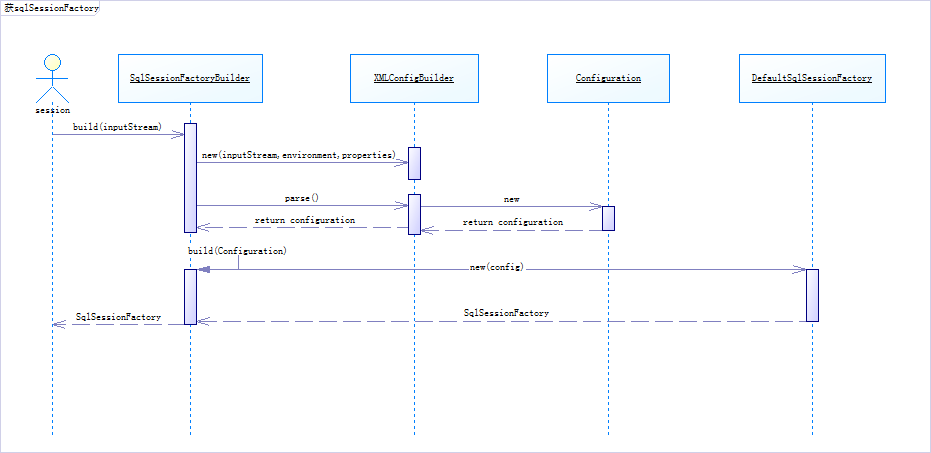
**MyBatis主流程分析之(一)-环境准备**

原创 2016年05月13日 23:45:10

* 标签：
* [mybatis](http://so.csdn.net/so/search/s.do?q=mybatis&t=blog) /
* [配置](http://so.csdn.net/so/search/s.do?q=%E9%85%8D%E7%BD%AE&t=blog) /
* [config-xml](http://so.csdn.net/so/search/s.do?q=config-xml&t=blog) /
* [解析](http://so.csdn.net/so/search/s.do?q=%E8%A7%A3%E6%9E%90&t=blog)
* 1579

**mybatis版本是3.2.6,请注意**

一、 获取SqlSessionFactory顺序图



二、 主要类及其说明

在这个获取SqlSessionFactory的顺序图中，我们可以清晰的看到，SqlSessionFactoryBuilder把读取 mybatis-config.xml交给给了XMLConfigBuilder类，XMLConfigBuilder类把读取的xml内容都记录在了Configuration类中， 最后把配置好的config传递给DefaultSqlSessionFactory类。

三、 XMLConfigBuilder类

我们先看看XMLConfigBuilder的parse方法

public Configuration parse() {

if (parsed) {

throw new BuilderException("Each XMLConfigBuilder can only be used once.");

}

parsed = true;

parseConfiguration(parser.evalNode("/configuration"));

return configuration;

}

private void parseConfiguration(XNode root) {

try {

propertiesElement(root.evalNode("properties"));

typeAliasesElement(root.evalNode("typeAliases"));

pluginElement(root.evalNode("plugins"));

objectFactoryElement(root.evalNode("objectFactory"));

objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

settingsElement(root.evalNode("settings"));

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);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

在parseConfiguration方法内，是不是有一种熟悉的感觉，是的他就是读取我们配置的mybatis-config.xml的各种配置，并且把他们分别配置到configuration类中，下面我讲逐个简单看看他们的源代码。

四、 propertiesElement

这些是外部化的, 可替代的属性, 这些属性也可以配置在典型的 Java 属性配置文件中, 或者通过 properties 元素的子元素来传递。   
例如:

* 配置

<properties resource="org/mybatis/example/config.properties">

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

* 1
* 2
* 3
* 4
* 源代码

//Configuration类中

protected Properties variables = new Properties();

//XMLConfigBuilder类中

private void propertiesElement(XNode context) throws Exception {

if (context != null) {

//保存属性的变量

Properties defaults = context.getChildrenAsProperties();

//下面是读取

String resource = context.getStringAttribute("resource");

String url = context.getStringAttribute("url");

//resource 和 url 配置不能共存

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.");

}

//resource 方式

if (resource != null) {

defaults.putAll(Resources.getResourceAsProperties(resource));

} else if (url != null) {

//外部url文件

defaults.putAll(Resources.getUrlAsProperties(url));

}

//已经包含的属性

Properties vars = configuration.getVariables();

if (vars != null) {

defaults.putAll(vars);

}

parser.setVariables(defaults);

//最后都保存到了configuration中

configuration.setVariables(defaults);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34

官方文档：   
如果在这些地方,属性多于一个的话,MyBatis 按照如下的顺序加载它们:   
•在 properties 元素体内指定的属性首先被读取。   
•从类路径下资源或 properties 元素的 url 属性中加载的属性第二被读取,它会 覆盖已经存在的完全一样的属性。   
•作为方法参数传递的属性最后被读取, 它也会覆盖任一已经存在的完全一样的 属性,这些属性可能是从 properties 元素体内和资源/url 属性中加载的。

