# 编写Maven插件：原始方式

## 1.1 编写Maven插件的一般步骤：

1. 创建一个maven-plugin项目：插件本身也是Maven项目，特殊的地方在于它的packaging必须是maven-plugin，用户可以使用maven-archetype-plugin快速创建一个Maven插件项目。
2. 为插件编写目标：每个插件都必须包含一个或多个目标，Maven称之为Mojo（与POJO对应，后者指Plain Old Java Object，这里指Maven Old Java Object）。编写插件的时候必须提供一个或者多个继承自AbstractMojo的类。
3. 为目标提供配置点：大部分Maven插件及其目标都是可配置的，因此在编写Mojo的时候需要注意提供可配置的参数。
4. 编写代码实现目标行为：根据实际的需要实现Mojo。
5. 错误处理日志：当Mojo发生异常时，根据情况控制Maven的运行状态。在代码中编写必要的日志以便为用户提供足够的信息。
6. 测试插件：编写自动化的测试代码测试行为，然后再实际运行插件以验证其行为。

## 1.2 案例：编写一个用于代码行统计的Maven插件

本小节详细演示如何实际编写一个简单的用于代码行统计的Maven插件。使用该插件，用户可以了解到Maven项目中各个源代码目录下文件的数量，以及它们加起来共有多少代码行。

要创建一个Maven插件项目，首先使用maven-archetype-plugin骨架(模板)生成模板代码：

|  |
| --- |
| mvn archetype:generate -DarchetypeArtifactId=maven-archetype-plugin -DgroupId=com.chenming -DartifactId=Hello-plugin -Dversion=1.1-SNAPSHOT -DinteractiveMode=false -DarchetypeCatalog=internal |

执行上面的命令，就可以使用maven-archetype-plugin模板创建一个Maven插件工程。

打开项目的pom.xml文件，可以看到如下代码清单所示的内容：



Maven插件项目的POM有两个特殊的地方：

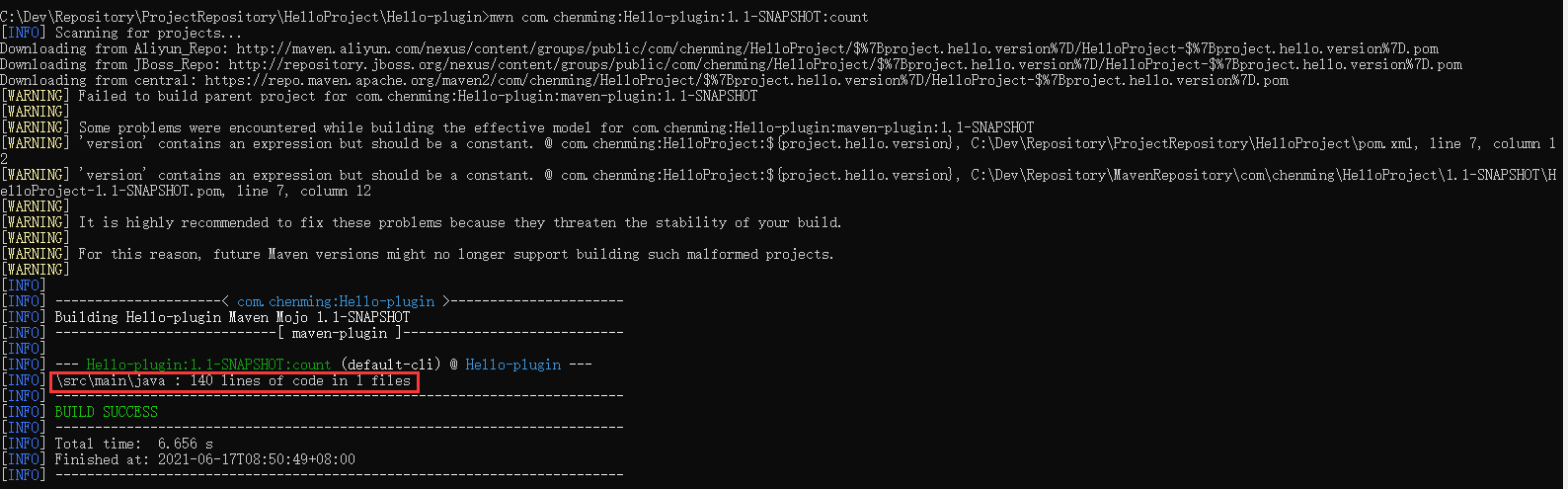
1. 它的packaging必须为maven-plugin，这种特殊的打包类型能控制Maven为其在生命周期阶段绑定插件处理相关的目标，例如在compile阶段，Maven需要为插件项目构建一个特殊插件描述符文件。
2. 从上述代码中可以看到一个artifactId为maven-plugin-api的依赖，该依赖中包含了插件开发所必须的类，例如稍后会看到的AbstractMojo。需要注意的是，上面的代码清单中并没有使用默认Archetype生成的maven-plugin-api版本，而是升级到了3.0，这样做的目的是与Maven的版本保持一致。

