OpenDoc Series’

OSGi 进阶

V 1.0

作者：BlueDavy

So many open source projects. Why not **Open** your **Doc**uments? ☺

文档说明

参与人员：

作者	Blog	联络
BlueDavy	http://www.blogjava.net/bluedavy	BlueDavy@gmail.com

本文中例子的代码请从以下地址下载：
http://www.osgi.org.cn/opendoc/osgiopendoc2-source.zip
本文中例子的可运行版本请从以下地址下载：
http://www.osgi.org.cn/opendoc/osgiopendoc2-dist.zip

发布记录

版本	日期	作者	说明
0.8	2007-08-02 至 2007-09-29	BlueDavy	国庆预览版，完成 Opendoc 的主体内容的编写；
1.0	2007-10-14	BlueDavy	文笔润色； 增加了一些过程描述的图例； 补充设计模式和最佳实践章节内容； 根据预览版收集的反馈意见进行修改完善。

OpenDoc 版权说明

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**
- **OSGi 实战**
- **OSWorkflow 中文手册**

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://www.redsaga.com获取最新更新信息。

目录

1. 序

1.1. 读者对象

此篇文档适合具备OSGi基础概念或已阅读《OSGi实战》的读者[1]阅读，尤其是希望将OSGi应用至实际的商业项目/产品的读者。

阅读此篇 Opendoc 的读者应具备以下知识：

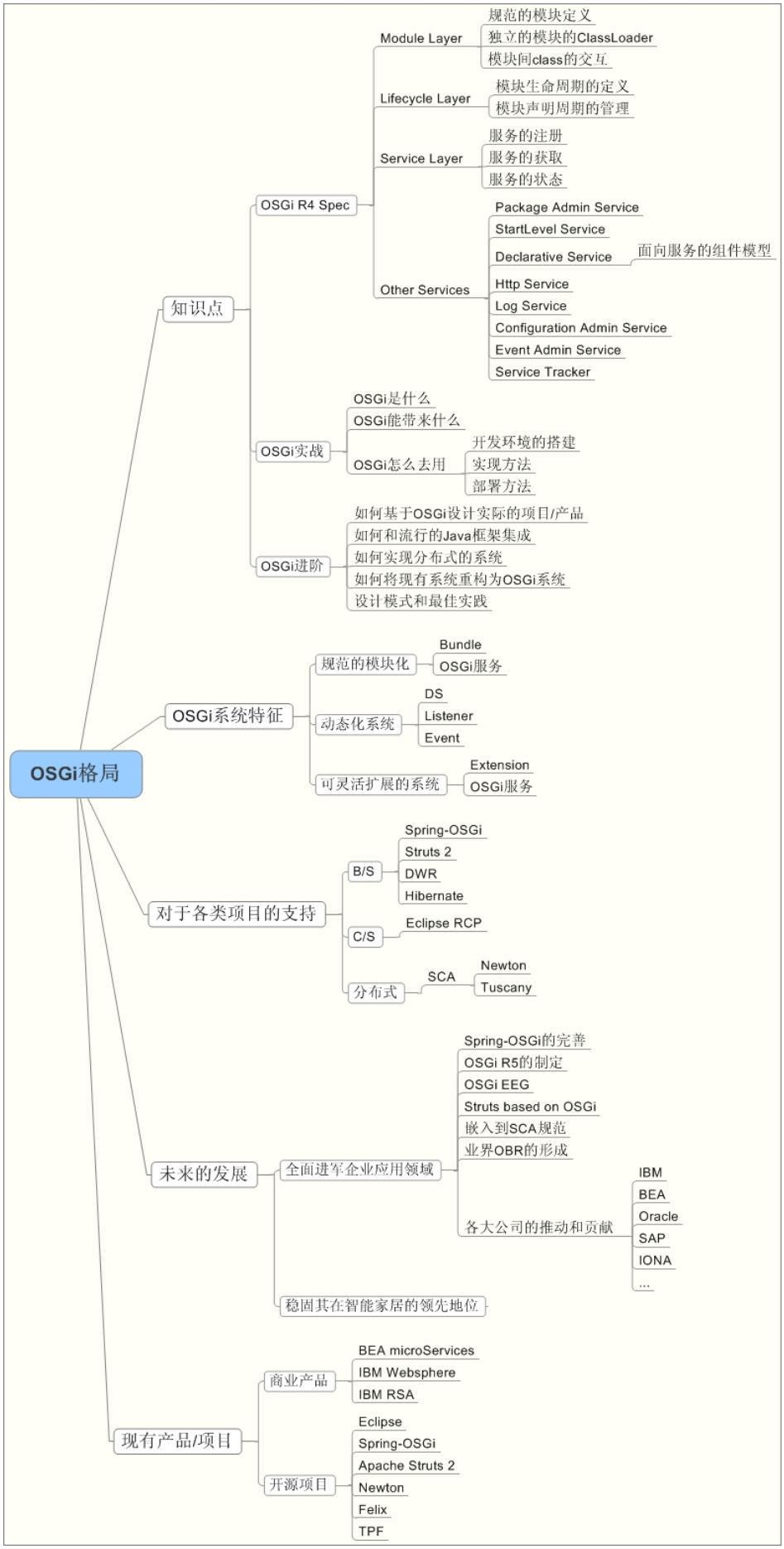
- 如何在 Eclipse 中搭建 OSGi 应用开发环境；
- 如何部署 OSGi 应用；
- 对于 OSGi 应用开发有大致的了解（如 DS 怎么使用）。

