**MyBatis Generator代码分析一**

[](http://www.jianshu.com/u/231b43e2c05f)

[小码哥Java学院](http://www.jianshu.com/u/231b43e2c05f) 关注

2015.09.05 17:50 字数 2410 阅读 1549评论 0喜欢 5

**注意：以下代码都有适当修改和删改，为了更好看清楚执行流程**

首先是简单分析使用Shell Runner执行MBG的最概略的执行流程分析：

**org.mybatis.generator.api.ShellRunner：运行MyBatis Generator 的Main入口类；**

核心代码（main方法）：

//解析命令行

Map<String, String> arguments = parseCommandLine(args);

//创建一个警告列表，整个MBG运行过程中的所有警告信息都放在这个列表中，执行完成后统一System.out

List<String> warnings = new ArrayList<String>();

//得到generatorConfig.xml文件

String configfile = arguments.get(CONFIG\_FILE);

File configurationFile = new File(configfile);

Set<String> fullyqualifiedTables = new HashSet<String>();//如果参数有tables，得到table名称列表

Set<String> contexts = new HashSet<String>();//如果参数有contextids，得到context名称列表

try {

//创建配置解析器

ConfigurationParser cp = new ConfigurationParser(warnings);

//调用配置解析器创建配置对象（Configuration对象非常简单，可以简单理解为包含两个列表，一个列表是List<Context> contexts，包含了解析出来的Context对象，一个是List<String> classPathEntries，包含了配置的classPathEntry的location值）

Configuration config = cp.parseConfiguration(configurationFile);

//创建一个默认的ShellCallback对象，之前说过，shellcallback接口主要用来处理文件的创建和合并，传入overwrite参数；默认的shellcallback是不支持文件合并的；

DefaultShellCallback shellCallback = new DefaultShellCallback(

arguments.containsKey(OVERWRITE));

//创建一个MyBatisGenerator对象。MyBatisGenerator类是真正用来执行生成动作的类

MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, shellCallback, warnings);

//创建一个默认的ProgressCallback对象，之前说过，在MBG执行过程中在一定的执行步骤结束后调用ProgressCallback对象的方法，达到执行过程监控的效果；

//如果在执行ShellRunner是传入了-verbose参数，那么创建一个VerboseProgressCallback（VerboseProgressCallback只是调用了System.out打印出了执行过程而已）

ProgressCallback progressCallback = arguments.containsKey(VERBOSE) ? new VerboseProgressCallback()

: null;

//执行真正的MBG创建过程

//注意，这里的contexts是通过-contextids传入的需要的上下文id列表；

//fullyqualifiedTables是通过-tables传入的本次需要生成的table名称列表；

myBatisGenerator.generate(progressCallback, contexts, fullyqualifiedTables);

}catch(...){...}

//输出警告信息

for (String warning : warnings) {

writeLine(warning);

}

**org.mybatis.generator.config.xml.ConfigurationParser：配置解析器，用于对generatorConfig.xml配置文件的解析；**

构造方法：

//初始化配置解析器中的一些基本数据内容

public ConfigurationParser(Properties properties, List<String> warnings) {

super();

if (properties == null) {

//properties：存放的系统配置信息

this.properties = System.getProperties();

} else {

this.properties = properties;

}

if (warnings == null) {

//warnings：存放的解析中的警告信息

this.warnings = new ArrayList<String>();

} else {

this.warnings = warnings;

}

//parseErrors ：存放的解析中的错误信息

parseErrors = new ArrayList<String>();

}

//执行配置解析，创建配置对象

private Configuration parseConfiguration(InputSource inputSource)

throws IOException, XMLParserException {

parseErrors.clear();

//使用DOM解析器解析XML

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

factory.setValidating(true);

DocumentBuilder builder = factory.newDocumentBuilder();

//设置实体对象处理器（对于MyBatis3来说，就是处理org/mybatis/generator/config/xml/mybatis-generator-config\_1\_0.dtd验证），

builder.setEntityResolver(new ParserEntityResolver());

//设置解析错误处理器，把解析过程中的异常和警告保存到warnings和parseErrors两个String列表中；

ParserErrorHandler handler = new ParserErrorHandler(warnings,

parseErrors);

builder.setErrorHandler(handler);

//得到配置文件对应的DOM对象；

Document document = builder.parse(inputSource);

//配置对象；

Configuration config;

Element rootNode = document.getDocumentElement();

//得到XML文件的xml描述符；

DocumentType docType = document.getDoctype();

if (rootNode.getNodeType() == Node.ELEMENT\_NODE

&& docType.getPublicId().equals(XmlConstants.IBATOR\_CONFIG\_PUBLIC\_ID)) {

//如果xml的PUBLIC\_ID为-//Apache Software Foundation//DTD Apache iBATIS Ibator Configuration 1.0//EN，则执行解析ibatis过程；

config = parseIbatorConfiguration(rootNode);

