

mybatis 3.x源码深度解析与最佳实践

html版离线文件可从<https://files.cnblogs.com/files/zhjh256/mybatis3.x%E6%BA%90%E7%A0%81%E6%B7%B1%E5%BA%A6%E8%A7%A3%E6%9E%90%E4%B8%8E%E6%9C%80%E4%BD%B3%E5%AE%9E%E8%B7%B5.rar>下载。

- - [1 环境准备](#)
 - [1.1 mybatis介绍以及框架源码的学习目标](#)
 - [1.2 本系列源码解析的方式](#)
 - [1.3 环境搭建](#)
 - [1.4 从Hello World开始](#)
 - [2 容器的加载与初始化](#)
 - [2.1 config文件解析XMLConfigBuilder.parseConfiguration](#)
 - [2.1.1 属性解析propertiesElement](#)
 - [2.1.2 加载settings节点settingsAsProperties](#)
 - [2.1.3 加载自定义VFS loadCustomVfs](#)
 - [2.1.4 解析类型别名typeAliasesElement](#)
 - [2.1.5 加载插件pluginElement](#)
 - [2.1.6 加载对象工厂objectFactoryElement](#)
 - [2.1.7 创建对象包装器工厂objectWrapperFactoryElement](#)
 - [2.1.8 加载反射工厂reflectorFactoryElement](#)
 - [2.1.9 加载环境配置environmentsElement](#)
 - [2.1.10 数据库厂商标识加载databaseIdProviderElement](#)
 - [2.1.11 加载类型处理器typeHandlerElement](#)
 - [处理枚举类型映射](#)
 - [2.1.12 加载mapper文件mapperElement](#)
 - [2.2 mapper加载与初始化](#)
 - [2.3 解析mapper文件XMLMapperBuilder](#)
 - [2.3.1 鉴别器discriminator的解析](#)
 - [解析SQL主体](#)
 - [sql语句解析的核心： mybatis语言驱动器XMLLanguageDriver](#)
 - [IfHandler](#)
 - [OtherwiseHandler](#)
 - [BindHandler](#)
 - [ChooseHandler](#)
 - [ForEachHandler](#)
 - [SetHandler](#)
 - [TrimHandler](#)
 - [WhereHandler](#)
 - [静态SQL创建RawSqlSource](#)
 - [动态SQL创建 DynamicSqlSource](#)
 - [4、二次解析未完成的结果映射、缓存参照、CRUD语句；](#)

- [3 关键对象总结与回顾](#)
 - [3.1 SqlSource](#)
 - [3.2 SqlNode](#)
 - [ChooseSqlNode](#)
 - [ForEachSqlNode](#)
 - [IfSqlNode](#)
 - [StaticTextSqlNode](#)
 - [TextSqlNode](#)
 - [VarDeclSqlNode](#)
 - [TrimSqlNode](#)
 - [SetSqlNode](#)
 - [WhereSqlNode](#)
 - [3.3 BaseBuilder](#)
 - [3.4 AdditionalParameter](#)
 - [3.5 TypeHandler](#)
 - [3.6 对象包装器工厂 ObjectWrapperFactory](#)
 - [3.7 MetaObject](#)
 - [3.8 对象工厂 ObjectFactory](#)
 - [3.13 LanguageDriver](#)
 - [3.14 ResultMap](#)
 - [3.15 ResultMapping](#)
 - [3.16 Discriminator](#)
- [4 SQL语句的执行流程](#)
 - [4.1 传统JDBC用法](#)
 - [4.2 mybatis执行SQL语句](#)
 - [4.2.1 获取openSession](#)
 - [4.2.2 sql语句执行方式一](#)
 - [mybatis结果集处理](#)
 - [selectMap实现](#)
 - [update/insert/delete实现](#)
 - [4.2.3 SQL语句执行方式二 SqlSession.getMapper实现](#)
 - [4.3 动态sql](#)
 - [4.4 存储过程与函数调用实现](#)
 - [4.5 mybatis事务实现](#)
 - [4.6 缓存](#)
- [5 执行期主要类总结](#)
 - [5.1 执行器Executor](#)
 - [5.4.1 SIMPLE执行器](#)
 - [5.4.2 REUSE执行器](#)

- [5.4.3 BATCH执行器](#)
- [5.4.4 缓存执行器CachingExecutor的实现](#)
- [5.2 参数处理器ParameterHandler](#)
- [5.3 语句处理器StatementHandler](#)
- [5.4 结果集处理器ResultSetHandler](#)
- [6 插件](#)
 - [6.1 分页插件PageHelper详解](#)
 - [6.2 自定义监控插件StatHelper实现](#)
- [7与spring集成](#)

1 环境准备

1.1 mybatis介绍以及框架源码的学习目标

MyBatis是当前最流行的java持久层框架之一，其通过XML配置的方式消除了绝大部分JDBC重复代码以及参数的设置，结果集的映射。虽然mybatis是最为流行的持久层框架之一，但是相比其他开源框架比如spring/netty的源码来说，其注释相对而言显得比较少。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。学习框架源码和学习框架本身不同，我们不仅要首先比较细致完整的熟悉框架提供的每个特性，还要理解框架本身的初始化过程等，除此之外，更重要的是，我们不能泛泛的快速浏览哪个功能是通过哪个类或者接口实现和封装的，对于核心特性和初始化过程的实现，我们应该达到下列目标：

1. 对于核心特性和内部功能，具体是如何实现的，采用什么数据结构，边研究边思考这么做是否合理，或者不合理，尤其是很多的特性和功能的调用频率是很低的，或者很多集合中包含的元素数量在99%以上的情况下都是1，这种情况下很多设计决定并不需要特别去选择或者适合于大数据量或者高并发的复杂实现。对于很多内部的数据结构和辅助方法，不仅需要知道其功能本身，还需要知道他们在上下文中发挥的作用。
2. 对于核心特性和内部功能，具体实现采用了哪些设计模式，使用这个设计模式的合理性；
3. 绝大部分框架都被设计为可扩展的，mybatis也不例外，它提供了很多的扩展点，比如最常用的插件，语言驱动器，执行器，对象工厂，对象包装器工厂等等都可以扩展，所以，我们应该知道如有必要的话，如何按照要求进行扩展以满足自己的需求；

1.2 本系列源码解析的方式

首先，和从使用层面相同，我们在理解实现细节的时候，也会涉及到学习A的时候，引用到B中的部分概念，B又引用了C的部分概念，C反过来又依赖于D接口和A的相关接口操作，D又有很多不同的具体实现。在使用层面，我们通常不需要深入去理解B的具体实现，只要知道B的概念和可以提供什么特性就可以了。在实现层面，我们必须去全部掌握这所有依赖的实现，直到最底层JDK原生操作或者某些我们熟悉的类库为止。这实际上涉及到横向流程和纵向切面的交叉引用，以及两个概念的接口和多种实现之间的交叉调用。所以，我们在整个系列中会先按照业务流程横向的方式进行整体分析，并附上必要的流程图，在整个大流程完成之后，在一个专门的章节安排对关键的类进行详细分析，它们的适用场景，这样就可以交叉引用，又不会在主流程中夹杂太多的干扰内容。

其次，因为我们免不了会对源码的主要部分做必要的注释，所以，对源码的讲解和注释会穿插进行，对于某些部分通过注释后已经完全能够知道其含义的实现，就不会在多此一举的重述。

1.3 环境搭建

从<https://github.com/mybatis/mybatis-3>下载mybatis 3源代码，导入eclipse工程，执行maven clean install，本书所使用的mybatis版本是3.4.6-SNAPSHOT，编写时的最新版本，读者如果使用其他版本，可能代码上会有细微差别，但其基本不影响我们对主要核心代码的分析。为了更好地理解mybatis的核心实现，我们需要创建一个演示工程mybatis-internal-example，读者可从github上下载，并将依赖的mybatis坐标改成mybatis-3.4.6-SNAPSHOT。

1.4 从Hello World开始

要理解mybatis的内部原理，最好的方式是直接从头开始，而不是从和spring的集成开始。每一个MyBatis的应用程序都以一个SqlSessionFactory对象的实例为核心。SqlSessionFactory对象的实例可以又通过SqlSessionFactoryBuilder对象来获得。SqlSessionFactoryBuilder对象可以从XML配置文件加载配置信息，然后创建SqlSessionFactory。先看下面的例子：

主程序：

```
package org.mybatis.internal.example;

import java.io.IOException;
import java.io.Reader;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.mybatis.internal.example.pojo.User;

public class MybatisHelloWorld {
    public static void main(String[] args) {
        String resource = "org/mybatis/internal/example/Configuration.xml";
        Reader reader;
        try {
            reader = Resources.getResourceAsReader(resource);
            SqlSessionFactory sqlMapper = new
SqlSessionFactoryBuilder().build(reader);

            SqlSession session = sqlMapper.openSession();
            try {
                User user = (User)
session.selectOne("org.mybatis.internal.example.mapper.UserMapper.getUser", 1);
                System.out.println(user.getLfPartyId() + "," + user.getPartyName());
            } finally {
                session.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

配置文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://10.7.12.4:3306/lfBase?
useUnicode=true"/>
                <property name="username" value="lfBase"/>
                <property name="password" value="eKffQV6wbh3sfQuFIG6M"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="org/mybatis/internal/example/UserMapper.xml"/>
    </mappers>
</configuration>
```

mapper文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.internal.example.mapper.UserMapper">
    <select id="getUser" parameterType="int"
resultType="org.mybatis.internal.example.pojo.User">
        select lfPartyId,partyName from LfParty where lfPartyId = #{id}
    </select>
</mapper>
```

mapper接口

```
package org.mybatis.internal.example.mapper;

import org.mybatis.internal.example.pojo.User;

public interface UserMapper {
    public User getUser(int lfPartyId);
}
```

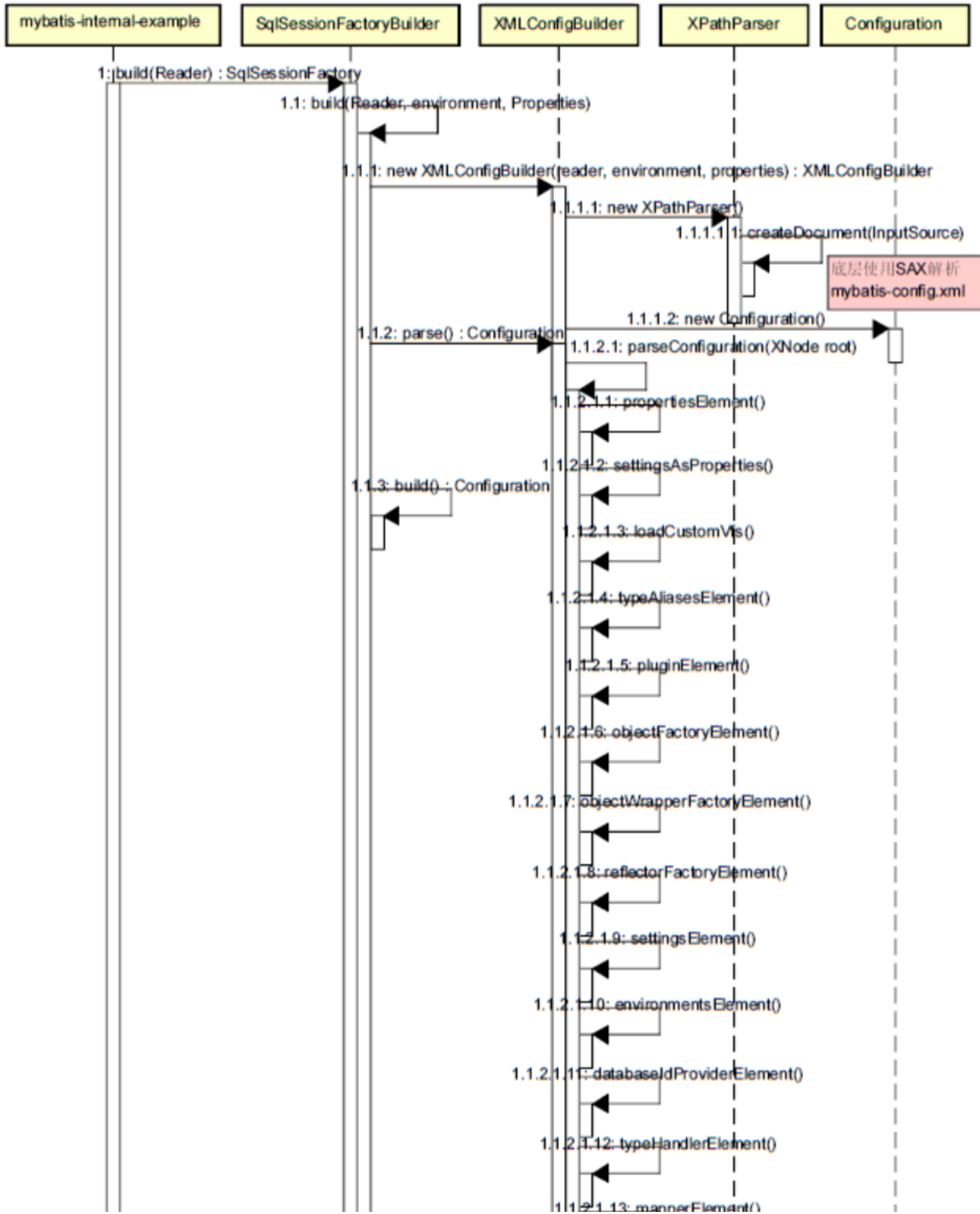
运行上述程序, 不出意外的话, 将输出:

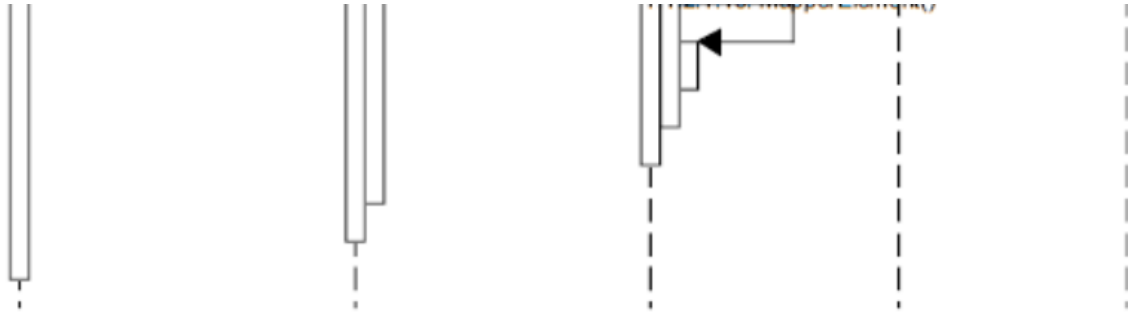
1, 汇金超级管理员

从上述代码可以看出，SqlSessionFactoryBuilder是一个关键的入口类，其中承担了mybatis配置文件的加载，解析，内部构建等职责。下一章，我们将重点分析mybatis配置文件的加载，配置类的构建等一系列细节。

2 容器的加载与初始化

SqlSessionFactory是通过SqlSessionFactoryBuilder工厂类创建的，而不是直接使用构造器。容器的配置文件加载和初始化流程如下：





SqlSessionFactoryBuilder的主要代码如下:

```
public class SqlSessionFactoryBuilder {
    // 使用java.io.Reader构建
    public SqlSessionFactory build(Reader reader) {
        return build(reader, null, null);
    }

    public SqlSessionFactory build(Reader reader, String environment) {
        return build(reader, environment, null);
    }

    public SqlSessionFactory build(Reader reader, Properties properties) {
        return build(reader, null, properties);
    }

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

    public SqlSessionFactory build(InputStream inputStream) {
        return build(inputStream, null, null);
    }

    public SqlSessionFactory build(InputStream inputStream, String environment) {
        return build(inputStream, environment, null);
    }
}
```

```

public SqlSessionFactory build(InputStream inputStream, Properties properties) {
    return build(inputStream, null, properties);
}

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

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

```

根据上述接口可知，SqlSessionFactory提供了根据字节流、字符流以及直接使用org.apache.ibatis.session.Configuration配置类（后续我们会详细讲到）三种途径的读取配置信息方式，无论是字符流还是字节流方式，首先都是将XML配置文件构建为Configuration配置类，然后将Configuration设置到SqlSessionFactory默认实现DefaultSqlSessionFactory的configurationz字段并返回。所以，它本身很简单，解析配置文件的关键逻辑都委托给XMLConfigBuilder了，我们以字符流也就是java.io.Reader为例进行分析，SqlSessionFactoryBuilder使用了XMLConfigBuilder作为解析器。

```

public class XMLConfigBuilder extends BaseBuilder {

    private boolean parsed;
    private XPathParser parser;
    private String environment;
    private ReflectorFactory localReflectorFactory = new DefaultReflectorFactory();
    ....
    public XMLConfigBuilder(Reader reader, String environment, Properties props) {
        this(new XPathParser(reader, true, props, new XMLMapperEntityResolver()),
environment, props);
    }
    ....
}

```


XMLConfigBuilder以及解析Mapper文件的XMLMapperBuilder都继承于BaseBuilder。他们对于XML文件本身技术上的加载和解析都委托给了XPathParser，最终用的是jdk自带的xml解析器而非第三方比如dom4j，底层使用了xpath方式进行节点解析。new XPathParser(reader, true, props, new XMLMapperEntityResolver())的参数含义分别是Reader，是否进行DTD 校验，属性配置，XML实体节点解析器。

entityResolver比较好理解，跟Spring的XML标签解析器一样，有默认的解析器，也有自定义的比如tx，dubbo等，主要使用了策略模式，在这里mybatis硬编码为了XMLMapperEntityResolver。

XMLMapperEntityResolver的定义如下：

```
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";

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

    /*
     * Converts a public DTD into a local one
     * 将公共的DTD转换为本地模式
     *
     * @param publicId The public id that is what comes after "PUBLIC"
     * @param systemId The system id that is what comes after the public id.
     * @return The InputSource for the DTD
     *
     * @throws org.xml.sax.SAXException If anything goes wrong
     */
    @Override
    public InputSource resolveEntity(String publicId, String systemId) throws
SAXException {
        try {
            if (systemId != null) {
                String lowerCaseSystemId = systemId.toLowerCase(Locale.ENGLISH);
                if (lowerCaseSystemId.contains(MYBATIS_CONFIG_SYSTEM) ||
lowerCaseSystemId.contains(IBATIS_CONFIG_SYSTEM)) {
                    return getInputSource(MYBATIS_CONFIG_DTD, publicId, systemId);
                } 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 {
            InputStream in = Resources.getResourceAsStream(path);
            source = new InputSource(in);
            source.setPublicId(publicId);
            source.setSystemId(systemId);
        } catch (IOException e) {
            // ignore, null is ok
        }
    }
    return source;
}
}

```

从上述代码可以看出，mybatis解析的时候，引用了本地的DTD文件，和本类在同一个package下，其中的ibatis-3-config.dtd应该主要是用于兼容用途。在其中getInputSource(MYBATIS_CONFIG_DTD, publicId, systemId)的调用里面有两个参数publicId（公共标识符）和systemId（系统标识符），他们是XML 1.0规范的一部分。他们的使用如下：

```

<?xml version="1.0"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
<greeting>Hello, world!</greeting>

```

系统标识符“hello.dtd”给出了此文件的 DTD 的地址（一个 URI 引用）。在mybatis中，这里指定的是mybatis-3-config.dtd和ibatis-3-config.dtd。publicId则一般从XML文档的DOCTYPE中获取，如下：

```

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">

```

继续往下看，

```

public XPathParser(Reader reader, boolean validation, Properties variables,
EntityResolver entityResolver) {
    commonConstructor(validation, variables, entityResolver);
    this.document = createDocument(new InputSource(reader));
}

private void commonConstructor(boolean validation, Properties variables,
EntityResolver entityResolver) {
    this.validation = validation;
    this.entityResolver = entityResolver;
    this.variables = variables;
    XPathFactory factory = XPathFactory.newInstance();
    this.xpath = factory.newXPath();
}

```

可知，commonConstructor并没有做什么。回过头到createDocument上，其使用了org.xml.sax.InputSource作为参数，createDocument的关键代码如下：

```

private Document createDocument(InputSource inputSource) {
    // important: this must only be called AFTER common constructor
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setValidating(validation);
        //设置由本工厂创建的解析器是否支持XML命名空间 TODO 什么是XML命名空间
        factory.setNamespaceAware(false);
        factory.setIgnoringComments(true);
        factory.setIgnoringElementContentWhitespace(false);
        //设置是否将CDATA节点转换为Text节点
        factory.setCoalescing(false);
        //设置是否展开实体引用节点，这里应该是sql片段引用的关键
        factory.setExpandEntityReferences(true);

        DocumentBuilder builder = factory.newDocumentBuilder();
        //设置解析mybatis xml文档节点的解析器，也就是上面的XMLMapperEntityResolver
        builder.setEntityResolver(entityResolver);
        builder.setErrorHandler(new ErrorHandler() {
            @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 {

```

```

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

```

主要是根据mybatis自身需要创建一个文档解析器，然后调用parse将输入input source解析为DOM XML文档并返回。

得到XPathParser实例之后，就调用另一个使用XPathParser作为配置来源的重载构造函数了，如下：

```

private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {
    super(new Configuration());
    ErrorContext.instance().resource("SQL Mapper Configuration");
    this.configuration.setVariables(props);
    this.parsed = false;
    this.environment = environment;
    this.parser = parser;
}

```

其中调用了父类BaseBuilder的构造器（主要是设置类型别名注册器，以及类型处理器注册器）：

```

public abstract class BaseBuilder {
    protected final Configuration configuration;
    protected final TypeAliasRegistry typeAliasRegistry;
    protected final TypeHandlerRegistry typeHandlerRegistry;

    public BaseBuilder(Configuration configuration) {
        this.configuration = configuration;
        this.typeAliasRegistry = this.configuration.getTypeAliasRegistry();
        this.typeHandlerRegistry = this.configuration.getTypeHandlerRegistry();
    }
}

```

我们等下再来看Configuration这个核心类的关键信息，主要是xml配置文件里面的所有配置的实现表示(几乎所有的情况下都要依赖与Configuration这个类)。

设置关键配置environment以及properties文件（mybatis在这里的实现和spring的机制有些不同），最后将上面构建的XPathParser设置为XMLConfigBuilder的parser属性值。

XMLConfigBuilder创建完成之后，SqlSessionFactoryBuild调用parser.parse()创建Configuration。所有，真正Configuration构建逻辑就在XMLConfigBuilder.parse()里面，如下所示：

```

public class XMLConfigBuilder extends BaseBuilder {
    public Configuration parse() {
        if (parsed) {
            throw new BuilderException("Each XMLConfigBuilder can only be used once.");
        }
        parsed = true;
        //mybatis配置文件解析的主流程
        parseConfiguration(parser.evalNode("/configuration"));
        return configuration;
    }
}

```

2.1 config文件解析XMLConfigBuilder.parseConfiguration

首先判断有没有解析过配置文件，只有没有解析过才允许解析。其中调用了 `parser.evalNode("/configuration")` 返回根节点的 `org.apache.ibatis.parsing.XNode` 表示，`XNode` 里面主要把关键的节点属性和占位符变量结构化出来，后面我们再看。然后调用 `parseConfiguration` 根据 mybatis 的主要配置进行解析，如下所示：

```

private void parseConfiguration(XNode root) {
    try {
        //issue #117 read properties first
        propertiesElement(root.evalNode("properties"));
        Properties settings = settingsAsProperties(root.evalNode("settings"));
        loadCustomVfs(settings);
        typeAliasesElement(root.evalNode("typeAliases"));
        pluginElement(root.evalNode("plugins"));
        objectFactoryElement(root.evalNode("objectFactory"));
        objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
        reflectorFactoryElement(root.evalNode("reflectorFactory"));
        settingsElement(settings);
        // read it after objectFactory and objectWrapperFactory issue #631
        environmentsElement(root.evalNode("environments"));
        databaseIdProviderElement(root.evalNode("databaseIdProvider"));
        typeHandlerElement(root.evalNode("typeHandlers"));
        mapperElement(root.evalNode("mappers"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e,
e);
    }
}

```

所有的 `root.evalNode` 底层都是调用 XML DOM 的 `evaluate()` 方法，根据给定的节点表达式来计算指定的 XPath 表达式，并且返回一个 `XPathResult` 对象，返回类型在 `Node.evalNode()` 方法中均被指定为 `NODE`。

在详细解释每个节点的解析前，我们先来粗略看下 mybatis-3-config.dtd 的声明：

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<!ELEMENT configuration (properties?, settings?, typeAliases?, typeHandlers?,
objectFactory?, objectWrapperFactory?, reflectorFactory?, plugins?, environments?,
databaseIdProvider?, mappers?)>
```

```
<!ELEMENT databaseIdProvider (property*)>
<!ATTLIST databaseIdProvider
type CDATA #REQUIRED
>
```

```
<!ELEMENT properties (property*)>
<!ATTLIST properties
resource CDATA #IMPLIED
url CDATA #IMPLIED
>
```

```
<!ELEMENT property EMPTY>
<!ATTLIST property
name CDATA #REQUIRED
value CDATA #REQUIRED
>
```

```
<!ELEMENT settings (setting+)>
```

```
<!ELEMENT setting EMPTY>
<!ATTLIST setting
name CDATA #REQUIRED
value CDATA #REQUIRED
>
```

```
<!ELEMENT typeAliases (typeAlias*,package*)>
```

```
<!ELEMENT typeAlias EMPTY>
<!ATTLIST typeAlias
type CDATA #REQUIRED
alias CDATA #IMPLIED
>
```

```
<!ELEMENT typeHandlers (typeHandler*,package*)>
```

```
<!ELEMENT typeHandler EMPTY>
<!ATTLIST typeHandler
javaType CDATA #IMPLIED
jdbcType CDATA #IMPLIED
handler CDATA #REQUIRED
>
```

```
<!ELEMENT objectFactory (property*)>
<!ATTLIST objectFactory
type CDATA #REQUIRED
```

```
>

<!ELEMENT objectWrapperFactory EMPTY>
<!ATTLIST objectWrapperFactory
type CDATA #REQUIRED
>

<!ELEMENT reflectorFactory EMPTY>
<!ATTLIST reflectorFactory
type CDATA #REQUIRED
>

<!ELEMENT plugins (plugin+)>

<!ELEMENT plugin (property*)>
<!ATTLIST plugin
interceptor CDATA #REQUIRED
>

<!ELEMENT environments (environment+)>
<!ATTLIST environments
default CDATA #REQUIRED
>

<!ELEMENT environment (transactionManager,dataSource)>
<!ATTLIST environment
id CDATA #REQUIRED
>

<!ELEMENT transactionManager (property*)>
<!ATTLIST transactionManager
type CDATA #REQUIRED
>

<!ELEMENT dataSource (property*)>
<!ATTLIST dataSource
type CDATA #REQUIRED
>

<!ELEMENT mappers (mapper*,package*)>

<!ELEMENT mapper EMPTY>
<!ATTLIST mapper
resource CDATA #IMPLIED
url CDATA #IMPLIED
class CDATA #IMPLIED
>

<!ELEMENT package EMPTY>
```

```
<!ATTLIST package
name CDATA #REQUIRED
>
```

从上述DTD可知，mybatis-config文件最多有11个配置项，分别是properties?, settings?, typeAliases?, typeHandlers?, objectFactory?, objectWrapperFactory?, reflectorFactory?, plugins?, environments?, databaseIdProvider?, mappers?，但是所有的配置都是可选的，这意味着mybatis-config配置文件本身可以什么都不包含。因为所有的配置最后保存到org.apache.ibatis.session.Configuration中，所以在详细查看每块配置的解析前，我们来看下Configuration的内部完整结构：

```
package org.apache.ibatis.session;
...省略import

public class Configuration {

    protected Environment environment;
    // 允许在嵌套语句中使用分页（RowBounds）。如果允许使用则设置为false。默认为false
    protected boolean safeRowBoundsEnabled;
    // 允许在嵌套语句中使用分页（ResultHandler）。如果允许使用则设置为false。
    protected boolean safeResultHandlerEnabled = true;
    // 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A_COLUMN 到经典 Java 属性名
    aColumn 的类似映射。默认false
    protected boolean mapUnderscoreToCamelCase;
    // 当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载。默认值false（true in
    ≤3.4.1）
    protected boolean aggressiveLazyLoading;
    // 是否允许单一语句返回多结果集（需要兼容驱动）。
    protected boolean multipleResultSetsEnabled = true;

    // 允许 JDBC 支持自动生成主键，需要驱动兼容。这就是insert时获取mysql自增主键/oracle sequence的
    开关。注：一般来说，这是希望的结果，应该默认值为true比较合适。
    protected boolean useGeneratedKeys;

    // 使用列标签代替列名，一般来说，这是希望的结果
    protected boolean useColumnLabel = true;

    // 是否启用缓存
    protected boolean cacheEnabled = true;

    // 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有
    Map.keySet() 依赖或 null 值初始化的时候是有用的。
    protected boolean callSettersOnNulls;

    // 允许使用方法签名中的名称作为语句参数名称。 为了使用该特性，你的工程必须采用Java 8编译，并且加上-
    parameters选项。（从3.4.1开始）
    protected boolean useActualParamName = true;
```


//当返回行的所有列都是空时，MyBatis默认返回null。 当开启这个设置时，MyBatis会返回一个空实例。 请注意，它也适用于嵌套的结果集（i.e. collection and association）。（从3.4.2开始） 注：这里应该拆分为两个参数比较合适，一个用于结果集，一个用于单记录。通常来说，我们会希望结果集不是null，单记录仍然是null

```
protected boolean returnInstanceForEmptyRow;
```

// 指定 MyBatis 增加到日志名称的前缀。

```
protected String logPrefix;
```

// 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。一般建议指定为slf4j或log4j

```
protected Class <? extends Log> logImpl;
```

// 指定VFS的实现，VFS是mybatis提供的用于访问AS内资源的一个简便接口

```
protected Class <? extends VFS> vfsImpl;
```

// MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。

```
protected LocalCacheScope localCacheScope = LocalCacheScope.SESSION;
```

// 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。

```
protected JdbcType jdbcTypeForNull = JdbcType.OTHER;
```

// 指定对象的哪个方法触发一次延迟加载。

```
protected Set<String> lazyLoadTriggerMethods = new HashSet<String>(Arrays.asList(new String[] { "equals", "clone", "hashCode", "toString" }));
```

// 设置超时时间，它决定驱动等待数据库响应的秒数。默认不超时

```
protected Integer defaultStatementTimeout;
```

// 为驱动的结果集设置默认获取数量。

```
protected Integer defaultFetchSize;
```

// SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。

```
protected ExecutorType defaultExecutorType = ExecutorType.SIMPLE;
```

// 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。

```
protected AutoMappingBehavior autoMappingBehavior = AutoMappingBehavior.PARTIAL;
```

// 指定发现自动映射目标未知列（或者未知属性类型）的行为。这个值应该设置为WARNING比较合适

```
protected AutoMappingUnknownColumnBehavior autoMappingUnknownColumnBehavior = AutoMappingUnknownColumnBehavior.NONE;
```

// settings下的properties属性

```
protected Properties variables = new Properties();
```

```

// 默认的反射器工厂,用于操作属性、构造器方便
protected ReflectorFactory reflectorFactory = new DefaultReflectorFactory();

// 对象工厂,所有的类resultMap类都需要依赖于对象工厂来实例化
protected ObjectFactory objectFactory = new DefaultObjectFactory();

// 对象包装器工厂,主要用来在创建非原生对象,比如增加了某些监控或者特殊属性的代理类
protected ObjectWrapperFactory objectWrapperFactory = new
DefaultObjectWrapperFactory();

// 延迟加载的全局开关。当开启时,所有关联对象都会延迟加载。特定关联关系中可通过设置fetchType属性来覆
盖该项的开关状态。
protected boolean lazyLoadingEnabled = false;

// 指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。MyBatis 3.3+使用JAVASSIST
protected ProxyFactory proxyFactory = new JavassistProxyFactory(); // #224 Using
internal Javassist instead of OGNL

// MyBatis 可以根据不同的数据库厂商执行不同的语句,这种多厂商的支持是基于映射语句中的 databaseId
属性。
protected String databaseId;

/**
 * Configuration factory class.
 * Used to create Configuration for loading deserialized unread properties.
 * 指定一个提供Configuration实例的类。这个被返回的Configuration实例是用来加载被反序列化对象的懒
加载属性值。这个类必须包含一个签名方法static Configuration getConfiguration()。(从 3.2.3 版本
开始)
 */
protected Class<?> configurationFactory;

protected final MapperRegistry mapperRegistry = new MapperRegistry(this);

// mybatis插件列表
protected final InterceptorChain interceptorChain = new InterceptorChain();
protected final TypeHandlerRegistry typeHandlerRegistry = new TypeHandlerRegistry();

// 类型注册器,用于在执行sql语句的出入参映射以及mybatis-config文件里的各种配置比如
<transactionManager type="JDBC"/><dataSource type="POOLED">时使用简写,后面会详细解释
protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();
protected final LanguageDriverRegistry languageRegistry = new
LanguageDriverRegistry();

protected final Map<String, MappedStatement> mappedStatements = new
StrictMap<MappedStatement>("Mapped Statements collection");
protected final Map<String, Cache> caches = new StrictMap<Cache>("Caches
collection");
protected final Map<String, ResultMap> resultMaps = new StrictMap<ResultMap>("Result
Maps collection");

```

```

protected final Map<String, ParameterMap> parameterMaps = new StrictMap<ParameterMap>
("Parameter Maps collection");
protected final Map<String, KeyGenerator> keyGenerators = new StrictMap<KeyGenerator>
("Key Generators collection");

protected final Set<String> loadedResources = new HashSet<String>();
protected final Map<String, XNode> sqlFragments = new StrictMap<XNode>("XML fragments
parsed from previous mappers");

protected final Collection<XMLStatementBuilder> incompleteStatements = new
LinkedList<XMLStatementBuilder>();
protected final Collection<CacheRefResolver> incompleteCacheRefs = new
LinkedList<CacheRefResolver>();
protected final Collection<ResultMapResolver> incompleteResultMaps = new
LinkedList<ResultMapResolver>();
protected final Collection<MethodResolver> incompleteMethods = new
LinkedList<MethodResolver>();

/*
 * A map holds cache-ref relationship. The key is the namespace that
 * references a cache bound to another namespace and the value is the
 * namespace which the actual cache is bound to.
 */
protected final Map<String, String> cacheRefMap = new HashMap<String, String>();

public Configuration(Environment environment) {
    this();
    this.environment = environment;
}

public Configuration() {
    // 内置别名注册
    typeAliasRegistry.registerAlias("JDBC", JdbcTransactionFactory.class);
    typeAliasRegistry.registerAlias("MANAGED", ManagedTransactionFactory.class);

    typeAliasRegistry.registerAlias("JNDI", JndiDataSourceFactory.class);
    typeAliasRegistry.registerAlias("POOLED", PooledDataSourceFactory.class);
    typeAliasRegistry.registerAlias("UNPOOLED", UnpooledDataSourceFactory.class);

    typeAliasRegistry.registerAlias("PERPETUAL", PerpetualCache.class);
    typeAliasRegistry.registerAlias("FIFO", FifoCache.class);
    typeAliasRegistry.registerAlias("LRU", LruCache.class);
    typeAliasRegistry.registerAlias("SOFT", SoftCache.class);
    typeAliasRegistry.registerAlias("WEAK", WeakCache.class);

    typeAliasRegistry.registerAlias("DB_VENDOR", VendorDatabaseIdProvider.class);

    typeAliasRegistry.registerAlias("XML", XMLLanguageDriver.class);
    typeAliasRegistry.registerAlias("RAW", RawLanguageDriver.class);

```

```

        typeAliasRegistry.registerAlias("SLF4J", Slf4jImpl.class);
        typeAliasRegistry.registerAlias("COMMONS_LOGGING",
JakartaCommonsLoggingImpl.class);
        typeAliasRegistry.registerAlias("LOG4J", Log4jImpl.class);
        typeAliasRegistry.registerAlias("LOG4J2", Log4j2Impl.class);
        typeAliasRegistry.registerAlias("JDK_LOGGING", Jdk14LoggingImpl.class);
        typeAliasRegistry.registerAlias("STDOUT_LOGGING", StdOutImpl.class);
        typeAliasRegistry.registerAlias("NO_LOGGING", NoLoggingImpl.class);

        typeAliasRegistry.registerAlias("CGLIB", CglibProxyFactory.class);
        typeAliasRegistry.registerAlias("JAVASSIST", JavassistProxyFactory.class);

        languageRegistry.setDefaultDriverClass(XMLLanguageDriver.class);
        languageRegistry.register(RawLanguageDriver.class);
    }

```

... 省去不必要的getter/setter

```

public Class<? extends VFS> getVfsImpl() {
    return this.vfsImpl;
}

public void setVfsImpl(Class<? extends VFS> vfsImpl) {
    if (vfsImpl != null) {
        this.vfsImpl = vfsImpl;
        VFS.addImplClass(this.vfsImpl);
    }
}

public ProxyFactory getProxyFactory() {
    return proxyFactory;
}

public void setProxyFactory(ProxyFactory proxyFactory) {
    if (proxyFactory == null) {
        proxyFactory = new JavassistProxyFactory();
    }
    this.proxyFactory = proxyFactory;
}

/**
 * @since 3.2.2
 */
public List<Interceptor> getInterceptors() {
    return interceptorChain.getInterceptors();
}

public LanguageDriverRegistry getLanguageRegistry() {

```

```

        return languageRegistry;
    }

    public void setDefaultScriptingLanguage(Class<?> driver) {
        if (driver == null) {
            driver = XMLLanguageDriver.class;
        }
        getLanguageRegistry().setDefaultDriverClass(driver);
    }

    public LanguageDriver getDefaultScriptingLanguageInstance() {
        return languageRegistry.getDefaultDriver();
    }

    /** @deprecated Use {@link #getDefaultScriptingLanguageInstance()} */
    @Deprecated
    public LanguageDriver getDefaultScriptingLanguageInstance() {
        return getDefaultScriptingLanguageInstance();
    }

    public MetaObject newMetaObject(Object object) {
        return MetaObject.forObject(object, objectFactory, objectWrapperFactory,
            reflectorFactory);
    }

    public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object
parameterObject, BoundSql boundSql) {
        ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement, parameterObject,
boundSql);
        parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);
        return parameterHandler;
    }

    public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement
mappedStatement, RowBounds rowBounds, ParameterHandler parameterHandler,
        ResultHandler resultHandler, BoundSql boundSql) {
        ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor,
mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);
        resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);
        return resultSetHandler;
    }

    public StatementHandler newStatementHandler(Executor executor, MappedStatement
mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler
resultHandler, BoundSql boundSql) {
        StatementHandler statementHandler = new RoutingStatementHandler(executor,
mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);
        statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
    }

```

```

    return statementHandler;
}

public Executor newExecutor(Transaction transaction) {
    return newExecutor(transaction, defaultExecutorType);
}

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    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);
    }
    if (cacheEnabled) {
        executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}

public void addKeyGenerator(String id, KeyGenerator keyGenerator) {
    keyGenerators.put(id, keyGenerator);
}

public Collection<String> getKeyGeneratorNames() {
    return keyGenerators.keySet();
}

public Collection<KeyGenerator> getKeyGenerators() {
    return keyGenerators.values();
}

public KeyGenerator getKeyGenerator(String id) {
    return keyGenerators.get(id);
}

public boolean hasKeyGenerator(String id) {
    return keyGenerators.containsKey(id);
}

public void addCache(Cache cache) {
    caches.put(cache.getId(), cache);
}

```

```
public Collection<String> getCacheNames() {
    return caches.keySet();
}

public Collection<Cache> getCaches() {
    return caches.values();
}

public Cache getCache(String id) {
    return caches.get(id);
}

public boolean hasCache(String id) {
    return caches.containsKey(id);
}

public void addResultMap(ResultMap rm) {
    resultMaps.put(rm.getId(), rm);
    checkLocallyForDiscriminatedNestedResultMaps(rm);
    checkGloballyForDiscriminatedNestedResultMaps(rm);
}

public Collection<String> getResultMapNames() {
    return resultMaps.keySet();
}

public Collection<ResultMap> getResultMaps() {
    return resultMaps.values();
}

public ResultMap getResultMap(String id) {
    return resultMaps.get(id);
}

public boolean hasResultMap(String id) {
    return resultMaps.containsKey(id);
}

public void addParameterMap(ParameterMap pm) {
    parameterMaps.put(pm.getId(), pm);
}

public Collection<String> getParameterMapNames() {
    return parameterMaps.keySet();
}

public Collection<ParameterMap> getParameterMaps() {
    return parameterMaps.values();
}
```

```
public ParameterMap getParameterMap(String id) {
    return parameterMaps.get(id);
}

public boolean hasParameterMap(String id) {
    return parameterMaps.containsKey(id);
}

public void addMappedStatement(MappedStatement ms) {
    mappedStatements.put(ms.getId(), ms);
}

public Collection<String> getMappedStatementNames() {
    buildAllStatements();
    return mappedStatements.keySet();
}

public Collection<MappedStatement> getMappedStatements() {
    buildAllStatements();
    return mappedStatements.values();
}

public Collection<XMLStatementBuilder> getIncompleteStatements() {
    return incompleteStatements;
}

public void addIncompleteStatement(XMLStatementBuilder incompleteStatement) {
    incompleteStatements.add(incompleteStatement);
}

public Collection<CacheRefResolver> getIncompleteCacheRefs() {
    return incompleteCacheRefs;
}

public void addIncompleteCacheRef(CacheRefResolver incompleteCacheRef) {
    incompleteCacheRefs.add(incompleteCacheRef);
}

public Collection<ResultMapResolver> getIncompleteResultMaps() {
    return incompleteResultMaps;
}

public void addIncompleteResultMap(ResultMapResolver resultMapResolver) {
    incompleteResultMaps.add(resultMapResolver);
}

public void addIncompleteMethod(MethodResolver builder) {
    incompleteMethods.add(builder);
}
```



```

}

public Collection<MethodResolver> getIncompleteMethods() {
    return incompleteMethods;
}

public MappedStatement getMappedStatement(String id) {
    return this.getMappedStatement(id, true);
}

public MappedStatement getMappedStatement(String id, boolean
validateIncompleteStatements) {
    if (validateIncompleteStatements) {
        buildAllStatements();
    }
    return mappedStatements.get(id);
}

public Map<String, XNode> getSqlFragments() {
    return sqlFragments;
}

public void addInterceptor(Interceptor interceptor) {
    interceptorChain.addInterceptor(interceptor);
}

public void addMappers(String packageName, Class<?> superType) {
    mapperRegistry.addMappers(packageName, superType);
}

public void addMappers(String packageName) {
    mapperRegistry.addMappers(packageName);
}

public <T> void addMapper(Class<T> type) {
    mapperRegistry.addMapper(type);
}

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

public boolean hasMapper(Class<?> type) {
    return mapperRegistry.hasMapper(type);
}

public boolean hasStatement(String statementName) {
    return hasStatement(statementName, true);
}

