

OpenDoc Series'

OSGI 实战

V1.0

作者: BlueDavy

So many open source projects. Why not **Open** your **Documents**? ©

文档说明

参与人员:

作者	联络
BlueDavy	BlueDavy@gmail.com

文中的代码请参见随文发布的 `code.rar`。classic 目录放置了基于 Equinox 的实战部分的代码；ds 目录放置了基于 ds 重构后的代码；EventAdmin 目录放置了使用 EventAdmin Service 的演示代码。

随文还发布了一个可直接运行的环境 `dist.rar`，解压后直接运行其中的 `run.bat`，就可通过 <http://localhost:8080/demo/page/login.htm> 来访问用户登录验证模块。

发布记录

版本	日期	作者	说明
1.0 Beta	2006.08.08	BlueDavy	原创 预览版
1.0	2006.08.25	BlueDavy	增加基于 Bridge 方式开发 B/S 应用的章节； 增加对于 Configuration Admin Service 和 Event Admin Service 讲解的章节； 增加 OSGI 关键部分讲解章节； 增加面向接口开发章节； 格式编排；

OpenDoc 版权说明

本文档版权归原作者所有。

在免费、且无任何附加条件的前提下，可在网络媒体中自由传播。

如需部分或者全文引用，请事先征求作者意见。

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

Open Doc Series 目前包括以下几份文档：

- **Spring 开发指南**
- **Hibernate 开发指南**
- **ibatis2 开发指南**
- **Webwork2 开发指南**
- **持续集成实践之 CruiseControl**

请订阅 <http://groups.google.com/group/redsaga-notify>，以获得新版本及其他 Opendoc 的 release 通知，或从 <http://wiki.redsaga.com> 获取最新更新信息。

目录

一.	序.....	5
二.	体验OSGI.....	7
2.1.	需求实现.....	7
2.2.	技术角度.....	9
三.	OSGI带来什么.....	11
四.	OSGI案例.....	13
五.	OSGI框架.....	15
5.1.	Equinox.....	15
5.2.	Oscar.....	15
5.3.	Knopflerfish.....	15
六.	基于OSGI框架（Equinox）的实战.....	17
6.1.	做好准备.....	17
6.2.	工具箱.....	18
6.3.	开发Bundle.....	22
6.4.	开发、发布和使用Service.....	29
6.5.	测试和调试.....	33
6.6.	发布基于OSGI的系统.....	34
6.7.	Equinox基于OSGI的扩展.....	37
6.8.	现有类型系统基于OSGI的开发.....	38
6.8.1.	B/S.....	38
6.8.2.	C/S.....	39
6.8.3.	嵌入式.....	39
6.9.	注意事项.....	39
七.	深入OSGI.....	41
7.1.	关于OSGI.....	41
7.2.	OSGI R4 规范.....	41
7.2.1.	Core Framework.....	41
7.2.2.	StartLevel Service.....	58
7.2.3.	Declarative Services.....	62
7.2.4.	Configuration Admin Service.....	72
7.2.5.	Event Admin Service.....	75
7.3.	OSGI关键部分讲解.....	77
7.3.1.	ClassLoader.....	77
7.3.2.	Bundle的生命周期.....	80
7.3.3.	Bundle的通讯机制.....	80
7.3.4.	DS中Component的生命周期.....	82
7.3.5.	DS中Component的通讯机制.....	83
八.	应用OSGI.....	84
8.1.	模块化设计.....	84
8.2.	面向服务的组件模型设计.....	85
8.3.	动态性设计.....	85
8.4.	面向接口的开发.....	85

九.	OSGI资源.....	87
十.	OSGI框架前瞻.....	88
十一.	OSGI带来的遐想.....	89
十二.	参考文献.....	90

一. 序

自工作以来就一直很关注插件体系结构的东西，从最早对于 ant、maven 这些部分支持插件思想的开源工具的关注，到对于 portal 时代 portlet 这种较为完整的插件体系结构的关注，到对于 Eclipse 3.0 采用 OSGI 作为其插件体系结构的思想的关注。

但真正开始在实际的产品中使用插件体系结构却是 04 年的事，在 04 年开始在产品中采用插件体系结构作为系统的基础架构，不过由于当时对于 OSGI 的陌生，所以在产品中采用的是自主编写插件架构实现的方式，事实证明插件体系结构的实现远不如想像中的那么简单，最终基于那个自主实现的插件架构来开发系统的插件并不容易，而且也不方便，也是基于这次的失败上让自己真正深刻的去思考插件体系结构，开始关注 Eclipse 采用的 OSGI，OSGI 在插件体系结构的独到的设计让它得到了 Eclipse 的认可。

在对 OSGI 规范进行研读的同时，无意中发现 Eclipse 成立了一个作为 OSGI R4 RI 的工程——Equinox，出于对 OSGI 的认同以及对于 Eclipse 系列 Project 的高质量的信任，尽管开源界中还有象 Oscar、Knopflerfish 这样的知名的 OSGI 框架，还是极度的关注 Equinox 这个工程，对 Equinox 进行了试用，不出意料，尽管 Equinox 还没有正式的发布版本，甚至连里程碑版本都还没有，但 Equinox 作为 OSGI R4 RI，表现非常出色，特别是借助 Eclipse 提供的 IDE，更是令它在 OSGI 框架的竞争中占据优势。

刚好在这个时候手上有一个新的产品，之前的试用让自己对于 Equinox 有了些理解，便决定采用 Equinox 作为这个产品的底层框架，经过几个月的开发后，Equinox 确实给这个产品带来了很多的优点，但同时由于 Equinox 文档的缺乏以及国内对于 OSGI 不高的关注度，在开发的过程中也是碰到了不少的问题，但总体而言 Equinox 给这个产品带来的优点还是多于缺点。

OSGI 在国内的关注度目前仍然很低，而 Equinox 更是，其实在国外 OSGI、Equinox 的关注度都算挺高的，目前国外已经有不少的项目采用 Equinox 作为基础框架了，OSGI 的推广之所以比较难的原因就在于 OSGI 的引入并不象决定项目是采用 struts 还是 webwork 那么简单，OSGI 带来的是设计思想以及开发方式的改变，这也就一定程度上要求系统设计师以及程序开发人员要接受一种新的开发方式，形象的说我觉得就是要让你从 jsp+javaBean 的方式转为采用 MVC 框架的方式，自然会有些不适应的感觉，但相信只要接受了，会体现出它的足够优势，而插件体系结构是我认为在未来几年内将流行的开发方式，目前国外对这方面其实也属于摸索阶段，如果现在我们就能够对 OSGI 这

样优秀的插件体系结构进行足够的熟悉和关注,那么也许我们能够有机会第一次提出更为领先的设计思想,而不是一直在思想级别远落后于国外,被国外牵着鼻子走,希望能够通过这篇 Opendoc 对 OSGI、Equinox 做一个较为完整的介绍,从而有更多的人能够对 OSGI、Equinox 进行关注,同时也希望更多的人将 OSGI、Equinox 应用到实际的项目、产品中去并提出基于插件体系结构的最佳实践。

本篇 Opendoc 按照学习开源框架的基本流程进行编写,从体验 OSGI 到基于 OSGI 框架的实战,到深入 OSGI,完成对于 OSGI 从入门到深入学习的过程,最后对于 OSGI 的现状和发展发表些自己的看法和思考,限于笔者的水平以及时间,文内难免有些错误,还请大家不吝指正,也希望本文能作为国内 OSGI 的抛砖之作,引出更多的关于 OSGI 的 Opendoc,在我的 blog 上也会不断的编写关于自己在 OSGI、Equinox 上的实战的体会和心得,欢迎大家多多交流。

在编写这篇 Opendoc 的过程中,感谢众多网友提出的意见,在此特别感谢一餐三碗、jazzy、caoxg 以及 brokendoor 等朋友对预览版提出的建议,使得正式版得以早日完成。

二. 体验 OSGI

在这个章节中分别从需求实现以及技术角度两方面来体验 OSGI 带给我们的享受。

2.1. 需求实现

以一个用户登录验证模块来体验下OSGI有什么不同的地方,在下面的描述中会涉及到一些OSGI的专业术语,暂时先不管它,在后续**OSGI R4 规范**的章节中会详细介绍。

用户登录验证模块

要求可动态的替换(系统运行期)用户登录验证的方式,如目前需要的有三种验证方式:LDAP 验证、DB 验证以及配置文件验证;同时要求可随时增加新的验证方式(系统运行期),如 CA 验证方式等。

这是常见的一种较为简单的动态修改和改变系统行为的需求,分别来看看传统的实现方式和基于 OSGI 的实现方式。

- 传统的实现方式

- 动态替换用户登录验证方式的实现

定义用户登录验证的接口,按照需求编写实现此接口的三个类,分别为 LDAP 验证类、DB 验证类以及配置文件验证类。

编写配置文件,格式类似如下:

```
LDAP=org.riawork.auth.LDAPAuthImpl
```

```
DB=org.riawork.auth.DBAuthImpl
```

```
FILE=org.riawork.auth.FileAuthImpl
```

```
DEFAULT=DB
```

编写登录验证类,登录验证类读取配置文件,获取目前系统采用的验证方式,根据验证方式获取对应的验证类,通过 `Class.forName().newInstance` 的方式获取到验证类的实例,造型成用户登录验证的接口,调用验证方法完成用户登录验证。

提供登录验证方式的管理端,用户可选择采用何种方式进行登录的验证,当用户选择不同的验证方式时相应的维护配置文件。

经过这样的几步后动态替换用户登录验证方式的需求得以实现了。

- 随时增加新的验证方式的实现

这个在采用传统的实现方式的时候就比较麻烦,Java 是编译性质的语言,而不

是解释性质的语言，所以在动态加载类方面是要专门处理的，在 Java 中通常采用自定义 `ClassLoader` 的方法去实现，为了实现能随时增加新的验证方式，也需要采用这样的方法。

编写自定义的 `ClassLoader`，该 `ClassLoader` 负责扫描验证类文件夹目录，加载其中所有的类文件，在有新文件放入时，将自动的将其加载到这个自定义的 `ClassLoader` 中。

修改之上的登录验证类，登录验证类读取配置文件，获取目前系统采用的验证方式，根据验证方式获取对应的验证类，这个时候通过自定义的 `ClassLoader` 来获取验证类的实例。

经过这样的步骤后，随时增加新的验证方式的需求也得以实现了。

可以看出，在采用这样的方式实现一个简单的动态修改和改变系统行为的需求是比较的复杂，需要利用到 Java 的底层特性，从一定程度上来说是有不小的难度和复杂度的，可想而知如果要想实现复杂的动态修改和改变系统行为的需求的时候，传统的开发方式是很难胜任的，接下来我们来看看基于 OSGI 会怎么样去实现这样的两个需求呢。

- **基于 OSGI 的实现方式**

- **动态替换用户登录验证方式的实现**

在基于 OSGI 时可以采用这样两种方式来实现这个需求：

- ◆ **动态替换 Service 的方式**

编写登录验证类，该类中拥有 `setAuthService(AuthService service)` 这样的方法，在做验证时通过注入的这个 `service` 来验证。

编写用户登录验证(`AuthService`)接口。

编写实现 `AuthService` 的三个类。

将以上的验证类和用户登录验证接口作为一个 `Bundle`¹。

将实现 `AuthService` 的 `LDAPAuthServiceImpl`、`DBAuthServiceImpl` 以及 `FileAuthServiceImpl` 分别作为 `Bundle` 部署。

将这四个 `Bundle` 部署到 OSGI 框架中并启动。

用户需要动态替换用户验证方式的时候，只需要通过 OSGI 框架的管理端修改登录验证类 `Bundle` 中获取 `Service` 的标识即可，如之前配置的登录验证类 `Bundle` 中获取 `Service` 的标识为 `DB`，用户可修改为 `LDAP` 或者 `FILE`，

¹ `Bundle`是OSGI中的核心概念，可以看成就是一个jar文件，具体参见后续的**OSGI R4 规范**。

或者用户可以直接修改登录验证类 **Bundle** 的配置文件，在修改完毕后登录验证的方式就被动态的替换了。

◆ **Bundle** 控制的方式

和动态替换 **Service** 的方式不同的方法就是在部署和用户使用的时候稍有不同。

部署的时候部署四个 **Bundle**，但只启动里面用来做验证的那个 **Bundle**，如目前用户要使用的为 **LDAP** 验证，就只启动 **LDAPBundle**。

当用户需要改变验证方式时，可以先停止 **LDAPBundle**，然后启动 **DBBundle** 或 **FileBundle**，这样就完成了验证方式的动态替换。

■ 随时增加新的验证方式的实现

基于 **OSGI** 实现这个需求就简单了，编写一个实现 **AuthService** 接口的 **Bundle**，然后将 **Bundle** 部署到 **OSGI** 框架中即完成了，之后通过 **OSGI** 管理端即可启动这个 **Bundle**，采用之上的动态替换用户登录验证方式的方法即可在系统中使用这个新增的 **Bundle** 了。

可以看出，基于 **OSGI** 的实现方式要实现这两个需求非常的简单，可能现在你看基于 **OSGI** 的实现方式还有点看不太懂，这没关系，在后续**基于OSGI框架（Equinox）的实战**章节中会进行详细的讲解，这里主要是让大家体验下 **OSGI** 对于动态修改和改变系统行为的支持。

2.2. 技术角度

在搭建系统时，模块的组织方式决定了系统将如何进行开发以及如何部署，而模块的复用和扩展则是公司希望通过项目形成的积累，避免重复的投入，同样，分别看看传统的方式和基于 **OSGI** 的方式。

● 传统的方式

■ 模块的组织

传统的方式下通常采用整个系统作为一个工程或每个模块一个工程的方式，在整个系统作为一个工程的方式下区分模块主要是通过包名的方式来区分，而每个模块一个工程的方式在传统的方式下会很麻烦，主要是在包的引用上，很容易出现模块交叉引用的现象，导致了在开发的时候非常的麻烦，同时也导致了部署的时候非常麻烦，鉴于此，在传统的方式下通常都是采用整个系统一个工程，通过包名来区分模块的方式。

- 模块的复用和扩展

在采用整个系统一个工程，通过包名来区分模块的方式自然使得模块的复用变得特别的复杂，在每个模块一个工程的方式则可以让模块的复用比较的简单，由于模块的组织通常来说和系统的基础架构有直接的关联，而由于系统的基础架构没有形成规范，这样就导致有可能因为模块的组织方式不同而无法复用的现象。

模块的扩展在传统的方式通常来讲只能通过修改原有模块的代码来实现。

- 基于 OSGI 的方式

- 模块的组织

基于 OSGI 的方式下可采用每个模块一个 Bundle 的方式来进行组织，而在 OSGI 框架的支持下不会出现需要引用其他模块 Bundle 的情况，而只需要引用接口就可以了，这就保证了每个模块一个 Bundle 的开发方式并不会很复杂，同时 Bundle 的部署也是非常的简单。

- 模块的复用和扩展

在基于 OSGI 的方式下，模块的复用就非常简单了，由于 OSGI 是规范性质的定义，只需要将 Bundle 部署上去即可为系统增加相应的模块。

模块的扩展在 OSGI 标准的规范中也是没有定义的，基本上同样只能采用直接修改原有代码的方式来实现，不过 Equinox 吸取了 Eclipse Extension Points 的设计，使得 Bundle 的扩展变得完全可行。

三. OSGI 带来什么

在上一个章节中，从需求实现和技术角度分别体验了 OSGI，对于基于 OSGI 搭建的系统传统的系统有什么不同有了个大概的印象，那么到底 OSGI 给我们带来了什么呢？

从需求实现方面，OSGI 为动态扩充、修改系统功能和改变系统行为提供了支撑，而在传统的开发方式下，要实现系统的动态扩充、修改以及改变是一件很麻烦的事。

从技术角度方面，OSGI 带来了规范化的模块组织以及统一的开发方式，这为传统的模块的组织、模块开发以及模块积累提供了一种全新的指导以及支撑。

来具体看看 OSGI 提供的这些支撑能给具体的项目带来什么不一样的感觉。

- **可插拔的系统**

估计很多人都接触过路由器，大部分的路由器都支持模块的热插拔，这就意味着可以在路由器运行的状况下给它动态的增加新的功能或者卸载不需要的功能，硬件界的这种热插拔技术一直就是软件界所追求的，而 OSGI 则使得热插拔技术在软件界成为现实。

基于 OSGI 的系统，可通过安装新的 Bundle、更新或停止现有的 Bundle 来实现系统功能的插拔。

- **可动态改变行为的系统**

在业界可插拔的系统其实并不少，但可动态改变行为的系统并不多，JMX, no no no, JMX 和 OSGI 在动态改变行为方面绝对不是一个级别的，OSGI 有一整套完整的机制去实现动态改变系统行为。

改变系统行为的典型例子见上章节中的动态改变登录验证方式，无需我说，基于这个基础上可以做出更为复杂的多的动态改变系统行为的支持。

可插拔、可动态改变行为从根本上保证了系统在运行期足够的灵活性和扩展性。

- **稳定、高效的系统**

基于 OSGI 的系统采用的是微核机制，微核机制保证了系统的稳定性，微核机制的系统只要微核是稳定运行的，那么系统就不会崩溃，也就是说基于 OSGI 的系统不会受到运行在其中的 Bundle 的影响，不会因为 Bundle 的崩溃而导致整个系统的崩溃。

OSGI 的动态性原则保证了系统的高效，只有在请求发生时 OSGI 才去完全加载、启动相应的 Bundle、Service。

- **规范的、可积累模块**

规范的模块开发方式其实是大部分软件公司都期盼的，规范的模块开发方式就意味着规范的人员技能培养体系以及规范的人员技能要求，这对于软件公司来讲是很重要的。

但为什么大部分软件公司都形成不了规范的模块开发方式呢，因为没有统一的规范的基础架构体系的定义，往往每个项目、每个产品都会因为架构师的偏好、技术的发展而导致模块的开发方式完全不同，这就使得软件公司在人员技能要求、培养上很难形成统一，而 OSGI 为这个问题提供了解决方案，基于 OSGI 的系统采用规范的模块开发、部署方式构建系统。

当然，采用 OSGI 作为规范的模块开发、部署方式自然给现有梯队提出了新的要求，对于设计师而言，需要学习新的基于 OSGI 的模块分解、设计方式，对于开发人员而言，需要学习新的基于 OSGI 的开发方式，但对于公司形成规范的模块开发方式能带来的回报而言，这样的付出是值得的，而且，这个学习成本并不是很高。

模块的积累是软件公司发展的基础，只有公司独特的竞争力的项目经验模块被积累下来了，公司的发展才能一直的持续和高速，而在形成了规范的模块开发、部署方式后，模块的积累自然水到渠成。

OSGI 除了带来这些的明显的优点和挑战之外，还带给我们更多，例如面向服务的组件模型设计思想、高效系统设计思想等，也希望在实际的项目中 OSGI 能让你享受到更多，同时也希望你能分享更多给大家。

四. OSGI 案例

说了这么多 OSGI 的好，吹了这么多的“牛”，来以事实证明下 OSGI 并不是什么实验品，而是已经在业界被证明切实可行的东西。

OSGI 典型的应用案例主要有两个，都非常的著名，分别是 Eclipse 和 BMW 汽车的应用控制系统。

- **Eclipse**

Eclipse 作为 Java 业界成功的 IDE project，在 3.0 以前的版本它采用的是自己设计的一套插件体系结构，而 Eclipse 的插件体系结构在整个业界都是非常知名的，也是被认为非常成功的一种设计，但 Eclipse 在 3.0 版本时却做了一个重大决度，就是推翻它自己以前的插件体系结构（虽然开始只是做兼容的方式，随着版本的逐渐升高已经开始逐步的替换工作了），而转为直接采用 OSGI 作为其插件体系结构，这到底是为什么呢？

Eclipse 的插件体系结构和 OSGI 的思想非常的耦合，都强调微核+系统插件+应用插件的概念，Eclipse 之所以要抛弃自己那套已经比较成熟的插件体系结构而转而采用 OSGI，就是因为 OSGI 的规范性以及 OSGI 对于插件体系结构更为完整的定义，当然，还有一些官方性质的原因，这些原因在这里暂且不提，Eclipse 采用 OSGI 作为其插件体系结构的成功是很明显的，在 Eclipse 3.1 版本以后大家可以明显的感觉到启动速度的提升，同时也使得可以在运行时对插件进行管理，更明显的提升是插件的开发更加的规范，从而可以使用很多已有的 OSGI 插件。

Eclipse 同时也带来了业界良好的插件系统的体验以及插件系统的开发经验。

- **BMW 汽车的应用控制系统**

BMW 汽车的应用控制系统采用 OSGI 作为其底层架构，估计这一定程度上颠覆了很多人对于 Java 的认识，很多人都认为基于 java 的系统低效，不可能用于汽车这样的应用控制系统上，在 EclipseCon 2006 会议上 BMW 采用 OSGI 得到了证实，估计是猜想会被很多人怀疑，演讲者在 PPT 上讲了下 BMW 汽车的应用控制系统，这套系统主要用来控制汽车上的音箱、灯光等等设备，总共由 1000 多个 Bundle 构成，但 BMW 汽车的应用控制系统启动时间却只需要 3.5 秒，是不是很令人惊讶呢，这也从很大程度上反应了采用 OSGI 的系统的效率并不会低。

这两个非常成功的案例向大家证明了基于 OSGI 开发系统的可行性，同时这个两个成功

案例的足够的知名性以及优秀的使用、技术效果也为 OSGI 的推广铺设了不错的基础，到这篇文档完稿之时，关于 OSGI 被商业领域（例如 IBM P5 服务器系列、Websphere V6.1、Lotus Sametime、Adobe CS2 等）、知名开源软件领域（例如 Apache 等）采用的消息已经是不断的传出，可以看出 OSGI 在服务器端应用、企业应用中已经开始广泛流行了，而这对于 OSGI 更好的发展成为支撑服务器端应用和企业应用的规范会起到很好的推动作用。

五. OSGI 框架

在开源界中实现 OSGI 的框架比较知名的有：Equinox、Knopflerfish、Oscar。

5.1. Equinox

Equinox 是 Eclipse 中的项目之一，Equinox 是作为 OSGI R4 RI 而知名的，同时由于 Equinox 有 Eclipse IDE 这么个成功案例，反应出了 Equinox 作为 OSGI 框架的优势。

Equinox 目前是随着 Eclipse 版本而发布的，同时，它也提供独立的下载，在独立的下载页面中可以下载到 Equinox 对于 OSGI R4 的所有实现以及 Equinox 扩展 OSGI R4 而提供的 Bundle。

Equinox 开发小组由 IBM 的 Jeff 领衔，开发状态非常的活跃，从它的开发者 maillist 可以看出，讨论非常的热闹，大家感兴趣的话可以申请加入开发者 maillist：

<http://dev.eclipse.org/mailman/listinfo/equinox-dev>。

想了解更多 Equinox 信息请参看：

官方站：<http://www.eclipse.org/equinox>

中文站：<http://www.riawork.org>

5.2. Oscar

Oscar 是一个遵循 OSGI R3 框架的实现，目前它的开发状态不怎么的活跃，最新的新闻都是 2005 年的了。

Oscar 的优势在于提供了大量 OSGI R3 标准之外的 Bundle，为开发基于 OSGI 的系统提供了方便。

更多 Oscar 的信息请参看：

官方站：<http://oscar.objectweb.org/>

5.3. Knopflerfish

Knopflerfish 是一个知名的 OSGI 框架，目前提供的最新版本也已经完整的实现了 OSGI R4，Knopflerfish 的开发状态非常的活跃，同时它也提供了为方便开发基于 OSGI 系统的大量的 Bundle。

