Bnd - Bundle Tool

介绍

BND 工具可用于创建符合 OSGi R4 规范的 Bundle 和检测 Jar 文件是否符合 OSGi R4 规范。其主要功能有：

- 显示 Bundle 的 manifest 信息和其所包含的 Jar 文件；
- 将 Jar 文件打包为 Bundle；
- 按照 OSGi R4 规范把类路径下的文件打包为 OSGi Bundle；
- 检测 Bundle 中的 manifest 信息是否符合 OSGi R4 规范。

BND 工具包含了：

- 命令行工具
- Eclipse 插件
- Ant Task
- Maven 插件

背景

通常情况下 jar 文件是通过 Sun jar tool、ant jar task 或 maven packager 打包生成的，所有的工具采用的为同样的方式，开发人员复制需要打包到 jar 的文件到目录中，然后打包这个目录，很明显这个方法是可行的。
Bnd 采用的是不同的处理方式，它根据 classpath 来组装需要打包的 java classes 和 packages。使用 Bnd 工具可从源代码、目录或者其他的 jar 文件中创建出 jar 文件，你不需要将文件复制来复制去，当需要的时候 Bnd 可以根据元信息找到它所需要的文件。

Bnd 在生成 jar 时根据 3 个不同的参数来生成：
Export-Package、Private-Package 和 Include-Resource

Private-Package 和 Export-Package 的值由很多带语法的语言片段组成的，这些语言片段由表达式、属性和描述组成，就像 OSGi 的属性和描述。例如：
Export-Package: com.acme.*;version=1.2
每个语言片段按照顺序应用到 classpath 中的每个 package 上，顺序的意思就是如果有一个更先的语言片段符合 package 的话，那么后面的语言片段就不会生效了。
如果语言片段中的属性和描述都符合 package，那么对于此 package 的描述就生效了。
Private-Package 和 Export-Package 的不同在于 Export-Package 是可以带版本的，如果 package 既属于 Private-Package，又属于 Export-Package，那么这个 package 会作为 Export-Package 而生效。
语言片段可以使用"!"开头，这种情况下!后的包不会被打入 jar 文件中，例如：

Export-Package: !com.acme.impl,com.acme.*;version=1.2
这类过滤性质的语言片段也是按顺序生效的，如果带!的语言片段写在最后，那么由于 com.acme.*已经生效了，最后打包出来的还是会有 com.acme.impl。

Include-Resource 参数可以用来定义需要复制哪些资源文件到 Jar 中，这可以用来打包 licenses、images 等等资源文件，参数里的语言片段可以是目录、文件或者 jar。如文件需要预先处理，可采用{}的方式来定义，如{license.txt}。

创建 jar 时，bnd 会分析 classes，并将其中引用到了但不在 jar 中的 packages 列入 import list。这些 import list 信息对应的放入 Import-Package 项中。正常情况下，Import-Package 的参数为*，也就是说所有引用的 packages 都会被导入。但有些时候可能需要忽略其中的个别引用的 package，那么可通过把 import 中的 package 定义为 optional 或取消 import 该 package，例如：
Import-Package: !com.acme.*, *;resolution:=optional

bnd 文件由很多的属性构成，以大写字母开头的属性会作为元信息复制到 manifest 文件中，小写字母开头的属性则作为变量而使用。

Jar 创建后，bnd 会校验创建的结果，主要是依据 mainfest 文件来进行详细的校验，以确认 bundle 的有效性。bnd 程序比传统的 jar 更高一级，除了 jar 的打包之外它还提供了一些其他的功能，它更强调以 packages 的角度来进行打包，而不仅仅是文件的角度，Bnd 的打包方式保证了在打包生成 Bundle 时就可以检查出 Bundle 是否符合 OSGi R4 规范。