插件项目创建好之后，下一步是为插件编写目标。使用Archetype生成的插件项目包含了一个名为MyMojo的java文件，我们将其删除，然后自己创建一个CountMojo，如下面的代码清单所示：

|  |
| --- |
| public class CountMojo extends AbstractMojo {   private static final String[] *INCLUDES\_DEFAULT* = {"java","xml","properties"};   */\*\*  \* @parameter expression="${project.basedir}"  \* @required  \* @readonly  \*/* private File basedir;   */\*\*  \* @parameter expression="${project.build.sourceDirectory}"  \* @required  \* @readonly  \*/* private File sourceDirectory;   */\*\*  \* @parameter expression="${project.build.testSourceDirectory}"  \* @required  \* @readonly  \*/* private File testSourceDirectory;   */\*\*  \* @parameter expression="${project.build.resources}"  \* @required  \* @readonly  \*/* private List<Resource> resources;   */\*\*  \* @parameter expression="${project.build.testResources}"  \* @required  \* @readonly  \*/* private List<Resource> testResources;   */\*\*  \* the file type which will be included for counting  \*  \* @parameter  \*/* private String[] includes;   @Override  public void execute() throws MojoExecutionException, MojoFailureException {  if (includes == null || includes.length == 0) {  includes = *INCLUDES\_DEFAULT*;  }  try {  countDir(sourceDirectory);  countDir(testSourceDirectory);  for (Resource resource : resources) {  countDir(new File(resource.getDirectory()));  }  for (Resource resource : testResources) {  countDir(new File(resource.getDirectory()));  }  } catch (IOException e) {  throw new RuntimeException("Unable to count lines of code",e);  }  }   private void countDir(File dir) throws IOException {  if (!dir.exists()) {  return;  }  List<File> collected = new ArrayList<>();  collectedFiles(collected,dir);  int lines = 0;  for (File sourceFile : collected) {  lines += countLine(sourceFile);  }  }   private void collectedFiles(List<File> collected,File file) {  if (file.isFile()) {  for (String include : includes) {  if (file.getName().endsWith("." + include)) {  collected.add(file);  break;  }  }  } else {  for (File sub : file.listFiles()) {  collectedFiles(collected,sub);  }  }  }   private int countLine(File file) throws IOException {  BufferedReader bufferedReader = new BufferedReader(new FileReader(file));  int lines = 0;  try {  while (bufferedReader.ready()) {  bufferedReader.readLine();  lines++;  }  } finally {  bufferedReader.close();  }  return lines;  } } |

这里简单解释一下：

1. CountMojo统计Java和SQL文件，而不是默认的Java、XML和Properties。
2. 代码中还包含了basedir、sourceDirectory、testSourceDirectory等字段，它们都是用了@parameter标注，但同时关键字expression表示从系统属性读取这几个字段的值。${project.build.testSourceDirectory}，${project.build.sourceDirectory}，${project.basedir}等表达式读者应该已经很熟悉了，它们分别代表了项目的基础目录、主代码目录和测试代码目录。@readonly标注表示不允许用户对其进行配置，因为对于一个项目来说，这几个目录位置都是固定的。
3. execute方法中大家可以看到这样一些信息：如果用户没有配置includes则就是默认的统计包含配置，然后再分别统计、测试代码目录、主资源目录、以及测试资源目录。这里涉及一个countDir()方法。
4. collectFiles()方法用来递归的收集一个目录下所有应当被统计的文件，countLine()方法用来统计单个文件的行数，而countDir()则借助上述两个方法来统计某个目录下共有多少文件被统计，以及这些文件共包含多少代码行。
5. execute()方法包含了简单的异常处理，代码行统计的时候由于涉及了文件操作，因此可能会抛出IOException。当捕获到IOException的时候，使用MojoExecutionException对其简单包装后再抛出，Maven插件执行目标的时候如果遇到MojoExecutionException，就会在命令行显示“BUILD ERROR”的信息。
6. 使用mvn clean install命令将该插件目标构建并安装到本地仓库后，就能使用它统计Maven项目的代码行了。如下所示：