更多 Knopflerfish 的信息请查看：

官方站：<http://www.knopflerfish.org>

三种框架各有优劣，我也没有仔细的对三个框架做过具体的比较，出于对 Eclipse 项目的信任、Equinox 作为 OSGI R4 RI 以及 Equinox 有 Eclipse 这个强大的开发 IDE 作为其

开发环境的原因,在实际的应用中我选用了 Equinox 作为项目/产品中使用的 OSGI 框架,由于 OSGI 是个标准的规范,在开发需要的情况下可以选用 Oscar、Knopflerfish 提供的大量 Bundle 做为项目/产品中的基础设施。

六. 基于 OSGI 框架 (Equinox) 的实战

讲了这么多 OSGI 的相关“废话”，开始走入正题，任何东西，仅仅说是没用的，在本章节中我们基于 OSGI 框架(Equinox)来实现体验 OSGI 章节中的用户登录验证模块。

6.1. 做好准备

OSGI 框架是一个微核结构的容器，所有的模块都需要运行在容器范围内，在 OSGI 中所有模块的部署都必须以 Bundle 的方式来进行部署，那么到底什么是 Bundle 呢？

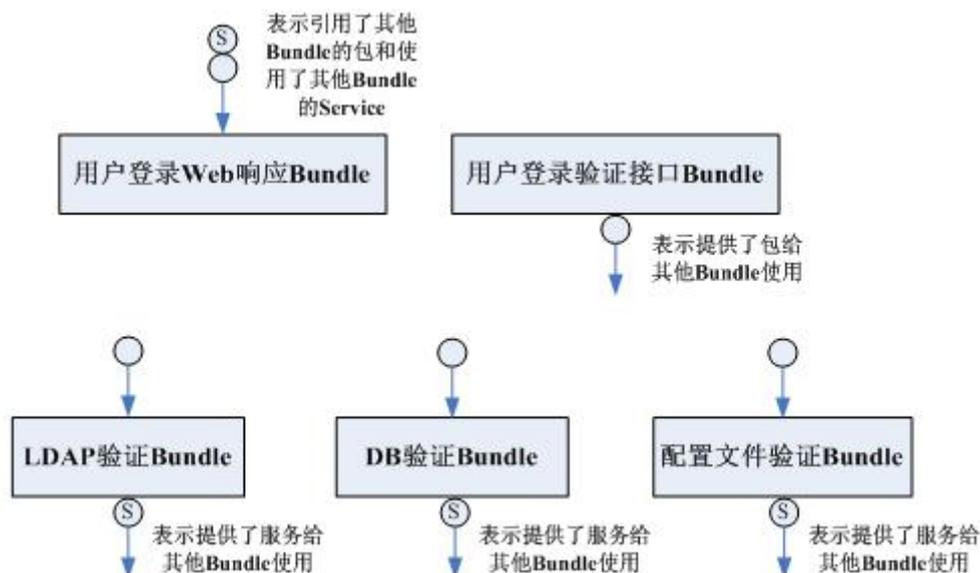
Bundle 其实就是一个 jar 文件，这个 jar 文件和普通的 jar 文件唯一不同的地方就是 Meta-inf 目录下的 MANIFEST.MF 文件的内容，关于 Bundle 的所有信息都在 MANIFEST.MF 中进行描述，说的时髦点，可以称它为 bundle 的元数据，这些信息中包含有象 Bundle 的名称、描述、开发商、classpath、需要导入的包以及输出的包等等，在后续的开发 Bundle 中将会详细的介绍 Bundle 的元数据以及如何去开发 Bundle。

Bundle 通过实现 BundleActivator 接口去控制其生命周期，在 Activator 中编写 Bundle 启动、停止时所需要进行的工作，同时也可以可以在 Activator 中发布或者监听框架的事件状态信息，以根据框架的运行状态做出相应的调整，但同时要注意，如果应用是被类似才用 Ctrl+C 等方式强行终止的话，那么 Activator 中的 stop 方法是不会被调用的。

Bundle 是个独立的概念，在 OSGI 框架中对于每个 Bundle 采用的是独立的 classloader 机制，这也就意味着不能采用传统的如引用其他 Bundle 的工程来实现 Bundle 间的协作了，那么在 OSGI 框架中 Bundle 之间是怎么协作的呢，在 OSGI 框架中对于每个 Bundle 可以定义输出的包以及引用的包，这样在 Bundle 间就可以共享包中的类了，尽管这样也可以直接实现简单的 Bundle 的协作，但在 OSGI 框架中更加推荐的是采用 Service 的方式，Service-Oriented 的概念(例如 SOA)大家都接触多了，OSGI 框架也同样是如此的，每个 Bundle 可以通过 BundleContext 注册对外提供的服务，同时也可以通过 BundleContext 来获得需要引用的服务，采用 Service-Oriented 的方式可以使得对外提供的服务能够更加的封闭，不需要为了使用别的 Bundle 提供的 Service 而做环境依赖等的设置，同时，Bundle 还可以采用 Require-Bundle 的方式来直接引用其他的 Bundle(相当于引用其他 Bundle 的工程或 jar)，在后续的开发、发布和使用 Service 中将会详细的介绍如何去开发、部署、使用 Service。

好，在对 OSGI 中的核心概念有了了解后，摩拳擦掌，开始来设计下基于 OSGI 框架如

何实现用户登录验证模块，由于目前Declarative Services²的实现尚处于测试阶段，在这里先采用Bundle控制的方式来实现用户登录验证模块，按照体验OSGI中的描述，我们将用户登录验证模块分为如下五个Bundle来实现：



图表 1 用户登录验证模块 Bundle 设计示意图

- 用户登录 Web 响应 Bundle

提供输入登录所需信息的页面，接受用户的登录请求，从 BundleContext 中获取用户登录验证的 Service，造型为用户登录验证接口，调用验证方法得到用户是否允许登录的许可，相应的返回结果至页面。

- 用户登录验证接口 Bundle

对外提供用户登录验证接口。

- LDAP 验证 Bundle

以 LDAP 验证方式实现用户登录验证接口，对外提供用户登录验证服务。

- DB 验证 Bundle

以 DB 验证方式实现用户登录验证接口，对外提供用户登录验证服务。

- 配置文件验证 Bundle

以配置文件验证方式实现用户登录验证接口，对外提供用户登录验证服务。

6.2. 工具箱

设计完毕了，是不是急着开始动手了呢，稍微忍耐下，不能空着手就开始呀，得先把工具箱里的工具准备好，好，下工具去。

² 详见Declarative Services

既然是基于 Equinox，自然要先下 Equinox，但由于 Equinox 是 Eclipse 的工程，而且 Eclipse 3.1 以后的版本都是通过它来启动的，这也就意味在 Eclipse 3.1 以后的版本其实本身就包含了 Equinox 的，所以如果你采用的是 Eclipse 3.1 以后的版本，那么就省事了，在你的 Eclipse 的 plugins 目录下你可以找到类似 org.eclipse.osgi_3.2.0.v20060510.jar 这样的文件，它其实就是 Equinox 的 OSGI R4 Core Framework 的实现了，如果你采用的不是 Eclipse 3.1 以后的版本，由于我们在开发基于 Equinox 的应用时采用的也是 Eclipse，那么还是建议你去下个 Eclipse 3.1 以后的版本。

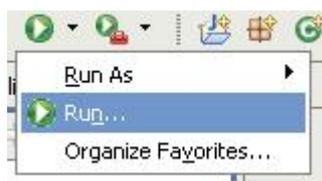
由于用户登录验证模块是个web性质的模块，我们得使用OSGI R4 中定义的Http Service，Equinox目前已实现了这个，到

<http://download.eclipse.org/eclipse/equinox/drops/S-3.2RC7-200606021317/index.php>上下载org.eclipse.equinox.http_1.0.0.v20060601a.jar 以及

org.eclipse.equinox.servlet.api_1.0.0.v20060601.jar，这是到目前为止Equinox对于Http Service实现的最新的两个相关的jar，如果在看到这篇文档时已经有更新的版本，那么可以去Equinox的最新版本中下载这两个jar包，下完这两个包后，把这两个包放到Eclipse的plugins目录下。

OK，启动Eclipse³，来检查下工具箱有没有准备好：

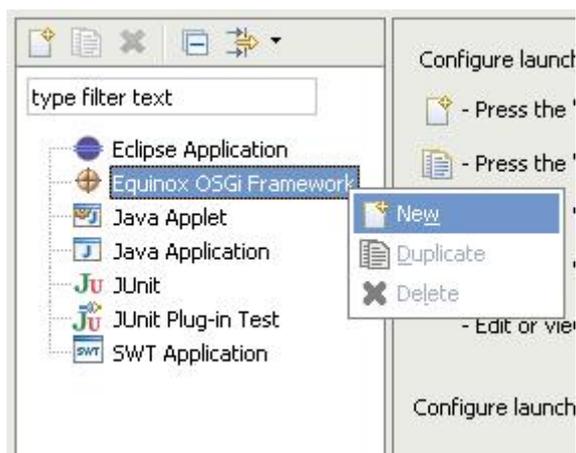
- 第一步



图表 2 工具箱—第一步

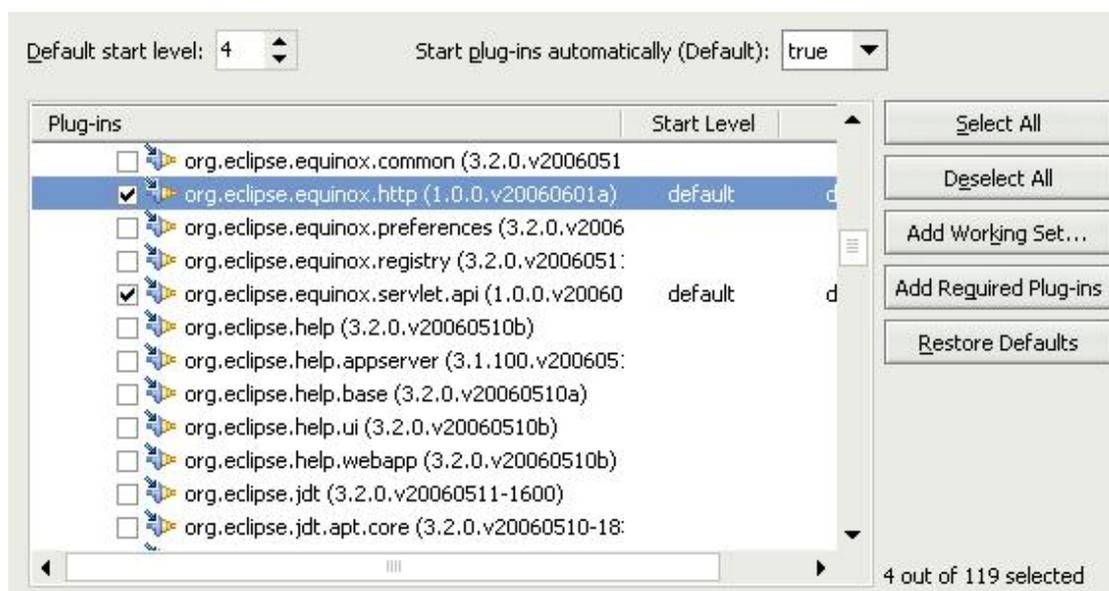
- 第二步

³ 笔者使用的Eclipse为 3.2 RC5



图表 3 工具箱—第二步

- 第三步

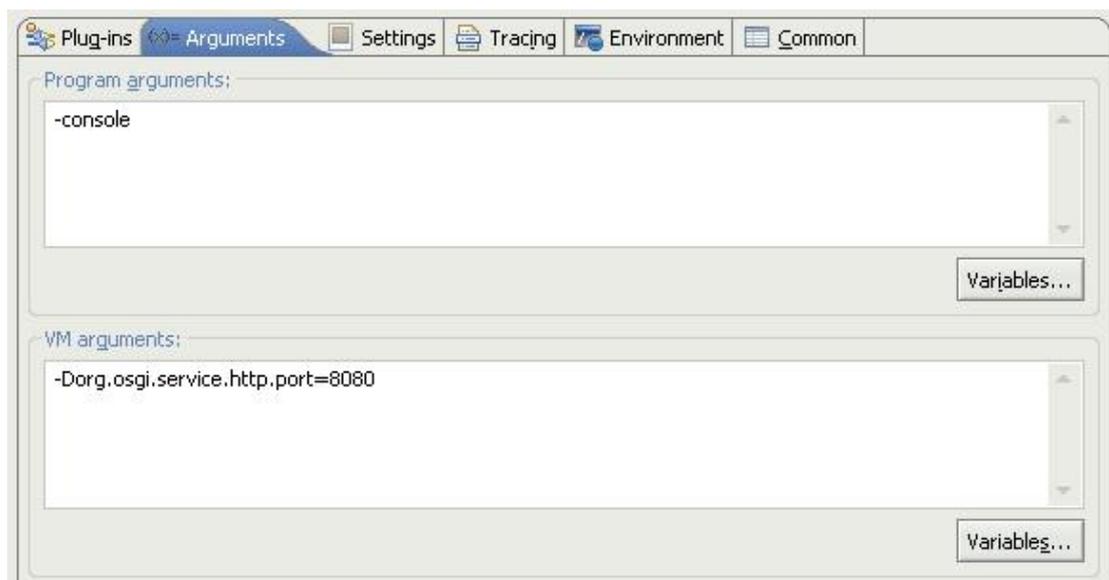


图表 4 工具箱—第三步

选中上图中的 `org.eclipse.equinox.http`、`org.eclipse.equinox.servlet.api`、`org.eclipse.osgi` 以及 `org.eclipse.osgi.services`。

- 第四步

点击上图中的 Run，如 Console 中看到 `osgi>` 而且没有错误信息，则表明工具箱配置已成功了，而如果看到类似 `Address already in use: JVM_Bind` 则说明本机的 80 端口已被占用，由于 Equinox 的 Http Service 实现默认使用的是 80 端口，所以会报这个错，那么我们可以通过在这里指定 Http Service 所使用的端口(进入第二步的那个窗口)：



图表 5 工具箱—第四步

在 VM arguments 中填入这样的一行 `-Dorg.osgi.service.http.port=8080`，后面的 8080 可以指定为你想要设定的端口，设置好后点 Run，是不是正常的在 Console 中看到了 `osgi>`，恭喜恭喜。

- 第五步

经过上步后整个工具箱的工具其实已经准备好了，最后可以验证下：

在 console 的 `osgi>`后输入 `ss`，回车，是不是看到类似这样的界面呢：

```
Console x
Equinox [Equinox OSGi Framework] C:\j2sdk1.4.2_03\bin\javaw.exe (2006-6-8 下午09:47:17)

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE    system.bundle_3.2.0.v20060510
1       ACTIVE    org.eclipse.equinox.http_1.0.0.v20060601a
2       ACTIVE    org.eclipse.equinox.servlet.api_1.0.0.v20060601
3       ACTIVE    org.eclipse.osgi.services_3.1.100.v20060511

osgi>
```

图表 6 工具箱—第五步

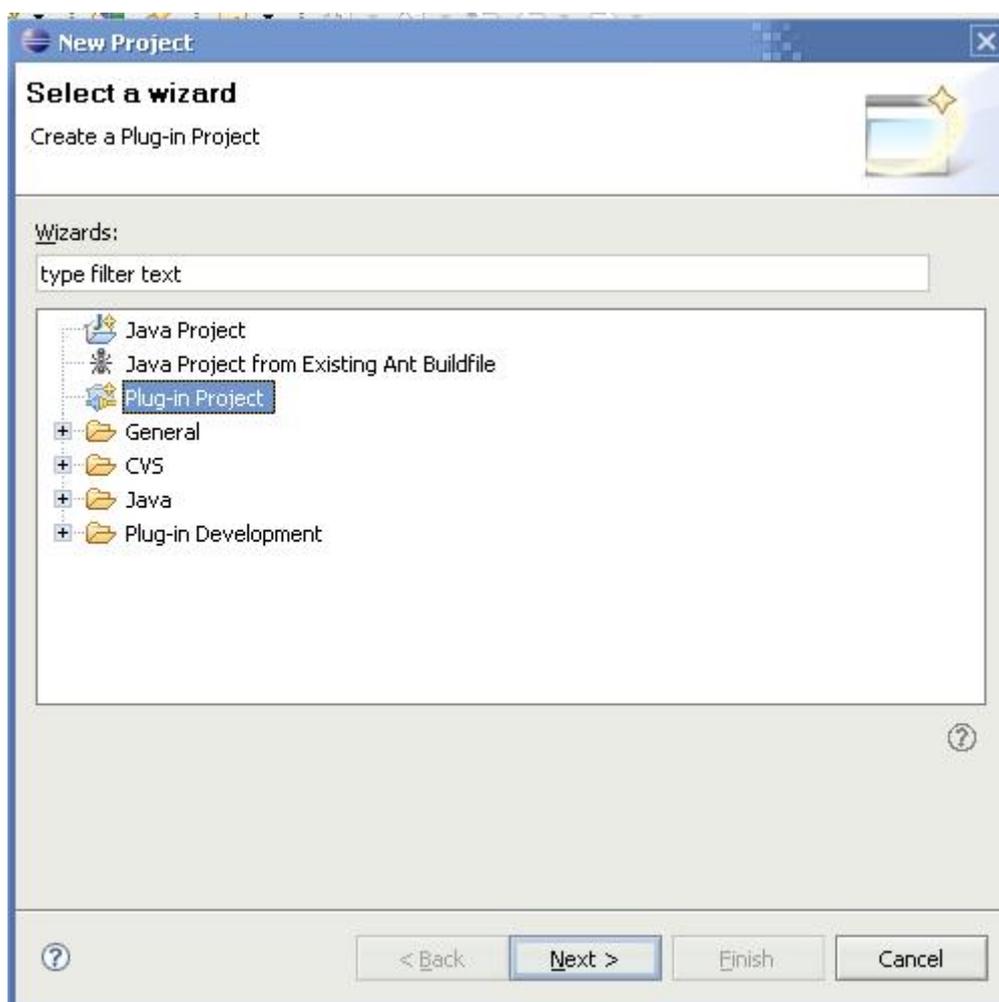
既然用了 `http service`的实现，那么我们应该可以通过 `web`去访问，OK，到 `web`浏览器中输入 <http://localhost:8080>，后面的 8080 替换为你自己的端口，应该会出现找不到网页而不是找不到服务器的提示。

经过上面的五个步骤，可以确定工具箱中的工具已经准备好了。

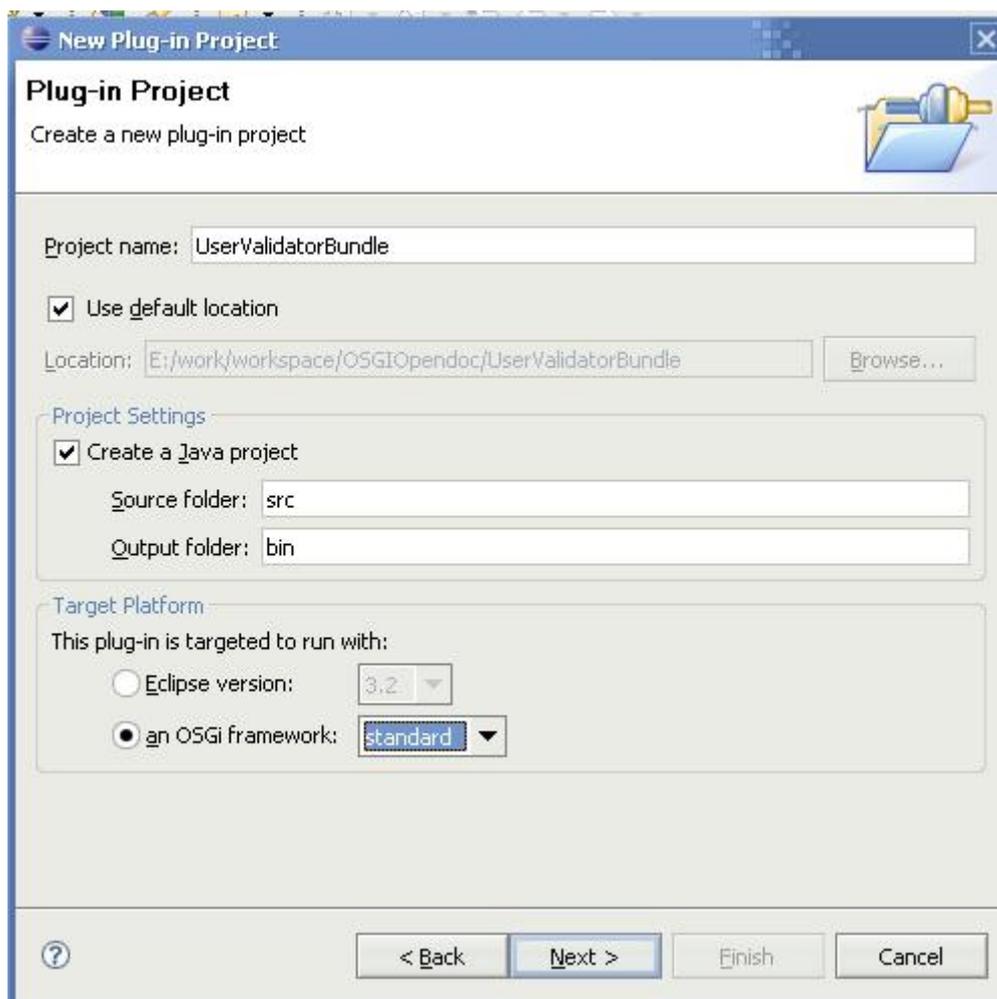
6.3. 开发 Bundle

一切准备工作完毕，开始 OSGI 实战之旅，首先来开发我们的第一个 Bundle，按照设计，根据依赖性，首先来做用户登录验证接口 Bundle：

- 第一步：建立 Bundle 工程
 - 在 Eclipse 中通过建立 Plug-in 工程来建立 Bundle 工程；



- 输入工程的相关信息，这里和建立普通 JAVA 工程唯一不同的就是需要选下 this plug-in is targeted to run with，在这里选择 an OSGI Framework 的 standard 选项，也就是说建立的是标准的 OSGI Bundle 工程；



■ 输入 Bundle 的相关元数据信息;

Plug-in ID 指的是 Bundle 的唯一标识, 在实际的项目中可以采用类似 java 的包名组织策略来保证标识的唯一性;

