OSGi R4 服务平台核心规范中译本

翻译：范里程
审校：曹晓钢

2008.7

www.redsaga.com 满江红开放技术研究组织

文档说明

参与人员：

作译者	联络
范里程	
曹晓钢	Caoxg <at> yahoo (dot) com

(at) 为 email @ 符号

发布记录

版本	日期	作译者	说明
R4	2007.11	范里程	翻译
R4	2008.7	曹晓钢	审校

OpenDoc 版权说明

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体无偿贡献自己的所学为最好的回馈。

本文档可从http://www.redsaga.com获取最新更新信息

OSGi 服务平台核心规范

OSGi联盟

版本4

2005年8月

Feedback

This specification can be downloaded from the OSGi web site:
http://www.osgi.org
Comments about this specification can be mailed to:
speccomments@mail.osgi.org

OSGi Member Companies

Alpine Electronics Europe Gmbh, Aplix Corporation, Belgacom, BMW Group, Cablevision Systems, Computer Associates, Deutsche Telekom AG, Echelon Corporation, Electricité de France (EDF), Ericsson Mobile Platforms AB, Esmertec, Espial Group, Inc., ETRI Electronics and Telecommunications

Research Institute, France Telecom, Gatespace Telematics AB, Gemplus, Harman/Becker Automotive Systems GmbH, Hitachi, Ltd., IBM Corporation, Industrial Technology Research Institute, Insignia Solutions, Intel Corporation, KDDI R&D Laboratories, Inc., KT Corporation, Mitsubishi Electric Corporation, Motorola, Inc., NEC Corporation, Nokia Corporation, NTT, Oracle Corporation, Panasonic Technologies, Inc., Philips Consumer Electronics, ProSyst Software GmbH, Robert Bosch Gmbh, Samsung Electronics Co., Ltd., SavaJe Technologies, Inc., Sharp Corporation, Siemens AG, Sun Microsystems, Inc., Telcordia Technologies, Inc., Telefonica I+D, TeliaSonera, Toshiba Corporation, Vodafone Group Services Limited

1. 简介

OSGi™联盟成立于 1999 年 3 月，致力于为网络和本地设备建立开放的网络管理服务规范，并制定下一代互联网向住宅、汽车、移动电话、电脑、小型办公室等提供服务的标准。

OSGi 服务平台规范是一个开放的一般性架构，主要为供应商如服务提供商、开发者、软件提供商、网关运营商和设备提供商等开发、部署和管理服务提供一种一致的协作环境。由于该规范对服务部署的的自适应性和可管理性，使得智能设备有了全新的用武之地。本规范的目标是机顶盒、服务网关、调制解调器、消费类电子设备、计算机、工业电脑、汽车、移动电话等众多产品，通过 OSGi 规范技术的实施将使服务提供商如：电信、电话公司等能提供更具有吸引力服务。

这个版本是 OSGi 服务平台规范的第四个版本，由 OSGi 成员公司的代表共同开发完成，这个版本将原来的 API 扩展到了新的领域，修正了部分现有的 API，以前的应用可以直接运行在新版本的框架中，如果需要，原来的版本管理机制也支持向前兼容。

1.1. OSGi 框架概述

OSGi 服务平台标准的核心是提供一个通用、安全并且可管理的 Java 框架，该框架支持可扩展部署和下载的应用程序(称之为 bundle)。

OSGi 设备可以下载和安装 bundle，并且可以在不需要的时候将其移除，在一个 OSGi 环境中，bundle 的安装、更新由框架统一动态管理，为此，需要框架对服务和 bundle 之间的依赖细节进行管理。

由于 Java 平台的独立性和和动态代码加载能力，bundle 的开发者可以很容易的开发出在嵌入设备上也能大规模部署的程序。

框架从功能上分为下面几个层次：

- 安全层 Security Layer
- 模块层 Module Layer
- 生命周期层 Life Cycle Layer
- 服务接口层 Service Layer
- 服务实现层 Actual Service

安全层基于 Java2 的安全机制，但是增加了一些限制，并且弥补了 Java 标准的一些不足，详细信息可以参考关于“安全层”的章节。

模块层定义了一个模块化的 Java 模型，它针对 Java 部署模式的一些缺点进行了改进，对 bundle 或者隐藏包与其他 bundle 之间共享包有严格规定，模块层独立于生命周期层和服务层，使用时可以不需要生命周期层和服务层。生命周期层提供了对模块层的 bundle 进行管理的 API，而服务层提供了 bundle 之间的通信模型。关于模块层的详细信息请参考“模块层”的章节。

生命周期层为 bundle 提供了生命周期管理的 API，为 bundle 提供了一个运行时的模型，定义了一个 bundle 如何启动、停止、安装、卸载等，另外，生命周期层也提供全面的事件机 API，允许管理 bundle 去控制和操作服务平台。有关生命周期层的更多信息可以参考相关的章节。

服务层为 Java bundle 开发者提供了一个灵活、简单并且一致的编程模型，简化服务 bundle 的开发和部署，并以非耦合的服务标准（Java 接口）来实现。这个模型允许 bundle 开发者使用他们自己的接口规范来绑定服务。这样就可以在运行时根据具体情况或需求选择接口的不同的实现。

一致的编程模型可以帮助 bundle 开发者应对扩充的问题，因为在很多情况下，框架需要运行在各种各样的硬件设备上，一致的接口可以确保软件稳定的运行。

在框架中，bundle 可以在运行时通过框架服务注册中心选择一个可用的实现，bundle 可以注册新服务、接受关于服务状态的通知或者查找适合当前设备的服务等。框架可以支持动态安装新的 bundle，支持对一个已经部署后的 Bundle 进行更改、更新而不需要重新启动系统。

关于对服务层的详细描述，请参考有关“服务层”的章节。

1.2. 更新概述

新版本的框架变化较大，框架的规范已经完全重写，标准也进行分层描述，因为改动较大，所以不可能给出一个很详细列表，以下是对更新的一个大概描述：

- 改进的模块化支持－现在，新的框架支持加载同一个包的多个版本，为了支持匹配处理，在 Import-Package 和 Export-Package 的标记定义了一系列属性和指令，请参考类加载架构。
- 可选的 bundle 级别连接—bundle 可以直接导入其他 bundle 的所有导出，而不必把他们作为 imports 进行说明，在需要的 bundle 一节中进行了讨论。
- 可选的 bundle fragment(片断)—bundle fragment 是指没有类加载器的 bundle，他们附加在一个主 bundle 之上，由主 bundle 提供类加载器。详情参考 bundle fragment(片断)一节。
- 扩展 bundle (Extension Bundle)—一个扩展的 bundle 扩展自启动类路径或者是框架类路径，在扩展 bundle 一节中进行了讨论。
- 本地化 Manifest—在 Manifest 中的实体可以通过不同的标记进行定位。在本地化一节中进行了详细描述。
- 签名 bundle—包管理被条件权限管理所替代，支持 bundle 签名，参考条件权限管理规格说明。
- 细粒度的权限管理 (fine grained admin permission)—权限管理类扩充了一系列操作来支持良好的控制。参考权限管理一节。
- bundle 权限 (bundle permission)—引入了一个新的 *BundlePermission* 类，描述对其它 bundle 的类和资源访问权限。参考 bundle 权限一节。

- 本地代码模型—为了支持一些复杂的情况，对本地代码处理作了一些细微的改变，参考加载本地代码库一节。
- 附加的事件类型—引入新的事件类型，描述解析 bundle 和无法解析 bundle 事件以及其他一些事件，参考事件一节。
- 扩展 Bundle-Classpath—Bundle-Classpath 中的内容不仅可以是 jar 文件，也可以是目录。参考 Bundle-Classpath 一节。
- Bundle 资源 URLs—本次发布已经定义了 URLs 如何描述相关资源，参考加载资源一节。

1.3. 面向的读者

本规范的主要读者是：

- 应用开发人员
- 框架和系统服务开发人员（系统开发者）
- 架构设计师

本 OSGi 规范假定读者已经至少拥有一年以上的 Java 开发经验，或者是在嵌入式领域和服务器环境下的开发经验。OSGi 环境要比传统桌面和服务器环境具有显著的动态性，应用开发人员一定要注意到这一点。

系统开发人员需要对 Java 有非常深入的了解，最好需要有三年以上的系统环境下 Java 开发经验，这是由于框架实现需要应用的是非常规领域的 Java 的应用。需要详细了解 Java 类加载机制，垃圾收集机制，Java 2 安全机制，还有 Java 本地代码加载机制。

架构师需要关注本说明的每一个方面。本说明包括了对影响设计的需求、操作的简要描述和用到的实体的一个概述，阅读说明这一节需要有 Java 的基本概念如类和接口等。但是不一定需要有编码经验。大部分要求也适用于应用开发人员和系统开发人员。

1.4. 约定和术语

1.4.1. 印刷样式

固定宽度，没有加衬线字样的文字一般描述的是 Java 包、类、接口或者成员名称，这种字体的文字一般都与代码有关。

加粗的字体表示介绍的是一个重要的术语或概念，在接下来的文字里将有对这个术语的详细解释说明。

如果一行文字需要拆分成多行，那么就会在每一行的末尾添加“<<”字符。例如：

```
http://www.acme.com/sp/ «
file?abc=12
```

表示：

http://www.acme.com/sp/file?abc=12

1.4.2. 常规语法定义

在本说明的很多地方都有语法说明，这些语法说明是基于以下的语法元素：

*	重复当前元素 0 到多次，例如(',' element) *
+	重复当前元素 1 到多次
?	当前元素是可选的
(...)	组
'...'	文字
|	Or
[...]	集合
..	列表的省略描述，如 1…5 表示 1，2，3，4，5
<...>	预定义标记

以下这些标记都是预定义的，其含义描述如下：

digit	[0..9]
alpha	[a..zA..Z]
alphanum	alpha | digit
token	(alphanum | '_' | '-')*
number	digit+
jletter	java 保留字
jletterordigit	java 保留字或者数字
qname	有效的 java 标志符
identifier	jletter jletterordigit *
quoted-string	' "' ([^"\#x0D#x0A#x00] | '\"'|'\\')* '"'
argument	token | quoted-string
parameter	directive | attribute
directive	token ':=' argument
attribute	token '=' argument
unique-name	identifier ('.' identifier)*
symbolic-name	token('.'token)*
package-name	unique-name
path	path-unquoted | ('"' path-unquoted '"')
path-unquoted	path-sep | path-sep? path-element (path-sep path-element)*
path-element	[^/"\#x0D#x0A#x00]+
path-sep	'/'

字符类中的 isJavaIdentifierStart 和 isJavaIdentifierPart 方法没有包括在最小运行环境中，这使得嵌入式系统比较难于遵循这个规范，因此，假定大于\u00FF 的为 jletterordigit 或者 jletter。

除非明确说明，空格是忽略不计的。

1.4.3. 面向对象技术

类、接口、对象、服务这些面向对象的概念都是很明确的，但是也容易混淆。比如“LogService”可以理解为 LogService 类的一个接口，或者是指类 LogService，或者是指明了 LogService 的所有功能。专家可以通过上下文环境来理解这些细微的差别，但是这样很费脑力，为了突出这些细微的差别，我们采用以下约定：

如果一个类已经确定，则类名和它在 java 代码中的名称一样，采用一种固定的拼写。例如：“HttpService class”，“类 HttpContext 中的一个方法”和“一个 javax.servlet.Servlet 对象”。如果类所在的包在上下文中并不明显，或者类并不是在常用的包如 java.lang，java.io，java.util，java.net 中，则需要通过包名和类名结合进行描述，比如：javax.servlet.Servlet。

为了增加了可读性，异常和权限类不能用“对象”来形容。比如：“抛出一个 Security Exception”和 “具有 File Permission” 比“具有 FilePermission 对象”更具有可读性。

带操作的权限描述更易于理解。比如，权限服务[com.acme.*,GET|REGISTER]表示所有服务名前缀为 com.acme 的有 GET 和 REGISTER 操作的权限服务。权限服务[Producer|Consumer, REGISTER]表示带 REGISTER 操作的 Producer 或者 Consumer 类的权限服务。

在讨论一个类的功能说明而不是实现细节时，类名采用普通文本描述，这在讨论服务的时候经常采用这样的描述。比如：“用户管理服务”更具有可读性。

有的服务名嵌入了“service”，在这样的情况中，“Service”只出现一次，而且采用大写“S”的形式，比如“the Log Service performs”。

服务对象注册到 OSGi 框架中，注册信息包括有：服务对象，属性，服务对象实现的类和接口。类和接口用于类型安全检查和命名服务。因此，我们可以说一个服务对象是注册在一个类或者接口之下的。比如，我们说“这个服务是注册在 PermissionAdmin 之下”。

1.4.4. 图表说明

本文档中所举例的图表并不是规范的。列举图表的目的在于在简单页面上提供一个高层次的视图。下面的这些举例说明的图形描述了本文档中图表的约定。

类和接口采用矩形描述，如下图所示。接口在第一行中采用《interface》标注。类和接口如果是规范的一部分，则其名称采用粗体标注，继承类名称采用简单的无格式文本。有时候类名称被简写了，同时在缩写后加上时期标注。

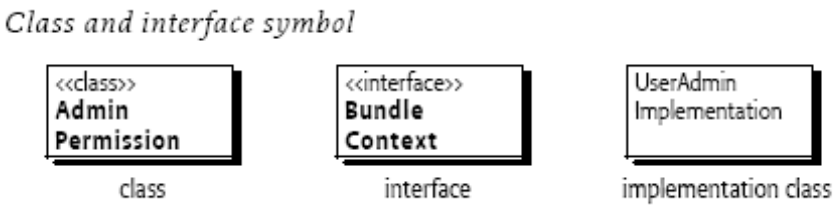


图 1.4-1 类和接口图形符号

如果一个接口或者类是服务对象，那么在图像右下部有一个黑三角标志。如图所示：

图 1.4-2 服务类图形符号

采用箭头表示类之间的继承关系。如下图描述了类 AdminPermission 继承或实现了类 Permission。

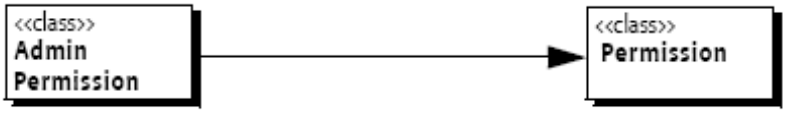


图 1.4-3 继承描述

关联关系采用直线联结，在直线的两端标明关联的基数。如下图所示。每一个 BundleContext 对象对应于 0 个或多个 BundleListener 对象。同时还有一些对关系的文字描述，文字描述按照从左到右，从上往下的顺序读。

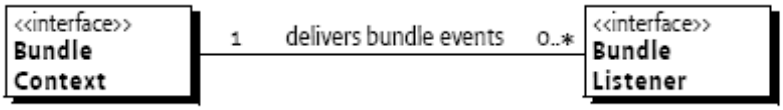


图 1.4-4 关联关系

聚集关系采用虚线描述，聚集关系表示类之间的关系，可以用关联关系替代。比如：每一个 ServiceRegistration 对象都有一个关联的 ServiceReference 对象，这种关联不是硬关联，而是可以通过派生来实现。

当关联采用对象或名称标志时，画一条点线从垂直于关联线，图 5 就是这种描述法的一个示例，表示了 UserAdmin 类和 Role 类之间的关系。

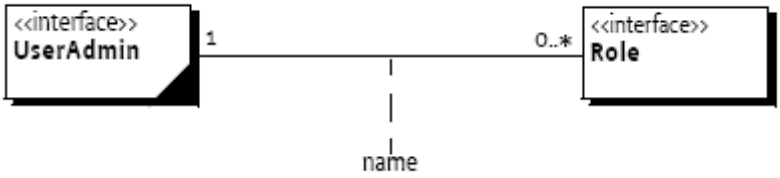


图 1.4-5 聚集关系

Bundle 是在一般应用程序中可见的实体。例如，如果一个 bundle 停止了，那么它的所有服务都将被取消注册。因此，Bundle 中的类和对象都是采用组的形式用一个灰色的矩形进行描述。如图 6 所示。

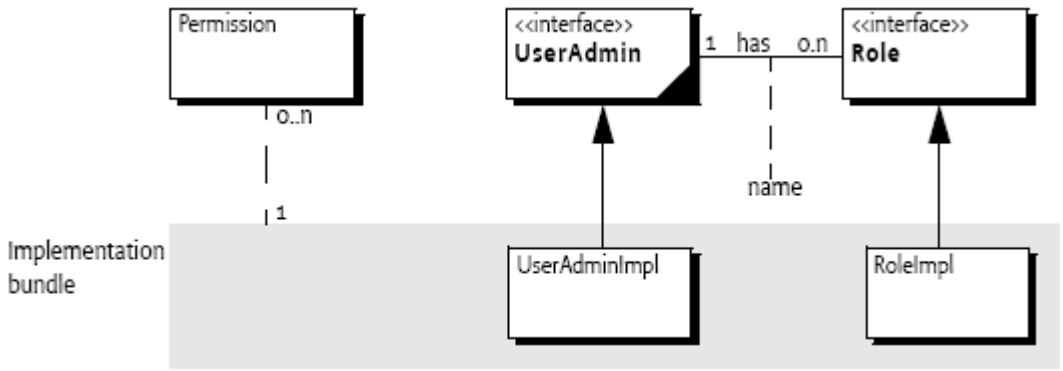


图 1.4-6 Bundle

1.4.5. 关键字

本规范沿用了关键字：可以（may 或者 can）、应该（should）和必须（must）。他们的含义在参考文献[1]中有详细定义。概括说明如下：

必须：绝对的需求，包括框架和 Bundle 都必须满足这种规范。

应该：建议的需求，建议最好采用如描述所述那样，但是也可以有理由不这么做。

可以：可选的需求，如果没有实现也可以正常工作。

1.5. 版本信息

本文档规范是 OSGi 服务平台核心规范的第 4 个版本。本规范向后支持第 1、2、3 版本。所有的安全层、模块层、生命周期层和服务层都是框架规范的一部分。

本规范中的组件都有自己的版本，独立于本文档的版本。下表描述了包和规范的信息。如果一个组件在 bundle 中引入，那么需要在输入输出包中的 Manifest 文件中进行描述这些版本信息。

Item	Package	Version
Framework Specification (all layers)	org.osgi.framework	Version 1.3
9 Conditional Permission Admin Specification	org.osgi.service.condpermission-admin	Version 1.0
7 Package Admin Service Specification	org.osgi.service.packageadmin	Version 1.2
10 Permission Admin Service Specification	org.osgi.service.permissionadmin	Version 1.2
8 Start Level Service Specification	org.osgi.service.startlevel	Version 1.0
11 URL Handlers Service Specification	org.osgi.service.url	Version 1.0

图 1.5-1 包和版本信息

1.6. 参考

[1] Bradner, S., Key words for use in RFCs to Indicate Requirement Levels
http://www.ietf.org/rfc/rfc2119.txt, March 1997.

[2] OSGi Service Gateway Specification 1.0, May 2000
http://www.osgi.org/resources/spec_download.asp

[3] OSGi Service Platform, Release 2, October 2001
http://www.osgi.org/resources/spec_download.asp

[4] OSGi Service Platform, Release 3, March 2003
http://www.osgi.org/resources/spec_download.asp

[5] Lexical Structure Java Language
http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html

2. 安全层

版本号： 1.3

2.1. 简介

OSGi 安全层是 OSGi 服务框架的一个可选的层。它基于 Java 2 安全体系结构，提供了对精密控制环境下的应用部署和管理的基础架构。

2.1.1. 要点

- 精密控制(fine grained)—在 OSGi 框架下的应用控制必须达到精细粒度的控制。
- 可管理性—安全层本身没有定义 API 来控制应用，对安全层的管理交由生命周期层。
- 可选性—安全层是可选的。

2.2. 安全概览

框架安全模型基于 Java 2 规范。如果运行安全检查，那么需要遵循 Java 2 安全架构规范。本规范假定读者熟悉这些 Java 规范，安全层是可选的。

2.2.1. 代码验证

OSGi 服务平台采用两种方式对代码进行校验：
- 位置验证
- 签名验证

权限和验证单元的代码关联，在高层定义了管理这些权限的服务，如下：
- 权限管理服务（Permission Admin service）：基于位置字符串的权限管理
- 条件权限管理服务（Conditional Permission Admin service）：基于复杂条件模型的权限管理，可以使用位置或签名来验证条件。

对于签名验证，需要 JAR 文件也是采用签名，在数字签名 JAR 文件中进行了详细描述。

2.2.2. 可选情况

框架运行的 Java 平台必须为 Java 2 权限提供 Java 安全 API。在资源约束的平台，这些 Java 安全 API 可以是一个存根，以便于可以运行和加载 bundle 类。在安全检查时，这些 API 存根并不实际执行，这些存根必须遵循一些约束：
- checkPermission—不抛出安全异常（Security Exception）返回

- checkGuard—不抛出安全异常（Security Exception）返回
- implies—返回 true

这种处理默认所有的 Bundle 具有所有的处理权限。

2.3. 数字签名 JAR 文件

本节详细描述了数字签名的 JAR 文件。本节覆盖了 JAR 文件规范的一部分，这是由于在这些规范中，存在这一些可选的方面或者是没有良好定义的方面。因此，一份参考是不足以说明的。

数字签名是一种安全机制，采用如下方法进行验证：

- 验证签名者
- 确保签名后的内容没有被修改过。

在 OSGi 框架中，对 JAR 文件的签名关联到该 JAR 文件。这种关联可用于：

- 在认证的基础上进行授权给 JAR 文件。
- 通过对 bundle 权限操作选定这些 bundle

例如：操作员给 ACME 公司赋予在他们设备上使用网络的权限，如果带有 ACME 公司签名的 bundle 部署在操作员的设备上，那么这些 bundle 就使用网络了。同理，一个特定的 bundle 也可以赋予它只有管理带 ACME 公司签名的 bundle 的生命周期。

签名提供了一种强大的代理模型。它允许操作员赋予一个公司一系列受限的权限，然后，这个公司就可以创建 JAR 文件来使用这些权限，而不需要进行其他的任何干涉或者通讯。

这种代理模型描述如下图所示：

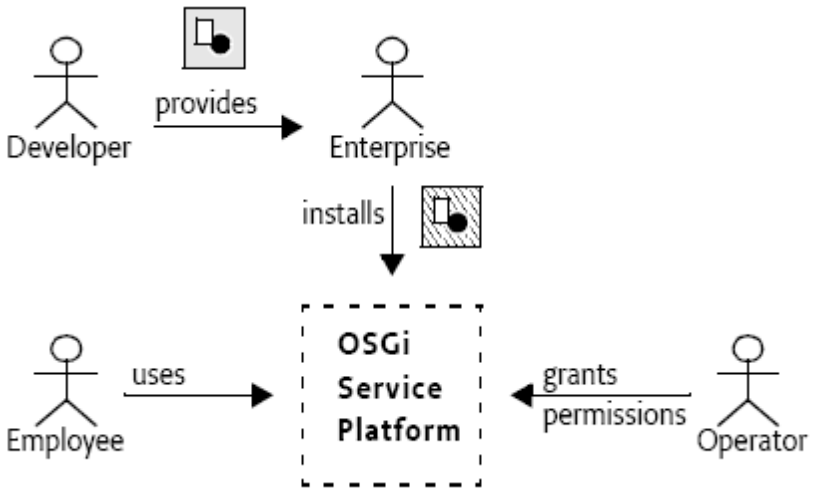


图 2.3-1 代理模型

数字签名采用了公钥密码机制。公钥密码系统采用了两个数学上相关的数字作为密钥，一个公钥，一个是私钥。公钥是公开、自由散发的，一般采用证书的形式，私钥必须要保密的。

用私钥加密的消息只有通过公钥才能够正确的验证。这可以用来验证消息的签名者的身份（假设公钥是可信的，参考证书一节）。数字签名处理过程基于 Java 的 JAR 文件签名。为了提高 OSGi bundle 的互操作性，签名过程是重复、受限并且进行了强化。

2.3.1. JAR 结构和 Manifest

一个 JAR 文件可以含有多份签名信息，每一份签名信息必须在 JAR 文件中存储两条资源信息：

- 签名指令资源文件：和 Manifest 有着类似的结构，采用.SF 的后缀名。描述了整个 Manifest 文件的摘要信息。
- 一份 PKCS#7 资源：包含对签名指令文件的数字签名，见参考文献[16]公钥密码标准。

这些 JAR 文件的签名资源必须放在 META-INF 文件夹下。在 META-INF 文件夹中的文件采用的是非常规的签名方法。在 JAR 文件流中，这些签名资源文件必须紧随着 MANIFEST.MF 文件之后，而在其他任何资源文件之前。如果不这样做，那么框架就不会接收签名信息而认为 bundle 是没有经过签名的。排列顺序是如此重要是由于 JAR 文件的接收是采用流的形式而不是缓冲形式。在其他任何资源加载之前，需要完成安全信息的加载。这种模型如下图所示：

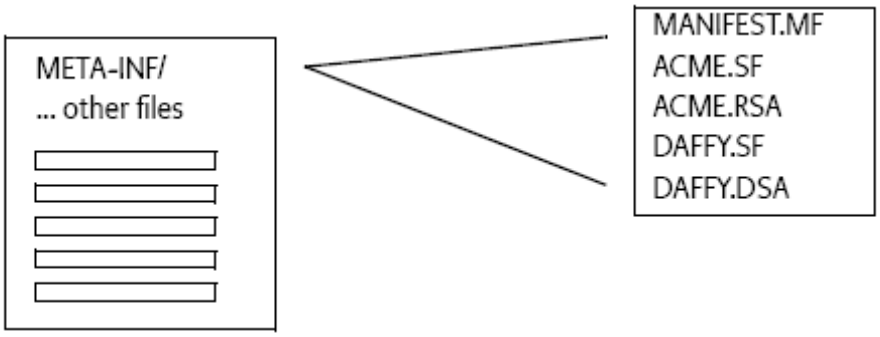


图 2.3-2 JAR 文件的签名信息

签名指令资源文件包括了 Manifest 资源的摘要，而不是实际的数据信息。摘要信息是对资源文件字节数据的一种计算结果，通过这种方式，其他人很难创建同样的匹配摘要信息的字节。

因此，JAR Manifest 必须包含有一份或多份实际资源的摘要信息。这些摘要信息在 Manifest 文件中有描述，放置在以他们名称命名的段中。摘要名称的标记是算法描述，接下来是“-Digest”。例如：“SHA1-Digest”。OSGi 框架推荐以下的摘要算法：

- MD5—Message Digest 5，在 MD4 基础上的一种改进算法，生成 128 位的哈希值，在 RFC 1321 中对 MD5 进行了详细描述。
- SHA1—对 SHA 的一种改进，生成 160 位的哈希值，在安全哈希算法一文中进行了定义。

哈希值必须采用 Base 64 位编码，关于 Base64 编码可以参考 RFC1421

下面是一个 Manifest 的例子：

```
Manifest-Version: 1.0
Bundle-Name: DisplayManifest

Name: x/A.class
SHA1-Digest: RJpDp+igoJ1kxs8CSFeDtMbMq78=
```

Name: x/B.class

SHA1-Digest: 3EuIPcx414w2QfFSXSZEBfLgKYA=

如下图所示：

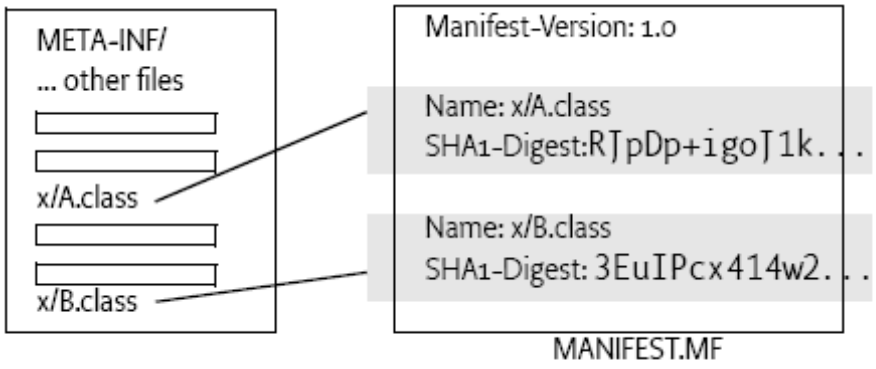


图 2.3-3 JAR 文件的签名文件

OSGi 的 JAR 文件必须对所有的资源进行签名，除了 META-INF 文件夹下的资源（默认的 Jar 签名工具）。这需要遵循 java 的 JAR 签名标准，而没有部分签名的 OSGi JAR 文件。OSGi 规范只支持完整签名的 bundle。这样做的部分是由于签名会破坏包的私有性，同时由于一个 bundle 的所有代码都使用了同样的保护域，也简化了安全 API 的处理。

在嵌套的 JAR 文件（比如在 Bundle-Classpath 中的 JAR 文件）中，签名文件被忽略处理，这些嵌套 JAR 文件和包含它们的 bundle 共享同样的保护域。必须把它们的资源文件看作是保存在外部的 JAR 文件中。

每一份签名信息都是基于两份资源，第一个文件描述签名的指令信息，第一个文件采用.SF 后缀结尾。一份签名文件的结构类似于 Manifest 文件，除了开头部分。开头如下：

Signature-Version: 1.0 instead of Manifest-Version: 1.0.

和签名信息有关联的文件只有 Manifest 资源中的摘要，标记采用算法名称，接下来就是“-Digest-Manifest”，如下例所示：

Signature-Version: 1.0
SHA1-Digest-Manifest: RJpDp+igoJ1kxs8CSFeDtMbMq78=
MD5-Digest-Manifest: IIsI6HranRNHMY27SK8M5qMunR4=

签名资源也可以包括名称片断，可是，这样的片断是被忽略处理的。

如果拥有多个签名者，并且它们采用了同样的摘要算法，则也可以同时读取出来。但是，对于每一个签名者，必须有单独的签名指令文件，而不允许在签名者之间共享签名指令文件。

签名指令文件在下图中进行了描述，描述了两个签名者：ACME 和 DAFFY

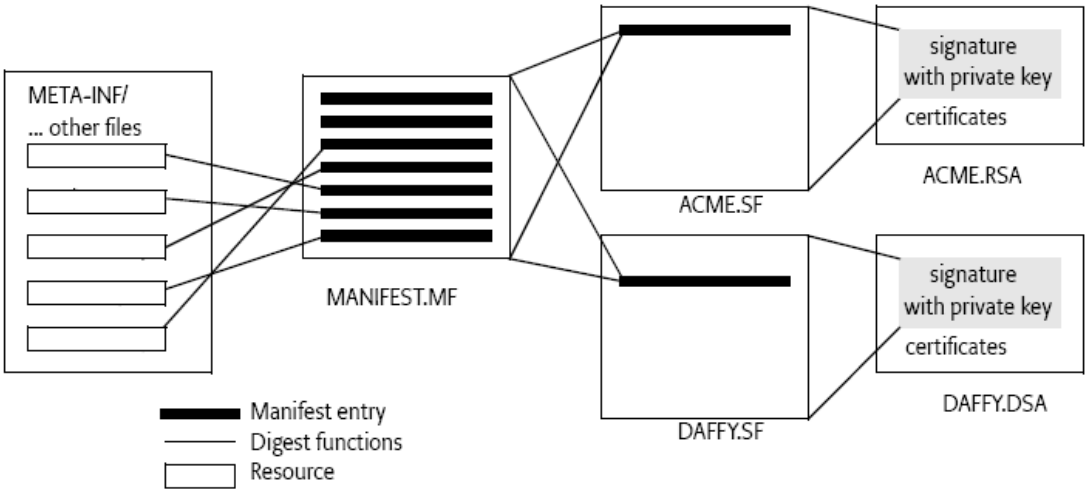


图 2.3-4 JAR 中的 Manifest、签名文件和摘要

2.3.2. JAR 文件约束

OSGi bundle 常常都是有效的 JAR 文件，但是，对于 bundle 还有一些其他的约束。

- Bundle 不支持部分签名的 bundle。除了 META-INF 文件夹，在 Manifest 文件中必须包括对所有资源的描述。
- 在签名文件中，名称部分是被忽略的，只使用了 Manifest 信息

2.3.3. 签名算法

有几种不同的算法可以支持数字签名，OSGi 框架的实现应该支持以下算法：

- DSA：The Digital Signature Algorithm。美国政府数字签名标准。用.DSA 结尾。
- RSA：Rivest, Shamir and Adleman。签名文件必须采用.RSA 结尾。

用 DSA 和 RSA 签名文件采用 PCKS#7 格式，这种格式在公钥密码标准#7 中进行了定义。PKCS#7 标准提供了利用算法加密信息和利用公钥对信息进行验证的方法。验证算法利用公钥信息进行验证：

- 数字签名和签名指令文件中的签名信息一致
- 通过私钥和证书创建签名信息

2.3.4. 证书

一份证书就是指包含名称和公钥信息的签名文件，这样的证书可以有多种格式。在 OSGi 中，Jar 签名是基于 X.509 证书格式。该格式已经形成多年，目前是 OSI 组织标准的一部分。在参考文献[7] X.509 Certificates 中有详细描述。

一份 X.509 证书由以下部分组成：

- 主题名称：主题名称惟一标志了证书持有人信息。例如，一个人可能拥有姓名、国籍、

邮箱地址信息、组织、部门等信息。这个标志符是一个 DN（Distinguished Name），下文将对 DN 进行解释。

- 认证机构：发出证书的认证机构名称，也是一个 DN。
- 证书扩展：证书也可以包括有图片、指纹编码信息、护照号码等其他信息。
- 公钥信息：公钥可用于加密过程，而私钥用于解密过程。公钥可以自由散发，而私钥必须保密。公钥信息指定了加密的算法和公钥的主题。
- 有效日期：证书只是在特定时期有效。
- 认证机构数字签名：如果接收者信任该认证机构，意味着信任签发证书的实体。

证书的结构如下图所示：

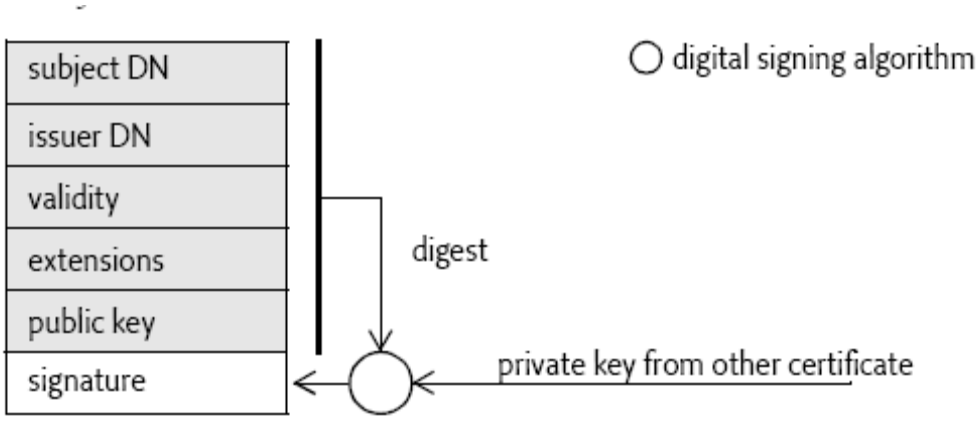


图 2.3-5 证书结构

证书可以自由散发，并不包含任何保密信息。因此，PKCS#7 资源库包含了签名证书。由于证书信息是包含在 bundle 内部的，所以在表面上证书不能进行校验。入侵者可以轻易的创建自己的证书。而接收者只需要验证证书的发行者是否可信的。在证书处于可信状态之前，必须要对证书进行校验。因此就有必要建立可信模型。

OSGi 支持这样一种可信模型，但不是必须实现。这种可信模型将所有的证书放置在一个库中，在库中的证书默认为可信的。但是，将所有的证书放置在一个库中过于庞大，在一个开发模型中，一种设备将有成千上万个证书，对证书的管理将是一件非常困难的事情。

解决的办法是采用一个证书验证其他证书的形式，通过多次迭代处理，形成了一个证书链。所有证书链上的证书都在 PKCS#7 文件中。如果一个证书可以在可信证书库中找到，那么与这个证书相关的证书都是可信的，模型的尺度缩小到了合理状态，而只有很少的一部分证书需要进行管理，这也是在浏览器中所用到的证书模型。在下图进行了描述。

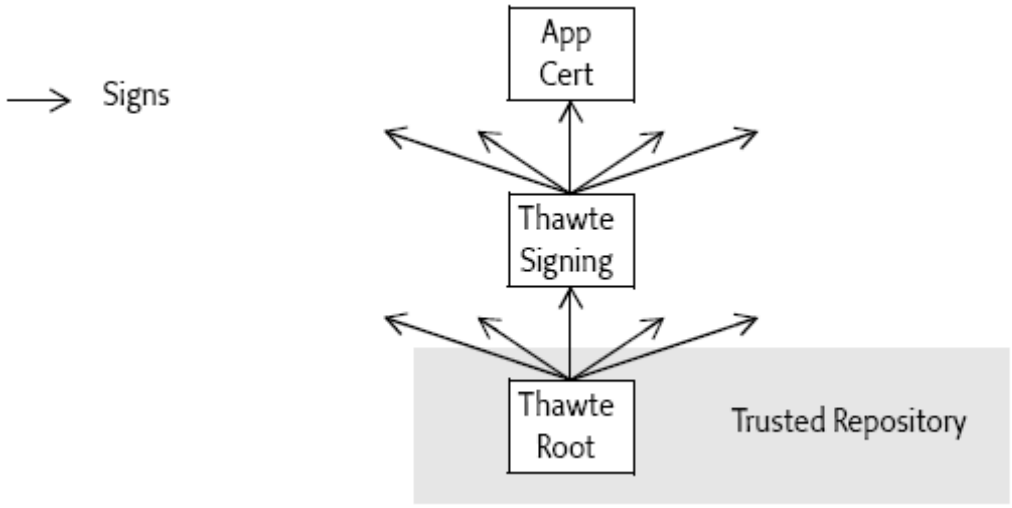


图 2.3-6 证书扇形展开

本规范没有描述如何对可信证书库进行存取，只是描述了证书库的组装和组织结构。

2.3.5. 惟一标识名（DN，Distinguished Name）

X.509 命名采用 DN 命名法。一个 DN 名称是一个高度结构化的名称，在一个层次化的命名空间对节点进行验证。DN 概念来源于 X.500 的目录服务，这种目录服务预想世界范围的命名空间由 PTTs 进行管理的。现在，DN 被用于可以由操作员设计的局部命名空间的标志。例如：验证 Bugs 的 DN 名称如下所示：

cn=Bug,o=ACME,c=US

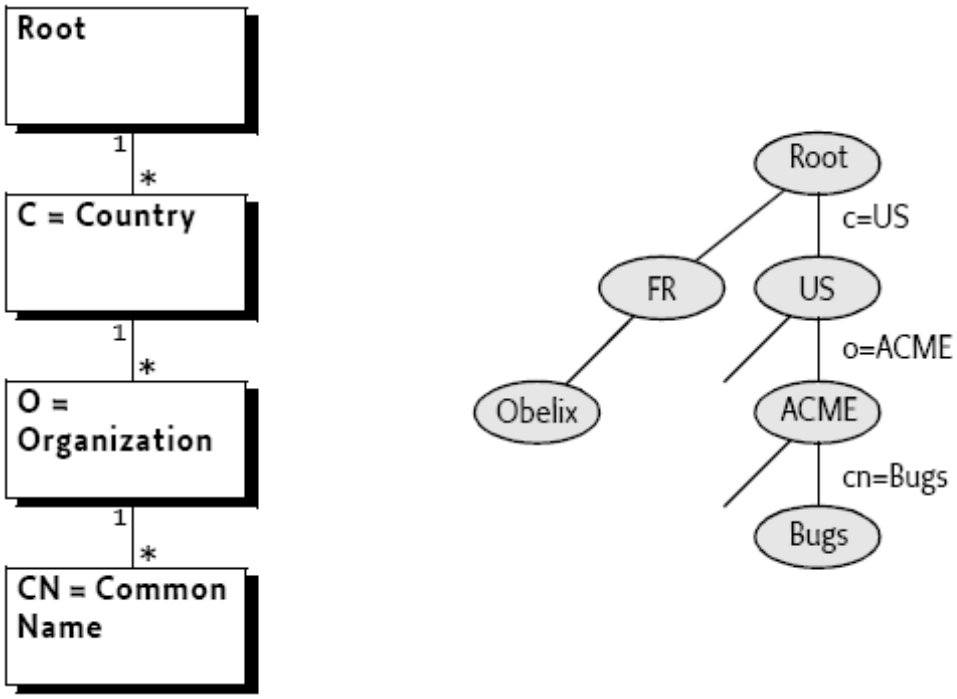


图 2.3-7 命名空间中的国籍，公司和个人信息

对名称空间的遍历是逆序进行的，开始部分是最不重要的信息，但是却是最具体的信息。所以，属性的排序是很重要的，两个具有同样属性但是不同排序的 DNs 是不等同的。

在示例中，对节点的搜索从根部开始，根部节点有一个属性 c（countryName 国家名称）的值为 US，然后搜索 c 节点的字节点有一个属性 o（organizationName 组织名称）的值为 ACME，然后对 o 节点搜索子节点，有一个属性为 cn（commonName 普通名称）的子节点的值为“Bugs Bunny”。

树模型在 X.500 标准中对 DN 进行了正式定义。但是，现在实际上很多 DNs 有很多和数结构没有关系的属性，例如：很多 DNs 在 ou（organizationalUnit 组织单元）节点还有批注和版权信息。

X.509 证书中的 DN 名称用 ASN.1（ISO 定义的一种类型语言）这种二进制结构进行表达。但是，DN 经常用于进行交互，有时，系统用户需要去确认对一个证书的使用，或者操作员需要根据客户证书对其赋予权限。因此，DN 名称需要具有很好的可读性，这点尤为重要。而采用 ASN.1 的表达方法可读性很差，本规范只用了在 RFC 2253 中用在类 javax.security.auth.x500.X500Principal 中规定的范式来定义的 DN 字符串。

然而，由于使用了一些不常用的类型和特征（二进制数据，多值的 RDNs，多种语言字母，具有特殊匹配规则的属性），使得编码和解码的复杂度提高。框架的实现必须要支持这些特性，但是用户应该避免这样做。因为这些特性现在已经很少使用了。

DN 字符串格式如下所示：

```
dn         ::= rdn ( ',' rdn ) *
rdn        ::= attribute ( '+' attribute ) *
attribute  ::= name '=' value
name       ::= readable | oid
oid        ::= number ( '.' number ) *   // See 1.4.2
readable   ::= <see attribute table>
value      ::= <escaped string>
```

分隔符前面和后面的空格是忽略不计的，值中间的空格是有用的，值中间的多个空格看作是一个空格。值中不允许出现通配符（'*' \u002A）。下面的值必须用反斜杠进行转义。

```
comma         ','   \u002C
plus          '+'   \u002B
double quote  '"'   \u0022
back slash    '\'   \u005C
less then     '<'   \u003C
greater then  '>'   \u003E
semicolon     ';'   \u003B
```

由于在 java 中反斜杠也必须要转义，所以应该在 Java 代码中添加两个反斜杠，如下所示：

DN: cn = Bugs Bunny, o = ACME++, C=US

Canonical form: cn=bugs bunny, o=acme\+\+,c=us

Java String: "cn=bugs bunny, o=acme\\+\\+,c=us"

在 DNs 中可以使用 unicode 编码字符集。字符串必须规范化并用范式进行表述之后才可以进行比较。

```
DN:              cn = Bugs Bunny, o = Ð Þ, C=US
Canonical form:  cn=bugs bunny,o=ð þ,c=us
Java String:     "cn = Bugs Bunny, o = Ð Þ, C=US"
```

属性名称（也称之为属性类型）实际转义成对象标志（OID，Object IDentifier），OID 采用点分表示法，如 2.5.4.3 表示属性名称为 cn。由于在比较规则中使用的是 OID 别名，因此，在实现框架中不能使用属性名称，而且只有在下表中列出的属性才可以使用。

commonName	cn	2.5.4.3 ITU X.520
surName	sn	2.5.4.4
countryName	c	2.5.4.6
localityName	l	2.5.4.7
stateOrProvinceName	st	2.5.4.8
organizationName	o	2.5.4.10
organizationalUnitName	ou	2.5.4.11
title		2.5.4.12
givenName		2.5.4.42
initials		2.5.4.43
generationQualifier		2.5.4.44
dnQualifier		2.5.4.46
streetAddress	street	RFC 2256
domainComponent	dc	RFC 1274
userid	uid	RFC 1274/2798?
emailAddress		RFC 2985
serialNumber		RFC 2985

如下 DN：

2.5.4.3=Bugs Bunny,organizationName=ACME,2.5.4.6=US

等同于：

cn=Bugs Bunny,o=ACME,c=US

属性类型在匹配规则中进行了正式定义，潜在的允许灵活和不灵活两种情况（cases sensitive and case insensitive）。在上面列表中的都是不灵活匹配。OSGi DN不依赖于灵活的情况。在X.500标准中支持多值的RDNs，然而，并不推荐使用这种方法。可以参考文献[18]（理解和部署LDAP目录服务）中的描述。多值的RDNs采用加号连接多个值，值之间没有顺序。如下所示：

cn=Bugs Bunny+dc=x.com+title=Manager, o=ACME, c=US

等同于：

dc=x.com+cn=Bug Bunny+title=Manager, o=ACME, c=US"

2.3.6. 证书匹配

证书根据主题 DN 进行匹配。在匹配之前，DNs 需要转换成范式，转换的算法在类 javax.security.auth.x500.X500Principal 中进行了定义。

DNs 也可以使用通配符（'*' \u002A）进行匹配。一个通配符可以代替所有可能的值。由于 DN 的结构不同，这种匹配比基于字符串格式的通配符匹配要复杂很多。

一个通配符可以代替 RDNs 中的一个号码，或者是一个单独的 RDN 值。有通配符的 DNs 必须在比较之前进行规范化。这也就是说，除了在值中的空格，其他空格都要去掉。带通配符的 DN 格式如下：

```
CertificateMatch::= dn-match ( ';' dn-match ) *
dn-match        ::= ( '*' | rdn-match )
                        ( ',' rdn-match ) * | '-'
rdn-match       ::= name '=' value-match
value-match     ::= '*' | value-star
value-star      ::= < value, requires escaped '*' and '-' >
```

最简单的例子是只有一个通配符的情况，可以匹配任何 DN。一个通配符也可以代替 DN 中第一个 RDNs 序列。第一个 RDNs 描述了最没有价值的信息，这样匹配的 RDNs 结果为空。

例如：一个带通配符的 DN 可以和图 2.3-5 的 DN 的 ACME 节点的所有子节点匹配。DN 如下：

*, o=ACME, c=US

这个带通配符的DN与下面这些DNs匹配：

```
cn = Bugs Bunny, o = ACME, c = US
ou = Carots, cn=Daffy Duck, o=ACME, c=US
street = 9C\, Avenue St. Drézéry, o=ACME, c=US
dc=www, dc=acme, dc=com, o=ACME, c=US
o=ACME, c=US
```

与下面这些DNs不匹配：

```
street = 9C\, Avenue St. Drézéry, o=ACME, c=FR
dc=www, dc=acme, dc=com, c=US
```

如果在RDN中使用通配符，那么属性值必须匹配通配符代表的值，而不能匹配其他的属性值，如下示例：

```
cn=*,o=ACME,c=*
```

可以匹配的 DNs：

```
cn=Bugs Bunny,o=ACME,c=US
cn = Daffy Duck , o = ACME , c = US
cn=Road Runner, o=ACME, c=NL
```

不能匹配：

```
o=ACME, c=NL
dc=acme.com, cn=Bugs Bunny, o=ACME, c=US
```

两种格式的通配符可以同时使用，例如，为了匹配来自 ACME 公司的所有 DN，可以使用：

```
*, o=ACME, c=*
```

匹配需要根据上下文环境进行，作为证书链的一部分，每一个证书都有一个证书的主题 DN 和发行机构 DN，发行机构的 DN 作为主题 DN 验证证书链的第一个证书，因此，DN 匹配可扩展到对签名者匹配，分号（“;”，\u003B）用于分隔证书链的 DNs。
下面这个例子可用于匹配由美国的 Tweety 公司发行的证书：

```
* ; ou=S & V, o=Tweety Inc., c=US
```

单独的一个通配符匹配一个空值或者一个证书。
但是有时候需要匹配更长的证书链，减号（“-”，\u002D）代表零个或者多个证书，而星号只代表一个证书，例如，为了匹配在一个证书链中的所有 Tweety 公司发行的证书，表达式如下：

```
- ; *, o=Tweety Inc., c=US
```

这个例子可以匹配所有的 Tweety 公司的可信证书，或者是证书链上被其他可信证书认证的 Tweety 公司的证书。某些证书是框架定义可信的。

2.4. 参考

[6] RFC 2253
http://www.ietf.org/rfc/rfc2253.txt
[7] X.509 Certificates
http://www.ietf.org/rfc/rfc2459.txt
[8] Java 2 Security Architecture
Version 1.2, Sun Microsystems, March 2002
[9] The Java 2 Package Versioning Specification
http://java.sun.com/j2se/1.4/docs/guide/versioning/index.html
[10] Manifest Format
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR%20Manifest
[11] Secure Hash Algorithm 1
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
[12] RFC 1321 The MD5 Message-Digest Algorithm
http://www.ietf.org/rfc/rfc1321.txt
[13] RFC 1421 Privacy Enhancement for Internet Electronic Mail
http://www.ietf.org/rfc/rfc1421.txt
[14] DSA
http://www.itl.nist.gov/fipspubs/fip186.htm
[15] RSA
http://www.ietf.org/rfc/rfc2313.txt which is superseded by
http://www.ietf.org/rfc/rfc2437.txt
[16] Public Key Cryptography Standard #7
http://www.rsasecurity.com/rsalabs/node.asp?id=2129
[17] Unicode Normalization UAX # 15

http://www.unicode.org/reports/tr15/
[18] Understanding and Deploying LDAP Directory Services
ISBN 1-57870-070-1

3. 模块层

版本号： 1.3

3.1. 介绍

Java 平台只提供了对打包、部署和对 Java 应用和组件检验的最小支持。因此，很多基于 java 的项目，如 JBoss、NetBeans，常常借助于专用的类加载器来创建用户模块层，以实现打包、部署和对 Java 应用和组件检验。OSGi 框架提供了对 java 模型化的一般和标准的解决方案。

3.2. Bundle

框架定义了模型化单元，称之为一个 bundle。一个 bundle 由 java 的类和其他资源组成，可以为终端用户提供功能。通过良好定义的方式，Bundle 可以和导入(importer)及导出(exporter) Bundle 之间共享 Java 包。

在 OSGi 服务框架中，bundle 是仅有的需要部署的 Java 应用实体。

Bundle 以 JAR 文件的方式进行部署。JAR 文件使用 ZIP 的格式存储应用程序以及所需的资源。在文献[27]中对 Zip 文件格式进行了描述。

一个 bundle 是一个如下的 JAR 文件：

- 拥有提供服务所必须的资源。这些资源可以是 java 的 class 文件，或者是其他的数据如 HTML 文件，帮助文件，图标文件等。一个 bundle JAR 文件也可以嵌入其他 JAR 文件作为资源，但是不支持多层嵌套的 JAR。
- 有一个 manifest 文件描述 JAR 文件内容和 bundle 的信息。该文件处于 JAR 的头部，提供框架需要的安装和激活 bundle 所需的信息。例如，它对其他资源如 JAR 文件的依赖这种状态信息必须在 bundle 运行之前加载。
- 可以在 OSGI-OPT 文件夹提供可选的文档信息，该文件夹可以位于 JAR 文件根目录或者它的子文件夹中。OSGI-OPT 文件夹中的内容都是可选的。例如，可以在其中保存 bundle 的源代码。管理系统可以删除该文件夹内容，以便于节约 OSGi 服务平台的存储空间。

当一个 bundle 开始运行，通过 OSGi 服务平台，它开始对安装在平台内的其他 bundle 提供功能和服务。

3.2.1. Bundle Manifest Headers

Bundle 的描述信息在一个 manifest 文件中，在 JAR 文件中的 META-INF 目录下的 MANIFEST.MF 文件。框架在 manifest 文件头中定义了 Export-Package 和 Bundle-Classpath 这样的 OSGi manifest 头，bundle 的开发人员可以使用它们提供 bundle 的描述信息。Manifest 头部必须严格遵照文献[28]中的“manifest format”规则进行定义。

框架的实现必须：

- 处理 manifest 的主要部分。manifest 中的单独部分只用于对 bundle 的签名验证。
- 忽略不可识别的 manifest 标记信息。因此，bundle 开发人员可以在 manifest 文件中定义

附加的其他信息。

- 忽略不可识别的指令和属性。

下面列出了对 manifest 标记的所有规定，除了特别指明，所有的标记信息都是可选的。

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

描述指出启动和停止 bundle 的类名称。

3.2.1.2 Bundle-Category: osgi, test, nursery

描述用逗号分隔的分类名称。

3.2.1.3 Bundle-Classpath: /jar/http.jar,.

定义用逗号分隔的路径，包含的内容有 JAR 文件和包含类和资源的目录（bundle 内部）。点号（‘.’）代表 JAR 文件的根目录，同时也是默认的。参考 bundle 类路径一节。

3.2.1.4 Bundle-ContactAddress: 2400 Oswego Road, Austin, TX 74563

标识 bundle 发行者的联系信息。

3.2.1.5 Bundle-Copyright: OSGi (c) 2002

描述 bundle 的版权信息。

3.2.1.6 Bundle-Description: Network Firewall

对 bundle 的简短描述信息。

3.2.1.7 Bundle-DocURL: http:/www.acme.com/Firewall/doc

Bundle 文档的链接地址。

3.2.1.8 Bundle-Localization: OSGI-INF/l10n/bundle

描述 bundle 的本地文件地址，默认值是 OSGI-INF/l10n/bundle。

3.2.1.9 Bundle-ManifestVersion: 2

定义了 bundle 遵循本规范的那种规则。默认值为 1，表示第三个版本的 bundle，2 表示第四个版本及其后发布的版本。也可以为 OSGi 新发布的版本定义更高的数字。

3.2.1.10 Bundle-Name: Firewall

定义了一个具有可读性的名字来标识 bundle。应该是一个简短易读没有空格的名字。

3.2.1.11 Bundle-NativeCode: /lib/http.DLL; osname = QNX; osversion = 3.1

对 bundle 中包含的本地代码库的规范。可以参考加载本地代码库一节。

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

描述在服务平台上必须出现的了可执行环境的，用逗号分割。参考可执行环境一节。

3.2.1.13 Bundle-SymbolicName: com.acme.daffy

提供了 bundle 的一个全局的惟一的标志符。名称应该是基于反域名解析的。参考 bundle 标志符一节。这部分是必须的。

3.2.1.14 Bundle-UpdateLocation: http://www.acme.com/Firewall/bundle.jar
描述 bundle 的更新地址。如果 bundle 需要更新，则使用这个地址进行更新。

3.2.1.15 Bundle-Vendor: OSGi Alliance
描述 bundle 的发行者信息。

3.2.1.16 Bundle-Version: 1.1
描述 bundle 的版本信息。默认值为 0.0.0

3.2.1.17 DynamicImport-Package: com.acme.plugin.*
包含了一个逗号分隔的动态导入包清单。参考动态导入包。

3.2.1.18 Export-Package: org.osgi.util.tracker;version=1.3
描述导出包声明。参考导出包。

3.2.1.19 Export-Service: org.osgi.service.log.LogService
不建议使用

3.2.1.20 Fragment-Host: org.eclipse.swt; bundle-version="[3.0.0,4.0.0)"
描述本片断中的主 bundle，参考片断主体（Fragment-Host）一节。

3.2.1.21 Import-Package: org.osgi.util.tracker,org.osgi.service.io;version=1.4
声明 bundle 导入的包。参考导入包一节。

3.2.1.22 Import-Service: org.osgi.service.log.LogService
不建议使用

3.2.1.23 Require-Bundle: com.acme.chess
指定 bundle 中需要其他 bundle 导出的内容。参考 bundle 需求一节。

3.2.2. 标记语构

每一个manifest标记都有自己的语法结构，语法结构在文献[29]W3C EBNF中进行了描述。接下来的部分定义了一些常用的规则。

3.2.3. 通用标记语构

很多manifest标记都有一个通用的结构，一般如下所示：

```
header ::= clause ( ',' clause ) *
clause ::= path ( ';' path ) *
           ( ';' parameter ) *        // See 1.4.2
```

参数可以是指令也可以是属性。一般指令在框架中用于潜在语义支持，属性用于匹配和比较。

3.2.4. 版本

版本规格在很多地方都会用到，版本说明采用以下的结构：

```
version    ::=
   major( '.' minor ( '.' micro ( '.' qualifier )? )? )?
major      ::= number                    // See 1.4.2
minor      ::= number
micro      ::= number
qualifier  ::= ( alphanum | '_' | '-' )+
```

版本标记不能有任何空格，默认值为0.0.0

3.2.5. 版本范围

用数字间隔表示法描述版本的范围，参考数字间隔表示法惯例。
版本范围表示的结构如下：

```
version-range ::= interval | atleast
interval ::= ( '[' | '(' ) floor ',' ceiling ( ']' | ')' )
atleast ::= version
floor ::= version
ceiling ::= version
```

如果一个版本范围指定是一个单独的版本，必须写成 [version,∞) 式样。默认的没有指定版本范围的值为 0，等价于：[0.0.0,∞)
注意在版本范围定义中使用逗号分隔，双引号结束，如下所示：

```
Import-Package: com.acme.foo;version="[1.23, 2)",
   com.acme.bar;version="[4.0, 5.0)"
```

下表中，对于左边列规定的范围，如果 x 满足右边列的谓词判定，则可以认为 x 在范围之内。
示例：

Predicate
1.2.3 <= x < 4.5.6
1.2.3 <= x <= 4.5.6
1.2.3 < x < 4.5.6
1.2.3 < x <= 4.5.6
1.2.3 <= x

3.2.6. 过滤器正则表达式

OSGi 规范广泛使用了过滤器表达式。过滤器表达式为约束提供了简洁的表达方法。
过滤器正则表达式是基于 LDAP 的查找过滤器字符串表达法，在文献[23]中有详细描述。需要注意的是 RFC 2254，一种 LDAP 查找过滤器的字符串表示法，替代了 RFC 1960，只是添加了扩展的匹配，但是 OSGi 框架的 API 不提供对它的支持。
一个 LDAP 查找过滤器使用前缀形式字符串表示，采用如下的语法定义：

```
filter       ::=  '(' filter-comp ')'
filter-comp  ::=  and | or | not | operation
and          ::=  '&' filter-list
or           ::=  '|' filter-list
not          ::=  '!' filter
filter-list  ::=  filter | filter filter-list
operation    ::=  simple | present | substring
simple       ::=  attr filter-type value
filter-type  ::=  equal | approx | greater | less
equal        ::=  '='
approx       ::=  '~='
greater      ::=  '>='
less         ::=  '<='
present      ::=  attr '=*'
substring    ::=  attr '=' initial any final
inital       ::=  () | value
any          ::=  '*' star-value
star-value   ::=  () | value '*' star-value
final        ::=  () | value
value         ::=   <see text>
```

在属性中，左边字符串表示一个属性或者是键。属性名称是大小写不敏感的，也就是说 cn 和 CN 代表的是同样的属性。在属性名称中不能含有'='、'>'、'<'、'~'、'(' 或者 ')'字符，在属性名称中可以嵌入空格，但是开始和结尾的空格都被忽略的。右边表示值，也可以是通过和过滤属性的值进行比较而得到的部分值。
如果值中间含有字符'*'、'(' 或者 ')'中的一个，那么必须要采用反斜杠进行转义处理。在属性值中，空格是由含义的，空格由 Character.isWhiteSpace()定义。
虽然在子串和现在产品中都可以生成“attr=*”结构，但是这种结构只是用于表示一个现有过滤器。
对过滤器求值有具体的框架实现来完成，但是最少需要忽略大小写和空格。需要使用到探测法和其他一些智能逼近的代码。
过滤器求值之后，对具体的属性值和过滤器的值进行比较。对这些值的比较并不是直接进行的。字符的比较和数字比较不一样，也同样适用于多值属性值的比较。属性键值必须是字符串类型的，这样大小写不敏感的属性就可以获得属性值。

```
type       ::=  scalar  | primitive | vector | array
scalar     ::=     String | Integer | Long | Float
                   | Double | Byte | Short
                   | Character | Boolean
primitive  ::= int | long | float | double | byte | short |
char | boolean
array      ::= <Array of primitive>
              | <Array of scalar>
vector     ::= Vector of scalar
```

下面的规则用于进行比较：

- 字符串—利用字符串进行比较
- 整数、长整形数、浮点数、双精度数、字节、短整形数、字符对象和图元—利用数字进行比较。
- 布尔型对象—利用等式比较。
- 数组或者是向量—根据保存内容进行比较。有可能其中的内容具有多种类型，或者为空。如果其中保存的内容不是上述类型，而且这种类型有一个带 String 类型参数的构造方法，那么框架将通过把属性值传递给构造函数中的这个 String 参数，构造出一个这样的对象，然后再根据上述规则进行比较。
- 可比较的对象—通过比较接口进行比较。
- 其他对象—相等关系。

如果没有上述规则可以对应，那么比较的值为 false。

如果一个属性具有多个值，那么过滤器只要满足其中一个就是匹配的。

例如：

```
Dictionary dict = new Hashtable();
dict.put( "cn", new String[] { "a", "b", "c" } );
```

那么 dict 可以满足这样的过滤器：（cn=a）和（cn=b）

3.3. 运行环境

在多个运行环境下执行的 bundle 必须在其 manifest 文件中标注这种对环境的约束。标记名称是 Bundle-RequiredExecutionEnvironment，这个节点的结构是用逗号分割的多个运行环境的清单。

```
Bundle-RequiredExecutionEnvironment ::=
   ee-name ( ',' ee-name )*

ee-name ::= <defined execution environment name>
```

例如：

```
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0,
 OSGi/Minimum-1.1
```

如果一个 bundle 有这样一个标记信息，那么 bundle 只能使用在提及的运行环境中签名了的方法。在 bundle 中，必须列出所有的可以运行的环境。

3.3.1. 运行环境命名

运行环境需要一个合适的名字，理由如下：

- bundle 在安装之前需要知道框架提供的运行环境。
- 提供框架使用的运行环境信息。

运行环境的命名可以使用除了空格之外的其他任何字符，也可以使用逗号（',', or \u002C），OSGi 联盟定义了一系列的运行环境。

以后，命名模式将采用 J2ME 格局形式，对于命名模式没有一个非常清晰的定义，但是在不同的规范中，名称都是类似的。

J2ME 模式通过配置和设备简表（profile）来设计运行环境。OSGi 将这两部分合并为一个名称，用来描述运行环境。

在 J2ME 已经存在一些定义好的运行环境，可以用于定义服务平台的服务器。运行环境的标题必须和现有的规范兼容。

一个 J2ME 运行环境名称包括配置和设备简表。在 J2ME 中，下面是两个不同的运行环境：

microedition.configuration

microedition.profiles

例如，Foundation 有这样一个命名的运行环境：CDC-1.0/Foundation-1.0，命名结构采用以下的规则：

```
ee-name = [ <configuration> '-' <version> '/' ]
          <profile> '-' <version>
```

配置和设备简表由JCP或者是OSGi联盟定义，一个运行环境不能没有配置或者设备简表，设备简表用于区别不同运行环境。这些原则并不是标准化的。

下表展示了部分常见的运行环境描述：

Table 3 Sample EE names

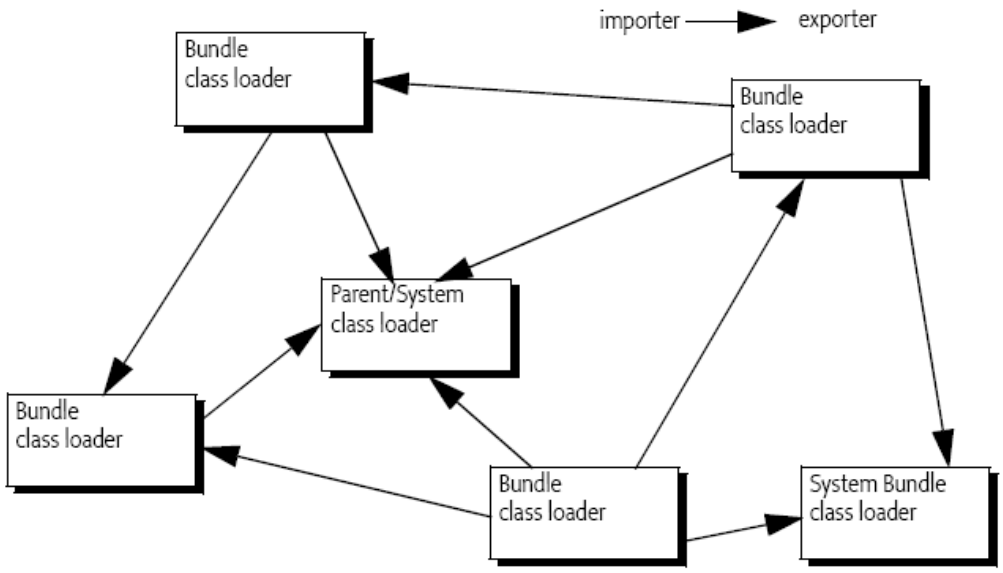
Name	Description
CDC-1.0/Foundation-1.0	Equal to J2ME Foundation Profile
OSGi/Minimum-1.1	OSGi EE that is a minimal set that allows th implementation of an OSGi Framework.
JRE-1.1	Java 1.1.x
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
PersonalJava-1.1	Personal Java 1.1
PersonalJava-1.2	Personal Java 1.2
CDC-1.0/PersonalBasis-1.0	J2ME Personal Basis Profile
CDC-1.0/PersonalJava-1.0	J2ME Personal Java Profile

org.osgi.framework.executionenvironment属性来自于BundleContext.getProperty(String)，必须要包含在框架的运行环境中。这个属性是易变的。由于bundle有可能随时会改变这个属性值，因此框架实现者不能缓存这个信息。这样做的目的是为了进行测试和对运行时的执行环境的可扩展性。

3.4. 类加载机制

许多bundle可以共享虚拟机（VM）。在VM内部，bundle可以相互隐藏包和类，也可以和其他bundle共享包。

隔离和共享包关键是由java的类加载器来实现，类加载器通过仔细定义的规则从bundle空间的一个子集中加载类。每一个bundle只会有一个单独的类加载器，类加载器形成了一个类加载的代理网络结构，如下所示：



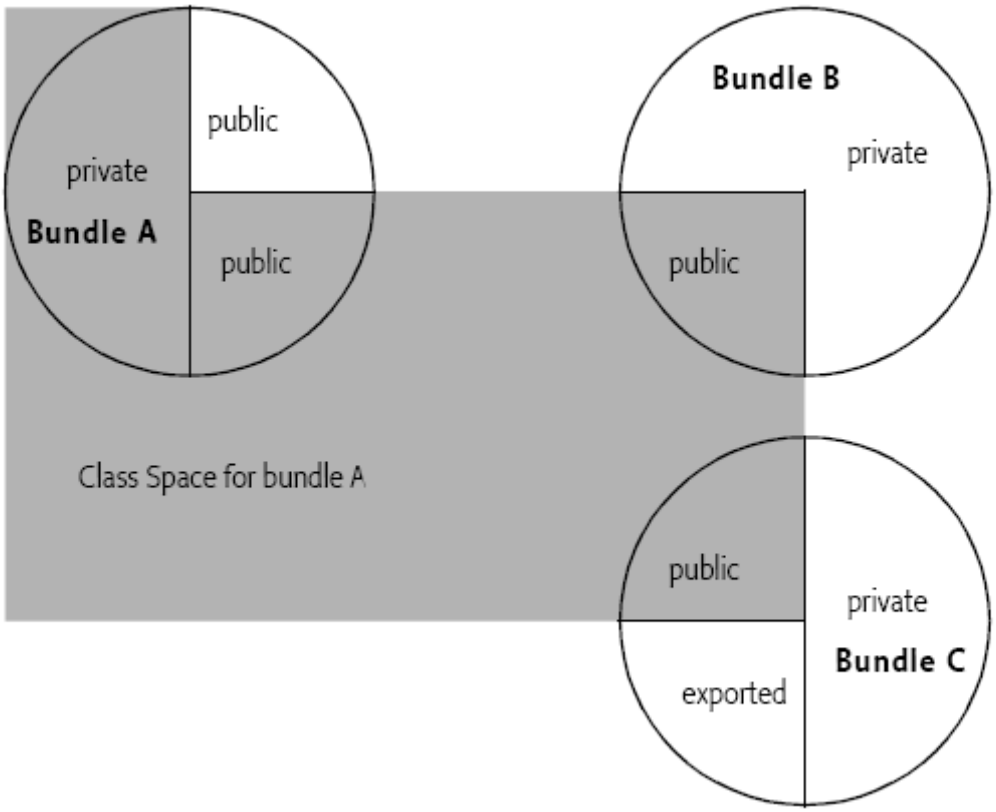
类加载器可以加载类和资源。加载途径有：

- 启动类路径：启动类路径中有一个java.*的包以及它实现的包。
- 框架类路径：在框架中通常有一个单独的类加载器，加载框架实现的类和关键的服务接口类。
- Bundle类空间：bundle的类空间由和bundle相关的JAR文件组成，以及其他和bundle紧密相关的JAR文件，比如bundle片断（参考bundle片断一节）