```

```

    public boolean hasStatement(String statementName, boolean
validateIncompleteStatements) {
        if (validateIncompleteStatements) {
            buildAllStatements();
        }
        return mappedStatements.containsKey(statementName);
    }

    public void addCacheRef(String namespace, String referencedNamespace) {
        cacheRefMap.put(namespace, referencedNamespace);
    }

    /*
     * Parses all the unprocessed statement nodes in the cache. It is recommended
     * to call this method once all the mappers are added as it provides fail-fast
     * statement validation.
     */
    protected void buildAllStatements() {
        if (!incompleteResultMaps.isEmpty()) {
            synchronized (incompleteResultMaps) {
                // This always throws a BuilderException.
                incompleteResultMaps.iterator().next().resolve();
            }
        }
        if (!incompleteCacheRefs.isEmpty()) {
            synchronized (incompleteCacheRefs) {
                // This always throws a BuilderException.
                incompleteCacheRefs.iterator().next().resolveCacheRef();
            }
        }
        if (!incompleteStatements.isEmpty()) {
            synchronized (incompleteStatements) {
                // This always throws a BuilderException.
                incompleteStatements.iterator().next().parseStatementNode();
            }
        }
        if (!incompleteMethods.isEmpty()) {
            synchronized (incompleteMethods) {
                // This always throws a BuilderException.
                incompleteMethods.iterator().next().resolve();
            }
        }
    }

    /*
     * Extracts namespace from fully qualified statement id.
     *
     * @param statementId

```

```

    * @return namespace or null when id does not contain period.
    */
protected String extractNamespace(String statementId) {
    int lastPeriod = statementId.lastIndexOf('.');
    return lastPeriod > 0 ? statementId.substring(0, lastPeriod) : null;
}

// Slow but a one time cost. A better solution is welcome.
protected void checkGloballyForDiscriminatedNestedResultMaps(ResultMap rm) {
    if (rm.hasNestedResultMaps()) {
        for (Map.Entry<String, ResultMap> entry : resultMaps.entrySet()) {
            Object value = entry.getValue();
            if (value instanceof ResultMap) {
                ResultMap entryResultMap = (ResultMap) value;
                if (!entryResultMap.hasNestedResultMaps() &&
entryResultMap.getDiscriminator() != null) {
                    Collection<String> discriminatedResultMapNames =
entryResultMap.getDiscriminator().getDiscriminatorMap().values();
                    if (discriminatedResultMapNames.contains(rm.getId())) {
                        entryResultMap.forceNestedResultMaps();
                    }
                }
            }
        }
    }
}

// Slow but a one time cost. A better solution is welcome.
protected void checkLocallyForDiscriminatedNestedResultMaps(ResultMap rm) {
    if (!rm.hasNestedResultMaps() && rm.getDiscriminator() != null) {
        for (Map.Entry<String, String> entry :
rm.getDiscriminator().getDiscriminatorMap().entrySet()) {
            String discriminatedResultMapName = entry.getValue();
            if (hasResultMap(discriminatedResultMapName)) {
                ResultMap discriminatedResultMap =
resultMaps.get(discriminatedResultMapName);
                if (discriminatedResultMap.hasNestedResultMaps()) {
                    rm.forceNestedResultMaps();
                    break;
                }
            }
        }
    }
}

protected static class StrictMap<V> extends HashMap<String, V> {

    private static final long serialVersionUID = -4950446264854982944L;
    private final String name;

```

```

public StrictMap(String name, int initialCapacity, float loadFactor) {
    super(initialCapacity, loadFactor);
    this.name = name;
}

public StrictMap(String name, int initialCapacity) {
    super(initialCapacity);
    this.name = name;
}

public StrictMap(String name) {
    super();
    this.name = name;
}

public StrictMap(String name, Map<String, ? extends V> m) {
    super(m);
    this.name = name;
}

@SuppressWarnings("unchecked")
public V put(String key, V value) {
    if (containsKey(key)) {
        throw new IllegalArgumentException(name + " already contains value for " +
key);
    }
    if (key.contains(".")) {
        final String shortKey = getShortName(key);
        if (super.get(shortKey) == null) {
            super.put(shortKey, value);
        } else {
            super.put(shortKey, (V) new Ambiguity(shortKey));
        }
    }
    return super.put(key, value);
}

public V get(Object key) {
    V value = super.get(key);
    if (value == null) {
        throw new IllegalArgumentException(name + " does not contain value for " +
key);
    }
    if (value instanceof Ambiguity) {
        throw new IllegalArgumentException(((Ambiguity) value).getSubject() + " is
ambiguous in " + name
        + " (try using the full name including the namespace, or rename one of the
entries)");
    }
}

```

```

    }
    return value;
}

private String getShortName(String key) {
    final String[] keyParts = key.split("\\.");
    return keyParts[keyParts.length - 1];
}

protected static class Ambiguity {
    final private String subject;

    public Ambiguity(String subject) {
        this.subject = subject;
    }

    public String getSubject() {
        return subject;
    }
}
}
}

```

mybatis中所有环境配置、resultMap集合、sql语句集合、插件列表、缓存、加载的xml列表、类型别名、类型处理器等全部都维护在Configuration中。Configuration中包含了一个内部静态类StrictMap，它继承于HashMap，对HashMap的装饰在于增加了put时防重复的处理，get时取不到值时候的异常处理，这样核心应用层就不需要额外关心各种对象异常处理,简化应用层逻辑。

从设计上来说，我们可以说Configuration并不是一个thin类(也就是仅包含了属性以及getter/setter)，而是一个rich类，它对部分逻辑进行了封装便于用户直接使用,而不是让用户各自散落处理，比如addResultMap方法和getMappedStatement方法等等。

最新的完整mybatis每个配置属性含义可参考<http://www.mybatis.org/mybatis-3/zh/configuration.html>。

从Configuration构造器和protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();可以看出，所有我们在mybatis-config和mapper文件中使用的类似int/string/JDBC/POOLED等字面常量最终解析为具体的java类型都是在typeAliasRegistry构造器和Configuration构造器执行期间初始化的。下面我们来每块分析。

2.1.1 属性解析propertiesElement

解析properties的方法为：

```

propertiesElement(root.evalNode("properties"));
private void propertiesElement(XNode context) throws Exception {
    if (context != null) {
        // 加载property节点为property
        Properties defaults = context.getChildrenAsProperties();
        String resource = context.getStringAttribute("resource");
        String url = context.getStringAttribute("url");
        // 必须至少包含resource或者url属性之一
    }
}

```

```

        if (resource != null && url != null) {
            throw new BuilderException("The properties element cannot specify both a URL
and a resource based property file reference. Please specify one or the other.");
        }
        if (resource != null) {
            defaults.putAll(Resources.getResourceAsProperties(resource));
        } else if (url != null) {
            defaults.putAll(Resources.getUrlAsProperties(url));
        }
        Properties vars = configuration.getVariables();
        if (vars != null) {
            defaults.putAll(vars);
        }
        parser.setVariables(defaults);
        configuration.setVariables(defaults);
    }
}

```

总体逻辑比较简单，首先加载properties节点下的property属性，比如：

```

<properties resource="org/mybatis/internal/example/config.properties">
    <property name="username" value="dev_user"/>
    <property name="password" value="F2Fa3!33TYyg"/>
</properties>

```

然后从url或resource加载配置文件，都先和configuration.variables合并，然后赋值到XMLConfigBuilder.parser和BaseBuilder.configuration。此时开始所有的属性就可以在随后的整个配置文件中使用了。

2.1.2 加载settings节点settingsAsProperties

```

private Properties settingsAsProperties(XNode context) {
    if (context == null) {
        return new Properties();
    }
    Properties props = context.getChildrenAsProperties();
    // Check that all settings are known to the configuration class
    // 检查所有从settings加载的设置,确保它们都在Configuration定义的范围内
    MetaClass metaConfig = MetaClass.forClass(Configuration.class,
localReflectorFactory);
    for (Object key : props.keySet()) {
        if (!metaConfig.hasSetter(String.valueOf(key))) {
            throw new BuilderException("The setting " + key + " is not known. Make sure
you spelled it correctly (case sensitive).");
        }
    }
    return props;
}

```

首先加载settings下面的setting节点为property，然后检查所有属性,确保它们都在Configuration中已定义，而非未知的设置。注：MetaClass是一个保存对象定义比如getter/setter/构造器等的数据类,localReflectorFactory则是mybatis提供的默认反射工厂实现，这个ReflectorFactory主要采用了工厂类，其内部使用的Reflector采用了facade设计模式，简化反射的使用。如下所示：

```
public class MetaClass {

    private ReflectorFactory reflectorFactory;
    private Reflector reflector;

    private MetaClass(Class<?> type, ReflectorFactory reflectorFactory) {
        this.reflectorFactory = reflectorFactory;
        this.reflector = reflectorFactory.findForClass(type);
    }

    public static MetaClass forClass(Class<?> type, ReflectorFactory reflectorFactory) {
        return new MetaClass(type, reflectorFactory);
    }
    ...
}

@Override
public Reflector findForClass(Class<?> type) {
    if (classCacheEnabled) {
        // synchronized (type) removed see issue #461
        Reflector cached = reflectorMap.get(type);
        if (cached == null) {
            cached = new Reflector(type);
            reflectorMap.put(type, cached);
        }
        return cached;
    } else {
        return new Reflector(type);
    }
}

public Reflector(Class<?> clazz) {
    type = clazz;
    addDefaultConstructor(clazz);
    addGetMethods(clazz);
    addSetMethods(clazz);
    addFields(clazz);
    readablePropertyNames = getMethods.keySet().toArray(new
String[getMethods.keySet().size()]);
    writeablePropertyNames = setMethods.keySet().toArray(new
String[setMethods.keySet().size()]);
    for (String propName : readablePropertyNames) {
        caseInsensitivePropertyMap.put(propName.toUpperCase(Locale.ENGLISH), propName);
    }
    for (String propName : writeablePropertyNames) {
        caseInsensitivePropertyMap.put(propName.toUpperCase(Locale.ENGLISH), propName);
    }
}
```

```
}  
}
```

得到setting之后，调用settingsElement(Properties props)将各值赋值给configuration，同时在这里有重新设置了默认值，所有这一点很重要，configuration中的默认值不一定是真正的默认值。

2.1.3 加载自定义VFS loadCustomVfs

VFS主要用来加载容器内的各种资源，比如jar或者class文件。mybatis提供了2个实现 JBoss6VFS 和 DefaultVFS，并提供了用户扩展点，用于自定义VFS实现，加载顺序是自定义VFS实现 > 默认VFS实现 取第一个加载成功的，默认情况下会先加载JBoss6VFS，如果classpath下找不到jboss的vfs实现才会加载默认VFS实现，启动打印的日志如下：

```
org.apache.ibatis.io.VFS.getClass(VFS.java:111) Class not found: org.jboss.vfs.VFS  
org.apache.ibatis.io.JBoss6VFS.setInvalid(JBoss6VFS.java:142) JBoss 6 VFS API is not available in this  
environment.  
org.apache.ibatis.io.VFS.getClass(VFS.java:111) Class not found: org.jboss.vfs.VirtualFile  
org.apache.ibatis.io.VFS$VFSHolder.createVFS(VFS.java:63) VFS implementation  
org.apache.ibatis.io.JBoss6VFS is not valid in this environment.  
org.apache.ibatis.io.VFS$VFSHolder.createVFS(VFS.java:77) Using VFS adapter  
org.apache.ibatis.io.DefaultVFS
```

jboss vfs的maven仓库坐标为：

```
<dependency>  
  <groupId>org.jboss</groupId>  
  <artifactId>jboss-vfs</artifactId>  
  <version>3.2.12.Final</version>  
</dependency>
```

找到jboss vfs实现后，输出的日志如下：

```
org.apache.ibatis.io.VFS$VFSHolder.createVFS(VFS.java:77) Using VFS adapter  
org.apache.ibatis.io.JBoss6VFS
```

2.1.4 解析类型别名typeAliasesElement

```
private void typeAliasesElement(XNode parent) {  
    if (parent != null) {  
        for (XNode child : parent.getChildren()) {  
            if ("package".equals(child.getName())) {  
                String typeAliasPackage = child.getStringAttribute("name");  
                configuration.getTypeAliasRegistry().registerAliases(typeAliasPackage);  
            } else {  
                String alias = child.getStringAttribute("alias");  
                String type = child.getStringAttribute("type");  
                try {  
                    Class<?> clazz = Resources.classForName(type);  
                    if (alias == null) {  
                        typeAliasRegistry.registerAlias(clazz);  
                    }  
                }  
            }  
        }  
    }  
}
```



```

        } else {
            typeAliasRegistry.registerAlias(alias, clazz);
        }
    } catch (ClassNotFoundException e) {
        throw new BuilderException("Error registering typeAlias for '" + alias +
            "'. Cause: " + e, e);
    }
}
}
}
}
}

```

从上述代码可以看出，mybatis主要提供两种类型的别名设置，具体类的别名以及包的别名设置。类型别名是为Java类型设置一个短的名字，存在的意义仅在于用来减少类完全限定名的冗余。

```

<typeAliases>
    <typeAlias alias="Blog" type="domain.blog.Blog"/>
</typeAliases>

```

当这样配置时，Blog可以用在任何使用domain.blog.Blog的地方。设置为package之后，MyBatis 会在包名下面搜索需要的Java Bean。如：

```

<typeAliases>
    <package name="domain.blog"/>
</typeAliases>

```

每一个在包domain.blog中的Java Bean，在没有注解的情况下，会使用Bean的首字母小写的非限定类名来作为它的别名。比如domain.blog.Author的别名为author；若有注解，则别名为其注解值。所以所有的别名，无论是内置的还是自定义的，都一开始被保存在configuration.typeAliasRegistry中了，这样就可以确保任何时候使用别名和FQN的效果是一样的。

2.1.5 加载插件pluginElement

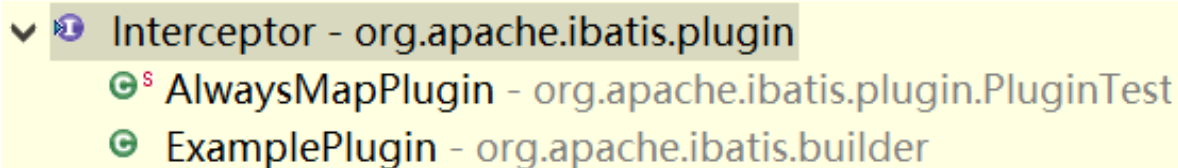
几乎所有优秀的框架都会预留插件体系以便扩展，mybatis调用pluginElement(root.evalNode("plugins"));加载mybatis插件，最常用的插件应该算是分页插件PageHelper了，再比如druid连接池提供的各种监控、拦截、预发检查功能，在使用其它连接池比如dbcp的时候，在不修改连接池源码的情况下，就可以借助mybatis的插件体系实现。加载插件的实现如下：

```

private void pluginElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            String interceptor = child.getStringAttribute("interceptor");
            Properties properties = child.getChildrenAsProperties();
            //将interceptor指定的名称解析为Interceptor类型
            Interceptor interceptorInstance = (Interceptor)
resolveClass(interceptor).newInstance();
            interceptorInstance.setProperties(properties);
            configuration.addInterceptor(interceptorInstance);
        }
    }
}

```

插件在具体实现的时候，采用的是拦截器模式，要注册为mybatis插件，必须实现org.apache.ibatis.plugin.Interceptor接口，每个插件可以有自己的属性。interceptor属性值既可以完整的类名，也可以是别名，只要别名在typealias中存在即可，如果启动时无法解析，会抛出ClassNotFoundException异常。实例化插件后，将设置插件的属性赋值给插件实现类的属性字段。mybatis提供了两个内置的插件例子，如下所示：



The screenshot shows a list of plugins in an IDE. The first plugin is 'AlwaysMapPlugin' with package 'org.apache.ibatis.plugin.PluginTest'. The second plugin is 'ExamplePlugin' with package 'org.apache.ibatis.builder'.

我们会在第6章详细讲解自定义插件的实现。

2.1.6 加载对象工厂 objectFactoryElement

什么是对象工厂？MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。默认的对象工厂DefaultObjectFactory做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。如下所示：

```

public class DefaultObjectFactory implements ObjectFactory, Serializable {

    private static final long serialVersionUID = -8855120656740914948L;

    @Override
    public <T> T create(Class<T> type) {
        return create(type, null, null);
    }

    @SuppressWarnings("unchecked")
    @Override
    public <T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object>
constructorArgs) {
        Class<?> classToCreate = resolveInterface(type);
        // we know types are assignable
        return (T) instantiateClass(classToCreate, constructorArgTypes, constructorArgs);
    }
}

```

```

@Override
public void setProperties(Properties properties) {
    // no props for default
}
...
protected Class<?> resolveInterface(Class<?> type) {
    Class<?> classToCreate;
    if (type == List.class || type == Collection.class || type == Iterable.class) {
        classToCreate = ArrayList.class;
    } else if (type == Map.class) {
        classToCreate = HashMap.class;
    } else if (type == SortedSet.class) { // issue #510 Collections Support
        classToCreate = TreeSet.class;
    } else if (type == Set.class) {
        classToCreate = HashSet.class;
    } else {
        classToCreate = type;
    }
    return classToCreate;
}
...
}

```

无论是创建集合类型、Map类型还是其他类型，都素hi同样的处理方式。如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。ObjectFactory 接口很简单，它包含两个创建用的方法，一个是处理默认构造方法的，另外一个处理带参数的构造方法的。最后，setProperties 方法可以被用来配置 ObjectFactory，在初始化你的 ObjectFactory 实例后，objectFactory元素体中定义的属性会被传递给setProperties方法。例如：







```

public class ExampleObjectFactory extends DefaultObjectFactory {
    public Object create(Class type) {
        return super.create(type);
    }
    public Object create(Class type, List<Class> constructorArgTypes, List<Object>
constructorArgs) {
        return super.create(type, constructorArgTypes, constructorArgs);
    }
    public void setProperties(Properties properties) {
        super.setProperties(properties);
    }
    public <T> boolean isCollection(Class<T> type) {
        return Collection.class.isAssignableFrom(type);
    }
}
<!-- mybatis-config.xml -->
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
    <property name="someProperty" value="100"/>
</objectFactory>

```

2.1.7 创建对象包装器工厂 objectWrapperFactoryElement

对象包装器工厂主要用来包装返回result对象，比如说可以用来设置某些敏感字段脱敏或者加密等。默认对象包装器工厂是DefaultObjectWrapperFactory，也就是不使用包装器工厂。既然看到包装器工厂，我们就得看下对象包装器，如下：

- ▼  ObjectWrapper - org.apache.ibatis.reflection.wrapper
 - ▼  BaseWrapper - org.apache.ibatis.reflection.wrapper
 -  BeanWrapper - org.apache.ibatis.reflection.wrapper
 -  MapWrapper - org.apache.ibatis.reflection.wrapper
 -  CollectionWrapper - org.apache.ibatis.reflection.wrapper
 -  CustomObjectWrapper - org.apache.ibatis.submitted.custom_collection_handling

BeanWrapper是BaseWrapper的默认实现。其中的两个关键接口是getBeanProperty和setBeanProperty，它们是实现包装的主要位置：

```
private Object getBeanProperty(PropertyTokenizer prop, Object object) {
    try {
        Invoker method = metaClass.getGetInvoker(prop.getName());
        try {
            return method.invoke(object, NO_ARGUMENTS);
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
    } catch (RuntimeException e) {
        throw e;
    } catch (Throwable t) {
        throw new ReflectionException("Could not get property '" + prop.getName() + "'
from " + object.getClass() + ". Cause: " + t.toString(), t);
    }
}

private void setBeanProperty(PropertyTokenizer prop, Object object, Object value) {
    try {
        Invoker method = metaClass.getSetInvoker(prop.getName());
        Object[] params = {value};
        try {
            method.invoke(object, params);
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
    } catch (Throwable t) {
        throw new ReflectionException("Could not set property '" + prop.getName() + "' of
'" + object.getClass() + "' with value '" + value + "' Cause: " + t.toString(), t);
    }
}
```

要实现自定义的对象包装器工厂，只要实现ObjectWrapperFactory中的两个接口hasWrapperFor和getWrapperFor即可，如下：

```

public class CustomBeanWrapperFactory implements ObjectWrapperFactory {
    @Override
    public boolean hasWrapperFor(Object object) {
        if (object instanceof Author) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public ObjectWrapper getWrapperFor(MetaObject metaObject, Object object) {
        return new CustomBeanWrapper(metaObject, object);
    }
}

```

2.1.8 加载反射工厂reflectorFactoryElement

因为加载配置文件中的各种插件类等等，为了更好的灵活性，mybatis支持用户自定义反射工厂，不过总体来说，用的不多，要实现反射工厂，只要实现ReflectorFactory接口即可。默认反射工厂是DefaultReflectorFactory。一般来说，使用默认的反射工厂就可以了。

2.1.9 加载环境配置environmentsElement

环境可以说是mybatis-config配置文件中最重要的一部分，它类似于spring和maven里面的profile，允许给开发、生产环境同时配置不同的environment，根据不同的环境加载不同的配置，这也是常见的做法，如果在SqlSessionFactoryBuilder调用期间没有传递使用哪个环境的话，默认会使用一个名为default的环境。找到对应的environment之后，就可以加载事务管理器和数据源了。事务管理器和数据源类型这里都用到了类型别名，JDBC/POOLED都是在mybatis内置提供的，在Configuration构造器执行期间注册到TypeAliasRegister。

mybatis内置提供JDBC和MANAGED两种事务管理方式，前者主要用于简单JDBC模式，后者主要用于容器管理事务，一般使用JDBC事务管理方式。mybatis内置提供JNDI、POOLED、UNPOOLED三种数据源工厂，一般情况下使用POOLED数据源。

```

<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <property name="url" value="jdbc:mysql://10.7.12.4:3306/lfBase?
useUnicode=true"/>
            <property name="username" value="lfBase"/>
            <property name="password" value="eKffQV6wbh3sfQuFIG6M"/>
        </dataSource>
    </environment>
</environments>

private void environmentsElement(XNode context) throws Exception {
    if (context != null) {

```

```

    if (environment == null) {
        environment = context.getStringAttribute("default");
    }
    for (XNode child : context.getChildren()) {
        String id = child.getStringAttribute("id");
        //查找匹配的environment
        if (isSpecifiedEnvironment(id)) {
            // 事务配置并创建事务工厂
            TransactionFactory txFactory =
transactionManagerElement(child.evalNode("transactionManager"));
            // 数据源配置加载并实例化数据源，数据源是必备的
            DataSourceFactory dsFactory =
dataSourceElement(child.evalNode("dataSource"));
            DataSource dataSource = dsFactory.getDataSource();
            // 创建Environment.Builder
            Environment.Builder environmentBuilder = new Environment.Builder(id)
                .transactionFactory(txFactory)
                .dataSource(dataSource);
            configuration.setEnvironment(environmentBuilder.build());
        }
    }
}
}
}
}

```

2.1.10 数据库厂商标识加载databaseIdProviderElement

MyBatis 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 databaseId 属性。MyBatis 会加载不带 databaseId 属性和带有匹配当前数据库 databaseId 属性的所有语句。如果同时找到带有 databaseId 和不带 databaseId 的相同语句，则后者会被舍弃。为支持多厂商特性只要像下面这样在 mybatis-config.xml 文件中加入 databaseIdProvider 即可：

```
<databaseIdProvider type="DB_VENDOR" />
```

这里的 DB_VENDOR 会通过 DatabaseMetaData#getDatabaseProductName() 返回的字符串进行设置。由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短，如下：

```

<databaseIdProvider type="DB_VENDOR">
    <property name="SQL Server" value="sqlserver" />
    <property name="MySQL" value="mysql" />
    <property name="Oracle" value="oracle" />
</databaseIdProvider>

```

在有 properties 时，DB_VENDOR databaseIdProvider 的将被设置为第一个能匹配数据库产品名称的属性键对应的值，如果没有匹配的属性将会设置为“null”。

因为每个数据库在实现的时候，getDatabaseProductName() 返回的通常并不是直接的 Oracle 或者 MySQL，而是“Oracle (DataDirect)”，所以如果希望使用多数据库特性，一般要实现 org.apache.ibatis.mapping.DatabaseIdProvider 接口 并在 mybatis-config.xml 中注册来构建自己的

DatabaseIdProvider:

```
public interface DatabaseIdProvider {
    void setProperties(Properties p);
    String getDatabaseId(DataSource dataSource) throws SQLException;
}
```

典型的实现比如:

```
public class VendorDatabaseIdProvider implements DatabaseIdProvider {

    private static final Log log = LogFactory.getLog(VendorDatabaseIdProvider.class);

    private Properties properties;

    @Override
    public String getDatabaseId(DataSource dataSource) {
        if (dataSource == null) {
            throw new NullPointerException("dataSource cannot be null");
        }
        try {
            return getDatabaseName(dataSource);
        } catch (Exception e) {
            log.error("Could not get a databaseId from dataSource", e);
        }
        return null;
    }
    ...
    private String getDatabaseName(DataSource dataSource) throws SQLException {
        String productName = getDatabaseProductName(dataSource);
        if (this.properties != null) {
            for (Map.Entry<Object, Object> property : properties.entrySet()) {
                // 只要包含productName中包含了property名称,就算匹配,而不是使用精确匹配
                if (productName.contains((String) property.getKey())) {
                    return (String) property.getValue();
                }
            }
            // no match, return null
            return null;
        }
        return productName;
    }

    private String getDatabaseProductName(DataSource dataSource) throws SQLException {
        Connection con = null;
        try {
            con = dataSource.getConnection();
            DatabaseMetaData metaData = con.getMetaData();
        }
    }
}
```

```

        return metaData.getDatabaseProductName();
    } finally {
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                // ignored
            }
        }
    }
}
}
}
}
}

```

2.1.11 加载类型处理器typeHandlerElement

```

private void typeHandlerElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            if ("package".equals(child.getName())) {
                String typeHandlerPackage = child.getStringAttribute("name");
                typeHandlerRegistry.register(typeHandlerPackage);
            } else {
                String javaTypeName = child.getStringAttribute("javaType");
                String jdbcTypeName = child.getStringAttribute("jdbcType");
                String handlerTypeName = child.getStringAttribute("handler");
                Class<?> javaTypeClass = resolveClass(javaTypeName);
                JdbcType jdbcType = resolveJdbcType(jdbcTypeName);
                Class<?> typeHandlerClass = resolveClass(handlerTypeName);
                if (javaTypeClass != null) {
                    if (jdbcType == null) {
                        typeHandlerRegistry.register(javaTypeClass, typeHandlerClass);
                    } else {
                        typeHandlerRegistry.register(javaTypeClass, jdbcType, typeHandlerClass);
                    }
                } else {
                    typeHandlerRegistry.register(typeHandlerClass);
                }
            }
        }
    }
}
}
}
}
}

```

无论是 MyBatis 在预处理语句 (PreparedStatement) 中设置一个参数时, 还是从结果集中取出一个值时, 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。

mybatis提供了两种方式注册类型处理器, package自动检索方式和显示定义方式。使用自动检索 (autodiscovery) 功能的时候, 只能通过注解方式来指定 JDBC 的类型。

```
<!-- mybatis-config.xml -->
```



```

<typeHandlers>
  <package name="org.mybatis.example"/>
</typeHandlers>

public void register(String packageName) {
    ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<Class<?>>();
    resolverUtil.find(new ResolverUtil.IsA(TypeHandler.class), packageName);
    Set<Class<? extends Class<?>>> handlerSet = resolverUtil.getClasses();
    for (Class<?> type : handlerSet) {
        //Ignore inner classes and interfaces (including package-info.java) and abstract
        classes
        if (!type.isAnonymousClass() && !type.isInterface() &&
!Modifier.isAbstract(type.getModifiers())) {
            register(type);
        }
    }
}

```

为了简化使用，mybatis在初始化TypeHandlerRegistry期间，自动注册了大部分的常用的类型处理器比如字符串、数字、日期等。对于非标准的类型，用户可以自定义类型处理器来处理。要实现一个自定义类型处理器，只要实现 org.apache.ibatis.type.TypeHandler 接口，或继承一个实用类 org.apache.ibatis.type.BaseTypeHandler，并将它映射到一个 JDBC 类型即可。例如：

```

@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String> {

    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, String parameter,
JdbcType jdbcType) throws SQLException {
        ps.setString(i, parameter);
    }

    @Override
    public String getNullableResult(ResultSet rs, String columnName) throws SQLException
    {
        return rs.getString(columnName);
    }

    @Override
    public String getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
        return rs.getString(columnIndex);
    }

    @Override
    public String getNullableResult(CallableStatement cs, int columnIndex) throws
SQLException {
        return cs.getString(columnIndex);
    }
}

<!-- mybatis-config.xml -->

```

```
<typeHandlers>
  <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

使用这个的类型处理器将会覆盖已经存在的处理 Java 的 String 类型属性和 VARCHAR 参数及结果的类型处理器。要注意 MyBatis 不会窥探数据库元信息来决定使用哪种类型，所以你必须要在参数和结果映射中指明那是 VARCHAR 类型的字段， 以使其能够绑定到正确的类型处理器上。这是因为：MyBatis 直到语句被执行才清楚数据类型。

通过类型处理器的泛型，MyBatis 可以得知该类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

- 在类型处理器的配置元素（typeHandler element）上增加一个 javaType 属性（比如：javaType="String"）；
 - 在类型处理器的类上（TypeHandler class）增加一个 @MappedTypes 注解来指定与其关联的 Java 类型列表。如果在 javaType 属性中也同时指定，则注解方式将被忽略。
- 可以通过两种方式来指定被关联的 JDBC 类型：

1. 在类型处理器的配置元素上增加一个 jdbcType 属性（比如：jdbcType="VARCHAR"）；
2. 在类型处理器的类上（TypeHandler class）增加一个 @MappedJdbcTypes 注解来指定与其关联的 JDBC 类型列表。如果在两个位置同时指定，则注解方式将被忽略。

当决定在 ResultMap 中使用某一 TypeHandler 时，此时 java 类型是已知的（从结果类型中获得），但是 JDBC 类型是未知的。因此 Mybatis 使用 javaType=[TheJavaType], jdbcType=null 的组合来选择一个 TypeHandler。这意味着使用 @MappedJdbcTypes 注解可以限制 TypeHandler 的范围，同时除非显示的设置，否则 TypeHandler 在 ResultMap 中将是无效的。如果希望在 ResultMap 中使用 TypeHandler，那么设置 @MappedJdbcTypes 注解的 includeNullJdbcType=true 即可。然而从 Mybatis 3.4.0 开始，如果只有一个注册的 TypeHandler 来处理 java 类型，那么它将是 ResultMap 使用 Java 类型时的默认值（即使没有 includeNullJdbcType=true）。

还可以创建一个泛型类型处理器，它可以处理多于一个类。为达到此目的， 需要增加一个接收该类作为参数的构造器，这样在构造一个类型处理器的时候 MyBatis 就会传入一个具体的类。

```
public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

    private Class<E> type;

    public GenericTypeHandler(Class<E> type) {
        if (type == null) throw new IllegalArgumentException("Type argument cannot be null");
        this.type = type;
    }
}
```

我们映射枚举使用的 EnumTypeHandler 和 EnumOrdinalTypeHandler 都是泛型类型处理器（generic TypeHandlers）。

处理枚举类型映射

若想映射枚举类型 Enum，则需要从 EnumTypeHandler 或者 EnumOrdinalTypeHandler 中选一个来使用。

比如说我们想存储取近似值时用到的舍入模式。默认情况下，MyBatis 会利用 EnumTypeHandler 来把 Enum 值转换成对应的名字。

注意 EnumTypeHandler 在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了 Enum 的类。

不过，我们可能不想存储名字，相反我们的 DBA 会坚持使用整形值代码。那也一样轻而易举：在配置文件中把 EnumOrdinalTypeHandler 加到 typeHandlers 中即可，这样每个 RoundingMode 将通过他们的序数值来映射成对应的整形。

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
javaType="java.math.RoundingMode"/>
</typeHandlers>
```

但是怎样能将同样的 Enum 既映射成字符串又映射成整形呢？

自动映射器（auto-mapper）会自动地选用 EnumOrdinalTypeHandler 来处理，所以如果我们想用普通的 EnumTypeHandler，就需要为那些 SQL 语句显式地设置要用到的类型处理器。比如：

```
<result column="roundingMode" property="roundingMode"
typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>
```

2.1.12 加载 mapper 文件 mapperElement

mapper 文件是 mybatis 框架的核心之处，所有的用户 sql 语句都编写在 mapper 文件中，所以理解 mapper 文件对于所有的开发人员来说都是必备的要求。

```
private void mapperElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            // 如果要同时使用 package 自动扫描和通过 mapper 明确指定要加载的 mapper，一定要确保 package 自动扫描的范围不包含明确指定的 mapper，否则在通过 package 扫描的 interface 的时候，尝试加载对应 xml 文件的 loadXmlResource() 的逻辑中出现判重出错，报 org.apache.ibatis.binding.BindingException 异常，即使 xml 文件中包含的内容和 mapper 接口中包含的语句不重复也会出错，包括加载 mapper 接口时自动加载的 xml mapper 也一样会出错。
            if ("package".equals(child.getName())) {
                String mapperPackage = child.getStringAttribute("name");
                configuration.addMappers(mapperPackage);
            } else {
                String resource = child.getStringAttribute("resource");
                String url = child.getStringAttribute("url");
                String mapperClass = child.getStringAttribute("class");
                if (resource != null && url == null && mapperClass == null) {
                    ErrorContext.instance().resource(resource);
                    InputStream inputStream = Resources.getResourceAsStream(resource);
```

```

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

```

mybatis提供了两类配置mapper的方法，第一类是使用package自动搜索的模式，这样指定package下所有接口都会被注册为mapper，例如：

```

<mappers>
  <package name="org.mybatis.builder"/>
</mappers>

```

另外一类是明确指定mapper，这又可以通过resource、url或者class进行细分。例如：

```

<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>

```

需要注意的是，如果要同时使用package自动扫描和通过mapper明确指定要加载的mapper，则必须先声明mapper，然后声明package，否则DTD校验会失败。同时一定要确保package自动扫描的范围不包含明确指定的mapper，否则在通过package扫描的interface的时候，尝试加载对应xml文件的loadXmlResource()的逻辑中出现判重出错，报org.apache.ibatis.binding.BindingException异常。

对于通过package加载的mapper文件，调用mapperRegistry.addMappers(packageName);进行加载，其核心逻辑在org.apache.ibatis.binding.MapperRegistry中，对于每个找到的接口或者mapper文件，最后调用XMLMapperBuilder进行具体解析。
































对于明确指定的mapper文件或者mapper接口，则主要使用XMLMapperBuilder进行具体解析。

下一章节，我们进行详细说明mapper文件的解析加载与初始化。

2.2 mapper加载与初始化

前面说过mybatis mapper文件的加载主要有两大类，通过package加载和明确指定的方式。

一般来说，对于简单语句来说，使用注解代码会更加清晰，然而Java注解对于复杂语句比如同时包含了构造器、鉴别器、resultMap来说就会非常混乱，应该限制使用，此时应该使用XML文件，因为注解至少至今为止不像XML/Gradle一样能够很好的表示嵌套关系。mybatis完整的注解列表以及含义可参考<http://www.mybatis.org/mybatis-3/java-api.html>或者<http://blog.51cto.com/computerdragon/1399742>或者源码org.apache.ibatis.annotations包：

- ▼  org.apache.ibatis.annotations
 - >  Arg.java
 - >  AutomapConstructor.java
 - >  CacheNamespace.java
 - >  CacheNamespaceRef.java
 - >  Case.java
 - >  ConstructorArgs.java
 - >  Delete.java
 - >  DeleteProvider.java
 - >  Flush.java
 - >  Insert.java
 - >  InsertProvider.java
 - >  Lang.java
 - >  Many.java
 - >  MapKey.java
 - >  Mapper.java
 - >  One.java
 - >  Options.java
 - >  package-info.java
 - >  Param.java
 - >  Property.java
 - >  Result.java
 - >  ResultMap.java
 - >  Results.java
 - >  ResultType.java
 - >  Select.java
 - >  SelectKey.java
 - >  SelectProvider.java
 - >  TypeDiscriminator.java
 - >  Update.java
 - >  UpdateProvider.java

为了更好的理解上下文语义，建议读者对XML配置对应的注解先了解，这样看起源码来会更加顺畅。我们先来回顾一下通过注解配置的典型mapper接口：

```
@Select("select *from User where id=#{id} and userName like #{name}")
```

```

public User retrieveUserByIdAndName(@Param("id")int id,@Param("name")String names);

@Insert("INSERT INTO user(userName,userAge,userAddress) VALUES(#{userName},#{userAge},#{userAddress})")
public void addNewUser(User user);

@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")
@SelectKey(statement="call next value for TestSequence", keyProperty="nameId",
before=true, resultType=int.class)
int insertTable3(Name name);
@Results(id = "userResult", value = {
    @Result(property = "id", column = "uid", id = true),
    @Result(property = "firstName", column = "first_name"),
    @Result(property = "lastName", column = "last_name")
})
@TypeDiscriminator(column = "type",
    cases={
        @Case(value="1",type=RegisterEmployee.class,results={
            @Result(property="salary")
        }),
        @Case(value="2",type=TimeEmployee.class,results={
            @Result(property="time")
        })
    }
)
@Select("select * from users where id = #{id}")
User getUserById(Integer id);

@Results(id = "companyResults")
@ConstructorArgs({
    @Arg(property = "id", column = "cid", id = true),
    @Arg(property = "name", column = "name")
})
@Select("select * from company where id = #{id}")
Company getCompanyById(Integer id);

@ResultMap(id = "xmlUserResults")
@SelectProvider(type = UserSqlBuilder.class, method = "buildGetUsersByName")
List<User> getUsersByName(String name);

```

// 注：建议尽可能避免使用SqlBuild的模式生成的,如果因为功能需要必须动态生成SQL的话，也是直接写SQL拼接返回，而不是一堆类似SELECT()、FROM()的函数调用，这只会让维护成为噩梦，这思路的设计者不是知道怎么想的，此处仅用于演示XXXProvider功能，但是XXXProvider模式本身的设计在关键时候还是比较清晰的。

```

class UserSqlBuilder {
    public String buildGetUsersByName(final String name) {
        return new SQL(){
            SELECT("*");
            FROM("users");
            if (name != null) {

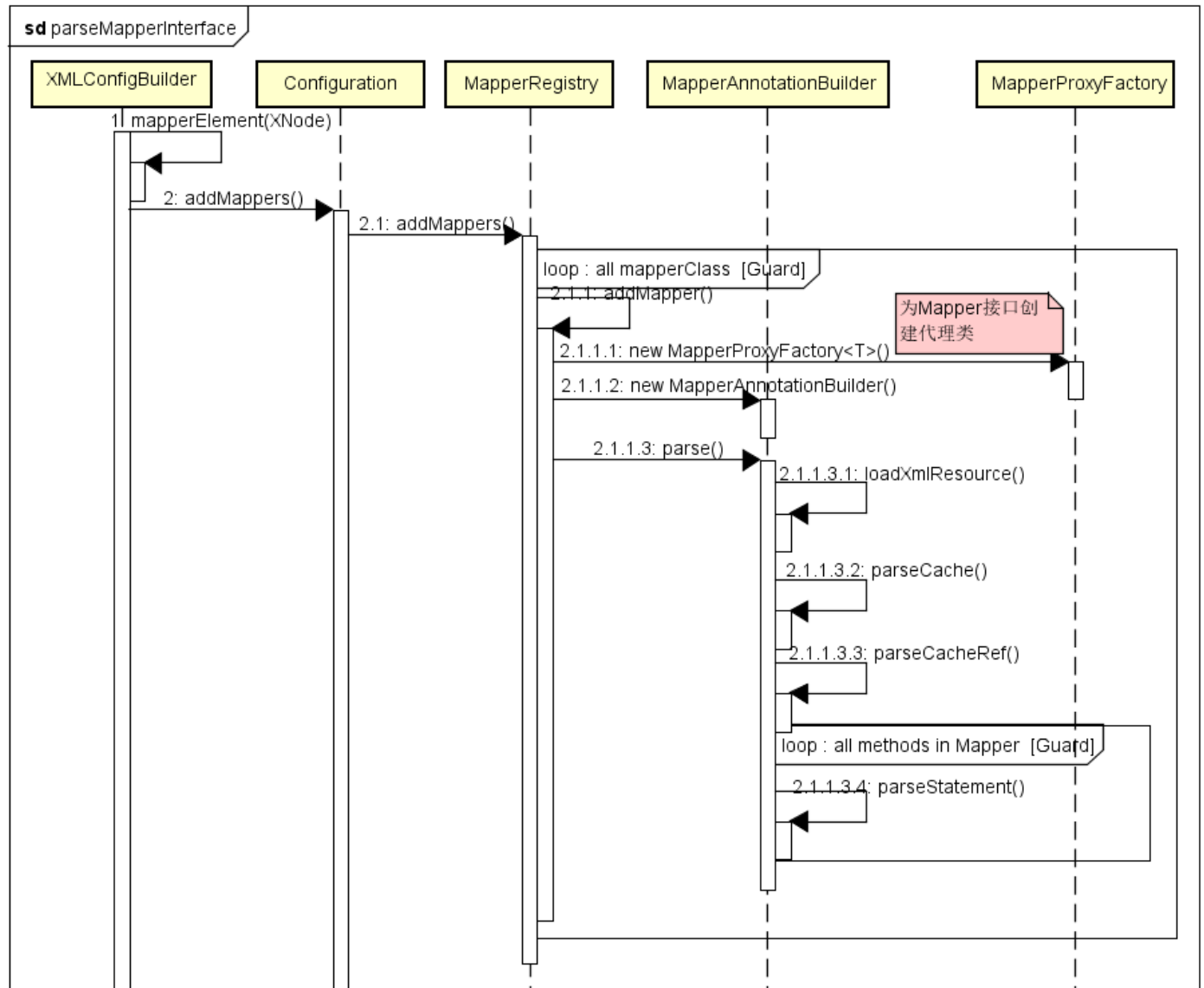
```

```

        WHERE("name like #{value} || '%'");
    }
    ORDER_BY("id");
  }}.toString();
}
}
}

```

我们先来看通过package自动搜索加载的方式，它的范围由addMappers的参数packageName指定的包名以及父类superType确定，其整体流程如下：



```

/**
 * @since 3.2.2
 */
public void addMappers(String packageName, Class<?> superType) {
    // mybatis框架提供的搜索classpath下指定package以及子package中符合条件(注解或者继承于某个类/接口)的类，默认使用Thread.currentThread().getContextClassLoader()返回的加载器，和spring的工具类殊途同归。
    ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<Class<?>>();
    // 无条件的加载所有的类，因为调用方传递了Object.class作为父类，这也给以后的指定mapper接口预留了余地
}