1.2. 编写目的

《OSGi 实战》Opendoc 发布已有一年多了，《OSGi 实战》Opendoc 作为一篇 OSGi 入门级的 Opendoc，主要是起到介绍 OSGi 的作用，从而吸引更多的人认识 OSGi 和对 OSGi 产生兴趣，很高兴《OSGi 实战》在过去的一年中得到了很多朋友的关注，更值得高兴的是短短的一年来 OSGi 取得了很大的进步，几乎可以称得上是 OSGi 发展历史上具备里程碑意义的一年。

OSGi 目前的现状格局从下图中可以管窥一二：

[1] 如尚未阅读《OSGi实战》Opendoc，可从此处下载：
http://www.osgi.org.cn/opendoc/OSGI_Opendoc.rar


OSGi格局
知识点
OSGi R4 Spec
Module Layer
规范的模块定义
独立的模块的ClassLoader
模块间class的交互
Lifecycle Layer
模块生命周期的定义
模块声明周期的管理
Service Layer
服务的注册
服务的获取
服务的状态
Other Services
Package Admin Service
StartLevel Service
Declarative Service
面向服务的组件模型
Http Service
Log Service
Configuration Admin Service
Event Admin Service
Service Tracker
OSGi实战
OSGi是什么
OSGi能带来什么
OSGi怎么去用
开发环境的搭建
实现方法
部署方法
OSGi进阶
如何基于OSGi设计实际的项目/产品
如何和流行的Java框架集成
如何实现分布式的系统
如何将现有系统重构为OSGi系统
设计模式和最佳实践
OSGi系统特征
规范的模块化
Bundle
OSGi服务
动态化系统
DS
Listener
Event
可灵活扩展的系统
Extension
OSGi服务
对于各类项目的支持
B/S
Spring-OSGi
Struts 2
DWR
Hibernate
C/S
Eclipse RCP
分布式
SCA
Newton
Tuscany
未来的发展
全面进军企业应用领域
Spring-OSGi的完善
OSGi R5的制定
OSGi EEG
Struts based on OSGi
嵌入到SCA规范
业界OBR的形成
各大公司的推动和贡献
IBM
BEA
Oracle
SAP
IONA
...
稳固其在智能家居的领先地位
现有产品/项目
商业产品
BEA microServices
IBM Websphere
IBM RSA
开源项目
Eclipse
Spring-OSGi
Apache Struts 2
Newton
Felix
TPF

图表 1 OSGi 格局

在这一年 OSGi 成功进入企业应用领域，相继受到各大厂商的认可和采用，例如 IBM 的 Websphere V6.1、BEA 更是将其所有产品都重构为基于 OSGi 而开发的 microServices 上等等；在开源界 OSGi 也是引起了很大的反响，例如 Spring 和 OSGi 的结合、Apache 宣布 Struts 2 要采用 OSGi 等等。

OSGi 联盟很好的抓住的了这样的机会，成立了专门的 EEG 小组，以更好、更快的推进 OSGi 在企业领域的发展，而同时在 EclipseCon2007 上，OSGi 也成为了主要的话题，这对在整个企业 IT 界中推广 OSGi 起到了巨大的作用。

在 Java 规范领域，OSGi 成为了 JSR291 的规范标准，尽管没有成为 JSR277 的标准，但一定程度上还是反应了 OSGi 已经得到了各大厂商的认可。

