# OSGi模块层规范与原理

## Bundle描述元数据

Bundle的元数据信息定义在/META-INF/MANIFEST.MF文件之中，OSGi规范中明确要求实现框架必须能够正确识别那些被预定义过的标记（在R5.0规范中预定义了28项标记），对于不可识别的标记以及不符合MANIFEST.MF标记格式的内容都要忽略且不能影响Bundle的正常解析。

### 预定义标记

以下列出了MANIFEST.MF文件中常用的预定义标记，除非特别说明，所列举的标记项都是可选的。对于其中某些较重要的标记（如Import-Package和Export-Package等），我们将在后续章节着重分析讲解。

1. Bundle-ActivationPolicy

标记Bundle-ActivationPolicy设置Bundle的加载策略，该参数目前只有一个值：lazy，设置该参数后，Bundle将延迟激活，延迟至有其他的Bundle请求加载该Bundle中的类或资源时它才会被激活，如果不设置这个参数，那么Bundle启动时就会被激活。示例： Bundle-ActivationPolicy: lazy

1. Bundle-Activator

标记Bundle-Activator指明一个Activator 类，在Bundle启动和停止时会分别调用该类的start()和stop()方法，以便执行程序员所希望的动作，该类必须实现org.osgi.framework.BundleActivator接口。Activator类通常用于在Bundle启动时注册和初始化服务，在Bundle卸载时注销这些服务。它很常用，但并不是必须的。

示例： Bundle-Activator: com.acme.fw.Activator

1. Bundle-Category

标记Bundle-Category指明该Bundle的功能类别，可使用逗号分隔多个类别名称。这个功能类别仅供人工分类和阅读，OSGi框架并不会使用它。示例： Bundle-Category: osgi, test, nursery

1. Bundle-Classpath

标记Bundle-Classpath指明该Bundle所引用的类路径，该路径应为Bundle包内部的一个合法路径，如果有多个Classpath，使用逗号分隔。在介绍Bundle类加载过程时我们会详细介绍这个标记。

示例： Bundle-Classpath: /jar/http.jar,.

1. Bundle-ContactAddress

标记Bundle-ContactAddress描述Bundle发行者的联系信息，仅供人工阅读，OSGi框架并不会使用它。示例： Bundle-ContactAddress: 2400 Oswego Road, Austin, TX 74563

1. Bundle-Copyright

标记Bundle-Copyright描述Bundle的版权信息，仅供人工阅读，OSGi框架并不会使用它。示例： Bundle-Copyright: OSGi (c) 2002

1. Bundle-Description

标记Bundle-Description给出关于该Bundle的简短描述信息，仅供人工阅读，OSGi框架并不会使用它。示例： Bundle-Description: Network Firewall

1. Bundle-DocURL

标记Bundle-DocURL给出该Bundle文档的链接地址，仅供人工阅读，OSGi框架并不会使用它。示例： Bundle-DocURL: http:/www.acme.com/Firewall/doc

1. Bundle-Icon

标记Bundle-Icon给出该Bundle的显示图标，图标应为一张正方形的图片，并通过参数size指出图标的宽度。OSGi规范要求实现框架至少要支持PNG图片格式。示例： Bundle-Icon: /icons/acme-logo.png;size=64

1. Bundle-License

标记Bundle-License给出该Bundle的授权协议信息。示例： Bundle-License: http://www.opensource.org/licenses/jabberpl.php

1. Bundle-Localization

标记Bundle-Localization给出该Bundle在不同语言系统下的本地化信息，如果不设置此标记，它的默认值为OSGI-INF/l10n/bundle。

示例： Bundle-Localization: OSGI-INF/I10n/bundle

1. Bundle-ManifestVersion

标记Bundle-ManifestVersion指出该Bundle应遵循哪个版本的OSGi规范，默认值为1。对于OSGi R3规范，该值为1；对于OSGi R4/R5规范，该值为2。也可能在以后的OSGi规范中使用更高的数字,但现在仅允许将它设置为1或2。

示例： Bundle-ManifestVersion: 2

1. Bundle-Name

标记Bundle-Name定义该Bundle的名称。注意该名称只供人工阅读，在Bundle-SymbolicName标记中定义的名称才会作为程序使用的Bundle的唯一标识来使用。根据一般开发习惯，Bundle-Name中所定义的名称会在打包发布时与Bundle-Version一起构成该Bundle的文件名，所以这个名称一般不含空格或其他不能在文件名中出现的字符。

示例： Bundle-Name: Firewall

1. Bundle-NativeCode

如果Bundle中需要使用JNI加载其他语言实现的本地代码，那么必须使用Bundle-NativeCode标记进行说明。这个标记有如下附加参数：

osname：操作系统名称，如Windows等。

osversion：操作系统版本号，如3.1等。

processor：处理器指令集架构，如x86等。

language：遵循ISO编码的语言，如en，zh等。

seleciton-filter：选择过滤器，该值为一个过滤器表达式，指定被选中或未被选中的本地代码。示例： Bundle-NativeCode: /lib/http.DLL; osname = QNX; osversion = 3.1

1. Bundle-RequiredExecutionEnvironment

标记Bundle-RequiredExecutionEnvironment定义该Bundle所需的执行环境，支持多种执行环境的Bundle使用逗号分隔。OSGi在设计上就有非常广泛的应用范围，从嵌入式系统至大型服务器执行环境必然会有许多差异，因此在这个标记中需要指出该Bundle所适合的执行环境。在后续章节中我们还会继续介绍OSGi执行环境。

示例： Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0

1. Bundle-SymbolicName

标记Bundle-SymbolicName给出该Bundle在OSGi容器中的全局唯一标识符。与其他可选标记不同，这个标记没有默认值，并且是Bundle元数据信息之中唯一一个必须设置的标记。程序将基于此标记和版本号在OSGi容器中定位到一个独一无二的Bundle。当且仅当两个Bundle的Bundle-SymbolicName和Bundle-Version属性都相同的时候，它们才是完全相同的，不允许同时安装两个完全相同的Bundle到同一个OSGi容器之中。 Bundle-SymbolicName有以下两个附加参数。 singleton：表示Bundle是单例的。如果OSGi系统中同时存在两个Bundle-SymbolicName相同的（当然，要求Bundle-Version不相同，否则是不可能同时存在的）单例Bundle，那么仅有其中一个会被解析。如果其中一个没有声明为单例Bundle，则不会受到另外一个单例Bundle的影响，默认值为false。 fragment-attachment：定义Fragment Bundle是否能附加到该Bundle之上。允许值为always、never和resolve-time，含义为允许附加、禁止附加和只允许在解析过程中附加，默认值为always，即允许附加。