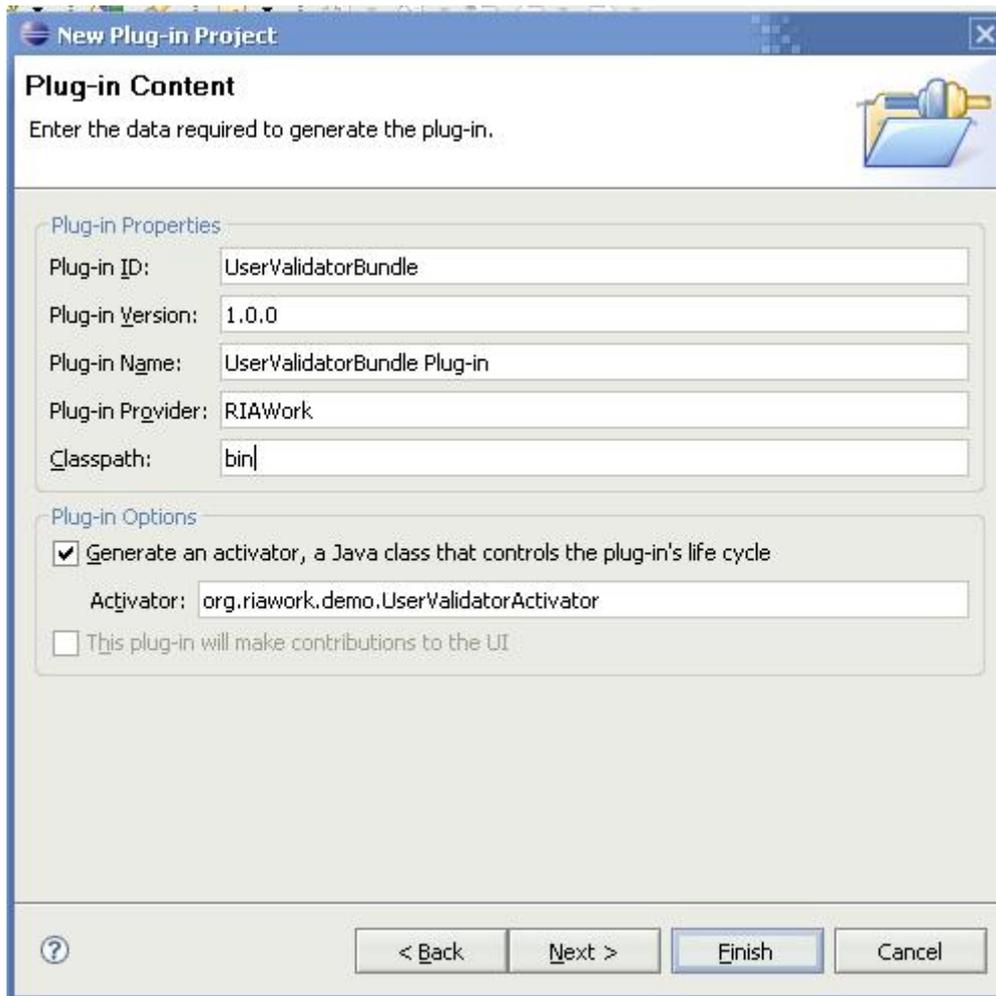
Plug-in Version 指的是 Bundle 的版本;

Plug-in Name 指的是 Bundle 的更具有意义的名称;

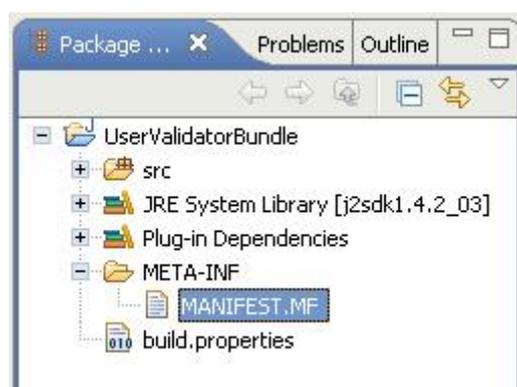
Plug-in Provider 指的是 Bundle 的提供商;

Classpath 指 Bundle 运行时的类路径, 由于在建立工程时 Output folder 设置的 bin, 在 classpath 这里同样设置为 bin;

剩下的最关键的就是 Activator 这个部分了, 这里填入自己的一个类名就可以了, 在工程建立时 Eclipse 会自动的建立这个类。



- 完成 Bundle 工程的建立，在 package 视图中可以看到类似这样的视图，表明工程建立成功了。



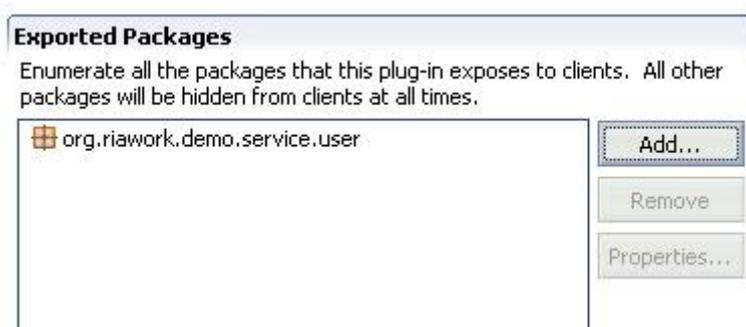
- 第二步：对外提供用户验证接口 package
 - 建立一个 Validator Interface;

```
Validator.java X
/*
 * RIAWork.org
 *
 * OSGI Opendoc Demo
 */
package org.riawork.demo.service.user;

/**
 * desc: 用户登录验证服务接口
 *
 * @author bluedavy
 */
public interface Validator {

    /**
     * 根据用户名和密码验证用户是否能够登录
     *
     * @param username
     * @param password
     * @return boolean
     * @throws Exception
     */
    public boolean validate(String username,String password) throws Exception;
}
```

- 对外提供用户验证接口 package，在之前的介绍中已经提及在 OSGI 框架中通过 Export-package 元数据来标识 Bundle 对外提供的 package，双击 META-INF 下的 MANIFEST.MF，选择其中的 Runtime 标签项，在 Exported Packages 中点击 add，在弹出的窗口中选择所建立的 Validator 接口所在的 package: org.riawork.demo.service.user，保存就完成这步了。



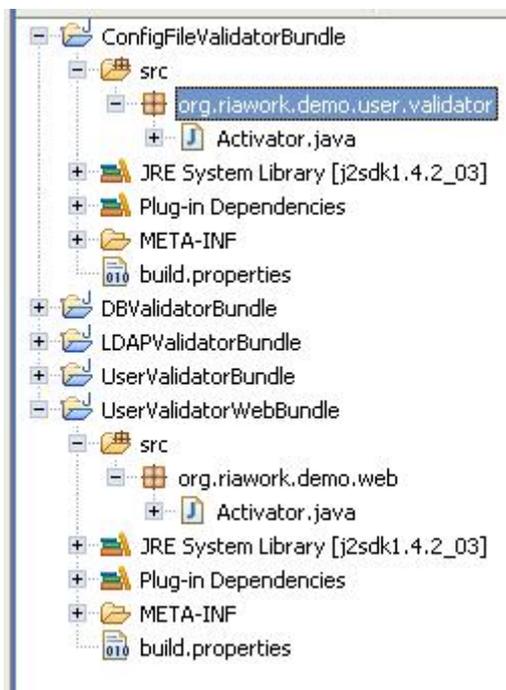
在完成后可以去看看 MANIFEST.MF 文件的内容，会看到其中有这项：

```
Export-Package: org.riawork.demo.service.user
```

由于用户登录验证接口 Bundle 在启动时和停止时都不需要做什么处理，因此无需对 UserValidatorActivator 做什么改动，经过上面两个步骤后，就完成了用户登录验证接口 Bundle 的开发了，是不是挺简单的呢，可以看到在 Bundle 的开发中与普通 Java 工程的开发基本相似，不同点在于需要使用

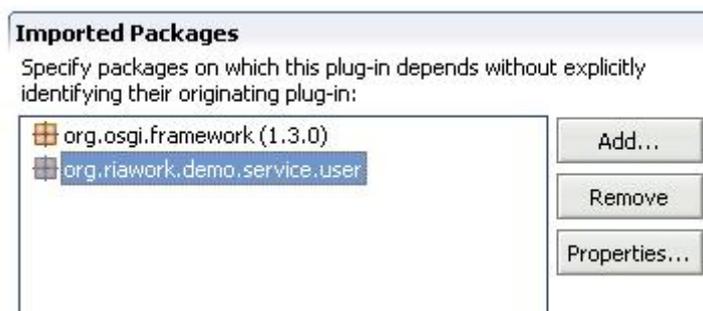
Bundle 的元数据信息，OK，继续来完成其他几个 Bundle 的开发。

按照之前建立 Bundle 工程的方式分别建立 LDAPValidatorBundle、DBValidatorBundle、ConfigFileValidatorBundle 和 UserValidatorWebBundle，建立后如下图所示：



- LDAPValidatorBundle

- 第一步：在 Bundle 元数据中引入用户验证接口的包，这样在 Bundle 中才可去实现接口，这里可以看出 Bundle 开发和传统的方式不同的地点，在传统的方式下就需要引用这个接口的 jar 文件或者引用接口所在的工程才行。



- 第二步：编写实现接口的类，鉴于本文是演示性质，在这里固定了登录的用户名和密码。

```
LDAPValidatorImpl.java x
+ * RIAWork.org
package org.riawork.demo.service.user.impl;

import org.riawork.demo.service.user.Validator;

/**
 * desc: LDAP方式的实现
 *
 * @author bluedavy
 */
public class LDAPValidatorImpl implements Validator{

    /*
     * (non-Javadoc)
     * @see org.riawork.demo.service.user.Validator#validate(java.lang.String, java.lan
     */
    public boolean validate(String username, String password) throws Exception {
        System.out.println("LDAP验证方式");
        if(("jerry".equals(username)) && ("bluedavy".equals(password)))
            return true;
        return false;
    }
}
```

- DBValidatorBundle

按照 LDAPValidatorBundle 方式同样实现。

- ConfigFileValidatorBundle

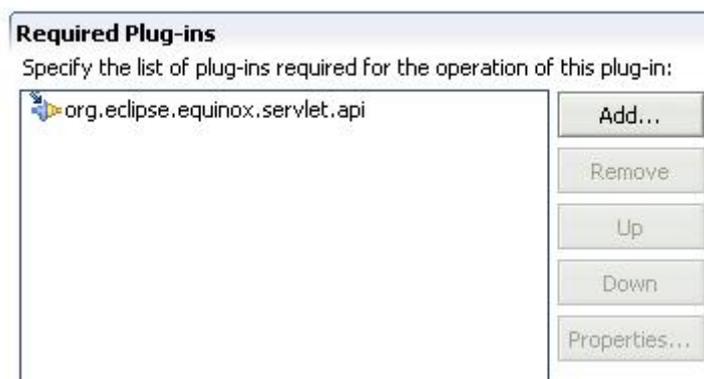
按照 LDAPValidatorBundle 方式同样实现。

- UserValidatorWebBundle

和传统的 Web 开发方式不同，由于 OSGI 框架中并没有象 web 应用服务器那样的 Bundle，就不能象 web 应用直接部署到 web 服务器那样简单了，需要通过 HttpService 将 Servlet 以及资源文件(象图片、css、html 等)进行注册，这样才可访问了，详细在后续的 HttpService 中介绍，在这里暂时就不提及 Service 的使用方法，先搭建好架子，鉴于这是个 Demo，就采用简单的方式实现了，这个 Bundle 由登录的首页面以及响应的 Servlet 构成。

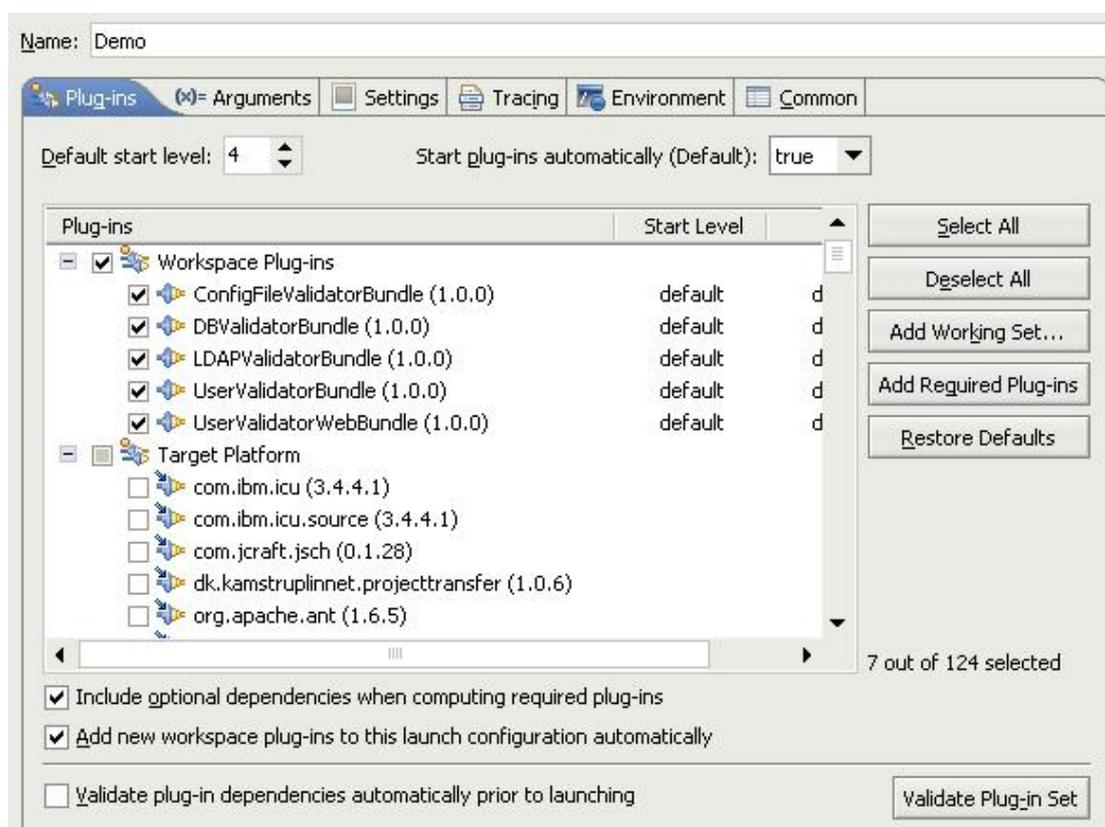
- 在 src 目录下建立一个 page 的目录，在其中编写 login.htm;

- 登录响应的 Servlet，由于 Servlet 需要继承 HttpServlet，需要引用下 Servlet API，Equinox 在实现 Http Service 时为了大家使用方便，提供了一个 Servlet API 的 Bundle，在准备工具箱时我们已经下了这个 Bundle 了，那么现在我们就可以通过这样来引用 Servlet API:

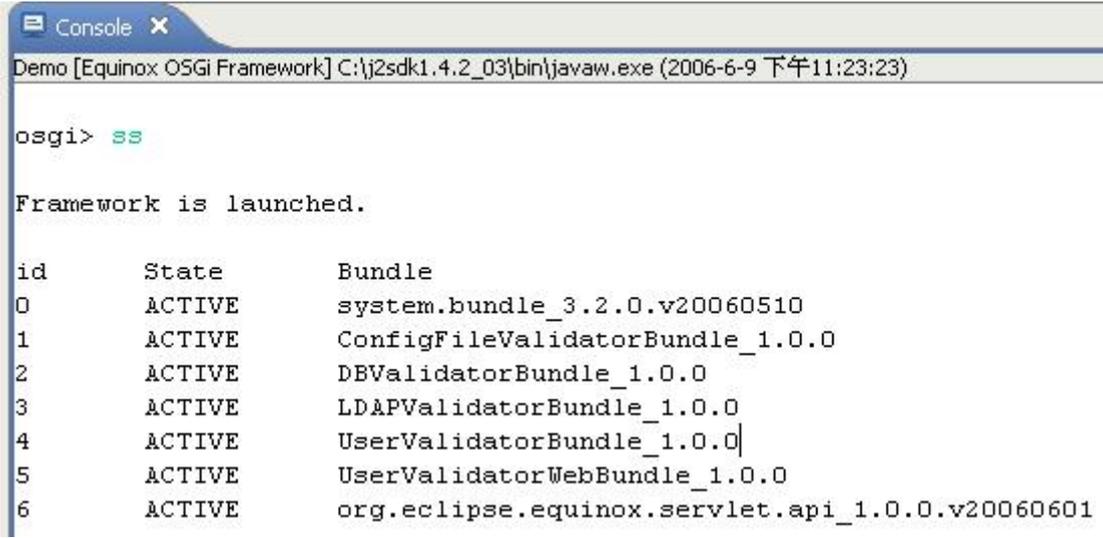


这样之后就可以开始来编写登录响应的 Servlet, 写法和普通的 Servlet 没有任何区别, 具体见附件中的代码。

经过这些步骤后, 用户登录验证模块所需要的 Bundle 就全部搭建好了, 启动一下看看, 点 Eclipse 的 Run..., 新建一个 Equinox OSGI Framework 的运行工程:



默认已经选中了 workspace 下的 Plugin 工程, 点击旁边的 Add Required Plug-ins, 保存这个配置 (如需要更改启动时的 web 端口, 仍然按照之前在准备工具箱时的方法), 点击运行, 启动正常后可以通过在 osgi>后输入 ss 回车:



```
Console X
Demo [Equinox OSGi Framework] C:\j2sdk1.4.2_03\bin\javaw.exe (2006-6-9 下午11:23:23)

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       ACTIVE     ConfigFileValidatorBundle_1.0.0
2       ACTIVE     DBValidatorBundle_1.0.0
3       ACTIVE     LDAPValidatorBundle_1.0.0
4       ACTIVE     UserValidatorBundle_1.0.0
5       ACTIVE     UserValidatorWebBundle_1.0.0
6       ACTIVE     org.eclipse.equinox.servlet.api_1.0.0.v20060601
```

OK, 到此, 用户登录验证模块的 Bundle 开发完成了, 经过这些学习后, 可以看出 Bundle 的开发并不复杂, 和做普通的 java 工程开发唯一不同的是借助使用 Bundle 的元数据来实现工程之间 package 的共享, 接下来我们开始学习 Service 的开发、发布以及使用, 来完成用户登录验证模块的开发。

6.4. 开发、发布和使用 Service

从开发角度来看, Service 有点象虚拟的概念, 因为在编写 Service 时和编写普通的 Java 类(POJO)没有任何的区别, 其实在上面已经写过三个 Service 类了, 分别是 LDAPValidatorImpl、DBValidatorImpl 以及 ConfigFileValidatorImpl。

在 OSGI 框架中, Service 是个实际的概念, 只有通过 BundleContext 注册成 Service 才能使得一个 POJO 作为 Service 在 OSGI 框架中被使用, 同时也只有通过 BundleContext 来获取发布到框架中的 Service, 通过 Service 的方式来实现 Bundle 之间实例级的依赖, 和 Import-Package、Require-Bundle 不同的地方在于通过 Service 获取的是其他 Bundle 中类的实例。

来看看怎么样去发布之前写好的几个 Service:

- LDAP 验证 Bundle

打开之前在搭建 LDAP 验证 Bundle 时的 Activator 文件, 可以看到 start 和 stop 两个方法, 这两个方法就是用来管理 Bundle 的生命周期的, 我们需要在 Bundle 启动的时候注册需要发布的 Service, 在 Bundle 停止的时候卸载发布的 Service, 在 OSGI 框架中通过调用 BundleContext 来注册 Service, 方法是这样的:

```
context.registerService(服务标识名,服务实例,服务实例属性);
```

方法返回的是 ServiceRegistration, 可以通过返回的这个 ServiceRegistration 来卸

载这个 Service，在 stop 方法中通过这样的方法来卸载注册的这个 Service：

```
serviceRegistration.unregister();
```

通过这样两个方法后就完成了 Service 的注册和卸载，代码详见附件。

- DB 验证 Bundle

和 LDAP 验证 Bundle 同样的完成 Service 的注册和卸载。

- 配置文件验证 Bundle

和 LDAP 验证 Bundle 同样的完成 Service 的注册和卸载。

Service 的发布写好了，开始来使用 Service，根据之前的设计，在这个用户登录验证模块中，只有 UserValidatorWebBundle 需要使用到其他 Bundle 提供的 Service，那么就来完成 UserValidatorWebBundle：

- 首先完成之前的登录响应的 Servlet，在这个 Servlet 中需要通过 BundleContext 获取用户登录验证服务，在这个 Servlet 增加一个构造器，以供外部在实例化时传入 BundleContext；在 doGet 方法中增加获取用户登录验证服务的代码，并将验证的结果展现给用户，在 OSGI 框架中通过这样的方法来获取服务：

```
ServiceReference serviceRef=context.getServiceReference(服务标识名);  
Object service=context.getService(serviceRef);
```

获取到用户登录验证服务实例后，通过调用验证方法获取到用户登录验证的结果，由于目前 Http Service 尚不支持 jsp，在这里就直接在 servlet 中把验证的结果写入 response 返回。

经过这样一个步骤后就完成了登录响应 Servlet 的编写。

- 接着来完成用 HttpService 来将实例化的登录响应的 servlet 以及登录页面的资源文件注册到 Web 应用中，和传统 web 应用开发不同，这里的 servlet 的实例化是由开发人员来操作的，当然，其实也很简单，就是直接的：

```
Servlet servlet=new LoginServlet(context);
```

由于要使用到 HttpService，需要首先在 import-package 导入 org.osgi.service.http 包，对于 HttpService 采用监听的方式来获取，也就是说，当监听到 HttpService 可用时，则使用这个 HttpService 来将 Servlet 和资源文件注册，当 HttpService 不可用时，则将 servlet 和资源文件卸载，在 OSGI 框架中，要监听服务的状态的方式也很简单，通过实现 ServiceListener 接口中的 serviceChanged 方法以及

将监听器实例注册到 `BundleContext` 中即可实现，通过 `BundleContext` 将 `ServiceListener` 注册采用的是这个方法：

```
context.addServiceListener(ServiceListener 实例,"(objectClass=服务标识名)");
```

`HttpService` 通过这样的方法来注册 `Servlet`：

```
HttpService.registerServlet(Servlet Mapping Url,Servlet 实例,配置信息,HttpContext 实例);
```

示例：

```
service.registerServlet("/demo/login", servlet, null, null);
```

`HttpService` 通过这样的方法来注册资源文件：

```
HttpService.registerResources(资源 Mapping Url,资源文件路径, HttpContext 实例);
```

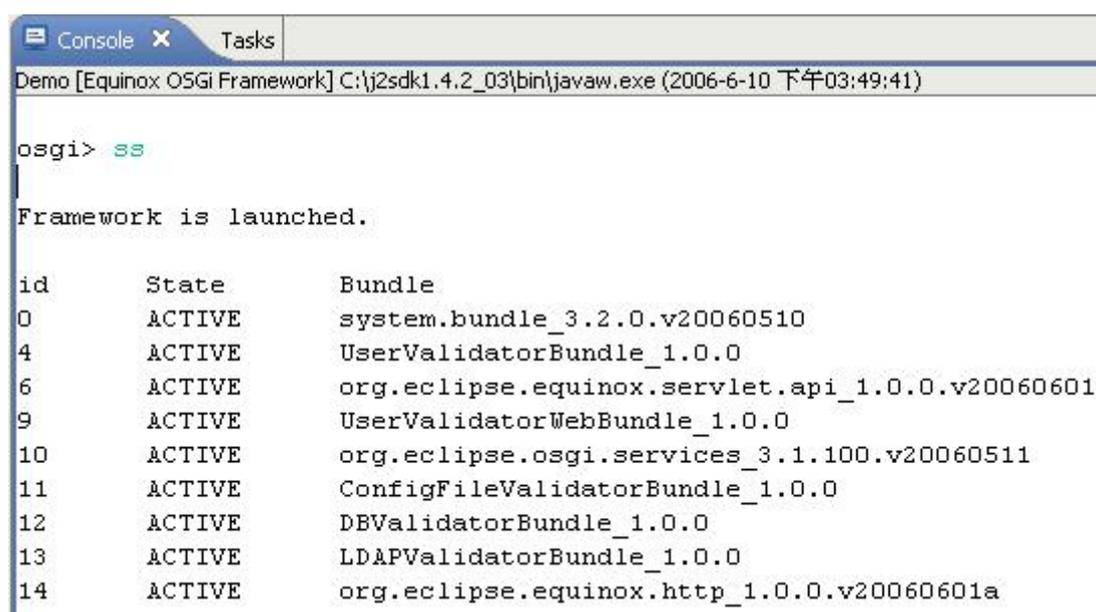
示例：

```
service.registerResources("/demo/page","page",null);
```