类空间是指一个给定的bundle类加载器可以访问到的所有的类。因此，一个指定bundle的类空间来自：

- 父类加载器（通常是来自启动类路径的java.*包中的）
- 导入的包
- 必须的bundle
- Bundle类路径（私有包）
- 附加的片断

类空间必须是一致的，也就是说不能存在相同全名的两个类（为了防止类声明错误）。但是，在OSGi框架中，不同的类空间可以存在同名的类。在模块层，支持不同版本的类加载到相同的虚拟机中。

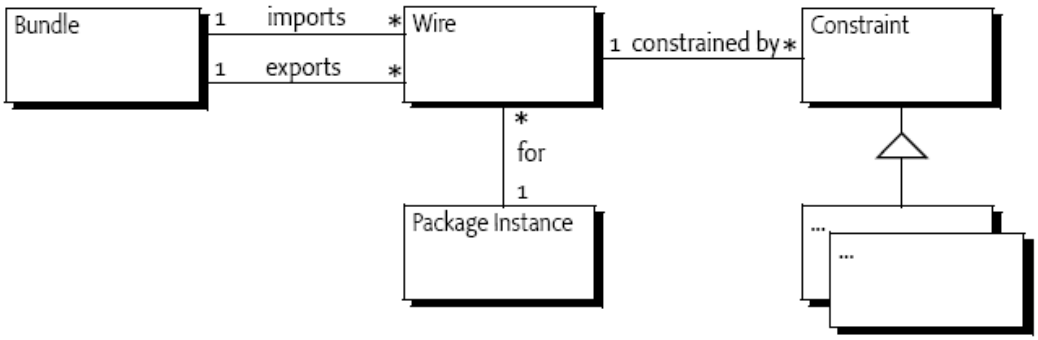


在类加载过程中，框架需要完成一系列的职责。在使用一个 bundle 之前，必须对共享的包之间的约束关系进行解析。选择一个最合适的类并创建连接（wiring），想请参考解析过程一节。在运行时类加载一节中也描述了一些运行时的特性。

3.4.1. 解析过程

框架必须支持bundle的解析。解析过程就是确定导入包如何连接到导出包(importers are *wired* to exporters)。解析过程需要满足约束条件。解析过程必须在bundle中任何代码加载或运行之前。

连接线是指导入包和导出包（都是bundle）之间的实际联系，连接线关联到一系列的约束，这些约束由导入导出包的manifest文件进行定义。一个有效的连接是满足它所有约束的连接。下图描述了连接模型的类结构。



3.5. 元数据处理

本节描述了 manifest 为解析过程提供的元数据信息。

3.5.1. Bundle-ManifestVersion

bundle 的 manifest 文件必须在名称为 Bundle-ManifestVersion 的头标中提供，它遵循 OSGi manifest header 语法。采用本规范或者后续版本规范的 bundle 必须要指定这个头标。头标的结构如下所示：

```
Bundle-ManifestVersion ::= number      // See 1.4.2
```

符合框架版本1.3的bundle的manifest版本必须是‘2’，在1.3之前版本的bundle的manifest版本为‘1’，虽然在这样的manifest中无法表述版本。因此，如果版本值不为‘2’，除非框架明确支持这样的后续版本，否则都是无效的标记。

OSGi框架的实现也可以支持没有Bundle-ManifestVersion头标的manifest，这样可以和框架1.2兼容。

版本号为‘2’的bundle必须指定bundle的符号名称（Symbolic Name）。而不需指定bundle的版本，这个头标有一个默认值。

3.5.2. Bundle-SymbolicName

头标 Bundle-SymbolicName(符号名称)是必须指定的。通过 bundle 的符号名称和版本号可以在框架中惟一的确定一个 bundle。也就是说，如果一个 bundle 和另外一个 bundle 有着同样的符号名称和版本号，那么这两个 bundle 就是等价的。

如果一个 bundle 的符号名称和版本号与已有的 bundle 相同，那么对这个 bundle 的安装肯定是失败的。

bundle 的符号名称由开发者确定（manifest 中的头标 Bundle-Name 是设计为人可读的，因此这个头标不会代替 Bundle-Name）。

bundle 的符号名称标记必须采用以下的规则：

```
Bundle-SymbolicName ::= symbolic-name
                        ( ';' parameter ) *    // See 1.4.2
```

框架必须识别在头标 Bundle-SymbolicName 中的以下指令：

- singleton：表示这个 bundle 只有一个版本可以解析。如果这个值为 true，那么表示是一个单态的 bundle。它的默认值是为 false。当同名且设为单态的 bundle 的多个版本同时安装在框架中，那么框架最多只能对一个 bundle 进行解析。如果两个 bundle 的符号名称相同，那么单态的 bundle 并不会影响到非单态的 bundle 的处理。
- fragment-attachment：定义了哪些片断(fragment)可以附加到 bundle 上。参考 bundle 片断一节。下面定义了哪些值是合法的 fragment-attachment 值：
  - always：片断可以在附主（host）解析完成之后，或者是在解析过程中附加。
  - never：不允许附加片断。
  - resolve-time：只允许在解析过程中附加。

示例：

```
Bundle-SymbolicName: com.acme.foo;singleton:=true
```

3.5.3. Bundle-Version

Bundle-Version 是一个可选的头标，它的默认值是0.0.0。如下例：

```
Bundle-Version ::= version  // See 3.2.4
```

如果小版本号（minor）和微版本号（micro）没有指定，那么他们的默认值为0。如果限定部分()没有指定，那么默认值为一个空字符串。

版本是可以进行比较的，比较是根据主版本号、小版本号、微版本号的顺序进行比较。最后是字符串的限定符比较。

例如：

```
Bundle-Version: 22.3.58.build-345678
```

3.5.4. Import-Package

Import-Package 头标定义了共享包的导入约束。本头标的规则如下：

```
Import-Package ::= import ( ',' import )*
import ::= package-names ( ';' parameter )*
package-names ::= package-name
                  ( ';' package-name )* // See 1.4.2
```

在这个头标中可以定义多个导入包。每条记录描述一个单独的导入包。其中指令如下：

- resolution：如果这个值为 mandatory，这也是默认值，那么这个包必须要解析，如果不能解析，那么对 bundle 的解析也会失败。如果这个值为 optional，那么这个导入包是可

选的。参考可选导入包一节

开发人员可以指定任意匹配的属性，参阅属性匹配一节。下面预定义了任意匹配属性：

- version：定义了导入包的版本范围。字段必须遵循版本范围一节中描述的规则。详情参阅版本匹配一节。如果没有指定这个属性，默认值为[0.0.0, ∞)。
- specification-version：是 version 属性的同义词，只是为了兼容原来的框架版本。如果指定了 version 属性，那么这两个值必须相等。
- bundle-symbolic-name：导出 bundle 的符号（symbolic）名称。
- bundle-version：选择导出 bundle 的版本范围。默认值为[0.0.0, ∞)，参阅版本匹配一节

为了允许导入包（除了以java开头的包之外），bundle必须有

PackagePermission[<package-name>,IMPORT]

参阅包权限一节。

以下错误会导致安装和更新的终止：

- 同样的指令或属性重复出现多次
- 同样的包进行多次导入定义

下面是一个正确的定义：

```
Import-Package: com.acme.foo;com.acme.bar;
      version="[1.23,1.24]";
      resolution:=mandatory
```

3.5.5. Export-Package

Export-Package头标和上述的Import-Package的规则类似。只是属性和指令不同。

```
Export-Package  ::= export ( ',' export )*
export          ::= package-names ( ';' parameter )*
package-names   ::= package-name           // See 1.4.2
                        ( ';' package-name )*
```

在头标中可以定义多个导出包。一个导出定义描述了一个 bundle 的导出包。可以对同一个包的多个导出定义，对不同的 importer 定义不同的属性。

导出指令如下：

uses：定义导出包所用到的包清单，采用逗号分隔包名。注意，逗号需要用双引号包括起来。如果对导出包进行导出，那么解析器必须保证这个包的导入连接到了本清单中描述的相同的版本的包。参阅包约束一节。

- mandatory：逗号分隔的属性名称清单。同样，逗号需要用双引号包括起来。导入这个包的 bundle 必须指定 mandatory 值描述的属性，以便于进行导入包的解析。参考强制属性一节。
- include：逗号分隔的类名清单，描述对 importer 必须可见的类。逗号需要用双引号包括起来。参考类过滤一节。
- exclude：描述对 importer 必须是不可见的类和资源名称清单，采用逗号分割，逗号需要用双引号包括起来。参阅类过滤一节。

下面的属性是本规范的一部分：

- version：描述包的版本，定义了相关包的版本，默认值为 0.0.0。
- specification-version：版本的别名，为了支持以前的版本。如果指定了 version 属性，那么这两个值必须相等。

另外，任意匹配的属性也可以进行说明。参阅属性匹配一节。
框架自动关联每一个定义了如下属性的导出包：

- bundle-symbolic-name：导出 bundle 的符号名称。
- bundle-version：导出 bundle 的版本号。

如果以下任何条件为真，那么将对导致安装或者更新的终止：

- 一条指令或者属性出现多次。
- 在头标 Export-Package 中指定了 bundle-symbolic-name 或者 bundle-version

在导出定义中没有隐含自动导入定义。如果一个 bundle 导出一个包而没有导入这个包，这个 bundle 将通过 bundle 类路径获得这个包。只有导出的包可以用于其它 bundle，但是导出的 bundle 不能使用来自其它 bundle 的同样的包。
为了导出一个包，bundle 必须有以下字段：
PackagePermission[<package>, EXPORT]。
例如：

```
Export-Package: com.acme.foo;com.acme.bar;version=1.23
```

3.5.6. 导入和导出一个包

导出一个包并不意味了导入同样的包（在 OSGi R3 中，导出意味着导入同样的包）。这样改进的原因在于使得 bundle 可以对其他 bundle 提供包，而无需考虑导出包由解析器从其它 bundle 中用同样的包来替代。如果一个应用是由一系列的相互关联紧密的 bundle 组成，而在这些 bundle 中，实现包提供给其他 bundle，那么这样的情况是很常见的。
在包的互操作中，这种包的替代至关重要。在 java 中，只有 bundle 中同样的类使用相同的类加载器，那么 bundle 才能进行互操作（inter-operate）。因此，如果两个 bundle 导出了同样的包，但是没有导入这个包，那么这两个 bundle 就不能共享这个包中的对象。这一点对于协作机制非常重要。例如在服务层中，如果 bundle 的服务对象是来自同样的类加载器，那么他们只能使用同样的服务对象。
bundle 应该导入已经导出的包，允许解析器替代包含了接口和其他共享类型的包。这种替代允许 bundle 通过服务注册和其他机制进行互操作。另外，为了使解析器保持最大限度的灵活性，导入者应该尽可能的减少约束。

3.5.7. 对遗留 bundle 的处理

如果在头标 Bundle-ManifestVersion 中没有指定值为 2 或者更大数，那么就根据发布的版本 3（R3）中的定义来解释。也就是说，框架必须根据 R3 头标定义转换为 R4 头标：

- Import-Package：将 specification-version 转换为 version 属性。如果没有定义这个属性，那么就无需转换，因为 R3 中的 version 属性默认值为 0.0.0，具有相同的语义。
- Export-Package：将 specification-version 转换为 version 属性。导出的定义中必须含有 uses 指令，uses 指令必须包含所有的给定 bundle 的导入导出包。另外，如果这个包没有导入定义，那么必须添加给定版本的导入定义。

- DynamicImport-Package: 定义没有修改。

如果在 bundle 的 manifest 中 version 属性标记为 2，但是又同时存在着遗留语法标记，那么这个错误会导致安装失败。

在版本为 2 的 manifest 中，不赞成使用 specification-version 属性。

3.6. 约束处理

OSGi 框架包解析器提供了一系列的导入与导出匹配的机制。本节描述了这些详细的机制。

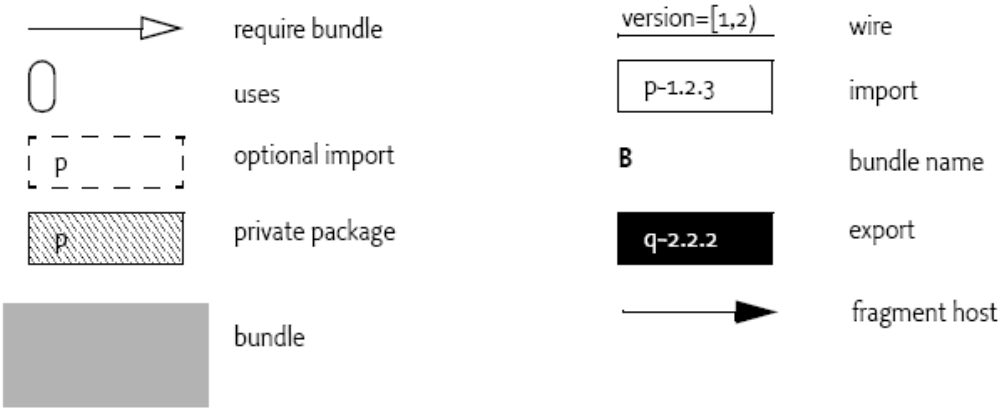
3.6.1. 图表和标记

线（wire）将节点（nodes）关联到一起形成了图（graph）。线和节点（即 bundle）一样，包含了大量重要的信息。下一节，采用以下约定来解释一些细节：

bundle 的名称为 A、B、C 等，也就是从 A 开始的大写字母。用 p、q、r、s、t 等命名包。也就是从 p 开始的小写字母。如果一个版本很重要，那么版本的标记后加上一个短划线，如：q-1.0。标记 A.p 表示在名称为 A 的 bundle 中定义（导入或者导出）了包 p。

采用白色方框表示导入定义，黑色方框表示导出定义，没有导入导出的包称之为私有包，用斜线网格背景表示。

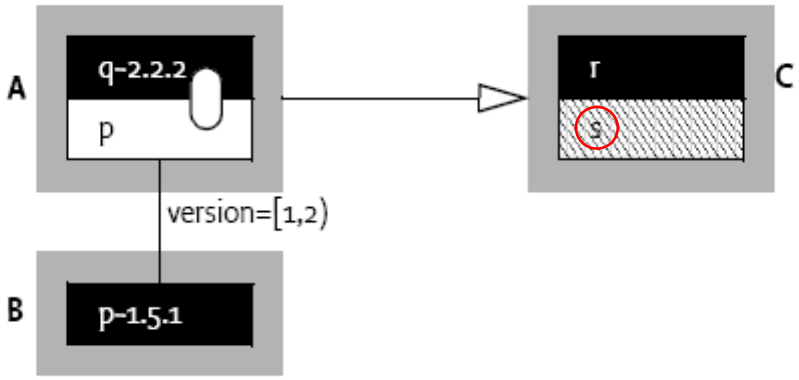
Bundle 是一系列关联的方框的集合。Bundle 之间的关联用线(wire)表示，而约束写在这些线上。



例如：

```
A: Import-Package: p; version="[1,2)"
   Export-Package: q; version=2.2.2; uses:=p
   Require-Bundle: C
B: Export-Package: p; version=1.5.1
C: Export-Package: r
```

可以用下图来描述 A、B、C 这三个 bundle：



3.6.2. 版本匹配

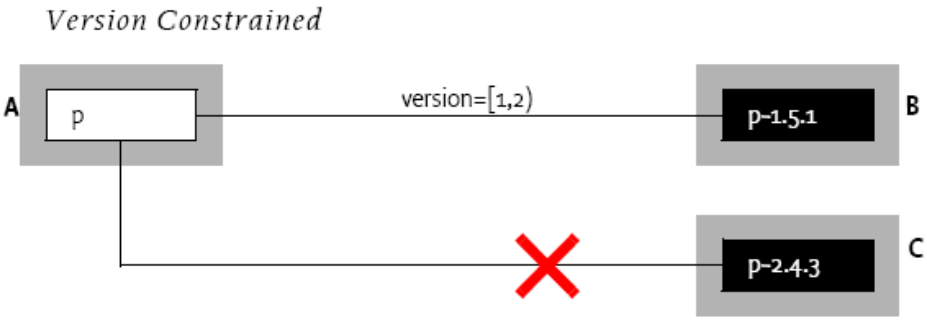
在版本机制中，对匹配导出定义的版本和版本范围采用了精确的描述。版本范围的编码考虑了兼容性。本规范没有定义任何兼容性原则，而留待 importer 制定一个版本范围。一个版本的范围嵌入了兼容性的原则。
但是，最常见的版本兼容原则如下：

- 主版本号（major）：不兼容的更新
- 副版本号（minor）：向后兼容的更新
- 小版本号（micro）：不影响接口的更新。例如，修正了一个错误。

一个导入定义必须指定一个版本范围作为导入的 version 属性值，exporter 必须指定一个版本号作为它的 version 属性。根据版本范围一节中的描述进行版本的匹配。
在下例中，A 中的导入和 B 中的导出可以正确进行匹配。

```
A: Import-Package: p; version="[1,2)"
B: Export-Package: p; version=1.5.1
```

下图描述了根据规则排除 exporter。



3.6.3. 可选包

在 bundle 中可以指定它不需要正确的解析一个包，但是如果这个包可用就使用它。例如，登录非常重要，但是如果没有登录服务，bundle 也应该可以运行的。
可选导入包可以通过以下方法指定

- Dynamic Imports 动态导入：DynamicImport-Package 头标描述了需要的导出包。如果 bundle 在调用之前不知道类名，关键技术是通过使用类的 forName 方法来加载类。
- Resolution Directive resolution 指令：在导入定义中通过 resolution 指令指定一个可选值。如果没有合适的可选包，那么 bundle 也可以成功解析。

这两种机制的区别是：

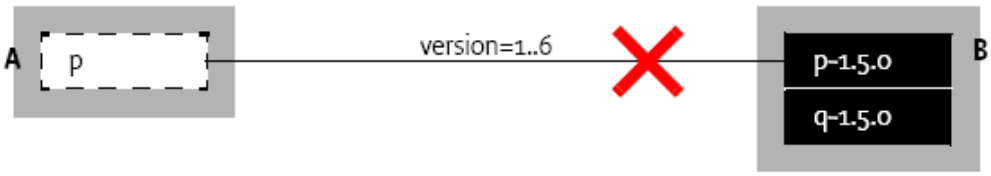
- 可选 vs 动态：对于动态导入，每次加载包中的类都会尝试与动态导入包进行建立连接，而可选导入包只有在 bundle 解析的时候才进行连接尝试。

导入定义的 resolution 指令中，它的值可以为 mandatory(强制的)或者是 optional(可选的)。

- mandatory：默认值，指定包必须连接到 bundle。
- optional：指定在包没有连接时也可以对导入 bundle 进行解析。

下例中，即使 bundle B 没有提供正确的匹配版本，bundle A 也可以进行解析。

```
A: Import-Package: p;
        resolution:=optional;
        version=1.6
B: Export-Package: p;
        q;
        version=1.5.0
```



bundle 的实现中，必须要考虑到可选包没有加载的情况。比如，抛出一个相关的找不到包的异常。

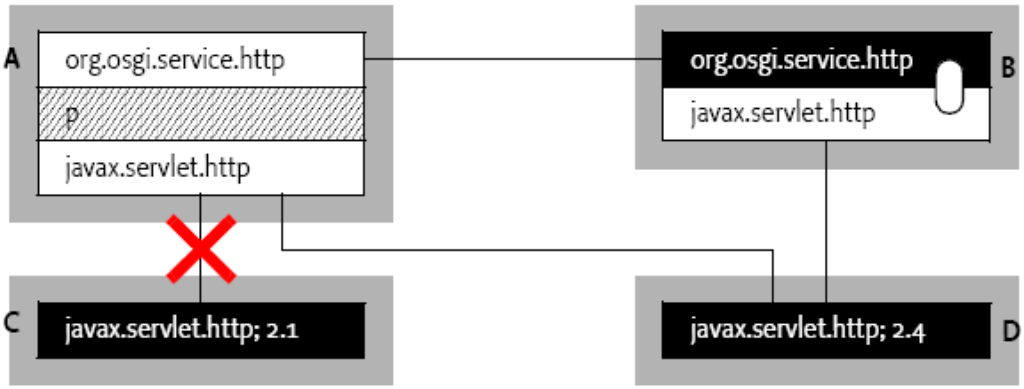
3.6.4. 包约束

一个类可以依赖于其他包中的类。例如，从其他包中的类继承，或者这些类出现在方法的声明中。这种情况称之为一个包使用了其他包。这种包之间的关系通过在 Export-Package 头标中使用 uses 指令来描述。

例如：org.osgi.service.http 使用包 javax.servlet 中的 API，所以这两个包存在依赖关系。在包 org.osgi.service.http 中必须包含值为 javax.servlet 的 uses 指令。

只有设置 bundle 中每一个包都只有一个 exporter，才可以保证类空间的一致性。

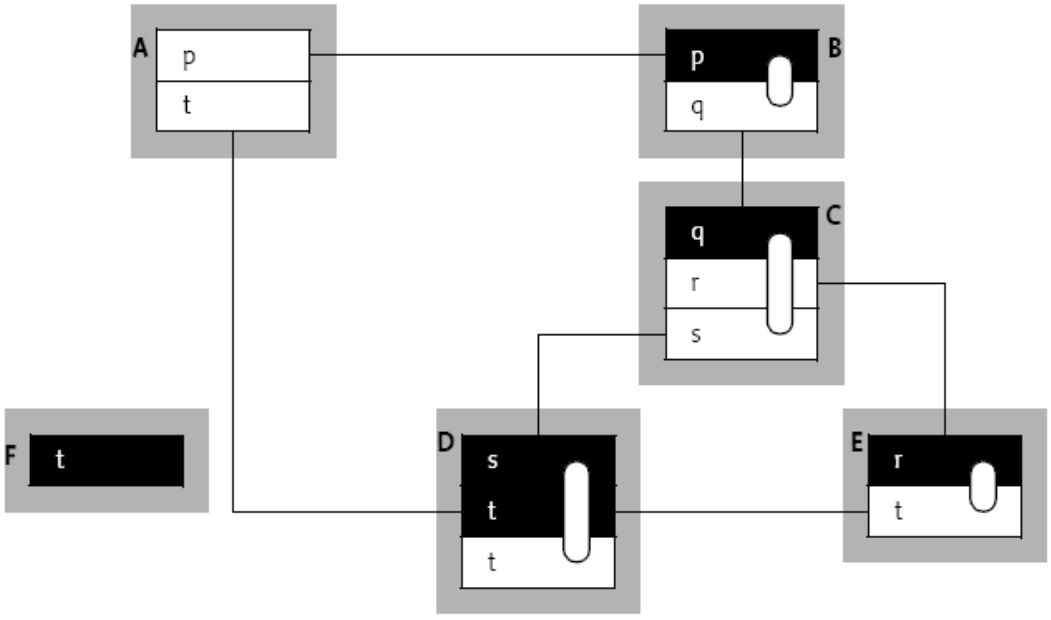
例如，Http 服务的实现需要从 javax.servlet.http.HttpServlet 中继承的 servlets 类，如果 Http 服务 bundle 导入 version 值为 2.4 而客户端 bundle 导入 version 为 2.1，那么必然发生类声明错误。如下所示：



如果 bundle 从一个 exporter 中导入了一个包，那么在导出定义中可以通过 uses 指令隐含一系列对其他包的约束。uses 指令列出了 exporter 依赖的一系列的包，这些约束保证了这一系列的 bundle 对于相同的包共享同样的类加载器。

如果一个导入者导入了含有约束的包，解析器必须通过约束建立 importer 和 exporter 之间的连接（wire）。而 exporter 可能也有同样的约束。这样，建立单个导入包和 exporter 之间的连接可以隐含有一大串的约束。术语导出包约束（*implied package constraints*）是指通过这种递归循环构建的约束集合，也隐含包约束并不包括自动导入，进一步说，隐含包约束只包括了在导入定义中描述的必须解析的内容。

如下图所示，bundle A 导入了包 p，假设定义的 p 连接到 bundle B。由于 uses 指令（在此省略 uses 指令描述）隐含了对包 q 的约束。进一步说，假设对包 q 的导入关联到 bundle C，那么也就隐含了对包 r 和 s 的约束。接下来，假设 C.s 和 C.r 分别连接到 bundle D 和 E，这两个 bundle 都将包 t 添加到 bundle A 的依赖包集合中。



为了维护类空间的一致性，框架必须确保 bundle 的导入不能和 bundle 依赖包中存在冲突。例如，这也就是说框架必须保证 A.t 的导入定义连接到 D.t。如果导入定义连接到包 F.t，那么就违背了类空间的一致性。这是由于在 bundle A 中，同时存在着来自 bundle D 和 F 的两个同名的类。这将导致一个类声明异常（ClassCastExceptions）。或者，如果所有的 bundle 连接到 F.t，也同样可以解决问题。

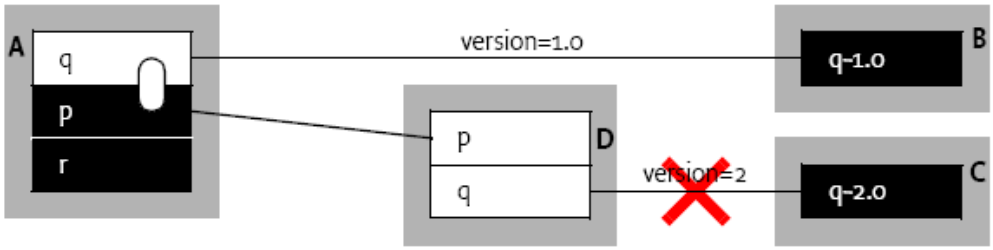
另一种情况是图 21 所示，bundle A 从 B 中导入了 Http 服务类，bundle B 中包含了 org.osgi.service.http 和 javax.servlet，bundle A 和 bundle B 同样的存在连接到 javax.servlet 的约束。
下面的示例说明了通过对 uses 指令的设置，使得不可能进行解析。

```
A: Import-Package: q; version="[1.0,1.0]"
   Export-Package: p; uses:="q,r"
B: Export-Package: q; version=1.0
C: Export-Package: q; version=2.0
```

由于 A.q 可以连接到 B.q 而不是 C.q，这种约束可以解析的。
添加 bundle D 之后就不可能解析了：

```
D: Import-Package: p, q; version=2.0
```

由于bundle A是p惟一的exporter，包D.p必须连接到包A.p。但是，根据包A.q中的uses指令隐含了包q，包A.q连接到B.q-1.0，而D.q需要版本为2.0的包，由于违背了类空间约束的导致解析失败。
下图描述了这种情况：



3.6.5. 属性匹配

属性匹配允许 importer 和 exporter 通过说明性的途径来影响匹配过程。为了使得导入定义属性可以解析为导出定义，导入中定义的属性必须要和导出中定义的属性匹配。默认情况下，如果导出定义中包含的属性没有出现在导入定义中，匹配过程继续进行。如果在导出定义中指定了 mandatory 指令，则框架在导入定义必须要匹配这些属性。在解析阶段，对于出现在字段 DynamicImport-Package 中的任何信息都是忽略处理。
例如，下例的语句是匹配的：

```
A: Import-Package: com.acme.foo;company=ACME
B: Export-Package: com.acme.foo;
           company="ACME";
           security=false
```

除了字段version和bundle-version，其他的属性值都是通过字符串方式进行比较。这两个字段采用版本范围的比较方法。属性中开始和结尾部分的空格都是忽略不计的。

3.6.6. 强制属性

属性分为两种：mandatory（强制的） 和 optional（可选的）。mandatory 属性表示必须匹配的属性。optional 属性表示可以在导入时不考虑的属性，默认值为 optional。
exporter 可以在导出定义中通过 mandatory 指令指定强制属性，这条指令包含了一个逗号分割的属性名称列表，表示必须在导入描述中匹配的属性。如下例所示，导入包和导出包不是匹配的，由于在 A 中没有指定 security 属性。

```
A: Import-Package: com.acme.foo;company=ACME

B: Export-Package: com.acme.foo;
           company="ACME";
           security=false;
           mandatory:=security
```

3.6.7. 类过滤

exporter 可以通过在导出定义中使用 include 和 exclude 指令来限制对类的访问范围。这两条指令的值都是逗号分割的类名列表。注意，逗号需要用双引号包括起来。类名不能包含有包名称，而且不能含有后缀.class。也就是说，类 com.acme.foo.Daffy 在指令中名称为 Daffy。类名称中可以含有通配符（'*'）。
对于 include 指令，它的默认值为通配符'*'（表示所有的名称），也就是说所有的类和资源。而 exclude 指令的默认值为空，一个空的名称表示没有匹配的名称。如果指定了这两个指令的值，那么默认值就被覆盖了。
如果满足以下条件，一个类是可见的：

- 在 include 中一条记录匹配，而且
- 在 exclude 中没有记录可以匹配

在其他情况下，加载或者搜索失败，类加载器抛出一个无法找到类的异常（Class Not Found Exception）。在指令中，不考虑排列顺序。
下例列出了一个导出段和其中文件的可见性。

```
Export-Package: com.acme.foo; include:="Qux*,BarImpl";
        exclude:=QuxImpl

com/acme/foo
   QuxFoo           visible
   QuxBar           visible
   QuxImpl          excluded
   BarImpl          visible
```

使用过滤器时必须小心。例如，模块的一个新版本需要和旧版本兼容，那么就不应该将旧版本中没有排除的类和资源排除在外（exclude）。另外，模块化现有的代码时，从导出包中排除的类和资源有可能中断和包连接的用户。例如，正是定义的包通常在正式包中有一个实现类，而正式

包对定义包有访问权限。

```
package org.acme.open;
public class Specified {
    static Specified implementation;
    public void foo() { implementation.foo(); }
}
package org.acme.open;
public class Implementation {
        public void initialize(Specified implementation) {
            Specified.implementation = implementation;
    }
}
```

由于实现类允许设置，因此对于扩展bundle必须是不可见的。可以通过将实现类排除在外，使得只有导出bundle可以访问这个类。导出定义的标记描述如下：

```
Export-Package: org.acme.open; exclude:=Implementation
```

3.6.8. 选择提供者(Provider selection)

“提供者选择（provider selection）”指允许 importer 选择一个 bundle 来作为 exporter。如果在一个 importer 和 exporter 之间没有规格约束，则进行“提供者选择”。importer 与一个特定的 exporter 紧密关联，一般情况下是用于测试的 bundle。为了降低连接的脆弱性，importer 可以指定一个可以选择的范围。

importer 可以通过导入属性 bundle-symbolic-name 和 bundle-version 选择一个 exporter。框架自动为每个导出定义(export definition)提供了这些属性(attributes)。这些属性不能在导出定义中进行指定。

导出定义的 bundle-symbolic-name 属性包含了 bundle 的符号名称，它是在头标 Bundle-SymbolicName 中不带参数进行指定的。导出定义中的 bundle-version 属性被设置为头标 Bundle-Version 的值，或者默认值为 0.0.0。

属性 bundle-symbolic-name 的匹配采用属性匹配方法。bundle-version 的匹配采用版本范围比较方法，在版本范围一节中有详细描述。导入定义必须有一个版本范围而导出定义是一个版本号。

如下例，定义是可以匹配的：

```
A: Bundle-SymbolicName: A
   Import-Package: com.acme.foo;
        bundle-symbolic-name=B;
        bundle-version="[1.41,2.0.0)"

B: Bundle-SymbolicName: B
   Bundle-Version: 1.41
   Export-Package: com.acme.foo
```

下面的例子是不能进行匹配的，B没有指定一个版本，取其默认值为0.0.0。

```
A: Bundle-SymbolicName: A
   Import-Package: com.acme.foo;
          bundle-symbolic-name=B;
          bundle-version="[1.41,2.0.0)"

B: Bundle-SymbolicName: B
   Export-Package: com.acme.foo;version=1.42
```

通过符号名称选择一个 exporter 可能会导致脆弱性，这是由于将包和 bundle 强耦合在一起。例如，如果一个 exporter 被重构为多个单独的 bundle，那么所有涉及的 importer 都需要进行修改。其他的任意匹配属性由于是独立的，所以没有这样的缺陷。
Bundle 重构时由符号名称带来的脆弱性问题，可以通过使用一个与原 bundle 同名称的 façade 模式 bundle 来取得部分的解决。

3.7. 解析过程

解析过程就是建立 bundle 之间的连接(wire)的过程。连接之间的约束由以下条件静态定义：

- 导入导出包（此阶段不考虑 DynamicImport-Package 的内容）
- required bundles，需要的 bundle,它们导入了 bundle 所有的导出包，具体定义参见后文的“bundle 需求”一节
- Fragments,片断，向宿主提供内容和定义，在 bundle 片断一节有详细定义

在 bundle 被解析之前，它所有的片断都必须已经 attach 到 bundle 之上。解析过程就变成了一个约束求解算法，可以使用“关联关系的要求”进行描述。解析处理就是对解空间的迭代搜索过程。
如果一个模块(module)对同一个包有导入和导出定义，那么框架需要判断选择哪一个定义。必须首先处理重复的导入定义。下面是可能的结果：

- 外部(external)—如果解析到另一个 bundle 的导出定义，那么不考虑在这个 bundle 中的重复导出定义。
- 内部(internal)—如果解析到了这个模块中的导出定义，那么不考虑这个模块的重复导入定义。
- 无法解析—没有匹配的导出定义。这是由于开发者的错误，也就是说重复的导出定义和对应的导入定义不相匹配。

如果满足以下条件，bundle 可以进行解析：

- 所有的必需的导入都已经连接
- 所有 required bundle 都是可用的而且建立了他们的导出连接。

而连接的建立需要满足以下条件：

- importer 的版本范围和 exporter 的版本匹配。参见版本匹配的说明。
- importer 具有 exporter 指定的所有的强制属性。参见强制属性说明一节。
- importer 的所有的属性和 exporter 的相应属性匹配。参见属性匹配一节。
- 如果连接到同一个 exporter，那么也就隐式关联到同一个包。参见包约束一节。
- 连接关联到一个合法的 exporter。

下面列表定义了优先级别，如果有多个选择，根据优先级降序排列：

- 一个已经解析的 exporter 优先于一个未解析的 exporter。

- 具有更高版本的 exporter 优先于一个低版本的 exporter。
- 具有较低 bundle ID 的 exporter 优先于较高 ID 的 exporter。

3.8. 运行时类加载

框架中安装的 bundle 只有在解析之后才能关联到类加载器。bundle 解析之后，框架必需为每一个非 fragment 的 bundle 创建一个类加载器。框架也可以延迟创建类加载器，直到它实际需要才创建。

每个 bundle 对应一个类加载器的机制使得 bundle 中的所有资源对于 bundle 中位于同一个包的其他资源具有包级别的访问权限。此类加载器为每一个 bundle 提供了它自己的命名空间，以避免命名冲突，并允许 bundle 之间的资源共享。

类加载器必须通过利用在解析过程中建立的连接(wire)来找到适当的 exporter。如果一个类没有在导入包中找到，那么根据在附加的 manifest 中的定义在附加的空间进行查找。

本节定义了影响运行时类加载的因素，并定义了类和资源加载时框架必须遵循的查找顺序。

3.8.1. bundle 类路径

在manifest中的Bundle-Classpath定义了bundle的内部类路径依赖。这种定义允许bundle通过在其JAR文件内使用多个JAR文件或者目录声明“嵌入”类路径。

在Bundle-Classpath manifest header中定义了逗号分割的文件名称列表，文件名称可以是：

- 点号（‘.’，\u002E）代表bundle本身的JAR文件，如果没有指定Bundle-Classpath，那么它的默认值就是这个。
- 在bundle的JAR文件中包含的JAR文件路径
- 在bundle的JAR文件中包含的目录路径

Bundle-Classpath必须符合以下语法：

```
Bundle-Classpath::=  entry ( ',' entry )*
entry            ::= target ( ';' target )*
     ( ';' parameter ) *
target           ::= path | '.'          // See 1.4.2
```

框架必须忽略任何不可识别的参数。

如果在 Bundle-Classpath 中指定的目标（目录或者是 JAR 文件）不能在需要的时候进行定位，那么框架必须忽略这样的目标，这在 bundle 解析之后随时可能发生。然而，在这种情况下，框架还应该发出一个类型为 INFO 的框架事件（Framework Event），带有关于它无法定位的每个条目的信息。

在定位 bundle 的类路径时，框架必需从 bundle 的 JAR 文件的根开始进行相对定位。如果一个类路径不能在 bundle 中定位，框架必需试图在附加的 fragment bundle 中进行定位。附加的 bundle fragment 根据 bundle ID 升序查找。这就允许 fragment 将一些条目插入到附主的 Bundle-Classpath 中。

如下示例：

```
A: Bundle-SymbolicName: A
   Bundle-Classpath: required.jar,optional.jar,default.jar
   content ...
   required.jar
   default.jar
B: Bundle-SymbolicName: B
   Bundle-Classpath: fragment.jar
   Fragment-Host: A
   content ...
   optional.jar
   fragment.jar
```

在这个例子中，bundle A 的Bundle-Classpath描述了三个条目（required.jar，optional.jar，和default.jar）。required.jar条目可以是必须出现在bundle中的类和资源，optional.jar类路径条目可以是在可用的时候bundle可以使用的类和资源。

default.jar类路径条目描述的是如果optional.jar条目不可用时候的默认值，可以在optional.jar中对其覆盖。bundle A只有required.jar和default.jar条目，而bundle B可以为A提供optional.jar条目。

片断bundle B也有Bundle-Classpath描述了一个条目信息（fragment.jar），当bundle A解析后，片断bundle B附加到A之上，那么现在bundle A的类路径如下：

```
required.jar, optional.jar, default.jar, fragment.jar
```

3.8.2. 动态导入包(Dynamic Import Package)

动态导入和导出定义进行匹配（形成包连接）是在 class loading 的时候进行的，从而不会影响模块的解析。

动态导入只会作用于没有建立连接，而且又找不到其他定义的包。动态导入是最后的解决方法。

```
DynamicImport-Package ::= dynamic-description
          ( ',' dynamic-description )*

dynamic-description::= wildcard-names ( ';' parameter )*
wildcard-names     ::= wildcard-name ( ';' wildcard-name )*
wildcard-name      ::= package-name
        | ( package-name '.*' )   // See 1.4.2
        | '*'
```

DynamicImport-Package 中没有规定任何指令。可以指定一些匹配属性。下面的属性会由框架进行匹配：

- version—版本范围，用于选择导出定义的版本。默认值为 0.0.0。
- bundle-symbolic-name — 导出 bundle 的符号标记。
- bundle-version — 版本范围，用于选择 bundle 版本。默认值为 0.0.0。

包名可以是明确指定的，也可以是含有通配符如 org.foo.*或者是*，通配符后缀可以表示任

何标记，包括多个子包。
动态导入必须根据指定的顺序查找，尤其是在通过通配符指定包名的情况尤为重要。排序器在匹配的时候进行排序。也就是说越是明确指定的包，出现的也越靠前。例如，下面的例子指定了优先选择由 ACME 公司提供的包。

DynamicImport-Package: *;vendor=acme, *

如果多个包需要通过同一个参数进行动态导入，那么可以在参数之前通过在其中指定多个包，由分号分割。
在类加载过程中，类所在的包根据指定的包名（可能包含通配符）进行加载。每一个匹配的包名循环使用，试图使用和 Import-Package 同样的规则与 exporter 建立连接。如果连接尝试获得成功（考虑任何 uses 约束），搜索结果转发到导出的类加载器，继续进行类加载过程。这个连接在随后过程中都不能修改，即使类加载失败。这也就是说一旦包被动态解析了，那么随后的类或者资源的加载和普通导入是没有区别的。
为了将 DynamicImport-Package 解析为导出声明，动态导入中定义的所有属性必须和导出声明中的属性匹配。所有的强制任意属性（由 exporter 指定，参见强制属性）必须在动态导入定义中指定和匹配。
一旦建立了连接，在以后的动态导入中必须遵循 exporter 的任何 uses 约束。
动态导入非常类似于可选包，参见可选包一节，但它是在 bundle 解析之后处理的。

3.8.3. 父级代理(Parent Delegation)

框架必须将以 java.开头的包交由父类加载器代理。
有一些 Java 虚拟机，包括 SUN 公司的虚拟机，错误的假定父级代理是经常发生的。这种关于严格分层的类加载器代理的假定会导致类没有定义的错误（NoClassDefFoundErrors）。如果虚拟机期待从任何类加载器中都能找到自己实现的类，而没有严格要求启动类加载器(boot class loader)只加载 java.*包，就会发生上述错误。
其他必须通过启动类加载器加载的包可以通过在系统属性中指定：

org.osgi.framework.bootdelegation

这个属性必须包含一个如下格式的列表：

```
org.osgi.framework.bootdelegation ::= boot-description
          ( ',' boot-description )*
boot-description::= package-name                    // See 1.4.2
        | ( package-name '.*' )
        | '*'
```

通配符可以进行深度匹配，与这个序列匹配的包必须通过父类加载器来加载。而 java.*前缀是默认指定的，不需要声明。
单个的通配符表示框架必须首先交由父类加载器代理，这和发布版本 R3 是一样的。例如，如果运行于一个 SUN 的虚拟机，那么这个属性值可以指定如下：

org.osgi.framework.bootdelegation=sun.*,com.sun.*

通过这个属性值，框架必须将所有的 java.*，sun.*和 com.sun.*的包交由父类加载器代理。

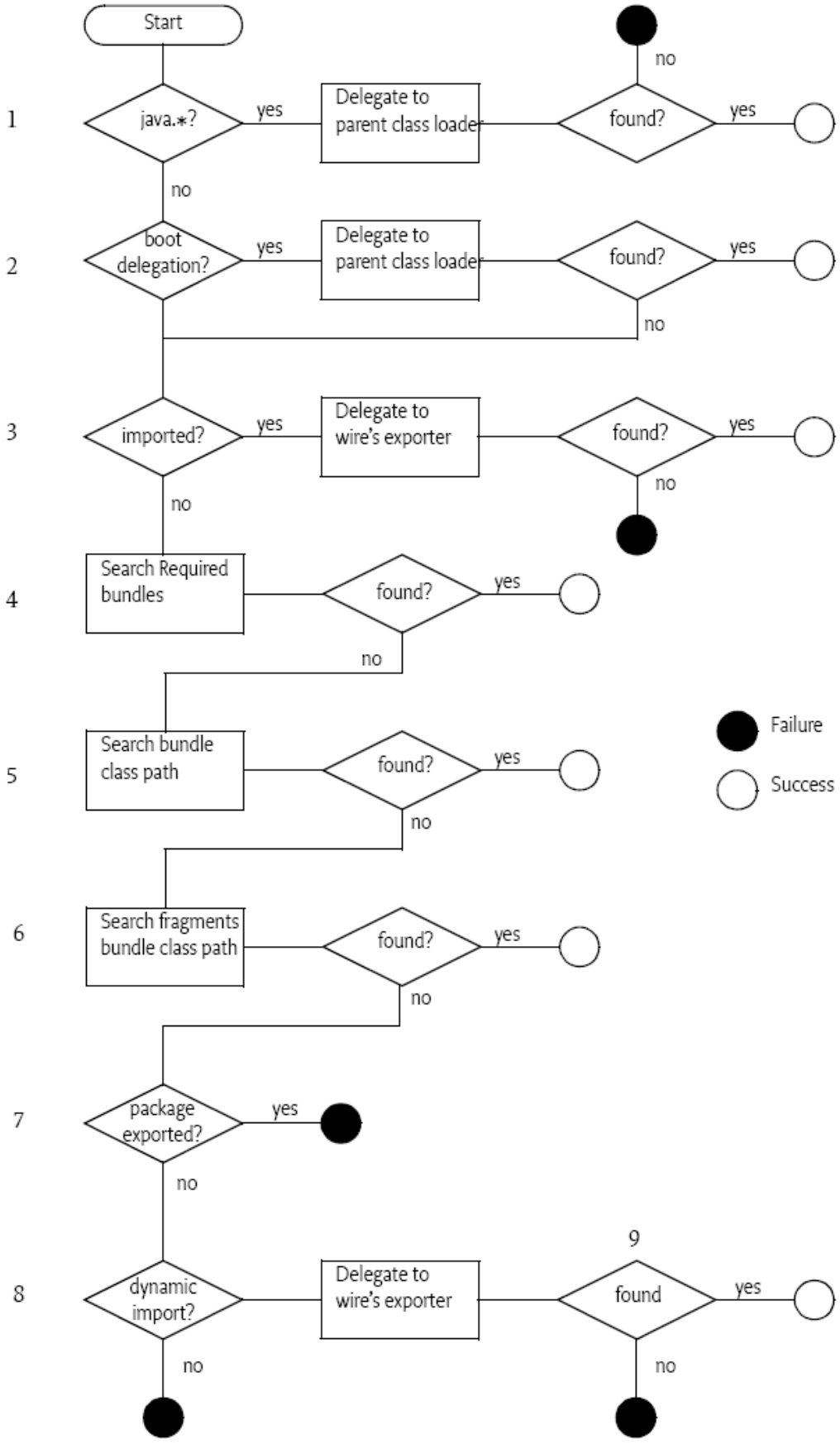
3.8.4. 完整查找顺序

在类和资源加载过程中，框架必须遵循以下规则。当请求一个 bundle 类加载器进行类加载或者资源的查找，查找必须按照以下顺序执行：

1. 如果类或者资源是在包 java.*中，那么交由父级类加载器代理完成，否则，搜索过程进入第二步。如果父级类加载器加载失败，那么查找过程结束，加载失败。
2. 如果类或者资源在启动代理序列（org.osgi.framework.bootdelegation）中定义，那么交由父级代理完成，如果找到了类或者资源，那么查找过程结束。
3. 如果类或者资源所在的包是在 Import-Package 中指定的，或者是在此之前通过动态导入加载的，那么将请求转发到导出 bundle 的类加载器，否则搜索继续进行下一步。如果将请求交由导出类加载器代理，而类或者资源又没有找到，那么查找过程中止，同时请求失败。
4. 如果类或者资源所在的包是使用 Require-Bundle 从一个或多个其他 bundle 进行导入的，那么请求交由那些 bundle 的类加载器，按照 bundle 的 manifest 中指定的顺序进行查找。如果没有找到类或者资源，搜索继续进行。
5. 使用 bundle 本身的内部 bundle 类路径查找。如果类或者资源还没有找到，搜索继续到下一步。
6. 查找每一个附加的 fragment 的内部类路径，fragment 的查找根据 bundle ID 顺序升序查找。如果没有找到类或者资源，查找过程继续下一步。
7. 如果类或者资源所在的包由 bundle 导出，或者包由 bundle 导入（使用 Import-Package 或者 Require-Bundle），查找结束，类或者资源没有找到。
8. 否则，如果类或者资源所在的包是通过使用 DynamicImport-Package 进行导入，那么试图进行包的动态导入。exporter 必须符合包约束。如果找到了合适的 exporter，然后建立连接，以后的包导入就可以通过步骤三进行。如果连接建立失败，那么请求失败。
9. 如果动态导入建立了，请求交由导出 bundle 的类加载器代理。如果代理查找失败，那么查找过程中止，请求失败。

如果代理转移到有另一个 bundle 类加载器，被代理请求的进入到算法第三步。

下面的非标准流程图展示了查找的过程：


Start
1
java.*?
yes
Delegate to parent class loader
found?
yes
no
no
2
boot delegation?
yes
Delegate to parent class loader
found?
yes
no
3
imported?
yes
Delegate to wire's exporter
found?
yes
no
no
4
Search Required bundles
found?
yes
no
5
Search bundle class path
found?
yes
no
6
Search fragments bundle class path
found?
yes
no
7
package exported?
yes
no
8
dynamic import?
Delegate to wire's exporter
9
found
yes
no
no
Failure
Success

3.8.5. 父类加载器

隐含导入包都是 java.*包，这是由于它们都是 Java Runtime 所需要的，而且同时进行多版本控制不是那么容易。例如，所有的对象都是扩展自同一个类 Object。
bundle 中绝对不能导入或者导出 java.*包，这样做是一个错误，任何这样的 bundle 必须安装失败。对正在执行的 bundle 来说，父类加载器可以获得的其他所有包必须被隐藏。
但是，框架必须明确导出和父类加载器关联的包。系统属性：

org.osgi.framework.system.packages

包括了系统 bundle 导出的包的描述。这个属性采用标准的 Export-Package 语法描述

```
org.osgi.framework.system.packages ::= package-description (
',' package-description )*
```

在启动类路径上的一些类假定他们可以使用任何类加载器来加载位于根类路径上的其他类，这对于 bundle 的类加载器来说是错误的。框架实现者应该试图从启动类路径加载这些类。
系统 bundle（bundle 的 ID 为 0）用于从父类加载器导出非 java.*的包。系统 bundle 导出定义看作是普通的导出，也就是说他们可以有版本号码，可以作为普通 bundle 的解析过程的一部分来用于解析导入定义。其他 bundle 也可以为同样的包提供一个替代的实现。
父类加载器中的导出定义序列也可以根据这个属性设置，或者由框架计算。导出定义必须实现了特定 bundle 符号名称的实现和系统 bundle 的版本的值。
这种风格的父类加载器中对外暴露的包必须考虑到其下的包的 uses 指令。例如，包 javax.crypto.spec 的定义必须声明包 javax.crypto.interfaces 和包 javax.crypto 的使用。

3.8.6. 资源加载

bundle 中的资源可以通过 bundle 的类加载器进行访问，也可以通过方法 getResource，getEntry 或者 findEntries 进行访问。所有这些方法都返回一个对象的 URL 或者是关于多个 URL 对象的枚举列表（Enumeration），这些方法返回的 URL 的模式可以是不一样的，由具体的实现来决定。
通常，bundle 的入口 URL 是由框架来创建的，然而，在特定情况下，bundle 需要操作 URL 来查找相关资源。这样框架需要保证：

- bundle 入口的 URL 必须是分层次的（参见统一资源标志符 URI 一节）
- 作为构建其他 URL 的上下文环境
- java.net.URLStreamHandler 类用于 bundle 入口 URL 地址，这个类必须是类 java.net.URL 可以访问的，这样来根据框架定义的协议模式来初始化一个 URL
- bundle 入口 URL 的 getPath 方法必须返回资源或者 bundle 入口的绝对路径（以‘/’开头）。例如，getEntry("myimages/test.gif")方法返回的路径必须包含有/myimages/test.gif。

例如，一个类可以通过一个 URL 访问 bundle 资源 index.html，也可以是在同一个 JAR 文件目录下的对其他文件的 URL 映射表。

```
public class BundleResource implements HttpContext {
   URL root;  // to index.html in bundle
   URL getResource( String resource ) {
      return new URL( root, resource );
   }
   ...
}
```

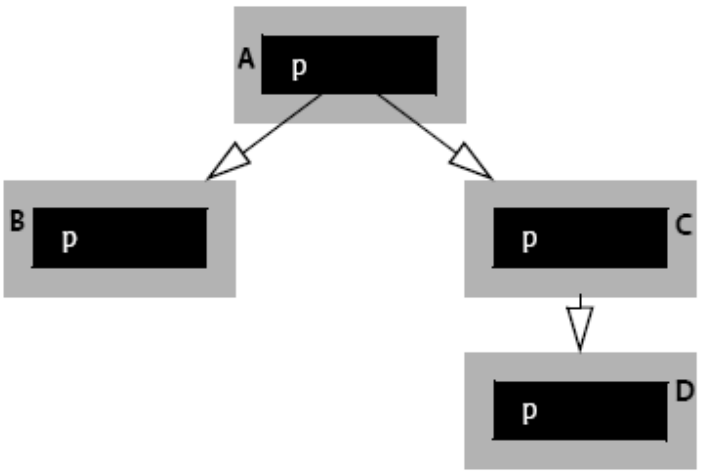
3.8.7. Bundle 环

多重需求的 bundle 可能会导出同样的包。在包中类和资源的查找过程中，这样做会导致查找形成的 bundle 环。
考虑如下定义：

A: Require-Bundle: B, C

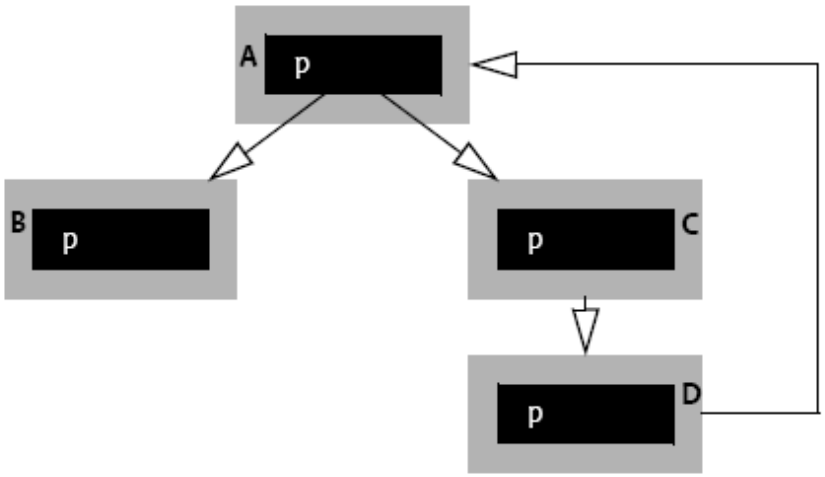
C: Require-Bundle: D

下图是对定义的描述：



每一个 bundle 都导出包 p，在这个例子中，bundle A 需要 bundle B，bundle C 需要 bundle D。当 bundle A 加载包 p 中的资源时，按照以下顺序进行查找：B，D，C，A。这是一个深度优先的搜索。如果在搜索路径中存在环，那么深度优先搜索会导致无限循环的搜索。
还是使用前面的例子，如果 bundle D 需要 bundle A，就形成了一个环。

D: Require-Bundle: A



如果 bundle A 的类加载器从包 p 中加载类和资源，而且不考虑环，那么 bundle 的查找顺序将是：B，B，B……
由于生成了这样的环，每次查找到达 bundle D 时，将重新返回到 A 进行搜索。框架必须组织这样导致无限循环查找的依赖环形成。
为了避免无限循环，框架必须在第一次访问 bundle 的时候做一个标记，在以后的搜索中，对做了标记的 bundle 将不再访问。通过这样的访问模式，上述示例的查找路径如下：B，D，C，A。

3.8.8. 启动前执行代码

在 bundle 解析之后，bundle 中导出的包就暴露给其他 bundle 了。这种状态也就是意味着其他 bundle 可以在导出包的 bundle 启动之前调用这些方法。

3.9. 本地代码加载

当一个 bundle 的类加载器试图通过调用 System.loadLibrary 来调用本地代码时，bundle 类加载器的 findLibrary 方法被调用，然后返回一个文件路径名，这个路径名在框架的可用的请求的本地库中。bundle 的类加载器必须通过测试选择的本地代码子句来试图找到本地代码库，包括 bundle 关联的类加载器和每一个附加的 fragment。fragment 的测试根据 bundle 的 ID 按照升序测试。如果在选择的本地代码子句中没有关联到搜索的库，那么返回一个 null 值，同时由父类加载器继续搜索。
为了在 OSGi 框架中加载本地代码，bundle 必须有运行权限[loadLibrary.<库名称>]，在 manifest 中的 Bundle-NativeCode 必须使用以下的语法形式进行描述：

```
Bundle-NativeCode ::= nativecode
      ( ',' nativecode )* ( ',' optional) ?
nativecode ::= path ( ';' path )*              // See 1.4.2
                    ( ';' parameter )+
optional   ::= '*'
```

当在 bundle 定位路径时，框架必须试图在启动 bundle 中进行路径的定位，启动 bundle 在其

manifest 中包含了相应本地代码子句。

定义了以下属性：

- osname—操作系统名称。这个属性的值必须是本地代码运行的操作系统平台。在环境属性一节中定义了名称的规范。
- osversion—操作系统版本。是一个版本的范围，参考版本范围一节的定义。
- processor—处理器架构。运行本地代码的处理器架构的名称，参考环境属性一节。
- language—ISO 代码定义的语言名称。这个属性的名称必须是本地代码库使用的语言。
- selection-filter—选择的过滤器。定义一个过滤器表达式，描述了本地代码子句中应该选择或者不应该选择的部分。

如下是一个典型的bundle中声明的本地代码示例：

```
Bundle-NativeCode: lib/http.dll ; lib/zlib.dll ;
      osname = Windows95 ;
      osname = Windows98 ;
      osname = WindowsNT ;
      processor = x86 ;
      selection-filter=
         "(org.osgi.framework.windowing.system=win32)";
      language = en ;
      language = se ,
    lib/solaris/libhttp.so ;
      osname = Solaris ;
      osname = SunOS ;
      processor = sparc,
    lib/linux/libhttp.so ;
      osname = Linux ;
      processor = mips;
      selection-filter
         = "(org.osgi.framework.windowing.system = gtk)"
```

如果多个本地代码需要安装在同一个平台，那么他们应该在同一个子句中进行声明。

```
// The effect of this header has probably
// not the intended effect!
Bundle-NativeCode: lib/http.DLL ;
   osname = Windows95 ;
   osversion = 3.1 ;
   osname = WindowsXP ;
   osversion = 5.1 ;
   processor = x86
```

上面的例子描述了本地代码库可以在 Windows XP，3.1 以及之后的操作系统上加载，这种描述是不正确的，单个的子句应该拆分成两个。

```
Bundle-NativeCode: lib/http.DLL ;
      osname = Windows95 ;
      osversion = 3.1;
      processor = x86,
    lib/http.DLL ;
      osname = WindowsXP ;
      osversion = 5.1;
      processor = x86
```

如果在描述子句中有一个可选的'*'号，那么即使在 Bundle-NativeCode 中没有找到匹配的子句，bundle 的安装也不会报错。
如下就是一个典型的有星号的在 bundle 的 manifest 中的本地代码声明：

```
Bundle-NativeCode: lib/win32/winxp/optimized.dll ;
      lib/win32/native.dll ;
      osname = WindowsXP ;
      processor = x86 ,
    lib/win32/native.dll ;
      osname = Windows95 ;
      osname = Windows98 ;
      osname = WindowsNT ;
      osname = Windows2000;
      processor = x86 ,
      *
```

3.9.1. 本地代码算法

在算法描述中，[X]表示框架属性 X 的值，~=表示匹配操作，匹配是大小写不敏感的。
某些属性可以使用别名。在这种情况下，在 manifest 中使用一般的名称，而在框架中应该试图通过别名来进行匹配（参考环境属性一节）。如果一个属性不是别名，或者有一个错误的值，操作者（Operator）应该将系统属性设置为一般名称或者一个有效的值，这是由于 java 的系统属性会覆盖框架构造器的这些属性。例如，如果操作系统返回的版本号码是：2.4.2-kwt，那么操作者应该设置系统属性 org.osgi.framework.os.version 的值为 2.4.2。
框架必须采用以下算法来选择本地代码子句：

1. 只能选择以下所有表达式的值为真的子句：
   - osname ~= [org.osgi.framework.os.name]
   - processor ~= [org.osgi.framework.processor]
   - osversion 包括了 [org.osgi.framework.os.version] 或者没有指定 osversion
   - language ~= [org.osgi.framework.language] 或者没有指定
   - selection-filter 使用系统属性的值或者没有指定
2. 如果在第一步中没有选择本地代码子句，算法没有中止，而是抛出一个 bundle 异常（BundleException）。

3. 选择的代码子句按照以下规则进行优先级别的排序：
   - osversion：按照操作系统版本号的降序排列，然后再是没有指定 osversion 的
   - language：按照指定了语言的排前面，没有指定的排后面
   - 在 Bundle-NativeCode 中出现的顺序，从左到右的顺序
4. 步骤 3 中选出的第一个子句用于作为本地代码子句。

无论是否指定了可选，如果在本地代码库中选择本地代码子句失败，那么 bundle 的安装失败，抛出 bundle 异常。

如果对一个选择过滤器计算，它的语法无效，那么 bundle 的安装过程失败，同时抛出一个 bundle 异常。如果没有计算出一个选择过滤器的值（有可能在一个操作系统版本或者处理器这些属性不匹配的本地代码子句中），那么这种无效的过滤器不应该导致安装的失败。在指定了可选的情况下，上述规定同样有效。

由于使用了不同的操作系统，代码库和语言，导致了设计 bundle 本地代码的头标变得非常复杂。可以采用将所有的参数用一个表格描述，而每一个目标环境都是表格的一行，这不失为一种很好的方法。如下所示：

Libraries	osname	osversion	processor	language	filter
nativecodewin32.dll, delta.dll	win32		x86	en	
nativecodegtk.so	linux		x86	en	(org.osgi.framework.windowing.systems=gtk)
nativecodeqt.so	linux		x86	en	(org.osgi.framework.windowing.system=qt)

通过上述表格，我们可以很容易就检查出遗漏的组合。然后，这个表格就和下述表格进行映射处理。

```
Bundle-NativeCode: nativecodewin32.dll;
      delta.dll;
      osname=win32;
      processor=x86;
      language=en,
   nativecodegtk.so;
      osname=linux;
      processor=x86;
      language=en;
      selection-filter=
         "(org.osgi.framework.windowing.system = gtk)",
   nativecodeqt.so;
      osname=linux;
      processor=x86;
      language=en;
      selection-filter =
         "(org.osgi.framework.windowing.system = qt)"
```

3.9.2. 考虑使用本地库

基于类加载器的本质特点，在加载本地代码时存在一些约束。为了保持类名称空间的独立性，只允许一个类加载来进行本地代码的加载，本地代码通过一个绝对的路径指定。多个类加载来加载本地代码（例如多个 bundle）会导致连接错误。
直到加载本地代码的类加载器被垃圾回收器回收，加载的本地代码才被释放。
如果卸载或者是更新一个 bundle，任何由 bundle 加载的本地代码都保留在内存中，直到 bundle 的类加载器由垃圾回收器回收才释放。而这需要所有的对象引用都已经被回收，而且所有从更新或者卸载的 bundle 中导入了包的 bundle 都已经更新完毕。这也就是说，系统类加载器加载的本地代码会一直停留在内存中，这是由于系统类加载器是绝对不会被垃圾回收器回收的。

3.10. 本地化

bundle 中包括了大量的可读性很强的有效信息。有些信息需要根据用户的语言，国籍以及其他指定的参数，或者称之为本地化（locale）的这样一些参数来变换。本节描述了 bundle 中为 manifest 和其他依赖于本地化的资源配置所提供的一种通用的变换方法。
bundle 本地化条目有一个通用的名称。为了找到潜在的本地化条目，下划线加上一个编号作为后缀，用下划线分开，最后加上一个.properties 的后缀。在 java.util.Locale 中定义了这样的后缀规格。后缀的排序必须是：

- 语言（language）
- 国籍（country）
- 变量（variant）

例如，下面的示例表示用英语，德语和瑞典语描述的 manifest：

OSGI-INF/l10n/bundle_en.properties
OSGI-INF/l10n/bundle_nl_BE.properties
OSGI-INF/l10n/bundle_nl_NL.properties
OSGI-INF/l10n/bundle_sv.properties

框架通过在基本名称后面添加后缀来查找本地化条目，即添加指定的本地化参数最后加上.properties 后缀。如果没有找到对应的变换，那么就逐步的减少参数，首先去掉变量信息，然后去掉国籍信息，最后是语言信息，直到找到了一个匹配的变换为止。例如，查找本地化的信息 en_GB_welsh 的查找过程如下：

OSGI-INF/l10n/bundle_en_GB_welsh.properties
OSGI-INF/l10n/bundle_en_GB.properties
OSGI-INF/l10n/bundle_en.properties
OSGI-INF/l10n/bundle.properties

这样允许本地化文件通过指定更详细的信息来覆盖没有指定详细信息的本地化文件。

3.10.1. 查找本地化条目

本地化条目既可以在 bundle 中，也可以在片断 fragment 中。框架必须首先查找 bundle，然后再查找 fragment。而 fragment 必须将查找委托给具有最小 bundle ID 的附主 bundle。

bundle的类加载器不能进行本地化条目的搜索。只对bundle的目录和附加的fragment进行搜索。而即使在bundle的类路径中没有逗号，也会对bundle搜索本地化条目。

3.10.2. 本地化 manifest

本地化的值保存在bundle的属性资源中，bundle的基准本地化属性文件是OSGI-INF/l10n/bundle。Bundle-Localization可能已经为本地化文件定义了一个基准的名称。这样的本地化依赖于启动bundle和bundle片断。

本地化的条目包含了本地化信息的键值对信息。bundle中所有的manifest信息都可以本地化。但是，框架必须使用一个与本地化无关包含有框架语义的的版本信息。

本地化的键可以通过以下的规则指定为bundle的manifest中的值：

header-value ::= ’%’text

text ::= < any value which is both a valid manifest header

value and a valid property key name >

例如：考虑如下的bundle条目：

Bundle-Name: %acme bundle

Bundle-Vendor: %acme corporation

Bundle-Description: %acme description

Bundle-Activator: com.acme.bundle.Activator

Acme-Defined-Header: %acme special header

用户定义的头标也可以本地化。在本地化的键中可以使用空格。

前面的例子中的条目可以通过在OSGI-INF/l10n/bundle.properties的manifest中的条目进行本地化。

```
# bundle.properties
acme bundle=The ACME Bundle
acme corporation=The ACME Corporation
acme description=The ACME Bundle provides all of the ACME \services
acme special header=user-defined Acme Data
```

也可以将上述的manifest条目本地化为法国的OSGI-INF/l10n/bundle_fr_FR.properties文件

3.11. Bundle 有效性

如果没有指定 Bundle-ManifestVersion，那么默认的 manifest 属性是 1，一些特定的 R4 语法，例如一些新的 manifest header，将被忽略而非被认为是一个错误。R3 的 bundle 必须根据 R3 的规范处理。

下面的清单（不完全包括）列举了导致 bundle 安装失败的错误：

- 不能根据 Bundle-RequireExecutionEnvironment 找到一个匹配可执行环境
- 没有 Bundle-SymbolicName.
- 重复的属性或者指令
- 对一个包的重复导入
- 导入或者导出了 java.*包
- 没有定义带有强制属性的 Export-Package
- 安装一个和已经安装好的 bundle 具有同样的符号名称和版本的 bundle

- 将一个 bundle 更新成为和已经安装好的 bundle 具有同样的符号名称和版本的 bundle
- 任何语法错误（例如，不合法的版本格式或者是符号名称，不能识别的指令等）
- 如果认为 Specification-version 是 version 的一个替代，而且指定了他们的值但是却不一样，例如：
  Import-Package p;specification-version=1;version=2
  将导致一个错误。而
  Import-Package p;specification-version=1, q;version=2
  不会导致错误。
- 在 manifest 中列举了 OSGI-INF/permission.perm 但是没有这样的一个文件
- Bundle-ManifestVersion 的值不等于 2，除非在今后的版本中有规定。

3.12. 可选项

规范提供了一系列的可选机制（optional machanism）。这些机制被设定为“可选”，是为了使得框架实现者可以选择性的让总体代码变得更小一些。所有的必选机制都必须实现，而所有的可选机制可以或多或少的实现。
下面定义了一些框架的可选部分，他们的名称是自解释的：

- org.osgi.supports.framework.requirebundle
- org.osgi.supports.framework.fragments
- org.osgi.supports.framework.extension
- org.osgi.supports.bootclasspath.extension

如果这些属性没有设置或者是他们的值不可认，那么就解析成默认值为 false。
如果框架没有实现和上述机制相关的头标，那么框架必须拒绝安装或者更新带有这些头标的 bundle。并且必须在安装和更新的时候抛出一个异常。

3.13. bundle 的需求(Requiring Bundles)