示例： Bundle-SymbolicName: com.acme.daffy

1. Bundle-UpdateLocation

标记Bundle-UpdateLocation给出Bundle的网络更新地址。如果Bundle需要更新版本，将使用这个地址。

示例： Bundle-UpdateLocation: file://www.acme.com/Firewall/bundle.jar

1. Bundle-Vendor

标记Bundle-Vendor给出该Bundle的发行者信息。示例： Bundle-Vendor: OSGi Alliance

1. Bundle-Version

标记Bundle-Version给出该Bundle的版本信息，默认值为“0.0.0”。注意，这项信息并不是仅供人工阅读的，“版本”在OSGi中是一项受系统管理的信息。维护一个Bundle的不同版本也是运行OSGi框架的重要特征之一，当一个Bundle依赖另一个Bundle时，经常需要指明它依赖的是什么版本范围内的Bundle。版本号是有序的，在Symbolic-Name相同的前提下，两个Bundle的版本可比较大小。完整的版本号会由“主版本号（Major）”+“副版本号（Minor）”+“微版本号（Micro）”+“限定字符串（Qualifier）”构成。示例： Bundle-Version: 22.3.58.build-345678 根据一般的开发习惯，上述4项版本号约定俗成地表示如下含义。

主版本号：表示与之前版本不兼容的重大功能升级。

副版本号：表示与之前版本兼容，但可能提供新的特性或接口。

微版本号：表示API接口没有变化，只是内部实现改变，或者修正了错误。

限定字符串：通常用于表示编译时间戳或者编译次数。

在比较版本大小时，从前往后逐项（含限定字符串）进行比较，当且仅当4个比较项都对应相等，两个Bundle的版本才相等，否则以第一个出现差异的版本号的大小决定整个Bundle版本的大小。示例： 1.2.3 < 3.2.1 < 4.0 有一点必须注意，对于限定字符串的处理，OSGi和Maven是恰恰相反的，在Maven里，版本“1.2.3.2012”<=“1.2.3”，但在OSGi里则是版本“1.2.3.2012”>=“1.2.3”。

1. DynamicImport-Package

标记Dynamic Import-Package描述运行时动态导入的Package。Package的导入和导出构成了OSGi多模块之间的组织协作关系，这是一个相对复杂而又很重要的内容，我们将在后续章节中专门介绍。示例： DynamicImport-Package: com.acme.plugin.\*

1. Export-Package

标记Export-Package描述被导出的Package。导入导出Package是模块层的核心功能，该内容将在后续章节中具体介绍。

示例： Export-Package: org.osgi.util.tracker;version=1.3

（22）Export-Service

标记Export-Service描述被导出的服务，这个标记在OSGi规范中已经被声明为Deprecated了，不推荐继续使用此标记。示例： Export-Service: org.osgi.service.log.LogService

1. Fragment-Host

当该Bundle是一个Fragment Bundle时，标记Fragment-Host指明它的宿主Bundle。示例： Fragment-Host: org.eclipse.swt; bundle-version="[3.0.0,4.0.0)"

1. Import-Package

标记Import-Package描述该Bundle需要导入的Package。导入导出Package是模块层的核心功能，该内容将在后续章节中具体介绍。

示例： Import-Package:org.osgi.service.io;version=1.4

1. Import-Service

标记Import-Service描述导入的服务。这个标记在OSGi规范中已经被声明为Deprecated了，不推荐继续使用此标记。示例： Import-Service: org.osgi.service.log.LogService

1. Provided-Capability

标记Provided-Capability描述该Bundle提供的服务特性（Capability）。服务特性是在OSGi R4.3规范中加入的新概念。在此之前，Bundle只能通过Bundle-RequiredExecution-Environment来声明所需的执行环境；但是在某些场景下一些执行环境特性是由其他Bundle提供的，这样依赖运行环境来描述所需特性就受到限制了。因此在R4.3规范中加入了Provided-Capability和Require-Capability来声明Bundle所需要和能够提供的特性。示例： Provided-Capability: com.acme.dict; from=nl; to=de; version:Version=1.2

1. Require-Capability

标记Require-Capability描述该Bundle所需要的服务特性。示例： Require-Capability: osgi.ee; filter:="(&(osgi.ee=AcmeMin)(version=1.1))"

1. Require-Bundle

标记Require-Bundle描述该Bundle所依赖的其他Bundle，一旦声明了依赖某个Bundle，就意味着可以直接使用所有从这个Bundle中导出的Package。示例： Require-Bundle: com.acme.chess

## Bundle的组织与依赖

既然是以模块化方式开发一个系统，那么必不可少的步骤是根据业务和技术的需要，将系统划分为多个模块，通过这些模块互相协作完成系统的功能。系统中绝大部分模块都不是孤立的，通常会依赖其他模块所导出的某些Package，又会被另外一些模块所依赖。这种依赖关系在元数据配置中简单体现为Import-Package、Export-Package和Require-Bundle标记的配置，使用起来并不算复杂，但是各个OSGi框架的实现者都要花费大量心思在组织Bundle与管理依赖上，如何查找最合适的Bundle、如何处理循环依赖关系、如何导入导出Bundle中的类和资源等一系列问题都是需要OSGi框架去解决的。

### 导出和导入Package

Export-Package用于声明Bundle要导出哪些Package，Import-Package用于声明Bundle需要导入哪些Package。这两个标记最简单的方式是直接跟随导入或导出的Package名称，如果导入或导出多个Package，则使用逗号分隔，如下所示： Export-Package: org.osgi.service.io, org.osgi.service.web Import-Package: org.osgi.service.io, org.osgi.service.web

这种写法是OSGi中导入导出Package的基本用法，也是最常见用法，已经能满足相当多的应用场景，但是在开发过程中总会遇见各种不同的需求，要根据特定规则去选择适合的Package。下面对OSGi规范中定义的几种对导入导出进行筛选的过滤方式进行介绍。

#### 根据类名过滤

如果仅在Package层次上控制导出的粒度不够精细，无法满足应用需求，那么可以使用附加参数include和exclude进一步描述要导出Package中哪一部分内容。比如，只希望导出org.osgi.service.io包中命名以“IO”开头的接口，不导出实现类（假设实现类都以Impl结尾），那么可以这样写： Export-Package: org.osgi.service.io;include:="IO\*"; exclude:="\*Impl"

include和exclude参数的具体使用方法如下：

附加参数include用于列举需要导出Package中哪些类，类名使用逗号分隔，可以使用通配符。如果加入这个参数，那么只有符合规则的类才会被导出。

附加参数exclude用于列举禁止导出Package中哪些类，类名使用逗号分隔，可以使用通配符。如果加入这个参数，那么只要符合规则的类就不会被导出。

include和exclude的限制是在导出时处理的，导入时无需对应用做任何特殊声明，Import-Package标记也无法与这两个参数搭配使用。例如与前面配对的导入声明依然为： Import-Package: org.osgi.service.io