因此, 最高优先级的属性是那些作为方法参数的, 然后是资源/url 属性, 最后是 properties 元素中指定的属性。

五、 typeAliasesElement

类型别名是为 Java 类型命名一个短的名字。 它只和 XML 配置有关, 只用来减少类完全 限定名的多余部分。

* 配置

<typeAliases>

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

</typeAliases>

* 1
* 2
* 3

使用这个配置, “Blog”可以任意用来替代“domain.blog. Blog”所使用的地方。

* 源代码

//Configuration类中

protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();

//TypeAliasRegistry中通过一个map保存这个关系

private final Map<String, Class<?>> TYPE\_ALIASES = new HashMap<String, Class<?>>();

//XMLConfigBuilder类中

private void typeAliasesElement(XNode parent) {

if (parent != null) {

for (XNode child : parent.getChildren()) {

if ("package".equals(child.getName())) {

String typeAliasPackage = child.getStringAttribute("name");

//package模式

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);

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

无论是package的情况还是直接在mybatis-config.xml配置，他们都统一调用了TypeAliasRegistry类来完成注册。在 TypeAliasRegistry类中通过TYPE\_ALIASES 来保存这个别名关系。

六、pluginElement

插件–允许你在某一点拦截已映射语句执行的调用。例如：[MyBatis 入门(六)–分页查询（2） －插件方式](http://blog.csdn.net/likewindy/article/details/51351221)   
例如配置文件：

* 配置

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

<plugins>

<plugin interceptor="org.mybatis.example.ExamplePlugin">

<property name="someProperty" value="100"/>

</plugin>

</plugins>

* 1
* 2
* 3
* 4
* 5
* 6

// ExampleTypeHandler.java

@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);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 源代码

//Configuration类中 拦截器责任链

protected final InterceptorChain interceptorChain = new InterceptorChain();

// InterceptorChain中有一个list保存注册好的Interceptor

private final List<Interceptor> interceptors = new ArrayList<Interceptor>();

//XMLConfigBuilder类中

private void pluginElement(XNode parent) throws Exception {

if (parent != null) {

for (XNode child : parent.getChildren()) {

//

String interceptor = child.getStringAttribute("interceptor");

Properties properties = child.getChildrenAsProperties();

//获取类

Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();

//注册到拦截器链中

interceptorInstance.setProperties(properties);

//这里向Configuration类的interceptorChain 注册

configuration.addInterceptor(interceptorInstance);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

在Configuration类中定义了一个 拦截器的链protected final InterceptorChain interceptorChain = new InterceptorChain();熟悉责任链模式的应该应该不陌生。在这里可以把我们实现Interceptor接口的类，注册到这里链条中。

Interceptor 的接口：

public interface Interceptor {

Object intercept(Invocation invocation) throws Throwable;

Object plugin(Object target);

void setProperties(Properties properties);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

七、objectFactoryElement

Mybatis中有一个默认的创建类的对象，名字叫做DefaultObjectFactory，这个类用于负责创建对象实体类。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。 默认的对象工厂需要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。   
- 配置

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

<objectFactory type="org.mybatis.example.ExampleObjectFactory">

<property name="someProperty" value="100"/>

</objectFactory>

* 1
* 2
* 3
* 4

// ExampleObjectFactory.java

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);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

源代码：

//Configuration类中

protected ObjectFactory objectFactory = new DefaultObjectFactory();

//XMLConfigBuilder类中

private void objectFactoryElement(XNode context) throws Exception {

if (context != null) {

String type = context.getStringAttribute("type");

Properties properties = context.getChildrenAsProperties();

ObjectFactory factory = (ObjectFactory) resolveClass(type).newInstance();

factory.setProperties(properties);

configuration.setObjectFactory(factory);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

ObjectFactory 接口：

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.

\*

\* @since 3.1.0

\* @param type Object type

\* @return whether it is a collection or not

\*/

<T> boolean isCollection(Class<T> type);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36

这里可以参考 <http://blog.csdn.net/ziwen00/article/details/38385069>   
除非你由自己创建新对象及方法的特殊之处，可以重新构造次创建对象，否则一般的对象,mat,list等都使用mybatis的默认就已经足够。

八、 objectWrapperFactoryElement

这个部分化了笔者比较长的时间， 原因网络上的资料很少。直到笔者分析到mybatis的反射部分，动态获取和设置属性的值，才明白mybatis预留这个接口的作用。不过mybatis已经考虑基本类型，基本对象，map和集合的包装，除非你有比较特殊的对象需要自己实现，才有用到这个配置。

* 源代码

//Configuration类中

protected ObjectWrapperFactory objectWrapperFactory = new DefaultObjectWrapperFactory();

//XMLConfigBuilder类中

private void objectWrapperFactoryElement(XNode context) throws Exception {

if (context != null) {

String type = context.getStringAttribute("type");