如果嫌命令行太长太复杂，可以执行两步：配置插件组和配置插件前缀。配置完这两步之后，就可以直接使用 **前缀:目标** 的方式方便执行，如compiler:compile

* 配置插件组：

在**settings.xml**文件中(可以是全局配置文件或者用户配置文件)，配置插件组：

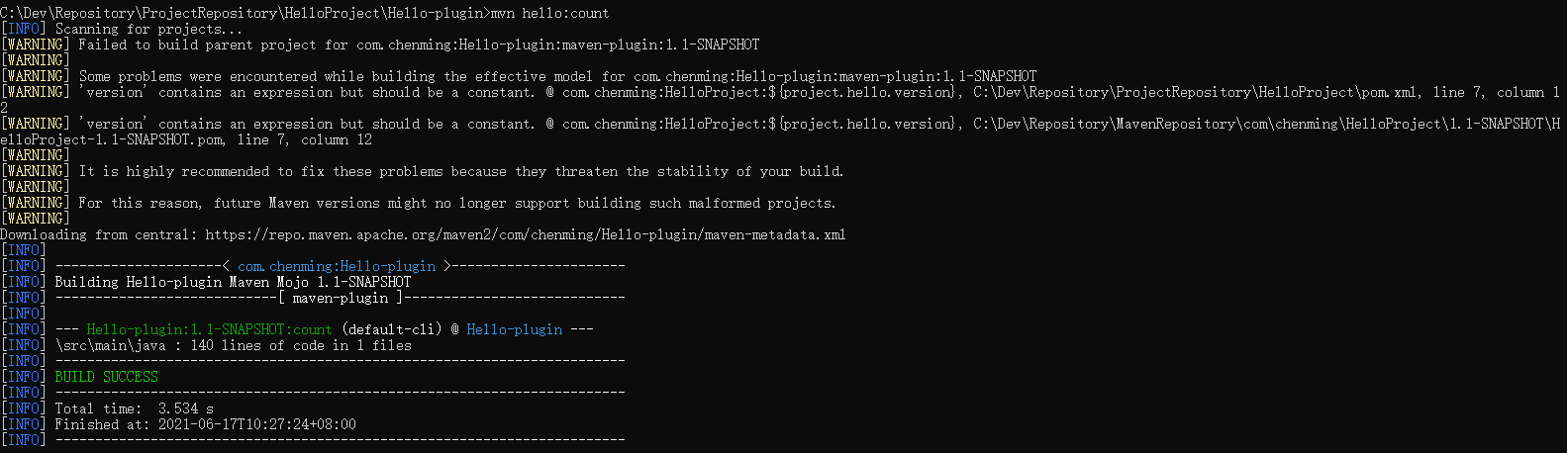
|  |
| --- |
| <settings>  <pluginGroups>  <pluginGroup>com.chenming</pluginGroup>  </pluginGroups>  <settings> |

* 配置Hello-plugin插件的前缀：

在插件项目的 pom.xml 中配置前缀：

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-plugin-plugin</artifactId>  <version>3.2</version>  <configuration>  <goalPrefix>hello</goalPrefix>  <skipErrorNoDescriptorsFound>true</skipErrorNoDescriptorsFound>  </configuration>  </plugin>  </plugins> </build> |

* 运行：



## 1.3 Mojo标注：

每个Mojo都必须使用@Goal标注来注明其目标名称，否则Maven将无法识别该目标。Mojo的标注不仅限于@Goal，以下是一些可以用来控制Mojo行为的标注：

### 1.3.1 @goal <name>：

这是唯一必须声明的标注，当用户使用命令行调用插件，或者在POM中配置插件的时候，都需要使用该目标名称。

### 1.3.2 @phase <phase>：

默认将目标绑定到生命周期的某个阶段，这样在配置使用该插件目标的时候就不需要声明phase。例如：maven-surefire-plugin的test目标就带有@phase test标注。