框架可以支持 bundle 之间的直接连接机制，而不管在 package 中指定的信息。本节定义了一些相关的头标，并讨论的可能的情形。最后说明了使用 Require-Bundle 会导致的一些后果（有些时候是未预见到的）。

3.13.1. Require-Bundle

在 manifest 中的 Require-Bundle 头标中定义了一个 bundle symbolic name 的列表，它们会在 import 之后，在 bundle 的类路径搜索之前进行搜索。但是，对提出 require 的 bundle，只有在被 require 的 bundle 中，标记为 exported 的包才是可见的。
Require-Bundle 必须遵循以下规范：

```
Require-Bundle ::= bundle-description
   ( ',' bundle-description )*
bundle-description ::= symbolic-name          // See 1.4.2
                       (';' parameter )*
```

在 Require-Bundle 中可以使用以下指令：

- visibility—如果值为 private （默认），那么来自被需求 bundle 的所有的可见包不会再次导出。如果值为 reexport，那么所有的这些包都会被提出需求的包进行导出，就好像这些包是位于提出需求 bundle 本地的一样。
- resolution—如果它的值为 mandatory（默认），那么被需求 bundle 必须在解析提出需求的 bundle 时存在；如果这个值为 optional，那么即使被需求 bundle 不存在，也不会影响提出需求 bundle 的解析。

在框架中还使用了以下属性：

- bundle-version—这个属性是一个版本范围，描述需求 bundle 的版本范围，参考版本范围一节。默认值为[0.0.0,∞)

某个给定包可能同时在一个或者多个被需要的 bundle 中提供，这样的情况是被明确允许的，这种包叫做拆分包（split packages）。拆分包没有在惟一的提供者中一次性提供，它的内容可以来自于不同的 bundle。如下例：

```
A:  Require-Bundle: B
    Export-Package: p
B:  Export-Package: p;partial=true;mandatory:=partial
```

如果 bundle C 导入了包 p，那么 C 将连接到包 A.p，但是，实际的内容来自于 B.p > A.p。B 导出定义中的 mandatory 属性确保了 B 并不是随机的作为包 p 的提供者。拆分包也有一些缺陷，“在 bundle 需求中的一些问题”中进行了讨论。

来自拆分包的类和资源是必须按照 Require-Bundle 中的顺序进行搜索。

例如，假设一个 bundle 由一系列的 bundle 和可选的语言资源（也是 bundle）组成：

```
Require-Bundle: com.acme.facade;visibility:=reexport,
 com.acme.bar.one;visibility:=reexport,
 com.acme.bar.two;visibility:=reexport,
 com.acme.bar._nl;visibility:=reexport;resolution:=optional,
 com.acme.bar._en;visibility:=reexport;resolution:=optional
```

bundle 可以进行同时 import 包（通过 Import-Package）和 require 一个或多个 bundle（通过 Require-Bundle），但是如果包通过 Import-Package 导入，那么它对于 Requre-Bundle 就是不可见的，这是由于 Import-Package 比 Requre-Bundle 具有更高的优先级别，而且，由 require bundle 导出和通过 Import-Package 导入的包不能被看作是拆分包。

如果要 require 一个具有名字的 bundle，提出请求的 bundle 必须有 bundle 权限（BundlePermission [<bundle symbolic name>, REQUIRE]），bundle 的符号名称即为被需要的 bundle 名称。被需求的 bundle 必须提供这样的 bundle 而且必须有这样的权限：BundlePermission[<bundle symbolic name>, PROVIDE]，其中，name 和提出请求者指定的相同。

3.13.2. bundle 需求中的一些问题

提倡的连接到bundle的方法是通过Import-Package和Export-Package来进行，这是由于通过这个途径，importer和exporter之间耦合度更小。可以将bundle进行重构，来具有不同的包结构，而不会导致其他bundle的安装失败。

Require-Bundle提供了一种无视export内容，将某个bundle的所有输出都绑定到另一个bundle的方法。尽管这种方法初看上去很方便，它也有一系列的缺陷：

- 拆分包(Split Packages)——同一个包中的类来自不同的required bundle，这样的包称之为

拆分包。拆分包有以下缺陷：

- 完整性(Completeness)——拆分包是开放式的，没有一种方法可以保证拆分包中所有的部分是否已经完全被包括了。
- 有序性(Ordering)——如果相同的类出现在多个需求的bundle中，那么对Require-Bundle来说顺序就非常重要了。排序的错误会导致难以跟踪的错误，类似于传统的类路径模型。
- 性能(Performance)——如果是拆分包，那么类搜索必须要搜索所有的提供者，这样会导致抛出ClassNotFoundException的次数上升，从而产生显著的额外开销。

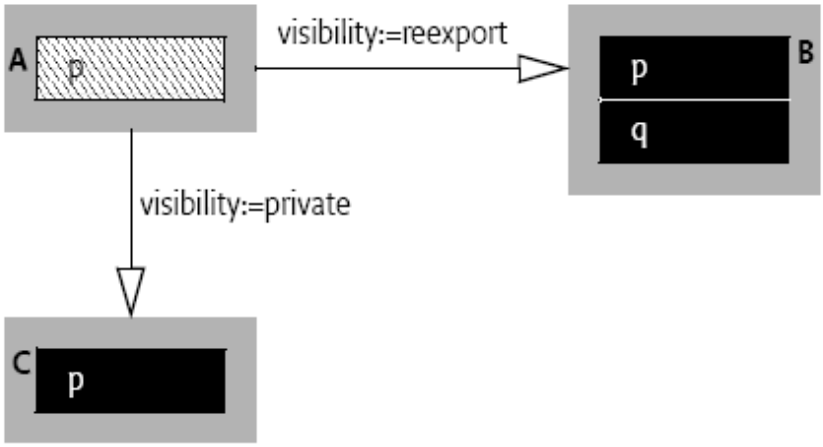
- 导出可变(Mutable Exports)——visibility:=reexport 这一特性可以导致发起请求的bundle的导出特征（export signature），受到被需求bundle的导出特征改变的影响，从而发生不可预料的改变。
- 遮蔽(Shadowing)—— 在发起请求的bundle中的类，可能被被需求的bundle中的类遮蔽，取决于被需求的bundle的导出特征和其中包含的类。（相反的是，在Import-Package中，除非指定resolution:=optional，将屏蔽整个export中的包）。
- 不可预料的声明改变（unexpected signature changes）——在Require-Bundle中的指令visibility:=private（默认值），可能会在某些特定环境下形成意外的重载。如下所示：

```
A:                 p     (private, not exposed in manifest)
   Require-Bundle: B;visibility:=reexport,
                   C;visibility:=private

B: Export-Package: p
C: Export-Package: p
```

bundle A的导出特征中只有包p，而p是一个拆分包。框架查找包p中的类时按照以下顺序：bundle B，C，最后是A。

因此， 导致”Require-Bundle C ;visibility:=private”这条指令对于包p来说是无效的。但是，如果B停止了导出包p，那么这条指令重新生效，导致在A的导出声明中失去了包p。如下图所描述：



3.14. Bundle 片断 (Fragment Bundles)

片断(fragment)指的是由框架附加(attach)在附主 bundle(Host bundle)之上的 bundle。附加的过程是解析过程的一部分：在附主 bundle 解析之前，框架将片断 bundle 的相关定义添加到附主的定义之上。Fragment 被认为是 Host 的一部分，它们不能有自己的类加载器。
片断的一个关键用途是用于提供不同区域的翻译文件。这样就可以实现翻译文件从主要应用的 bundle 中独立出来。
当更新已经 attach 的 fragment 时，原先 fragment 的内容必须仍然 attach 在附主 bundle 之上。更新的片断内容直到框架 restart，或者附主 bundle 更新之后才可以附加上来。
当附加一个 fragment 到附主 bundle 之上时，框架必须按照以下步骤运行：

1. 将与 Host 的导入定义无冲突的 fragment 的导入定义附加在 Host 的导入定义上。冲突是指：有同名的包，但是却存在不同的指令或者是属性。如果有一个这样的冲突存在，那么 fragement bundle 就不会附加在 Host 上。Fragement 可以为 Host 的私有包增加导入。在这种情况下，附主中的私有包就被遮蔽。
2. 将与 Host 的 Require- Bundle 无冲突的 fragment 的 Require-Bundle 条目附加到 Host 的 Require-Bundle 条目上。冲突仅指：如果 fragement 和 Host 的 Require-Bundle 条目中 bundle 的符号名称相同，但版本范围不同。如果有一个这样的冲突存在，那么 fragement bundle 就不会附加在 Host 上。
3. 将与 Host 的导出定义不冲突的 fragment 的导出定义附加到 Host 的导出定义上。冲突指的是：包名相同，如果存在冲突，那么忽略 fragement 中的导出定义，但继续将 fragement 附加到 host 上。

只有当 fragement 成功的附加在附主 bundle 上之后，才进入对 frament 的解析阶段。
在运行时，fragement 的 JAR 文件将在 Host bundle 类路径的搜索之后进行搜索，对此后文“运行时 Fragment”一节中有描述。

3.14.1. Fragment-Host

Fragement(片断)是一个附加到其他的称之为附主（host bundle）之上的 bundle。Fragment 的组成部分，例如 Bundle-Classpath 以及其他定义，都添加到附主的相关定义之后。片断中所有的类和资源必须使用附主的类加载器进行加载。
在 manifest 中的 Fragment-Host 的头标定义必须遵循以下语法：

```
Fragment-Host         ::= bundle-description
bundle-description    ::= symbolic-name
                          (';' parameter ) * // See 1.4.2
```

同时框架在 Fragment-Host 中还提供了以下指令：

- extension—指出了扩展的是一个系统类路径还是启动类路径的扩展。此指令只有在 Fragment-Host 是一个系统 bundle 时才有作用。在后文“扩展 bundle”一节中进行了讨论。它支持以下值：
  - framework——片断 bundle 是一个框架扩展 bundle
  - bootclasspath——片断 bundle 是一个启动类路径的扩展 bundle

fragment 必须具有“实现”所指定的系统 bundle 的 bundle 符号名称，或者是 system.bundle 这一别名。如果 bundle 的符号名称和系统 bundle 不相符，那么框架对扩展 bundle 的安装失

败。

框架在Fragment-Host中还提供了以下属性：

- bundle-version—用于选择附主 bundle 的版本范围。参考版本匹配一节，默认值为[0.0.0,∞)。

当片断 bundle 解析完毕，框架必须将片断附加在选择的具有最高版本的附主 bundle 之上。当片断附加在附主之上后，逻辑上它成为附主的一部分。片断中所有的类和资源必须使用附主的类加载器载入。一个附主的所有片断必须按照片断安装的顺序进行添加，即按照 bundle ID 的升序排列。如果在片断附加过程中发生错误，那么片断就不能附加在附主上。只有当片断附加成功之后才可以对片断进行解析。

如果一个 bundle 指定了 Fragment-Host，那么就不能指定：

- Bundle-Activator

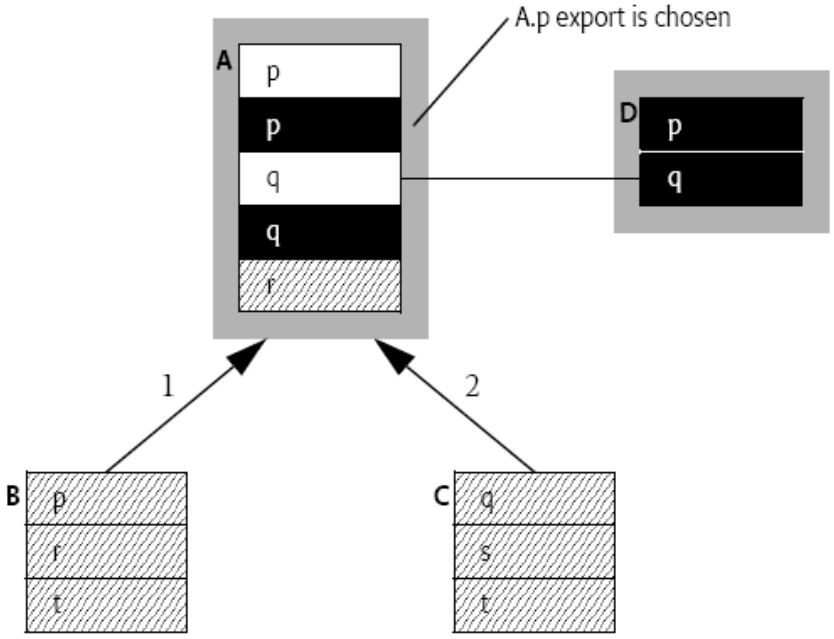
附主要允许附加片断，必须有权限 BundlePermission[<bundle symbolic name>,HOST]。而在片断中为了允许附加，片断必须要有权限 BundlePermission[<bundle symbolic name>,FRAGMENT]。

3.14.2. 运行时 Fragment (Fragment During Runtime)

Fragment 中所有的类和资源都是通过附主的类加载器来处理，而片断是不能有自己的类加载器的。片断被看作是附主的内在的一部分。

虽然片断没有自己的类加载器，当它不是扩展片断时，它还是必须有一个单独的保护域（Protection Domain）。每一个片断都可以有一个自己的关联到到片断 bundle 的位置和签名的权限。

附主 bundle 的类路径在片断类路径之前进行搜索。这也就是说包可以在附主和它的片断之间拆分。对片断的查找必须是按照 bundle ID 升序的顺序进行，这也是片断安装的顺序。



上图展示了两个片断，bundle B 在 C 之前安装，而 B 和 C 都附加在 A 之上。下面的表格描述了在初始化过程中不同包的起源。注意其中的（>）符号。

Package Requested	From	Remark
p	A.p > B.p	Bundle A exports package p, therefore, it will search its class path for p. This class path consists of the JAR and then its Fragment bundles.
q	D.q	The import does not handle split packages and package q is imported from bundle D. Therefore, C.q is not found.
r	A.r > B.r	Package r is not imported and therefore comes from the class path.
s	C.s	
t	B.t > C.t	

在上面的例子中，如果包 p 从 bundle D 中导入，那么这个表格将变得完全不同。包 p 来自 bundle D，而 bundle A 以及 bundle B 的内容将被忽略。

如果 bundle D 中的包 p 没有，那么就搜索类路径，而选择了 A.q，这是由于 A.q > C.q。

框架必须保持附加状态，只要附主是解析的。当附主变为没有解析的状态时，所有附加之上的片断必须要从附主分离。当片断变为没有解析状态时，框架必须：

- 将它从附主分离
- 重新解析附主 bundle
- 重新附加其他的片断

片断可以通过调用自己或者附主的 refreshPackages 方法或者是 resolveBundle 方法变为未解析状态。

3.15. bundle 的扩展

框架可以有选择的部分实现扩展 bundle，或者提供在启动类路径上必须的功能。这些包不能通过一般的导入导出机制来提供。

启动类路径的扩展是必须的，这是由于特定包的实现是假定它们在启动类路径上，或者是要对所有的客户端可用的。扩展启动类路径的一个例子是 java.sql 的一个实现 JSR 169。

框架扩展对于框架的实现来说是必须的。例如，一个框架的供应商可以通过框架扩展的 bundle 提供可选的一些服务如权限管理服务，启动级别服务。

扩展的 bundle 应该使用 bundle 的符号名称来实现一个系统的 bundle。或者是使用同样也是系统 bundle 的系统 bundle 替换。

下面的例子使用了 Fragment-Host 来为特定的框架实现来指定一个扩展的 bundle：

```
Fragment-Host: com.acme.impl.framework; extension:=framework
```

下面的例子使用了 Fragment-Host 来指定扩展 bundle 的启动类路径：

```
Fragment-Host: system.bundle; extension:=bootclasspath
```

下面的步骤描述了扩展 bundle 的生命周期：

1. 扩展 bundle 安装完成之后，进入安装完毕（INSTALLED）状态。
2. 通过框架的判断，扩展 bundle 可以进入到解析完毕（RESOLVED）状态。
3. 如果扩展 bundle 进行了更新，框架必须要关闭，附主虚拟机必须终止，而框架也需要重新启动。
4. 如果是一个处在解析完毕状态的扩展 bundle 进行刷新，框架必须要关闭，附主虚拟机必须终止，而框架也要重新启动。
5. 如果对一个处在解析完毕状态的扩展 bundle 进行更新或者是卸载（UNINSTALLED），

那么就不能再次进入到解析完毕状态。如果对扩展 bundle 进行刷新，必须关闭框架，附主虚拟机终止，然后重新启动框架。

3.15.1. 扩展 bundle 的非法 manifest

如果在bundle中指定了以下的头标，那么在扩展bundle安装或者是更新的时候，必将抛出bundle异常（BundleException）:

- Import-Package
- Require-Bundle
- Bundle-NativeCode
- DynamicImport-Package
- Bundle-Activator

规范允许同时在启动类路径和框架扩展bundle中指定Export-Package。当对扩展bundle解析时，任何由框架扩展bundle指定的导出的包必须由系统bundle来导出。

3.15.2. 类路径处理

bundle 的 JAR 文件的扩展启动类路径必须添加到附主的虚拟机的启动类路径上。框架扩展的 bundle 的 JAR 文件附加在框架的类路径上。
扩展 bundle 的类路径的附加顺序按照它们的安装顺序来进行，即按照 bundle ID 的增序排列。框架如果配置自己以及启动类路径如果附加扩展 bundle 是由具体的实现来完成。在一些执行环境下，也许并不支持扩展 bundle。在这样的环境下，如果安装扩展 bundle 将抛出 bundle 异常。产生的 bundle 异常有一个原因标志为 UnsupportedOperationException

3.16. 安全

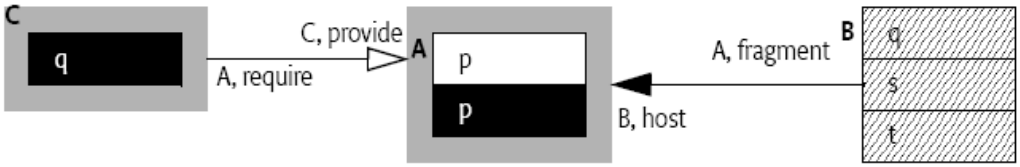
3.16.1. 扩展 bundle

在 Java 2 安全机制下的框架环境下允许安装扩展 bundle 之前还需要进行额外的安全检查。为了保证扩展 bundle 的成功安装，框架必须检查扩展 bundle 是否拥有分配给它的所有权限（All Permissions）。这也就是说扩展 bundle 的权限必须要在它安装之前设立。
必须授予扩展 bundle 的 AllPermission 权限，这是由于扩展 bundle 需要在启动类路径或者是框架实现的保护域（Protection Domain）之下加载。这两个保护域都有 AllPermission 权限。这样，如果一个扩展 bundle 没有 AllPermission 权限，那么是不允许安装的。扩展 bundle 的安装者必须要有管理权限（AdminPermission[<extension bundle>,EXTENSIONLIFECYCLE]）来安装扩展 bundle。

3.16.2. bundle 权限

大部分的包在包权限管理之上共享权限，而片断和需求的 bundle 通过 bundle 的符号名称来进行共享处理。bundle 权限就用于这种类型的包共享。bundle 权限是可选的，但是如果框架

支持 Require-Bundle，那么也必须要支持 bundle 的权限。
bundle 权限的名称参数使用了 bundle 的符号名称。符号名称用于鉴别目标 bundle（target bundle）。可以在名称后面使用通配符（'.*'）。
例如，如果片断 A 附加到 bundle B 之上，那么 A 需要 BundlePermission("B", "fragment")权限，这样 A 才允许附主 bundle。这个指令的动作描述如下：



定义了如下动作：

- provide – 对目标 bundle 提供包权限。
- require – 从目标 bundle 中获取包权限。
- host – 附加目标片断的权限。
- fragment – 作为片断附加到附主上的权限。

3.16.3. 包权限

bundle 只能导入导出有获取权限的包。包权限（PackagePermission）必须要用于一个包的所有的版本。
包权限由两个参数：

- 可能导入或者导出的包，可以使用通配符。权限的粒度是针对包而不是包中的类。
- 动作，导入（IMPORT）或者导出（EXPORT）。如果 bundle 有导出一个包的权限，那么框架必须自动授予导入这个包的权限。

如果一个包权限的参数为通配符*和 EXPORT，那么就允许导入导出任何包。

3.16.4. 资源权限

为了访问以下资源，框架必须授予bundle RESOURCE，METADATA和CLASS 管理权限（AdminPermission）：

- 自身
- 任何附加的片断
- 导入包中的任何资源

bundle中的资源也可以通过使用bundle中特定的方法来访问。这些方法的访问者必须要拥有管理权限AdminPermission[bundle,RESOURCE]。
如果访问者没有必需的权限，那么就不能对资源进行访问，并且返回一个空值。否则返回一个资源的URL对象。这些URL对象称之为bundle资源链接（*bundle resource URLs*）。一旦返回了URL对象，当对资源进行访问时就不再进行权限检查。URL对象必须使用框架实现中定义的配置（scheme）。
bundle链接通常由框架来创建，但是，在一些情况下，bundle需要通过链接来查找到相关资源，例如，对于和给定资源处于同一个目录下的资源，可以创建它的资源链接。
对于不是由框架创建的资源链接，由于java.net.URL类的设计的原因，它的安全策略略有不

同。 并不是所有的URL类构造方法会和URL流处理类（URL Stream Handler）进行交互（实现规范部分）。其它的构造方法至少会调用URL流处理类中的parseURL方法，而在URL流处理类中会进行安全检查。这种设计使得框架不可能在构建bundle链接时进行权限检查。
下面的构建方法在构建bundle资源链接时使用parseURL方法，并进行检查：

URL(String spec)
URL(URL context, String spec)
URL(URL context, String spec, URLStreamHandler handler)

当使用上述构造方法进行bundle资源链接的创建时，框架的实现必须要在parseURL方法中对调用者进行必要的权限检查。如果没有必需的权限，parseURL方法抛出安全异常（Security Exception）。这将导致URL的构造方法抛出一个非正常的URL异常。如果调用者有必需的权限，那么就创建对资源访问的链接对象，而且以后再也不需要进行检查了。
在下面的构造方法中没有调用parseURL方法，这样在创建URL类期间就不可能进行权限的检查了：

URL(String protocol, String host, int port, String file)
URL(String protocol, String host, int port, String file,
URLStreamHandler handler)
URL(String protocol, String host, String file)

这些构造方法没有进行权限检查就创建了bundle的资源链接，因此，对权限的检查就推迟到调用时进行。访问这样创建的URL对象时，如果访问者没有权限（AdminPermission[bundle, RESOURCE]）那么框架就会抛出一个安全异常。

3.16.5. 权限检查

由于多个 bundle 会导出具有相同类名的权限类，框架必须要保证权限检查时使用了正确的类。例如，调用 checkPermission 方法提供了权限类的一个接口：

```
void foo(String name) {
   checkPermission(new FooPermission(name,"foo"));
}
```

权限接口类来自于独特的途径，权限只能用相同来源的实例来测试。
因此，框架需要基于类而不是类名来查找权限。如果需要实例化一个权限，那么必须要使用检查为实例的权限类。这对框架的实现来说很复杂，但是不会影响到 bundle 的开发人员。
考虑如下示例：

Bundle A
    Import-Package: p
    Export-Package: q
Bundle B
    Import-Package: p

- bundle A 使用 p.FooService，使用这个类来检查 q.FooPermission 当调用它的任何方法时。
- bundle B 在其保护域中的权限信息对象中有一个 FooPermission。
- bundle B 调用 bundle A 中 FooService 中的方法。
- FooService 通过使用一个新的 FooPermission 接口来调用 checkPermission 方法。
- 在调用 FooPermission 中的方法之前，框架必须使用来自同一个类加载器中的 FooPermission 对象作为给定的 FooPermission 对象。在这种情况下，FooPermission 类来自包 A.q。

权限检查完毕之后，bundle B 就有了一个 FooPermission 实例，这个实例使用了没有导入的包中类的。因此，框架需要实例化多种不同 FooPermission 类来满足不同 bundle 的需求。

3.17. 参考

[19] The Standard for the Format of ARPA Internet Text Messages
STD 11, RFC 822, UDEL, August 1982
http://www.ietf.org/rfc/rfc822.txt

[20] The Hypertext Transfer Protocol - HTTP/1.1
RFC 2068 DEC, MIT/LCS, UC Irvine, January 1997
http://www.ietf.org/rfc/rfc2068.txt

[21] The Java 2 Platform API Specification
Standard Edition, Version 1.3, Sun Microsystems
http://java.sun.com/j2se/1.4

[22] The Java Language Specification
Second Edition, Sun Microsystems, 2000
http://java.sun.com/docs/books/jls/index.html

[23] A String Representation of LDAP Search Filters
RFC 1960, UMich, 1996
http://www.ietf.org/rfc/rfc1960.txt

[24] The Java Security Architecture for JDK 1.2
Version 1.0, Sun Microsystems, October 1998

[25] The Java 2 Package Versioning Specification
http://java.sun.com/j2se/1.4/docs/guide/versioning/index.html

[26] Codes for the Representation of Names of Languages
ISO 639, International Standards Organization
http://lcweb.loc.gov/standards/iso639-2/langhome.html

[27] Zip File Format
The Zip file format as defined by the java.util.zip package.

[28] Manifest Format
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR%20Manifest

[29] W3C EBNF
http://www.w3c.org/TR/REC-xml#sec-notation

[30] Lexical Structure Java Language
http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html

[31] Mathematical Convention for Interval Notation
http://planetmath.org/encyclopedia/Interval.html

[32] Uniform Resource Identifiers URI: Generic Syntax
RFC 2396
http://www.ietf.org/rfc/rfc2396.txt

[33] Codes for the Representation of Names of Languages
ISO 639, International Standards Organization
http://lcweb.loc.gov/standards/iso639-2/langhome.html

4. 生命周期层

版本号： 1.3

4.1. 简介

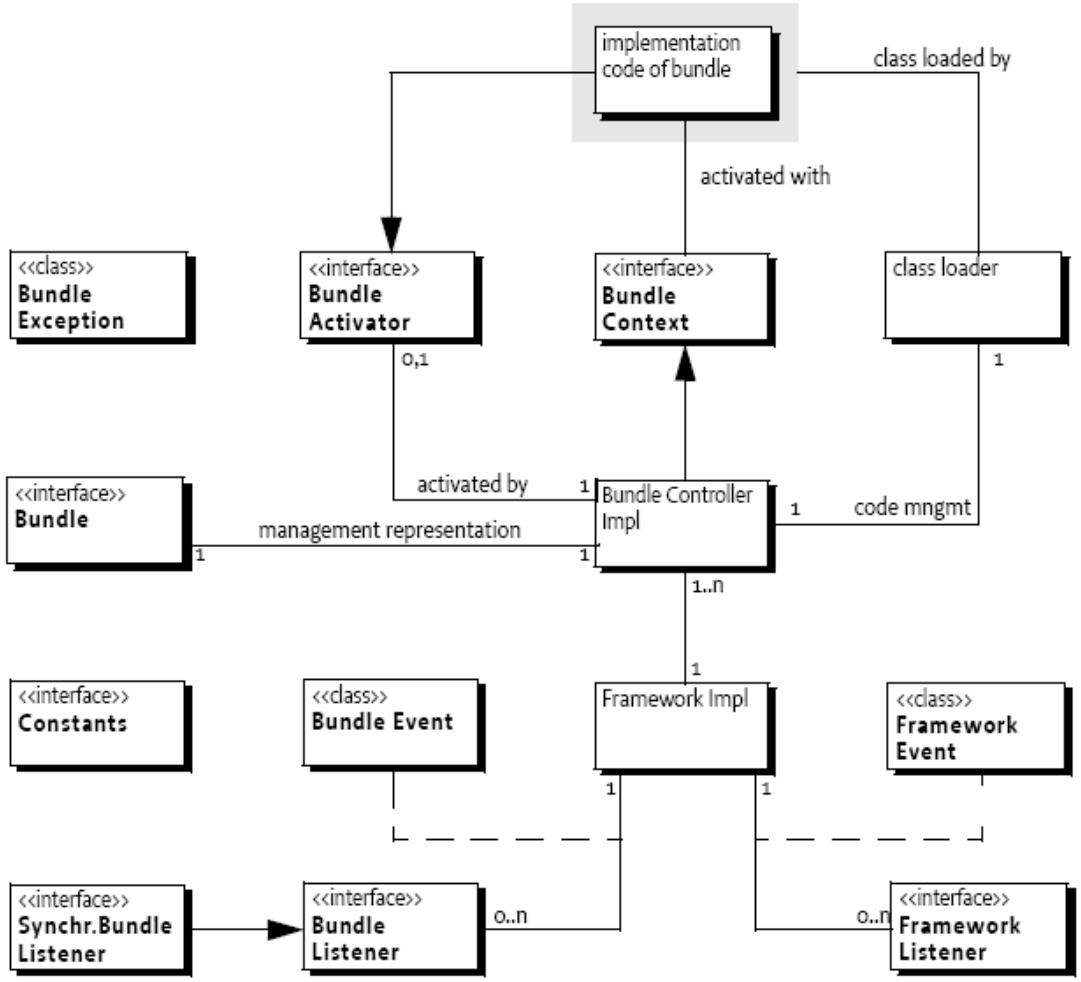
生命周期层提供了 bundle 的生命周期管理和安全控制的 API。本层是建立在在模型和安全层之上。

4.1.1. 要点

- 完整性—生命周期层必须提供包括 bundle 安装、启动、停止、更新、卸载和管理的所有状态的 API。
- 深入性—API 必须要提供深入到框架实际状态的视点。
- 安全—通过使用小粒度的权限来实现的安全环境下，API 必须可以在这样的环境下使用。但是安全必须要是可选的。
- 可管理性—必须可以对远程平台的服务进行管理。

4.1.2. 名词

- Bundle — 框架中的安装完毕的 bundle。
- bundle 上下文（Bundle Context） — bundle 在框架中的执行上下文环境。当启动或者停止一个 bundle 的时候，框架将它发送到一个 bundle 的激活器（Bundle Activator）。
- bundle 激活器（Bundle Activator） — 用于启动和停止 bundle 的，在 bundle 中实现的一个接口。
- bundle 事件（Bundle Event） — 在 bundle 内用于标志一个生命周期操作的事件。通过 bundle 的监听器来接收（同步）这样的事件。
- 框架事件（Framework Event） — 标志错误或者框架状态改变的事件。通过框架的监听器来接收框架事件。
- bundle 监听器（Bundle Listener） — bundle 事件的监听器。
- 同步 bundle 监听器（Synchronous Bundle Listener） — 同步传送 bundle 事件的监听器。
- 框架监听器（Framework Listener） — 框架事件的监听器。
- bundle 异常（Bundle Exception） — 当框架错误错误时抛出的异常。
- 系统 bundle（System Bundle） — 框架声明的 bundle。



4.2. Bundle

在 OSGi 框架中，bundle 表现为 JAR 文件的形式。在模块层中说明了类加载这方面的含义。但是，模块层没有定义 bundle 是如何安装、更新和卸载的，而生命周期操作对这些进行了定义。

对 bundle 的安装只能够是另一个 bundle 来执行或者是由规范实现的方法（例如作为框架实现的一个命令行参数）。

通过 bundle 激活器来启动一个 bundle。bundle 激活器是根据 manifest 中的 Bundle-Activator 来确定。指定的类必须要实现 BundleActivator 接口。这个接口有一个 start 和 stop 方法，用于 bundle 的编程人员来注册自己的监听器，并启动任何必需的线程。stop 方法中必须要清除和停止任何执行的线程。

在 bundle 激活中，激活器接收 bundle 的上下文环境。bundle 上下文环境接口的方法可以粗略的分为以下三类：

- 信息 — 访问框架中的其他信息
- 生命周期 — 安装其他 bundle 的可能性
- 服务注册 — 在服务层详细讨论了服务注册

4.3. Bundle 实体

对于 OSGi 框架中安装的每一个 bundle，都有一个关联的 bundle 对象。这个对象就用于 bundle 的生命周期管理。通常是由一个管理代理（Agent，也是一个 bundle）来完成。

4.3.1. Bundle 标识

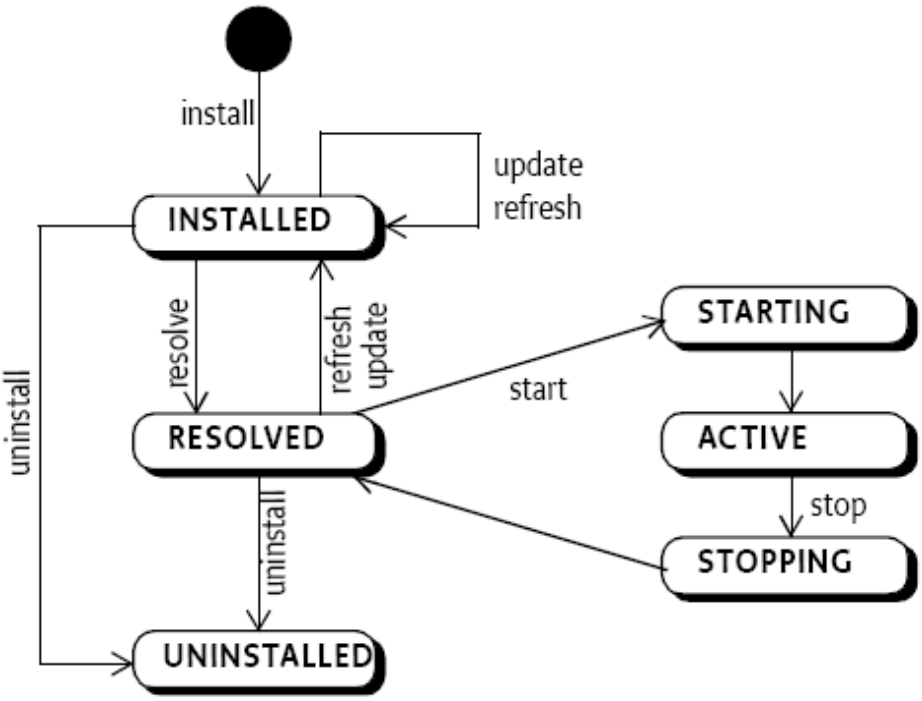
bundle 是通过一系列的名称来区分的：

- bundle 标志符（*Bundle identifier*）— 由框架分配的一个长整形数字，用于在整个生命周期期间惟一标识一个 bundle，即使 bundle 更新或者框架重启。它的目的是用于在框架中区分 bundle。bundle 标志符在安装的时候按照升序来分配。可以通过 getBundle()方法来获取一个 bundle 的标志符。
- bundle 位置（*Bundle location*）— 由管理代理（Agent，操作者）在安装过程中分配给一个 bundle 的名称。这个字符串通常解释为一个 JAR 文件 URL，但是这并不是强制性的。在一个特定的框架中，位置是惟一的，位置字符串可以惟一标志一个 bundle，而且即使 bundle 更新也不能改变这个位置字符串。可以通过 getLocation()方法来获取一个 bundle 的位置。
- bundle 符号名称（*Bundle Symbolic Name*）— 由开发者分配的名称。通过 bundle 的版本和符号名称就可以在全局惟一定位一个 bundle。可以通过 getSymbolicName()获取一个分配给 bundle 的名称。

4.3.2. Bundle 状态

bundle 可以处于以下状态中的一种：

- INSTALLED — 成功安装 bundle
- RESOLVED — 所有 bundle 需要的 Java 类都准备好了。这个状态标志着 bundle 已经是启动就绪或者是已经停止。
- STARTING — 正在启动 bundle。调用了 bundle 激活器的 start 方法，而且还没有从方法中返回。
- ACTIVE — bundle 已经启动完毕，正在运行中。
- STOPPING — 正在停止 bundle。调用了 bundle 激活器的 stop 方法，而且还没有从方法中返回。
- UNINSTALLED — bundle 已经卸载完毕，不能进入其他状态。



bundle 状态图

当一个 bundle 安装完毕，它一直存储在框架的固定存储区域，直到确定被卸载。bundle 是否已经启动或者停止的信息必须记录到框架的固定存储区域。如果一个 bundle 在固定存储中记录为开始的，那么无论何时框架启动，bundle 也要启动，直到明确的停止 bundle 为止。启动级别服务影响了实际的启动和停止 bundle 的过程。参考启动级别服务规范。

bundle 接口定义了一个 getState()方法来返回 bundle 的状态。

如果本规范采用了术语 active 来描述一个状态，那么它包括了 STARTING 和 STOPPING 两种状态。

bundle 的状态采用位掩码（bit-mask）来描述，尽管一个 bundle 只能处于一种状态。下面的代码示例可以用于检测 bundle 是否处于 STARTING、ACTIVE 或者是 STOPPING 状态。

```
if ((b.getState() & (STARTING | ACTIVE | STOPPING) != 0)
doActive()
```

4.3.3. Bundle 安装

在 BundleContext 接口是用于提供给 bundle 激活器的，为了安装 bundle，在这个接口中定义了以下方法：

- installBundle(String) — 根据指定的位置字符串（通常应该是一个 URL）来安装一个 bundle。
- installBundle(String,InputStream) — 从指定的输入流中安装一个 bundle。

在安装一个 bundle 之前，应该对它进行有效性校验，否则会导致安装失败。在 bundle 有效性校验一节中对校验进行了描述。

每一个 bundle 都可以通过为止字符串来惟一标识。如果根据一个已经指定的位置来安装，那么方法 installBundle 返回这个已经安装的 bundle 而且不会在安装新的 bundle。

框架必须要分配给新安装的 bundle 一个比原来任何 bundle 标志符要大的一个惟一的 bundle 标志符。

框架中对 bundle 的安装必须要满足：

- 持久性—bundle 必须要在整个框架和 java 虚拟机中保持直到被明确卸载。
- 原子性—install 方法必须是原子的，也就是说要么 bundle 安装完成，要么安装失败。在方法调用前和调用后，OSGi 框架必须要是同一种状态。

一旦 bundle 安装完成，则创建了一个 bundle 对象，其他所有的对 bundle 的生命周期操作必要要由这个对象来执行。返回的这个 bundle 对象可以用于启动、停止、更新和卸载 bundle。

4.3.4. Bundle 解析

当框架根据 manifest 中的描述成功解析了一个 bundle 中的依赖关系时，bundle 进入了 RESOLVED 状态。在解析过程一节中描述了详细的依赖关系。

4.3.5. Bundle 启动

为了启动一个 bundle，在 Bunlde 这个接口中定义了 start()方法。如果成功调用了这个方法，那么 bundle 的状态就被设置为 ACTIVE，而且一直保持这个状态直到停止 bundle。可选的启动级别服务会影响到启动和停止 bundle 的实际顺序。参阅启动级别服务规范一节。

为了启动一个 bundle，首先需要对它进行解析。当试图启动一个 bundle 时，如果 bundle 还没有被解析，框架会尝试对 bundle 进行解析。如果解析失败，那么 start 方法必须要抛出一个 bundle 异常（BundleException）。在这种情况下，标记 bundle 为启动（started），那么当 bundle 变为已解析之后，它的启动级别就必须自动启动 bundle，即使重新启动了框架。

如果框架已经解析完毕，bundle 必须要通过调用 BundleActivator 对象进行激活。在接口 BundleActivator 中定义了框架启动和停止 bundle 需要调用的方法。

为了将 bundle 的激活器类信息发送给 OSGi 环境，bundle 开发人员必须要在 bundle 的 manifest 文件中的 Bundle-Activator 部分进行声明。而框架必须要实例化这样的一个类，而且声明为 BundleActivator 实例。然后调用它的 start 方法来启动这个 bundle。

如下就是一个 Bundle-Activator 声明的例子：

```
Bundle-Activator: com.acme.Activator
```

作为 bundle 激活器的类比需要实现 BundleActivator 接口，并且声明为 public 类型，同时还需要有一个默认的构造方法，这样就可以通过调用 Class.newInstance 来创建一个新的实例。bundle 激活器的提供是可选的。例如，如果是一个导出一系列包的库文件 bundle，那么就没有必要来定义一个 bundle 激活器了。另外，还可以通过其他的机制来对取得 bundle 的上下文的控制和信息的获取，例如，服务组件运行时（Service Component Runtime）。

BundleActivator 接口定义了启动和停止一个 bundle 的方法：

- start(BundleContext) — 可以通过这个方法来申请启动 bundle 所需的资源，启动线程，注册服务等等。如果这个方法没有注册任何服务，那么 bundle 就会在随后注册需要的服务。例如，只要 bundle 是在 ACTIVE 状态下，就可以通过在回调处理中，或者是一个扩展的事件来注册。
- stop(BundleContext) — 这个方法可以看作是 start 方法的一个逆过程。但是，对服务和框架事件的取消注册并不是必需的，因为这些可以由框架来清除。

当 bundle 启动时，必须要创建一个 bundle 激活器，也就是说创建一个类加载器。在一个更大的系统中，这种贪心策略使得启动事件显著增加，而且内存消耗也存在没必要的增长。可以通过服务组件运行时这种机制来减轻这样的问题。

4.3.6. Bundle 停止

在 Bundle 接口中定义了 stop 方法来停止一个 bundle。通过调用这个方法来停止一个 bundle 并将 bundle 的状态设置为 RESOLVED。
在 BundleActivator 中定义了方法 stop(BundleContext)，这个方法是供由框架调用来停止一个 bundle。在这个方法中，必须要释放 bundle 从激活开始所申请的所有资源。必须马上中止 bundle 相关的所有线程。当 stop 方法返回之后，线程代码中不能再使用框架相关的类（例如服务和 BundleContext 类）。
如果在 bundle 生命周期期间注册了任何服务，那么当停止 bundle 之后，框架必须自动的取消注册所有的这些服务。但是在 stop 方法中注册的服务例外，可以不必取消注册。
框架必须要确保方法 BundleActivator.start 的成功执行，当 bundle 不是活动时，必须调用这个 bundle 的 BundleActivator 对象中的 stop 方法，之后，这个专用的 BundleActivator 必须不能再被使用。
其他 bundle 还可以使用处于停止状态的 bundle 导出的包，这种持续导出意味着其他 bundle 可以执行停止状态的 bundle 中的代码，这就需要 bundle 的设计者来保证这样做没有危害。而只导出接口可以防止这种代码的执行。通常，为了保证不被执行，导出的接口中不应该包含有可执行的代码。

4.3.7. Bundle 更新

在 Bundle 接口中定义了以下两种方法来更新 bundle：

- update() – 更新 bundle
- update(InputStream) – 通过一个指定的输入流来更新 bundle

更新处理支持从一个版本的 bundle 移植到这个 bundle 的另一个更新的版本。
一个更新了的 bundle 必须将它导出的包直接提供给系统。同时，对于已经存在和以后安装的 bundle 来说，原来旧版本导出的包也是可用的，直到调用了包的 refreshPackages 方法或者框架重新启动之后。
bundle 的更新者对安装的 bundle 以及新版本的 bundle 必须有管理权限：AdminPermission[<bundle>,LIFECYCLE]，在权限管理一节中对 AdminPermission 进行了解释。

4.3.8. Bundle 卸载

在 Bundle 接口中定义了 uninstall 方法来从框架中卸载 bundle。这个方法的调用使得框架将 bundle 卸载的信息通知给其他 bundle，并设置 bundle 的状态为 UNINSTALLED。为了将来的扩展，框架必须要清除与 bundle 相关的任何资源。这个方法必须要将 bundle 从框架的固定存储区卸载。
一旦从 uninstall 方法中返回，OSGi 服务平台的状态必须要和 bundle 安装之前一样，除非以下原因：

- 卸载的 bundle 导出了包（通过它的 Export-Package 头标）
- 卸载的 bundle 被框架选作是包的 exporter

如果 bundle 导出了在其他 bundle 中使用的包，框架必须要保留这些包对 importer 可用，直

到满足以下任意一个条件：

- 调用了方法 org.osgi.service.packageadmin.PackageAdmin.refreshPackages
- 框架重新启动

卸载 bundle 的包必须是不能用于最新安装的 bundle，但是对于原来的 importer 来说还是可用的，直到调用了 refreshPackages 方法或者是框架的重新启动。

4.3.9. Bundle 更改检测

在 Bundle 类中提供了检测 bundle 更改的便捷方法。在框架中，必须记录对 bundle 使用生命周期操作而使得 bundle 发生改变的时间。使用 getLastModified()方法可以返回 bundle 最后的安装、更新和卸载时间。这种最后更改时间必须要持久存储。

这个方法返回一系列的自从 1970 年 1 月 1 日 0 点开始的毫秒级时间，而最近的一次更新时间值一定要比上次更新时间的值要大。

如果一个 bundle 需要从另一个 bundle 中更新资源，而当另一个 bundle 发生改变之后，这个 bundle 需要刷新缓冲，这时 getLastModified()方法就非常有用。当需要刷新的 bundle 不是活动状态，则可以改变目标 bundle 的生命状态。因此最后更改时间是记录是跟踪目标 bundle 的一种便利方法。

4.3.10. 重新取得 Manifest

在 Bundle 接口中定义了两个方法来取得 manifest 信息：

- getHeaders() – 返回一个包括了 manifest 头标以及对应值的键值对的 Dictionary 对象。返回值根据 java.util.Locale.getDefault 获得的默认区域信息来本地化。.
- getHeaders(String) –返回一个包括了 manifest 头标以及对应值的键值对的 Dictionary 对象。通过指定的参数来进行本地化。locale 参数的值可以是以下值：
  - null – 如果为空，则使用 java.util.Locale.getDefault 返回的 locale 信息。这种情况等同于 getHeaders()方法。
  - 空字符串 – 返回的 Dictionary 信息包含了没有本地化的 manifest 信息，包括任何以‘%’开始的头标。
  - 一个指定的 Locale 信息 – 通过这个指定的 Locale 信息来本地化 manifest

本地化的过程根据前面的本地化一节的描述来进行。如果某一个键没有找到变换，那么 Bundle 的 getHeaders 方法返回的 Dictionary 中将返回一个在 manifest 中指定的不是以‘%’开始的字符。

这些方法需要 AdminPermission[<bundle>, METADATA]权限，这是由于有些头标信息也许比较敏感，例如在 Export-Package 中列出来的包。通常 bundle 都有对自己头标的读权限。

对处于 UNINSTALLED 状态的 bundle，getHeaders 必须继续提供它的 manifest 信息。在 bundle 卸载之后，getHeaders 方法只能返回一个原始的 manifest 信息，或者是在 bundle 卸载时通过默认的 locale 信息本地化的 manifest。

处理 manifest 时，框架实现必须使用原始的 manifest 信息。而本地化不能影响到框架的操作。

4.3.11. 类加载

在特定情况下，类的加载要像是从 bundle 内部进行的加载。通过 loadClass(String)方法来访问 bundle 的类加载器。这个方法可用于：

- 从其它 bundle 中加载插件
- 启动一个应用模型激活器
- 和遗留代码的交互

例如，应用模型可以使用这个特性来从 bundle 中加载初始化类，并且通过应用模型的规则来启动它。

```
void appStart() {
   Class initializer = bundle.loadClass(activator);
   if ( initializer != null ) {
      App app = (App) initializer.newInstance();
      app.activate();
   }
}
```

4.3.12. 资源访问

bundle 中的资源可以来自于不同的途径。可以是来自于原始的 JAR 文件，bundle 片断，导入包，或者是 bundle 类路径。不同的情况下的查找策略是不一样的。在 Bunlde 接口中提供了一系列使用不同策略的方法来访问这些资源。支持如下的查找策略：

- 类空间—方法 getResource(String) 和 getResources(String)提供了对类空间组成的资源的访问，在全局搜索顺序一节中描述了这种类空间。按照这个查找顺序，JAR 中的某些类变得不可访问了。调用这两个方法要求 bundle 必须已经解析完毕，如果 bundle 还没有解析，那么框架必须尝试进行解析。

  同时，这种查找顺序也会隐藏 JAR 中的一些目录。比如考虑到拆分包的情况，也就说，包名相同的资源来自于不同的 JAR 文件。如果 bundle 是未解析的（或者是不可解析的），方法 getResource 和 getResources 必须只能从 bundle 类路径中加载资源。这种查找策略可用于对自己资源的查找。对这两个方法的访问都会导致类加载器的创建和 bundle 的解析。
- JAR 文件—方法 getEntry(String) 和 getEntryPaths(String)提供了对 bundle 的 JAR 文件资源的访问，只考虑原始的 JAR 文件的情况。这两个方法的目的在于提供一个低层次的资源访问而无需解析 bundle。
- bundle 空间—方法 findEntries(String,String,boolean)是一种中间媒介。当从另一个 bundle 中配置或者初始化信息时非常有用。在这个方法中，考虑到了片断 bundle 的情况，但是不会创建一个类加载器或者对 bundle 进行解析。这个方法提供了对相关 JAR 文件的所有目录的访问。

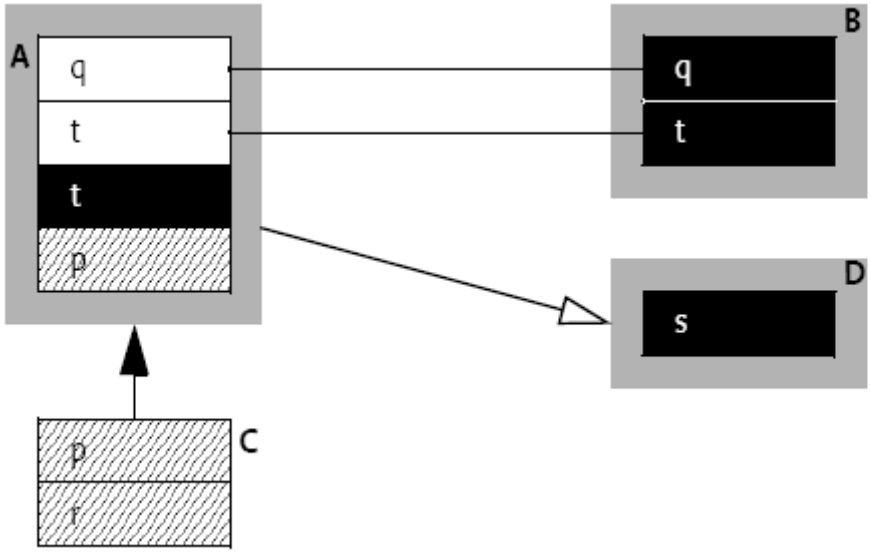
例如，考虑如下设置：

```
A:  Require-Bundle: D
    Import-Package: q,t
    Export-Package: t
B:  Export-Package: q,t
```

C: Fragment-Host: A

D: Export-Package: s

如下图所示：



下面的表格描述了解析 bundle A 的时候资源获取的情况：

Resource	getResource	getEntry	findEntries
q	B.q	null	null
p	A.p > C.p	A.p	A.p > C.p
r	C.r	null	C.r
s	D.s	null	null
t	B.t	A.t	A.t

下面的表格则描述了同样情况下的 bundle A 未解析的情况：

Resource	getResource	getEntry	findEntries
q	null	null	null
p	A.p	A.p	A.p
r	null	null	null
s	null	null	null
t	A.t	A.t	A.t

4.3.13. Bundle 权限

在 Bundle 接口中定义了一个方法来返回和 bundle 权限相关的信息：hasPermission(Object)。如果 bundle 的保护域中有指定的权限则返回 true，否则返回 false，或者如果参数指定的对象没有实现 java.security.Permission 接口也会返回 false。

参数的类型为 Object，这样，在不支持 java 2 安全的平台上也可以实现框架。

4.4. Bundle 上下文环境

通过使用 BundleContext 对象来实现框架和安装的 bundle 之间的关系。一个 BundleContext 对象描述了 OSGi 服务平台中一个 bundle 的执行上下文环境，作为框架之下的一个代理。
当启动一个 bundle 时，框架就创建这个 bundle 的 BundleContext 对象，bundle 使用这个私有的 BundleContext 对象用于以下目的：

- 在 OSGi 环境下安装一个新的 bundle。参阅安装 bundle 一节。
- 监视安装在 OSGi 环境下的其他 bundle。参阅获取 bundle 信息一节。
- 获取一个持久存储区。参阅持久存储一节。
- 重新获取注册服务的服务类。参阅服务参考一节。
- 在框架服务中注册一个服务。参阅注册服务一节。
- 事件注册到框架中或者取消注册。参阅监听器一节。

当启动一个 bundle 时，框架就创建一个 BundleContext 对象，并将这个对象作为参数提供给 bundle 激活器的 start 方法，即调用 start(BundleContext)。每一个 bundle 都有一个自己的 BundleContext 对象，而不能在 bundle 之间传递这些对象，这是由于 BundleContext 关系到 bundle 的安全和资源分配这些方面。
方法 stop(BundleContext)返回之后，就不能再使用 BundleContext 对象。而如果在 bunle 停止之后继续使用 BundleContext 对象，则框架必须抛出异常。

4.4.1. 获取 bundle 信息

接口 BundleContext 中定义了获取 OSGi 服务平台下安装的 bundle 信息的方法：

- getBundle() – 返回和 BundleContext 相关的惟一的一个 Bundle 对象。
- getBundle() – 返回一个数组，描述框架中安装的 bundle。
- getBundle(long) – 返回指定 bundle 标志符的 Bundle 对象，如果没有找到匹配的 Bundle 对象，则返回空值。

对 Bundle 的访问是不受限制的，任何 bundle 都可以遍历已经安装的 bundle 的列表。而可以标识一个 bundle 的信息（例如区域信息，或者是它的 manifest 信息等）则需要调用者有管理权限：AdminPermission[<bundle>,METADATA]。

4.4.2. 持久存储

框架应该对每一个在平台下安装的 bundle 提供一个私有的持久存储区，而且应该支持一些常见文件系统。
BundleContext 接口中通过 File 类来定义这个持久存储区，在 File 类中提供了与平台无关的文件和目录的定义。
BundleContext 接口中还定义了一个方法来访问私有的持久存储区：getDataFile(String)方法。这个方法使用一个相关的文件名作为参数，在方法中，则将这个相对文件名转换成 bundle 持久存储的绝对文件路径。如果不支持这样的持久存储区，则方法返回空值 null。
框架必须自动提供给 bundle 以下权限：FilePermission[<storage area>, READ | WRITE | DELETE]，拥有这样的权限，bundle 就可以读、写、删除存储区中的文件。
如果需要执行权限（EXECUTE），那么可以在文件权限定义中使用相对路径。例如，使用

FilePermission[bin/*,EXECUTE]来指定 bin 目录下所有文件具有可执行权限。这种方法只是提供了在 Java 环境下的执行权限，而不会处理基于操作系统层次上的可执行性分析。
这种特殊的处理只是用于处理分配给 bundle 的 FilePermission 对象。默认的权限并不是这样处理的。而必须忽略通过使用相对路径的 setDefaultPermission 方法分配的 FilePermission 对象。

4.4.3. 环境属性

在 BundleContext 中定义了一个方法来返回与框架相关的信息：getProperty(String)方法。可以用这个方法来返回如下框架信息：

<table>
<tr><td>属性名称</td><td colspan="3">描述</td></tr>
<tr><td>org.osgi.framework.version</td><td colspan="3">框架规范版本，必须是 1.3</td></tr>
<tr><td>org.osgi.framework.vendor</td><td colspan="3">框架实现厂商</td></tr>
<tr><td>org.osgi.framework.language</td><td colspan="3">框架使用的语言。ISO 639 规定</td></tr>
<tr><td>org.osgi.framework. « executionenvironment</td><td colspan="3">逗号分隔的执行环境描述，在每一个执行环境中的所有方法必须要在平台上实现。平台实现必须要提供在执行环境下的定义的所有特征。因此，框架实现必须要实现在这个环境属性中定义的所有环境定义特征。</td></tr>
<tr><td rowspan="11">org.osgi.framework.processor</td><td colspan="3">处理器名称。下表列出了一系列的处理器名称，新的处理器名称在 OSGi 网站公布。处理器名称是不区分大小写的。</td></tr>
<tr><td>名称</td><td>别名</td><td>描述</td></tr>
<tr><td>68k</td><td></td><td>68000 and up</td></tr>
<tr><td>ARM</td><td></td><td>Intel Strong ARM</td></tr>
<tr><td>Alpha</td><td></td><td>Compaq</td></tr>
<tr><td>Ignite</td><td>psc1k</td><td>PTSC</td></tr>
<tr><td>Mips</td><td></td><td>SGI</td></tr>
<tr><td>PArisc</td><td></td><td>Hewlett Packard</td></tr>
<tr><td>PowerPC</td><td>power ppc</td><td>Motorola/IBM</td></tr>
<tr><td>Sparc</td><td></td><td>SUN</td></tr>
<tr><td>x86</td><td>pentium i386 i486 i586 i686</td><td>Intel</td></tr>
<tr><td>x86-64</td><td>amd64</td><td>New 64 bit x86 architecture</td></tr>
<tr><td>org.osgi.framework.os.version</td><td colspan="3">操作系统版本。如果版本与标准格式（x.y.z）不匹配，那么操作者应该使用这个名称来定义这样一个系统属性。</td></tr>
<tr><td rowspan="6">org.osgi.framework.os.name</td><td colspan="3">框架服务器操作系统名称。下表定义了一系列操作系统名称，新的可用操作系统名称在 OSGi 网站公布。名称匹配是不区分大小写的。</td></tr>
<tr><td>名称</td><td>别名</td><td>描述</td></tr>
<tr><td>AIX</td><td></td><td>IBM</td></tr>
<tr><td>DigitalUnix</td><td></td><td>Compaq</td></tr>
<tr><td>FreeBSD</td><td></td><td>Free BSD</td></tr>
<tr><td>HPUX</td><td></td><td>Hewlett Packard</td></tr>
</table>

	IRIX		Silicon Graphics
	Linux		Open source
	MacOS		Apple
	Netware		Novell
	OpenBSD		Open source
	NetBSD		Open source
	OS2	OS/2	IBM
	QNX	procnto	
	Solaris		Sun Micro Systems
	SunOS		Sun Micro Systems
	VxWorks		WindRiver Systems
	Win32	Win*	All Microsoft Windows operating systems
	Windows95	Win95 Windows 95	Microsoft Windows 95
	Windows98	Win98 Windows 98	Microsoft Windows 98
	WindowsNT	WinNT Windows NT	Microsoft Windows NT
	WindowsCE	WinCE Windows CE	Microsoft Windows CE
	Windows2000	Win2000 Windows 2000	Microsoft Windows 2000
	WindowsXP	Windows XP, WinXP	Microsoft Windows XP
org.osgi.supports.« framework.extension	参考 3.12 一节		
org.osgi.supports.« bootclasspath.extension	参考 3.12 一节		
org.osgi.supports.« framework.fragment	参考 3.12 一节		
org.osgi.supports.« framework.requirebundle	参考 3.12 一节		
org.osgi.framework.« bootdelegation	参考 3.8.3 父级代理一节		
org.osgi.framework.« system.packages	参考 3.8.5 父类加载器一节		

所有的框架属性应该由系统用户定义为系统属性。如果在系统属性中没有定义这些属性，那么框架必须要根据Java系统环境来设置这些属性。

在别名一栏中描述了特定的操作系统返回的名称。框架应该将这些别名转换成为标准的处理器名称或者是操作系统名称。bundle开发人员则在manifest中的Bundle-NativeCode使用标准的名称。

4.5. 系统 bundle

与普通 bundle 相比，框架本身也是一个 bundle。这些 bundle 称之为系统 bundle。通过系统 bundle，框架可以注册供其他 bundle 使用的服务。例如包管理和权限管理服务。
在接口 BundleContext 中定义了方法 getBundle()，可以用来返回安装的系统 bundle 的集合。
系统 bundle 在以下方面有别于其他 bundle：

- 系统 bundle 的标志符通常是零（0）。
- 系统 bundle 的 getLocation 方法返回的是这样一个字符串："System Bundle"，在 Constants 接口中定义了这个常量。
- 系统 bundle 的符号名称有一个特殊的版本。因此，将名称 system.bundle 看作是 implementation-defined 的一个别名。
- 系统 bundle 的生命周期管理和一般 bundle 不一样，它的生命周期方法必须符合以下规范：
  - start — 由于系统 bundle 已经启动，所以不做任何操作
  - stop — 从方法中立即返回，并在另一个线程中关闭框架
  - update — 从方法中立即返回，并在另一个线程中重新启动框架
  - uninstall — 由于系统框架是不能卸载的，因此，这个方法抛出一个 bundle 异常（BundleException）
  - 更多信息可参考框架的启动和关闭一节
- 系统 bundle 的 Bundle.getHeaders 方法返回一个带头标 implementation-specific 的 Dictionary 对象。例如，系统 bundle 的 manifest 文件应该包括一个 Export-Package 头标声明，来描述框架导出的包。

4.6. 事件

OSGi 框架支持以下生命周期层事件：

- BundleEvent — 报告 bundle 的生命周期改变
- FrameworkEvent — 报告框架的启动、启动级别的改变、包的更新或者是捕获的错误。

事件的实际类型可以通过 getType 方法获得。返回的是一个整形数据，在类中定义了整形数据的含义。因此，可以在以后对事件进行扩充。忽略处理不可识别的事件。

4.6.1. 监听器

每一种类型的事件对应有一个监听器接口。下面描述了这些监听器：

- BundleListener 和 SynchronousBundleListener — 当 bundle 的生命周期信息改变后使用一个 BundleEvent 类型的事件来调用。
  对 SynchronousBundleListener 的调用是同步的，在处理这个事件的时候，必须要先于调用其他任何 BundleListener 对象。下面描述了当框架进入到另一个状态时，发出的一系列的事件：
  - INSTALLED – 框架安装后发出。
  - RESOLVED– 框架解析一个 bundle 后发出。
  - STARTING – 当框架即将启动一个 bunlde 时发出。只发送到

SynchronousBundleListener 对象。

- STARTED – 当框架已经启动了一个 bundle 后发出。
- STOPPING – 当框架即将停止一个 bundle 时发出。只发送到 SynchronousBundleListener 对象。
- STOPPED– 当框架停止了一个 bundle 时发出。
- UNINSTALLED – 当框架卸载了一个 bundle 后发出。
- UNRESOLVED – 当框架检测到一个 bundle 变成不可解析的时候发出；当刷新或者更新一个 bundle 的时候发生这样的事件。当使用包管理 API 来更新一系列的 bundle 时，那么序列中的每一个 bundle 都有抛出一个 UNRESOLVED BundleEvent 事件。这个事件必须要等到序列中所有的 bundle 都已经停止，并且序列中没有任何 bundle 重新启动才抛出，这是由于使用同步 bunle 监听器。RESOLVED 和 UNRESOLVED 并不需要成对出现。
- UPDATED – 在 bundle 更新之后发出。

- FrameworkListener — 发生 FrameworkEvent 类型的事件后调用。
  框架事件有以下类型：
  - ERROR – 需要操作者立即处理的重要错误。
  - INFO – 在特殊情况下需要的一般的信息。
  - PACKAGES_REFRESHED – 框架更新了包。
  - STARTED – 框架已经完成了初始化，正在普通模式下运行。
  - STARTLEVEL_CHANGED – 在设置和处理一个新的启动级别后由框架发出。
  - WARNING – 警告，提示操作者不是至关重要的但是存在潜在错误的信息。

在接口 BundleContext 中定义了可用于添加和移出每一种类型的监听器的方法。
除非一些特殊情况，事件是可以进行异步处理的，也就说并不是必须要使用和产生事件的同一个线程中来处理事件。并没有定义事件监听器的线程。
如果对于监听器的回调产生了不可检查的异常，那么在框架中必须要抛出一个框架事件 FrameworkEvent.ERROR。除非回调发生在传递事件 FrameworkEvent.ERROR 过程中（为了防止死循环）。

4.6.2. 事件发送

如果框架是异步传递事件的，那么框架必须：

- 在事件传递之前，保存一个事件发生时的监听器列表的快照（而不是在以后进行是在事件发送之前）。这样监听器就在事件发生之后不需要访问列表。
- 确保在保存快照的时候监听器所属的 bundle 处于活动状态。

如果框架在事件发生时没有捕获到当前的监听器列表，而是等到事件传递之前进行，那么可能导致以下错误：一个 bundle 已经启动并注册了一个事件监听器，bundle 就可以通过自己的 BundleEvent.INSTALLED 来查看自己的事件。
下面三种情景说明了上述的情况：

1. 情景一事件顺序：
   - 发生事件 A
   - 注册监听器 1
   - 尝试异步传递事件 A

   异常状态：监听器 1 不能接收到事件 A，由于它不是在事件发生前注册的。
2. 情景二事件顺序：

- 注册监听器 2
- 发生事件 B
- 取消注册监听器 2
- 尝试异步传递事件 B

异常状态：监听器 2 不能接收到事件 B，由于监听器 2 时在事件 B 发生时注册的。

3. 情景三事件顺序：
   - 注册监听器 3
   - 发生事件 C
   - 停止注册监听器 3 的 bundle
   - 尝试异步传递事件 C

   异常状态：监听器 3 肯定不能接收到事件 C，这是由于它的 BundleContext 对象不存在。

4.6.3. 同步处理缺陷

通常，调用监听器的 bundle 不应该保留有任何 Java 监控器。这也就是说在初始化回调时，框架和同步事件的发送者都应该不是处于监控器之中。
Java 监控器的目的在于保护更新数据的结构。也就是一小段不能调用任何不可监视代码的代码。在同步代码中调用 OSGi 框架可能导致不可预测的结果。其中之一有可能是导致死锁。死锁是指两个线程由于相互等待而阻塞。
可以通过超时来解决死锁问题，但是 java 监控器并没有使用超时。因此，线程一直挂起直到系统重新设置（Java 中不赞成所有能停止一个线程的方法）。这种死锁的预防方法就是在同步代码中不调用框架（否则其他代码可能导致回调）。
如果再调用其他代码时，必须要使用锁，那么使用 Java 监控器来创建一个信号量，这个信号量可以超时，以此来预防死锁。

4.7. 框架的启动和关闭

框架的实现必须要在提供任何服务可用之前启动。本规范中没有详细定义操作者应该如果来启动框架，这是由于不同的实现有不同的方法。一些框架的实现可能提供命令行的方式，还有一些可能是通过配置文件的方式来提供。在所有的情况中，框架的实现必须要按照给定的顺序完成以下操作。

4.7.1. 启动

当启动一个框架时，必须要进行以下操作：

1. 激活事件处理。这样可以将事件发送给监听器，在事件一节中进行了细述。
2. 系统 bundle 进入 STARTING 状态。参阅系统 bundle 一节。
3. 使用 Bundle.start 方法启动所有标记为启动的 bundle。启动过程中抛出的任何异常必须要包装成 BundleException，然后抛出为 FrameworkEvent.ERROR 的异常。在 Bundle 对象一节中讨论了 bundle 的状态。如果框架实现了可选的启动级别，那么情况会有所不同，详情参见启动级别服务规范一节。
4. 系统 bundle 进入活动状态（ACTIVE）。

5. 广播一个 FrameworkEvent.STARTED 事件

4.7.2. 关闭

由于某种原因，框架也需要关闭。同样，停止一个系统 bundle 也导致关闭框架，下面的操作必须按照顺序执行：

1. 系统 bundle 进入 STOPPING 状态
2. 调用 Bundle.stop 方法停止所有活动的 bundle，除非持久标记其必须在框架下次启动时重新启动。在关闭期间发生的异常必须包装成 BundleException 然后再发布为 FrameworkEvent.ERROR 类型的框架事件。如果框架实现了可选的启动级别，那么情况会有所不同，详情参阅启动级别规范一节。
3. 禁止事件处理。

4.8. 安全

4.8.1. 管理权限

管理权限是指用于授予管理框架权力的权限，包括了对子序列 bundle 的可选的约束，这些子序列称之为目标（targets）。例如，操作者可以给一个 bundle 赋予只有对主题名称为 ACME 的 bundle 拥有管理权限：

```
org.osgi.framework.AdminPermission(
    "(signer=\*, o=ACME, c=us)", ... )
```

管理权限是细粒度定义的。允许开发者只对一个 bundle 授予必需的权限。例如，一个 HTTP 的实现可能授予了访问所有 bundle 的所有资源的权限：

```
org.osgi.framework.AdminPermission("*",
    "resource" )
```

需要检查管理权限的代码必须要在其构造函数中使用一个 bundle 作为参数：AdminPermission(Bundle,String)中只有一个动作，这是为了确保权限检查的快速进行。

例如，在 loadClass 方法实现中必须要检查调用者是否有访问类空间的权限：

```
public class BundleImpl implements Bundle {

    Class loadClass(String name) {
        securityManager.checkPermission(
            new AdminPermission(this,"class") );
        ...
    }
}
```

当通过（有条件的）权限管理服务来分配 bundle 权限时，系统管理员很难事先知道分配给 bundle 的 ID。为了给管理权限提供一种更加实用的 bundle 命名方法，（有条件的）权限管理必须要提供从 PermissionInfo 对象重创建 AdminPermission 对象的一种特殊的支持。如果一个 PermissionInfo 对象指定了一个管理权限，那么参数 PermissionInfo 对象必须是一个过滤字符串。这个过滤器和 OSGi 中的过滤器有着相同的语法，但是关于位置和署名属性中的通

配规则却不一样。
过滤器中可以包括以下键：

- id – 指定 bundle 的 ID，例如：(id=256)
- location – bundle 的区域信息。支持通配符，可以指定一系列的 bundle。例如：(location=https://www.acme.com/download/*)
- signer – 一组 DN 名称。参阅证书匹配一节中的 DN 匹配。在 DN 中不能识别通配符。通配符要在其前加上一个反斜杠，以避免解释为一个过滤器通配符。例如：(signer=*,o=ACME,c=NL)
- name – bundle 的符号名称。通过使用通配符来支持指定一组 bundle。例如：(name=com.acme.*)

整个过滤器也可以是一个通配符。这样就匹配所有 bundle。

4.8.1.1. Actions

权限管理中的 Action 参数指定了一组框架允许的权限管理操作。操作如下表所示。同时，由于会有新版本的规范和增加系统服务，都可能添加其他的操作，所以这个表的内容并不是一成不变的。

Action	Used in
metadata	Bundle.getHeaders Bundle.getLocation
resource	Bundle.getResource Bundle.getResources Bundle.getEntry Bundle.getEntryPaths Bundle.findEntries Bundle resource/entry URL creation
class	Bundle.loadClass
lifecycle	BundleContext.installBundle Bundle.update Bundle.uninstall
execute	Bundle.start Bundle.stop StartLevel.setBundleStartLevel
listener	BundleContext.addBundleListener for SynchronousBundleListener BundleContext.removeBundleListener for SynchronousBundleListener
extensionLifecycle	BundleContext.installBundle for extension bundles Bundle.update for extension bundles Bundle.uninstall for extension bundles

Action	Used in
resolve	PackageAdmin.refreshPackages PackageAdmin.resolveBundles
startlevel	StartLevel.setStartLevel StartLevel.setInitialBundleStartLevel

通配符“*”表示所有的 Action。
必须要授予每一个 bundle 如下权限：AdminPermission(<bundle identifier>,"resource, metadata, class")，这样 bundle 才可以访问自己的资源。这个隐含的权限是由框架自动授予 bundle 的。resolve 和 startlevel 这两个 action 必须要使用系统 bundle 作为目标。
AdminPermission 类的实现需要框架实现来集成。在规范中也包含了一个 AdminPermission 类。如果有必要，框架的实现也可以修改这个类。然而，框架实现默认提供的类必须加载框架实现所提供的一个类。这个类来自的包根据以下属性定义：

org.osgi.vendor.framework

在这个包中的名称为 AdminPermission 的类必须是用于构造一个新的 Permission 实例，实现了委托的所有 AdminPermission 方法。

4.8.2. 对目标使用签名

Admin 权限管理使用 bundle 的签名者来选择一个目标。例如，通过授予一个 bundle 的权限来运行具有特定首要签名的 bundle 的生命周期操作。
通过使用首要签名者来作为目标，权限的维护变得要简单很多，这是由于这样就不需要对单独的 bundle 来进行设置了：有效使用了签名者组的机制。但是，也必须考虑到可以任意添加签名，这样需要考虑对管理特定签名者的权限。
使用多个签名者是一种解决方案同时也存在潜在的危险。从管理的角度来看，可以使用签名来处理组的情况。但是，这无法用于管理一个可信的 bundle。
例如，一个由 T 签名的可信 bundle，又增加了一个不可信的 U 的签名。这样就会授予 T 和 U 的 bundle 权限。但是，如果关联到 U 的权限也允许对 U 签名的 bundle 的管理，那么 U 就会意外获得了对这个可信 bundle 的管理权限。例如，U 现在就可以启动和停止这个可信 bundle。这种意外获得的管理权限应该要在使用多个签名者的时候仔细考虑。

4.8.3. 特权回调

下面定义了框架为 bundle 回调实现的接口：

- BundleActivator
- ServiceFactory
- Bundle-, Service-, and FrameworkListener

当框架调用了上述的回调接口时，导致回调的 bundle 也许还在堆栈中。例如，一个 bundle 安装并启动了另一个 bundle，当 BundleActivator.start 方法被调用时，bundle 安装者也许在堆栈中。同样，当一个 bundle 注册了一个服务对象，当框架回调所有限定的 ServiceListener 对象的 serviceChanged 方法时，注册 bundle 还在堆栈中。
当任意 bundle 的回调在任何时候试图访问一个保护域中的资源或者操作时，访问控制机制不仅要考虑到接收回调 bundle 的权限，同时还要考虑到框架和堆栈中的任何其他 bundle。这也就是说在这些回调中，bundle 开发人员应该在权限检查中使用 doPrivileged 调用（例如

在获取或者注册服务对象中）。
为了减少 bundle 编程人员对 doPrivileged 的调用，框架必须要在任何 bundle 的回调中调用 doPrivileged。框架应该实现 java.security.AllPermission。因此，对于 bundle 开发者来说，可以假定 bundle 除了自身的权限，并没有其他的权限限制。
在框架注册或调用的任何回调实现中，bundle 开发者不需要使用 doPrivileged。

For any other callbacks that are registered with a service object and therefore get invoked by the service-providing bundle directly, doPrivileged calls must be used in the callback implementation if the bundle's own privileges are to be exercised. Otherwise, the callback must fail if the bundle that initiated the callback lacks the required permissions.
A framework must never load classes in a doPrivileged region, but must instead use the current stack. This means that static initializers should never assume that they are privileged. Any privileged code in a static initializer must be guarded with a doPrivileged region in the static initializer.