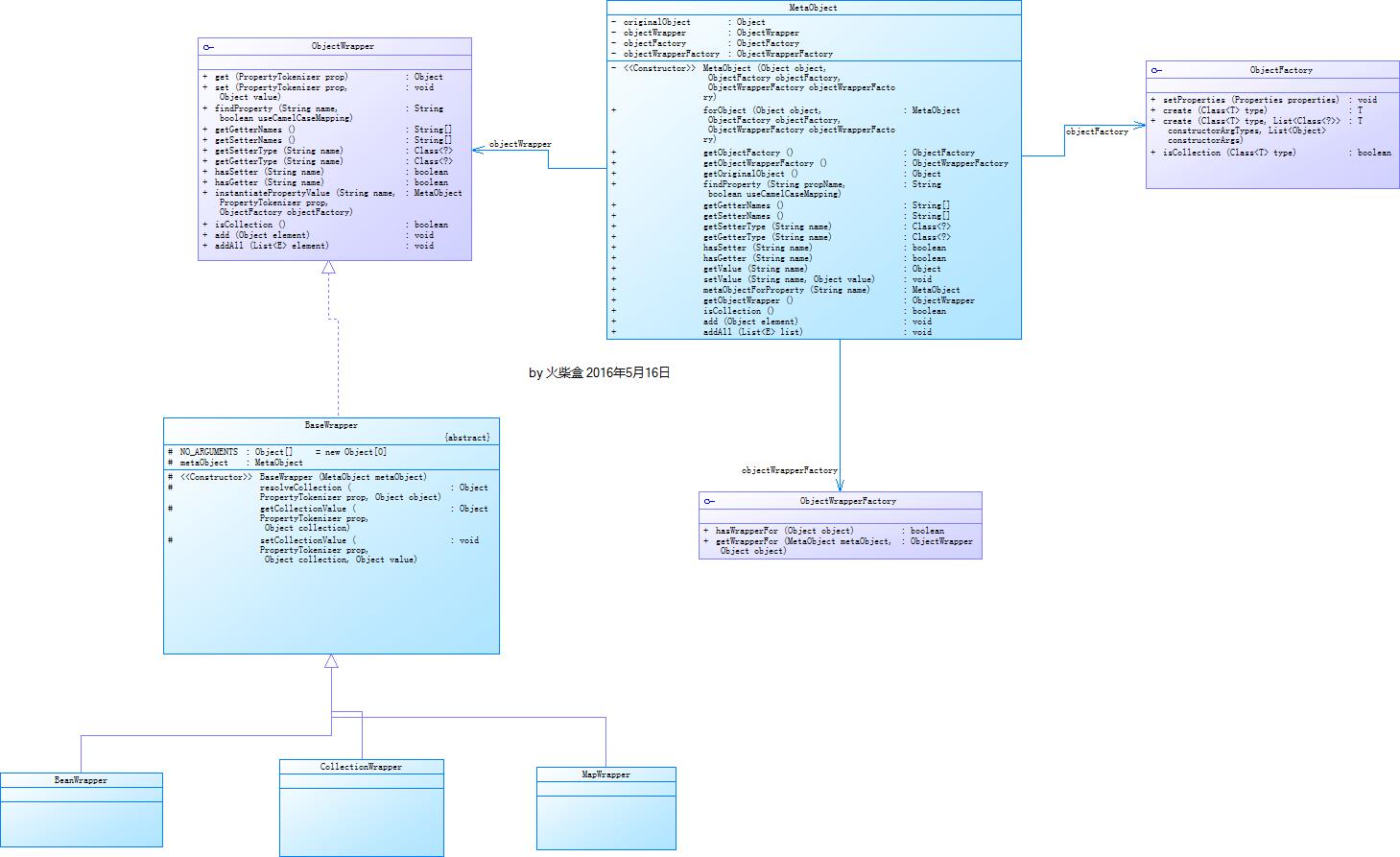
ObjectWrapperFactory factory = (ObjectWrapperFactory) resolveClass(type).newInstance();

configuration.setObjectWrapperFactory(factory);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

  
在这个图片的左半部分，我们可看到objectWrapper的基础情况。mybatis已经实现基本的对象 包装，这个类的作用是获取或设置bean，map或者list属性的值。

ObjectWrapperFactory 接口

public interface ObjectWrapperFactory {

boolean hasWrapperFor(Object object);

ObjectWrapper getWrapperFor(MetaObject metaObject, Object object);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

通过分析MetaObject的源代码

//构造函数，私有

private MetaObject(Object object, ObjectFactory objectFactory, ObjectWrapperFactory objectWrapperFactory) {

this.originalObject = object;

this.objectFactory = objectFactory;

this.objectWrapperFactory = objectWrapperFactory;

if (object instanceof ObjectWrapper) {

//如果参数对象实现了ObjectWrapper

this.objectWrapper = (ObjectWrapper) object;

} else if (objectWrapperFactory.hasWrapperFor(object)) {

//如果objectWrapperFactory已经包装了对象，对用objectWrapperFactory的getWrapperFor

this.objectWrapper = objectWrapperFactory.getWrapperFor(this, object);

} else if (object instanceof Map) {

//是一个Map对象，使用mybatis的MapWrapper

this.objectWrapper = new MapWrapper(this, (Map) object);

} else if (object instanceof Collection) {

//是一个CollectionWrapper对象

this.objectWrapper = new CollectionWrapper(this, (Collection) object);

