基于最新OSGi R5.0规范撰写，继《深入理解Java虚拟机》后的又一实力之作

全面解读OSGi规范，深刻揭示OSGi原理，详细讲解OSGi服务，系统介绍Equinox框架的用法，并通过源代码分析其工作机制，包含大量可操作性极强的解决方案和最佳实践

华章原创精品

深入理解OSGi

Equinox原理、应用与最佳实践

Understanding the OSGi Principles, Using and Best Practices

周志明 谢小明 著

机械工业出版社
China Machine Press

深入理解 OSGi：Equinox 原理、应用与最佳实践

周志明　谢小明　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 OSGi：Equinox 原理、应用与最佳实践 / 周志明，谢小明著 . —北京：机械工业出版社，2013.1

ISBN 978-7-111-40887-1

I. 深… II. ① 周… ② 谢… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2012）第 302863 号

本书是原创 Java 技术图书领域继《深入理解 Java 虚拟机》后的又一实力之作，也是全球首本基于最新 OSGi R5.0 规范的著作。理论方面，既全面解读了 OSGi 规范，深刻揭示了 OSGi 原理，详细讲解了 OSGi 服务，又系统地介绍了 Equinox 框架的使用方法，并通过源码分析了该框架的工作机制；实践方面，不仅包含一些典型的案例，还总结了大量的最佳实践，极具实践指导意义。

全书共 14 章，分 4 个部分。第一部分（第 1 章）：走近 OSGi，主要介绍了什么是 OSGi 以及为什么要使用 OSGi。第二部分（第 2 ～ 4 章）：OSGi 规范与原理，对最新的 OSGi R5.0 中的核心规范进行了全面的解读，首先讲解了 OSGi 模块的建立、描述、依赖关系的处理，然后讲解了 Bundle 的启动原理和调度管理，最后讲解了与本地及远程服务相关的内容。第三部分：OSGi 服务与 Equinox 应用实践（第 5 ～ 11 章），不仅详细讲解了 OSGi 服务纲要规范和企业级规范中最常用的几个子规范和服务的技术细节，还通过一个基于 Equinox 的 BBS 案例演示了 Equinox 的使用方法，最重要的是还通过源码分析了 Equinox 关键功能的实现机制和原理。第四部分：最佳实践（第 12 ～ 14 章），总结了大量关于 OSGi 的最佳实践，包括从 Bundle 如何命名、模块划分、依赖关系处理到保持 OSGi 动态性、管理程序启动顺序、使用 API 基线管理模块版本等各方面的实践技巧，此外还介绍了 Spring DM 的原理以及如何在 OSGi 环节中进行程序测试。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：孙海亮

印刷

2013 年 2 月第 1 版第 1 次印刷

186mm × 240mm • 27 印张

标准书号：ISBN 978-7-111-40887-1

定 价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

前　言

为什么写这本书

随着软件规模的日益增大，程序按模块拆分、按模块开发和按模块部署等需求变得越来越迫切，“模块化”成为了 Java 社区中最热门的话题之一。而 OSGi 技术——Java 业界事实上的模块化标准，也被越来越多的中间件、第三方类库和各类应用程序所认可和采用。与此相对的是，有关 OSGi 技术的资料，尤其是中文的资料却显得异常的贫乏。

笔者自 2007 年接触 OSGi 以来，曾在数个大型系统中使用过 OSGi 作为软件的基础架构，这一方面使笔者深刻感受到了 OSGi 带来的诸多好处以及 OSGi 设计思想的魅力；另一方面也使笔者注意到 OSGi 的入门门槛相对较高，如果没有足够的指导材料，开发人员从零开始学习并探索出 OSGi 的最佳实践需要很高的成本。因此，笔者就萌生了写一本关于 OSGi 原理、应用与最佳实践的中文书籍的想法。

自从 1999 年 OSGi 联盟成立以来，OSGi 技术随着 Java 一起飞速发展，现已经成为一种被广泛认可的软件架构技术和方法。OSGi 联盟的成员数量也已经从最开始的几个增长到目前的 100 多个，许多世界著名的 IT 企业都加入到 OSGi 的阵营之中，如 Adobe、IBM、Oracle、SAP、RedHat 和 Siemens 等。这些软件厂商推出的许多产品都支持 OSGi 技术，甚至产品本身就使用了 OSGi 技术构建，例如 IBM 的 WebSphere、Lotus 和 JAZZ，Oracle 的 GlassFish 和 Weblogic，RedHat 的 JBoss，Eclipse 基金会的 Eclipse IDE、Equinox 及之下的众多子项目，Apache 基金会的 Karaf、Aries、Geronimo、Felix 及之下的众多子项目等。这些业界巨头的踊跃参与，从侧面证明了 OSGi 技术有着非常广阔的市场前景。

OSGi 能获得如此广泛的认可，一方面是它的诞生迎合了软件模块化的需求；另一方面是因为它足够全面和开放，OSGi 所具有的动态性、伸缩性正好是许多生产系统所需的。基于 OSGi 的程序更新升级或者缺陷修复，能够像电脑更换 USB 接口的鼠标键盘或者插拔其他 USB 设备那样可以即插即用，无须重启，甚至无须停顿，这是许多基于 Java 的、需要 7×24 小时运转的生产系统长期以来迫切希望而又无法实现的需求。把软件开发中公用的、通用的功能抽象成标准件，令各个软件可以使用同样的模块来完成特定需求，把软件开发变为搭建模块，这更是软件工业化的最终理想，而 OSGi 为这个目标带来了实现的曙光。

目前，虽然 OSGi 背后有庞大的厂商支持，对实现框架、中间件和类库的支撑也足够完善，但是在国内基于 OSGi 架构的系统还不是很多。很多软件企业都跃跃欲试，尝试迈出使

用 OSGi 的第一步，但往往被其复杂性阻挡于大门之外——如果要用 OSGi 开发一个入门程序，并不需要多高深的知识，但如果要把 OSGi 运用于生产系统，则要求该软件系统的架构师和至少一部分中高级开发人员必须对 OSGi 有比较深入的了解，业界对这一部分人才的需求也一直非常大。通过本书，读者可以通过一个相对轻松的方式学习到 OSGi 的运作原理，这对自身能力的提高有较大的帮助。

本书面向的读者

1. 系统架构师

OSGi 目前已经成为互联网、桌面程序、移动设备和企业级应用等领域中最流行的系统架构方法之一，OSGi 本身的设计思想也非常值得系统架构师借鉴。

2. 使用 Java 技术体系的中、高级开发人员

鉴于 OSGi 技术本身的复杂性和相对陡峭的学习曲线，开发人员入门和深入理解 OSGi 时要付出较多的努力，本书的理论讲解和案例实践将帮助对 OSGi 感兴趣的开发人员跨过初学 OSGi 的那道门槛。

3. 系统运维工程师

保障系统的性能，更新和维护程序版本是系统运维工程师的主要职责。目前 Java 业界主流的中间件均开始直接支持甚至基于 OSGi 架构实现。伴随 OSGi 的发展，越来越多使用 OSGi 技术的程序被部署到生产系统之中，OSGi 已经成为一个优秀的运维工程师必须了解的专业知识。本书中的大量案例、代码、调优实战将会对系统运维工程师日常的工作起到直接的参考作用。

如何阅读本书

本书一共分为 4 个部分：走近 OSGi、OSGi 规范与原理、基于 Equinox 的 OSGi 应用与实践、最佳实践。各个部分之间基本上是互相独立的，没有必然的前后依赖关系，读者可以从任何一个感兴趣的专题开始阅读，但是每个部分中的各个章节间会有先后顺序。

本书并不要求读者在 Java 领域具备很高的技术水平，而且在讲解各个知识点时会在保证逻辑准确的前提下、尽量用通俗的语言和案例去讲述 OSGi 中与开发关系最为密切的内容；但是由于探讨 OSGi 过程中涉及的许多问题不可避免地需要读者有一定技术基础，因此本书的定位依然是面向中、高级的程序员群体，对于一些常用的开发框架、Java API、Java 语法等基础知识点，将认为读者已有所了解。

下面简要介绍一下本书各部分的内容。

第一部分　走近 OSGi

本书第一部分为后文的研究和讲解打下了基础，让读者了解 OSGi 的来龙去脉以及它出现的意义，这是走近 Java 技术和 Java 虚拟机的第一步。第一部分包括第 1 章：

第 1 章　在这本书中，笔者尝试阐述与 OSGi 相关的三个问题：什么是 OSGi？为什么要使用 OSGi？如何使用 OSGi？在第 1 章中，笔者将针对前面两个问题进行分析介绍并给出答案。

第二部分　OSGi 规范与原理

最新的 OSGi R5 规范文档包含了数十个子规范、数百页的文档和近 2000 个 API。一般的开发人员很难、也没有必要完全了解 OSGi 规范的所有细节。但是，要学习 OSGi 技术，阅读 OSGi 核心规范（OSGi Core）是必需的过程。在第二部分中，笔者将介绍 OSGi 核心规范中的内容。第二部分包括第 2 至 4 章：

第 2 章　介绍了 OSGi 中模块这个最基础的概念，并讲解了 OSGi 如何建立模块、如何描述模块、模块间的依赖关系如何处理等内容。

第 3 章　介绍了 Bundle 是如何启动的，Bundle 自安装到卸载所经历的生命周期状态和这些状态的转换条件过程，还介绍了 OSGi 框架是如何使用启动级别对系统中的 Bundle 进行管理调度的。

第 4 章　介绍了本地及远程服务定义、注册、查找和使用方法，这个过程涉及服务事件监听，以及使用服务钩子干涉服务与 OSGi 框架的交互过程等知识。

第三部分　基于 Equinox 的 OSGi 应用与实践

如果说 OSGi 核心规范是 OSGi 技术的灵魂，OSGi 的服务纲要规范（OSGi Service Compendium）和 OSGi 企业级规范（OSGi Service Enterpress）就是其躯体。在第三部分中，笔者会详细介绍其中最常用的几个子规范的内容。在这一部分中，将会把 OSGi 技术从理论落地到实践之中，结合目前使用范围最广的 OSGi 实现 Equinox 和一个穿插整个部分的完整案例来讲解 OSGi 技术的使用。第三部分包括第 5 至 11 章：

第 5 章　介绍了 Equinox 和 OSGi 的关系、Equinox 的组成，以及如何获取、安装、使用和调试 Equinox 的代码。本章奠定了整个第三部分的技术基础。

第 6 章　尝试实现了一个名为“Neonat”的基于 Equinox 框架的 Telnet BBS，这个项目将迭代贯穿于第三部分。另外，还从浩瀚庞大的 Equinox 源码中挑选出 4 个关键功能点进行源码解析。读者在理解了 Equinox 这几个关键功能点是如何实现的之后，与前面第一部分介绍的 OSGi 规范的理论相互对照和印证，可以更好地理解 OSGi 的技术原理，知道为何要这样设计和实现。

第 7 章　介绍了服务端 OSGi 的应用，分析了 OSGi 的 HTTP Service 规范的使用和原理，及如何为 Neonat 添加 Web 访问模块。

第 8 章　介绍了 OSGi 的 User Admin 服务的基本使用，以及如何为 Neonat 添加用户管理模块，如何实现用户权限控制。

第 9 章　介绍了 OSGi 的 Preferences 服务，重新实现了 Neonat 的数据访问模块，展示了如何使用 Preferences 服务来持久化非事务性数据。

第 10 章　介绍了声明式服务的使用。声明式服务规范的制定，推动了 OSGi 服务从程

序化的服务模型向声明式的服务模型发展，这对整个 OSGi 来说也是一项非常有意义的进化，令 OSGi 的服务真正成为一项可统一分配、使用和管理的资源。

第 11 章　介绍了 OSGi R5 中最新出现的 Subsystem 规范，这是 OSGi R5 相对 R4.3 最重要的改进。在制定了 Subsystem 规范（及其相关的 Respository 和 Resolver 等规范）后，表明 OSGi 对企业级开发的支持已经达到相当完善的程度了。Subsystem 让各个软件厂商的 OSGi 企业级容器有了通用的程序发布格式，让 OSGi 支持者能向同一个方向努力，推动企业级 OSGi 的发展，这点将是有深远影响的。

第四部分　最佳实践

学习过 OSGi 的原理、规范和各种服务的使用后，第四部分笔者将针对开发实践中遇到的具体问题进行分析、讲解，介绍 OSGi 的各种最佳实践。第四部分包括 12 至 14 章：

第 12 章　介绍了 OSGi 的最佳实践，包括 Bundle 命名方法、模块划分、依赖关系处理、保持 OSGi 动态性、管理程序启动顺序、使用 API 基线管理模块版本等各方面的实践技巧。

第 13 章　介绍了 Spring DM 的原理、使用，以及如何把基于 Spring 的程序引入到 OSGi 环境之中。

第 14 章　讲解了单元测试和集成测试在 OSGi 程序中的意义，展示了在 OSGi 环境中如何进行程序测试，以及相关测试工具的使用。

勘误和支持

在本书交稿的时候，笔者并没有想象中那样兴奋或轻松，写作之时那种“战战兢兢、如履薄冰”的感觉依然萦绕在心头。在写作过程中，在每一章、每一节落笔之时，笔者都在考虑如何写才能把这个知识点有条理地讲述出来，都在担心会不会由于自己理解有偏差而误导了大家。囿于作者水平和写作时间，书中存在不妥之处在所难免，所以特别开辟了一个读者邮箱（understandingosgi@gmail.com），读者有任何意见、建议都欢迎与笔者联系。另外，读者可以从华章公司的网站本书的相应页面下载书中所涉及的源码，华章公司地址为：http://www.hzbook.com。写书与写程序一样，作品一定都有不完美之处，因为不完美，我们才有不断追求完美的动力。

目　　录

第四部分 最佳实践

第一部分

走近 OSGi

第 1 章　Java 模块化之路

Java 可能是近 20 年来最成功的开发技术，因其具备通用性、高效性、平台移植性和安全性而成为不同硬件平台理想的开发工具。从笔记本电脑到数据中心，从游戏控制台到科学超级计算机，从手机到互联网，Java 技术无处不在。

Java 能够让程序员使用同一种语言为服务器、智能卡、移动电话和嵌入式设备开发程序，极大地提升了软件的研发效率。不过，仅靠统一的语言还不足以让软件业迅速提升至成熟的工业化阶段。不同软件系统、不同硬件设备下的程序都经常会有相同的业务需求和设备间交互通信的需求，例如很多设备都需要互联网接入的功能，如果通用于不同设备的网络标准件不存在，那就只能为每个设备都开发一个连接互联网的模块，这样效率和质量都难以保证。假如把开发中经常遇到的需求进行抽象，将它们统一规范起来作为标准件提供，任何设备都通过预定义好的协议和接口来使用这些标准件，那么构造一个大型程序的主要工作很可能就只是根据需求选择合适的模块，然后再写少量的黏合代码而已。

标准件是区别小手工作坊和大工业化最明显的标志。今天，个人计算机的硬件已经到达了工业化阶段，无论哪个公司生产的显示器、键盘、鼠标、内存和 CPU，都遵循统一规范的接口工作。要获得不同功能、性能的计算机，只要选择适当的硬件模块进行组装即可。与此相对的，大部分计算机软件都还是从零开始进行编码开发的。软件业还远不如硬件成熟，但是软件工业化是一股不可逆转的潮流，实现这个目标的第一步就是要制定不同功能模块的标准，以及模块间的黏合及交互方式。Java 业界内已经有了很多的技术规范，例如 EJB、JTA、JDBC、JMS 等，欠缺的是一个组织者或扮演黏合剂的角色，直到 Java 有了 OSGi……

1.1　什么是 OSGi

OSGi（Open Service Gateway Initiative，直译为“开放服务网关”）实际上是一个由 OSGi 联盟（OSGi Alliance，如图 1-1 所示）发起的以 Java 为技术平台的动态模块化规范。

OSGi 联盟是由 Sun Microsystems、IBM、Ericsson 等公司于 1999 年 3 月成立的一个世界性的开放标准化组织，最初的名称为 Connected Alliance，该组织成立的主要目的原本在于使服务提供商通过住宅网关为各种家庭智能设备提供服务。最初的 OSGi 规范也只是关注于嵌入式领域，前三个版本的 OSGi 规范主要满足诸如机顶盒、服务网关、手机等应用环境的模块化需求。从第四个版本开始，OSGi 将主要关注点转向了 Java SE 和 EE 领域，并且在这些领域中获得了很大的发展，成为 Java 平台事实上的模块化规范。

图 1-1　OSGi 联盟

随着 OSGi 技术的不断发展，OSGi 联盟的成员数量已经由最开始的几个增长到目前超过 100 个[⊖]，很多世界著名的 IT 企业都加入到 OSGi 的阵营之中，如 Adobe、IBM、Oracle、SAP、RedHat 和 Siemens 等。它们推出的许多产品都支持 OSGi 技术，甚至产品本身就使用了 OSGi 技术构建，例如 IBM 的 WebSphere、Lotus 和 JAZZ，Oracle 的 GlassFish 和 Weblogic，RedHat 的 JBoss，Eclipse 基金会的 Eclipse IDE、Equinox 及之下的众多子项目，Apache 基金会的 Karaf、Aries、Geronimo、Felix 及之下的众多子项目等。这些 IT 巨头的踊跃参与，也从侧面证明了 OSGi 技术有着非常广阔的市场前景。

OSGi 技术的影响同时也延伸到了 Java 社区，JSR–232 提案的通过说明 OSGi 技术已经被 Java ME 领域所认可，而 JSR–291 提案则奠定了 OSGi 技术在 Java SE 和 Java EE 领域标准模块化规范的地位（OSGi 与 Java 模块化规范的历史将在 1.1.2 节会详细介绍）。

OSGi 的诸多优秀特性，如动态性、模块化和可扩展能力逐渐被越来越多的开发者所认识和欣赏，越来越多的系统基于 OSGi 架构进行开发。在这些系统的开发过程中，又会向 OSGi 提出一个又一个新的需求，所以 OSGi 规范所包括的子规范与技术范畴也在不断发展、日益壮大，如图 1-2 所示。

今天，OSGi 的已经不再是原来 Open Service Gateway Initiative 的字面意义能涵盖的了，OSGi 联盟给出的最新 OSGi 定义是 The Dynamic Module System for Java，即面向 Java 的动态模块化系统。

2012 年 7 月，OSGi 联盟发布了最新版的 OSGi R5.0 规范，这次发布的规范包括 OSGi 核心规范 R5.0 和 OSGi 企业级规范 R5.0。在 Java SE 领域，Eclipse 和 NetBean 两款集成开发工具的成功已经完全证明了 OSGi 在桌面领域是能担当重任的。最近两三年来，OSGi 的发展方向主要集中在 Java EE 领域，在 OSGi 企业专家组（EEG）的努力下，OSGi 的企业级规范 R5.0 版相比两年前发布的 R4.2 版又增加了许多新的内容，OSGi 技术在服务端和企业级领域正迅速走向成熟。

⊖　OSGi 联盟成员：http://www.osgi.org/About/Members。

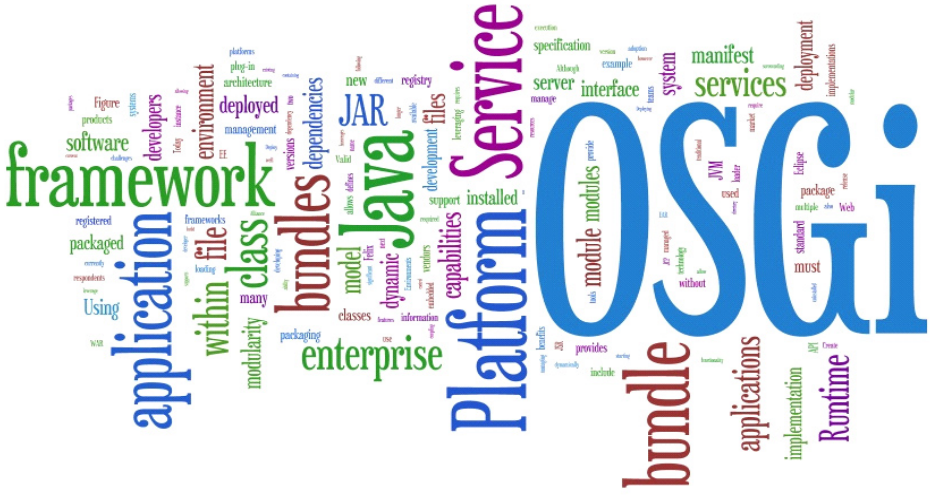

图 1-2 OSGi 及相关技术

1.1.1 OSGi 规范的演进

OSGi 在 R4 版之前都处于初级阶段，规范的主要关注点是在移动和嵌入式设备上的 Java 模块化应用。在这个初级阶段中有一些很成功的案例，比如 BMW（宝马）汽车使用 OSGi 架构实现的多媒体设备控制程序（内部是西门子 VDO 系统），要使用不同型号的电子设备，只更换对应程序模块便取得了很好的效果。但是初级阶段的 OSGi 在 Java 其他主流应用领域（企业级、互联网、服务端、桌面端等）的影响力还比较有限，因此下面简要介绍这部分的历史。OSGi 规范在初级阶段一共发布了三个版本：

- ❑ OSGi Release 1 (R1)：2000 年 5 月发布。
- ❑ OSGi Release 2 (R2)：2001 年 10 月发布。
- ❑ OSGi Release 3 (R3)：2003 年 3 月发布。