```



```

resolverUtil.find(new ResolverUtil.IsA(superType), packageName);
// 所有匹配的calss都被存储在ResolverUtil.matches字段中
Set<Class<? extends Class<?>>> mapperSet = resolverUtil.getClasses();
for (Class<?> mapperClass : mapperSet) {
    //调用addMapper方法进行具体的mapper类/接口解析
    addMapper(mapperClass);
}
}

/**
 * 外部调用的入口
 * @since 3.2.2
 */
public void addMappers(String packageName) {
    addMappers(packageName, Object.class);
}

public <T> void addMapper(Class<T> type) {
    // 对于mybatis mapper接口文件, 必须是interface, 不能是class
    if (type.isInterface()) {
        // 判重, 确保只会加载一次不会被覆盖
        if (hasMapper(type)) {
            throw new BindingException("Type " + type + " is already known to the
MapperRegistry.");
        }
        boolean loadCompleted = false;
        try {
            // 为mapper接口创建一个MapperProxyFactory代理
            knownMappers.put(type, new MapperProxyFactory<T>(type));
            // It's important that the type is added before the parser is run
            // otherwise the binding may automatically be attempted by the
            // mapper parser. If the type is already known, it won't try.
            MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);
            parser.parse();
            loadCompleted = true;
        } finally {
            //剔除解析出现异常的接口
            if (!loadCompleted) {
                knownMappers.remove(type);
            }
        }
    }
}
}

```

knownMappers是MapperRegistry的主要字段, 维护了Mapper接口和代理类的映射关系,key是mapper接口类, value是MapperProxyFactory, 其定义如下:

```

private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<Class<?>, MapperProxyFactory<?>>();

```

```

public class MapperProxyFactory<T> {

    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method,
MapperMethod>();

    public MapperProxyFactory(Class<T> mapperInterface) {
        this.mapperInterface = mapperInterface;
    }

    public Class<T> getMapperInterface() {
        return mapperInterface;
    }

    public Map<Method, MapperMethod> getMethodCache() {
        return methodCache;
    }

    @SuppressWarnings("unchecked")
    protected T newInstance(MapperProxy<T> mapperProxy) {
        return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] {
mapperInterface }, mapperProxy);
    }

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

}

```

从定义看出，MapperProxyFactory主要是维护mapper接口的方法与对应mapper文件中具体CRUD节点的关联关系。其中每个Method与对应MapperMethod维护在一起。MapperMethod是mapper中具体映射语句节点的内部表示。

首先为mapper接口创建MapperProxyFactory，然后创建MapperAnnotationBuilder进行具体的解析，MapperAnnotationBuilder在解析前的构造器中完成了下列工作：

```

public MapperAnnotationBuilder(Configuration configuration, Class<?> type) {
    String resource = type.getName().replace('.', '/') + ".java (best guess)";
    this.assistant = new MapperBuilderAssistant(configuration, resource);
    this.configuration = configuration;
    this.type = type;

    sqlAnnotationTypes.add(Select.class);
    sqlAnnotationTypes.add(Insert.class);
    sqlAnnotationTypes.add(Update.class);
    sqlAnnotationTypes.add>Delete.class);
}

```

```

sqlProviderAnnotationTypes.add(SelectProvider.class);
sqlProviderAnnotationTypes.add(InsertProvider.class);
sqlProviderAnnotationTypes.add(UpdateProvider.class);
sqlProviderAnnotationTypes.add>DeleteProvider.class);
}

```

其中的MapperBuilderAssistant和XMLConfigBuilder一样，都是继承于BaseBuilder。
 Select.class/Insert.class等注解指示该方法对应的真实sql语句类型分别是select/insert。
 SelectProvider.class/InsertProvider.class主要用于动态SQL，它们允许你指定一个类名和一个方法在具体执行时返回要运行的SQL语句。MyBatis会实例化这个类，然后执行指定的方法。

MapperBuilderAssistant初始化完成之后，就调用build.parse()进行具体的mapper接口文件加载与解析，如下所示：

```

public void parse() {
    String resource = type.toString();
    //首先根据mapper接口的字符串表示判断是否已经加载,避免重复加载,正常情况下应该都没有加载
    if (!configuration.isResourceLoaded(resource)) {
        loadXmlResource();
        configuration.addLoadedResource(resource);
        // 每个mapper文件自成一个namespace, 通常自动匹配就是这么来的, 约定俗成代替人工设置最简化常见的
        开发
        assistant.setCurrentNamespace(type.getName());
        parseCache();
        parseCacheRef();
        Method[] methods = type.getMethods();
        for (Method method : methods) {
            try {
                // issue #237
                if (!method.isBridge()) {
                    parseStatement(method);
                }
            } catch (IncompleteElementException e) {
                configuration.addIncompleteMethod(new MethodResolver(this, method));
            }
        }
    }
    parsePendingMethods();
}

```

整体流程为：

- 1、首先加载mapper接口对应的xml文件并解析。loadXmlResource和通过resource、url解析相同，都是解析mapper文件中的定义，他们的入口都是XMLMapperBuilder.parse()，我们稍等会儿专门专门分析，这一节先来看通过注解方式配置的mapper的解析（注：对于一个mapper接口,不能同时使用注解方式和xml方式,任何时候只能之一,但是不同的mapper接口可以混合使用这两种方式）。

- 2、解析缓存注解；

mybatis中缓存注解的定义为：

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface CacheNamespace {
    Class<? extends org.apache.ibatis.cache.Cache> implementation() default
    PerpetualCache.class;

    Class<? extends org.apache.ibatis.cache.Cache> eviction() default LruCache.class;

    long flushInterval() default 0;

    int size() default 1024;

    boolean readWrite() default true;

    boolean blocking() default false;

    /**
     * Property values for a implementation object.
     * @since 3.4.2
     */
    Property[] properties() default {};
}

```

从上面的定义可以看出，和在XML中的是一一对应的。缓存的解析很简单，这里不展开细细讲解。

3、解析缓存参照注解。缓存参考的解析也很简单，这里不展开细细讲解。

4、解析非桥接方法。在正式开始之前，我们先来看下什么是桥接方法。桥接方法是JDK 1.5引入泛型后，为了使Java的泛型方法生成的字节码和1.5版本前的字节码相兼容，由编译器自动生成的方法。那什么时候，编译器会生成桥接方法呢，举个例子，一个子类在继承（或实现）一个父类（或接口）的泛型方法时，在子类中明确指定了泛型类型，那么在编译时编译器会自动生成桥接方法。参考：

<http://blog.csdn.net/mhmyqn/article/details/47342577>

<https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.6>

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.4.5>

所以正常情况下，只要在实现mybatis mapper接口的时候，没有继承根Mapper或者继承了根Mapper但是没有写死泛型类型的时候，是不会成为桥接方法的。现在来看parseStatement的主要实现代码(提示:因为注解方式通常不用于复杂的配置,所以这里我们进行简单的解析，在XML部分进行详细说明):

```

void parseStatement(Method method) {
    // 获取参数类型,如果有多个参数,这种情况下就返回
    org.apache.ibatis.binding.MapperMethod.ParamMap.class, ParamMap是一个继承于HashMap的类, 否则返回实际类型
    Class<?> parameterTypeClass = getParameterType(method);
    // 获取语言驱动器
    LanguageDriver languageDriver = getLanguageDriver(method);
}

```

```

// 获取方法的SqlSource对象,只有指定了@Select/@Insert/@Update/@Delete或者对应的Provider的
方法才会被当作mapper,否则只是和mapper文件中对应语句的一个运行时占位符
SqlSource sqlSource = getSqlSourceFromAnnotations(method, parameterTypeClass,
languageDriver);
if (sqlSource != null) {

// 获取方法的属性设置, 对应<select>中的各种属性
Options options = method.getAnnotation(Options.class);

final String mappedStatementId = type.getName() + "." + method.getName();
Integer fetchSize = null;
Integer timeout = null;
StatementType statementType = StatementType.PREPARED;
ResultSetType resultSetType = ResultSetType.FORWARD_ONLY;

// 获取语句的CRUD类型
SqlCommandType sqlCommandType = getSqlCommandType(method);
boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
boolean flushCache = !isSelect;
boolean useCache = isSelect;

KeyGenerator keyGenerator;
String keyProperty = "id";
String keyColumn = null;

// 只有INSERT/UPDATE才解析SelectKey选项,总体来说, 它的实现逻辑和XML基本一致, 这里不展开详述
if (SqlCommandType.INSERT.equals(sqlCommandType) ||
SqlCommandType.UPDATE.equals(sqlCommandType)) {
// first check for SelectKey annotation - that overrides everything else
SelectKey selectKey = method.getAnnotation(SelectKey.class);
if (selectKey != null) {
keyGenerator = handleSelectKeyAnnotation(selectKey, mappedStatementId,
getParameterType(method), languageDriver);
keyProperty = selectKey.keyProperty();
} else if (options == null) {
keyGenerator = configuration.isUseGeneratedKeys() ?
Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
} else {
keyGenerator = options.useGeneratedKeys() ? Jdbc3KeyGenerator.INSTANCE :
NoKeyGenerator.INSTANCE;
keyProperty = options.keyProperty();
keyColumn = options.keyColumn();
}
} else {
keyGenerator = NoKeyGenerator.INSTANCE;
}

if (options != null) {
if (FlushCachePolicy.TRUE.equals(options.flushCache())) {

```

```

        flushCache = true;
    } else if (FlushCachePolicy.FALSE.equals(options.flushCache())) {
        flushCache = false;
    }
    useCache = options.useCache();
    fetchSize = options.fetchSize() > -1 || options.fetchSize() ==
Integer.MIN_VALUE ? options.fetchSize() : null; //issue #348
    timeout = options.timeout() > -1 ? options.timeout() : null;
    statementType = options.statementType();
    resultSetType = options.resultSetType();
}

```

```
String resultMapId = null;
```

```
// 解析@ResultMap注解,如果有@ResultMap注解,就是用它, 否则才解析@Results
```

// @ResultMap注解用于给@Select和@SelectProvider注解提供在xml配置的<resultMap>,如果一个方法上同时出现@Results或者@ConstructorArgs等和结果映射有关的注解,那么@ResultMap会覆盖后面两者的注解

```

ResultMap resultMapAnnotation = method.getAnnotation(ResultMap.class);
if (resultMapAnnotation != null) {
    String[] resultMaps = resultMapAnnotation.value();
    StringBuilder sb = new StringBuilder();
    for (String resultMap : resultMaps) {
        if (sb.length() > 0) {
            sb.append(",");
        }
        sb.append(resultMap);
    }
    resultMapId = sb.toString();
} else if (isSelect) {
    //如果是查询, 且没有明确设置ResultMap, 则根据返回类型自动解析生成ResultMap
    resultMapId = parseResultMap(method);
}

```

```

assistant.addMappedStatement(
    mappedStatementId,
    sqlSource,
    statementType,
    sqlCommandType,
    fetchSize,
    timeout,
    // ParameterMapID
    null,
    parameterTypeClass,
    resultMapId,
    getReturnType(method),
    resultSetType,
    flushCache,
    useCache,

```

```

        // TODO gcode issue #577
        false,
        keyGenerator,
        keyProperty,
        keyColumn,
        // DatabaseID
        null,
        languageDriver,
        // ResultSets
        options != null ? nullOrEmpty(options.resultSets()) : null);
    }
}

```

重点来看没有带@ResultMap注解的查询方法parseResultMap(Method):

```

private String parseResultMap(Method method) {
    // 获取方法的返回类型
    Class<?> returnType = getReturnType(method);
    // 获取构造器
    ConstructorArgs args = method.getAnnotation(ConstructorArgs.class);

    // 获取@Results注解,也就是注解形式的结果映射
    Results results = method.getAnnotation(Results.class);

    // 获取鉴别器
    TypeDiscriminator typeDiscriminator =
method.getAnnotation(TypeDiscriminator.class);

    // 产生resultMapId
    String resultMapId = generateResultMapName(method);
    applyResultMap(resultMapId, returnType, argsIf(args), resultsIf(results),
typeDiscriminator);
    return resultMapId;
}

// 如果有resultMap设置了Id, 就直接返回类名.resultMapId. 否则返回类名.方法名.以-分隔拼接的方法参数
private String generateResultMapName(Method method) {
    Results results = method.getAnnotation(Results.class);

    if (results != null && !results.id().isEmpty()) {
        return type.getName() + "." + results.id();
    }
    StringBuilder suffix = new StringBuilder();
    for (Class<?> c : method.getParameterTypes()) {
        suffix.append("-");
        suffix.append(c.getSimpleName());
    }
    if (suffix.length() < 1) {

```

```

        suffix.append("-void");
    }
    return type.getName() + "." + method.getName() + suffix;
}

private void applyResultMap(String resultMapId, Class<?> returnType, Arg[] args,
Result[] results, TypeDiscriminator discriminator) {
    List<ResultMapping> resultMappings = new ArrayList<ResultMapping>();
    applyConstructorArgs(args, returnType, resultMappings);
    applyResults(results, returnType, resultMappings);
    Discriminator disc = applyDiscriminator(resultMapId, returnType, discriminator);
    // TODO add AutoMappingBehaviour
    assistant.addResultMap(resultMapId, returnType, null, disc, resultMappings, null);
    createDiscriminatorResultMaps(resultMapId, returnType, discriminator);
}

private void createDiscriminatorResultMaps(String resultMapId, Class<?> resultType,
TypeDiscriminator discriminator) {
    if (discriminator != null) {
        // 对于鉴别器来说, 和XML配置的差别在于xml中可以外部公用的resultMap,在注解中, 则只提供了内嵌式的resultMap定义
        for (Case c : discriminator.cases()) {
            // 从内部实现的角度, 因为内嵌式的resultMap定义也会创建resultMap, 所以XML的实现也一样, 对于内嵌式鉴别器每个分支resultMap, 其命名为映射方法的resultMapId-Case.value()。这样在运行时, 只要知道resultMap中包含了鉴别器之后, 获取具体的鉴别器映射就很简单了, map.get()一下就得到了。
            String caseResultMapId = resultMapId + "-" + c.value();
            List<ResultMapping> resultMappings = new ArrayList<ResultMapping>();
            // issue #136
            applyConstructorArgs(c.constructArgs(), resultType, resultMappings);
            applyResults(c.results(), resultType, resultMappings);
            // TODO add AutoMappingBehaviour
            assistant.addResultMap(caseResultMapId, c.type(), resultMapId, null,
resultMappings, null);
        }
    }
}

private Discriminator applyDiscriminator(String resultMapId, Class<?> resultType,
TypeDiscriminator discriminator) {
    if (discriminator != null) {
        String column = discriminator.column();
        Class<?> javaType = discriminator.javaType() == void.class ? String.class :
discriminator.javaType();
        JdbcType jdbcType = discriminator.jdbcType() == JdbcType.UNDEFINED ? null :
discriminator.jdbcType();
        @SuppressWarnings("unchecked")
        Class<? extends TypeHandler<?>> typeHandler = (Class<? extends TypeHandler<?>>)
            (discriminator.typeHandler() == UnknownTypeHandler.class ? null :
discriminator.typeHandler());
    }
}

```



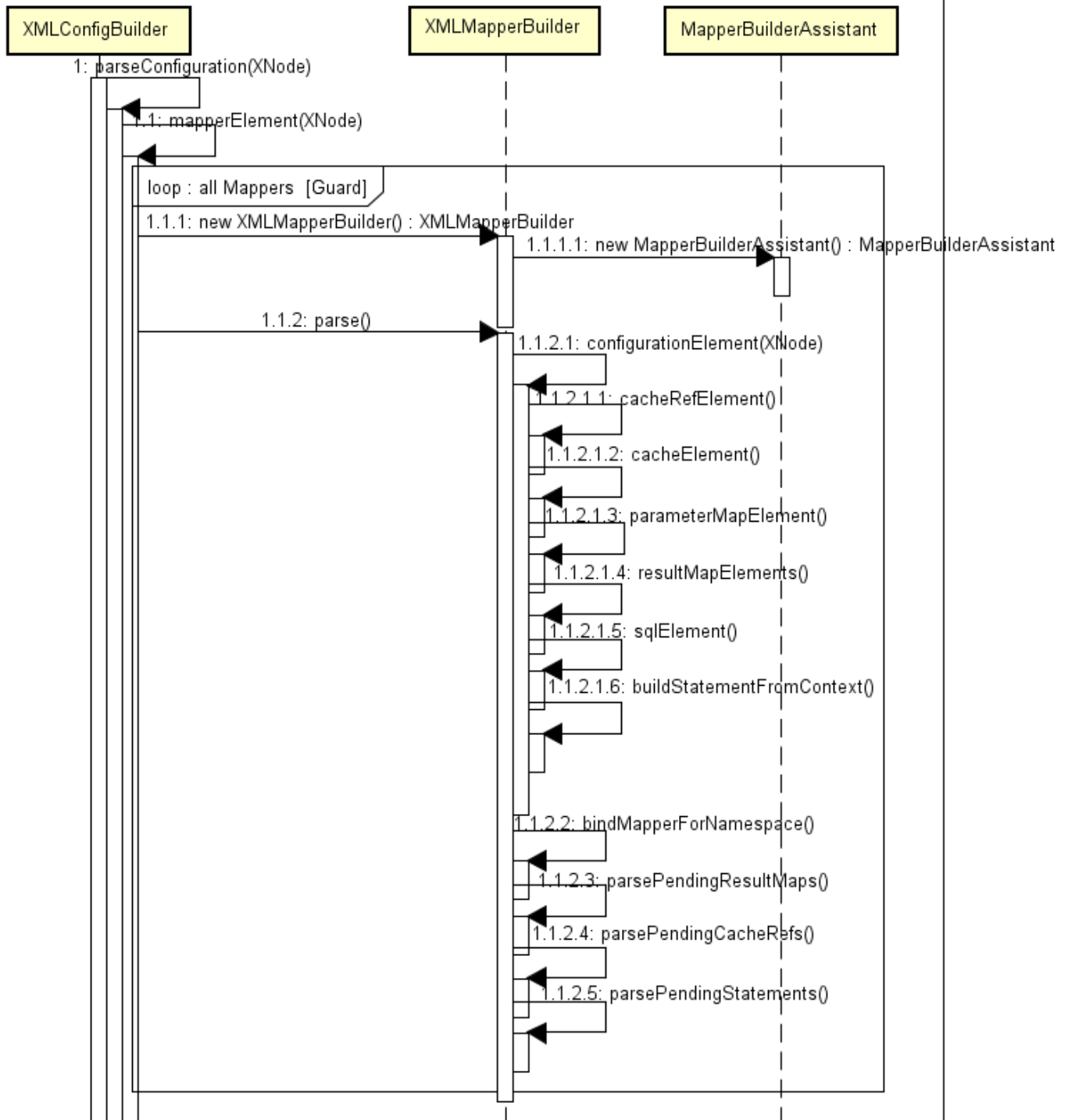
```
Case[] cases = discriminator.cases();
Map<String, String> discriminatorMap = new HashMap<String, String>();
for (Case c : cases) {
    String value = c.value();
    String caseResultMapId = resultMapId + "-" + value;
    discriminatorMap.put(value, caseResultMapId);
}
return assistant.buildDiscriminator(resultType, column, javaType, jdbcType,
typeHandler, discriminatorMap);
}
return null;
}
```

5、二次解析pending的方法。

2.3 解析mapper文件XMLMapperBuilder

Mapper文件的解析主要由XMLMapperBuilder类完成，Mapper文件的加载流程如下：

sd parseMapperXML



我们以package扫描中的loadXmlResource()为入口开始。

```

private void loadXmlResource() {
    // Spring may not know the real resource name so we check a flag
    // to prevent loading again a resource twice
    // this flag is set at XMLMapperBuilder#bindMapperForNamespace
    if (!configuration.isResourceLoaded("namespace:" + type.getName())) {
        String xmlResource = type.getName().replace('.', '/') + ".xml";
        InputStream inputStream = null;
        try {

```

```

        inputStream = Resources.getResourceAsStream(type.getClassLoader(),
xmlResource);
    } catch (IOException e) {
        // ignore, resource is not required
    }
    if (inputStream != null) {
        XMLMapperBuilder xmlParser = new XMLMapperBuilder(inputStream,
assistant.getConfiguration(), xmlResource, configuration.getSqlFragments(),
type.getName());
        xmlParser.parse();
    }
}
}

```

根据package自动搜索加载的时候，约定俗称从classpath下加载接口的完整名，比如org.mybatis.example.mapper.BlogMapper，就加载org/mybatis/example/mapper/BlogMapper.xml。对于从package和class进来的mapper，如果找不到对应的文件，就忽略，因为这种情况下是允许SQL语句作为注解打在接口上的，所以xml文件不是必须的，而对于直接声明的xml mapper文件，如果找不到的话会抛出IOException异常而终止，这在使用注解模式的时候需要注意。加载到对应的mapper.xml文件后，调用XMLMapperBuilder进行解析。在创建XMLMapperBuilder时，我们发现用到了configuration.getSqlFragments()，这就是我们在mapper文件中经常使用的可以被包含在其他语句中的SQL片段，但是我们并没有初始化过，所以很有可能它是在解析过程中动态添加的，创建了XMLMapperBuilder之后，在调用其parse()接口进行具体xml的解析，这和mybatis-config的逻辑基本上是一致的思路。再来看XMLMapperBuilder的初始化逻辑：

```

public XMLMapperBuilder(InputStream inputStream, Configuration configuration, String
resource, Map<String, XNode> sqlFragments, String namespace) {
    this(inputStream, configuration, resource, sqlFragments);
    this.builderAssistant.setCurrentNamespace(namespace);
}

public XMLMapperBuilder(InputStream inputStream, Configuration configuration, String
resource, Map<String, XNode> sqlFragments) {
    this(new XPathParser(inputStream, true, configuration.getVariables(), new
XMLMapperEntityResolver()),
        configuration, resource, sqlFragments);
}

```

加载的基本逻辑和加载mybatis-config一样的过程，使用XPathParser进行总控，XMLMapperEntityResolver进行具体判断。

接下来来看XMLMapperBuilder.parse()的具体实现。

```

public void parse() {
    if (!configuration.isResourceLoaded(resource)) {
        configurationElement(parser.evalNode("/mapper"));
        configuration.addLoadedResource(resource);
        bindMapperForNamespace();
    }

    parsePendingResultMaps();
    parsePendingCacheRefs();
    parsePendingStatements();
}

```

其中，解析mapper的核心又在configurationElement中，如下所示：

```

private void configurationElement(XNode context) {
    try {
        String namespace = context.getStringAttribute("namespace");
        if (namespace == null || namespace.equals("")) {
            throw new BuilderException("Mapper's namespace cannot be empty");
        }
        builderAssistant.setCurrentNamespace(namespace);
        cacheRefElement(context.evalNode("cache-ref"));
        cacheElement(context.evalNode("cache"));
        parameterMapElement(context.evalNodes("/mapper/parameterMap"));
        resultMapElements(context.evalNodes("/mapper/resultMap"));
        sqlElement(context.evalNodes("/mapper/sql"));
        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);
    }
}

```

其主要过程是：

1、解析缓存参照cache-ref。参照缓存顾名思义，就是共用其他缓存的设置。

```

private void cacheRefElement(XNode context) {
    if (context != null) {
        configuration.addCacheRef(builderAssistant.getCurrentNamespace(),
context.getStringAttribute("namespace"));
        CacheRefResolver cacheRefResolver = new CacheRefResolver(builderAssistant,
context.getStringAttribute("namespace"));
        try {
            cacheRefResolver.resolveCacheRef();
        } catch (IncompleteElementException e) {
            configuration.addIncompleteCacheRef(cacheRefResolver);
        }
    }
}

<cache-ref namespace="com.someone.application.data.SomeMapper"/>

```

缓存参考因为通过namespace指向其他的缓存。所以会出现第一次解析的时候指向的缓存还不存在的情况，所以需要在所有的mapper文件加载完成后进行二次处理，不仅仅是缓存参考，其他的CRUD也一样。所以在XMLMapperBuilder.configuration中有很多的incompleteXXX，这种设计模式类似于JVM GC中的mark and sweep，标记、然后处理。所以当捕获到IncompleteElementException异常时，没有终止执行，而是将指向的缓存不存在的cacheRefResolver添加到configuration.incompleteCacheRef中。

2、解析缓存cache

```

private void cacheElement(XNode context) throws Exception {
    if (context != null) {
        String type = context.getStringAttribute("type", "PERPETUAL");
        Class<? extends Cache> typeClass = typeAliasRegistry.resolveAlias(type);
        String eviction = context.getStringAttribute("eviction", "LRU");
        Class<? extends Cache> evictionClass = typeAliasRegistry.resolveAlias(eviction);
        Long flushInterval = context.getLongAttribute("flushInterval");
        Integer size = context.getIntAttribute("size");
        boolean readWrite = !context.getBooleanAttribute("readOnly", false);
        boolean blocking = context.getBooleanAttribute("blocking", false);
        Properties props = context.getChildrenAsProperties();
        builderAssistant.useNewCache(typeClass, evictionClass, flushInterval, size,
readWrite, blocking, props);
    }
}

```

默认情况下，mybatis使用的是永久缓存PerpetualCache，读取或设置各个属性默认值之后，调用builderAssistant.useNewCache构建缓存，其中的CacheBuilder使用了build模式（在effective里面，建议有4个以上可选属性时，应该为对象提供一个builder便于使用），只要实现org.apache.ibatis.cache.Cache接口，就是合法的mybatis缓存。

我们先来看下缓存的DTD定义：

```

<!ELEMENT cache (property*)>
<!--ATTLIST cache
type CDATA #IMPLIED
eviction CDATA #IMPLIED
flushInterval CDATA #IMPLIED
size CDATA #IMPLIED
readOnly CDATA #IMPLIED
blocking CDATA #IMPLIED
-->

```

所以，最简单的情况下只要声明就可以启用当前mapper下的缓存。

3、解析参数映射parameterMap

```

private void parameterMapElement(List<XNode> list) throws Exception {
    for (XNode parameterMapNode : list) {
        String id = parameterMapNode.getStringAttribute("id");
        String type = parameterMapNode.getStringAttribute("type");
        Class<?> parameterClass = resolveClass(type);
        List<XNode> parameterNodes = parameterMapNode.evalNodes("parameter");
        List<ParameterMapping> parameterMappings = new ArrayList<ParameterMapping>();
        for (XNode parameterNode : parameterNodes) {
            String property = parameterNode.getStringAttribute("property");
            String javaType = parameterNode.getStringAttribute("javaType");
            String jdbcType = parameterNode.getStringAttribute("jdbcType");
            String resultMap = parameterNode.getStringAttribute("resultMap");
            String mode = parameterNode.getStringAttribute("mode");
            String typeHandler = parameterNode.getStringAttribute("typeHandler");
            Integer numericScale = parameterNode.getIntAttribute("numericScale");
            ParameterMode modeEnum = resolveParameterMode(mode);
            Class<?> javaTypeClass = resolveClass(javaType);
            JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
            @SuppressWarnings("unchecked")
            Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends
TypeHandler<?>>) resolveClass(typeHandler);
            ParameterMapping parameterMapping =
builderAssistant.buildParameterMapping(parameterClass, property, javaTypeClass,
jdbcTypeEnum, resultMap, modeEnum, typeHandlerClass, numericScale);
            parameterMappings.add(parameterMapping);
        }
        builderAssistant.addParameterMap(id, parameterClass, parameterMappings);
    }
}

<!--ELEMENT parameterMap (parameter+)?>
<!--ATTLIST parameterMap
id CDATA #REQUIRED
type CDATA #REQUIRED
-->

```

```

<!ELEMENT parameter EMPTY>
<!--ATTLIST parameter
property CDATA #REQUIRED
javaType CDATA #IMPLIED
jdbcType CDATA #IMPLIED
mode (IN | OUT | INOUT) #IMPLIED
resultMap CDATA #IMPLIED
scale CDATA #IMPLIED
typeHandler CDATA #IMPLIED
-->

```

总体来说，目前已经不推荐使用参数映射，而是直接使用内联参数。所以我们这里就不展开细讲了。如有必要，我们后续会补上。

4、解析结果集映射resultMap

结果集映射早期版本可以说是用的最多的辅助节点了，不过有了mapUnderscoreToCamelCase属性之后，如果命名规范控制做的好的话，resultMap也是可以省略的。每个mapper文件可以有多个结果集映射。最终来说，它还是使用频率很高的。我们先来看下DTD定义：

```

<!--ELEMENT resultMap (constructor?,id*,result*,association*,collection*,
discriminator?)>
<!--ATTLIST resultMap
id CDATA #REQUIRED
type CDATA #REQUIRED
extends CDATA #IMPLIED
autoMapping (true|false) #IMPLIED
-->

<!--ELEMENT constructor (idArg*,arg*)>

<!--ELEMENT id EMPTY>
<!--ATTLIST id
property CDATA #IMPLIED
javaType CDATA #IMPLIED
column CDATA #IMPLIED
jdbcType CDATA #IMPLIED
typeHandler CDATA #IMPLIED
-->

<!--ELEMENT result EMPTY>
<!--ATTLIST result
property CDATA #IMPLIED
javaType CDATA #IMPLIED
column CDATA #IMPLIED
jdbcType CDATA #IMPLIED
typeHandler CDATA #IMPLIED
-->

<!--ELEMENT idArg EMPTY>

```

```
<!--ATTLIST idArg
javaType CDATA #IMPLIED
column CDATA #IMPLIED
jdbcType CDATA #IMPLIED
typeHandler CDATA #IMPLIED
select CDATA #IMPLIED
resultMap CDATA #IMPLIED
name CDATA #IMPLIED
-->
```

```
<!--ELEMENT arg EMPTY-->
<!--ATTLIST arg
javaType CDATA #IMPLIED
column CDATA #IMPLIED
jdbcType CDATA #IMPLIED
typeHandler CDATA #IMPLIED
select CDATA #IMPLIED
resultMap CDATA #IMPLIED
name CDATA #IMPLIED
-->
```

```
<!--ELEMENT collection (constructor?,id*,result*,association*,collection*,
discriminator?)-->
<!--ATTLIST collection
property CDATA #REQUIRED
column CDATA #IMPLIED
javaType CDATA #IMPLIED
ofType CDATA #IMPLIED
jdbcType CDATA #IMPLIED
select CDATA #IMPLIED
resultMap CDATA #IMPLIED
typeHandler CDATA #IMPLIED
notNullColumn CDATA #IMPLIED
columnPrefix CDATA #IMPLIED
resultSet CDATA #IMPLIED
foreignColumn CDATA #IMPLIED
autoMapping (true|false) #IMPLIED
fetchType (lazy|eager) #IMPLIED
-->
```

```
<!--ELEMENT association (constructor?,id*,result*,association*,collection*,
discriminator?)-->
<!--ATTLIST association
property CDATA #REQUIRED
column CDATA #IMPLIED
javaType CDATA #IMPLIED
jdbcType CDATA #IMPLIED
select CDATA #IMPLIED
resultMap CDATA #IMPLIED
```



```

typeHandler CDATA #IMPLIED
notNullColumn CDATA #IMPLIED
columnPrefix CDATA #IMPLIED
resultSet CDATA #IMPLIED
foreignColumn CDATA #IMPLIED
autoMapping (true|false) #IMPLIED
fetchType (lazy|eager) #IMPLIED
>

<!ELEMENT discriminator (case+)>
<!ATTLIST discriminator
column CDATA #IMPLIED
javaType CDATA #REQUIRED
jdbcType CDATA #IMPLIED
typeHandler CDATA #IMPLIED
>

<!ELEMENT case (constructor?,id*,result*,association*,collection*, discriminator?)>
<!ATTLIST case
value CDATA #REQUIRED
resultMap CDATA #IMPLIED
resultType CDATA #IMPLIED
>

```

从DTD的复杂程度就可知，resultMap相当于前面的cache/parameterMap等来说，是相当灵活的。下面我们来看resultMap在运行时到底是如何表示的。

```

private void resultMapElements(List<XNode> list) throws Exception {
    for (XNode resultMapNode : list) {
        try {
            resultMapElement(resultMapNode);
        } catch (IncompleteElementException e) {
            // ignore, it will be retried
            // 在内部实现中将未完成的元素添加到configuration.incomplete中了
        }
    }
}

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

private ResultMap resultMapElement(XNode resultMapNode, List<ResultMapping>
additionalResultMappings) throws Exception {
    ErrorContext.instance().activity("processing " +
resultMapNode.getValueBasedIdentifier());
    String id = resultMapNode.getStringAttribute("id",
        resultMapNode.getValueBasedIdentifier());
    String type = resultMapNode.getStringAttribute("type",

```

```

        resultMapNode.getStringAttribute("ofType",
            resultMapNode.getStringAttribute("resultType",
                resultMapNode.getStringAttribute("javaType"))));
String extend = resultMapNode.getStringAttribute("extends");
Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");
Class<?> typeClass = resolveClass(type);
Discriminator discriminator = null;
List<ResultMapping> resultMappings = new ArrayList<ResultMapping>();
resultMappings.addAll(additionalResultMappings);
List<XNode> resultChildren = resultMapNode.getChildren();
for (XNode resultChild : resultChildren) {
    if ("constructor".equals(resultChild.getName())) {
        processConstructorElement(resultChild, typeClass, resultMappings);
    } else if ("discriminator".equals(resultChild.getName())) {
        discriminator = processDiscriminatorElement(resultChild, typeClass,
resultMappings);
    } else {
        List<ResultFlag> flags = new ArrayList<ResultFlag>();
        if ("id".equals(resultChild.getName())) {
            flags.add(ResultFlag.ID);
        }
        resultMappings.add(buildResultMappingFromContext(resultChild, typeClass,
flags));
    }
}

ResultMapResolver resultMapResolver = new ResultMapResolver(builderAssistant, id,
typeClass, extend, discriminator, resultMappings, autoMapping);
try {
    return resultMapResolver.resolve();
} catch (IncompleteElementException e) {
    configuration.addIncompleteResultMap(resultMapResolver);
    throw e;
}
}

private void processConstructorElement(XNode resultChild, Class<?> resultType,
List<ResultMapping> resultMappings) throws Exception {
    List<XNode> argChildren = resultChild.getChildren();
    for (XNode argChild : argChildren) {
        List<ResultFlag> flags = new ArrayList<ResultFlag>();
        flags.add(ResultFlag.CONSTRUCTOR);
        if ("idArg".equals(argChild.getName())) {
            flags.add(ResultFlag.ID);
        }
        resultMappings.add(buildResultMappingFromContext(argChild, resultType, flags));
    }
}
}

```

```

private Discriminator processDiscriminatorElement(XNode context, Class<?> resultType,
List<ResultMapping> resultMappings) throws Exception {
    String column = context.getStringAttribute("column");
    String javaType = context.getStringAttribute("javaType");
    String jdbcType = context.getStringAttribute("jdbcType");
    String typeHandler = context.getStringAttribute("typeHandler");
    Class<?> javaTypeClass = resolveClass(javaType);
    @SuppressWarnings("unchecked")
    Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends TypeHandler<?
>>) resolveClass(typeHandler);
    JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
    Map<String, String> discriminatorMap = new HashMap<String, String>();
    for (XNode caseChild : context.getChildren()) {
        String value = caseChild.getStringAttribute("value");
        String resultMap = caseChild.getStringAttribute("resultMap",
processNestedResultMappings(caseChild, resultMappings));
        discriminatorMap.put(value, resultMap);
    }
    return builderAssistant.buildDiscriminator(resultType, column, javaTypeClass,
jdbcTypeEnum, typeHandlerClass, discriminatorMap);
}

```

我们先来看下ResultMapping的定义：

```

public class ResultMapping {

    private Configuration configuration;
    private String property;
    private String column;
    private Class<?> javaType;
    private JdbcType jdbcType;
    private TypeHandler<?> typeHandler;
    private String nestedResultMapId;
    private String nestedQueryId;
    private Set<String> notNullColumns;
    private String columnPrefix;
    private List<ResultFlag> flags;
    private List<ResultMapping> composites;
    private String resultSet;
    private String foreignColumn;
    private boolean lazy;
    ...
}

```

所有下的最底层子元素比如、、等本质上都属于一个映射,只不过有着额外的标记比如是否嵌套，是否构造器等。

总体逻辑是先解析resultMap节点本身，然后解析子节点构造器，鉴别器discriminator，id。最后组装成真正的resultMappings。我们先来看个实际的复杂resultMap例子，便于我们更好的理解代码的逻辑：

```

<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
    <arg column="blog_name" javaType="string"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" javaType="Author"/>
    <collection property="comments" ofType="Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" ofType="Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>

```

resultMap里面可以包含多种子节点，每个节点都有具体的方法进行解析，这也体现了单一职责原则。在resultMapElement中，主要是解析resultMap节点本身并循环遍历委托给具体的方法处理。下面来看构造器的解析。构造器主要用于没有默认构造器或者有多个构造器的情况，比如：

```

public class User {
  //...
  public User(Integer id, String username, int age) {
    //...
  }
  //...
}

```

就可以使用下列的构造器设置属性值，比如：

```

<constructor>
  <idArg column="id" javaType="int"/>
  <arg column="username" javaType="String"/>
  <arg column="age" javaType="_int"/>
</constructor>

```

遍历构造器元素很简单：

```

private void processConstructorElement(XNode resultChild, Class<?> resultType,
List<ResultMapping> resultMappings) throws Exception {
    List<XNode> argChildren = resultChild.getChildren();
    for (XNode argChild : argChildren) {
        List<ResultFlag> flags = new ArrayList<ResultFlag>();
        flags.add(ResultFlag.CONSTRUCTOR);
        if ("idArg".equals(argChild.getName())) {
            flags.add(ResultFlag.ID);
        }
        resultMappings.add(buildResultMappingFromContext(argChild, resultType, flags));
    }
}

```

构造器的解析比较简单，除了遍历构造参数外，还可以构造器参数的ID也识别出来。最后调用 buildResultMappingFromContext 建立具体的 resultMap。buildResultMappingFromContext 是个公共工具方法，会被反复使用，我们来看下它的具体实现（不是所有元素都包含所有属性）：

```

private ResultMapping buildResultMappingFromContext(XNode context, Class<?> resultType,
List<ResultFlag> flags) throws Exception {
    String property;
    if (flags.contains(ResultFlag.CONSTRUCTOR)) {
        property = context.getStringAttribute("name");
    } else {
        property = context.getStringAttribute("property");
    }
    String column = context.getStringAttribute("column");
    String javaType = context.getStringAttribute("javaType");
    String jdbcType = context.getStringAttribute("jdbcType");
    String nestedSelect = context.getStringAttribute("select");
    // resultMap中可以包含association或collection复合类型,这些复合类型可以使用外部定义的公用
    resultMap或者内嵌resultMap, 所以这里的处理逻辑是如果有resultMap就获取resultMap,如果没有,那就动态
    生成一个。如果自动生成的话, 他的resultMap id通过调用XNode.getValueBasedIdentifier()来获得
    String nestedResultMap = context.getStringAttribute("resultMap",
        processNestedResultMappings(context, Collections.<ResultMapping> emptyList()));
    String notNullColumn = context.getStringAttribute("notNullColumn");
    String columnPrefix = context.getStringAttribute("columnPrefix");
    String typeHandler = context.getStringAttribute("typeHandler");
    String resultSet = context.getStringAttribute("resultSet");
    String foreignColumn = context.getStringAttribute("foreignColumn");
}

```

```

        boolean lazy = "lazy".equals(context.getStringAttribute("fetchType",
configuration.isLazyLoadingEnabled() ? "lazy" : "eager"));
        Class<?> javaTypeClass = resolveClass(javaType);
        @SuppressWarnings("unchecked")
        Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends TypeHandler<?
>>) resolveClass(typeHandler);
        JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
        return builderAssistant.buildResultMapping(resultType, property, column,
javaTypeClass, jdbcTypeEnum, nestedSelect, nestedResultMap, notNullColumn,
columnPrefix, typeHandlerClass, flags, resultSet, foreignColumn, lazy);
    }

```

上述过程主要用于获取各个属性，其中唯一值得注意的是processNestedResultMappings，它用于解析包含的association或collection复合类型，这些复合类型可以使用外部定义的公用resultMap或者内嵌resultMap，所以这里的处理逻辑如果是外部resultMap就获取对应resultMap的名称，如果没有，那就动态生成一个。如果自动生成的话，其resultMap id通过调用XNode.getValueBasedIdentifier()来获得。由于collection和association、discriminator里面还可以包含复合类型，所以将进行递归解析直到所有的子元素都为基本列位置，它在使用层面的目的在于将关系模型映射为对象树模型。例如：

```

<resultMap id="blogResult" type="Blog">
    <id property="id" column="blog_id" />
    <result property="title" column="blog_title"/>
    <association property="author" column="blog_author_id" javaType="Author"
resultMap="authorResult"/>
    <collection property="posts" javaType="ArrayList" column="id" ofType="Post"
select="selectPostsForBlog"/>
</resultMap>

```

注意“ofType”属性，这个属性用来区分JavaBean(或字段)属性类型和集合中存储的对象类型。

```

private String processNestedResultMappings(XNode context, List<ResultMapping>
resultMappings) throws Exception {
    if ("association".equals(context.getName())
        || "collection".equals(context.getName())
        || "case".equals(context.getName())) {
        if (context.getStringAttribute("select") == null) {
            resultMap resultMap = resultMapElement(context, resultMappings);
            return resultMap.getId();
        }
    }
    return null;
}

```

对于其中的每个非select属性映射，调用resultMapElement进行递归解析。其中的case节点主要用于鉴别器情况，后面我们会细讲。

注：select的用途在于指定另外一个映射语句的ID,加载这个属性映射需要的复杂类型。在列属性中指定的列的值将被传递给目标 select 语句作为参数。在上面的例子中，id的值会作为selectPostsForBlog的参数，这个语句会为每条映射到blogResult的记录执行一次selectPostsForBlog，并将返回的值添加到blog.posts属性中，其类型为Post。

得到各属性之后，调用builderAssistant.buildResultMapping最后创建ResultMap。其中除了 javaType,column 外，其他都是可选的，property也就是中的name属性或者中的property属性，主要用于根据@Param或者jdk 8 - parameters形参名而非依赖声明顺序进行映射。

```
public ResultMapping buildResultMapping(  
    Class<?> resultType,  
    String property,  
    String column,  
    Class<?> javaType,  
    JdbcType jdbcType,  
    String nestedSelect,  
    String nestedResultMap,  
    String notNullColumn,  
    String columnPrefix,  
    Class<? extends TypeHandler<?>> typeHandler,  
    List<ResultFlag> flags,  
    String resultSet,  
    String foreignColumn,  
    boolean lazy) {  
    Class<?> javaTypeClass = resolveResultJavaType(resultType, property, javaType);  
    TypeHandler<?> typeHandlerInstance = resolveTypeHandler(javaTypeClass,  
typeHandler);  
    List<ResultMapping> composites = parseCompositeColumnName(column);  
    return new ResultMapping.Builder(configuration, property, column, javaTypeClass)  
        .jdbcType(jdbcType)  
        .nestedQueryId(applyCurrentNamespace(nestedSelect, true))  
        .nestedResultMapId(applyCurrentNamespace(nestedResultMap, true))  
        .resultSet(resultSet)  
        .typeHandler(typeHandlerInstance)  
        .flags(flags == null ? new ArrayList<ResultFlag>() : flags)  
        .composites(composites)  
        .notNullColumns(parseMultipleColumnNames(notNullColumn))  
        .columnPrefix(columnPrefix)  
        .foreignColumn(foreignColumn)  
        .lazy(lazy)  
        .build();  
}
```

在实际使用中，一般不会对构造器中包含association和collection。

2.3.1 鉴别器discriminator的解析

鉴别器非常容易理解,它的表现很像Java语言中的switch语句。定义鉴别器也是通过column和javaType属性来唯一标识, column是用来确定某个字段是否为鉴别器, javaType是需要被用来保证等价测试的合适类型。例如:

```
<discriminator javaType="int" column="vehicle_type">
  <case value="1" resultMap="carResult"/>
  <case value="2" resultMap="truckResult"/>
  <case value="3" resultMap="vanResult"/>
  <case value="4" resultMap="suvResult"/>
</discriminator>
```

对于上述的鉴别器, 如果 vehicle_type=1, 那就会使用下列这个结果映射。

```
<resultMap id="carResult" type="Car">
  <result property="doorCount" column="door_count" />
</resultMap>

private Discriminator processDiscriminatorElement(XNode context, Class<?> resultType,
List<ResultMapping> resultMappings) throws Exception {
    String column = context.getStringAttribute("column");
    String javaType = context.getStringAttribute("javaType");
    String jdbcType = context.getStringAttribute("jdbcType");
    String typeHandler = context.getStringAttribute("typeHandler");
    Class<?> javaTypeClass = resolveClass(javaType);
    @SuppressWarnings("unchecked")
    Class<? extends TypeHandler<?>> typeHandlerClass = (Class<? extends TypeHandler<?
>>) resolveClass(typeHandler);
    JdbcType jdbcTypeEnum = resolveJdbcType(jdbcType);
    Map<String, String> discriminatorMap = new HashMap<String, String>();
    for (XNode caseChild : context.getChildren()) {
        String value = caseChild.getStringAttribute("value");
        String resultMap = caseChild.getStringAttribute("resultMap",
processNestedResultMappings(caseChild, resultMappings));
        discriminatorMap.put(value, resultMap);
    }
    return builderAssistant.buildDiscriminator(resultType, column, javaTypeClass,
jdbcTypeEnum, typeHandlerClass, discriminatorMap);
}
```

其逻辑和之前处理构造器的时候类似, 同样的使用build建立鉴别器并返回鉴别器实例, 鉴别器中也可以嵌套association和collection。他们的实现逻辑和常规resultMap中的处理方式完全相同, 这里就不展开重复讲。和构造器不一样的是, 鉴别器中包含了case分支和对应的resultMap的映射。

最后所有的子节点都被解析到resultMappings中, 在解析完整个resultMap中的所有子元素之后, 调用ResultMapResolver进行解析, 如下所示:


```

    resultMapResolver resultMapResolver = new resultMapResolver(builderAssistant, id,
typeClass, extend, discriminator, resultMappings, autoMapping);
    try {
        return resultMapResolver.resolve();
    } catch (IncompleteElementException e) {
        configuration.addIncompleteResultMap(resultMapResolver);
        throw e;
    }
}

```

- ▼ • addResultMap(String, Class<?>, String, Discriminator, List<ResultMapping>, Boolean) : ResultMap - org.apache.ibatis.builder.MapperBuilderAssistant
 - > ■ applyResultMap(String, Class<?>, Arg[], Result[], TypeDiscriminator) : void - org.apache.ibatis.builder.annotation.MapperAnnotationBuilder
 - > ■ createDiscriminatorResultMaps(String, Class<?>, TypeDiscriminator) : void - org.apache.ibatis.builder.annotation.MapperAnnotationBuilder
- ▼ • resolve() : ResultMap - org.apache.ibatis.builder.ResultMapResolver
 - > ◆ buildAllStatements() : void - org.apache.ibatis.session.Configuration
 - > ■ parsePendingResultMaps() : void - org.apache.ibatis.builder.xml.XMLMapperBuilder
- ▼ ■ resultMapElement(XNode, List<ResultMapping>) : ResultMap - org.apache.ibatis.builder.xml.XMLMapperBuilder
 - > ■ processNestedResultMappings(XNode, List<ResultMapping>) : String - org.apache.ibatis.builder.xml.XMLMapperBuilder
- ▼ ■ resultMapElement(XNode) : ResultMap - org.apache.ibatis.builder.xml.XMLMapperBuilder
 - ▼ ■ resultMapElements(List<XNode>) : void - org.apache.ibatis.builder.xml.XMLMapperBuilder
 - ▼ ■ configurationElement(XNode) : void - org.apache.ibatis.builder.xml.XMLMapperBuilder
 - > • parse() : void - org.apache.ibatis.builder.xml.XMLMapperBuilder

```

public ResultMap addResultMap(
    String id,
    Class<?> type,
    String extend,
    Discriminator discriminator,
    List<ResultMapping> resultMappings,
    Boolean autoMapping) {
    // 将id/extend填充为完整模式,也就是带命名空间前缀,true不需要和当前resultMap所在的namespace相同,比如extend和cache,否则只能是当前的namespace
    id = applyCurrentNamespace(id, false);
    extend = applyCurrentNamespace(extend, true);

    if (extend != null) {
        // 首先检查继承的resultMap是否已存在,如果不存在则标记为incomplete,会进行二次处理
        if (!configuration.hasResultMap(extend)) {
            throw new IncompleteElementException("Could not find a parent resultMap with id '" + extend + "'");
        }
        ResultMap resultMap = configuration.getResultMap(extend);
        List<ResultMapping> extendedResultMappings = new ArrayList<ResultMapping>(resultMap.getResultMappings());
        // 剔除所继承的resultMap里已经在当前resultMap中的那个基本映射
        extendedResultMappings.removeAll(resultMappings);
        // Remove parent constructor if this resultMap declares a constructor.
        // 如果本resultMap已经包含了构造器,则剔除继承的resultMap里面的构造器
        boolean declaresConstructor = false;
        for (ResultMapping resultMapping : resultMappings) {
            if (resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR)) {
                declaresConstructor = true;
                break;
            }
        }
    }
}

```

```

    }
}
if (declaresConstructor) {
    Iterator<ResultMapping> extendedResultMappingsIter =
extendedResultMappings.iterator();
    while (extendedResultMappingsIter.hasNext()) {
        if
        (extendedResultMappingsIter.next().getFlags().contains(ResultFlag.CONSTRUCTOR)) {
            extendedResultMappingsIter.remove();
        }
    }
}
// 都处理完成之后,将继承的resultMap里面剩下那部分不重复的resultMap子元素添加到当前的
resultMap中,所以这个addResultMap方法的用途在于启动时就创建了完整的resultMap, 这样运行时就不需要去
检查继承的映射和构造器,有利于性能提升。
resultMappings.addAll(extendedResultMappings);
}
ResultMap resultMap = new ResultMap.Builder(configuration, id, type,
resultMappings, autoMapping)
    .discriminator(discriminator)
    .build();
configuration.addResultMap(resultMap);
return resultMap;
}

```

mybatis源码分析到这里, 应该来说上面部分代码不是特别显而易见,它主要是处理extends这个属性,将resultMap各子元素节点的去对象关系, 也就是去重、合并属性、构造器。处理完继承的逻辑之后, 调用了ResultMap.Builder.build()进行最后的resultMap构建。build应该来说是真正创建根resultMap对象的逻辑入口。我们先看下ResultMap的定义:

```

public class ResultMap {
    private Configuration configuration;

    // resultMap的id属性
    private String id;

    // resultMap的type属性,有可能是alias
    private Class<?> type;

    // resultMap下的所有节点
    private List<ResultMapping> resultMappings;

    // resultMap下的id节点比如<id property="id" column="user_id" />
    private List<ResultMapping> idResultMappings;

    // resultMap下的构造器节点<constructor>
    private List<ResultMapping> constructorResultMappings;

    // resultMap下的property节点比如<result property="password" column="hashed_password"/>
}

```

```

private List<ResultMapping> propertyResultMappings;

//映射的列名
private Set<String> mappedColumns;

// 映射的javaBean属性名,所有映射不管是id、构造器或者普通的
private Set<String> mappedProperties;

// 鉴别器
private Discriminator discriminator;

// 是否有嵌套的resultMap比如association或者collection
private boolean hasNestedResultMaps;

// 是否有嵌套的查询,也就是select属性
private boolean hasNestedQueries;

// autoMapping属性,这个属性会覆盖全局的属性autoMappingBehavior
private Boolean autoMapping;

...
}

```

其中定义了节点下所有的子元素,以及必要的辅助属性, 包括最重要的各种ResultMapping。
再来看下build的实现：

```

public ResultMap build() {
    if (resultMap.id == null) {
        throw new IllegalArgumentException("ResultMaps must have an id");
    }
    resultMap.mappedColumns = new HashSet<String>();
    resultMap.mappedProperties = new HashSet<String>();
    resultMap.idResultMappings = new ArrayList<ResultMapping>();
    resultMap.constructorResultMappings = new ArrayList<ResultMapping>();
    resultMap.propertyResultMappings = new ArrayList<ResultMapping>();
    final List<String> constructorArgNames = new ArrayList<String>();
    for (ResultMapping resultMapping : resultMap.resultMappings) {
        // 判断是否有嵌套查询, nestedQueryId是在buildResultMappingFromContext的时候通过读取节点的select属性得到的
        resultMap.hasNestedQueries = resultMap.hasNestedQueries ||
resultMapping.getNestedQueryId() != null;

        // 判断是否嵌套了association或者collection, nestedResultMapId是在
buildResultMappingFromContext的时候通过读取节点的resultMap属性得到的或者内嵌resultMap的时候自动
计算得到的。注：这里的resultSet没有地方set进来,DTD中也没有看到,不确定是不是有意预留的,但是
association/collection的子元素中倒是有声明
        resultMap.hasNestedResultMaps = resultMap.hasNestedResultMaps ||
(resultMapping.getNestedResultMapId() != null && resultMapping.getResultSet() == null);
    }
}

```

```

        // 获取column属性，包括复合列，复合列是在
org.apache.ibatis.builder.MapperBuilderAssistant.parseCompositeColumnName(String)中解析
的。所有的数据库列都被按顺序添加到resultMap.mappedColumns中
        final String column = resultMapping.getColumn();
        if (column != null) {
            resultMap.mappedColumns.add(column.toUpperCase(Locale.ENGLISH));
        } else if (resultMapping.isCompositeResult()) {
            for (ResultMapping compositeResultMapping : resultMapping.getComposites()) {
                final String compositeColumn = compositeResultMapping.getColumn();
                if (compositeColumn != null) {
                    resultMap.mappedColumns.add(compositeColumn.toUpperCase(Locale.ENGLISH));
                }
            }
        }
    }

    // 所有映射的属性都被按顺序添加到resultMap.mappedProperties中,ID单独存储
    final String property = resultMapping.getProperty();
    if (property != null) {
        resultMap.mappedProperties.add(property);
    }

    // 所有映射的构造器被按顺序添加到resultMap.constructorResultMappings
    // 如果本元素具有CONSTRUCTOR标记,则添加到构造函数参数列表,否则添加到普通属性映射列表
resultMap.propertyResultMappings
    if (resultMapping.getFlags().contains(ResultFlag.CONSTRUCTOR)) {
        resultMap.constructorResultMappings.add(resultMapping);
        if (resultMapping.getProperty() != null) {
            constructorArgNames.add(resultMapping.getProperty());
        }
    } else {
        resultMap.propertyResultMappings.add(resultMapping);
    }

    // 如果本元素具有ID标记,则添加到ID映射列表resultMap.idResultMappings
    if (resultMapping.getFlags().contains(ResultFlag.ID)) {
        resultMap.idResultMappings.add(resultMapping);
    }
}

// 如果没有声明ID属性,就把所有属性都作为ID属性
if (resultMap.idResultMappings.isEmpty()) {
    resultMap.idResultMappings.addAll(resultMap.resultMappings);
}

// 根据声明的构造器参数名和类型,反射声明的类,检查其中是否包含对应参数名和类型的构造器,如果不存在
匹配的构造器,就抛出运行时异常,这是为了确保运行时不会出现异常
if (!constructorArgNames.isEmpty()) {
    final List<String> actualArgNames =
argNamesOfMatchingConstructor(constructorArgNames);

```

```

        if (actualArgNames == null) {
            throw new BuilderException("Error in result map '" + resultMap.id
                + "'. Failed to find a constructor in '"
                + resultMap.getType().getName() + "' by arg names " + constructorArgNames
                + ". There might be more info in debug log.");
        }
        //构造器参数排序
        Collections.sort(resultMap.constructorResultMappings, new
Comparator<ResultMapping>() {
            @Override
            public int compare(ResultMapping o1, ResultMapping o2) {
                int paramIdx1 = actualArgNames.indexOf(o1.getProperty());
                int paramIdx2 = actualArgNames.indexOf(o2.getProperty());
                return paramIdx1 - paramIdx2;
            }
        });
    }
    // lock down collections
    // 为了避免用于无意或者有意事后修改resultMap的内部结构, 克隆一个不可修改的集合提供给用户
    resultMap.resultMappings =
Collections.unmodifiableList(resultMap.resultMappings);
    resultMap.idResultMappings =
Collections.unmodifiableList(resultMap.idResultMappings);
    resultMap.constructorResultMappings =
Collections.unmodifiableList(resultMap.constructorResultMappings);
    resultMap.propertyResultMappings =
Collections.unmodifiableList(resultMap.propertyResultMappings);
    resultMap.mappedColumns = Collections.unmodifiableSet(resultMap.mappedColumns);
    return resultMap;
}

private List<String> argNamesOfMatchingConstructor(List<String>
constructorArgNames) {
    Constructor<?>[] constructors = resultMap.type.getDeclaredConstructors();
    for (Constructor<?> constructor : constructors) {
        Class<?>[] paramTypes = constructor.getParameterTypes();
        if (constructorArgNames.size() == paramTypes.length) {
            List<String> paramNames = getArgNames(constructor);
            if (constructorArgNames.containsAll(paramNames)
                && argTypesMatch(constructorArgNames, paramTypes, paramNames)) {
                return paramNames;
            }
        }
    }
    return null;
}

private boolean argTypesMatch(final List<String> constructorArgNames,
    Class<?>[] paramTypes, List<String> paramNames) {

```

```

        for (int i = 0; i < constructorArgNames.size(); i++) {
            Class<?> actualType =
paramTypes[paramNames.indexOf(constructorArgNames.get(i))];
            Class<?> specifiedType =
resultMap.constructorResultMappings.get(i).getJavaType();
            if (!actualType.equals(specifiedType)) {
                if (log.isDebugEnabled()) {
                    log.debug("While building result map '" + resultMap.id
                        + "', found a constructor with arg names " + constructorArgNames
                        + ", but the type of '" + constructorArgNames.get(i)
                        + "' did not match. Specified: [" + specifiedType.getName() + "]
Declared: ["
                        + actualType.getName() + "]);
                }
                return false;
            }
        }
        return true;
    }

    private List<String> getArgNames(Constructor<?> constructor) {
        List<String> paramNames = new ArrayList<String>();
        List<String> actualParamNames = null;
        final Annotation[][] paramAnnotations = constructor.getParameterAnnotations();
        int paramCount = paramAnnotations.length;
        for (int paramIndex = 0; paramIndex < paramCount; paramIndex++) {
            String name = null;
            for (Annotation annotation : paramAnnotations[paramIndex]) {
                if (annotation instanceof Param) {
                    name = ((Param) annotation).value();
                    break;
                }
            }
            if (name == null && resultMap.configuration.isUseActualParamName() &&
Jdk.parameterExists) {
                if (actualParamNames == null) {
                    actualParamNames = ParamNameUtil.getParamNames(constructor);
                }
                if (actualParamNames.size() > paramIndex) {
                    name = actualParamNames.get(paramIndex);
                }
            }
            paramNames.add(name != null ? name : "arg" + paramIndex);
        }
        return paramNames;
    }
}

```

ResultMap构建完成之后，添加到Configuration的resultMaps中。我们发现在addResultMap方法中，还有两个针对鉴别器嵌套resultMap处理：

```
public void addResultMap(ResultMap rm) {
    resultMaps.put(rm.getId(), rm);
    // 检查本resultMap内的鉴别器有没有嵌套resultMap
    checkLocallyForDiscriminatedNestedResultMaps(rm);
    // 检查所有resultMap的鉴别器有没有嵌套resultMap
    checkGloballyForDiscriminatedNestedResultMaps(rm);
}
```

应该来说，设置resultMap的鉴别器有没有嵌套的resultMap在解析resultMap子元素的时候就可以设置，当然放在最后统一处理也未尝不可，也不见得放在这里就一定更加清晰，只能说实现的方式有多种。

到此为止，一个根resultMap的解析就完整的结束了。不得不说resultMap的实现确实是很复杂。

5、解析sql片段

sql元素可以被用来定义可重用的SQL代码段，包含在其他语句中。比如，他常被用来定义重用的列：

```
<sql id="userColumns"> id,username,password </sql>
```

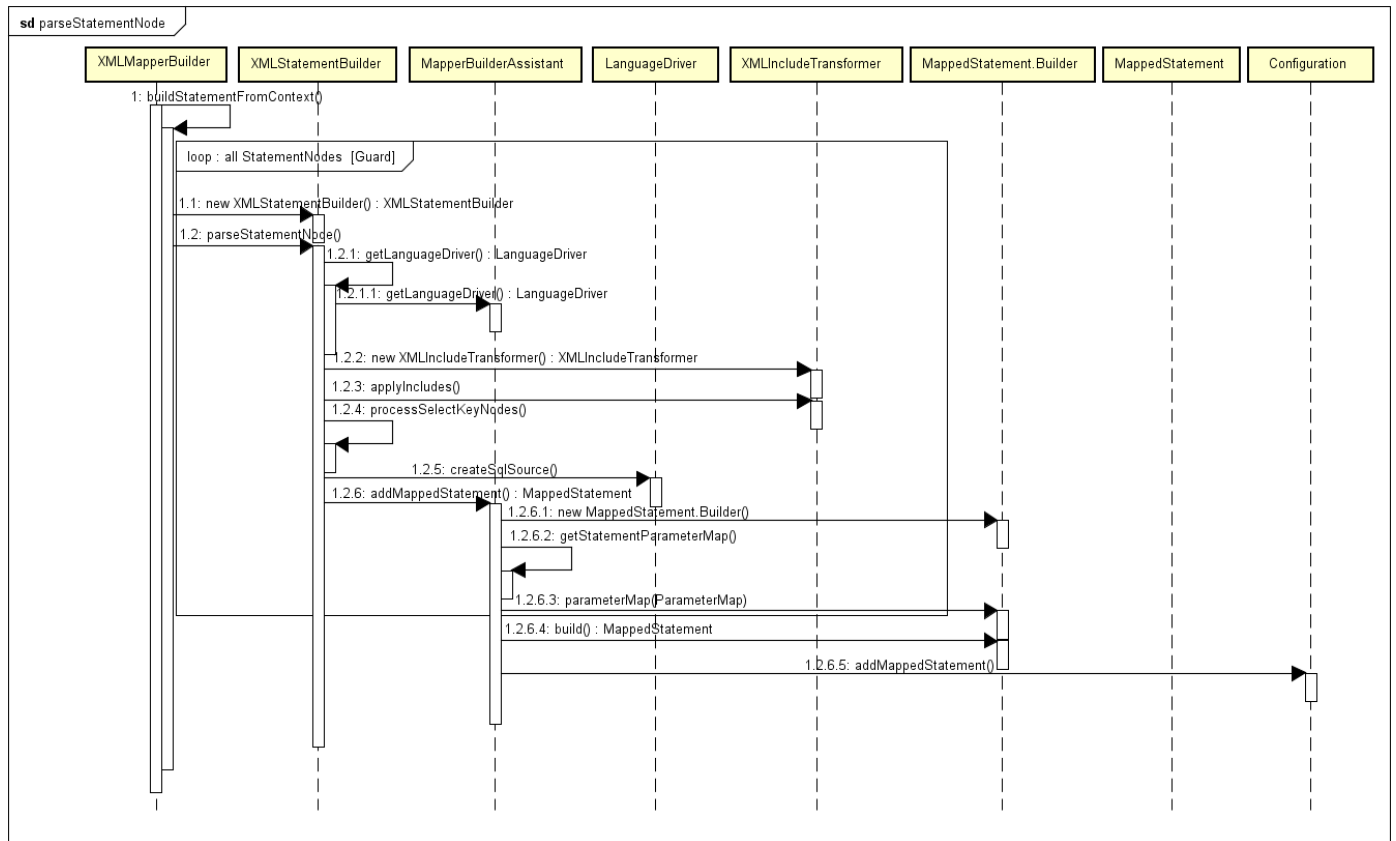
sql片段的解析逻辑为：

```
private void sqlElement(List<XNode> list, String requiredDatabaseId) throws Exception {
    for (XNode context : list) {
        String databaseId = context.getStringAttribute("databaseId");
        String id = context.getStringAttribute("id");
        id = builderAssistant.applyCurrentNamespace(id, false);
        if (databaseIdMatchesCurrent(id, databaseId, requiredDatabaseId)) {
            sqlFragments.put(id, context);
        }
    }
}
```

前面讲过，mybatis可以根据不同的数据库执行不同的sql，这就是通过sql元素上的databaseId属性来区别的。同样，首先设置sql元素的id，它必须是当前mapper文件所定义的命名空间。sql元素本身的处理很简单，只是简单的过滤出databaseId和当前加载的配置文件相同的语句以备以后再解析crud遇到时进行引用。之所以不进行解析，是因为首先能够作为sql元素子元素的所有节点都可以作为crud的子元素，而且sql元素不会在运行时单独使用，所以也没有必要专门解析一番。下面我们重点来看crud的解析与内部表示。

6、解析CRUD语句

CRUD是SQL的核心部分，也是mybatis针对用户提供的最有价值的部分，所以研究源码有必要对CRUD的完整实现进行分析。不理解这一部分的核心实现，就谈不上对mybatis的实现有深刻理解。CRUD解析和加载的整体流程如下：



crud的解析从buildStatementFromContext(context.evalNodes("select|insert|update|delete"));语句开始，通过调用调用链，我们可以得知SQL语句的解析主要在XMLStatementBuilder中实现。

```

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) {
    for (XNode context : list) {
        final XMLStatementBuilder statementParser = new
XMLStatementBuilder(configuration, builderAssistant, context, requiredDatabaseId);
        try {
            statementParser.parseStatementNode();
        } catch (IncompleteElementException e) {
            configuration.addIncompleteStatement(statementParser);
        }
    }
}

```

在看具体实现之前，我们先大概看下DTD的定义（因为INSERT/UPDATE/DELETE属于一种类型，所以我们以INSERT为例），这样我们就知道整体关系和脉络：

```

<!ELEMENT select (#PCDATA | include | trim | where | set | foreach | choose | if |
bind)*>

```



```

<!--ATTLIST select
id CDATA #REQUIRED
parameterMap CDATA #IMPLIED
parameterType CDATA #IMPLIED
resultMap CDATA #IMPLIED
resultType CDATA #IMPLIED
resultSetType (FORWARD_ONLY | SCROLL_INSENSITIVE | SCROLL_SENSITIVE) #IMPLIED
statementType (STATEMENT|PREPARED|CALLABLE) #IMPLIED
fetchSize CDATA #IMPLIED
timeout CDATA #IMPLIED
flushCache (true|false) #IMPLIED
useCache (true|false) #IMPLIED
databaseId CDATA #IMPLIED
lang CDATA #IMPLIED
resultOrdered (true|false) #IMPLIED
resultSets CDATA #IMPLIED
-->

<!--ELEMENT insert (#PCDATA | selectKey | include | trim | where | set | foreach | choose
| if | bind)*-->
<!--ATTLIST insert
id CDATA #REQUIRED
parameterMap CDATA #IMPLIED
parameterType CDATA #IMPLIED
timeout CDATA #IMPLIED
flushCache (true|false) #IMPLIED
statementType (STATEMENT|PREPARED|CALLABLE) #IMPLIED
keyProperty CDATA #IMPLIED
useGeneratedKeys (true|false) #IMPLIED
keyColumn CDATA #IMPLIED
databaseId CDATA #IMPLIED
lang CDATA #IMPLIED
-->

<!--ELEMENT selectKey (#PCDATA | include | trim | where | set | foreach | choose | if |
bind)*-->
<!--ATTLIST selectKey
resultType CDATA #IMPLIED
statementType (STATEMENT|PREPARED|CALLABLE) #IMPLIED
keyProperty CDATA #IMPLIED
keyColumn CDATA #IMPLIED
order (BEFORE|AFTER) #IMPLIED
databaseId CDATA #IMPLIED
-->

```

我们先来看statementParser.parseStatementNode()的实现主体部分。

```

public void parseStatementNode() {
    String id = context.getStringAttribute("id");

```

```
String databaseId = context.getStringAttribute("databaseId");

if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId)) {
    return;
}
```

```
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");
```

// MyBatis 从 3.2 开始支持可插拔的脚本语言，因此你可以在插入一种语言的驱动（language driver）之后来写基于这种语言的动态 SQL 查询。

```
String lang = context.getStringAttribute("lang");
LanguageDriver langDriver = getLanguageDriver(lang);
```

```
Class<?> resultTypeClass = resolveClass(resultType);
```

// 结果集的类型，FORWARD_ONLY, SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 unset（依赖驱动）。

```
String resultSetType = context.getStringAttribute("resultSetType");
```

// 解析crud语句的类型，mybatis目前支持三种,prepare、硬编码、以及存储过程调用

```
StatementType statementType =
StatementType.valueOf(context.getStringAttribute("statementType",
StatementType.PREPARED.toString()));
ResultSetType resultSetTypeEnum = resolveResultSetType(resultSetType);
```

```
String nodeName = context.getNode().getNodeName();
```

// 解析SQL命令类型，目前主要有UNKNOWN, INSERT, UPDATE, DELETE, SELECT, FLUSH

```
SqlCommandType sqlCommandType =
SqlCommandType.valueOf(nodeName.toUpperCase(Locale.ENGLISH));
boolean isSelect = sqlCommandType == SqlCommandType.SELECT;
```

// insert/delete/update后是否刷新缓存

```
boolean flushCache = context.getBooleanAttribute("flushCache", !isSelect);
```

// select是否使用缓存

```
boolean useCache = context.getBooleanAttribute("useCache", isSelect);
```

// 这个设置仅针对嵌套结果 select 语句适用：如果为 true，就是假设包含了嵌套结果集或是分组了，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值：false。我猜测这个属性为true的意思是查询的结果字段根据定义的嵌套resultMap进行了排序，后面在分析sql执行源码的时候，我们会具体看到他到底是干吗用的

```
boolean resultOrdered = context.getBooleanAttribute("resultOrdered", false);
```

```

// Include Fragments before parsing
// 解析语句中包含的sql片段，也就是
// <select id="select" resultType="map">
//   select
//     field1, field2, field3
//     <include refid="someinclude"></include>
// </select>

XMLIncludeTransformer includeParser = new XMLIncludeTransformer(configuration,
builderAssistant);
includeParser.applyIncludes(context.getNode());

// Parse selectKey after includes and remove them.
processSelectKeyNodes(id, parameterTypeClass, langDriver);

// Parse the SQL (pre: <selectKey> and <include> were parsed and removed)
SqlSource sqlSource = langDriver.createSqlSource(configuration, context,
parameterTypeClass);
String resultSets = context.getStringAttribute("resultSets");
String keyProperty = context.getStringAttribute("keyProperty");
String keyColumn = context.getStringAttribute("keyColumn");
KeyGenerator keyGenerator;
String keyStatementId = id + SelectKeyGenerator.SELECT_KEY_SUFFIX;
keyStatementId = builderAssistant.applyCurrentNamespace(keyStatementId, true);
if (configuration.hasKeyGenerator(keyStatementId)) {
    keyGenerator = configuration.getKeyGenerator(keyStatementId);
} else {
    keyGenerator = context.getBooleanAttribute("useGeneratedKeys",
configuration.isUseGeneratedKeys() &&
SqlCommandType.INSERT.equals(sqlCommandType))
        ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;
}

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

```

从DTD可以看出，sql/crud元素里面可以包含这些元素：#PCDATA(文本节点) | include | trim | where | set | foreach | choose | if | bind，除了文本节点外，其他类型的节点都可以嵌套，我们看下解析包含的sql片段的逻辑。

```

/**
 * Recursively apply includes through all SQL fragments.
 * @param source Include node in DOM tree
 * @param variablesContext Current context for static variables with values

```

```

    */
    private void applyIncludes(Node source, final Properties variablesContext, boolean
included) {
        if (source.getNodeName().equals("include")) {
            Node toInclude = findSqlFragment(getStringAttribute(source, "refid"),
variablesContext);
            Properties toIncludeContext = getVariablesContext(source, variablesContext);
            applyIncludes(toInclude, toIncludeContext, true);
            if (toInclude.getOwnerDocument() != source.getOwnerDocument()) {
                toInclude = source.getOwnerDocument().importNode(toInclude, true);
            }
            source.getParentNode().replaceChild(toInclude, source);
            while (toInclude.hasChildNodes()) {
                toInclude.getParentNode().insertBefore(toInclude.getFirstChild(), toInclude);
            }
            toInclude.getParentNode().removeChild(toInclude);
        } else if (source.getNodeType() == Node.ELEMENT_NODE) {
            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));
                }
            }
            NodeList children = source.getChildNodes();
            for (int i = 0; i < children.getLength(); i++) {
                applyIncludes(children.item(i), variablesContext, included);
            }
        } else if (included && source.getNodeType() == Node.TEXT_NODE
&& !variablesContext.isEmpty()) {
            // replace variables in text node
            source.setNodeValue(PropertyParser.parse(source.getNodeValue(),
variablesContext));
        }
    }
}

```

总的来说，将节点分为文本节点、include、非include三类进行处理。因为一开始传递进来的是CRUD节点本身，所以第一次执行的时候，是第一个else if，也就是source.getNodeType() == Node.ELEMENT_NODE，然后在这里开始遍历所有的子节点。

对于include节点：根据属性refid调用findSqlFragment找到sql片段，对节点中包含的占位符进行替换解析，然后调用自身进行递归解析，解析到文本节点返回之后。判断下include的sql片段是否和包含它的节点是同一个文档，如果不是，则把它从原来的文档包含进来。然后使用include指向的sql节点替换include节点，最后剥掉sql节点本身，也就是把sql下的节点上移一层，这样就合法了。举例来说，这里完成的功能就是把：

```

<sql id="userColumns"> id,username,password </sql>
<select id="selectUsers" parameterType="int" resultType="hashmap">
    select <include refid="userColumns"/>
    from some_table
    where id = #{id}
</select>

```

转换为下面的形式：

```

<select id="selectUsers" parameterType="int" resultType="hashmap">
    select id,username,password
    from some_table
    where id = #{id}
</select>

```

对于文本节点：根据入参变量上下文将变量设置替换进去；

对于其他节点：首先判断是否为根节点，如果是非根且变量上下文不为空，则先解析属性值上的占位符。然后对于子节点，递归进行调用直到所有节点都为文本节点为止。

上述解析完成之后，CRUD就没有嵌套的sql片段了，这样就可以进行直接解析了。现在，我们回到parseStatementNode()刚才中止的部分继续往下看。接下来是解析selectKey节点。selectKey节点用于支持数据库比如Oracle不支持自动生成主键，或者可能JDBC驱动不支持自动生成主键时的情况。对于数据库支持自动生成主键的字段（比如MySQL和SQL Server），那么你可以设置useGeneratedKeys="true"，而且设置keyProperty到你已经做好的目标属性上就可以了，不需要使用selectKey节点。由于已经处理了SQL片段节点，当前在处理CRUD节点，所以先将包含的SQL片段展开，然后解析selectKey是正确的，因为selectKey可以包含在SQL片段中。

```

private void parseSelectKeyNode(String id, XNode nodeToHandle, Class<?>
parameterTypeClass, LanguageDriver langDriver, String databaseId) {
    String resultType = nodeToHandle.getStringAttribute("resultType");
    Class<?> resultTypeClass = resolveClass(resultType);
    StatementType statementType =
StatementType.valueOf(nodeToHandle.getStringAttribute("statementType",
StatementType.PREPARED.toString()));
    String keyProperty = nodeToHandle.getStringAttribute("keyProperty");
    String keyColumn = nodeToHandle.getStringAttribute("keyColumn");
    boolean executeBefore = "BEFORE".equals(nodeToHandle.getStringAttribute("order",
"AFTER"));

    //defaults
    boolean useCache = false;
    boolean resultOrdered = false;
    KeyGenerator keyGenerator = NoKeyGenerator.INSTANCE;
    Integer fetchSize = null;
    Integer timeout = null;
    boolean flushCache = false;
    String parameterMap = null;
    String resultMap = null;
    ResultSetType resultSetTypeEnum = null;

```

```

        SqlSource sqlSource = langDriver.createSqlSource(configuration, nodeToHandle,
parameterTypeClass);
        SqlCommandType sqlCommandType = SqlCommandType.SELECT;

        builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,
            fetchSize, timeout, parameterMap, parameterTypeClass, resultMap,
resultTypeClass,
            resultSetTypeEnum, flushCache, useCache, resultOrdered,
            keyGenerator, keyProperty, keyColumn, databaseId, langDriver, null);

        id = builderAssistant.applyCurrentNamespace(id, false);

        MappedStatement keyStatement = configuration.getMappedStatement(id, false);
        configuration.addKeyGenerator(id, new SelectKeyGenerator(keyStatement,
executeBefore));
    }

```

解析SelectKey节点总的来说比较简单：

- 1、加载相关属性；
- 2、使用语言驱动器创建SqlSource，关于创建SqlSource的细节，我们会在下一节详细分析mybatis语言驱动器时详细展开。SqlSource是一个接口，它代表了从xml或者注解上读取到的sql映射语句的内容，其中参数使用占位符进行了替换，在运行时，其代表的SQL会发送给数据库，如下：

```

/**
 * Represents the content of a mapped statement read from an XML file or an annotation.
 * It creates the SQL that will be passed to the database out of the input parameter
received from the user.
 *
 * @author Clinton Begin
 */
public interface SqlSource {

    BoundSql getBoundSql(Object parameterObject);

}

```

mybatis提供了两种类型的实现org.apache.ibatis.scripting.xmltags.DynamicSqlSource和org.apache.ibatis.scripting.defaults.RawSqlSource。

- 3、SelectKey在实现上内部和其他的CRUD一样，被当做一个MappedStatement进行存储；我们来看下MappedStatement的具体构造以及代表SelectKey的sqlSource是如何组装成MappedStatement的。

```

public final class MappedStatement {

    private String resource;
    private Configuration configuration;
    private String id;
    private Integer fetchSize;

```

```

private Integer timeout;
private StatementType statementType;
private ResultSetType resultSetType;
private SqlSource sqlSource;
private Cache cache;
private ParameterMap parameterMap;
private List<ResultMap> resultMaps;
private boolean flushCacheRequired;
private boolean useCache;
private boolean resultOrdered;
private SqlCommandType sqlCommandType;
private KeyGenerator keyGenerator;
private String[] keyProperties;
private String[] keyColumns;
private boolean hasNestedResultMaps;
private String databaseId;
private Log statementLog;
private LanguageDriver lang;
private String[] resultSets;
...
}

```

对于每个语句而言，在运行时都需要知道结果映射，是否使用缓存，语句类型，sql文本，超时时间，是否自动生成key等等。为了方便运行时的使用以及高效率，MappedStatement被设计为直接包含了所有这些属性。

```

public MappedStatement addMappedStatement(
    String id,
    SqlSource sqlSource,
    StatementType statementType,
    SqlCommandType sqlCommandType,
    Integer fetchSize,
    Integer timeout,
    String parameterMap,
    Class<?> parameterType,
    String resultMap,
    Class<?> resultType,
    ResultSetType resultSetType,
    boolean flushCache,
    boolean useCache,
    boolean resultOrdered,
    KeyGenerator keyGenerator,
    String keyProperty,
    String keyColumn,
    String databaseId,
    LanguageDriver lang,
    String resultSets) {

    if (unresolvedCacheRef) {
        throw new IncompleteElementException("Cache-ref not yet resolved");
    }
}

```

```

    }

    id = applyCurrentNamespace(id, false);
    boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

    MappedStatement.Builder statementBuilder = new
MappedStatement.Builder(configuration, id, sqlSource, sqlCommandType)
        .resource(resource)
        .fetchSize(fetchSize)
        .timeout(timeout)
        .statementType(statementType)
        .keyGenerator(keyGenerator)
        .keyProperty(keyProperty)
        .keyColumn(keyColumn)
        .databaseId(databaseId)
        .lang(lang)
        .resultOrdered(resultOrdered)
        .resultSets(resultSets)
        .resultMaps(getStatementResultMaps(resultMap, resultType, id))
        .resultSetType(resultSetType)
        .flushCacheRequired(valueOrDefault(flushCache, !isSelect))
        .useCache(valueOrDefault(useCache, isSelect))
        .cache(currentCache);

    // 创建语句参数映射
    ParameterMap statementParameterMap = getStatementParameterMap(parameterMap,
parameterType, id);
    if (statementParameterMap != null) {
        statementBuilder.parameterMap(statementParameterMap);
    }

    MappedStatement statement = statementBuilder.build();
    configuration.addMappedStatement(statement);
    return statement;
}

```

MappedStatement本身构建过程很简单，没什么复杂的逻辑，关键部分都在SqlSource的创建上。其中configuration.addMappedStatement里面进行了防重复处理。

4、最后为SelectKey对应的sql语句创建并维护一个KeyGenerator。

解析SQL主体

这一部分到mybatis语言驱动器一节一起讲解。

到此为止，crud部分的解析和加载就完成了。现在我们来看看这个语言驱动器。

sql语句解析的核心：mybatis语言驱动器XMLLanguageDriver

虽然官方名称叫做LanguageDriver，其实叫做解析器可能更加合理。MyBatis 从 3.2 开始支持可插拔的脚本语言，因此你可以在插入一种语言的驱动（language driver）之后来写基于这种语言的动态 SQL 查询比如mybatis除了XML格式外，还提供了mybatis-velocity，允许使用velocity表达式编写SQL语句。可以通过实现LanguageDriver接口的方式来插入一种语言：

```
public interface LanguageDriver {
    ParameterHandler createParameterHandler(MappedStatement mappedStatement, Object
parameterObject, BoundSql boundSql);
    SqlSource createSqlSource(Configuration configuration, XNode script, Class<?>
parameterType);
    SqlSource createSqlSource(Configuration configuration, String script, Class<?>
parameterType);
}
```

一旦有了自定义的语言驱动，你就可以在 mybatis-config.xml 文件中将它设置为默认语言：

```
<typeAliases>
  <typeAlias type="org.sample.MyLanguageDriver" alias="myLanguage"/>
</typeAliases>
<settings>
  <setting name="defaultScriptingLanguage" value="myLanguage"/>
</settings>
```

除了设置默认语言，你也可以针对特殊的语句指定特定语言，这可以通过如下的 lang 属性来完成：

```
<select id="selectBlog" lang="myLanguage">
  SELECT * FROM BLOG
</select>
```

默认情况下，mybatis使用org.apache.ibatis.scripting.xmltags.XMLLanguageDriver。通过调用createSqlSource方法来创建SqlSource，如下：

```
@Override
public SqlSource createSqlSource(Configuration configuration, XNode script, Class<?>
parameterType) {
    XMLScriptBuilder builder = new XMLScriptBuilder(configuration, script,
parameterType);
    return builder.parseScriptNode();
}

@Override
public SqlSource createSqlSource(Configuration configuration, String script, Class<?>
parameterType) {
    // issue #3
    if (script.startsWith("<script>")) {
```

```

        XPathParser parser = new XPathParser(script, false, configuration.getVariables(),
new XMLMapperEntityResolver());
        return createSqlSource(configuration, parser.evalNode("/script"), parameterType);
    } else {
        // issue #127
        script = PropertyParser.parse(script, configuration.getVariables());
        TextSqlNode textSqlNode = new TextSqlNode(script);
        if (textSqlNode.isDynamic()) {
            return new DynamicSqlSource(configuration, textSqlNode);
        } else {
            return new RawSqlSource(configuration, script, parameterType);
        }
    }
}

```

他有两个重载版本，第二个主要是传递文本的格式，所以我们重点分析第一个版本。XMLScriptBuilder构造器中设置相关字段外，还硬编码了每个支持的动态元素对应的处理器。如果我们要支持额外的动态元素比如else/elseif，只要在map中添加对应的key/value对即可。

```

public XMLScriptBuilder(Configuration configuration, XNode context, Class<?>
parameterType) {
    super(configuration);
    this.context = context;
    this.parameterType = parameterType;
    initNodeHandlerMap();
}

private void initNodeHandlerMap() {
    nodeHandlerMap.put("trim", new TrimHandler());
    nodeHandlerMap.put("where", new WhereHandler());
    nodeHandlerMap.put("set", new SetHandler());
    nodeHandlerMap.put("foreach", new ForEachHandler());
    nodeHandlerMap.put("if", new IfHandler());
    nodeHandlerMap.put("choose", new ChooseHandler());
    nodeHandlerMap.put("when", new IfHandler());
    nodeHandlerMap.put("otherwise", new OtherwiseHandler());
    nodeHandlerMap.put("bind", new BindHandler());
}

```

这里采用了内部类的设计，内部类只有在有外部类实例的情况下才会存在。在这里因为内部类不会单独被使用，所以应该设计为内部类而不是内部静态类。每个动态元素处理类都实现了NodeHandler接口且有对应的SqlNode比如IfSqlNode。

现在来看parseScriptNode()的实现：

```

public SqlSource parseScriptNode() {
    // 解析动态标签
    MixedSqlNode rootSqlNode = parseDynamicTags(context);
    SqlSource sqlSource = null;
}

```

```

    if (isDynamic) {
        sqlSource = new DynamicSqlSource(configuration, rootSqlNode);
    } else {
        sqlSource = new RawSqlSource(configuration, rootSqlNode, parameterType);
    }
    return sqlSource;
}