在这样的大环境下，OSGi 在国内也受到了越来越多的关注和认同，不少朋友也都有了将 OSGi 使用到实际的项目/产品中去的想法，其中有部分朋友已经在实际的项目/产品中开始使用 OSGi，但其中更多的朋友由于这样那样的原因暂时放弃了在实际项目/产品中使用 OSGi 的想法，总结下来主要有这么几点原因：

- 基于 OSGi 怎么来设计/实现/部署/测试项目和产品

  OSGi 对于模块化有严格而规范的定义，这对于传统的设计/实现/部署/测试模式都会有一定的影响，而同时如何来充分发挥 OSGi 带来的动态性和可扩展性也是关键的问题，对于系统的架构设计者而言会非常的关心这个问题，否则的话采用 OSGi 就没有任何意义了，但从目前可获取到的 OSGi 资源中很难找到这方面的指导，这成了很多架构设计者不敢冒险选择 OSGi 应用到实际项目/产品中的原因。

- OSGi 怎么和流行的 Java 领域的 B/S 体系架构集成

  这个问题困扰了很多的朋友，因为如果 OSGi 无法和流行的 Java 领域的 B/S 体系架构集成的话，那也就意味着如果采用 OSGi 的话，项目/产品的很多基础框架都得从头再来，同时也意味着整个团队的知识体系得重建，这肯定是不可取的。

  而从各方面 OSGi 的资源中确实找不到集成的方法，因此这也成了很多朋友放弃在实际的项目/产品中使用 OSGi 的原因。

- 基于 OSGi 怎么来实现分布式的系统

  这个和做 B/S 系统的朋友的疑问是一样的。

- 怎么把原有的项目/产品部署为 OSGi 应用

  有部分项目可能是长期的项目，产品的话就肯定是长期发展的，这些项目/产品都

已经有一定的积累了，不可能因为要重构为 OSGi 应用而推倒从来，从已有的 OSGi 资源中又获取不到相关的指导，因此这也成为了很多朋友不得不放弃使用 OSGi 的原因。

作为 OSGi 的拥护者和实践者，我也非常希望能有越来越多的人能够将 OSGi 应用到实际的项目/产品中去，《OSGi 实战》Opendoc 吸引了不少的人关注 OSGi，但由于上面的这些原因大家没有把 OSGi 应用到实际项目/产品上去，我感到非常的遗憾，因此有了编写一篇新的 Opendoc 的想法，把自己有限的知识和经验分享给大家，尽量的解除大家的担忧和忧患，而将 OSGi 应用到实际的项目/产品中去，这也是本篇 Opendco 编写的目的。

随着本篇 Opendoc 也会诞生出一些的 OSGi 开源项目，另外目前我已将在商业产品中使用的插件管理框架开源出来了，详细信息请大家访问以下地址：

http://www.blogjava.net/BlueDavy/archive/2007/10/05/150598.html

1.3. 导读

本篇 Opendoc 遵照着让大家放心的将 OSGi 使用到实际的项目/产品的指导思想而编写，以实际的例子来解答大家心中的疑问。

- 解答如何基于 OSGi 怎么来设计/实现/部署/测试项目和产品

  Opendoc以此开篇，以一个留言板系统实例讲解了OSGi应用在设计时应把握的几个重点原则，并遵照设计完成留言板系统的设计/实现和部署，具体内容请参见基于OSGi的留言板。

- 解答 OSGi 怎么和 Java 领域流行的 B/S 体系架构进行集成

  Opendoc中详细讲解了如何将OSGi与Hibernate、Spring以及Webwork进行集成，并由此诞生了OSGi+Hibernate+Spring+Webwork（OHSW）的脚手架，基于此脚手架完成了对于留言板系统的重构，具体内容请参见与流行的Java B/S体系架构进行集成。

- 解答基于 OSGi 怎么来实现分布式的系统

  Opendoc中详细讲解了怎么实现分布式的OSGi应用的通讯，并由此诞生了一个简单的分布式OSGi应用通讯的脚手架，基于此脚手架完成了对于留言板系统的重构，具体内容请参见基于OSGi搭建分布式系统。

- 解答怎么把原有的项目/产品部署为 OSGi 应用

  Opendoc中详细介绍了将一个传统的基于Hibernate+Spring+Webwork的留言板系

统重构为部署至OHSW的OSGi应用的步骤，以此说明如何将原有的额项目/产品部署为OSGi应用，并重构原有代码使其具备OSGi应用的模块化、动态化以及可扩展性的特征，具体内容请参见将原系统重构为基于OSGi的系统。

