第 17 章

编写 Maven 插件

本章内容

- ❑ 编写 Maven 插件的一般步骤
- ❑ 案例：编写一个用于代码行统计的 Maven 插件
- ❑ Mojo 标注
- ❑ Mojo 参数
- ❑ 错误处理和日志
- ❑ 测试 Maven 插件
- ❑ 小结

本书第7章已经讲过，Maven 的任何行为都是由插件完成的，包括项目的清理、编译、测试以及打包等操作都有其对应的 Maven 插件。每个插件拥有一个或者多个目标，用户可以直接从命令行运行这些插件目标，或者选择将目标绑定到 Maven 的生命周期。

大量的 Maven 插件可以从 Aapche[㊀]和 Codehaus[㊁]获得，这里的近百个插件几乎能够满足所有 Maven 项目的需要。除此之外，还有很多 Maven 插件分布在 Googlecode、Sourceforge、Github 等项目托管服务中。因此，当你发现自己有特殊需要的时候，首先应该搜索一下看是否已经有现成的插件可供使用。例如，如果想要配置 Maven 自动为所有 Java 文件的头部添加许可证声明，那么可以通过关键字 maven plugin license 找到 maven- license- plugin[㊂]，这个托管在 Googlecode 上的项目完全能够满足我的需求。

在一些非常情况下（几率低于1%），你有非常特殊的需求，并且无法找到现成的插件可供使用，那么就只能自己编写 Maven 插件了。编写 Maven 插件并不是特别复杂，本章将详细介绍如何一步步编写能够满足自己需要的 Maven 插件。

17.1 编写 Maven 插件的一般步骤

为了能让读者对编写 Maven 插件的方法和过程有一个总体的认识，下面先简要介绍一下编写 Maven 插件的主要步骤。

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者多个目标，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）错误处理及日志：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。

17.2 案例：编写一个用于代码行统计的 Maven 插件

为了便于大家实践，下面将详细演示如何实际编写一个简单的用于代码行统计的 Maven

㊀ 网址为：http://maven. apache. org/plugins/index. html。

㊁ 网址为：http://mojo. codehaus. org/plugins. html

㊂ 网址为：http://code. google. com/p/maven-license-plugin/。

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。不过，笔者强烈反对使用代码行来考核程序员，因为大家都知道，代码的数量并不能真正反映一个程序员的价值。

要创建一个 Maven 插件项目，首先使用 maven-archetype-plugin 命令：

```
$ mvn archetype:generate
```

然后选择：

```
maven-archetype-plugin (An archetype which contains a sample Maven plugin.)
```

输入 Maven 坐标等信息之后，一个 Maven 插件项目就创建好了。打开项目的 pom.xml 可以看到如代码清单 17-1 所示的内容。

代码清单 17-1　代码行统计插件的 POM

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd" >
    <modelVersion >4.0.0 < /modelVersion >
    <groupId >com.juvenxu.mvnbook < /groupId >
    <artifactId >maven-loc-plugin < /artifactId >
    <packaging >maven-plugin < /packaging >
    <version >0.0.1-SNAPSHOT < /version >
    <name >Maven LOC Plugin < /name >
    <url >http://www.juvenxu.com/ < /url >

    <properties >
        <maven.version >3.0 < /maven.version >
    < /properties >

    <dependencies >
        <dependency >
            <groupId >org.apache.maven < /groupId >
            <artifactId >maven-plugin-api < /artifactId >
            <version > ${maven.version} < /version >
        < /dependency >
    < /dependencies >
< /project >
```

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需的类，例如稍后会看到的 AbstractMojo。需要注意的是，代码清单 17-1 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，如代码清单 17-2 所示。

代码清单 17-2 CountMojo 的主要代码

```
/**
 * Goal which counts lines of code of a project
 *
 * @goal count
 */
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 types which will be included for counting
     *
     * @parameter
     */
    private String[] includes;

    public void execute()
        throws MojoExecutionException
    {
        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 );
        }
    }
}
```

首先，每个插件目标类，或者说 Mojo，都必须继承 AbstractMojo 并实现 execute（）方法，只有这样 Maven 才能识别该插件目标，并执行 execute（）方法中的行为。其次，由于历史原因，上述 CountMojo 类使用了 Java 1.4 风格的标注（将标注写在注释中），这里要关注的是@ goal，任何一个 Mojo 都必须使用该标注写明自己的目标名称，有了目标定义之后，我们才能在项目中配置该插件目标，或者在命令行调用之。例如：