如打包生成的 jar 文件已存在并且没有文件需要更新，则 bnd 不会产生新的打包文件。

可以以几种方式来使用 bnd：命令行、ant task、maven 插件和 Eclipse 插件。

快速上手

假设我们需要在 Eclipse 中创建一个 bundle，Eclipse 中的每个 java 工程都是有类路径和源码的，因此 Bnd 可以获取到所有的 classes，但它仍然无法知道你想怎么样组建 jars/bundles，这就需要编写一个 bnd file 来告诉它，bnd file 的文件名默认为 bundle 的 symbolic name 加.bnd 来构成，例如 aQute.example.bnd 就是一个典型的 bnd 文件的文件名，只要文件名不是 bnd.bnd，那么.bnd 前的字符串就会作为 Bundle 的 symbolic name。

以 aQute OSGi tutorial Chat 为例，来创建一个 bundle，这个 bundle 中有两个 package：

- aQute.service.channel

- aQute.tutorial.chat

aQute.service.channel 包须对外提供，其他包则保持私有的状态，所有源码中引用到的不在 classpath 中的包都必须导入，要达到这个要求，bnd 文件的写法如下所示：

Export-Package: aQute.service.channel; version=1.0

Private-Package: aQute.tutorial.chat

你要做的就只有这些，在 Eclipse 中，你可以选择这个 bnd 文件，然后点击右键选择其中的 Make Bundle，将会创建一个包含以下目录和文件的 jar：

META-INF

MANIFEST.MF

aQute/service/channel

Channel.class

aQute/tutorial/chat

Chat$ChannelTracker.class

Chat.class

也可以通过命令行运行：

bnd aQute.tutorial.chat.bnd

来看看产生的 jar 文件里的 manifest.mf 文件，在命令行中可以通过 bnd aQute.tutorial.chat.jar 来查看，或直接使用 winzip 打开 jar 文件来看：

Manifest-Version: 1

Bundle-Name: aQute.tutorial.chat

Private-Package: aQute.tutorial.chat

Import-Package: aQute.service.channel;version=1.0,

org.osgi.framework; version=1.3,

org.osgi.util.tracker;version=1.3

Bundle-ManifestVersion: 2

Bundle-SymbolicName: aQute.tutorial.chat

Export-Package: aQute.service.channel;version=1.0

Bundle-Version: 0

就像你所看到的，bnd 补充了很多的头信息。第一个头信息：Manifest-Version 是 jar 文件必须的；Bundle-Name 是根据 Bundle-SymbolicName 来生成的，因为在 bnd 文件中未指定；Private-Package 头信息指定了不对外提供的 packages；Import-Package 头信息是 bnd 根据 classes 所引用的 packages 来生成的，可以看到，bnd 还提取了所

引用的 package 的版本信息，这些版本信息是从这些 package 的 jar 文件的 manifest 中获取的；Export-Package 显示了需要对外提供的 package。在生成 jar 文件时，bnd 会校验这些头信息是否符合 OSGi 规范，如不符合则会提供错误信息或警告信息。

Bnd 文件格式

Bnd 文件的格式和 manifest.mf 非常相似。尽管它是以 properties 文件的方式来读取的，还是可以使用':'作为分隔符。唯一要注意的就是 bnd 文件是不支持以空格开头的，每行的长度不限，key 和 value 前后的空格都会被去除，请参见 Properties 来了解更多关于此格式的信息。

bnd 文件具备几种不同的参数类型：

类型	举例	说明
Manifest headers	Bundle-Description: ...	当首字母大写时，bnd 会复制这些信息到 manifest 中作为头信息。
Variables	Version=3.0	当首字母小写时bnd视其为变量。元信息可通过宏方式来获取变量的值，变量是不会被复制到manifest中的。请参见Macros 。
Directives	-include: deflts.bnd	声明以'-'开头。 声明是为了告诉bnd需要做一些特殊的处理。请参见 Directives