在解答了上面的几点问题后，为了能够让大家更好的使用 OSGi，在最后的章节中总结了一些 OSGi 应用的设计模式和最佳实践。

1.4. 致谢

在编写这篇 Opendoc 的过程中，得到了很多朋友无私的支持，在此略表谢意：

Kyang：感谢你对于预览版的反馈意见；

Jlinux（唐勇）：感谢你对于预览版的反馈意见；

Caoxg（曹晓刚）：感谢你对于预览版的反馈意见和封面的改进；

Xiaodao：感谢你的新闻报道；

霍泰稳：感谢你将此文档制作为InfoQ中文站迷你书（近期发布）并宣传；

还有其他未在此列出名字的关注此文档的朋友们，在此一并表示感谢。

2. 基于 OSGi 的留言板

2.1. 需求

此留言板主要为展示基于 OSGi 的 B/S 系统的通用设计和实现方法，因此功能方面只是一些基础功能的需求：

- 分页浏览留言

  进入留言板的首页时，以每页 20 条的列表的方式列出系统中所有的留言，并提供下一页、上一页、首页和末页的链接导航；

  留言列表按留言时间顺序显示，最新的留言显示在最前；

  每条留言显示的内容分别为：留言内容、留言人和留言时间。

- 新增留言

  进入留言板的首页后，用户通过点击新增留言进入新增留言页面；

  在新增留言页面中填入留言人、留言内容，此两项均为必填项，填写完毕后点击保存完成新增留言动作，或点击返回放弃新增留言。

- 删除留言

  管理员首先进入留言板的管理功能的登录页面，输入管理员用户名和密码登录管理页面；

  管理页面和分页浏览页面相同，只是在每条留言的最后增加了删除留言的链接。

另外，此留言板还要求具备可扩展性，如增加编辑留言功能、回复留言功能、搜索留言功能等。

2.2. 基于 OSGi 的留言板的设计

2.2.1. OSGi 框架的基础功能和设计思想

此留言板基于 OSGi 而实现，我们可以把 OSGi 看做为一个框架（尽管实际上它不仅仅是框架），此框架提供的基础功能有：

- 支持模块化的动态部署

  基于 OSGi 而构建的系统可以以模块化的方式（例如 jar 文件等）动态的部署至框架中，从而增加、扩展或改变系统的功能。

  要以模块化的方式部署到 OSGi 中，必须遵循 OSGi 的规范要求，那就是将工程建设为符合规范的 Bundle 工程（就是 Eclipse 中的插件工程），或使用工具将工程打包成符合规范的 Jar 文件，在后续的章节中会介绍如何将一个传统的 Java 工程

打包生成为符合 Bundle 要求的 Jar 文件，从而顺利的部署到 OSGi 上运行。

- 支持模块化的封装和交互

  OSGi 支持模块化的部署，因此可以将系统按照模块或其他方式划分为不同的 Java 工程，这和以往做 Java 系统时逻辑上的模块化是有很大的不同的，这样做就使得模块从物理级别上隔离了，也就不可能从这个模块直接调用另外模块的接口或类了，那么这个时候应该怎么去实现模块间的互相调用呢？

  根据 OSGi 规范，每个工程可通过声明 Export-Package 对外提供访问此工程中的类和接口，也可通过将需要对外提供的功能声明为 OSGi 的服务实现面向接口、面向服务式的设计。

  还有一种方法来实现模块间的交互，那就是基于 OSGi 的 Event 服务来对外发布事件，订阅了此事件的模块就会相信的接收到消息，做出处理。

- 支持模块的动态配置

  OSGi 通过提供 Configuration Admin 服务来实现模块的动态配置和统一管理，基于此服务各模块的配置可在运行期进行增加、修改和删除，所有对于模块配置的管理统一调用 Configuration Admin 服务接口来实现。