```
$ mvn com.juvenxu.mvnbook:maven-loc-plugin:0.0.1-SNAPSHOT:count
```

创建一个 Mojo 所必要的工作就是这三项：继承 AbstractMojo、实现 execute（）方法、提供@ goal 标注。

下一步是为插件提供配置点。我们希望该插件默认统计所有 Java、XML，以及 properties 文件，但是允许用户配置包含哪些类型的文件。代码清单 17-2 中的 includes 字段就是用来为用户提供该配置点的，它的类型为 String 数组，并且使用了@

parameter 参数表示用户可以在使用该插件的时候在 POM 中配置该字段，如代码清单 17-3 所示。

代码清单 17-3 配置 CountMojo 的 includes 参数

```
<plugin>
    <groupId>com.juvenxu.mvnbook</groupId>
    <artifactId>maven-loc-plugin</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <configuration>
        <includes>
            <include>java</include>
            <include>sql</include>
        </includes>
    </configuration>
    </executions>
</plugin>
```

代码清单 17-3 配置了 CountMojo 统计 Java 和 SQL 文件，而不是默认的 Java、XML 和 Properties。

代码清单 17-2 中还包含了 basedir、sourceDirectory、testSourceDirectory 等字段，它们都使用了@ 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。${project.basedir}、${project.build.sourceDirectory}、${project.build.testSourceDirectory} 等表达式读者应该已经熟悉，它们分别表示了项目的基础目录、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。

了解这些简单的配置点之后，下一步就该实现插件的具体行为了。从代码清单 17-2 的 execute（）方法中大家能看到这样一些信息：如果用户没有配置 includes 则就是用默认的统计包含配置，然后再分别统计项目主代码目录、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，其具体实现如代码清单 17-4 所示。

代码清单 17-4 CountMojo 的具体行为实现

```
private void countDir( File dir )
    throws IOException
{
    if ( !dir.exists() )
    {
        return;
    }

    List<File> collected = new ArrayList<File>();

    collectFiles( 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 collectFiles( 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() )
            {
                collectFiles( collected, sub );
            }
        }
    }

    private int countLine( File file )
        throws IOException
    {
        BufferedReader reader = new BufferedReader( new FileReader( file ) );

        int line = 0;

        try
        {
            while ( reader.ready() )
            {
                reader.readLine();

                line++;
            }
        }
        finally
        {
            reader.close();
```

```
        }

        return line;
    }
```

这里简单解释一下上述三个方法：collectFiles()方法用来递归地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统计某一目录下共有多少文件被统计，以及这些文件共包含了多少代码行。

代码清单 17-2 中的 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 MojoExecutationException 对其简单包装后再抛出，Maven 执行插件目标的时候如果遇到 MojoExecutationException，就会在命令行显示“BUILD ERROR”信息。

代码清单 17-4 中的 countDir()方法的最后一行使用了 AbstractMojo 的 getLog()方法，该方法返回一个类似于 Log4j 的日志对象，可以用来将输出日志到 Maven 命令行。这里使用了 info 级别的日志告诉用户某个路径下有多少文件被统计，共包含了多少代码行，因此在使用该插件的时候可以看到如下的 Maven 输出：

```
[INFO] ---maven-loc-plugin:0.0.1-SNAPSHOT:count (default) @ app ---
[INFO] \src\main\java: 13 lines of code in 1 files
[INFO] \src\test\java: 38 lines of code in 1 files
```

使用 mvn clean install 命令将该插件项目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：