Bnd Directives

声明参数	格式	说明
-classpath	LIST	把 list 中的文件加到 bundle 的 classpath 中。这些文件的路径必须是相对 bnd 文件的相对路径，且为 jar 文件或目录，例如： -classpath= acme.jar, junit.jar, bin
-donotcopy	REGEX	当复制文件到 jar 中时，可以通过此参数来控制不需要复制的文件。例如，CVS 文件和 SVN 文件不需要复制到 jar 中的，可以这么写来实现： -donotcopy= (CVS|.svn|.+.bak|~.+)
-include	LIST	这个属性用来指定需要打入 jar 的文件。文

		件路径为相对 bnd 文件的相对路径。如果引用的是外部的文件，则外部文件中的文件路径为相对该外部文件的相对路径。Includes 对于统一放置 Bundle-Vendor 和 Bundle-Copyright 这样的属性值非常的有用。如果文件的扩展名为 mf，那么这个文件将作为 manifest 文件进行解析。如果有很多个同样的变量，只有最后一个变量会生效。文件名可使用{user.home}这样的变量。如果文件不存在，将会产生错误。如果文件名以'-'开头，那么当文件不存在的时候将会忽略此错误。例如： -include= ${user.home}/deflts.bnd, META-INF/MANIFEST.MF
-failok	true | false	在某些情况下，出现错误也不应该导致打包退出。例如测试用例在打包时通常也需要运行。如果这个参数设置为 true，那么无论出现什么错误都会产生 bundle 的 jar 文件，并将错误信息列在错误文件中。如 failok 设置为 false，那么只要出错就不会产生 Bundle 的 jar 文件。例如： -failok= true
Export-Package	LIST of PATTERN	Export-Package列出了Bundle需要对外提供的package。参见ExportPackage.
Include-Resource	LIST of iclause	Include-Resource使得Bundle可包含多个资源文件的路径。参见Include Resource.
Private-Package	LIST of PATTERN	Private-Package中列出了Bundle中需要包含但不对外提供的package。参见Private Package.
Import-Package	LIST of PATTERN	Import-Package列出了Bundle所需引用的package。参见Import Package.
Conditional-Package	LIST of	这是一个用来补充 Private-Package 的参

	PATTERN	数，使用这个参数后就指定了需要打包到 Bundle 中的 package，而且也只有这些 package 会打包到 Bundle 中。
Bundle-SymbolicName		用户可设置 Bundle-SymbolicName。默认情况下为 bnd 文件去除.bnd 后的文件名，如 bnd 文件的文件名为 bnd.bnd，那么就默认为 bnd 文件所在的目录名。也可使用 ${project}作为其名称。
Bundle-Name		如未设置 Bundle-Name，将会将 Bundle-Name 赋值为 Bundle-SymbolicName 的值。
Bundle-ManifestVersion	2	Bundle-ManifestVersion 被设置为固定值 2。
Bundle-Version	VERSION	bundle 的版本，如未设置此值，则默认为 0。
Service-Component	LIST of component	参见Service Component Header.

Export-Package

bnd 定义允许使用模式(经过修改的正则表示式)的方式来定义此参数。定义中所有的模式作用于 classpath 中的每个包。如模式为过滤模式(以!开头)并且与 classpath 中 package 匹配，那么打包时就会忽略此 package。其他方式定义的 Export-Package 的 package 都会复制到目标 bundle 中。模式包含 directives 和属性，参数中的模式是按顺序生效的，排在前面的模式优先后面的模式而生效。下面的例子为对外提供除以'com'开头外的 classpath 中的所有的 packages。默认的 Export-Package 为'*'，对于非常大的 bundles 而言是很有效的。如果输出的 packages 有相关的版本信息(packageinfo 文件的元数据信息中)，那么这些版本信息将会自动添加到 Export-Package 中。

```
Export-Package= !com.*, *
```

输出的 package 自动的列入 Import-Package 中，这个特性可通过在 Export-Package 中添加一个特殊的 directive 来取消：-noimport:=true，例如：

```
Export-Package= com.acme.impl.*;-noimport:=true, *
```