4.9. 参考

[34] The Standard for the Format of ARPA Internet Text Messages
STD 11, RFC 822, UDEL, August 1982
http://www.ietf.org/rfc/rfc822.txt

[35] The Hypertext Transfer Protocol - HTTP/1.1
RFC 2068 DEC, MIT/LCS, UC Irvine, January 1997
http://www.ietf.org/rfc/rfc2068.txt

[36] The Java 2 Platform API Specification
Standard Edition, Sun Microsystems
http://java.sun.com/j2se

[37] The Java Language Specification
Second Edition, Sun Microsystems, 2000
http://java.sun.com/docs/books/jls/index.html

[38] A String Representation of LDAP Search Filters
RFC 1960, UMich, 1996
http://www.ietf.org/rfc/rfc1960.txt

[39] The Java Security Architecture for JDK 1.2
Version 1.0, Sun Microsystems, October 1998

[40] The Java 2 Package Versioning Specification
http://java.sun.com/j2se/1.4/docs/guide/versioning/index.html

[41] Codes for the Representation of Names of Languages
ISO 639, International Standards Organization
http://lcweb.loc.gov/standards/iso639-2/langhome.html

[42] Manifest Format
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR%20Manifest

[43] W3C EBNF
http://www.w3c.org/TR/REC-xml#sec-notation

[44] Lexical Structure Java Language
http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html
[45] Interval Notation
http://www.math.ohio-state.edu/courses/math104/interval.pdf

5. 服务层

版本号： 1.3

5.1. 简介

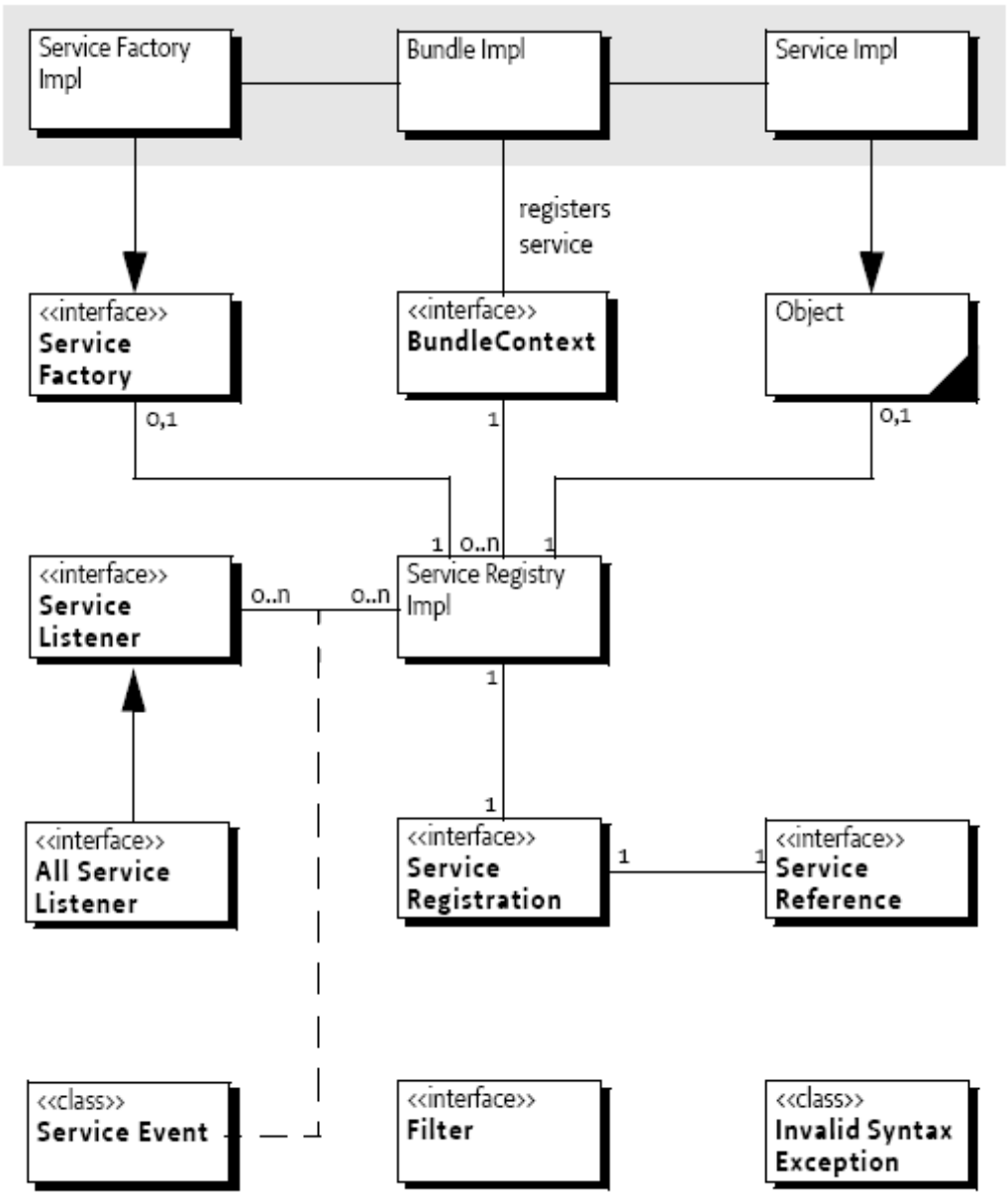
OSGi 服务层定义了一个和生命周期层紧密结合的动态协作模型。服务模型包括发布、查找和绑定模型。一个服务（service）就是一个通过服务登记来注册到一个或者多个 Java 接口下的 Java 对象。bundle 可以注册服务，查找服务，接收注册服务的状态改变信息。

5.1.1. 要点

- 协作性 — 服务层必须为 bundle 提供一种机制来发布、查找和绑定 bundle 之间的服务，而无需事先知道 bundle 的信息。
- 动态性 — 服务层机制必须能够处理外界或者是子结构的变化。
- 安全 — 必须能够限定对服务的访问。
- 深入性 — 对服务层的内部状态可以进行完全控制。
- 版本控制 — 提供对 bundle 以及对它们服务更新的控制。
- 持久性 — 在框架重启过程中提供方法来跟踪服务。

5.1.2. 名词

- 服务（Service） – 在一个或多个接口下注册了服务登记的具有属性值的类。可以由 bundle 查找发现并使用。
- 服务注册中心（Service Registry） – 保持服务注册。
- 服务引用（Service Reference） – 对一个服务的引用。提供对一个服务属性的访问但不是对实际的服务对象。对服务对象的访问必须要通过 bundle 的 BundleContext 来进行。
- 服务注册（Service Registration） – 当一个服务注册之后提供凭据。服务注册允许对服务进行更新和取消注册。
- 服务权限（Service Permission） – 当注册或者使用一个服务的时候，使用一个接口名称来进行权限控制。
- 服务工厂（Service Factory） – 由提供服务的 bundle 来为每一个使用服务的 bundle 定制一个服务对象的便捷方法。
- 服务监听器（Service Listener） – 服务事件监听器。
- 服务事件（Service Event） – 一个服务对象的注册、更改、取消注册的事件。
- 过滤器（Filter） – 实现了一个简单但是功能强大的过滤器语言的对象。可以用于属性选择。
- 语法错误异常（Invalid Syntax Exception） – 当一个过滤器表达式包含错误的时候抛出的异常。



5.2. 服务

在 OSGi 服务平台下，bundle 建立在一系列的相互协作的可用服务之上，这些服务共享一个服务注册中心。这样一个 OSGi 服务在语义上通过它的服务接口来定义，并实现为一个服务对象。

在服务接口中应该尽可能少的指定实现细节。OSGi 规定了很多常用的服务接口，以后还会增加。

服务对象是属于 bundle 的，而且在 bundle 之内运行。bundle 必须要将服务对象注册到框架的服务注册中心，这样，才可以在框架的控制下来为其他 bundle 提供服务。

提供服务的 bundle 和使用服务的 bundle 之间的依赖关系由框架来进行管理。例如，当停止一个 bundle 后，这个 bundle 在框架中注册的服务必须要自动的取消注册。

框架将服务映射到框架下的服务对象，而且，框架提供了一种简单而强大的查询机制，通过使用这种机制，bundle 就可以请求它需要的服务。框架也提供了一种事件机制，这样，bundle

就可以接收到服务注册、更改和取消注册的消息。

5.2.1. 服务引用

通常，注册的服务是通过一个 ServiceReference 对象来引用的。这样，在 bundle 只需要知道一个服务而不是一个服务对象的情况下，就可以避免创建 bundle 之间的不必要的动态服务依赖关系

可以把一个 ServiceReference 对象保存并发送给其他 bundle，而不会带来依赖关系。当 bundle 需要使用服务时，就可以通过将 ServiceReference 对象作为方法 getService 的参数来获取服务，即为 BundleContext. getService(ServiceReference)。详情参阅服务定位一节。

在 ServiceReference 对象中封装了属性和服务对象的其他元数据信息。其他 bundle 可以通过查找元数据信息来选择最合适的服务。

当 bundle 在框架的服务注册中心查找服务时，框架必须将请求服务目标的 ServiceReference 对象发送给请求者 bundle，而不是直接发送服务对象本身。也可以通过 ServiceRegistration 对象来得到一个 ServiceReference 对象。

只有当服务对象注册之后，ServiceReference 对象才是有效的。而只要 ServiceReference 对象存在，那么它的属性应该是可以使用的。

5.2.2. 服务接口

在服务接口中定义了服务的公有方法。实际上，bundle 的开发人员通过实现服务接口来创建一个服务对象，并将服务对象注册到框架的服务注册中心。一旦 bundle 通过一个接口名称注册了一个服务对象，那么 bundle 就可以通过这个接口名称来获得相关服务，并可以通过接口中定义的方法来使用服务。同时，框架也支持使用类名称来注册服务，因此，在本规范中，服务接口同时包括了接口和类。

当从框架来请求一个服务时，bundle 可以指定提供服务的对象必须要实现的服务接口名称。在请求中，bundle 也可以通过一个过滤器字符串来限定搜索范围。

5.2.3. 注册服务

bundle 通过在框架的服务注册中心注册一个服务对象来发布一个服务。那么安装在 OSGi 环境下的其它 bundle 就可以访问到在框架中注册的服务对象。

每一个注册服务对象都有一个惟一的 ServiceRegistration 对象，而且有一个或者多个引用的 ServiceReference 对象。这些 ServiceReference 对象公开了服务注册的属性，包括服务实现的一系列接口信息。可以使用 ServiceReference 对象来获取实现了特定接口的服务对象。

在框架中，允许 bundle 动态的注册和取消注册服务对象。因此，bundle 可以在 STARTING、ACTIVE 或者 STOPPING 状态下注册服务对象。

bundle 通过使用 BundleContext.registerService 方法，在框架中注册一个服务对象：

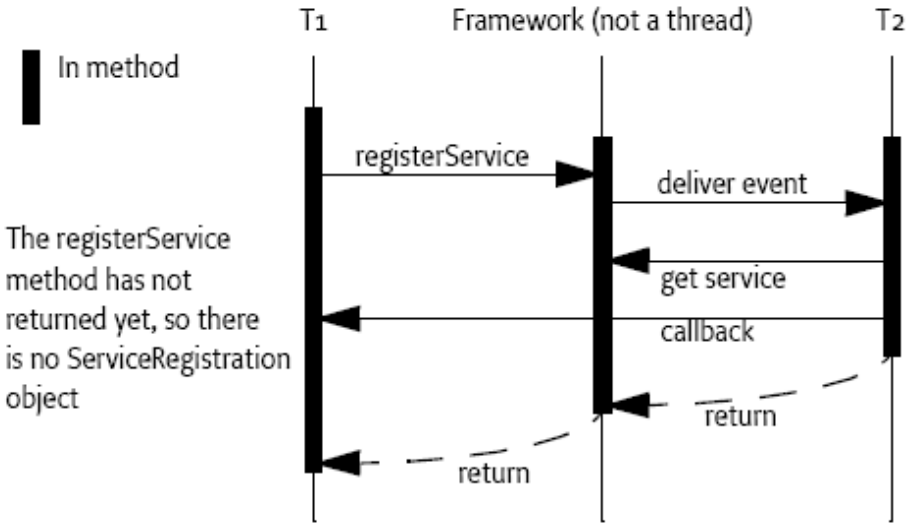
- registerService(String,Object,Dictionary) – 只实现了一个服务接口的服务对象注册。
- registerService(String[],Object,Dictionary) – 实现了多个服务接口的服务对象注册。

bundle需要注册的服务实现的服务接口名称以参数形式提供给registerService方法。框架必须要确保服务对象是每一个指定的服务接口的一个实例，或者这个对象是一个服务工厂。参阅

服务工厂一节。
为了进行上述检查，框架必须要从bundle或者一个共享包中为每一个指定的服务接口加载Class对象。对于每一个Class对象，在以服务对象为参数的Class对象中，必须调用它的Class.isInstance方法，并且返回true。
以后，需要注册的服务对象也许会使用一个Dictionary对象来描述，在这个Dictionary 对象中，包含了服务的属性的键值对序列。
如果一个服务对象成功注册，则将它实现的服务接口名称自动添加到服务对象的objectClass属性下。这个值必须由框架来自动设置，而由bundle提供的值则被覆盖了。
服务对象成功注册之后，框架给调用者返回一个ServiceRegistration对象。对服务的取消注册只能由ServiceRegistration对象（参阅unregister方法）的持有者进行。即使一个服务对象进行了多次注册，每次成功注册的服务对象必须产生一个惟一的ServiceRegistration对象。
使用ServiceRegistration对象是在服务对象注册后对它的属性进行可靠更改的惟一途径（参考setProperties方法）。在服务对象注册之后对它的Dictionary对象进行更改可能不会对服务属性有任何影响。
服务对象的注册处理过程包含了权限检查。注册服务的bundle必须要有ServicePermission权限：ServicePermission [<interface name>,REGISTER]，用来注册一个指定了所有服务接口的服务对象。否则，不能注册服务对象，并抛出一个安全异常（SecurityException）。

5.2.4. 服务注册对象的过早请求

服务注册的消息会通知所有注册了 ServiceListener 的对象。这是一个同步发送的过程。也就是说监听器可以在 registerService 方法返回之前就访问服务并调用它的方法。特定情况下，在这种回调中需要访问到 ServiceRegistration 对象。但是，注册服务的 bundle 还没有接收到 ServiceRegistration 对象。下图展示了这样一个序列：



在前面的例子中，对 ServiceRegistration 对象的访问可以通过 ServiceFactory 对象来获取。如果已经注册了一个 ServiceFactory 对象，框架在注册服务的 bundle 中回调 ServiceFactory 的 getService 方法。那么在这个方法执行完之后，通过这个方法的参数就可以获得 ServiceRegistration 对象。

5.2.5. 服务属性

属性信息保存在键值对中。键值为一个字符串对象，值则为 Filter 对象可以识别的类型。如果一个属性有多个值，则可以使用数组和 Vector 对象。
属性值应该限定为标准或原始的 Java 类型，以防止对 bundle 内部的依赖。框架不能检测到 bundle 之间通过服务属性产生的依赖关系。
属性键名称是不区分大小写的。ObjectClass、OBJECTCLASS 和 objectclass 都是指同样的含义。如果调用方法 ServiceReference.getPropertyKeys，那么框架必须返回它上次设置的属性值。如果 Dictionary 对象中存在只是大小写不同的键，那么框架必须要抛出一个异常。
服务属性提供了服务对象的信息，不应该将这些属性值用于服务的实际功能中。对注册服务的属性值更改时一种潜在的代价颇高的操作。例如，框架会提前将属性值加入到索引中以加快以后的查找速度。
Filter 接口支持复杂的过滤规则；可以用于查找匹配的服务对象。因此，框架的服务注册中心中所有属性共享一个名称空间。导致的结果就是为了防止产生名称冲突，需要使用描述性名称或者是形式化定义的更短的名称来区别不同的属性，所有的属性名称前缀为 service，同时在 OSGi 规范中保留了 objectClass 属性。
下表描述了预定义的标准服务属性：

属性	类型	常量	描述
objectClass	String[]	OBJECTCLASS	objectClass 属性包含了注册到框架中的服务对象所有实现的接口的集合。这个属性必须由框架自动设置。当使用 BundleContext 的 getService 方法获取服务对象时，框架必须要保证这时候服务对象可以声明为属性中指定的任何接口名称。
service.description	String	SERVICE_DESCRIPTION	service.description 属性用于文档性的描述，这个属性是可选的。在框架和 bundle 中可以利用这个属性来对其提供的服务注册对象作一个简短的描述。这个属性的主要目的在于进行调试，这是由于没有对这个属性的本地化支持。
service.id	Long	SERVICE_ID	每一个注册了的服务对象都由框架分配了一个惟一的 service.id。将这个标志数字添加到服务对象的属性中。框架给每一个注册的服务对象分配一个惟一的标志值，这个值要比原来分配的任何值要大，也就是说是递增分配的。
service.pid	String	SERVICE_PID	service.pid 属性是可选的，标记了服务对象的持久惟一标记。

service.ranking	Integer	SERVICE_RANKING	当注册一个服务对象时，bundle 也可以可选指定一个 service.ranking 数字作为服务对象的属性值。如果存在多个可用的服务接口，那么如果一个服务对象具有最大的 SERVICE_RANKING 值或者是它的值等于最小的 SERVICE_ID 值，那么就由框架返回这个服务对象。
service.vendor	String	SERVICE_VENDOR	这是一个可选属性，描述服务对象的开发商信息。

5.2.6. 持久标志符（PID）

持久标志符（PID）用于在框架重启之后标志服务。对于每次注册引用同一个实体的服务来说，应该使用包含了 PID 的服务属性。PID 的服务属性定义为 service.pid。对于不同愿景的框架中都持久性存在的服务，PID 是这种服务的惟一标记。对于一个给定的服务，应该使用相同的 PID。如果停止了一个 bundle 之后又重新启动它，那么必须使用相同的 PID。
PID 的格式如下所示：

pid ::= symbolic-name // 参考 1.4.2

PID 对于每一个服务来说是惟一的。bundle 不能使用同样的 PID 来注册多个服务，而且也不能和其他 bundle 使用相同的 PID，如果使用了相同的 PID，那么这是一种错误的用法。

5.2.7. 服务定位

为了使用一个服务对象并调用它的方法，bundle 必须首先获得服务的 ServiceReference 对象，在接口 BundleContext 中定义了两个方法来从框架中获取一个 ServiceReference 对象：

- getServiceReference(String) – 这个方法返回一个服务对象实现的 ServiceReference 对象，这个服务对象实现并注册了指定字符串参数的接口。如果存在多个这样的服务对象，那么选择一个具有最大 SERVICE_RANKING 值的服务对象。如果在队列中存在多个符合这样条件的服务对象，那么就使用具有最小 SERVICE_ID 值的服务对象（也就是最先注册的）。
- getServiceReferences(String,String) – 这个方法返回一个 ServiceReference 对象数组，返回数据符合以下条件：
  - 实现并注册了指定的服务接口
  - 符合指定的查找过滤器条件。在 Filter 一节中解释了过滤器语法。

如果没有找到符合条件的服务对象，那么上述两个方法都返回 null 值。否则，调用者接收到一个或多个 ServiceReference 对象。可以通过 ServiceReference 对象来取得服务对象的属性值，还可以通过 BundleContext 对象使用 ServiceReference 对象获得实际的服务对象。
这两种方法的调用都需要调用者具有 ServicePermission 权限：ServicePermission[<name>, GET]来通过指定服务接口获得服务对象。如果调用者没有权限，那么这两个方法都是返回 null 值。

5.2.8. 获取服务属性

为了允许访问服务对象，ServiceReference 接口定义了以下两种方法：

- getPropertyKeys() – 返回所与可用的属性键数组。
- getProperty(String) – 返回指定键名称的键值。

即使从框架中取消注册了服务对象，也可以通过上述方法来获得信息。当 ServiceReference 对象和登录服务关联时，这种机制就非常有用。

5.2.9. 获取服务对象

通过使用 BundleContext 对象来获取一个实际的服务对象，这样框架就可以来管理这些依赖关系了。如果一个 bundle 需要一个服务对象，这样这个 bundle 就依赖于注册的服务对象的生命周期。对于这种依赖可以通过可以获取服务对象的 BundleContext 对象来跟踪，这也就是为什么在 bundle 之间共享 BundleContext 对象需要非常的小心。

BundleContext对象的方法BundleContext.getService(ServiceReference)返回一个实现了接口的对象，在 objectClass 属性中定义了需要这些接口。getService 方法有以下特征：

- 如果服务对象取消注册之后，那么方法返回 null 值。
- 检查调用者是否有权限：ServicePermission[<interface name>,GET]，至少需要有一个注册的服务接口的权限。权限检查是非常有必要的，这样就可以传递 ServiceReference 对象而无须考虑泄密这样的安全问题。
- BundleContext 的对服务对象的使用计数加一。
- 如果服务对象没有实现 ServiceFactory 接口，那么直接返回。否则如果 BundleContext 的服务对象是一个，如果这个对象声明为一个 ServiceFactory 对象，则调用 getService 方法来为调用的 bundle 创建一个定制的服务对象。再则，返回一个定制对象的缓冲拷贝。关于 ServiceFactory 对象参阅服务工厂一节。

5.2.10. 服务的其它信息

在 Bundle 接口中定义了以下两种方法来返回与服务使用的 bundle 有关的信息：

- getRegisteredServices() – 返回 bundle 在框架中注册的服务对象。
- getServicesInUse() – 返回 bundle 使用的服务对象。

5.3. 服务事件

- ServiceEvent – 报告服务对象的注册、取消注册和属性改变事件。所有这些事件都是同步发送的。可以通过 getType 方法来获取事件的类型，这个方法返回的类型是 int 类型的，关于事件类型的定义以后可能会扩充，而不可识别的事件类型则是忽略处理的。
- ServiceListener – 当对一个服务对象进行注册或者改变或者是在取消注册的过程中时，使用一个 ServiceEvent 事件来调用。当一个 ServiceEvent 发生之后，对于每一个注册的监听器必须要对其进行安全检查。只有注册监听器的 bundle 具有对至少一个注册的服务接口的以下权限时：ServicePermission[<interface name>,GET]，才可以调用监听器。

使用服务对象的 bundle 应该注册一个 ServiceListener 对象来跟踪服务对象的可用性，并在取消注册服务对象时采取合适的动作。

5.4. 过期引用

框架必须要管理 bundle 之间的依赖关系。这种管理受限于框架的结构。bundle 通过监听框架产生的事件来清理和移除过期引用。

过期引用是指对这样一个 Java 对象的引用，这个对象所在的类加载器所属的 bundle 已经停止，或者所关联的服务对象已经是取消注册了。标准 Java 处理中并没有提供任何方法来清除这些过期引用，这就需要 bundle 的开发人员必须对他们的代码进行分析，并确保删除了过期引用。

过期引用存在潜在的危害，这是由于它们阻止了 Java 的垃圾收集器对已经停止的 bundle 的类以及可能的实例进行回收。这样就会导致显著的内存消耗，并可能使得对本地代码的更新失败。因此，强烈建议使用服务的 bundle 使用服务跟踪器（Service Tracker）或者是公布服务（Declarative Services）。

服务开发人员可以通过使用如下机制来减小（并不能完全消除）过期引用的危害：

- 通过使用 ServiceFactory 接口来实现服务对象。ServiceFactory 接口中的方法可以简单跟踪使用服务对象的 bundle。参阅服务工厂一节。
- 间接使用服务对象。服务对象提供给其他 bundle 的应该是一个对实际服务对象的一个指针。当服务对象不可用之后，将指针设置为 null 值，这样就有效的移除了对实际服务对象的引用。

一个已经取消注册的服务的行为是不确定的。这些服务也许可以正常工作，也许会抛出一个异常，这取决于它们的实现情况。应该将这种类型的错误记录到日志中。

5.5. 过滤器

框架提供了一个 Filter 接口，并且在方法 getServiceReferences 中使用过滤器语法（在过滤器语法一节中定义）。通过调用 BundleContext 的方法 createFilter(String)或者是 FrameworkUtil 的方法 createFilter(String)来创建一个 Filter 对象，方法中的参数为一个选定的过滤规则字符串。过滤字符串支持以下匹配方法：

- match(ServiceReference) – 通过对 Service Reference 的属性进行匹配，匹配不区分大小写。
- match(Dictionary) – 通过给定的 Dictionary 对象来查找，也是不区分大小写的。
- matchCase(Dictionary) –通过给定的 Dictionary 对象来查找，不同之处在于是区分大小写的。

可以对一个 Filter 对象进行多次匹配来确定是否匹配。如果找到了匹配的 ServiceReference 对象或者 Dictionary 对象，那么就通过它们来创建 Filter 对象。

匹配的过程需要将过滤器中的字符串和目标对象包括服务属性或者是 Dictionary 进行比较。如果目标对象的类实现了带有一个字符串参数的构造方法，并且实现了 Comparable 接口，那么就可以通过 Comparable 接口来进行比较。也就是说，如果目标对象是一个 Target 类，那么这个 Target 类必须实现：

- 构造方法 Target(String)
- 实现了接口 java.lang.Comparable

如果目标类没有实现接口 Comparable，那么当对象相等（使用 equals 方法），对对象使用操作符=、~=、<=、>=将只能返回 true。Target 类不一定要是 public 类，下面的例子说明了类可以通过过滤器对一个枚举分类进行校验：

```
public class B implements Comparable {
    String keys[] = {"bugs", "daffy", "elmer", "pepe"};
    int index;
    public B(String s) {
        for ( index=0; index<keys.length; index++ )
            if ( keys[index].equals(s) )
                return;
    }
    public int compareTo( Object other ) {
        B vother = (B) other;
        return index - vother.index;
    }
}
```

可以对这个类使用如下过滤器：

```
(!(enum>=elmer)) -> matches bugs and daffy
```

方法 Filter.toString 必须返回一个不带空格的过滤字符串。

5.6. 服务工厂

通过服务工厂，可以在 bundle 调用方法 BundleContext.getService(ServiceReference)时返回一个定制的服务对象。

通常，对一个 bundle 注册的服务对象的调用是直接返回这个服务对象。但是，如果这个注册的服务对象实现了 ServiceFactory 接口，那么框架必须调用这个服务对象的方法来为每一个使用服务的 bundle 创建一个惟一的服务对象。当 bundle 不再使用服务对象——比如停止了 bundle 之后——那么框架必须要将事件通知 ServiceFactory 对象。

可以用 ServiceFactory 来管理 bundle 间的依赖关系，而框架并没有很明确的管理这种依赖关系。通过绑定一个请求 bundle 的返回服务对象，那么当 bundle 停止使用服务之后，比如停止 bundle，服务对象就可以接收到事件通知，然后就可以释放提供给那个 bundle 的服务资源。

在 ServiceFactory 接口中定义了以下方法：

- getService(Bundle,ServiceRegistration) –如果是方法 BundleContext.getService 来调用，并且下述情况为真，则由框架调用这个方法：
  - BundleContext.getService 中的参数 ServiceReference 关联的服务对象实现了 ServiceFactory 接口。
  - bundle 对那个服务对象的使用计数是 0，也就说 bundle 当前并没有对这个服务对象的任何依赖。

  对方法 BundleContext.getService 的调用请求必须要是由框架发出到 getService 方法，将 bundle 对象发送给调用者。框架必须要缓存请求的 bundle-to-service 映射，并且如果以后 bundle 对这个服务对象的使用计数要大于 0，则对于 BundleContext.getService 调用，返回给 bundle 缓存的服务对象。

  框架必须检查方法返回的服务对象。如果它不是服务工厂注册时的类的一个实例，那么

对 getService 方法的调用返回 null 值。这种检查必须根据注册服务一节中描述的规范来进行。

- ungetService(Bundle,ServiceRegistration,Object) – 如果是 BundleContext.ungetService 方法调用，并且下述条件为真，则由框架调用这个方法：
  - BundleContext.getService 中的参数 ServiceReference 关联的服务对象实现了 ServiceFactory 接口。
  - 调用返回之后，bundle 对服务对象的使用计数降为零，也就说 bundle 即将释放对这个服务对象的依赖。

  对方法 BundleContext.getService 的调用请求必须要是由框架发出到 getService 方法，这样 ServiceFactory 对象就可以将事先创建的服务对象释放。

  另外，服务对象事先创建的缓存拷贝必须是和框架没有引用关联的，这样就可以通过 gc 进行回收。

5.7. 服务释放

如果 bundle 需要释放一个服务对象，那么 bundle 必须移除它和注册服务的 bundle 之间的动态依赖。在接口 BundleContext 中定义了一个方法来释放服务对象：ungetService(ServiceReference)，在这个方法中使用了一个 ServiceReference 对象作为参数。方法返回一个布尔值：

- false：如果调用方法时，bundle对服务对象的使用计数为0，或者是已经取消注册了服务对象。
- true：如果调用方法时，bundle对服务对象的使用计数大于0。

5.8. 取消注册服务

在接口 ServiceRegistration 中定义了方法 unregister()来取消注册一个服务对象。这个方法必须要从框架的服务注册中心移除指定服务对象。ServiceRegistration 对象的 ServiceReference 对象不能再对服务对象进行访问。

ServiceRegistration 对象中的这个方法的实际作用在于确保了只有拥有这个对象的 bundle 才可以取消注册相关的服务对象。那么取消服务注册的 bundle 与注册服务的 bundle 可能不是同一个 bundle。例如，注册服务的 bundle 将 ServiceRegistration 对象发送给另一个 bundle，这样，就将取消注册服务的职责赋予给另一个 bundle。对 ServiceRegistration 对象的传递应该特别小心。

方法 ServiceRegistration.unregister 成功返回之后，服务对象必须达到如下要求：

- 从框架的服务注册中心完全的移除了。因此，服务对象的 ServiceReference 对象也就不能再用于进行访问服务对象了。对方法 BundleContext.getService 的调用也将返回 null 值。
- 处于未注册状态，即使存在其他 bundle 的依赖。所有的 bundle 都应该接收到取消注册的事件通知，事件类型是 ServiceEvent.UNREGISTERING。这个事件是同步发送的，这样 bundle 就可以释放服务对象。

  bundle 接收到 ServiceEvent.UNREGISTERING 事件之后，bundle 应该释放服务对象，释放对这个对象的任何引用，这样 Java VM 的 gc 就可以回收服务对象的资源。
- 所有使用服务的 bundle 释放资源。对于调用事件监听器之后，如果 bundle 对服务对象

的使用计数还是大于 0，那么就由框架来将使用计数清零，并释放服务对象。

5.9. 多版本导出

允许多个 bundle 导出同名的包给框架实现和 bundle 开发带来了复杂度的提高：不能用类名来惟一标志导出的类。这影响到了服务注册中心和权限的检查。

5.9.1. 服务注册中心

bundle 不能暴露存在冲突的类加载器的服务。bundle 获取的服务应该是可以安全的声明为服务注册时登记的接口或者类的类型，并且应该是可以访问获取的服务。而不能因为这些接口来自于不同的类加载器就抛出类声明异常（ClassCastExceptions）。而服务注册中心应该确保 bundle 所访问的服务都是没有冲突的。bundle 获取的服务是没有冲突的是指：和注册服务的 bundle 相比较，这个 bundle 没有和接口包的另一个类加载器进行连接。也就是说，它要么连接到同一个类加载器，要么就和那个包没有进行连接。
要求 bundle 中不能有存在冲突的服务是非常重要的，因此，可以通过以下方法来过滤存在冲突的 ServiceReferenc 对象。通过 BundleContext 对象来标记 bundle：

- getServiceReference(String) – 返回和调用 bundle 的指定接口不冲突的一个 ServiceReference 对象。
- getServiceReferences(String,String) – 返回和调用 bundle 的指定接口不冲突的多个 ServiceReference 对象。

方法 getAllServiceReferences(String,String)提供了对服务注册中心的所有没有任何兼容约束的服务的访问，通过这个方法获取的服务可能会导致抛出类声明异常。
ServiceReference 类中定义的方法 isAssignableTo(Bundle,String)也可以用来测试这个 ServiceReference 对象关联的服务的注册 bundle 和指定的 bundle 是否关联到了指定接口的同样的源。

5.9.2. 服务事件

服务事件只发送到和 ServiceReference 不矛盾的事件监听器。
而有一些 bundle 需要监听所有的服务事件而不考虑是否存在冲突的问题。这样增加了一种新的服务监听器：AllServiceListener。这是一个标记接口，继承了 ServiceListener。使用了这个标记接口的监听器向框架标明了它们需要可以访问到所有的服务，包括存在冲突的服务。

5.10. 安全

5.10.1. 服务权限控制

ServicePermission 中包括以下变量：

- Interface Name – 接口名称中可能包括了一个通配符来匹配多个接口名称（在

java.security.BasicPermission 中描述了通配符）。

- Action – 支持的 Action 包括：
  - REGISTER – 权限的持有者可以注册这个服务对象。
  - GET – 权限持有者可以获取服务。

当使用 BundleContext.registerService 来注册一个服务对象时，注册服务的 bundle 必须要有的 ServicePermission 权限。参阅服务注册一节。

当通过 BundleContext.getServiceReference 或者 BundleContext.getServiceReferences 来获取一个 ServiceReference 对象时，调用方法的 bundle 必须要 ServicePermission[<interface name>, GET]权限来获取服务对象。参阅服务引用一节。

当通过 BundleContext.getService(ServiceReference)方法来获取一个服务对象，调用方法的代码必须至少有关于注册的一个类的权限：ServicePermission[<name>, GET]。

ServicePermission 必须用于对服务事件监听器接收的事件的过滤器，同时也是列举服务的方法的过滤器，方法包括了 Bundle.getRegisteredServices 和 Bundle.getServicesInUse。框架必须要确保 bundle 不能够检测到它没有权限访问的服务的状态。

6. Framework API

版本 1.3

6.1. org.osgi.framework

OSGi 框架包。规范版本：1.3。
如果 bundle 需要使用这些包，那么在它的 manifest 中的 Import-Package 必须要列出使用的包名称，例如：

Import-Package: org.osgi.framework;version=1.3

6.1.1. 要点

- AdminPermission – 描述调用者对 bundle 的管理操作或者获取信息的权限。[p.120]
- AllServiceListener – 服务事件监听器。[p.123]
- Bundle – 框架中已经安装的 bundle。[p.123]
- BundleActivator – bundle 启动和停止的定制化。[p.136]
- BundleContext – 框架中 bundle 执行的上下文环境。[p.137]
- BundleEvent – 框架描述 bundle 的生命周期改变的事件。[p.148]
- BundleException – 描述 bundle 的生命周期中发生的意外的框架异常。[p.149]
- BundleListener – 描述 bundle 事件监听器。[p.150]
- BundlePermission – 描述 bundle 访问或者提供一个 bundle，接收或者附加片断的权限。[p.151]
- Configurable – 支持一个配置对象。[p.152]
- Constants – 定义了 OSGi 环境中属性名称、服务属性名称和 Manifest 头标属性名称的标准。[p.153]
- Filter – 基于 RFC 1960-based 的过滤器。[p.167]
- FrameworkEvent – 框架产生的事件。[p.168]
- FrameworkListener – 框架事件的监听器。[p.170]
- FrameworkUtil – 框架的实用类。[p.170]
- InvalidSyntaxException – 一种框架异常。[p.171]
- PackagePermission – bundle 导入或者导出包的权限。[p.172]
- ServiceEvent – 描述服务的生命周期改变的来自框架的事件。[p.173]
- ServiceFactory – 在 OSGi 环境下允许服务定制的方法。[p.175]
- ServiceListener –服务事件监听器。[p.176]
- ServicePermission – 描述了 bundle 注册或者获取一个服务的权限。[p.176]
- ServiceReference – 对一个服务的引用。[p.177]
- ServiceRegistration – 一个已经注册的服务。[p.179]
- SynchronousBundleListener – 一个同步的 bundle 事件监听器。[p.180]
- Version – bundle 和包的版本标记。[p.181]

6.1.2. public final class AdminPermission extends BasicPermission

描述调用者对 bundle 进行特殊的管理操作或者是获取 bundle 敏感信息的权限。所有的操作权如下：

Action	Methods
class	Bundle.loadClass
execute	Bundle.start
	Bundle.stop
	StartLevel.setBundleStartLevel
extensionLifecycle	BundleContext.installBundle for extension bundles
	Bundle.update for extension bundles
	Bundle.uninstall for extension bundles
lifecycle	BundleContext.installBundle
	Bundle.update
	Bundle.uninstall
listener	BundleContext.addBundleListener for SynchronousBundleListener
	BundleContext.removeBundleListener for SynchronousBundleListener
metadata	Bundle.getHeaders
	Bundle.getLocation
resolve	PackageAdmin.refreshPackages
	PackageAdmin.resolveBundles
resource	Bundle.getResource
	Bundle.getResources
	Bundle.getEntry
	Bundle.getEntryPaths
	Bundle.findEntries
	Bundle resource/entry URL creation
startlevel	StartLevel.setStartLevel
	StartLevel.setInitialBundleStartLevel

通配符“*”表示所有的操作。

权限描述是一个过滤器表达式。表达式具有以下参数描述：

- signer – 用于给一个 bundle 签名特异名称链（Distinguished Name chain）。根据过滤表达式规则，在 DN 中不能匹配通配符，虽然在 DN 链的定义中可以匹配。
- location – bundle 的位置。
- id – 指定 bundle 的 bundle ID。
- name – bundle 的符号名称。

6.1.2.1. public static final String CLASS = “class”

class 操作（值为“class”）

Since 1.3

6.1.2.2. public static final String EXECUTE = “execute”

execute 操作（值为“execute”）

Since 1.3

6.1.2.3. public static final String EXTENSIONLIFECYCLE = “extensionLifecycle”

extensionLifecycle 操作（值为“extensionLifecycle”）

Since 1.3

6.1.2.4. public static final String LIFECYCLE = “lifecycle”

lifecycle 操作（值为“lifecycle”）

Since 1.3

6.1.2.5. public static final String LISTENER = “listener”

listener 操作（值为“listener”）

Since 1.3

6.1.2.6. public static final String METADATA = “metadata”

metadata 操作（值为“metadata”）

Since 1.3

6.1.2.7. public static final String RESOLVE = “resolve”

resolve 操作（值为“resolve”）

Since 1.3

6.1.2.8. public static final String RESOURCE = “resource”

resource 操作（值为“resource”）

Since 1.3

6.1.2.9. public static final String STARTLEVEL = “startlevel”

startlevel 操作（值为“startlevel”）

Since 1.3

6.1.2.10.public AdminPermission( )

- 创建一个匹配所有 bundle 的所有的操作的 AdminPermission 对象，等同于 AdminPermission(”*”,”*”)

6.1.2.11.public AdminPermission(String filter, String actions)

filter 过滤器表达式，拥有键 signer，location，id，和 name。如果值为“*”那么可以匹配所有的 bundle。

actions 操作，可以为：class, execute, extensionLifecycle, lifecycle, listener, metadata, resolve, resource 或者 startlevel。通配符“*”表示所有的操作。

- 创建一个新的 AdminPermission 对象。这个构造器只能用于创建一个即将检查的权限。

例如：

```
(signer=\*,o=ACME,c=US)
(&(signer=\*,o=ACME,c=US)(name=com.acme.*)(location=http://
www.acme.com/bundles/*))
(id>=1)
```

当在过滤表达式中使用 signer ，那么 signer 的值不能为特殊的字符(’*’, ‘(’, ‘)’)。

如果参数值为 null，则等同于“*”。

6.1.2.12.public AdminPermission(Bundle bundle, String actions)

bundle 一个 bundle

actions class, execute, extensionLifecycle, lifecycle, listener, metadata, resolve, resource, startlevel

- 创建一个新的 AdminPermission 对象，由必须检测 Permission 对象的代码使用。

Since 1.3

6.1.2.13. public boolean equals(Object obj)

obj 使用这个对象来进行比较

- 确定两个 AdminPermission 对象是否相等

Returns 如果两个 AdminPermission 对象相等返回 true，否则返回 false

6.1.2.14.public String getActions( )

□ 返回表示 AdminPermission 对象的操作的范式字符串。
通常返回的顺序如下：
class, execute, extensionLifecycle, lifecycle, listener, metadata, resolve, resource, startlevel

Returns 表示 AdminPermission 对象操作的范式字符串。

6.1.2.15.public int hashCode( )

□ 返回对象的哈希值

Returns 对象的哈希值

6.1.2.16.public boolean implies(Permission p)

p 询问的权限

□ 确定对象是否蕴含了指定的权限。如果指定的 Permission 构建时没有关联到 bundle，那么方法的调用会抛出异常。
如果指定权限是 AdminPermission，并且满足以下条件：

- 这个 AdminPermission 对象的过滤器和指定 Permission 对象的 bundle ID，bundle 符号名称，bundle 位置和 bundle 签名 DN 链匹配，或者
- 过滤器为“*”

而且满足条件：AdminPermission 对象的操作中包括了指定的 Permission 对象的操作。那么返回 true。
特例：如果指定的 Permission 对象通过使用“*”过滤器来构建，那么如果 AdminPermission 对象的过滤器是“*”而且它的操作中包含了指定 Permission 的操作。

Returns 如果 AdminPermission 对象中隐含指定 Permission 对象，那么返回 true，否则返回 false。

Throws RuntimeException – 如果指定 Permission 对象没有使用 bundle 来构建或者是使用了“*”

6.1.2.17.public PermissionCollection newPermissionCollection( )

□ 返回一个适合于存储 AdminPermission 对象的 PermissionCollection 对象。

Returns 一个新的 PermissionCollection 对象。

6.1.3. public interface AllServiceListener extends ServiceListener

一个服务监听器。
AllServiceListener 是一个监听器接口，bundle 开发人员可以实现这个接口。
框架可以通过 BundleContext.addServiceListener 方法来注册一个 AllServiceListener 对象。当注册、修改或者正在处理取消注册一个服务，就会通过一个 ServiceEvent 对象来调用 AllServiceListener 对象。
发送给 AllServiceListener 对象的 ServiceEvent 对象是已经过滤了的，使用的过滤器是在注册监听器时指定的过滤器。如果 Java 运行环境支持权限控制，那么还有附加的过滤。只有当定义监听器的 bundle 具有适当 ServicePermission 权限来获得服务，其中获得服务是使用了至少一个注册服务的类，那么这样才会把 ServiceEvent 发送给监听器。
和普通的 ServiceListener 不一样的是，AllServiceListener 对象接收所有的 ServiceEvent 对象，而不管来自监听器所在 bundle 的包是否等于来自注册服务 bundle 的包。这也就是说，如果在检索服务对象时，监听器不会广播服务对象到任何相关的服务接口。

See Also ServiceEvent[p.173] , ServicePermission[p.176]

Since 1.3

6.1.4. public interface Bundle

框架中一个已经安装的 bundle。
可以通过 Bundle 对象来对一个已经安装的 bundle 的生命周期进行控制。
在 OSGi 框架中安装的每一个 bundle 都有一个相关的 Bundle 对象。
bundle 必须要有一个惟一标识，这是框架提供的一个长整形数。
在 bundle 的生命周期期间，这个标识符是不能修改的，即使对 bundle 进行了更新。而卸载然后重新安装的 bundle 必须重新分配一个标识符。
bundle 具有六种状态：

- UNINSTALLED[p.125]
- INSTALLED[p.124]
- RESOLVED[p.124]
- STARTING[p.125]
- STOPPING[p.125]
- ACTIVE[p.124]

每种状态分配的值是没有规定顺序的；值为位串，可以通过 OR 操作来获得 bundle 的有效状态。
只有当 bundle 的状态是在启动、活动或者是停止时才可以运行代码。处于卸载状态的 bundle 是不能设置为其他状态的；这种状态只是由于其他地方存在引用而形成的一种僵死状态。
只有框架才能创建 Bundle 对象，而且只有在创建它们的框架内有效。

6.1.4.1. public static final int ACTIVE = 32

表示 bundle 正在运行。
当 bundle 成功启动之后就进入了 ACTIVE 状态。
ACTIVE 状态的值为 0x00000020。

6.1.4.2. public static final int INSTALLED = 2

描述 bundle 已经安装但是还没有解析。
当框架已经安装了 bundle 但是还不能运行时的状态称为 INSTALLED 状态。如果 bundle 代码的依赖关系还没有解析，这个状态是可见的。框架将尝试解析处于 INSTALLED 状态的 bundle 代码依赖关系，并且将 bundle 转移到 RESOLVED 状态。
INSTALLED 状态的值为 0x00000002。

6.1.4.3. public static final int RESOLVED = 4

描述 bundle 已经解析，而且可以启动。
当框架成功的解析了 bundle 的依赖关系之后，bundle 就处于 RESOLVED 状态。这些依赖关系包括：

- 来自于清单文件中 Constants.BUNDLE_CLASSPATH 描述的 bundle 类路径。
- 来自于清单文件中 Constants.EXPORT_PACKAGE 和 Constants.IMPORT_PACKAGE 描述的包的依赖关系。
- 来自于清单文件中 Constants.REQUIRE_BUNDLE 描述的需要的 bundle 依赖关系。
- 来自于清单文件中 Constants.FRAGMENT_HOST 所描述的片断 bundle 依赖关系。

需要注意的是此时 bundle 尚未激活。在启动 bundle 之前必须将其状态置为 RESOLVED。框架可能在任何时候都会尝试对 bundle 进行解析。
RESOLVED 状态的值为 0x00000004。

6.1.4.4. public static final int STARTING = 8

描述 bundle 处于启动过程中。
只有当调用 start 方法启动 bundle 之后，bundle 进入 STARTING 状态。
当调用了 bundle 的 BundleActivator.start 方法时，bundle 必须进入 STARTING 状态。如果方法成功返回，而没有抛出异常，那么 bundle 成功启动，并且进入到 ACTIVE 状态。
STARTING 状态的值为 0x00000008。

6.1.4.5. public static final int STOPPING = 16

描述 bundle 处于停止状态。
当调用 bundle 的 stop 方法时，bundle 进入到 STOPPING 状态。当调用 BundleActivator.stop 方法之后，bundle 必须要进入这个状态。当方法成功返回而且完全停止了 bundle 之后，bundle 进入到 RESOLVED 状态。
STOPPING 状态的值为 0x00000010。

6.1.4.6. public static final int UNINSTALLED = 1

描述卸载 bundle 之后而且不能再使用的状态。
只有当卸载了 bundle 之后，bundle 的状态才是 UNINSTALLED 状态；虽然不能再使用 bundle，但是对这个 Bundle 对象的引用还有有效的，可以根据引用来进行自省（introspection）。
UNINSTALLED 状态的值为 0x00000001。

6.1.4.7. public Enumeration findEntries(String path, String filePattern, boolean recurse)

path 搜索的路径名称。特殊路径“/”表示 bundle 根路径。路径和 bundle 的来源有关，而且必须不能为 null 值。

filePattern 在指定路径下选择条目（entry）的文件名称模式。只和条目地址的最后单元进行匹配，而且支持子串匹配，如同 Filter 规范一节中的描述，可以使用通配符“*”。如果 filePattern 为 null 值，那么就和“*”的处理一样，匹配所有的文件。

recurse 如果为 true，那么递归遍历所有的子目录；如果为 false，那么就只会返回当前目录中符合条件的条目。

□ 返回 bundle 以及它的附加片断中的匹配条目。不使用 bundle 的类加载器来查找条目。只会搜索 bundle 以及它附加片断中的内容。如果 bundle 处于 INSTALLED 状态，那么在查找之前会尝试对 bundle 进行解析。
这个方法的意图在于提供从 bundle 中获取配置、设置、本地化以及其他信息的途径。这个方法考虑到 bundle 的“目录”中也扩加了片断的信息。这样的一个“bundle 空间”并不是一个惟一成员的命名空间；同样名称的条目可能出现多次。因此这个方法返回的是 URL 对象的枚举对象。这些 URL 对象可以是来自不同的 JAR 文件，但是具有相同的路径名。这个方法既可以指返回指定路径的条目，也可以返回指定路径的子目录下的条目。在解析 bundle 之后，就可以附加片断，而可能会因此改变了这个方法返回的 URL 集合。如果还没有解析这个 bundle，那么只会返回这个 bundle 的 JAR 文件的条目。
例如：

```
// List all XML files in the OSGI-INF directory and below
Enumeration e = b.findEntries("OSGI-INF", "*.xml", true);

// Find a specific localization file
Enumeration e = b.findEntries("OSGI-INF/l10n",
                              "bundle_nl_DU.properties",
                              false);
if (e.hasMoreElements())
return (URL) e.nextElement();
```

Returns URL 对象的枚举对象，其中每一个 URL 对象是对应于一个匹配条目的。如果不能找到匹配条目，那么返回 null 值，或者如果 Java 运行环境支持权限控制而调用这没有适应的管理权限：AdminPermission[this,RESOURCE]。返回条目的 URL 的排列顺序如下：首先返回 bundle 片断中条目，而且按照 bundle 的 ID 升序排列。如果该 bundle 是一个片断，那么只会返回这个片断中的匹配结果。

Since 1.3

6.1.4.8. public long getBundleId( )

- 返回 bundle 的惟一标识。当在 OSGi 环境中安装 bundle 时，框架会给 bundle 分配一个惟一的标识符。

  bundle 的标识符具有以下特征：
  - 惟一且持久性
  - 长整型
  - 其他 bundle 不能再次使用，即使在卸载 bundle 之后。
  - 当 bundle 还在安装时不会改变。
  - 当 bundle 更新时不会改变。

  对这个方法的调用将返回 bundle 的惟一标识符，即使 bundle 处于 UNINSTALLED 状态。

Returns 返回 bundle 的惟一标识符。

6.1.4.9. public URL getEntry(String name)

name 条目名称。参阅 java.lang.ClassLoader.getResource 来获得关于资源名称的更多描述格式信息。

- 返回 bundle 中指定名称的条目 URL。不会使用 bundle 的类加载器来查找指定的条目。只会对 bundle 的目录来查找指定条目。特殊路径“/”表示 bundle 根路径。

Returns 指定条目的 URL；或者如果没有找到指定条目则返回 null；或者如果 Java 运行环境支持权限控制而调用者没有管理权限：AdminPermission[this, RESOURCE]，那么也返回 null。

Throws IllegalStateException – 如果 bundle 已经卸载。

Since 1.3 specified

6.1.4.10.public Enumeration getEntryPaths(String path)

path 返回条目路径的路径名称

□ 返回 bunlde 中条目的最长子路径和参数提供的路径匹配的所有条目的路径枚举对象。不能使用 bundle 的类加载器来查找条目。只搜索 bundle 的目录。特殊路径“/”表示 bundle 根路径。
以“/”结尾的路径表示是一个子目录，返回的路径都和 bundle 根路径相关。

Returns 返回条目路径的枚举对象；如果没有找到匹配条目，返回 null 值；或者如果 Java 运行环境支持权限控制，而调用者没有合适的管理权限：AdminPermission[this,RESOURCE]，也返回 null 值。

Throws IllegalStateException – 如果 bundle 已经卸载了

Since 1.3

6.1.4.11.public Dictionary getHeaders( )

□ 返回 bundle 的清单文件的头标和值对。这个方法返回的是 bundle 清单文件中主要部分的所有的清单信息，也就是说从第一行到第一个空白行。
清单文件中的头标是不区分大小写的。方法返回的 Dictionary 对象中对于头标名称的处理也是不区分大小写的。如果头标名称的起始值为“%”，那么必须要使用默认的区域来本地化这个头标。
例如，对于下面的清单头标，如果它们出现在清单文件中，那么它们将出现在返回的 Dictionary 对象中：
Bundle-Name
Bundle-Vendor
Bundle-Version
Bundle-Description
Bundle-DocURL
Bundle-ContactAddress
如果 bundle 的状态是 UNINSTALLED，那么这个方法依然要返回 bundle 的清单文件信息。

Returns 返回一个包括了 bundle 清单头标和值对的 Dictionary 对象。

Throws SecurityException – 如果 Java 运行环境支持权限控制，而调用者没有适当的管理权限：AdminPermission[this,METADATA]

See Also Constants.BUNDLE_LOCALIZATION

6.1.4.12.public Dictionary getHeaders(String locale)

locale 头标的值需要进行本地化的区域。如果指定的区域为 null，那么使用 java.util.Locale.getDefault 返回的区域。如果如果指定的区域字符串是空串，那么这个方法将返回所有的以“%”开头的原始的（未本地化的）清单头标。

□ 返回通过指定区域进行本地化的 bundle 的头标。
这个方法和 Bundle.getHeaders()方法的处理是一样的，除了增加区域。如果清单头标的起始值为“%”，那么必须要通过指定区域进行本地化。如果不能

找到指定区域，那么那么使用默认的区域的头标的值。如果指定的区域为一个空字符串（""），那么返回没有本地化的值，返回所有的以“%”开始的头标的值。
如果 bundle 的状态是 UNINSTALLED，那么这个方法依然要返回 bundle 的清单文件信息。但是返回的值只能是原始的值和默认区域的本地化值。

Returns 包含 bundle 清单头标和值对的 Dictionary 对象

Throws SecurityException – 如果 Java 运行环境支持权限控制，而调用者没有适当的管理权限：AdminPermission[this,METADATA]

See Also getHeaders()[p.127] , Constants.BUNDLE_LOCALIZATION[p.154]

Since 1.3

6.1.4.13.public long getLastModified( )

- 返回 bundle 上次的最新更新时间。对 bundle 的修改包括安装、更新和卸载。时间值为自从 1 月 1 日, 1970, 00:00:00 GMT 开始的时间。

Returns bundle 上次更新的时间

Since 1.3

6.1.4.14.public String getLocation( )

- 返回 bundle 区域的标识符。
  bundle 区域标识符是当安装 bundle 时，发送给 BundleContext.installBundle 的区域。只要当 bundle 还是安装好的，那么 bundle 的区域标识符就不会改变，即使对 bundle 进行更新。
  如果 bundle 的状态是 UNINSTALLED，那么本方法必须还要能返回 bundle 的区域标识符。

Returns 描述 bundle 区域的标识符字符串。

Throws SecurityException – 如果 Java 运行环境支持权限控制，而调用者没有适当的管理权限：AdminPermission[this,METADATA]

6.1.4.15.public ServiceReference[] getRegisteredServices( )

- 返回 bundle 已经注册的所有 ServiceReference 对象数组，如果 bundle 没有注册服务，返回 null 值。
  如果 Java 运行环境支持权限，那么只有当调用者具有 ServicePermission 来使用至少一个注册的服务类获得服务，才能返回这个 ServiceReference 对象。
  在调用方法时返回的列表是有效的，然则框架是一个动态的环境，服务可能被随时修改或者取消注册。

Returns ServiceReference 对象数组或者 null 值。

Throws IllegalStateException – 如果 bundle 已经是卸载的。

See Also ServiceRegistration[p.179] , ServiceReference[p.177] , ServicePermission[p.176]

6.1.4.16.public URL getResource(String name)

name 资源名称。参阅 java.lang.ClassLoader.getResource 关于资源格式的描述。

- 从 bundle 中找到指定的资源。通过调用 bundle 类加载器来查找资源。如果 bundle 的状态是 INSTALLED，那么只会对这个 bundle 进行搜索，而由于还没有解析 bundle，故不会搜索导入包。如果指定 bundle 是一个片断（fragment），那么返回 null 值。

Returns 指定资源名称的 URL，或者如果不能找到指定资源，或者如果 bundle 是一个片断，或者如果 Java 运行环境支持权限而调用者没有管理权限：AdminPermission[this,RESOURCE]，那么返回 null 值。

Throws IllegalStateException – 如果 bundle 已经是卸载的。

Since 1.1

6.1.4.17.public Enumeration getResources(String name) throws IOException

name 资源名称。参阅 java.lang.ClassLoader.getResource 关于资源格式的描述。

- 从 bundle 中查找指定资源。在查找过程中调用 bundle 类加载器。如果 bundle 状态为 INSTALLED，那么只对 bundle 进行查找。由于 bundle 还没有解析，所以不查找导入包。如果这是一个片断 bundle，那么返回 null。

Returns 返回指定名称的资源 URL 的枚举对象，或者如果不能找到指定资源，或者如果 bundle 是一个片断，或者如果 Java 运行环境支持权限而调用者没有管理权限：AdminPermission[this,RESOURCE]，那么返回 null 值。

Throws IllegalStateException – 如果 bundle 已经是卸载的。
IOException – 如果发生 I/O 错误。

Since 1.3

6.1.4.18.public ServiceReference[] getServicesInUse( )

- 返回 bundle 使用的所有服务的 ServiceReference 对象数组，或者如果 bundle 没有使用任何服务，那么返回 null。如果 bundle 对一个服务的使用计数大于 0，那么就说 bundle 使用了这个服务。
  如果 Java 运行环境支持权限，那么只有当调用者具有 ServicePermission 权限来使用至少一个注册时的服务类获得服务，才能返回这个 ServiceReference 对象。
  在调用方法时返回的列表是有效的，然则框架是一个动态的环境，服务可能被随时修改或者取消注册。

Returns An array of ServiceReference objects or null.

Throws IllegalStateException – If this bundle has been uninstalled.