} else {

//其他默认使用BeanWrapper

this.objectWrapper = new BeanWrapper(this, object);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

在这里mybatis预留两个接口对象，可以自己封装。

* 参数对象实现了ObjectWrapper接口
* 实现了ObjectWrapperFactory的接口，并且通过配置应用到了Configuration类中。   
  它有一个默认的实现类DefaultObjectWrapperFactory。

如果的某个参数对象object想要有用自己的 WrapperFactory来进行设置和获取属性的值就可以实现它的ObjectWrapperFactory。并且在 getWrapperFor(MetaObject metaObject, Object object);的方法中实现 ObjectWrapper 的接口。   
ObjectWrapper 接口

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();

public void add(Object element);

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

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29

九、settingsElement

这些是极其重要的调整, 它们会修改 MyBatis 在运行时的行为方式。一个设置信息元素的示例,完全的配置如下所示:

* 配置

<settings>

<setting name="cacheEnabled" value="true"/>

<setting name="lazyLoadingEnabled" value="true"/>

<setting name="multipleResultSetsEnabled" value="true"/>

<setting name="useColumnLabel" value="true"/>

<setting name="useGeneratedKeys" value="false"/>

<setting name="autoMappingBehavior" value="PARTIAL"/>

<setting name="defaultExecutorType" value="SIMPLE"/>

<setting name="defaultStatementTimeout" value="25"/>

<setting name="safeRowBoundsEnabled" value="false"/>

<setting name="mapUnderscoreToCamelCase" value="false"/>

<setting name="localCacheScope" value="SESSION"/>

<setting name="jdbcTypeForNull" value="OTHER"/>

<setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>

</settings>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 源代码

////XMLConfigBuilder类中

private void settingsElement(XNode context) throws Exception {

if (context != null) {

Properties props = context.getChildrenAsProperties();

// Check that all settings are known to the configuration class

MetaClass metaConfig = MetaClass.forClass(Configuration.class);

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).");

}

}

configuration.setAutoMappingBehavior(AutoMappingBehavior.valueOf(props.getProperty("autoMappingBehavior", "PARTIAL")));

configuration.setCacheEnabled(booleanValueOf(props.getProperty("cacheEnabled"), true));

configuration.setProxyFactory((ProxyFactory) createInstance(props.getProperty("proxyFactory")));

configuration.setLazyLoadingEnabled(booleanValueOf(props.getProperty("lazyLoadingEnabled"), false));

configuration.setAggressiveLazyLoading(booleanValueOf(props.getProperty("aggressiveLazyLoading"), true));

configuration.setMultipleResultSetsEnabled(booleanValueOf(props.getProperty("multipleResultSetsEnabled"), true));

configuration.setUseColumnLabel(booleanValueOf(props.getProperty("useColumnLabel"), true));

configuration.setUseGeneratedKeys(booleanValueOf(props.getProperty("useGeneratedKeys"), false));

configuration.setDefaultExecutorType(ExecutorType.valueOf(props.getProperty("defaultExecutorType", "SIMPLE")));

configuration.setDefaultStatementTimeout(integerValueOf(props.getProperty("defaultStatementTimeout"), null));

configuration.setMapUnderscoreToCamelCase(booleanValueOf(props.getProperty("mapUnderscoreToCamelCase"), false));

configuration.setSafeRowBoundsEnabled(booleanValueOf(props.getProperty("safeRowBoundsEnabled"), false));

configuration.setLocalCacheScope(LocalCacheScope.valueOf(props.getProperty("localCacheScope", "SESSION")));

configuration.setJdbcTypeForNull(JdbcType.valueOf(props.getProperty("jdbcTypeForNull", "OTHER")));

configuration.setLazyLoadTriggerMethods(stringSetValueOf(props.getProperty("lazyLoadTriggerMethods"), "equals,clone,hashCode,toString"));

configuration.setSafeResultHandlerEnabled(booleanValueOf(props.getProperty("safeResultHandlerEnabled"), true));

configuration.setDefaultScriptingLanguage(resolveClass(props.getProperty("defaultScriptingLanguage")));

configuration.setCallSettersOnNulls(booleanValueOf(props.getProperty("callSettersOnNulls"), false));

configuration.setLogPrefix(props.getProperty("logPrefix"));

configuration.setLogImpl(resolveClass(props.getProperty("logImpl")));

configuration.setConfigurationFactory(resolveClass(props.getProperty("configurationFactory")));

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34

十、 environmentsElement

MyBatis 可以配置多种环境。这会帮助你将 SQL 映射应用于多种数据库之中。例如, 你也许为开发要设置不同的配置, 测试和生产环境   
**一个很重要的问题要记得:你可以配置多种环境,但你只能为每个 SqlSessionFactory 实例选择一个。**

* environments 的配置文件

<environments default="dev">

<environment id="dev">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

<property name="driver" value="${driver}" />

<property name="url" value="${url}" />

<property name="username" value="${username}" />

<property name="password" value="${password}" />

</dataSource>

</environment>

</environments>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 读取数据库环境的函数

//Configuration类中

protected Environment environment;