### 1.3.3 @requiresDependencyResolution <scope>：

表示在运行该Mojo之前必须解析所有指定范围的依赖。如：maven-surefire-plugin的test目标带有@requiresDependencyResolution test标注，表示在执行测试之前，所有测试范围的依赖必须得到解析。这里可用的依赖范围有compile、test和runtime，默认值为runtime。

### 1.3.4 @requiresProject <true/false>：

表示该目标是否必须在一个Maven项目中运行，默认为true。大部分插件目标必须依赖一个项目才能执行，但有一些例外，如maven-help-plugin的system目标，它用来显示系统属性和环境变量信息，不需要实际项目，因此使用了@requireProject false标注。另外，Maven的maven-archetype-plugin的generate也是一个很好的目标。

### 1.3.5 @requiresDirectInvocation <true/false>：

当值为true的时候，该目标就只能通过命令行直接调用，如果试图在POM中将其绑定到生命周期阶段，Maven就会报错，默认值为false。如果你希望编写的插件只能在命令行独立运行，就应该使用标注。

### 1.3.6 @requiresOnline <true/false>：

表示否是要求Maven必须是在线状态，默认值是false

### 1.3.7 @requiresReport <true/false>：

表示是否要求项目报告已经生成，默认值时false

### 1.3.8 @aggregator：

当Mojo在多模块项目上运行时，使用该标注表示该目标只会在顶层模块运行。例如maven-javadoc-plugin插件的aggregator-jar目标使用@aggregator标注，它不会为多模块的每个目标生成javadoc，而是在顶层项目生成一个已经聚合的javadoc文档。

### 1.3.9 @execute goal=“<goal>”：

在运行该目标之前先让Maven运行另外一个目标，如果是本插件的目标，则直接使用目标名称，否则使用“prefix：goal”的形式，即注明目标前缀。例如，maven-pmd-plugin是一个使用PMD来分析源码的工具，它包含pmd和check等目标，其中pmd用来生成报告，而check用来验证报告。由于check是依赖于pmd生成的内容的，因此可以看到它使用了标注@execute goal=”pmd”。

### 1.3.10 @execute phase=“<phase>”：

在运行该目标之前让Maven先运行一个并行的生命周期，到指定的阶段为止。例如maven-dependency-plugin的analyze使用了@execute phase=“test-compile”，因此当用户在命令行执行dependency:analyze的时候，Maven会首先执行default生命周期所有至test-compile的阶段。

### 1.3.11 @execute lifecycle=“<lifecycle>”phase=“<phase>”：

在运行该目标之前，让Maven先运行一个自定义的生命周期，到指定的阶段为止。例如maven-surefire-report-plugin这个用来生成测试报告的插件，它有一个report目标，标注了@execute phase=“test”lifecycle=“surefire”，表示运行这个自定义的surefire生命周期至test阶段。自定义生命周期的配置文件位于src/main/resource/META-INF/maven/lifecycle.xml，内容如代码清单所示：

|  |
| --- |
| <lifecycles>  <lifecycle>  <id>surefire</id>  <phases>  <phase>  <id>test</id>  <configuration>  <testFailureIgnore>true</testFailureIgnore>  </configuraiton>  </phase>  </phases>  </lifecycle>  </lifecycles> |

## 1.4 Mojo参数：

正如在代码清单中看到的那样，我们可以使用@parameter将Mojo的某个字段标注为可配置的参数，即Mojo参数。事实上几乎每个Mojo都有一个或者多个Mojo参数，通过配置这些参数，Maven用户可以自定义插件的行为。

Maven支持多种多样的Mojo参数，包括单值的boolean、int、float、String、Date、File和URL，多值的数组、Collection、Map、Properties等。

### 1.4.1 boolean（ 包括boolean和Boolean）：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private boolean sampleBoolean; |

对应的配置如下：

|  |
| --- |
| <sampleBoolean>true</sampleBoolean> |

### 1.4.2 int（包括Integer、long、Long、short、Short、byte、Byte）：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private int sampleInt; |

对应配置如下：

|  |
| --- |
| <sampleInt>8</sampleInt> |

### 1.4.3 float（包括Float、double、Double）：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private float sampleFloat; |

对应配置如下：

|  |
| --- |
| <sampleFloat>8.8</sampleInt> |

### 1.4.4 String（包括StringBuffer、char、Charater）：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private String sampleString; |