See Also ServiceReference[p.177] , ServicePermission[p.176]

6.1.4.19.public int getState( )

- 返回 bundle 当前状态。
  bundle 在任何时候都只能处于一种状态。

Returns UNINSTALLED，INSTALLED，RESOLVED，STARTING，STOPPING，ACTIVE 这六个值中的一个。

6.1.4.20.public String getSymbolicName( )

- 返回根据 Bundle-SymbolicName 指定的 bundle 的符号名称。这个名称是惟一的，推荐通过倒置域名的形式来命名，如同 java 包的命名一样。
  如果没有指定符号名称，那么返回 null。
  即使 bundle 的状态是 UNINSTALLED，那么本方法依然能返回 bundle 符号名称。

Returns bundle 的符号名称。

Since 1.3

6.1.4.21.public boolean hasPermission(Object permission)

permission 需要检查的权限。

- 检查 bundle 是否具有指定权限。
  如果 Java 运行环境不支持权限，那么始终返回 true。
  permission 的类型是 Object，这是为了避免直接引用 java.security.Permission 类。这样，允许框架可以在不支持 Java 权限的环境下实现。
  如果 Java 运行环境不支持权限，那么 bundle 和包含在 JAR 文件中的其他资源是属于同一个 java 安全域（java.security.ProtectionDomain）；也就是说，它们是共享相同的权限集合。

Returns 如果 bundle 有指定权限或者由 bundle 分配的权限蕴含了指定权限，那么返回 true，否则，如果没有指定权限，或者 permission 不是 instanceofjava.security.Permission 的实例。

Throws IllegalStateException – 如果已经卸载了 bundle。

6.1.4.22.public Class loadClass(String name) throws ClassNotFoundException

name 需要加载的类。

- 使用 bundle 的；类加载器来加载指定类。
  如果 bundle 为一个片断 bundle，那么方法抛出 ClassNotFoundException 异常。
  如果 bundle 状态为 INSTALLED，那么在加载类的时候会尝试解析 bundle。
  如果不能解析 bundle，那么框架发出一个包含 BundleException 的 FrameworkEvent.ERROR 事件，说明 bundle 不能解析的原因。同时方法抛出 ClassNotFoundException 异常。

如果 bundle 状态为 UNINSTALLED，那么方法抛出 IllegalStateException 异常。

Returns 请求类的类对象。

Throws ClassNotFoundException – 如果不能找到指定类，或者是一个片断 bundle，或者 Java 运行环境支持权限，而调用者没有相应的 AdminPermission[this, CLASS]权限。

IllegalStateException – 如果 bundle 已经被卸载。

Since 1.3

6.1.4.23.public void start( ) throws BundleException

- 启动 bundle。

  如果框架实现了可选的启动级别服务，而且框架当前的启动级别要小于 bundle 的启动级别，那么框架必须要持久标记 bundle 状态为 started，延迟到当框架的启动级别大于或者等于 bundle 的启动级别时，再启动 bundle。

  否则，按照以下步骤启动 bundle：

  1. 如果 bundle 的状态是 UNINSTALLED，那么抛出 IllegalStateException 异常。
  2. 如果 bundle 的状态为 STARTING 或者 STOPPING，那么必须要等到状态改变后再继续执行方法，如果不能在一个有效的时间内继续执行启动，那么方法抛出 BundleException 表示不能启动 bundle。
  3. 如果 bundle 的状态是 ACTIVE，那么方法立即返回。
  4. 框架持久标记已经启动 bundle。当框架重新启动后，必须要自动启动已经启动了的 bundle。
  5. 如果 bundle 的状态不是 RESOLVED，那么尝试解析 bundle 的包依赖。如果框架不能解析 bundle，抛出 BundleException 异常。
  6. 设置 bundle 状态为 STARTING。
  7. 发出 BundleEvent.STARTING 事件。这个事件只发送到 bundle 同步监听器 SynchronousBundleListeners，而不会发送到 BundleListeners。
  8. 如果指定了一个 BundleActivator，那么就调用它的 start 方法。如果没有指定或者是抛出了异常，那么将 bundle 状态设置为 RESOLVED。

     取消注册任何 bundle 注册的服务。

     释放 bundle 使用的任何服务。

     移除 bundle 注册的任何监听器。

     然后抛出 bundle 异常 BundleException。
  9. 如果 bundle 的状态是 UNINSTALLED，那么由于在 BundleActivator.start 运行中卸载了 bundle，那么抛出 BundleException。
  10. 设置 bundle 状态为 ACTIVE。
  11. 发出 bundle 事件 BundleEvent.STARTED。

  前置条件是：

  - getState() 方法返回值为{INSTALLED}，{ RESOLVED}。

  如果没有抛出异常, 后置条件：

  - bundle 的持久状态标记为活动（active）；
  - getState() 方法返回值为{ ACTIVE}；

- 调用了 BundleActivator.start()方法，而且没有抛出异常。

如果抛出了异常，后置条件：

- 依赖于异常抛出的时机，bundle 持久状态为活动（active）；
- getState()方法返回值不是{STARTING}或者{ ACTIVE}。

Throws BundleException – 如果不能启动 bundle。可能是由于不能解析代码依赖，或者是不能加载指定的启动器（BundleActivator），或者在加载中抛出了异常。
IllegalStateException – 如果已经卸载了 bundle 或者 bundle 试图修改自己的状态。
SecurityException – 如果 Java 运行环境支持权限而调用者没有相应的管理权限：AdminPermission[this,EXECUTE]。

6.1.4.24.public void stop( ) throws BundleException

□ 停止 bundle。

按照以下步骤进行：

1. 如果 bundle 的状态为 UNINSTALLED，那么抛出 IllegalStateException 异常。
2. 如果 bundle 的状态为 STARTING 或者 STOPPING，那么必须要等到状态改变后再继续执行方法，如果不能在一个有效的时间内继续执行停止方法，那么方法抛出 BundleException 表示不能停止 bundle。
3. 框架持久标记已经停止 bundle。当框架重新启动后，必须不能启动这个 bundle。
4. 如果 bundle 的状态不是 ACTIVE，那么方法立即返回。
5. 设置 bundle 状态为 STOPPING。
6. 发出 BundleEvent.STOPPING 事件。这个事件只发送到 bundle 同步监听器 SynchronousBundleListeners，而不会发送到 BundleListeners。
7. 如果指定了一个 BundleActivator，那么就调用它的 stop 方法。如果调用这个方法时抛出了异常，那么本方法继续进行停止 bundle 的过程，而且在执行完剩余代码之后抛出异常：BundleException。
8. 取消注册任何 bundle 注册的服务。
9. 释放 bundle 使用的任何服务。
10. 移除 bundle 注册的任何监听器。
11. 如果 bundle 的状态是 UNINSTALLED，那么由于在 BundleActivator. stop 运行中卸载了 bundle，那么抛出 BundleException。
12. 设置 bundle 状态为 RESOLVED。
13. 发出 bundle 事件 BundleEvent.STOPPED。

前置条件是：

- getState() 方法返回值为{ACTIVE}。

如果没有抛出异常, 后置条件：

- bundle 的持久状态标记为停止（stopped）；
- getState() 方法返回值不是{ACTIVE, STOPPING}；
- 调用了 BundleActivator.stop()方法，而且没有抛出异常。

如果抛出了异常，后置条件：

- bundle 的持久状态标记为停止（stopped）；

Throws BundleException – 如果不能加载 bundle 启动器，或者执行中抛出了异常。
IllegalStateException – 如果已经卸载了 bundle 或者 bundle 试图修改自己的状态。
SecurityException – 如果 Java 运行环境支持权限而调用者没有相应的管理权限：AdminPermission[this,EXECUTE]。

6.1.4.25.public void uninstall( ) throws BundleException

□ 卸载 bundle。
这个方法的调用会通知其他 bundle 已经卸载了这个 bundle，而且将 bundle 的状态设置为 UNINSTALLED。框架必须要移除所有可以移除的相关资源。如果这个 bundle 已经导出了包，框架必须要保证这些卸载后的 bundle 的导出包对导入的 bundle 还是可用的，直到重新启动框架或者是调用了方法 PackageAdmin.refreshPackages。
卸载过程步骤如下：

1. 如果 bundle 状态为 UNINSTALLED，那么抛出 IllegalStateException 异常。
2. 如果 bundle 状态为 ACTIVE，STARTING 或者 STOPPING，那么使用 Bundle.stop 方法来停止 bundle。如果 stop 方法抛出了异常，那么发出包含这个异常的框架事件：FrameworkEvent.ERROR。
3. 设置 bundle 状态为 UNINSTALLED。
4. 发出 bundle 事件 BundleEvent.UNINSTALLED。
5. bundle 和其他框架分配给 bundle 的固定存储区域由框架回收。

前置条件是：

- getState() 方法返回值不是{UNINSTALLED}。

如果没有抛出异常, 后置条件：

- getState() 方法返回值是{UNINSTALLED}
- 已经卸载了 bundle。

如果抛出了异常，后置条件：

- getState() 方法返回值不是{UNINSTALLED}
- 没有卸载 bundle。

Throws BundleException – 如果卸载失败。如果任何线程试图修改 bundle 状态，但是却没有及时返回会抛出这个异常。
IllegalStateException – 如果已经卸载了 bundle 或者 bundle 试图修改自己的状态。
SecurityException – 如果 Java 运行环境支持权限而调用者没有相应的管理权限：AdminPermission[this, LIFECYCLE]。

6.1.4.26.public void update( ) throws BundleException

□ 更新 bundle。
如果 bundle 状态为 ACTIVE，那么必须要停止 bundle 而且等更新完毕之后再启动 bundle。如果更新的 bundle 导出了包，那么不能对这些导出包进行更新。

直到重新启动框架或者是调用了方法 PackageAdmin.refreshPackages，这些导出包都维持原来的版本。

更新过程步骤如下：

1. 如果 bundle 状态为 UNINSTALLED，那么抛出 IllegalStateException 异常。
2. 如果 bundle 状态为 ACTIVE，STARTING 或者 STOPPING，那么使用 Bundle.stop 方法来停止 bundle。如果 stop 方法抛出了异常，那么重新抛出这个异常来终止更新过程。
3. 根据 bundle 的 Constants.BUNDLE_UPDATELOCATION 描述的位置信息（如果有提供），或者是 bundle 的原始位置信息来下载新的版本。
4. 根据框架的实现来翻译位置信息，通常为 URL，然后从这个位置获得新的版本。
5. 安装 bundle 的新版本。如果框架不能安装 bundle 的新版本，那么保留原来的版本，而且在完成剩下步骤之后抛出 BundleException 异常。
6. 如果 bundle 声明了 Bundle-RequiredExecutionEnvironment 头标，那么将执行环境和列出的执行环境进行比较。如果不是全部匹配，那么保留原来版本，而且在完成剩下步骤之后抛出 BundleException 异常。
7. 设置 bundle 状态为 INSTALLED。
8. 如果成功安装了 bundle 的新版本，那么发出 BundleEvent.UPDATED bundle 事件。
9. 如果 bundle 最初状态为 ACTIVE，那么按照 Bundle.start 所描述的那样启动 bundle。如果 Bundle.start 方法抛出了异常，那么发出一个包含了异常的框架事件：FrameworkEvent.ERROR。

前置条件是：

- getState() 方法返回值不在{ UNINSTALLED }中

如果没有抛出异常, 后置条件：

- getState() 方法返回值在{ INSTALLED, RESOLVED,ACTIVE }中
- 已经更新了 bundle。

如果抛出了异常，后置条件：

- getState() 方法返回值在{ INSTALLED, RESOLVED,ACTIVE }中
- 没有更新 bundle。

Throws BundleException – 如果更新失败。如果任何线程试图修改 bundle 状态，但是却没有及时返回会抛出这个异常。

IllegalStateException – 如果已经卸载 bundle 或者 bundle 试图修改自己状态。

SecurityException – 如果 Java 运行环境支持权限而调用者没有对当前 bundle 和更新后 bundle 相应的管理权限：AdminPermission[this, LIFECYCLE]。

See Also stop()[p.132] , start()[p.131]

6.1.4.27.public void update(InputStream in) throws BundleException

in 读入新 bundle 的输入流（InputStream）

□ 从一个输入流更新 bundle。

按照 Bundle.update()方法中描述的步骤来进行，除了读取的是从一个提供的输入流而不是 URL。方法执行完毕之后一定要关闭输入流，即使执行抛出了

异常。

Throws BundleException – 如果更新失败或者是不能读取输入流。
IllegalStateException – 如果已经卸载 bundle 或者 bundle 试图修改自己状态。
SecurityException – 如果 Java 运行环境支持权限而调用者没有对当前 bundle 和更新后 bundle 相应的管理权限：AdminPermission[this, LIFECYCLE]。

See Also update()[p.134]

6.1.5. public interface BundleActivator

对 bundle 的启动和停止进行定制。

BundleActivator 是一个当 bundle 启动或者停止时实现的接口。如果需要，框架可以创建 bundle 的 BundleActivator。如果 BundleActivator 的实例的 start 方法成功执行，那么就可以保证同一个实例的 stop 方法将在停止 bundle 的时候进行调用。

通过头标 Bundle-Activator 来指定 BundleActivator。

在 bundle 清单文件中只能指定一个 BundleActivator。而且片断 bundle 是不能有 BundleActivator 的。头标格式如下：

Bundle-Activator: class-name

其中 class-name 是全名的 Java 类名称。

指定的 BundleActivator 必须要有一个公有的不带参数的构造方法，这样就可以通过 Class.newInstance()来创建 BundleActivator 实例。

6.1.5.1. public void start(BundleContext context) throws Exception

context 待启动 bundle 的执行上下文

- bundle 启动时调用，这样，框架就可以执行 bundle 指定的动作来启动 bundle。可以用于注册服务，或者是分配 bundle 需要的资源。
  本方法必须要执行完成，并且在允许时间内返回。

Throws Exception – 如果方法抛出了异常，那么标记 bundle 为 stopped，然后框架将移除 bundle 的监听器，取消注册 bundle 所注册的所有服务，并且释放 bundle 使用的所有服务。

See Also Bundle.start[p.131]

6.1.5.2. public void stop(BundleContext context) throws Exception

context 待停止 bundle 的执行上下文

- 当停止 bundle 时调用，这样框架就运行 bundle 指定的动作来停止 bundle。通常，方法执行的是 start 方法的逆操作。bundle 返回后，应该没有由这个 bundle 创建的活动状态的线程。一个停止的 bundle 不能调用框架的任何对象。
  本方法必须要执行完成，并且在允许时间内返回。

Throws Exception – 如果方法抛出了异常，那么标记 bundle 为 stopped，然后框架将移除 bundle 的监听器，取消注册 bundle 所注册的所有服务，并且释放 bundle 使用的所有服务。

See Also Bundle.stop[p.132]

6.1.6. public interface BundleContext

框架内的 bundle 执行上下文。上下文用于授予其他方法和框架进行交互的访问途径。

通过使用 bundel 上下文（BundleContext）允许 bundle：

- 预定框架事件。
- 在框架服务注册中心注册服务对象。
- 从框架服务注册中心检索服务引用（ServiceReferences）。
- 从服务引用对象获取并释放服务对象。
- 在框架中安装新的 bundle。
- 获取框架中安装的 bundle 列表。
- 从 bundle 中获得 Bundle 对象。
- 从框架为 bundle 提供的持久存储区中为文件创建 File 对象。

当使用 BundleActivator.start 方法来启动 bundle 时，创建一个 BundleContext 对象关联到这个 bundle。而当使用 BundleActivator.stop 方法来停止一个 bundle 时，传递和创建时同样的一个 BundleContext 对象来关联上下文。通常，BundleContext 都是 bundle 私有的，也就是说在 OSGi 环境下，不能在 bundle 之间共享 BundleContext 对象。

和对象 BundleContext 关联的 Bundle 对象称之为上下文 bundle。

只有在执行 BundleContext 的上下文 bundle 时，BundleContext 对象才是有效的；也就是说，当上下文 bundle 处于 STARTING、STOPPING 和 ACTIVE 状态时。如果随后使用了 BundleContext 对象，那么必须抛出 IllegalStateException 异常。而在 BundleContext 的上下文 bundle 停止之后，不能再使用这个 BundleContext 对象。

框架是创建 BundleContext 对象的惟一来源，也只在创建 BundleContext 对象的框架内有效。

6.1.6.1. public void addBundleListener(BundleListener listener)

listener 需要添加的监听器（BundleListener）

- 如果指定的 BundleListener 对象在上下文 bundle 中不存在，那么将它添加到上下文 bundle 的监听器列表中。当 bundle 的生命周期状态改变之后，监听器就可以获得通知。

  如果上下文 bundle 中已经包含了监听器 l（l==listener），那么本方法不做任何处理。

Throws IllegalStateException – 如果这个 BundleContext 不再有效。

SecurityException – 如果这个监听器是一个同步 bundle 监听器（SynchronousBundleListener），而且 Java 运行环境支持权限但是调用者没有相应的管理权限：AdminPermission[context bundle,LISTENER]

See Also BundleEvent[p.148] , BundleListener[p.150]

6.1.6.2. public void addFrameworkListener(FrameworkListener listener)

listener 待添加的框架监听器（FrameworkListener）对象

- 如果指定的 FrameworkListener 对象在上下文 bundle 中不存在，那么将它添加到上下文 bundle 的监听器列表中。如果发生了框架事件，监听器就可以获

得通知。
如果上下文 bundle 中已经包含了监听器 l（l==listener），那么本方法不做任何处理。

Throws IllegalStateException – 如果这个 BundleContext 不再有效。

See Also FrameworkEvent[p.168] , FrameworkListener[p.170]

6.1.6.3. public void addServiceListener(ServiceListener listener, String filter) throws InvalidSyntaxException

listener 待添加的服务监听（ServiceListener）对象

filter 过滤条件

□ 将带有指定过滤器的指定的服务监听对象添加到上下文 bundle 的监听器列表中。关于过滤语法可以参考 Filter 一节。当一个服务的生命周期发生改变之后，服务监听器对象可以获得通知。
如果上下文 bundle 中已经包含了监听器 l（l==listener），那么本方法将这个监听器 l 的过滤器用参数中指定的过滤器替代。
如果满足过滤器条件，那么就调用监听器。为了在服务类的基础上过滤，过滤器应该参考 Constants.OBJECTCLASS 属性。如果过滤器为 null，那么可以认为所有的服务都匹配过滤器。
使用过滤器时，有可能服务的整个生命周期服务事件都不会通知监听器。例如，如果过滤器之匹配 x 属性的值为 1，如果服务注册时，x 属性的值没有设置为 1。随后，当将 x 属性的值修改为 1，那么过滤器就可以匹配了，如果发生了类型为 MODIFIED 的服务事件，将调用这个监听器。但是对于类型为 REGISTERED 的服务事件，将不会调用监听器。
如果 Java 运行环境支持权限，而只有当注册服务的 bundle 具有服务权限使用至少一个注册的服务类来获得服务，服务监听器对象才会获得服务事件通知。

Throws InvalidSyntaxException – 如果过滤器包含了无法处理的过滤器字符串。
IllegalStateException – 如果上下文 bundle 不再有效。

See Also ServiceEvent[p.173]，ServiceListener[p.176]，ServicePermission[p.176]

6.1.6.4. public void addServiceListener(ServiceListener listener)

listener 待添加的服务监听器对象（ServiceListener）

□ 将指定的服务监听对象添加到上下文 bundle 的监听器列表中。
这个方法和调用如下方法：

BundleContext.addServiceListener(ServiceListener listener, String filter)
其中 filter 为 null

的结果是相同的。

Throws IllegalStateException – 如果上下文 bundle 不再有效。

See Also addServiceListener(ServiceListener, String)[p.138]

6.1.6.5. public Filter createFilter(String filter) throws InvalidSyntaxException

filter 过滤器字符串

□ 创建一个 Filter 对象。使用这个 Filter 对象来匹配 ServiceReference 对象或者是 Dictionary 对象。

如果不能分析这个过滤器对象，那么抛出 InvalidSyntaxException 异常，并且带有可读的信息来描述哪一部分是不可读的。

Returns 对过滤字符串进行压缩包装的 Filter 对象。

Throws InvalidSyntaxException – 如果过滤字符串中包含了不能分析的过滤器字符串。

NullPointerException – 如果过滤器为 null。

IllegalStateException – 如果指定的 BundleContext 不再有效。

See Also Framework specification for a description of the filter string syntax., FrameworkUtil.createFilter(String)[p.170]

6.1.6.6. public ServiceReference[] getAllServiceReferences(String clazz, String filter) throws InvalidSyntaxException

clazz 服务注册的类名称，或者是所有的服务。

filter 过滤条件

□ 返回一个服务引用对象（ServiceReference）数组。返回的服务引用数组中包含了符合条件的服务，即匹配指定的过滤器，并且是在参数中指定的类下注册的服务。

调用方法时返回的结果是有效的，但是由于框架是一个的动态的环境，服务任何时候都有可能改变或者是取消注册。

过滤器是用来选择注册服务，那些注册服务包含了和过滤器匹配的属性键值对。关于过滤器语法参阅 Filter 一节。

如果过滤器为 null，那么表示匹配所有的注册服务。如果不能解析过滤字符串，那么抛出一个 InvalidSyntaxException 异常，其中包含了对那么不能解析部分的可读信息。

按照如下步骤来选择 ServiceReference 对象：

1. 如果过滤字符串不为 null，那么分析过滤字符串，然后得到了满足过滤器的 ServiceReference 对象集合。
2. 如果 Java 运行环境支持权限控制，那么第一步中生成的集合将由于对权限的检查而缩减，如果调用者没有权限获得至少一个注册服务的类，那么将从集合中删除这个 ServiceReference 对象；如果对于某一个 ServiceReference 对象，调用者没有正确的权限，那么删除这个对象。
3. 如果参数中 clazz 的值为非空，那么进一步检查集合中引用的服务是否注册了指定类。可以通过服务的 Constants.OBJECTCLASS 属性来获得所有的服务接口类列表和服务注册类列表。
4. 返回剩下的 ServiceReference 对象数组。

Returns ServiceReference 对象数组

或者如果没有注册服务，返回 null

Throws InvalidSyntaxException – 如果过滤器包含不能分析的不合法的过滤字符串。
IllegalStateException – 如果 BundleContext 对象已经不再有效。
Since 1.3

6.1.6.7. public Bundle getBundle( )

- 返回 BundleContext 关联的 Bundle 对象，这个 bundle 称之为上下文 bundle。

Returns 和 BundleContext 关联的 Bundle 对象。
Throws IllegalStateException – 如果 BundleContext 对象已经不再有效。

6.1.6.8. public Bundle getBundle(long id)

id 待获得的 bundle 的标识符

- 返回指定标识符的 bundle。

Returns 一个 Bundle 对象；
返回 null，如果指定标识符不能匹配任何安装的 bundle。

6.1.6.9. public Bundle[] getBundle( )

- 返回所有的已经安装的 bundle。
  这个方法返回调用方法时安装在 OSGi 环境下的所有的 bundle 列表。但是，由于框架是一个动态的环境，任何时候都有可能安装或者是卸载 bundle。

Returns Bundle 对象数组，对于每一个安装的 bundle，对应一个 Bundle 对象。

6.1.6.10.public File getDataFile(String filename)

filename 访问的文件的相对路径名。

- 在框架为 bundle 提供的持久存储区中创建一个指定文件名称的 File 对象。如果平台不支持这种文件系统，那么返回 null。
  如果使用一个空串作为文件名参数来调用这个方法，那么可以获得一个框架为 bundle 提供存储区的根目录的 File 对象。
  如果 Java 运行环境支持权限控制，那么框架要确保 bundle 具有文件权限（java.io.FilePermission）来读取、写入、删除所有的上下文 bundle 的持久存储区中的文件（递归）。

Returns 表示请求文件的 File 对象，
或者如果平台不支持文件系统，返回 null。
Throws IllegalStateException – 如果 BundleContext 对象已经不再有效。

6.1.6.11.public String getProperty(String key)

key 请求的属性名称

- 返回指定属性的值。如果没有在框架中找到指定的属性，那么查找系统属性，

如果没有找到属性，返回 null。
框架定义了如下标准属性键：

- Constants.FRAMEWORK_VERSION[p.161] — OSGi 框架版本。
- Constants.FRAMEWORK_VENDOR[p.161] — 框架实现者。
- Constants.FRAMEWORK_LANGUAGE[p.160] — 使用的语言。参阅 ISO 639 的标准。
- Constants.FRAMEWORK_OS_NAME[p.160] —主服务器操作系统名称。
- Constants.FRAMEWORK_OS_VERSION[p.160] — 主服务器操作系统版本号。
- Constants.FRAMEWORK_PROCESSOR[p.160] — 主服务器处理器名称。

所有的 bundle 必须要有权限读取这些属性。
注意：最后的四个标准属性是用于 Constants.BUNDLE_NATIVECODE 清单头标选择本地语言代码的匹配算法。

Returns 请求的属性值；
或者如果没有定义这个属性，则返回 null。

Throws SecurityException – 如果 Java 运行环境支持权限控制而调用者没有相应的权限来读取这个属性。

6.1.6.12.public Object getService(ServiceReference reference)

reference 服务的一个引用（ServiceReference）

□ 返回指定服务的一个服务对象。
bundle 对服务的使用通过 bundle 对服务的使用计数来跟踪。每当通过这个方法 getService(ServiceReference)获得了一个对某个服务的服务对象，那么上下文 bundle 对该服务的使用计数加 1。
如果 bundle 对某个服务的使用计数降为 0，那么表示 bundle 不再使用这个服务。
通常如果 ServiceReference 关联的服务已经取消注册了，那么方法返回 null。
通过以下步骤来获得服务对象：

1. 如果已经取消注册服务，那么返回 null。
2. 上下文 bundle 对这个服务的使用计数加 1。
3. 如果上下文 bundle 对服务的使用计数为 1，而且服务注册了一个实现服务工厂（ServiceFactory）的类，那么调用 ServiceFactory. getService(Bundle, ServiceRegistration)方法来为上下文 bundle 创建一个服务对象。框架将缓存这个服务对象。当上下文 bundle 对服务使用计数大于 0 时，接下来对本方法的调用将返回这个缓存的服务对象。
   如果服务工厂对象返回的服务对象不是注册的服务类的实例对象，或者服务工厂对象中抛出了异常，那么返回 null，而且发出框架事件 FrameworkEvent.ERROR。
4. 返回获得的服务对象。

Returns 和指定 ServiceReference 关联的服务对象；
如果服务没有注册，或者如果使用服务工厂而没有实现服务注册类，那么返回 null 值。

Throws SecurityException – 如果 Java 运行环境支持权限，而调用者没有使用至少一个服务注册类获得服务的权限。
IllegalStateException – 如果 BundleContext 对象已经不再有效。

See Also ungetService(ServiceReference)[p.147] , ServiceFactory[p.175]

6.1.6.13.public ServiceReference getServiceReference(String clazz)

clazz 服务注册的类名称

□ 返回实现并且注册了指定类的服务的一个引用对象（ServiceReference）。
当调用方法时，ServiceReference 对象有效。但是由于框架是一个动态环境，服务随时都可能修改或取消注册，那么返回的 ServiceReference 对象不一定一直有效。
这个方法等同于使用一个 null 值的过滤字符串来调用如下方法：BundleContext.getServiceReferences(String, String)，这是为了方便调用者只对实现了指定类的服务感兴趣的情况。
如果存在多个这样的服务，那么返回具有最高等级的服务（在 Constants.SERVICE_RANKING 中指定）；也就是说，返回最先注册的服务。

Returns ServiceReference 对象；
如果没有任何注册服务实现了指定类，返回 null。

Throws IllegalStateException – 如果 BundleContext 对象已经不再有效。

See Also getServiceReferences(String, String)[p.143]

6.1.6.14.public ServiceReference[] getServiceReferences(String clazz, String filter) throws InvalidSyntaxException

clazz 服务注册的类名称，或者 null 表示所有的服务

filter 过滤器条件

□ 返回 ServiceReference 对象数组。返回的对象都是注册了指定类，符合过滤条件，而且服务注册的类名所在包要和上下文 bundle 中通过 ServiceReference.isAssignableTo(Bundle, String)定义的包匹配。
当调用方法时，ServiceReference 对象有效。但是由于框架是一个动态环境，服务随时都可能修改或取消注册，那么返回的 ServiceReference 对象不一定一直有效。
过滤器用于选择注册服务，选择的服务的属性对象中包含的键值对和过滤器匹配。参阅 Filter 一节中过滤器语法规则描述。
如果过滤器字符串为 null，那么匹配所有的注册服务。如果不能分析过滤字符串，那么抛出 InvalidSyntaxException 异常，其中包含了过滤字符串不能分析的原因。
按照以下步骤选择 ServiceReference 对象集合：

1. 如果过滤字符串不为 null，那么分析过滤字符串对象，产生一个满足过滤条件的 ServiceReference 对象集合。
   如果过滤字符串为 null，那么可以认为所有的注册服务都满足过滤器。
2. 如果 Java 运行环境支持权限控制，那么通过权限检查来缩减这个集合。

如果调用者没有服务权限来获得至少一个注册服务类，那么将移除这个 ServiceReference 对象。

3. 如果 clazz 不为 null，那么将集合缩减到指定类的的实例并且注册了指定类的服务。可以通过服务的 Constants.OBJECTCLASS 属性来获得所有的服务接口类列表和服务注册类列表。
4. 最后对集合中的所有 ServiceReference 对象作最后一次检查。对于每一个 ServiceReference 对象，使用这个上下文 bundle 和 ServiceReference 对象注册的每一个类，调用 ServiceReference.isAssignableTo(Bundle, String)方法，如果任何调用返回 false，那么从集合中移除这个 ServiceReference 对象。
5. 返回剩下的 ServiceReference 对象数组。

Returns ServiceReference 对象数组；
如果没有找到满足条件的服务，返回 null。

Throws InvalidSyntaxException – 如果过滤器包含不能分析的不合法的过滤字符串。
IllegalStateException – 如果 BundleContext 对象已经不再有效。

6.1.6.15.public Bundle installBundle(String location) throws BundleException

location bundle 待安装的位置的标识符

□ 从指定位置安装 bundle。框架将位置字符串翻译为一个与实现无关的位置信息，从这个位置获得 bundle。

每一个安装的 bundle 都可以通过位置字符串来惟一标识，例如，URL 的形式。

按照如下步骤安装 bundle：

1. 如果已经安装了同样位置字符串标识的 bundle，那么返回已经安装的 Bundle 对象。
2. 从位置字符串标识的位置读取 bundle，如果读取失败，抛出 bundle 异常（BundleException）。
3. 解析 bundle 的 Bundle-NativeCode 依赖，如果失败，抛出 bundle 异常（BundleException）。
4. 分配 bundle 关联的资源。分配最低限度的资源，由一个惟一标识符和一块持久存储区域（如果平台支持文件系统）组成。如果本步骤失败，那么抛出 bundle 异常（BundleException）。
5. 如果 bundle 已经声明了 Bundle-RequiredExecutionEnvironment 头标，那么根据列出运行环境对安装的运行环境进行检测。如果当前运行环境没有在其中列出，那么抛出 bundle 异常（BundleException）。
6. 设置 bundle 状态为 INSTALLED。
7. 发出 bundle 事件：BundleEvent.INSTALLED。
8. 返回最新或者原来安装的 bundle 的 Bundle 对象。

后置条件，没有抛出异常：

- getState() 返回值在集合{INSTALLED,RESOLVED}中。
- Bundle 对象具有一个惟一 ID

后置条件，抛出异常：

- 没有安装 bundle，而且没有 bundle 存在的记录。

Returns 安装的 bundle 的 Bundle 对象。

Throws BundleException – 如果安装失败。

SecurityException – 如果 Java 运行环境支持权限控制而调用者没有合适的管理权限：AdminPermission[installed bundle,LIFECYCLE]。

IllegalStateException – 如果 BundleContext 不再有效。

6.1.6.16.public Bundle installBundle(String location, InputStream input) throws BundleException

location 待安装的 bundle 的位置标识符

input bundle 读取的输入流对象（InputStream）

□ 从指定输入流中安装 bundle。

本方法需要执行 BundleContext.installBundle(String location)中列出的所有步骤，另外本方法需要从输入流中读取 bundle 内容。位置标识符字符串指定了 bundle 的标识符。

本方法必须要关闭输入流，即使抛出了异常。

Returns 安装的 Bundle 对象

Throws BundleException – 如果安装失败或者是不能读取提供的输入流。

SecurityException – 如果 Java 运行环境支持权限控制而调用者没有合适的管理权限：AdminPermission[installed bundle,LIFECYCLE]。

IllegalStateException – 如果 BundleContext 不再有效。

See Also installBundle(java.lang.String)[p.144]

6.1.6.17.public ServiceRegistration registerService(String[] clazzes, Object service, Dictionary properties)

clazzes 服务可以定位的类名称。在这个数组中的类名称存储在服务属性中的键 Constants.OBJECTCLASS 之下

service 服务对象或者是服务工厂对象

properties 服务的属性。属性对象中的键必须是字符串对象。参阅常量（Constants）一节中描述的标准属性键。调用本方法之后，不能改变这个对象。可以通过调用如下方法来修改服务的属性值：ServiceRegistration.setProperties。如果服务没有属性值，那么这个集合可能为 null。

□ 通过指定属性，在框架中使用指定类来注册指定的服务对象。返回一个 ServiceRegistration 对象。这个 ServiceRegistration 对象是 bundle 服务注册时私有的，而且不能在 bundle 之间共享。注册的 bundle 即为当前上下文 bundle。其他的 bundle 可以通过 getServiceReferences 方法或者 getServiceReference 方法来定位服务。

也可以注册实现了 ServiceFactory 接口的服务对象，这样为其他 bundle 提供服务对象的方式更加灵活。

注册服务时按照如下步骤进行：

1. 如果注册的服务不是服务工厂（ServiceFactory），那么如果服务不是所

有命名的类的接口，那么抛出异常 IllegalArgumentException。

2. 框架将服务属性添加到指定的 Dictionary 中（可能为 null）：属性名称为 Constants.SERVICE_ID 的属性描述服务的注册编号，属性名称为 Constants.OBJECTCLASS 的属性包含了所有的指定类。如果注册 bundle 已经定义了这些属性，那么这些属性的值将被框架覆盖。
3. 将服务添加到框架服务注册中心，现在其他 bundle 就可以使用服务。
4. 发出服务事件：ServiceEvent.REGISTERED[p.174]。
5. 返回一个 ServiceRegistration 对象。

Returns 一个 ServiceRegistration 对象，注册服务的 bundle 可以使用这个对象来更新服务属性或者取消注册服务。

Throws IllegalArgumentException – 如果满足以下条件中的一个：参数 service 为 null 值；参数 service 不是一个服务工厂类，而且不是在 clazzes 参数中指定的所有类的实例；参数 properties 中包含了同名的属性。

SecurityException –如果 Java 运行环境支持权限控制而调用者没有服务权限（ServicePermission）来注册所有命名类的服务。

IllegalStateException –如果 BundleContext 不再有效。

See Also ServiceRegistration[p.179] , ServiceFactory[p.175]

6.1.6.18.public ServiceRegistration registerService(String clazz, Object service,Dictionary properties)

clazz 服务可以定位的类名

service 服务对象或者是服务工厂对象

properties 服务的属性

□ 通过指定属性，在框架中使用指定类来注册指定的服务对象。

这个方法等同于上文中的 registerService(java.lang.String[],java.lang.Object, java.util.Dictionary)[p.145]方法在使用一个类来注册的情况是等同的。在这种情况下，服务的 Constants.OBJECTCLASS[p.162]属性是一个字符串数组，而不仅仅是一个单独字符串。

Returns 一个 ServiceRegistration 对象，注册服务的 bundle 可以使用这个对象来更新服务属性或者取消注册服务。

Throws IllegalStateException –如果 BundleContext 不再有效。

See Also registerService(java.lang.String[], java.lang.Object,java.util.Dictionary)[p.145]

6.1.6.19.public void removeBundleListener(BundleListener listener)

listener 待删除的 bundle 监听器对象（BundleListener）

□ 从上下文 bundle 的监听器列表中移除指定的 BundleListener 对象。

如果在上下文 bundle 的监听器列表中没有这个监听器，那么方法将不作任何操作。

Throws IllegalStateException – 如果 BundleContext 不再是有效的。

SecurityException – 如果监听器是同步的，而且 Java 运行环境支持权限控制而调用者没有管理权限： AdminPermission[context bundle, LISTENER]。

6.1.6.20.public void removeFrameworkListener(FrameworkListener listener)

listener 待删除的框架监听器对象（FrameworkListener）

- 从上下文 bundle 的监听器列表中移除指定的 FrameworkListener 对象。
  如果在上下文 bundle 的监听器列表中没有这个监听器，那么方法将不作任何操作。

Throws IllegalStateException – 如果 BundleContext 不再是有效的。

6.1.6.21.public void removeServiceListener(ServiceListener listener)

listener 待删除的服务监听器对象（ServiceListener）

- 从上下文 bundle 的监听器列表中移除指定的 ServiceListener 对象。
  如果在上下文 bundle 的监听器列表中没有这个监听器，那么方法将不作任何操作。

Throws IllegalStateException – 如果 BundleContext 不再是有效的。

6.1.6.22.public boolean ungetService(ServiceReference reference)

reference 待释放的服务引用

- 释放指定的服务引用所引用的服务对象。如果上下文 bundle 对这个服务的使用计数为 0，那么这个方法返回 false，否则，将上下文 bundle 对服务的使用计数减 1。
  应该不再使用服务的 service 对象，而且如果当服务的引用计数减为 0 时，销毁这个服务的所有引用。按照如下步骤来释放服务对象：
  1. 如果上下文 bundle 对服务的使用计数为 0，或者已经取消注册了服务，那么返回 false。
  2. 将上下文 bundle 对这个服务的使用计数减 1。
  3. 如果上下文 bundle 对这个服务的使用计数在执行第二步后为 0，而且注册的是一个服务工厂对象，那么调用服务工厂的服务释放方法：ServiceFactory.ungetService(Bundle, ServiceRegistration, Object)[p.175]。
  4. 返回 true。

Returns 如果上下文 bundle 对服务的使用计数为 0，或者已经取消注册了服务，那么返回 false；
否则返回 true。

Throws IllegalStateException – 如果 BundleContext 不再是有效的。

6.1.7. public class BundleEvent extends EventObject

框架描述的 bundle 生命周期改变的事件。

当在 bundle 的生命周期内发生了改变，那么发出一个 BundleEvent 对象。为了将来扩展，定义了类型代码来标识这种事件类型。

OSGi 联盟保留对事件类型的扩充权力。

6.1.7.1. public static final int INSTALLED = 1

表示 bundle 已经安装。
INSTALLED 值为 0x00000001。

See Also BundleContext.installBundle(String)[p.144]

6.1.7.2. public static final int RESOLVED = 32

表示 bundle 已经解析。
RESOLVED 值为 0x00000020。

See Also Bundle.RESOLVED[p.124]

Since 1.3

6.1.7.3. public static final int STARTED = 2

表示 bundle 已经启动。
STARTED 值为 0x00000002。

See Also Bundle.start[p.131]

6.1.7.4. public static final int STARTING = 128

表示 bundle 即将启动。
STARTING 值为 0x00000080。

See Also Bundle.start()[p.131]

Since 1.3

6.1.7.5. public static final int STOPPED = 4

表示 bundle 已经停止。
STOPPED 值为 0x00000004。

See Also Bundle.stop[p.132]

6.1.7.6. public static final int STOPPING = 256

表示 bundle 即将停止。
STOPPING 值为 0x00000100。

See Also Bundle.stop()[p.132]

Since 1.3

6.1.7.7. public static final int UNINSTALLED = 16

表示 bundle 已经卸载。
UNINSTALLED 值为 0x00000010。

See Also Bundle.uninstall[p.133]

6.1.7.8. public static final int UNRESOLVED = 64

表示 bundle 已经未解析。
UNRESOLVED 值为 0x00000040。

See Also Bundle.INSTALLED[p.124]

Since 1.3

6.1.7.9. public static final int UPDATED = 8

表示 bundle 已经更新。
UPDATED 值为 0x00000008。

See Also Bundle.update()[p.134]

6.1.7.10.public BundleEvent(int type, Bundle bundle)

type 事件类型

bundle 发生了生命周期改变的 bundle

□ 创建一个指定类型的 bundle 事件。

6.1.7.11.public Bundle getBundle( )

□ 返回发生了生命周期改变的 bundle。是事件的发生源。

Returns 生命周期发生改变了的 bundle。

6.1.7.12.public int getType( )

□ 返回生命周期事件类型。类型如下：

- INSTALLED[p.148]
- RESOLVED[p.148]
- STARTING[p.148]
- STARTED[p.148]
- STOPPING[p.149]
- STOPPED[p.148]
- UPDATED[p.149]
- UNRESOLVED[p.149]
- UNINSTALLED[p.149]

Returns 生命周期事件的类型。

6.1.8. public class BundleException extends Exception

用于指出发生了 bundle 生命周期问题的框架异常。

由框架来创建 BundleException 对象，用于标识在 bundle 生命周期中的异常状况。BundleExceptions 不应该由 bundle 开发人员来创建。

为了和一般的异常链机制一致，可以对异常进行更新。

6.1.8.1. public BundleException(String msg, Throwable cause)

msg 相关消息

cause 异常的原因

- 创建一个封装了另一个异常的 BundleExceptions 对象。

6.1.8.2. public BundleException(String msg)

msg 相关消息

- 创建一个指定了消息的 BundleExceptions 异常对象。

6.1.8.3. public Throwable getCause( )

- 返回异常的原因，或者如果创建时没有指定原因，返回 null。

Returns 异常的原因，或者如果没有指定原因，返回 null。

Since 1.3

6.1.8.4. public Throwable getNestedException( )

- 返回异常中任何嵌套的异常。
  这个方法要比一般意义下的异常链机制产生时间早。现在更推荐使用 getCause()方法来获得信息。

Returns 嵌套的异常，或者如果没有嵌套异常，返回 null。

6.1.8.5. public Throwable initCause(Throwable cause)

cause 异常的原因

- 只有当构造的时候才可以设置异常的原因。

Returns 异常原因

Throws IllegalStateException – 通称这个方法将抛出异常，因为只有在构造的时候才可以设置原因。

Since 1.3

6.1.9. public interface BundleListener extends EventListener

bundle 事件监听器。当发生了 BundleEvent，将这个 BundleEvent 事件异步发送到 bundle 监听器（BundleListener）。

BundleListener 是一个监听器接口，bundle 开发人员可以实现这个接口。

框架可以使用 BundleContext.addBundleListener[p.137] 方法来注册 BundleListener 对象。当安装、解析、启动、停止、更新、取消解析或者卸载 bundle 时，发生了 Bundle 事件，这时就会使用这个事件来调用 BundleListener。

参阅：BundleEvent[p.148]。

6.1.9.1. public void bundleChanged(BundleEvent event)

event BundleEvent

□ 收到 bundle 发生了生命周期改变的通知。

6.1.10. public final class BundlePermission extends BasicPermission

需要或者是提供一个 bundle 的权限，或者是接收或者附加一个片断的 bundle 权限。

bundle 的符号名称定义了一个惟一的完全资格的名称。

例如：

org.osgi.example.bundle

BundlePermission 权限有四种操作：PROVIDE，REQUIRE，HOST 和 FRAGMENT。

Since 1.3

6.1.10.1.public static final String FRAGMENT = “fragment”

操作“fragment”

6.1.10.2.public static final String HOST = “host”

操作“host”

6.1.10.3.public static final String PROVIDE = “provide”

操作“provide”

6.1.10.4.public static final String REQUIRE = “require”

操作“require”

6.1.10.5.public BundlePermission(String symbolicName, String actions)

symbolicName bundle 符号名称

actions PROVIDE，REQUIRE，HOST，FRAGMENT（规范排序）

- 定义了提供而且/或者请求一个在 OSGi 框架中指定的符号名称的片断的权限。

  将 bundle 的权限授予给 bundle 的所有的版本。需要提供 bundle 的一个 bundle 必须要具有相应的对符号名称的 BundlePermission；而如果一个 bundle 请求其他 bundle，则必须要有相应的对符号名称的 BundlePermission；指定了片断附主的 bundle 必须要有相应的对符号名称的 BundlePermission。

6.1.10.6.public boolean equals(Object obj)

obj 和这个 BundlePermission 类进行比较的 Object 对象

- 检测两个 BundlePermission 对象是否相等。这个方法检测指定对象是否和这个 BundlePermission 具有相同的 bundle 符号名称和操作。

Returns 如果 obj 对象是一个 BundlePermission 对象，而且它们具有相同的 bundle 符号名称和操作，那么返回 true；否则返回 false。

6.1.10.7.public String getActions( )

- 返回 BundlePermission 对象的操作，表示为一个范式字符串。

  通常按照如下顺序返回：

  PROVIDE,REQUIRE, HOST,FRAGMENT

Returns 表示 BundlePermission 对象操作的范式字符串

6.1.10.8.public int hashCode( )

- 返回这个对象的哈希码

Returns 这个对象的哈希码

6.1.10.9.public boolean implies(Permission p)

p 待查询的目标权限

- 判断这个 BundlePermission 对象是否蕴含了指定的权限。在这个方法中，检测对象的符号名称中是否蕴含了目标的符号名称。BundlePermission 的操作列表必须要么匹配目标对象的列表，要么是允许目标对象的权限列表的，这样，才可以说是蕴含了目标 BundlePermission 操作。

如果 bundle 需要隐含权限，那么对权限描述应该使用符号名称描述。

x.y.*,"provide" -> x.y.z,"provide" is true

*,"require" -> x.y, "require" is true

*,"provide" -> x.y, "require" is true

x.y,"provide" -> x.y.z, "provide" is false

Returns 如果这个对象隐含了指定的 bundle 权限操作，那么返回 true；否则返回 false。

6.1.10.10. public PermissionCollection newPermissionCollection( )

- 返回一个新的 PermissionCollection 对象，这样就可以对 BundlePermission 进行排序。

Returns 一个新的 PermissionCollection 对象

6.1.11. public interface Configurable

支持可配置的对象。

Configurable 是一个接口，bundle 的开发人员使用这个接口来开发一个可配置的服务。如果一个 bundle 需要配置一个服务，那么需要测试这个服务对象是否为 Configurable 的一个实例对象。

Deprecated 从 1.2 后废弃，请使用配置管理服务

6.1.11.1.public Object getConfigurationObject( )

- 返回这个服务的配置对象。

实现了 Configurable 的服务在返回一个服务的 Configurable 对象时应该小心，因为这个对象是易变的。

如果 Java 运行环境支持权限，那么强烈建议在返回一个 Configurable 对象之前对调用者的权限进行检查。

Returns 这个服务的 Configurable 对象

Throws SecurityException – 如果 Java 运行环境支持权限，而调用者没有相应的权限。

Deprecated 从 1.2 后开始废弃，请使用配置管理服务（Configuration Admin service）。

6.1.12. public interface Constants

定义了 OSGi 环境特性、服务特性和清单标记的标准名称。

如果没有特别说明，这些键的值的类型均为 java.lang.String。

Since 1.1

6.1.12.1.public static final String BUNDLE_ACTIVATOR = "Bundle-Activator"

清单标记（Bundle-Activator）的属性，标识这个 bundle 的启动类。

如果有这个属性，那么也就指定了当启动和停止 bundle 时框架所调用的 start 方法和 stop 方法的来源，来自于指定的 bundle 资源实现了 BundleActivator 接口的类。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.2.public static final String BUNDLE_CATEGORY = “Bundle-Category”

清单标记（Bundle-Category）的属性，标识这个 bundle 的分类。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.3.public static final String BUNDLE_CLASSPATH = “Bundle-ClassPath”

清单标记（Bundle-ClassPath）的属性，标识这个 JAR 文件中的目录列表，表示用于扩展 bundle 类路径的 bundle 资源。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.4.public static final String BUNDLE_CONTACTADDRESS = “Bundle-ContactAddress”

清单标记（Bundle-ContactAddress）的属性，标识 bundle 发生问题之后报送的地址；例如，一个 Email 地址。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.5.public static final String BUNDLE_COPYRIGHT = “Bundle-Copyright”

清单标记（Bundle-Copyright）的属性，标识 bundle 的版权信息。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.6.public static final String BUNDLE_DESCRIPTION = “Bundle-Description”

清单标记（Bundle-Description）的属性，包含了 bundle 功能的简单说明。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.7.public static final String BUNDLE_DOCURL = “Bundle-DocURL”

清单标记（Bundle-DocURL）的属性，标识 bundle 文档说明的 URL 地址，从这个地址可以获得更多的信息。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.8.public static final String BUNDLE_LOCALIZATION =

“Bundle-Localization”

清单标记（Bundle-Localization）的属性，标识 bundle 本地化条目的基准名称。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

See Also BUNDLE_LOCALIZATION_DEFAULT_BASENAME[p.154]

Since 1.3

6.1.12.9.public static final String

BUNDLE_LOCALIZATION_DEFAULT_BASENAME =

“OSGI-INF/l10n/bundle”

清单标记 Bundle-Localization 的默认值

See Also BUNDLE_LOCALIZATION[p.154]

Since 1.3

6.1.12.10. public static final String BUNDLE_MANIFESTVERSION =

“Bundle-ManifestVersion”

清单标记（Bundle-ManifestVersion）的属性，标识 bundle 的清单版本。bundle 的清单的版本表示了这个清单文件是采用了某一个特定的 bundle 清单版本所完成的。在 OSGi R4 中，或者是以后开发的 bundle 中，必须要指明 bundle 的清单文件版本号码。bundle 清单版本由 OSGi R4 定义，或者更高级别。例如 V1.3 的 OSGi 框架规范表示为“2”
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

Since 1.3

6.1.12.11. public static final String BUNDLE_NAME = “Bundle-Name”

清单标记（Bundle-Name）的属性，标识 bundle 的名称。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.12. public static final String BUNDLE_NATIVECODE =

“Bundle-NativeCode”

清单标记（Bundle-NativeCode）的属性，标识硬件环境的编号和 bundle 在每一个环境下的本地代码库。

可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.13. public static final String BUNDLE_NATIVECODE_LANGUAGE = "language"

清单标记属性（language），标识编写 bundle 本地代码所使用的语言。参考 ISO 639 中关于语言的规定。
属性值在 Bundle-NativeCode 中描述，例如：
Bundle-NativeCode: http.so ; language=nl_be ...

6.1.12.14. public static final String BUNDLE_NATIVECODE_OSNAME = "osname"

清单标记属性（osname），标识运行 Bundle-NativeCode 中指定的本地代码需要的操作系统名称。
属性值在 Bundle-NativeCode 中描述，例如：
Bundle-NativeCode: http.so ; osname=Linux ...

6.1.12.15. public static final String BUNDLE_NATIVECODE_OSVERSION = "osversion"

清单标记属性（osversion），标识运行 Bundle-NativeCode 中指定的本地代码的操作系统版本。
属性值在 Bundle-NativeCode 中描述，例如：
Bundle-NativeCode: http.so ; osversion="2.34" ...

6.1.12.16. public static final String BUNDLE_NATIVECODE_PROCESSOR = "processor"

清单标记属性（processor），标识运行 Bundle-NativeCode 中指定的本地代码的处理器。
属性值在 Bundle-NativeCode 中描述，例如：
Bundle-NativeCode: http.so ; processor=x86 ...

6.1.12.17. public static final String

BUNDLE_REQUIREDEXECUTIONENVIRONMENT =

“Bundle-RequiredExecutionEnvironment”

清单文件标记（Bundle-RequiredExecutionEnvironment），标识 bundle 需要的执行环境。如果在这个标记中的执行环境和 bundle 实现的一个执行环境匹配，那么服务平台就会运行这个 bundle。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

Since 1.2

6.1.12.18. public static final String BUNDLE_SYMBOLICNAME =

“Bundle-SymbolicName”

清单文件标记（Bundle-SymbolicName），标识 bundle 的符号名称。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

Since 1.3

6.1.12.19. public static final String

BUNDLE_SYMBOLICNAME_ATTRIBUTE =

“bundle-symbolic-name”

清单标记属性（bundle-symbolic-name），标识导出了在 Import-Package 中指定的导入包的 bundle 的符号名称。
属性值在 Import-Package 中描述，例如：
Import-Package: org.osgi.framework; bundle-symbolicname=
”com.acme.module.test”

Since 1.3

6.1.12.20. public static final String BUNDLE_UPDATELOCATION =

“Bundle-UpdateLocation”

清单文件标记（Bundle-UpdateLocation），标识在更新 bundle 时，新版本的 bundle 的下载位置。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.21. public static final String BUNDLE_VENDOR = “Bundle-Vendor”

清单文件标记（Bundle-Vendor），标识 bundle 的供应商。

可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.22. public static final String BUNDLE_VERSION = “Bundle-Version”

清单文件标记（Bundle-Version），标识 bundle 的版本。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.23. public static final String BUNDLE_VERSION_ATTRIBUTE = “bundle-version”

清单标记属性（bundle-version），标识在 Require-Bundle 或者 Fragment-Host 中指定的 bundle 的版本范围。缺省值为 0.0.0。
属性值在 Require-Bundle 中描述，例如：
Require-Bundle: com.acme.module.test; bundle-version=”1.1”
Require-Bundle: com.acme.module.test; bundle-version=”[1.0,2.0)”
这个属性值使用一种点分十进制数字来表示 bundle 版本的范围。如果一个 bundle-version 属性值为一个版本号而不是版本范围，那么就表示大于或者等于这个版本值的所有版本。

Since 1.3

6.1.12.24. public static final String DYNAMICIMPORT_PACKAGE = “DynamicImport-Package”

清单文件标记（DynamicImport-Package），标识在执行过程中可能会动态导入的包。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

Since 1.2

6.1.12.25. public static final String EXCLUDE_DIRECTIVE = “exclude”

清单文件指令（exclude），列出了在 Export-Package 中的导出包中不允许导出的类或者资源的列表。
属性值在 Export-Package 中描述，例如：
Export-Package: org.osgi.framework; exclude:=”MyStuff*”

Since 1.3

6.1.12.26. public static final String EXPORT_PACKAGE = “Export-Package”

清单文件标记（Export-Package），标识 bundle 提供给框架的导出包。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.27. public static final String EXPORT_SERVICE = “Export-Service”

清单文件标记（Export-Service），标识 bundle 可能注册（只是用于说明的目的）的服务的全局类名称。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

6.1.12.28. public static final String EXTENSION_DIRECTIVE = “extension”

清单文件指令（extension），标识扩展片断的类型。
指令值在 Fragment-Host 中描述，例如：
Fragment-Host: system.bundle; extension:=”framework”

See Also Constants.EXTENSION_FRAMEWORK[p.158] ,
Constants.EXTENSION_BOOTCLASSPATH[p.158]

Since 1.3

6.1.12.29. public static final String EXTENSION_BOOTCLASSPATH = “bootclasspath”

清单文件指令值（bootclasspath），标识扩展片断的类型。类型为 bootclasspath 的扩展片断由启动类加载器加载。
指令值在 Fragment-Host 中描述，例如：
Fragment-Host: system.bundle; extension:=”bootclasspath”

See Also Constants.EXTENSION_DIRECTIVE[p.158]

Since 1.3

6.1.12.30. public static final String EXTENSION_FRAMEWORK = “framework”

清单文件指令值（framework），标识扩展片断的类型。类型为 framework 的扩展片断由框架类加载器加载。
指令值在 Fragment-Host 中描述，例如：
Fragment-Host: system.bundle; extension:=”framework”

See Also Constants.EXTENSION_DIRECTIVE[p.158]

Since 1.3

6.1.12.31. public static final String FRAGMENT_ATTACHMENT_ALWAYS = “always”

清单文件指令值（always），标识附加片断类型。如果类型为“always”，那么表示任何时候都允许附加到附主上（附主进行解析或者是已经解析）。

指令值在 Bundle-SymbolicName 中描述，例如：
Bundle-SymbolicName: com.acme.module.test; fragment-attachment:=”always”

See Also Constants.FRAGMENT_ATTACHMENT_DIRECTIVE[p.159]

Since 1.3

6.1.12.32. public static final String FRAGMENT_ATTACHMENT_DIRECTIVE = “fragment-attachment”

清单文件指令（fragment-attachment），标识是否进行片断附加和什么时候进行附加。缺省值为“always”。
指令值在 Bundle-SymbolicName 中描述，例如：
Bundle-SymbolicName: com.acme.module.test; fragment-attachment:=”never”

See Also Constants.FRAGMENT_ATTACHMENT_ALWAYS[p.158] , Constants.FRAGMENT_ATTACHMENT_RESOLVETIME[p.159] , Constants.FRAGMENT_ATTACHMENT_NEVER[p.159]

Since 1.3

6.1.12.33. public static final String FRAGMENT_ATTACHMENT_NEVER = “never”

清单文件指令值（never），标识附加片断类型。如果类型为“never”，那么表示任何时候都不允许附加到附主上。
指令值在 Bundle-SymbolicName 中描述，例如：
Bundle-SymbolicName: com.acme.module.test; fragment-attachment:=”never”

See Also Constants.FRAGMENT_ATTACHMENT_DIRECTIVE[p.159]

Since 1.3

6.1.12.34. public static final String FRAGMENT_ATTACHMENT_RESOLVETIME = “resolve-time”

清单文件指令值（resolve-time），标识附加片断类型。如果类型为“resolve-time”，那么表示只有在解析 bundle 的时候允许附加到附主上。
指令值在 Bundle-SymbolicName 中描述，例如：
Bundle-SymbolicName: com.acme.module.test; fragment-attachment:=
”resolve-time”

See Also Constants.FRAGMENT_ATTACHMENT_DIRECTIVE[p.159]

Since 1.3

6.1.12.35. public static final String FRAGMENT_HOST = “Fragment-Host”

清单文件标记（Fragment-Host），标识另外一个片断 bundle 的符号名称。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象获得这个属性的值。

Since 1.3

6.1.12.36. public static final String FRAMEWORK_BOOTDELEGATION = “org.osgi.framework.bootdelegation”

框架环境属性（org.osgi.framework.bootdelegation），标识框架加载启动类路径中的类时必须要委托的包。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

See Also Constants.FRAGMENT_ATTACHMENT_DIRECTIVE[p.159]

Since 1.3

6.1.12.37. public static final String FRAMEWORK_EXECUTIONENVIRONMENT = “org.osgi.framework.executionenvironment”

框架环境属性（org.osgi.framework.executionenvironment），标识框架提供的运行环境。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

Since 1.2

6.1.12.38. public static final String FRAMEWORK_LANGUAGE = “org.osgi.framework.language”

框架环境属性（org.osgi.framework.language），标识的实现语言（ISO 639 种规定的值）。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.39. public static final String FRAMEWORK_OS_NAME = “org.osgi.framework.os.name”

框架环境属性（org.osgi.framework.os.name），标识框架的服务器操作系统。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.40. public static final String FRAMEWORK_OS_VERSION = "org.osgi.framework.os.version"

框架环境属性（org.osgi.framework.os.version），标识框架运行服务器的操作系统版本编号。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.41. public static final String FRAMEWORK_PROCESSOR = "org.osgi.framework.processor"

框架环境属性（org.osgi.framework.processor），标识框架运行服务器的处理器名称。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.42. public static final String FRAMEWORK_SYSTEMPACKAGES = "org.osgi.framework.system.packages"

框架环境属性（org.osgi.framework.system.packages），标识系统 bundle 必须要提供的包。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

Since 1.3

6.1.12.43. public static final String FRAMEWORK_VENDOR = "org.osgi.framework.vendor"

框架环境属性（org.osgi.framework.vendor），标识框架的供应商。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.44. public static final String FRAMEWORK_VERSION = "org.osgi.framework.version"

框架环境属性（org.osgi.framework.version），标识框架的版本。
可以通过 BundleContext.getProperty 方法获得这个属性的值。

6.1.12.45. public static final String IMPORT_PACKAGE = "Import-Package"

清单标记（Import-Package），标识 bundle 依赖的包。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象得到这个属性的值。

6.1.12.46. public static final String IMPORT_SERVICE = “Import-Service”

清单标记（Import-Service），标识 bundle 需要的服务的全局类名称（只用于报告的目的）。
可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象得到这个属性的值。

6.1.12.47. public static final String INCLUDE_DIRECTIVE = “include”

清单标记的指令（include），标识在 Export-Package 中必须要允许导出的类和/或者资源的列表。
这个指令在 Export-Package 中使用，示例如下：
Export-Package: org.osgi.framework; include:=”MyStuff*”

Since 1.3

6.1.12.48. public static final String MANDATORY_DIRECTIVE = “mandatory”

清单标记指令（mandatory），标识在 Export-Package 中必须要和 Import-Package 中指定的属性匹配的属性值。
这个指令在 Export-Package 中使用，示例如下：
Export-Package: org.osgi.framework; mandatory:=”bundle-symbolic-name”

Since 1.3

6.1.12.49. public static final String OBJECTCLASS = “objectClass”

服务属性（objectClass），标识服务在框架中注册的所有类名称（类型为字符串数组）。
当服务注册时，由框架设置这个属性。

6.1.12.50. public static final String PACKAGE_SPECIFICATION_VERSION = “specification-version”

清单标记属性（specification-version），标识在清单标记 Export-Package 或者 Import-Package 中指定保的版本。
这个属性在 Export-Package 或者 Import-Package 中使用，示例如下：
Import-Package: org.osgi.framework ; specification-version=”1.1”

Deprecated 从 1.3 之后废弃。由 VERSION_ATTRIBUTE[p.166]代替。

6.1.12.51. public static final String REQUIRE_BUNDLE = “Require-Bundle”

清单标记（Require-Bundle），标识这个 bundle 需要的其他 bundle 的符号名称。可以通过 Bundle.getHeaders 方法获得的 Dictionary 对象得到这个属性的值。

Since 1.3

6.1.12.52. public static final String RESOLUTION_DIRECTIVE = “resolution”

清单标记指令（resolution），标识在 Import-Package 或者 Require-Bundle 中的处理类型。
这个指令在 Export-Package 或者 Import-Package 中使用，示例如下：
Import-Package: org.osgi.framework; resolution:=”optional”
Require-Bundle: com.acme.module.test; resolution:=”optional”

See Also Constants.RESOLUTION_MANDATORY[p.163] , Constants.RESOLUTION_OPTIONAL[p.163]

Since 1.3

6.1.12.53. public static final String RESOLUTION_MANDATORY = “mandatory”

清单标记指令的值（mandatory），表示强制类型。如果指定了 mandatory 类型，那么在解析 bundle 的时候，必须要解析导入包或者是需求的 bundle。如果不能解析 mandatory 类型的导入包或者是需求 bundle，那么解析失败。
这个指令在 Require-Bundle 或者 Import-Package 中使用，示例如下：
Import-Package: org.osgi.framework; resolution:=”manditory”
Require-Bundle: com.acme.module.test; resolution:=”manditory”

See Also Constants.RESOLUTION_DIRECTIVE[p.162]

Since 1.3

6.1.12.54. public static final String RESOLUTION_OPTIONAL = “optional”

清单标记指令的值（optional），表示可选类型。如果指定了 optional 类型，那么在解析 bundle 的时候，不一定需要解析导入包或者是需求的 bundle。如果在解析 bundle 的时候没有解析导入包或者需求 bundle，那么在刷新 bundle 之前，可能不会解析导入包或者需求 bundle。
这个指令在 Require-Bundle 或者 Import-Package 中使用，示例如下：
Import-Package: org.osgi.framework; resolution:=”optional”
Require-Bundle: com.acme.module.test; resolution:=”optional”

See Also Constants.RESOLUTION_DIRECTIVE[p.162]

Since 1.3

6.1.12.55. public static final String SELECTION_FILTER_ATTRIBUTE = "selectionfilter"

清单标记属性（selection-filter），用于在系统属性上使用过滤器选择。
这个属性在清单标记中使用，示例如下：
Bundle-NativeCode: libgtk.so; selection-filter="(ws=gtk)";
...

Since 1.3

6.1.12.56. public static final String SERVICE_DESCRIPTION = "service.description"

服务属性（service.description），服务描述。
这个属性可以通过传递给 BundleContext.registerService 方法的 Dictionary 对象中提供。

6.1.12.57. public static final String SERVICE_ID = "service.id"

服务属性（service.id），服务的注册编号（类型为 java.lang.Long）。
这个属性值由框架在注册服务的时候分配。框架分配的值是一个比原来所分配的任何值都要大的一个惟一值。框架重新启动之后，这个值是会变化的。

6.1.12.58. public static final String SERVICE_PID = "service.pid"

服务属性（service.pid），服务的持久编号。
这个属性由传递给 BundleContext.registerService 方法的 propertiesDictionary 对象中提供的。
服务的持久惟一标记符是在框架的多次调用中保持不变的。
根据约定，每一个 bundle 都有自己的命名空间，名称空间的起始符号为 bundle 的标识符（参考 Bundle.getBundleId[p.126]），接下来是点号（.）。bundle 可以使用这个作为注册服务的持久标记的前缀。

6.1.12.59. public static final String SERVICE_RANKING = "service.ranking"

服务属性（service.ranking），服务的顺序编号（类型为 java.lang.Integer）。
可以通过传递给 BundleContext.registerService 方法包含属性的 Dictionary 对象获得这个属性。
框架使用这个顺序号莱检查调用 BundleContext.getServiceReference 方法返回的缺省服务；如果有多个服务实现了指定的类，那么返回具有更大顺序号的服务引用对象（ServiceReference）。
缺省的顺序号为 0。具有最大值即 Integer.MAX_VALUE 的服务是最有可能返

回的服务。如果是具有最小值即 Integer.MIN_VALUE 是最不可能返回的服务。
如果提供的属性值不是 java.lang.Integer 类型，那么就将其看作是缺省值 0。

6.1.12.60. public static final String SERVICE_VENDOR = “service.vendor”

服务属性（service.vendor），表示服务的提供者。
可以由传递给 BundleContext.registerService 方法的包含属性的 Dictionary 对象获得这个属性。

6.1.12.61. public static final String SINGLETON_DIRECTIVE = “singleton”

清单标记指令（singleton），表示 bundle 是否为单态模式。缺省值为 false。
这个指令在 Bundle-SymbolicName 中描述，如下：
Bundle-SymbolicName: com.acme.module.test; singleton:=true

Since 1.3

6.1.12.62. public static final String SUPPORTS_BOOTCLASSPATH_EXTENSION = “org.osgi.supports.bootclasspath.extension”

框架环境属性（org.osgi.supports.bootclasspath.extension），表示框架是否支持启动类路径上的扩展 bundle。如果这个值为 true，那么框架支持启动类路径上的扩展 bundle。缺省值为 false。
可以通过调用 BundleContext.getProperty 获得这个属性值。

Since 1.3

6.1.12.63. public static final String SUPPORTS_FRAMEWORK_EXTENSION = “org.osgi.supports.framework.extension”

框架环境属性（org.osgi.supports.framework.extension），表示框架是否支持框架扩展 bundle。如果这个值为 true，那么框架支持框架扩展 bundle。缺省值为 false。
可以通过调用 BundleContext.getProperty 获得这个属性值。

Since 1.3

6.1.12.64. public static final String SUPPORTS_FRAMEWORK_FRAGMENT = "org.osgi.supports.framework.fragment"

框架环境属性（org.osgi.supports.framework.fragment），表示框架是否支持片断 bundle。如果这个值为 true，那么框架支持片断 bundle。缺省值为 false。
可以通过调用 BundleContext.getProperty 获得这个属性值。

Since 1.3

6.1.12.65. public static final String SUPPORTS_FRAMEWORK_REQUIREBUNDLE = "org.osgi.supports.framework.requirebundle"

框架环境属性（org.osgi.supports.framework.requirebundle），表示框架是否支持清单标记 Require-Bundle。如果这个值为 true，那么框架支持清单标记 Require-Bundle。缺省值为 false。
可以通过调用 BundleContext.getProperty 获得这个属性值。

Since 1.3

6.1.12.66. public static final String SYSTEM_BUNDLE_LOCATION = "System Bundle"

OSGi 系统 bundle 的位置标志符，定义了是一个"系统 bundle（System Bundle）"

6.1.12.67. public static final String SYSTEM_BUNDLE_SYMBOLICNAME = "system.bundle"

OSGi 系统 bundle 的符号名称的同义词。定义了是一个"system.bundle"。

Since 1.3

6.1.12.68. public static final String USES_DIRECTIVE = "uses"

清单标记指令（uses），表示一个导出包使用的包列表。
这个指令的值在 Export-Package 标记中描述，如下：
Export-Package: org.osgi.util.tracker; uses:="org.osgi. framework"

Since 1.3

6.1.12.69. public static final String VERSION_ATTRIBUTE = “version”

清单标记属性（version），表示一个 Export-Package 或者 Import-Package 中包的版本号。
这个指令的值在 Export-Package 或者 Import-Package 标记中描述，如下：
Import-Package: org.osgi.framework; version=”1.1”

Since 1.3

6.1.12.70. public static final String VISIBILITY_DIRECTIVE = “visibility”

清单标记指令（visibility），表示在 Require-Bundle 中一个需要的 bundle 的可见性。
这个指令的值在 Require-Bundle 标记中描述，如下：
Require-Bundle: com.acme.module.test; visibility:=”reexport”

See Also Constants.VISIBILITY_PRIVATE[p.166] ,
Constants.VISIBILITY_REEXPORT[p.167]

Since 1.3

6.1.12.71. public static final String VISIBILITY_PRIVATE = “private”

清单标记指令的值（private），表示私有的可见性。如果为 private，那么表示需求 bundle 的提供的包在导出声明中是不可见的。
这个指令的值在 Require-Bundle 标记中描述，如下：
Require-Bundle: com.acme.module.test; visibility:=”private”

See Also Constants.VISIBILITY_DIRECTIVE[p.166]

Since 1.3

6.1.12.72. public static final String VISIBILITY_REEXPORT = “reexport”

清单标记指令的值（reexport），表示私有的可见性。如果为 reexport，那么表示需求 bundle 的提供的包在导出声明中重新进行说明的。删除任何和需求 bundle 导出的属性匹配的属性。
这个指令的值在 Require-Bundle 标记中描述，如下：
Require-Bundle: com.acme.module.test; visibility:=”reexport”

See Also Constants.VISIBILITY_DIRECTIVE[p.166]

Since 1.3

6.1.13. public interface Filter

基于 RFC 1960 的表达式。
可以通过选择一个过滤表达式调用 BundleContext.createFilter 方法来创建一个 Filter 对象。

可以多次使用一个Filter对象来检测匹配参数是否和创建Filter对象的表达式字符串匹配。

下面列举了一些 LDAP 表达式的例子：

“(cn=Babs Jensen)”

“(!(cn=Tim Howes))”

“(&(” + Constants.OBJECTCLASS +

“=Person)(|(sn=Jensen)(cn=Babs J*)))”

“(o=univ*of*mich*)”

See Also 框架规范中关于表达式字符串语法的描述。

Since 1.1

6.1.13.1.public boolean equals(Object obj)

obj 和这个 Filter 对象比较的 Object 对象。

□ 比较这个 Filter 对象和另外一个 obj 对象。

Returns 如果 obj 对象是一个 Filter 对象，那么返回 this.toString().equals(obj.toString()) 的值。

如果这个 obj 对象不是一个 Filter 对象，返回 false。

6.1.13.2.public int hashCode( )

□ 返回这个 Filter 对象的哈希码

Returns 这个表达式字符串的哈希码，也就是 this.toString().hashCode()

6.1.13.3.public boolean match(ServiceReference reference)

reference 待比较的服务属性的引用对象

□ 使用服务属性的键值对来进行匹配比较。键的匹配是不考虑大小写的。

Returns 如果服务的属性和这个表达式匹配，返回 true，否则返回 false。

6.1.13.4.public boolean match(Dictionary dictionary)

dictionary 匹配过程中使用了这个 Dictionary 对象的键

□ 使用 Dictionary 对象来匹配。使用 Dictionary 对象的键值对来进行匹配。匹配过程不考虑键的大小写。

Returns 如果 Dictionary 对象的键值和这个表达式的匹配，返回 true，否则返回 false。

Throws IllegalArgumentException – 如果这个 dictionary 对象包含了相同键名称的变量。

6.1.13.5.public boolean matchCase(Dictionary dictionary)

dictionary 匹配过程中使用了这个 Dictionary 对象的键

- 使用 Dictionary 对象来匹配，匹配过程要区分大小写。使用 Dictionary 对象的键值对来进行匹配。匹配过程要考虑键的大小写。

Returns 如果 Dictionary 对象的键值和这个表达式的匹配，返回 true，否则返回 false。

Since 1.3

6.1.13.6.public String toString( )

- 返回这个 Filter 对象的过滤表达式字符串。
  并且删除了字符串中没有作用的空格。

Returns 表达式字符串。

6.1.14. public class FrameworkEvent extends EventObject

框架的一般事件。
当在 OSGi 环境下发生了一般的事件后用于通知监听器的事件类。为了以后的扩展性，使用一个类型编码来标识事件类型。
OSGi 联盟保留对事件类型扩展的权利。

6.1.14.1.public static final int ERROR = 2

发生了一个错误
相关 bundle 发生了一个错误。
值为 0x00000002。

6.1.14.2.public static final int INFO = 32

发生了一个通告事件
相关 bundle 发生了一个通告事件。
值为 0x00000020。

Since 1.3

6.1.14.3.public static final int PACKAGES_REFRESHED = 4

PackageAdmin.refreshPackage 操作已经完成。
当框架调用 PackageAdmin.refreshPackage 方法完成了刷新包操作之后发出这样的事件。
PACKAGES_REFRESHED 的值为 0x00000004。

See Also PackageAdmin.refreshPackages

Since 1.2

6.1.14.4.public static final int STARTED = 1

框架已经启动。
当框架中所有的标记为 started 的 bundle 都已经启动，然后框架已经启动，并且框架达到了初始启动级别后发出的事件。
STARTED 的值为 0x00000001。

See Also StartLevel

6.1.14.5.public static final int STARTLEVEL_CHANGED = 8

已经完成 StartLevel.setStartLevel 操作
当框架完成调用 StartLevel.setStartLevel 来修改激活启动级别之后发出的事件。
STARTLEVEL_CHANGED 的值为 0x00000008