//XMLConfigBuilder类中

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");

if (isSpecifiedEnvironment(id)) {

//读取 <transactionManager type="JDBC" />，并且创建TransactionFactory 工厂

TransactionFactory txFactory = transactionManagerElement(child.evalNode("transactionManager"));

//读取 <dataSource type="POOLED">

DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));

//得到数据源

DataSource dataSource = dsFactory.getDataSource();

Environment.Builder environmentBuilder = new Environment.Builder(id)

.transactionFactory(txFactory)

.dataSource(dataSource);

//保存到配置中

configuration.setEnvironment(environmentBuilder.build());

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27

在Environment类中申明了 三个重要变量

private final String id;

private final TransactionFactory transactionFactory;

private final DataSource dataSource;

* 1
* 2
* 3
* TransactionFactory 是我们的事务处理对象。

public interface TransactionFactory {

/\*\*

\* Sets transaction factory custom properties.

\* @param props

\*/

void setProperties(Properties props);

/\*\*

\* Creates a {@link Transaction} out of an existing connection.

\* @param conn Existing database connection

\* @return Transaction

\* @since 3.1.0

\*/

Transaction newTransaction(Connection conn);

/\*\*

\* Creates a {@link Transaction} out of a datasource.

\* @param dataSource DataSource to take the connection from

\* @param level Desired isolation level

\* @param autoCommit Desired autocommit

\* @return Transaction

\* @since 3.1.0

\*/

Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, boolean autoCommit);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* DataSource使我们数据库连接的对象。

public interface DataSource extends CommonDataSource,Wrapper {

/\*\*

\* <p>Attempts to establish a connection with the data source that

\* this <code>DataSource</code> object represents.

\*

\* @return a connection to the data source

\* @exception SQLException if a database access error occurs

\*/

Connection getConnection() throws SQLException;

/\*\*

\* <p>Attempts to establish a connection with the data source that

\* this <code>DataSource</code> object represents.

\*

\* @param username the database user on whose behalf the connection is

\* being made

\* @param password the user's password

\* @return a connection to the data source

\* @exception SQLException if a database access error occurs

\* @since 1.4

\*/

Connection getConnection(String username, String password)

throws SQLException;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

十一、databaseIdProviderElement

解释及实际应用场景[MyBatis之databaseIdProvider](http://blog.csdn.net/likewindy/article/details/51396576)   
- 源代码

//XMLConfigBuilder类中

private void databaseIdProviderElement(XNode context) throws Exception {

DatabaseIdProvider databaseIdProvider = null;

if (context != null) {//有配置数据库供应商的情况

String type = context.getStringAttribute("type");

if ("VENDOR".equals(type)) type = "DB\_VENDOR"; // awful patch to keep backward compatibility

Properties properties = context.getChildrenAsProperties();

databaseIdProvider = (DatabaseIdProvider) resolveClass(type).newInstance();

databaseIdProvider.setProperties(properties);

}

Environment environment = configuration.getEnvironment();

if (environment != null && databaseIdProvider != null) {

String databaseId = databaseIdProvider.getDatabaseId(environment.getDataSource());

configuration.setDatabaseId(databaseId);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

十二、typeHandlerElement

无论是 MyBatis 在预处理语句中设置一个参数, 还是从结果集中取出一个值时, 类型处理器被用来将获取的值以合适的方式转换成 Java 类型。你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。例如：[MyBatis 入门(五)–typeHandlers](http://blog.csdn.net/likewindy/article/details/51350995)

* 配置

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

<typeHandlers>

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

</typeHandlers>

* 1
* 2
* 3
* 4

// ExampleTypeHandler.java

@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);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 源代码:

//Configuration类中

protected final TypeHandlerRegistry typeHandlerRegistry = new TypeHandlerRegistry();

//XMLConfigBuilder类中

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 {

//java类型

String javaTypeName = child.getStringAttribute("javaType");

//jdbc类型

String jdbcTypeName = child.getStringAttribute("jdbcType");

//处理类

String handlerTypeName = child.getStringAttribute("handler");

Class<?> javaTypeClass = resolveClass(javaTypeName);

JdbcType jdbcType = resolveJdbcType(jdbcTypeName);

Class<?> typeHandlerClass = resolveClass(handlerTypeName);

//注册typeHandler

if (javaTypeClass != null) {

if (jdbcType == null) {

typeHandlerRegistry.register(javaTypeClass, typeHandlerClass);

} else {

typeHandlerRegistry.register(javaTypeClass, jdbcType, typeHandlerClass);

}

} else {

typeHandlerRegistry.register(typeHandlerClass);