- 支持模块的动态扩展

  基于 OSGi 提供的面向服务的组件模型的设计方法以及 OSGi 实现框架提供的扩展点方法可实现模块的动态扩展。

要使用 OSGi 框架提供的这些基本功能，在设计系统时就要遵循 OSGi 框架的设计思想：

- 模块化的设计

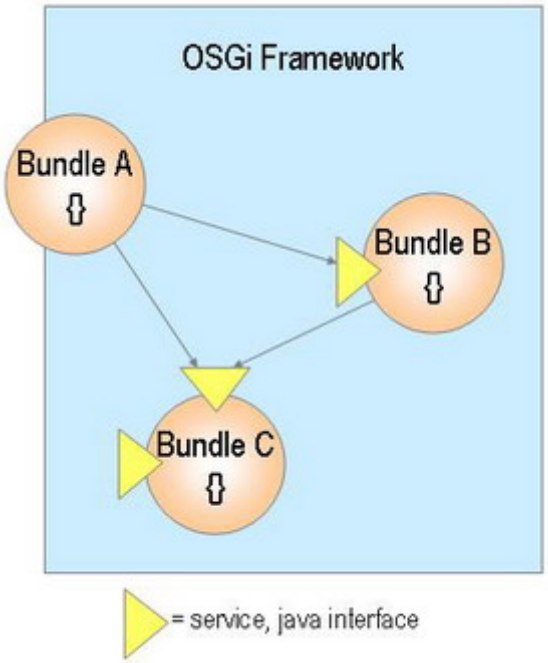
  模块化的设计已经是大家在做系统设计时遵循的基本设计原则，但只有基于OSGi来做模块化的时候才会真正的体验到何谓模块化[2]，因为OSGi中的模块化是物理隔离的，而不基于OSGi的话很难做到物理隔离方式的模块化实现，也就很难使系统真正的做到模块化，通常切换到基于OSGi后就会发现以前的模块化设计做的还是很不足。

  基于 OSGi 进行模块化的设计时和传统做设计时的模块设计并没有多大的差别，均为定义模块的范围、模块对外提供的服务和所依赖的服务，相信大家在这点上

[2] 此话来源于Bea microServices的研发人员在EclipseCon2007 的讲话，原话是：Until modularity is not enforced,it is not there.

很容易适应，在 OSGi 中只是更为的规范，更为的遵循面向服务的设计思想。

在 OSGi 中模块由一个或多个 Bundle 构成，模块之间的交互通过 Import-Package、Export-Package 以及 OSGi Service 的方式来实现。



图表 2 OSGi 模块及交互模型

- 面向服务的组件模型的设计

  面向服务的组件模型（Service-Oriented Component Model）的设计思想是 OSGi 的核心设计思想，OSGi 推崇系统采用 Bundle 的方式来划分，Bundle 由多个 Component（组件）来实现，Component 通过对外提供服务接口和引用其他 Bundle 的服务接口来实现 Component 间的交互。

  从这个核心的设计思想上可以看出，基于 OSGi 实现的系统自然就是符合 SOA 体系架构的。

  在 OSGi 中 Component 以 POJO 的方式编写，通过 DI 的方式注入其所引用的服务，以一个标准格式的 XML 描述 Component 引用服务的方式、对外提供的服务以及服务的属性。

- 动态化的设计

  动态化的设计是指系统中所有的模块均需支持动态的插拔和修改，系统的模块需要遵循对具体实现的零依赖和配置的统一维护（基于 Configuration Admin 服务），在设计时要记住的是所依赖的 OSGi 服务或 Bundle 都是有可能动态的卸载或安装的。

对于模块的动态插拔和修改，OSGi 框架本身提供了支持，模块可通过 OSGi 的 Console（命令行 Console、Web console 等）安装、更新、卸载、启动、停止相应的 Bundle。

为保持系统的动态性，在设计时要遵循的原则是不要静态化的依赖任何服务，避免服务不可用时造成系统的崩溃，另外保证系统的“即插即用，即删即无”。

- 可扩展的设计

  OSGi 在设计时提倡采用可扩展式的设计，即可通过系统中预设的扩展点来扩充系统的功能，有两种方式来实现：

  - 引用服务的方式

    通过在组件中允许引用服务接口的多个实现来实现组件功能的不断扩展，例如 A 组件的作用为显示菜单，其通过引用菜单服务接口来获取系统中所有的菜单服务，此时系统中有两个实现此服务的组件，分别为文件菜单组件和编辑菜单组件，那么 A 组件相应的就会显示出文件菜单和编辑菜单，而当从系统中删除编辑菜单的组件时，A 组件显示的菜单就只剩文件菜单了，如此时再部署一个实现菜单服务接口的视图菜单组件模块到系统中，那么显示出来的菜单则会为文件、视图。

  - 定义扩展点的方式

    按照 Eclipse 推荐的扩展点插件的标准格式定义 Bundle 中的扩展点，其他需要扩展的 Bundle 可通过实现相应的扩展点来扩展该 Bundle 的功能。