See Also StartLevel

Since 1.2

6.1.14.6.public static final int WARNING = 16

发生了警告。
相关 bundle 中发生了警告。
WARNING 的值为 0x00000010。

Since 1.3

6.1.14.7.public FrameworkEvent(int type, Object source)

type 事件类型

source 事件源对象。可能为 null。

□ 创建一个框架事件。

Deprecated 从 1.2 后废弃。这个构造方法已经废弃。使用一个系统 bundle（System Bundle）来作为事件源更加合适。

6.1.14.8.public FrameworkEvent(int type, Bundle bundle, Throwable throwable)

type 事件类型

bundle 事件源。

throwable 相关的异常。如果没有相关的异常，这个参数可以为 null。

□ 通过指定 bundle 创建一个框架事件。

6.1.14.9.public Bundle getBundle( )

□ 返回这个事件相关的 bundle，同时这个 bundle 也是事件源。

Return 事件相关的 bundle。

6.1.14.10. public Throwable getThrowable( )

□ 返回和这个事件相关的异常。

Return 相关的异常，如果没有返回 null。

6.1.14.11. public int getType( )

□ 返回框架事件的类型。

类型的值为：

- STARTED[p.169]
- ERROR[p.168]
- WARNING[p.169]
- INFO[p.168]
- PACKAGES_REFRESHED[p.168]
- STARTLEVEL_CHANGED[p.169]

Returns 状态改变的类型。

6.1.15. public interface FrameworkListener extends EventListener

框架事件监听器。当发出了一个框架事件，那么将这个事件异步发送到框架事件监听器（FrameworkListener）。

框架事件监听器是一个监听器借款，bundle 开发人员可以实现这个接口。通过使用 BundleContext.addFrameworkListener 方法来在框架中注册一个框架监听器对象。

当框架启动或者异步错误发生之后，使用一个框架事件调用框架监听器。

See Also FrameworkEvent[p.168]

6.1.15.1.public void frameworkEvent(FrameworkEvent event)

event 框架事件对象。

□ 从一个一般框架事件对象获得通知。

6.1.16. public class FrameworkUtil

框架实用类
这个包含了访问框架功能的通用的方法，对 bundle 来说，这些方法可能很有用。

Since 1.3

6.1.16.1.public static Filter createFilter(String filter) throws InvalidSyntaxException

filter 表达式字符串。

□ 创建一个 Filter 对象。可以使用这个 Filter 对象来和一个 ServiceReference 对象或者是 Dictionary 对象进行匹配。
如果不能解析表达式字符串，那么抛出异常 InvalidSyntaxException，并且在异常中包含了可读信息描述不能解析的部分。

Returns 封装了表达式字符串的 Filter 对象。

Throws InvalidSyntaxException – 如果在 Filter 中包含了不能解析的部分。
NullPointerException – 如果 filter 参数为 null。

See Also Filter[p.167]

6.1.17. public class InvalidSyntaxException extends Exception

框架异常
一个 InvalidSyntaxException 对象中包含了一个具有无效语法而不能解析的表达式字符串参数。

See Filter[p.167]关于表达式字符串的语法描述。

6.1.17.1.public InvalidSyntaxException(String msg, String filter)

msg 消息

filter 非法的表达式字符串

□ 创建一个 InvalidSyntaxException 类型的异常。
这个方法创建一个 InvalidSyntaxException 对象，对象中包含了指定的消息和产生异常的表达式字符串。

6.1.17.2.public InvalidSyntaxException(String msg, String filter, Throwable cause)

msg 消息
filter 非法的表达式字符串
cause 异常的原因
□ 创建一个 InvalidSyntaxException 类型的异常。
这个方法创建一个 InvalidSyntaxException 对象，对象中包含了指定的消息和产生异常的表达式字符串。
Since 1.3

6.1.17.3.public Throwable getCause( )

□ 返回异常的原因，如果创建这个对象的时候没有指定原因，那么返回 null。
Returns 异常的原因，如果创建这个对象的时候没有指定原因，那么返回 null。
Since 1.3

6.1.17.4.public String getFilter( )

□ 返回生成这个 InvalidSyntaxException 对象的表达式字符串。
Returns 非法的表达式字符串。
See Also BundleContext.getServiceReferences[p.143]
BundleContext.addServiceListener(ServiceListener,String)[p.138]

6.1.17.5.public Throwable initCause(Throwable cause)

cause 异常的原因
□ 异常的原因，只有在构造的时候才可以设置。
Returns 这个对象。
Throws IllegalStateException – 这个方法通常都会抛出异常 IllegalStateException，这是由于只有在创建的时候才可以设置的。
Since 1.3

6.1.18. public final class PackagePermission extends BasicPermission

bundle 导入或者导出包的权限。其中包名称是点号分割的包全名。例如：org.osgi.service.http
包权限具有两个操作：EXPORT 和 IMPORT，其中 EXPORT 隐含了 EXPORT

6.1.18.1.public static final String EXPORT = “export”

操作名称：export。

6.1.18.2.public static final String IMPORT = “import”

操作名称：import。

6.1.18.3.public PackagePermission(String name, String actions)

name 包名称。

actions EXPORT,IMPORT（规范顺序）

- 定义了在 OSGi 环境下的导入和/或导出包的权限。
  其中包名称使用 Java 包名称：点号分割的字符串，可以使用通配符。例如：
  org.osgi.service.http
  javax.servlet.*
  *
  包权限将授予包的所有版本。如果一个 bundle 需要导出一个包，那么必须要有合适的包权限；同样的，如果 bundle 需要导入一个包，也需要具有相应的包权限。
  可以将权限赋予给类或者资源。

6.1.18.4.public boolean equals(Object obj)

obj 和这个包权限对象进行比较的 Object 对象。

- 判断这两个 PackagePermission 对象是否相等，这个方法检查这两个对象是否具有相同的包名称和权限操作。

Returns 如果 obj 对象是一个 PackagePermission 对象，而且具有相同的包名称和操作，那么返回 true，否则返回 false。

6.1.18.5.public String getActions( )

- 返回这个 PackagePermission 对象的权限操作的规范字符串。
  通常返回的操作顺序如下：
  EXPORT,IMPORT

Returns 这个 PackagePermission 对象的权限操作的规范字符串

6.1.18.6.public int hashCode( )

□ 返回这个对象的哈希码

Returns 这个对象的哈希码

6.1.18.7.public boolean implies(Permission p)

p 待查询的目标权限

□ 检查这个对象是否隐含了指定的目标权限。这个方法检查目标 Permission 对象的包名称是否隐含在这个对象的包名称中。这个 PackagePermission 对象的操作列表必须要和目标对象的操作匹配或者是隐含了目标对象的操作。
导出包的权限（export）隐含了导入同名包的权限（import）：
x.y.*,"export" -> x.y.z,"export" 为 true
*,"import" -> x.y, "import" 为 true
*,"export" -> x.y, "import" 为 true
x.y,"export" -> x.y.z, "export" 为 false

Returns 如果这个对象隐含了指定的包权限操作，那么返回 true；否则 false。

6.1.18.8.public PermissionCollection newPermissionCollection( )

□ 为对 PackagePermission 进行排序返回一个新的 PermissionCollection 对象。

Returns 个新的 PermissionCollection 对象

6.1.19. public class ServiceEvent extends EventObject

来自框架的事件，描述服务的生命周期改变。
当发生了服务生命周期改变的事件发生之后，将 ServiceEvent 对象发送到 ServiceListener 监听器对象。使用类型码来区分事件的类型。

See Also ServiceListener[p.176]

6.1.19.1.public static final int MODIFIED = 2

修改了一个注册的服务的属性。
当服务属性被修改之后，将这个事件同步发送到监听器。
MODIFIED 的值为 0x00000002

See Also ServiceRegistration.setProperties[p.180]

6.1.19.2.public static final int REGISTERED = 1

服务已经注册。
当在框架中注册了服务之后，将这个事件同步发送到监听器。

REGISTERED 的值为 0x00000001

See Also BundleContext.registerService(String[],Object, java.util.Dictionary)[p.145]

6.1.19.3.public static final int UNREGISTERING = 4

正在取消注册服务。
在完成取消注册服务之前，将这个事件同步发送到监听器。
如果一个 bundle 使用了一个 UNREGISTERING 的服务，那么当它接收到这个事件之后应该释放对这个服务的使用，当完成了取消注册服务操作之后，框架自动的释放 bundle 对服务的使用。
UNREGISTERING 的值为 0x00000004

See Also ServiceRegistration.unregister[p.180] ,
BundleContext.ungetService[p.147]

6.1.19.4.public ServiceEvent(int type, ServiceReference reference)

type 事件类型

reference 生命周期改变了的服务的 ServiceReference 对象。

- 创建一个新的服务事件对象。

6.1.19.5.public ServiceReference getServiceReference( )

- 返回在生命周期中发生改变的服务的一个服务引用。
  这个引用是事件源。

Returns 在生命周期中发生改变的服务的一个服务引用

6.1.19.6.public int getType( )

- 返回事件的类型。事件类型值为：
  - REGISTERED[p.174]
  - MODIFIED[p.174]
  - UNREGISTERING[p.174]

Returns 服务生命周期改变的类型。

6.1.20. public interface ServiceFactory

允许在 OSGi 环境中提供定制的服务对象。
当注册服务时，可以使用一个 ServiceFactory 对象来代替一个服务对象，这样，bundle 开发人员就可以控制分配的服务对象，服务对象分配给使用这个服务的 bundle。
当 bundle 请求服务对象时，方法 BundleContext.getService(ServiceReference) 调用 ServiceFactory.getService 方法来根据指定的请求 bundle 来创建一个服务

对象。ServiceFactory 返回的对象由框架暂存，直到 bundle 释放了对服务的使用。
当 bundle 对这个服务的使用计数变为 0 时（包含停止 bundle 或者是服务取消注册的情况），调用 ServiceFactory.ungetService 方法。
在 OSGi 环境中，只有框架才能使用 ServiceFactory 对象，而且其他 bundle 不能获得这个对象。

See Also BundleContext.getService[p.142]

6.1.20.1.public Object getService(Bundle bundle, ServiceRegistration registration)

bundle 使用服务的 bundle
registration 服务的服务注册 ServiceRegistration 对象
□ 创建一个新的服务对象。框架第一次调用这个方法时，指定的 bundle 调用方法 BundleContext.getService(ServiceReference)请求一个服务对象。服务工厂为每一个 bundle 返回一个指定的服务对象。
框架暂存返回的值（除非为 null），而且如果同样的 bundle 调用方法 BundleContext.getService 后，都返回这个暂存的对象。
框架检查返回的服务对象是否为注册时的所有类的实例，如果不是，返回 null。

Returns 服务对象，这个对象必须是服务注册时的所有类的实例。

See Also BundleContext.getService[p.142]

6.1.20.2.public void ungetService(Bundle bundle, ServiceRegistration registration, Object service)

bundle 释放服务的 bundle
registration 服务的 ServiceRegistration 对象
service 原来调用 ServiceFactory.getService 方法返回的服务对象。
□ 释放一个服务对象。
当 bundle 释放了一个服务之后，框架调用这个方法。有可能已经销毁了这个服务对象。

See Also BundleContext.ungetService[p.147]

6.1.21. public interface ServiceListener extends EventListener

服务事件监听器（ServiceEvent）。当发出了一个 ServiceEvent，将这个事件同步发送到 bundle 监听器（BundleListener）。
ServiceListener 是一个接口，bundle 的开发人员可以实现这个接口。

框架使用 BundleContext.addServiceListener 方法来注册一个 ServiceListener 对象。当注册、修改或者是处理取消注册服务时，传入一个 ServiceEvent 对象调用 ServiceListener 对象。

发送给 ServiceListener 对象的 ServiceEvent 对象是通过注册监听器时指定的表达式来过滤选择的。如果 Java 运行环境支持权限，那么还要增加其他的过滤。只有当定义这个监听器对象类的 bundle 具有相应的 ServicePermission 来使用至少一个注册的类名获得服务的时候，ServiceEvent 对象才会发送到这个监听器。

而且，还要根据 ServiceReference.isAssignableTo(Bundle, String)[p.179]中定义的包资源来对 ServiceEvent 对象进行过滤。

See Also ServiceEvent[p.173] , ServicePermission[p.176]

6.1.21.1.public void serviceChanged(ServiceEvent event)

event ServiceEvent 对象

□ 收到关于服务生命周期改变的通知。

6.1.22. public final class ServicePermission extends BasicPermission

表示对 bundle 注册或者是获得一个服务的授权。

- ServicePermission.REGISTER 操作允许 bundle 在一个指定的名称上注册一个服务。
- ServicePermission.GET 操作允许 bundle 检测并获得一个服务。

6.1.22.1.public static final String GET = “get”

权限操作 get

6.1.22.2.public static final String REGISTER = “register”

权限操作 register

6.1.22.3.public ServicePermission(String name, String actions)

name 类名称

actions get,register（规范顺序）

□ 创建一个 ServicePermission 对象。

服务的名称通过一个指定的全局类名称来确定。

ClassName ::= <class name> | <class name ending in “.*”>

例如：

org.osgi.service.http.HttpService
org.osgi.service.http.*
org.osgi.service.snmp.*

有两种可能的操作：get 和 register。get 权限允许拥有权限者通过名称来获得服务。register 权限允许 bundle 在这个名称下注册服务。

6.1.22.4.public boolean equals(Object obj)

obj 待检查的对象

□ 检查这两个 ServicePermission 对象是否相等。检测指定的 obj 是否和这个 ServicePermission 类具有相同的类名和操作。

Returns 如果 obj 是一个 ServicePermission 类，而且和这个 ServicePermission 类具有相同的名称和操作，返回 true；
否则返回 false。

6.1.22.5.public String getActions( )

□ 返回规范排列的操作字符串。通常返回的顺序如下：get, register。

Returns 表示操作的规范排列的字符串。

6.1.22.6.public int hashCode( )

□ 返回这个对象的哈希值

Returns 这个对象的哈希值

6.1.22.7.public boolean implies(Permission p)

p 检测的目标权限。

□ 检查这个 ServicePermission 对象是否隐含了指定的权限。

Returns 如果这个对象隐含了指定权限，返回 true；否则返回 false。

6.1.22.8.public PermissionCollection newPermissionCollection( )

□ 为对 ServicePermission 对象排序，返回一个新的 PermissionCollection 对象。

Returns 一个新的 PermissionCollection 对象，应用于 ServicePermission 对象排序。

6.1.23. public interface ServiceReference

对服务的引用。当调用 BundleContext.getServiceReference 方法和 BundleContext.getServiceReferences 方法时，框架返回一个 ServiceReference 对象。

可以在 bundle 之间共享 ServiceReference 对象，而且可以通过使用

ServiceReference 对象来检测服务属性，获得服务对象。
每一个在框架中注册的服务都具有一个惟一的 ServiceRegistration 对象，但是却可能有多个，截然不同的 ServiceReference 对象，关联到同一个 ServiceRegistration 对象的多个 ServiceReference 对象具有相同的哈希值，而且可以看作是相等的（当调用 equal 方法时返回 true）。
如果多次注册了同一个服务对象，那么关联到不同 ServiceRegistration 对象的 ServiceReference 对象之间是不相等的。

See Also BundleContext.getServiceReference[p.142] ,
BundleContext.getServiceReferences[p.143] ,
BundleContext.getService[p.142]

6.1.23.1.public Bundle getBundle( )

□ 返回注册这个 ServiceReference 对象引用服务的 bundle。
如果服务已经取消注册，那么这个方法返回 null。
这可以用来检测服务是否已经取消注册。

Returns 注册这个 ServiceReference 对象所引用服务的 bundle；
如果已经取消注册了这个服务，那么返回 null。

See Also BundleContext.registerService(String[],Object,java.util.Dictionary)[p.145]

6.1.23.2.public Object getProperty(String key)

key 属性键

□ 返回这个 ServiceReference 对象引用服务的属性 Dictionary 对象中和指定的属性键匹配的属性值。
属性键是不区分大小写的。
即使服务已经是取消注册的，这个方法依然要返回属性值。这样就可以对已经取消注册的服务（例如，存储在日志中的 ServiceReference 对象）也可以进行询问。

Returns 匹配的属性键的值；如果没有属性和这个键匹配，返回 null。

6.1.23.3.public String[] getPropertyKeys( )

□ 返回这个 ServiceReference 对象所引用服务的属性 Dictionary 对象中的属性键数组。
即使服务已经取消注册了，对这个方法的调用依然可以返回属性键。这样就可以对已经取消注册的服务（例如，存储在日志中的 ServiceReference 对象）也可以进行询问。
这个方法是保留大小写的；也就是说返回的数组中的属性键名称必须要和注册服务时传递给方法 BundleContext.registerService(String[],Object, java.util.Dictionary)[p.145] 或者方法 ServiceRegistration.setProperties 的属性 Dictionary 中的属性键是大小写一样的。

Returns 属性键数组

6.1.23.4.public Bundle[] getUsingBundle( )

□ 使用这个 ServiceReference 对象所引用服务的 bundle。具体来说，这个方法返回的是对这个服务的使用计数大于 0 的 bundle。

Returns 对这个 ServiceReference 对象所引用服务的使用计数大于 0 的 bundle 数组。如果没有使用这个服务的 bundle，返回 null。

Since 1.1

6.1.23.5.public boolean isAssignableTo(Bundle bundle, String className)

bundle 待检测的 Bundle 对象

className 待检测的类的名称

□ 检查指定 bundle 和注册了这个 ServiceReference 所引用服务的 bundle 是否对指定类使用了同样的包来源。

这个方法按照以下顺序来检查：

1. 从指定类名称来获得包名称；
2. 为注册了这个 ServiceReference 所引用服务的 bundle 查找包的来源。如果没有找到来源，而且注册 bundle 等于传入的参数 bundle，返回 true；否则返回 false；
3. 如果注册 bundle 的包来源等于传入参数 bundle 的包来源，返回 true；否则返回 false。

Returns 如果注册了这个 ServiceReference 所引用服务的 bundle 和传入的参数 bundle 对指定的类具有同样的包来源，返回 true；否则返回 false。

Since 1.3

6.1.24. public interface ServiceRegistration

一个注册的服务。

当成功调用了 BundleContext.registerService 方法之后，框架返回一个 ServiceRegistration 对象。ServiceRegistration 对象是注册服务的 bundle 的私有属性，不能和其他 bundle 共享。

可以使用这个 ServiceRegistration 对象来对服务进行属性更新或者是取消注册服务。

See Also BundleContext.registerService(String[],Object,Dictionary)[p.145]

6.1.24.1.public ServiceReference getReference( )

□ 返回一个已经注册了的服务的 ServiceReference 对象。

可以在 bundle 之间共享 ServiceReference 对象。

Returns ServiceReference 对象

Throws IllegalStateException – 如果这个服务已经取消注册

6.1.24.2.public void setProperties(Dictionary properties)

properties 服务的属性。参阅 Constants[p.153]中的标准服务属性键。调用这个方法之后，就不能再做这个对象修改了。如果要更新服务属性，那么应该重新调用这个方法。

- 更新相关服务的属性。

  这个方法不能修改属性：Constants.OBJECTCLASS[p.162] 和 Constants.SERVICE_ID[p.163]。在 OSGi 环境中注册服务时，由框架来设置这两个属性。

  修改服务属性的步骤如下：

  1. 根据提供的属性替换服务属性。
  2. 发出类型为 ServiceEvent.MODIFIED[p.174]的服务事件。

Throws IllegalStateException – 如果已经取消注册了这个 ServiceRegistration 对象。
IllegalArgumentException – 如果属性中存在只有大小写不同的同名的键。

6.1.24.3.public void unregister( )

- 取消注册一个服务。从框架的服务注册中心删除一个 ServiceRegistration 对象。不能再使用和这个 ServiceRegistration 对象相关的 ServiceReference 对象来访问服务了。

  按照如下步骤来取消注册一个服务：

  1. 从框架的服务注册中心删除这个服务，这样就不能再使用这个服务了。也不能再使用这个服务的 ServiceRegistration 对象来获得这个服务的服务对象。
  2. 发出事件 ServiceEvent.UNREGISTERING[p.174]，这样使用这个服务的 bundle 就可以释放对服务的使用。
  3. 对于每一个对这个服务的使用计数要大于 0 的 bundle，进行如下操作：bundle 对这个服务的使用计数清零。

     如果注册的是服务工厂对象（ServiceFactory[p.175]），调用服务工厂对象的 ServiceFactory.ungetService 方法来释放 bundle 使用的服务对象。

Throws IllegalStateException – 如果已经取消注册了这个 ServiceRegistration 对象。

See Also BundleContext.ungetService[p.147] ,
ServiceFactory.ungetService[p.175]

6.1.25. public interface SynchronousBundleListener extends BundleListener

一个同步的 bundle 事件监听器（BundleEvent）。当发出了 bundle 事件，将事件同步发送到监听器。

SynchronousBundleListener 是一个监听器接口，bundle 的开发人员可以实现这个接口。

框架使用 BundleContext.addBundleListener[p.137]方法来注册一个同步监听

器对象。当对 bundle 执行如下操作：安装、解析、开始启动、启动完毕、开始停止、停止完毕、更新、取消解析或者卸载的时候，将 BundleEvent 发送到 BundleListener 对象，调用这个 BundleListener 对象。
和一般的 BundleListener 监听器对象不一样的是，SynchronousBundleListeners 监听器是和 bundle 的生命周期操作处理同步调用的。如果监听器还没有处理完毕，那么对生命周期的操作也就没有结束。对 SynchronousBundleListeners 对象的调用的优先级要高于 BundleListener 对象。
为了删除 SynchronousBundleListeners 对象，需要有如下权限：
AdminPermission[bundle,LISTENER]

See Also BundleEvent[p.148]

Returns 1.1

6.1.26. public class Version implements Comparable

Version 标识了 bundle 和包。
Version 具有四部分：
1. 主版本号：非负整数
2. 子版本号：非负整数
3. 微版本号：非负整数
4. 限定符：文本字符串。参考 Version(String)获得关于字符串的格式说明。

Version 对象是不可变的。

Since 1.3

6.1.26.1.public static final Version emptyVersion

空版本号 0.0.0。等价于调用 new Version(0,0,0)。

6.1.26.2.public Version(int major, int minor, int micro)

major 版本的主要部分标识
minor 版本的次要部分标识
micro 版本的微部分标识
□ 根据指定的数字来创建一个版本标识。其中限定符为一个空字符串。
Throws IllegalArgumentException – 如果参数中存在负数。

6.1.26.3.public Version(int major, int minor, int micro, String qualifier)

major 版本的主要部分标识
minor 版本的次要部分标识
micro 版本的微部分标识
qualifier 限定符
□ 根据指定的数字来创建一个版本标识。

Throws IllegalArgumentException – 如果参数中存在负数，或者限定符非法。

6.1.26.4.public Version(String version)

version 表示版本限定符的字符串

□ 根据指定字符串创建一个版本标志。

下面是版本字符串中的语法格式：

version ::= major('.'minor('.'micro('.'qualifier)?)?)?

major ::= digit+

minor ::= digit+

micro ::= digit+

qualifier ::= (alpha|digit|'_'|'-')+

digit ::= [0..9]

alpha ::= [a..zA..Z]

在版本中不能存在空格。

Throws IllegalArgumentException – 如果限定符格式不正确。

6.1.26.5.public int compareTo(Object object)

object 待比较的 Version 对象

□ 将这个 Version 对象和传入的 object 对象进行比较。

如果一个 Version 对象的主版本号要小，那么这个 Version 对象的版本就要小于另外一个 Version 对象的版本；如果主版本号相等，子版本号要小，那么这个 Version 对象的版本要小于另外一个 Version 对象的版本；如果主版本号和子版本号都相等，那么就比较微版本号；如果这三者都相等，就比较限定符。限定符的比较使用 String.compareTo 方法。

Returns 如果这个 object 对象要小于次 Version 对象，返回一个负整数

相等返回 0

大于责返回一个正整数

6.1.26.6.public boolean equals(Object object)

object 待比较的 Version 对象。

□ 将这个 Version 对象和另外一个 object 对象进行比较。

只有当两个 Version 对象的四部分都相等的时候，才返回 true。

限定符的比较使用 String.equals 方法。

Returns 如果这个 Version 对象和 object 对象相等，返回 true；

否则返回 false。

6.1.26.7.public int getMajor( )

□ 返回 Version 对象的主版本号

Returns Version 对象的主版本号

6.1.26.8.public int getMicro( )

- 返回 Version 对象的子版本号

Returns Version 对象的子版本号

6.1.26.9.public int getMinor( )

- 返回 Version 对象的微版本号

Returns Version 对象的微版本号

6.1.26.10. public String getQualifier( )

- 返回 Version 对象的限定符

Returns Version 对象的限定符

返回 Version 对象
Version 对象的子

6.1.26.11. public int hashCode( )

- 返回这个对象的哈希码

Returns 一个整数，即为这个对象的哈希码

6.1.26.12. public static Version parseVersion(String version)

version 描述这个版本标记的字符串。不考虑起始和结束的空格。

- 分析传入的字符串。
  参考 Version(String)中关于字符串格式的描述。

Returns 这个字符串描述的 Version 对象。如果参数 version 为 null 或者是一个空串，返回一个 emptyVersion 对象。

Throws IllegalArgumentException – 如果参数 version 的格式不正确。

6.1.26.13. public String toString( )

- 返回这个 Version 对象的字符串表示。
  如果限定符为一个空串，那么字符串格式为：主版本号.子版本号.微版本号
  否则，格式为：主版本号.子版本号.微版本号.限定符

Returns 这个 Version 对象的字符串表示

7. 包管理服务规范

版本号： 1.2

7.1. 简介

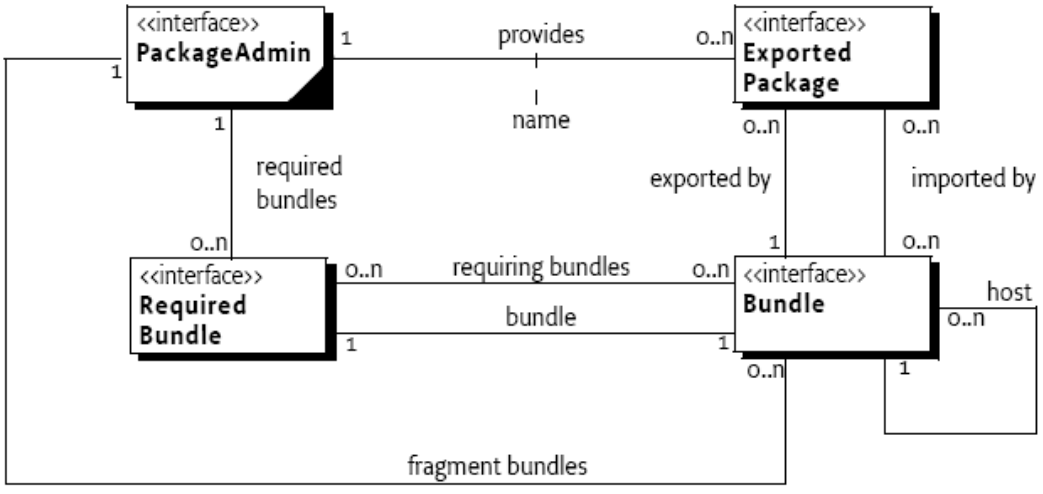
bundle 可以将包提供给其他 bundle。这种导出建立了导出包的 bundle 和使用导出包的 bundle 之间的依赖关系。当对导出包的 bundle 进行卸载或者更新，那么需要对它共享的包进行决策处理。
Package Admin 服务给管理代理提供了这种决策的接口。

7.1.1. 要点

- 信息（Information） – 包管理服务必须提供所有包的共享状态信息。包括 importerbundle 和 exporterbundle 的信息。
- 策略（Policy） – 当对 bundle 进行更新和卸载时，包管理服务必须要允许管理代理通过策略来共享包。
- 最小更新（Minimal update） – 当强制刷新包时，只有依赖于这个包的 bundle 才会重新启动。

7.1.2. 名词

- PackageAdmin – 对框架共享包进行访问的接口。
- ExportedPackage – 提供包信息和它的共享状态的接口。
- RequiredBundle – 提供了对需求 bundle 的绑定信息的接口。
- 管理代理（Management Agent） – 操作者提供的 bundle，实现了操作者指定的策略。



7.1.3. 操作

框架的系统 bundle 应该为管理代理提供了包管理服务。包管理服务必须要由系统 bundle 注册在接口 org.osgi.service.packageadmin.PackageAdmin 之下。它必须提供对框架中包共享、片断以及需求的 bundle 的内部结构进行访问的机制。这是一个可选的单态模式的服务，因此，在任何时候，最多只能有一个包管理服务。
框架必须要保证卸载或者更新后的 bundle 导出的包的共享完整性。管理代理则可以通过包管理服务来强制框架刷新这些包。可以采用一直使用安装的 bundle 最近导出的包这样一种策略，这种策略的实现可以通过管理代理监听框架关于卸载或者更新 bundle 的事件，如果发生了这样的事件，则通过包管理服务对这些 bundle 的包进行更新。

7.2. 包管理

通过包管理服务服务允许管理代理定义包共享的管理策略。提供了共享包状态的检测，并允许管理代理来刷新包并在必要情况下停止或者重启 bundle。

7.2.1. 包共享

在类 PackageAdmin 中提供了以下方法：

- getExportedPackage(String) – 返回一个提供了请求包信息的导出包。可以用这些信息来确定对包的更新。
- getExportedPackages(Bundle) – 返回指定的 bundle 导出的包列表。
- refreshPackages(Bundle[]) – 管理代理通过调用这个方法来刷新指定 bundle 的所有导出包。实际上这个过程是异步进行的。当所有的包的刷新完毕之后，框架发送一个 Framework.PACKAGES_REFRESHED 信号。
- resolveBundle(Bundle[]) –框架尝试解析指定的 bundle。

7.2.2. Bundle 信息

在框架的 API 中，只有通过 Bundle 接口，bundle 才可以在框架中扮演不同角色。包管理服务提供了对这种结构信息的访问。

- getBundle(Class) – 返回指定类的加载器所在的 bundle。
- getBundle(String,String) – 通过指定的 bundle 符号名称和版本标记来查找符合条件的所有 bundle。如果版本标记为 null 值，那么返回所有符号名称为指定值的 bundle。
- getBundleType(Bundle) – 返回 bundle 的类型。这是位类型的，定义了如下类型：
  - BUNDLE_TYPE_FRAGMENT– 返回的 bundle 是一个片断。

7.2.3. 片断和需求的 bundle

包管理服务提供了对由需求 bundle 和附主片断组成的网状结构的访问。

- getFragments(Bundle) – 返回以片断形式附加到指定 bundle 的所有 bundle 的列表。如果

没有附加的片断，则返回 null 值。

- getHosts(Bundle) – 返回指定片断 bundle 的附主 bundle。参数中的 bundle 必须要是一个 bundle 片断，否则返回 null 值。
- getRequiredBundle(String) – 返回和指定名称匹配的 RequiredBundle 对象的数组列表，如果给定名称为 null 值则返回所有的 RequiredBundle 对象。RequiredBundle 对象提供了一个需求 bundle 的结构信息。

7.2.4. 导出包

对象 ExportedPackage 提供了共享包的信息。这些对象提供了导入导出包的 bundle 的详细信息。管理代理可以使用这些信息进行决策。

7.2.5. 更新包和启动级别服务

当调用方法 refreshPackages(Bundle[])时，bundle 可能会停止运行并随后再启动。如果存在启动级别服务，那么启动 bundle 必须不能违背启动级别的约束。这也就是说具有更高的启动级别的 bundle 必须要在具有较低启动级别的 bundle 停止之前停止运行。同理，在较低启动级别的 bundle 都启动完成之前，不能启动高层 bundle。参阅启动顺序一节。

7.3. 安全

包管理服务是一个很容易就造成滥用的系统服务，这是由于它提供了对框架内部结构的访问。很多 bundle 都有权限：
ServicePermission[org.osgi.service.packageadmin.PackageAdmin, GET]
这是由于对于调用改变框架环境的任何方法都必须要有权限：
AdminPermission[System Bundle,RESOLVE]
而 bundle 都不一定要有权限：
ServicePermission [org.osgi.service.packageadmin.PackageAdmin,REGISTER]
因为只有框架本身才应该注册这样的系统服务。
这个服务是由管理代理来使用的。

7.4. 更改

包管理实现了在模块层介绍的新特性：

- getExportedPackages(String) – 提供了通过名称来访问导出包。
- resolveBundle(Bundle[]) – 强制解析一系列的 bundle。
- getFragments(Bundle) –返回和指定 bundle 相关的片断。
- getHosts(Bundle) – 返回指定 bundle 的附主。
- getRequiredBundle(String) – 返回指定符号名称的 RequiredBundle 对象组。
- getBundle(String,String) – 返回指定符号名称和版本标记的 bundle。
- getBundleType(Bundle) – 返回 bundle 的类型；目前只支持类型为片断的情况。

7.5. org.osgi.service.packageadmin

OSGi 包管理包，规范版本为 1.2。
如果 bundle 需要使用这些包，那么在它的 manifest 中的 Import-Package 必须要列出使用的包名称，例如：

Import-Package: org.osgi.service.packageadmin; version=1.2

7.5.1. 概要

- ExportedPackage – 一个导出包。[p.188]
- PackageAdmin – 框架服务，允许 bundle 开发人员来检查框架中 bundle 的包连接状态，也包括和 bundle 之间的类加载网络相关的功能。[p.190]
- RequiredBundle – 一个需求 bundle。[p.193]

7.5.2. public interface ExportedPackage

一个导出包。实现了这个接口的类由包管理服务来创建对象。
术语 exported package 表示从一个解析的 bundle 中导出的包。这个包不一定和其他 bundle 建立了连接。
由这个对象提供的导出包的信息是可以修改的。如果 ExportedPackage 所引用的包由于调用 PackageAdmin 的 refreshPackages 方法之后进行了更新或者移除，那么这个 ExportedPackage 就变成一个过期的对象。如果这个对象过期了，那么它的 getName()和 getVersion()方法继续返回正常的值，isRemovalPending()方法返回 true，getExportingBundle()方法和 getImportingBundle()方法返回 null 值。

7.5.2.1. public Bundle getExportingBundle( )

□ 返回和这个导出包相关的包的导出 bundle

Returns 导出 bundle，或者如果当前的 ExportedPackage 对象过期，则返回 null 值

7.5.2.2. public Bundle[] getImportingBundle( )

□ 返回和这个导出包建立了连接的所有解析了的 bundle 数组。在此假定如果一个 bundle 需要与导出包关联的导出 bundle，那么这个 bundle 是连接到了返回的值 bundle 的导出包。参阅方法 RequiredBundle.getRequiringBundle()。

Returns 和当前导出包建立了连接的的已经解析的 bundle 数组，或者如果当前的 ExportedPackage 对象过期，则返回 null 值

7.5.2.3. public String getName( )

- 返回和这个导出包相关的包的名称

Returns 导出包的名称

7.5.2.4. public String getSpecificationVersion( )

- 返回导出包的版本

Returns 导出包的版本，如果没有版本信息则返回 null 值。

Deprecated 从 1.2 后已过时。由方法 getVersion 代替

7.5.2.5. public Version getVersion( )

- 返回导出包的版本

Returns 导出包的版本，如果没有版本信息则返回 Version.emptyVersion 值。

Since 1.2

7.5.2.6. public boolean isRemovalPending( )

- 如果这个 ExportedPackage 对象关联的包由一个已经更新或者卸载的 bundle 导出，则返回 true

Returns 如果这个 ExportedPackage 对象关联的包由一个已经更新或者卸载的 bundle 导出，或者 ExportedPackage 对象已经过期，返回 true；否则返回 false。

7.5.3. public interface PackageAdmin

这个接口是一个框架服务，允许 bunlde 的开发人员审查框架中 bundle 的连接状态，同时提供了 bundle 之间的类加载网络的相关功能。
目前，只能允许有一个 PackageAdmin 的实例对象注册到框架。
更多信息可以参考

org.osgi.service.packageadmin.ExportedPackage[p.188]
org.osgi.service.packageadmin.RequiredBundle[p.193]

7.5.3.1. public static final int BUNDLE_TYPE_FRAGMENT = 1

- 标志 bundle 的类型，描述 bundle 是否为一个片断 bundle。
  BUNDLE_TYPE_FRAGMENT 的值为 0x00000001。

Since 1.2

7.5.3.2. public Bundle getBundle(Class clazz)

clazz 用于定位 bundle 的 Class 对象

- 返回加载指定类的 bundle。返回的 bundle 必须使用了类加载器来加载指定的 Class 对象。如果指定的 Class 对象还没有被加载，则返回 null 值。

Returns 加载指定的类的 bundle；如果指定类没有由任何 bundle 的类加载器加载，则返回 null 值。

Since 1.2

7.5.3.3. public Bundle[] getBundle(String symbolicName, String versionRange)

symbolicName 需要的 bundle 的符号名称

versionRange 需要的 bundle 的版本范围，或者如果是所有版本则为 null 值

- 返回指定符号名称而且版本标记在指定的版本范围之内的 bundle 数组。如果没有安装指定的符号名称的 bundle，那么返回 null 值。如果指定了版本范围，那么只返回符号名称一致并且版本标记在指定的版本范围之内的 bundle。返回的 bundle 按照版本大小逆序排列，这样返回的第一个值具有最大的版本编号。

Returns 符合如下条件的 bundle 数组：和指定的名称匹配，并且在指定的版本范围之内，按照版本大小逆序排列。如果没有找到符合条件的 bundle，返回 null 值。

See Also org.osgi.framework.Constants.BUNDLE_VERSION_ATTRIBUTE

Since 1.2

7.5.3.4. public int getBundleType(Bundle bundle)

bundle 需要返回类型的 bundle

- 返回指定 bundle 的特殊类型，bundle 类型的值为
  - BUNDLE_TYPE_FRAGMENT

  如果 bundle 的类型不是上述值，那么返回 0x00000000

Returns bundle 的类型

Since 1.2

7.5.3.5. public ExportedPackage getExportedPackage(String name)

name 返回的导出包的名称

- 通过指定包名称来获得导出包。如果存在多个符合条件的导出包，那么返回具有最大版本标号的导出包。

Returns 导出包；如果没有指定名称的导出包，则返回 null 值。

See Also getExportedPackages(String)

7.5.3.6. public ExportedPackage[] getExportedPackages(Bundle bundle)

bundle 返回的导出包所属的 bundle，如果返回所有的导出包，则这个参数值为 null。如果指定的是系统 bundle（也就是说指定 bundle 的 ID 为 0），那么这个方法返回由这个系统 bundle 所导出的所有包。返回值中也包括了通过系统属性 org.osgi.framework.system.packages 指定的包以及其他框架实现导出的包。

□ 获取指定 bundle 的导出包。

Returns 导出包的数组，或者如果指定 bundle 没有导出包则返回 null 值。

7.5.3.7. public ExportedPackage[] getExportedPackages(String name)

name 返回的导出包名称

□ 通过指定包名称获得导出包

Returns 导出包数组；或者如果不存在指定名称的导出包，返回 null 值。

Since 1.2

7.5.3.8. public Bundle[] getFragments(Bundle bundle)

bundle 返回的片断所附加的附主 bundle

□ 返回指定 bundle 所附加的片断 bundle 数组。如果指定的 bundle 是一个片断，那么返回 null 值。如果没有片断附加在指定 bundle 之上，那么返回 null 值。这个方法并不会去尝试解析 bundle，如果还没有解析指定 bundle，那么返回 null 值。

Returns 指定 bundle 上附加的片断；如果 bundle 上没有附加片断或者指定 bundle 还没有解析，那么返回 null 值。

Since 1.2

7.5.3.9. public Bundle[] getHosts(Bundle bundle)

bundle 待返回其附主的 bundle

□ 指定片断所附加的附主 bundle 数组，如果指定 bundle 没有附加到其他 bundle，或者指定 bundle 不是一个片断，那么返回 null 值。同时片断有可能只附加到一个 bundle 之上。

Returns 一个包含了附主 bundle 的数组；如果指定 bundle 没有附主，返回 null

Since 1.2

7.5.3.10.public RequiredBundle[] getRequiredBundle(String symbolicName)

symbolicName bundle 的符号名称或者 null 值来返回所有需求的 bundle

□ 返回指定符号名称的需求 bundle 数组，如果指定 null，返回所有的需求 bundle

Returns 需求 bundle 数组；或者如果不存在指定名称的需求 bundle 则返回 null 值

Since 1.2

7.5.3.11.public void refreshPackages(Bundle[] bundle)

bundle 待更新或者移除导出包的 bundle，或者是 null 值表示所有自从上次调用本方法之后更新或者卸载了的 bundle。

□ 强制更新（代替）或者移除制定 bundle 的导出包。

如果没有指定 bundle，本方法将对自从上次调用该方法之后进行了更新或者卸载的 bundle 的导出包进行更新或者移除。不同的框架实现也许所实现的技术会不一样。一种允许的实现就是停止并重启框架。

这个方法会立即返回给调用者，并且在另外的线程中执行以下步骤：

1. 从指定 bundle 开始计算一个 bundle 的集合。如果没有指定 bundle，那么就从上次调用本方法后更新或者卸载了的 bundle 开始计算。如果任何 bundle 和集合中的 bundle 的包有连接（wired），那么就将此 bundle 添加到当前集合。如果没有任何 bundle 和集合中的包有连接，那么就完成了集合的构建。在集合中，可能还包括了已经卸载了但是还导出包的 bundle。
2. 通过调用 Bundle.stop 方法停止集合中处于 ACTIVE 状态的 bundle。
3. 集合中处于 RESOLVED 状态的 bundle 是未解析的，状态转为 INSTALLED 状态。这个步骤的目的是使集合中不存在 RESOLVED 状态的 bundle。
4. 移除集合中处于 UNINSTALLED 状态的 bundle，并从框架中完全删除。
5. 调用 Bundle.start 方法启动在原来步骤 2 中处于 ACTIVE 状态的 bundle，使得重启 bundle 所需求的 bundle 得到解析。由于上述步骤的操作，可能导致的结果就是，原来导出的包已经改变了。因此，一些 bundle 可能会保持未解析直到其他的 bundle 提供了合适的导出包安装在框架中。
6. 发出框架事件：FrameworkEvent.PACKAGES_REFRESHED。

对于上述步骤中所抛出的任何异常，框架产生一个类型 ERROR 的框架事件。产生这些事件的源 bundle 也是这个框架事件所指定的 bundle。如果没有指定 bundle，那么就使用 SystemBundle 作为源 bundle。

Throws SecurityException — 如果 Java 允许环境支持权限而调用者没有权限：AdminPermission[SystemBundle, RESOLVE]

7.5.3.12.public boolean resolveBundle(Bundle[] bundle)

bundle 待解析的 bundle，如果为 null 表示在框架中安装的所有未解析的 bundle。

□ 解析指定 bundle。框架尝试解析指定的未解析的 bundle。这个方法的调用可能会导致并不包含在指定 bundle 数组中的 bundle 得到解析。这个方法的一个可行实现就是解析框架中所有未解析的 bundle。

如果参数指定为 null 值，那么框架尝试解析所有未解析 bundle。这个方法的调用绝不能导致任何 bundle 的刷新、停止或启动。所有操作完成才能返回。

Returns 如果所有指定的 bundle 都得到解析，返回 true。

Throws SecurityException —如果 Java 允许环境支持权限而调用者没有权限：AdminPermission[SystemBundle, RESOLVE]

Since 1.2

7.5.4. public interface RequiredBundle

需求 bundle。实现了这个接口的类对象由包管理服务创建。
术语需求 bundle（required bundle）是指一个已经解析的了带有符号名称的 bundle 而不是一个片断。也就是，一个 bundle 可能被其他 bundle 所需要，而当前这个 bundle 不一定被其他 bundle 所需要。
这个对象所提供的的需求 bundle 的信息可能会改变。如果一个 RequiredBundle 对象所引用的 bundle 的一个导出包通过调用 PackageAdmin.refreshPackages() 方法进行了更新或者移除，那么这个 RequiredBundle 对象就成为一个过期对象。如果对象已经过期，那么它的 getSymbolicName() 方法和 getVersion() 方法可以返回正常值，而方法 isRemovalPending()返回 true，getBundle()和 getRequiringBundle()方法返回 null 值。

Since 1.2

7.5.4.1. public Bundle getBundle( )

- 返回这个需求 bundle 所关联的 bundle。

Returns 该 bundle，或者如果这个 RequiredBundle 对象过期，返回 null 值

7.5.4.2. public Bundle[] getRequiringBundle( )

- 返回需要当前需求 bundle 的所有 bundle 的数组。
  如果一个 bundle 需要当前需求 bundle，然后其他 bundle 重新导出了这个需求 bundle，那么对于所有对重新导出 bundle 有需求的 bundle 都包括的返回数组中。

Returns 需要当前需求 bundle 的 bundle 数组，或者如果当前 RequiredBundle 对象过期，返回 null 值。

7.5.4.3. public String getSymbolicName( )

- 返回需求 bundle 的符号名称

Returns 需求 bundle 的符号名称

7.5.4.4. public Version getVersion( )

- 需求 bundle 的版本。

Returns 需求 bundle 的版本。
如果没有可用的版本信息，返回 Version.emptyVersion。

7.5.4.5. public boolean isRemovalPending( )

□ 如果 RequiredBundle 对象关联的 bundle 已经更新或者卸载，那么返回 true。

Returns 返回 true，如果需求 bundle 已经更新或者卸载，或者 RequiredBundle 对象已经过期。

其他情况则返回 false。

8. 启动级别服务规范

版本号： 1.0

8.1. 简介

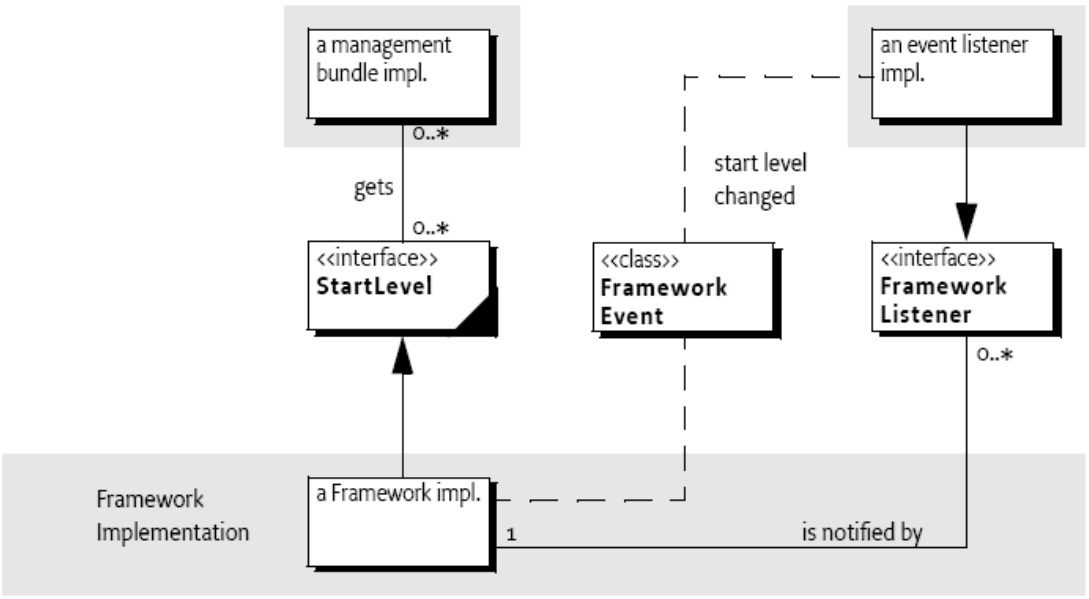
本章规范描述了在 OSGi 服务平台下，如何实现管理代理对启动和停止 bundle 的顺序进行控制。启动级别服务给每一个 bundle 分配一个启动级别（start level）。管理代理可以修改 bundle 的启动级别，并通过设置框架激活启动级别（active start level）来启动和停止相关的 bundle。只有启动级别小于或者等于激活启动级别的 bundle 才可以激活。
启动级别服务的目的在于允许管理代理对启动和停止 bundle 时进行控制。

8.1.1. 要点

- 排序（Ordering）– 管理代理可以对启动和关闭 bundle 的顺序进行排序。
- 级别（Levels） – 管理代理应该支持虚拟的无限制的级别。
- 向后兼容（Backward compatible） – 启动级别应该和 OSGi 的 R2 规范兼容。

8.1.2. 名词

- 启动级别服务（Start Level Service）– 管理代理使用的一种服务，用于对启动和停止 bundle 的顺序进行排序。
- 管理代理（Management Agent） – 参阅管理代理。
- 框架事件（Framework Event）– 参阅框架事件。
- 框架监听器（Framework Listener） – 参阅框架监听器。



8.2. 启动级别服务

启动级别服务提供了以下功能：

- 对 OSGi 框架的开始启动级别进行控制。
- 用于对框架的激活启动级别进行修改。
- 可以用于给 bundle 分配指定的启动级别。
- 初始化最新安装的 bundle 的启动级别。

对于 bundle 的启动和停止顺序的定义用于以下情况：

- 安全模式（Safe mode） – 管理代理可以实现安全模式，在这种模式下，这能启动完全信任的 bundle，如果 bundle 在启动时失败并导致了正常操作的破坏而且阻止了对问题的修正，那么在这种情况下，是有必要使用安全模式的。
- 启动快照（Splash screen） – 如果整个启动时间很长，那么就最好能在安装过程中显示启动快照。这样有助于用户对设备安装时间的把握。启动排序要确保首先启动正确的 bundle。
- 处理不稳定 bundle（Handling erratic bundle） – 由于 bundle 在激活的时候需要服务是可用的（这是一个编程错误），这样会产生一些问题。通过对启动顺序的控制，管理代理就可以预防这些问题。
- 高优先级 bundle（High priority bundle） – 有些任务如测量等是需要尽快启动的，而不能有长时间的等待，可以首先启动这些 bundle。

8.2.1. 启动级别的概念

启动级别是一个非负的整数。启动级别为 0 表示框架还没有运行或者框架已经关闭（根据环境来确定这两种状态）。在这种为 0 状态下，没有 bundle 正在运行。增长的更大的整数代表了更高的启动级别。例如，启动级别 2 要大于 1。框架必须支持 int 类型的所有整数的启动级别（最大值为 Integer.MAX_VALUE）。

框架有激活启动级别（active start level），用于确定可以启动哪些bundle。所有的bundle都有一个

bundle启动级别（bundle start level）。这是指启动bundle的最小启动级别。可以通过方法setBundletartLevel(Bundle,int)来设置bundle的启动级别。当安装完bundle之后，最初分配的启动级别是调用方法getInitialBundletartLevel()的返回值。这个对bundle安装时进行启动级别初始化的值可以通过方法setInitialBundletartLevel(int)来设置。
另外，可以通过Bundle的start和stop方法来持久标记bundle为started或者stopped。除非标记bundle为started，否则bundle就不会运行，而不考虑bundle的启动级别。

8.2.2. 修改激活启动级别

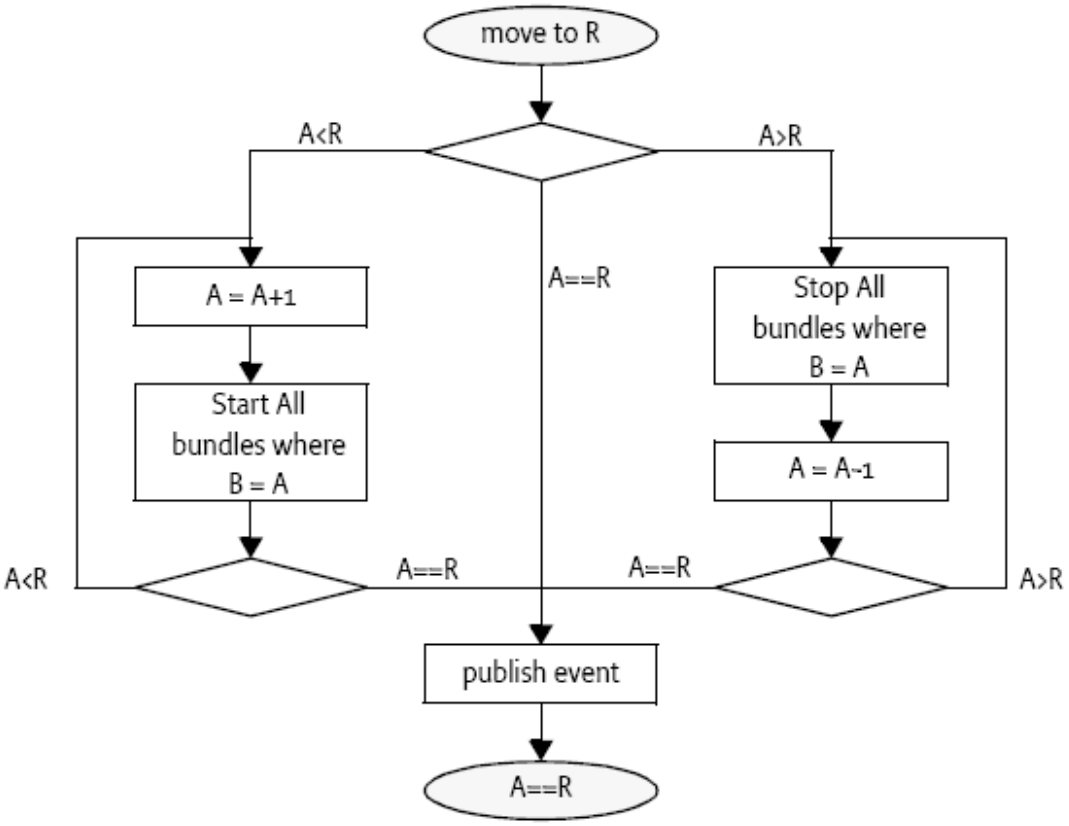
管理代理可以通过方法setStartLevel(int)来设置激活启动级别。框架通过加减1来设置激活启动级别的值直到达到了设定值。通过方法setStartLevel(int)来进行启动或者停止bundle的过程必须是异步进行的。
这也就是说必须要将激活启动级别（特定时候激活）修改为一个新的启动级别，称之为请求启动级别（requested start level）。在框架启动或者停止某些bundle的一个特定期间，激活和请求的级别是不同的。从激活启动级别转变为请求启动级别时是通过递增1完成的。
如果请求启动级别要大于激活启动级别，那么框架就将启动级别加1，并且启动所有满足以下条件的bundle：

- bundle持久标记为started，并且
- bundle的启动级别等于新的激活启动级别。

框架继续增加激活启动级别的值，并且启动符合条件的bundle，直到启动了所有启动级别和请求启动级别的值相同的bundle。
直到所有的启动bundle都从BundleActivator.start方法中返回才可以继续增加激活启动级别到下一个值，返回可以是正常返回或者是抛出了异常。如果抛出了异常，则框架必须广播一个FrameworkEvent.ERROR事件。
如果请求启动级别要比激活启动级别小，那么框架必须要停止所有的启动级别等于激活值的bundle。然后框架必须要将激活值减1。如果激活值还是小于请求值，那么继续停止符合条件的bundle并继续递减激活值直到激活值等于请求值。如果在调用BundleActivator.stop方法来停止bundle的过程中抛出了异常，那么框架必须要广播一个FrameworkEvent.ERROR事件。
如果请求值等于激活值，那么框架不会停止或者启动任何bundle。
当达到了请求值之后，而且所有符合条件（bundle启动级别<=激活启动级别）的bundle都已经启动，然后框架将FrameworkEvent.STARTLEVEL_CHANGED事件发送给所有注册了框架监听器（FrameworkListener）的对象。如果请求值和激活值相等，那么，这个事件到达时间也许要比方法setStartLevel返回的时间更早。
因此，以下情况必须为真：

- 如果bundle启动级别小于或者等于激活值，那么这个bundle是启动完成了，或者即将启动的。
- 如果bundle启动级别要大于激活值，那么这个bundle是已经停止了，或者即将停止的。

下图描述了这样的一个过程：



如果框架还没有完成上次对激活值的修改，那么在将它设置为新的激活值之前，它必须要完成上次的设置。例如，激活值是 5，框架的请求值为 3。在达到 3 之前，另外一个请求要把激活值修改为 7。在这种情况下，OSGi 框架必须要先完成将激活值修改为 3，然后再将其修改为 7。

8.2.3. 启动顺序

在启动时，框架的激活启动级别必须是 0。然后再将激活值转换到初始启动级别（beginning start level）。初始值可以来自于启动框架时输入的参数，或者通过其他未定义途径。如果没有给定初始值，框架默认初始值为 1。
框架运行后将请求值设置为初始值。然后根据前文中修改激活启动级别一节所描述那样，使得激活启动级别的值等于初始启动级别的值，如果在转换过程中抛出异常，则发出事件 FrameworkEvent.START_LEVEL_CHANGED。在运行过程中，当到达了初始的启动级别后，框架必须广播 FrameworkEvent.STARTED 事件。

8.2.4. 关闭顺序

当关闭框架时，请求启动级别必须要设置为 0。根据上文中修改激活启动级别一节所描述的那样将激活启动级别设置为 0。

8.2.5. 修改 bundle 的启动级别

当 bundle 安装完毕之后，分配给 bundle 一个初始的启动级别。默认的初始值为 1，可以通过 setInitialBundletartLevel(int)方法对这个值进行修改。当方法 setInitialBundletartLevel(int)修改了默认的初始值之后就不能再进行修改了。
安装之后，bundle的启动级别可以通过setBundletartLevel(Bundle,int)方法来修改。如果修改了bundle的启动级别，而且bundle持久标记为started，那么OSGi框架必须要将新的bundle启动级别和框架的激活值相比较。例如，假设激活值为5，启动级别为5的bundle标记为started，如果将bundle的启动级别修改为6，那么框架必须要将bundle停止，而且bundle的持久标记依然为started。

8.2.6. 启动 bundle

如果通过 Bundle.start 方法来启动 bundle，那么 OSGi 必须要持久标记 bundle 为 started。如果框架的激活值小于 bundle 的启动级别，OSGi 框架并不会实际启动 bundle，在这种情况下，bundle 的状态不会改变。

8.2.7. BundleActivator 中的异常

如果 BundleActivator 中的 start 或者 stop 方法抛出了异常，那么对异常的处理根据不同的方法调用者而不同。
如果 bundle 的启动或者停止是由于框架的激活启动级别值的修改或者是 bundle 的启动级别改变而引起的，那么必须要将异常封装成一个 BundleException 并作为 FrameworkEvent 的一个错误事件：FrameworkEvent.ERROR 广播。
否则，创建一个包含了这个异常的新的 BundleException，并将这个 BundleException 抛给调用者。

8.2.8. 系统 bundle

System Bundle 的启动级别为 0。System Bundle 的启动级别是不能修改的。如果试图修改 System Bundle 的启动级别，那么会抛出一个 IllegalArgumentException 异常。

8.3. 兼容模式

兼容模式需要完成所有 bundle 的启动级别的一致性。所有 bundle 分配的启动级别为 1。在兼容模式下，OSGi 框架可以在启动时输入参数来指定初始的启动级别为 1。然后框架启动所有持久标记为 started 的 bundle。那么当到达启动级别 1，框架也就启动了所有的 bundle，然后框架发出事件 FrameworkEvent.STARTED。这样由于启动了所有的 bundle 而没有进行任何启动控制，就和 OSGi 框架原来的标准兼容了。OSGi 框架的实现必须要支持兼容模式。

8.4. 应用实例

启动级别服务允许管理代理来实现很多不同的启动模式。本节列出了一些应用实例。

8.4.1. 安全模式启动方案

管理代理可以实现一种安全模式，在这种模式下，管理代理可以运行启动级别为 1 的可信 bundle，并且本身的级别为 2。当管理代理获得控制权之后，就构建一个待启动应用的列表。可以通过方法 BundleContext.getBundle()来构建这个列表。管理代理可以通过调用启动级别服务的方法：isBundlePersistentlyStarted(Bundle)，来检测每个标记为 started 的 bundle 是否已经启动。
在启动 bundle 之前，管理代理将 bundle 持久标记为 started 然后再启动 bundle，直到所有的 bundle 都完成启动。当需要启动一个 bundle 时，管理代理持久标记 bundle 为 started。如果重新启动服务框架，那么管理代理应该根据持久保存的信息来运行。如果持久信息中描述了 bundle 失败，那么管理代理应该试图重新启动系统，并且不处理标记为失败的应用 bundle。或者，也可以通过远程管理代理来获得帮助。

8.4.2. 启动快照方案

启动快照是包含了应用的启动信息的窗口。窗口提示用户系统还在安装 bundle。在其它 bundle 启动之前，budle 可以通过使用启动级别服务来弹出启动快照，而当启动完毕所有的 bundle 之后，将快照删除。启动快照 bundle 的启动级别可以为 1，而其他所有的 bundle 的启动级别可以为 2 或者更大的数值。

```
class SplashScreen implements
      BundleActivator, FrameworkListener {
            Screen screen;
            public void start(BundleContext context) {
                  context.addFrameworkListener( this );
                  screen = createSplash();
                  screen.open();
            }
            public void stop(BundleContext context) {
                  screen.close();
            }
      public void frameworkEvent( FrameworkEvent event ) {
            if ( event.getType() == FrameworkEvent.STARTED )
            screen.close();
      }
      Screen createSplash() { ... }
}
```

8.5. 安全

如果启动级别服务是可用的，那么就要防止非可信的bundle来访问这些服务。一个怀有恶意的

bundle如果控制了启动级别，那么也就可以控制整个服务平台。
启动级别服务是为管理代理服务的，也就是说使用这个服务的bundle必须要有管理权限：AdminPermission[bundle,EXECUTE]，这样才可以修改bundle的启动级别，或者是拥有管理权限：AdminPermission[System Bundle,STARTLEVEL]，这样才可以修改框架的激活启动级别值。如果bundle只需要访问服务，那么应该有服务权限：ServicePermission[StartLevel, GET]。
启动级别服务必须要由框架来进行注册，这样其他bundle就不可能拥有服务权限：ServicePermission[StartLevel, REGISTER]。

8.6. org.osgi.service.startlevel

OSGi 框架的 StartLevel 服务包。规范版本为 1.0。
如果 bundle 需要使用这些包，那么在它的 manifest 中的 Import-Package 必须要列出使用的包名称，例如：

Import-Package: org.osgi.service.startlevel; version=1.0

8.6.1. public interface StartLevel

StartLevel 服务允许管理代理对分配给每一个 bundle 的启动级别和框架的激活启动级别值进行管理。在 OSGi 环境中，只能有一个 StartLevel 服务。
启动级别定义了框架环境下的一个执行状态。如果启动级别值为 0，那么框架是处于非运行状态。递增的数值表示递增的启动级别值，例如 2 要大于级别 1。
通过服务权限（ServicePermission）来对启动级别服务进行访问控制。同时对启动级别信息的修改也需要有管理权限（AdminPermission）。
框架启动级别支持对框架初始启动级别的控制，可以通过初始值来修改框架的激活启动级别值，也可以用来分配给 bundle 一个指定的启动级别。初始启动级别的可以根据不同情况有不同的实现。也可以通过在框架启动的时候通过命令行参数来指定。
框架第一次启动的时候，启动级别为 0，在这种状态下，没有运行 bundle。这是框架运行前的初始状态。框架运行后，进入启动级别为 1 的状态，那么所有分配的启动级别为 1 而且持久标记为 started 的 bundle 都通过 Bundle.start 方法进行启动。在同样的启动级别下，所有的 bundle 按照 Bundle.getBundleId 方法获得的 ID 值升序的顺序启动。然后，框架持续增加启动级别的值，启动每一个级别的 bundle，直到框架达到了初始启动级别值。这时，框架才完成了 bundle 启动，发出事件 FrameworkEvent.STARTED，通知监听器已经完成了框架启动。StartLevel 服务可以通过框架激活启动级别来对 bundle 进行管理。

8.6.1.1. public int getBundletartLevel(Bundle bundle)

bundle 目标 bundle
□ 返回分配给指定 bundle 的启动级别
Returns 指定 bundle 的启动级别
Throws IllegalArgumentException — 指定 bundle 已经被卸载

8.6.1.2. public int getInitialBundletartLevel( )

□ 返回当 bundle 安装时分配给它的默认启动级别值

Returns bundle 的初始启动值
See Also setInitialBundletartLevel

8.6.1.3. public int getStartLevel( )

- 框架的激活启动级别值。如果框架正在改变激活值，那么必须要返回请求修改的目标值。

Returns 框架的激活启动级别值。

8.6.1.4. public boolean isBundlePersistentlyStarted(Bundle bundle)]

bundle 返回持久状态的目标 bundle。

- 返回指定 bundle 的持久状态。持久状态指定了如果到达了 bundle 的启动级别，是否需要标记 bundle 为 started。

Returns 如果 bundle 持久标记为 started，那么返回 true；否则返回 false。
Throws IllegalArgumentException — 如果指定 bundle 已经被卸载。

8.6.1.5. public void setBundletartLevel(Bundle bundle, int startlevel)

bundle 目标 bundle
startlevel 指定的新的启动级别

- 给目标 bundle 分配一个新的启动级别。
  给指定 bundle 分配一个指定的启动级别。分配的启动级别是需要框架进行持久存储的。如果新的启动级别要低于或者等于框架的激活值，并且 bundle 的持久标记为 started，那么框架将通过 Bundle.start 方法启动指定的 bundle。对 bundle 的启动必须是异步进行的。如果新的启动级别要高于框架激活值，那么框架将通过 Bundle.stop 方法来停止指定的 bundle，除非 bundle 的持久标记指定了 bundle 必须要进行重新启动。停止 bundle 的操作必须是异步发生的。

Returns
Throws

- IllegalArgumentException – 如果指定 bundle 已经被卸载，或者指定启动级别值要小于或者等于 0，或者指定 bundle 是一个系统 bundle。
- SecurityException – 如果运行环境支持权限控制，而调用者没有对 bundle 的管理权限：AdminPermission[bundle,EXECUTE]。

8.6.1.6. public void setInitialBundletartLevel(int startlevel)

startlevel 新安装的 bundle 的默认启动级别

- 设置用于在 bundle 安装后设置的默认启动级别。
  安装完毕的 bundle 将给它设置一个指定的启动级别。默认的启动级别由框架进行持久存储。当通过 BundleContext.installBundle 安装一个 bundle 之后，就给这个 bundle 分配设置的默认启动级别。
  如果没有调用这个方法，默认启动级别的缺省值为 1。
  这个方法的调用不会改变已经安装的 bundle 的启动级别。

Throws

- IllegalArgumentException — 如果指定的值要小于或者等于 0。

- SecurityException — 如果运行环境支持权限控制，而调用者没有对系统 bundle 的管理权限：AdminPermission[SystemBundle,STARTLEVEL]。

8.6.1.7. public void setStartLevel(int startlevel)

startlevel 对框架的请求启动级别

□ 修改框架的激活启动级别。

框架的启动值将修改到指定的值。这个方法将立即返回给调用者，对启动级别的修改是在另一个线程中异步进行的。

如果指定的启动级别要比现在的激活值高，那么框架将单步增加启动级别直到达到了指定的启动级别。对于每一个启动级别，使用 Bundle.start 方法启动所有符合条件的 bundle。在每一个到达的启动级别，也包括了指定的目标值，框架按照以下步骤进行操作：

1. 修改框架激活启动级别为临时的中间启动级别。
2. 启动所有启动级别等于这个中间值的 bundle，启动顺序由方法 Bundle.getBundleId 获得的 ID 按照升序排列。

如果框架完成了启动级别的设置，那么发出框架事件通知框架已经设置完成：FrameworkEvent.STARTLEVEL_CHANGED。

如果指定的启动值要小于框架的激活值，那么框架将单步递减框架激活值，直到达到了指定的启动值。在每一个达到的中间启动级别，如果 bundle 的启动级别等于这个中间值，那么框架调用方法 Bundle.stop 停止这个 bundle，除非 bundle 的持久标记描述了将要重新启动 bundle。在每一步的递减过程中，框架必须按照以下步骤进行：

1. 停止处于中间启动级别的 bundle，按照 Bundle.getBundleId 方法返回的 ID 值降序进行。
2. 将框架的激活启动级别修改为中间值。

当完成设置后，框架发出事件：FrameworkEvent.STARTLEVEL_CHANGED

如果指定的启动级别等于框架原来的激活值，那么不会启动或者停止任何 bundle。但是，框架必须发出事件：

FrameworkEvent.STARTLEVEL_CHANGED

这个事件可能会在方法返回之前就已经发出。

Throws

- IllegalArgumentException – 如果指定的启动值要小于或者等于 0。
- SecurityException – 如果运行环境支持权限控制，而调用者没有对系统 bundel 的管理权限： AdminPermission[SystemBundle,STARTLEVEL]