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 扩展类TypeHandlerRegistry

private final Map<JdbcType, TypeHandler<?>> JDBC\_TYPE\_HANDLER\_MAP = new EnumMap<JdbcType, TypeHandler<?>>(JdbcType.class);

private final Map<Type, Map<JdbcType, TypeHandler<?>>> TYPE\_HANDLER\_MAP = new HashMap<Type, Map<JdbcType, TypeHandler<?>>>();

private final TypeHandler<Object> UNKNOWN\_TYPE\_HANDLER = new UnknownTypeHandler(this);

private final Map<Class<?>, TypeHandler<?>> ALL\_TYPE\_HANDLERS\_MAP = new HashMap<Class<?>, TypeHandler<?>>();

private void register(Type javaType, JdbcType jdbcType, TypeHandler<?> handler) {

if (javaType != null) {

Map<JdbcType, TypeHandler<?>> map = TYPE\_HANDLER\_MAP.get(javaType);

if (map == null) {//该javaType类型没有注册

map = new HashMap<JdbcType, TypeHandler<?>>();

TYPE\_HANDLER\_MAP.put(javaType, map);

}

map.put(jdbcType, handler);

if (reversePrimitiveMap.containsKey(javaType)) {

register(reversePrimitiveMap.get(javaType), jdbcType, handler);

}

}

ALL\_TYPE\_HANDLERS\_MAP.put(handler.getClass(), handler);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

十三、mapperElement

定义 SQL 映射语句了。 但是, 首先我们需要告诉 MyBatis 到哪里去找到这些语句。 Java 在这方面没有提供一个很好 的方法, 所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。 你可以使用相对于类路径的 资源引用,或者字符表示,或 url 引用的完全限定名(包括 file:///URLs) 。例如:

* 配置

<!-- Using classpath relative resources -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

<!-- Using url fully qualified paths -->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

<!-- Using mapper interface classes -->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

<!-- Register all interfaces in a package as mappers -->

<mappers>

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

</mappers>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 源代码

//已经加载过的

protected final Set<String> loadedResources = new HashSet<String>();

//XMLConfigBuilder类中

private void mapperElement(XNode parent) throws Exception {

if (parent != null) {

for (XNode child : parent.getChildren()) {

if ("package".equals(child.getName())) {

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

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) {

//resource 模式<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

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) {

//url模式 <mapper url="file:///var/mappers/AuthorMapper.xml"/>

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) {

// 接口模式<mapper class="org.mybatis.builder.AuthorMapper"/>

Class<?> mapperInterface = Resources.classForName(mapperClass);

configuration.addMapper(mapperInterface);

} else {

throw new BuilderException("A mapper element may only specify a url, resource or class, but not more than one.");

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39

这里有好几种注册模式，我们简单看看resource 模式的 XMLMapperBuilder处理类。有兴趣的可以看看其他模式 接口模式（接口模式把接口都注册到了Configuration类中   
protected MapperRegistry mapperRegistry = new MapperRegistry(this);中   
）和package模式

-14. 1 XMLMapperBuilder

一个简单的配置文件UserMapper.xml

<mapper namespace="com.elements.user.mapper">

<resultMap type="com.elements.user.model.User" id="usermap">

<id column="userId" property="userId"/>

<result column="UserName" property="UserName"/>

<result column="UserEmail" property="UserEmail"/>

</resultMap>

<select id="getAllUser" resultMap="usermap">

select \* from mybatis.user

</select>

<select id="getUserById" resultMap="usermap" parameterType="String">

select \* from mybatis.user where userId= #{userId}

</select>

<insert id="insert" parameterType="com.elements.user.model.User">

insert into mybatis.user (UserName, UserEmail) values (

#{UserName}, #{UserEmail}

)

</insert>

<update id="update" parameterType="com.elements.user.model.User">

update mybatis.user set UserName=#{UserName},

UserEmail=#{UserEmail}

where userId= #{userId}

</update>

<delete id="delete" parameterType="String">

delete mybatis.user where userId= #{userId}

</delete>

</mapper>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 14.1.1在XMLMapperBuilder的parse方法中

//Configuration类中

//已经解析过的资源

protected final Set<String> loadedResources = new HashSet<String>();

//每一个resultMap ，id+resultMap 对象

protected final Map<String, ResultMap> resultMaps = new StrictMap<ResultMap>("Result Maps collection");

//XMLMapperBuilder

public void parse() {

//先看看资源是否已经被解析过了

if (!configuration.isResourceLoaded(resource)) {

//这里解析我们上面的UserMapper.xml

configurationElement(parser.evalNode("/mapper"));

configuration.addLoadedResource(resource);

bindMapperForNamespace();

}

parsePendingResultMaps();

parsePendingChacheRefs();

parsePendingStatements();

}

//解析一个UserMapper.xml文件

private void configurationElement(XNode context) {

try {

//获取命名空间 namespace="com.elements.user.mapper"

String namespace = context.getStringAttribute("namespace");

if (namespace.equals("")) {

throw new BuilderException("Mapper's namespace cannot be empty");

}

builderAssistant.setCurrentNamespace(namespace);

//缓存的解析，这里我们先忽略

cacheRefElement(context.evalNode("cache-ref"));

cacheElement(context.evalNode("cache"));

//参数的解析，我们的配置文件没有也先忽略

parameterMapElement(context.evalNodes("/mapper/parameterMap"));