对应配置如下：

|  |
| --- |
| <sampleString>Hello World</sampleString> |

### 1.4.5 Date（格式为yyyy-MM-dd HH:mm:ss.S a或者yyyy-MM-dd HH:mm:ssa）：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private Date sampleDate; |

对应配置如下：

|  |
| --- |
| <sampleDate>2010-06-06 3:14:51.1 PM</sampleDate> |

或者

|  |
| --- |
| <sampleDate>2010-06-06 3:14:51PM</sampleDate> |

### 1.4.6 File：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private File sampleFile; |

对应配置如下：

|  |
| --- |
| <sampleFile>c:\tmp</sampleFile> |

### 1.4.7 URL：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private URL sampleURL; |

对应配置如下：

|  |
| --- |
| <sampleURL>http://www.baidu.com</sampleURL> |

### 1.4.8 数组：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private String[] includes; |

对应配置如下：

|  |
| --- |
| <includes>  <include>java</include>  <include>sql</include>  </includes> |

### 1.4.9 Collection(任何实现Collection接口的类，如ArrayList、HashSet)：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private list includes; |

对应配置如下：

|  |
| --- |
| <includes>  <include>java</include>  <include>sql</include>  </includes> |

### 1.4.10 Map：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private Map sampleMap; |

对应配置如下：

|  |
| --- |
| <sampleMap>  <key1>java</key1>  <key2>sql</key2>  </sampleMap> |

### 1.4.11 Properties：

|  |
| --- |
| /\*\*  \* @parameter  \*/  private Properties sampleProperties; |

对应配置如下：

|  |
| --- |
| <sampleProperties>  <property>  <name>p\_name\_1</name>  <value>p\_value\_1</value>  </property>  <property>  <name>p\_name\_2</name>  <value>p\_value\_2</value>  </property>  </sampleProperties> |

一个简单的@parameter标注就能让用户配置各种类型的Mojo字段，不过在此基础上，用户还能为@parameter标注提供一些额外的属性，进一步自定义Mojo参数。

* @parameter alias=“<aliasName>”：

使用alias，用户就可以为Mojo参数使用别名，当Mojo字段名称太长或者可读性不强时，这个别名就非常有用。例如：

|  |
| --- |
| /\*\*  \* @parameter alias=”uid”  \*/  private String uniqueIdentity; |

对应配置如下：

|  |
| --- |
| <uid>juven</uid> |

* @parameter expression=“${aSystemProperty}”

使用系统属性表达式对Mojo参数进行赋值，这是非常有用的特性。配置了@parameter的expression之后，用户可以在命令行配置Mojo参数。例如，maven-surefire-plugin的test目标有如下源码：

|  |
| --- |
| /\*\*  \* @parameter expression=”${maven.test.skip}”  \*/  private boolean skip; |

用户可以在POM中配置skip参数，同时也可以直接在命令行使用-Dmaven.test.skip=true来跳过测试。如果Maven参数没有提供expression，那就意味着该参数无法在命令行直接配置。好需要注意的是，Mojo参数的名称和expression名称不一定相同。

* @parameter defaultValue=“aValue/${anExpression}”：

如果用户没有配置该Mojo参数，就为其提供一个默认值。该值可以是一个简单字面量如“true”、“hello”、或者“1.5”，也可以是一个表达式，以方便使用POM的某个元素。

例如，下面代码中参数sampleBoolean默认值为true：

|  |
| --- |
| /\*\*  \* @parameter defaultValue=”true”  \*/  private boolean skip; |

* @readonly：

表示该Mojo参数是只读的，如果使用了该标注，用户就无法对其进行配置。通常在应用POM中元素内容的时候，我们不希望用户干涉。

* @required：

表示该Mojo参数是必须的，如果使用了该标注，但是用户没有配置该Mojo参数并且其没有默认值，Maven就会报错。

## 1.5 错误处理和日志：

如果大家看一下源码，会发现AbstractMojo实现了Mojo接口，execute()方法正是这个接口中定义的。具体代码如下：

|  |
| --- |
| void execute()  throws MojoExecutionException,MojoFailureException |

这个方法可以抛出两种异常，分别是MojoExecutionException和MojoFailureException。

如果Maven执行插件目标的时候遇到MojoFailureException，就会显式“BUILD FAILURE”的错误信息。这种异常表示Mojo在运行时发现了预期的错误。例如maven-surefire-plugin运行后若发现有失败的测试就会抛出该异常。