具体请参见代码。

经过这样两步后就完成了 `UserValidatorWebBundle` 的开发了。

最后按照之前的方法，在 Eclipse `Run...`中重新点 `Add Required Plugins`，之后再选中其中的 `org.eclipse.equinox.http`，然后点击 `Run`，同样，当在 `console` 中看到 `osgi>`则表明启动成功，再在 `osgi>`后输入 `ss`，看看是不是类似如下的画面：



```
osgi> ss
Framework is launched.
id      State      Bundle
0       ACTIVE    system.bundle_3.2.0.v20060510
4       ACTIVE    UserValidatorBundle_1.0.0
6       ACTIVE    org.eclipse.equinox.servlet.api_1.0.0.v20060601
9       ACTIVE    UserValidatorWebBundle_1.0.0
10      ACTIVE    org.eclipse.osgi.services_3.1.100.v20060511
11      ACTIVE    ConfigFileValidatorBundle_1.0.0
12      ACTIVE    DBValidatorBundle_1.0.0
13      ACTIVE    LDAPValidatorBundle_1.0.0
14      ACTIVE    org.eclipse.equinox.http_1.0.0.v20060601a
```

这个时候可以看到 `ConfigFileValidatorBundle`、`DBValidatorBundle`、

LDAPValidatorBundle都启动了⁴，如果想现在采用LDAP方式来验证用户的登录，那可以通过stop命令来停止另外两个Bundle的运行：

```
stop 11
stop 12
```

好，通过 web 来访问下用户登录验证模块：



输入用户名和密码后，点击登录，看看是不是会进入一个提示登录结果的页面：



再看看 console 中的提示信息：osgi> LDAP 验证方式

好，这个时候再来动态切换下登录的验证方式，将 LDAP 验证方式切换为 DB 验证方式：

```
start 12
stop 13
```

再登录下，去看看 console 中的提示信息：osgi> DB 验证方式

经过了这些步骤后，完整的实现了用户登录验证模块的需求，当要增加新的验证方式时，只需要按照之上编写 LDAPValidatorBundle 的方法同样编写即可。

在对开发、发布和使用Service进行学习后，估计大家会觉得Service的注册和使用过于复杂，确实如此，可能大家都会想为什么不采用DI的方式来解决这个问题，在OSGI R4 中提出了Declarative Services来解决这个问题，Declarative Services不仅仅支持DI方式，还推出了更为完整的Service-Oriented Component Model，在之后的**Declarative Services**章节中将会详细介绍，并重新实现Service的注册和使用，到时会看到一个更好用且更加支持动态策略的模型。

⁴ 在后续StartLevel Service将介绍如何设定StartLevel来控制哪些Bundle启动，哪些不启动

6.5. 测试和调试

对于测试的支持无疑是现在对于框架的重要考评点，那么来看看基于 OSGI 框架的系统怎么做测试呢，由于做集成测试的方法只和系统的结构(B/S、C/S)有关，和框架没什么关系，所以在这里只谈基于 OSGI 框架的系统是如何做单元测试的。

按照正确的方法，应该是先写测试，由于需要先介绍下基于 OSGI 框架的开发，为了避免注意力的分散，所以就把测试的部分挪到这里专门讲解了。

编写单元测试时最重要的就两点：

- 设置测试类的依赖(通过 Mock、实例化等方法)；
- 测试在某种输入的情况下方法执行的输出是否和预期的结果一致。

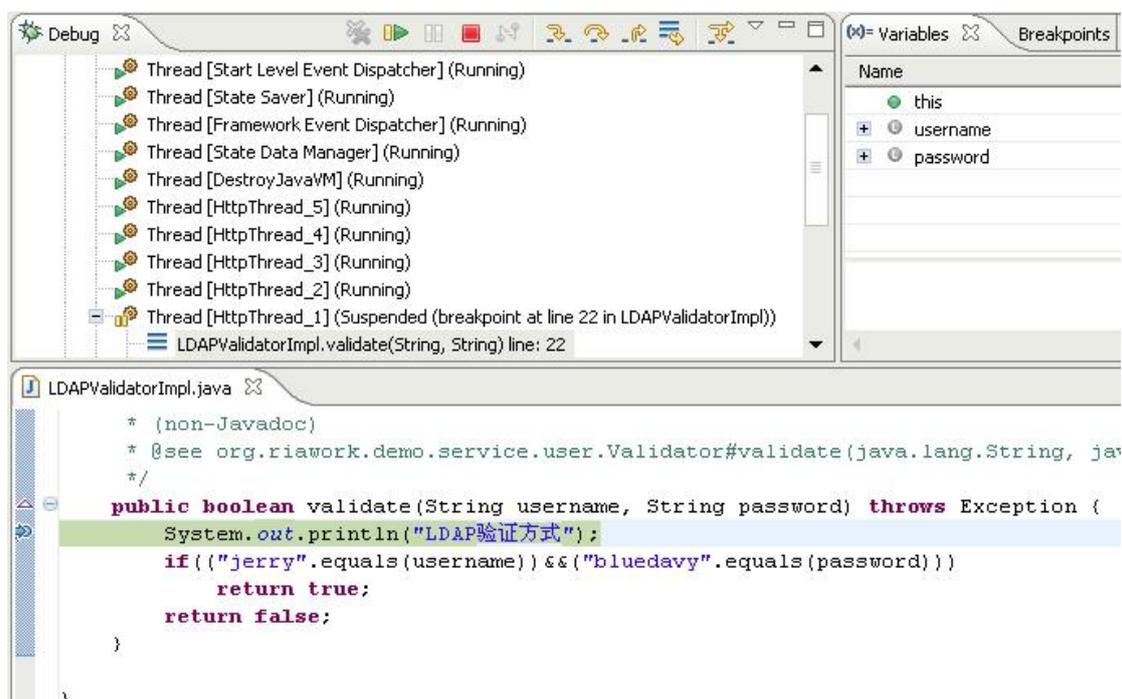
在做单元测试时最复杂的部分往往就是设置测试类的依赖，典型的就像 EJB 中的 session bean 的测试，由于它需要依赖 EJB Container，所以是比较麻烦的。

之前编写的几个 Service 没有依赖的情况，就只需要测试在某种输入的情况下方法执行的输出和预期的结果是否一致，这就很容易编写了，具体详见附件 classic 目录下的 LDAPValidatorImplTest、DBValidatorImplTest 以及 ConfigFileValidatorImplTest。

在用户登录模块这个例子中，最复杂的就是登录响应 Servlet 的测试，这个登录响应 Servlet 需要依赖 OSGI 框架的 BundleContext 获取用户登录验证服务，同时还需要依赖 HttpServletRequest 和 HttpServletResponse，在没法实例化类所依赖的环境时，只能采用 Mock 的方法来实现，代码见附件 classic 目录下的 LoginServletTest。

在这种情况下会发现基于OSGI框架的单元测试并不好做，这主要是因为之上的例子中对于服务的获取、注册都是采用BundleContext来完成的(也就意味着需要依赖OSGI框架， HttpServletRequest需要Mock和OSGI框架没什么关系)，在后续**Declarative Services**的章节中介绍采用DI方式来实现时，就不需要Mock BundleContext来获取服务了，到时会将测试这部分的代码进行重写。

调试也是关注的重点，由于可以在 Eclipse 中启动基于 Equinox 开发的系统，那么一切都不是问题，和普通 Java 工程进行调试的方法没有任何区别，设置断点，Debug，就 OK 了，在运行到断点对应的代码时就进入熟悉的调试视图了。



6.6. 发布基于 OSGI 的系统

经过上面的步骤，完成了用户登录验证模块的开发工作，一直我们都是通过 Eclipse 直接来启动这个系统的，但发布给用户的系统不可能让用户通过 Eclipse 来运行系统，那么如何发布一个单独运行的基于 OSGI 的系统呢？

- 第一步：建立独立的 Equinox 运行环境；

在硬盘上建立一个 Demo 目录，从 Eclipse/Plugins 目录下复制 org.eclipse.osgi_3.2.0.v20060510.jar 到此目录下，改名为 equinox.jar，编写一个 run.bat，其中内容为：

```
java -Dorg.osgi.service.http.port=8080 -jar equinox.jar -console
```

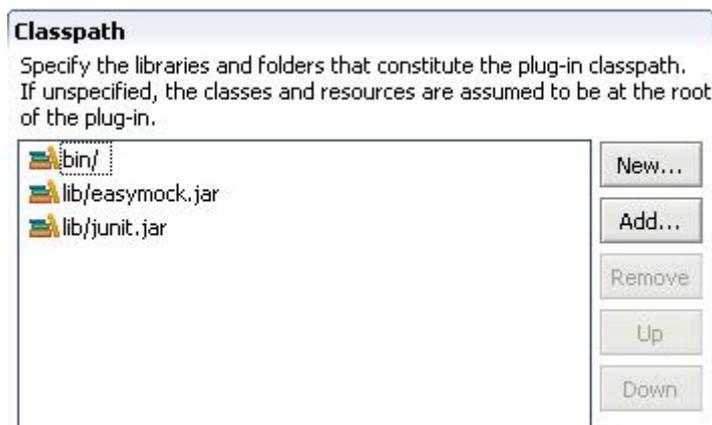
如果不需要指定 http 的端口的话，可以编写成：

```
java -jar equinox.jar -console
```

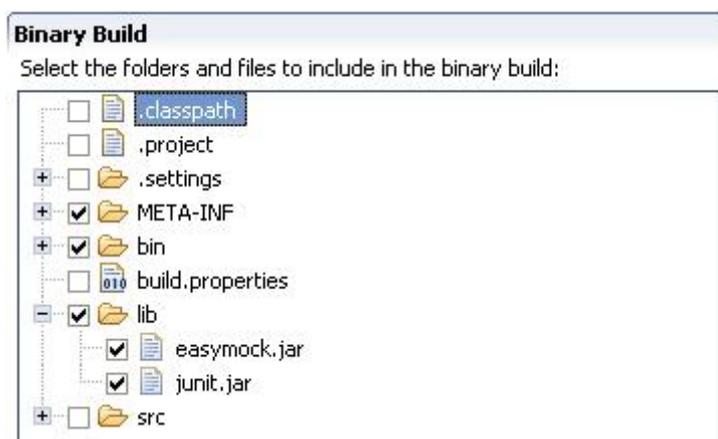
双击 run.bat，如看到 osgi>则表明启动成功了，输入 ss 回车，这个时候会看到列表中只有一个 system bundle。

- 第二步：导出各 Bundle 工程为 jar；

以最为复杂的 UserValidatorWebBundle 为例，首先打开 MANIFEST.MF，在里面 Runtime 标签项的 Classpath 中增加对于 lib 中 jar 包的引用：



打开工程里的 build.properties 文件，选中其中的 lib 目录：



选中 UserValidatorWebBundle 工程，右键，选择 Export，选中弹出页面中的 Deployable plug-ins and fragments：



在进入到窗口中的 Available Plug-ins and Fragments 中已经默认选中了 UserValidatorWebBundle，选中 Destination 标签项中的 Directory，browse，填写或选择导出的目录，点击 Finish，在导出的目录中可以看到 plugins 目录，其中有 UserValidatorWebBundle_1.0.0.jar 文件，这就是我们所需要的。

按照同样类似的方法导出其他的几个工程，在目前这个用户登录验证模块中会形成五个 jar 文件，也可以一次性的把五个工程全部导出。



- 第三步：安装各 Bundle 到 Equinox 中。

在用户登录验证模块还用到了 Equinox 提供的其他三个 Bundle: OSGI Services API、ServletAPI、HttpService, 在 demo 下建立一个 bundles 目录, 将第二步时生成的五个 Bundle 复制到 bundles 目录下, 同样再从 Eclipse/Plugins 目录中复制 org.eclipse.equinox.http_1.0.0.v20060601a.jar 、 org.eclipse.equinox.servlet.api_1.0.0.v20060601.jar 、 org.eclipse.osgi.services_3.1.100.v20060511.jar 到 bundles 目录中。

有两种方法将 Bundle 安装到 Equinox 中:

- 第一种

运行之前编写的 run.bat, 在 osgi>后输入 install reference:file:bundles/UserValidatorBundle_1.0.0.jar, 回车, 这样就完成了 UserValidatorBundle 的安装了, 输入同样的命令把 bundles 目录中的其他 Bundle 也安装上, 安装完毕之后, 在 osgi>后输入 start 1 回车, 之后相聚输入 start 2、start 3、start 4、start 5、start 6、start 7、start 8; 这样所有的 Bundle 就启动好了, 输入 ss 看看目前安装的 Bundle 的状态:

```
osgi> ss
Framework is launched.
id      State      Bundle
0       ACTIVE    system.bundle_3.2.0.v20060510
1       ACTIVE    ConfigFileValidatorBundle_1.0.0
2       ACTIVE    DBValidatorBundle_1.0.0
3       ACTIVE    LDAPValidatorBundle_1.0.0
4       ACTIVE    org.eclipse.equinox.http_1.0.0.v20060601a
5       ACTIVE    org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       ACTIVE    org.eclipse.osgi.services_3.1.100.v20060511
7       ACTIVE    UserValidatorBundle_1.0.0
8       ACTIVE    UserValidatorWebBundle_1.0.0
```

打开浏览器, 输入 <http://localhost:8080/demo/page/login.htm>, 看到了之前在 Eclipse 启动后同样的页面吧。

之后在 osgi>输入 exit, 回车后退出系统, 以后需要启动系统时只需要运行

run.bat 就可以了。

经过这些步骤后，就形成了单独运行的基于 OSGI 的系统。

■ 第二种

这种方法比较简单，在 Demo 目录下建立 configuration 目录，在其中放入 config.ini 文件，这个文件示例如下：

```
osgi.noShutdown=true
# 当前系统下运行的 Bundle，可以在此指定 Bundle 的启动顺序，在后续的
# StartLevel Service 章节中会详细的介绍
osgi.bundles=reference\;file\;bundles/ConfigFileValidatorBundle_1.0.0.jar@start,reference\;file\;bundles/DBValidatorBundle_1.0.0.jar@start,reference\;file\;bundles/LDAPValidatorBundle_1.0.0.jar@start,reference\;file\;bundles/org.eclipse.equinox.http_1.0.0.v20060601a.jar@start,reference\;file\;bundles/org.eclipse.equinox.servlet.api_1.0.0.v20060601.jar@start,reference\;file\;bundles/org.eclipse.osgi.services_3.1.100.v20060511.jar@start,reference\;file\;bundles/UserValidatorBundle_1.0.0.jar@start,reference\;file\;bundles/UserValidatorWebBundle_1.0.0.jar@start
osgi.bundles.defaultStartLevel=4
```

配置完毕后双击run.bat，就启动这个单独运行的基于OSGI的系统了，在后续的StartLevel Service章节中会详细的介绍config.ini文件。

当然，可以把上面的这些手动部署的步骤转为采用 ant 或 maven 编写成脚本来实现自动部署，但总体来说，目前发布单独运行的基于 OSGI 的系统还不是非常的方便，也许在不久的将来可以直接导出在 Eclipse Run...中配置的 OSGI 运行系统。

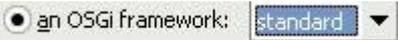
6.7. Equinox 基于 OSGI 的扩展

Equinox 除了完整实现 OSGI R4 规范以外，还吸取了 Eclipse 3.0 以前版本中那套插件框架中优秀的地方，如 Extension registry 机制、Eclipse Adapter 机制等，其中最重要的就是 Extension Registry，Extension Registry 是 Eclipse 插件机制中关键的部分，它为插件的扩展提供了一种实现的机制，Bundle 通过发布扩展点的方式来定义 Bundle 可扩展的部分，当需要扩展 Bundle 的时候只需要实现 Bundle 提供的扩展点的接口就可以了，通过这样的方式就可以完成 Bundle 的扩展了，在 Eclipse 中有很多地方采取了这种方式来扩展插件，如 Eclipse 中的菜单、视图等，都可以通过实现扩展点来增加

新的菜单、新的视图。

网上关于如何使用 Eclipse 扩展点的介绍非常的多, Equinox 的使用方法是基本相同的, 将 org.eclipse.equinox.registry Bundle 放入 Eclipse 的 Plugins 目录即可, 其他过程就和 Eclipse 扩展点的使用方法完全相同, 在此就不再重复去写了。

Equinox 作为 OSGI R4 的 RI, 其对于 OSGI 的扩展一定程度上会对 OSGI R5 产生影响, 但在目前的阶段使用的时候要注意, 如果开发要保证符合 OSGI 标准, 那么就on不要使用 Equinox 对于 OSGI 的扩展, 这个在使用 Eclipse 进行开发时可以指定仅使用标准的

OSGI 框架。 

6.8. 现有类型系统基于 OSGI 的开发

基于 OSGI 进行系统开发的时候和传统的开发方式有所不同, 来看看对于 B/S、C/S 以及嵌入式系统基于 OSGI 怎么去开发。

6.8.1. B/S

例子中的用户登录验证模块是个典型的 B/S 结构的系统模块, 采用 HttpService 来作为实现 B/S 结构的系统是基于 OSGI 开发 B/S 结构系统的方法之一, 不过 HttpService 对于 B/S 结构的系统来说支持的还是不够, 例如目前 B/S 结构中通常使用的 jsp、传统的 MVC 框架的嵌入等等, 对于 jsp 的支持已发布, 目前正在测试阶段中, 使用 HttpService 适合用于开发较为简单的 B/S 应用, 不过以我目前的应用情况来看, 我觉得要支持较为复杂的 B/S 应用也是可以的, 在我目前的产品的 B/S 部分采用的是一个自主开发的简单的 MVC Framework+Velocity 的结构来开发的, 至于到了后台则可以用大家熟悉的各种后台结构方法来实现, 采用这种方式的好处是无需应用服务器。

6.8.1.1. 基于 Bridge 方式开发 B/S 应用

而对于需要使用应用服务器特性的应用来说, 基于 Bridge 来开发 B/S 应用就是另外一种可选择的方法了, 在使用这种方法的情况下就可以按照传统的 B/S 开发方式将系统打包为一个 war 包放入应用服务器下运行, 目前 Bridge 方式还处于 Equinox 的 server-side incubator 中, 根据 maillist 的情况来看, 主要是 IBM 的人现在在试用这个东西, 在 Equinox 网站上提供了一篇如何使用 bridge.war 来实现这种开发方式的 [指导文章](#), 在这里我就只简单的说说了, 大家感兴趣的话可以把例子中的用户登录验证模块改为部署到 bridge.war 里。

bridge.war的使用还是比较简单的,不过目前确实存在着一些bug,从[指导文章](#)这个页面中可以找到bridge.war的下载,下载完毕后将它部署到应用服务器(Tomcat、Jetty、Jboss、Websphere、Weblogic等)中,启动应用服务器,启动完毕后可以直接在应用服务器的console中直接操作OSGI框架的管理console了,可以输入已经熟悉的ss、start、stop、install等命令,通过这样的方式就可以把Bundle部署到Bridge这个Web应用中了,在[指导文章](#)页面中还可以找到sample.http和sample.http.registry两个例子,这两个例子向我们展示了如何增加新的servlet、资源文件(页面文件、图片、js等)到web应用中去,两个例子功能相同,但采用的方法不同,一个是采用我们已经熟悉的HttpService去注册新的servlet以及资源文件的方法,另一个则是采用Equinox对OSGI增强的扩展点的方法,通过扩展点的方式的好处在于无需通过HttpService来注册Servlet和资源文件,从这个例子中也可以简单的看出如何在Equinox中使用扩展点,需要注意的是在使用这个扩展点的Bundle的例子时必须保留其中的plugin.xml,并且该Bundle需要先安装到OSGI框架中再重新启动方能生效。

6.8.2. C/S

基于OSGI开发C/S结构的系统和开发传统的C/S结构系统没什么很多不同的地方,只需将系统中所有的模块编写为Bundle的方式进行部署,并通过启动OSGI框架来启动整个系统。

6.8.3. 嵌入式

在嵌入式结构的系统方面,OSGI是绝对的强项,OSGI的诞生就是为嵌入式系统提供支撑的,所以在嵌入式系统方面采用OSGI没有任何的问题。

综合对于B/S、C/S、嵌入式这三种体系结构系统的支持,可以看出对于C/S、嵌入式系统而言采用OSGI框架没有什么太多的问题,对于B/S结构的系统而言其实大部分情况下是可以通过采用HttpService来实现的,实在不行的话可以采用Equinox提供的server-side incubator中的bridge.war来实现,相信随着OSGI在Server-Side应用和企业应用中的不断发展,对于B/S结构应用的支撑将会越来越优秀。

6.9. 注意事项

在使用Equinox时,特别要注意的是需要使用classloader去加载资源文件的时候,由于每个Bundle拥有独立的classloader,而有些开源框架会采用使用顶级classloader去加载文件的现象,这个时候就会导致在equinox中运行时出错,如在equinox使用

spring 时，如配置文件中采用了 `classpath:file` 这样的方式去加载其他的配置文件的话，就会出现找不到文件的现象，这个错误就是由于 `classloader` 引起的，关于 `Classloader` 的问题在后续的章节中将会详细的进行讲解。

另外要注意的就是包的命名问题，如果引用了其他 `Bundle Export` 的 `package`，那么在当前 `Bundle` 中就不能再建同样的包了，举例说：

在 `LDAPValidatorBundle` 中引用了 `UserValidatorBundle Export` 的 `org.riawork.demo.service.user`，那么在 `LDAPValidatorBundle` 中就不能再采用这个包名来编写类了，可以看到附件中的实现 `Validator` 接口的代码的包名是 `org.riawork.demo.service.user.impl`，大家可以把这个包名重构为 `org.riawork.demo.service.user`，再次启动后会发现 `LDAPValidatorBundle` 的状态为 `RESOLVED`，而不是其他 `Bundle` 的 `ACTIVE` 状态，这个时候通过命令行 `start 3` 之类的方法启动 `LDAPValidatorBundle` 就会报找不到类的错误，在实际使用 `Equinox` 进行项目开发时要注意这个问题。

在使用 `Equinox` 时可能还会碰到这样那样的问题，在碰到问题的时候可以到 `equinox` 查找相关的文档，或问使用过 `equinox` 的同行们，或发邮件到 `equinox` 邮件组中进行询问。

七. 深入 OSGI

经过对 OSGI 框架使用的学习，相信大家对于 OSGI 中的概念有了或多或少的理解，同时也会在使用的过程中产生各种各样的疑问，在这个章节中将深入学习 OSGI 框架背后的思想，学习 OSGI 规范中是怎么定义 Bundle 的元数据的、怎么来管理 Bundle 的、怎么来管理 Service 的等等，以在实践中更好的使用 OSGI 框架搭建系统，同时通过对于 OSGI 的学习，不断的改善和提升之前在实战章节中的用户登录验证模块，以使其更加的贴合实际的应用。

7.1. 关于 OSGI

OSGI联盟⁵成立于 1999 年 3 月，致力于制定管理本地网络设备服务的规范。OSGI 组织是为家用设备、汽车、手机、桌面、小型办公环境以及其他环境制定下一代网络服务标准的领导者。