从 OSGi R4 版开始，OSGi 的目标就从“在移动和嵌入式设备上的 Java 模块化应用”发展为“Java 模块化应用”，去掉了“在移动和嵌入式设备上的”这个限定语，这意味着 OSGi 开始脱离 Java ME 的约束，向 Java 其他领域进军。同时也意味着 OSGi 需要考虑如何去迁移遗留的异构系统、如何去支持大规模开发等非嵌入式领域的问题了。因此，OSGi R4 版规范的复杂度相应地高出 R3 版许多。笔者可以给出两个最直观的数据：规范文档的页数从 R3 的 450 页增加到 900 页，规范中定义的外部接口（统计 API 中 public 方法）数量从 R3 的 661 个增加到 1432 个[⊖]。

⊖ 此数据来源于 World Congress 2005 的 PPT：http://www.osgi.org/wiki/uploads/Congress2005/1012_1015_lopez。pdf。

OSGi R4 版本分为两个部分，2005 年 10 月发布的核心规范（包含服务纲要规范）和 2006 年 9 月发布的移动设备规范（移动设备规范已停止发展，目前最新的 OSGi 移动设备规范依然是这个版本）。从更具体的角度来讲，OSGi R4 解决了 R3 的许多遗留问题和限制，以下列举了部分在核心规范中比较关键的改进：

- 在 OSGi R3 版本中，模块导出的 Package 是全局唯一的，不允许同一个 Package 存在多个版本。这点限制放在资源受限的嵌入式环境中一般不会有问题，但是放在整个 Java 领域就不妥了，因为引用不同版本的第三方包对于规模稍大一点的程序来说是很常见的事情。
- OSGi R3 中的模块缺乏对模块本身的扩展机制，所有的资源、代码都必须在模块中是静态存在的，无法运行时动态添加。在 OSGi R4 中，出现了 Fragment Bundle 的概念。
- OSGi R3 的 Package 导入和导出无论是版本、可见性和可选择性都很粗糙，例如在导入时指明一个 Package 的版本，语义就只能是导入不小于这个版本的 Package，而对于要明确具体版本范围（如 [2.5，3.0)）的需求就不适用；又如在导出 Package 时，一个 Package 中的所有类要么全部导出，要么全部隐藏。在 OSGi R4 中改进了 version 参数，也为导入导出加入了许多子参数来方便精确过滤范围。

除了在核心规范中对 R3 版的改进外，许多目前非常常用的、在服务纲要规范中的 OSGi 服务也是在 R4 版才开始出现的。这些服务对提升开发人员的工作效率及系统的鲁棒性[⊖]有很大帮助，例如 R4 版首次出现的声明式服务就是对 R3 版之前的程序化服务模型的重大改进。

尽管从 OSGi R3 到 OSGi R4 发生了很大的变化，R4 版规范依然保持了很好的向后兼容性，绝大部分能运行于 OSGi R3 的模块都可以不经修改地迁移到 OSGi R4 之中。

2007 年 5 月发布的 OSGi R4.1 是一个修正版性质的规范，只是核心规范发生了很小的变化，服务纲要规范和移动设备规范并没有跟随发布 R4.1 版，整个 R4.1 版没有新增任何服务。OSGi R4.1 版本的推出，最重要的任务是适配 JSR-291 提案，让 JSR-291 提案顺利通过 JCP 的投票，成为整个 Java 业界标准的一部分。

在 OSGi R4.1 版本中，值得一提的改进是处理了 Bundle 延迟初始化的问题，增加了 Bundle-ActivationPolicy 标识来指明 Bundle 的启动策略。在此之前，OSGi 实现框架只能通过自己的非规范的标识来完成类似的事情，例如 Equinox 的私有的 Eclipse-LazyStart 标识。

2009 年 9 月，OSGi R4.2 版核心规范发布；在次年 3 月，还发布了 OSGi R4.2 企业级规范。OSGi R4.2 是一个包含了许多重要改进的版本。首先，随着 OSGi 实现框架的数量逐渐增多，OSGi R4.2 开始着手解决 OSGi 框架自身的启动问题，提供了操作 OSGi 框架的统一 API。在此之前，启动 Felix、Equinox、Knopflerfish 或其他 OSGi 框架，必须使用完全不同

⊖ 鲁棒性是 Robustness 的音译，是指当系统受到不正常干扰时，是否还能保证主体功能正常运作。可参考维基百科：http://zh.wikipedia.org/zh/ 鲁棒性 _(计算机科学)。

的私有 API 来实现，这点不利于程序在不同 OSGi 上实现平滑迁移。另外，OSGi R4.2 还向开发者提供了影响 OSGi 框架运作的能力，如 Bundle Tracker、Service Hooks 这些工具的出现，让由非 OSGi 实现框架的开发人员去实现 OSGi 系统级的模块成为可能。

在具体服务方面，OSGi R4.2 的主旋律是企业级服务的改进。许多企业级服务在这个版本中首次出现，例如远程服务规范和 Blueprint 容器（提供依赖注入和反转控制的容器，类似于 Spring 的功能）规范。说到企业级服务，OSGi R4.2 专门独立发布企业级服务规范的一个重要任务就是解决 OSGi 与 Java EE 服务之间关系的问题。Java EE 体系中的许多重要服务如 JNDI、JPA 和 JDBC 等在企业级开发中都是不可或缺的，因此在 OSGi R4.2 中相应定义了 JNDI、JPA 和 JDBC 等服务规范，将 Java EE 的服务引入到 OSGi 容器中来。

除此之外，OSGi R4.2 还制定了 Web Applications 规范，使 OSGi 中包含 Web 页面的模块可以使用标准 WAR 格式来打包（打包后的产品为以 .wab 为扩展名的 JAR 格式文件），允许将这些模块直接安装到支持 OSGi 和 Web Applications 规范的应用服务器之中。

2011 年 4 月和 2012 年 3 月，分别发布了 OSGi R4.3 的核心规范和服务纲要规范。在这个版本中，OSGi 的 API 接口终于开始使用已经有 8 年历史的、从 Java SE 5 开始提供的泛型[⊖]。OSGi 对 Java 平台版本迟缓响应，很大程度是因为顾及到嵌入式环境的虚拟机版本，很多设备还没有升级到 Java 1.5，同时还顾及其中存在的遗留系统。OSGi R4.3 有了这样的变化，也从侧面说明 OSGi 的重心已经开始向服务端应用等领域偏移了。

OSGi R4.3 的另一个重要改进是在核心规范中添加了 Bundle Wiring API 子规范，该规范引入了 Capabilities 和 Requirements 的概念。在此之前，OSGi 中的依赖单元要么是某个 Package，要么是整个 Bundle（分别使用 Import-Package 和 Require-Bundle 标识来描述），这种粒度的依赖单元能够满足代码上的依赖需要，却无法描述某些非代码的依赖特性，例如说明一个功能要依赖某个 Java 虚拟机版本、依赖某种架构的操作系统、依赖某些资源文件或依赖一定数量的 CPU 或内存等。以前虽然可以使用 Bundle-RequiredExecutionEnvironment 标识来描述部分执行环境的特性（如指明 JDK 版本是 Java SE 6），但是有一些特性不是在执行环境中天然存在的，是由某个 Bundle 安装后带来的，有了 Require-Capability 和 Provide-Capability 标识之后，才可以精确描述这类依赖关系。

继在 OSGi R4.2 中引入 Service Hooks 之后，OSGi R4.3 大幅增加了 OSGi 的 Hooks 挂接点数量，新增了 Weaving Hook、Resolver Hook、Bundle Hook、Weaving Hook 和 Service EventListener Hook。

Weaving Hook 让用户模块可以获得在其他类加载时动态植入增强的能力。Resolver Hook 和 Bundle Hook 代替了以前的 OSGi 框架嵌套和组合模块（Composite Bundle）的功能，

⊖ 因为只使用了泛型，没有使用其他新语言特性（如枚举、注解等），而 Java 的泛型又是使用擦除法实现的，所以接口仍然可以利用 javac -source 1.5 -target jsr14 编译出可运行于 Java 1.4 的版本。

让用户可以创建虚拟的模块集合，使不同集合之间的模块互不可见（这点在 OSGi R5 中提供了更完美的解决方案）。Service EventListener Hook 让用户可以插手服务事件分派的过程。

2012 年 7 月，OSGi R5 发布（同时发布了核心规范和企业级规范的 OSGi R5 版本），这是目前最新的 OSGi 规范版本。OSGi R5 的一个主要目标是建立一套基于 OSGi 的模块仓库系统（为下一步的 OSGi in Cloud 做准备）。Apache Maven 已经建成了类似的仓库系统，它的中央仓库中保存了 Java 业界中许多项目的依赖信息和 JAR 包。其实 OSGi 在这个领域本应有着得天独厚的优势，模块的元数据信息在某种意义上就是依赖描述的信息，但迟迟未在规范上踏出这一步。在 OSGi R5 出现之前，就已经有了 Equinox P2、OSGi Bundle Repository（OBR）等技术出现，在 OSGi R5 版规范中提出的 Subsystem Service 子规范和 Repository Service 子规范终于把这些技术统一起来。

OSGi R5 还对许多之前发布的子规范进行了更新和功能增强，例如 JMX Management Model 规范开始支持 Bundle Wiring API 了，Configuration Admin 在这个版本中能够支持多个 Bundle 共享同一个配置对象，声明式服务规范开始支持注解（这些注解用于供 BndTools 这类工具自动生成 XML 配置文件之用，实际上 OSGi 运行期还是没有使用泛型之外的 Java SE 5 后的语法特性）。

1.1.2　Java 模块化规范之争

经过近 20 年的发展，Java 语言已成为今日世界上最成功、使用的开发者人数最多的语言之一，Java 世界中无数商业的或开源的组织、技术和产品共同构成了一个无比庞大的生态系统。

与大多数开发人员的普遍认知不同，Java 的生态系统和演进路线并不是由 Sun Microsystems 公司[⊖]来决策和管理的。虽然 Sun 公司拥有“Java™”这个商标的所有权，并且拥有 Java 中使用最广泛的 HotSpot 虚拟机和 Sun JDK，但它并不能直接制定 Java 世界中的规则、确定 Java 技术的发展走向。

1998 年，在 Sun 公司的推动下成立了 JCP（Java Community Process）组织[⊜]，这个组织负责制定 Java 的技术标准，发布技术标准的参考实现（RI）和用于检验标准的技术兼容包（TCK）。目前，除了 Sun（目前是 Oracle 继承了 Sun 的席位）公司外，还有 Google、IBM、Motorola、Nokia、Sybase、SAP 等数以百计的公司、组织和独立个人参与了 JCP 组织，共同监督和维护 Java 标准的制定。这些参与者中的 16 位代表组成了 JCP 执行委员会（JCP Executive Committees，一共有两个这样的委员会，分别对应 Java SE/EE 方向和 Java ME 方向），对提交给 JCP 的提案进行投票表决。

JCP 允许任何组织或个人对 Java 的演进路线提出建议，这些建议在审查立项后会以 JSR（Java Specification Requests）的形式形成正式的规范提案文档；在经过专家组审核和执

⊖ Sun 公司已被 Oracle 公司收购，因本节涉及较多历史，为避免混乱，仍然沿用 Sun 公司的称谓。

⊜ JCP 组织官方主页地址：http://www.jcp.org。

行委员会投票通过之后，这些 JSR 文档就将成为正式的 Java 技术规范。J2EE、EJB、JSP、JDBC、JMX、JVMS 等规范都由对应的 JSR 文档“孵化”而来，甚至连 JCP 本身的组织和运作方式也是由特定的 JSR 文档进行定义的[⊖]。

以上介绍的内容虽然与 OSGi 没有直接关系，但是它们是读者了解后面介绍的 OSGi 与 Java 之间关系的必要背景知识。

OSGi 源于 JSR-8（Open Services Gateway Specification，开放服务网关规范），这是一份由 Sun 发起并主导（共同发起的还有 IBM、Ericsson 和 Sybase 等公司）、在 1999 年 3 月提交到 JCP 的规范提案。这份规范定义了不同设备之间服务互相依赖和调用的交互接口。1999 年，Java 2 刚发布不久，互联网也刚刚兴起，“支持各种移动设备、嵌入式设备、智能家电”这个最初建立 Java 语言的目标，对于 Java 来说依然是最重要领域之一。

不过，JSR-8 提案很快（1999 年 5 月，即提交之后的 2 个月）就被发起者撤回。撤回并不是因为这份 JSR 不够资格成为 Java 规范发布，主要是发起者希望另外建立一个独立于 JCP 的组织来发展运作这份规范，让更多不适合参与到 JCP 的设备厂商能够参与 OSGi 的规范制定。因此，1999 年独立于 JCP 的 OSGi 联盟成立，并于 2000 年发布了 OSGi 规范的第一个版本：OSGi R1.0。

在 OSGi 的前三个版本中，OSGi 主要领域依然维系在移动和嵌入式设备之上，在这三个版本的发展中 Sun 公司起了很大的推动作用，这段时间可谓是 OSGi 和 Sun 的蜜月期。比如 Sun 的 JES（Java Embedded Server）就是当时使用最广泛的 OSGi R2 的实现。从 OSGi R4 开始，OSGi 开始尝试跨越移动和嵌入式领域的限制，进入 Java SE/EE 领域，与此同时，OSGi 联盟的各个成员在发展和商业选择上也产生了分歧，各自（主要是 IBM 和 Sun）都在争夺 OSGi 联盟的主导权。在这个过程中各厂商是如何争夺规范控制权的我们不得而知，总之最终的结果是 Sun 公司于 2006 年离开了当年它一手主导建立的 OSGi 联盟。OSGi 规范分为面向 Java 主流领域的 OSGi R4 核心规范和依然专门面向嵌入式和移动领域的 OSGi 移动设备规范（即 JSR-232 Mobile Operational Management，OSGi R4 Mobile 与 JSR-232 的内容是完全一致的）。

在今天看来，Sun 的离开恰恰证明了当时它在 Java 模块化方向上的错误。OSGi R4 的目标平台转变为 Java SE/EE，进军桌面、服务端和互联网的举措获得了很大的成功，关于这点不得不提到在 IBM 支持下 Eclipse 基金会对 OSGi 快速流行所做出的贡献。自 Eclipse 3.0 M4 版本开始，这款著名的集成开发工具被重构为完全基于 OSGi 架构实现的产品，支持 Eclipse 运行的底层框架 Equinox 成为 OSGi R4.x 使用最广泛的实现框架。伴随着 Eclipse IDE 的流行，OSGi 迅速在 Java ME 以外的领域站稳脚跟。许多人（包括笔者）都是从 Eclipse 开始关注 OSGi 的，IBM 的很多后续产品，如 WebSphere、JAZZ 等都继续支持 OSGi 或直接基于

⊖ JSR-215：Java Community Process (JCP)。

OSGi 来构建，其他公司也迅速跟进[㊀]。

OSGi R4 的迅速流行带来一个强烈的信号：Java SE/EE 支持模块化已经成为一股不可逆转的潮流，支持模块化的呼声已经强大到令 Sun 公司不能再忽视的程度。Sun 公司期望能借助 JCP 的力量重新争夺 Java 模块化规范的控制权。

2006 年 10 月，由 Sun 公司提交的 JSR-277 规范提案（Java Module System，Java 模块化系统）发布了第一个早期预览版（Early Draft Review）；2007 年年底，Sun 携带着 JSR-277 重新加入 OSGi 联盟。

JSR-277 是一个全静态的模块化规范，它在模块化和依赖描述方面与 OSGi 有着相似的能力，在构建模块仓库（Repository）上对比当时的 OSGi 规范占有一定优势（OSGi 这方面的弱点在 2012 年 7 月 OSGi R5 发布并拥有了 Subsystem Service 和 Repository Service 之后已经被填补）。但是 JSR-277 是完全静态的，没有任何关于动态化的考虑，这样模块就无法在运行时安装、更新和卸载，因此也就不存在类空间一致性、类和类加载器卸载等问题。这点相对于 OSGi 的动态模块化来说是极大的退步。另外 JSR-277 引入新的“.jam”格式文件作为模块分发格式也被大家所诟病。OSGi 技术专家、OSGi 联盟主席 Peter Kriens 在 OSGi 联盟的官方博客上发表文章批评道：“JSR-277 的目标如同儿戏，只能相当于 OSGi 在 8 年前的技术水平[㊁]”。

在 JCP 中争夺模块化规范控制权，对于 Sun 来说会比在 OSGi 联盟中更为有利。Sun 在 JCP 拥有很大的影响力，它是 JCP 的发起者，担任 JCP 的主席，在执委会中拥有无须选举的无限期执委会投票权（其他 15 个执委席位三年选举一次），Sun 肯定希望能永远保持在 JCP 中的领导地位，但是 JCP 的其他成员都希望能够在 Java 的规范和发展路线上拥有更大的话语权。这样，不可避免会产生一些利益冲突。Sun 力挺 JSR-277，希望用它代替 OSGi 成为 Java 模块化标准，IBM 则将 OSGi R4.1 提交到 JCP 成为 JSR-291（Dynamic Component Support for Java SE，Java SE 动态组件支持）来与 JSR-277 对抗，这样，在 OSGi 联盟中的规范之争的战场又重新回到 JCP 之中。

这两个 JSR 竞争的结果是 JSR-277 没有得到通过，尽管 Sun 曾经做出了一些让步，比如承诺 JSR-277 可以引用和操作 OSGi 的遗留模块，它最终还是没有得到足够的支持，被废止在早期预览版阶段。另一方面，在对 JSR-291 进行表决时，虽然 Sun 明确投了反对票，但是 JSR-291 仍然在 JCP 执委会最终投票中通过。这样，OSGi 终于确立了 Java 唯一的模块化规范的地位。不过，事情并没有结束，JSR-291 得到通过并不意味着 OSGi 规范立刻就会成为现实。

虽然 OSGi R4.1 在 2007 年 5 月通过投票之后就应该是正式的 Java 规范，但是 Sun 依然

㊀ 业界其他著名的 OSGi 项目可见本书附录 D。

㊁ 原文为：“JSR 277 proposal feels rather toyish...just like the OSGi framework minus 8 years of experience”。全文地址：http://blog.osgi.org/2006/10/jsr-277-review.html。

在 JSR-316（Java Platform, Enterprise Edition 6 Specification，Java 企业版规范第 6 版，这个规范提案在 2007 年 7 月提交给 JCP，2009 年 12 月发布最终版，提交时 JSR-277 与 JSR-291 之争已尘埃落定）的规范目标中明确写道：“为了更好地支持这个平台（指 Java EE 6）在扩展性方面的目标，应该有一个更加宽泛的模块化概念。这项工作正由‘JSR-277 Java 模块系统’来实现，它的目标平台是 Java SE 7。我们预期 Java EE 7 将建立在这项技术的基础上，因此我们将推迟任何可能与将来的版本冲突的技术规范”。在这段话中所谓的“可能与将来的版本冲突的技术规范”毫无疑问就是 JSR-291 了。Sun（当时已经被 Oracle 收购了）这种一意孤行地坚持自己的 JSR-277 而无视 JSR-291 的行为招来了许多非议[㊀]。但是非议无法解决问题，Sun 仍然实质性地控制着 JDK 的发展，最终结果就如 JSR-316 中的那句话所表达的那样，整个 Java SE 6 期间没有任何模块化相关的改进以 JDK 功能的形式进入到 Java 平台中。Sun 对待 OSGi 的态度不应解读为 Sun 反对 Java 模块化，相反，这是 Sun 极为重视 Java 模块化的体现，它一定要把 Java 模块化的主导权抓在手中。尽管 Sun 的做法拖延了 Java 模块化的进程，但模块化依然是不可避免地向前发展了，到 Java SE 7 中又如何呢？

讲到 Java SE 7 的模块化，我们不得不再多介绍一个规范提案：JSR-294（Improved Modularity Support in the Java Programming Language，Java 语言的模块化改进）。如此之多（提交的时间很集中，并且是并行发展的，在 JSR-294 提交时 JSR-277 并没有被废止）的 JSR 来解决重复的问题在 JCP 历史上也是极为罕见的。这个规范提案的提交者也是 Sun 公司，它试图在 Java 语言和 Java 虚拟机层面上对模块化进行支持，直接修改 JLS（Java 语言规范）和 JVMS（Java 虚拟机规范），加入 module 关键字和超级包（Super Package）等概念。JSR-294 的模块化实现思想与 OSGi 的差异是非常大的，通俗地讲，OSGi 是在 Java 平台之上建立的模块化，而 JSR-294 是直接在 Java 平台之内建立的模块化。

从纯粹技术角度来看，Java 模块化如果真能通过直接修改 Java 语法、Class 文件格式和 Java 虚拟机来实现，我们相信这种实现方案的性能、完善程度肯定能够超越 OSGi，仅从技术角度看，这种改进方式无疑是 Java 模块化的最佳结果。不过，修改 Java 语法在 JCP 中历来都是“慎重而敏感的话题”，增加新的语言特性相对缓慢，这点也是社区管理的劣势。对比一下微软一家单独掌控的 C# 语言，就能看出明显差距。

话题再回到 JSR-294，很遗憾，这个规范提案的结果与 JSR-277 一样，也被废弃在早期预览版阶段[㊁]。Sun 似乎早就预料到这个结果，它在提交 JSR-294 的同时，就在 OpenJDK 中启动了 Jigsaw 项目[㊂]的孵化进程。Jigsaw 项目最开始的目标是作为 JSR-294 的参考实现

㊀ 例如：http://www.eclipsezone.com/eclipse/forums/t98330.rhtml。

㊁ 根据 Alex Buckley（JSR-294、277 的 Specification Leader）的回应，JSR-294 的废止并非没有得到足够支持，而是由于提交人超过提案发布时间（18 个月）未发布早期预览版规范而自动废止的，此回应的邮件可参见：http://altair.cs.oswego.edu/pipermail/jsr294-modularity-observer/2009-December/000352.html。