```
$ mvn com.juvenxu.mvnbook:maven-loc-plugin:0.0.1-SNAPSHOT:count
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Account Captcha 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] ---maven-loc-plugin:0.0.1-SNAPSHOT:count (default-cli) @ account-captcha ---
[INFO] \src\main\java: 179 lines of code in 4 files
[INFO] \src\test\java: 112 lines of code in 2 files
[INFO] \src\main\resources: 11 lines of code in 1 files
[INFO] \src\test\resources: 0 lines of code in 0 files
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.423s
[INFO] Finished at: Sat Jun 05 16:28:35 CST 2010
[INFO] Final Memory: 1M/4M
[INFO] ------------------------------------------------------------------------
```

如果嫌命令行太长太复杂，可以将该插件的 groupId 添加到 settings.xml 中。如下所示：

```
<settings>
  <pluginGroups>
    <pluginGroup>com.juvenxu.mvnbook</pluginGroup>
```

```
  </pluginGroups>
</settings>
```

现在 Maven 命令行就可以简化成：

```
$ mvn loc:count
```

这里面的具体原理可参考 7.8.4 节。

17.3　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。

❑ **@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。

❑ **@phase <phase>**

默认将该目标绑定至 Default 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。

❑ **@requiresDependencyResolution <scope>**

表示在运行该 Mojo 之前必须解析所有指定范围的依赖。例如，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。

❑ **@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。大部分插件目标都需要依赖一个项目才能执行，但有一些例外。例如 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@requiresProject false 标注。另外，maven-archetype-plugin 的 generate 目标也是一个很好的例子。

❑ **@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，就应当使用该标注。

❑ **@requiresOnline <true/false>**

表示是否要求 Maven 必须是在线状态，默认值是 false。

❑ **@requiresReport <true/false>**

表示是否要求项目报告已经生成，默认值是 false。

❑ **@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 的 aggregator-jar 使用了@aggregator 标注，它不会为多模块项目的每个

模块生成 Javadoc，而是在顶层项目生成一个已经聚合的 Javadoc 文档。

❑ **@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 来分析项目源码的工具，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”。

❑ **@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 使用了标注@ 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。

❑ **@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 声明周期至 test 阶段。自定义生命周期的配置文件位于 src/main/resources/META- INF/maven/lifecycle. xml。内容如代码清单 17-5 所示。

代码清单 17-5　maven- surefire- report- plugin 的自定义生命周期

```
<lifecycles>
  <lifecycle>
    <id>surefire</id>
    <phases>
      <phase>
        <id>test</id>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </phase>
    </phases>
  </lifecycle>
</lifecycles>
```

17.4　Mojo 参数

正如在代码清单 17-2 中所看到的那样，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。7. 5. 2 节和 7. 5. 3 节就分别配置了 maven- compiler- plugin 和 maven- antrun- plugin 的 Mojo 参数。

Maven 支持种类多样的 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。

❑ **boolean**（包括 boolean 和 Boolean）

```
/**
 * @parameter
 */
private boolean sampleBoolean
```

对应的配置如下：

```
<sampleBoolean>true</sampleBoolean>
```

❑ **int**（包括 Integer、long、Long、short、Short、byte、Byte）

```
/**
 * @parameter
 */
private int sampleInt
```

对应的配置如下：

```
<sampleInt>8</sampleInt>
```

❑ **float**（包括 Float、double、Double）

```
/**
 * @parameter
 */
private float sampleFloat
```

对应的配置如下：

```
<sampleFloat>8.8</sampleFloat>
```

❑ **String**（包括 StringBuffer、char、Character）

```
/**
 * @parameter
 */
private String sampleString
```

对应的配置如下：

```
<sampleString>Hello World</sampleString>
```

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

```
/**
 * @parameter
 */
private Date sampleDate
```

对应的配置如下：

```
<sampleDate>2010-06-06 3:14:55.1 PM</sampleDate>
```

或者

```
<sampleDate>2010-06-06 3:14:55 PM</sampleDate>
```

❑ **File**

```
/**
 * @parameter
 */
private File sampleFile
```

对应的配置如下：

```
<sampleFile>c:\tmp</sampleFile>
```

❑ **URL**

```
/**
 * @parameter
 */
private URL sampleURL
```

对应的配置如下：

```
<sample-URL>http://www.juvenxu.com/</sampleURL>
```

❑ **数组**

```
/**
 * @parameter
 */
private String[] includes
```

对应的配置如下：

```
<includes>
  <include>java</include>
  <include>sql</include>
</includes>
```

❑ **Collection**（任何实现 Collection 接口的类，如 ArrayList 和 HashSet）