OSGI Service Platform 规范提供了开放和通用的架构，使得服务提供商、开发人员、软件提供商、网关操作者和设备提供商以统一的方式开发、部署和管理服务。OSGI 通过提供灵活的服务部署机制和强大的管理功能增强了设备的智能性。OSGI 规范制定的目标是为机顶盒、服务网关、Cable Modems、PC、汽车、手机等等提供服务。

7.2. OSGI R4 规范

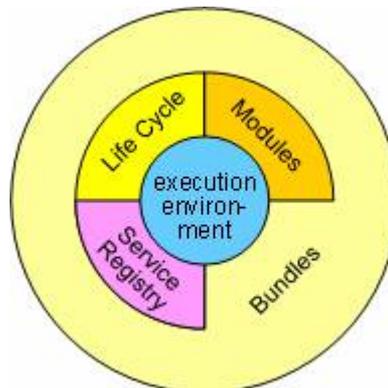
OSGI R4 规范由 Framework、Standard Services、Framework Services、System Services、Protocol Services、Miscellaneous Services 共同组成，在规范中对以上的各个方面进行了详细的介绍和规定，在这个章节中会深入的介绍 OSGI R4 中的关键部分，同时结合理论对之前的用户登录验证模块的实现进行讲解和改善。

7.2.1. Core Framework

Core Framework 是 OSGI 规范中的核心部分，它为基于 OSGI 的应用系统提供了标准的运行环境，从本质上保证了基于 OSGI 规范化的开发和部署动态性的系统。

OSGI 框架由 4 层组成：

- L0: 运行环境
- L1: 模块
- L2: 生命周期管理
- L3: 服务注册



⁵ 官方网站: www.osgi.org

可参见右边这张 OSGI Framework 的经典图：



L0 运行环境是指标准的 JAVA 环境。只要具备 Java 2 的构造和轮廓的都是被认可的运行环境。OSGI 同时也定义了一个可运行 Bundles 的最小环境的标准。



L1 模块层定义了所采用的类加载(Classloader)机制。OSGI 是一个强大、严格、规范的类加载模型，基于 Java 但增加了模块化。在 Java 中，通常都是由一个 Classloader 来加载所有的类和资源文件。在 OSGI 模块层中则为模块提供各自的 classloader，同时为模块的关联提供控制。



L2 生命周期管理层则为 Bundles 的动态安装、启动、停止、更新和卸载提供了支持。基于 L1 提供的模块类加载机制的基础上，增加了一个对于 Bundle 的管理的 API。



L3 增加了服务注册。服务注册为 Bundles 提供了一个动态的协作模型。本来 Bundles 可通过传统的 class 共享方式来实现协作，但在动态的安装和卸载代码的环境下这种方法是不适用的。服务注册为 Bundles 间共享 Objects 提供了一种可用的模型，OSGI 提供了一堆的事件来通知服务的使用者关于服务的注册和卸载，服务其实就是简单的 Java objects。很多服务象 objects 一样，例如 http server，而有些服务则代表了现实世界中的对象，例如蓝牙手机。

在 OSGI Framework 中还包括一个安全层次 (Security Layer)，OSGI 的安全层次基于 Java 的安全机制进行了扩展，增加了一些新的约束以及填补了 java 安全机制中的遗漏。

OSGI Framework 作为 OSGI 规范的核心，其中的定义决定了基于 OSGI 框架如何去设计、开发以及部署系统，下面就按照 OSGI Core Framework 的分层规则来分别讲解各个层次。

7.2.1.1. Module Layer

Module Layer 定义了如何在 OSGI 框架中是怎么去按照 Module 的思想去开发的，由于在目前的 Java 标准中并没有明确的按照 Module 方式定义的开发规范⁶，但按照 Module 的思想进行开发系统是很流行的思想，这也就使得象 Jboss、Netbeans 都有一套自己的 Module 规范，在这样的一套规范中，需要定义的主要是 Module 是如何去组织的、如何去部署的以及如何去共享 Module 的 package 的。

⁶ 目前正在制定的 JSR277 就是 Java Module System 的规范了

- **Module 的定义**

在 OSGI 规范中,将 Module 命名为 Bundle,所以,在 OSGI 框架中是采用 Bundle 的方式来组织和部署系统的, Bundle 的概念在之前的章节中已经介绍了,经过实战的演练,也已经知道 Bundle 和普通的 Java 工程唯一不同的是需要在 MANIFEST.MF 中编写 Bundle 的元数据信息,来看看在 MANIFEST.MF 中可以定义哪些 Bundle 的元数据信息:

属性	属性描述
Bundle-Activator	Bundle 的 Activator 类名。 示例: Bundle-Activator:org.riawork.demo.Activator
Bundle-Category	Bundle 的分类属性描述。 示例: Bundle-Category:Opendoc,OSGI
Bundle-Classpath	Bundle 的 Classpath。 示例: Bundle-Classpath:/bin,/lib/log4j.jar
Bundle-ContactAddress	提供 Bundle 的开发商的联系地址。 示例: Bundle-ContactAddress:ShangHai
Bundle-Copyright	Bundle 的版权。
Bundle-Description	Bundle 的描述信息。
Bundle-DocURL	Bundle 的文档 URL 地址。
Bundle-Localization	Bundle 的国际化文件。 示例: Bundle-Localization: OSGI-INF/110n/bundle
Bundle-ManifestVersion	定义 Bundle 所遵循的规范的版本, OSGI R3 对应的值为 1, OSGI R4 对应的值为 2。
Bundle-Name	Bundle 的有意义的名称。
Bundle-NativeCode	Bundle 所引用的 NativeCode 的地址。

Bundle-RequiredExecutionEnvironment	Bundle 运行所需要的环境，如可指定为需要 OSGI R3、Java 1.4、Java 1.3 等。
Bundle-SymbolicName	Bundle 的唯一标识名，可采用类似 java package 名的机制来保证唯一性。
Bundle-UpdateLocation	Bundle 更新时连接的 URL 地址。
Bundle-Vendor	Bundle 的开发商。
Bundle-Version	Bundle 的版本。
DynamicImport-Package	Bundle 动态引用的 package。
Export-Package	Bundle 对外暴露的 package。
Fragment-Host	Fragment 类型 Bundle 所属的 Bundle 名。
Import-Package	Bundle 引用的 package。
Require-Bundle	Bundle 所需要引用的其他的 Bundle。

对于之上的 Bundle 的元数据属性的值，都支持增加附加过滤属性的方式，如：

Import-Package 可以是这样的格式：

```
Import-Package:
org.riawork.opendoc.osgi;version="[1.0,2.0]";resolution:=mandatory,org.riawork.op
endoc.riawork;Company=RIAWork
```

● Module 的 package 共享机制

在开发 Bundle 章节中已经学习使用 Export-Package 和 Import-Package 来实现 Bundle 的包的提供和引用，但其实 Module Layer 还定义了更为实用和严格的 package 共享机制。

■ 过滤引用需要的 package

在开发 Bundle 章节中我们采用的是直接通过 import-package 方式来引用所需要的 package 的，未做任何的过滤设置，其实可以通过在 import-package 中设置其他的过滤属性，以更加准确的获取所需要引用的 package，可以通过版本过滤、Bundle 元数据信息过滤、自定义属性过滤、必须的属性过滤来实现过滤获取所需的 package。

◆ 版本过滤

可在 Import-Package 指定需要导入的 package 的版本范围或版本，示

例：

```
Import-Package: org.riawork.opendoc.osgi;version="1.0"
```

说明导入时只导入版本为 1.0 的 package, 那么相应的在导出的 package 中可以指定其版本：

```
Export-Package: org.riawork.opendoc.osgi;version="1.0"
```

版本过滤时还可以采用一种版本范围过滤的方式，即采用在数学中经常使用的区间的方式，如 [1.0,2.0]，说明可引用版本为 1.0= \leq version \leq 2.0 的 Package：

```
Import-Package: org.riawork.opendoc.osgi;version="[1.0,2.0]"
```

而[1.0,2.0]则表明引用的版本需要 1.0= \leq version \leq 2.0。

版本过滤的作用在于能够动态的更新系统的版本或者更好的去控制系统的版本兼容性问题等。

◆ Bundle 元数据信息过滤

在引用 package 时还可使用 Bundle 元数据信息来进行过滤，同样通过在 import-package 增加附加属性来实现，例：

```
Import-Package: org.riawork.opendoc.osgi;bundle-symbolic-name=B
```

◆ 自定义属性过滤

由于 Bundle 中的元数据基本都是可自定义属性的，所以在 import-package 时也可以采用自定义属性过滤的方法，例：

```
Import-Package:org.riawork.opendoc.osgi;company=RIAWork
```

那么它所引用的就是这个 Bundle export 的 package：

```
Export-Package:org.riawork.opendoc.osgi;company="RIAWork"
```

◆ 必须的属性过滤

之上的自定义属性过滤是一种非强制性的，而必须的属性过滤是一种强制性的，在 Export-Package 中通过 mandatory 来指定必须匹配的属性，例：

```
Export-Package:org.riawork.opendoc.osgi;company="RIAWork";security=false;mandatory:=security
```

```
Import-Package:org.riawork.opendoc.osgi;company=RIAWork
```

在这样的情况下是引用不到上面 Export 的 package，因为在 Export-Package 中指定了必须匹配 security 的属性。

■ Package 约束

Package 约束是在使用 import-package 时要较为注意的一个地方，举例来说：

A Bundle:

```
Import-package: org.riawork.opendoc.osgi;version="2.0"
Export-package: org.riawork.opendoc.riawork
```

B Bundle:

```
Export-package: org.riawork.opendoc.osgi; version=1.0
```

C Bundle:

```
Export-package: org.riawork.opendoc.osgi; version=2.0
```

这个时候如果 D Bundle 按照下面这样的方式引用包则会导致错误：

D Bundle:

```
Import-package:
org.riawork.opendoc.osgi,org.riawork.opendoc.riawork;version="1.0"
```

这个时候出错的原因就在于 D Bundle 去引用了版本均为 1.0 的 org.riawork.opendoc.osgi 和 org.riawork.opendoc.riawork，但唯一的暴露 org.riawork.opendoc.riawork 的 A Bundle 引用的却是版本为 2.0 的 org.riawork.opendoc.osgi，这个时候就出现冲突的问题了。

在 import 其他 bundle 提供的 package 时要注意这个问题，在 package 引用时如有和其他 bundle 提供的 package 同样的引用时，要避免出现版本上的冲突，其实上面的 D Bundle 只要改成这样就可以了：

```
Import-package:org.riawork.opendoc.osgi;version="2.0",org.riawork.opendoc.riawork
```

■ 限定导出的 package 中的类

在 OSGI 中还支持限定导出的 package 中的类，例如在同一个 package 下同时有接口类和实现类，在对外暴露 package 时只希望接口类被暴露出，

则可使用这个来限定：

```
Export-package: org.riawork.opendoc.osgi;exclude:=*"Impl";include="Val*"
```

在这样的设置下，只有 org.riawork.opendoc.osgi 下以 Val 开头且不以 Impl 结尾的类被暴露出来，也就是说在别的 Bundle 即使引用了 org.riawork.opendoc.osgi package，也只能使用其中 Validator 这样的类，但不能使用 ValidatorImpl 这样的类。

■ 动态的获取引用的 package

动态的获取引用的package和直接引用package的策略不同的地方在于：如果采用直接引用package的策略(也就是import-package)，那么OSGI框架在 resolve Bundle时就会对其import的package做检测，如不可用的话就会导致 resolve失败；但如果采用动态获取引用package的策略，则直到使用这个 package时才会去获取，而不是在resolve阶段获取⁷。

在 OSGI 框架有这么两种方法来实现动态的获取引用的 package：

◆ Dynamic Imports

这种方式为不使用 import-package，而是改为使用 DynamicImport-Package 的方式来引用其他 Bundle export 的 package。

如 Import-package: org.riawork.opendoc.osgi

改为

```
DynamicImport-Package:org.riawork.opendoc.osgi
```

◆ Resolution Directive

这种方式为继续使用 import-package，只是增加附加属性来声明引用的 package 采用 dynamic 方式载入，例：

```
Import-package: org.riawork.opendoc.osgi;resolution:=optional
```

● Module 的 Classloader 机制

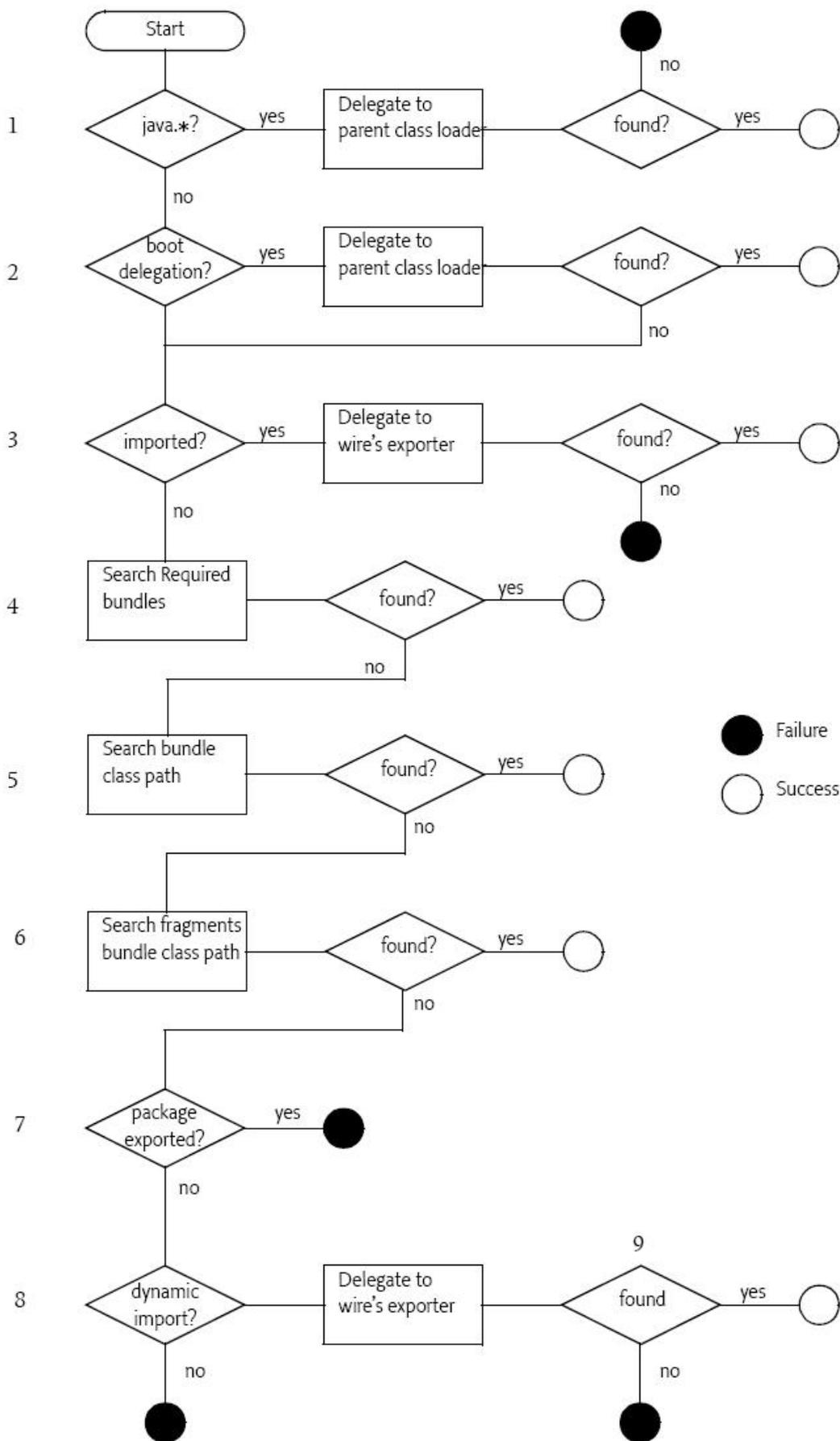
Module 机制中重要的一点就是拥有独立分离的 classloader 机制而得以保证 Module 的闭合，那么来看看在 OSGI 规范中是如何管理 Bundle 的 classloader 的。

OSGI 框架的 classloader 由 System Classloader、Bundle Classloader 共同组成，

⁷ 在 Lifecycle Layer 章节中会详细介绍 resolve

每个 Bundle 拥有独立的 Classloader。

在 OSGI 规范中有张经典的图可说明 Bundle 的 class 的加载机制的：



图表 7 OSGI Class loading 流程图

如图所示，OSGI 框架在加载 Bundle 中的类时按照这样的步骤进行：

- 如需要加载的为 java.* 的类，则直接委派给 Parent Classloader，如在 parent Classloader 中找到了相应的类，则直接返回，如未找到，则抛出 NoClassDefFoundException。
- 如加载的不是 java.* 的类，则进入这一步。判断加载的类是否属于 boot delegation 中配置的范围，如不属于则进入下一步，如属于则继续委派给 Parent Classloader，如在 Parent Classloader 中找到则直接返回，如未找到，则进入下一步。可在配置文件中编写 org.osgi.framework.bootdelegation 的属性来决定 boot delegation 的范围，示例：

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

- 如属于 Bundle Import package 中的类，则交给 export package 的 Bundle 的 classloader 进行加载，如加载失败，则直接抛出 NoClassDefFoundException，如加载成功则直接返回。这步就解释了之前在注意事项中所写的需要注意的包的问题。
- 如不属于 Bundle Import package 中的类，则搜索是否属于 Require Bundles 中 export 的 package 的类，如属于则交由 export package 的 Bundle 的 Classloader 进行加载，如加载成功则直接返回，如加载失败则进入下一步。
- 在 Bundle classpath(就是在 Bundle-Classpath 所配置的路径)中搜索需要加载的类，如加载成功，则直接返回，如加载失败则继续进入下一步。
- 搜索 Fragment Bundle(还记得配置的 Fragment-Host 吧)的 classpath，如加载成功，则直接返回，如加载失败则继续进入下一步。
- 判断是否属于 export 的 package，如属于则直接抛出 NoClassDefFoundException，如不属于则进入下一步。
- 判断是否属于 DynamicImport 的 package，如不属于则直接抛出 NoClassDefFoundException，如属于则使用 export package 的 Bundle 的 ClassLoader 进行加载，如加载成功则直接返回，如加载失败则抛出 NoClassDefFoundException。

对于 Bundle 中的资源文件，可使用 bundle.getResource、bundle.getEntry 或 bundle.findEntries 来获取，返回的为一个可被转变为 java.net.URL 的对象，通

过 URL 就可加载到相应的资源文件，如果要获取到其他 Bundle 的资源文件则需通过设置 Require-Bundle 的方式才可获取，Require-Bundle 也可视为实现资源文件共享的一种方式，不过 Require-Bundle 并不是被推崇的一种方式，在 OSGI 规范中，认为 Require-Bundle 会造成 split packages。

● Module 的国际化

标准的 Java 国际化的方案，默认国际化文件的目录为：OSGI-INF/i10n，配置文件名为类似 bundle_语言_国家_变量.properties，如：bundle_en.propties，同样，可以在 Bundle 的元数据中通过 Bundle-Localization 来指定国际化文件所在的目录，在使用国际化文件的情况下，在 MANIFEST.MF 文件中可采用%属性名的方式来使用国际化文件中的配置：

```
Bundle-Name: %BundleName  
Bundle-Version:% Bundle Version
```

bundle_en.properties 文件中配置如下：

```
BundleName=OSGI Opendoc  
Bundle Version=1.0
```

● Module 的校验

如果 Bundle-ManifestVersion 没有设置，则会使用其默认值 1，此时对应的为遵循 OSGI R3 规范，OSGI R4 规范中定义的新的 Bundle 的元数据信息则被忽略。

这里列出了一些会导致 Bundle 安装失败的原因：

- Bundle-RequiredExecutionEnvironment 中的值和可用的执行环境不符；
- 缺少 Bundle-SymbolicName；
- 重复的导入同一个 package；
- 导出或导入 java.*；
- 导出的 package 中必须的属性未定义；
- 安装一个已经安装了的同版本、同样标识名的 Bundle；
- 更新一个已经安装了的同版本、同样标识名的 Bundle；
- 同时使用了 specification-version 和 version；
- Bundle-ManifestVersion 的值不是 1 或 2，除非将来推出的新的 OSGI 规范

接受新的值。

- **三种特殊形式的 Bundle**

- **Require Bundles**

Require Bundle 其实不能算什么特殊形式的 Bundle, 它只是可以直接被其他 Bundle 通过 Require-Bundle 来使用的 Bundle。

如果使用了 Require-Bundle, 那么就可以使用该 Bundle 中所有的资源文件和 export 的 package。

- **Fragment Bundles**

Fragment Bundle 是一种比较特殊的 Bundle, 它本身并不拥有独立的 classloader, 可以把它看成是 Bundle 的一种附属, 它通过在元数据中指定 Fragment-Host 来说明其所依附的 Bundle, 只有在该 Bundle 使用时才会激活到这个 Fragment Bundle。

- **Extension Bundles**

Extension Bundle 也是一种比较特殊的 Bundle, 它用于扩展 system bundle, 通过 Fragment-Host 指定到 system bundle 的方式来实现对 system bundle 的扩展。

经过上面对于 Module Layer 规范的解读, 可以看出 OSGI 的 Module Layer 定义了完整的 Module 开发、部署、共享 package 等的规范, 而基于 Module Layer 一定程度上实现了系统的动态性, 使得设计人员在基于 OSGI 框架时可放心的按照 Module 的方式进行设计, 使得开发人员在基于 OSGI 框架时可按照规范的 Module 开发方式来进展。

7.2.1.2. Lifecycle Layer

Lifecycle Layer 基于 Module Layer, 使得基于 OSGI 框架可以动态的对 Bundle 的生命周期进行管理。

- **Bundle 的状态**

Bundle 的状态分为六种:

- **INSTALLED**

Bundle 已经成功的安装了。

- **RESOLVED**

Bundle 中所需要的类都已经可用了, RESOLVED 状态表明 Bundle 已经准

备好了用于启动或者说 Bundle 已被停止。

■ STARTING

Bundle 正在启动中，BundleActivator 的 start 方法已经被调用，不过还没返回。

■ ACTIVE

Bundle 已启动，并在运行中。

■ STOPPING

Bundle 正在停止中，BundleActivator 的 stop 方法已经被调用，不过还没返回。

■ UNINSTALLED

Bundle 已经被卸载了。

● 管理 Bundle 的状态

先来形象的看看基于 OSGI 框架是如何管理 Bundle 的状态的：

启动之前我们实现的用户登录验证模块，输入 ss 可以查看此时系统中 Bundle 的状态：

```
osgi> ss
Framework is launched.
id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       ACTIVE     ConfigFileValidatorBundle_1.0.0
2       ACTIVE     DBValidatorBundle_1.0.0
3       ACTIVE     LDAPValidatorBundle_1.0.0
4       ACTIVE     org.eclipse.equinox.http_1.0.0.v20060601a
5       ACTIVE     org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       ACTIVE     org.eclipse.osgi.services_3.1.100.v20060511
7       ACTIVE     UserValidatorBundle_1.0.0
8       ACTIVE     UserValidatorWebBundle_1.0.0
```

此时系统中所有的 Bundle 都处于 ACTIVE 状态，输入 stop 8 回车后再看看 id 为 8 的 Bundle 的状态：

```
osgi> stop 8

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       ACTIVE     ConfigFileValidatorBundle_1.0.0
2       ACTIVE     DBValidatorBundle_1.0.0
3       ACTIVE     LDAPValidatorBundle_1.0.0
4       ACTIVE     org.eclipse.equinox.http_1.0.0.v20060601a
5       ACTIVE     org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       ACTIVE     org.eclipse.osgi.services_3.1.100.v20060511
7       ACTIVE     UserValidatorBundle_1.0.0
8       RESOLVED   UserValidatorWebBundle_1.0.0
```