Private-Package

这个参数和 Export-Package 的使用方法相同，唯一不同的就是在这里指定的 packages 是不对外暴露的，这个头信息只是复制到 manifest 中而已。如果一个包既在 Export-Package 里定义了，又在 Private-Package 里定义了，那么将会做为 Export-Package。

```
Private-Package= com.*
```

Import-Package

Import-Package 用来指定需要引用的 packages。默认值为'*'，也就是导入所有引用的 packages，因此 Import-Package 很少被设置。但在某些特殊的情况下可能会不希望引用某些包，因为这些包其实从来就没用到过，在这种情况下可以使用过滤模式来实现，还有一种就是需要导入没有明确定义需要引用的 package(例如采用 Class.forName 方式加载的类)，这种就需要手工在 Import-Package 中来定义，否则 bnd 是不会自动生成。

例如：

```
Import-Package: !org.apache.commons.log4j, com.acme.*,
com.foo.extra
```

在处理过程中，bnd 会尝试寻找引用的 packages 导出时所指定的版本。如未定义引用 package 的版本或版本范围，bnd 就会使用导出版本的基本版本号。 例如导出的版本为 1.2.3.build123，那么导入的版本为 1.2.3，如指定了导入的版本或版本范围，那么 bnd 将使用找到的符合的版本来覆盖，在版本范围的定义中${@}可用来代表找到的范围内的版本。

```
Import-Package: org.osgi.framework;version="[1.3,2.0)"
Import-Package: org.osgi.framework;version="[${@},2.0)"
```

Include-Resource

Include Resources 指定需要复制到目标 jar 文件中的资源文件。有以下几种方式来定义：

```
iclause ::= inline | copy
copy ::= '{' process '}' | process
process ::= assignment | simple
assignment ::= PATH '=' PATH
simple ::= PATH
inline ::= '@' PATH ( '!/' PATH? ('/**' | '/*')? )?
```

对于 assignment 或 simple 而言，可通过设置 PATH 参数指定文件或目录。也可以使用 classpath 中 JAR 文件的 name.ext 路径，也就是说，忽略目录。simple 方式采用的是直接把文件不带目录的放入目标 jar 文件中，例如，工程中有 src/a/b.c 文件，以 simple 方式处理的话 bnd 将会把 b.c 直接放到目录 JAR 文件的根目录下。

如资源文件必须放在目标 JAR 中的子目录下，可以使用 assignment 方式来实现。

如文件未找到，bnd 将会寻找整个 classpath 来查找是否有匹配的文件。

inline 参数必须是 ZIP 或 JAR 文件， 文件可以指定为 jar 中的具体的文件名，或者为一个带**或*的目录。**表示在查找时采用的为递归查询的方式，*则只在指定的目录下查找，如仅指定了目录名，就相当于目录名**。

simple 和 assigment 方式都可以使用{}的方式，使用{}的这种文件意味着是会被率先处理的，同样也可以使用变量或宏定义，关于变量和宏定义请参见 macro 章节。

```
Include-Resource: @osgi.jar,
{LICENSE.txt}, acme/Merge.class=src/acme/Merge.class
```

Service-Component Header

Service-Component 头信息和标准的 OSGi 的头信息是兼容的。在列表中未带属性的元素必须相应的有资源文件在 Jar 中，并且该元素信息会复制到 manifest 中。组件定义的标识为：

```
component ::= <implementation-class> ( ';' parameter ) *
parameter ::= provide | reference | multiple | optional
| reference | properties | factory | servicefactory
| immediate | enabled
reference ::= <name> '=' <interface-class>
provide ::= 'provide:=' LIST
multiple ::= 'multiple:=' LIST
optional ::= 'optional:=' LIST
dynamic ::= 'dynamic:=' LIST
factory ::= 'factory:=' true | false
servicefactory := 'servicefactory:=' true | false
immediate ::= 'immediate:=' true | false
enabled ::= 'enabled:=' true | false
properties::= 'properties:=' key '=' value \
( ',' key '=' value ) *
```