9. 条件权限管理规范

版本号： 1.0

9.1. 简介

OSGi 安全模型是基于功能强大，适应性强的 Java 2 安全架构，特别是它的权限模型。本规范在此基础上增加了一些特性，以适应 OSGi 开发人员的一些特殊要求。
与其他 Java 执行模型相比，OSGi 框架提供了良好定义的 API 来进行权限管理，而其他的执行模型将权限管理留给具体的开发实现。
安全管理 API 的一个关键特性就是进行实时的权限控制。这就使得可以对其他应用的权限管理可以立即生效，而不需要重新启动。
权限管理是基于一般的条件权限模型。bundle 使用 OSGi 联盟或者是用户定义的条件来和条件权限进行匹配。这种模型的优点就是组权限可以通过签名者和区域信息进行共享。条件模型也可以用于当满足特定条件时，则启用一组权限，例如，内置的 SIM 卡，建立了管理系统的在线连接，或者是当提示后用户批准了一项权限。这个模型允许操作者为他们的设备创建并执行一个动态的安全策略。
本规范定义了条件权限管理来代理权限管理（虽然它和权限管理的关系在本规范中已经进行了定义）。通过条件权限来提供一个安全模型，条件定义了这些权限应用的 bundle 的选集。

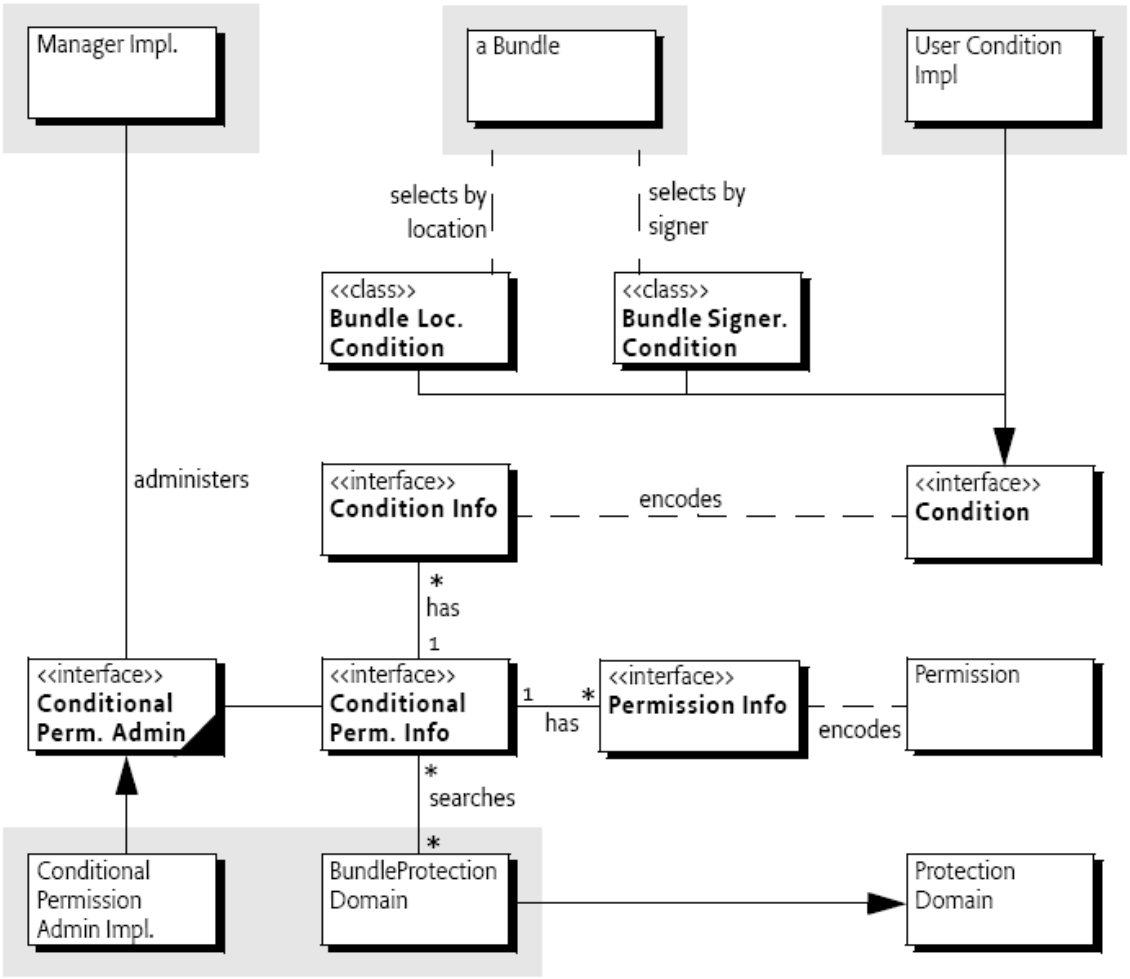
9.1.1. 要点

- Policies - 提供了一个安全策略系统，在这个系统中，外部条件控制对某一时刻的 bundle 的权限进行控制。
- Java 2 Security - 对 Java 2 安全模型提供了全部兼容，现有的系统无须修改。
- Delegation - 提供了代理管理模型，操作者可以将对设备的部分安全管理委托给另一方。
- Digital Signatures - 在 bundle 的决策机制中支持使用数字签名。
- Real Time - 环境的改变必须马上影响到 bundle 的权限。
- Operator Specific Conditions - 操作者、开发厂商、选择的开发者和其他人可以提供定制的条件。
- User Confirmation - 策略模型必须支持终端用户的审批和配置。
- Backward Compatibility - 模型必须要向后兼容原来发布的权限管理。

9.1.2. 名词

- Conditional Permission Admin – 提供了对权限表格进行操作功能的管理服务。
- Permission Table – 一种包含了所有条件权限信息的概念上的表格。
- Conditional Permission Info – 条件信息对象集合和权限信息的集合的笛卡儿积。

- Permission Info – 带有字符串编码的权限对象。
- Condition Info – 带有字符串编码的条件对象。
- Condition – 条件对象关联到一个 bundle 的保护域，并且可以在任何时候通过计算抽象出一个条件的表示。
- Bundle Location Condition –提供一个不变的条件对象来满足当关联的 bundle 具有给定位置时的情况。
- Bundle Signer Condition – 提供一个不变的条件对象来满足当关联的 bundle 的证书是由一个指定 DN 签名的情况。
- Permission – 定义了特定权限类型的对象。
- Bundle Protection Domain – 实现了 bundle 保护域的类，这个规范没有定义类的接口，但是在这个规范中起着很重要的作用。



9.1.3. 大纲

条件权限管理服务包括了一个系统级的 ConditionalPermissionInfo 对象表格，这些对象编码格式为条件对应权限。管理员可以列举、删除和添加新的条目到表格中。
当创建了一个 bundle 之后，也创建了一个对应的惟一的 bundle 保护域。这个保护域给通过初始化在权限表格中定义的条件和权限值来给 bundle 计算其系统权限，计算过程中也同时删除了 bundle 不会使用的部分，并优化其经常使用的条目。

在 bundle 中可以有通过 bundle 权限资源来定义的局部权限。这些是 bundle 实际需要的操作权限。bundle 的有效权限集合是局部权限和系统权限的交集。在基于 Java 安全管理进行权限检查时，保护域首先检查局部权限，如果检查失败，那么整个检查过程失败。
否则，bundle 保护域检查调用 bundle 是否隐含请求的权限。为了隐含请求权限，bungle 保护域必须在它的所有满足条件的权限表格中查找到一个元组，这个元组的权限隐含了请求权限。但是，特定的条件必将推迟计算以便于将计算成组进行。例如，一个用户命令必须延期执行，这样来清除终端用户的任何冗余问题。只有当这些条件关联的权限隐含了请求权限，而请求又不能马上满足的时候，这些条件式才会延期执行。
权限检查的最后阶段，由延期的条件式所在的类对它们进行成组计算。

9.1.4. 阅读指南

9.1.4.1. 架构师

- 权限管理模型
- 条件权限
- 条件
- 数字签名JAR

9.1.4.2. 应用开发人员

- 数字签名JAR

9.1.4.3. 管理人员

- 权限管理模型
- 条件权限
- 条件
- 条件权限
- 权限管理
- 数字签名JAR
- 标准条件式

9.2. 权限管理模型

条件权限管理模型提供了对 bundle 的灵活性非常强的安全模型。然而，高度灵活性也带来了复杂度的提高。在这种情况下，为了设置一个工作环境需要进行大量的配置，这是非常繁重的工作。因此，框架实现的时候要特别注意对安全模型的实现。本节定义了一系列可行的实现方案，并定义了一些术语来详细解释这种机制。

9.2.1. 局部权限

一种好的原则就是尽可能最小化权限，并且尽可能早的设置权限。这种原则的一个实现就是通过局部权限来控制。局部权限通过嵌入到 bundle 中的 bundle 权限资源（Bundle Permission Resource）来定义；它定义了一系列权限。框架通过这些权限来限制指定 bundle，也就是说，bundle 可以获得比局部权限少的权限，而绝不能获得比局部权限更多的权限。如果没有这样的资源，那么局部权限默认就是全部的权限（All Permission）。关于 bundle 权限资源可以参阅 Bundle Permission Resource 一节。

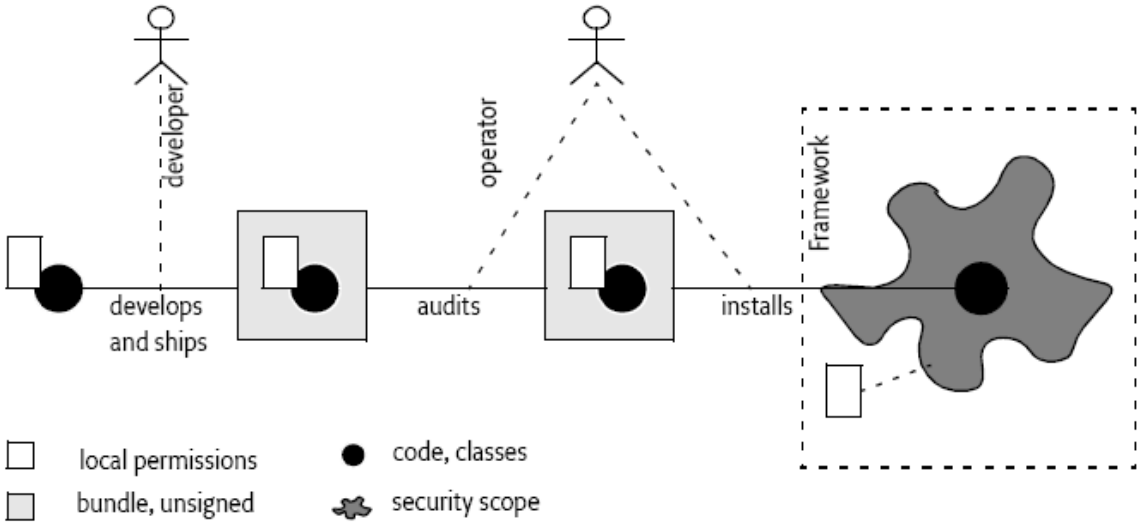
例如，如果局部权限中没有定义 ServicePermission[org.osgi.service.log.LogService,GET]，那么 bundle 就绝不能获得 LogService 这个对象，而不管是否有其他的安全设置。

OSGi 支持的局部权限是一种小粒度的权限，这种方式非常有效，这是由于可以通过开发人员来定义而不是由部署人员进行定义。开发人员非常明确的知道他所需要的服务，所需要的 bundle 的包，所需要访问的网络服务器等。而且，可以通过工具来分析 bundle，获得适当的局部权限。这样就可以简化开发人员的工作。但是，如果不知道 bundle 的内部结构，那么对于这种小粒度的权限，是很难创建出局部权限的。

初步看来，bundle 带有自己的权限是多余的。但是，局部权限定义了 bundle 所需要的最大权限，由于框架不允许 bundle 使用其它的权限，那么就没必要提供给 bundle 的更多不相关的权限。因此，局部权限的目的就是由部署人员来进行审核。为安全需求而对 bundle 的字节码进行分析是非常麻烦的事情，而不是不可能的。与之相比，对 bundle 的权限资源进行审核则更加直接。例如，如果局部权限请求访问因特网的权限，这也就说明了 bundle 潜在的希望访问网络。通过对局部权限的检查，操作者可以很快就能获得 bundle 的安全区域。核查是可信的，这是由于在执行 bundle 的时候，是由框架强制进行的。

运行封闭系统的操作者可以通过局部权限来运行第三方的应用程序，这些程序是不可信的需要进行运行检查，这样就降低了风险。框架绝不会授予在局部权限中没有隐含的权限。对应用的局部权限进行简单的检查就可以暴露出潜在的威胁。

本方案可以通过下图来进行描述。开发人员通过局部权限创建了 bundle，操作者对局部权限进行确认，如果局部权限与预期值匹配， 那么就部署这个设备，而框架则对设备进行局部权限检查。



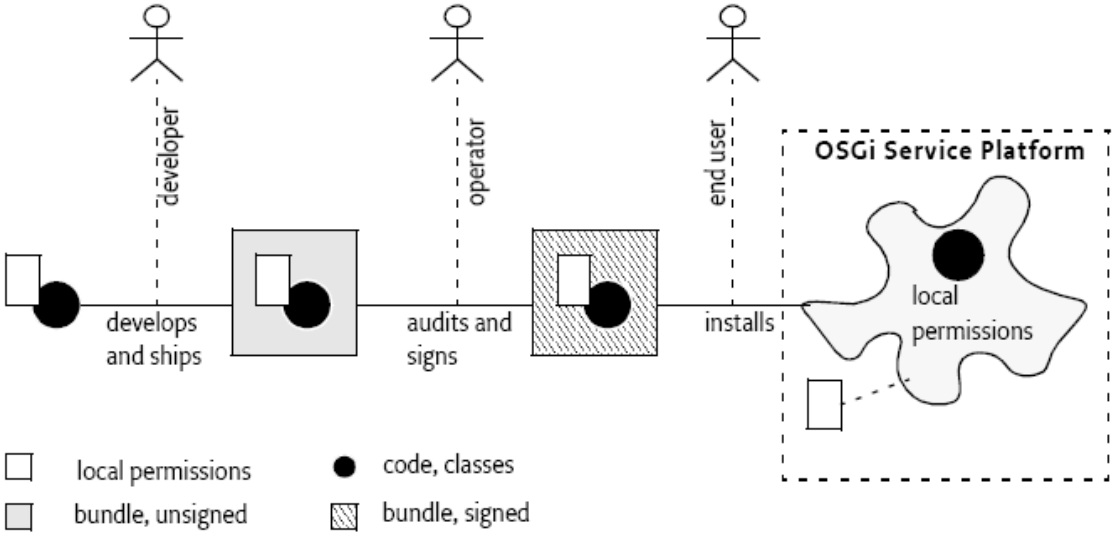
对于局部权限的优点总结如下：

- *细粒度（Fine-grained）* – 开发人员熟知对于bundle这个沙盒最少需要提供哪些细粒度的权限而使得bundle不会有约束。

- *核查（Auditable）* – 与操作者的关联度很小，而且可以通过可读文件来展示需要的沙盒。因此可以接受运行bundle的风险。
- *沙盒（Sandboxed）* – 操作者通过框架来授权，而bundle的权限不会超过局部权限。

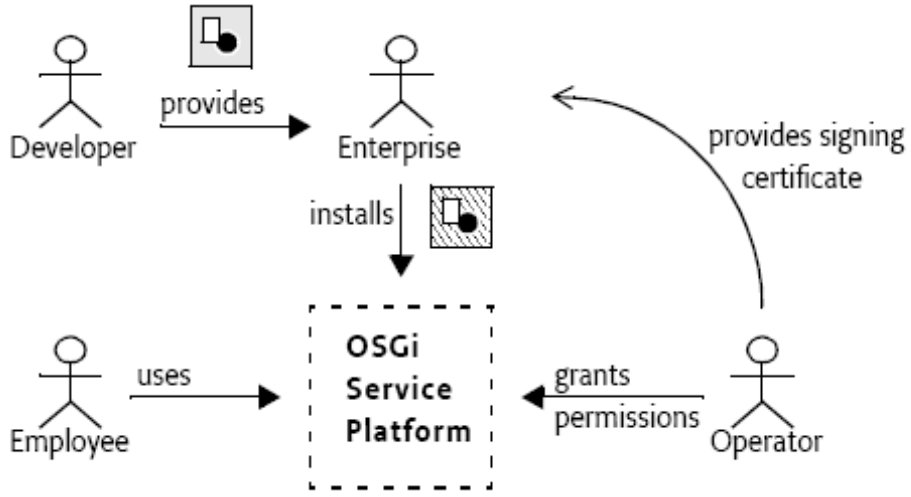
9.2.2. 开放式部署通道

从业务角度来看，如果是维持一个完全封闭的系统是过于严格。很多情况下，用户都需要从 CD、其他 PC、或者是因特网来下载部署 bundle。在这些情况下，仅仅依赖于局部权限不足以满足这样的要求，这是由于框架不能够确认局部权限是否已经被篡改过了。
实际篡改的是 bundle 的数字签名。OSGi 签名规则在 JAR 文件的数字签名一节中已经定义。数字签名算法可以检测到对 JAR 文件的修改以及对签名者验证信息的修改。因此，框架必需拒绝运行一个签名不匹配或者是签名者不可识别的 bundle。因此，签名机制使得框架可以使用一种不可信的部署通道，同时也依赖于局部权限的强制性。
例如，操作者可以通过因特网来提供应用。当用户从一个不可信的网站下载了一个应用之后，框架对签名信息进行校验。只有当签名是可信的，或者对不可信 bundle 具有默认权限时，框架才会安装这个应用。



9.2.3. 代理

具有局部权限的模型通过操作者签名来对设备进行安全控制。在提供服务之前，操作者必须要对所有的 bundle 进行签名。这种情况下，操作者实际上扮演了一个看门人的角色，而没有任何授权代理。
当包含了第三方的情况下，这种操作的代价就非常高昂。例如，一个企业通过对顾客移动电话提供应用服务，由操作者进行管理。下图描述了这种模型。如果企业在每次发布一个新版本都需要与操作者进行交涉，那么瓶颈就凸现了。



这个瓶颈问题可以通过签名来解决。签名不仅仅提供了篡改检测机制，也提供了已确认负责人（*principal*）。负责人是通过证书链来验证为已确认的。设备中包括了一系列的可信证书（根据具体实现而不同），用于对签名者的证书进行校验。

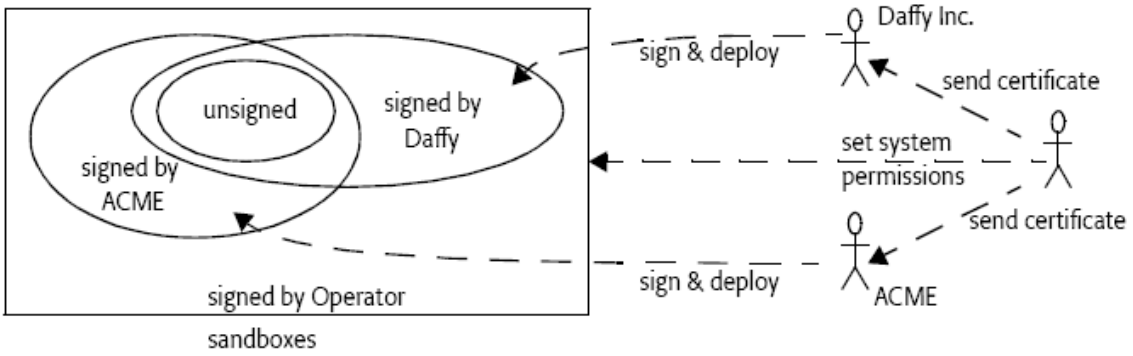
因此，操作者就可以安全的将一个负责人与一系列的权限关联。这些权限称之为系统权限（*system permissions*）。而框架对所有负责人签名的 bundle 都授予这一系列系统权限。

在这种模型中，操作者依然拥有全部的控制。在任何时候，操作者都可以根据需要来改变关联的负责人的系统权限，即使是正在运行这些应用。同样的，当签名者的一个新的服务可用之后，操作者也可以实时的给负责人授予需要的系统权限，可以给负责人授予权限来使用服务。例如，如果操作者安装了一个新的服务 org.tourist.PointOfInterest service，那么就可以给所有的负责人授予权限： ServicePermission[org.tourist.PointOfInterest,GET]和 PackagePermission[org.tourist, IMPORT]权限，授权后就被允许使用这些服务了。然后，操作者就可以通知相关的第三方。因此，这个模型就不会产生瓶颈。

使用数字签名来分配系统权限也因此可以交由第三方来代理。操作者只要定义了负责人的权限的限界，而签名和部署就可以交由第三方来完成。

例如，操作者可以定义 ACME 公司可以提供 bundle 而无须操作者干涉。操作者只需一次性的提供签名证书，并将 ACME 负责人和适当的系统权限在设备上关联。

这种模型的主要优点是减少了 ACME 和操作者之间的通讯。操作者可以修改 ACME 公司应用的系统权限，并且在任何时候都可以及时的控制。ACME 公司开发的新应用也不需要进行协调了。下图描述了这种模型，同时也展示了 Daffy 公司的未验证 bundle 沙盒的可能情况。



在代理模型中，局部权限依然起着重要的作用，这是由于它提供给签名者降低风险的可能，就如同它降低了操作者的风险一样。签名者可以在对 bundle 签名之前校验局部权限。例如，如果一个游戏 bundle 请求权限：AdminPermission[*,*]，那么很可能 bundle 不会通过签名者的安全检查。但是，即使在不太可能的情况下确实通过了检查，那么如果操作者没有授予签名负责人在

设备上这样的权限，bundle 将不会获得这样的权限。

9.2.4. 组模型

使用传统上的组模型是由于这样可以减小安全设置的管理。例如，操作者可以定义如下的安全级别：

- *不可信（Untrusted）* – 不可信的应用。这些应用必须要在最小安全范围内运行。这些应用可能是未签名的。
- *可信（Trusted）* – 可信的应用。不允许对设备进行管理或者是提供系统服务。
- *系统（System）* – 提供系统服务的应用。
- *管理（Manage）* – 管理设备的应用。

在部署 bundle 之前，操作者使用适当的证书给 bundle 进行签名，当安装 bundle 时，将自动把 bundle 放置到适当的安全区域中。
然而，也可以通过使用 bundle 的局部权限来达到同样的目的。

9.2.5. 典型示例

这些示例提供了代理模型的一个简单设置。为增加可读性，这个示例中有一些概念，将在后文中进行解释。同样也是为了可读性，可以猜测的包前缀都使用“…”来替代。
基本的权限定义了对于所有 bundle 都有的权限。因此，基本权限是没有条件关联的，所有的 bundle 都可以使用这些权限：

```
{
   (..ServicePermission "..LogService" "get" )
   (..PackagePermission "..log "import" )
   (..PackagePermission "..framework" "import" )
}
```

下一个权限元组是有条件的，定义了 ACME 公司签名的 bundle 具有的权限界限。ACME 签名的 bundle 具有管理其他 bundle 的权限。条件式在方括号中进行描述：

```
{
    [ ..BundleSignerCondition "* ; o=ACME" ]
   ( ..AdminPermission "(signer=\* ; o=ACME)" "*" )
   ( ..ServicePermission "..ManagedService" "register" )
   ( ..ServicePermission "..ManagedServiceFactory"
"register" )
   ( ..PackagePermission "..cm" "import" )
}
```

最后一个权限元组是定义了操作者签名的bundle。操作者的bundle具有所有的管理权限，并且具有提供系统服务的权限：

```
{
   [ ..BundleSignerCondition "*; o=Operator" ]
   ( ..AdminPermission "*" "*" )
   ( ..ServicePermission "*" "get,register" )
   ( ..PackagePermission "*" "import,export" )
}
```

授权结果如下表所示：

表格 11 分配的权限和通过其他权限隐含，x表示直接包含，i表示通过其他权限隐含

签名者		权限	未签名	ACME	操作者
Service Permission	..LogService	GET	x	x	x/i
	..ManagedService *	REGISTER		x	i
		GET			i
	*	*			x
PackagePermission	..log	IMPORT	x	x	x/i
	..cm	IMPORT		x	i
	..framework	IMPORT	x	x	x/i
	*	*			x
AdminPermission	(signer=*;o=ACME)	*		x	i
	*	*			x

9.3. 有效权限

一旦安装完成 bundle 之后，也就关联了 Java 2 的权限。这个集合称之为有效权限。当调用安全管理的 checkPermission 方法时，访问有效权限。

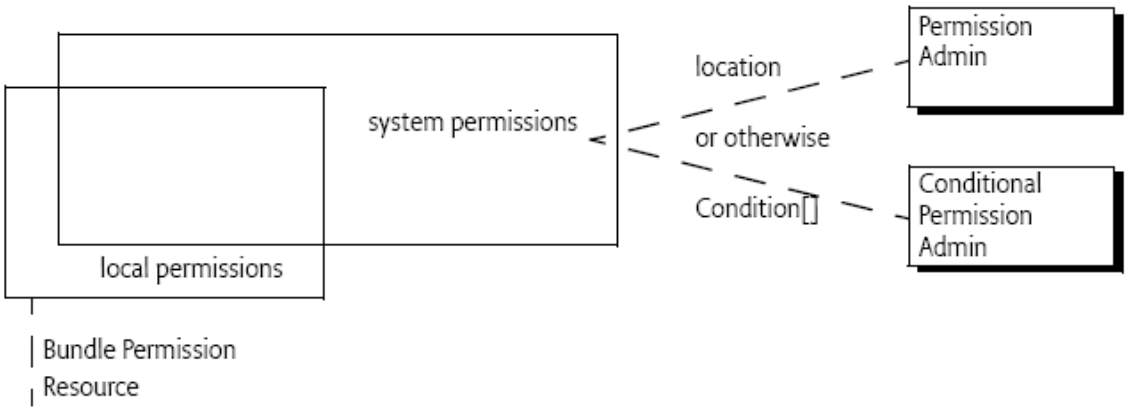
管理应用可以通过权限管理服务和有条件权限管理服务来定义系统权限（*system permissions*）。另外，bundle 也有自身的权限，称之为局部权限（*local permissions*）。所有的这些权限集合通过内部交互后得到有效的权限。

局部权限的作用是减小 bundle 签名者的风险。框架保证了 bundle 的有效权限通常要比小于或者等于局部权限，这是由于有效权限集合是局部权限与系统权限的交集。

$$Effective = Local \cap System$$

系统权限来自于两种可能的途径。一种是通过位置的权限管理服务。这种机制仅仅是为了兼容原来的版本。新开发的管理应用应该尽可能使用条件权限管理服务。

如果不能对权限管理进行定位，那么所有来自条件权限管理的条件权限充当系统权限。系统权限和局部权限的关系如图 43 所示：



9.4. 条件权限

条件权限提供了和 Java 2 策略模型（Policy mode）相关但是又不一样的更加一般化的模型。Java 2 策略模型将一系列的权限分配给代码或者是签名者。
在条件权限管理服务模型中，采用了一种更加通用的方法。在系统范围内拥有一个概念上的权限表（*permission table*）。如果权限表中的任一条目符合正确的条件式，那么它对任何 bundle 都是可用的。
表格中的元组由以下元素组成：

- 一组条件式
- 一组权限

只有当权限检查时，满足了所有的条件式，那么才能应用元组中的权限。本规范提供了某些条件类型，其他的条件类型可以由用户代码提供。例如，当相关 bundle 是由一个指定负责人签名时，可以满足 bundle 签名条件。如果 bundle 不是由这个负责人签名的，那么在检查过程中，不能应用元组的权限。
条件式的一种实现是通过 Condition 接口来表示，它表示为一个表达式，在权限检查过程中，可以计算出表达式的值为 true 或者 false。必须将条件集合进行 AND 操作。只有满足了所有的条件，那么元组才能匹配并应用权限。也就是说，元组（conditions, permissions）满足以下条件时，隐含了权限 P：

- 满足了所有的条件。
- 至少有一个权限隐含了 P，就如在 Java 2 安全中定义的那样。

Condition 对象的另一个特点就是对它们的计算可以推迟到权限检查的最后进行。推迟计算使得一种 Condition 类型进行成组对相同的条件表达式进行求值。这也就是说，例如，提示用户的重要信息。直接计算即时的条件。在条件生命周期中进行了详细描述。
例如，考虑如下对 bundle A 的设置：

```
{
  [ ...BundleSignerCondition "cn=*, o=ACME, c=US" ]
  [ com.acme.Online ]
   (...AdminPermission "*", "lifecycle")
}
```

花括号{}中定义了一个单独的元组。在方括号[]中包含了条件式，权限则用小括号（）表示。通

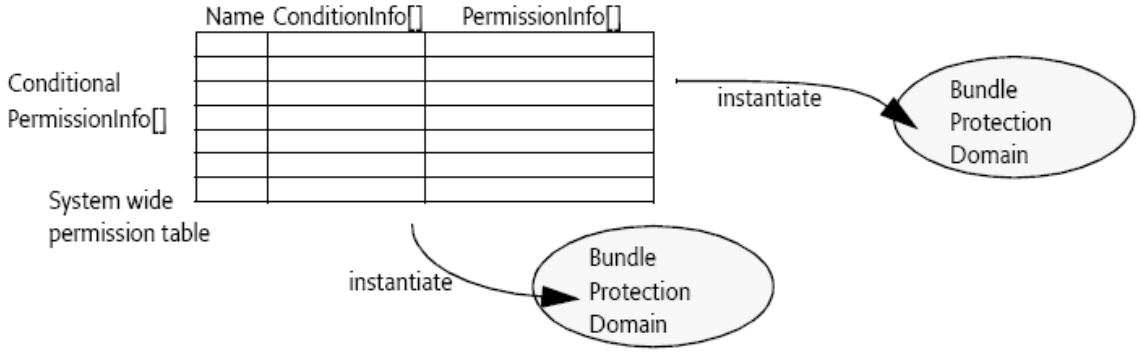
常，为了简短起见，省略了包前缀。在文档的其余部分也使用了这种语法结构。
示例展示了在对任何一个 bundle 授予运行生命周期操作的管理权限之前，必须要同时满足的条件 org.osgi.service.condpermadmin.BundleignerCondition 和条件 com.acme.Online。
将条件权限和 bundle 进行绑定是很常见的，虽然不是必须的。因此，提供了一系列的 Condition 类来定义 bundle 的系统权限。这种关联依赖于具体的 bundle：

- Signer – 实现了类 BundleignerCondition。.
- Location –实现了类 BundleLocationCondition。

bundle 的系统权限是动态易变的，这也就是为什么隐含方法的调用依赖于当权限检查时满足的条件式。原则上，系统范围内的权限表中任何元组都是可以匹配的。

9.4.1. 实例对应编码

系统范围权限表由方法 addConditionalPermissionInfo(ConditionInfo[],PermissionInfo[]) 或者方法 setConditionalPermissionInfo(String,ConditionInfo[],PermissionInfo[])来维护。这些方法使用了条件式和权限表达式的编码形式。这种编码形式保存在权限表中。权限表就象是 Bundle Protection Domain 的一个动态模板，Bundle Protection Domain 创建相关 bundle 的一个接口作为上下文环境。这是动态的，是由于 Bundle Protection Domain 必须要即时跟踪对权限表格的修改，并且立即更新其为最新的编码形式。一旦方法 addConditionalPermissionInfo(ConditionInfo[], PermissionInfo[]) 或者 setConditionalPermissionInfo(String,ConditionInfo[], PermissionInfo[])返回，那么以后对 Bundle Protection Domain 的使用必须是基于这个新的组合体。

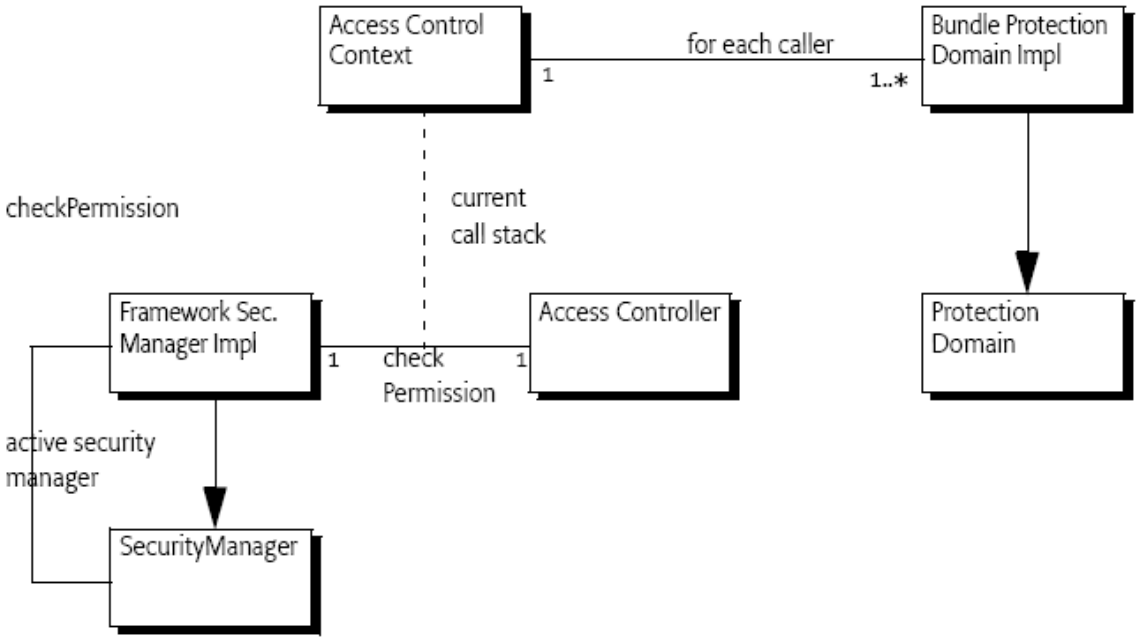


方法 addConditionalPermissionInfo 的参数是 ConditionInfo 和 PermissionInfo 对象（来自包 org.osgi.service.permissionadmin）。这些对象的目的在于对条件和权限进行编码而不是初始化。
方法 addConditionalPermission 的返回值也是一个对条件和权限进行编码的对象：一个 ConditionalPermissionInfo 对象。
权限表中的条件和权限必须在进行条件检查之前进行初始化。。初始化的过程发生在创建 Bundle Protection Domain 时或者是由于权限检查而第一次需要条件权限的时候。因此，Condition 对象必须是属于一个单独的 Bundle Protection Domain，而且不能共享。相反的，Permission 对象是上下文无关的，可以在不同的 Bundle Protection Domain 之间进行共享。

9.5. 权限检查

Java 2 安全模型同时拥有安全管理和访问控制来实现权限检查。其核心功能集中在访问控制类（AccessController）和访问控制上下文（AccessControlContext）类中，同时这两个类与检查是否已经授予权限的类保护域（ProtectionDomain）和权限（Permission）进行配合使用。保护域

（ProtectionDomain）对象中存有很多权限类。在OSGi框架中，bundle必须拥有一个单独的bundle保护域对象。
访问控制类提供了对权限检查的所有功能。但是，为了向后兼容，所有的系统检查都是通过SecurityManager对象的checkPermission方法进行。也可以使用定制方式，而不是访问控制类（是一个final类）来实现安全管理。在图45种描述了这种模型。
条件权限管理通过一系列条件来对权限控制。只有当符合条件时才会授予权限。Conditions会请求用户交互，并导致其他影响。其中一个后果就是这种执行条件的处理模型必须是高度优化的。这也就要求框架必须完成权限检查。因此，框架必须利用条件权限管理的一种实现来替代安全管理。
如果由于其他部分已经安装了，导致框架实现不能替代安全管理器（Security Manager）。那么并不是本规范的所有特性都可以实现。



9.5.1. 权限检查算法

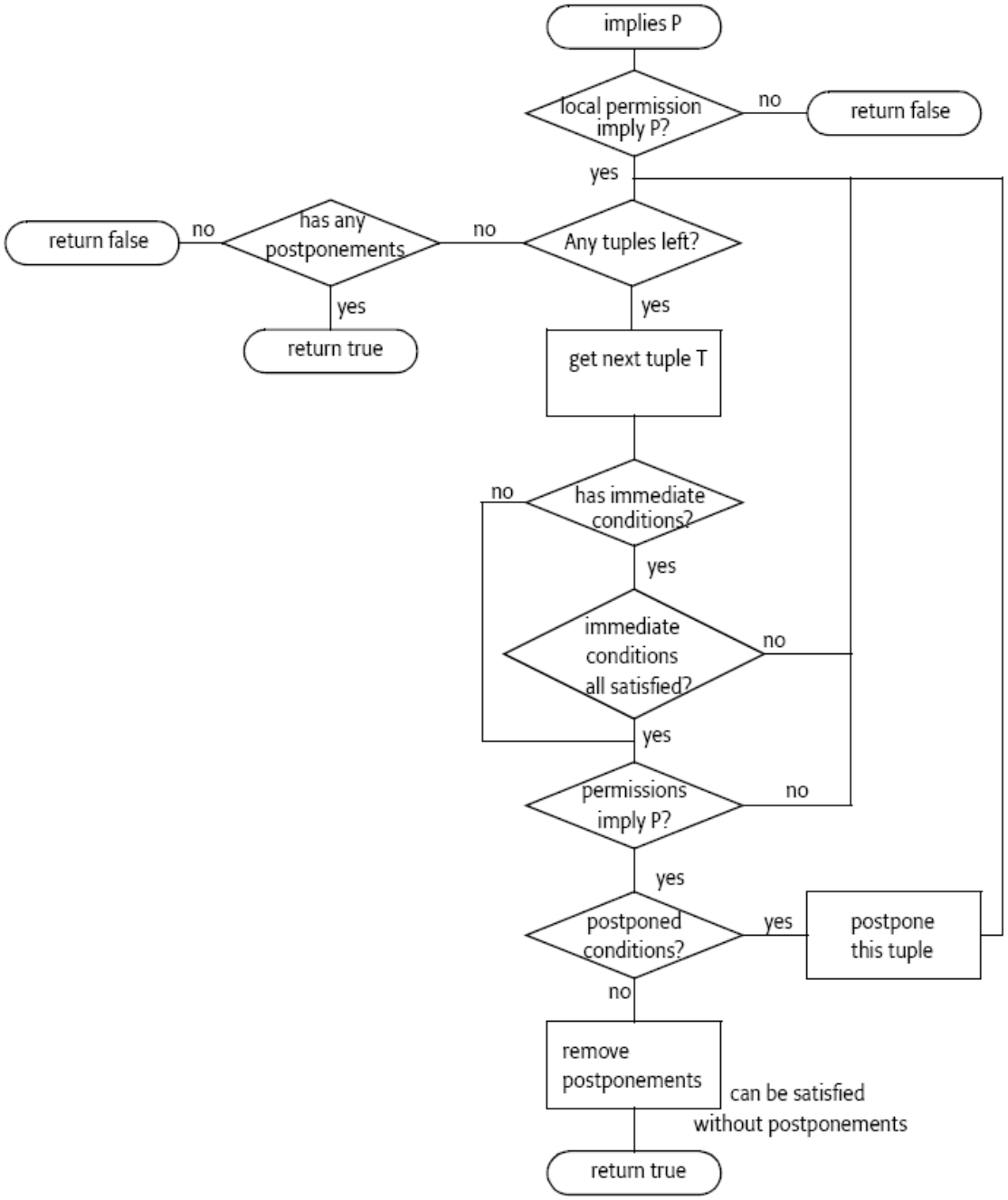
当安全管理器使用权限P为参数来调用checkPermission方法时，就开始了权限检查。框架必须实现安全管理器，称之为框架安全管理器（Framework Security Manager）。并且必须将它完全和条件权限管理服务（Conditional Permission Admin service）集成。
框架安全管理器必须有效获得访问控制上下文（Access Control Context）。如果没有对它指定一个上下文，那么必须调用AccessController. getContext()方法来获得一个默认的上下文。
然后必须调用对象AccessControlContext的checkPermission方法，这个方法将对调用栈进行遍历。在每一个栈级别上，使用ProtectionDomain对象的隐含方法对权限P计算调用类的bundle保护域。计算的过程必须在同一个线程中进行。
在Bundle Protection Domain的局部权限中必须隐含了权限P。如果没有，那么结束检查过程，检查结果为失败。在局部权限一节和bundle权限资源一节中描述了局部权限。
Bundle Protection Domain必须要确定在例示权限表中哪些元组是适用的，并且是隐含了P的。因此，它必须要执行以下步骤，或者是通过其他替代方法来获得同样的结果：

- 对于例示权限表格中的每一个元组T：

- 如果 T 中有立即条件式，计算所有的立即条件式对象，如果有任何一个条件对象不满足，那么计算下一个元组。
- 如果 T 的权限中没有隐含 P，继续下一个元组。
- 如果 T 中含有延迟的条件对象，那么延迟计算 T，转到下一个元组。
- 否则，移除所有的延迟对象，返回 true。

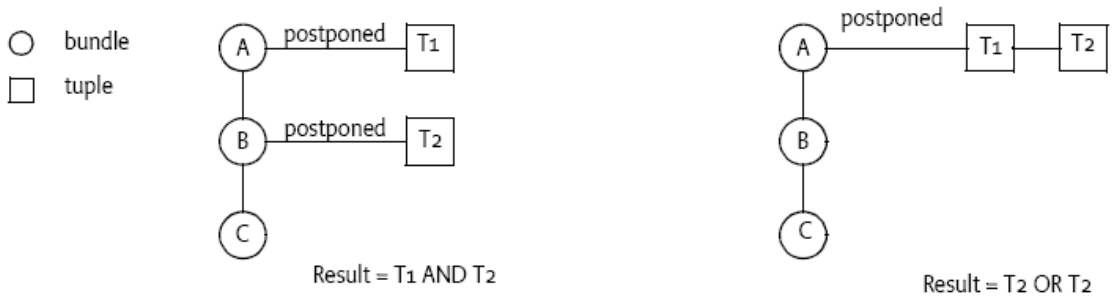
- 当所有的元组都已经处理过之后
  - 如果还有任何延迟对象，那么返回 true，否则返回 false。

在图 46 中描述了算法的流程：



在框架安全管理器调用访问控制上下文的 checkPermission 方法之后，必须确定是检查失败还是处理延迟计算的元组。如果方法返回 false，那么框架安全管理器的 checkPermission 方法检查失

败。
如果返回 true，那么还应该处理延迟计算的元组。这些元组中都蕴含了权限 P，否则就不会将它们放在延迟计算元组集合中。但是，在框架安全管理器的 checkPermission 方法返回 true 之前，还要对它们的条件进行检查确定是否满足。
延迟计算的元组集合并不是一个线性的列表。很多的 bundle 保护域（Bundle Protection Domains）都和这个列表有关。而每一个 bundle 保护域必须蕴含了需要的权限，因此，对于每一个 bundle 保护域，至少要能找到一个元组来满足，这样框架安全管理器的 checkPermission 方法才可以返回 true。
例如，在延迟计算列表中，T1 是 bundle A 的，而 T2 是 bundle B 的，那么必须要同时满足 T1 和 T2。但是，如果 T1 和 T2 都是 bundle A 的，那么只需要满足 T1 或者 T2 就可以返回 true。图 47 描述了这个示例：



条件式的计算必须是成组进行的，这样相同实现的 Condition 对象就可以一起计算。例如，如果需要用户命令输入，那么就有必须移除复制的条件式，并且所有的问题都只询问一次。

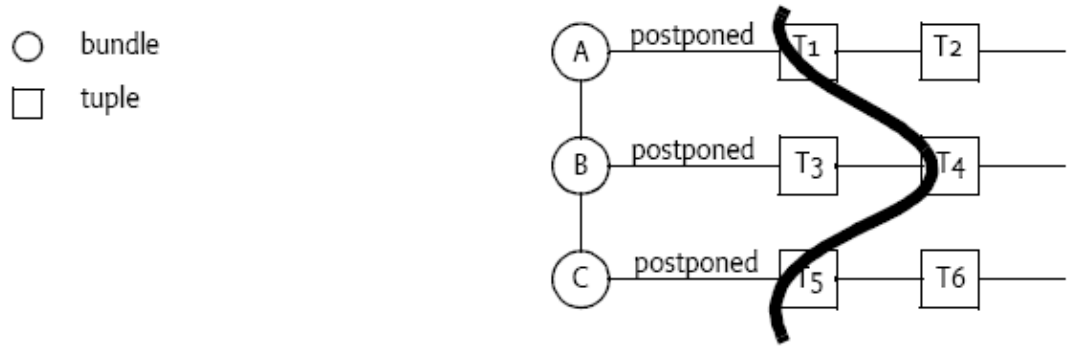
可以通过方法 isSatisfied(Condition[], Dictionary)来计算多条件式。这个方法是一个接口中的方法，应该实现这个方法。Condition 类使用这个集合来移除复制的条件式，移除覆盖的条件式，将需要用户交互的放置到一组中。这个方法可以使用语义来判断集中计算的条件式的结果是 true 还是 false。

在框架安全管理器的 checkPermission 方法调用期间，Condition 的实现类对象使用方法 isSatisfied(Condition[], Dictionary)中的 Dictionary 参数来保存状态信息。Dictionary 对象的作用：

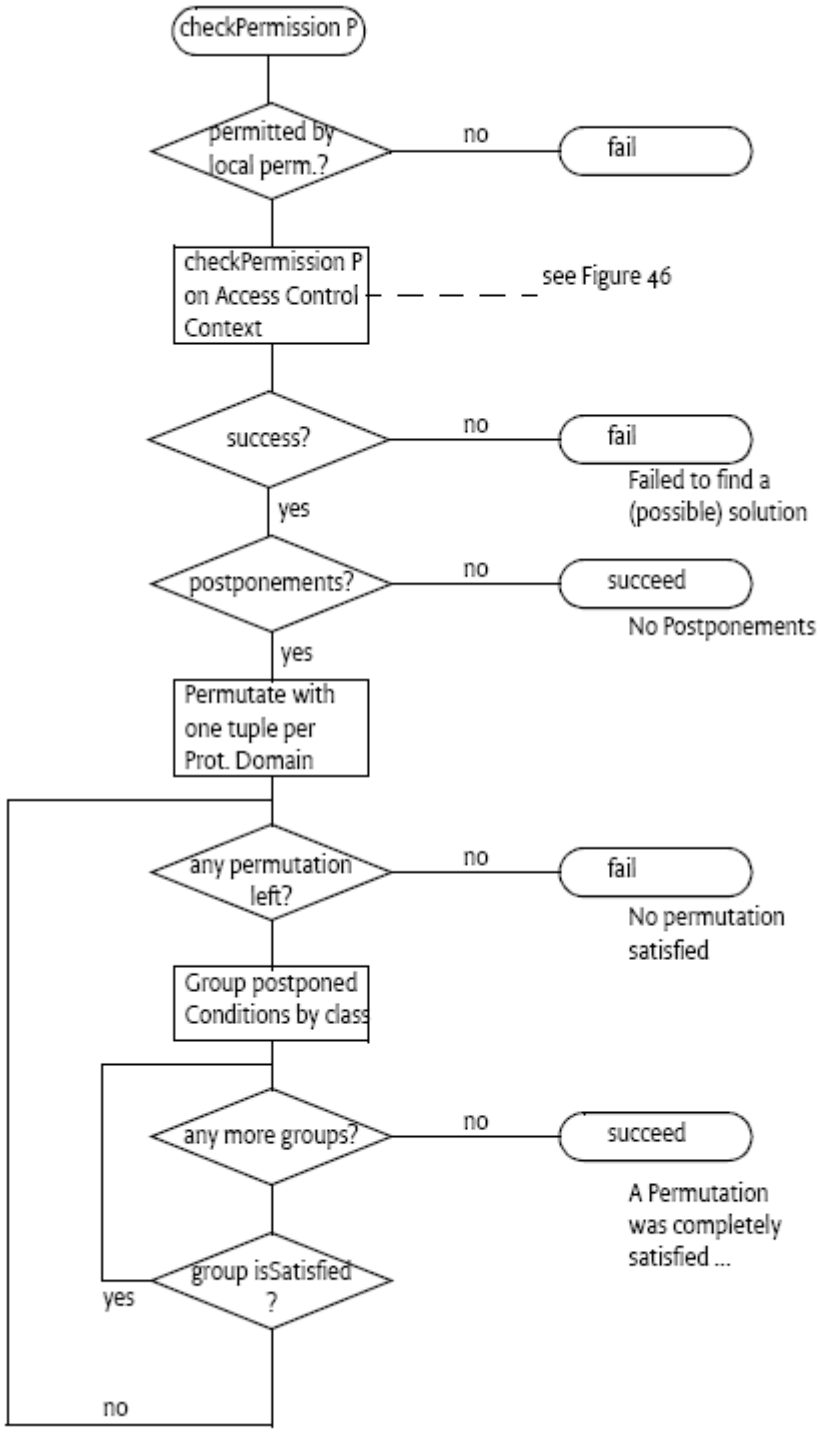
- 是 Condition 实现类的一个特例，不同的实现类不能共享这样的 Dictionary 对象。
- 在方法 isSatisfied(Condition[],Dictionary)第一次调用之前创建。
- 在调用一个单独的权限检查会话中有效。也就是说，在不同的权限检查调用期间没有保存 Dictionary 对象。
- 在不同的 bundle 保护域之间的 isSatisfied(Condition[], Dictionary)是共享的。

框架安全管理器必须对每一个 bundle 保护域找到至少一个元组来满足。也可能是每一个 bundle 保护域有多个元组，那么只需要满足其中一个元组就可以了。在方法 isSatisfied(Condition[],Dictionary)中，必须要对所有的条件进行检查确定是否满足，不允许在条件之间使用 OR 操作。因此，只能在不同的 bundle 保护域之间的元组进行成组计算。也就是说，分组必须是使用每一个 bundle 保护域中的一个条件元组。

在图 48 中，每一个 bundle 保护域都有两个元组是延迟计算的。框架安全管理器必须要将这些元组分成序列：(T1,T3,T5)、(T2,T3,T5)、(T1,T4,T5)、(T2,T4, T5)、(T1, T3, T6)、(T2,T3, T6)、(T1, T4, T6)、(T2, T4, T6)，并且计算每一个序列。对于每一个序列，使用方法 isSatisfied(Condition[],Dictionary)来计算所有的条件式对象。计算顺序可以有不同实现，而且可以进行优化。



框架安全管理器的 checkPermission 方法的整个执行算法如图 49 所示：



9.5.2. 示例

当 bundle A、B 和 C 都在调用栈中时，检查权限 P。权限配置过程如下（IC = 立即计算的条件，PC 为延迟计算的条件，P、Q 和 R 表示权限。元组已经预处理，只和给定 bundle 匹配。也就是，已经计算并清除了所有的 Bundle*Condition 对象）：

```
C: {                    (Q)      }
   { [IC0]              (P)      }
   { [PC2]              (P)      }
```

首先，如果查询 bundle C 的保护域对象中是否蕴含权限 P。C 中有三个元组。第一个元组没有条件表达式，只有一个不蕴含权限 P 的权限表达式。

第二个元组中有一个不满足的立即条件式 IC0，因此，不考虑这个元组的权限表达式。

最后一个元组包含了一个延迟计算元组 PC2，在它的权限表达式中蕴含了权限 P。这时候还不可能做出判断，因此元组 C3 的计算延迟进行。但是，还是返回 true 表示这个 bundle 是潜在允许的。

```
B: { [IC1] [PC1] [PC2]  (P)  (R) }
   { [PC2]              (P)  (R) }
   {                    (Q)      }
```

接下来，考虑 bundle B。在第一个元组中，有立即条件对象 IC1。这个条件是满足的。将抵一个元组归入潜在候选是由于元组中还有两个延迟计算的条件。因此还是有必要检查这个元组是否是一个可能结果。而且由于权限表达式中蕴含权限 P，那么这个元组放置到延迟计算列表中。

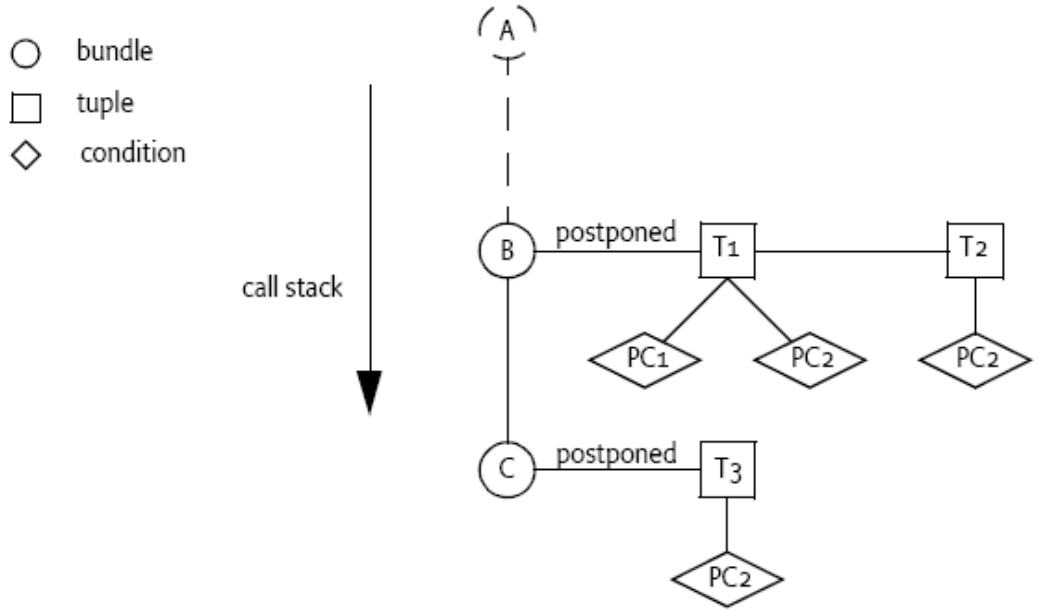
第二个元组也是类似的。由于它蕴含了权限 P，而且有一个延迟计算条件式 PC2，因此也必须将它放置到延迟计算列表。

对第三个元组的判断结果是拒绝的，这是由于它没有蕴含权限 P。但是，由于存在两个延迟计算元组，因此这个 bundle 也是潜在允许的。

```
A: {  [IC1]  [PC1]      (P)  (Q) }
   {  [IC2]             (P)  (R) }
   {                    (S)      }
```

对于上述例子，首先计算 bundle A 的 IC1 表达式，并且结果是满足的。而在 A 的权限表达式中蕴含了权限 P，因此延迟计算这个元组。但是，第二个元组的立即条件表达式也满足，而且蕴含了 P，所以，对删除对第一个元组的延迟计算，而是直接就允许 bundle A，同时也不必考虑了第三个元组。

调用访问控制上下文的 checkPermission 方法之后，框架的安全管理器必须要计算所有的延迟计算元组。延迟计算元组如图 50 所示：



现在框架安全管理器必须计算排列 (T1,T3) 或者 (T3,T2)。PC2的条件类同时在两个元组中出现，因此，必须将T1:PC2和T3:PC2组成一个带有版本标记的组来使用isSatisfied(Condition[], Dictionary)方法计算。而PC1是必须匹配的，它的计算是使用不带组标记的isSatisfied()方法。

长远来说，假设排列(T1, T3)的计算结果返回为失败，虽然对用户进行了交互而且用户同意了权限，那么PC1的返回结果还是失败。而在PC2中的Dictionary对象上增加标记防止再次询问用户同样的权限确认问题。

然后，框架安全管理器尝试序列(T2,T3)。必须通过方法isSatisfied(Condition[],Dictionary)来计算T2:PC2和T3:PC2。在上一个排列的计算中已经调用了这个方法，而且在Dictionary对象中保留了计算信息。而且，由于Dictionary对象中保留了原来调用的标记。再次调用时，方法发现了这个标记，从而直接返回true而不是再次询问用户。也就是说序列(T2,T3)是匹配的，从而框架安全管理器的checkPermission方法返回结果为成功。

9.5.3. 直接使用上下文访问控制

bundle 的开发人员可以直接使用标准 Java API 来完成安全检查。但是，如果直接使用访问控制器（Access Controller）来代替安全管理器进行安全检查（或者是使用访问控制上下文，Access Control Context），那么就不能处理延迟判断的条件式。因此，在这种情况下，bundle 保护域必须将延迟条件式看作是立即条件式一样同等处理。

上文中隐含了这样一个事实：权限检查的结果依赖于检查的实现策略。一般说来，安全管理器的 checkPermission 方法可访问控制上下文的 checkPermission 方法的执行过程是不一样的，依赖于运行的检查策略。

例如，栈中的一个 bundle 需要权限 P，而 P 和一个用户提示条件式相关，而另一个 bundle 没有权限 P。如果是调用安全管理器，那么检测结果是失败的，而且不会提示用户进行交互。如果直接调用访问控制上下文，那么用户将在返回检测结果为失败之前获得提示来交互。

9.5.4. 优化

理论上，每一个 checkPermission 方法必须要计算调用栈中的每一个 bundle。也就是，矿井安全管理器必须遍历栈中的所有 bundle，对这个 bundle 检查权限表中的每一个元组，计算所有的条件式，测试所有的权限式，直到找到了隐含的权限。这样的模型的计算代价是非常大的。

因此，框架实现者必须要尽可能的优化计算的过程。他们可以修改本规范描述的算法而只要保持外在的表现是一致的。

第一步的优化就是对权限表的裁剪。如果条件是固定的那么这样的条件对象是可以裁剪的。

如果满足一个固定的条件对象，那么就可以移除这个条件对象，这是由于它对计算结果不会再有影响。如果不满足，那么就可以不管这样的条件式，这是由于不满足这个条件式，使得元组也是不可能用的。

如果裁剪后元组中不存在其他元组，那么就可以将所有的这样的权限集中到一个权限对象中（包括了 PermissionCollection 对象）。那么 Java 2 安全机制中就有高度优化的代码来进行权限检查了。

例如，假设有以下权限表格：

```
{
   [ ...BundleLocationCondition
        "http://www.acme.com/*" ]
   ( ...SocketPermission "www.acme.com" )
} {
   [ ...BundleLocationCondition
        "http://www.et.com/*" ]
   [ ...Prompt "Call home?" ]
   ( ...SocketPermission "www.et.com" )
}
```

假设这个表格是通过一个来自 http://www.acme.com/bundle.jar 的 bundle 而初始化的。第一个元组的权限可以使用一个特定的权限集合来代替，这是由于 Bundle Location 的条件是固定的，而且满足这个条件。

而第二个元组可以删除，这是由于它的条件式是永远都不会满足的。

9.6. 权限管理

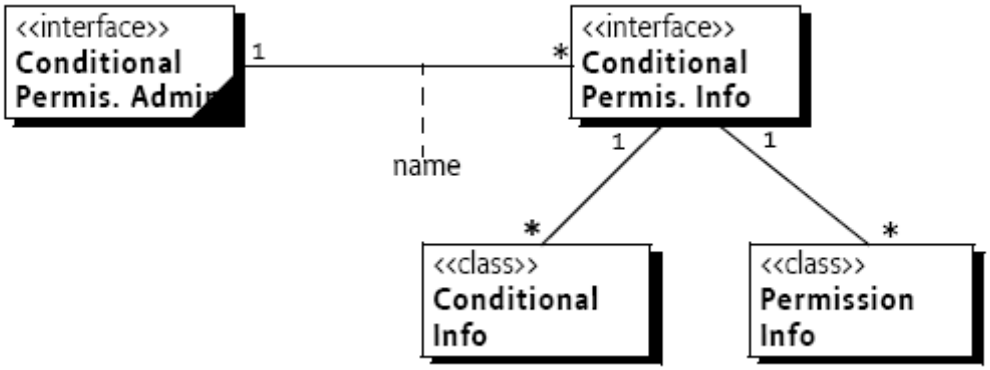
系统权限通过条件权限管理服务（Conditional Permission Admin service）来进行管理。条件权限管理也可以注册权限管理服务，这种情况下，这两种服务的相互作用必须根据下文中与权限管理的关系一节中所描述的方式。

通过使用方法 addConditionalPermissionInfo(ConditionInfo[],PermissionInfo[]) 或者方法 setConditionalPermissionInfo(String,ConditionInfo[],PermissionInfo[])来将权限添加到元组的权限编码中。这两种方法的不同在于参数名称的不同。

可以将 ConditionalPermissionInfo 对象匿名或者自命名添加权限。每一个 ConditionalPermissionInfo 对象都具有一个惟一名称用于区分其他 ConditionalPermissionInfo 对象，同时也将它和一个管理服务器区分开来。如果没有指定名称或者是名称为空，那么条件权

限管理服务将自动为它创建一个名称。

上述方法都是返回一个 ConditionalPermissionInfo 对象，可以通过这个对象来访问元组。这些对象也可以通过方法 getConditionalPermissionInfos()来进行枚举访问，或者是指定一个名称来调用方法 getConditionalPermissionInfo(String)访问。可以通过方法 getName()获得返回对象的名称。可以通过方法 ConditionalPermissionInfo.delete()来删除一个 ConditionalPermissionInfo 对象。在元组中包含了一个 ConditionInfo 对象数组和一个 PermissionInfo 数组，如图 51 所描述的：



ConditionalInfo 对象和 PermissionInfo 对象都可以通过编码字符串来组建。字符串的编码格式如下：

```
ConditionalInfo ::= '[' qname ( quoted ) * ']' // See 1.4.2
PermissionInfo  ::= '(' qname quoted quoted ')'
```

编码中不考虑多余的空格符号。

作为一种约定，ConditionalPermissionInfo 对象是包含在花括号中的：('{}', \u007B, \u007D)。而且没有编码和解码的方法调用。下面的例子是读取按照约定包含在花括号中的条件权限流的代码片段。解析是一行一行的解析处理。删除每一行中多余的空格，然后和控制字符相比较。如果找到了匹配的的‘}’那么也就设置完成了当前的条件和权限。

```
ConditionInfo   EMPTY_CS = new ConditionInfo[0];
PermissionInfo  EMPTY_PS = new PermissionInfo[0];
```

```
public void setConditionalPermissionInfo(
   ConditionalPermissionAdmin admin,
   InputStream in) throws Exception {

   Vector conditions = null;
   Vector permissions = null;

   InputStreamReader ir = new InputStreamReader(
      in, "UTF-8");
   BufferedReader br = new BufferedReader(ir);

   String line = br.readLine();

   while (line != null) {
      line = line.trim();
      switch (line.charAt(0)) {
         case '[' :
            conditions.add(new ConditionInfo(line));
            break;

         case '(' :
            permissions.add(new PermissionInfo(line));
            break;

         case '#':
            break;

         case '{':
            conditions = new Vector();
            permissions = new Vector();
            break;

         case '}' :
            admin.addConditionalPermissionInfo(
                  (ConditionInfo[])
                     conditions.toArray(EMPTY_CS),
                  (PermissionInfo[])
                     permissions.toArray(EMPTY_PS));
            conditions = permissions = null;
            break;

         default:
            throw new RuntimeException("Invalid format");

      }
      line = br.readLine();
   }
}
```

9.6.1. 缺省权限

条件权限管理中没有缺省权限的情况。缺省权限是从不包括任何条件表达式对象的ConditionalPermissionInfo对象中导出的。这些ConditionalPermissionInfo对象蕴含了所有的bundle，可以有效的处理缺省权限。这是和权限管理（Permission Admin）中不一样的地方；在权限管理中，只有当没有指定权限集合时才会使用缺省权限。

9.7. 条件式

条件式的目的在于确定是否应有一个权限集合。也就是说扮演着守卫的角色。因此，当对照 bundle 的有效权限来检查权限对象时必须计算条件。

Condition 对象可以通过它的方法 isSatisfied()来获得对象的状态。如果返回 true，那么就称之为满足条件，如果抛出了异常，那么应该在日志中记录异常信息，而且将这种情况看作是不满足条件的情况。

如果在同一个权限检查中某个特定的 Condition 对象重复进行了多次计算，那么可以将这个过程进行优化。例如，在权限检查中，可能需要多次进行用户提示确认，而用户的输入最好的一次完成。这些条件式称之为延迟条件式（*postponed conditions*）。可以立即得出值的条件式称之为立即条件式（*immediate conditions*）。可以通过方法 isPostponed()来判断一个条件式是否需要延迟计算。。如果方法返回 false，那么就可以马上调用 isSatisfied 方法来进行判断，否则，只能等到权限检查的最后阶段调用 isSatisfied 方法。

例如，一个条件式是用于判断移动电话是否处于漫游状态的。可以很容易在内存中得到这些信息，那么方法 isPostponed()通常是返回 false。对应的，一个通过网络获得授权的条件式应该在权限检查时最多只进行一次计算。对于这样的条件式 isPostponed 方法应该返回 true，以便于在权限检查的最后阶段来统一进行这些 Condition 对象的验证。

只有当条件式的答案是变化的时候才需要对 Condition 对象进行多次计算。这种 Condition 对象的可满足性是可变化的特性称之为易变性，可以通过方法 isMutable()来检查这个特性。如果条件式结果是固定的，那么条件权限管理就可以通过对权限表中的元组进行剪枝来获得显著的优化。例如，bundle 保护域可以将权限表中的包含了不满足的立即条件对象的元组移除。由于，这样的条件式是不可能满足的，因此在 bundle 保护域中可以不管这样的权限元组。

这种优化是由提供的 BundleLocationCondition 和 BundleignerCondition 对象共同决定。在保护域中不会考虑和保护域的 bundle 不匹配的条件权限。但是，Condition 对象也许启动的时候是可变的，而以后可以变成是固定的。例如，一个用户命令行提示可能具有以下状态：

- *Prompt* – 必须要对用户进行提示以获得答案，然后条件权限管理来根据答案判断是否满足条件。
- *Blanket* – 由于用户在以前的提示中指出了以后遇到同样的提示也采用同样的处理方式。在这种状态下，Condition对象是固定不变的。

对于指定bundle，规范提供了一系列的Condition对象来绑定权限。但是，客户化的代码也可以提供条件式。下文中描述了怎样创建条件类，并定义了标准的Condition类。

9.7.1. 客户化条件式

如果根据 bundle 保护域来初始化权限表，那么 Condition 对象是从 ConditionInfo 对象中构建的。ConditionInfo 对象支持一系列参数。框架的安全管理器必须要通过反射来从实现了

Condition 的类中查找一个公有的静态 getCondition 方法，这个 Condition 的实现类有两个属性：一个 Bundle 对象和一个 ConditionInfo 对象。getCondition 方法返回一个实现了 Condition 接口的对象。

但是，并不是要求这必须是一个新对象，如果需要，getCondition 方法也可以重用对象。例如，有一个 Bundle Location Condition 对象是不易变的，因此只由两个实例对象：一个分配给与指定位置匹配的 bundle，另一个分配给其他 bundle。这种情况下，通过比较 bundle 的位置和给定的参数信息来确定返回哪一个实例对象。

Condition 接口提供这样两种实例的模式是一种非常常见的模式。

- TRUE –计算结果通常为 true，而且从来不会延迟计算的条件对象。
- FALSE – 计算结果为 false，而且从来不会延迟计算的条件对象。

如果没有找到静态的 getCondition 方法，那么条件权限管理服务必须要尝试找到一个公有的使用一个 Bundle 对象和一个 ConditionInfo 对象作为参数的构造方法。条件权限管理必须查找：

```
public static Condition com.acme.AcmeCondition.getCondition(
      Bundle, ConditionInfo )
public com.acme.AcmeCondition( Bundle, ConditionInfo )
```

如果不能创建一个条件对象，那么必须将给定的条件对象看作是一个 Condition.FALSE 对象，而且在日志中记录一个错误。

对于一个 bundle 保护域来说，Condition 对象是惟一的，在实例编码一节中进行了解释。因此，在 Condition 对象上的任何查询将以给定的 Bundle 作为上下文。如果框架安全管理器（Framework Security Manager）不能够找到一种合适的方法来构造 Condition 对象，那么应该记录这个事件，而且要确保条件结果为 false。

另一个需要考虑的方面是计算的时间和易变性。

廉价的 Condition 对象是不可变的；而且几乎没有计算开销。如果一个 Condition 对象在创建之后就是不可变的，那么框架安全管理器就可以立即简化以后的计算操作。也就是，如果不满足一个不可变的 Condition 对象，那么就可以不考虑上层元组，甚至可以不需要实例化一些 Condition 和 Permission 对象。

易变的 Condition 对象必须要在权限检查时进行计算。权限检查是非常常见的操作，因此需要对权限计算进行优化。应该避免不必要的权限检查。易变条件属于系统代码，必须要将它设计为在一种受限环境下工作。应该将方法 isSatisfied()设计为是立即返回的。应该是根据变量来确定结果，而且要最小化副作用。

但是，有时候副作用是必需的；一个典型的例子就是用户提示。就如在权限检查算法[P220]一节中讨论的那样，对方法 isSatisfied()的计算是延迟到检查的最后阶段进行。为了进行延迟，Condition 对象必须要对方法 isPostponed()返回 true。

延迟计算的 Condition 对象必须要通过实现接口方法 isSatisfied(Condition[],Dictionary)来对计算进行优化。在这个方法中，必须要计算一系列的条件式；这和指定的接口是不相关的。通过将计算分组进行可以最小化用户提示，最小化网络访问等。

下面的代码是检测一个操作是否由网络服务器进行授权。这是一个分组计算，在请求服务器校验之前，将所有的请求分组。其中的 Host 类代码是抽象的，在这里也没有展示。

```
public class HostCondition implements Condition {
    String action;

    public HostCondition( Bundle, ConditionInfo info ) {
        action = info.getArgs()[0];
    }

    public boolean isSatisfied() { return false; }
    public boolean isPostponed() { return true; }
```

```
        public boolean isImmutable() { return false; }

        static Host host = new Host();

        public synchronized boolean isSatisfied(Condition[] conditions, Dictionary state ) {
            Set granted = (Set) state.get("granted");
            if ( granted == null ) {
                granted = new TreeSet();
                state.put("granted", granted );
            }
            Set pending = new TreeSet();
            for ( int i=0; i<conditions.length; i++ ) {
                String a = ((HostCondition)conditions[i]).action;
                if ( !granted.contains(a) )
                pending.add( a );
            }
            if ( pending.isEmpty() )
                return true;

            if ( ! host.permits( pending ) )
                return false;

            granted.addAll( pending );
            return true;
        }
    }
```

在 Host Condition 中有如下的 Condition Info：

```
    [ HostCondition "payment" ]
```

在 isSatisfied 方法中包含了大部分代码，这个方法中有一个 Condition 对象数组的参数。而构造方法中只存储了操作（action）。

首先 isSatisfied 方法获得授予的权限集合。第一次调用这个方法的时候，这个集合并不存在，然后就创建并存储在 state 中，以后调用就可以直接使用这个 state 了。

然后，创建一个临时的集合 pending，保存需要检查的条件的所有操作，并且减去所有在调用安全管理器（Security Manager）的 checkPermission 方法时，已经授予的操作权限。如果由于已经授予了所有的操作而导致 pending 列表为空，返回 true。否则，查询服务器。如果服务器允许操作，那么将 pending 列表中的操作添加到 state 中的 granted 集合。

9.7.2. 实现问题

9.7.2.1. 在条件式中使用权限检查

如果在调用方法 isSatisfied 的代码中有几次进行权限检查，那么 Condition 的实现者应该使用 AccessController doPrivileged 来确保需要的权限。例如，由于用户提示条件式需要和用户通过界面进行交互，也就潜在的导致了很多权限检查。

但是，对于同一个 Condition 对象必须不能递归计算。框架必须要检测对条件式的递归计算，对第二次的调用返回一个不满足，而不是延迟计算 Condition 对象。

例如，如果计算一个用户提示条件式，而且需要访问 UI，这样会导致对同样的用户提示 Condition 的计算，那么就不要进行第二次的计算，而是不对条件式延迟计算，直接返回 false。

9.7.2.2. 线程

Condition 的实现需要保证对一个单独的 checkPermission 调用应该是发生在同一个线程中。但是，多权限的检查可以是在不同的线程中进行。Condition 类的实现必须要确保这些同步问题可以得到解决。

9.7.2.3. 类加载

所有的条件式都是来自启动类路径或者来自框架类加载器。这是由于安全原因同时也是由于实现包的多版本性。可以使用框架扩展 bundle 来下载带有 bundle 的条件式。

9.7.2.4. 条件式生命周期

当框架重新启动或者是创建了 bundle 保护域（Bundle Protection Domain），则对 Condition 对象实例化。框架的实现可以通过优化来使得在一个 bundle 保护域的生命周期中多次创建并销毁 Condition 对象。Condition 类的实现必须不能承担创建或者是间接引用了 Condition 类的职责。

9.8. 标准条件式

本标准提供了一系列的标准条件式。OSGi 规范中的 JAR 文件中包含了规范的类，在 JAR 文件中，也包含了关于这些条件式的非功能实现。框架的实现者必须实现这些类，并替代原来的非功能实现类，这些类的实现和框架的有效性相关。

9.8.1. Bundle 签名条件式

当条件式的相关 bundle 是采用了和条件式参数匹配的证书进行签名时候，那么这个 bundle 签名条件式是满足的。也就是，这个条件式可以用于来给按照指定规则进行签名的 bundle 分配权限，必须通过静态方法 getCondition(Bundle,ConditionInfo)来创建 bundle 签名条件式。其中的字符串参数是和证书匹配中定义的 DN 名称进行匹配。例如：

```
[ ...BundleignerCondition "* ;cn=S&V, o=Tweety Inc., c=US"]
```

bundle 签名条件式不是易变的，而且可以在调用 getCondition 方法时进行全部计算。

9.8.2. Bundle 位置条件式

bundle 位置条件式中的参数和相关 bundle 的位置参数进行匹配。bundle 位置的匹配提供了很多签名的优势而且没有额外开销。但是，使用位置来进行校验需要确保下载的位置是安全的，而且不能被诱骗。例如，操作者也许会允许可以从指定位置下载对某个企业的相关 bundle。为了使这样的下载安全，最少应该使用 HTTPS 协议。然后操作者才可以使用位置来分配权限。

```
https://www.acme.com/download/* Apps from ACME
https://www.operator.com/download/* Operator apps
```

必须要通过它的静态方法 getCondition(Bundle,ConditionInfo)来创建 bundle 位置条件式。字

符串参数是一个位置字符串，可以使用通配符“*”。通配符的匹配是采用过滤器字符串匹配机制。例如：

```
http://www.acme.com/*
*://www.acme.com/*
```

当这个位置参数和 bundle 的位置字符串匹配的时候，bundle 位置条件式满足。

bundle 位置条件式不是易变的，而且可以在调用 getCondition 方法时进行全部计算。

9.9. bundle 权限资源

bundle 可以使用文件 OSGI-INF/permissions.perm 来传送局部权限。这必须是一个采用 UTF-8 编码的文件，文件格式采用行形式，对于每一行不限制长度但是必须要可以通过 BufferedReader 的 readLine 方法来读取：

```
permission.perm ::= line *
line ::= ( comment | pinfo ) '\r\n'
comment ::= ( '#' | '//' | /* blank */ )
pinfo ::= '(' qname quoted-string
quoted-string ')'
// 参阅 1.4.2
```

每一个权限必须要是在一行中使用 Permission Info 的格式进行描述。在行中可以使用注释。使用#和//开始的行是注释行，行的起始空格是忽略处理的。引号外的多个空格看作是一个空格。

例如（…为适当的包前缀）：

```
# Friday, Feb 24 2005
# ACME, chess game
( ..ServicePermission "..log.LogService" "GET" )
( ..PackagePermission "..log" "IMPORT" )
( ..ServicePermission "..cm.ManagedService" "REGISTER" )
( ..PackagePermission "..cm" "IMPORT" )
( ..ServicePermission "..useradmin.UserAdmin" "GET" )
( ..PackagePermission "..cm" "SET" )
( ..PackagePermission "com.acme.chess" "IMPORT,EXPORT" )
( ..PackagePermission "com.acme.score" "IMPORT" )
```

如果再 bundle 的 JAR 文件中有如上文件，那么将对本地权限进行设置。

如果没有上述文件，那么本地权限为所有权限。

9.9.1. 移除 bundle 权限资源

攻击者可以通过删除 bundle 中的 permissions.perm 文件来获得本地权限。这样也会删除 bundle 的签名者需要的权限。为了防止这种攻击，条件权限管理（Conditional Permission Admin）必须检测到 permissions.perm 资源，也就是，已经在清单文件中列出，但是却不在 JAR 中。如果检测时，正在进行 bundle 安装，那么安装过程必将失败，而且抛出 bundle 异常。

9.10. 与权限管理的关系

如果框架提供了条件权限管理服务和权限管理服务，那么权限管理服务提供的信息必须覆盖条件权限管理服务提供的相同信息，只有在权限管理服务没有提供的情况下，条件权限管理

才会生效。

权限管理定义了缺省权限的概念，而条件权限管理不支持缺省权限。缺省权限使用一组空的条件来模仿。条件式空集应用于所有的 bundle，除此之外，还有一些具体的条件式。这和权限管理是不一样的，在权限管理中，只有当 bundle 没有位置界限权限时才应用缺省权限。因此，如果存在条件权限管理，那么是决不会使用权限管理的缺省条件式的。

新的应用应该使用条件权限管理服务。在以后的发布版本中，将废弃权限管理服务。

9.11. 安全

9.11.1. 服务注册安全

9.11.1.1.条件权限管理服务

条件权限管理服务应该是框架的一部分，因此具有所有的权限。

9.11.1.2.客户端

```
ServicePermission    ..ConditionalPermissionAdmin GET
PackagePermission    ..condpermadmin              IMPORT
AllPermission
```

设置权限的条件权限服务客户端必须要用有所有的权限，这是由于它们可以将所有权限赋予给其他 bundle。

9.12. org.osgi.service.condpermadmin

OSGi 条件权限管理规范版本 1.0。

需要使用这个包的 bundle 必须在清单头标 Import-Package 中列出这个包，例如：

The OSGi Conditional Permission Admin Specification Version 1.0.