//返回的map定义，这是一个经常使用的，它主要解析我们上面对应的地方

/\*

<resultMap type="com.elements.user.model.User" id="usermap">

<id column="userId" property="userId"/>

<result column="UserName" property="UserName"/>

<result column="UserEmail" property="UserEmail"/>

</resultMap>

\*/

resultMapElements(context.evalNodes("/mapper/resultMap"));

sqlElement(context.evalNodes("/mapper/sql"));

//下面这个主要解析我们写sql的地方

/\* <select id="getAllUser" resultMap="usermap">

select \* from mybatis.user

</select>\*/

buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

} catch (Exception e) {

throw new BuilderException("Error parsing Mapper XML. Cause: " + e, e);

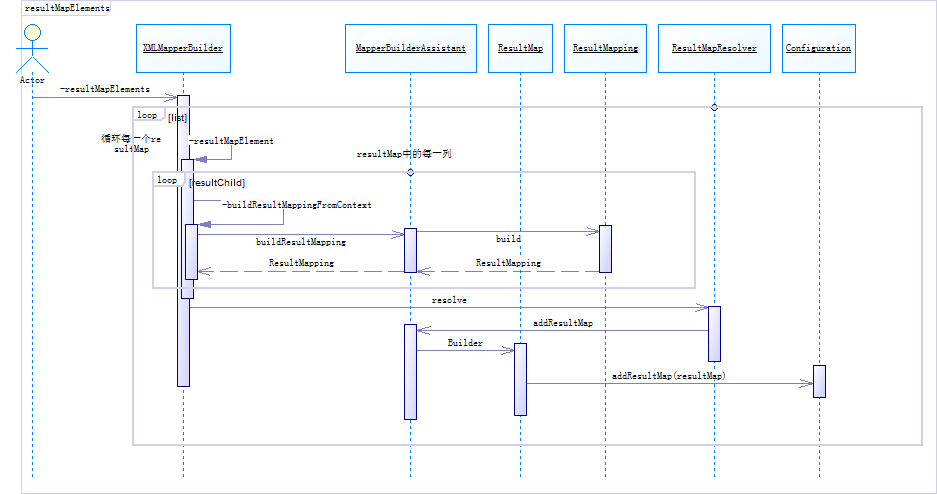
}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58

在这里我们主要分析 resultMapElements和buildStatementFromContext，其他我们暂时先忽略。

* 14.1.2 resultMapElements方法

  
通过上面的顺序图，我们可以看到XMLMapperBuilder读取XML配置的resultMap，交给MapperBuilderAssistant构建每一列的ResultMapping，再通过ResultMapResolver类（处理的还是MapperBuilderAssistant），构建出一个ResultMap类，ResultMap的属性见下面代码。最后把这个ResultMap注册到Configuration类中。

ResultMap类的主要属性

private String id;//命名空间+id的唯一代码

private Class<?> type;//java pojo类

private List<ResultMapping> resultMappings;//resultMap中定义的每一列，都在这里

private List<ResultMapping> idResultMappings;//resultMap中id列

private List<ResultMapping> constructorResultMappings;

private List<ResultMapping> propertyResultMappings;//resultMap中的property

private Set<String> mappedColumns;//resultMap中的column列

private Discriminator discriminator;

private boolean hasNestedResultMaps;

private boolean hasNestedQueries;

private Boolean autoMapping;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

ResultMapping类主要属性，这里是resultMap中定义的每一列

private Configuration configuration;

private String property;//java 属性

private String column; //数据库列

private Class<?> javaType;//java 类型

private JdbcType jdbcType; //数据类型

private TypeHandler<?> typeHandler;//类型处理

private String nestedResultMapId;

private String nestedQueryId;

private Set<String> notNullColumns;

private String columnPrefix;

private List<ResultFlag> flags;

private List<ResultMapping> composites;

private String resultSet;