组件名也就是实现的类名。之后为其引用的服务，每个服务的名称对应的为 ComponentContext.locateService 中调用的服务名，如果名称以小写开头，则被认为是 bean 属性，在这种情况下，这个引用将按照 set<名称>和 unset<名称>的标准 bean 规则来使用。Bnd 负责解析这个头信息，并在输出的 jar 文件的 OSGI-INF 目录下创建一个<id>.xml 的文件。

下面的例子演示了一个通过 setLog 来调用 log 服务的组件的头信息的定义方法：

```
Service-Component=aQute.tutorial.component.World; \
```

```
log=org.osgi.service.log.LogService
```

在 Declarative Services 的规范中 SCR 还提供了很多的附加选项的支持。有些选项像 target 是不支持的，但像 policy 和 cardinality 是支持的，如果这个引用是 optional 性质的，那么应该在 optional 声明中进行定义，这也就意味着 cardinality 从 0 开始。如该组件需要引用服务接口的多个实现，则需要在 multiple 声明中定义。policy 是通过 dynamic 声明来定义的，需要 dynamic 引用的就在 dynamic 声明中定义。SCR 赋的默认值为：1..1 的 cardinality 和 static 的 policy。看一个复杂的例子：

```
Service-Component=aQute.tutorial.component.World; \
log=org.osgi.service.log.LogService; \
http=org.osgi.service.http.HttpService; \
dynamic:='log,http'; \
optional:=log; \
PROCESSORS=xierpa.service.processor.Processor; \
multiple:=PROCESSORS; \
properties:="wazaabi=true"
```

根据上面的方式编写的头信息 bnd 产生一个 OSGI-INF/aQute.tutorial.component.World.xml 文件，文件的内容为：

```
<?xml version="1.0" encoding="utf-8" ?>
<component name="aQute.tutorial.component.World">
<implementation class="aQute.tutorial.component.World" />
<reference name="log"
interface="org.osgi.service.log.LogService"
cardinality="0..1"
bind="setLog"
unbind="unsetLog"
policy="dynamic" />
<reference name="http"
interface="org.osgi.service.http.HttpService"
bind="setHttp"
unbind="unsetHttp"
policy="dynamic" />
<reference name="PROCESSORS"
interface="xierpa.service.processor.Processor"
cardinality="1..n" />
```

</component>

在头信息的描述里还支持 immediate、enabled、factory 和 servicefactory 属性。如需了解这些属性的信息，请参见 Declarative Services。

Macros

宏定义处理器可用于对头信息进行复杂的处理。宏变量可使用单一定义的值，也可使用复杂的函数。每个头信息都相当于一个可扩展的宏，作为宏的头信息是不以大写字母开头的，也不会复制到 manifest 中去，因此他们也可被当成宏变量而使用。可通过${<name>}(大括号方式)或$(<name>)(小括号方式)来引用变量。另外，也可以使用[]、<>、《》和‹›。

例如：

```
version=1.23.87.200109111023542
Bundle-Version= ${version}
Bundle-Description= This bundle has version ${version}
```

bnd 默认的设置了一些属性值：

属性名	说明
project	工程名，默认为去除.bnd 后的文件名，如文件名为 bnd.bnd，则采用 bnd 文件所在的目录名。
project.file	bnd 文件的绝对路径。
project.name	所在的绝对路径的文件名称。
project.dir	bnd 文件所在的目录的绝对路径。

bnd 自带了一些实现基本功能的 macros。采用如下格式使用和定义函数：

```
macro ::= '${' function '}'
| function
| '$(' function ')'
| '$<' function '>'
| '$«' function '»'
| '$‹' function '›'
function ::= name ( ':' argument ) *
```