㊂ Jigsaw 项目地址：http://openjdk.java.net/projects/jigsaw/。

（RI），但是该项目的开发过程却是在 jigsaw-dev 邮件列表上进行的，该邮件列表游离于 JSR-294 专家组的邮件列表之外。目前的种种迹象表明，Sun 决定让 Jigsaw 项目采取“SunJDK 专有的方式”来实现 Java 语言模块化，JSR-294 没有得到通过，也就意味着 Jigsaw 项目是 Sun 私有的，使用了 Jigsaw 的 Java 程序无法运行于其他公司提供的 JDK 之上。因此，即使 Jigsaw 本身的设计再好，只要无法做到“一次编译，到处运行”的模块化，就必然是对 Java 语言最重要一块基石的巨大损害。

很难相信 Sun 最后会以私有化的方式强行推出 Jigsaw，这是非常不明智的。最后的结果肯定还要重新激活 JSR-294，或者再提交另外一个 JSR 使之在 JCP 上通过，在此之前，Jigsaw 只能无限期拖延下去。目前 Jigsaw 已经拖得足够长了，最初它是作为 Java SE 7 的特性进行设计的，后来因 Java 7 进度压力被推迟到 Java 8 之中，在 2012 年 7 月，在 Jigsaw 项目的主页上宣布它将进一步被延迟到 Java 9 中发布。这样导致的结果是，即使后续一切顺利，用户也要到 2015 年 9 月才能见到 Jigsaw，那时已经是 OSGi 出现的 16 年之后了；再等到 Java 9 被应用到主流的生产环境中，Jigsaw 就显得更加遥遥无期了。如果系统要兼容 Java 2 至 Java 8 平台，OSGi 还是唯一的选择。Jigsaw 不得不正视 OSGi 事实上的模块化规范的地位，建立了一个名为的 Penrose[⊖]子项目让 Jigsaw 可以与 OSGi 互相操作。

目前 OSGi 是 Java 世界中唯一的模块化规范，从纯技术角度看，它未必是最先进的模块化技术，从学习使用来看，它也不是使用最简单方便的模块化技术。但是从整个 Java 业界整体来看，OSGi 确实是过去、现在乃至未来至少 5 年内可预见的最有生命力、最标准、使用范围最广泛的 Java 模块化技术。随着 Java 应用规模的日益庞大，越来越多的大型系统使用 OSGi 架构进行建设，因此，OSGi 是具有广阔发展前景和使用、学习价值的。

1.2　为什么使用 OSGi

没有什么技术是万能的，任何一门技术都有它的适用场景和最佳实践方法。OSGi 不只是一门技术，更多的是一种做系统架构的工具和方法论，如果在不适用的场景中使用 OSGi，或者在适用的场景中不恰当地使用 OSGi，都会使整个系统产生架构级的缺陷。因此，了解什么时候该用 OSGi 是与学会如何使用 OSGi 同样重要的事情。

每个系统遇到的业务环境都是不一样的，笔者不希望以经验式的陈述去回答“什么时候该用 OSGi”或“为什么要使用 OSGi”这样的问题，而试图通过几个问题的讨论和利弊权衡，让读者自己去思考为什么这些场景适用 OSGi。如果读者是第一次通过本书接触 OSGi，那么可能对某些讨论的内容会感到困惑，笔者建议尽可能在阅读完全书或者在准备真正在项目中使用 OSGi 的时候，再回过头读一遍本节的内容。

⊖　Penrose 项目地址：http://openjdk.java.net/projects/penrose/。

1.2.1 OSGi 能让软件开发变得更容易吗

不可否认，OSGi 的入门门槛在 Java 众多技术中算是比较高的，相对陡峭的学习曲线会为第一次使用 OSGi 开发系统的开发人员带来额外的复杂度。

OSGi 规范由数十个子规范组成，包含了上千个不同用途的 API 接口。OSGi 规范显得这样庞杂的主要原因是实现“模块化”本身需要解决的问题就非常多。模块化并不仅仅是把系统拆分成不同的块而已——这是 JAR 包就能做的事情，真正的模块化必须考虑到模块中类的导出、隐藏、依赖、版本管理、生命周期变化和模块间交互等一系列的问题。

鉴于 OSGi 本身就具有较高的复杂度，“引入 OSGi 就能让软件开发变得更容易”无论如何是说不通的，小型系统使用 OSGi 可能导致开发成本更高。但是这句话又不是完全错误的，随着系统不断发展，在代码量和开发人员都达到一定规模之后，OSGi 带来的额外成本就不是主要的关注点了，这时候的主要矛盾是软件规模扩大与复杂度随之膨胀间的矛盾。如图 1-3 所示，代码量越大、涉及人员越多的系统，软件复杂度就会越高，两者成正比关系。这个观点从宏观角度看是正确的，具体到某个系统，良好的架构和设计可以有效减缓这个比率。基于 OSGi 架构的效率优势在这时候才能体现出来：模块化推动架构师设计出能在一定范围内自治的代码，可以使开发人员只了解当前模块的知识就能高效编码，也有利于代码出现问题时隔断连锁反应。OSGi 的依赖描述和约束能力，强制开发人员必须遵循架构约束，这些让开发人员“不自由”的限制，在系统规模变大后会成为开发效率的强大推动力。

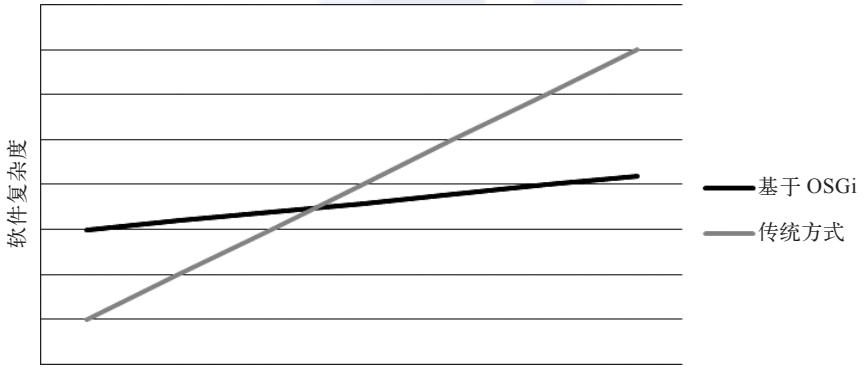

图 1-3 引入 OSGi 对软件复杂度的影响

可以用一个更具体的场景来论述上面的观点，解析 OSGi 架构如何在开发效率上发挥优势。有经验的架构师会有这样的感受：设计一个具有“自约束能力”的系统架构非常不容易。最常见的情况是设计人员设想得很美好，开发人员在实现时做出来的产品却不是那样。大部分软件公司是通过“开发过程”、“编码规范”、“测试驱动”，甚至“人员熟练度”来保证开发人员实现的代码符合设计人员的意图。这样即使在开发阶段做到符合设计需求，也很难保证日后维护人员能够继续贯彻原有的设计思想；随着开发的时间越来越长，系统最终实现的

样子可能和原有的设计产生越来越大的偏差。在软件工程中，将这种现象称为“架构腐化”。架构的“自约束能力”就是指限定不同开发人员在实现功能的时候，实现方式都是一致的，最好只有唯一一条遵循设计意愿的路可走，别的方法无法达到目的。更通俗地说就是，尽可能使程序员不写出烂代码。

举个最浅显的例子，如果有开发人员在 Web 层中使用 DAO 直接操作数据库，或者在 DAL 层直接从 HttpSession 对象中取上下文信息，这样的代码也许能逃过测试人员的黑盒测试，但是显然是不符合软件开发基本理论的。前者可能因绕过 Service 层中的事务配置而出现数据安全问题；后者限制了这样的 DAO 就只能从 Web 访问，无法重用和进行单元测试。如果项目中出现这样的代码，笔者认为首要责任在架构师，因为架构师没有把各层的依赖分清，如果 Web 层只依赖 Service 层的 JAR 包，那么程序员就无法访问到 DAO，如果 DAL 层没有依赖 Servlet API 的 JAR 包，那么程序员就不可能访问 HttpSession 对象，这就是一种架构缺乏自约束能力的表现。

大概没有哪个架构师会犯上面例子那样幼稚的错误。但是，实际情况也远比例子中的复杂，甚至有一些问题是 Java 语言本身的缺陷带来的，例如，依赖了一个 JAR 包就意味着能够访问这个 JAR 包中的一切类和资源，因为 JAR 包中的内容没有 Public、Private 和 Protected 之分，无法限制用户能访问什么、不能访问什么。更复杂的情况是在引入了同一个 JAR 包的不同版本时怎么办？如果依赖包需要动态变化怎么办？使用 OSGi 一个很重要的目的就是弥补 Java 中资源精细划分的缺陷，加强架构的自约束能力。

虽然 OSGi 起源于精小软件占多数的嵌入式领域，但是在 Java SE/EE 领域中，对于越庞大的系统，使用 OSGi 进行模块化拆分就越能发挥出优势。在商业上已经有一些使用 OSGi 控制软件复杂度增长、延缓架构腐化速度的成功案例，如 Eclipse Marketplace，它已经拥有了上千个插件，插件的开发者来自全球各地，技术水平差异很大，插件实现的功能也各不相同，是 OSGi 让这些插件基本遵循了统一的架构约束，并且一般不会因为某个插件的缺陷影响整个 Eclipse 的质量。

1.2.2　OSGi 能让系统变得更稳定吗

笔者遇到过许多由 OSGi 框架引发的问题，例如，最典型的 ClassNotFoundException 异常、类加载器死锁或者在动态环境下的 OutOfMemoryError 问题等，这些都是基于 OSGi 架构开发软件时很常见的。从这一方面看，使用 OSGi 确实会增加系统不稳定的风险，所以，在开发过程中团队中有一两个深入了解 OSGi 的成员是必要的。

不过，软件是否稳定不是只看开发阶段可能出现多少异常就能衡量的，软件的“稳定”应是多方面共同作用的结果。除了关注开发阶段是否稳定之外，还要关注是否能积累重用稳定的代码，问题出现时能否隔断连锁反应蔓延，缺陷是否容易修复等。在这些方面，OSGi 就可以带来相当多的好处，例如：

- OSGi 会引导程序员开发出可积累可重用的软件。我们无法要求程序刚开发出来就是完全稳定的，但可以在开发过程中尽可能重用已稳定的代码来提升程序质量。大家知道，写日志可以使用 Log4j，做 ORM 会引入 Hibernate，Java 中有许多经过长期实践检验的、被证实为稳定的开源项目，这些开源项目的共同特征是都经过良好的设计，能够很方便地在其他项目中使用。相对而言，在自己开发项目时很多人没有注意到要进行可积累的设计。一种典型现象是项目中出现一些“万能的包”，通常名字会是 XXXCommons.jar、XXXUtils.jar 等，这些包中存放了在项目中被多次调用的代码，但是这样的包不能叫做可重用包。当这些包越来越大、类越来越多、功能越来越强时，与这个项目的耦合就越紧密，一般也就无法用在其他项目中了。在 OSGi 环境下，“大杂烩”形式的模块是很难生存的，如果某个模块有非常多的依赖项，那么没有人愿意为了使用其中少量功能去承担这些间接依赖的代价。因此设计者必须把模块设计得粒度合理，精心挑选对外发布的接口和引入的依赖，把每个模块视为一个商业产品来对待，这样才能积累出可重用的模块，也利于提高程序稳定性。
- 基于 OSGi 比较容易实现强鲁棒性的系统。普通汽车坏掉一个轮胎就会抛锚，但是飞机在飞行过程中即使坏了其中一个引擎，一般都还能保持正常飞行。对于软件系统来说，如果某一个模块出了问题，能够不波及其他功能的运作，这也是稳定性的一种体现。大多数系统都做不到在某部分出现问题时隔离缺陷带来的连锁反应。试想一下，在自己做过的项目中把 Common Logging（或 slf4j）的包拿掉，系统能只损失日志功能而其他部分正常运作吗？但是对于基于 OSGi 架构开发系统，在设计时自然会考虑到模块自治和动态化，当某部分不可用时如何处理是每时每刻都会考虑的问题，如果软件在开发阶段跟随着 OSGi 的设计原则来进行，自然而然会实现强鲁棒性的系统。
- 在 OSGi 环境下可以做到动态修复缺陷。许多系统都有停机限制，要求 7×24 小时运行，对于这类系统，OSGi 的动态化能力在出现问题时就非常有用，可以做到不停机地增加或禁止某项功能、更新某个模块，甚至建立一个统一更新的模块仓库，让系统在不中断运行的情况下做到自动更新升级。

1.2.1 节和 1.2.2 节提出的两个问题可以总结为 OSGi 是否能提升开发效率和软件质量。OSGi 在这两方面的作用与软件设计得是否合理关系非常密切，这时 OSGi 好比一个针对“设计”这个因素的放大杠杆，配合好的设计它会更加稳定、高效，而遇到坏的设计，反而会带来更多问题。

1.2.3 OSGi 能让系统运行得更快吗

系统引入 OSGi 的目的可能有很多种，但一般不包括解决性能问题。如果硬要说 OSGi 对性能有什么好处，大概就是让那些有“系统洁癖”的用户可以组装出为自己定制的系统

了。例如 GlassFish v3.0 服务器是基于 OSGi 架构的，它由 200 多个模块构成，如果不需要 EJB 或 JMS 这类功能，就可以把对应的模块移除掉，以获得一个更精简的服务器，节省一些内存。总体上讲，OSGi 框架对系统性能是有一定损耗的，我们从执行和内存两方面来讨论。

首先，OSGi 是在 Java 虚拟机之上实现的，它没有要求虚拟机的支持，完全通过 Java 代码实现模块化，在执行上不可避免地会有一些损耗。例如，OSGi 类加载的层次比普通 Java 应用要深很多，这意味着需要经过更多次的类加载委派才能找到所需的类。在两个互相依赖的模块间发生调用时，可能会由于类加载器互相锁定而产生死锁；要避免死锁的出现，有时候不得不选用有性能损失的串行化的加载策略[⊖]。在服务层上，动态性（表现为服务可能随时不可用）决定了应用不能缓存服务对象，必须在每次使用前查找，这种对 OSGi 服务注册表的频繁访问也会带来一些开销。使用一些具体的 OSGi 服务，例如使用 HTTP Service 与直接部署在 Web 容器中的 Servlet 相比会由于请求的桥接和转发产生一些性能损耗。

其次，从内存用量来看，OSGi 允许不同版本的 Package 同时存在，这是个优点，但是客观上会占用更多内存。例如，一个库可能需要 ASM 3.0，而同一应用程序使用的另一个库可能需要 ASM 2.0，要解决这种问题，通常需要更改代码，而在 OSGi 中只需要付出一点 Java 方法区的内存即可解决。不过，如果对 OSGi 动态性使用不当，可能会因为不正确持有某个过期模块（被更新或卸载的模块）中一个类的实例，导致该类的类加载器无法被回收，进而导致该类加载器下所有类都无法被 GC 回收掉。

仅从性能角度来说，OSGi 确实会让系统性能略微下降，但是这完全在可接受范围之内。使用 OSGi 开发时应该考虑到性能的影响，但不应当将其作为是否采用 OSGi 架构的主要决策依据。

1.2.4　OSGi 能支撑企业级开发吗

不管关于“OSGi 是否能支撑企业级开发”的讨论结果如何，一个必须正视的事实是 OSGi 对企业级开发的支撑能力正在迅速增强。从 2007 年 OSGi 联盟建立企业专家组以来，OSGi 的发展方向已经逐渐调整到企业级应用领域。在 IBM、Apache 和 Eclipse 基金会等公司和组织推动下，企业级 OSGi 正在变得越来越成熟。

在企业级 OSGi 出现之前，企业级开发要么是走 Java EE 的重量级路线，要么是走 SSH 的轻量级路线。企业级 OSGi 被引入后并没有扮演一个“革命者”的角色，没有把 Java EE 或 SSH 中积累的东西推倒重来，OSGi 更像是在扮演一个“组织者”的角色，把各种企业级技术变为它的模块和服务，使以前的企业级开发技术在 OSGi 中依然能够发挥作用。

OSGi 企业级规范中定义了 JDBC、JPA、JMX、JTA 和 JNDI 等各种 Java EE 技术以及

⊖ 例如 Equinox 中的 osgi.classloader.singleThreadLoads 机制。即使有了这个机制，非树状类加载架构下还是有一些情况仅靠 OSGi 本身是没有办法解决的，需要等到 Java 7 中类加载器结构改进才能解决，可参见：http://openjdk.java.net/projects/jdk7/features/#f352。

SCA、SDO 这些非 Java EE 标准的企业级技术在 OSGi 环境中的应用方式，这些容器级的服务都可以映射为 OSGi 容器内部的服务来使用。并且到现在，企业级规范定义的内容已经不仅停留在规范文字中，已经有不少专注于 OSGi 企业级服务实现框架出现（例如 Apache Aries [⊖]）了。

另一方面，OSGi 的 Blueprint 容器规范统一了 Java 大型程序中几乎都会用到的依赖注入（DI）方式，使基于 Blueprint 的 OSGi 模块可以在不同的 DI 框架中无缝迁移。这个规范得到 Apache、SpringSource 等组织的大力支持，目前这些组织已经发布了若干个 Blueprint 规范的实现容器（例如 Apache Geronimo 和 Equinox Virgo，Virgo 前身就是 SpringSource 捐献的 Spring DM 2.0）。在最近两三年时间里，企业级 OSGi 成为 Java 社区技术发展的主要方向之一，其发展局面可以说是如火如荼。

不过，我们在使用企业级 OSGi 的时候也要意识到它还很年轻，其中很多先进的思想可能是遗留程序根本没有考虑过的，还有不少问题的解决都依赖于设计约束来实现。因此，如果是遗留系统的迁移，或者设计本来就做得不好，那么使用 OSGi 会遇到不少麻烦。以最常见的数据访问为例，如果以前遗留系统使用了 ORM 方式访问数据库，而迁移到 OSGi 时没有把实体类统一抽取到一个模块，那么 ORM 模块的依赖就很难配置了，这时不得不使用 Equinox Buddy 甚至 DynamicImport-Package 这类很不优雅的方式来解决。另一个问题是集群，OSGi 拥有支持分布式的远程服务规范，而 OSGi 的动态性是针对单 Java 虚拟机实例而言的，因此要在集群环境下保持 OSGi 的动态性，就必须自己做一些工作才行。

1.3 本章小结

在本书中，笔者尝试阐述与 OSGi 相关的三个问题：什么是 OSGi？为什么要使用 OSGi？以及如何使用 OSGi？在本章中，我们已经解答了前两个问题，本书后面的 13 章都将围绕“如何使用 OSGi”这个问题来进行，通过研读 OSGi 核心规范、分析最常用的 OSGi 实现框架 Equinox 以及与读者一起通过实践来了解如何使用 OSGi 技术开发程序。

⊖ Apache Aries 官方主页：http://aries.apache.org/。

第二部分

OSGi 规范与原理

第 2 章　模块层规范与原理

从本章开始，我们将为读者讲解 OSGi 中最常用和最重要的内容。“最常用和最重要”意味着并不会涵盖 OSGi 规范的所有方面。“讲解”也不是对 OSGi 规范的直接翻译，笔者简化了 OSGi 规范中一些近乎于数学公式的严谨描述，改用尽可能通俗的语言进行介绍，并添加了目前业界在实际应用中使用规范中定义的内容的例子。

如果要学习 Java 语言，相信没有人会推荐从《Java 语言规范》和《Java 虚拟机规范》学起，因为刚开始接触 Java 的人没有必要把所有 Java 语言和 Java 虚拟机的细节都记下来。同样，对于之前没有接触过 OSGi 的读者，没有必要一字不漏地把本部分内容读完，这很枯燥。我们推荐初学者花上一、两个小时把本书第一部分的内容大致浏览一遍，然后迅速转入第三部分，配合第三部分的应用案例来学习会更有效率。如果读者此前已有基于 OSGi 的开发经验，或者已经阅读过本书第三部分，那么细读这部分 OSGi 规范内容将有助于了解 OSGi 的原理和完整的面貌。了解 OSGi 的规范和原理，是深入理解 OSGi 必不可少的过程。

2.1　OSGi 规范概要

目前最新的 OSGi 规范是 2012 年 7 月发布的 Release 5，Version5.0（后文简称为 R5.0）⊖版本，该规范定义了 Java 模块化系统所涉及的各种场景（开发、打包、部署、更新和交互等），以及其中用到的标准接口和参考模型。它是一份内容很全面、涉及范围很广泛的技术规范，从嵌入式系统到大型服务器系统，从模块的编码开发到部署使用，从 OSGi 核心框架到外围扩展服务都有专门的定义。

OSGi 规范并不是单一的规范文档，而是由一系列子规范构成，这些子规范主要可分为两大部分，其中一部分用于描述 OSGi 的核心框架（OSGi Framework）。OSGi 核心框架是一个可运行 OSGi 系统的最小集合，它由以下内容组成：

- **执行环境**（Execution Environment）。由于 OSGi 所适用的目标范围非常广泛，为了更好地处理不同硬件、软件环境对 OSGi 造成的兼容性问题，在建立其他约定之前，必须先定义好系统的执行环境。
- **安全层**（Security Layer）。描述了基于 Java 2 安全架构实现的代码验证、JAR 文件数

⊖ OSGi 的核心规范（Core Specification）和企业级服务规范（Enterprise Specification）更新到了 R5.0 版。但截至笔者完成初稿时，最新的服务纲要规范（Compendium Specification）仍是 R4.3 版，最新的 Mobile 规范（本书不会涉及）为 R4.0 版，Mobile 规范已经停止更新，其中的子规范大多并入到了服务纲要规范之中。