在导出时建议附加参数include和exclude一起使用。

如：

|  |
| --- |
| Manifest-Version: 1.0  Bundle-ManifestVersion: 2  Bundle-Name: VersionFilter  Bundle-SymbolicName: com.chinasofti.samples.osgi.importexport.version.filter.sayhello  Bundle-Version: 1.0.0.qualifier  Bundle-Activator: com.chinasofti.samples.osgi.importexport.sayhello.Activator  Bundle-Vendor: CHINASOFTI  Bundle-RequiredExecutionEnvironment: JavaSE-1.6  Import-Package: org.osgi.framework;version="1.3.0"  Export-Package: com.chinasofti.samples.osgi.importexport.version.filter.sayhello.api;exclude:=”\*Impl”;include:=”I\*” |

#### 根据版本过滤

在OSGi系统中，同一个名称的Package可能存在多个不同版本。假设Bundle C开发时引入了Spring 2.0版的Package，并且使用了某些只在这个版本私有的特征，而Bundle D开发时使用的是Spring 3.0版的Package，那么从这个系统中导出Spring的Bundle就必须明确指明Spring的版本号，以便导入时区分。示例如下：

Bundle A（导出Spring2.0）:

Export-Package: org.springframework.core; version="2.0.0"

Bundle B（导出Spring3.0）:

Export-Package: org.springframework.core; version="3.0.0"

对应的，导入时也要指明版本，准确地说应指明某个版本范围，例如：

Bundle C（导入Spring2.x版）:

Import-Package: org.springframework.core; version="[2.0.0,2.9.9]"

Bundle D（导入Spring3.0以上版本）:

Import-Package: org.springframework.core; version="3.0.0"

这里要注意我们导入的是“版本范围”而不是某个具体的“版本”，例如示例中Bundle D的Import-Package写法，只声明了version="3.0.0"的含义并不是“只导入”3.0.0版本的Package，而是导入3.0.0或以上版本的Package，因为这更符合一般的开发使用场景。如果需要指定只导入3.0.0版的Package，需要这样写：

Bundle E（只导入Spring3.0版本）:

Import-Package: org.springframework.core; version="[3.0.0,3.0.0]"

version参数的具体使用方法是：在导出Package时，此参数声明导出Package的版本号，如果不设置，默认值为0.0.0；在导入Package时，此参数声明要导入Package的版本范围，如果不设置，默认值为[0.0.0, ∞)。在声明版本范围时，方括号“[”和“]”表示“范围包含此数值”，圆括号“（”和“）”表示“范围不含此数值”。

在OSGi R3.x及之前的版本中，version标记原本叫做specification-version，在R4.x规范中新增了version标记，但依然保留了specification-version这个别名，但是已将它声明为Deprecated，如果同时设置了version和specification-version的值，那么这两个值必须相等。

#### 根据提供者过滤

OSGi还允许开发人员根据Bundle名称、版本等信息来过滤Package，这种过滤方式在规范中被称为“选择提供者（Provider Selection）”。由开发人员明确指明导入的Package必须来自某个提供者的做法在实际开发之中是很少见的，它会增加系统的兼容性风险和人为的不确定因素。就如同我们组装电脑选择显示卡一样，理性的选择方式是根据性能需求确定显示芯片、显存大小和位宽等参数，再来选择合适的产品，而不是明确要求必须选择某公司出产的某型号产品。

根据提供者过滤一般使用在测试Bundle的时候，Import-Package标记提供了两个附加参数bundle-symbolic-name和bundle-version来完成这项功能，它们分别用于对Bundle的名称和版本进行过滤，示例如下：

Bundle A：

Bundle-SymbolicName:BundleA

Import-Package: com.acme.foo; bundle-symbolic-name="BundleB";

bundle-version="[1.4.1,2.0.0)"

Bundle B

Bundle-SymbolicName:BundleB

Bundle-Version:1.4.1

Export-Package: com.acme.foo

上面配置指明一定要是来自于名称为“BundleB”并且版本在1.4.1至2.0.0之间的com.acme.foo包才会被选择。

bundle-symbolic-name和bundle-version参数的具体使用方法如下。

附加参数bundle-symbolic-name：参数值为某个Bundle的名称，只有符合该名称的Bundle所导出的Package才会被导入。

附加参数bundle-version：参数值为导入Bundle（注意不是Package）的版本范围，只有版本在该范围之内的Bundle所导出的Package才会被导入。

#### 根据属性过滤

导入和导出的Package除了使用include、exclude对类名进行过滤，使用version对版本进行过滤和使用bundle-symbolic-name、bundle-version对提供者信息进行过滤外，还有第四种方式：使用自定义的扩展属性进行过滤。包名、类名和版本这几项信息都是很客观的数据，在代码开发完成那一刻就已确定下来，不可能随意改变。而自定义的扩展属性可以满足某些根据开发人员自己加入的属性进行过滤的需求。示例如下： Export-Package: org.osgi.simple;filter="true";

在导出org.osgi.simple时加入了自定义属性filter，它的值为“true”（属性值按照字符串处理），那下面三句Import-Package中，只有第Bundle B、C可以成功导入前面发布的org.osgi.simple包，而Bundle A中由于自定义属性冲突导致而导致匹配失败，示例如下：

Bundle A：

Import-Package: org.osgi.simple; filter="false"

Bundle B：

Import-Package: org.osgi.simple; filter="true"

Bundle C

Import-Package: org.osgi.simple;

注意，Bundle C虽然没有声明自定义属性filter，但在默认情况下这并不会产生匹配冲突。如果要改变这种情况，可以使用mandatory附加参数，强制要求必须存在扩展属性才能成功匹配，示例如下： Export-Package: org.osgi.simple; filter="true";mandatory:="filter"

这样，在下面两句Import-Package中，只有第一句能成功导入前面发布的“org.osgi.simple”包，因为它发布时filter属性被声明为“必须”的，示例如下： Bundle A： Import-Package: org.osgi.simple; filter="true"

Bundle B：

Import-Package: org.osgi.simple

如果没有在mandatory中指定属性名称，那这种属性被默认为“optional”，即可选的，在导入时没有声明这个属性也不影响正常导入。

mandatory参数的具体使用方法：只适用于Export-Package，用于声明哪些属性是必选的，只有导入Package时提供了必选的属性，才能正确匹配到导出的Package。多个属性用逗号分隔，用双引号包裹。

#### 可选导入与动态导入

在大多数情况下，导入某个Package，就说明当前这个Bundle的正常运行是必须依赖导入的Package的，比如一个基于Spring开发的程序，没有导入Spring的Package就肯定无法运行。但还有另外一些场景，导入某个Package是为了实现一些不影响Bundle正常运行的附加功能，比如为某个英文软件开发了一个实现中文语言支持的插件，不导入这样的Package也不应当影响整个系统正常运行，只不过软件仍以英文形式显示而已。