再输入 `uninstall 2`, 回车:

```
osgi> uninstall 2

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       ACTIVE     ConfigFileValidatorBundle_1.0.0
3       ACTIVE     LDAPValidatorBundle_1.0.0
4       ACTIVE     org.eclipse.equinox.http_1.0.0.v20060601a
5       ACTIVE     org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       ACTIVE     org.eclipse.osgi.services_3.1.100.v20060511
7       ACTIVE     UserValidatorBundle_1.0.0
8       RESOLVED   UserValidatorWebBundle_1.0.0
```

再输入 `start 8`, 回车:

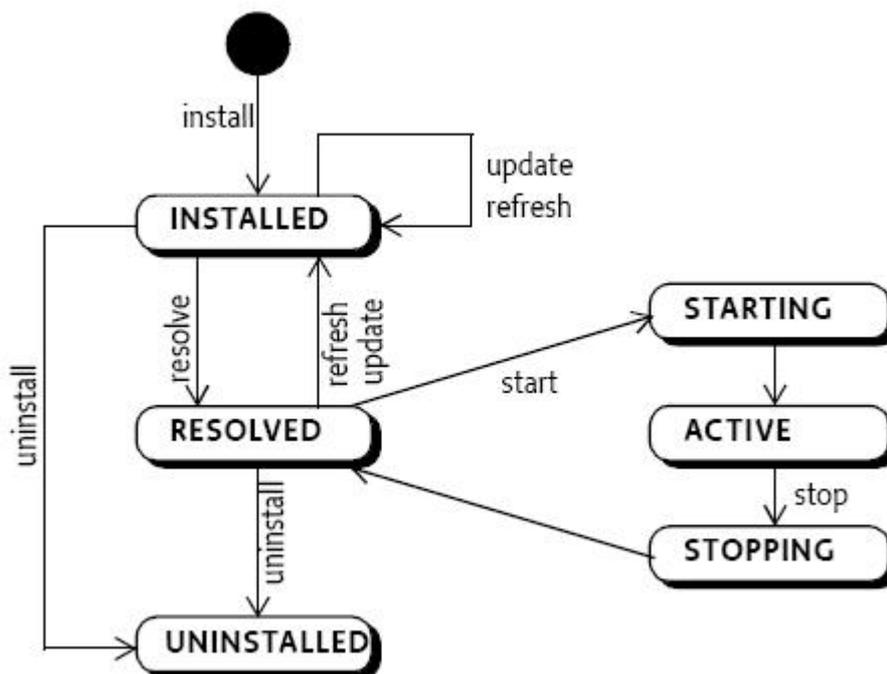
```
osgi> start 8

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       ACTIVE     ConfigFileValidatorBundle_1.0.0
3       ACTIVE     LDAPValidatorBundle_1.0.0
4       ACTIVE     org.eclipse.equinox.http_1.0.0.v20060601a
5       ACTIVE     org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       ACTIVE     org.eclipse.osgi.services_3.1.100.v20060511
7       ACTIVE     UserValidatorBundle_1.0.0
8       ACTIVE     UserValidatorWebBundle_1.0.0
```

下面这张图说明了 Bundle 的状态是怎么转换的:



来看看基于 Lifecycle Layer 是怎么完成这些状态的转变的:

■ 安装 Bundle

通过 `BundleContext` 的 `installBundle` 方法来安装 Bundle, 在安装前首先需要对 Bundle 进行校验, 导致 Bundle 安装失败的原因在之前 `Module Layer` 章节中已经描述过了, 如校验通过, OSGI 框架中将安装 Bundle 到系统中, 此时 OSGI 框架会分配一个高于现在系统中所有的 Bundle 的 ID 给新的 Bundle, 安装完毕后 Bundle 的状态就变为 **INSTALLED** 了, 同时会返回 bundle 对象, 在 Bundle 安装后就要使用 bundle 对象来管理 Bundle 的生命周期状态了。

■ 解析 Bundle

Bundle 安装完毕后, OSGI 框架将对 Bundle 进行解析, 以检测 Bundle 中的类依赖等是否正确, 如有错误则仍然处于 **INSTALLED** 状态, 如成功 Bundle 的状态则转变为 **RESOLVED**。

■ 启动 Bundle

在启动 Bundle 前需检测 Bundle 的状态, 如 Bundle 状态不为 **RESOLVED**, 那么需要先解析 Bundle, 如启动一个解析失败的 Bundle, 则会抛出 `BundleException`, 但此时 Bundle 的状态仍然会被设置为 **ACTIVE**; 如 Bundle 的状态已经是 **ACTIVE**, 那么启动 Bundle 对它不会产生任何影响。

通过 `BundleContext` 的 `getBundle` 方法可获取指定 `Bundle ID` 的 `Bundle` 对象,在获取到 `Bundle` 对象后可使用 `Bundle` 对象的 `start` 方法来启动 `Bundle`,此时会调用 `MANIFEST.MF` 中的 `Bundle-Activator` 属性对应的 `BundleActivator` 类的 `start` 方法(如存在 `BundleActivator` 类),在 `start` 方法执行的过程中 `Bundle` 的状态为 `STARTING`,当 `start` 方法执行完毕后 `Bundle` 的状态转变为 `ACTIVE`,如 `start` 方法执行失败, `Bundle` 的状态转变为 `RESOLVED`。

`BundleActivator` 类是可以不需要的,建议不要在 `BundleActivator` 中初始化过多的东西,这样会使得系统的启动速度会变得很慢,同时也会消耗大量的内存,而且 OSGI 对于动态性的良好支持使得尽可以在需要的时候才去获取所需的资源。

■ 停止 Bundle

通过 `BundleContext` 的 `getBundle` 方法可获取指定 `Bundle ID` 的 `Bundle` 对象,在获取到 `Bundle` 对象后可使用 `Bundle` 对象的 `stop` 方法来启动 `Bundle`,此时会调用 `MANIFEST.MF` 中的 `Bundle-Activator` 属性对应的 `BundleActivator` 类的 `stop` 方法,在 `stop` 方法执行的过程中 `Bundle` 的状态为 `STOPPING`,当 `stop` 方法执行完毕后 `Bundle` 的状态转变为 `RESOLVED`,如 `stop` 方法执行失败, `Bundle` 的状态则继续保留原状态。

即使 `Bundle` 已经停止,其 `export` 的 `package` 仍然是可以使用的,这就意味着可以执行 `RESOLVED` 状态的 `Bundle` 中 `export package` 的类。

■ 卸载 Bundle

通过调用 `Bundle` 对象的 `uninstall` 方法可完成 `Bundle` 的卸载,此时 `Bundle` 的状态转变为 `UNINSTALLED`。

即使 `Bundle` 已卸载,其 `export` 的 `package` 对于已经在使用的 `Bundle` 而言仍然是可用的,但对于新增的 `Bundle` 则不可使用已卸载的 `Bundle` `export` 的 `package`。

在管理 `Bundle` 的状态时,OSGI 主要是通过 `Bundle`、`BundleContext` 这两个对象来实现, `Bundle` 对象中除了对于 `Bundle` 的生命周期管理的方法之外,还提供了象 `getHeaders`、`loadClass`、`getResource` 这些方法, `getHeaders` 方法可用于获取 `MANIFEST.MF` 中的属性值, `loadClass` 可用于加载 `bundle` 中的类,

getResource 可用于获取 Bundle 中的资源文件。

- **监听 Bundle 的状态**

在监听 Bundle 的状态上 OSGI 采用的是典型的 Java 中的事件机制，在 OSGI 中事件分为 Framework Event 和 Bundle Event 两种，Framework Event 用于报告 Framework 已启动、改变了 StartLevel、刷新了 packages 或是出现了错误；而 Bundle Event 则用于报告 Bundle 的生命周期的改变。

可通过实现 BundleListener 或 SynchronousBundleListener 来监听 Bundle Event，可通过实现 FrameworkListener 来监听 Framework Event。

7.2.1.3. Service Layer

Service Layer 定义了 Bundle 动态协作的服务发布、查找和绑定模型，Service Layer 基于 Module Layer 和 Lifecycle Layer，使得 OSGI 形成了完整的动态模型。

不过 Service Layer 的定义比较简单，是一个典型的 Service Locator 模式的模型，Service 通过 BundleContext 完成注册和获取。

- **服务的注册**

可在任何时候通过 BundleContext 的 registerService 方法来完成服务的注册，和其他的服务框架一样，在 OSGI 中注册服务时也可以注册一个 ServiceFactory 的类，服务成功注册后会返回 ServiceRegistration 对象，通过这个对象的 unregister 方法可卸载服务。

- **服务的获取**

在获取服务时除了按照实战章节中的示例方式获取之外，还可通过构造 Dictionary 对象来增加过滤属性，以更加准确的获取所需要的服务。

同样的，可在任何时刻通过 BundleContext 来获取服务，而通过 BundleContext 在需要的时候获取服务则可保证获取服务的动态性。

- **服务的监听**

服务的监听在实战章节中已经举例说明了，通过实现 ServiceListener 可监听 Service 的状态，通过 BundleContext 注册监听器，在注册监听器时可增加过滤的属性，以更加准确的监听希望监听的服务的事件。

在 OSGI R4 推出 Declarative Services 之后，Service Layer 其实就已经成为鸡肋了，Declarative Services 提供了更好的服务注册、获取、监听等方式，使得其成为了 OSGI R4 中的重要角色，并由此替代了 Service Layer。

7.2.1.4. Security Layer

Security Layer 在 OSGI 中采用的主要是 java 本身的 security 策略和数字签名策略，这方面就不在这里做过多介绍，如需要了解的话请翻阅《OSGI Service Platform Release 4》。

7.2.2. StartLevel Service

StartLevel Service 是 OSGI 规范中的核心服务，是 OSGI 框架必须实现的服务，通过 StartLevel Service 可以动态的设置和改变 Bundle 的启动顺序，OSGI 框架在启动 Bundle 时按照启动顺序和 Bundle ID 的倒序来启动系统。

在 OSGI 中，启动顺序称为 StartLevel，在之前的实战章节中，没有去控制 Bundle 的启动顺序，通过 config.ini 可以静态的设置系统的启动级别和各 Bundle 的启动级别，而通过在运行期获取 StartLevelService 则可动态的改变系统、各 Bundle 的启动级别以及新安装的 Bundle 的启动级别。

在未设置 OSGI 系统的启动级别时，系统将会启动所有的 Bundle，然后将系统的启动级别设置为当前启动的 Bundle 的最高级别，在未设置默认的 Bundle 的级别时，OSGI 框架将采用默认设置 Bundle 的启动级别为 4。

7.2.2.1. 静态的设置系统和各 Bundle 的启动级别

在 equinox 中，通过在 config.ini 中设置 osgi.startLevel 的值来设置基于 OSGI 系统的启动级别，打开之前在实战章节中编写的 config.ini，其中没有 osgi.startLevel 属性的设置，在这种情况下 equinox 会将系统的启动级别默认的设置 6，在 config.ini 中设置了 osgi.bundles.defaultStartLevel 的值，这个属性是用来设置 bundle 的默认启动级别的，在 config.ini 中并没有去设置各 bundle 的启动级别，这个时候就会采用 osgi.bundles.defaultStartLevel 作为 Bundle 的级别，根据这样的说法，之前的用户登录验证模块的系统启动级别为 6，各 Bundle 的启动级别则为 4，由于系统的启动级别比各 Bundle 的启动级别高，所以在启动系统时所有的 Bundle 都启动了，可以试着把系统的级别改为比各 Bundle 的启动级别低来看看会造成什么样的效果，在 config.ini 中增加一行这样的设置：

```
osgi.startLevel=3
```

这样系统的启动级别就被设置为 3 了，而各 Bundle 的启动级别则为 4，这个时候再运行 run.bat 试试：

```
osgi> ss
Framework is launched.

id      State      Bundle
0       ACTIVE     system.bundle_3.2.0.v20060510
1       RESOLVED   ConfigFileValidatorBundle_1.0.0
3       RESOLVED   LDAPValidatorBundle_1.0.0
4       RESOLVED   org.eclipse.equinox.http_1.0.0.v20060601a
5       RESOLVED   org.eclipse.equinox.servlet.api_1.0.0.v20060601
6       RESOLVED   org.eclipse.osgi.services_3.1.100.v20060511
7       RESOLVED   UserValidatorBundle_1.0.0
8       RESOLVED   UserValidatorWebBundle_1.0.0
9       RESOLVED   DBValidatorBundle_1.0.0
```

从上图中，可以看出在这种情况下除了 system bundle，其他所有的 bundle 都没有启动，这是由于 system bundle 的启动级别设置的为 0，并且是不可改变的，而其他 bundle 的启动级别比系统启动级别高，所以就没启动，这个时候再手动去启动 bundle，看看会发生什么，输入 start 1，再输入 ss 查看下状态，会看到 id 为 1 的 state 仍然是 RESOLVED，这就说明了当 Bundle 启动级别比系统启动级别高时，Bundle 无论通过什么方法都是无法启动的，这个时候要启动 Bundle 的做法只能是去动态的改变系统的启动级别，使其等于或者大于 Bundle 启动级别，才可启动 Bundle。

静态的设置各 Bundle 的启动级别有两种做法，一种就是通过设置 Bundle 的默认启动级别来改变所有未设置启动级别 Bundle 的启动级别，另外一种就是分别的设置各 Bundle 的启动级别。

之前已经说过，可以通过设置 osgi.bundles.defaultStartLevel 的值来改变 Bundle 的默认启动级别，打开刚刚修改后的 config.ini，把其中的 osgi.bundles.defaultStartLevel 的值改为 2，再运行 run.bat，输入 ss 来查看系统中 Bundle 的状态，可以看到这个时候所有的 Bundle 的状态均为 ACTIVE，表明所有的 Bundle 的都启动了。

如果要单独的设置 Bundle 的启动级别，打开 config.ini，可以看到其中有这样的格式：

```
reference\:file\:bundles/ConfigFileValidatorBundle_1.0.0.jar@start
```

可以通过在 @ 后增加启动级别来单独的设置该 Bundle 的启动级别：

```
reference\:file\:bundles/ConfigFileValidatorBundle_1.0.0.jar@4:start
```

再运行 run.bat，输入 ss 查看系统 bundle 的状态，可以看到 ConfigFileValidatorBundle 的 state 为 RESOLVED，而其他的 Bundle 由于采用的是默认的启动级别，state 则

为 ACTIVE。

通过这样的方法可以静态的设置系统和各 Bundle 的启动级别,在启动系统时,OSGI 框架将按照各 Bundle 的启动级别从小到大的启动,如启动级别相同,则按照 Bundle ID 从小到大的启动,一直启动到系统的启动级别为止。

7.2.2.2. 动态的设置系统和各 Bundle 的启动级别

动态的设置系统和各 Bundle 的启动级别可通过在运行时调用 StartLevelService 来实现。

同样的,既然要使用 StartLevelService,先打开 MANIFEST.MF,在 import packages 中 import org.osgi.service.startlevel,通过 BundleContext 获取 StartLevelService:

```
ServiceReference serviceRef=bc.getServiceReference(StartLevel.class.getName());  
StartLevel startLevelSrv=(StartLevel) bc.getService(serviceRef);
```

StartLevel 提供了管理系统和各 Bundle 启动级别的 API:

- `getBundleStartLevel(Bundle bundle)`
获取 Bundle 的启动级别。
- `getInitialBundleStartLevel()`
获取默认的 Bundle 的启动级别。
- `getStartLevel()`
获取系统的启动级别。
- `setBundleStartLevel(Bundle bundle,int startLevel)`
设置 Bundle 的启动级别,如设置的启动级别高于系统的启动级别,那么 OSGI 框架将会停止此 Bundle,如设置的启动级别低于系统的启动级别,如此时 Bundle 尚未启动,OSGI 框架则会启动此 Bundle,如已启动,OSGI 框架不会做任何动作。
- `setInitialBundleStartLevel(int startLevel)`
设置 Bundle 的默认启动级别,对于新安装的 Bundle 将使用此默认启动级别。
- `setStartLevel(int startLevel)`
设置系统的启动级别。
如设置的系统启动级别和目前系统的启动级别一致,则 OSGI 框架会发布 `FrameworkEvent.STARTLEVEL_CHANGED` 的事件;

如设置的系统启动级别比目前的系统启动级别低，那么 OSGI 框架将现有的启动级别按 1 递减，并停止相应启动级别的 Bundle，直至设置的系统启动级别，当系统的启动级别和设置的启动级别一致时，OSGI 框架会发布 FrameworkEvent.STARTLEVEL_CHANGED 的事件；

如设置的系统启动级别比目前的系统启动级别高，那么 OSGI 框架将现有的启动级别按 1 递增，并启动相应启动级别的 Bundle，直至设置的系统启动级别，当系统的启动级别和设置的启动级别一致时，OSGI 框架会发布 FrameworkEvent.STARTLEVEL_CHANGED 的事件。

7.2.2.3. 适用场景

在 OSGI 规范中列举了 StartLevel Service 的适用场景：

- 安全模式

安全模式可用于只启动系统所需要的 Bundles，就象 windows 的安全模式一样，可只启动必须的 Bundles，而其他不是必须的的 Bundles 则不启动。

可通过静态或动态的设置系统的启动级别来实现安全模式。。

- Splash Screen

就像 Eclipse 一样，在启动前先出现一个等待的屏幕，等启动完毕后再进入系统界面，通过设置 Bundle 的启动级别，就可以实现这点了。

典型的实现方法就是写一个用于提供 Splash Screen 的 Bundle，将该 Bundle 的启动级别设置为 1，而其他的 Bundle 则设置比 1 高的启动级别，Splash Screen 的 Bundle 同时实现 FrameworkListener，当监听到 FrameworkEvent.STARTED 时，则表明系统已启动完毕，这时可以将 Splash Screen 关闭。

- 处理不稳定的 Bundles

不稳定的 Bundles 是指那些依赖别的 Bundle 提供的 Service 才能正常启动的 Bundle，这其实是 OSGI 实践时要注意的一点，就是在启动时不要去静态的依赖其他的 Bundle，充分的发挥基于 OSGI 的动态性。

如果一定依赖其他的 Bundle，那么就可以通过设置 Bundle 的启动级别来避免不稳定的 Bundles 启动造成的错误。

- 高优先级的 Bundles

系统有些时候会要求某些 Bundles 快速的启动，这个时候就可以将他们的启动级别设置的低一点，使得在运行时先启动这些 Bundles。

7.2.3. Declarative Services

尽管 Declarative Services (DS) 和别的 Standard Services (例如 Log Service、Http Service) 放在同样的章节级别中, 但 DS 绝对称得上是 OSGI R4 的重大改进, 它对 OSGI 的 Core Framework 进行了完善和提升, 在 OSGI Core Framework 对于系统采用的是 Module+Service 的开发方式, 但按照分结构的开发方式而言, 系统按照的是 Module 分解为 Component+Service, 在 OSGI Core Framework 中, 并没有明确的 Component 的概念, 只能借助 BundleActivator 作为 Module 中的 Component, 而其他的都作为 Service, 尽管 OSGI Core Framework 中的 Service Layer 对于 Service-Oriented 的系统已经提供了完整的服务注册、查找、绑定和监听的定义, 但对于 Service 的发布、使用而言并不是非常方便, DS 则提升了 Service 的发布和使用的简便性, 同时更好的去实现 Service 的动态性, 在 DS 上会看到 DI Container 的影子, 不过 DS 除了具备 DI Container 的特征之外, 还有一点更为根本的是保证了系统的动态性。

DS 提出了完整的 Service-Oriented Component Model (SOCM) 概念, 使得在 Bundle 中可以按照 Component+Service 的方式进行开发, 来看看在 DS 的支撑下怎么去做。

7.2.3.1. Component 的概念和定义

Component 和 Service 从定义上来看是差不多的, 任何一个普通的 Java 对象都可以通过在配置文件中定义成 Component, 那么 Component 到底有什么作用呢:

- 对外提供 Service;
- 使用其他 Component 提供的 Service;
- 交由 OSGI 框架管理生命周期。

只要具备上面三点之一的 Java 对象就可以定义为 Component, 从而借助 DS 来最简化的达到上面三点需求。

首先来看看怎么把一个普通的 Java 对象定义为 DS 中的 Component:

假设需要把 org.riawork.opendoc.osgi.DemoComponent 类定义为 DS 中的 Component, 首先需要编写一个 xml 文件来声明, xml 文件格式类似如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
```

```
</component>
```

把这个文件存为OSGI-INF/components.xml。

然后在MANIFEST.MF文件中引用这个文件就可以了：

Service-Component: OSGI-INF/components.xml

关于 Component 对外提供 Service 和使用其他 Component 提供的 Service 的部分在后续章节做详细讲解,在这里主要讲讲交由 OSGI 框架管理生命周期的 Component,在 DS 中对于 Component 分为三种类型:

- Immediate

对于 Immediate Component, DS 会立刻激活这个 Component。

在这个 Component 的配置文件被改动、配置信息文件(properties 文件)被改动以及所引用的 Service 被改动的情况下, DS 都会重新装载并激活这个 Component,同时对于引用了这个 Component 中 Service 的 Component 也会做同样的动作。

- Delayed

对于 Delayed Component, DS 会根据配置文件中的 Service 的配置,注册 Service 的信息,但此时不会激活这个 Component。

直到这个 Component 被请求的时候 DS 才会去激活这个 Component,相应的设置引用的 Service。

在这个 Component 引用的 Service 发生改变的情况下, DS 不会重新装载这个 Component,而会采用调用配置中设置的 bind、unbind 方法来重新设置引用的 Service。

- Factory

Factory Component 和 Immediate Component 基本相同,不过它在激活后注册的只是一个 ComponentFactory 服务,只有在调用 ComponentFactory 的 newInstance 后才会激活里面的各个组件。

对于这三种类型的 Component 大家是不是会觉得和 Spring 中的 Bean 有所类似呢?

这三种类型的组件中无疑 Delayed Component 是最需要的,为系统的动态性带来了很大的好处,同时也节省了内存的占用。

组件的生命周期受 bundle 生命周期影响,当 bundle 停止时 bundle 中所有组件也会

随之停止，组件的生命周期可以通过 `activate` 和 `deactivate` 这两个方法来主动控制，`activate` 方法在组件被激活时被调用，`deactivate` 方法在组件被停止时调用，这两个方法也是可以不被实现的，同时，通过这两个方法可以获取到 `ComponentContext`，而通过它则可以获取到 `BundleContext`，通常来说这是非常有用的。

一个简单的 `Component` 示例如下：

```
public class LifecycleComponent{

    public void activate(ComponentContext context){
        // 进行组件的初始化工作
    }

    public void deactivate(ComponentContext context){
        // 进行组件被停止前的工作
    }

}
```

7.2.3.2. Service 的发布

在 **Service Layer** 中 `Service` 的发布需要通过 `BundleContext` 注册，而在 `DS` 中则不需要这么做了，在 `DS` 中只需要通过在之前 `component` 的配置文件中进行定义即可完成：例如需要提供 `org.riawork.opendoc.osgi.DemoComponent` 中实现的 `ValidatorService`，那么只需要修改之前编写的 `xml` 文件为如下格式：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="org.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<service>
    <provide interface="org.riawork.opendoc.osgi.ValidatorService"/>
</service>
</component>
```