字签名、数字证书服务，安全层贯穿了 OSGi 框架的其他各个层次。

- **模块层**（Module Layer）。模块层从“静态”的角度描述了一个模块的元数据信息、执行环境定义、模块约束和解析过程、类加载顺序等内容。模块层是整个 OSGi 中最基础、最底层的层次。
- **生命周期层**（Life Cycle Layer）。生命周期层从“动态”的角度描述了一个模块从安装到被解析、启动、停止、更新、卸载的过程，以及在这些过程中的事件监听和上下文支持环境。
- **服务层**（Service Layer）。描述了如何定义、注册、导出、查找、监听和使用 OSGi 中的服务。服务层是所有 OSGi 标准服务的基础。
- **框架 API**（Framework API）。由一系列通过 Java 语言实现的接口和常量类构成，为上面各层提供面向 Java 语言的编程接口。

构成 OSGi 规范的另外一部分内容是 OSGi 标准服务，这些标准服务试图以 OSGi 为基础，在软件开发的各种场景中（如配置管理、设备访问、处理网络请求等），建立一套标准服务和编程接口。软件开发所遇到的场景是多种多样、极其复杂的，因此 OSGi 对应定义的标准服务也非常庞大和复杂，OSGi 所包含的数十个子规范大部分都用于定义这些标准服务。以下列举了一小部分较为常用的 OSGi 标准服务。

- 事件服务（Event Admin Service）
- 包管理服务（Package Admin Service）
- 日志服务（Log Service）
- 配置管理服务（Configuration Admin Service）
- HTTP 服务（HTTP Service）
- 用户管理服务（User Admin Service）
- 设备访问服务（Device Access Service）
- IO 连接器服务（IO Connector Service）
- 声明式服务（Declarative Services）
- 其他 OSGi 标准服务

大部分 OSGi 标准服务都没有写入 OSGi 核心（Core）规范之中，而是定义在 OSGi 服务纲要（Service Compendium）规范和企业级（Service Enterprise）规范之中。从上面对 OSGi 规范简要介绍我们可以总结出来，平时所说的“OSGi”大致包含了如图 2-1 所示的内容。

虽然伴随着 OSGi 规范文档还发布了一些代码性质的内容，例如 XML Schema 定义和少量的 JAR 包，但是这些 JAR 包仅仅包含 OSGi 框架 API 及一些标准服务的接口。换句话说，仅仅靠这些随规范发布的代码是无法建立一个可运行的 OSGi 系统的。要让 OSGi 运行起来，还需要具体实现 OSGi 规范的程序才行，我们把这些程序称为“实现框架”或“OSGi 实现”，如 Eclipse 的 Equinox、Apache 的 Felix 和 Makewave 的 Knopflerfish 等都是常见的 OSGi 实现。

图 2-1 OSGi 内容总览

2.2 Bundle

从本节开始，一直到本书结束，我们都会不断地提起“Bundle”这个词。Bundle 是 OSGi 中最基本的单位，通俗地讲，如果说 OSGi 是基于 Java 平台的“模块化开发体系”，那么 Bundle 便是其中的“模块”。

OSGi 中的 Bundle 是在 JAR 文件格式规范[⊖]基础上扩展而来的，一个符合 OSGi 规范的 Bundle 首先必须是一个符合 JAR 文件格式规范的 JAR 包。与 JAR 文件格式兼容这点虽然没有太多技术含量可言，但是这个简单的举措极大地加速了 OSGi 的发展传播，它令 OSGi 的 Bundle 可以不经任何修改就直接应用于非 OSGi 的系统之中，也为将非 OSGi 的 JAR 包转换为可在 OSGi 系统运行的 Bundle 提供了很大的便利。

Bundle 相对普通的 JAR 文件主要进行了以下三个方面扩展。

- JAR 文件格式规范里定义的 /META-INF/MANIFEST.MF 文件用于描述 JAR 包的元数据信息，如 JAR 包的版本、数字签名信息等，Bundle 在 MANIFEST.MF 文件中添加了大量扩展定义，如描述该 Bundle 可以提供哪些资源、依赖哪些其他 Bundle、启动或卸载时要执行哪些动作等，这部分内容我们会在 2.3 节中详细介绍。
- 加入了一个可选的 /OSGI-OPT 文件夹，可以在其中保存一些与 Bundle 运行无关的信息，比如 Bundle 源码、软件说明书等。Bundle 的使用者可以从中获取一些额外的信息，也可以安全地删除该文件夹，以节约 OSGi 系统的存储空间。

⊖ JAR 文件格式规范：http://download.oracle.com/javase/6/docs/technotes/guides/jar/jar.html。

- Bundle 中可以包含一些具备特殊含义的程序和资源，如使用 Bundle-Activator 定义的初始化类、定义在 OSGI-INF/l10n 目录中的本地化信息等。

Fragment Bundle 是一种特殊的 Bundle，它无法独立存在，必须依附于某个其他的普通 Bundle 来使用，可以将它视为“Bundle 的插件”、“模块中的模块”。

Fragment Bundle 经常用来提供某些可选的功能，譬如为某个实现具体功能的 Bundle 提供一个中文语言包。有这个语言包，实现功能的 Bundle 能显示中文界面；在没有这个中文语言包时，实现功能的 Bundle 也能够正常使用。Fragment Bundle 的另一项主要用途是隔离 Bundle 中经常变动的部分，譬如把系统的内部配置文件（开发模式还是生产模式、连接的数据库地址、调试级别等）集中在 Fragment Bundle 中，通过更换不同的 Fragment Bundle 来实现配置快速切换。

从静态角度（开发期）来看，Fragment Bundle 与普通 Bundle 没有太大区别，它们都以 JAR 文件格式为基础，具备相同的元数据信息标记，标记的含义与设置方式也一样。区别仅仅是 Fragment Bundle 的元数据中会使用 Fragment-Host 标记说明它的宿主 Bundle。

从动态角度（运行期）来看，Fragment Bundle 与普通 Bundle 在运行时的处理差别却非常大，最重要的一点差异是 Fragment Bundle 不具备自己独立的类加载器。OSGi 利用每个 Bundle 独立的类加载器互相协作来维护 Bundle 间导入、导出的依赖关系。没有类加载器，就无法直接与其他 Bundle 交互，必须依附于宿主，使用宿主 Bundle 的类加载器完成。关于这部分内容，我们在后面会有更详尽的介绍。

2.3　描述元数据

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

2.3.1　预定义标记

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

（1）Bundle-ActivationPolicy

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

示例：

```
Bundle-ActivationPolicy: lazy
```

（2）Bundle-Activator

标记 Bundle-Activator 指明一个 Activator 类，在 Bundle 启动和停止时会分别调用该类的 start() 和 stop() 方法，以便执行程序员所希望的动作，该类必须实现 org.osgi.framework.BundleActivator 接口。

Activator 类通常用于在 Bundle 启动时注册和初始化服务，在 Bundle 卸载时注销这些服务。它很常用，但并不是必须的。

示例：

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

（3）Bundle-Category

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

示例：

```
Bundle-Category: osgi, test, nursery
```

（4）Bundle-Classpath

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

示例：

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

（5）Bundle-ContactAddress

标记 Bundle-ContactAddress 描述 Bundle 发行者的联系信息，仅供人工阅读，OSGi 框架并不会使用它。

示例：

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

（6）Bundle-Copyright

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

示例：

```
Bundle-Copyright: OSGi (c) 2002
```

（7）Bundle-Description

标记 Bundle-Description 给出关于该 Bundle 的简短描述信息，仅供人工阅读，OSGi 框

架并不会使用它。

示例：

```
Bundle-Description: Network Firewall
```

（8）Bundle-DocURL

标记 Bundle-DocURL 给出该 Bundle 文档的链接地址，仅供人工阅读，OSGi 框架并不会使用它。

示例：

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

（9）Bundle-Icon

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

示例：

```
Bundle-Icon: /icons/acme-logo.png;size=64
```

（10）Bundle-License

标记 Bundle-License 给出该 Bundle 的授权协议信息。

示例：

```
Bundle-License: http://www.opensource.org/licenses/jabberpl.php
```

（11）Bundle-Localization

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

示例：

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

（12）Bundle-ManifestVersion

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

示例：

```
Bundle-ManifestVersion: 2
```

（13）Bundle-Name

标记 Bundle-Name 定义该 Bundle 的名称。注意该名称只供人工阅读，在 Bundle-SymbolicName 标记中定义的名称才会作为程序使用的 Bundle 的唯一标识来使用。根据一般开发习惯，

Bundle-Name 中所定义的名称会在打包发布时与 Bundle-Version 一起构成该 Bundle 的文件名，所以这个名称一般不含空格或其他不能在文件名中出现的字符。

示例：

```
Bundle-Name: Firewall
```

（14）Bundle-NativeCode

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

- ❑ osname：操作系统名称，如 Windows 等。
- ❑ osversion：操作系统版本号，如 3.1 等。
- ❑ processor：处理器指令集架构，如 x86 等。
- ❑ language：遵循 ISO 编码的语言，如 en，zh 等。
- ❑ seleciton-filter ：选择过滤器，该值为一个过滤器表达式，指定被选中或未被选中的本地代码。

示例：

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

（15）Bundle-RequiredExecutionEnvironment

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

示例：

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

（16）Bundle-SymbolicName

标记 Bundle-SymbolicName 给出该 Bundle 在 OSGi 容器中的全局唯一标识符。与其他可选标记不同，这个标记没有默认值，并且是 Bundle 元数据信息之中唯一一个必须设置的标记。程序将基于此标记和版本号在 OSGi 容器中定位到一个独一无二的 Bundle。

当且仅当两个 Bundle 的 Bundle-SymbolicName 和 Bundle-Version 属性都相同的时候，它们才是完全相同的，不允许同时安装两个完全相同的 Bundle 到同一个 OSGi 容器之中。

Bundle-SymbolicName 有以下两个附加参数。

- ❑ singleton：表示 Bundle 是单例的。如果 OSGi 系统中同时存在两个 Bundle-SymbolicName 相同的（当然，要求 Bundle-Version 不相同，否则是不可能同时存在的）单例 Bundle，那么仅有其中一个会被解析。如果其中一个没有声明为单例 Bundle，则不会受到另外

一个单例 Bundle 的影响，默认值为 false。

- fragment-attachment：定义 Fragment Bundle 是否能附加到该 Bundle 之上。允许值为 always、never 和 resolve-time，含义为允许附加、禁止附加和只允许在解析过程中附加，默认值为 always，即允许附加。

示例：

```
Bundle-SymbolicName: com.acme.daffy
```

（17）Bundle-UpdateLocation

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

示例：

```
Bundle-UpdateLocation: http://www.acme.com/Firewall/bundle.jar
```

（18）Bundle-Vendor

标记 Bundle-Vendor 给出该 Bundle 的发行者信息。

示例：

```
Bundle-Vendor: OSGi Alliance
```

（19）Bundle-Version

标记 Bundle-Version 给出该 Bundle 的版本信息，默认值为“0.0.0”。注意，这项信息并不是仅供人工阅读的，“版本”在 OSGi 中是一项受系统管理的信息。维护一个 Bundle 的不同版本也是运行 OSGi 框架的重要特征之一，当一个 Bundle 依赖另一个 Bundle 时，经常需要指明它依赖的是什么版本范围内的 Bundle。

版本号是有序的，在 Symbolic-Name 相同的前提下，两个 Bundle 的版本可比较大小。完整的版本号会由“主版本号（Major）”+“副版本号（Minor）”+“微版本号（Micro）”+“限定字符串（Qualifier）”构成。

示例：

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

根据一般的开发习惯，上述 4 项版本号约定俗成地表示如下含义。

- 主版本号：表示与之前版本不兼容的重大功能升级。
- 副版本号：表示与之前版本兼容，但可能提供新的特性或接口。
- 微版本号：表示 API 接口没有变化，只是内部实现改变，或者修正了错误。
- 限定字符串：通常用于表示编译时间戳或者编译次数。

在比较版本大小时，从前往后逐项（含限定字符串）进行比较，当且仅当 4 个比较项都对应相等，两个 Bundle 的版本才相等，否则以第一个出现差异的版本号的大小决定整个

Bundle 版本的大小。

示例：

```
1.2.3 < 3.2.1 < 4.0
```

有一点必须注意，对于限定字符串的处理，OSGi 和 Maven 是恰恰相反的，在 Maven 里，版本“1.2.3.2012”<=“1.2.3”，但在 OSGi 里则是版本“1.2.3.2012”>=“1.2.3”。

（20）DynamicImport-Package

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

示例：

```
DynamicImport-Package: com.acme.plugin.*
```

（21）Export-Package

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

示例：

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

（22）Export-Service

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

示例：

```
Export-Service: org.osgi.service.log.LogService
```

（23）Fragment-Host

当该 Bundle 是一个 Fragment Bundle 时，标记 Fragment-Host 指明它的宿主 Bundle。

示例：

```
Fragment-Host: org.eclipse.swt; bundle-version="[3.0.0,4.0.0)"
```

（24）Import-Package

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

示例：

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

（25）Import-Service

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

示例：

```
Import-Service: org.osgi.service.log.LogService
```

（26）Provided-Capability

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

示例：

```
Provided-Capability: com.acme.dict; from=nl; to=de; version:Version=1.2
```

（27）Require-Capability

标记 Require-Capability 描述该 Bundle 所需要的服务特性。

示例：

```
Require-Capability: osgi.ee; filter:="(&(osgi.ee=AcmeMin)(version=1.1))"
```

（28）Require-Bundle

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

示例：

```
Require-Bundle: com.acme.chess
```

2.3.2　使用可视化工具

记录 Bundle 元数据的 MANIFEST.MF 文件是一个纯文本文件，这意味着手工编辑它是完全可行的。不过这样很繁琐且易于出错，在实际开发中一般不会手工去做，而通常会使用一些可视化的工具来配置 Bundle 的元数据信息。本节将介绍如何在 Eclipse IDE 下配置 Bundle 的元数据信息。

普通的 Java 工程是不存在 OSGi 元数据标记的，因此要先在 Eclipse 中建立一个“Plug-in Project”[⊖]，如图 2-2 所示。

⊖ 因为 Eclipse 本身是建立在 Equinox 平台之上的，所以每个 Eclipse 插件就是一个 OSGi 的 Bundle，因此在 Eclipse 的新建工程菜单中，把开发 Eclipse 插件与开发 OSGi Bundle 的菜单或视图合二为一了。

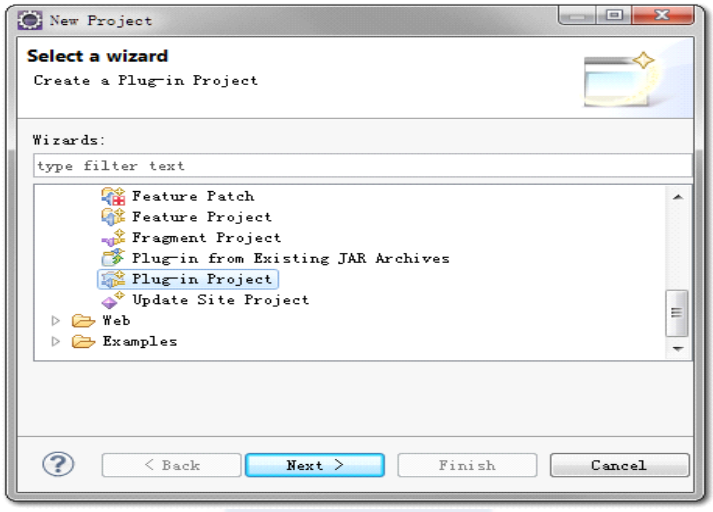

图 2-2 新建工程

在 New Project 中选中“Plug-in Project”，单击“Next”后出现如图 2-3 所示的界面。

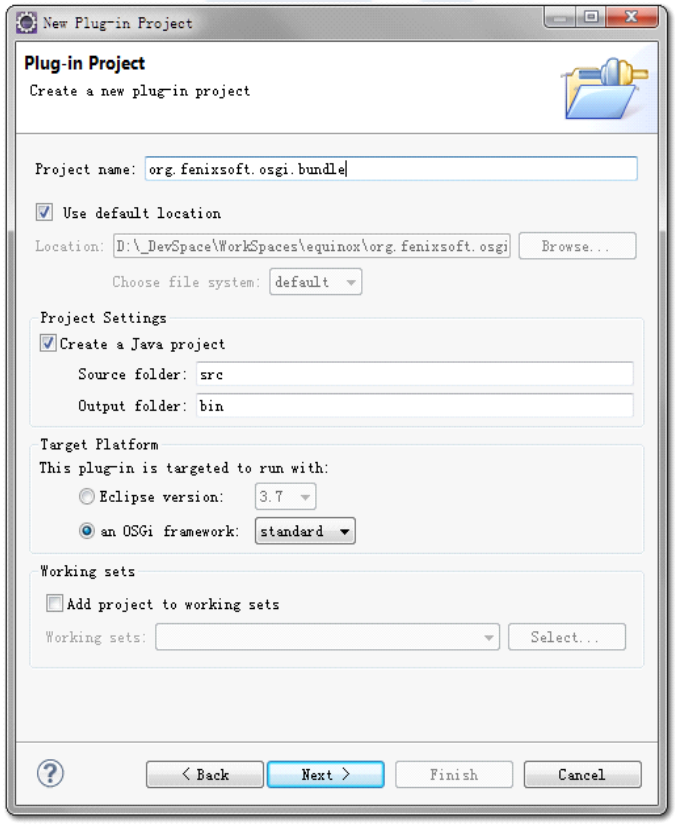

图 2-3 设置目标平台

从图 2-3 中可见，新建 Plug-in 工程的界面与新建普通 Java 工程界面的主要不同是增加了“Target Platform”项。如果要开发的是运行在 Eclipse 之上的插件，那么在此选择插件所支持的 Eclipse 版本号，系统会根据不同的 Eclipse 版本所支持的 API 为插件建立可用的目标平台。

如果准备开发一个标准的 OSGi Bundle，那么应该选择“an OSGi framework”，后面的下拉框有“standard”和“Equinox”两个选项。它们的区别是当选择 standard 时，只能使用 OSGi 规范所定义的标准 API，这样建立出来的 OSGi Bundle 在任何一个符合 OSGi 规范的实现中都能执行；而选择 Equinox 时，除了可以使用 OSGi 的标准 API 外，还可以使用 Equinox 框架自己的扩展功能，譬如专有 API、扩展点机制、Buddy 加载器和 P2 更新平台等，不过这样建立的 OSGi Bundle 只能运行于 Equinox 框架之上，无法在 Felix、Knopflerfish 等其他 OSGi 实现框架上运行。

就本书的例子来说，除了第三、四部分中专门介绍 Equinox 框架特有功能的例子外，其他各部分内容都可以选择“standard”，通过标准的 OSGi 框架 API 来完成。

继续单击“Next”按钮，将出现如图 2-4 所示的“Content”对话框。

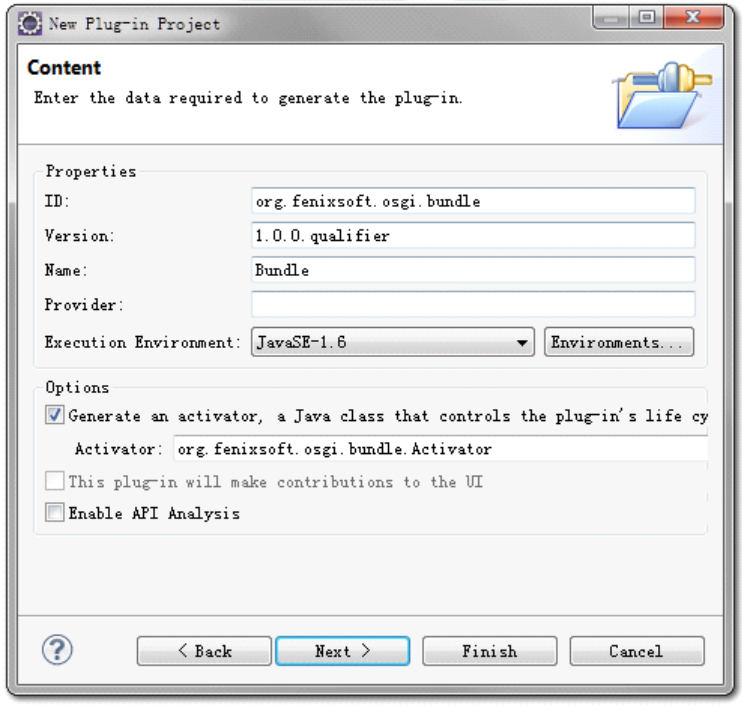

图 2-4　设置工程基本信息

在这个对话框中可以设置一些最基本的 Bundle 元数据信息，其中各输入项与前面元数据标记的对应如下：

- ID 对应于 Bundle-SymbolicName
- Version 对应于 Bundle-Version
- Name 对应于 Bundle-Name
- Provider 对应于 Bundle-Vendor
- Execution Environment 对应于 Bundle-RequiredExecutionEnvironment
- Activator 对应于 Bundle-Activator

在这里可以直接单击“Finish”按钮，之后出现如图 2-5 所示的 MANIFEST.MF 编辑界面，这个界面可以作为可视化工具对 Bundle 的元数据进行编辑。