Import-Package: org.osgi.service.condpermadmin; version=1.0

9.12.1. 总结

- BundleLocationCondition – 测试bundle的位置是否和一个模式串匹配的条件式。[p.237]
- BundleignerCondition – 用于测试 bundle 的签名者是否和一个模式串匹配的条件式。[p.237]
- Condition – 条件式接口。 [p.238]
- ConditionalPermissionAdmin – 为管理条件权限的框架服务。[p.239]
- ConditionalPermissionInfo – 绑定了一系列权限的条件式集合。[p.240]
- ConditionInfo – 使用条件权限管理服务的条件式表达形式。[p.241]

9.12.2. public class BundleLocationCondition

测试 bundle 的位置是否和模式串匹配的条件式。模式匹配使用过滤字符串的匹配机制。

9.12.2.1.public static Condition getCondition(Bundle bundle, ConditionInfo info)

bundle 待计算的 bundle

info 构造条件式的 ConditionInfo 对象。这个 ConditionInfo 对象的参数必须是一个单独的字符串指定了匹配的位置模式串。根据过滤器字符串匹配机制进行匹配。其中“*”当作是通配符，除非使用了“\”转义。

□ 构造一个条件式来尝试对传递的 bundle 的位置和位置模式串进行匹配。

Returns 请求条件的条件式对象。

9.12.3. public class BundleignerCondition

用于测试 bundle 签名者是否和模式串匹配的条件式。由于只有当更新 bundle 时才可以改变 bundle 的签名者，所以这是一个不易变的条件式。

条件式中使用一个 DN 名称链字符串来匹配 bundle 的签名者。DN 名称的编码使用 IETF RFC 2253。通常，签名者使用证书授权来发布证书，证书授权中也包括一个对应的 DN 和证书。可以形成一个可信的证书授权链，最后的 DN 和证书是框架所认可的。bundle 的签名者的描述如下：签名者的 DN，然后是发行者的 DN，然后是发行发行者的 DN，直到一个根证书的 DN。对于每一个 DN，用分号隔开。

如果 bundle 的一个签名者具有一个 DN 链和构造条件式的 DN 链匹配，那么 bundle 可以满足这个条件式。可以使用通配符（“*”），这样在指定 DN 链中的匹配具有更大的灵活性。通配符可以用来代替 DN 名称，RDN 名称或者是一个 RDN 的值。如果使用通配符来代替一个 RDN 的值，那么这个通配符必须是“*”，那么可以匹配 RDN 中相应类型的所有值。如果通配符用来代替一个 RDN，那么必须是第一个 RDN，而且可以匹配任意数目的 RDNs（包括 0 个 RDNs）。

9.12.3.1.public static Condition getCondition(Bundle bundle, ConditionInfo info)

bundle 待计算的 bundle

info 用于构造条件式的 ConditionInfo 对象。ConditionInfo 对象的参数是一个字符串，指定了和 bundle 签名者进行匹配的 DN 名称链。

□ 构造一个条件式，尝试对传递的 bundle 的位置和给定的位置模式串进行匹配。

Returns 对指定 bundle 的签名者进行检查的条件式

9.12.4. public interface Condition

条件式的接口。使用条件权限信息（ConditionalPermissionInfo）来限制权限。其中，ConditionalPermissionInfo 的权限只能在相关条件式满足的情况下使用。

9.12.4.1.public static final Condition FALSE

条件式对象，计算结果为 false，而且从不延迟计算。

9.12.4.2.public static final Condition TRUE

条件式对象，计算结果为 true，而且从不延迟计算。

9.12.4.3.public boolean isMutable( )

- 返回条件式是否为易变的。

Returns 如果返回 true，表示 isSatisfied()[p.239]方法的返回值是可以改变的；否则 false 表示这个方法的返回值是不能改变的。

9.12.4.4.public boolean isPostponed( )

- 返回计算是否必须要延迟到权限检查的最后进行。如果对条件式的计算必须延迟到权限检查最后进行，那么返回 true，如果返回 false，那么条件式必须对方法 isSatisfied()[p.239]的调用能直接返回值。也就是说，方法,isSatisfied()直接返回，而不需要进行其他考虑，如用户的同意等。

Returns 返回 true 表示计算必须要延迟进行，否则，false 表示对计算必须要能直接运行。

9.12.4.5.public boolean isSatisfied( )

- 返回是否满足条件式。

Returns 如果满足条件式，返回 true，否则返回 false

9.12.4.6.public boolean isSatisfied(Condition[] conditions, Dictionary context)

conditions 条件数组

context 实现者可以用来跟踪状态的 Dictionary 对象。如果在同一个权限计算中多次调用这个方法，那么将多次传递这个 Dictionary 对象。安全管理器将这个 Dictionary 对象看作是一个不传导的对象，如果需要多次调用，那么安全管理器简单创建一个空的 Dictionary 来传递给子调用。

- 返回是否满足 Condition 集合。尽管这个方法不是静态的，但是必须要实现为静态形式。所有传递的条件式具有同样的类型，而且和调用这个方法的对象的类型相应。

Returns 如果所有的条件式满足，返回 true，否则，只要有一个条件式不满足，返回 false。

9.12.5. public interface ConditionalPermissionAdmin

管理条件权限的框架服务。可以将条件权限添加到框架，从框架中获得和删除条件权限。

9.12.5.1.public ConditionalPermissionInfo addConditionalPermissionInfo(ConditionInfo[] conds, PermissionInfo[] perms)

conds 为具有相应权限而需要满足的条件式集合
perms 当满足相应条件式而具有的权限
□ 川剧一个新的条件权限信息（ConditionalPermissionInfo）对象。这个对象具有惟一的，不会重用的名称。
Returns 指定条件和权限的 ConditionalPermissionInfo 对象
Throws SecurityException – 如果调用这没有 AllPermission 权限

9.12.5.2.public AccessControlContext getAccessControlContext(String[] signers)

signers 返回的访问控制上下文（AccessControlContext）对象的签名者
□ 返回指定签名者的 AccessControlContext 对象。
Returns 具有和指定签名者相关权限的 AccessControlContext 对象。

9.12.5.3.public ConditionalPermissionInfo getConditionalPermissionInfo(String name)

name 返回的 ConditionalPermissionInfo 对象的名称
□ 返回指定名称的 ConditionalPermissionInfo 对象
Returns 指定名称的 ConditionalPermissionInfo 对象

9.12.5.4.public Enumeration getConditionalPermissionInfos( )

□ 返回由当前的条件权限管理（ConditionalPermissionAdmin）对象管理的条件权限信息（ConditionalPermissionInfos）对象集合。调用这个方法 ConditionalPermissionInfo.delete()[p.240]将删除从条件权限管理对象中删除这个条件权限信息对象。
Returns 当前的 ConditionalPermissionAdmin 管理的 ConditionalPermissionInfos 对象的枚举

9.12.5.5.public ConditionalPermissionInfo setConditionalPermissionInfo(String name, ConditionInfo[] conds, PermissionInfo[] perms)

name 条件权限信息对象的名称，或者为 null 值
conds 为了获得相应权限需要满足的条件式
perms 当满足了相应条件式后具有的权限
□ 通过一个指定的名称设置或者创建一个 ConditionalPermissionInfo 对象。如果指定的名称为 null，那么必须要创建一个 ConditionalPermissionInfo 对象，而且必须分配一个惟一的，不会重用的名称。如果没有和当前的名称匹配的 ConditionalPermissionInfo 对象，那么必须使用指定的名称创建一个 ConditionalPermissionInfo 对象。否则，更新名称为指定名称的这个 ConditionalPermissionInfo 对象。
Returns 指定名称、条件式和权限的 ConditionalPermissionInfo 对象
Throws SecurityException – 如果调用这没有 AllPermission 权限

9.12.6. public interface ConditionalPermissionInfo

条件式集合和权限集合的汇集。这个接口的实例来自于条件权限管理服务（Conditional Permission Admin service）。

9.12.6.1.public void delete( )

□ 删除来自于 Conditional Permission Admin 的 ConditionalPermissionInfo 对象
Throws SecurityException – 如果调用这没有 AllPermission 权限

9.12.6.2.public ConditionInfo[] getConditionInfos( )

□ 返回为了获得权限必须满足的条件式的条件信息（ConditionInfo）
Returns 在这个条件权限信息（ConditionalPermissionInfo）对象中，条件式的条件信息（ConditionInfo）。

9.12.6.3.public String getName( )

□ 返回 ConditionalPermissionInfo 对象的名称
Returns ConditionalPermissionInfo 对象的名称

9.12.6.4.public PermissionInfo[] getPermissionInfos( )

□ 返回 ConditionalPermissionInfo 对象的权限的 PermissionInfo 对象集合
Returns ConditionalPermissionInfo 对象的权限的 PermissionInfo 对象集合

9.12.7. public class ConditionInfo

使用条件权限管理服务的条件表示。在这个类中封装了两部分信息：条件类型（类型名称，必须要是实现了 Condition 接口），和传递给构造方法的参数。

同时 Condition 类必须：

- 声明一个静态的 getCondition 方法，并且使用一个 Bundle 对象和一个 ConditionInfo 对象作为参数。这个方法返回一个实现了 Condition 接口的实例对象。
- 实现了 Condition 接口，定义了一个公有的带有一个 Bundle 对象和 ConditionInfo 对象的参数的构造方法。

9.12.7.1.public ConditionInfo(String type, String[] args)

type 这个 ConditionInfo 对象表示的 Condition 对象的类名称。

args Condition 类的参数。对于最近创建的 Condition 对象调用 getArgs()[p.242]方法的返回值为这个参数。

□ 通过指定类型和参数来创建一个 ConditionInfo 对象。

Throws NullPointerException – 如果类型 null。

9.12.7.2.public ConditionInfo(String encodedCondition)

encodedCondition 编码的 ConditionInfo。

□ 通过指定 ConditionInfo 对象的编码字符串来创建 ConditionInfo 对象。在编码中忽略处理空格。

Throws IllegalArgumentException – 如果参数 encodedCondition 具有不适当的格式。

See Also getEncoded[p.242]

9.12.7.3.public boolean equals(Object obj)

obj 与 ConditionInfo 对象相比较的对象。

□ 判断两个 ConditionInfo 对象是否相等。这个方法检查参数 obj 对象是否和这个 ConditionInfo 对象具有相同的类型和 args 参数。

Returns 如果 obj 是一个 ConditionInfo 对象，而且和这个 ConditionInfo 对象具有相同的类型和 args 参数，返回 true。
否则返回 false。

9.12.7.4.public final String[] getArgs( )

□ 返回这个 ConditionInfo 对象的 args 参数。

Returns 返回 ConditionInfo 对象的 args 参数。
如果这个 ConditionInfo 对象没有 args 参数，那么返回一个空的数组。

9.12.7.5.public final String getEncoded( )

- 返回这个 ConditionInfo 对象的编码字符串，字符串采用一个合适的格式来存储这个 ConditionInfo 对象。
  编码格式如下：
  [type “arg0” “arg1” ...]
  其中每一个 arg 都是相应语法分析的编码。对于换行符通过“\”来转义。在编码字符串中没有起始和终止的空格符。在 type 和“arg0”，以及 arg 之间都有一个空格符。

Returns ConditionInfo 对象的编码字符串。

9.12.7.6.public final String getType( )

- 返回这个 ConditionInfo 所代表的 Condition 类的全名类名称

Returns 这个 ConditionInfo 所代表的 Condition 类的全名类名称。

9.12.7.7.public int hashCode( )

- 返回这个 ConditionInfo 类的哈希码

Returns 这个 ConditionInfo 类的哈希码。

9.12.7.8.public String toString( )

- 返回这个 ConditionInfo 类的表示字符串。可以通过调用这个对象的 getEncoded 方法来创建这个字符串

Returns 这个 ConditionInfo 类的表示字符串。

9.13. 参考

[46] Java 1.3
http://java.sun.com/j2se/1.3

10. 权限管理服务规范

版本号： 1.2

10.1. 简介

在框架中，bundle 都有一个单独的权限集合。通过使用这些权限来验证 bundle 是否具有对一些特权代码的执行权限。例如，一个 FilePermission 定义了使用哪个文件以及如何使用文件的方法。

对 bundle 的权限授予的执行应该交由管理代理（Management Agent）来实行。由于这个原因，框架提供了权限管理服务，这样，管理代理就可以对 bundle 的权限进行管理，而且可以给 bundle 赋予缺省的权限。

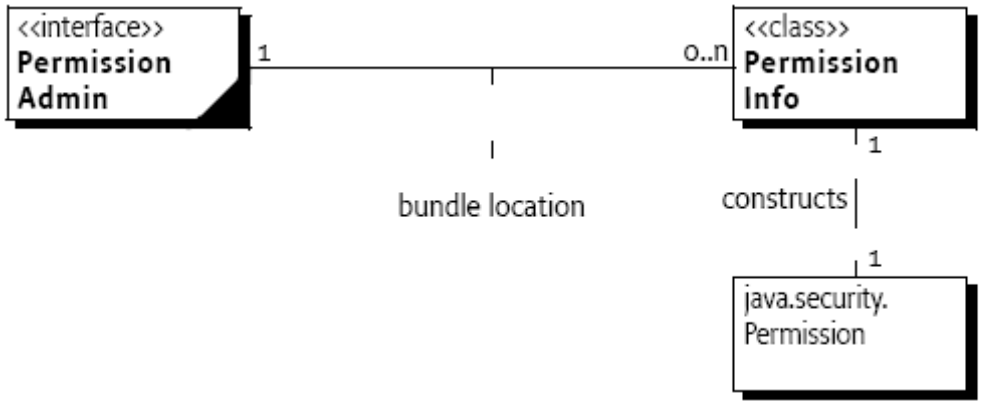
其他的框架相关机制在安全概要[P13]中进行了说明。

10.1.1. 要点

- 状态信息（Status information）– 权限管理服务必须要提供 bundle 的当前权限状态信息。
- 可管理性（Administrative）– 权限管理服务必须要允许管理代理在 bundle 的安装之前、安装进行中和安装完毕之后都可以进行权限设置。
- 缺省性（Defaults） – 权限管理服务必须提供对缺省权限的控制。这些是 bundle 不用指定而直接拥有的权限。

10.1.2. 实体

- PermissionAdmin – 提供了对框架权限库的访问的服务。
- PermissionInfo – 保存了构造一个 Permission 类所需要的信息的类。
- Bundle location – 描述了 bundle 的位置的字符串。在 bundle 标识符[81]一节中对它进行了详细描述。



10.1.3. 操作

框架包括了一个权限库。这些权限使用了 bunde 的位置字符串来标识。使用 bundle 位置字符串可以允许在下载 bundle 之前对权限进行设置。当框架需要某个 bundle 的权限时，需要从这个库中获得该 bundle 的权限。如果没有设置这个 bundle 的权限，那么就必须使用缺省权限。如果没有设置缺省权限，那么就使用 java.security.AllPermission 中设置的权限。如果修改了缺省权限，那么必须要立即重新启动那些未设置权限的 bundle，而且使用修改后的缺省权限。这是由于这些 bundle 使用了修改前的缺省权限。

框架的系统 bundle 通过接口 the org.osgi.service.permissionadmin.PermissionAdmin 来注册权限管理服务。这是一个可选的单态服务，因此，在任何时候都最多只有一个注册的权限管理服务。

权限管理服务提供了对权限库的访问。管理代理可以获得、修改、更新和删除从这个库中获得的权限。管理代理也可以通过使用同步 bundle 监听器（SynchronousBundleListener）对象，在安装和更新 bundle 的时候对 bundle 权限进行设置。

10.2. 权限管理服务

权限管理服务需要操作缺省权限和指定 bundle 的相关权限。缺省权限和指定 bundle 的相关权限都是持久存储的。我们可以在安装 bundle 之前来设置 bundle 的权限，这是由于使用了 bundle 的位置字符串来设置 bundle 权限。

但是，对 bundle 的权限操作却有可能是实时的，这有可能在下载 bundle 的时候完成，甚至在下载之前就完成。为了支持这种适应性要求，在管理代理就需要使用一个同步 bundle 监听器对象（SynchronousBundleListener）来检测安装或者是更新一个 bundle，同时也可以在安装完成之前就设置好 bundle 需要的权限。

在对 bundle 进行权限检查之前，权限就已经设置好的。这也就是说，如果一个 bundle 打开了一个文件，那么即使现在删除了 bundle 对这个文件的打开权限，但是 bundle 依然还是可以使用这个文件。

权限信息不是通过使用 java.security.Permission 来指定的。这是由于框架重新启动之后需要这些持久信息，以及框架的类加载器的机制的原因。实际的权限类（Permission）必须是 Permission 类的一个子类或者是可以从任一 bundle 中导出提供。在 bundle 导出之后，框架就可以访问这些权限了，而管理代理应该导入所有可能包含权限的包。这种需求将极大限制了权限的类型。因

此，权限管理服务使用权限信息类（PermissionInfo）来指定权限的信息。框架使用这个类的实例来创建 Permission 对象。

权限信息对象限定了可能使用的权限（Permission）对象。一个 Permission 类的子类可以使用一个 PermissionInfo 对象来描述，这个 Permission 类的子类必须要满足以下特性：

- 必须是 java.security.Permission 的一个子类。
- 必须拥有带有两个参数的公有构造方法，形如 type(name,actions)。
- 框架代码必须可以通过系统类路径或者是导出包来使用这个类，这样框架就可以加载这个类。
- 这个类必须是公有的。

如果不满足以上任意一个条件，那么忽略处理这个 PemissionInfo 对象，而且在日志中记录这个错误。

通常，将权限设置为一个 PemissionInfo 对象的数组，这样就可以自动的完成权限的赋值。

PermissionAdmin 接口提供了以下方法：

- getLocations() – 返回一个位置的列表，具有对这些位置的权限描述。管理代理使用这个方法来测试当前的权限集合。
- getPermissions(String) – 返回 PermissionInfo 对象的列表，这些 PermissionInfo 对象具有对指定位置的权限，如果没有设置权限，那么返回 null。
- setPermissions(String,PermissionInfo[]) – 将权限和一个指定位置关联。或者如果应该移除这个权限，则返回 null。
- getDefaultPermissions() – 这个方法返回缺省缺陷列表。
- setDefaultPermissions(PermissionInfo[]) – 设置缺省权限。

10.2.1. 相对路径名称的文件权限

通过 setPermissions 方法分配给 bundle 的一个 java.io.FilePermission 对象，如果这个 FilePermission 对象的路径参数是一个相对路径名称，那么必须要特殊处理这样的对象。相对路径名称不是绝对路径。可以参阅 java.io.File.isAbsolute 方法，获得更多的关于绝对路径名称的信息。

当给 bundle 分配一个带有相对路径名称的 FilePermission 对象时，在 bundle 的持久存储区中也是将这个路径看作是一个相对路径。这样允许附加的权限，例如分配了对 bundle 持久存储区中文件的执行权限。如下例：

```
java.io.FilePermission "-" "execute"
```

这样允许 bundle 执行 bundle 持久存储区中的任何文件。

上述说明只是应用于通过 setPermission 方法来分配 bundle 的 FilePermission 对象。而不适用于缺省的权限。不考虑通过 setDefaultPermission 方法来设置的带有相对路径的 FilePermission 对象。

10.3. 安全

权限管理服务是一个有缺陷的系统服务。具有访问和使用权限管理服务的 bundle 就可以对 OSGi 服务平台进行完全控制了。然而，很多 bundle 具有权限 ServicePermission[PermissionAdmin, GET]，这是由于所有的改变框架状态的方法都需要 AdminPermission 权限。

任何 bundle 都不能具有权限 ServicePermission[PermissionAdmin,REGISTER]，这是由于只有框架应该提供这样的服务。

10.4. 更改

下面描述的是由于和原来发布的版本的相关内容：

- 对权限信息（Permission Info）进行了更新，在编码格式中允许使用空格。
- 修改权限的包管理方法需要调用者具有 AllPermission 权限。由于如果只有一个简单指定的权限就可以修改 bundle 的权限，那么将会导致对 bundle 权限提升，甚至可以提升到 AllPermission。

10.5. org.osgi.service.permissionadmin

OSGi 权限管理服务包，版本规范：1.2。

需要使用这个包的 bundle 必须要导入这个包，通过在 bundle 清单文件中的 Import-Package 头标中描述。例如：

Import-Package: org.osgi.service.permissionadmin; version=1.2

10.5.1. 概要

- PermissionAdmin – 权限管理服务允许管理代理来对 bundle 的权限进行管理。[p.248]
- PermissionInfo – 权限管理服务中使用的权限的表示形式。[p.250]

10.5.2. public interface PermissionAdmin

权限管理服务允许管理代理来对 bundle 的权限进行管理。在 OSGi 环境中，最多只能有一个权限管理服务存在。

对权限管理服务的访问由 ServicePermission 权限来进行保护。另外设置权限时还需要有 AdminPermission。通过使用权限表来管理 bundle 的权限。将 bundle 的位置字符串用来作为权限表中的码。权限表的条目都是权限（类型为 PermissionInfo）的集合，这些权限授予给名称为指定位置的 bundle。在将 bundle 安装到框架中之前，bundle 也可能在权限表中有一个条目。

通过 setDefaultPermissions 设置的权限可以当作缺省权限使用，将这些权限赋予给所有那些在权限表中没有条目的 bundle。

在权限表中对 bundle 的任何权限修改将导致对 bundle 的 java.security.ProtectionDomain 进行权限检查，而且是持久性保存。

在权限检查时，只考虑在系统类路径中的权限类或者是从导出包中的权限类。另外，只能使用 java.security.Permission 类的子类，而且在这个子类中，定义了带有一个名称字符串参数和一个动作字符串参数的构造函数。

由框架授予的权限是不能修改的（例如，bundle 对于它的持久存储区的访问权），而且也不会影响到 getPermissions 和 getDefaultPermissions 返回的权限值。

10.5.2.1.public PermissionInfo[] getDefaultPermissions( )

- 获得缺省权限
  对于没有通过位置字符串来定义权限的 bundle，系统所授予的权限。

Returns 返回缺省权限，或者如果没有设置缺省权限，返回 null。

10.5.2.2.public String[] getLocations( )

- 返回 bundle 的位置字符串，根据这个字符串来定义权限，也就是，用在权限表的条目中。

Returns 返回分配了任何权限的 bundle 的位置字符串，如果权限表中为空，返回 null。

10.5.2.3.public PermissionInfo[] getPermissions(String location)

location 待返回权限的 bundle 的位置字符串

- 获得分配给指定位置的 bundle 的权限。

Returns 通过指定位置的 bundle 的权限，如果对于这个 bundle 没有分配任何权限，返回 null。

10.5.2.4.public void setDefaultPermissions(PermissionInfo[] permissions)

permissions 缺省权限，或者为 null，表示从权限表中删除缺省缺陷。

- 设置缺省权限。
  这些权限是授予给那些没有通过位置字符串来分配权限的 bundle。

Throws SecurityException – 如果调用者没有 AllPermission 权限。

10.5.2.5.public void setPermissions(String location, PermissionInfo[] permissions)

location 待设置权限的 bundle 的位置字符串

permissions 待设置的权限，或者如果删除指定 bundle 的权限，则为 null。

- 给通过位置字符串指定的 bundle 分配指定权限。

Throws SecurityException – 如果调用者没有 AllPermission 权限。

10.5.3. public class PermissionInfo

权限管理服务中使用的权限表示形式。

这个类中封装了三部分信息：Permission 类型（类名称），必须是 java.security.Permission 类的子类；从构造方法中传递过来的名称和操作参数。

为了实例化一个使用 PermissionInfo 表示的权限类，考虑在权限检查的时候，Permission 必须是通过系统类路径或者是一个导出包来提供。这也就是说对一个通过 PermissionInfo 表示的权限实例有可能要等到直到 bundle 导出了包含 Permission 类的包。

10.5.3.1.public PermissionInfo(String type, String name, String actions)

type 这个 PermissionInfo 类的表示的权限类的类名称。
必须是 java.security.Permission 类的子类，而且必须定义了带有两个参数的构造方法，其中一个参数是一个名称字符串，另一个参数是一个操作字符串。

name 权限名称，将这个名称作为 Permission 类的构造方法的第一个参数。
actions 权限操作，作为 Permission 类的构造方法的第二个参数。
□ 通过指定类型，名称和操作构造一个指定类型的 PermissionInfo 类。
Throws NullPointerException – 如果类型为 null。
IllegalArgumentException – 如果操作不为 null 而名称为 null。

10.5.3.2.public PermissionInfo(String encodedPermission)

encodedPermission 编码的 PermissionInfo
□ 从指定编码的 PermissionInfo 字符串中构造一个 PermissionInfo 类。不考虑编码中的空格。
Throws IllegalArgumentException – 如果 encodedPermission 的格式不正确。
See Also getEncoded[p.251]

10.5.3.3.public boolean equals(Object obj)

obj 和这个 PermissionInfo 类进行相等比较的类
□ 确定这两个 PermissionInfo 类是否相等。这个方法检查指定类是否和这个类具有相同的类型、名称、和操作。
Return 如果 obj 是一个 PermissionInfo 类，而且具有相同的类型、名称、和操作，那么返回 true。否则返回 false。

10.5.3.4.public final String getActions( )

□ 返回这个 PermissionInfo 表示的权限的操作。
Return 这个 PermissionInfo 表示的权限的操作，如果这个权限没有关联任何操作，返回 null。

10.5.3.5.public final String getEncoded( )

□ 返回这个 PermissionInfo 类的编码字符串。编码格式如下：
(type)
或者
(type “name”)
或者
(type “name” “actions”)
其中的名称和操作的编码采用语法分析。其中换行符分别使用\”，\\，\r 和 \n 来转义。
在编码中没有起始和结尾的空格，在类型和“name”以及“name”和“action”之间使用一个空格分隔。
Return PermissionInfo 类的编码字符串。

10.5.3.6.public final String getName( )

- 返回这个 PermissionInfo 表示的权限的名称。

Return 这个 PermissionInfo 类表示的权限的名称，或者如果这个权限没有名称，返回 null。

10.5.3.7.public final String getType( )

- 返回这个 PermissionInfo 表示的权限的全局类名称。

Return 这个 PermissionInfo 表示的权限的全局类名称。

10.5.3.8.public int hashCode( )

- 返回这个类的哈希码

Return 这个类的哈希码

10.5.3.9.public String toString( )

- 返回这个 PermissionInfo 表示的字符串。通过调用 getEncoded 方法来创建这个字符串。

Return 表示这个 PermissionInfo 类的字符串。

11. URL 处理服务规范

版本号： 1.0

11.1. 简介

本规范定义了如何注册一个新的 URL 类型，如何将一个 java.io.InputStream 类转换为一个指定的 Java 类。

本规范定义了对 Java 运行时的通过 bundle 来对 URL 资源类型和内容处理器的扩充处理标准机制。在 OSGi 平台中，支持对 URL 类型的动态扩展，这是 OSGi 服务平台的一个具有重要作用的特性。

这部分规范是非常必要的，这是由于标准 Java 中的通过新的类型和内容类型来扩展 URL 类的机制和 OSGi 服务平台的动态特性不相适应。在 Java 中，一次性的处理注册新的资源类型和 URL 内容类型，而且一旦注册之后，就不会取消对资源类型或者内容类型的注册。这样的单一的注册模式使得我们不可能在不同的独立的 bundle 之间来使用这样的机制。因此，OSGi 服务框架必须要实现一种新的机制来覆盖原有的机制。

R4 规范提出了一个标准的 Connector 服务，具有类似的功能，参考 IO Connector 服务规范 [P219]。

11.1.1. 要点

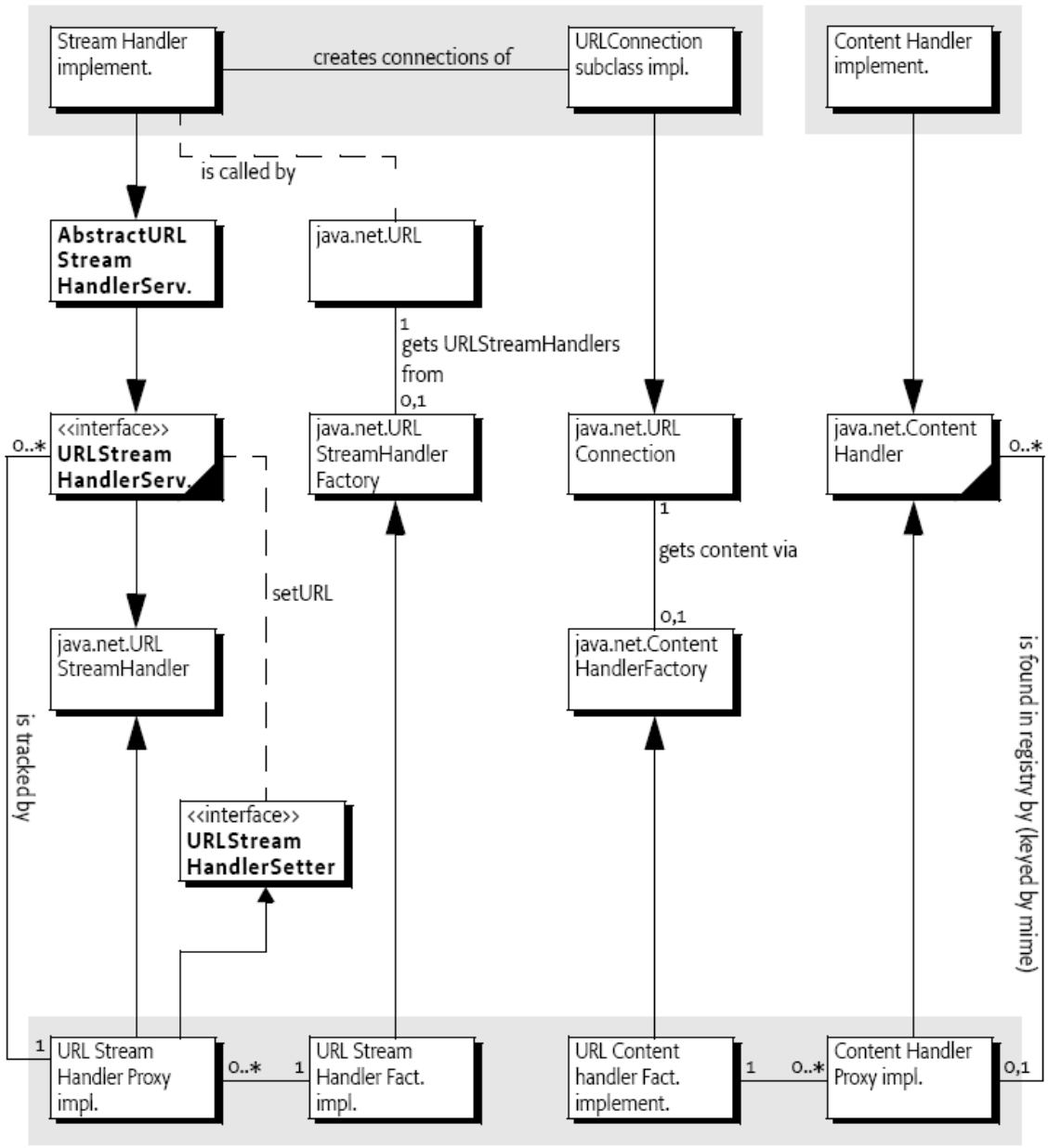
- 多样性（Multiple Access） – 多个 bundle 都可以注册 ContentHandler 类和 URLStreamHandler 类
- 支持现有类型（Existing Schemes Availability） – 不能覆盖在 OSGi 服务平台中的现有资源类型。
- 生命周期监控（life cycle Monitored） – 必须支持 bundle 的生命周期。当注册了资源类型处理器和内容类型处理处理器的 bundle 停止之后，服务不再可用。
- 简单性（Simplicity） – 如果 bundle 需要提供一个新的 URL 资源类型或者内容类型处理器，最小化 bundle 需要完成的工作。

11.1.2. 名词

- 资源类型（Scheme） – 指定协议的标志符。例如，“http”表示超文本传输协议。资源类型是 java.net.URLStreamHandler 的子类。
- 内容类型（Content Type） – 内容的类型标志符。内容类型一般都是引用了 MIME 类型。内容类型处理器是 java.net.ContentHandler 类的子类。
- 统一资源定位（URL） –java.net.URL 类的实例，具有一个资源类型名称和足够的参数来表示一个资源。

- 工厂对象（Factory）– 创建其他对象的对象。使用工厂对象的目的在于对调用者隐藏实现类（可能有多个）。创建的对象是一个指定类型的子类或者是实现类。
- 代理对象（Proxy） – 在 Java 中注册的一个类，将调用转发到在服务注册中心中的实现类。
- java.net.URLStreamHandler –java.net.URLStreamHandler 类的实例，可以创建一个指定协议的连接的 URLConnection 对象。
- 单一操作（Singleton Operation） – 只能执行一次的操作。
- URLStreamHandlerService – OSGi 服务接口，包含了的 URLStreamHandler 类的公有方法，因此，框架可以调用这些方法。
- AbstractURLStreamHandlerService – 接口 URLStreamHandlerService 的一个实现，通过调用超类的实现（java.net.url.URLStreamHandler)来实现这个接口的方法. 同时，这个类也实现了通过 java.net.URLStreamHandlerSetter 对 java.net.URL 类的设置。
- URLStreamHandlerSetter – 接口，其中定义了对 java.net.URL 类设置的抽象方法。这个接口和一个代理以及安全检查相关。
- java.net.URLStreamHandlerFactory – 工厂对象，注册了 java.net.URL 类。用于查找实现了在 Java 环境下没有的资源类型的 java.net.URLStreamHandler 类。在 Java 环境下只能注册一个 java.net.URLStreamHandlerFactory 类。
- java.net.URLConnection – 指定基于类型的协议的连接。当调用 java.net.URL.openConnection 时，通过 java.net.URLStreamHandler 对象来创建一个 java.net.URLConnection 类。
- java.net.ContentHandler – 将一个字节流转换而来的 Java 对象。这个 Java 类是基于字节流的 MIME 类型。
- java.net.ContentHandlerFactory – 工厂对象，通过创建一个需要的实例来扩展 java.net.URLConnection 提供的 java.net.ContentHandler 类。java.net.URLConnection 只能够注册一个 java.net.ContentHandlerFactory 类。
- MIME Type – 字节流格式的名称。参考 [49] MIME 多用途网际邮件扩充协议。Multipurpose Internet Mail Extension.

下图非常复杂，这是由于 Java 使用的复杂的策略来实现扩展流处理和内容处理。



11.1.3. 操作

实现了一个新的 URL 资源类型的 bundle 应该在 OSGi 框架中的 URLStreamHandlerService 接口下注册一个服务对象。这个接口包含了 java.net.URLStreamHandler 类的公有方法，因此可以通过代理器（通常在 Java 运行时注册的对象）来调用这些方法。

OSGi 框架实现中，必须在 java.net 中实现这个服务对象。这是由于只能调用一次方法 java.net.URL.setStreamHandlerFactory。这样过去和以后的 bundle 都不能使用它了。

对一个内容标记的流进行转化的 bundle 应该在 java.net.ContentHandler 下注册一个服务对象。OSGi 框架应该提供这些对象给 java.net.URLConnection 类。

11.2. java.net 中的工厂模式

Java 提供了 OSGi 框架和很多 OSGi 服务平台运行的 bundle 所使用的 java.net.URL 类，使用 URL 类的一个关键好处在于简化了将 URL 字符串转化为对资源请求的过程。

java.net.URL 类具有可扩展性，允许动态的通过 java.net.URLStreamHandlerFactory 添加新的资源类型（协议）和内容类型。现有的应用也可以通过这些新的处理器来使用新的资源类型和内容类型，使用方式和 Java 运行环境提供的处理器是相同的。在 Javadoc 的 URLStreamHandler 和 ContentHandler 中描述了这种机制。参阅文献[47] Java。

例如，通过 URL http://www.osgi.org/sample.txt 来定位一个 OSGi web 服务的一个文件，这个文件使用 HTTP 协议来获得（通常 Java 运行环境都提供了这样的资源类型）。这样的一个 URL：rsh://www.acme.com/agent.zip 定位了一个 ZIP 文件，可以通过 Java 环境中没有定义的 RSH 资源类型来获得。在成功使用 RSH 资源之前，必须要在 java.net.URL 类下注册一个 java.net.URLStreamHandlerFactory 对象。

只使用在 Java 中现有的 java.net.URL 资源处理器会导致一些问题：

- 工厂对象是单一操作 – 一个 java.net.URLStreamHandlerFactory 类只能使用 java.net.URL 注册一次。同样一个 java.net.ContentHandlerFactory 也只能使用 java.net.URLConnection 注册一次。也就是说，不能对工厂对象的取消注册或者是替代一个工厂对象。
- 资源类型的缓存（Caching Of Schemes） – 当 java.net.URL 类第一次使用一个原来没有使用过的资源类型，java.net.URL 类从当前注册的 java.net.URLStreamHandlerFactory 类中请求一个 java.net.URLStreamHandler 类。缓存返回的 java.net.URLStreamHandler 类，然后对这个类型的并发请求使用了同样的 java.net.URLStreamHandler 类。这也就是说，一旦构造了指定类型的处理器，就不能删除这个处理器，不能替换为一个新的 java.net.ContentHandler 类。

这两方面的问题都影响到了 OSGi 操作模型，这个模型允许一个 bundle 在不同的生命周期阶段中涉及暴露的服务、删除服务、更新代码、从另一个 bundle 中提供同样的服务来替代等等。bundle 并不能兼容现在的 Java 机制。

11.3. 框架规程

在 OSGi 框架中必须注册一个的 java.net.URLStreamHandlerFactory 对象和一个 java.net.ContentHandlerFactory 对象，其中包含的方法分别为：

java.net.URL.setURLStreamHandlerFactory
java.net.URLConnection.setContentHandlerFactory

当注册了这两个工厂对象，OSGi 框架服务注册中心必须跟踪 URLStreamHandlerService 服务和 java.net.ContentHandler 服务的注册。

一个 URL 流处理器服务（URL Stream Handler Service）必须要关联一个服务注册属性，属性名称为 URL_HANDLER_PROTOCOL。这个属性 url.handler.protocol 的值必须是一个资源类型数组（String []）。

内容处理器服务必须关联一个服务注册属性，属性名称为 URL_CONTENT_MIMETYPE。这个属性的值必须是一个 MIME 类型名称数组（String []），其中字符串的格式为类型/子类型。参阅文献[49]MIME 多用途网际邮件扩充协议。

11.3.1. 构建代理和处理器

当 URL 中使用一个原来没有使用过的资源类型时，查询已经注册的 java.net.URLStreamHandlerFactory 对象（应该已经在 OSGi 框架中注册）。然后，OSGi 框架必须查询服务注册中心注册在 URLStreamHandlerService 下的而且匹配请求资源类型的服务。

如果找到了一个或者多个服务对象，那么构造一个代理对象。必须使用代理对象，这是由于提供 java.net.URLStreamHandler 实现类的服务对象是未注册的，而且 Java 中没有提供一种机制来撤销一个从 java.net.URLStreamHandlerFactor 类返回的 java.net.URLStreamHandler 类。

一旦创建了代理对象，这个代理对象必须要跟踪服务注册中心中和它相关资源类型的注册和取消注册。代理对象必须关联和资源类型匹配，而且任何时候都具有最大的服务属性 org.osgi.framework.Constants.SERVICE_RANKING（参考服务属性[P.107]）都是具有最大值的服务。如果一个代理对象关联到一个 URL 流处理服务，则当具有更大的这个服务属性值的的服务注册之后，代理对象必须要关联到这样的最新注册的处理器。

代理对象必须将所有的方法请求发送到相关的 URL 流处理服务，直到这个服务变成未注册状态。一旦创建了代理对象，就不能撤销这个对象，这是由于 Java 运行时缓存了这个对象。尽管如此，服务对象是可以撤销的，代理对象可以不关联任何 URLStreamHandlerService 或者 java.net.ContentHandle 对象而单独存在。

在这种情况下，代理对象必须要处理接下来的请求，直到注册了另外一个相应的服务。如果发生了这样的情况，代理类必须要能够处理这样的错误。

如果是 URL 流处理代理对象，如果在方法的声明中允许抛出这样的异常，那么就必须要抛出 java.net.MalformedURLException 异常。否则，抛出异常 java.lang.IllegalStateException。

如果是内容处理器代理对象，必须返回这些数据的输入流（InputStream）。

bundle 必须要确保它们的 URLStreamHandlerService 或者是 java.net.ContentHandler 服务对象在变为未注册状态的情况下也可以抛出异常。

内容处理服务的代理和 URL 流处理服务代理的操作稍微有些不同。如果一个注册的 ContentHandlerFactory 对象返回了 null，在这种情况下，工厂对象将没有机会来为那种类型的内容提供一个 ContentHandler 对象。因此，对于一种类型的内容，如果没有内置的处理器，也没有一个注册的处理器，必须构造一个 ContentHandler 代理类，这个代理类从 URLConnection 对象中返回一个输入流对象（InputStream）作为内容对象，直到注册了一个这样类型的处理器。

11.3.2. 内置处理器

Java 实现中提供了一系列的 java.net.URLStreamHandler 类的子类来处理这样的协议，例如：HTTP，FTP，NEWS 等等。大部分 Java 实现提供了一种机制来通过类名称添加新的处理器，添加的处理器可以通过类路径查找到。

如果对于一个内置处理器（或者是一个可以通过类名称构造机制查找获得的），一个注册的 java.net.URLStreamHandlerFactory 工厂对象返回了 null，那么对于这样类型的请求就不会再次调用，这是由于 Java 实现将会使用内置的处理器，或者使用类名称构造。

因此，不能保证一定会使用注册的 URLStreamHandlerService 服务对象，内置的处理器也许会优先来处理，这样做是为了保证兼容性。在 OSGi 执行环境中定义的内置的处理器，是不能对其进行覆盖的。同样的，内容处理器工厂对象的实现也是采用了同样的技术，当然也具有同样的问题存在。

为了简化内置处理器的发现，使得可以通过名称来解释构造，在 bundle 查找服务注册中心之前，框架必须要使用下一节描述的方法。

11.3.3. 查找内置处理器

如果定义了如下系统属性：java.protocol.handler.pkgs 或者 java.content.handler.pkgs，那么必须要在内置处理器中使用这些属性。属性值为一系列的包名称，采用垂直线（‘|’, \u007C）来分隔，查找的时候按照从左到右的顺序，例如，

org.osgi.impl.handlers | com.acme.url

包名称是作为资源类型或者内容类型的前缀，来形成一个处理资源类型或者内容类型的类名。

某种资源类型的 URL 流处理器名称就是将一个字符串“.Handler”附加到资源类型名称的后面。名称前缀是包名。例如，上述例子中，对于 rsh 资源类型的处理器的查找按照如下顺序进行：

org.osgi.impl.handlers.rsh.Handler
com.acme.url.rsh.Handler

MIME 类型名称中包含了‘/’字符，和其他字符，其他字符不能是 Java 类名称中不允许出现的字符。将一个 MIME 名称转化为一个类名称的处理必须要按照以下规定进行：

1. 首先，将名称中的所有斜线转化为点号（‘.’，\u002E）。所遇其他 Java 类名称中不允许出现的字符必须要转化为下划线（‘_’,\u005F）。

application/zip	application.zip
text/uri-list	text.uri_list
image/vnd.dwg	image.vnd_dwg

2. 转换完成之后，将这个名称附加到在属性 java.content.handler.pkgs 中指定的包列表名称之后，例如，如果转化前的类型为 application/zip，而包列表为前面所示的例子，那么查找的类名称如下：

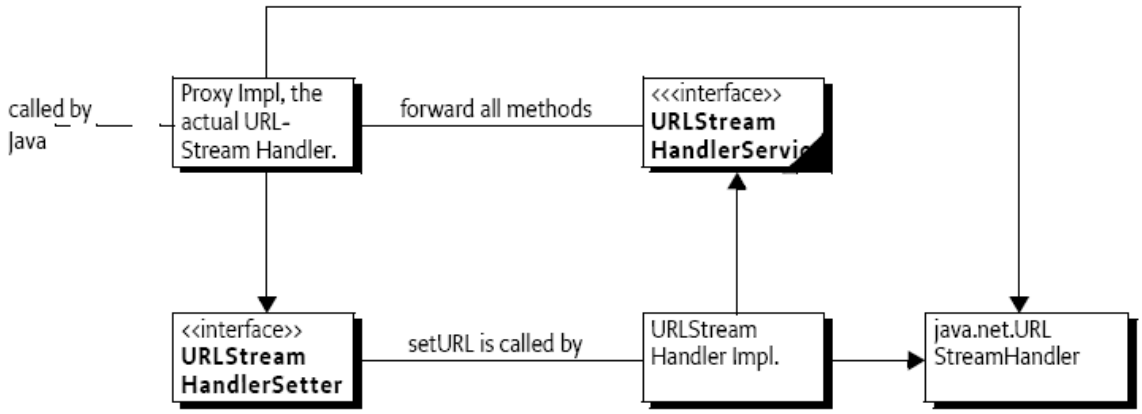
```
org.osgi.impl.handlers.application.zip
com.acme.url.application.zip
```

Java 运行时规定了包应该出现在适当的属性中，这样 URL 流处理器工厂类和内容处理器工厂的实现就可以知道这些包。

11.3.4. 保护方法和代理器

不能在服务注册中心注册 java.net.URLStreamHandler 类的实现，这是由于类 URLStreamHandler 的方法都是 protected 的，这样代理器就不能使用这些方法。同样，URLStreamHandler 类检查那些只有从 URLStreamHandlerFactory 对象中返回的 URLStreamHandler 才可以调用 setURL 方法。这也就是说，服务注册中心的 URLStreamHandler 类不能调用 setURL 方法，当实现 parseURL 方法时，是必须要调用这个方法的。

因此，创建了接口 URLStreamHandlerService 和 URLStreamHandlerSetter。接口 URLStreamHandlerService 提供了 URLStreamHandler 的公有方法，除了没有 setURL 方法，而且 parseURL 方法有一个新的类型参数，类型为 URLStreamHandlerSetter。通常，可以将 URLStreamHandler 的子类转化为 URLStreamHandlerService 服务类，只需要对代码作很小的改动。通过将相关方法设置为 public，parseURL 方法为了调用 URLStreamHandlerService 对象传递的 URLStreamHandlerSetter 对象的 setURL 方法，需要对 parseURL 方法作一些改动，相当于 URLStreamHandler 类的 setURL 方法。



为了帮助 URLStreamHandler 实现类的转换，提供了一个 AbstractURLStreamHandlerService 类。分别完成了相关方法的公有化，AbstractURLStreamHandlerService 类中的一个私有变量保存了 URLStreamHandlerSetter 对象。为了让 setURL 方法能够顺利执行，覆盖了 setURL 方法来调用保存 URLStreamHandlerSetter 的 setURL 方法，而不是 URLStreamHandler.setURL 方法。这也就是说 URLStreamHandler 类的子类应该改为 AbstractURLStreamHandlerService 类的一个子类而且需要重新编译。

通常，parseURL 方法的代码如下格式：

```
class URLStreamHandlerImpl {
    ...
    protected URLStreamHandlerSetter realHandler;
    ...
    public void parseURL(URLStreamHandlerSetter realHandler,
            URL u, String spec, int start, int limit) {
            this.realHandler = realHandler;
            parseURL(u, spec, start, limit);
    }

    protected void setURL(URL u,
            String protocol, String host,
            int port, String authority,
            String userInfo, String path,
            String query,String ref) {
            realHandler.setURL(u, protocol, host,
            port, authority, userInfo, path,
            query, ref);
    }
    ...
}
```

URLStreamHandler.parseURL 将调用 setURL 方法，这个方法必须要通过代理器调用。这也就是为什么覆盖 setURL 方法委托给 URLStreamHandlerSetter。

11.4. 提供新的模式

下面的例子提供了一种资源类型，这种资源类型返回了 URL 的路径部分。第一部分是 URLStreamHandlerService 类的一个实现。服务启动之后，在 OSGi 框架中这个类将自己注册。

然后如果必须创建一个新的 java.net.URLConnection 对象，那么 OSGi 框架调用方法 openConnection 方法。在这个例子中，返回的是 DataConnection 对象。

```
public class DataProtocol
    extends AbstractURLStreamHandlerService
    implements BundleActivator {

    public void start( BundleContext context ) {
        Hashtable properties = new Hashtable();
        properties.put( URLConstants.URL_HANDLER_PROTOCOL,
        new String[] { "data" } );
        context.registerService(
        URLStreamHandlerService.class.getName(),
        this, properties );
    }

    public void stop( BundleContext context ) {}

    public URLConnection openConnection( URL url ) {
        return new DataConnection(url);
    }
}
```

下面的例子中，DataConnection 类继承了 java.net.URLConnection 类，而且覆盖了构造方法，这样就可以为超类、connect 方法和 getInputStream 方法提供 URL 对象。最后一个方法返回了 URL 的路径部分，返回的形式为一个 java.io.InputStream 对象。

```
class DataConnection extends java.net.URLConnection {
    DataConnection( URL url ) {super(url);}

    public void connect() {}

    public InputStream getInputStream() throws IOException {
        String s = getURL().getPath();
        byte [] buf = s.getBytes();
        return new ByteArrayInputStream(buf,1,buf.length-1);
    }

    public String getContentType() {
        return "text/plain";
    }
}
```

11.5. 内容处理器

内容处理器（Content Handler）应该继承 java.net.ContentHandler 类，而且实现 getContent 方法。这个方法必须要从 java.net.URLConnection 参数中获得一个 InputStream 对象，而且将输入流中的子介转化为应用类型。在这个例子中，MIME 类型是 text/plain，返回的对象是一个字符串对象。

```
public class TextPlainHandler extends ContentHandler
    implements BundleActivator {

    public void start( BundleContext context ) {
```

```
            Hashtableproperties = new Hashtable();
            properties.put( URLConstants.URL_CONTENT_MIMETYPE,
            new String[] { "text/plain" } );
            context.registerService(
            ContentHandler.class.getName(),
            this, properties );
        }
        public void stop( BundleContext context ) {}

        public Object getContent( URLConnection conn )
            throws IOException {
            InputStream in = conn.getInputStream();
            InputStreamReader r = new InputStreamReader( in );
            StringBuffer sb = new StringBuffer();
            int c;
            while ( (c=r.read()) >= 0 )
            sb.append( (char) c );
            r.close(); in.close();
            return sb.toString();
        }
    }
```

11.6. 安全问题

指定一种协议类型并且添加内容处理器会导致程序可能会直接影响到 Java 虚拟机中的核心代码类的运行。在网络应用中，广泛使用了 java.net.URL 类，而且 OSGi 框架本身也可以使用这个类。

因此，当提供了注册处理器的能力之后，必须要小心谨慎。支持的两种类型的处理器是 URLStreamHandlerService 和 java.net.ContentHandler。只有可信的 bundle 才可以进行注册这些服务，并且具有对这些类的权限：ServicePermission[URLStreamHandlerService|ContentHandler, REGISTER]。由于其他 bundle 都可以通过 java.net.URL 类和 java.net.URLConnection 类来使用这些服务，最好还是对于所有 bundle 拒绝这样的服务（ServicePermission[<name>,GET]），而只有框架才可以获得它们。这样可以阻止一些恶意代码攻击，恶意代码可以通过 java.net.URL 类完成权限检查，然后就可以直接使用 URLStreamHandlerServices 类。

11.7. org.osgi.service.url

OSGi URL 流和内容处理器 API 包，版本规范：1.0。

需要使用这个包的 bundle 必须要导入这个包，通过在 bundle 清单文件中的 Import-Package 头标中描述。例如：

```
Import-Package: org.osgi.service.url; version=1.0
```

11.7.1. 概要

- AbstractURLStreamHandlerService – URLStreamHandlerService 接口的抽象实现。[p.263]
- URLConstants – 定义了与 URLStreamHandlerService[p.265]和 java.net.ContentHandler

服务相关的属性键名称。[p.264]
URLStreamHandlerService – 服务接口，实现了 java.net.URLStreamHandler 中私有方法的公开化。[p.265]
URLStreamHandlerSetter – URLStreamHandlerService 对象使用的接口，用来调用代理器 URLStreamHandler 对象的 setURL 方法。

11.7.2. public abstract class AbstractURLStreamHandlerService extends URLStreamHandler implements URLStreamHandlerService

URLStreamHandlerService 接口的抽象实现。所有的方法只是简单的调用了在类 java.net.URLStreamHandler 中相应的方法，除了 parseURL 和 setURL 方法，这两个方法使用了 URLStreamHandlerSetter 参数。这个抽象类的子类应该没有必要覆盖 setURL 和 parseURL(URLStreamHandlerSetter,...)方法。

11.7.2.1. protected URLStreamHandlerSetter realHandler

传递给 parseURL 方法的 URLStreamHandlerSetter 对象。

11.7.2.2. public AbstractURLStreamHandlerService( )

11.7.2.3. public boolean equals(URL u1, URL u2)

- 调用 super.equals(URL,URL)

See Also java.net.URLStreamHandler.equals(URL,URL)

11.7.2.4. public int getDefaultPort( )

- 调用 super.getDefaultPort。

See Also java.net.URLStreamHandler.getDefaultPort

11.7.2.5. public InetAddress getHostAddress(URL u)

- 调用 super.getHostAddress

See Also java.net.URLStreamHandler.getHostAddress

11.7.2.6.public int hashCode(URL u)

- 调用 super.hashCode(URL)

See Also java.net.URLStreamHandler.hashCode(URL)

11.7.2.7.public boolean hostsEqual(URL u1, URL u2)

- 调用 super.hostsEqual

See Also java.net.URLStreamHandler.hostsEqual

11.7.2.8.public abstract URLConnection openConnection(URL u) throwsIOException

See Also java.net.URLStreamHandler.openConnection

11.7.2.9.public void parseURL(URLStreamHandlerSetter realHandler, URL u, String spec, int start, int limit)

realHandler 必须要对指定 URL 调用这个对象的 setURL 方法

- 使用一个 URLStreamHandlerSetter 对象来分析一个 URL，通过这个方法来设置 realHandler 的值，设置为指定的 URLStreamHandlerSetter 对象，然后调用 parseURL(URL,String,int,int)。

See Also java.net.URLStreamHandler.parseURL

11.7.2.10. public boolean sameFile(URL u1, URL u2)

- 调用 super.sameFile

See Also java.net.URLStreamHandler.sameFile

11.7.2.11. protected void setURL(URL u, String proto, String host, int port, String file, String ref)

- 调用 realHandler.setURL(URL,String,String,int,String,String)

See Also java.net.URLStreamHandler.setURL(URL,String,String,int,String,String)

Deprecated 这个方法只适用于使用 JDK1.1 写的处理器

11.7.2.12. protected void setURL(URL u, String proto, String host, int port, String auth, String user, String path, String query, String ref)

- 调用 realHandler.setURL(URL,String,String,int,String,String,String,String)

See Also java.net.URLStreamHandler.setURL(URL,String,String,int,String, String,String,String)

11.7.2.13. public String toExternalForm(URL u)

- 调用 super.toExternalForm

See Also java.net.URLStreamHandler.toExternalForm

11.7.3. public interface URLConstants

定义了与 URLStreamHandlerService[p.265]和 java.net.ContentHandler 服务相关的属性键名称。
如果没有特别声明，属性的值为 java.lang.String[]。

11.7.3.1.public static final String URL_CONTENT_MIMETYPE = "url.content.mimetype"

服务属性，表示 MIME 类型，提供给 java.net.ContentHandler。这个属性的值为一个 MIME 类型的数组。

11.7.3.2.public static final String URL_HANDLER_PROTOCOL = "url.handler.protocol"

服务属性，表示协议名称，提供给 URLStreamHandlerService。
这个属性的值为一个协议名称的数组。

11.7.4. public interface URLStreamHandlerService

服务接口，提供了对 java.net.URLStreamHandler 的私有方法的公有化。
这个接口和 URLStreamHandler 类的重要不同点在于没有 setURL 方法，而且 parseURL 方法中有一个 URLStreamHandlerSetter 对象作为第一个参数。实现这个接口的类必须要调用通过 parseURL 方法获得的 URLStreamHandlerSetter 对象的 setURL 方法，而不是调用 URLStreamHandler.setURL，这样可以避免抛出安全异常（SecurityException）。

See Also AbstractURLStreamHandlerService[p.263]

11.7.4.1.public boolean equals(URL u1, URL u2)

See Also java.net.URLStreamHandler.equals(URL, URL)

11.7.4.2.public int getDefaultPort( )

See Also java.net.URLStreamHandler.getDefaultPort

11.7.4.3.public InetAddress getHostAddress(URL u)

See Also java.net.URLStreamHandler.getHostAddress

11.7.4.4.public int hashCode(URL u)

See Also java.net.URLStreamHandler.hashCode(URL)

11.7.4.5.public boolean hostsEqual(URL u1, URL u2)

See Also java.net.URLStreamHandler.hostsEqual

11.7.4.6.public URLConnection openConnection(URL u) throws IOException

See Also java.net.URLStreamHandler.openConnection

11.7.4.7.public void parseURL(URLStreamHandlerSetter realHandler, URL u, String spec, int start, int limit)

realHandler 必须要对指定 URL 调用这个对象的 setURL 方法

- 分析一个 URL。这个方法通过 URLStreamHandler 代理来调用，而不是 java.net.URLStreamHandler.parseURL，通过传递一个 URLStreamHandlerSetter 对象来完成。

See Also java.net.URLStreamHandler.parseURL

11.7.4.8.public boolean sameFile(URL u1, URL u2)

See Also java.net.URLStreamHandler.sameFile

11.7.4.9.public String toExternalForm(URL u)

See Also java.net.URLStreamHandler.toExternalForm

11.7.5. public interface URLStreamHandlerSetter

URLStreamHandlerService 类使用这个接口来调用 URLStreamHandler 代理器的 setURL 方法。

这个类型的对象传递给 URLStreamHandlerService.parseURL[p.265]方法。调用为这个协议而实际注册在 java.net.URL 之下的 URLStreamHandlerSetter 对象的 setURL 方法。

11.7.5.1. public void setURL(URL u, String protocol, String host, int port, String file, String ref)

See Also java.net.URLStreamHandler.setURL(URL,String,String,int,String,String)
Deprecated 这个方法只适用于使用 JDK1.1 编写的处理器。

11.7.5.2. public void setURL(URL u, String protocol, String host, int port, String authority, String userInfo, String path, String query, String ref)

See Also java.net.URLStreamHandler.setURL(URL,String,String,int,String, String,String,String)

11.8. 参考

[47] Java
http://java.sun.com/j2se/1.4/docs/api/java/net/package-summary.html
[48] URLs
http://www.ietf.org/rfc/rfc1738.txt
[49] MIME Multipurpose Internet Mail Extension
http://www.nacs.uci.edu/indiv/ehood/MIME/MIME.html
[50] Assigned MIME Media Types
http://www.iana.org/assignments/media-types

完