  系统对于可扩展性的需求很大程度会影响到 Bundle 的划分和设计，这需要结合实际情况来进行设计。

2.2.2. 留言板的设计

留言板的设计遵循 OSGi 的设计思想以及职责单一的原则而完成。

2.2.2.1. 系统层次划分

首先根据留言板的体系结构来确定系统层次的划分。

根据留言板的功能需求，此留言板为一个 B/S 系统，对其功能需求进行简单分析可以看出此系统基本不涉及业务逻辑处理，因此在系统的层次上简单的划分为页面层+持久层两个部分，页面层负责信息的显示以及页面操作的提供；持久层负责信息的存储和获取。

2.2.2.2. 系统模块划分

根据留言板的功能需求并结合系统层次来划分系统模块，确定各模块对外提供的服务以及需要引用的服务。

模块划分的粒度没有明确的指导法则，需要根据设计师们的经验来确定，最基本的原则是职责单一的原则，由于留言板系统本身的功能较为简单，在模块的划分上就按照功能职责来划分，每个模块中直接包含了页面层+持久层的实现，不再划分为页面层的 Bundle 和持久层的 Bundle，按照留言板的功能需求将系统的模块划分为：

- 留言列表模块

  此模块负责实现将留言从数据库中分页提取，并显示至页面中。

  无对外提供的接口，同时也不需要引用外部的接口。

- 新增留言模块

  此模块负责提供留言编写页面，在用户提交后将留言信息存入数据库中。

  无对外提供的接口，同时也不需要引用外部的接口。

- 管理员登录模块

  此模块负责提供登录页面，在用户提交后查询数据库确认用户是否能够登录。

  无对外提供的接口，同时也不需要引用外部的接口。

- 删除留言模块

  此模块负责提供根据留言序号从数据库中删除留言的功能。

  无对外提供的接口，同时也不需要引用外部的接口。

功能模块的划分就是这么设计了，另外还需考虑的就是模块的部署问题了，在传统的 B/S 开发模式下，我们是采用将整个 web 应用打包为 war 的形式部署至应用服务器下，在 OSGi 中则通过 OSGi 的 Http Service 或通过 Equinox 的 Bridge 方式和 web 应用服务器集成来部署为 Web 应用，对于留言板系统我们选用直接通过 OSGi Http Service 来注册 web 应用的方法。

OSGi 通过 Http Service 将相应的 Servlet 以及资源文件绑定至相应的路径的请求上，当访问此路径时，Http Service 会转发到相应的 Servlet 进行处理，为了避免编写 Servlet 以及页面的复杂性，在此可编写一个简单的 MVC 框架，提供一个公用的 Servlet 来作为 Controller，同时允许每个 web 应用对此 servlet 进行扩展覆盖，Servlet 将请求转发给相应的响应类，响应类处理完毕后可直接返回相应的页面地址，页面采用 Velocity 的模板方式来进行编写，因此在模块上需要增加一个简单的 MVC 框