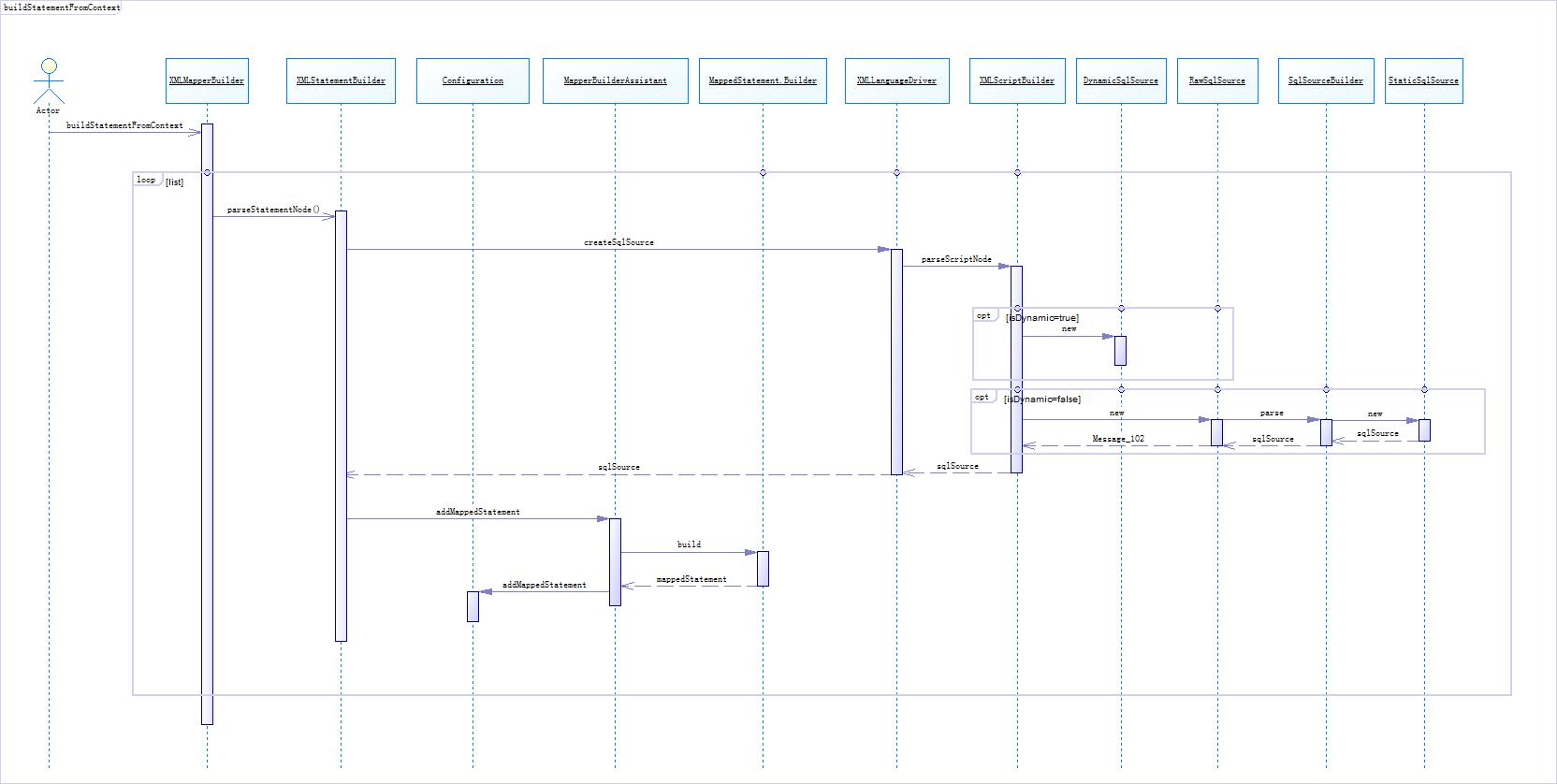
private String foreignColumn;

private boolean lazy;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 14.1.3 buildStatementFromContext

buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

* 1

顺序图   


处理UserMapper.xml中的查询，新增，修改和删除

private void buildStatementFromContext(List<XNode> list) {

if (configuration.getDatabaseId() != null) {

buildStatementFromContext(list, configuration.getDatabaseId());

}

buildStatementFromContext(list, null);

}

* 1
* 2
* 3
* 4
* 5
* 6

在这里我们看到他把XML中的查询、新增、修改和删除的语句，都保持到了MappedStatement对象中，这些执行的语句同时都注册到了Configuration中 。   
在MappedStatement类的属性有：

private String resource;//com/elements/user/mapper/UserMapper.xml

private Configuration configuration;

private String id;//命名空间+id

private Integer fetchSize;

private Integer timeout;

private StatementType statementType;//PREPARED，CALLABE等

private ResultSetType resultSetType;

private SqlSource sqlSource;//这是另外一个比较重要的属性

private Cache cache;

private ParameterMap parameterMap;//参数

private List<ResultMap> resultMaps;//返回的

private boolean flushCacheRequired;

private boolean useCache;

private boolean resultOrdered;

private SqlCommandType sqlCommandType;//执行语句类型：SELECT等

private KeyGenerator keyGenerator;

private String[] keyProperties;

private String[] keyColumns;

private boolean hasNestedResultMaps;

private String databaseId;

private Log statementLog;

private LanguageDriver lang;

private String[] resultSets;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

SqlSource 接口

public interface SqlSource {

BoundSql getBoundSql(Object parameterObject);

}

* 1
* 2
* 3
* 4
* 5

StaticSqlSource属性

private String sql;//替换后的sql语句。例如：select \* from mybatis.user where userId= ?

private List<ParameterMapping> parameterMappings;//替换的参数对象

private Configuration configuration;

* 1
* 2
* 3

ParameterMapping参数对象

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;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

十四、DefaultSqlSessionFactory

最后我们把已经配置好的Configuration类传递给了DefaultSqlSessionFactory的构造函数，此后DefaultSqlSessionFactory可以读取已经配置好的Configuration类。至此我们的SqlSessionFactory已经创建完成。

**注意事项： 摘录官方文档**   
SqlSessionFactoryBuilder   
这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳范围是方法范围（也就是局部方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在以保证所有的 XML 解析资源开放给更重要的事情。

SqlSessionFactory   
SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳范围是应用范围。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

版权声明：FREE