// 动态标签解析实现
protected MixedSqlNode parseDynamicTags(XNode node) {
    List<SqlNode> contents = new ArrayList<SqlNode>();
    NodeList children = node.getNode().getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        XNode child = node.newXNode(children.item(i));
        if (child.getNode().getNodeType() == Node.CDATA_SECTION_NODE ||
child.getNode().getNodeType() == Node.TEXT_NODE) {
            String data = child.getStringBody("");
            TextSqlNode textSqlNode = new TextSqlNode(data);
            if (textSqlNode.isDynamic()) { // 判断文本节点中是否包含了${}, 如果包含则为动态文本节点, 否则为静态文本节点, 静态文本节点在运行时不需要二次处理
                contents.add(textSqlNode);
                isDynamic = true;
            } else {
                contents.add(new StaticTextSqlNode(data));
            }
        } else if (child.getNode().getNodeType() == Node.ELEMENT_NODE) { // issue #628 标签节点
            String nodeName = child.getNode().getNodeName();
            // 首先根据节点名称获取到对应的节点处理器
            NodeHandler handler = nodeHandlerMap.get(nodeName);
            if (handler == null) {
                throw new BuilderException("Unknown element <" + nodeName + "> in SQL statement.");
            }
            // 使用对应的节点处理器处理本节点
            handler.handleNode(child, contents);
            isDynamic = true;
        }
    }
    return new MixedSqlNode(contents);
}

```

在解析mapper语句的时候, 很重要的一个步骤是解析动态标签(动态指的是SQL文本里面包含了\${}动态变量或者包含等元素的sql节点, 它将合成为SQL语句的一部分发送给数据库), 然后根据是否动态sql决定实例化的SqlSource为DynamicSqlSource或RawSqlSource。

可以说, mybatis是通过动态标签的实现来解决传统JDBC编程中sql语句拼接这个步骤的(就像现代web前端开发使用模板或vdom自动绑定代替jquery字符串拼接一样), mybatis动态标签被设计为可以相互嵌套, 所以对于动态标签的解析需要递归直到解析至文本节点。一个映射语句下可以包含多个根动态标签, 因此最后返回的是一个MixedSqlNode, 其中有一个List类型的属性, 包含树状层次嵌套的多种SqlNode实现类型的列表, 也就是单向链

表的其中一种衍生形式。MixedSqlNode的定义如下：

```
public class MixedSqlNode implements SqlNode {
    private final List<SqlNode> contents;

    public MixedSqlNode(List<SqlNode> contents) {
        this.contents = contents;
    }

    @Override
    public boolean apply(DynamicContext context) {
        // 遍历每个根SqlNode
        for (SqlNode sqlNode : contents) {
            sqlNode.apply(context);
        }
        return true;
    }
}
```

这样在运行的时候，就只要根据对应的表达式结果(mybatis采用ONGL作为动态表达式语言)调用下一个SqlNode进行计算即可。

IfHandler

IfHandler的实现如下：

```
private class IfHandler implements NodeHandler {
    public IfHandler() {
        // Prevent Synthetic Access
    }

    @Override
    public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
        MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
        // 获取if属性的值，将值设置为IfSqlNode的属性，便于运行时解析
        String test = nodeToHandle.getStringAttribute("test");
        IfSqlNode ifSqlNode = new IfSqlNode(mixedSqlNode, test);
        targetContents.add(ifSqlNode);
    }
}
```

OtherwiseHandler

otherwise标签可以说并不是一个真正有意义的标签，它不做任何处理，用在choose标签的最后默认分支，如下所示：

```
private class OtherwiseHandler implements NodeHandler {
    public OtherwiseHandler() {
        // Prevent Synthetic Access
    }

    @Override
    public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
        MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
        targetContents.add(mixedSqlNode);
    }
}
```

BindHandler

bind 元素可以使用 OGNL 表达式创建一个变量并将其绑定到当前SQL节点的上下文。

```
<select id="selectBlogsLike" parameterType="BlogQuery" resultType="Blog">
    <bind name="pattern" value="'%' + title + '%'" />
    SELECT * FROM BLOG
    WHERE title LIKE #{pattern}
</select>
```

对于这种情况，bind还可以用来预防 SQL 注入。

```
private class BindHandler implements NodeHandler {
    public BindHandler() {
        // Prevent Synthetic Access
    }

    @Override
    public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
        // 变量名称
        final String name = nodeToHandle.getStringAttribute("name");
        // OGNL表达式
        final String expression = nodeToHandle.getStringAttribute("value");
        final VarDeclSqlNode node = new VarDeclSqlNode(name, expression);
        targetContents.add(node);
    }
}
```

ChooseHandler

choose节点应该说和switch是等价的，其中的when就是各种条件判断。如下所示：

```
private class ChooseHandler implements NodeHandler {
    public ChooseHandler() {
        // Prevent Synthetic Access
    }
}
```

```

@Override
public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
    List<SqlNode> whenSqlNodes = new ArrayList<SqlNode>();
    List<SqlNode> otherwiseSqlNodes = new ArrayList<SqlNode>();
    // 拆分出when 和 otherwise 节点
    handleWhenOtherwiseNodes(nodeToHandle, whenSqlNodes, otherwiseSqlNodes);
    SqlNode defaultSqlNode = getDefaultSqlNode(otherwiseSqlNodes);
    ChooseSqlNode chooseSqlNode = new ChooseSqlNode(whenSqlNodes, defaultSqlNode);
    targetContents.add(chooseSqlNode);
}

private void handleWhenOtherwiseNodes(XNode chooseSqlNode, List<SqlNode>
ifSqlNodes, List<SqlNode> defaultSqlNodes) {
    List<XNode> children = chooseSqlNode.getChildren();
    for (XNode child : children) {
        String nodeName = child.getNode().getNodeName();
        NodeHandler handler = nodeHandlerMap.get(nodeName);
        if (handler instanceof IfHandler) {
            handler.handleNode(child, ifSqlNodes);
        } else if (handler instanceof OtherwiseHandler) {
            handler.handleNode(child, defaultSqlNodes);
        }
    }
}

private SqlNode getDefaultSqlNode(List<SqlNode> defaultSqlNodes) {
    SqlNode defaultSqlNode = null;
    if (defaultSqlNodes.size() == 1) {
        defaultSqlNode = defaultSqlNodes.get(0);
    } else if (defaultSqlNodes.size() > 1) {
        throw new BuilderException("Too many default (otherwise) elements in choose
statement.");
    }
    return defaultSqlNode;
}
}

```

ForEachHandler

foreach可以将任何可迭代对象（如列表、集合等）和任何的字典或者数组对象传递给foreach作为集合参数。当使用可迭代对象或者数组时，index是当前迭代的次数，item的值是本次迭代获取的元素。当使用字典（或者Map.Entry对象的集合）时，index是键，item是值。

```

private class ForEachHandler implements NodeHandler {
    public ForEachHandler() {
        // Prevent Synthetic Access
    }
}

```

```

@Override
public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
    MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
    String collection = nodeToHandle.getStringAttribute("collection");
    String item = nodeToHandle.getStringAttribute("item");
    String index = nodeToHandle.getStringAttribute("index");
    String open = nodeToHandle.getStringAttribute("open");
    String close = nodeToHandle.getStringAttribute("close");
    String separator = nodeToHandle.getStringAttribute("separator");
    ForEachSqlNode forEachSqlNode = new ForEachSqlNode(configuration, mixedSqlNode,
collection, index, item, open, close, separator);
    targetContents.add(forEachSqlNode);
}
}

```

SetHandler

set标签主要用于解决update动态字段，例如：

```

<update id="updateAuthorIfNecessary">
    update Author
    <set>
        <if test="username != null">username=#{username},</if>
        <if test="password != null">password=#{password},</if>
        <if test="email != null">email=#{email},</if>
        <if test="bio != null">bio=#{bio}</if>
    </set>
    where id=#{id}
</update>

```

一般来说，在实际中我们应该增加至少一个额外的最后更新时间字段(mysql内置)或者更新人比较合适，并不需要使用动态set。

```

private class SetHandler implements NodeHandler {
    public SetHandler() {
        // Prevent Synthetic Access
    }

    @Override
    public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
        MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
        SetSqlNode set = new SetSqlNode(configuration, mixedSqlNode);
        targetContents.add(set);
    }
}

```

因为在set中内容为空的时候，set会被trim掉，所以set实际上是trim的一种特殊实现。

TrimHandler

trim使用最多的情况是截掉where条件中的前置OR和AND，update的set子句中的后置“,”，同时在内容不为空的时候加上where和set。

比如：

```
select * from user
<trim prefix="WHERE" prefixoverride="AND |OR">
  <if test="name != null and name.length()>0"> AND name=#{name}</if>
  <if test="gender != null and gender.length()>0"> AND gender=#{gender}</if>
</trim>
update user
<trim prefix="set" suffixoverride="," suffix=" where id = #{id} ">
  <if test="name != null and name.length()>0"> name=#{name} , </if>
  <if test="gender != null and gender.length()>0"> gender=#{gender} , </if>
</trim>
```

虽然这两者都可以通过非动态标签来解决。从性能角度来说，这两者是可以避免的。

```
private class TrimHandler implements NodeHandler {
    public TrimHandler() {
        // Prevent Synthetic Access
    }

    @Override
    public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {
        MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);
        // 包含的子节点解析后SQL文本不为空时要添加的前缀内容
        String prefix = nodeToHandle.getStringAttribute("prefix");
        // 要覆盖的子节点解析后SQL文本前缀内容
        String prefixOverrides = nodeToHandle.getStringAttribute("prefixOverrides");
        // 包含的子节点解析后SQL文本不为空时要添加的后缀内容
        String suffix = nodeToHandle.getStringAttribute("suffix");
        // 要覆盖的子节点解析后SQL文本后缀内容
        String suffixOverrides = nodeToHandle.getStringAttribute("suffixOverrides");
        TrimSqlNode trim = new TrimSqlNode(configuration, mixedSqlNode, prefix,
            prefixOverrides, suffix, suffixOverrides);
        targetContents.add(trim);
    }
}
```

WhereHandler

和set一样，where也是trim的特殊情况，同样where标签也不是必须的，可以通过方式解决。


```

public class WhereSqlNode extends TrimSqlNode {

    private static List<String> prefixList = Arrays.asList("AND ", "OR ", "AND\n", "OR\n",
"AND\r", "OR\r", "AND\t", "OR\t");

    public WhereSqlNode(Configuration configuration, SqlNode contents) {
        super(configuration, contents, "WHERE", prefixList, null, null);
    }

}

```

静态SQL创建RawSqlSource

先来看静态SQL的创建，这是两方面原因：一、静态SQL是动态SQL的一部分；二、静态SQL最终被传递给JDBC原生API发送到数据库执行。

```

public class RawSqlSource implements SqlSource {

    private final SqlSource sqlSource;

    public RawSqlSource(Configuration configuration, SqlNode rootSqlNode, Class<?>
parameterType) {
        this(configuration, getSql(configuration, rootSqlNode), parameterType);
    }

    public RawSqlSource(Configuration configuration, String sql, Class<?> parameterType)
{
        SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration);
        Class<?> clazz = parameterType == null ? Object.class : parameterType;
        sqlSource = sqlSourceParser.parse(sql, clazz, new HashMap<String, Object>());
    }

    private static String getSql(Configuration configuration, SqlNode rootSqlNode) {
        DynamicContext context = new DynamicContext(configuration, null);
        rootSqlNode.apply(context);
        return context.getSql();
    }

    ...
}

```

其中的关键之处在getSql()方法的逻辑实现，因为给DynamicContext()构造器传递的parameterObject为空,所以没有参数，也不需要反射，反之就通过反射将object转为map。因为rootSqlNode是StaticTextSqlNode类型，所以getSql就直接返回原文本，随后调用第二个构造器，首先创建一个SqlSourceBuilder实例，然后调用其parse()方法，其中ParameterMappingTokenHandler符号处理器的目的是把sql参数解析出来，如下所示：

```

public SqlSource parse(String originalSql, Class<?> parameterType, Map<String,
Object> additionalParameters) {
    ParameterMappingTokenHandler handler = new
ParameterMappingTokenHandler(configuration, parameterType, additionalParameters);
    GenericTokenParser parser = new GenericTokenParser("#{", "}", handler);
    String sql = parser.parse(originalSql);
    return new StaticSqlSource(configuration, sql, handler.getParameterMappings());
}

```

其中参数的具体解析在ParameterMappingTokenHandler.buildParameterMapping()中，如下：

```

private ParameterMapping buildParameterMapping(String content) {
    Map<String, String> propertiesMap = parseParameterMapping(content);
    String property = propertiesMap.get("property");
    Class<?> propertyType;
    if (metaParameters.hasGetter(property)) { // issue #448 get type from additional
params
        propertyType = metaParameters.getGetterType(property);
    } else if (typeHandlerRegistry.hasTypeHandler(parameterType)) {
        propertyType = parameterType;
    } else if (JdbcType.CURSOR.name().equals(propertiesMap.get("jdbcType"))) {
        propertyType = java.sql.ResultSet.class;
    } else if (property == null || Map.class.isAssignableFrom(parameterType)) {
        propertyType = Object.class;
    } else {
        MetaClass metaClass = MetaClass.forClass(parameterType,
configuration.getReflectorFactory());
        if (metaClass.hasGetter(property)) {
            propertyType = metaClass.getGetterType(property);
        } else {
            propertyType = Object.class;
        }
    }
    ParameterMapping.Builder builder = new ParameterMapping.Builder(configuration,
property, propertyType);
    Class<?> javaType = propertyType;
    String typeHandlerAlias = null;
    for (Map.Entry<String, String> entry : propertiesMap.entrySet()) {
        String name = entry.getKey();
        String value = entry.getValue();
        if ("javaType".equals(name)) {
            javaType = resolveClass(value);
            builder.javaType(javaType);
        } else if ("jdbcType".equals(name)) {
            builder.jdbcType(resolveJdbcType(value));
        } else if ("mode".equals(name)) {
            builder.mode(resolveParameterMode(value));
        } else if ("numericScale".equals(name)) {

```

```

        builder.numericScale(Integer.valueOf(value));
    } else if ("resultMap".equals(name)) {
        builder.resultMapId(value);
    } else if ("typeHandler".equals(name)) {
        typeHandlerAlias = value;
    } else if ("jdbcTypeName".equals(name)) {
        builder.jdbcTypeName(value);
    } else if ("property".equals(name)) {
        // Do Nothing
    } else if ("expression".equals(name)) {
        throw new BuilderException("Expression based parameters are not supported yet");
    } else {
        throw new BuilderException("An invalid property '" + name + "' was found in mapping #{' + content + "}. Valid properties are " + parameterProperties);
    }
}
if (typeHandlerAlias != null) {
    builder.typeHandler(resolveTypeHandler(javaType, typeHandlerAlias));
}
return builder.build();
}

```

其总体逻辑还是比较简单的，唯一值得注意的是，只不过在这里可以看出每个参数被绑定的javaType获取的优先级分别为：#{ }中明确指定的，调用parse时传递的map(metaParameters)中包含的(这主要用于动态SQL的场景)，调用parse时传递的参数类型(如果该类型已经在typeHandlerRegistry中的话)，参数指定的jdbcType类型为JdbcType.CURSOR，Object.class(如果该类型是Map的子类或者参数本身为null)，参数包含在调用parse时传递的参数类型对象的字段中，最后是Object.class。最后构建出ParameterMapping，如下所示：

```

public class ParameterMapping {

    private Configuration configuration;

    private String property;
    private ParameterMode mode;
    private Class<?> javaType = Object.class;
    private JdbcType jdbcType;
    private Integer numericScale;
    private TypeHandler<?> typeHandler;
    private String resultMapId;
    private String jdbcTypeName;
    private String expression;
    ...
}

```

对于下列调用：

```
RawSqlSource rawSqlSource = new RawSqlSource(conf, "SELECT * FROM PERSON WHERE ID = #{id}", int.class);
```

将返回

sql语句: `SELECT * FROM PERSON WHERE ID = ?`

以及参数列表: `[ParameterMapping{property='id', mode=IN, javaType=int, jdbcType=null, numericScale=null, resultMapId='null', jdbcTypeName='null', expression='null'}]`

到此为止，静态SQL的解析就完成了。

动态SQL创建 DynamicSqlSource

再来看动态SQL的创建。动态SQL主要在运行时由上下文调用SqlSource.getBoundSql()接口获取。它在处理了动态标签以及\${}之后,调用静态SQL构建器返回PreparedStatement, 也就是静态SQL形式。

到此为止，整个mapper文件的第一轮解析就完成了。

4、二次解析未完成的结果映射、缓存参照、CRUD语句；

在第一次解析期间，有可能因为加载顺序或者其他原因，导致部分依赖的resultMap或者其他没有完全完成解析，所以针对pending的结果映射、缓存参考、CRUD语句进行递归重新解析直到完成。

3 关键对象总结与回顾

3.1 SqlSource

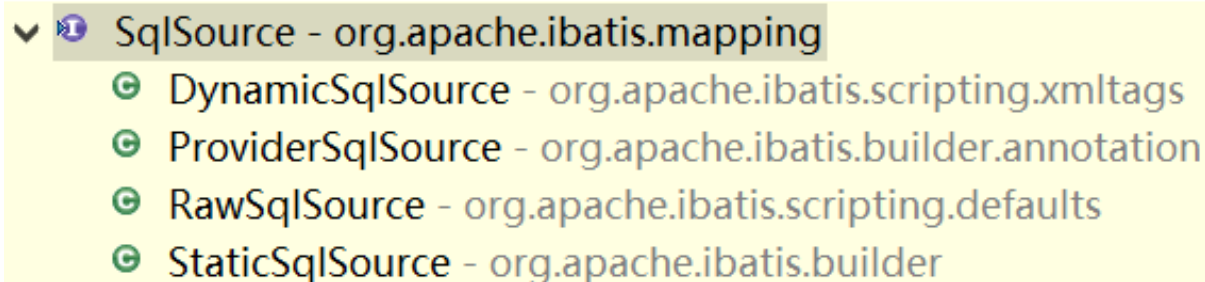
SqlSource是XML文件或者注解方法中映射语句的实现时表示，通过SqlSourceBuilder.parse()方法创建，SqlSourceBuilder中符号解析器将mybatis中的查询参数#{转换为?, 并记录了参数的顺序。它只有一个方法getBoundSql用于获取映射语句对象的各个组成部分，它的定义如下：

```
/**
 * Represents the content of a mapped statement read from an XML file or an annotation.
 * It creates the SQL that will be passed to the database out of the input parameter
 * received from the user.
 * 代表从XML文件或者注解读取的映射语句的内容,它创建的SQL会被传递给数据库。
 * @author Clinton Begin
 */
public interface SqlSource {

    BoundSql getBoundSql(Object parameterObject);

}
```

根据SQL语句的类型不同，mybatis提供了多种SqlSource的具体实现，如下所示：



- StaticSqlSource：最终静态SQL语句的封装,其他类型的SqlSource最终都委托给StaticSqlSource。
- RawSqlSource：原始静态SQL语句的封装,在加载时就已经确定了SQL语句,没有、等动态标签和\${} SQL拼接,比动态SQL语句要快,因为不需要运行时解析SQL节点。
- DynamicSqlSource：动态SQL语句的封装，在运行时需要根据参数处理、等标签或者\${} SQL拼接之后才能生成最后要执行的静态SQL语句。
- ProviderSqlSource：当SQL语句通过指定的类和方法获取时(使用@XXXProvider注解)，需要使用本类，它会通过反射调用相应的方法得到SQL语句。

3.2 SqlNode

SqlNode接口主要用来处理CRUD节点下的各类动态标签比如、，对每个动态标签，mybatis都提供了对应的SqlNode实现，这些动态标签可以相互嵌套且实现上采用单向链表进行应用，这样后面如果需要增加其他动态标签，就只需要新增对应的SqlNode实现就能支持。mybatis使用OGNL表达式语言。对SqlNode的调用在SQL执行期间的DynamicSqlSource.getBoundSql()方法中，SQL执行过程我们后面会讲解。

当前版本的SqlNode有下列实现：

SqlNode实现类

其中MixedSqlNode代表了所有具体SqlNode的集合，其他分别代表了一种类型的SqlNode。下面对每个SqlNode的实现做简单的分析：

ChooseSqlNode

```
public class ChooseSqlNode implements SqlNode {
    private final SqlNode defaultSqlNode;
    private final List<SqlNode> ifSqlNodes;

    public ChooseSqlNode(List<SqlNode> ifSqlNodes, SqlNode defaultSqlNode) {
        this.ifSqlNodes = ifSqlNodes;
        this.defaultSqlNode = defaultSqlNode;
    }

    @Override
    public boolean apply(DynamicContext context) {
        // 遍历所有when分支节点，只要遇到第一个为true就返回
        for (SqlNode sqlNode : ifSqlNodes) {
            if (sqlNode.apply(context)) {
                return true;
            }
        }
        // 全部when都为false时，走otherwise分支
    }
}
```

```

        if (defaultSqlNode != null) {
            defaultSqlNode.apply(context);
            return true;
        }
        return false;
    }
}

```

ForEachSqlNode

```

public class ForEachSqlNode implements SqlNode {
    public static final String ITEM_PREFIX = "__frch_";

    private final ExpressionEvaluator evaluator;
    private final String collectionExpression;
    private final SqlNode contents;
    private final String open;
    private final String close;
    private final String separator;
    private final String item;
    private final String index;
    private final Configuration configuration;

    public ForEachSqlNode(Configuration configuration, SqlNode contents, String
collectionExpression, String index, String item, String open, String close, String
separator) {
        this.evaluator = new ExpressionEvaluator();
        this.collectionExpression = collectionExpression;
        this.contents = contents;
        this.open = open;
        this.close = close;
        this.separator = separator;
        this.index = index;
        this.item = item;
        this.configuration = configuration;
    }

    @Override
    public boolean apply(DynamicContext context) {
        Map<String, Object> bindings = context.getBindings();
        // 将Map/Array/List统一包装为迭代器接口
        final Iterable<?> iterable = evaluator.evaluateIterable(collectionExpression,
bindings);
        if (!iterable.iterator().hasNext()) {
            return true;
        }
        boolean first = true;
        applyOpen(context);
        int i = 0;
    }
}

```

```

// 遍历集合
for (Object o : iterable) {
    DynamicContext oldContext = context;
    if (first || separator == null) {
        context = new PrefixedContext(context, "");
    } else {
        context = new PrefixedContext(context, separator);
    }
    int uniqueNumber = context.getUniqueNumber();
    // Issue #709
    if (o instanceof Map.Entry) { //Map条目处理
        @SuppressWarnings("unchecked")
        Map.Entry<Object, Object> mapEntry = (Map.Entry<Object, Object>) o;
        applyIndex(context, mapEntry.getKey(), uniqueNumber);
        applyItem(context, mapEntry.getValue(), uniqueNumber);
    } else { // List条目处理
        applyIndex(context, i, uniqueNumber);
        applyItem(context, o, uniqueNumber);
    }
    // 子节点SqlNode处理, 很重要的一个逻辑就是将#{item.XXX}转换为#{__frch_item_N.XXX}, 这样在
    JDBC设置参数的时候就能够找到对应的参数值了
    contents.apply(new FilteredDynamicContext(configuration, context, index, item,
uniqueNumber));
    if (first) {
        first = !((PrefixedContext) context).isPrefixApplied();
    }
    context = oldContext;
    i++;
}
applyClose(context);
context.getBindings().remove(item);
context.getBindings().remove(index);
return true;
}

private void applyIndex(DynamicContext context, Object o, int i) {
    if (index != null) {
        context.bind(index, o);
        context.bind(itemizeItem(index, i), o);
    }
}

private void applyItem(DynamicContext context, Object o, int i) {
    if (item != null) {
        context.bind(item, o);
        context.bind(itemizeItem(item, i), o);
    }
}

```

```

private void applyOpen(DynamicContext context) {
    if (open != null) {
        context.appendSql(open);
    }
}

private void applyClose(DynamicContext context) {
    if (close != null) {
        context.appendSql(close);
    }
}

private static String itemizeItem(String item, int i) {
    return new
String
Builder(ITEM_PREFIX).append(item).append("_").append(i).toString();
}

private static class FilteredDynamicContext extends DynamicContext {
    private final DynamicContext delegate;
    private final int index;
    private final String itemIndex;
    private final String item;

    public FilteredDynamicContext(Configuration configuration, DynamicContext delegate,
String itemIndex, String item, int i) {
        super(configuration, null);
        this.delegate = delegate;
        this.index = i;
        this.itemIndex = itemIndex;
        this.item = item;
    }

    @Override
    public Map<String, Object> getBindings() {
        return delegate.getBindings();
    }

    @Override
    public void bind(String name, Object value) {
        delegate.bind(name, value);
    }

    @Override
    public String getSql() {
        return delegate.getSql();
    }

    @Override
    public void appendSql(String sql) {

```



```

        GenericTokenParser parser = new GenericTokenParser("#{", "}", new TokenHandler()
{
    @Override
    // 将#{item.XXX}转换为#{__frch_item_N.XXX}
    public String handleToken(String content) {
        String newContent = content.replaceFirst("^\\s*" + item + "(?![^\s])",
itemizeItem(item, index));
        if (itemIndex != null && newContent.equals(content)) {
            newContent = content.replaceFirst("^\\s*" + itemIndex + "(?![^\s])",
itemizeItem(itemIndex, index));
        }
        return new StringBuilder("#{").append(newContent).append(")").toString();
    }
});

    delegate.appendSql(parser.parse(sql));
}

@Override
public int getUniqueNumber() {
    return delegate.getUniqueNumber();
}

}

private class PrefixedContext extends DynamicContext {
    private final DynamicContext delegate;
    private final String prefix;
    private boolean prefixApplied;

    public PrefixedContext(DynamicContext delegate, String prefix) {
        super(configuration, null);
        this.delegate = delegate;
        this.prefix = prefix;
        this.prefixApplied = false;
    }

    public boolean isPrefixApplied() {
        return prefixApplied;
    }

    @Override
    public Map<String, Object> getBindings() {
        return delegate.getBindings();
    }

    @Override
    public void bind(String name, Object value) {

```

```

        delegate.bind(name, value);
    }

    @Override
    public void appendSql(String sql) {
        if (!prefixApplied && sql != null && sql.trim().length() > 0) {
            delegate.appendSql(prefix);
            prefixApplied = true;
        }
        delegate.appendSql(sql);
    }

    @Override
    public String getSql() {
        return delegate.getSql();
    }

    @Override
    public int getUniqueNumber() {
        return delegate.getUniqueNumber();
    }
}
}

```

IfSqlNode

```

public class IfSqlNode implements SqlNode {
    private final ExpressionEvaluator evaluator; //表达式执行器
    private final String test; //条件表达式
    private final SqlNode contents;

    public IfSqlNode(SqlNode contents, String test) {
        this.test = test;
        this.contents = contents;
        this.evaluator = new ExpressionEvaluator();
    }

    @Override
    public boolean apply(DynamicContext context) {
        if (evaluator.evaluateBoolean(test, context.getBindings())) {
            contents.apply(context);
            return true;
        }
        return false;
    }
}

```

ExpressionEvaluator的定义如下:

```
public class ExpressionEvaluator {
    // 布尔表达式解析, 对于返回值为数字的if表达式, 0为假, 非0为真
    public boolean evaluateBoolean(String expression, Object parameterObject) {
        Object value = OgnlCache.getValue(expression, parameterObject);
        if (value instanceof Boolean) {
            return (Boolean) value;
        }
        if (value instanceof Number) {
            return new BigDecimal(String.valueOf(value)).compareTo(BigDecimal.ZERO) != 0;
        }
        return value != null;
    }

    // 循环表达式解析, 主要用于foreach标签
    public Iterable<?> evaluateIterable(String expression, Object parameterObject) {
        Object value = OgnlCache.getValue(expression, parameterObject);
        if (value == null) {
            throw new BuilderException("The expression '" + expression + "' evaluated to a null value.");
        }
        if (value instanceof Iterable) {
            return (Iterable<?>) value;
        }
        if (value.getClass().isArray()) {
            // the array may be primitive, so Arrays.asList() may throw
            // a ClassCastException (issue 209). Do the work manually
            // Curse primitives! :) (JGB)
            int size = Array.getLength(value);
            List<Object> answer = new ArrayList<Object>();
            for (int i = 0; i < size; i++) {
                Object o = Array.get(value, i);
                answer.add(o);
            }
            return answer;
        }
        if (value instanceof Map) {
            return ((Map) value).entrySet();
        }
        throw new BuilderException("Error evaluating expression '" + expression + "'. Return value (" + value + ") was not iterable.");
    }
}
```

StaticTextSqlNode

静态文本节点不做任何处理，直接将本文本节点的内容追加到已经解析了的SQL文本的后面。

```
public class StaticTextSqlNode implements SqlNode {
    private final String text;

    public StaticTextSqlNode(String text) {
        this.text = text;
    }

    @Override
    public boolean apply(DynamicContext context) {
        context.appendSql(text);
        return true;
    }
}
```

TextSqlNode

TextSqlNode主要是用来将\${}转换为实际的参数值，并返回拼接后的SQL语句，为了防止SQL注入，可以通过标签来创建OGNL上下文变量。

```
public class TextSqlNode implements SqlNode {
    private final String text;
    private final Pattern injectionFilter;

    public TextSqlNode(String text) {
        this(text, null);
    }

    public TextSqlNode(String text, Pattern injectionFilter) {
        this.text = text;
        this.injectionFilter = injectionFilter;
    }

    public boolean isDynamic() {
        DynamicCheckerTokenParser checker = new DynamicCheckerTokenParser();
        GenericTokenParser parser = createParser(checker);
        parser.parse(text);
        return checker.isDynamic();
    }

    @Override
    public boolean apply(DynamicContext context) {
        GenericTokenParser parser = createParser(new BindingTokenParser(context,
            injectionFilter));
        context.appendSql(parser.parse(text));
    }
}
```

```

    return true;
}

private GenericTokenParser createParser(TokenHandler handler) {
    return new GenericTokenParser("${", "}", handler);
}

private static class BindingTokenParser implements TokenHandler {

    private DynamicContext context;
    private Pattern injectionFilter;

    public BindingTokenParser(DynamicContext context, Pattern injectionFilter) {
        this.context = context;
        this.injectionFilter = injectionFilter;
    }
    // 将${}中的值替换为查询参数中实际的值并返回, 在StaticTextSqlNode中, #{}返回的是?
    @Override
    public String handleToken(String content) {
        Object parameter = context.getBindings().get("_parameter");
        if (parameter == null) {
            context.getBindings().put("value", null);
        } else if (SimpleTypeRegistry.isSimpleType(parameter.getClass())) {
            context.getBindings().put("value", parameter);
        }
        Object value = OgnlCache.getValue(content, context.getBindings());
        String srtValue = (value == null ? "" : String.valueOf(value)); // issue #274
return "" instead of "null"
        checkInjection(srtValue);
        return srtValue;
    }

    private void checkInjection(String value) {
        if (injectionFilter != null && !injectionFilter.matcher(value).matches()) {
            throw new ScriptingException("Invalid input. Please conform to regex" +
injectionFilter.pattern());
        }
    }
}

private static class DynamicCheckerTokenParser implements TokenHandler {

    private boolean isDynamic;

    public DynamicCheckerTokenParser() {
        // Prevent Synthetic Access
    }

    public boolean isDynamic() {

```

```

        return isDynamic;
    }

    @Override
    public String handleToken(String content) {
        this.isDynamic = true;
        return null;
    }
}
}

```

VarDeclSqlNode

```

public class VarDeclSqlNode implements SqlNode {

    private final String name;
    private final String expression;

    public VarDeclSqlNode(String var, String exp) {
        name = var;
        expression = exp;
    }

    @Override
    public boolean apply(DynamicContext context) {
        final Object value = OgnlCache.getValue(expression, context.getBindings());
        // 直接将ognl表达式加到当前映射语句的上下文中，这样就可以直接获取到了
        context.bind(name, value);
        return true;
    }
}

```

DynamicContext.bind方法的实现如下：

```

private final ContextMap bindings;
public void bind(String name, Object value) {
    bindings.put(name, value);
}

```

TrimSqlNode

```

public class TrimSqlNode implements SqlNode {

    private final SqlNode contents;
    private final String prefix;
    private final String suffix;
}

```

```

private final List<String> prefixesToOverride; // 要trim多个文本的话,|分隔即可
private final List<String> suffixesToOverride; // 要trim多个文本的话,|分隔即可
private final Configuration configuration;

public TrimSqlNode(Configuration configuration, SqlNode contents, String prefix,
String prefixesToOverride, String suffix, String suffixesToOverride) {
    this(configuration, contents, prefix, parseOverrides(prefixesToOverride), suffix,
parseOverrides(suffixesToOverride));
}

protected TrimSqlNode(Configuration configuration, SqlNode contents, String prefix,
List<String> prefixesToOverride, String suffix, List<String> suffixesToOverride) {
    this.contents = contents;
    this.prefix = prefix;
    this.prefixesToOverride = prefixesToOverride;
    this.suffix = suffix;
    this.suffixesToOverride = suffixesToOverride;
    this.configuration = configuration;
}

@Override
public boolean apply(DynamicContext context) {
    FilteredDynamicContext filteredDynamicContext = new
FilteredDynamicContext(context);
    // trim节点只有在至少有一个子节点不为空的时候才有意义
    boolean result = contents.apply(filteredDynamicContext);
    // 所有子节点处理完成之后,filteredDynamicContext.delegate里面就包含解析后的静态SQL文本了,此
    时就可以处理前后的trim了
    filteredDynamicContext.applyAll();
    return result;
}

private static List<String> parseOverrides(String overrides) {
    if (overrides != null) {
        final StringTokenizer parser = new StringTokenizer(overrides, "|", false);
        final List<String> list = new ArrayList<String>(parser.countTokens());
        while (parser.hasMoreTokens()) {
            list.add(parser.nextToken().toUpperCase(Locale.ENGLISH));
        }
        return list;
    }
    return Collections.emptyList();
}

private class FilteredDynamicContext extends DynamicContext {
    private DynamicContext delegate;
    private boolean prefixApplied;
    private boolean suffixApplied;
    private StringBuilder sqlBuffer;

```

```

public FilteredDynamicContext(DynamicContext delegate) {
    super(configuration, null);
    this.delegate = delegate;
    this.prefixApplied = false;
    this.suffixApplied = false;
    this.sqlBuffer = new StringBuilder();
}

public void applyAll() {
    sqlBuffer = new StringBuilder(sqlBuffer.toString().trim());
    String trimmedUppercaseSql = sqlBuffer.toString().toUpperCase(Locale.ENGLISH);
    if (trimmedUppercaseSql.length() > 0) {
        applyPrefix(sqlBuffer, trimmedUppercaseSql);
        applySuffix(sqlBuffer, trimmedUppercaseSql);
    }
    delegate.appendSql(sqlBuffer.toString());
}

@Override
public Map<String, Object> getBindings() {
    return delegate.getBindings();
}

@Override
public void bind(String name, Object value) {
    delegate.bind(name, value);
}

@Override
public int getUniqueNumber() {
    return delegate.getUniqueNumber();
}

@Override
public void appendSql(String sql) {
    sqlBuffer.append(sql);
}

@Override
public String getSql() {
    return delegate.getSql();
}

// 处理前缀
private void applyPrefix(StringBuilder sql, String trimmedUppercaseSql) {
    if (!prefixApplied) {
        prefixApplied = true;
        if (prefixesToOverride != null) {
            for (String toRemove : prefixesToOverride) {

```



```

        if (trimmedUppercaseSql.startsWith(toRemove)) {
            sql.delete(0, toRemove.trim().length());
            break;
        }
    }
}
if (prefix != null) {
    sql.insert(0, " ");
    sql.insert(0, prefix);
}
}
}

// 处理后缀
private void applySuffix(StringBuilder sql, String trimmedUppercaseSql) {
    if (!suffixApplied) {
        suffixApplied = true;
        if (suffixesToOverride != null) {
            for (String toRemove : suffixesToOverride) {
                if (trimmedUppercaseSql.endsWith(toRemove) ||
trimmedUppercaseSql.endsWith(toRemove.trim())) {
                    int start = sql.length() - toRemove.trim().length();
                    int end = sql.length();
                    sql.delete(start, end);
                    break;
                }
            }
        }
        if (suffix != null) {
            sql.append(" ");
            sql.append(suffix);
        }
    }
}
}
}
}

```

SetSqlNode

SetSqlNode直接委托给TrimSqlNode处理。参见TrimSqlNode。

WhereSqlNode

WhereSqlNode直接委托给TrimSqlNode处理。参见TrimSqlNode。

3.3 BaseBuilder

从整个设计角度来说，BaseBuilder的目的是为了统一解析的使用，但在实现上却出入较大。首先，BaseBuilder是所有解析类的MapperBuilderAssistant、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder等的父类。如下所示：

BaseBuilder实现类

BaseBuilder中提供类型处理器、JDBC类型、结果集类型、别名等的解析，因为在mybatis配置文件、mapper文件解析、SQL映射语句解析、基于注解的mapper文件解析过程中，都会频繁的遇到类型处理相关的解析。但是BaseBuilder也没有定义需要子类实现的负责解析的抽象接口，虽然XMLMapperBuilder、XMLConfigBuilder的解析入口是parse方法，XMLStatementBuilder的入口是parseStatementNode，不仅如此，MapperBuilderAssistant继承了BaseBuilder，而不是MapperAnnotationBuilder，实际上MapperAnnotationBuilder才是解析Mapper接口的主控类。

所以从实现上来说，BaseBuilder如果要作为具体Builder类的抽象父类，那就应该定义一个需要子类实现的parse接口，要么就用组合代替继承。

3.4 AdditionalParameter

额外参数主要是维护一些在加载时无法确定的参数，比如标签中的参数在加载时就无法尽最大努力确定，必须通过运行时执行org.apache.ibatis.scripting.xmltags.DynamicSqlSource.getBoundSql()中的SqlNode.apply()才能确定真正要执行的SQL语句，以及额外参数。比如，对于下列的foreach语句，它的AdditionalParameter内容为：

```
{frch_index_0=0, item=2, frch_index_1=1, parameter=org.mybatis.internal.example.pojo.UserReq@5ccddd20, index=1, frch_item_1=2, _databaseld=null, frch_item_0=1}
```

其中parameter和_databaseld在DynamicContext构造器中硬编码，其他值通过调用ForEachSqlNode.apply()计算得到。与此相对应，此时SQL语句在应用ForEachSqlNode之后，对参数名也进行重写，如下所示：

```
select lfPartyId,author as authors,subject,comments,title,partyName from
LfParty where partyName = #{partyName}

AND partyName like #{partyName}

and lfPartyId in
(
  #{__frch_item_0.prop}
,
  #{__frch_item_1}
)
```

然后通过SqlSourceBuilder.parse()调用ParameterMappingTokenHandler计算出该sql的ParameterMapping列表，最后构造出StaticSqlSource。

3.5 TypeHandler

当MyBatis将一个Java对象作为输入/输出参数执行CRUD语句操作时，它会创建一个PreparedStatement对象，并且调用setXXX()为占位符设置相应的参数值。XXX可以是Int, String, Date等Java内置类型，或者用户自定义的类型。在实现上，MyBatis是通过使用类型处理器（type handler）来确定XXX是具体什么类型的。MyBatis对于下列类型使用内建的类型处理器：所有的基本数据类型、基本类型的包裹类型、byte[]、java.util.Date、java.sql.Date、java.sql.Time、java.sql.Timestamp、java 枚举类型等。对于用户自定义的类型，我们可以创建一个自定义的类型处理器。要创建自定义类型处理器，只要实现TypeHandler接口即可，TypeHandler接口的定义如下：

```
public interface TypeHandler<T> {

    void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType) throws
SQLException;

    T getResult(ResultSet rs, String columnName) throws SQLException;

    T getResult(ResultSet rs, int columnIndex) throws SQLException;

    T getResult(CallableStatement cs, int columnIndex) throws SQLException;

}
```

虽然我们可以直接实现TypeHandler接口，但是在实践中，我们一般选择继承BaseTypeHandler，BaseTypeHandler为TypeHandler提供了部分骨架代码，使得用户使用方便，几乎所有mybatis内置类型处理器都继承于BaseTypeHandler。下面我们实现一个最简单的自定义类型处理器MobileTypeHandler。

```
public class MobileTypeHandler extends BaseTypeHandler<Mobile> {

    @Override
    public Mobile getNullableResult(ResultSet rs, String columnName)
        throws SQLException {
        // mobile字段是VARCHAR类型，所以使用rs.getString
        return new Mobile(rs.getString(columnName));
    }

    @Override
    public Mobile getNullableResult(ResultSet rs, int columnIndex)
        throws SQLException {
        return new Mobile(rs.getString(columnIndex));
    }

    @Override
    public Mobile getNullableResult(CallableStatement cs, int columnIndex)
        throws SQLException {
        return new Mobile(cs.getString(columnIndex));
    }

}
```

```

@Override
public void setNonNullParameter(PreparedStatement ps, int i,
    Mobile param, JdbcType jdbcType) throws SQLException {
    ps.setString(i, param.getFullNumber());
}
}

```

我们实现了自定义的类型处理器后，只要在mybatis配置文件mybatis-config.xml中注册就可以使用了，如下：

```

<typeHandlers>
    <typeHandler handler="org.mybatis.internal.example.MobileTypeHandler" />
</typeHandlers>

```

上述完成之后，当我们在parameterType或者resultType或者resultMap中遇到Mobile类型的属性时，就会调用MobileTypeHandler进行代理出入参的设置和获取。

3.6 对象包装器工厂 ObjectWrapperFactory

ObjectWrapperFactory是一个对象包装器工厂,用于对返回的结果对象进行二次处理,它主要在org.apache.ibatis.executor.resultset.DefaultResultSetHandler.getRowValue方法中创建对象的MetaObject时作为参数设置进去,这样MetaObject中的objectWrapper属性就可以被设置为我们自定义的ObjectWrapper实现而不是mybatis内置实现,如下所示：

```

private MetaObject(Object object, ObjectFactory objectFactory, ObjectWrapperFactory
objectWrapperFactory, ReflectorFactory reflectorFactory) {
    this.originalObject = object;
    this.objectFactory = objectFactory;
    this.objectWrapperFactory = objectWrapperFactory;
    this.reflectorFactory = reflectorFactory;

    if (object instanceof ObjectWrapper) {
        this.objectWrapper = (ObjectWrapper) object;
    } else if (objectWrapperFactory.hasWrapperFor(object)) { // 如果有自定义的
ObjectWrapperFactory,就不会总是返回false了,这样对于特定类就启用了的我们自定义的ObjectWrapper
        this.objectWrapper = objectWrapperFactory.getWrapperFor(this, object);
    } else if (object instanceof Map) {
        this.objectWrapper = new MapWrapper(this, (Map) object);
    } else if (object instanceof Collection) {
        this.objectWrapper = new CollectionWrapper(this, (Collection) object);
    } else {
        this.objectWrapper = new BeanWrapper(this, object);
    }
}

```

典型的下划线转驼峰,我们就可以使用ObjectWrapperFactory来统一处理(当然,在实际中,我们一般不会这么做,而是通过设置mapUnderscoreToCamelCase来实现)。ObjectWrapperFactory 接口如下：

```

public interface ObjectWrapperFactory {

    boolean hasWrapperFor(Object object);

    ObjectWrapper getWrapperFor(MetaObject metaObject, Object object);

}

```

通过实现这个接口，可以判断当object是特定类型时，返回true，然后在下面的getWrapperFor中返回一个可以处理key为驼峰的ObjectWrapper 实现类即可。ObjectWrapper类可以说是对象反射信息的facade模式，它的定义如下：

```

public interface ObjectWrapper {

    Object get(PropertyTokenizer prop);

    void set(PropertyTokenizer prop, Object value);

    String findProperty(String name, boolean useCamelCaseMapping);

    String[] getGetterNames();

    String[] getSetterNames();

    Class<?> getSetterType(String name);