```
/**
 * @parameter
 */
private List includes
```

对应的配置如下：

```
<includes>
  <include>java</include>
  <include>sql</include>
</includes>
```

❑ **Map**

```
/**
 * @parameter
 */
private Map sampleMap
```

对应的配置如下：

```
<sampleMap>
  <key1>value1</key2>
  <key1>value2</key2>
</sampleMap>
```

❑ **Properties**

```
/**
 * @parameter
 */
private Properties sampleProperties
```

对应的配置如下：

```
<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
```

对应的配置如下：

```
<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该 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 Mojo 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。还需要注意的是，Mojo 参数的名称和 expression 名称不一定相同。

❑ **@parameter default-value = "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，下面代码中的参数 sampleBoolean 默认值为 true：

```
/**
 * @parameter defaultValue="true"
 */
private boolean sampleBoolean
```

代码清单 17-2 中有如下代码：

```
/**
 * @parameter expression="${project.build.sourceDirectory}"
 * @required
 * @readonly
 */
private File sourceDirectory;
```

表示默认使用 POM 元素 <project><build><sourceDirectory> 的值。

除了@parameter 标注外，还看到可以为 Mojo 参数使用@readonly 和@required 标注。

❑ **@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 元素内容的时候，我们不希望用户干涉。代码清单 17-2 就是很好的例子。

❑ **@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 参数且其没有默认值，Maven 就会报错。

17.5 错误处理和日志

如果大家看一下 Maven 的源码，会发现 AbstractMojo 实现了 Mojo 接口，execute() 方法正是在这个接口中定义的。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，就会显示“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 MojoExecutationException，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，例如代码清单 17-2

中我们不知道代码行统计插件何时会遇到 IOException，这个时候只能将其嵌套进 MojoExecutionException 后再抛出。

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。该对象支持四种级别的日志方法，它们从低到高分别为：

- **debug**：调试级别的日志。Maven 默认不会输出该级别的日志，不过用户可以在执行 mvn 命令的时候使用 -X 参数开启调试日志，该级别的日志是用来帮助程序员了解插件具体运行状态的，因此应该尽量详细。需要注意的是，不要指望你的用户会主动去看该级别的日志。
- **info**：消息级别的日志。Maven 默认会输出该级别的日志，该级别的日志应该足够简洁，帮助用户了解插件重要的运行状态。例如，maven-compiler-plugin 会使用该级别的日志告诉用户源代码编译的目标目录。
- **warn**：警告级别的日志。当插件运行的时候遇到了一些问题或错误，不过这类问题不会导致运行失败的时候，就应该使用该级别的日志警告用户尽快修复。
- **error**：错误级别的日志。当插件运行的时候遇到了一些问题或错误，并且这类问题导致 Mojo 无法继续运行，就应该使用该级别的日志提供详细的错误信息。

上述每个级别的日志都提供了三个方法。以 debug 为例，它们分别为：

- void debug (CharSequence content) ;
- void debug (CharSequence content, Throwable error) ;
- void debug (Throwable error) ;

用户在编写插件的时候，应该根据实际情况选择适应的方法。基本的原则是，如果有异常出现，就应该尽量使用适宜的日志方法将异常堆栈记录下来，方便将来的问题分析。

如果使用过 Log4j 之类的日志框架，就应该不会对 Maven 日志支持感到陌生，日志不是一个 Maven 插件的核心代码，但是为了方便使用和调试，完整的插件应该具备足够丰富的日志代码。

17.6　测试 Maven 插件

编写 Maven 插件的最后一步是对其进行测试，单元测试较之于一般的 Maven 项目无异，可以参考第 10 章。手动测试 Maven 插件也是一种做法，读者可以将插件安装到本地仓库后，再找个项目测试该插件。本节要介绍的并非上述两种读者已经十分熟悉的测试方法，而是如何编写自动化的集成测试代码来验证 Maven 插件的行为。