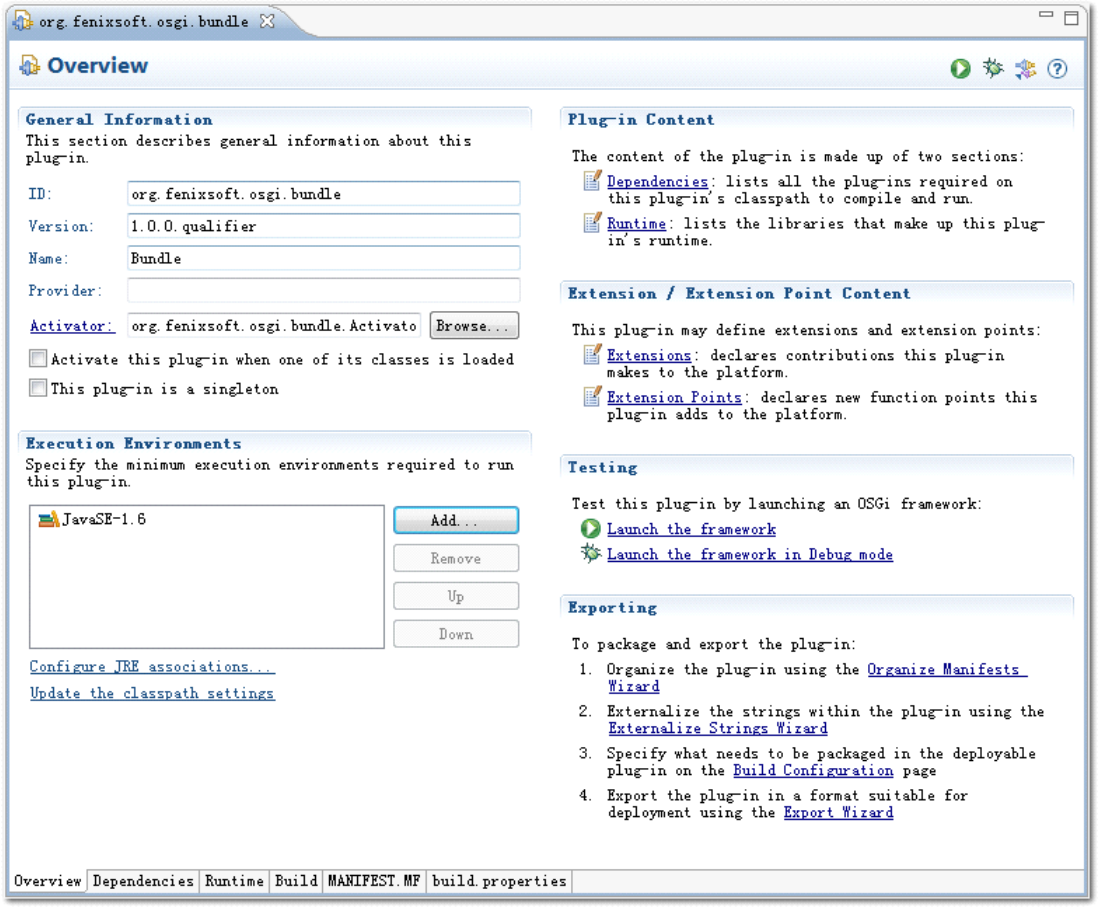

图 2-5 Bundle 元数据编辑界面

这个元数据编辑界面中有 Overview、Dependencies、Runtime、Build 四个面板，这四个面板之后的 MANIFEST.MF 和 build.properties 显示对应文件的文本内容编辑器。无论是使用面板进行可视化操作，还是直接用文本编辑的方式修改 MANIFEST.MF 文件的内容，这两种

修改方式都是等效的且内容会互相同步。

可以使用 Overview 面板来编辑 Bundle 的基础信息（在图 2-4 中设置的那些信息）及测试、打包 Bundle；使用 Dependencies 面板来配置 Bundle 的依赖关系（元数据中的 Import-Package 和 Require-Bundle 标记）；使用 Runtime 面板来配置 Bundle 的导出 Package 和该 Bundle 的 Classpath（元数据中的 Export-Package 和 Bundle-Classpath 标记）。如果在这两个面板中修改了 Bundle 的依赖关系，那么 Eclipse 会自动同步更新工程的“.classpath”文件，以便用户编码时能使用依赖包中提供的 API；还可以使用 Build 面板来配置打包发布工程时是否需要发布源码、说明文档和其他资源等细节信息。

在这个例子中，工具生成的元数据如下：

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle111
Bundle-SymbolicName: org.fenixsoft.osgi.bundle
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: org.fenixsoft.osgi.bundle.Activator
Import-Package: org.osgi.framework;version="1.3.0"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

2.4　Bundle 的组织与依赖

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

2.4.1　导出和导入 Package

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

```
Export-Package: org.osgi.service.io, org.osgi.service.web
Import-Package: org.osgi.service.io, org.osgi.service.web
```

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

1. 根据类名过滤

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

```
Export-Package: org.osgi.service.io;include="IO*"; exclude:="*Impl"
```

include 和 exclude 参数的具体使用方法如下：

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

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

```
Import-Package: org.osgi.service.io
```

2. 根据版本过滤

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

```
Bundle A（导出 Spring 2.0[⊜]）：
Export-Package: org.springframework.core; version="2.0.0"
Bundle B（导出 Spring 3.0）：
Export-Package: org.springframework.core; version="3.0.0"
```

对应的，导入时也要指明版本，准确地说应指明某个版本范围，例如：

```
Bundle C（导入 Spring 2.x 版）：
Import-Package: org.springframework.core; version="[2.0.0,2.9.9]"
Bundle D（导入 Spring 3.0 以上版本）：
Import-Package: org.springframework.core; version="3.0.0"
```

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

⊖ 这种解决方案是不提倡的，最佳的处理方案是把接口和实现分开到不同的包中。

⊜ 要导出一个可使用的 Spring，需要导出很多包，例子中仅使用“org.springframework.core”进行说明。

需要指定只导入 3.0.0 版的 Package，需要这样写：

```
Bundle E（只导入 Spring 3.0 版本）：
Import-Package: org.springframework.core; version="[3.0.0,3.0.0]"
```

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

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

3. 根据提供者过滤

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

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

```
Bundle A:
Bundle-SymbolicName: BundleA
Import-Package: com.acme.foo; bundle-symbolic-name="BundleB";
bundle-version="[1.4.1,2.0.0)"

Bundle B
Bundle-SymbolicName: BundleB
Bundle-Version: 1.4.1
Export-Package: com.acme.foo
```

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

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

- ❑ 附加参数 bundle-symbolic-name ：参数值为某个 Bundle 的名称，只有符合该名称的 Bundle 所导出的 Package 才会被导入。
- ❑ 附加参数 bundle-version ：参数值为导入 Bundle（注意不是 Package）的版本范围，

只有版本在该范围之内的 Bundle 所导出的 Package 才会被导入。

4. 根据属性过滤

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

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

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

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

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

Bundle C
Import-Package: org.osgi.simple;
```

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

```
Export-Package: org.osgi.simple; filter="true";mandatory:="filter"
```

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

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

Bundle B:
Import-Package: org.osgi.simple
```

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

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

5. 可选导入与动态导入

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

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

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

Bundle B:
Import-Package: org.osgi.simple

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

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

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

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

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

```
Bundle A:
Import-Package: org.osgi.simple

Bundle B:
DynamicImport-Package: org.osgi.simple
```

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

DynamicImport-Package 也可是使用 version 附加参数来过滤导入 Package 的版本，在导

入 Package 时声明属性，以便符合 Package 导出时 mandatory 参数中声明必须存在属性。

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

```
DynamicImport-Package: *
```

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

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

6. 导出 Package 的依赖限制

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

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

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

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

```
Bundle A:
Export-Package: org.osgi.service.http;uses:="javax.servlet.http"
Import-Package: javax.servlet.http; version="2.4"
```

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

```
Bundle B:
Import-Package: org.osgi.service.http  // 由 Bundle A 提供
,javax.servlet.http // 这里默认会导入 2.4 版的 Package
```

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

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

7. 导入整个 Bundle

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

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

```
Require-Bundle：BundleA、BundleB、BundleC⊖
```

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

```
Bundle A：
Require-Bundle: BundleB
Export-Package: p

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

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

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

与 Import-Package 类似，Require-Bundle 标记也有附加参数 bundle-version 和 resolution。bundle-version 用于过滤导入 Bundle 的版本，默认值为 [0.0.0, ∞)，即不限制版本范围。resolution 用于确定导入的 Bundle 是否是必需的，可选值为 mandatory 和 optional，默认值为

⊖ 根据一般的开发习惯，Bundle 的符号名称也会采用类似 Package 全限定名的命名方式，即类似 org.fenixsoft.osgi.bundle，而不是类似 BundleA 这样的命名（尽管这样的命名在技术上是没有任何问题的）。但在本章中，为了便于读者从命名上区分 Package 和 Bundle，Bundle 的命名没有遵循一般习惯。

mandatory，含义与 Import-Package 标记的 resolution 参数相同，这里就不再详细举例介绍了。

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

```
Bundle A
Require-Bundle: BundleB;visibility:=reexport

Bundle B
Export-Package: org.osgi.service.http
```

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

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

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

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

2.4.2 约束规则与示例

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

1. 图示

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

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

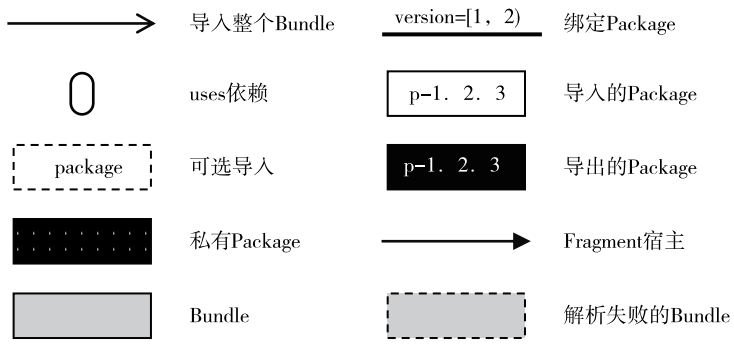

图 2-6　OSGi 标准图示与标记

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

Bundle B
Export-Package: p; version=1.5.1

Bundle C
Export-Package: r
```

这 3 个 Bundle 以及它们之间的依赖关系可以使用图 2-7 来表示。

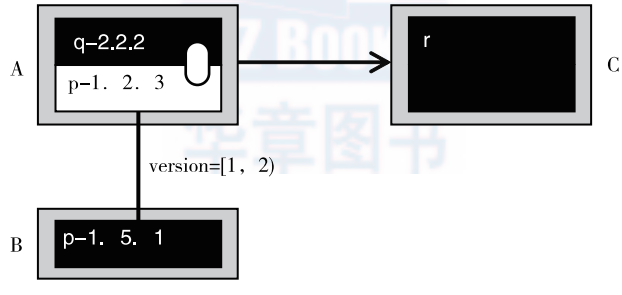

图 2-7　图示示例

2. 多版本 Package 依赖选择

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

```
Bundle A:
Import-Package: org.osgi.service.http
Import-Package: javax.servlet.http

Bundle B:
```

```
Export-Package: org.osgi.service.http;uses:="javax.servlet.http";
Import-Package: javax.servlet.http; version="2.4"

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

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

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

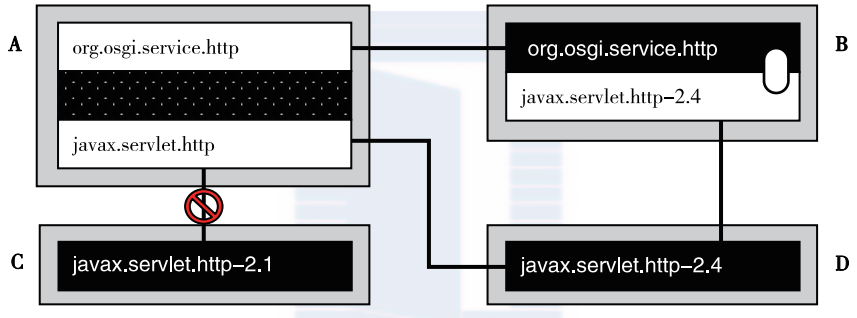

图 2-8 多版本 Package 依赖选择示例 (1)

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

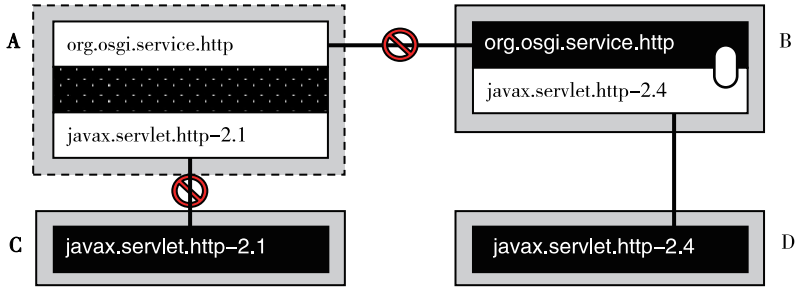

图 2-9 多版本 Package 依赖选择示例 (2)

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

```
org.osgi.framework.BundleException:
The bundle "BundleA_1.0.0.qualifier [8]" could not be resolved. Reason: Package
uses conflict: Import-Package: org.osgi.service.http; version="0.0.0"
```

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

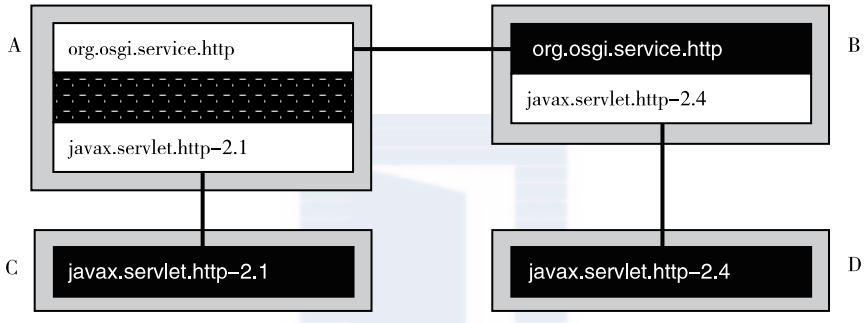

图 2-10　多版本 Package 依赖选择示例 (3)

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

3. 相同 Package 循环导出

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

```
Bundle A:
Require-Bundle: BundleB, BundleC
Export-Package: p

Bundle C:
Require-Bundle: BundleD
Export-Package: p
```

```
Bundle B
Export-Package: p

Bundle D:
Export-Package: p
```

相同 Package 循环导出的图形化描述如图 2-11 所示。

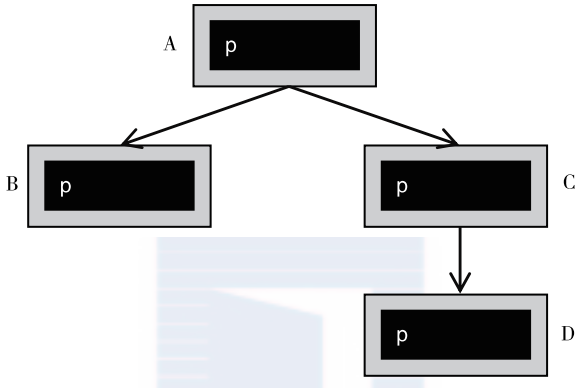

图 2-11 相同 Package 循环导出 (1)

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

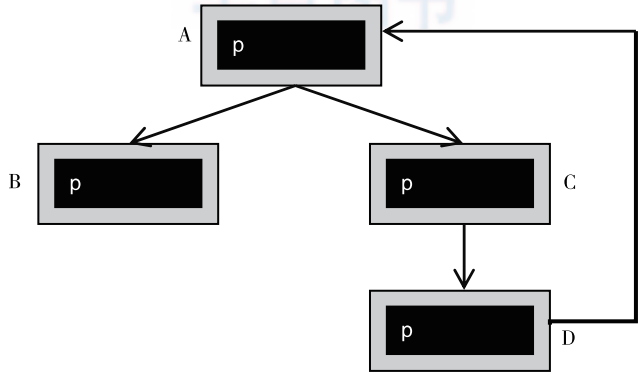

图 2-12 相同 Package 循环导出 (2)

⊖ 在 Bundle 查找类的顺序优先级中，Require-Bundle 高于自身 Classpath，因此 Bundle A 在最后，这一点我们将在稍后介绍 OSGi 类加载架构时介绍。

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

4.Require-Bundle 带来的问题

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

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

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

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

Bundle B:
Export-Package: p

Bundle C:
Export-Package: p
```

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

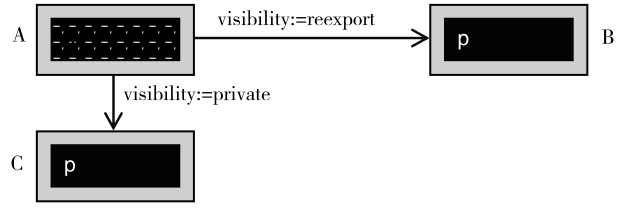

图 2-13 Require-Bundle 导致的 Package 导出问题

2.4.3 校验 Bundle 有效性

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

- ❑ 无法根据 Bundle-RequireExecutionEnvironment 的值找到一个可匹配执行环境。
- ❑ 在 Bundle 中没有设置 Bundle-SymbolicName 标记。
- ❑ 存在重复的标记。
- ❑ 对某个 Package 重复导入。
- ❑ Bundle 导入或导出了以 java.* 开头的 Package。
- ❑ 在 Export-Package 标记中用 mandatory 参数指明了某些强制属性，却没有定义这些属性。
- ❑ 尝试安装某个和 OSGi 框架中某个已经安装好的 Bundle 具有同样名称和版本的 Bundle。
- ❑ 将某个 Bundle 更新为和 OSGi 框架中一个已经安装好的 Bundle 具有同样名称和版本的 Bundle。
- ❑ 元数据信息中存在语法错误（例如，不合法的版本格式或 Bundle 名称）。
- ❑ 同时使用 Specification-version 和 version 参数，但是为它们指定了不一样的值，例如：

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

这将导致一个错误。而下面的指定：

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

不会导致错误。

- ❑ 在元数据信息中列出了权限文件：OSGI-INF/permission.perm，但此文件不存在。
- ❑ Bundle-ManifestVersion 的值不等于 1 或 2，在今后的 OSGi 规范中另有规定的情况除外。

2.5　OSGi 的类加载架构

OSGi 为 Java 平台提供了动态模块化的特性，但是它并没有对 Java 的底层实现如类库和 Java 虚拟机等进行修改，OSGi 实现的模块间引用与隔离、模块的动态启用与停用的关键在于它扩展的类加载架构。

OSGi 的类加载架构并未遵循 Java 所推荐的双亲委派模型（Parents Delegation Model），它的类加载器通过严谨定义的规则从 Bundle 的一个子集中加载类。除了 Fragment Bundle 外，每一个被正确解析的 Bundle 都有一个独立的类加载器支持，这些类加载器之间互相协作形成了一个类加载的代理网络架构，因此 OSGi 中采用的是网状的类加载架构，而不是 Java 传统的树状类加载架构，如图 2-14 所示。

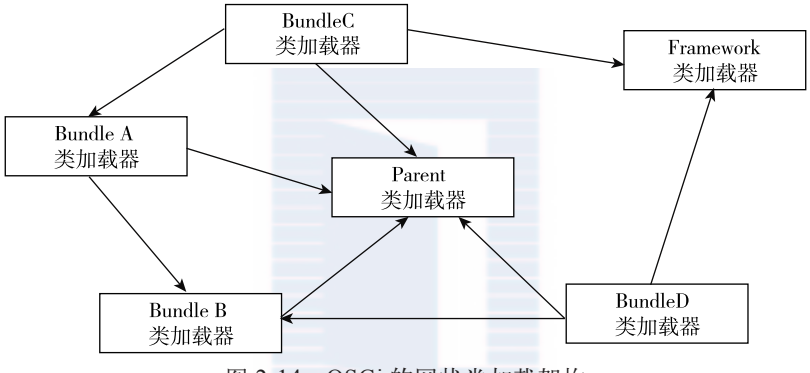

图 2-14　OSGi 的网状类加载架构

在 OSGi 中，类加载器可以划分为 3 类。

- 父类加载器：由 Java 平台直接提供，最典型的场景包括启动类加载器（Bootstrap ClassLoader）、扩展类加载器（Extension ClassLoader）和应用程序类加载器（Application ClassLoader）。在一些特殊场景中（如将 OSGi 内嵌入一个 Web 中间件）还会有更多的加载器组成。它们用于加载以“java.*”开头的类以及在父类委派清单中声明为要委派给父类加载器加载的类。
- Bundle 类加载器：每个 Bundle 都有自己独立的类加载器，用于加载本 Bundle 中的类和资源。当一个 Bundle 去请求加载另一个 Bundle 导出的 Package 中的类时，要把加载请求委派给导出类的那个 Bundle 的加载器处理，而无法自己去加载其他 Bundle 的类。
- 其他加载器：譬如线程上下文类加载器、框架类加载器等。它们并非 OSGi 规范中专门定义的，但是为了实现方便，在许多 OSGi 框架中都会使用。例如框架类加载器，OSGi 框架实现一般会将这个独立的框架类加载器用于加载框架实现的类和关键的服务接口类。

不同类加载器所能完成的（无论是自己完成加载，还是委派给其他类加载器来加载）加载请求的范围构成了该 Bundle 的类名称空间（Class Name Space）。在同一个类名称空间中，类必须是一致的，也就是说不会存在完全重名的两个类[㊀]。但是在整个 OSGi 的模块层，允许多个相同名称的类同时存在，因为 OSGi 模块层是由多个 Bundle 的类名称空间组成的。单独一个 Bundle 的类名称空间由如下内容组成：

- ❑ 父类加载器提供的类（以 java.* 开头的类以及在委派名单中列明的类）；
- ❑ 导入的 Package(Import-Package)；
- ❑ 导入的 Bundle（Require-Bundle）；
- ❑ 本 Bundle 的 Classpath（私有 Package，Bundle-Classpath）；
- ❑ 附加的 Fragment Bundle（fragment-attachment）；
- ❑ 动态导入的 Package（DynamicImport-Package）。