Import-Package标记也可以很好地支持上述需求，它有一个名为resolution的附加参数，用于说明一个依赖的Package是否是必需的。示例如下：

Bundle A：

Import-Package: org.osgi.simple;resolution:="mandatory"

Bundle B：

Import-Package: org.osgi.simple

Bundle C：

Import-Package: org.osgi.simple; resolution:="optional"

在上面例子中，Bundle A和Bundle B的Import-Package是等价的，因为resolution的默认值就是mandatory。在这个设置下，如果没有任何Bundle导出org.osgi.simple包，那么Bundle A 和Bundle B将会解析失败。而Bundle C不会，因为它明确指出了这个Package的导入是可选的（resolution:="optional"）。

resolution参数的具体使用方法是这样的：附加参数resolution只适用于Import-Package标记，用于确定某个Package是可选的还是必须的。

还有一种场景，可能某个被导入的Package确实是必需的，但是提供这个Package的Bundle并不会在系统启动时就被安装。在这种情况下，只有真正使用到这个Package的类时，才会去导入这个Package，而不是在启动时就查找是否有提供Package的Bundle存在。

DynamicImport-Package标记可以处理这样的需求，这个标记的语义和Import-Package标记很类似，不同点在于DynamicImport-Package标记不在Bundle的解析阶段进行处理，无论它要求导入的Package是否存在，都不会影响Bundle的解析，示例如下：

Bundle A：

Import-Package: org.osgi.simple

Bundle B：

DynamicImport-Package: org.osgi.simple

如果没有任何Bundle提供org.osgi.simple包，那么Bundle A将无法解析，而Bundle B不受影响；但是如果真正使用到org.osgi.simple中的类时还是没有任何Bundle可提供，那么Bundle B依然要抛出ClassNotFoundException异常。

DynamicImport-Package也可是使用version附加参数来过滤导入Package的版本，在导入Package时声明属性，以便符合Package导出时mandatory参数中声明必须存在属性。

与Import-Package有所不同的是，DynamicImport-Package可以使用通配符，举一个极端的例子： DynamicImport-Package: \*

如果某个Bundle的元数据信息中有上面这行定义，那么它就成了一个“管理员级别”的Bundle，它可以访问整个系统中所有被导出过的Package。这样做很方便，却违背了OSGi中提倡的封装与按需使用的原则，会带来许多遗患，因此这并不是一种好的使用方式。

动态导入和可选导入实现的功能有些类似，它们的共同特征是在Bundle解析期间即使找不到要导入的依赖，也不会导致解析失败。它们的区别是，动态导入每次加载包中的类都会尝试去查找动态导入包，而可选导入包只有在Bundle解析时才进行连接尝试。

#### 导出Package的依赖限制

如果导出Package的全部依赖都集中在一个Bundle之中，那么这个Package的导出是完全不受限制的。但是如果要导出的Package中有某些类还依赖于其他Bundle所提供的类，比如从其他Bundle所导出的Package中的类继承，或者其他Bundle的类出现在方法的声明中，在这种情况下，要导出这个Package就会受到依赖限制，必须先保证依赖的Package是可用的，才能保证导出的Package是可用的。

这种Package之间的关系可以通过在Export-Package标记中的uses附加参数来描述。例如：包org.osgi.service.http使用了包javax.servlet.http中的API，因此这两个包存在依赖关系。在导出org.osgi.service.http的Export-Package标记中就应当包含值为javax.servlet的uses参数，如下所示： Export-Package: org.osgi.service.http;uses:="javax.servlet.http"

当一个系统中同时存在不同版本的Package时，uses参数对于协调依赖关系很有用。比如上面例子中的javax.servlet.http包同时存在2.4和2.1两个版本，而Bundle A导入的是2.4版，如下所示：

Bundle A：

Export-Package: org.osgi.service.http;uses:="javax.servlet.http"

Import-Package：javax.servlet.http; version="2.4"

对于任意一个Bundle，只要导入Bunlde A中发布org.osgi.service.http包，同时又导入了javax.servlet.http包，即使在导入的时候不明确指明其版本，OSGi实现也必须保证它只会使用2.4版的javax.servlet.http，如下所示： Bundle B： Import-Package：org.osgi.service.http // 由Bundle A提供 ,javax.servlet.http // 这里默认会导入2.4版的Package

uses参数的具体使用方法是：附加参数uses只适用于Export-Package标记，用于说明导出Package的依赖关系。如果同时依赖多个Package，使用逗号分隔，并用双引号包裹。

当OSGi容器中的Bundle不断增加，依赖关系逐渐变得复杂时，uses参数就是协调依赖的必要手段，也是配置OSGi依赖的难点之一。在2.4.2节中我们还会对如何使用uses参数协调Package依赖约束进行更详细介绍。

#### 导入整个Bundle

前面几项介绍的都是Package级别的导入、导出和依赖，OSGi还可以支持Bundle级别的依赖关系。我们可以使用Require-Bundle标记来声明要导入这个Bundle，如果成功导入了这个Bundle，那就意味着导入了这个Bundle中所有声明为导出的Package。

Require-Bundle标记后面应跟随一个或多个其他Bundle的名称（由Bundle-SymbolicName定义的名称），多个名称之间使用逗号分隔，如下所示：

Require-Bundle：BundleA、BundleB、BundleC

如果导入了某个Bundle，就可以重复导出该Bundle中已导出过的Package，就如同导出自己的Package一样，例子如下所示：

Bundle A：

Require-Bundle:BundleB

Export-Package: p

Bundle B:

Export-Package: p;partial=true;mandatory:=partial

这时如果有Bundle C要导入Package p，而又没有声明属性partial，那它将会从Bundle A中导入Package p。实际上Bundle A只承担了转发的工作，真正的Package p在Bundle B之中。这种情况属于拆分包的一个特例，应尽可能避免。

注意：拆分包（Split Packages）是指OSGi容器中有两个或两个以上的Bundle导出了相同名称的Package。容器存在拆分包会令包导入过程变得复杂，因为只带有包名的Import-Package标识无法保证能正确导入到所需的Package。必须通过过滤或者使用Require-Bundle才能导入正确的Package。

与Import-Package类似，Require-Bundle标记也有附加参数bundle-version和resolution。bundle-version用于过滤导入Bundle的版本，默认值为[0.0.0,∞)，即不限制版本范围。resolution用于确定导入的Bundle是否是必需的，可选值为mandatory和optional，默认值为mandatory，含义与Import-Package标记的resolution参数相同，这里就不再详细举例介绍了。

在默认设置下，导入了某个Bundle，仅表示在本Bundle中可以访问被导入Bundle中声明导出的Package，除非明确用Export-Package声明过，否则这些Package在本Bundle中默认不会再次被导出。如果有必要，可以使用Import-Package标记的visibility附加参数来改变这种行为，示例如下：