读者可以想象一下，既然是集成测试，那么就一定需要一个实际的 Maven 项目，配置该项目使用插件，然后在该项目上运行 Maven 构建，最后再验证该构建成功与否，可能还

需要检查构建的输出。

既然有数以千计的 Maven 插件，那么很可能已经有很多人遇到过上述的需求，因此 Maven 社区有一个用来帮助插件集成测试的插件，它就是 maven-invoker-plugin。该插件能够用来在一组项目上执行 Maven，并检查每个项目的构建是否成功，最后，它还可以执行 BeanShell 或者 Groovy 脚本来验证项目构建的输出。

BeanShell 和 Groovy 都是基于 JVM 平台的脚本语言，读者可以访问 http://www.beanshell.org/和 http://groovy.codehaus.org/以了解更多的信息。本章下面的内容会用到少许的 Groovy 代码，不过这些代码十分简单，很容易理解。

回顾一下前面的代码行统计插件，可以使用 Archetype 创建一个最简单的 Maven 项目，然后在该项目中配置 maven-loc-plugin。如果一切正常，就应该能够看到如下的 Maven 构建输出：

```
[INFO] \src\main\java:13 lines of code in 1 files
[INFO] \src\test\java:38 lines of code in 1 files
```

为了验证这一行为，先配置 maven-loc-plugin 的 POM 使用 maven-invoker-plugin，如代码清单 17-6 所示。

代码清单 17-6　配置 maven-loc-plugin 使用 maven-invoker-plugin

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-invoker-plugin</artifactId>
    <version>1.5</version>
    <configuration>
        <projectsDirectory>src/it</projectsDirectory>
        <goals>
            <goal>install</goal>
        </goals>
        <postBuildHookScript>validate.groovy</postBuildHookScript>
    </configuration>
    <executions>
        <execution>
            <id>integration-test</id>
            <goals>
                <goal>install</goal>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

代码清单 17-6 中 maven-invoker-plugin 有三项配置。首先 projectDirectory 用来配置测试项目的目录，也就是说在 src/it 目录下存放要测试的 Maven 项目源码；其次 goals 表示在测试项目上要运行的 Maven 目标，这里的配置就表示 maven-invoker-plugin 会在 src/it 目录下的各个 Maven 项目中运行 mvn install 命令；最后的 postBuildHookScript 表示在测试完成后要运行的验证脚本，这里是一个 groovy 文件。

从代码清单 17-6 中我们还看到，maven-invoker-plugin 的两个目标 install 和 run 被绑定到了 integration-test 生命周期阶段。这里的 install 目标用来将当前的插件构建并安装到仓库中供测试项目使用，run 目标则会执行定义好的 mvn 命令并运行验证脚本。

当然仅仅该配置还不够，src/it 目录下必须有一个或者多个供测试的 Maven 项目，我们可以使用 maven-archetype-quickstart 创建一个项目并修改 POM 使用 mvn-loc-plugin，如代码清单 17-7 所示。该测试项目的其余代码不再赘述。

代码清单 17-7　maven-loc-plugin 的测试项目 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.juvenxu</groupId>
    <artifactId>app</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>app</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>com.juvenxu.mvnbook</groupId>
                <artifactId>maven-loc-plugin</artifactId>
                <version>0.0.1-SNAPSHOT</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>count</goal>
                        </goals>
                        <phase>verify</phase>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

代码清单 17-7 就是一个最简单的 POM，然后配置 maven-loc-plugin 的 count 目标绑定到了 verify 生命周期阶段。

测试项目准备好了，现在要准备的是与该项目对应的验证脚本文件，即 validate.groovy，

它应该位于 src/ it/ app 目录下（即上述测试项目的根目录），内容如代码清单 17-8 所示。

代码清单 17-8 maven- loc- plugin 的集成测试验证脚本

```
def file = new File(basedir,'build.log')

def countMain = false
def countTest = false

file.eachLine {
  if ( it = ~ /src.main.java: 13 lines of code in 1 files/)
    countMain = true
  if ( it = ~ /src.test.java: 38 lines of code in 1 files/)
    countTest = true
}

if ( !countMain )
  throw new RuntimeException( "incorrect src/main/java count info" );

if ( !countTest )
  throw new RuntimeException( "incorrect src/test/java count info" );
```