} else if (rootNode.getNodeType() == Node.ELEMENT\_NODE

&& docType.getPublicId().equals(XmlConstants.MYBATIS\_GENERATOR\_CONFIG\_PUBLIC\_ID)) {

//如果xml的PUBLIC\_ID为-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN，则执行解析mybatis过程；

config = parseMyBatisGeneratorConfiguration(rootNode);

}

//返回解析出的Configuration对象

return config;

}

//执行MyBatis生成器的配置

private Configuration parseMyBatisGeneratorConfiguration(Element rootNode)

throws XMLParserException {

//创建一个MyBatisGeneratorConfigurationParser

MyBatisGeneratorConfigurationParser parser = new MyBatisGeneratorConfigurationParser(

properties);

//使用配置解析器执行XML解析

return parser.parseConfiguration(rootNode);

}

**MyBatisGeneratorConfigurationParser ：用于把MBG配置文件解析为MyBatis3需要样式；**

public Configuration parseConfiguration(Element rootNode)

throws XMLParserException {

//创建一个新的配置对象

Configuration configuration = new Configuration();

//得到<generatorConfiguration>下的所有元素，并遍历

NodeList nodeList = rootNode.getChildNodes();

for (int i = 0; i < nodeList.getLength(); i++) {

Node childNode = nodeList.item(i);

if (childNode.getNodeType() != Node.ELEMENT\_NODE) {

continue;

}

if ("properties".equals(childNode.getNodeName())) { //$NON-NLS-1$

//如果是<properties>元素，执行properties解析

parseProperties(configuration, childNode);

} else if ("classPathEntry".equals(childNode.getNodeName())) { //$NON-NLS-1$

//如果是<classPathEntry>，执行classPathEntry解析

parseClassPathEntry(configuration, childNode);

} else if ("context".equals(childNode.getNodeName())) { //$NON-NLS-1$

//如果是<context>元素，执行context解析

parseContext(configuration, childNode);

}

}

return configuration;

}

//所以重点是三个方法：parseProperties/parseClassPathEntry/parseContext

//parseProperties方法最重要的就是加载指定的properties配置到properties中，【注意】，因为在<generatorConfiguration>元素中的<properties>元素最重要的就是用来替换在配置文件中所有的${key}占位符，所以，properties元素只需要在解析过程存在，所以可以看到properties属性是只需要在MyBatisGeneratorConfigurationParser中使用；

private void parseProperties(Configuration configuration, Node node)

throws XMLParserException {

//解析得到URL或者resource属性（两种配置的加载方式）

Properties attributes = parseAttributes(node);

String resource = attributes.getProperty("resource");

String url = attributes.getProperty("url");

//统一把resource/URL转成URL；

URL resourceUrl;

if (stringHasValue(resource)) {

resourceUrl = ObjectFactory.getResource(resource);

} else {

resourceUrl = new URL(url);

}

//从URL加载properties文件并载入；

InputStream inputStream = resourceUrl.openConnection()

.getInputStream();

properties.load(inputStream);

inputStream.close();

}

//上面是解析properties的方法，主要就是提供给这个方法使用：在配置文件中所有的属性值都先使用${}占位符去测试一下，如果是占位符，就把${}中的值作为key去properties中查找，把查找到的值作为属性真正的值返回；

private String parsePropertyTokens(String string) {

final String OPEN = "${"; //$NON-NLS-1$

final String CLOSE = "}"; //$NON-NLS-1$

//中间代码略，就是解析得到${}中的值，并去properties中查询；

return newString;

}

//解析classPathEntry元素，只是很简单的把所有的classPathEntry元素的location添加到配置对象的classpathEntry列表中

private void parseClassPathEntry(Configuration configuration, Node node) {

Properties attributes = parseAttributes(node);

configuration.addClasspathEntry(attributes.getProperty("location")); //$NON-NLS-1$

}

//最重要的，最复杂的，解析context元素

private void parseContext(Configuration configuration, Node node) {

//解析出context元素上的所有属性，并把所有属性放到一个properties中；

Properties attributes = parseAttributes(node);

/\*\*

\* 得到默认的生成对象的样式(ModeType是一个简单的枚举)

\* public enum ModelType {

HIERARCHICAL("hierarchical"),FLAT("flat"),CONDITIONAL("conditional");

}

ModelType的getModelType只是很简单的根据string返回对应的类型或者报错

\*/

ModelType mt = defaultModelType == null ? null : ModelType

.getModelType(defaultModelType);

//创建一个Context对象

Context context = new Context(mt);

//先添加到配置对象的context列表中，

configuration.addContext(context);