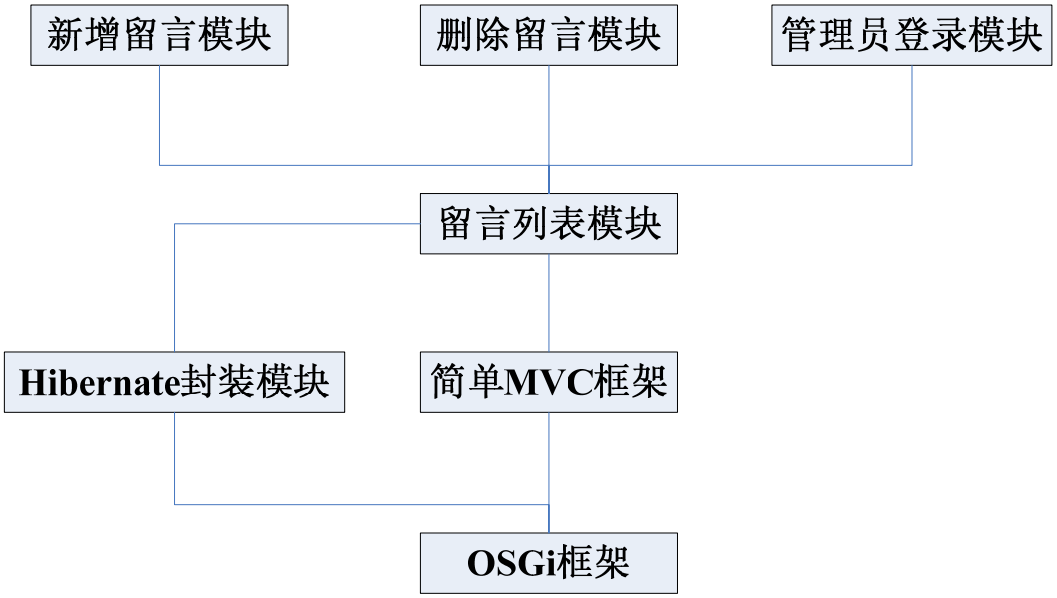
架模块。

2.2.2.3. 系统模块设计

从“即插即用、即删即无”的部署角度来看需要增加什么新的模块来支持现有的功能模块的部署：

- 部署留言列表模块，此时访问留言板的路径即可查看到留言列表；
- 部署新增留言模块，此时可在留言列表页面的右上角看到新增留言链接，点击新增留言链接进入新增留言页面，点击保存后返回到留言列表页面；
- 部署管理员登录模块，此时可通过留言板路径/admin 进入管理员登录页面，登录成功后进入留言列表页面；
- 部署删除留言模块，当以管理员身份登录后在留言列表中的每条留言后可看到删除按钮，点击删除即可删除相应的留言。

从部署角度上可以看出，系统就像是一颗树般的慢慢长出来了，形式的图示如下：



图表 3 留言板系统树状部署图

在分析完系统的部署方式后，结合各模块的功能要求以及 OSGi 的设计思想来进行系统模块的设计，与数据库的操作基于一个封装了 Hibernate 的通用 Bundle 来实现，这个 Bundle 在后续的章节中将详细的介绍，在此就不多解释了：

- 简单的 MVC 框架模块

  此模块需要简单的实现 MVC 模式，同时对外提供 Controller 类以及 Command 接口的 package，以供外部 Bundle 扩展 Controller 进行自定义的实现，并提供 Controller 以及 Command 注册的扩展点，以供外部 bundle 注册 web 应用

和 Web 响应服务。

做为 MVC 框架需要对外提供通用的 Controller 以及 Command 接口，并确定 Web 请求整个交互过程的方式。

通用 Controller 类的功能为：

通用 Controller 为一个 Servlet 类，负责接收 Web 请求转发至相应的 Command 类，并执行 Command 类将返回的结果 HTML 进行显示，同时提供一些方法供外部覆盖，如默认的 Command 名等。

Command 接口：

传入 HttpServletRequest 和 HttpServletResponse 作为参数，返回 HTML 字符串，方法示意如下：

public String execute(HttpServletRequest request，HttpServletResponse response) throws Exception;

同时为了 Controller 以及 Command 的注册，需要编写 Controller 的注册管理组件以及 Command 的注册管理组件。

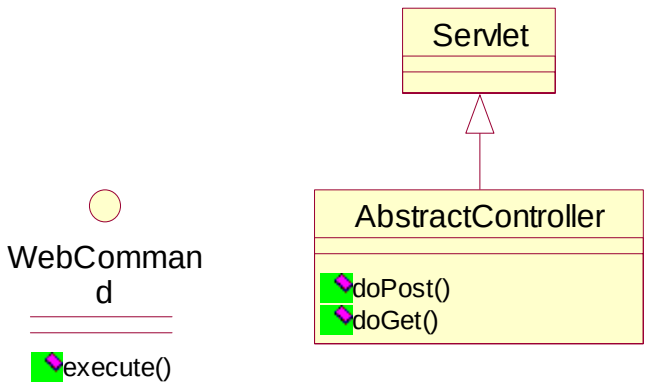
Controller 的注册管理组件：

负责注册系统中扩展出的 Controller 以及相应的资源文件的访问路径，在这里 Controller 采用扩展点的方式来实现，Equinox 已提供了这块的扩展点，在此处直接使用 org.eclipse.equinox.http.registry。

Command 的注册管理组件：

负责注册系统中的 Command，以便 Controller 能够根据请求调用相应的 Command 实现类来完成请求的响应，在这里 Command 的注册管理基于 OSGi Declarative Services 来实现，因此不需要编写注册管理组件。