BundleRequire-Bundle:BundleB;visibility:=reexport

Bundle B

Export-Package: org.osgi.service.http

在上面例子中由于明确设置了visibility的值为reexport，Bundle A中就会重复导出Bundle B的声明导出的包，即org.osgi.service.http。

visibility参数的具体使用方法是：附加参数visibility仅用于Require-Bundle标记，描述来自被导入Bundle中的Package是否要在导入Bundle中重新导出。默认值为private，代表不会在导入Bundle中重新导出。如果值为reexport，则说明需要重新导出。

在元数据信息中可以同时使用Import-Package和Require-Bundle标记来获取Bundle所需的依赖包，但是如果某个依赖的Package同时在Import-Package列表和Require-Bundle的Bundle中存在，那么OSGi实现框架必须保证要以Import-Package列表中的包优先。

使用Require-Bundle有时候确实会获得一些便利，但从长远来看，依赖的粒度越小越好。依靠Require-Bundle来处理依赖关系并非一种好的开发实践，甚至可能会带来一些令人头痛的问题，我们将在2.4.2节中通过实际例子来介绍Require-Bundle的缺陷。

### 约束规则与示例

导入和导出Package并不总是一帆风顺的，随着系统复杂性的增加，模块数量不断变多，依赖关系随之变得越来越错综复杂，尤其有拆分包或在多个不同版本的Package同时存在和出现循环依赖等情况时，理解依赖的约束规则就很重要了，本节将介绍这方面的内容。

1.图示

在本书后续章节中，我们将会以图示来表示各个Bundle、Package以及它们之间的依赖关系，如图2-6所示。这些图示最初定义在OSGi规范之中（除了最后一个“解析失败的Bundle”，这个图示是笔者为了讲解方便自行定义的），因此在本书之外的其他OSGi文档也会遇到类似的图示。

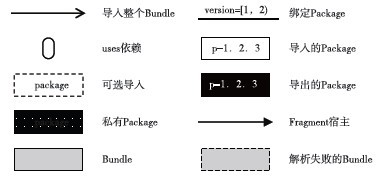


图2-6　OSGi标准图示与标记

下面是一个如何使用图示来表示Bundle、Package及其依赖关系的例子，假设存在A、B、C三个Bundle，其定义如下所示：

Bundle A:

Import-Package: p; version="[1,2)"

Export-Package: q; version=2.2.2; uses:=p

Require-Bundle: C

Bundle B

Export-Package: p; version=1.5.1

Bundle C

Export-Package: r

这3个Bundle以及它们之间的依赖关系可以使用图2-7来表示。

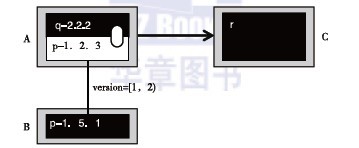


图2-7　图示示例

2.多版本Package依赖选择

前面介绍过如何使用Export-Package标记uses附加参数来协调在同一个OSGi系统中选择多个不同版本Package的问题。本节我们继续沿用前面的例子深化理解多版本Package共存时的约束。Bundle A、B、C、D的定义如下：

Bundle A：

Import-Package：org.osgi.service.http

Import-Package：javax.servlet.http

Bundle B：

Export-Package: org.osgi.service.http;uses:="javax.servlet.http";

Import-Package：javax.servlet.http; version="2.4"

Bundle C:

Export-Package: javax.servlet.http; version="2.1"

Bundle D

Export-Package: javax.servlet.http; version="2.4"

对于Bundle A来说，由于没有指明所需要的javax.servlet.http的版本号，Bundle C和Bundle D对它来说都是满足需求的。但是它依赖的org.osgi.service.http来自于Bundle B之中，Bundle B明确声明了org.osgi.service.http包要用到版本为2.4的javax.servlet.http的类，在这层隐含限制下，Bundle A就不允许从Bundle C获取2.1版的javax.servlet.http了，如图2-8所示。

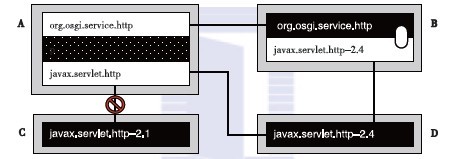


图2-8　多版本Package依赖选择示例(1)

如果Bundle A在导入javax.servlet.http的时候声明了只支持版本2.1（在导入时加入version="[2.1.0,2.1.0]"），其他条件不变，那么就会发生版本冲突，Bundle A在这种设置下是无法解析通过的，但是不会影响到Bundle B、Bundle C和Bundle D，它们依然能够解析成功，如图2-9所示。

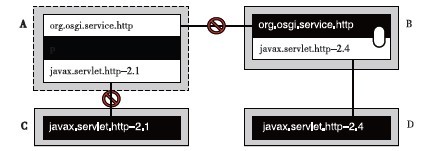


图2-9　多版本Package依赖选择示例(2)

遇到上述Bundle解析问题，OSGi实现框架会抛出一个BundleException异常，可能的异常信息（根据实现框架而定，这里以Equinox框架为例）类似如下所示：

org.osgi.framework.BundleException:

The bundle "BundleA\_1.0.0.qualifier [8]" could not be resolved.Reason:Package

uses conflict:Import-Package: org.osgi.service.http; version="0.0.0"

再把场景修改一下，假设在Bundle B导出org.osgi.service.http时没有使用uses参数，在实际代码之中org.osgi.service.http也不会依赖javax.servlet.http中的类，其他条件不变，那么Bundle A、B、C、D全部能正常共存。因为Bundle A认为在使用org.osgi.service.http时不会遇到任何与javax.servlet.http有依赖关系的类，这样即使Bundle B中有其他的包需要2.4版的javax.servlet.http支持，也对Bundle A没有任何影响，因为根本不会使用到uses参数，如图2-10所示（注意Bundle B中已经没有了表示uses参数的图示）。

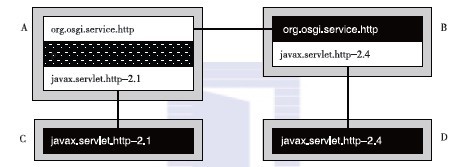


图2-10　多版本Package依赖选择示例(3)

我们继续假设，如果仅在元数据信息中没有使用uses参数描述出org.osgi.service.http对javax.servlet.http的依赖关系，而在实际代码之中它们却有依赖关系，那会怎样呢？OSGi规范并未说明这种元数据与实际代码不符的场景应当如何处理，因此这时不同实现框架表现出来的行为是不可预测的。在Equinox的实现中，Bundle A在实际使用到Bundle B发布出来的org.osgi.service.http中的类时并不会报错，但是没有遵循Bundle B的定义采用2.4版的javax.servlet.http包，而是采用了在Bundle A中引用的2.1版。