    Class<?> getGetterType(String name);

    boolean hasSetter(String name);

    boolean hasGetter(String name);

    MetaObject instantiatePropertyValue(String name, PropertyTokenizer prop,
    ObjectFactory objectFactory);

    boolean isCollection();

    void add(Object element);

    <E> void addAll(List<E> element);

}

```

当然，我们不需要从头实现ObjectWrapper接口,可以选择继承BeanWrapper或者MapWrapper。比如对于Map类型，我们可以继承MapWrapper，让参数useCamelCaseMapping起作用。MapWrapper默认的findProperty方法并没有做驼峰转换处理，如下：

```

@Override
public String findProperty(String name, boolean useCamelCaseMapping) {
    return name;
}

```

我们可以改成：

```

public class CamelMapWrapper extends MapWrapper {
    public CamelMapWrapper(MetaObject metaObject, Map<String, Object> map) {
        super(metaObject, map);
    }

    @Override
    public String findProperty(String name, boolean useCamelCaseMapping) {
        if (useCamelCaseMapping
            && ((name.charAt(0) >= 'A' && name.charAt(0) <= 'Z')
                || name.indexOf("_") >= 0)) {
            return underlineToCamelhump(name);
        }
        return name;
    }

    /**
     * 将下划线风格替换为驼峰风格
     */
    public String underlineToCamelhump(String inputString) {
        StringBuilder sb = new StringBuilder();

        boolean nextUpperCase = false;
        for (int i = 0; i < inputString.length(); i++) {
            char c = inputString.charAt(i);
            if (c == '_') {
                if (sb.length() > 0) {
                    nextUpperCase = true;
                }
            } else {
                if (nextUpperCase) {
                    sb.append(Character.toUpperCase(c));
                    nextUpperCase = false;
                } else {
                    sb.append(Character.toLowerCase(c));
                }
            }
        }
        return sb.toString();
    }
}

```

同时，创建一个自定义的objectWrapperFactory如下：

```
public class CustomWrapperFactory implements ObjectWrapperFactory {

    @Override
    public boolean hasWrapperFor(Object object) {
        return object != null && object instanceof Map;
    }

    @Override
    public ObjectWrapper getWrapperFor(MetaObject metaObject, Object object) {
        return new CamelMapWrapper(metaObject, (Map) object);
    }

}
```

然后，在 MyBatis 配置文件中配置上objectWrapperFactory：

```
<objectWrapperFactory type="org.mybatis.internal.example.CustomWrapperFactory" />
```

同样，useCamelCaseMapping最终是通过mapUnderscoreToCamelCase设置注入进来的，所以settings要加上这个设置：

```
<setting name="mapUnderscoreToCamelCase" value="true" />
```

此时，如果resultType是map类型的话,就可以看到key已经是驼峰式而不是columnName了。

注意：mybatis提供了一个什么都不做的默认实现DefaultObjectWrapperFactory。

3.7 MetaObject

MetaObject是一个对象包装器，其性质上有点类似ASF提供的commons类库，其中包装了对象的元数据信息，对象本身，对象反射工厂，对象包装器工厂等。使得根据OGNL表达式设置或者获取对象的属性更为便利，也可以更加方便的判断对象中是否包含指定属性、指定属性是否具有getter、setter等。主要的功能是通过其ObjectWrapper类型的属性完成的，它包装了操作对象元数据以及对象本身的主要接口，操作标准对象的实现是BeanWrapper。BeanWrapper类型有个MetaClass类型的属性，MetaClass中有个Reflector属性，其中包含了可读、可写的属性、方法以及构造器信息。

3.8 对象工厂ObjectFactory

MyBatis 每次创建结果对象的新实例时，都会使用一个对象工厂（ObjectFactory）实例来完成。默认的对象工厂DefaultObjectFactory仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。如果想覆盖对象工厂的默认行为比如给某些属性设置默认值(有些时候直接修改对象不可行，或者由于不是自己拥有的代码或者改动太大)，则可以通过创建自己的对象工厂来实现。ObjectFactory接口定义如下：

```
public interface ObjectFactory {
```

```

/**
 * Sets configuration properties.
 * @param properties configuration properties
 */
void setProperties(Properties properties);

/**
 * Creates a new object with default constructor.
 * @param type Object type
 * @return
 */
<T> T create(Class<T> type);

/**
 * Creates a new object with the specified constructor and params.
 * @param type Object type
 * @param constructorArgTypes Constructor argument types
 * @param constructorArgs Constructor argument values
 * @return
 */
<T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object>
constructorArgs);

/**
 * Returns true if this object can have a set of other objects.
 * It's main purpose is to support non-java.util.Collection objects like Scala
collections.
 *
 * @param type Object type
 * @return whether it is a collection or not
 * @since 3.1.0
 */
<T> boolean isCollection(Class<T> type);
}

```

从这个接口定义可以看出，它包含了两种通过反射机制构造实体类对象的方法，一种是通过无参构造函数，一种是通过带参数的构造函数。同时，为了使工厂类能设置其他属性，还提供了setProperties()方法。

要自定义对象工厂类，我们可以实现ObjectFactory这个接口，但是这样我们就需要自己去实现一些在DefaultObjectFactory已经实现好了的东西，所以也可以继承这个DefaultObjectFactory类，这样可以使得实现起来更为简单。例如，我们希望给Order对象的属性hostname设置为本地机器名，可以像下面这么实现：

```

public class CustomObjectFactory extends DefaultObjectFactory{
    private static String hostname;
    static {
        InetAddress addr = InetAddress.getLocalHost();
        String ip=addr.getHostAddress().toString(); //获取本机ip
        hostName=addr.getHostName().toString(); //获取本机计算机名称
    }
}

```



```

    }

    private static final long serialVersionUID = 1128715667301891724L;

    @Override
    public <T> T create(Class<T> type) {
        T result = super.create(type);
        if(type.equals(Order.class)){
            ((Order)result).setIp(hostname);
        }
        return result;
    }
}

```

接下来，在配置文件中配置对象工厂类为我们创建的对象工厂类CustomObjectFactory。

```
<objectFactory type="org.mybatis.internal.example.CustomObjectFactory"></objectFactory>
```

此时执行代码，就会发现返回的Order对象中ip字段的值为本机名。

3.9 MappedStatement

mapper文件或者mapper接口中每个映射语句都对应一个MappedStatement实例，它包含了所有运行时需要的信息比如结果映射、参数映射、是否需要刷新缓存等。MappedStatement定义如下：

```

public final class MappedStatement {

    private String resource;
    private Configuration configuration;
    private String id;
    private Integer fetchSize;
    private Integer timeout;
    private StatementType statementType;
    private ResultSetType resultSetType;
    private SqlSource sqlSource;
    private Cache cache;
    private ParameterMap parameterMap;
    private List resultMaps;
    private boolean flushCacheRequired;
    private boolean useCache;
    private boolean resultOrdered;
    private SqlCommandType sqlCommandType;
    private KeyGenerator keyGenerator;
    private String[] keyProperties;
    private String[] keyColumns;
    private boolean hasNestedResultMaps;
    private String databaseId;
    private Log statementLog;
    private LanguageDriver lang;
    private String[] resultSets;
    ...
}

```

```

public MappedStatement build() {
    assert mappedStatement.configuration != null;
    assert mappedStatement.id != null;
    assert mappedStatement.sqlSource != null;
    assert mappedStatement.lang != null;
    mappedStatement.resultMaps = Collections.unmodifiableList(mappedStatement.resultMaps);
    return mappedStatement;
}
}
...
}

```

唯一值得注意的是resultMaps被设计为只读,这样应用可以查看但是不能修改。

3.10 ParameterMapping

每个参数映射<>标签都被创建一个ParameterMapping实例, 其中包含和结果映射类似的信息, 如下:

```

public class ParameterMapping {

    private Configuration configuration;

    private String property;
    private ParameterMode mode;
    private Class<?> javaType = Object.class;
    private JdbcType jdbcType;
    private Integer numericScale;
    private TypeHandler<?> typeHandler;
    private String resultMapId;
    private String jdbcTypeName;
    private String expression;

    private ParameterMapping() {
    }
    ...
}

```

3.11 KeyGenerator

```

package org.apache.ibatis.executor.keygen;

public interface KeyGenerator {
    // before key generator 主要用于oracle等使用序列机制的ID生成方式
    void processBefore(Executor executor, MappedStatement ms, Statement stmt, Object
parameter);
    // after key generator 主要用于mysql等使用自增机制的ID生成方式
    void processAfter(Executor executor, MappedStatement ms, Statement stmt, Object
parameter);

}

```

3.12 各种Registry

mybatis将类型处理器, 类型别名, mapper定义, 语言驱动器等各种信息包装在Registry中维护, 如下所示:

```

public class Configuration {
    ...
    protected final MapperRegistry mapperRegistry = new MapperRegistry(this);
    protected final InterceptorChain interceptorChain = new InterceptorChain();
    protected final TypeHandlerRegistry typeHandlerRegistry = new TypeHandlerRegistry();
    protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();
    protected final LanguageDriverRegistry languageRegistry = new
LanguageDriverRegistry();
    ...
}

```

各Registry中提供了相关的方法，比如TypeHandlerRegistry中包含了判断某个java类型是否有类型处理器以及获取类型处理器的方法，如下：

```

public boolean hasTypeHandler(TypeReference<?> javaTypeReference, JdbcType jdbcType)
{
    return javaTypeReference != null && getTypeHandler(javaTypeReference, jdbcType) !=
null;
}

public <T> TypeHandler<T> getTypeHandler(Class<T> type) {
    return getTypeHandler((Type) type, null);
}

```

3.13 LanguageDriver

从3.2版本开始，mybatis提供了LanguageDriver接口，我们可以使用该接口自定义SQL的解析方式。先来看下LanguageDriver接口中的3个方法：

```

public interface LanguageDriver {

    /**
     * Creates a {@link ParameterHandler} that passes the actual parameters to the the
     JDBC statement.
     * 创建一个ParameterHandler对象，用于将实际参数赋值到JDBC语句中
     *
     * @param mappedStatement The mapped statement that is being executed
     * @param parameterObject The input parameter object (can be null)
     * @param boundSql The resulting SQL once the dynamic language has been executed.
     * @return
     * @author Frank D. Martinez [mnesarco]
     * @see DefaultParameterHandler
     */
    ParameterHandler createParameterHandler(MappedStatement mappedStatement, Object
parameterObject, BoundSql boundSql);

    /**

```

```

    * Creates an {@link SqlSource} that will hold the statement read from a mapper xml
    file.
    * It is called during startup, when the mapped statement is read from a class or an
    xml file.
    * 将XML中读入的语句解析并返回一个sqlSource对象
    *
    * @param configuration The MyBatis configuration
    * @param script XNode parsed from a XML file
    * @param parameterType input parameter type got from a mapper method or specified in
    the parameterType xml attribute. Can be null.
    * @return
    */
    SqlSource createSqlSource(Configuration configuration, XNode script, Class<?>
    parameterType);

    /**
    * Creates an {@link SqlSource} that will hold the statement read from an annotation.
    * It is called during startup, when the mapped statement is read from a class or an
    xml file.
    * 将注解中读入的语句解析并返回一个sqlSource对象
    *
    * @param configuration The MyBatis configuration
    * @param script The content of the annotation
    * @param parameterType input parameter type got from a mapper method or specified in
    the parameterType xml attribute. Can be null.
    * @return
    */
    SqlSource createSqlSource(Configuration configuration, String script, Class<?>
    parameterType);
}

```

实现了LanguageDriver之后，可以在配置文件中指定该实现类作为SQL的解析器，在XML中我们可以使用lang 属性来进行指定，如下：

```

<typeAliases>
  <typeAlias type="org.sample.MyLanguageDriver" alias="myLanguage" />
</typeAliases>

<select id="selectBlog" lang="myLanguage">
  SELECT * FROM BLOG
</select>

```

除了可以在语句级别指定外，也可以全局设置，如下：

```

<settings>
  <setting name="defaultScriptingLanguage" value="myLanguage" />
</settings>

```

对于mapper接口，也可以使用@Lang注解，如下所示：

```
public interface Mapper {
    @Lang(MyLanguageDriver.class)
    @Select("SELECT * FROM users")
    List<User> selectUser();
}
```

LanguageDriver的默认实现类为XMLLanguageDriver。Mybatis默认是XML语言，所以我们来看看XMLLanguageDriver的实现：

```
public class XMLLanguageDriver implements LanguageDriver {
    // 创建参数处理器，返回默认的实现
    @Override
    public ParameterHandler createParameterHandler(MappedStatement mappedStatement,
Object parameterObject, BoundSql boundSql) {
        return new DefaultParameterHandler(mappedStatement, parameterObject, boundSql);
    }

    // 根据XML定义创建SqlSource
    @Override
    public SqlSource createSqlSource(Configuration configuration, XNode script, Class<?>
parameterType) {
        XMLScriptBuilder builder = new XMLScriptBuilder(configuration, script,
parameterType);
        return builder.parseScriptNode();
    }

    // 解析注解中的SQL语句
    @Override
    public SqlSource createSqlSource(Configuration configuration, String script, Class<?>
parameterType) {
        // issue #3
        if (script.startsWith("<script>")) {
            XPathParser parser = new XPathParser(script, false, configuration.getVariables(),
new XMLMapperEntityResolver());
            return createSqlSource(configuration, parser.evalNode("/script"), parameterType);
        } else {
            // issue #127
            script = PropertyParser.parse(script, configuration.getVariables());
            TextSqlNode textSqlNode = new TextSqlNode(script);
            if (textSqlNode.isDynamic()) {
                return new DynamicSqlSource(configuration, textSqlNode);
            } else {
                return new RawSqlSource(configuration, script, parameterType);
            }
        }
    }
}
```

```
}
```

如上所示，LanguageDriver将实际的实现根据采用的底层不同，委托给了具体的Builder，对于XML配置，委托给XMLScriptBuilder。对于使用Velocity模板的解析器，委托给SQLScriptSource解析具体的SQL。

注：mybatis-velocity还提供了VelocityLanguageDriver和FreeMarkerLanguageDriver，可参见：

- <https://github.com/mybatis/velocity-scripting>
- <https://github.com/mybatis/freemarker-scripting>

3.14 ResultMap

ResultMap类维护了每个标签中的详细信息，比如id映射、构造器映射、属性映射以及完整的映射列表、是否有嵌套的resultMap、是否有鉴别器、是否有嵌套查询，如下所示：

```
public class ResultMap {
    private Configuration configuration;

    private String id;
    private Class<?> type;
    private List<ResultMapping> resultMappings;
    private List<ResultMapping> idResultMappings;
    private List<ResultMapping> constructorResultMappings;
    private List<ResultMapping> propertyResultMappings;
    private Set<String> mappedColumns;
    private Set<String> mappedProperties;
    private Discriminator discriminator;
    private boolean hasNestedResultMaps;
    private boolean hasNestedQueries;
    private Boolean autoMapping;
    ...
}
```

ResultMap除了作为一个ResultMap的数据结构表示外，本身并没有提供额外的功能。

3.15 ResultMapping

ResultMapping代表下的映射，如下：

```
public class ResultMapping {

    private Configuration configuration;
    private String property;
    private String column;
    private Class<?> javaType;
    private JdbcType jdbcType;
    private TypeHandler<?> typeHandler;
    private String nestedResultMapId;
    private String nestedQueryId;
```

```

private Set<String> notNullColumns;
private String columnPrefix;
// 标记是否构造器属性, 是否ID属性
private List<ResultFlag> flags;
private List<ResultMapping> composites;
private String resultSet;
private String foreignColumn;
private boolean lazy;
...
}

```

3.16 Discriminator

每个鉴别器节点都表示为一个Discriminator，如下所示：

```

public class Discriminator {
    // 所属的属性节点<result>
    private ResultMapping resultMapping;
    // 内部的if then映射
    private Map<String, String> discriminatorMap;
    ...
}

```

3.17 Configuration

Configuration是mybatis所有配置以及mapper文件的元数据容器。无论是解析mapper文件还是运行时执行SQL语句，都需要依赖与mybatis的环境和配置信息，比如databaseId、类型别名等。mybatis实现将所有这些信息封装到Configuration中并提供了一系列便利的接口方便各主要的调用方使用，这样就避免了各种配置和元数据信息到处散落的凌乱。

3.18 ErrorContext

ErrorContext定义了一个mybatis内部统一的日志规范，记录了错误信息、发生错误涉及的资源文件、对象、逻辑过程、SQL语句以及出错原因，但是它不会影响运行，如下所示：

```

public class ErrorContext {

    private static final String LINE_SEPARATOR =
System.getProperty("line.separator", "\n");
    private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>
();

    private ErrorContext stored;
    private String resource;
    private String activity;
    private String object;
    private String message;
    private String sql;
    private Throwable cause;
}

```

...

```
public ErrorContext reset() {
    resource = null;
    activity = null;
    object = null;
    message = null;
    sql = null;
    cause = null;
    LOCAL.remove();
    return this;
}

@Override
public String toString() {
    StringBuilder description = new StringBuilder();

    // message
    if (this.message != null) {
        description.append(LINE_SEPARATOR);
        description.append("### ");
        description.append(this.message);
    }

    // resource
    if (resource != null) {
        description.append(LINE_SEPARATOR);
        description.append("### The error may exist in ");
        description.append(resource);
    }

    // object
    if (object != null) {
        description.append(LINE_SEPARATOR);
        description.append("### The error may involve ");
        description.append(object);
    }

    // activity
    if (activity != null) {
        description.append(LINE_SEPARATOR);
        description.append("### The error occurred while ");
        description.append(activity);
    }

    // activity
    if (sql != null) {
        description.append(LINE_SEPARATOR);
        description.append("### SQL: ");
    }
}
```



```

        description.append(sql.replace('\n', ' ').replace('\r', ' ').replace('\t', '
').trim());
    }

    // cause
    if (cause != null) {
        description.append(LINE_SEPARATOR);
        description.append("### Cause: ");
        description.append(cause.toString());
    }

    return description.toString();
}

}

```

3.19 BoundSql

```

/**
 * An actual SQL String got from an {@link SqlSource} after having processed any
 * dynamic content.
 * The SQL may have SQL placeholders "?" and an list (ordered) of an parameter mappings
 * with the additional information for each parameter (at least the property name of
 * the input object to read
 * the value from).
 * <br>
 * Can also have additional parameters that are created by the dynamic language (for
 * loops, bind...).
 *
 * SqlSource中包含的SQL处理动态内容之后的实际SQL语句，SQL中会包含?占位符，也就是最终给JDBC的SQL语
 * 句，以及他们的参数信息
 * @author Clinton Begin
 */
public class BoundSql {
    // sql文本
    private final String sql;
    // 静态参数说明
    private final List<ParameterMapping> parameterMappings;
    // 运行时参数对象
    private final Object parameterObject;
    // 额外参数，也就是for loops、bind生成的
    private final Map<String, Object> additionalParameters;
    // 额外参数的facade模式包装
    private final MetaObject metaParameters;

    public BoundSql(Configuration configuration, String sql, List<ParameterMapping>
parameterMappings, Object parameterObject) {
        this.sql = sql;
        this.parameterMappings = parameterMappings;
    }
}

```

```

    this.parameterObject = parameterObject;
    this.additionalParameters = new HashMap<String, Object>();
    this.metaParameters = configuration.newMetaObject(additionalParameters);
}

public String getSql() {
    return sql;
}

public List<ParameterMapping> getParameterMappings() {
    return parameterMappings;
}

public Object getParameterObject() {
    return parameterObject;
}

public boolean hasAdditionalParameter(String name) {
    String paramName = new PropertyTokenizer(name).getName();
    return additionalParameters.containsKey(paramName);
}

public void setAdditionalParameter(String name, Object value) {
    metaParameters.setValue(name, value);
}

public Object getAdditionalParameter(String name) {
    return metaParameters.getValue(name);
}
}

```

4 SQL语句的执行流程

4.1 传统JDBC用法

在原生jdbc中，我们要执行一个sql语句，它的流程是这样的：

1. 注册驱动；
2. 获取jdbc连接；
3. 创建参数化预编译SQL；
4. 绑定参数；
5. 发送SQL给数据库进行执行；
6. 对于查询，获取结果集到应用；

我们先回顾下典型JDBC的用法：

```

package org.mybatis.internal.example;

import java.sql.Connection;

```

```

import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class JdbcHelloWord {

    /**
     * 入口函数
     * @param arg
     */
    public static void main(String arg[]) {
        try {
            Connection con = null; //定义一个MYSQL链接对象
            Class.forName("com.mysql.jdbc.Driver").newInstance(); //MYSQL驱动
            con = DriverManager.getConnection("jdbc:mysql://10.7.12.4:3306/lfBase?
useUnicode=true", "lfBase", "eKffQV6wbh3sfQuFIG6M"); //链接本地MYSQL

            //更新一条数据
            String updateSql = "UPDATE LfParty SET remark1 = 'mybatis internal example'
WHERE lfPartyId = ?";
            PreparedStatement pstmt = con.prepareStatement(updateSql);
            pstmt.setString(1, "1");
            long updateRes = pstmt.executeUpdate();
            System.out.print("UPDATE:" + updateRes);

            //查询数据并输出
            String sql = "select lfPartyId,partyName from LfParty where lfPartyId = ?";
            PreparedStatement pstmt2 = con.prepareStatement(sql);
            pstmt2.setString(1, "1");
            ResultSet rs = pstmt2.executeQuery();
            while (rs.next()) { //循环输出结果集
                String lfPartyId = rs.getString("lfPartyId");
                String partyName = rs.getString("partyName");
                System.out.print("\r\n\r\n");
                System.out.print("lfPartyId:" + lfPartyId + "partyName:" + partyName);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

4.2 mybatis执行SQL语句

同样的，在mybatis中，要执行sql语句，首先要拿到代表JDBC底层连接的一个对象，这在mybatis中的实现就是SqlSession。mybatis提供了下列实现：

image

获取SqlSession的API如下：

```
SqlSession session = SqlSessionFactory.openSession();
try {
    User user = (User)
session.selectOne("org.mybatis.internal.example.mapper.UserMapper.getUser", 1);
    System.out.println("sql from xml:" + user.getLfPartyId() + "," +
user.getPartyName());
    UserMapper2 mapper = session.getMapper(UserMapper2.class);
    List<User> users = mapper.getUser2(293);
    System.out.println("sql from mapper:" + users.get(0).getLfPartyId() + "," +
users.get(0).getPartyName());
} finally {
    session.close();
}
```

同样，首先调用SqlSessionFactory.openSession()拿到一个session，然后在session上执行各种CRUD操作。简单来说，SqlSession就是jdbc连接的代表，openSession()就是获取jdbc连接（当然其背后可能是从jdbc连接池获取）；session中的各种selectXXX方法或者调用mapper的具体方法就是集合了JDBC调用的第3、4、5、6步。SqlSession接口的定义如下：

image

可知，绝大部分的方法都是泛型方法，也可以说采用了模板方法实现。

4.2.1 获取openSession

获取openSession的总体流程为：

image

我们先来看openSession的具体实现。mybatis提供了两个SqlSessionFactory实现：SqlSessionManager和DefaultSqlSessionFactory，默认返回的是DefaultSqlSessionFactory，它们的区别我们后面会讲到。我们先看下SqlSessionFactory的接口定义：

```
public interface SqlSessionFactory {

    SqlSession openSession();

    SqlSession openSession(boolean autoCommit);
    SqlSession openSession(Connection connection);
    SqlSession openSession(TransactionIsolationLevel level);

    SqlSession openSession(ExecutorType execType);
    SqlSession openSession(ExecutorType execType, boolean autoCommit);
    SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level);
    SqlSession openSession(ExecutorType execType, Connection connection);
```

```
Configuration getConfiguration();
```

}
主要有多种形式的重载，除了使用默认设置外，可以指定自动提交模式、特定的jdbc连接、事务隔离级别，以及指定的执行器类型。关于执行器类型，mybatis提供了三种执行器类型：SIMPLE，REUSE，BATCH。后面我们会详细分析每种类型的执行器的差别以及各自的适用场景。我们以最简单的无参方法切入（按照一般的套路，如果定义了多个重载的方法或者构造器，内部实现一定是设置作者认为最合适的默认值，然后调用次多参数的方法，直到最后），它的实现是这样的：

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    ...

    @Override
    public SqlSession openSession() {
        // 使用默认的执行器类型(默认是SIMPLE)，默认隔离级别，非自动提交 委托给
        openSessionFromDataSource方法
        return openSessionFromDataSource(configuration.getDefaultExecutorType(), null,
false);
    }

    ...

    private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
        Transaction tx = null;
        try {
            final Environment environment = configuration.getEnvironment();
            // 获取事务管理器，支持从数据源或者直接获取
            final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);

            // 从数据源创建一个事务，同样,数据源必须配置，mybatis内置了JNDI、POOLED、UNPOOLED三种类型的
            数据源,其中POOLED对应的实现为org.apache.ibatis.datasource.pooled.PooledDataSource,它是
            mybatis自带实现的一个同步、线程安全的数据库连接池 一般在生产中,我们会使用dbcp或者druid连接池
            tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
            final Executor executor = configuration.newExecutor(tx, execType);
            return new DefaultSqlSession(configuration, executor, autoCommit);
        } catch (Exception e) {
            closeTransaction(tx); // may have fetched a connection so lets call close()
            throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
        } finally {
            ErrorContext.instance().reset();
        }
    }

    ...

    private TransactionFactory getTransactionFactoryFromEnvironment(Environment
environment) {
```

```

// 如果没有配置environment或者environment的事务管理器为空,则使用受管的事务管理器
// 除非什么都没有配置,否则在mybatis-config里面,至少要配置一个environment,此时事务工厂不允许为空
// 对于jdbc类型的事务管理器,则返回JdbcTransactionFactory,其内部操作mybatis的JdbcTransaction实现(采用了Facade模式),后者对jdbc连接操作
if (environment == null || environment.getTransactionFactory() == null) {
    return new ManagedTransactionFactory();
}
return environment.getTransactionFactory();
}
}

```

我们来看下transactionFactory.newTransaction的实现，还是以jdbc事务为例子。

```

public class JdbcTransactionFactory implements TransactionFactory {
    ...

    @Override
    public Transaction newTransaction(DataSource ds, TransactionIsolationLevel level,
    boolean autoCommit) {
        return new JdbcTransaction(ds, level, autoCommit);
    }
}

```

newTransaction的实现逻辑很简单，但是此时返回的事务不一定是底层连接的。

拿到事务后，根据事务和执行器类型创建一个真正的执行器实例。获取执行器的逻辑如下：

```

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    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);
    }
    if (cacheEnabled) {
        executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}

```

如果没有配置执行器类型，默认是简单执行器。如果启用了缓存，则使用缓存执行器。

拿到执行器之后，new一个DefaultSqlSession并返回，这样一个SqlSession就创建了，它从逻辑上代表一个封装了事务特性的连接，如果在此期间发生异常，则调用关闭事务（因为此时事务底层的连接可能已经持有了，否则会导致连接泄露）。

DefaultSqlSession的构造很简单，就是简单的属性赋值：

```
public class DefaultSqlSession implements SqlSession {

    private Configuration configuration;
    private Executor executor;

    private boolean autoCommit;
    // 含义是TODO
    private boolean dirty;
    private List<Cursor<?>> cursorList;

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

具体的对外API我们后面专门讲解。

根据sql语句使用xml进行维护或者在注解上配置,sql语句执行的入口分为两种：

第一种，调用org.apache.ibatis.session.SqlSession的crud方法比如selectList/selectOne传递完整的语句id直接执行；

第二种，先调用SqlSession的getMapper()方法得到mapper接口的一个实现，然后调用具体的方法。除非早期，现在实际开发中，我们一般采用这种方式。

4.2.2 sql语句执行方式一

我们先来看第一种形式的sql语句执行也就是SqlSession.getMapper除外的形式。这里还是以带参数的session.selectOne为例子。其实现代码为：

```
@Override
public <T> T selectOne(String statement, Object parameter) {
    // Popular vote was to return null on 0 results and throw exception on too many.
    List<T> list = this.<T>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;
    }
}
...
@Override
public <E> List<E> selectList(String statement, Object parameter) {
    return this.selectList(statement, parameter, RowBounds.DEFAULT);
}

@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
    try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        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();
    }
}
}

```

selectOne调用在内部将具体实现委托给selectList，如果返回的行数大于1就抛异常。我们先看下selectList的总体流程：

image

selectList的第三个参数是RowBounds.DEFAULT，我们来看下什么是RowBounds。

```

public class RowBounds {

    public static final int NO_ROW_OFFSET = 0;
    public static final int NO_ROW_LIMIT = Integer.MAX_VALUE;
    public static final RowBounds DEFAULT = new RowBounds();

    private int offset;
    private int limit;

    public RowBounds() {
        this.offset = NO_ROW_OFFSET;
        this.limit = NO_ROW_LIMIT;
    }
}

```

从定义可知，RowBounds是一个分页查询的参数封装，默认是不分页。

第三个selectList重载首先使用应用调用方传递的语句id判断configuration.mappedStatements里面是否有这个语句，如果没有将会抛出IllegalArgumentException异常，执行结束。否则将获取到的映射语句对象连同其他参数一起将具体实现委托给执行器Executor的query方法。

在这里对查询参数parameter进行了一次封装，封装逻辑wrapCollection主要是判断参数是否为数组或集合类型，是的话将他们包装到StrictMap中。同时设置结果处理器为null。

我们现在来看下Executor的query方法：

```
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler) throws SQLException {
    // 首先根据传递的参数获取BoundSql对象，对于不同类型的SqlSource，对应的getBoundSql实现不同，具体参见SqlSource详解一节 TODO
    BoundSql boundSql = ms.getBoundSql(parameter);
    // 创建缓存key
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);

    // 委托给重载的query
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}

@SuppressWarnings("unchecked")
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }

    // 如果需要刷新缓存(默认DML需要刷新,也可以语句层面修改), 且queryStack(应该是用于嵌套查询的场景)=0
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
        clearLocalCache();
    }
    List<E> list;
    try {
        queryStack++;
        // 如果查询不需要应用结果处理器,则先从缓存获取,这样可以避免数据库查询。我们后面会分析到localCache是什么时候被设置进去的
        list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
        if (list != null) {
            handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
        } else {
            // 不管是因为需要应用结果处理器还是缓存中没有,都从数据库中查询
            list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
        }
    } finally {
        queryStack--;
    }
}
```

```

    if (queryStack == 0) {
        for (DeferredLoad deferredLoad : deferredLoads) {
            deferredLoad.load();
        }
        // issue #601
        deferredLoads.clear();
        if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
            // issue #482
            clearLocalCache();
        }
    }
    return list;
}
...
@Override
public CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, BoundSql boundSql) {
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    // 根据映射语句id,分页信息,jdbc规范化的预编译sql,所有映射参数的值以及环境id的值,计算出缓存Key
    CacheKey cacheKey = new CacheKey();
    cacheKey.update(ms.getId());
    cacheKey.update(rowBounds.getOffset());
    cacheKey.update(rowBounds.getLimit());
    cacheKey.update(boundSql.getSql());
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    TypeHandlerRegistry typeHandlerRegistry =
ms.getConfiguration().getTypeHandlerRegistry();
    // mimic DefaultParameterHandler logic
    for (ParameterMapping parameterMapping : parameterMappings) {
        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);
            }
            cacheKey.update(value);
        }
    }
    if (configuration.getEnvironment() != null) {
        // issue #176

```

```

        cacheKey.update(configuration.getEnvironment().getId());
    }
    return cacheKey;
}

private void handleLocallyCachedOutputParameters(MappedStatement ms, CacheKey key,
Object parameter, BoundSql boundSql) {
    // 只处理存储过程和函数调用的出参，因为存储过程和函数的返回不是通过ResultMap而是ParameterMap来的
    // 的，所以只要把缓存的非IN模式参数取出来设置到parameter对应的属性上即可
    if (ms.getStatementType() == StatementType.CALLABLE) {
        final Object cachedParameter = localOutputParameterCache.getObject(key);
        if (cachedParameter != null && parameter != null) {
            final MetaObject metaCachedParameter =
configuration.newMetaObject(cachedParameter);
            final MetaObject metaParameter = configuration.newMetaObject(parameter);
            for (ParameterMapping parameterMapping : boundSql.getParameterMappings()) {
                if (parameterMapping.getMode() != ParameterMode.IN) {
                    final String parameterName = parameterMapping.getProperty();
                    final Object cachedValue = metaCachedParameter.getValue(parameterName);
                    metaParameter.setValue(parameterName, cachedValue);
                }
            }
        }
    }
}

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
SQLException {
    List<E> list;
    // 一开始放个占位符进去,这个还真不知道用意是什么???
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
        // doQuery是个抽象方法,每个具体的执行器都要自己去实现,我们先看SIMPLE的
        list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
    } finally {
        localCache.removeObject(key);
    }
    // 把真正的查询结果放到缓存中去
    localCache.putObject(key, list);

    // 如果是存储过程类型,则把查询参数放到本地出参缓存中, 所以第一次一定为空
    if (ms.getStatementType() == StatementType.CALLABLE) {
        localOutputParameterCache.putObject(key, parameter);
    }
    return list;
}

```

我们先看下缓存key的定义：

```
package org.apache.ibatis.cache;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.apache.ibatis.reflection.ArrayUtil;

/**
 * @author Clinton Begin
 */
public class CacheKey implements Cloneable, Serializable {

    private static final long serialVersionUID = 1146682552656046210L;

    public static final CacheKey NULL_CACHE_KEY = new NullCacheKey();

    private static final int DEFAULT_MULTIPLYER = 37;
    private static final int DEFAULT_HASHCODE = 17;

    private int multiplier;
    private int hashCode;
    private long checksum;
    private int count;
    private List<Object> updateList;

    public CacheKey() {
        this.hashCode = DEFAULT_HASHCODE;
        this.multiplier = DEFAULT_MULTIPLYER;
        this.count = 0;
        this.updateList = new ArrayList<Object>();
    }

    public CacheKey(Object[] objects) {
        this();
        updateAll(objects);
    }

    public int getUpdateCount() {
        return updateList.size();
    }

    public void update(Object object) {
        // ArrayUtil提供了可以计算包括数组的对象的hashCode, toString, equals方法
        int baseHashCode = object == null ? 1 : ArrayUtil.hashCode(object);

        count++;
        checksum += baseHashCode;
    }
}
```

```

        baseHashCode *= count;

        hashCode = multiplier * hashCode + baseHashCode;

        updateList.add(object);
    }

    public void updateAll(Object[] objects) {
        for (Object o : objects) {
            update(o);
        }
    }

    @Override
    public boolean equals(Object object) {
        if (this == object) {
            return true;
        }
        if (!(object instanceof CacheKey)) {
            return false;
        }

        final CacheKey cacheKey = (CacheKey) object;

        if (hashCode != cacheKey.hashCode) {
            return false;
        }
        if (checksum != cacheKey.checksum) {
            return false;
        }
        if (count != cacheKey.count) {
            return false;
        }

        for (int i = 0; i < updateList.size(); i++) {
            Object thisObject = updateList.get(i);
            Object thatObject = cacheKey.updateList.get(i);
            if (!ArrayUtil.equals(thisObject, thatObject)) {
                return false;
            }
        }
        return true;
    }
    ...

    @Override
    public CacheKey clone() throws CloneNotSupportedException {
        CacheKey clonedCacheKey = (CacheKey) super.clone();
        clonedCacheKey.updateList = new ArrayList<Object>(updateList);
    }

```

```

        return clonedCacheKey;
    }

}

```

我们来看下SIMPLE执行器的doQuery定义：

```

@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();
        // 根据上下文参数和具体的执行器new一个StatementHandler，其中包含了所有必要的信息，比如结果处理器、参数处理器、执行器等等，主要有三种类型的语句处理器UNPREPARE、PREPARE、CALLABLE。默认是PREPARE类型，通过mapper语句上的statementType属性进行设置，一般除了存储过程外不应该设置
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        // 这一步是真正和JDBC打交道
        stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.<E>query(stmt, resultHandler);
    } finally {
        closeStatement(stmt);
    }
}

private Statement prepareStatement(StatementHandler handler, Log statementLog) throws
SQLException {
    Statement stmt;
    // 获取JDBC连接
    Connection connection = getConnection(statementLog);
    // 调用语句处理器的prepare方法
    stmt = handler.prepare(connection, transaction.getTimeout());
    // 设置参数
    handler.parameterize(stmt);
    return stmt;
}

```

下面我们来看下PREPARE处理处理的prepare实现，从上述可知，具体的处理器继承了BaseStatementHandler，

```

@Override
public Statement prepare(Connection connection, Integer transactionTimeout) throws
SQLException {
    ErrorContext.instance().sql(boundSql.getSql());
    Statement statement = null;
    try {
        // 首先实例化语句，因为PREPARE和非PREPARE不同，所以留给具体子类实现
        statement = instantiateStatement(connection);
    }
}

```

```

// 设置语句超时时间
setStatementTimeout(statement, transactionTimeout);

// 设置fetch大小
setFetchSize(statement);
return statement;
} catch (SQLException e) {
    closeStatement(statement);
    throw e;
} catch (Exception e) {
    closeStatement(statement);
    throw new ExecutorException("Error preparing statement. Cause: " + e, e);
}
}

```

重点来看PREPARE语句处理器的初始化语句过程：

```

@Override
protected Statement instantiateStatement(Connection connection) throws SQLException {
    String sql = boundSql.getSql();
    // 只处理Jdbc3KeyGenerator，因为它代表的是自增，另外一个SelectKeyGenerator用于不支持自增的情况
    if (mappedStatement.getKeyGenerator() instanceof Jdbc3KeyGenerator) {
        String[] keyColumnNames = mappedStatement.getKeyColumns();
        if (keyColumnNames == null) {
            return connection.prepareStatement(sql,
                PreparedStatement.RETURN_GENERATED_KEYS);
        } else {
            return connection.prepareStatement(sql, keyColumnNames);
        }
    } else if (mappedStatement.getResultSetType() != null) {
        return connection.prepareStatement(sql,
            mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR_READ_ONLY);
    } else {
        return connection.prepareStatement(sql);
    }
}

```

基本上就是把我们在MAPPER中定义的属性转换为JDBC标准的调用。

接下去再来看参数是如何MAPPER中定义参数是如何转换为JDBC参数的，PreparedStatementHandler.parameterize将具体实现委托给了ParameterHandler.setParameters()方法，ParameterHandler目前只有一种实现DefaultParameterHandler。

```

@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);
            // 仅处理非出参
            if (parameterMapping.getMode() != ParameterMode.OUT) {
                Object value;
                String propertyName = parameterMapping.getProperty();

                // 计算参数值的优先级是 先判断是不是属于语句的AdditionalParameter; 其次参数是不是null;
                // 然后判断是不是属于注册类型; 都不是, 那估计参数一定是object或者map了, 这就要借助于MetaObject获取属性值
                // 了;
                if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first
                    for additional params
                        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 {
                    // jdbc下标从1开始, 由具体的类型处理器进行参数的设置, 对于每个jdbcType, mybatis都提供
                    // 了一个对应的Handler, 具体可参考上文TypeHandler详解, 其内部调用的是PreparedStatement.setXXX进行设
                    // 置。
                    typeHandler.setParameter(ps, i + 1, value, jdbcType);
                } catch (TypeException e) {
                    throw new TypeException("Could not set parameters for mapping: " +
parameterMapping + ". Cause: " + e, e);
                } catch (SQLException e) {
                    throw new TypeException("Could not set parameters for mapping: " +
parameterMapping + ". Cause: " + e, e);
                }
            }
        }
    }
}

```

我们在mapper中定义的所有ParameterType、ParameterMap、内嵌参数映射等在最后都在这里被作为ParameterMapping转换为JDBC参数。

Configuration中新StatementHandler的定义如下:

```
public StatementHandler newStatementHandler(Executor executor, MappedStatement
mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler
resultHandler, BoundSql boundSql) {
    StatementHandler statementHandler = new RoutingStatementHandler(executor,
mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);
    // 如果有拦截器的话, 则为语句处理器新生成一个代理类
    statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
    return statementHandler;
}
```

回到主体逻辑SimpleExecutor.doQuery, 创建了Statement具体实现的实例后, 调用SimpleExecutor.query进行具体的查询, 查询的主体逻辑如下:

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

从上述具体逻辑的实现可以看出, 内部调用PreparedStatement完成具体查询后, 将ps的结果集传递给对应的结果处理器进行处理。查询结果的映射是mybatis作为ORM框架提供的最有价值的功能, 同时也可以说是最复杂的逻辑之一。下面我们来专门分析mybatis查询结果集的处理。

mybatis结果集处理

对于结果集处理, mybatis默认提供了DefaultResultSetHandler, 如下所示:

```
@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;
    // 返回jdbc ResultSet的包装形式, 主要是将java.sql.ResultSetMetaData做了Facade模式, 便于使用
    ResultSetWrapper rsw = getFirstResultSet(stmt);

    List<ResultMap> resultMaps = mappedStatement.getResultMaps();
    // 绝大部分情况下一个查询只有一个ResultMap, 除非多结果集查询
    int resultMapCount = resultMaps.size();
    validateResultMapsCount(rsw, resultMapCount);

    // 至少执行一次
    while (rsw != null && resultMapCount > resultSetCount) {
```

```

    ResultMap resultMap = resultMaps.get(resultSetCount);
    // 根据resultMap的定义将resultSet打包到应用端的multipleResults中
    handleResultSet(rsw, resultMap, multipleResults, null);
    // 循环直到处理完所有的结果集, 一般情况下, 一个execute只会返回一个结果集, 除非语句比如存储过程返回多个resultSet
    rsw = getNextResultSet(stmt);
    // 清空嵌套结果集
    cleanUpAfterHandlingResultSet();
    resultSetCount++;
}

// 处理关联或集合
String[] resultSets = mappedStatement.getResultSets();
if (resultSets != null) {
    // 如果一个映射语句的resultSet数量比jdbc resultSet多的话, 多的部分就是嵌套结果集
    while (rsw != null && resultSetCount < resultSets.length) {
        // nextResultMaps初始化的时候为空, 它是在处理主查询每行记录的时候写进去的, 所以此时就可以得到主记录是哪个属性
        ResultMapping parentMapping = nextResultMaps.get(resultSets[resultSetCount]);
        if (parentMapping != null) {
            String nestedResultMapId = parentMapping.getNestedResultMapId();
            ResultMap resultMap = configuration.getResultMap(nestedResultMapId);
            // 得到子结果集的resultMap之后, 就可以进行填充了
            handleResultSet(rsw, resultMap, null, parentMapping);
        }
        rsw = getNextResultSet(stmt);
        cleanUpAfterHandlingResultSet();
        resultSetCount++;
    }
}

return collapseSingleResultList(multipleResults);
}