下面将介绍 Bundle 中各种类的加载过程，涉及类加载器[㊁]，以及类加载的优先级次序。

2.5.1 父类加载器

OSGi 框架必须将以 java.* 开头的 Package 交给父类加载器代理，这一点是无须设置且不可改动的。除此之外，OSGi 框架也允许用户通过系统参数“org.osgi.framework.bootdelegation”[㊂]自行指定一些 Package 委派给父类加载器加载，这个参数被称为“父类委派清单”（Boot Delegation List）。它的值应为一系列的包名，用逗号分隔，支持通配符，例如：

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

如果 org.osgi.framework.bootdelegation 的参数值如以上代码中所示，那么以 sun.* 和 com.sun.* 开头的类也会委派给父类加载器去加载。这个设定在特定场景下很有用。

例如某个部署在 Web 中间件上的 OSGi 应用需要使用 JDBC 访问数据库，与大多数应用一样，访问数据库的 Connection 是由应用服务器的 JNDI 提供的，这时候就应当把 JDBC 驱动设置为由父类加载器加载，而不是由 OSGi 中的某个 Bundle 包提供。因为 Web 中间件通常会带有连接池实现，为了实现事务控制和连接监视等功能，从 JNDI 中查到的 DataSource 是被中间件服务器包装过的，并非直接由原生的 JDBC 驱动所提供。为了保证中间件服务器

㊀ 准确地说，如果有重名的类，只有特定（由类加载器决定）的一个会被使用。比如在自己的 Bundle 包中建立一个名为 java.lang.Object 的类，可以创建和编译该类，但永远无法使用它。

㊁ 这一部分的主旨是介绍在 OSGi 规范下各个实现框架通用的知识，所以这里“类加载器”并没有特别指定具体实现和名称。另外，我们默认读者已对 Java 的类加载器和双亲委派模型有所了解，如果读者对此还比较陌生，建议先阅读本书附录 A。

㊂ 在实际开发的时候，一般不会使用“-D+JVM 参数”的形式去指定这些参数，各种 OSGi 实现都有自己的启动配置文件，如 Equinox 在 launch.ini 中可以配置，并在运行期将配置信息转换为 JVM 参数。

中一些需要把 Connection、Statement、ResultSet 等从接口转型为具体实现类的代码（大多数是操作大字段的代码）能正常执行，必须保证中间件服务器和 OSGi 应用所使用的 JDBC 驱动是同一个——不仅是同一个文件，还要是由同一个类加载器加载的，这样才能保证转型成功。

以 java.* 开头的 Package 是默认被隐式导出的，在所有 Bundle 中无需导入便可以直接使用，并且 OSGi 规范明确禁止在 Bundle 中导入或导出以 java.* 开头的 Package。与前面提到的父类委派清单类似，OSGi 也定义了添加隐式导出 Package 的参数“org.osgi.framework.system.packages”。这个参数使用标准的 Export-Package 语法描述，例如：

```
org.osgi.framework.system.packages=javax.crypto.interfaces
```

这里定义的 Package 将由系统 Bundle（ID 为 0 的 Bundle）导出，由父类加载器加载。这样导出的 Package 与普通的导出方式没有太大区别，可以带有属性和版本号，也可以使用 uses 参数描述依赖。

2.5.2　Bundle 类加载器

OSGi 框架为每一个 Bundle（不包括 Fragment Bundle）生成了一个 Bundle 类加载器的实例，这些类加载器负责处理其他 Bundle 委派的加载请求，根据元数据信息确定这些加载请求的类是否与该 Bundle 的导出列表相符合，然后对合法的加载请求进行响应，返回该 Bundle 的类供其他 Bundle 使用。

Bundle-Classpath 这个元数据标记与 Bundle 类加载器密切相关，它描述了 Bundle 加载器的 Classpath 范围，即 Bundle 加载器应该到哪里去查找类。

Bundle-Classpath 标记有默认值“.”，它代表该 Bundle 的根目录，或者说代表该 Bundle 的 JAR 文件。如果不在元数据信息中显式定义这个标记，那么 Bundle 类加载器就在整个 Bundle 的范围内查找类。但是要注意，在这种默认配置下，如果 Bundle 存在其他 JAR 文件，类加载器只能把它当作一个普通资源来读取，而无法查找到这些 JAR 文件内部包含的类。例如，在 Bundle 中有如下路径：

```
Bundle:
      lib/log4j.jar
      org/fenixsoft/osgi/Example.class

log4j.jar
      org/apache/log4j/Logger.class
```

Bundle 类加载器可以访问到 Example.class，但是无法访问到 Logger.class，最多只能把 log4j.jar 当作与图片、音频等类似的二进制资源整体提供出去。

要读取到 Logger.class，必须设置 Bundle-Classpath 标记为：

```
Bundle-Classpath: lib/log4j.jar,.
```

注意不要遗漏了后面的“,.”，这里有两个 Classpath 路径，它们之间使用逗号分隔，如果没有了后面的“.”，那么 Bundle 类加载器就只能处理 log4j.jar 中的类而无法处理本 Bundle 的 Example.class 了。

如果 Bundle-Classpath 标记的值是多个 Classpath 路径，那么它们之间还有优先级关系，例如下面这个定义：

```
Bundle-Classpath: required.jar,optional.jar,default.jar
```

该定义中 required.jar 是必须出现在 Bundle 中的类和资源；optional.jar 是某个可选的 JAR 包，其中存放着可选的类和资源；default.jar 中存放着 optional.jar 不可用时这些类和资源的默认值，如果 optional.jar 中有可用的内容便会对其覆盖。

如果一个 Bundle 被另一个 Fragment Bundle 附加，那么 Bundle-Classpath 也会相应叠加，例如下面定义：

```
Bundle A:
Bundle-Classpath: required.jar,optional.jar,default.jar

Bundle B:
Bundle-Classpath: fragment.jar
Fragment-Host: BundleA
```

此时 Bundle A 的 Bundle 类加载器能搜索到的 Classpath 依次为：required.jar、optional.jar、default.jar、fragment.jar。

Bundle 类加载器收到类加载请求时，会优先委托给导入包的其他 Bundle 类加载器处理，只有其他导入包的 Bundle 类加载器都无法处理时才会尝试自己处理。读者可以通俗地理解为“Import-Package”和“Require-Bundle”的优先级高于“Bundle-Classpath”，如果能在前者中找到所需的类，后者就不会起作用。这条规则读起来不复杂，但初接触 OSGi 的朋友在实际编码时候可能会对此有些不习惯，例如下面这个例子：

在 Bundle A、B 中都有 Package p，两者的 Package p 中都存在有类 ClassA。同时 ,Bundle B 还导入了 Bundle A 中的 Package p。在这个前提下，假设 Bundle A 中有下列代码：

```
ClassA anA = new ClassA();
```

这时候 ClassA 用的都是 Bundle A 中的类，符合一般思维习惯。但是如果 Bundle B 中有同样的代码，所使用的 ClassA 依然是 Bundle A 中的类，即使 Bundle B 自己的 Classpath 中也有这类 ClassA，甚至与调用 ClassA 的代码文件存在于同一个目录下紧紧相邻的就是 ClassA，都不会被使用，这就不符合一般的思维习惯了，如图 2-15 所示。

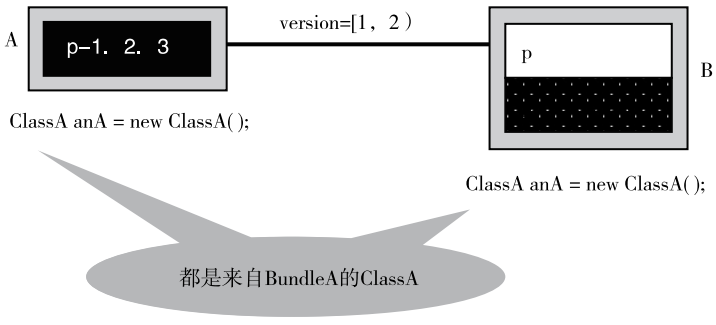

图 2-15　加载 Bundle 类加载器的示例

这里假设 Bundle A 导出的 p 中存在 ClassA 这个类，这样 Bundle B 的 ClassA 就无法派上用场。如果情况更极端一些，Bundle A 导出的 p 不存在 ClassA 这个类，那 Bundle B 的 ClassA 依然不会被使用，而会直接收到 ClassNotFoundException 异常，异常信息类似如下所示：

```
Caused by: java.lang.ClassNotFoundException: p.ClassA
    at org.eclipse.osgi.internal.loader.BundleLoader.findClassInternal(BundleLoader.
java:467)
    at org.eclipse.osgi.internal.loader.BundleLoader.findClass(BundleLoader.java:429)
    at org.eclipse.osgi.internal.loader.BundleLoader.findClass(BundleLoader.java:417)
    at org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader.loadClass(DefaultClass-
Loader.java:107)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:248)
```

对于在 Bundle 中发生的加载请求而言，当前 Bundle 的 Bundle 类加载器是使用到的类的初始类加载器[⊖]（Initiating Classloader，它表示加载请求最先发送到的类加载器），而哪个类加载器是定义类加载器（Defining Classloader，它表示加载请求被不断委派后，最终执行加载动作的类加载器）则要根据 OSGi 类加载顺序来判定。在类型强制转换和类型比较（譬如 instanceOf 操作）时理解类加载顺序很重要，因为即使是同一个类文件，由不同定义类加载器加载所形成的类在 Java 虚拟机中也是完全独立且不可互相转型的。

2.5.3　其他类加载器

在 OSGi 中还可能使用到其他的类加载器，比如 OSGi 实现框架中一般都会有框架类加载器（Framework Classloader）。OSGi 框架为每个 Bundle 创建 Bundle 类加载器的实例，而 OSGi 框架自身的代码——至少涉及 OSGi 框架启动的代码就没法使用 Bundle 类加载器来加载，因此需要一个专门的框架类加载器来完成这个任务。这个框架类加载器是各个 OSGi 实

⊖ “初始类加载器”和“定义类加载器”的概念在《Java 虚拟机规范》之中有定义，它们分别指“触发加载某个类”的类加载器和“真正加载某个类”的类加载器。引导类加载器和定义类加载器可能是同一个，但更多情况下是由引导类加载器委派给定义类加载器去加载某个类。

现框架自己定义的，有时候可能直接使用 Java 平台提供的应用程序类加载器（Application ClassLoader）。这个框架类加载器还可能同时充当父类加载器的角色，比如在 Equinox 框架中就可以选择是使用启动类加载器、扩展类加载器、应用程序类加载器还是使用框架类加载器来作为父类加载器。

另外一个在 OSGi 中比较常见的类加载器是线程上下文类加载器（Thread ContextClassLoaser），这个类加载器并不是在 OSGi 中才出现的，它在普通的 Java 应用中有广泛应用。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoaser() 方法进行设置，如果创建线程时未设置，那么它将会从父线程中继承一个；如果在应用程序的全局范围内都没有设置过，那么这个类加载器就默认是应用程序类加载器。有了线程上下文类加载器，就可以做一些“舞弊”的事情，例如直接加载没有经过导入和导出的类，或者让由框架类加载器加载的 OSGi 框架代码在运行期得以访问一些系统 Bundle 中的类。

OSGi 中其他的类加载器与具体实现密切相关，后面我们将会在确定具体 OSGi 实现框架和具体上下文的场景下再进行介绍，此处不再赘述。

2.5.4 类加载顺序

当一个 Bundle 类加载器遇到需要加载某个类或查找某个资源的请求时，搜索过程必须按以下指定步骤执行：

1）如果类或资源在以 java.* 开头的 Package 中，那么这个请求需要委派给父类加载器；否则，继续下一个步骤搜索。如果将这个请求委派给父类加载器后发现类或资源不存在，那么搜索终止并宣告这次类加载请求失败。

2）如果类或资源在父类委派清单 (org.osgi.framework. bootdelegation) 所列明的 Package 中，那么这个请求也将委派给父类加载器。如果将这个请求委派给父类加载器后，发现类或资源不存在，那么搜索将跳转到一个步骤。

3）如果类或资源在 Import-Package 标记描述的 Package 中，那么请求将委派给导出这个包的 Bundle 的类加载器，否则搜索过程将跳转到下一个步骤。如果将这个请求委派给 Bundle 类加载器后，发现类或资源不存在，那么搜索终止并宣告这次类加载请求失败。

4）如果类或资源在 Require-Bundle 导入的一个或多个 Bundle 的包中，这个请求将按照 Require-Bundle 指定的 Bundle 清单顺序逐一委派给对应 Bundle 的类加载器，由于被委派的加载器也会按照这里描述的搜索过程查找类，因此整个搜索过程就构成了深度优先的搜索策略。如果所有被委派的 Bundle 类加载器都没有找到类或资源，那么搜索将转到下一个步骤。

5）搜索 Bundle 内部的 Classpath。如果类或资源没有找到，那么这个搜索将转到下一个步骤。

6）搜索每个附加的 Fragment Bundle 的 Classpath。搜索顺序将按这些 Fragment Bundle 的 ID 升序搜索。如果这个类或资源没有找到，那么搜索转到下一个步骤。

7）如果类或资源在某个 Bundle 已声明导出的 Package 中，或者包含在已声明导入（Import-Package 或 Require-Bundle）的 Package 中，那么这次搜索过程将以没有找到指定的类或资源而终止。

8）如果类或资源在某个使用 DynamicImport-Package 声明导入的 Package 中，那么将尝试在运行时动态导入这个 Package。如果在某个导出该 Package 的 Bundle 中找到需要加载的类，那么后面的类加载过程将按照步骤 3）处理。

9）如果可以确定找到一个合适的完成动态导入的 Bundle，那么这个请求将委派给该 Bundle 的类加载器。如果无法找到任何合适的 Bundle 来完成动态导入，那么搜索终止并宣告此次类加载请求失败。当将动态导入委派给另一个 Bundle 类加载器时，类加载请求将按照步骤 3）处理。

上述加载过程如图 2-16 所示。

2.6　定义执行环境

某些 Bundle 必须在特定的执行环境之下才能正常运作，例如为大型服务端应用而设计的 Bundle 一般不能运行在嵌入式设备之中，而利用 JDK1.6 开发的 Bundle 无法运行在 JDK1.5 的应用服务器之中等。为了确保 Bundle 可用性，元数据信息中提供了 Bundle-RequiredExecutionEnvironment 标记来描述 Bundle 对执行环境的要求，示例如下：

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

OSGi 对执行环境定义的命名是直接继承于 Java 平台的执行环境名称，如图 2-17 所示为 Eclipse 配置界面中执行环境与 JRE 关联的设置对话框。

对于每一个执行环境，在 OSGi 框架中应该对应一套系统运行参数的默认配置。前面提到过的 org.osgi.framework.bootdelegation、org.osgi.framework.system.packages 等参数的默认值都由执行环境决定。以 Equinox 框架为例，对于 JavaSE-1.6 这个执行环境，在 Equinox 框架中对应的配置存储在 JavaSE-1.6.profile 文件中，内容为：

```
org.osgi.framework.system.packages = \
 javax.accessibility,\
 javax.activation,\
 javax.activity,\
 javax.annotation,\
……// 版面关系，省略其余的 Package
org.osgi.framework.bootdelegation = \
 javax.*,\
 org.ietf.jgss,\
 org.omg.*,\
 org.w3c.*,\
 org.xml.*,\
 sun.*,\
```

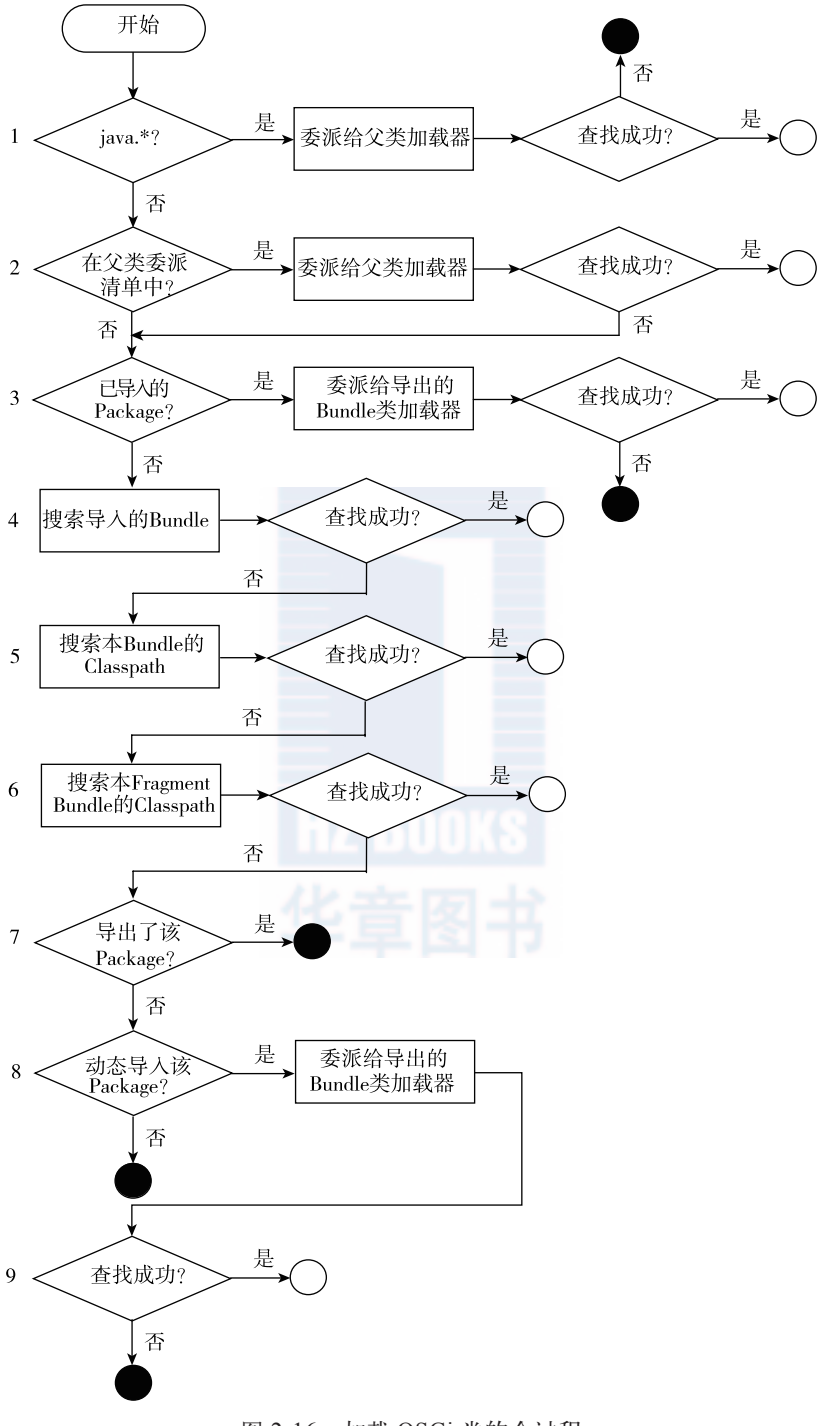

图 2-16 加载 OSGi 类的全过程

```
 com.sun.*
org.osgi.framework.executionenvironment = \
 OSGi/Minimum-1.0,\
 OSGi/Minimum-1.1,\
 OSGi/Minimum-1.2,\
 JRE-1.1,\
 J2SE-1.2,\
 J2SE-1.3,\
 J2SE-1.4,\
 J2SE-1.5,\
 JavaSE-1.6
org.osgi.framework.system.capabilities = \
 osgi.ee; osgi.ee="OSGi/Minimum"; version:List<Version>="1.0, 1.1, 1.2",\
 osgi.ee; osgi.ee="JavaSE"; version:List<Version>="1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6"
osgi.java.profile.name = JavaSE-1.6
org.eclipse.jdt.core.compiler.compliance=1.6
org.eclipse.jdt.core.compiler.source=1.6
org.eclipse.jdt.core.compiler.codegen.inlineJsrBytecode=enabled
org.eclipse.jdt.core.compiler.codegen.targetPlatform=1.6
org.eclipse.jdt.core.compiler.problem.assertIdentifier=error
org.eclipse.jdt.core.compiler.problem.enumIdentifier=error
```

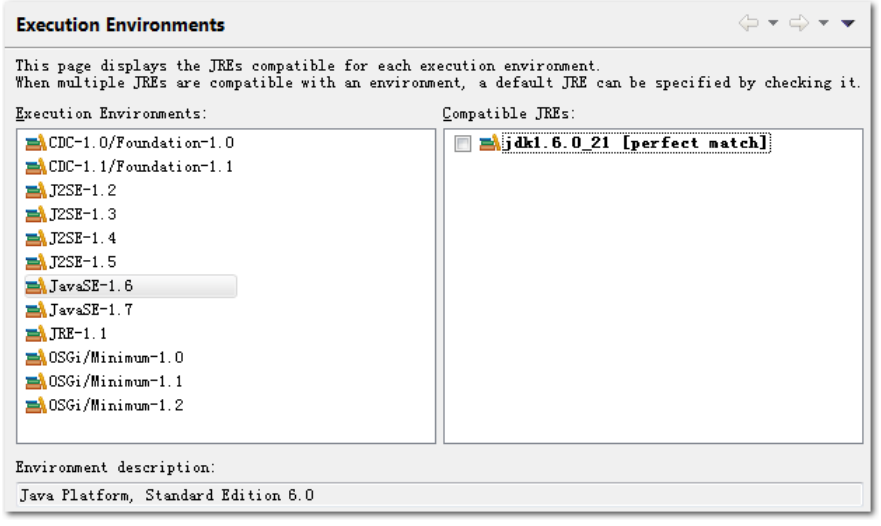

图 2-17　Eclipse 执行环境设置

在 OSGi R4.3 规范发布后，元数据对执行环境的描述能力被进一步增量，引入了通用 Capability 的概念，通过 Require-Capability 和 Provide-Capability 两个新的标记也可以定义执行环境，例如：

```
#R4.3 之前的方式：
Bundle-RequiredExecutionEnvironment: JavaSE-1.6

#R4.3 新提供的方式：
```

```
Require-Capability: osgi.ee;filter:="(&(osgi.ee="JavaSE")(version>=1.6))"
```