这段 Groovy 代码做的事情很简单。它首先读取 app 项目目录下的 build. log 文件，当 maven- invoker- plugin 构建测试项目的时候，会把 mvn 输出保存到项目下的 build. log 文件中。因此，可以解析该日志文件来验证 maven- loc- plugin 是否输出了正确的代码行信息。

上述 Groovy 代码首先假设没有找到正确的主代码统计信息和测试代码统计信息，然后它逐行遍历日志文件，紧接着使用正则表达式检查寻找要检查的内容（两个斜杠 / / 中间的内容是正则表达式，而 = ~ 表示寻找该正则表达式匹配的内容），如果找到期望的输出，就将 countMain 和 countTest 置为 true。最后，如果这两个变量的值有 false，就抛出对应的异常信息。

Maven 会首先在测试项目 app 上运行 mvn install 命令，如果运行成功，则再执行 validate. groovy 脚本。只有脚本运行通过且没有异常，集成测试才算成功。

现在在 maven- loc- plugin 下运行 mvn clean install，就能看到如下的输出：

```
[INFO] --- maven - invoker-plugin:1.5:install (integration-test) @ maven-loc-
plugin ---
[INFO] Installing D:\ws-maven-book\maven-loc-plugin\pom.xml to D:\java\reposi-
tory\com\juvenxu\mvnbook\maven-loc-plugin\0.0.1-SNAPSHOT\maven-loc-plugin-
0.0.1-SNAPSHOT.pom
[INFO] Installing D:\ws-maven-book\maven-loc-plugin\target\maven-loc-plugin-
0.0.1-SNAPSHOT.jar to D:\java\repository\com\juvenxu\mvnbook\maven-loc-plugin\
0.0.1-SNAPSHOT\maven-loc-plugin-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] ---maven-invoker-plugin:1.5:run (integration-test) @ maven-loc-plugin ---
[WARNING] Filtering of parent/child POMs is not supported without cloning the pro-
jects
[INFO] Building: app\pom.xml
[INFO] .. SUCCESS (3.4 s)
[INFO] ------------------------------------------------
[INFO] Build Summary:
```

```
[INFO] Passed: 1, Failed: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------
```

从输出中可以看到 maven-invoker-plugin 的 install 目标将当前项目 maven-loc-plugin 安装至本地仓库，然后它的 run 目标构建测试项目 app，并最后报告运行结果。

至此，所有 Maven 插件集成测试的步骤就都完成了。

上述样例只涉及了 maven-invoker-plugin 的很少一部分配置点，用户还可以配置：

- **debug**（boolean）：是否在构建测试项目的时候开启 debug 输出。
- **settingsFile**（File）：执行集成测试所使用的 settings.xml，默认为本机环境 settings.xml。
- **localRepositoryPath**（File）：执行集成测试所使用的本地仓库，默认就是本机环境仓库。
- **preBuildHookScript**（String）：构建测试项目之前运行的 BeanShell 或 Groovy 脚本。
- **postBuildHookScript**（String）：构建测试项目之后运行的 BeanShell 或 Groovy 脚本。

要了解更多的配置点，或者查看更多的样例。读者可以访问 maven-invoker-plugin 的站点：http://maven.apache.org/plugins/maven-invoker-plugin/。

17.7　小结

Maven 社区提供了成百上千的插件供用户使用，这些插件能够满足绝大部分用户的需求。然而，在极少数的情况下，用户还是需要编写 Maven 插件来满足自己非常特殊的需求。编写 Maven 插件的一般步骤包括创建一个插件项目、编写 Mojo、为 Mojo 提供配置点、实现 Mojo 行为、处理错误、记录日志和测试插件等。本章实现了一个简单的代码行统计插件，并逐步展示了上述步骤。用户在编写自己插件的时候，还可以参考本章描述的各种 Mojo 标注、Mojo 参数、异常类型和日志接口。本章最后介绍了如何使用 maven-invoker-plugin 实现插件的自动化集成测试。