//再解析<context>子元素

NodeList nodeList = node.getChildNodes();

for (int i = 0; i < nodeList.getLength(); i++) {

Node childNode = nodeList.item(i);

if (childNode.getNodeType() != Node.ELEMENT\_NODE) {

continue;

}

//以下的内容就很模式化了，只是依次把context的不同子元素解析，并添加到Context对象中；所以我们就先不看每一个具体的解析代码，先看一下Context对象的结构；

if ("property".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseProperty(context, childNode);

} else if ("plugin".equals(childNode.getNodeName())) { //$NON-NLS-1$

parsePlugin(context, childNode);

} else if ("commentGenerator".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseCommentGenerator(context, childNode);

} else if ("jdbcConnection".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseJdbcConnection(context, childNode);

} else if ("javaModelGenerator".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseJavaModelGenerator(context, childNode);

} else if ("javaTypeResolver".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseJavaTypeResolver(context, childNode);

} else if ("sqlMapGenerator".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseSqlMapGenerator(context, childNode);

} else if ("javaClientGenerator".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseJavaClientGenerator(context, childNode);

} else if ("table".equals(childNode.getNodeName())) { //$NON-NLS-1$

parseTable(context, childNode);

}

}

}

**org.mybatis.generator.config.Context：封装<context>元素内容**

public class Context extends PropertyHolder {

/\*\* context的id \*/

private String id;

/\*\* jdbc连接配置，包装成JDBCConnectionConfiguration 对象，对应<jdbcConnection>元素 \*/

private JDBCConnectionConfiguration jdbcConnectionConfiguration;

/\*\* 生成SQL MAP的xml配置，对应<sqlMapGenerator>元素，包装成 SqlMapGeneratorConfiguration 对象\*/

private SqlMapGeneratorConfiguration sqlMapGeneratorConfiguration;

/\*\* 生成java类型处理器配置，对应<javaTypeResolver>元素，包装成 JavaTypeResolverConfiguration 对象 \*/

private JavaTypeResolverConfiguration javaTypeResolverConfiguration;

/\*\* 生成java模型创建器配置，对应<javaModelGenerator>元素，包装成 JavaModelGeneratorConfiguration 对象 \*/

private JavaModelGeneratorConfiguration javaModelGeneratorConfiguration;

/\*\* 生成Mapper接口配置，对应<javaClientGenerator>元素，包装成 JavaClientGeneratorConfiguration 对象\*/

private JavaClientGeneratorConfiguration javaClientGeneratorConfiguration;

/\*\* 解析每一个<table>元素，并包装成一个一个的TableConfiguration对象 \*/

private ArrayList<TableConfiguration> tableConfigurations;

/\*\* 生成对象样式，对应context元素的defaultModelType属性(attribute) \*/

private ModelType defaultModelType;

/\*\* 对应context元素的beginningDelimiter这个property子元素（注意属性和property的区别） \*/

private String beginningDelimiter = "\"";

/\*\* 对应context元素的endingDelimiter 这个property子元素\*/

private String endingDelimiter = "\"";

/\*\* 对应<commentGenerator>元素，注解生成器的配置 \*/

private CommentGeneratorConfiguration commentGeneratorConfiguration;

/\*\* 注解生成器 \*/

private CommentGenerator commentGenerator;

/\*\* 这是一个包装了所有的plugin的插件执行对象，其中的插件就是由pluginConfigurations中的每一个PluginConfiguration生成\*/

private PluginAggregator pluginAggregator;

/\*\* 对应每一个<plugin>元素的配置 \*/

private List<PluginConfiguration> pluginConfigurations;

/\*\* 目标运行时，对应context元素的targetRuntime属性(attribute) \*/

private String targetRuntime;

/\*\* 对应context元素的introspectedColumnImpl属性(attribute) \*/

private String introspectedColumnImpl;

/\*\* 自动识别数据库关键字，对应context元素的autoDelimitKeywords这个property子元素 \*/

private Boolean autoDelimitKeywords;

/\*\*Java代码格式化工具，对应context元素的javaFormatter这个property子元素 \*/

private JavaFormatter javaFormatter;

/\*\* Xml代码格式化工具，对应context元素的xmlFormatter这个property子元素 \*/

private XmlFormatter xmlFormatter;

}

可以看到，其实MBG的初始化过程是非常简单的，说白了，最重要的目的就是把generatorConfig.xml中的DOM通过MyBatisGeneratorConfigurationParser类解析成一个Configuration对象，而主要的工作就是消耗在把<context>元素解析成Configuration对象中的List<Context>，而<context>刚好对应着Context对象，那么，实际的生成过程，就是MyBatisGenerator对象根据Configuration对象来生成了。

待续...