定义了 Require-Capability 之后，OSGi 框架在解析 Bundle 之前必须满足必要的 Capability 需求。典型使用场景是提供 OSGi 的声明式服务，该服务并不会表示为 Package 依赖，但为了能够正确解析 Bundle，它又是必需的。

2.7 本地化

前面提到过，Bundle 的元数据信息中包含一些供人工阅读的信息，如 Bundle-Name、Bundle-Vendor 等。这些信息可能需要根据用户的语言、国家和其他指定的参数（一般是指定“区域”的参数）翻译成不同的语言。OSGi 规范定义了 Bundle 应该如何自动根据系统语言、国家等参数自动翻译这些信息，即 Bundle 的本地化能力。

Bundle 的本地化信息必须遵循特定的命名规则，存放在 Bundle 的指定目录下，如果没有通过 Bundle-Localization 特别指定，那么这个目录默认为“OSGI-INFO/l10n”。为了方便实现框架查找存在的本地化信息，OSGi 规范规定了这些信息必是以“bundle”开头，以语言、国家、其他参数为内容，以下划线（‘_’,\u005F）分隔，以“.properties”为扩展名来命名的文本文件，即遵循以下格式命名：

```
OSGI-INF/l10n/bundle_[语言]_[国家]_[其他参数].properties
```

文件名中所使用到的语言、国家等会从 java.util.Locale 获取。例如，以下文件提供了英语、荷兰语（比利时和荷兰）和瑞典语的本地化信息：

```
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
```

实现框架不是通过精确匹配文件名来搜索本地化信息，而是采用一种渐进式的搜索方式。如果最佳匹配的文件没有找到，会先删除参数，然后是国家，最后是语言，直到找到一个包含有效信息的本地化文件。比如，参数为 welsh、国家是 GB、语言是 en 的本地化文件将通过以下顺序查找：

```
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
```

另外，查找过程也并不是找到某个可用的本地化文件就停止，而是会一直进行下去，这种策略允许在拥有更具体区域、语言信息的本地化文件时覆盖更少信息的本地化文件。

本地化文件中包含了以 Key-Value 值对表示的本地化信息。Bundle 的元数据信息文件中所有信息都可以进行本地化。但是，对于带有程序语义而非人工阅读的信息，OSGi 实现框

架必须使用非本地化版本，也就是只以 MANIFEST.MF 文件中的内容为准。

我们可以通过两种方式来使用本地化文件中的信息，第一种是直接覆盖元数据标记，例如：

```
Bundle-Name : The ACME Bundle
Bundle-Vendor : The ACME Corporation
Bundle-Description : The ACME Bundle provides all of the ACME
```

第二种是在 MANIFEST.MF 中使用本地化变量，然后在本地化文件中定义这些变量的值，例如：

```
# 在 MANIFEST.MF 文件中：
Bundle-Name : %acme_bundle
Bundle-Vendor : %acme_corporation
Bundle-Description : %acme_description
Acme-Defined-Header : %acme_special_header

# 在 OSGI-INF/l10n/bundle.properties 文件中：
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
```

本地化文件中定义的变量中间允许空格存在，把上面例子中的“_”替换成空格也是允许的。另外，在 MANIFEST.MF 文件中由用户自定义的非 OSGi 的标记也可以被本地化，例如上面的“Acme-Defined-Header”。

2.8　本章小结

OSGi 的模块层定义了一个模块化的 Java 模型，针对 Java 部署模式的一些缺点进行了改进，对哪些 Package 可以在模块之间交互、如何交互、版本管理等都有严格规定。

在 OSGi 中，模块层独立于生命周期层和服务层，这意味着它在使用时可以不需要生命周期层和服务层的支持，但是，这样的模块是“静态的”。生命周期层提供了对模块层的 Bundle 进行管理的各种 API，而服务层提供了 Bundle 之间的通信模型。在后面我们将继续探索模块层与生命周期层中的知识。

第 3 章　生命周期层规范与原理

OSGi 规范把模块化“静态”的一面，比如如何描述元数据、如何加载模块中的类和资源等内容定义于模块层规范之中；而把模块化“动态”的一面，比如模块从安装到解析、启动、停止、更新、卸载的过程，以及在这些过程中的事件监听和上下文支持环境等定义于生命周期层（Life Cycle Layer）之中。在第 2 章介绍了模块层相关知识后，这里再从 OSGi 运行时的角度去看一下模块化“动态”的相关知识。

因为生命周期层本身具有“动态”的特性，所以本章介绍的大多数例子的演示都要求 OSGi 框架真正运行起来才能进行。笔者演示例子采用的 OSGi 框架是 Equinox 3.8（用于支撑 Eclipse 4.2 的版本），后面不再单独说明。如果读者想了解如何部署运行此框架，可参考本书第 5 章的相关内容。

3.1　Bundle 标识

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

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

一刻它就被确定下来，不会因使用了不同的 OSGi 框架而有所不同（前面的 Bundle ID 和 Bundle Location 是由 OSGi 框架所决定的）。Bundle 的版本与符号名称一起可以唯一定位一个 Bundle，在代码中可以通过 Bundle 接口的 getSymbolicName() 方法获取当前 Bundle 的符号名称，通过 getVersion() 方法获取 Bundle 的版本号。

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

```
System.out.println("Location:" + bundleContext.getBundle().getLocation());
System.out.println("ID:" + bundleContext.getBundle().getBundleId());
System.out.println("SymbolicName:" + bundleContext.getBundle().getSymbolicName());
```

输出结果为：

```
Location: initial@reference:file:../WorkSpaces/equinox/BundleA/
ID: 1
SymbolicName: BundleA
```

3.2　Bundle 状态及转换

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

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

个方法退出之后，Bundle 的状态都将转为 RESOLVED。STOPPING 的状态值为整型数 16。

- ACTIVE，Bundle 处于激活状态，说明 BundleActivator 的 start() 方法已经执行完毕，如果没有其他动作，Bundle 将继续维持 ACTIVE 状态。ACTIVE 的状态值为整型数 32。

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

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

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

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

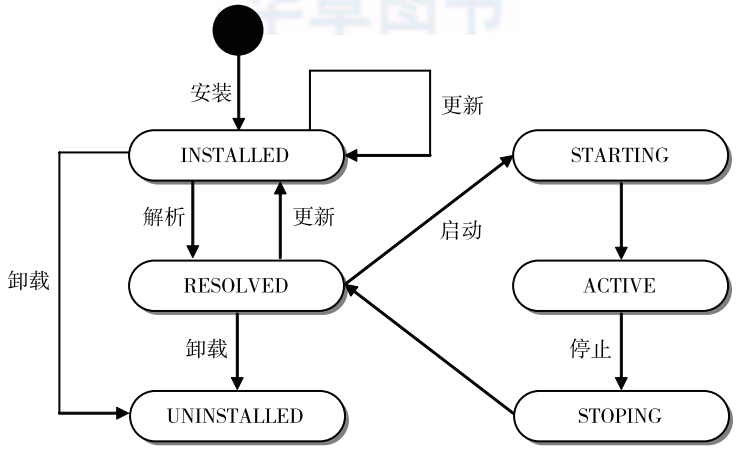

图 3-1 Bundle 状态转换示意图

3.2.1　安装过程

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

1）根据 Bundle Location 判定 Bundle 是否已经安装过，如果已安装，那么直接返回之前安装的 Bundle 实例。

```
// 代码位置：Framework.installWorker()
AbstractBundle bundle = getBundleByLocation(location);
// 如果 Bundle 已安装，直接返回之前安装的 Bundle 实例
if (bundle != null) {
    Bundle visible = origin.getBundle(bundle.getBundleId());
    if (visible == null) {
        BundleData data = bundle.getBundleData();
        String msg = NLS.bind(Msg.BUNDLE_INSTALL_SAME_UNIQUEID, new Object[] {data.
getSymbolicName(), data.getVersion().toString(), data.getLocation()});
        throw new BundleException(msg, BundleException.REJECTED_BY_HOOK);
    }
    return bundle;
}
```

2）根据 Bundle Location 生成 BundleData、BundleInstall 等访问对象，这时需要完成下面几个关键的动作：

- ❑ 产生新的 Bundle ID；
- ❑ 读取 MANIFEST.MF 文件中的内容，并且对其进行有效性校验（见 2.4.3 节），但不对内容进行具体的解析。
- ❑ 更新 Bundle 的 LastModified 等信息（在 Bundle 更新检测中会使用到）。

```
// 代码位置：BaseStorage.installBundle()
// 产生新的 BundleID
BaseData data = createBaseData(getNextBundleId(), location);
return new BundleInstall(data, source, this);
  ……
// 代码位置：BundleInstall.begin()
// 更新 Bundle 的 LastModified 等信息
data.setLastModified(System.currentTimeMillis());
data.setStartLevel(storage.getInitialBundleStartLevel());
  ……
// 代码位置：BundleInstall.begin()
// 读取 MANIFEST.MF 文件中的内容
Dictionary<String, String> manifest = storage.loadManifest(data, true);
```

3）检查将要安装的 Bundle 是否与系统中已有的 Bundle 重名，Bundle 的符号名称和版本确定其唯一标识，OSGi 不允许将两个符号名称和版本完全一样的 Bundle 安装到系统中。

```
// 代码位置：Framework.createAndVerifyBundle()
// 检查将要安装的 Bundle 是否设置了 Bundle-SymbolicName，
// 以及符号名称和版本是否在系统中出现过
if (!allowDuplicateBSNVersion && bundledata.getSymbolicName() != null) {
      AbstractBundle installedBundle = getBundleBySymbolicName(bundledata.
getSymbolicName(), bundledata.getVersion());
     if (installedBundle != null && installedBundle.getBundleId() != bundledata.
getBundleID()) {
          String msg = NLS.bind(Msg.BUNDLE_INSTALL_SAME_UNIQUEID, new Object[]
{installedBundle.getSymbolicName(), installedBundle.getVersion().toString(),
installedBundle.getLocation()});
          throw new DuplicateBundleException(msg, installedBundle);
     }
}
```

4）根据 Bundle 类型（Fragment、Bundle Host 等）确定 Bundle 对象的实现类，初始化 Bundle 对象实例。

```
// 代码位置：Framework.createBundle()
protected static AbstractBundle createBundle(BundleData bundledata, Framework
framework, boolean setBundle) throws BundleException {
     AbstractBundle result;
     if ((bundledata.getType() & BundleData.TYPE_FRAGMENT) > 0)
         result = new BundleFragment(bundledata, framework);
     else if ((bundledata.getType() & BundleData.TYPE_COMPOSITEBUNDLE) > 0)
         result = new CompositeImpl(bundledata, framework);
     else if ((bundledata.getType() & BundleData.TYPE_SURROGATEBUNDLE) > 0)
         result = new SurrogateImpl(bundledata, framework);
     else
         result = new BundleHost(bundledata, framework);
     if (setBundle)
         bundledata.setBundle(result);
     return result;
}
```

5）将 Bundle 实例加入到 OSGi 框架的 Bundle Repository 之中，在这个过程结束之后，新安装的 Bundle 在 Equinox Console 和其他 Bundle 中就已经可见了。

```
// 代码位置：Framework.installWorkerPrivileged()
bundles.add(bundle);
storage.commit(false);
```

6）发布 Bundle 的 INSTALLED 状态转换事件。

```
// 代码位置：Framework.installWorker()
publishBundleEvent(new BundleEvent(BundleEvent.INSTALLED, bundle, origin.
getBundle()));
```

虽然 Bundle 安装需要经过以上 6 个步骤，但是从 Equinox Console 或其他 Bundle 的角度观察，一个 Bundle 的安装过程是个原子过程，即要么 Bundle 已经安装了，要么 Bundle 还没有安装，不会观察到 Bundle“正在安装之中”的状态，也不会出现 Bundle 安装了一半的情况。并且，Bundle 的 INSTALLED 状态是一个持久状态，如果没有外部作用（改变启动级别、调用 start() 方法或卸载 Bundle），Bundle 将一直维持这个状态。

3.2.2　解析过程

解析过程是 OSGi 框架根据 Bundle 的 MANIFEST.MF 文件中描述的元数据信息分析处理 Bundle 依赖关系的过程。在实际开发中，经常会遇到“某个 Bundle 安装不上”这类问题，这种“安装不上”在大多数情况下不是指“安装过程”失败，而更多是指解析过程中的处理失败，抛出了异常。接下来我们了解一下 OSGi 框架是如何实现解析过程的。

1）先把 OSGi 框架中所有已安装的 Bundle（包括未解析的）按照以下规则进行排序，以便在某个 Package 有多个导出源可供选择时确定依赖包的优先级关系（下面的 Package 优先级依次递减）。

- ❑ 已经解析的 Bundle 导出的 Package 优先于未解析的 Bundle 导出的 Package。
- ❑ 具有更高版本的 Package 优先于低版本的 Package。
- ❑ 具有较低 Bundle ID 的 Bundle 导出的 Package 优先于较高 Bundle ID 的 Bundle 导出的 Package。

```
// 代码位置：ResolverImpl.resolve()
resolverExports.reorder();
resolverBundles.reorder();
reorderGenerics();
```

2）对将要解析的 Bundle 执行一些基本的检查校验，确定它们是否符合被解析的基本条件，包括以下检查项：

- ❑ 检查 OSGi 容器提供的执行环境是否能满足 Bundle 的需要；
- ❑ 检查当前系统是否能满足在 Bundle 中定义的本地代码（Native Code）的需求（例如 Bundle 中的本地代码是以 *.so 形式发布的就只能用于 Linux 系统，是基于 x86 指令集的就无法用于 ARM 架构的机器等）。

```
// 代码位置：ResolverImpl.resolveBundles()
for (ResolverBundle bundle : bundles) {
    state.removeResolverErrors(bundle.getBundleDescription());
    bundle.setResolvable(isResolvable(bundle, platformProperties, hookDisabled) ||
developmentMode);
}
```

3）将 Fragment Bundle 附加到宿主 Bundle 之中，在对任何一个 Bundle 解析之前，要保证框架中所有的 Fragment Bundle 都已经正确附加到宿主上。

```
// 代码位置：ResolverImpl.resolveBundles0()
Collection<String> processedFragments = new HashSet<String>(bundles.length);
for (int i = 0; i < bundles.length; i++)
   attachFragment(bundles[i], processedFragments);
```

4）确保 OSGi 容器能提供所有 Bundle 元数据信息中声明的依赖项，如 Require-Capability、Import-Package 和 Require-Bundle 中声明的依赖内容，并且这些依赖项都要符合要求，具体要求如下：

- ❑ Bundle 声明导入的 Package 在容器中能够匹配到符合版本范围要求的提供者；
- ❑ Bundle 声明导入的 Package 包含了提供者在导出时所有声明要求包含的强制属性；
- ❑ 如果 Bundle 声明导入的 Package 定义了属性，那么这些属性的值必须与导出时声明的属性值相匹配。
- ❑ 如果某个 Package 的导入者与提供者都依赖于同一个 Package，那么导入者必须满足导出 Package 时使用 uses 参数声明的约束。

  Equinox 对上述检查采用“Fast-Fail”方式实现，即检查到异常，校验程序立即中断并抛出异常，这些代码实现在 ResolverImpl.resolveBundle() 中。由于代码量较大，考虑版面关系，不再贴出具体代码，读者可自行阅读源码。

5）将 Bundle 的状态调整为已解析的状态。这里不仅要修改 Bundle 对象的 getState() 返回值，更重要的是要将前面计算出来的 Bundle 各个依赖项的提供者记录在 Bundle 对象实例上，并且把 Bundle 所能提供的导出包和 Capabilities 记录到 OSGi 容器之中。

这部分的实现主要在 ResolverImpl.stateResolveBundle() 之中，代码较多，考虑版面关系，读者可自行阅读源码。

3.2.3 启动过程

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

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

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

3.2.4　更新过程

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

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

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

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

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

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

下面对 Equinox 框架在更新 Bundle 时的代码进行分析。

1）检查 Bundle 目前是否处于可更新的状态，还没有安装、状态为 UNINSTALLED 的 Bundle 是不能被更新的。

```
// 代码位置：AbstractBundle.checkValid()
if (state == UNINSTALLED) {
      throw new IllegalStateException(NLS.bind(Msg.BUNDLE_UNINSTALLED_EXCEPTION,
getBundleData().getLocation()));
}
```

2）确定 Bundle 更新的数据来源。这个来源可以是用户指定的一个输入流，如果用户没有指定，默认通过 Bundle Location 创建一个输入流。

```
// 代码位置 :AbstractBundle.update()
if (in == null) {
      String updateLocation = bundledata.getManifest().get(Constants.BUNDLE_
```

```
UPDATELOCATION);
        if (updateLocation == null)
            updateLocation = bundledata.getLocation();
        if (Debug.DEBUG_GENERAL)
            Debug.println("   from location: " + updateLocation); //$NON-NLS-1$
        // 从 Bundle 原来的位置进行更新
        source = framework.adaptor.mapLocationToURLConnection(updateLocation);
    } else {
        // 通过一个指定的输入流获取 Bundle 的新内容进行更新
        source = new BundleSource(in);
    }
```

3）根据输入流创建 Bundle 对象实例，实例创建的过程与安装 Bundle 时是一样的。

```
    // 代码位置 :AbstractBundle.updateWorkerPrivileged()
    final AbstractBundle newBundle = framework.createAndVerifyBundle(newBundleData,
false);
```

4）将新 Bundle 对象实例的信息刷新到调用 update() 方法的 Bundle 对象中，并更新 OSGi 框架的 Bundle Repository。

```
    // 代码位置：AbstractBundle.updateWorkerPrivileged()
    synchronized (bundles) {
        String oldBSN = this.getSymbolicName();
        exporting = reload(newBundle);
        bundles.update(oldBSN, this);
        manifestLocalization = null;
    }
```

5）将新的 Bundle 对象的状态转换回更新之前的状态。因为刚刚更新完的 Bundle 是处于 INSTALLED 状态的，如果更新前是 ACTIVE、STARTING、STOPING 等状态的，需要自动切换回原来的状态。这时执行的逻辑跟对应的状态变换过程是一致的。

```
    // 代码位置：AbstractBundle.updateWorker()
    if ((previousState & (ACTIVE | STARTING)) != 0) {
        try {
             startWorker(START_TRANSIENT | ((previousState & STARTING) != 0 ? START_
ACTIVATION_POLICY : 0));
        } catch (BundleException e) {
            framework.publishFrameworkEvent(FrameworkEvent.ERROR, this, e);
        }
    }
```

6）发布 Bundle 的 UPDATED 状态转换事件。

```
    // 代码位置：AbstractBundle.updateWorker()
    framework.publishBundleEvent(BundleEvent.UPDATED, this);
```

3.2.5 停止过程

Bundle 的停止过程是启动过程的逆向转换，此时 OSGi 框架会自动调用 Bundle 的

Activator 类的 stop() 方法。在 stop() 方法执行期间，Bundle 的状态为 STOPPING。当 stop() 方法成功执行完毕后 Bundle 的状态转变为 RESOLVED。

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

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

3.2.6　卸载过程

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

3.3　启动级别

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

启动级别是一个非负的整数[⊖]，值为 0 时表示 OSGi 框架还没有运行或框架已经停止（根据具体上下文环境来区分这两种情况），只有 Bundle ID 为 0 的 System Bundle 的启动级别可以为 0，除此以外，其他 Bundle 的启动级别都大于 0（最大值为 Integer.MAX_VALUE）。

⊖ 此处的“启动级别”源于 OSGi 规范中的定义。在一些场景（例如图 3-2 中的 Eclipse 调试界面）中，系统 Bundle 的启动级别被列为“–1”，“0”作为其他 Bundle 未设置启动级别时的默认状态。这只是为了在 IDE 中显示方便，在 OSGi 框架运行时是遵循 OSGi 规范定义的。

启动级别的数值越高，所代表的启动阈值就越高，即启动顺序会越靠后。OSGi 框架还有一个活动启动级别（Active Start Level），用于确定目前的状态下应该启动哪些 Bundle。读者可以把它想象成一个指针，在这个指针位置及其之前（Bundle 启动级别小于或等于活动启动级别）的 Bundle 都是已经启动或即将启动的，而在它之后（Bundle 启动级别大于活动启动级别）的 Bundle 都是未曾启动或即将停止的。

下列几个实际开发中常见的场景都可以通过合理安排各 Bundle 的启动级别来实现。

- 安全模式（Safe Mode）：把实现系统核心功能的、可以充分信任的 Bundle 的启动级别限制在某个阈值以内，这样在某个扩展功能的 Bundle 启动失败而导致整个系统无法运作时，可以调整活动启动级别至这个阈值，使系统以安全模式运行。
- 闪屏（Splash Screen）：如果整个系统启动的时间很长，就可能需要在初始化阶段显示一个欢迎屏幕，可以为实现欢迎屏幕的 Bundle 指定一个最小的启动级别，让它最先启动。
- 模块优先级（Bundle Priority）：某些功能，比如性能监控需要尽可能快速运行且不能有很长时间的启动延迟，应当为实现这些功能的 Bundle 指定较小的启动级别以便其尽快启动，而对于一些非关键的附加功能给予较低的优先级，使其延后启动。