函数	参数	说明

filter	; list ; regex	filter 遍历所有给定的值，并只包含符合这些正则表达式的元素。下面的例子演示了如何用 filter 来实现只包含 list 中的 jar 文件： list= a,b,c,d,x.jar,z.jar List= ${filter;${list};.*\\.jar}
filterout	; list ; regex	filterout 遍历所有给定的值，并去除符合这些正则表达式的元素。下面的例子演示了如何用 filterout 来实现去除 list 中的 jar 文件： list= a,b,c,d,x.jar,z.jar List= ${filterout:${list};.*\\.jar}
Sort	; list	Sort 用于实现把所有的元素按字母排序。例如： List= ${sort:acme.jar, harry.jar, runner.jar, alpha.jar, bugs.jar}
Join	(; list) *	Join 用来实现把元素集合合并为同一个集合。看起来这个功能用两个宏直接连接起来也可以实现，但除非是用,隔开两个集合的元素，否则用两个宏直接连接形成的将不是 list，但如果其中的一个集合元素为空，则会出现错误。join 则很好的处理了这种情况，join 可支持给定的任何数量的元素集合的合并，例如： List= ${join;a,b,c;d,e,f}
if	; condition ; true (; false) ?	如果条件不为空，则返回 true 对应的值，如为空，则返回 false 对应的值，如未设置 false 对应的值，则返回空字符串，条件在运行前会做去空格处理。例如： Comment: ${if;${version};Ok;Version is NOT set!!!!}
now		以字符串形式返回当前日期，例如： Created-When: ${now}
fmodified	; file-path-list	返回给定的文件路径的最近的修改时间。这个函数是基于 Java 提供的 API 所编写的，所以其返回值为 long 类型。例如： Last-Modified:

		${long2date;${fmodified;${files}})
Long2date	; long	把 long 类型转换为 date 类型，例如： Last-Modified: ${long2date:${fmodified:${files}})
replace	; list ; regex ; replacement	替换符合表达式的值。函数基于 item.replaceAll 实现。如需给所有的文件添加.jar 的扩展名，可使用如下方法实现： List = ${replace;${impls};$;.jar}
toclassname	; list	把类文件(如：org/osgi/service/LogService.class)转换为类名(如 org.osgi.service.LogService)。
toclasspath	; list	把类名(如 org.osgi.service.LogService)转换为类文件名(org/osgi/service/LogService.class)。
findname	; regex [; replacement]	寻找到符合表达式的资源的路径，并将资源的名称以给定的值替换。仅用于替换资源的名称，并不包含/。
findpath	; regex [; replacement]	寻找到符合表达式的资源的路径，并将资源的路径以给定的值替换，在替换时包含/。

命令行

命令行可使用几种方式来执行：

bnd general-options cmd cmd-options

bnd general-options <file>.jar

bnd general-options <file>.bnd

全局选项

全局选项	说明
-failok	和-failok 属性是相同的，默认情况下即使出现错误也会完成打包。
-exceptions	当打包时出现导致程序退出的致命错误时打印出此异常信息。默认情况下只打印了异常的简要信息，添加此选项可打印出详细的异常堆栈信息，这对于调试和查找错误原因是非常有帮助的。

print (-verify | -manifest | -list | - all) * <file>.jar +

print 可用于查看 JAR 文件的某些信息，以下的信息可通过 print 来查看：

- -verify - 验证 jar 包是否符合 OSGi 规范的要求，如不符合则会在控制台中显示错误信息。
- -manifest - 显示 jar 包中的 manifest.mf 文件的信息。
- -list - 列出 jar 包中文件的信息。
- -all - 执行以上所有的动作(默认就是-all)。