`DemoComponent` 示例如下：

```
public class DemoComponent implements ValidatorService{

    public boolean validator(String username,String userpass) throws Exception{

        return true;

    }

}
```

7.2.3.3. Service 的引用

在 **Service Layer** 中需要通过 **BundleContext** 来获取其他 **Bundle** 提供的 **Service**，非常的不方便，因为通常要使用 **Service** 的类并不是 **Bundle**，这就导致了要在启动 **Bundle** 时获取 **Service** 或在 **Bundle** 中提供对于 **BundleContext** 的静态获取，而在 **DS** 中则可以象 **DI Container** 一样，采用注入的方式来获取需要引用的 **Service**。

例如 `org.riawork.opendoc.osgi.DemoComponent` 需要使用 `HttpService`，`DemoComponent` 这么写：

```
public class DemoComponent{

    private HttpService service;

    public void setHttpService(HttpService service){

        this.service=service;

    }

    public void unsetHttpService(HttpService service){

        if(this.service!=service)

            return;

        this.service=null;

    }

}
```

配置文件修改如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<reference name="HTTPService" interface="org.osgi.service.http.HttpService"
bind="setHttpService" unbind="unsetHttpService"/>
</component>
```

当 `HttpService` 可用时，DS 就会自动的将 `HttpService` 注入到 `DemoComponent` 中，如当 `HttpService` 不可用时，DS 就会自动调用 `DemoComponent` 的 `unsetHttpService` 方法。

7.2.3.4. Service 的引用策略

在 **Service Layer** 中提供了通过设置过滤属性来更加准确的获取说需要的 **Service**，而在 DS 中则提供了更加完整的 **Service** 引用策略：

- **Cardinality**

由于在 OSGI 框架中服务的提供通常都是提供接口的方式，那么就会产生系统中提供多个实现接口的服务的问题，例如在系统中可能同时有三个验证服务的实现，那么在引用服务时其实是可以同时引用这三个服务的，在 DS 中可以通过 **Cardinality** 的定义来只引用一个或多个服务。

Cardinality 有四种策略：

- 0..1 最多只引用其中的一个服务，但系统也可以不存在可用的服务，象 Log 服务就很适合这种情况，即使系统中不存在 Log 服务组件应该也同样保持可用。
- 1..1 引用的服务必须至少有一个可用的（默认）。
- 0..n 引用 0 到多个可用的服务。
- 1..n 引用的服务必须存在，使用时可使用多个服务。

设置 **Cardinality** 策略只需要在之前的 **service** 引用的配置上增加 **Cardinality** 属性的配置即可：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
```

```
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<reference name="HTTPService"
interface="org.osgi.service.http.HttpService"
bind="setHttpService"
unbind="unsetHttpService"
cardinality="0..n"/>
</component>
```

在这样的情况，假设系统中有三个可用的 `HttpService`，那么 `setHttpService` 方法就会被调用三次，如果不配置 `cardinality` 属性，那么系统将采用默认的 `cardinality="1..1"`，在这种情况下，`setHttpService` 只会被调用一次，并且要求系统中必须至少有一个可用的 `HttpService`，否则 `Component` 将会启动不了。

- 引用策略

引用策略是指在引用服务时可采用 `static` 和 `dynamic` 两种策略方式，`static` 策略是默认的方式，在 `Component` 启动后，如果引用的 `service` 发生变动时，整个 `Component` 会重启；而采用 `dynamic` 策略的情况下，在 `Component` 启动后，如果引用的 `service` 发生变动，并不会导致整个 `Component` 重启，而只会调用其中的 `unbind` 和 `bind` 方法。

要使用引用策略只需要在之前的 `service` 引用配置中增加 `policy` 属性的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<reference name="HTTPService"
interface="org.osgi.service.http.HttpService"
bind="setHttpService"
unbind="unsetHttpService"
cardinality="0..n"
policy="dynamic"/>
</component>
```

- 属性过滤

在**Service Layer**中是通过填充属性到Dictionary对象中来实现根据属性过滤获取服务，而在DS中同样可以通过在配置中定义需要过滤的属性：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<reference name="HTTPService"
interface="org.osgi.service.http.HttpService"
bind="setHttpService"
unbind="unsetHttpService"
cardinality="0..n"
policy="dynamic"
target="(component.version=1.0)"/>
</component>
```

那么这个属性是怎么配置到 component 中的呢？在 DS 中可以通过 property 和 properties 两种方式来配置 component 的属性：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="osgi.DemoComponent">
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>
<property name="component.version">1.0</property>
<property name="component.canuse" type="Boolean">true</property>
<properties entry="OSGI-INF/config.properties"/>
</component>
```

可以看出 property 用于配置单个属性，在 property 中可以指定 property 值的类型，而 properties 则用于引用 bundle 中的配置文件。

7.2.3.5. Service 的状态监听

在**Service Layer**中是采用实现ServiceListener的方式来监听服务的状态的，而在DS中则不需要，在引用的service发生变化时，DS会自动的通知相关的Component。

7.2.3.6. 基于 DS 重构用户登录验证模块

由于目前 DS 并没有融合到 OSGI R4 Core Framework 中, 需要单独的下载 DS 实现的 Bundle, 先到 equinox 的下载网站下载 org.eclipse.equinox.ds_1.0.0.v20060601a.jar (类似这样名称的 jar 包), 下载完毕后放入 eclipse 的 plugins 目录, 启动 Eclipse。在用户登录验证模块中用 DS 来改进服务的发布和引用:

- **重构服务的发布**

用户登录模块中 ConfigFileValidatorBundle、DBValidatorBundle 和 LDAPValidatorBundle 都是通过 BundleContext 的 registerService 方法来实现对外发布服务的, 在 DS 中则不需要这么做了, 只需要将之前提供服务的类配置为 Component, 然后在配置文件中提供对外发布的服务即可。

以 ConfigFileValidatorBundle 为例:

- **第一步**

由于不需要通过 BundleContext 来注册 Service 了, 那么 BundleActivator 类就没有意义了, 于是重构第一步就是把 BundleActivator 删了, 删了后同时删除 MANIFEST.MF 中 Bundle-Activator 属性;

- **第二步**

在 ConfigFileValidatorBundle 工程下建立 OSGI-INF 目录, 在该目录下建立 component.xml, 内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="ConfigFileValidator">
  <implementation
class="org.riawork.demo.service.user.impl.ConfigFileValidatorImpl"/>
    <service>
<provide interface="org.riawork.demo.service.user.Validator"/>
    </service>
</component>
```

通过这个文件将 ConfigFileValidatorImpl 定义成了 DS 中的 Component, 同时对外提供了接口为 org.riawork.demo.service.user.Validator 的服务。

- **第三步**

打开 MANIFEST.MF 文件，在其中加入对于 component 配置文件的引入：

```
Service-Component: OSGI-INF/component.xml
```

按照同样的方式重构 DBValidatorBundle 和 LDAPValidatorBundle。

- **重构服务的引用**

用户登录模块中只有 UserValidatorWebBundle 中的 LoginServlet 需要引用其他 Bundle 提供的服务，那么就只需要重构 UserValidatorBundle 即可。

- **第一步**

之前是通过在 BundleActivator 监听 HttpService 来注册 LoginServlet，在 DS 中不需要通过 BundleActivator 来完成，那么第一步就是删除 BundleActivator 类，同时删除 MANIFEST.MF 中的 Bundle-Activator 属性。

- **第二步**

LoginServlet 需要使用其他 Bundle 提供的 HttpService 和 Validator 服务，首先删除 LoginServlet 中的 LoginServlet(BundleContext context)的构造器，增加 setHttpService、unsetHttpService、setValidator 和 unsetValidator 方法，获取 HttpService 和 Validator 服务。

- **第三步**

LoginServlet 激活的前提条件是系统中必须有一个可用的 HttpService 服务，而且 LoginServlet 只使用一个 HttpService 服务，至于 Validator 服务，即使没有系统也仍然要可运行，LoginServlet 至多也只需要一个 Validator 服务。

在 UserValidatorWebBundle 工程下建立 OSGI-INF 目录，新建 component.xml 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="LoginServlet">
<implementation class="org.riawork.demo.web.servlet.LoginServlet"/>
<reference
                                name="ValidatorService"
interface="org.riawork.demo.service.user.Validator"
                                bind="setValidator"
unbind="unsetValidator" policy="dynamic" cardinality="0..1"/>
<reference name="HttpService" interface="org.osgi.service.http.HttpService"
```

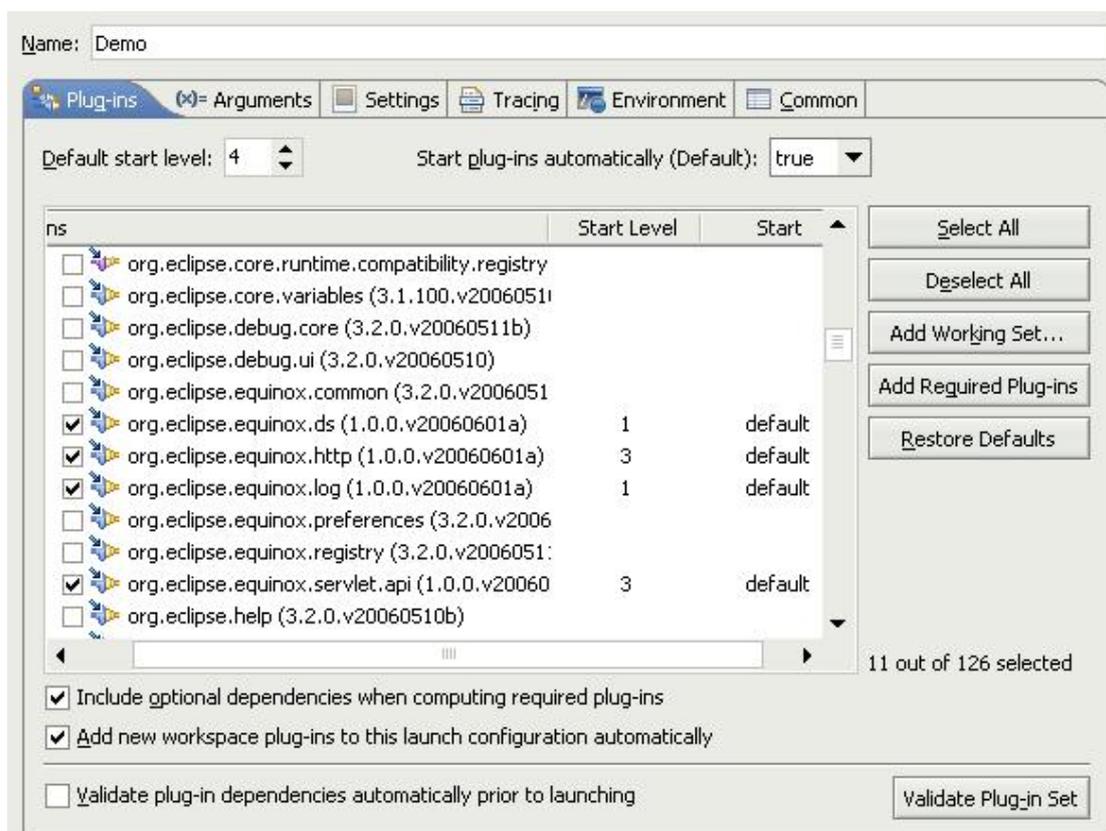
```
bind="setHttpService" unbind="unsetHttpService" policy="dynamic"/>
</component>
```

unsetValidator 方法是这么写的:

```
if(this.validator!=validator)
    return;
this.validator=null;
```

这么写的原因是当之前 set 的 Validator 服务不可用时, DS 会自动的寻找新的 Validator 服务注册到 LoginServlet, 之后再调用 unsetValidator 方法, 这个时候其中的 validator 参数就是告诉 LoginServlet 哪个 validator 服务已经不可用了。

重构完了服务的发布和引用后, 就可以重新运行用户登录模块了, 在运行用户登录模块中加入 org.eclipse.equinox.ds bundle, 并将它的启动级别设置为 1, 让它比其他的 Bundle 先启动, 配置完了后重新运行用户登录验证模块即可。



基于DS重构后的用户登录验证模块中的服务的注册、引用和状态的监听都变得非常容易了⁸, 重构后的代码具体见附件DS目录下的Bundle工程。

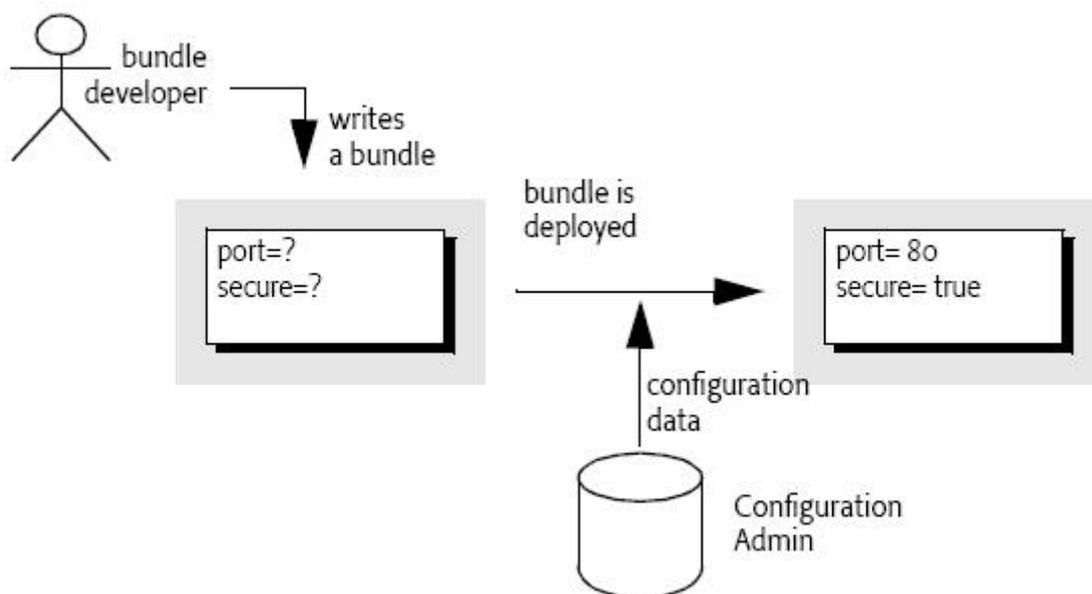
⁸ 目前DS的实现还处于测试阶段, equinox team认为使用DS可能会产生效率问题, 同时不适合在多线程情

7.2.4. Configuration Admin Service

7.2.4.1. 规范描述

Configuration Admin Service 是 OSGI 中非常重要的一个服务，它用于动态的管理 Bundle 的配置的属性，而这些属性的存储可能是在本地、也有可能是在远端、甚至可能会在某些设备上，基于 Configuration Admin Service 就可以统一的、动态的、远程的完成 Bundle 配置属性的管理工作了。

规范中的这张图挺形象的解释了 Configuration Admin Service 的作用：



图表 8 Configuration Admin Service Overview(摘自 OSGI R4)

Configuration Admin Service 的实现简单的去看可以认为它是一个事件订阅/通知模型，所以在使用 Configuration Admin Service 的时候也是基本可以按照事件订阅/通知模型的使用方法而开展，尽管实际上的 Configuration Admin Service 的实现并不是如此的简单，要知道这个服务的目的是要实现远程的管理 Bundle 的属性配置的。对于开发人员来说最重要的仍然是了解基于 Configuration Admin Service 如何去实现属性的管理和获取。

7.2.4.2. 跟踪属性的变化

假设 Bundle A 中有一个服务的接口是 `org.riawork.opendoc.service.DemoCM`，该 Service 需要使用的配置属性有 `cm.port`，那么怎么去跟踪这个属性值的变化呢？

为了获取配置的属性，需要了解 Configuration Admin Service 中的这么几个对象：

- ManagedService

`ManagedService` 接口只有一个 `updated` 方法，从这个方法可以看出事件订阅/通知的模型，既然是这样，必然要将当前实现对外提供 `ManagedService` 的服务，同时做为需要告诉外部它订阅的是什么，这个就需要在注册时对此服务增加 `Constants.SERVICE_PID` 属性的注册，这样 `Configuration Admin Service` 在服务属性被改动的情况下就会相应的根据 `Constants.SERVICE.PID` 的值来通知相应的服务。

- `ManagedServiceFactory`

`ManagedServiceFactory` 接口有三个方法需要实现，其功能和 `ManagedService` 是差不多的，区别在于 `ManagedService` 只能订阅一个服务的属性的变化，而 `ManagedServiceFactory` 可以订阅多个服务的属性的变化，这个对于一个组件是多个服务的实现的情况下就非常有用了。

通过实现 `ManagedService` 接口来完成上面的例子的要求：

```
private ServiceRegistration serviceReg=null;

private ServiceRegistration manSrvReg=null;

public void start(BundleContext context) throws Exception {
    DemoCM cm=new DemoCMImpl();
    Dictionary props=new Hashtable();
    props.put("cm.port", "1220");
    serviceReg=context.registerService(DemoCM.class.getName(), cm,
props);
    Dictionary manProps=new Hashtable();
    manProps.put(Constants.SERVICE.PID, DemoCM.class.getName());

    manSrvReg=context.registerService(ManagedService.class.getName(), this,
manProps);
}
```

```
public void stop(BundleContext context) throws Exception {  
    serviceReg.unregister();  
    manSrvReg.unregister();  
}  
  
public void updated(Dictionary props) throws ConfigurationException {  
    serviceReg.setProperties(props);  
}
```

当 DemoCM 的服务的属性被改变后，就会相应的通知到 DemoCM，在 updated 方法中可以根据需要做出处理。

7.2.4.3. 管理属性

管理属性包括对于属性的获取、增加、修改和删除，通过 Configuration Admin Service 可以较为容易的实现这些操作，例如要管理上面服务的属性：

```
ServiceReference  
serviceRef=context.getServiceReference(ConfigurationAdmin.class.getName());  
ConfigurationAdmin admin=(ConfigurationAdmin)context.getService(serviceRef);  
Configuration config=admin.getConfiguration(DemoCM.class.getName());
```

获取到了 config 之后就可以通过 config.getProperties 来获取、增加、修改和删除其中的属性，在完成了所有操作后通过 config.update() 方法即可完成对于相应服务的通知。

7.2.4.4. 其他作用

Configuration Admin Service 的管理配置属性的方法还可以被用于动态的调整系统行为方面，如用户登录验证模块在获取验证服务时的方法，就可以通过 Configuration Admin Service 来改变为动态的改变的方式，只需要将获取验证服务时的属性做为可配置的，在系统运行期就可以通过动态的修改这个属性值来切换对于验证服务的获取，同样的，这样的方法也可以被用于升级接口 API、接口的实现等，这对于动态的改变系统行为是具备很重要的意义的。

不过到本文编写完毕之时 Equinox 还没有实现 Configuration Admin Service，如果要试用的话可以从 equinox 的 cvs 中下载。

7.2.5. Event Admin Service

7.2.5.1. 规范描述

OSGI 提供了 Event Admin Service 以方便开发人员实现自定义的事件发布和事件处理，和 Java 的事件处理模型基本是一致的。



图表 9 Event Admin Service Architecture(摘自 OSGI R4)

事件发布者通过调用 Event Admin Service 的 sendEvent 或 postEvent 方法对外发布事件，事件订阅者则通过实现 Event Handler 并注册为 EventHandler 的服务来订阅感兴趣的事件。

Event Admin Service 从规范上来说比较的简单，在这里还是以如何使用它来进行介绍。

7.2.5.2. 发布自定义的事件

对于发布自定义的事件，需要接触的主要是这么两个对象：

- EventAdmin

EventAdmin 提供了 sendEvent 和 postEvent 两个方法供外部调用以发布事件，两个方法的区别在于 sendEvent 是同步调用，postEvent 是异步调用。

- Event

Event 的构造器有两个参数，一个是 String 类型的 Event 的 Topic，也就是事件的主题了，就像 BUNDLE_STARTED 这种事件，在规范中建议 Topic 采用包名 / 类名 / 事件主题名这样的方式来构成 Topic，就像这样：
org/riawork/opendoc/DemoEventPublisher/RIAWORKEVENT；第二个参数是 Event 的属性，其实也可以看出是用于传递 Event 的参数，这样可以方便事件订阅者通过获取属性值来处理事件。

实际应用的一个简单的例子如下：

```
EventAdmin
```

```
eventAdmin=(EventAdmin)context.getService(context.getServiceReference(EventAdmin.class.getName()));

        eventAdmin.postEvent(new Event("org/riawork/RIAWORKEVENT",new Hashtable());
```

在本文附带的代码的 EventAdmin 目录下可以找到更为详细的代码，在使用前请从 Equinox 的网站下载 EventAdmin 的实现并放入 Eclipse 的 plugins 目录中。

7.2.5.3. 处理自定义的事件

对于处理自定义的事件，需要涉及的对象同样有三个：

- EventHandler

EventHandler 接口中有 handleEvent(Event event)方法，通过完成此方法可完成事件的处理，如何订阅自己感兴趣的事件呢，通过在注册 EventHandler 的实现为 EventHandler 服务时通过属性进行控制。

- EventConstants

EventConstants 中有 EVENT_TOPIC 属性和 EVENT_FILTER 是常用的，EVENT_TOPIC 用于控制订阅的事件的主题名，主题名的值中可以采用*通配符，如 org/riawork/*，但不支持这样的格式 org/riawork/RIA*；EVENT_FILTER 用于过滤订阅的事件的属性。

- Event

通过 event.getProperties 获取 Event 的属性值，以进行事件的处理。

一个简单的事件处理器的例子如下：