```

// 因为调用handleResultSet的只有handleResultSets, 按说其中第一个调用永远不会出现parentMapping==null的情况, 只有第二个调用才会出现这种情况, 而且应该是连续的, 因为第二个调用就是为了处理嵌套resultMap。所以在handleRowValues处理resultMap的时候, 一定是主的先处理, 嵌套的后处理, 这样整个逻辑就比较清晰了。这里需要补个流程图。

```

private void handleResultSet(ResultSetWrapper rsw, ResultMap resultMap, List<Object>
multipleResults, ResultMapping parentMapping) throws SQLException {
    try {
        // 处理嵌套/关联的resultMap(collection或association)
        if (parentMapping != null) {
            // 处理非主记录 resultHandler传递null, RowBounds传递默认值, parentMapping不为空
            handleRowValues(rsw, resultMap, null, RowBounds.DEFAULT, parentMapping);
        } else {
            // 处理主记录, 这里有个疑问??? 问什么resultHandler不为空, 就不需要添加到multipleResults
            if (resultHandler == null) {

```

中

```

        DefaultResultHandler defaultResultHandler = new
DefaultResultHandler(objectFactory);
        // 处理主记录, resultHandler不为空,rowBounds不使用默认值,parentMapping传递null
        handleRowValues(rsw, resultMap, defaultResultHandler, rowBounds, null);
        multipleResults.add(defaultResultHandler.getResultList());
    } else {
        handleRowValues(rsw, resultMap, resultHandler, rowBounds, null);
    }
}
} finally {
    // issue #228 (close resultsets)
    closeResultSet(rsw.getResultSet());
}
}
// 处理每一行记录,不管是主查询记录还是关联嵌套的查询结果集, 他们的入参上通过
resultHandler,rowBounds,parentMapping区分, 具体见上述调用
public void handleRowValues(ResultSetWrapper rsw, ResultMap resultMap,
ResultHandler<?> resultHandler, RowBounds rowBounds, ResultMapping parentMapping)
throws SQLException {
    // 具体实现上,处理嵌套记录和主记录的逻辑不一样
    if (resultMap.hasNestedResultMaps()) {
        ensureNoRowBounds();
        checkResultHandler();
        // 含有嵌套resultMap的列处理, 通常是collection或者association
        handleRowValuesForNestedResultMap(rsw, resultMap, resultHandler, rowBounds,
parentMapping);
    } else {
        handleRowValuesForSimpleResultMap(rsw, resultMap, resultHandler, rowBounds,
parentMapping);
    }
}

// 处理主记录
private void handleRowValuesForSimpleResultMap(ResultSetWrapper rsw, ResultMap
resultMap, ResultHandler<?> resultHandler, RowBounds rowBounds, ResultMapping
parentMapping)
    throws SQLException {
    DefaultResultContext<Object> resultContext = new DefaultResultContext<Object>();
    // 设置从指定行开始,这是mybatis进行内部分页,不是依赖服务器端的分页实现
    skipRows(rsw.getResultSet(), rowBounds);
    // 确保没有超过mybatis逻辑分页限制,同时结果集中还有记录没有fetch
    while (shouldProcessMoreRows(resultContext, rowBounds) &&
rsw.getResultSet().next()) {
        // 解析鉴别器, 得到嵌套的最深的鉴别器对应的ResultMap, 如果没有鉴别器, 就返回最顶层的ResultMap
        ResultMap discriminatedResultMap =
resolveDiscriminatedResultMap(rsw.getResultSet(), resultMap, null);
        // 这个时候resultMap是非常干净的,没有嵌套任何其他东西了, 但是这也是最关键的地方, 将ResultSet
记录转换为业务层配置的对象类型或者Map类型
        Object rowValue = getRowValue(rsw, discriminatedResultMap);

```

```

        storeObject(resultHandler, resultContext, rowValue, parentMapping,
rsw.getResultSet());
    }
}

private void storeObject(ResultHandler<?> resultHandler, DefaultResultContext<Object>
resultContext, Object rowValue, ResultMapping parentMapping, ResultSet rs) throws
SQLException {
    if (parentMapping != null) {
        // 如果不是主记录,则链接到主记录
        linkToParents(rs, parentMapping, rowValue);
    } else {
        // 否则让ResultHandler(默认是DefaultResultHandler处理,直接添加到List<Object>中)
        callResultHandler(resultHandler, resultContext, rowValue);
    }
}

@SuppressWarnings("unchecked" /* because ResultHandler<?> is always
ResultHandler<Object> */)
private void callResultHandler(ResultHandler<?> resultHandler,
DefaultResultContext<Object> resultContext, Object rowValue) {
    resultContext.nextResultObject(rowValue);
    ((ResultHandler<Object>) resultHandler).handleResult(resultContext);
}

// 非主记录需要链接到主记录,这里涉及到collection和association的resultMap实现,我们来看下
// MULTIPLE RESULT SETS

private void linkToParents(ResultSet rs, ResultMapping parentMapping, Object
rowValue) throws SQLException {
    CacheKey parentKey = createKeyForMultipleResults(rs, parentMapping,
parentMapping.getColumn(), parentMapping.getForeignColumn());
    // 判断当前的主记录是否有待关联的子记录,是通过pendingRelations Map进行维护的,
pendingRelations是在addPendingChildRelation中添加主记录的
    List<PendingRelation> parents = pendingRelations.get(parentKey);
    if (parents != null) {
        for (PendingRelation parent : parents) {
            if (parent != null && rowValue != null) {
                linkObjects(parent.metaObject, parent.propertyMapping, rowValue);
            }
        }
    }
}

// 集合与非集合的处理逻辑对外封装在一起,这样便于用户使用,内部通过判断resultMapping中的类型确定是
否为集合类型。无论是否为集合类型,最后都添加到parent的metaObject所封装的原始Object对应的属性上
private void linkObjects(MetaObject metaObject, ResultMapping resultMapping, Object
rowValue) {

```

```

        final Object collectionProperty =
            instantiateCollectionPropertyIfAppropriate(resultMapping, metaObject);
        if (collectionProperty != null) {
            final MetaObject targetMetaObject =
                configuration.newMetaObject(collectionProperty);
            targetMetaObject.add(rowValue);
        } else {
            metaObject.setValue(resultMapping.getProperty(), rowValue);
        }
    }

    private Object instantiateCollectionPropertyIfAppropriate(ResultMapping
resultMapping, MetaObject metaObject) {
        final String propertyName = resultMapping.getProperty();
        Object propertyValue = metaObject.getValue(propertyName);
        if (propertyValue == null) {
            Class<?> type = resultMapping.getJavaType();
            if (type == null) {
                type = metaObject.getSetterType(propertyName);
            }
            try {
                if (objectFactory.isCollection(type)) {
                    propertyValue = objectFactory.create(type);
                    metaObject.setValue(propertyName, propertyValue);
                    return propertyValue;
                }
            } catch (Exception e) {
                throw new ExecutorException("Error instantiating collection property for result
'" + resultMapping.getProperty() + "'. Cause: " + e, e);
            }
        } else if (objectFactory.isCollection(propertyValue.getClass())) {
            return propertyValue;
        }
        return null;
    }

    private Object getRowValue(ResultSetWrapper rsw, ResultMap resultMap) throws
SQLException {
        final ResultLoaderMap lazyLoader = new ResultLoaderMap();
        // 有三个createResultObject重载，这三个重载完成的功能从最里面到最外面分别是：
        1、使用构造器创建目标对象类型；
            先判断是否有目标对象类型的处理器，有的话，直接调用类型处理器（这里为什么一定是原生类型）创建目标对
象
            如果没有，判断是否有构造器，有的话，使用指定的构造器创建目标对象（构造器里面如果嵌套了查询或者
ResultMap，则进行处理）
            如果结果类型是接口或者具有默认构造器，则使用ObjectFactory创建默认目标对象

```

最后判断是否可以应用自动映射，默认是对非嵌套查询，只要没有明确设置AutoMappingBehavior.NONE就可以，对于嵌套查询，AutoMappingBehavior.FULL就可以。自动映射的逻辑是先找目标对象上具有@AutomapConstructor注解的构造器，然后根据ResultSet返回的字段清单找匹配的构造器，如果找不到，就报错

2、如果此时创建的对象不为空，且不需要应用结果对象处理器，判断有没有延迟加载且具有嵌套查询的属性，如果有的话，则为对象创建一个代理，额外存储后面fetch的时候进行延迟加载所需的信息。返回对象。

3、如果此时创建的对象不为空，且不需要应用结果对象处理器，如果对象需要自动映射，则先进行自动映射（创建自动映射列表的过程为：先找到在ResultSet、不在ResultMap中的列，如果在目标对象上可以找到属性且可以类型可以处理，则标记为可以自动映射；然后进行自动映射处理，如果遇到无法处理的属性，则根据autoMappingUnknownColumnBehavior进行处理，默认忽略），其次进行属性映射处理

```
Object rowValue = createResultObject(rsw, resultMap, lazyLoader, null);
if (rowValue != null && !hasTypeHandlerForResultObject(rsw, resultMap.getType())) {
    final MetaObject metaObject = configuration.newMetaObject(rowValue);
    boolean foundValues = this.useConstructorMappings;
    if (shouldApplyAutomaticMappings(resultMap, false)) {
        foundValues = applyAutomaticMappings(rsw, resultMap, metaObject, null) ||
foundValues;
    }

    // 处理属性映射,这里会识别出哪些属性需要nestQuery, 哪些是nest ResultMap
    foundValues = applyPropertyMappings(rsw, resultMap, metaObject, lazyLoader, null)
|| foundValues;
    foundValues = lazyLoader.size() > 0 || foundValues;
    rowValue = foundValues || configuration.isReturnInstanceForEmptyRow() ? rowValue
: null;
}
return rowValue;
}

//
// PROPERTY MAPPINGS
//

private boolean applyPropertyMappings(ResultSetWrapper rsw, ResultMap resultMap,
MetaObject metaObject, ResultLoaderMap lazyLoader, String columnPrefix)
throws SQLException {
    final List<String> mappedColumnNames = rsw.getMappedColumnNames(resultMap,
columnPrefix);
    boolean foundValues = false;
    final List<ResultMapping> propertyMappings = resultMap.getPropertyResultMappings();
    for (ResultMapping propertyMapping : propertyMappings) {
        String column = prependPrefix(propertyMapping.getColumn(), columnPrefix);
        if (propertyMapping.getNestedResultMapId() != null) {
            // the user added a column attribute to a nested result map, ignore it
            column = null;
        }
        if (propertyMapping.isCompositeResult()
            || (column != null &&
mappedColumnNames.contains(column.toUpperCase(Locale.ENGLISH))))
```

```

        || propertyMapping.getResultSet() != null) {
    Object value = getPropertyMappingValue(rsw.getResultSet(), metaObject,
propertyMapping, lazyLoader, columnPrefix);
    // issue #541 make property optional
    final String property = propertyMapping.getProperty();
    if (property == null) {
        continue;
    } else if (value == DEFERED) {
        foundValues = true;
        continue;
    }
    if (value != null) {
        foundValues = true;
    }
    if (value != null || (configuration.isCallSettersOnNulls() &&
!metaObject.getSetterType(property).isPrimitive())) {
        // gcode issue #377, call setter on nulls (value is not 'found')
        metaObject.setValue(property, value);
    }
}
}
return foundValues;
}

private Object getPropertyMappingValue(ResultSet rs, MetaObject metaResultObject,
ResultMapping propertyMapping, ResultLoaderMap lazyLoader, String columnPrefix)
    throws SQLException {
    if (propertyMapping.getNestedQueryId() != null) {
        // 处理嵌套query类型的属性
        return getNestedQueryMappingValue(rs, metaResultObject, propertyMapping,
lazyLoader, columnPrefix);
    } else if (propertyMapping.getResultSet() != null) {
        // 处理resultMap类型的属性,主要是: 1、维护记录cacheKey和父属性/记录对Map的关联关系, 便于在处
理嵌套ResultMap时很快可以找到所有需要处理嵌套结果集的父属性; 2、维护父属性和对应resultSet的关联关系。
这两者都是为了在处理嵌套结果集是方便
        addPendingChildRelation(rs, metaResultObject, propertyMapping); // TODO is that
OK?
        return DEFERED;
    } else {
        final TypeHandler<?> typeHandler = propertyMapping.getTypeHandler();
        final String column = prependPrefix(propertyMapping.getColumn(), columnPrefix);
        return typeHandler.getResult(rs, column);
    }
}
}

```

对于嵌套resultmap的处理, 它的实现是这样的:

```

//
// HANDLE NESTED RESULT MAPS

```



```
//

private void handleRowValuesForNestedResultMap(ResultSetWrapper rsw, ResultMap
resultMap, ResultHandler<?> resultHandler, RowBounds rowBounds, ResultMapping
parentMapping) throws SQLException {
    final DefaultResultContext<Object> resultContext = new DefaultResultContext<Object>
();
    // mybatis 分页处理
    skipRows(rsw.getResultSet(), rowBounds);
    // 前一次处理的记录,只有在映射语句的结果集无序的情况下有意义
    Object rowValue = previousRowValue;
    // 一直处理直到超出分页边界或者结果集处理完
    while (shouldProcessMoreRows(resultContext, rowBounds) &&
rsw.getResultSet().next()) {
        //同主记录,先解析到鉴别器的最底层的ResultMap
        final ResultMap discriminatedResultMap =
resolveDiscriminatedResultMap(rsw.getResultSet(), resultMap, null);
        // 创建当前处理记录的rowKey, 规则见下文
        final CacheKey rowKey = createRowKey(discriminatedResultMap, rsw, null);
        // 根据rowKey获取嵌套结果对象map中对应的值, 因为在处理主记录时存储进去了, 具体见上面
addPending的流程图, 所以partialObject一般不为空
        Object partialObject = nestedResultObjects.get(rowKey);
        // issue #577 && #542
        // 根据映射语句的结果集是否有序走不同的逻辑
        if (mappedStatement.isResultOrdered()) {
            // 对于有序结果集的映射语句, 如果嵌套结果对象map中不包含本记录, 则清空嵌套结果对象, 因为此时嵌
套结果对象之前的记录已经没有意义了
            if (partialObject == null && rowValue != null) {
                nestedResultObjects.clear();
                // 添加记录到resultHandler的list属性中 或 如果是非主记录,添加到主记录对应属性的list或者
object中
                storeObject(resultHandler, resultContext, rowValue, parentMapping,
rsw.getResultSet());
            }
            rowValue = getRowValue(rsw, discriminatedResultMap, rowKey, null,
partialObject);
        } else { // 正常逻辑走这里
            rowValue = getRowValue(rsw, discriminatedResultMap, rowKey, null,
partialObject);
            if (partialObject == null) {
                storeObject(resultHandler, resultContext, rowValue, parentMapping,
rsw.getResultSet());
            }
        }
    }

    if (rowValue != null && mappedStatement.isResultOrdered() &&
shouldProcessMoreRows(resultContext, rowBounds)) {

```



```

        storeObject(resultHandler, resultContext, rowValue, parentMapping,
rsw.getResultSet());
        previousRowValue = null;
    } else if (rowValue != null) {
        previousRowValue = rowValue;
    }
}

//
// GET VALUE FROM ROW FOR NESTED RESULT MAP
// 为嵌套resultMap创建rowValue, 和非嵌套记录的接口分开
// getRowValue和applyNestedResultMappings存在递归调用,直到处理到不包含任何嵌套结果集的最后一层
resultMap为止
private Object getRowValue(ResultSetWrapper rsw, ResultMap resultMap/*主记录的
resultMap*/, CacheKey combinedKey, String columnPrefix, Object partialObject/*嵌套
resultMap的记录*/) throws SQLException {
    final String resultMapId = resultMap.getId();
    Object rowValue = partialObject;
    if (rowValue != null) { // 此时rowValue不应该空
        final MetaObject metaObject = configuration.newMetaObject(rowValue);
        putAncestor(rowValue, resultMapId);
        applyNestedResultMappings(rsw, resultMap, metaObject, columnPrefix, combinedKey,
false);
        ancestorObjects.remove(resultMapId);
    } else {
        final ResultLoaderMap lazyLoader = new ResultLoaderMap();
        rowValue = createResultObject(rsw, resultMap, lazyLoader, columnPrefix);
        if (rowValue != null && !hasTypeHandlerForResultObject(rsw, resultMap.getType()))
{
            final MetaObject metaObject = configuration.newMetaObject(rowValue);
            // 判断是否至少找到了一个不为null的属性值
            boolean foundValues = this.useConstructorMappings;
            if (shouldApplyAutomaticMappings(resultMap, true)) {
                foundValues = applyAutomaticMappings(rsw, resultMap, metaObject,
columnPrefix) || foundValues;
            }
            foundValues = applyPropertyMappings(rsw, resultMap, metaObject, lazyLoader,
columnPrefix) || foundValues;
            putAncestor(rowValue, resultMapId);
            foundValues = applyNestedResultMappings(rsw, resultMap, metaObject,
columnPrefix, combinedKey, true) || foundValues;
            ancestorObjects.remove(resultMapId);
            foundValues = lazyLoader.size() > 0 || foundValues;
            rowValue = foundValues || configuration.isReturnInstanceForEmptyRow() ?
rowValue : null;
        }
        if (combinedKey != CacheKey.NULL_CACHE_KEY) {
            nestedResultObjects.put(combinedKey, rowValue);
        }
    }
}

```

```

    }
    return rowValue;
}

private void putAncestor(Object resultObject, String resultMapId) {
    ancestorObjects.put(resultMapId, resultObject);
}

//
// NESTED RESULT MAP (JOIN MAPPING)
//

private boolean applyNestedResultMappings(ResultSetWrapper rsw, ResultMap resultMap,
MetaObject metaObject, String parentPrefix, CacheKey parentRowKey, boolean newObject) {
    boolean foundValues = false;
    for (ResultMapping resultMapping : resultMap.getPropertyResultMappings()) {
        final String nestedResultMapId = resultMapping.getNestedResultMapId();
        if (nestedResultMapId != null && resultMapping.getResultSet() == null) { // 仅仅处
理嵌套resultMap的属性
            try {
                final String columnPrefix = getColumnPrefix(parentPrefix, resultMapping);
                // 得到嵌套resultMap的实际定义
                final ResultMap nestedResultMap = getNestedResultMap(rsw.getResultSet(),
nestedResultMapId, columnPrefix);
                if (resultMapping.getColumnPrefix() == null) {
                    // try to fill circular reference only when columnPrefix
                    // is not specified for the nested result map (issue #215)
                    // 不管循环嵌套resultMap的情况
                    Object ancestorObject = ancestorObjects.get(nestedResultMapId);
                    if (ancestorObject != null) {
                        if (newObject) {
                            linkObjects(metaObject, resultMapping, ancestorObject); // issue #385
                        }
                        continue;
                    }
                }
            }
            final CacheKey rowKey = createRowKey(nestedResultMap, rsw, columnPrefix);
            final CacheKey combinedKey = combineKeys(rowKey, parentRowKey);
            Object rowValue = nestedResultObjects.get(combinedKey);
            boolean knownValue = rowValue != null; // 第一次一定是null
            // 为嵌套resultMap属性创建对象或者集合
            instantiateCollectionPropertyIfAppropriate(resultMapping, metaObject); //
mandatory
            // 至少有一个字段不为null
            if (anyNotNullColumnHasValue(resultMapping, columnPrefix, rsw)) {
                // 为嵌套resultMap创建记录
                rowValue = getRowValue(rsw, nestedResultMap, combinedKey, columnPrefix,
rowValue);
                if (rowValue != null && !knownValue) { //获取到记录,就绑定到主记录

```

```

        linkObjects(metaObject, resultMapping, rowValue/*嵌套resultMap的记录*/);
        foundValues = true;
    }
}
} catch (SQLException e) {
    throw new ExecutorException("Error getting nested result map values for '" +
resultMapping.getProperty() + "'. Cause: " + e, e);
}
}
}
return foundValues;
}

```

对于嵌套查询的属性处理，它的实现是这样的：

```

private Object getNestedQueryMappingValue(ResultSet rs, MetaObject metaResultObject,
ResultMapping propertyMapping, ResultLoaderMap lazyLoader, String columnPrefix)
    throws SQLException {
    // 获取queryId
    final String nestedQueryId = propertyMapping.getNestedQueryId();
    final String property = propertyMapping.getProperty();
    // 根据嵌套queryId获取映射语句对象
    final MappedStatement nestedQuery =
configuration.getMappedStatement(nestedQueryId);
    final Class<?> nestedQueryParameterType = nestedQuery.getParameterMap().getType();
    final Object nestedQueryParameterObject = prepareParameterForNestedQuery(rs,
propertyMapping, nestedQueryParameterType, columnPrefix);
    Object value = null;
    if (nestedQueryParameterObject != null) {
        final BoundSql nestedBoundSql =
nestedQuery.getBoundSql(nestedQueryParameterObject);
        final CacheKey key = executor.createCacheKey(nestedQuery,
nestedQueryParameterObject, RowBounds.DEFAULT, nestedBoundSql);
        final Class<?> targetType = propertyMapping.getJavaType();
        if (executor.isCached(nestedQuery, key)) {
            executor.deferLoad(nestedQuery, metaResultObject, property, key, targetType);
            value = DEFERED;
        } else {
            final ResultLoader resultLoader = new ResultLoader(configuration, executor,
nestedQuery, nestedQueryParameterObject, targetType, key, nestedBoundSql);
            if (propertyMapping.isLazy()) {
                lazyLoader.addLoader(property, metaResultObject, resultLoader);
                value = DEFERED;
            } else {
                value = resultLoader.loadResult();
            }
        }
    }
}
return value;
}

```

```

}

private Object prepareParameterForNestedQuery(ResultSet rs, ResultMapping
resultMapping, Class<?> parameterType, String columnPrefix) throws SQLException {
    if (resultMapping.isCompositeResult()) {
        return prepareCompositeKeyParameter(rs, resultMapping, parameterType,
columnPrefix);
    } else {
        return prepareSimpleKeyParameter(rs, resultMapping, parameterType, columnPrefix);
    }
}

private Object prepareSimpleKeyParameter(ResultSet rs, ResultMapping resultMapping,
Class<?> parameterType, String columnPrefix) throws SQLException {
    final TypeHandler<?> typeHandler;
    if (typeHandlerRegistry.hasTypeHandler(parameterType)) {
        typeHandler = typeHandlerRegistry.getTypeHandler(parameterType);
    } else {
        typeHandler = typeHandlerRegistry.getUnknownTypeHandler();
    }
    return typeHandler.getResult(rs, prependPrefix(resultMapping.getColumn(),
columnPrefix));
}

private Object prepareCompositeKeyParameter(ResultSet rs, ResultMapping
resultMapping, Class<?> parameterType, String columnPrefix) throws SQLException {
    final Object parameterObject = instantiateParameterObject(parameterType);
    final MetaObject metaObject = configuration.newMetaObject(parameterObject);
    boolean foundValues = false;
    for (ResultMapping innerResultMapping : resultMapping.getComposites()) {
        final Class<?> propType =
metaObject.getSetterType(innerResultMapping.getProperty());
        final TypeHandler<?> typeHandler = typeHandlerRegistry.getTypeHandler(propType);
        final Object propValue = typeHandler.getResult(rs,
prependPrefix(innerResultMapping.getColumn(), columnPrefix));
        // issue #353 & #560 do not execute nested query if key is null
        if (propValue != null) {
            metaObject.setValue(innerResultMapping.getProperty(), propValue);
            foundValues = true;
        }
    }
    return foundValues ? parameterObject : null;
}

```

selectMap实现

看完selectList/selectOne的实现，我们来看下selectMap的实现，需要注意的是，这个selectMap并不等价于方法public List selectList，它返回的格式直接是Map，Key是查询记录的某个字段，一般应该唯一，Value是查询记录本身，也就是Map<object.prop1,object>。

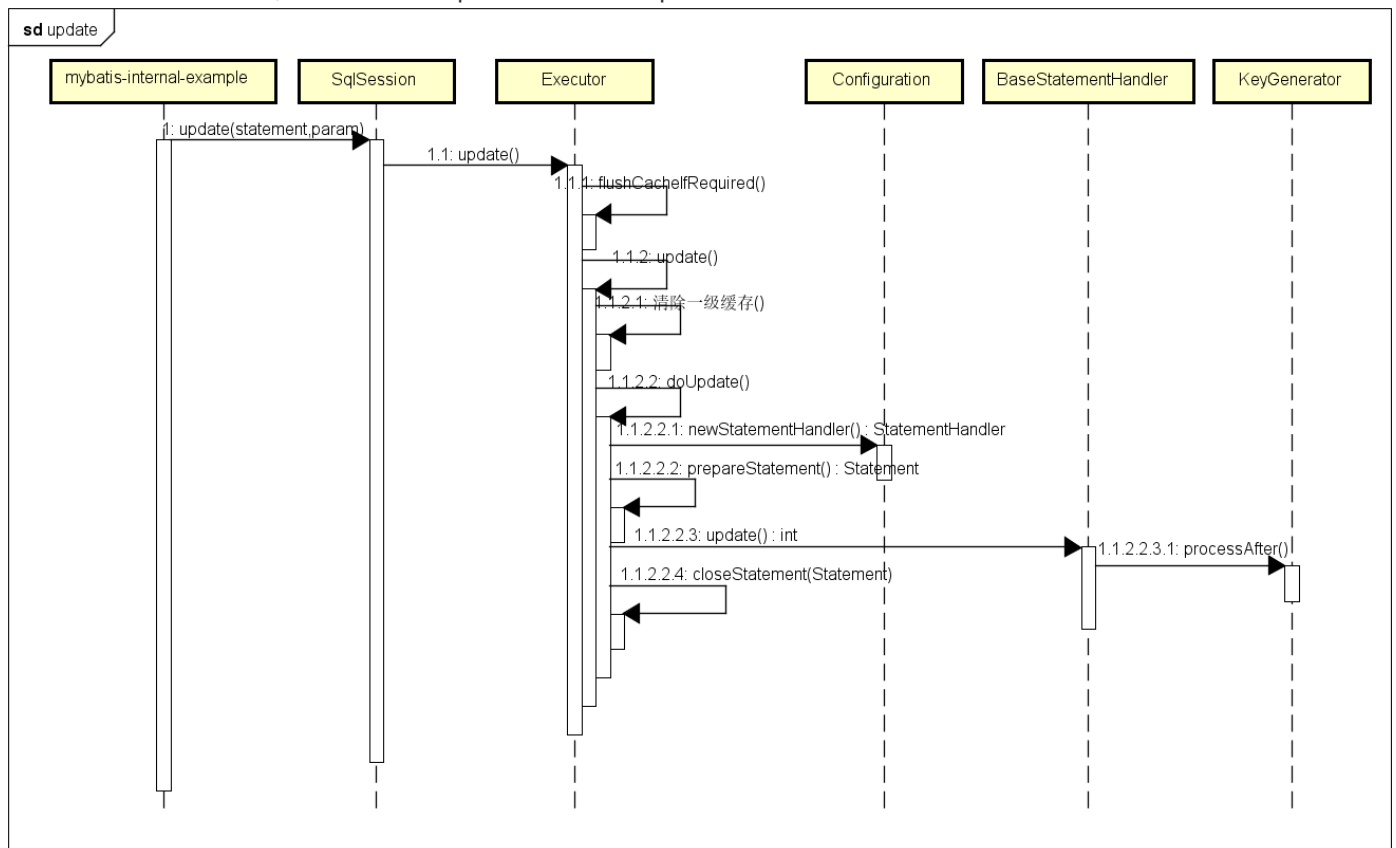
selectMap的入口为：

```
@Override
public <K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey,
RowBounds rowBounds) {
    final List<? extends V> list = selectList(statement, parameter, rowBounds);
    final DefaultMapResultHandler<K, V> mapResultHandler = new
DefaultMapResultHandler<K, V>(mapKey,
        configuration.getObjectFactory(), configuration.getObjectWrapperFactory(),
configuration.getReflectorFactory());
    final DefaultResultContext<V> context = new DefaultResultContext<V>();
    for (V o : list) {
        context.nextResultObject(o);
        mapResultHandler.handleResult(context);
    }
    return mapResultHandler.getMappedResults();
}
```

从方法签名上，我们可以看到，和selectList不同，selectMap多了一个参数mapKey，mapKey就是用来指定返回类型中作为key的那个字段名，具体的核心逻辑委托给了selectList方法，只是在返回结果后，mapResultHandler进行了二次处理。DefaultMapResultHandler是众多ResultHandler的实现之一。DefaultMapResultHandler.handleResult()的功能就是把List转换为Map<object.prop1,object>格式。

update/insert/delete实现

看完select的实现，我们再来看update的实现。update操作的整体流程为：



具体实现代码如下：

```
@Override
public int update(String statement, Object parameter) {
    try {
        dirty = true;
        MappedStatement ms = configuration.getMappedStatement(statement);
        return executor.update(ms, wrapCollection(parameter));
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error updating database. Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}
```

首先设置了字段dirty=true(dirty主要用在非自动提交模式下，用于判断是否需要提交或回滚，在强行提交模式下，如果dirty=true，则需要提交或者回滚，代表可能有pending的事务)，然后调用执行器实例的update()方法，如下：

```

@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.");
    }
    // 清空本地缓存, 与本地出参缓存
    clearLocalCache();
    // 调用具体执行器实现的doUpdate方法
    return doUpdate(ms, parameter);
}

```

我们以SimpleExecutor为例，看下doUpdate的实现：

```

@Override
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter,
RowBounds.DEFAULT, null, null);
        stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.update(stmt);
    } finally {
        closeStatement(stmt);
    }
}

```

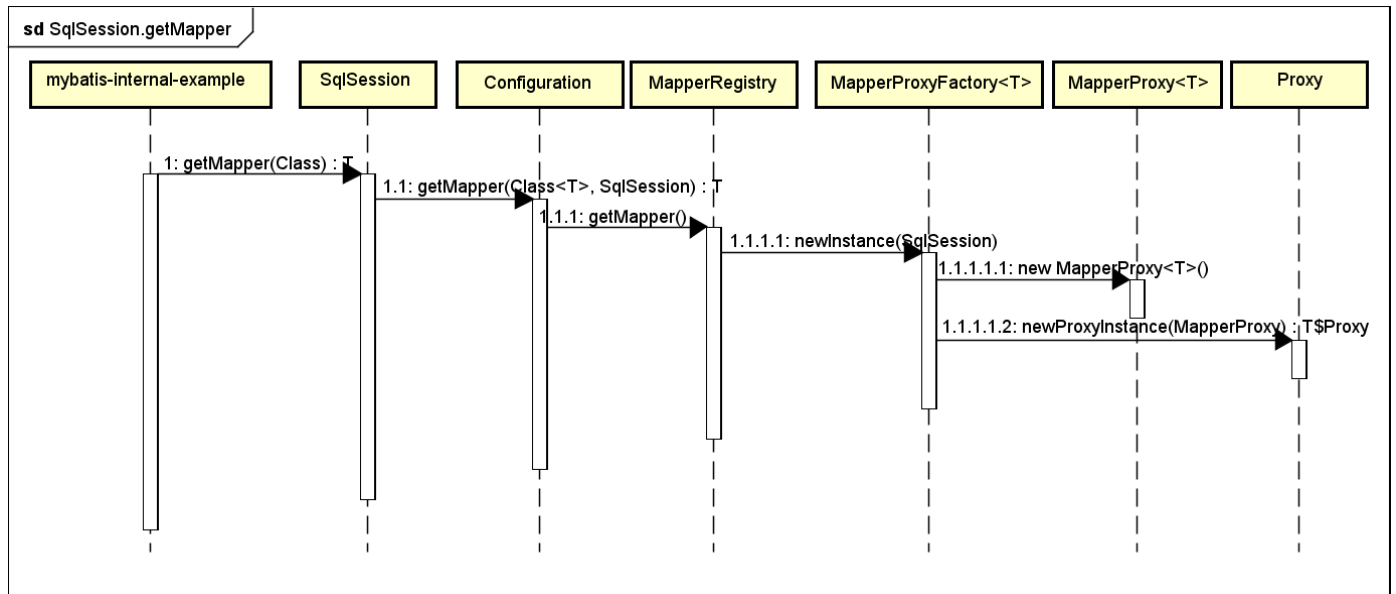
其中的逻辑可以发现，和selectList的实现非常相似，先创建语句处理器，然后创建Statement实例，最后调用语句处理的update，语句处理器里面调用jdbc对应update的方法execute()。和selectList的不同之处在于：

1. 在创建语句处理器期间，会根据需要调用KeyGenerator.processBefore生成前置id；
2. 在执行完成execute()方法后，会根据需要调用KeyGenerator.processAfter生成后置id；

通过分析delete/insert，我们会发现他们内部都委托给update实现了，所以我们就做重复的分析了。

4.2.3 SQL语句执行方式二 SqlSession.getMapper实现

通过SqlSession.getMapper执行CRUD语句的流程为：



我们现在来看下SqlSession的getMapper()是如何实现的。DefaultSqlSession将具体创建Mapper实现的任务委托给了Configuration的getMapper泛型方法，如下所示：

```

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

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

```

最后调用MapperRegistry.getMapper得到Mapper的实现代理,如下所示：

```

public class MapperRegistry {
    ...
    @SuppressWarnings("unchecked")
    public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
        final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>)
knownMappers.get(type);
        if (mapperProxyFactory == null) {
            throw new BindingException("Type " + type + " is not known to the
MapperRegistry.");
        }
        try {
            return mapperProxyFactory.newInstance(sqlSession);
        } catch (Exception e) {
            throw new BindingException("Error getting mapper instance. Cause: " + e, e);
        }
    }
}

```

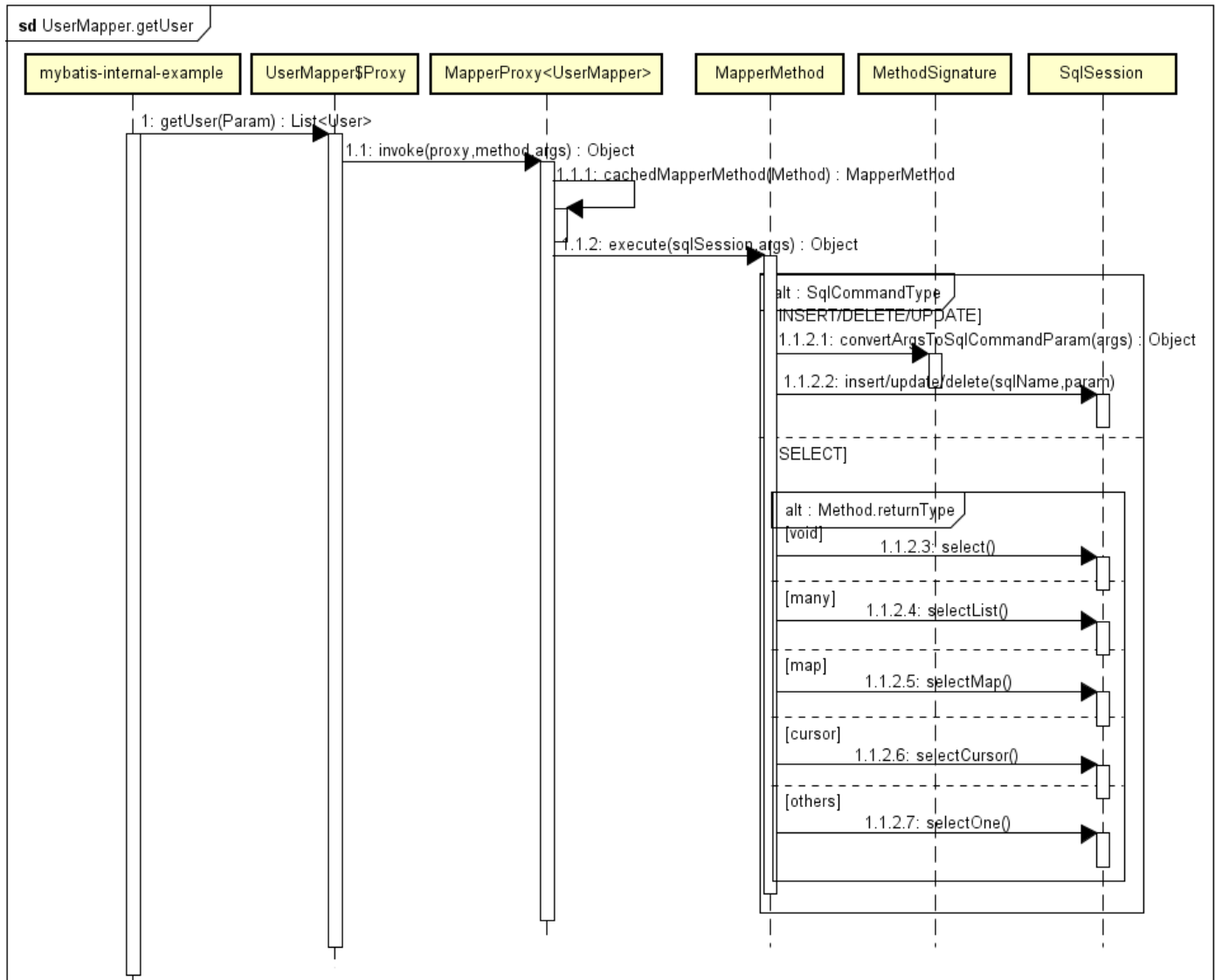


```
    }  
    }  
}
```

MapperRegistry又将创建代理的任务委托给MapperProxyFactory，MapperProxyFactory首先为Mapper接口创建了一个实现了InvocationHandler方法调用处理器接口的代理类MapperProxy，并实现invoke接口（其中为mapper各方法执行sql的具体逻辑），最后才调用JDK的java.lang.reflect.Proxy为Mapper接口创建动态代理类并返回。如下所示：

```
public class MapperProxyFactory<T> {  
    ...  
    @SuppressWarnings("unchecked")  
    protected T newInstance(MapperProxy<T> mapperProxy) {  
        return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] {  
mapperInterface }, mapperProxy);  
    }  
  
    public T newInstance(SqlSession sqlSession) {  
        final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface,  
methodCache);  
        return newInstance(mapperProxy);  
    }  
}
```

这样当我们应用层执行List users = mapper.getUser2(293);的时候，JVM会首先调用MapperProxy.invoke，如下：



具体实现代码如下：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try {
        if (Object.class.equals(method.getDeclaringClass())) {
            return method.invoke(this, args);
        } else if (isDefaultMethod(method)) {
            return invokeDefaultMethod(proxy, method, args);
        }
    } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
    }
    final MapperMethod mapperMethod = cachedMapperMethod(method);
    return mapperMethod.execute(sqlSession, args);
}
```

MapperMethod.execute实现如下：

```

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);
            }
            break;
        case FLUSH: // 主要用于BatchExecutor和CacheExecutor的场景,SimpleExecutor模式不适用
            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;
}

```

对非查询类SQL，首先将请求参数转换为mybatis内部的格式，然后调用sqlSession实例对应的方法，这就和第一种方式的SQL逻辑一样的。

对于查询类SQL，根据返回类型是void/many/map/one/cursor分别调用不同的实现入口，但主体逻辑都类似，除少数特别处理外，都是调用sqlSession.selectXXX，这里我们就不一一讲解。

4.3 动态sql

准确的说,只要mybatis的crud语句中包含了、等标签或者\${}之后，就已经算是动态sql了，所以只要在mybatis加载mapper文件期间被解析为非StaticSqlSource，就会被当做动态sql处理，在执行selectXXX或者update/insert/delete期间，就会调用对应的SqlNode接口和TextSqlNode.isDynamic()处理各自的标签以及\${}，并最终将每个sql片段处理到StaticTextSqlNode并生成最终的参数化静态SQL语句为止。所以，可以说，在绝大部分非PK查询的情况下，我们都是在使用动态SQL。

4.4 存储过程与函数调用实现

如果MappedStatement.StatementType类型为CALLABLE，在Executor.doQuery方法中创建语句处理器的时候，就会返回CallableStatementHandler实例，随后在调用语句处理器的初始化语句和设置参数 方法时，调用jdbc对应存储过程的prepareCall方法，如下：

```
@Override
protected Statement instantiateStatement(Connection connection) throws SQLException {
    String sql = boundSql.getSql();
    if (mappedStatement.getResultSetType() != null) {
        return connection.prepareCall(sql, mappedStatement.getResultSetType().getValue(),
            ResultSet.CONCUR_READ_ONLY);
    } else {
        return connection.prepareCall(sql);
    }
}

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

private void registerOutputParameters(CallableStatement cs) throws SQLException {
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    for (int i = 0, n = parameterMappings.size(); i < n; i++) {
        ParameterMapping parameterMapping = parameterMappings.get(i);
        if (parameterMapping.getMode() == ParameterMode.OUT || parameterMapping.getMode()
            == ParameterMode.INOUT) {
            if (null == parameterMapping.getJdbcType()) {
                throw new ExecutorException("The JDBC Type must be specified for output
                    parameter. Parameter: " + parameterMapping.getProperty());
            } else {
                if (parameterMapping.getNumericScale() != null &&
                    (parameterMapping.getJdbcType() == JdbcType.NUMERIC || parameterMapping.getJdbcType()
                        == JdbcType.DECIMAL)) {
```

```
        cs.registerOutParameter(i + 1, parameterMapping.getJdbcType().TYPE_CODE,  
parameterMapping.getNumericScale());  
    } else {  
        if (parameterMapping.getJdbcTypeName() == null) {  
            cs.registerOutParameter(i + 1, parameterMapping.getJdbcType().TYPE_CODE);  
        } else {  
            cs.registerOutParameter(i + 1, parameterMapping.getJdbcType().TYPE_CODE,  
parameterMapping.getJdbcTypeName());  
        }  
    }  
}  
}
```

4.5 mybatis事务实现

mybatis的事务管理模式分为两种，自动提交和手工提交，DefaultSqlSessionFactory的openSession中重载中，提供了一个参数用于控制是否自动提交事务，该参数最终被传递给 java.sql.Connection.setAutoCommit()方法用于控制是否自动提交事务(默认情况下，连接是自动提交的)，如下所示：

```
private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
    Transaction tx = null;
    try {
        final Environment environment = configuration.getEnvironment();
        final TransactionFactory transactionFactory =
getTransactionFactoryFromEnvironment(environment);
        tx = transactionFactory.newTransaction(environment.getDataSource(), level,
autoCommit);
        final Executor executor = configuration.newExecutor(tx, execType);
        return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
        closeTransaction(tx); // may have fetched a connection so lets call close()
        throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}
```

如上所示，返回的事务传递给了执行器，因为执行器是在事务上下文中执行，所以对于自动提交模式，实际上mybatis不需要去关心。只有非自动管理模式，mybatis才需要关心事务。对于非自动提交模式，通过sqlSession.commit()或sqlSession.rollback()发起，在进行提交或者回滚的时候会调用isCommitOrRollbackRequired判断是否应该提交或者回滚事务，如下所示：

```
private boolean isCommitOrRollbackRequired(boolean force) {
    return (!autoCommit && dirty) || force;
}
```

只有非自动提交模式且执行过DML操作或者设置强制提交才会认为应该进行事务提交或者回滚操作。

对于不同的执行器，在提交和回滚执行的逻辑不一样，因为每个执行器在一级、二级、语句缓存上的差异：

- 对于简单执行器，除了清空一级缓存外，什么都不做；
- 对于REUSE执行器，关闭每个缓存的Statement以释放服务器端语句处理器，然后清空缓存的语句；
- 对于批量处理器，则执行每个批处理语句的executeBatch()方法以便真正执行语句，然后关闭Statement；

上述逻辑执行完成后，会执行提交/回滚操作。对于缓存执行器，在提交/回滚完成之后，会将TransactionCache中的entriesMissedInCache和entriesToAddOnCommit列表分别移动到语句对应的二级缓存中或清空掉。

4.6 缓存

只要实现org.apache.ibatis.cache.Cache接口的任何类都可以当做缓存，Cache接口很简单：