使用示例：

```
bnd print -verify *.jar
```

build (-classpath LIST | -eclipse <file> | -noeclipse | -output <file>) * <file>.bnd +

build 可用于按照 bnd 的描述打包生成 bundle 文件。默认输出的 bundle 的文件名为 bnd 文件的文件名+.jar。

可用的参数有：

- -classpath - 放入 classpath 中的 jar 文件或目录。
- -eclipse - 把指定的文件作为 eclipse 的.classpath 文件而使用，如使用了此参数，则默认的.classpath 文件不会被读取。
- -noeclipse - 不解析 Eclipse 工程下的.classpath 文件。
- -output - 覆盖输出的 Bundle 的文件名或目录。如果指定的值为目录，那么输出的 bundle 的文件名也将会采用这个目录名。

```
bnd build -classpath bin -noeclipse -output test.jar xyz.bnd
```

wrap (-classpath (<file>(','<file>)*)-output <file|dir> | -properties <file>) * -ignoremanifest? <file>.jar *

wrap 命令用于将一个已存在的 jar 文件转换为符合 OSGi 规范的 Jar 文件或合并到已存在的 bundle 中。如输出的文件不可覆盖，则会生成一个以.bar 结尾的文件。默认的 bnd 文件信息为：

```
Export-Package: *
Import-Package: <packages inside the target jar>
```

如果目标 bundle 有 manifest 文件，那么两者的头信息将合并。

默认值可使用 properties 文件进行覆盖。

- -output - 设置输出的文件名或目录名。
- -classpath - 设置 classpath。
- -properties - 设置 manifest.mf 需要用到的属性文件。
- -ignoremanifest - 不包含目标 Bundle 本身的 manifest 的信息。

```
bnd wrap -classpath osgi.jar *.jar
```

eclipse

打印出当前目录下 Eclipse 工程的信息。

```
bnd eclipse
```

Eclipse 插件

只需将 bnd.jar 文件放入 eclipse/plugins 目录并重启 eclipse 即可完成安装。安装完这个插件后在以.bnd 作为扩展名的文件上点右键即可看到'Make Bundle'菜单，如在 jar 文件上点击右键，则有两个菜单可选择：'Wrap JAR'用于将此 JAR 文件生成一个包含了所有 imports 和所有 exports 的 bundle；'Verify Bundle'用于校验此 JAR 文件是否符合 OSGi Bundle 的规范，如有错误或警告信息均会以对话框的形式显示。

Ant Task

bnd.jar 也可作为 ANT task 使用。下面的例子演示了如何在 ANT 文件中使用 bnd.jar：

```
<target name="build">
<taskdef resource="aQute/bnd/ant/taskdef.properties"
classpath="bnd.jar"/>
<bnd
classpath="src"
eclipse="true"
failok="false"
exceptions="true"
files="test.bnd"/>
</target>
```

在此 ANT task 中可设置以下属性：

属性名	说明
classpath	相对于 ant 工程文件的路径。
eclipse	True 代表解析 Eclipse 的.classpath 文件，也就是和 eclipse 工程采用同样的 classpath；false 代表不解析 Eclipse 的.classpath 文件。
failok	true 代表即使出现错误，也生成 Bundle 文件；false 代表当出现错误时，则不生成 Bundle 文件。
exceptions	false 代表出现错误时，仅显示错误的简要描述；true 代表出现错误时，显示详细的错误堆栈信息。
files	bnd 文件。
sourcepath	源码所在路径。

output	文件输出的路径。

还可在 Ant 中使用以下的 Task:

task 名	类名	属性
bndeclipse	EclipseTask	prefix='project.'
bndexpand	ExpandPropertiesTask	propertyFile='<file>'
bndwrap	WrapTask	jars='<list>', output='<dir>', definitions='<dir>', classpath='<file-list>'

Maven Plugin

在Felix maven plugin中有关于此maven plugin的描述，maven plugin中默认的值为：

Bundle-SymbolicName: <groupId>.<artifactId>

Bundle-Name: project.getName();

Bundle-Version: <version>

Import-Package: *

Export-Package: <groupId>.<artifactId>.* (unless Private-package is set)

Bundle-Description: project.getDescription()

Bundle-License: project.getLicenses()

Bundle-Vendor: project.getOrganization()

Include-Resource: src/main/resources

错误和警告信息

有很多种错误和警告信息，看到信息就知道错误和警告的原因了。