3.3.1 设置启动级别

OSGi 框架的活动启动级别和 Bundle 的启动级别都可以在框架启动时设定，也可以在运行期间更改。图 3-2 为在 Eclipse 中调试 OSGi 程序的启动界面，可在“Default Start level”和 Bundle 列表的“Start Level”列中分别设置 OSGi 框架的默认活动启动级别和具体某个 Bundle 的启动级别。

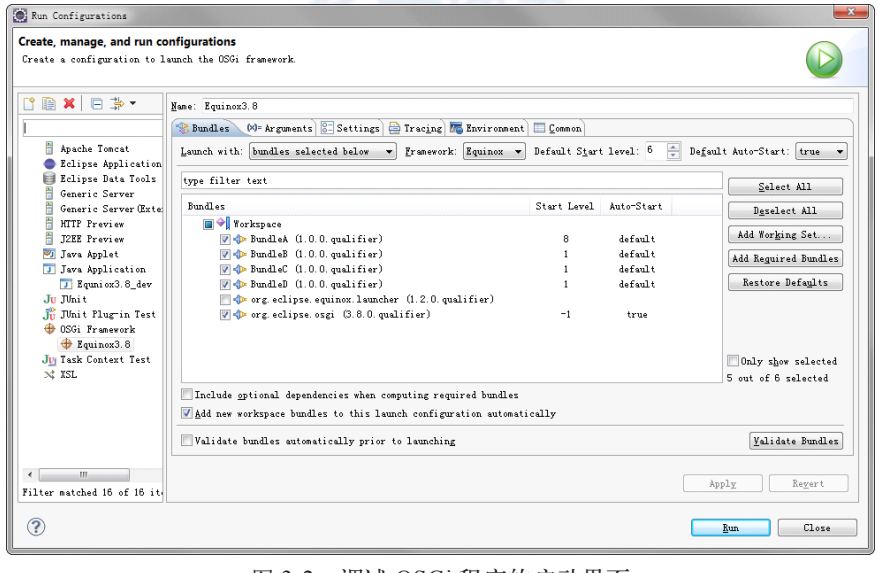

图 3-2 调试 OSGi 程序的启动界面

在启动界面设置的信息，Eclipse 最终都会通过 JVM 系统参数的方式传递给 OSGi 框架，默认启动级别存放到参数“osgi.bundles.defaultStartLevel”中，各个 Bundle 的启动级别和 Bundle 名称一起存放到“osgi.bundles”参数中。在对程序进行实际部署时，我们一般会使用配置文件来设置这些信息，图 3-2 中的配置就相当于在 Equinox 的 configuration\config.ini 配置文件中写入如下内容：

```
#Configuration File
#Wed Dec 07 09:40:32 CST 2011
osgi.bundles=reference\:file\:D\:/_DevSpace/WorkSpaces/equinox/BundleD@1\:start,re-
ference\:file\:D\:/_DevSpace/WorkSpaces/equinox/BundleA@8\:start,reference\:file\:D\:/_
DevSpace/WorkSpaces/equinox/BundleC@1\:start,reference\:file\:D\:/_DevSpace/WorkSpaces/
equinox/BundleB@1\:start
osgi.bundles.defaultStartLevel=6
```

在编码过程中，开发人员可以使用 StartLevel 接口中的以下方法来调整 Bundle 和 OSGi 框架的启动级别。

- ❑ setInitialBundletartLevel()：设置 Bundle 的初始启动级别（Bundle 初次安装时的启动级别）。
- ❑ getInitialBundletartLevel ()：获取 Bundle 的初始启动级别。
- ❑ setBundleStartLevel()：运行时调整 Bundle 的启动级别。
- ❑ getBundleStartLevel()：获取 Bundle 当前的启动级别。
- ❑ setStartLevel()：设置 OSGi 框架的活动启动级别。
- ❑ getStartLevel()：获取 OSGi 框架的活动启动级别。

在系统运行期间，用户还可以使用 Equinox Console 在不停机的情况下动态调整启动级别，Equinox 框架的控制台默认提供了 sl、setfwsl、setbsl 和 setibsl 命令，用于“显示启动 Bundle 的级别”、“设置活动级别”、“设置 Bundle 初始启动级别”和“设置 Bundle 启动级别”的功能，这些功能在后台也是通过调用 StartLevel 接口来实现的。

3.3.2　调整活动启动级别

调整 OSGi 容器活动启动级别是一个渐进的过程。如果要将活动启动级别修改为一个新的值，我们将这个新值称为“请求启动级别”（Requested Start Level）。在 OSGi 容器启动或停止某些 Bundle 的期间，活动和请求的启动级别是不相等的，活动启动级别必须以步长为 1 的速度来增加或减少，逐渐接近并最后与请求启动级别相等。

在活动启动级别向请求启动级别趋近变化的过程中，如果活动启动级别在逐渐增加，那么启动级别与框架活动启动级别相等的 Bundle 都会被启动（执行 Activator.start() 方法），直到达到请求启动级别为止（启动级别与请求启动级别相等的 Bundle 也会被启动）。如果活动启动级别在逐渐减少，那么启动级别与框架活动启动级别相等的 Bundle 都会被停止（执行 Activator.stop() 方法），直到达到请求启动级别为止（启动级别与请求启动级别相等的

Bundle 不会停止）。如果某个 Bundle 在启动或停止过程中抛出了异常，也不能中断活动启动级别的变化过程，但框架会广播一个 FrameworkEvent.ERROR 事件通知所有注册了框架监听器（FrameworkListener）的对象。

当活动启动级别与请求启动级别相等之后，OSGi 框架会广播出一个 FrameworkEvent.STARTLEVEL_CHANGED 事件通知所有框架监听器启动级别调整已经完成。

图 3-3 描述了活动启动级别渐进调整的过程，图中“A”代表活动启动级别的值，“R”代表请求启动级别的值。

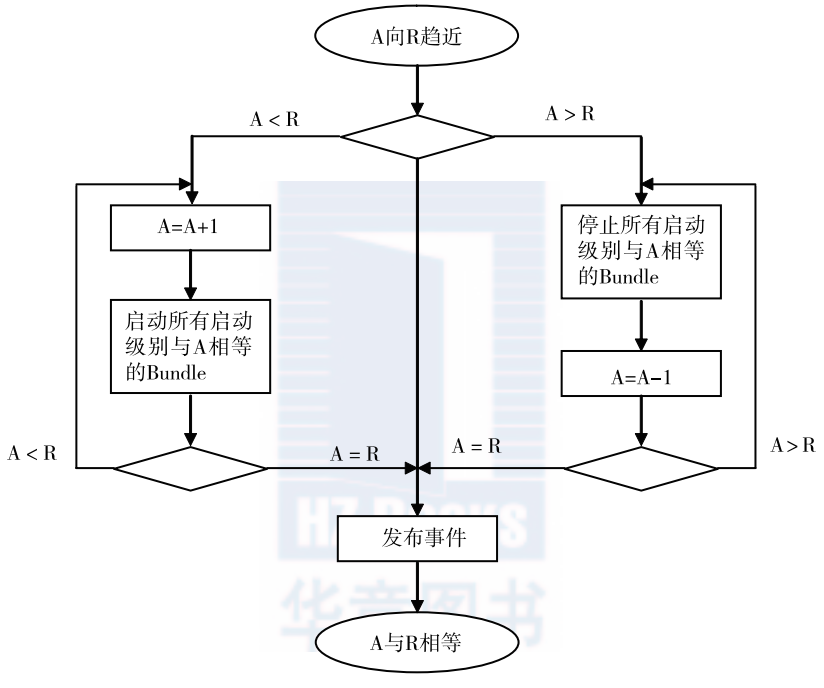

图 3-3 活动启动级别向请求启动级别渐进的过程

当目前框架的活动启动级别小于 Bundle 的启动级别时，即使在代码中直接调用 Bundle 对象的 start() 方法，Bundle 也无法启动。反过来却不成立，当目前框架的活动启动级别大于或等于 Bundle 的启动级别时，在代码中直接调用 Bundle 对象的 stop() 方法可以停止 Bundle。

3.4 事件监听

事件监听在 OSGi 中是一种很常见的设计模式，在 Bundle 生命周期的不同状态相互转换时，OSGi 框架会发布出各种不同的事件供事先注册好的事件监听器处理，这些事件被称为“生命周期层事件”。OSGi 框架支持的生命周期层事件包括继承于 BundleEvent 类的，用于报告 Bundle 的生命周期改变的 Bundle 事件，以及继承于 FrameworkEvent 类的，用于报告框

架的启动、启动级别的改变、包的更新或捕获错误的框架事件。

3.4.1　事件类型

BundleEvent 和 FrameworkEvent 中都定义了一个返回值为 int 的 getType() 方法，这个方法用于说明该事件对象代表了什么事件类型，每一类事件用一个整数来表示。为了以后能够对事件类型进行扩充，事件监听器应当忽略不可识别的事件。Bundle 事件所包含的事件类型和描述如表 3-1 中所示。

表 3-1　Bundle 事件类型

事件名称	描　　述	事件值
INSTALLED	当 Bundle 被成功安装后发出	1
STARTED	当 Bundle 被成功启动后发出	2
STOPPED	当 Bundle 被成功停止后发出	4
UPDATED	当 Bundle 被成功更新后发出	8
UNINSTALLED	当 Bundle 被成功卸载后发出	16
RESOLVED	当 Bundle 被成功解析后发出	32
UNRESOLVED	当 Bundle 转变为未解析状态时发出	64
STARTING	当 Bundle 正处于启动期间发出	128
STOPPING	当 Bundle 正处于停止期间发出	256
LAZY_ACTIVATION	当 Bundle 进入延迟启动状态时发出	512

框架事件包含的事件类型和描述如表 3-2 中所示。

表 3-2　框架事件类型

事件名称	描　　述	事件值
STARTED	当 OSGi 框架启动完成之后发出	1
ERROR	当 OSGi 框架检测到错误信息后发出	2
PACKAGES_REFRESHED	当 OSGi 框架中的 Bundle 被刷新（调用了 FrameworkWiring.refreshBundles() 方法）后发出	4
STARTLEVEL_CHANGED	当 OSGi 框架的活动启动级别被改变（调用了 FrameworkStartLevel.setStartLevel() 方法）后发出	8
WARNING	当 OSGi 框架检测到警告信息后发出	16
INFO	当 OSGi 框架检测到一般信息后发出	32
STOPPED_UPDATE	当 Bundle 更新操作导致 OSGi 框架暂时停顿时发出	64
STARTING	当 Bundle 正处于启动期间发出	128

（续）

事件名称	描　　述	事件值
STOPPED_BOOTCLASSPATH_MODIFIED	当 OSGi 框架因系统 Bundle 被更新或附加了一个提供 BootClasspath 的 Bundle 而停止时发出	256
WAIT_TIMEDOUT	当操作等待 OSGi 框架停止（比如调用了 Framework.waitForStop() 方法），但框架直到超时都仍未停止，此时发出该超时事件	512

与 Bundle 状态值类似，代表事件类型的整型值也使用位掩码来描述，这便于一个事件监听器同时处理多种事件类型，示例如下：

```
if((event.getType() & (INSTALLED | STARTED | STOPPED)) != 0){
   // 只能在 INSTALLED、STARTED、STOPPED 事件下执行的动作
}
```

3.4.2 事件分派

与 BundleEvent 和 FrameworkEvent 类相对应，OSGi 规范定义了 BundleListener 和 FrameworkListener 接口来描述 Bundle 事件和框架事件的监听器。这两个监听器接口中分别包含了 BundleListener.bundleChanged() 和 FrameworkListener.frameworkEvent() 方法，接收到事件广播之后在这两个方法中进行相关处理，这些方法的处理动作都是默认异步执行的，不会阻塞 Bundle 和框架的状态转换过程。

另外，OSGi 规范明确规定了必须以事件发生那一刻的监听器快照作为事件分派目标，因此可能出现在事件监听器执行时，这个监听器实际上已经不再监听当前所发生的事件的情况。具体情况笔者通过以下三个场景进行说明。

1）情景一：事件顺序如图 3-4 所示。

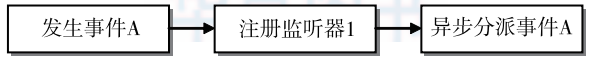

图 3-4　情景一示例

在这个场景中，监听器 1 不能接收到事件 A，因为 OSGi 框架以事件发生那一刻的监听器快照作为事件分派目标，而监听器 1 不是在事件发生前注册的。

2）情景二：事件顺序如图 3-5 所示。

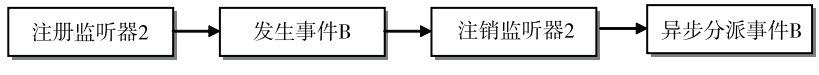

图 3-5　情景二示例

在这个场景中，已被注销的监听器 2 依然可以接收到事件 B，同样是因为 OSGi 框架以事件发生那一刻的监听器快照作为事件分派目标。

3）情景三：事件顺序如图 3-6 所示。

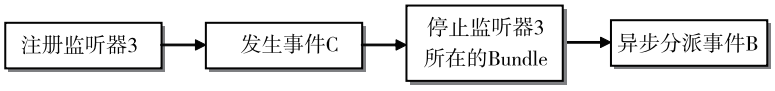

图 3-6　情景三示例

在这个场景中，监听器 3 也无法接收到事件 C，因为它的上下文对象（BundleContext）此时已经不存在（准确地说是存在但不可用）了。

BundleListener 接口还派生了一个 SynchronousBundleListener 子接口，顾名思义，这个子接口被用来进行同步处理。因为 STARTING 和 STOPPING 这两个 Bundle 事件描述的是一个持续过程而非瞬间的状态转换，所以它们的监听器应当同步执行才是合理的，对于这两个事件，OSGi 框架也只会分派给同步的监听器。

OSGi 分别通过 BundleContext.addBundleListener() 和 BundleContext.addFrameworkListener() 添加 Bundle 事件和框架事件的监听器到 OSGi 框架的事件监听列表之中，添加监听器的动作一般在 Bundle 的 Activator 类中实现。

3.5　系统 Bundle

OSGi 框架本身也会以一个 Bundle 的形式向其他 Bundle 提供资源、Package 和服务，比如已经在书中多次出现的 Bundle、BundleContext、FrameworkListener 等接口，以及后面将会介绍的 EventAdmin、PackageAdmin 等服务都是由系统 Bundle 提供的。OSGi 规范规定了系统 Bundle 的 Bundle ID 固定为 0，Bundle 的 getLocation() 方法返回固定字符串“System Bundle”，这些特征使得任何 Bundle 都可以很方便地从 BundleContex.getBundle(0) 或 BundleContex.getBundle("System Bundle") 方法中获取到系统 Bundle 的对象实例。

在 OSGi 容器中，系统 Bundle 可以认为是一定存在的，每一个 Bundle 都默认依赖这个系统 Bundle。下面列出了 Equinox 框架的系统 Bundle 的元数据信息。

```
osgi> headers 0
Bundle headers:
 Bundle-Activator = org.eclipse.osgi.framework.internal.core.SystemBundleActivator
 Bundle-Copyright = Copyright (c) 2003, 2004 IBM Corporation and others. All
rights reserved. This program and the accompanying materials  are made available under
the terms of the Eclipse Public License v1.0 which accompanies this distribution, and
is available at http://www.eclipse.org/legal/epl-v10.html
 Bundle-Description = OSGi System Bundle
 Bundle-DocUrl = http://www.eclipse.org
 Bundle-Localization = systembundle
 Bundle-ManifestVersion = 2
 Bundle-Name = OSGi System Bundle
 Bundle-RequiredExecutionEnvironment = J2SE-1.5,OSGi/Minimum-1.2
 Bundle-SymbolicName = org.eclipse.osgi; singleton:=true
 Bundle-Vendor = Eclipse.org - Equinox
 Bundle-Version = 3.8.0.qualifier
```

```
Eclipse-BundleShape = jar
Eclipse-ExtensibleAPI = true
Eclipse-SystemBundle = true
Export-Package = org.eclipse.osgi.event;version="1.0",
……// 版面关系省略其他 Package
Export-Service = org.osgi.service.packageadmin.PackageAdmin,org.osgi.service.
permissionadmin.PermissionAdmin,org.osgi.service.startlevel.StartLevel,org.eclipse.
osgi.service.debug.DebugOptions
Main-Class = org.eclipse.core.runtime.adaptor.EclipseStarter
Manifest-Version = 1.0
```

系统 Bundle 与 OSGi 框架密不可分，由于它的特殊性，其生命周期变化过程也与普通 Bundle 有所区别。以下是 OSGi 规范对系统 Bundle 生命周期几个过程执行的动作规定。

- ❑ 启动过程：Bundle 的 start() 方法为空操作，因为 OSGi 框架一启动，系统 Bundle 就已经启动。
- ❑ 停止过程：Bundle 的 stop() 方法会立即返回并在另外一条线程中关闭 OSGi 框架。
- ❑ 更新过程：Bundle 的 update() 方法会立即返回并在另外一条线程中重启 OSGi 框架。
- ❑ 卸载过程：系统 Bundle 无法卸载，如果执行了 Bundle 的 uninstall() 方法，那么框架会抛出一个 BundleException 异常。

系统 Bundle 的启动级别固定为 0，这个启动级别是无法使用 StartLevel 接口中的 setBundleStartLevel() 进行修改的；如果这样做了，那么 OSGi 框架将会抛出一个 IllegalArgumentException 异常。

3.6 Bundle 上下文

OSGi 容器中运行的各个 Bundle 共同构成了一个微型的生态系统，Bundle 的许多行为都无法孤立进行，必须在特定的上下文环境中才有意义，因为要与上下文的其他 Bundle 或 OSGi 框架进行互动。在代码中使用 BundleContext 对象来代表上下文环境，当 Bundle 启动的时候，OSGi 框架就创建这个 Bundle 的 BundleContext 对象，直到 Bundle 停止运行从 OSGi 容器卸载为止。Bundle 在进行以下操作时，需要通过 BundleContext 对象来完成。

- ❑ 安装一个新的 Bundle 到当前 OSGi 容器之中（BundleContext.installBundle() 方法）。
- ❑ 从当前 OSGi 容器中获取已有的 Bundle（BundleContext.getBundle() 方法）。
- ❑ 在 OSGi 容器中注册服务（BundleContext.registerService() 方法）。
- ❑ 从当前 OSGi 容器中获取已有的服务（BundleContext.getService() 方法）。
- ❑ 在 OSGi 容器中注册事件监听器（BundleContext.addBundleListener()、BundleContext.addFrameworkListener() 方法）。
- ❑ 从 Bundle 的持久储存区中获取文件（BundleContext.getDataFile() 方法）。
- ❑ 获取 Bundle 所处环境的环境属性（BundleContext.getProperty() 方法）。

每个 Bundle 的 BundleContext 对象都是在调用 Bundle.start() 方法时由 OSGi 框架创建并

以方法参数的形式传入到 Bundle 中，实现 Bundle 时一般会把这个对象的引用保留起来，以便在 Bundle 其他代码中使用，示例如下：

```
public class Activator implements BundleActivator {

    private static BundleContext context;

    // Bundle 中其他代码调用 Activator.getContext() 方法获取
    // BundleContext 对象
    public static BundleContext getContext() {
        return context;
    }

    public void start(BundleContext bundleContext) throws Exception {
        // 将 BundleContext 对象的引用保留起来
        Activator.context = bundleContext;
    }
}
```

上下文对象涉及 Bundle 的数据安全和资源分配，它应当是 Bundle 私有的，不应当传递给其他 Bundle 使用。在 Bundle 的 stop() 方法执行完之后，Bundle 对象就会随之失效，如果在 Bundle 停止后继续使用 BundleContext 对象，那么 OSGi 框架就会抛出 IllegalStateException 异常。

3.7　本章小结

本章介绍了 Bundle 如何启动，Bundle 自安装到卸载所经历的生命周期状态和这些状态的条件转换过程，还介绍了 OSGi 框架是如何使用启动级别对系统中的 Bundle 进行管理调度的。

掌握了模块层和生命周期层的知识，我们已经可以构建一个最基本的可运行的 OSGi 系统了。但是 OSGi 对不同开发和应用场景还提供了非常多的标准服务支持，使用这些标准服务不仅能使我们的开发事半功倍，还能保证我们编写的程序具备通用性，符合业界接口标准。因此接下来我们将介绍 OSGi 的服务层，它是定义和执行所有标准服务的基础。

51CTO.com
技术成就梦想
联合策划

Understanding the OSGi
Principles, Using and Best Practices

计算机硬件的工业化生产已经非常成熟，无论是哪个公司生产的硬件产品，都能遵循统一规范的接口，有无数可替换的标准件可以使用，但是整个软件行业还在经常为解决各种软件系统、硬件设备中相同的业务需求而从零开始制造一个又一个相似的“轮子”。硬件行业的工业化生产方式让软件行业的程序员们羡慕不已。

软件工业化必定是一股不可逆转的潮流，要实现这个目标：第一步就是要订立不同功能模块的标准，Java业界已经有了很多的这类技术规范，例如EJB、JTA、JDBC、JMS等；第二步是为这些技术规范提供一致的管理和交互方式，Java所欠缺的正是这样一个扮演组织者或黏合剂的角色，直到出现了OSGi。

本书一共分为四个部分，秉承理论与实践相结合的原则，对OSGi进行了深入且全面的讲解，既详细讲解了OSGi的规范、原理和核心服务，又深入分析了Equinox框架的源代码，最重要的是，本书还总结了大量关于OSGi的最佳实践。各个部分之间基本上是互相独立的，没有必然的前后依赖，读者可以从任何一个感兴趣的专题开始阅读，但是每个部分中的各个章节间则会有先后顺序。

上架指导：计算机/程序设计/Java
ISBN 978-7-111-40887-1
9 787111 408871 >
定价：79.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com