3.相同Package循环导出

我们知道，即使不考虑OSGi环境，在普通Java系统中，一个Package中的各个类也可以分布于不同JAR包中，这种情况很常见。因此OSGi必须允许多个Bundle导出相同的Package，即出现拆分包的情况。但是如果不加约束，在某些场景中会出现一些逻辑矛盾，例如接下来要介绍的相同Package循环导出的例子。Bundle A、B、C、D定义如下：

Bundle A：

Require-Bundle:BundleB,BundleC

Export-Package: p

Bundle C：

Require-Bundle:BundleD

Export-Package: p

Bundle B

Export-Package: p

Bundle D:

Export-Package: p

相同Package循环导出的图形化描述如图2-11所示。

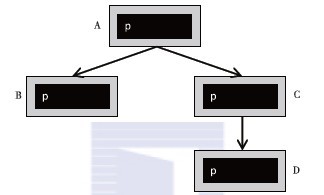


图2-11　相同Package循环导出(1)

这4个Bundle都共同导出了一个包p，如果这时某个Bundle向Bundle A请求加载p中的类，Bundle A会根据深度优先的规则去查找这个类（OSGi的类加载器规则决定了加载请求优先委派给导入Bundle的类加载器，后面我们再详细介绍这一点），即查找顺序是：B->D->C->A。在这个基础上，再让Bundle D依赖Bundle A（加入Require-Bundle: BundleA），即变为如图2-12所示的状态。

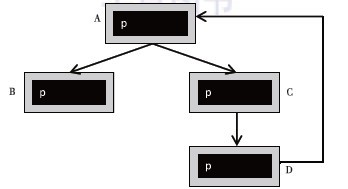


图2-12　相同Package循环导出(2)

在这种场景下，Bundle A、C、D构成了一个循环，如果仍然按照深度优先搜索，就会出现B->D->C->A->B->D->C->A->……的无限循环。为了避免出现这样的循环，OSGi规范明确要求实现框架必须对每个搜索过的Bundle记录状态，保证每个Bundle只被搜索一次。因此对于图2-11所示的情况，搜索顺序依然是B->D->C->A。

4.Require-Bundle带来的问题

OSGi提倡使用Import-Package和Export-Package来构成模块之间的依赖关系，不提倡使用Require-Bundle，从实践经验来说，依赖的粒度总是越小越好。Bundle级别的导入还可能会由于Package重复导入和导出而带来一系列逻辑缺陷和性能问题。

对于“拆分包”的场景——如果Bundle A把它导入的Bundle B中某个Package使用的Export-Package重新导出就会形成拆分包，假如这时如果通过Bundle A加载这个Package，类搜索的路径就会变得更长，类加载器中抛出并接住ClassNotFoundException异常的次数会变得更多，这样就会带来额外的性能开销。

另外，Require-Bundle会让导出Package的过程变得更为复杂，可能导致Bundle开发人员不可预料地声明改变。因为在Require-Bundle中的声明的visibility可能会令Bundle的调用者无所适从。例如，有如下定义：

Bundle A:

Require-Bundle:BundleB;visibility:=reexport,BundleC;visibility:=private

Bundle B:

Export-Package: p

Bundle C:

Export-Package: p

上面的定义会导致Bundle A只导出“一部分”的Package p。根据定义内容，Bundle A由于在导入Bundle B时声明了“visibility:=reexport”，暗示了它会重复导出包p，但这又与Bundle A直接声明“Export-Package p”有所不同。当Bundle A收到类查找的请求时，类查找顺序应为B->C->A，如果Bundle B中的Package p确实有这个类，那么通过visibility:=reexport还是Export-Package p实现导出并没有区别；但是如果Bundle B中的Package p没有请求的类，被请求的类在Bundle A中或在者Bundle C的Package p中，那么这时OSGi框架就会拒绝提供这些类，这是因为这些类在未被导出的私有Package中或在明确声明为visibility:=private的Bundle中。因此Bundle A只导出了“一部分”的Package p。这种导出的复杂性从Bundle A、B、C内部定义来还能够解释得过去，但是从在外部调用者的角度看，Bundle A只导出“一部分”的Package p就显得很别扭了。上面示例对应的图形表述如图2-13所示。

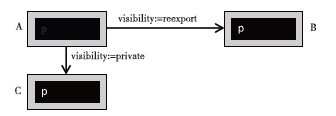


图2-13　Require-Bundle导致的Package导出问题

### 校验Bundle有效性

要确定某个JAR包是否是一个合法的OSGi Bundle，首先要根据其元数据信息中的Bundle-ManifestVersion（注意，不是ManifestVersion，ManifestVersion是JAR文件规范中定义的）来确定元数据信息的版本号。如果在元数据信息中没有指定Bundle-ManifestVersion，那么其默认属性是1，表示遵循OSGi R3规范；如果Bundle-ManifestVersion为2，则代表遵循OSGi R4/R5规范；OSGi R4规范中添加了一些特定的标记。在将Bundle安装到OSGi系统的时候，必须对其进行有效性校验。下面列举一些（不完全包括）会导致Bundle有效性校验失败的问题。

无法根据Bundle-RequireExecutionEnvironment的值找到一个可匹配执行环境。

在Bundle中没有设置Bundle-SymbolicName标记。

存在重复的标记。

对某个Package重复导入。

Bundle导入或导出了以java.\*开头的Package。

在Export-Package标记中用mandatory参数指明了某些强制属性，却没有定义这些属性。

尝试安装某个和OSGi框架中某个已经安装好的Bundle具有同样名称和版本的Bundle。

将某个Bundle更新为和OSGi框架中一个已经安装好的Bundle具有同样名称和版本的Bundle。

元数据信息中存在语法错误（例如，不合法的版本格式或Bundle名称）。

同时使用Specification-version和version参数，但是为它们指定了不一样的值，例如：

Import-Package p;specification-version=1;version=2

这将导致一个错误。而下面的指定：

Import-Package p;specification-version=1, q;version=2

不会导致错误。

在元数据信息中列出了权限文件：OSGI-INF/permission.perm，但此文件不存在。

Bundle-ManifestVersion的值不等于1或2，在今后的OSGi规范中另有规定的情况除外。

# OSGi生命周期层规范与原理

OSGi规范把模块化“静态”的一面，比如如何描述元数据、如何加载模块中的类和资源等内容定义于模块层规范之中；而把模块化“动态”的一面，比如模块从安装到解析、启动、停止、更新、卸载的过程，以及在这些过程中的事件监听和上下文支持环境等定义于生命周期层（Life Cycle Layer）之中。