如果Maven执行插件目标的时候遇到MojoExecutionException，就会显示“BUILD ERROR”的错误信息。这种异常表示Mojo在运行时发现了未预期的错误。

上述两种异常能够在Mojo执行出错的时候提供一定的信息，但这往往是不够的，用户在编写插件的时候还应该提供足够的日志信息，AbstractMojo提供了一个getLog()方法，用户可以使用该方法获得一个Log对象。

# 编写Maven插件：注解方式

## pom.xml：

|  |
| --- |
| <modelVersion>4.0.0</modelVersion>  <groupId>com.chenming</groupId> <artifactId>Hello-plugin-annotation</artifactId> <version>1.1-SNAPSHOT</version> <packaging>maven-plugin</packaging>  <name>Hello-plugin-annotation</name> *<!-- FIXME change it to the project's website -->* <url>http://www.example.com</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target> </properties>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.11</version>  <scope>test</scope>  </dependency>   <dependency>  <groupId>org.apache.maven</groupId>  <artifactId>maven-plugin-api</artifactId>  <version>3.0</version>  </dependency>  *<!-- https://mvnrepository.com/artifact/org.apache.maven.plugin-tools/maven-plugin-annotations -->* <dependency>  <groupId>org.apache.maven.plugin-tools</groupId>  <artifactId>maven-plugin-annotations</artifactId>  <version>3.6.1</version>  <scope>provided</scope>  </dependency> </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.1</version>  <configuration>  <encoding>UTF-8</encoding>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>   <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-plugin-plugin</artifactId>  <version>3.2</version>  <configuration>  <goalPrefix>hello-annotation</goalPrefix>  <skipErrorNoDescriptorsFound>true</skipErrorNoDescriptorsFound>  </configuration>  </plugin>  </plugins> </build> |

需要注意的是：

1. <packaging>：打包方式，要填写maven-plugin。
2. maven-plugin-plugin：使用maven-plugin-plugin插件帮助生成插件前缀

## CountMojo：

|  |
| --- |
| @Mojo(name = "count",requiresProject = false) public class CountMojo extends AbstractMojo {  private static final String[] *INCLUDES\_DEFAULT* = {"java","xml","properties"};   @Parameter(required = true,property = "project.basedir")  private File basedir;   @Parameter(required = true,property = "project.build.sourceDirectory")  private File sourceDirectory;   @Parameter(required = true,property = "project.build.testSourceDirectory")  private File testSourceDirectory;   @Parameter(required = true,property = "project.build.resources")  private List<Resource> resources;   @Parameter(required = true,property = "project.build.testResources")  private List<Resource> testResources;   @Parameter(required = false,property = "includes")  private String[] includes;   @Override  public void execute() throws MojoExecutionException {  System.*out*.println("==========Hello plugin Annotation==========");  if (includes == null || includes.length == 0) {  includes = *INCLUDES\_DEFAULT*;  }  try {  countDir(sourceDirectory);  countDir(testSourceDirectory);  for (Resource resource : resources) {  countDir(new File(resource.getDirectory()));  }  for (Resource resource : testResources) {  countDir(new File(resource.getDirectory()));  }  } catch (IOException e) {  throw new MojoExecutionException("Unable to count lines of code",e);  }  System.*out*.println("==========Hello plugin Annotation==========");  }   private void countDir(File dir) throws IOException {  if (!dir.exists()) {  return;  }  List<File> collected = new ArrayList<>();  collectedFiles(collected,dir);  int lines = 0;  for (File sourceFile : collected) {  lines += countLine(sourceFile);  }  String path = dir.getAbsolutePath().substring(basedir.getAbsolutePath().length());  getLog().info(path + " : " + lines + " lines of code in " + collected.size() + " files");  }   private void collectedFiles(List<File> collected,File file) {  if (file.isFile()) {  for (String include : includes) {  if (file.getName().endsWith("." + include)) {  collected.add(file);  break;  }  }  } else {  for (File sub : file.listFiles()) {  collectedFiles(collected,sub);  }  }  }   private int countLine(File file) throws IOException {  BufferedReader bufferedReader = new BufferedReader(new FileReader(file));  int lines = 0;  try {  while (bufferedReader.ready()) {  bufferedReader.readLine();  lines++;  }  } finally {  bufferedReader.close();  }  return lines;  } } |

## 测试：

在一个Maven项目中打开命令行，运行命令，执行插件:目标