```
public interface Cache {

    /**
     * @return The identifier of this cache
     */
    String getId();

    /**
     * @param key Can be any object but usually it is a {@link CacheKey}
     * @param value The result of a select.
     */
    void putObject(Object key, Object value);

    /**
     * @param key The key
     * @return The object stored in the cache.
     */
    Object getObject(Object key);

    /**
     * As of 3.3.0 this method is only called during a rollback
     * for any previous value that was missing in the cache.
     * This lets any blocking cache to release the lock that
     * may have previously put on the key.
     * A blocking cache puts a lock when a value is null
     * and releases it when the value is back again.
     * This way other threads will wait for the value to be
     * available instead of hitting the database.
     *
     *
     */
}
```

```

    * @param key The key
    * @return Not used
    */
    Object removeObject(Object key);

    /**
     * Clears this cache instance
     */
    void clear();

    /**
     * Optional. This method is not called by the core.
     *
     * @return The number of elements stored in the cache (not its capacity).
     */
    int getSize();

    /**
     * Optional. As of 3.2.6 this method is no longer called by the core.
     *
     * Any locking needed by the cache must be provided internally by the cache provider.
     *
     * @return A ReadWriteLock
     */
    ReadWriteLock getReadWriteLock();
}

```

mybatis提供了基本实现org.apache.ibatis.cache.impl.PerpetualCache，内部采用原始HashMap实现。第二个需要知道的方面是mybatis有一级缓存和二级缓存。一级缓存是SqlSession级别的缓存，不同SqlSession之间的缓存数据区域（HashMap）是互相不影响，MyBatis默认支持一级缓存，不需要任何的配置，默认情况下(一级缓存的有效范围可通过参数localCacheScope参数修改，取值为SESSION或者STATEMENT)，在一个SqlSession的查询期间，只要没有发生commit/rollback或者调用close()方法，那么mybatis就会先根据当前执行语句的CacheKey到一级缓存中查找，如果找到了就直接返回，不到数据库中执行。其实现在代码BaseExecutor.query()中，如下所示：

```

@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++;
    // 如果在一级缓存中就直接获取
    ==list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
    if (list != null) {
        handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
    } else {
        list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
    }
} finally {
    queryStack--;
}
if (queryStack == 0) {
    for (DeferredLoad deferredLoad : deferredLoads) {
        deferredLoad.load();
    }
    // issue #601
    deferredLoads.clear();
    if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        // issue #482
        // 如果设置了一级缓存是STATEMENT级别而非默认的SESSION级别，一级缓存就去掉了
        clearLocalCache();
    }
}
return list;
}

```

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession。二级缓存默认不启用，需要通过在Mapper中明确设置cache，它的实现在CachingExecutor的query()方法中，如下所示：

```

@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, parameterObject, boundSql);
            @SuppressWarnings("unchecked")
            // 如果二级缓存中找到了记录就直接返回,否则到DB查询后进行缓存
            List<E> list = (List<E>) tcm.getObject(cache, key);
            if (list == null) {
                list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key,
boundSql);
                tcm.putObject(cache, key, list); // issue #578 and #116
            }
        }
    }
}

```



```

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

```

在mybatis的缓存实现中，缓存键CacheKey的格式为：cacheKey=ID + offset + limit + sql + parameterValues + environmentId。对于本书例子中的语句，其CacheKey为：

```

-1445574094:212285810:org.mybatis.internal.example.mapper.UserMapper.getUser:0:2147483647:select lfPartyId,partyName from LfParty where partyName = ? AND partyName like ? and lfPartyId in ( ?, ?):p2:p2:1:2:development

```

- 对于一级缓存，commit/rollback都会清空一级缓存。
- 对于二级缓存，DML操作或者显示设置语句层面的flushCache属性都会使得二级缓存失效。

在二级缓存容器的具体回收策略实现上，有下列几种：

- LRU – 最近最少使用的：移除最长时间不被使用的对象，也是默认的选项，其实现类是org.apache.ibatis.cache.decorators.LruCache。
- FIFO – 先进先出：按对象进入缓存的顺序来移除它们，其实现类是org.apache.ibatis.cache.decorators.FifoCache。
- SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象，其实现类是org.apache.ibatis.cache.decorators.SoftCache。
- WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象，其实现类是org.apache.ibatis.cache.decorators.WeakCache。

在缓存的设计上，Mybatis的所有Cache算法都是基于装饰器/Composite模式对PerpetualCache扩展增加功能。

对于模块化微服务系统来说，应该来说mybatis的一二级缓存对业务数据都不适合，尤其是对于OLTP系统来说，CRM/BI这些不算，如果要求数据非常精确的话，也不是特别合适。对这些要求数据准确的系统来说，尽可能只使用mybatis的ORM特性比较靠谱。但是有一部分数据如果前期没有很少的设计缓存的话，是很有价值的，比如说对于一些配置类数据比如数据字典、系统参数、业务配置项等很少变化的数据。

5 执行期主要类总结

mybatis在执行期间，主要有四大核心接口对象：

- 执行器Executor，执行器负责整个SQL执行过程的总体控制。
- 参数处理器ParameterHandler，参数处理器负责PreparedStatement入参的具体设置。
- 语句处理器StatementHandler，语句处理器负责和JDBC层具体交互，包括prepare语句，执行语句，以及调用ParameterHandler.parameterize()设置参数。
- 结果集处理器ResultSetHandler，结果处理器负责将JDBC查询结果映射到java对象。

5.1 执行器Executor

什么是执行器？所有我们在应用层通过sqlSession执行的各类selectXXX和增删改操作在做了动态sql和参数相关的封装处理后，都被委托给具体的执行器去执行，包括一、二级缓存的管理，事务的具体管理，Statement和具体JDBC层面优化的实现等等。所以执行器比较像是sqlSession下的各个策略工厂实现，用户通过配置决定使用哪个策略工厂。只不过执行器在一个mybatis配置下只有一个，这可能无法适应于所有的情况，尤其是哪些微服务做得不是特别好的中小型公司，因为这些系统通常混搭了OLTP和ETL功能。先来看下执行器接口的定义：

```
public interface Executor {

    ResultHandler NO_RESULT_HANDLER = null;

    int update(MappedStatement ms, Object parameter) throws SQLException;

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

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

    <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds)
throws SQLException;

    List<BatchResult> flushStatements() throws SQLException;

    void commit(boolean required) throws SQLException;

    void rollback(boolean required) throws SQLException;

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

    boolean isCached(MappedStatement ms, CacheKey key);

    void clearLocalCache();

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

    Transaction getTransaction();








    void close(boolean forceRollback);

    boolean isClosed();

    void setExecutorWrapper(Executor executor);

}
```

mybatis提供了下列类型的执行器：

- ✓  **Executor** - org.apache.ibatis.executor
 - ✓  **BaseExecutor** - org.apache.ibatis.executor
 -  BatchExecutor - org.apache.ibatis.executor
 -  ClosedExecutor - org.apache.ibatis.executor.loader.ResultLoaderMap
 -  ReuseExecutor - org.apache.ibatis.executor
 -  SimpleExecutor - org.apache.ibatis.executor
 -  CachingExecutor - org.apache.ibatis.executor

从上述可以看出，mybatis提供了两种类型的执行器，缓存执行器与非缓存执行器（使用哪个执行器是通过配置文件中settings下的属性defaultExecutorType控制的，默认是SIMPLE），是否使用缓存执行器则是通过执行cacheEnabled控制的，默认是true。

缓存执行器不是真正功能上独立的执行器，而是非缓存执行器的装饰器模式。

我们先来看非缓存执行器。非缓存执行器又分为三种，这三种类型的执行器都基于基础执行器BaseExecutor，基础执行器完成了大部分的公共功能，如下所示：

```
package org.apache.ibatis.executor;
...
public abstract class BaseExecutor implements Executor {

    protected Transaction transaction;
    protected Executor wrapper;

    protected ConcurrentLinkedQueue<DeferredLoad> deferredLoads;
    // mybatis的二级缓存 PerpetualCache实际上内部使用的是常规的Map
    protected PerpetualCache localCache;
    // 用于存储过程出参
    protected PerpetualCache localOutputParameterCache;
    protected Configuration configuration;

    protected int queryStack;

    // transaction的底层连接是否已经释放
    private boolean closed;

    protected BaseExecutor(Configuration configuration, Transaction transaction) {
        this.transaction = transaction;
        this.deferredLoads = new ConcurrentLinkedQueue<DeferredLoad>();
        this.localCache = new PerpetualCache("LocalCache");
        this.localOutputParameterCache = new PerpetualCache("LocalOutputParameterCache");
        this.closed = false;
        this.configuration = configuration;
        this.wrapper = this;
    }

    @Override
    public Transaction getTransaction() {
```

```

        if (closed) {
            throw new ExecutorException("Executor was closed.");
        }
        return transaction;
    }

    // 关闭本执行器相关的transaction
    @Override
    public void close(boolean forceRollback) {
        try {
            try {
                rollback(forceRollback);
            } finally {
                if (transaction != null) {
                    transaction.close();
                }
            }
        } catch (SQLException e) {
            // Ignore. There's nothing that can be done at this point.
            log.warn("Unexpected exception on closing transaction. Cause: " + e);
        } finally {
            transaction = null;
            deferredLoads = null;
            localCache = null;
            localOutputParameterCache = null;
            closed = true;
        }
    }

    @Override
    public boolean isClosed() {
        return closed;
    }

    // 更新操作
    @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.");
        }
        clearLocalCache();
        return doUpdate(ms, parameter);
    }

    @Override
    public List<BatchResult> flushStatements() throws SQLException {
        return flushStatements(false);
    }

```

```

}

public List<BatchResult> flushStatements(boolean isRollBack) throws SQLException {
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    return doFlushStatements(isRollBack);
}

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

@SuppressWarnings("unchecked")
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
    ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing a
query").object(ms.getId());
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
        clearLocalCache();
    }
    List<E> list;
    try {
        queryStack++;
        list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
        if (list != null) {
            handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
        } else {
            list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
        }
    } finally {
        queryStack--;
    }
    if (queryStack == 0) {
        for (DeferredLoad deferredLoad : deferredLoads) {
            deferredLoad.load();
        }
        // issue #601
        deferredLoads.clear();
        if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {

```

```

        // issue #482
        clearLocalCache();
    }
}
return list;
}

@Override
public <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds
rowBounds) throws SQLException {
    BoundSql boundSql = ms.getBoundSql(parameter);
    return doQueryCursor(ms, parameter, rowBounds, boundSql);
}

@Override
public void deferLoad(MappedStatement ms, MetaObject resultObject, String property,
CacheKey key, Class<?> targetType) {
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    DeferredLoad deferredLoad = new DeferredLoad(resultObject, property, key,
localCache, configuration, targetType);
    if (deferredLoad.canLoad()) {
        deferredLoad.load();
    } else {
        deferredLoads.add(new DeferredLoad(resultObject, property, key, localCache,
configuration, targetType));
    }
}

@Override
public CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, BoundSql boundSql) {
    if (closed) {
        throw new ExecutorException("Executor was closed.");
    }
    CacheKey cacheKey = new CacheKey();
    cacheKey.update(ms.getId());
    cacheKey.update(rowBounds.getOffset());
    cacheKey.update(rowBounds.getLimit());
    cacheKey.update(boundSql.getSql());
    List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
    TypeHandlerRegistry typeHandlerRegistry =
ms.getConfiguration().getTypeHandlerRegistry();
    // mimic DefaultParameterHandler logic
    for (ParameterMapping parameterMapping : parameterMappings) {
        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);
        }
        cacheKey.update(value);
    }
}

if (configuration.getEnvironment() != null) {
    // issue #176
    cacheKey.update(configuration.getEnvironment().getId());
}

return cacheKey;
}

@Override
public boolean isCached(MappedStatement ms, CacheKey key) {
    return localCache.getObject(key) != null;
}

@Override
public void commit(boolean required) throws SQLException {
    if (closed) {
        throw new ExecutorException("Cannot commit, transaction is already closed");
    }
    clearLocalCache();
    flushStatements();
    if (required) {
        transaction.commit();
    }
}

@Override
public void rollback(boolean required) throws SQLException {
    if (!closed) {
        try {
            clearLocalCache();
            flushStatements(true);
        } finally {
            if (required) {
                transaction.rollback();
            }
        }
    }
}

```

```

}

@Override
public void clearLocalCache() {
    if (!closed) {
        localCache.clear();
        localOutputParameterCache.clear();
    }
}

// 接下去的4个方法由子类进行实现
protected abstract int doUpdate(MappedStatement ms, Object parameter)
    throws SQLException;

protected abstract List<BatchResult> doFlushStatements(boolean isRollback)
    throws SQLException;

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

protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter,
RowBounds rowBounds, BoundSql boundSql)
    throws SQLException;

protected void closeStatement(Statement statement) {
    if (statement != null) {
        try {
            statement.close();
        } catch (SQLException e) {
            // ignore
        }
    }
}

/**
 * Apply a transaction timeout.
 * @param statement a current statement
 * @throws SQLException if a database access error occurs, this method is called on a
closed <code>Statement</code>
 * @since 3.4.0
 * @see StatementUtil#applyTransactionTimeout(Statement, Integer, Integer)
 */
protected void applyTransactionTimeout(Statement statement) throws SQLException {
    StatementUtil.applyTransactionTimeout(statement, statement.getQueryTimeout(),
transaction.getTimeout());
}

```



```

private void handleLocallyCachedOutputParameters(MappedStatement ms, CacheKey key,
Object parameter, BoundSql boundSql) {
    if (ms.getStatementType() == StatementType.CALLABLE) {
        final Object cachedParameter = localOutputParameterCache.getObject(key);
        if (cachedParameter != null && parameter != null) {
            final MetaObject metaCachedParameter =
configuration.newMetaObject(cachedParameter);
            final MetaObject metaParameter = configuration.newMetaObject(parameter);
            for (ParameterMapping parameterMapping : boundSql.getParameterMappings()) {
                if (parameterMapping.getMode() != ParameterMode.IN) {
                    final String parameterName = parameterMapping.getProperty();
                    final Object cachedValue = metaCachedParameter.getValue(parameterName);
                    metaParameter.setValue(parameterName, cachedValue);
                }
            }
        }
    }
}

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws
SQLException {
    List<E> list;
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
        list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
    } finally {
        localCache.removeObject(key);
    }
    localCache.putObject(key, list);
    if (ms.getStatementType() == StatementType.CALLABLE) {
        localOutputParameterCache.putObject(key, parameter);
    }
    return list;
}

protected Connection getConnection(Log statementLog) throws SQLException {
    Connection connection = transaction.getConnection();
    if (statementLog.isDebugEnabled()) {
        return ConnectionLogger.newInstance(connection, statementLog, queryStack);
    } else {
        return connection;
    }
}

@Override
public void setExecutorWrapper(Executor wrapper) {
    this.wrapper = wrapper;
}

```

```

private static class DeferredLoad {

    private final MetaObject resultObject;
    private final String property;
    private final Class<?> targetType;
    private final CacheKey key;
    private final PerpetualCache localCache;
    private final ObjectFactory objectFactory;
    private final ResultExtractor resultExtractor;

    // issue #781
    public DeferredLoad(MetaObject resultObject,
                        String property,
                        CacheKey key,
                        PerpetualCache localCache,
                        Configuration configuration,
                        Class<?> targetType) {
        this.resultObject = resultObject;
        this.property = property;
        this.key = key;
        this.localCache = localCache;
        this.objectFactory = configuration.getObjectFactory();
        this.resultExtractor = new ResultExtractor(configuration, objectFactory);
        this.targetType = targetType;
    }

    public boolean canLoad() {
        return localCache.getObject(key) != null && localCache.getObject(key) !=
EXECUTION_PLACEHOLDER;
    }

    public void load() {
        @SuppressWarnings( "unchecked" )
        // we suppose we get back a List
        List<Object> list = (List<Object>) localCache.getObject(key);
        Object value = resultExtractor.extractObjectFromList(list, targetType);
        resultObject.setValue(property, value);
    }
}

```

我们先来看下BaseExecutor的属性，从上述BaseExecutor的定义可以看出：

1. 执行器在特定的事务上下文下执行；
2. 具有本地缓存和本地出参缓存（任何时候，只要事务提交或者回滚或者执行update或者查询时设定了刷新缓存，都会清空本地缓存和本地出参缓存）；
3. 具有延迟加载任务；

BaseExecutor实现了大部分通用功能本地缓存管理、事务提交、回滚、超时设置、延迟加载等，但是将下列4个方法留给了具体的子类实现：

```
protected abstract int doUpdate(MappedStatement ms, Object parameter)
    throws SQLException;

protected abstract List<BatchResult> doFlushStatements(boolean isRollback)
    throws SQLException;

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

protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter,
RowBounds rowBounds, BoundSql boundSql)
    throws SQLException;
```

从功能上来说，这三种执行器的差别在于：

- ExecutorType.SIMPLE：这个执行器类型不做特殊的事情。它为每个语句的每次执行创建一个新的预处理语句。
- ExecutorType.REUSE：这个执行器类型会复用预处理语句。
- ExecutorType.BATCH：这个执行器会批量执行所有更新语句，也就是jdbc addBatch API的facade模式。

所以这三种类型的执行器可以说时应用于不同的负载场景下，除了SIMPLE类型外，另外两种要求对系统有较好的架构设计，当然也提供了更多的回报。

5.4.1 SIMPLE执行器

我们先来看SIMPLE各个方法的实现，

```
public class SimpleExecutor extends BaseExecutor {

    public SimpleExecutor(Configuration configuration, Transaction transaction) {
        super(configuration, transaction);
    }

    @Override
    public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
        Statement stmt = null;
        try {
            Configuration configuration = ms.getConfiguration();
            StatementHandler handler = configuration.newStatementHandler(this, ms, parameter,
RowBounds.DEFAULT, null, null);
            stmt = prepareStatement(handler, ms.getStatementLog());
            return handler.update(stmt);
        } finally {
            closeStatement(stmt);
        }
    }
}
```

```

@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.<E>query(stmt, resultHandler);
    } finally {
        closeStatement(stmt);
    }
}

@Override
protected <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds
rowBounds, BoundSql boundSql) throws SQLException {
    Configuration configuration = ms.getConfiguration();
    StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, null, boundSql);
    Statement stmt = prepareStatement(handler, ms.getStatementLog());
    return handler.<E>queryCursor(stmt);
}

@Override
public List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException {
    return Collections.emptyList();
}

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

```

简单执行器的实现非常的简单，我们就不展开详述了。下面俩看REUSE执行器。

5.4.2 REUSE执行器

我们来看下REUSE执行器中和SIMPLE执行器不同的地方：

```
public class ReuseExecutor extends BaseExecutor {

    private final Map<String, Statement> statementMap = new HashMap<String, Statement>();

    public ReuseExecutor(Configuration configuration, Transaction transaction) {
        super(configuration, transaction);
    }

    @Override
    public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter,
RowBounds.DEFAULT, null, null);
        Statement stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.update(stmt);
    }

    @Override
    public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        Statement stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.<E>query(stmt, resultHandler);
    }

    @Override
    protected <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds
rowBounds, BoundSql boundSql) throws SQLException {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, null, boundSql);
        Statement stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.<E>queryCursor(stmt);
    }

    @Override
    public List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException {
        for (Statement stmt : statementMap.values()) {
            closeStatement(stmt);
        }
        statementMap.clear();
        return Collections.emptyList();
    }
}
```

```

    private Statement prepareStatement(StatementHandler handler, Log statementLog) throws
SQLException {
    Statement stmt;
    BoundSql boundSql = handler.getBoundSql();
    String sql = boundSql.getSql();
    if (hasStatementFor(sql)) {
        stmt = getStatement(sql);
        applyTransactionTimeout(stmt);
    } else {
        Connection connection = getConnection(statementLog);
        stmt = handler.prepare(connection, transaction.getTimeout());
        putStatement(sql, stmt);
    }
    handler.parameterize(stmt);
    return stmt;
}

private boolean hasStatementFor(String sql) {
    try {
        return statementMap.keySet().contains(sql) &&
!statementMap.get(sql).getConnection().isClosed();
    } catch (SQLException e) {
        return false;
    }
}

private Statement getStatement(String s) {
    return statementMap.get(s);
}

private void putStatement(String sql, Statement stmt) {
    statementMap.put(sql, stmt);
}
}

```

从实现可以看出，REUSE和SIMPLE在doUpdate/doQuery上有个差别，不再是每执行一个语句就close掉了，而是尽可能的根据SQL文本进行缓存并重用，但是由于数据库服务器端通常对每个连接以及全局的语句(oracle称为游标)handler的数量有限制，oracle中是open_cursors参数控制，mysql中是mysql_stmt_close参数控制，这就会导致如果sql都是靠if各种拼接出来，日积月累可能会导致数据库资源耗尽。其是否有足够价值，视创建Statement语句消耗的资源占整体资源的比例、以及一共有多少完全不同的Statement数量而定，一般来说，纯粹的OLTP且非自动生成的sqlmap，它会比SIMPLE执行器更好。

5.4.3 BATCH执行器

BATCH执行器的实现代码如下：

```
public class BatchExecutor extends BaseExecutor {

    public static final int BATCH_UPDATE_RETURN_VALUE = Integer.MIN_VALUE + 1002;
    // 存储在一个事务中的批量DML的语句列表
    private final List<Statement> statementList = new ArrayList<Statement>();
    // 存放DML语句对应的参数对象,包括自动/手工生成的key
    private final List<BatchResult> batchResultList = new ArrayList<BatchResult>();

    // 最新提交执行的SQL语句
    private String currentSql;

    // 最新提交执行的语句
    private MappedStatement currentStatement;

    public BatchExecutor(Configuration configuration, Transaction transaction) {
        super(configuration, transaction);
    }

    @Override
    public int doUpdate(MappedStatement ms, Object parameterObject) throws SQLException {
        final Configuration configuration = ms.getConfiguration();
        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;
        // 如果最新执行的一条语句和前面一条语句相同,就不创建新的语句了,直接用缓存的语句,只是把参数对象添
        加到该语句对应的BatchResult中
        // 否则的话,无论是否在未提交之前,还有pending的语句,都新插入一条语句到list中
        if (sql.equals(currentSql) && ms.equals(currentStatement)) {
            int last = statementList.size() - 1;
            stmt = statementList.get(last);
            applyTransactionTimeout(stmt);
            handler.parameterize(stmt); //fix Issues 322
            BatchResult batchResult = batchResultList.get(last);
            batchResult.addParameterObject(parameterObject);
        } else {
            Connection connection = getConnection(ms.getStatementLog());
            stmt = handler.prepare(connection, transaction.getTimeout());
            handler.parameterize(stmt); //fix Issues 322
            currentSql = sql;
            currentStatement = ms;
            statementList.add(stmt);
            batchResultList.add(new BatchResult(ms, sql, parameterObject));
        }
    }
}
```

```

// handler.parameterize(stmt);
// 调用jdbc的addBatch方法
handler.batch(stmt);
return BATCH_UPDATE_RETURN_VALUE;
}

@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler, BoundSql boundSql)
    throws SQLException {
    Statement stmt = null;
    try {
        flushStatements();
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameterObject, rowBounds, resultHandler, boundSql);
        Connection connection = getConnection(ms.getStatementLog());
        stmt = handler.prepare(connection, transaction.getTimeout());
        handler.parameterize(stmt);
        return handler.<E>query(stmt, resultHandler);
    } finally {
        closeStatement(stmt);
    }
}

@Override
protected <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds
rowBounds, BoundSql boundSql) throws SQLException {
    flushStatements();
    Configuration configuration = ms.getConfiguration();
    StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, null, boundSql);
    Connection connection = getConnection(ms.getStatementLog());
    Statement stmt = handler.prepare(connection, transaction.getTimeout());
    handler.parameterize(stmt);
    return handler.<E>queryCursor(stmt);
}

@Override
public List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException {
    try {
        List<BatchResult> results = new ArrayList<BatchResult>();
        if (isRollback) {
            return Collections.emptyList();
        }
        for (int i = 0, n = statementList.size(); i < n; i++) {
            Statement stmt = statementList.get(i);
            applyTransactionTimeout(stmt);
            BatchResult batchResult = batchResultList.get(i);

```



```

try {
    batchResult.setUpdateCounts(stmt.executeBatch());
    MappedStatement ms = batchResult.getMappedStatement();
    List<Object> parameterObjects = batchResult.getParameterObjects();
    KeyGenerator keyGenerator = ms.getKeyGenerator();
    if (Jdbc3KeyGenerator.class.equals(keyGenerator.getClass())) {
        Jdbc3KeyGenerator jdbc3KeyGenerator = (Jdbc3KeyGenerator) keyGenerator;
        jdbc3KeyGenerator.processBatch(ms, stmt, parameterObjects);
    } else if (!NoKeyGenerator.class.equals(keyGenerator.getClass())) { //issue
#141
        for (Object parameter : parameterObjects) {
            keyGenerator.processAfter(this, ms, stmt, parameter);
        }
    }
    // Close statement to close cursor #1109
    closeStatement(stmt);
} catch (BatchUpdateException e) {
    StringBuilder message = new StringBuilder();
    message.append(batchResult.getMappedStatement().getId())
        .append(" (batch index #")
        .append(i + 1)
        .append(")")
        .append(" failed.");
    if (i > 0) {
        message.append(" ")
            .append(i)
            .append(" prior sub executor(s) completed successfully, but will be
rolled back.");
    }
    throw new BatchExecutorException(message.toString(), e, results,
batchResult);
}
    results.add(batchResult);
}
return results;
} finally {
    for (Statement stmt : statementList) {
        closeStatement(stmt);
    }
    currentSql = null;
    statementList.clear();
    batchResultList.clear();
}
}
}

```

批量执行器是JDBC Statement.addBatch的实现,对于批量insert而言比如导入大量数据的ETL,驱动器如果支持的话,能够大幅度的提高DML语句的性能（首先最重要的是,网络交互就大幅度减少了），比如对于mysql而言,在5.1.13以上版本的驱动,在连接字符串上rewriteBatchedStatements参数也就是jdbc:mysql://192.168.1.100:3306/test?rewriteBatchedStatements=true后,性能可以提高几十倍,参见 <https://www.cnblogs.com/kxdblog/p/4056010.html> 以及 http://blog.sina.com.cn/s/blog_68b4c68f01013yog.html。因为BatchExecutor对于每个statementList中的语句,都执行executeBatch()方法,因此最差的极端情况是交叉执行不同的DML SQL语句,这种情况退化为原始的方式。比如下列形式就是最差的情况:

```
for(int i=0;i<100;i++) {  
    session.update("insertUser", userReq);  
    session.update("insertUserProfile", userReq);  
}
```

5.4.4 缓存执行器CachingExecutor的实现

```
public class CachingExecutor implements Executor {  
  
    private final Executor delegate;  
    private final TransactionalCacheManager tcm = new TransactionalCacheManager();  
  
    public CachingExecutor(Executor delegate) {  
        this.delegate = delegate;  
        delegate.setExecutorWrapper(this);  
    }  
  
    @Override  
    public Transaction getTransaction() {  
        return delegate.getTransaction();  
    }  
  
    @Override  
    public void close(boolean forceRollback) {  
        try {  
            //issues #499, #524 and #573  
            if (forceRollback) {  
                tcm.rollback();  
            } else {  
                tcm.commit();  
            }  
        } finally {  
            delegate.close(forceRollback);  
        }  
    }  
  
    @Override  
    public boolean isClosed() {  
        return delegate.isClosed();  
    }  
}
```

```

@Override
public int update(MappedStatement ms, Object parameterObject) throws SQLException {
    flushCacheIfRequired(ms);
    return delegate.update(ms, parameterObject);
}

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

@Override
public <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds
rowBounds) throws SQLException {
    flushCacheIfRequired(ms);
    return delegate.queryCursor(ms, parameter, rowBounds);
}

@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)
    throws SQLException {
    Cache cache = ms.getCache();
    // 首先判断是否启用了二级缓存
    if (cache != null) {
        flushCacheIfRequired(ms);
        if (ms.isUseCache() && resultHandler == null) {
            ensureNoOutParams(ms, boundSql);
            @SuppressWarnings("unchecked")
            // 然后判断缓存中是否有对应的缓存条目(正常情况下, 执行DML操作会清空缓存, 也可以语句层面明确明
            确设置), 有的话则返回, 这样就不用二次查询了
            List<E> list = (List<E>) tcm.getObject(cache, key);
            if (list == null) {
                list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key,
                boundSql);
                tcm.putObject(cache, key, list); // issue #578 and #116
            }
            return list;
        }
    }
    return delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key,
    boundSql);
}

@Override

```

```

public List<BatchResult> flushStatements() throws SQLException {
    return delegate.flushStatements();
}

@Override
public void commit(boolean required) throws SQLException {
    delegate.commit(required);
    tcm.commit();
}

@Override
public void rollback(boolean required) throws SQLException {
    try {
        delegate.rollback(required);
    } finally {
        if (required) {
            tcm.rollback();
        }
    }
}

// 存储过程不支持二级缓存
private void ensureNoOutParams(MappedStatement ms, BoundSql boundSql) {
    if (ms.getStatementType() == StatementType.CALLABLE) {
        for (ParameterMapping parameterMapping : boundSql.getParameterMappings()) {
            if (parameterMapping.getMode() != ParameterMode.IN) {
                throw new ExecutorException("Caching stored procedures with OUT params is not supported. Please configure useCache=false in " + ms.getId() + " statement.");
            }
        }
    }
}

@Override
public CacheKey createCacheKey(MappedStatement ms, Object parameterObject, RowBounds rowBounds, BoundSql boundSql) {
    return delegate.createCacheKey(ms, parameterObject, rowBounds, boundSql);
}

@Override
public boolean isCached(MappedStatement ms, CacheKey key) {
    return delegate.isCached(ms, key);
}

@Override
public void deferLoad(MappedStatement ms, MetaObject resultObject, String property, CacheKey key, Class<?> targetType) {
    delegate.deferLoad(ms, resultObject, property, key, targetType);
}

```

```

@Override
public void clearLocalCache() {
    delegate.clearLocalCache();
}

private void flushCacheIfRequired(MappedStatement ms) {
    Cache cache = ms.getCache();
    if (cache != null && ms.isFlushCacheRequired()) {
        tcm.clear(cache);
    }
}

@Override
public void setExecutorWrapper(Executor executor) {
    throw new UnsupportedOperationException("This method should not be called");
}
}

```

缓存执行器相对于其他执行器的差别在于，首先是在query()方法中判断是否使用二级缓存(也就是mapper级别的缓存)。虽然mybatis默认启用了CachingExecutor，但是如果在mapper层面没有明确设置二级缓存的话，就退化为SimpleExecutor了。二级缓存的维护由TransactionalCache(事务化缓存)负责，当在TransactionalCacheManager(事务化缓存管理器)中调用putObject和removeObject方法的时候并不是马上就把对象存放到缓存或者从缓存中删除，而是先把这个对象放到entriesToAddOnCommit和entriesToRemoveOnCommit这两个HashMap之中的一个里，然后当执行commit/rollback方法时再真正地把对象存放到缓存或者从缓存中删除，具体可以参见TransactionalCache.commit/rollback方法。

还有一个差别是使用了TransactionalCacheManager管理事务，其他逻辑就一样了。

5.2 参数处理器ParameterHandler

ParameterHandler的接口定义如下：

```

public interface ParameterHandler {

    Object getParameterObject();

    void setParameters(PreparedStatement ps)
        throws SQLException;

}

```

ParameterHandler只有一个默认实现DefaultParameterHandler，它的代码如下：

```

public class DefaultParameterHandler implements ParameterHandler {

    private final TypeHandlerRegistry typeHandlerRegistry;

    private final MappedStatement mappedStatement;
    private final Object parameterObject;
}

```

```

private final BoundSql boundSql;
private final Configuration configuration;

public DefaultParameterHandler(MappedStatement mappedStatement, Object
parameterObject, BoundSql boundSql) {
    this.mappedStatement = mappedStatement;
    this.configuration = mappedStatement.getConfiguration();
    this.typeHandlerRegistry =
mappedStatement.getConfiguration().getTypeHandlerRegistry();
    this.parameterObject = parameterObject;
    this.boundSql = boundSql;
}

@Override
public Object getParameterObject() {
    return parameterObject;
}
// 设置PreparedStatement的入参
@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)) { // issue #448 ask first
for additional params
                    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 e) {

```

```

        throw new TypeException("Could not set parameters for mapping: " +
parameterMapping + ". Cause: " + e, e);
    } catch (SQLException e) {
        throw new TypeException("Could not set parameters for mapping: " +
parameterMapping + ". Cause: " + e, e);
    }
}
}
}
}
}
}
}

```

ParameterHandler的实现很简单，上面在执行语句的时候详细解释了每个步骤，这里就不重复了。

5.3 语句处理器StatementHandler

先来看下StatementHandler的定义：

```

public interface StatementHandler {

    Statement prepare(Connection connection, Integer transactionTimeout)
        throws SQLException;

    void parameterize(Statement statement)
        throws SQLException;

    void batch(Statement statement)
        throws SQLException;

    int update(Statement statement)
        throws SQLException;

    <E> List<E> query(Statement statement, ResultHandler resultHandler)
        throws SQLException;

    <E> Cursor<E> queryCursor(Statement statement)
        throws SQLException;

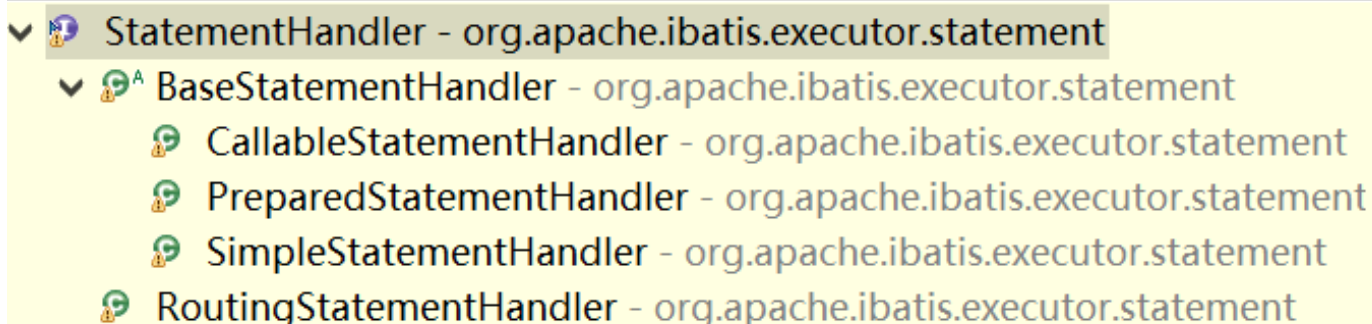
    BoundSql getBoundSql();

    ParameterHandler getParameterHandler();

}

```

从接口可以看出，StatementHandler主要包括prepare语句、给语句设置参数、执行语句获取要执行的SQL语句本身。mybatis包含了三种类型的StatementHandler实现：



分别用于JDBC对应的PreparedStatement,Statement以及CallableStatement。BaseStatementHandler是这三种类型语句处理器的抽象父类，封装了一些实现细节比如设置超时时间、结果集每次提取大小等操作，代码如下：

```
public abstract class BaseStatementHandler implements StatementHandler {

    protected final Configuration configuration;
    protected final ObjectFactory objectFactory;
    protected final TypeHandlerRegistry typeHandlerRegistry;
    protected final ResultSetHandler resultSetHandler;
    protected final ParameterHandler parameterHandler;

    protected final Executor executor;
    protected final MappedStatement mappedStatement;
    protected final RowBounds rowBounds;

    protected BoundSql boundSql;

    protected BaseStatementHandler(Executor executor, MappedStatement mappedStatement,
    Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql
    boundSql) {
        this.configuration = mappedStatement.getConfiguration();
        this.executor = executor;
        this.mappedStatement = mappedStatement;
        this.rowBounds = rowBounds;
        this.typeHandlerRegistry = configuration.getTypeHandlerRegistry();
        this.objectFactory = configuration.getObjectFactory();

        if (boundSql == null) { // issue #435, get the key before calculating the statement
            // 首先执行selectKey对应的SQL语句把ID生成
            generateKeys(parameterObject);
            boundSql = mappedStatement.getBoundSql(parameterObject);
        }

        this.boundSql = boundSql;

        this.parameterHandler = configuration.newParameterHandler(mappedStatement,
        parameterObject, boundSql);
    }
}
```



```

        this.resultSetHandler = configuration.newResultSetHandler(executor,
mappedStatement, rowBounds, parameterHandler, resultHandler, boundSql);
    }

    @Override
    public BoundSql getBoundSql() {
        return boundSql;
    }

    @Override
    public ParameterHandler getParameterHandler() {
        return parameterHandler;
    }
    // prepare SQL语句
    @Override
    public Statement prepare(Connection connection, Integer transactionTimeout) throws
SQLException {
        ErrorContext.instance().sql(boundSql.getSql());
        Statement statement = null;
        try {
            // 创建Statement
            statement = instantiateStatement(connection);
            setStatementTimeout(statement, transactionTimeout);
            setFetchSize(statement);
            return statement;
        } catch (SQLException e) {
            closeStatement(statement);
            throw e;
        } catch (Exception e) {
            closeStatement(statement);
            throw new ExecutorException("Error preparing statement. Cause: " + e, e);
        }
    }

    // 不同类型语句的初始化过程不同,比如Statement语句直接调用JDBC
    java.sql.Connection.createStatement, 而PreparedStatement则是调用
    java.sql.Connection.prepareStatement
    protected abstract Statement instantiateStatement(Connection connection) throws
SQLException;

    // 设置JDBC语句超时时间,注: 数据库服务器端也可以设置语句超时时间。mysql通过参数
    max_statement_time设置,oracle截止12.2c不支持
    protected void setStatementTimeout(Statement stmt, Integer transactionTimeout) throws
SQLException {
        Integer queryTimeout = null;
        if (mappedStatement.getTimeout() != null) {
            queryTimeout = mappedStatement.getTimeout();
        } else if (configuration.getDefaultStatementTimeout() != null) {
            queryTimeout = configuration.getDefaultStatementTimeout();
        }
    }

```

```

    }
    if (queryTimeout != null) {
        stmt.setQueryTimeout(queryTimeout);
    }
    StatementUtil.applyTransactionTimeout(stmt, queryTimeout, transactionTimeout);
}

// fetchSize设置每次从服务器端提取的行数,默认不同数据库实现不同,mysql一次性提取全部,oracle默认
10。正确设置fetchSize可以避免OOM并且对性能有一定的影响,尤其是在网络延时较大的情况下
protected void setFetchSize(Statement stmt) throws SQLException {
    Integer fetchSize = mappedStatement.getFetchSize();
    if (fetchSize != null) {
        stmt.setFetchSize(fetchSize);
        return;
    }
    Integer defaultFetchSize = configuration.getDefaultFetchSize();
    if (defaultFetchSize != null) {
        stmt.setFetchSize(defaultFetchSize);
    }
}

protected void closeStatement(Statement statement) {
    try {
        if (statement != null) {
            statement.close();
        }
    } catch (SQLException e) {
        //ignore
    }
}

protected void generateKeys(Object parameter) {
    KeyGenerator keyGenerator = mappedStatement.getKeyGenerator();
    ErrorContext.instance().store();
    keyGenerator.processBefore(executor, mappedStatement, null, parameter);
    ErrorContext.instance().recall();
}
}

```

5.4 结果集处理器ResultSetHandler

结果集处理器,顾名思义,就是用了查询结果集进行处理的,目标是将JDBC结果集映射为业务对象。其接口定义如下:

```

public interface ResultSetHandler {

    <E> List<E> handleResultSets(Statement stmt) throws SQLException;

    <E> Cursor<E> handleCursorResultSets(Statement stmt) throws SQLException;

    void handleOutputParameters(CallableStatement cs) throws SQLException;

}

```

接口中定义三个接口分别用于处理常规查询的结果集、游标查询的结果集以及存储过程调用的出参设置。和参数处理器一样，结果集处理器也只有一个默认实现DefaultResultSetHandler。结果集处理器的功能包括对象的实例化、属性自动匹配计算、常规属性赋值、嵌套ResultMap的处理、嵌套查询的处理、鉴别器结果集的处理等，每个功能我们在分析SQL语句执行selectXXX的时候都详细的讲解过了，具体可以参见selectXXX部分。

6 插件

插件几乎是所有主流框架提供的一种扩展方式之一，插件可以用于记录日志，统计运行时性能，为核心功能提供额外的辅助支持。在mybatis中，插件是在内部是通过拦截器实现的。要开发自定义插件，只要实现org.apache.ibatis.plugin.Interceptor接口即可，Interceptor接口定义如下：

```

public interface Interceptor {
    //执行代理类方法
    Object intercept(Invocation invocation) throws Throwable;
    // 用于创建代理对象
    Object plugin(Object target);
    // 插件自定义属性
    void setProperties(Properties properties);
}

```

mybatis提供了一个示例插件ExamplePlugin，如下所示：

```

@Intercepts({})
public class ExamplePlugin implements Interceptor {
    private Properties properties;
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        return invocation.proceed();
    }

    @Override
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
        this.properties = properties;
    }
}

```

```

    }

    public Properties getProperties() {
        return properties;
    }

}

```

不过这个例子一点也不完整，没有体现出插件/拦截器的强大之处，mybatis提供了为插件配置提供了两个注解：org.apache.ibatis.plugin.Signature和org.apache.ibatis.plugin.Intercepts。Intercepts注解用来指示当前类是一个拦截器，它的定义如下：

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Intercepts {
    Signature[] value();
}

```

它有一个类型为Signature数组的value属性，如果没有指定，它会拦截StatementHandler、ResultSetHandler、ParameterHandler和Executor这四个核心接口对象中的所有方法。如需改变默认行为，可以通过明确设置value的值，Signature的定义如下：

```

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({})
public @interface Signature {
    Class<?> type();

    String method();

    Class<?>[] args();
}

```

比如：

```

@Intercepts({@Signature(type = StatementHandler.class, method = "prepare", args =
{Connection.class}),
             @Signature(type = ResultSetHandler.class, method = "handleResultSets", args =
{Statement.class})})

```

在实际使用中，使用最频繁的mybatis插件应该算是分页查询插件了，最流行的应该是com.github.pagehelper.PageHelper了。下面我们就来看下PageHelper的详细实现。

6.1 分页插件PageHelper详解

6.2 自定义监控插件StatHelper实现

看过了PageHelper的实现，现在我们来实现一个简单的统计各个sql语句执行时间的插件StatHelper。

7 与spring集成

generated by [haroopad](#)