## Bundle标识

Bundle的唯一标识是由Bundle-SymbolicName和Bundle-Version标记共同构成的。对于生命周期层，我们依然可以采用Bundle-SymbolicName和Bundle-Version标记来确定唯一的Bundle。不过，基于API使用方便的考虑，在运行期还可以采用其他Bundle标识进行定位，包括：

* Bundle ID（Bundle Identifier）。Bundle ID是运行期最常用的标识符，尤其是在Equinox Console的命令中。它是由OSGi框架自动分配的一个长整型数字，在Bundle整个生命周期内（包括Bundle更新、卸载之后）都不会改变，甚至在OSGi框架重启后都能保留下来。Bundle ID是在Bundle安装过程中由OSGi框架根据Bundle安装时间的先后次序，由小到大进行分配的。在代码中可以通过Bundle接口的getBundleId ()方法来获取当前Bundle的ID。
* Bundle位置（Bundle Location）。Bundle位置是OSGi容器在Bundle安装过程中分配给Bundle的定位字符串。这个字符串通常是该Bundle的JAR文件地址，但是这并不是强制性的。在一个OSGi容器中，每个Bundle的定位字符串都必须是唯一的，即使Bundle更新时改变了JAR文件的路径，也不会修改这个定位字符串，所以它可以唯一确定一个Bundle。在代码中我们可以通过Bundle接口的getLocation()方法来获取一个Bundle的定位字符串。
* Bundle符号名称（Bundle Symbolic Name）。前面介绍过，Bundle的符号名称由开发人员设定，保存于Bundle元数据信息之中。它是静态的信息，在Bundle打包发布的那一刻它就被确定下来，不会因使用了不同的OSGi框架而有所不同（前面的Bundle ID和Bundle Location是由OSGi框架所决定的）。Bundle的版本与符号名称一起可以唯一定位一个Bundle，在代码中可以通过Bundle接口的getSymbolicName()方法获取当前Bundle的符号名称，通过getVersion()方法获取Bundle的版本号。

我们可以写一小段简单的代码，在Equinox框架中运行查看这3个唯一标识，示例如下：

|  |
| --- |
| System.out.println("Location:" + bundleContext.getBundle().getLocation());  System.out.println("ID:" + bundleContext.getBundle().getBundleId());  System.out.println("SymbolicName:" + bundleContext.getBundle().getSymbolicName());  输出结果为：  Location: initial@reference:file:../WorkSpaces/equinox/BundleA/  ID: 1  SymbolicName: Bundle |

## Bundle状态及转换

“状态”是Bundle在运行期的一项动态属性，不同状态的Bundle具有不同的行为。生命周期层规范定义了Bundle生命周期过程之中的6种状态，分别是：UNINSTALLED（未安装）、INSTALLED（已安装）、RESOLVED（已解析）、STARTING（启动中）、STOPPING（停止中）、ACTIVE（已激活），它们的含义为：

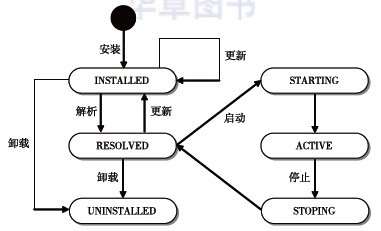
* UNINSTALLED，未安装状态。处于未安装状态的Bundle导出的Package和包含的其他资源都是不可使用的。但是OSGi容器中代表这个Bundle的对象实例仍然可以操作，在某些场景，比如自省（Introspection）中这个对象还是可用的。UNINSTALLED的状态值为整型数1。
* INSTALLED，已安装状态。Bundle处于已安装状态就意味着它已经通过OSGi框架的有效性校验（有效性校验的内容可参见第2章中模块层定义的介绍）并产生了Bundle ID，但这时还未对它定义的依赖关系进行解析处理。INSTALLED的状态值为整型数2。
* RESOLVED，已解析状态。Bundle处于已解析状态说明OSGi框架已经根据元数据信息中描述的依赖关系成功地在类名空间中找到它所有的依赖包，这时它导出的Package就可以被其他Bundle导入使用。RESOLVED的状态值为整型数4。
* STARTING，启动中状态。Bundle处于启动中状态说明它的BundleActivator的start()方法已经被调用，但是还没执行结束。如果start()方法正常执行结束，Bundle将自动转换到ACTIVE状态； 否则，如果start()方法抛出了异常，Bundle将退回到RESOLVED状态。STARTING的状态值为整型数8。
* STOPPING，停止中状态。Bundle处于停止中状态说明它的BundleActivator的stop()方法已经被调用，但是还没执行结束。无论stop()是正常结束还是抛出了异常，在这个方法退出之后，Bundle的状态都将转为RESOLVED。STOPPING的状态值为整型数16。
* ACTIVE，Bundle处于激活状态，说明BundleActivator的start()方法已经执行完毕，如果没有其他动作，Bundle将继续维持ACTIVE状态。ACTIVE的状态值为整型数32。

OSGi规范定义的部分API接口对Bundle的状态有要求，只有处于特定状态的Bundle才能调用这些API。在代码中可以使用Bundle接口中的getState()方法来检测Bundle目前的状态，示例如下：

|  |
| --- |
| import static org.osgi.framework.Bundle.\*;  Bundle bundle = bundleContext.getBundle();  if((bundle.getState() & (STARTING | ACTIVE | STOPPING)) != 0){  // 进行某些只允许在STARTING、ACTIVE、STOPPING状态下执行的动作  } |

目前Bundle的状态值采用由整型数存储的位掩码（Bit-Mask）来表示，而没有直接采用JDK 1.5后提供的枚举类型。从OSGi R4.3规范开始，规范中定义的标准接口中已经开始使用了部分JDK1.5的语言特性，主要是泛型支持，在对应的实现R4.3规范的Equinox 3.7框架的元数据信息中，运行环境项也升级到了“J2SE-1.5”。不过，其他JDK 1.5中的新语言特性，如注解和枚举，仍然没有被采用。这主要是考虑到已有代码的兼容性，Java基于擦除法实现的范型可以在编译时直接使用Javac的“-target jsr14”转换出能部署在JDK 1.4环境下的代码，而注解、枚举等还需要其他Backport运行库支持才可以。

Bundle状态是动态可变的，上述6种状态可以在特定条件下互相转换，图3-1描述了状态转换的规则和所需的条件，随后笔者将详细解释这些状态的转换过程。



### 安装过程

OSGi规范定义了BundleContext接口的installBundle()方法来安装新的Bundle，方法参数为要安装的Bundle的Bundle Location。但是OSGi规范没有详细规定Bundle的安装过程应当如何进行，只是很笼统地要求OSGi框架在实现这个方法时，至少要完成生成新的Bundle ID、对元数据信息进行有效性校验、生成Bundle对象实例这些工作。