```
public class Activator implements BundleActivator,EventHandler {

private static final String[] topics=new String[]{"org/riawork/*"};

        private ServiceRegistration serviceReg=null;

        public void start(BundleContext context) throws Exception {

                Dictionary props=new Hashtable();

                props.put(EventConstants.EVENT_TOPIC, topics);
```

```
serviceReg=context.registerService(EventHandler.class.getName(), this, props);
}

    public void handleEvent(Event event) {
        System.out.println(event.getTopic());
    }
}
```

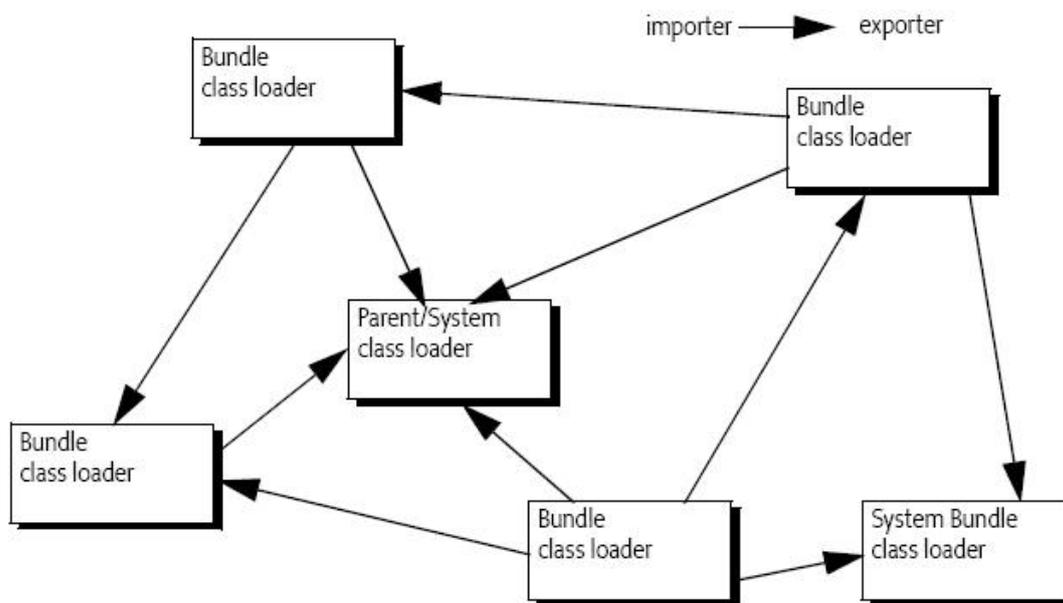
OSGI 规范中还定义了非常多值得关注的 Service 以支撑基于 OSGI 的应用，例如 Http Service、Log Service、Preferences Service、Metatype Service、XML Parser Service 等等，这些 Service 的详细介绍请大家参见《OSGI Service Platform Release 4》。

7.3. OSGI 关键部分讲解

7.3.1. ClassLoader

在 Java 中 ClassLoader 是非常重要的概念，而大家也知道，JVM 本身在 ClassLoader 上并没有提供非常强的功能，象对于模块开发非常重要的模块隔离 ClassLoader 的机制，对于版本升级非常重要的版本加载机制等，当然，基于 JVM 现有的 ClassLoader 可以来实现这些，OSGI 基于 JVM ClassLoader 形成模块隔离 ClassLoader 的机制，同时也增强了 ClassLoader 按版本加载、属性过滤等多种功能。

关于基于 OSGI 的系统的 ClassLoader，OSGI 规范中有张图可以解释：



图表 10 OSGI ClassLoader Delegation Model

在 OSGI 中，ClassLoader 可以通过下面几个区域来加载类和资源：

- Boot Class path

Boot Class path 中包含了 java.* 以及其实现的包。

- Framework Class path

框架通常会为其实现的类建立一个单独的 ClassLoader。

- Bundle Space

Bundle Space 则包含了与这个 Bundle 有关的所有 jar 文件。

Class space 是指通过一个给定的 Bundle 的 Classloader 可以获取的类，对于一个 Bundle 而言，它的 ClassLoader 能加载的类包括：

- Parent Classloader 加载的类(在 OSGI 的实现中 Bundle ClassLoader 的 Parent ClassLoader 通常都是 Boot ClassLoader，这里能加载的类通常是 java.*)；
- 当前 Bundle Imported 的 packages；
- 当前 Bundle Required 的 Bundles 的类；
- Bundle 自己的 Classpath 中的类；
- 附加的其他 Bundle。

关于 OSGI 在对一个 Bundle 进行类加载的顺序请见图表 7 OSGI Class loading 流程图。

按照规范的这种翻译式的讲解可能会很晦涩、难懂，在下面还是以基于 OSGI 构建系统时会经常碰到的一些 ClassLoader 的问题来进行实际性质的讲解，可能会让大

家更加容易去理解 OSGI 的 ClassLoader 机制，在讲解问题之前还是先解释下 ClassNotFoundException 和 NoClassDefError 两个异常，这也是在 ClassLoader 加载类出现问题时常常碰到的两个异常，这两个异常的区别在于前者 ClassNotFoundException 是指通过 ClassLoader 加载不到所需要的类，而后者 NoClassDefError 是指通过 ClassLoader 已经找到了所需要的类，但找不到该类所依赖的其他的类，明白了这两种异常后来看实际使用 OSGI 搭建系统时常碰到的 ClassLoader 问题：

- 在 Bundle A 的 Build Path 中引用了 lib 的 xstream.jar，但在运行时当加载到 XStream 类时报 ClassNotFoundException

这是为什么呢，参照图表 7 OSGI Class loading流程图，可以看出Bundle的 ClassLoader在加载类时是通过MANIFEST.MF中的bundle-classpath属性的描述来加载Bundle私有的类的，而上面这个问题就出在这了，xstream.jar是bundle私有引用的jar，那么就需要打开MANIFEST.MF文件，把它加入到Bundle-Classpath属性中，这其实是个习惯的问题，因为在通常的project的开发中我们都习惯把引用的lib加入到classpath，可能新手的话就会忘了同步MANIFEST.MF的改动，在Eclipse V3.2 的版本中在MANIFEST.MF中指定Bundle-Classpath属性的内容可自动同步到Build Path中去，以后要给Bundle引新的lib的时候可以直接到MANIFEST.MF中去改，然后选择自动同步就可以了。

在实际使用的过程中还有一个要养成的习惯就是如果要引用其他 Bundle 的类的时候，不要直接的引用那个 Bundle 的工程，而是通过 MANIFEST.MF 的 Import-Package 属性导入所需要的类的包。

- Bundle A 是个小的 Web 应用，使用了 Spring 完成业务逻辑处理的部分，但在运行过程当 Spring 加载配置在 xml 中的 bean 时报出了 ClassNotFoundException 这个是在基于 OSGI 搭建 B/S 应用时与一些开源产品集成时会碰到的典型问题，这个问题不完全算 OSGI ClassLoader 的问题，但因为是基于 OSGI 搭建 B/S 应用才会碰到的，所以还是放在这里来讲讲了。

会出现这个问题的原因是 Spring 在加载 xml 中 bean 的类时采用的是获取当前线程 ClassLoader 的方法，而由于 Bundle A 是个 Web 应用，当在页面请求 Bundle A 调用 bean 的时候，HttpService 的实现是要启动线程监听的，这个时候当前

线程的 `ClassLoader` 就变成 `HttpService` 实现的那个 `Bundle` 的 `ClassLoader`，这个时候 `spring` 再用当前线程的 `ClassLoader` 去加载类，自然就会找不到了，所以就抛出 `ClassNotFoundException` 了。

这个问题的解决方案有两种，一种是抛弃 `Spring`，如果你只是使用 `Spring IoC` 的话那么就可以选择这种，因为 `OSGI R4` 的 `DS` 本来就具备 `IoC` 的特性；另外一种修改 `Spring` 的 `ClassUtils` 类，将其使用当前线程 `ClassLoader` 加载的方法改为使用当前类的 `ClassLoader` 进行加载。

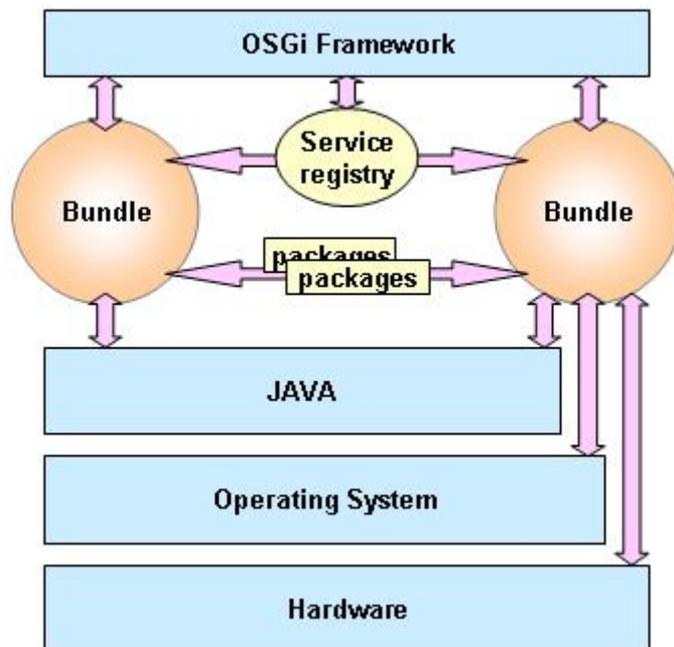
这个问题在使用 `Axis` 的时候也会碰到，同样的做法，修改它的 `ClassUtils` 类。其他的 `Bundle` 的 `ClassLoader` 的问题应该会碰得很少，碰到时只要参照图表 7 OSGI Class loading 流程图看看就差不多了，记住 `OSGI` 对于每个 `Bundle` 采用的是独立的 `ClassLoader` 机制，同时 `Bundle` 可以通过象 `Import-Package`、`Require-Bundle` 的方式引用其他 `Bundle` `Export` 的 `Package` 就可以了。

7.3.2. Bundle 的生命周期

具体见 `Lifecycle Layer`。

7.3.3. Bundle 的通讯机制

现在大家都知道，`OSGI` 对于每个 `Bundle` 采用的是独立的 `ClassLoader` 的，那么 `Bundle` 之间是怎样通讯的，在之前解释 `OSGI R4` 规范的 `Module Layer` 中也有部分的介绍，在这里来结合个人的实际使用来进行更为形象点的解释，`Bundle` 的通讯机制在 `OSGI Component Programming` 这份 PPT 中有个不错的图进行了解释：



图表 11 OSGI Bundle 通讯机制（摘自 OSGI Component Programming PPT）

从这张图中可以很清楚的看出 Bundle 之间通过 Service 和 Packages 两种方式进行通讯, Packages 方式是通过定义 Bundle 的 Import-Package 和 Export-Package 来实现, Packages 方式看起来更适合设计时的定义 Bundle 的依赖, 当然, 也可使用类如 DynamicImport-Package 的方式来实现动态的引用, Packages 的机制是的引用其他 Bundle 中的类成为了可能, 那么 Service 机制呢, 通过 Service 机制可以实现引用其他 Bundle 中提供的类的实例, 而 Service 机制同时还保证了基于面向接口的方式能够更加的简单的去完成, 举例来说:

- Bundle A 需要使用实现了 org.riawork.demo.Opendoc 接口的类, 在 OSGI 中一种典型的做法就是:

将 org.riawork.demo.Opendoc 做为一个单独的 Bundle, 并 Export 中 org.riawork.demo 包, Bundle A 引用 org.riawork.demo 包, 采用如下的方式调用实现了接口的实例:

```
Opendoc
service=(Opendoc)context.getService(context.getServiceReference(Opendoc.class.getName()));
```

可以看到在这样的方式中, Bundle A 和实现 Opendoc 接口的类的 Bundle 完全没有发生任何的关系, 真正的面向接口的编程得以实现。

而如果不采用 Service 的方式的话, 只能让实现了 Opendoc 接口的类的 Bundle Export 出实现类的 package, 然后 Bundle A Import 这个 package, 通过类似 Opendoc opendoc=new OpendocImpl(); 的方式来获取 Opendoc 的实现, 这样的方式还使得接口实现的依赖在程序中已经确定, 我们知道这是 IoC 反对的重点, 而在 OSGI 中使用 service 的话不仅不会造成这种现象, 而且还能做到实现了此接口的类的装载是动态的, 不像很多传统的 IoC 容器是设计期就确定的。

可以看出, 在 OSGI 中要实现 Bundle 之间的通讯并不是什么难事, 而且正因为 Bundle 的通讯机制是动态的, 从这方面 OSGI 保证了基于其构建的系统可在运行期动态的改变行为, 而在 OSGI 框架具备了 DS 的实现后 Bundle 间的通讯就更为容易去实现了。

7.3.4. DS 中 Component 的生命周期

DS 中的 Component 的生命周期是如何被控制的, 尽管对于 Component 的控制 DS 是提供了接口可以通过程序来实现控制的, 但在实际的系统中基本都是交由 OSGI 框架去控制, 而很少通过程序主动去控制, 仍然举个例子来看看 DS 中 Component 的生命周期:

- Component Opendoc 是 Bundle A 中的一个 Component;
- 当 Bundle A 启动时, Component Opendoc 的配置文件的信息被加载并解析, 此时 Component Opendoc 的状态设置为 Enabled;
- 之后根据 Component Opendoc 的配置寻找引用的服务, 如引用的服务的条件均满足了, 那么此时 OSGI 将此 Component 的状态设置为 Satisfied; 如引用的服务的条件未满足, 此时 OSGI 则将此 Component 的状态设置为 Unsatisfied;
- 当 Component Opendoc 被调用时, 如 Component 的状态为 Satisfied, 那么 OSGI 将激活此 Component, 首先设置 Component Opendoc 的依赖, 如 Component Opendoc 中存在 activate(ComponentContext context)方法, 则调用此方法。
- 当 Bundle 停止时 OSGI 将通知引用了 Component Opendoc 提供的服务的 Component, 如那些 Component 必须引用 Component Opendoc 提供的服务, 那么 OSGI 将调用该 Component 的 deactivate(ComponentContext context)(如果存在)方法, 之后把该 Component 的状态设置为 Unsatisfied, 在完成了这些工作后, OSGI 将调用 Component Opendoc 的 deactivate 方法, 之后将 Component

Opendoc 的状态设置为 Disabled。

从上面的描述可见，Component 的生命周期完全是动态的，也因为这个原因在使用 Component 时当发现 Component 并没被激活也不是什么很奇怪的事，这个时候就可以去查查 Bundle 是否启动了，Component 所必须的依赖在系统中是否可用，由于 Component 的变化完全是动态的，所以依靠跟踪去判断错误比较困难，大多数时候能采取的方法就是通过启动 Equinox 对于 LogService 的实现，然后在控制台中输入 log 来查看 ds 的日志；另外的方法就只能推导原因了。

7.3.5. DS 中 Component 的通讯机制

DS 中 Component 的通讯通过服务的方式来完成，所以 DS 称自己为 Component-Oriented Service Model，关于 DS 中 Service 的详细描述请见 **Declarative Services** 中 Service 的相关章节。

八. 应用 OSGI

之前的章节中对于 OSGI 的应用都只是让大家对于 OSGI 的实战有个较为实际的概念，并没有在整个系统级别去应用 OSGI，要在整个系统级别上应用 OSGI，还是会带来不少的挑战的，但相对于 OSGI 能带来的优势，相信这是值得的，在应用 OSGI 时应最大程度的发挥基于 OSGI 搭建系统的优势以及减少基于 OSGI 系统带来的挑战。

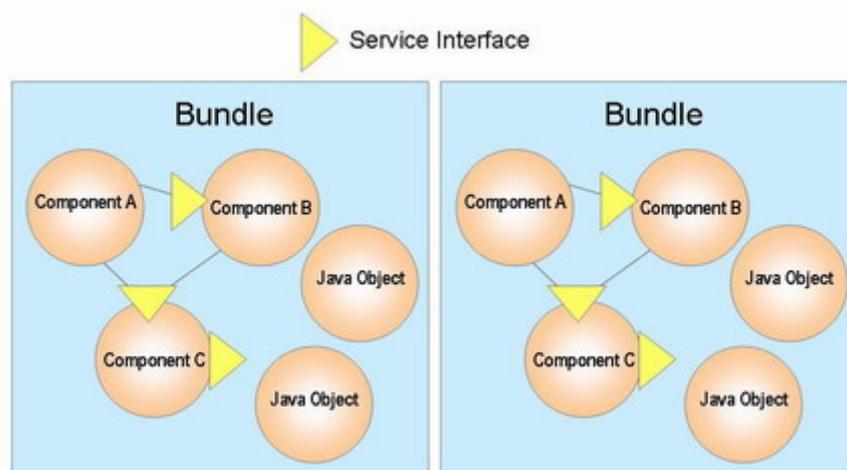
由于基于 OSGI 搭建系统带来的是架构级别的改变，带来的最大的影响莫过于设计层面，同时对于系统开发和部署也产生了影响。

先说说设计层面的，主要是模块化设计、面向服务的组件模型设计以及动态性的设计三个方面，只有在把握好了这三方面才能充分的发挥基于 OSGI 搭建系统的优势，否则也许会给项目或产品带来更大的痛苦。

8.1. 模块化设计

尽管大家在设计大部分的系统时候确实是按照模块化的思想去进行的，但可能不会考虑的象 OSGI 定义 Bundle 那么的清晰，但这点相信大家都能很快的适应，毕竟模块化的思想大致仍然是相同的，基于 OSGI 就可以按照统一的标准进行模块的设计，同时也会提升在模块设计时对于模块的输入（依赖）、输出（功能）、扩展的关注，从而使得模块的设计更加的完善和规范。

基于 OSGI 搭建的系统架构通常类似如下：



图表 12 基于 OSGI 搭建的系统架构示意图

可以看出基于 OSGI 搭建的系统由 Bundle 组成，而每个 Bundle 由多个 Component 或 Java Object 共同组成，每个 Component 又有可能引用或暴露了多个 Service，每个 Bundle 以及 Component 均可在其生命周期状态改变时做出相应的行为。

8.2. 面向服务的组件模型设计

模块化设计可以看成是架构级别的设计方式，而面向服务的组件模型设计则是对模块进行详细设计时的核心思想了，在 OSGI 中的模块划分为不同的 Component 和 Java Object 共同组成，Component 和普通 Java Object 不同的地方在之前的章节中已经做过解释了，在这就不多说了，Component 通过引用 Service 或暴露 Service 来完成模块中用例的实现，应该说，这种方式与传统的设计方法也没有太多的差别，面向服务的组件模型设计更加强调在设计时分解模块中用例的实现（形成组件和服务）以及组件依赖的关注。

基于 OSGI 框架构建模块、Component、Service 的依赖时，体现出了超越传统 IoC 容器的优势，在依赖的构建上 OSGI 更是提供了多种灵活的依赖控制方式，例如属性过滤、版本过滤以及延迟加载设置等。

8.3. 动态性设计

从 OSGI 规范的定义来讲，基于 OSGI 搭建的系统保持动态性是其最大的优势，包括模块、Component、Service 的依赖都具备动态性的特征，在构建模块、Component、Service 的依赖时 OSGI 框架采用的都是动态的方式实现，这也就是说 OSGI 框架本身保证了基于其实现的系统的动态性，但这并不意味着基于 OSGI 框架实现的系统就一定具备动态性特征，就像基于 JAVA 编写的系统不一定跨平台一样。

要充分的发挥基于 OSGI 的动态性，就要完全的采用面向接口的设计方式，而不是去依赖实现，要记住基于 OSGI 搭建的系统是在运行期才构成依赖的，这和传统的系统有很大的差别，所以在设计基于 OSGI 的系统时特别要注意依赖不要在设计时就定死，而应该在运行期由 OSGI 框架自动的注入。

OSGI 提供的对于依赖的各种管理策略（如版本过滤、多样性设置等）在保证系统动态性的同时也保证了系统获取到所需的模块、Component 或 Service。

8.4. 面向接口的开发

基于 OSGI 能让我们在开发时真正的做到面向接口的开发，尽管面向接口的开发的思想已经被几乎所有的开发人员所接受，但由于有些时候偷懒或者无意中就直接去引用了实现的类，尽管象 `A a=new AImpl()` 这样的代码在 Factory 模式、IoC 容器得到认同后出现的已经很少了，但无论是采用 Factory 模式还是 IoC 容器都是在设计期隐性的对实现设置了直接的依赖，而非运行期去寻找可用的依赖的，而同样由于 `A a=new AImpl()` 还是可以直接的写，导致了有些时候还是会出现这样的强制依赖的代码，而

OSGI 提供为每个 Bundle 提供独立的 ClassLoader 机制的方法则让我们可以真正的做到面向接口的开发，Equinox 本身就给出了一个这样的例子，在 Equinox 中将 OSGI 中所有服务的接口都统一放到了 OSGI Services Bundle 中，由该 Bundle 对外 Export 这些 Packages，而对于这些 Service 的实现则由其他单独的 Bundle 去实现，那些 Bundle 就无需对外提供 Package 了，这样就真正的隐藏了接口的实现，而加上 OSGI 的动态性，使用这些接口的类也只有在被使用的时候才会寻找相应可用的依赖，所以在基于 OSGI 框架进行开发时应该养成对外 Export 接口的 packages，而隐藏实现接口的 packages，更为好的方法就是把接口单独的放入一个 Bundle 中，这样对于更换接口的实现就更为方便了。

对于系统开发方面，产生的影响主要是让单元测试显得更为重要了，毕竟所有的依赖都是动态设置的，要依靠集成测试来发现问题就显得更为麻烦了，所以基于 OSGI 的系统会更多的依赖 Mock 进行单元测试，其他的影响就是不再通过引用 lib 或 project 的方式来设置对于其他工程的依赖等等。

对于系统部署方面，产生的影响是现在的部署是多工程共同部署的方式，这个时候自动化的部署脚本就显得非常有必要了。

在 OSGI 应用的好的情况下，基于它搭建的系统应具备统一的模块化构建和管理、可插拔、可积累以及可动态改变和扩展行为的特征。

九. OSGI 资源

OSGI 目前国内的资源相对还是比较的少, 国外的资源主要是 OSGI 的官方网站、Equinox 网站以及其他的一些开源 OSGI 框架 (例如 Oscar、Knopflerfish) 的网站。

- 网站

中文方面的主要有 www.riawork.org

英文方面的主要有 www.osgi.org、www.eclipse.org/equinox

- Blog

OSGI 官方 Blog: www.osgi.org/blog

OSGI 主席 Peter 的 Blog: www.aqute.biz

- 邮件列表

Equinox 邮件列表以及 OSGI 官方的邮件列表。

十. OSGI 框架前瞻

从基于 OSGI 框架搭建应用系统级别的视角来看目前的 OSGI 框架在这几方面还需要进一步改进:

- 对于 B/S 结构系统支持的不足

目前的 OSGI 框架对于 B/S 结构的系统支持仍然显的非常的不足, 这点也是现在 Equinox 的重要提升点, 考虑如何更好的和应用服务器的集成是 Equinox 的重点, 一定程度上这也会影响到 equinox 的接受度, 毕竟目前大部分的系统都是 B/S 结构的。

- 管理端不够强大

目前 OSGI 框架提供的管理端不够强大, 现在的管理端中仅提供了基本的 Bundle 状态管理、日志查看等功能, 象动态修改系统级别的配置(config.ini)、动态修改 Bundle 的配置(Manifest.mf)、启动级别等功能都尚未提供, 而这些在实际的项目或产品中都是非常必要的。

- 缺少基于微核统一管理应用系统的功能

基于微核统一管理应用系统是指可以通过微核来统一控制应用系统的启动、停止或更新, 在这样的情况下, 只要微核的运行是稳定的, 那么系统就可以一直的处于运行状态, 对于用户而言从项目启动的那天就可以看到一个在不间断运行的系统。

至于框架中缺少构建应用级别系统的通用 Bundle 的这个问题, 倒是可以通过依靠大家的力量来完成, 大家可以把在项目中使用的象持久层处理 Bundle、缓存处理 Bundle 等等贡献出来。

十一. OSGI 带来的遐想

基于 OSGI 搭建系统带来了不同的系统设计和开发的方式，OSGI 可以带动 Java 界模块化级别设计思想的统一，这点已经得到了证明，尽管 JSR277 目前未接受 OSGI 作为其规范，但 OSGI 带来的影响是必然的，也许在以后的某个 Java 版本发布的时候，大家就会按照同样的方式去设计模块以及模块间的依赖，编写模块中的 Component 和 Service，也许以后可以从网站上下载各种各样的 Bundle，从而搭建成自己所需要的系统，而每个公司在搭建自己新项目的脚手架甚至是原型系统时，可以直接从公司的 Bundle 库中获取相应的 Bundle 来搭建，对于公司的积累而言，无疑这是非常有利的。

基于 OSGI 的 C/S 结构的系统也许就能做成象硬件一样卖给用户，甚至可以同样的提供象路由器中一样的热插拔式的硬件模块，又或者可以采用 Server 端统一管理 Client 端功能模块的方式，由 Server 端分发、更新 Client 端的功能模块。

基于 OSGI 还会给我们带来更多更大的惊喜，都等待着大家一起去发掘.....

十二. 参考文献

- 《OSGI Service Platform Release 4》
- [OSGI Technology Introduction](#)
- 《OSGI Component Programming》PPT