### 解析过程

解析过程是OSGi框架根据Bundle的MANIFEST.MF文件中描述的元数据信息分析处理Bundle依赖关系的过程。在实际开发中，经常会遇到“某个Bundle安装不上”这类问题，这种“安装不上”在大多数情况下不是指“安装过程”失败，而更多是指解析过程中的处理失败，抛出了异常。

要解析成功，需要确保OSGi容器能提供所有Bundle元数据信息中声明的依赖项，如Require-Capability、Import-Package和Require-Bundle中声明的依赖内容满足条件。

### 启动过程

启动过程即执行Bundle的Activator.start()方法的过程，在此方法执行期间，Bundle的状态为STARTING。如果成功执行完这个方法，那么Bundle的状态会转变为ACTIVE，而且将一直保持这个状态直到Bundle被停止。

在Bundle启动时，OSGi框架需要通过调用Class.newInstance()方法来创建Activator类的实例，因此Bundle的Activator类必须保证有一个默认的（即不带参数的）构造函数。

在启动Bundle之前，OSGi框架首先对它进行解析。如果在OSGi框架试图启动一个Bundle时这个Bundle还没有被解析，框架会自动尝试对Bundle进行解析。如果解析失败，框架会在start()方法中抛出一个BundleException异常（即使还没有真正运行start()方法）。在这种情况下，OSGi框架会记住这个Bundle“已启动过”，但它的状态仍然保持为INSTALLED，一旦条件满足，Bundle变为可以解析之后，它就应该自动启动（当然，这个Bundle还要满足后面将介绍的启动级别的要求）。如果解析本身是成功的，但是start()方法本身的代码抛出了异常导致启动失败，那么Bundle应当退回到RESOLVED状态。

### 更新过程

前面曾经拿PC类比过OSGi模块化系统：如果把PC机看成一个OSGi系统，把组成PC的CPU、内存、键盘、鼠标等部件看做模块，这些模块中有一些（如CPU等）是必须停机更换的，另外一些（如键盘、鼠标等）可以支持热插拔。与热插拔类似，OSGi框架中同样可以支持模块的动态更新。

Bundle的更新过程其实就是重新从Bundle文件加载类、生成新的Bundle对象实例的过程。OSGi规范在Bundle接口中定义了以下两种方法来更新Bundle。

1. Bundle.update()：从Bundle原来的位置进行更新。
2. Bundle.update(InputStream)：通过一个指定的输入流获取Bundle新的内容进行更新。

更新Bundle意味着在代码中使用Bundle ID、Bundle位置和名称等方式来获取Bundle对象实例时，能够获取到包含新信息的Bundle对象。但是如果已经有代码在Bundle更新前读取了旧的信息并保持持有状态，那么持有的状态信息自然是不可能自动刷新的。所以某个Bundle是否支持热插拔，OSGi只提供了技术上的支持，如果开发人员在编写Bundle代码时就没有考虑过对状态的管理更新，那么所做出来的Bundle仍然是无法动态更新的。

另外，如果在新的Bundle中修改了原Bundle所导出Package或改变了导出Package的版本号，那么对于已经存在的、甚至是更新之后才安装的Bundle来说，原来导出的旧版本Package都依然是可用状态，直到调用了PackageAdmin服务的refreshPackages()方法或OSGi框架重新启动之后，这些旧版本的Package才会停止导出。

由于更新Bundle不意味着旧Bundle立即消失废弃，换句话说，与旧Bundle有关的对象实例、旧Bundle在Java虚拟机中形成的类型和类加载器都不会立刻消失，因此，如果有大量更新Bundle的操作，开发人员还应注意Java虚拟机方法区的内存占用压力，避免造成内存泄露问题。

### 停止过程

Bundle的停止过程是启动过程的逆向转换，此时OSGi框架会自动调用Bundle的Activator类的stop()方法。在stop()方法执行期间，Bundle的状态为STOPPING。当stop()方法成功执行完毕后Bundle的状态转变为RESOLVED。

我们一般用stop()方法来释放Bundle中申请的资源、终止Bundle启动的线程等。在执行完Bundle的stop()方法后，其他Bundle就不能再使用该Bundle的上下文状态（BundleContext对象）。如果在Bundle生命周期内在OSGi框架中注册了任何服务，那么在停止该Bundle之后，框架必须自动注销它注册的所有服务。但是如果Bundle在stop()方法中还注册了新的服务，这些服务就不会自动注销。

需要注意的是，即使Bundle已经停止，它导出的Package仍然是可以使用的，无论对停止前还是停止后安装的Bundle都是如此。这意味着其他Bundle可以执行停止状态的Bundle中的代码，Bundle的设计者需要保证这样做是符合预期、没有危害的。一般来说，停止状态的Bundle只导出接口是比较合理的做法。为了尽可能保证不执行代码，导出接口的类构造器（方法）中也不应该包含可执行的代码。

### 卸载过程

调用Bundle.uninstall()方法可以实现Bundle的卸载，此时该Bundle的状态会转变为UNINSTALLED。OSGi框架应尽可能释放被卸载的Bundle所占用的资源，尽可能把框架还原成该Bundle安装前的样子。如果该Bundle导出了Package，并且这个Package被其他Bundle导入过，那么对于这些Bundle来说，原来导入的Package都是可用的，直到调用了PackageAdmin的refreshPackages()方法或框架重新启动之后才会被卸载掉。这与停用和更新Bundle时遗留的旧Package不一样，这些旧Package在卸载后会变为不可见，卸载之后才安装到OSGi框架的Bundle是不能导入它们的。

## 启动级别

开发人员可以使用代码来启动、停止某些Bundle，用户也可以在Equinox控制台中完成这项工作。但是从OSGi系统整体来看，各个模块的启动和停止顺序不应当由代码或人工完成，尤其是在Bundle数量很多时，OSGi框架提供一种全局的控制Bundle启动、停止的方案就显得更有必要了。OSGi规范定义了“启动级别”来满足这个需求，对于熟悉Linux系统的读者，对比下文的介绍就会发现，OSGi中的启动级别和Linux系统的启动级别非常相似。

启动级别是一个非负的整数，值为0时表示OSGi框架还没有运行或框架已经停止（根据具体上下文环境来区分这两种情况），只有Bundle ID为0的System Bundle的启动级别可以为0，除此以外，其他Bundle的启动级别都大于0（最大值为Integer.MAX\_VALUE）。启动级别的数值越高，所代表的启动阈值就越高，即启动顺序会越靠后。OSGi框架还有一个活动启动级别（Active Start Level），用于确定目前的状态下应该启动哪些Bundle。读者可以把它想象成一个指针，在这个指针位置及其之前（Bundle启动级别小于或等于活动启动级别）的Bundle都是已经启动或即将启动的，而在它之后（Bundle启动级别大于活动启动级别）的Bundle都是未曾启动或即将停止的。