根据上述分析，简单 MVC 框架模块的类图如下：



- 留言列表模块

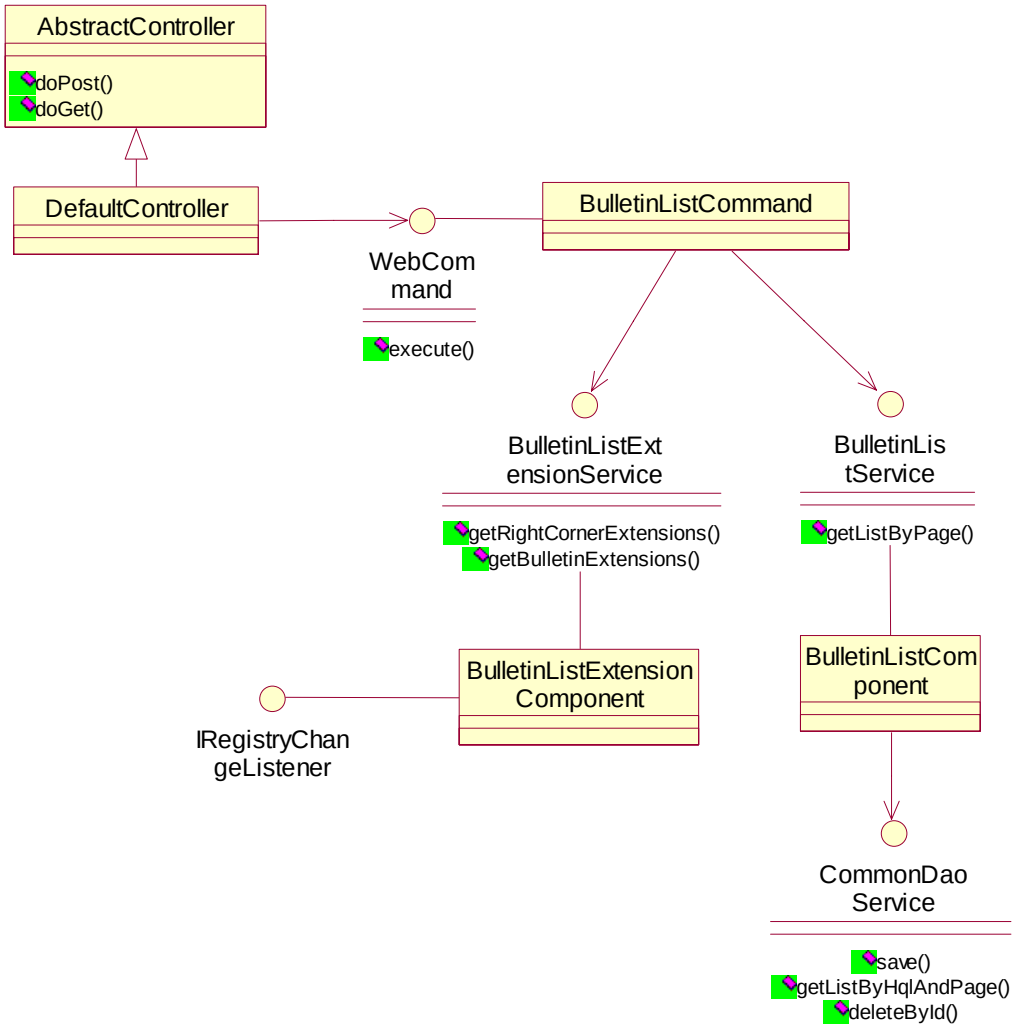
  按照需求新增留言模块安装上时，需要在留言列表页面的右上角产生新增留言的功能链接，相应的当新增留言模块卸载时，新增留言的功能链接也相应的从留言列表上删除；删除留言模块安装上时，如当前用户为管理员，则需在留言列表中增加删除留言的功能链接，相应的当删除留言模块卸载时，删除留言的功能链接也应从留言列表上删除，这就是所谓的“即插即用，即删即无”的实际例子的体现。

  按照传统方式，这些功能链接都是采用直接在留言列表页面上的，但现在采用物理分离的模块化的方法来做后，就会发现直接写在页面上的方式造成了留言列表模块对于新增留言模块、删除留言模块的强耦合，而 OSGi 应用的“即插即用，即删即无”则很好的完成解耦合，基于 Equinox 提供的扩展点的方式可很方便的实现上述扩展的需求，因此在留言列表功能的基础上需增加一个右上角功能的扩展点以及留言列表中功能链接的扩展点。

  根据系统分层方法以及留言列表的功能需求，留言列表模块分为获取留言列表 Command 服务、从数据库中获取留言列表服务类以及留言列表显示页面。留言列表模块作为留言板系统的入口页面，需要注册留言板系统的 Controller，此 Controller 相应的继承 MVC 框架中的 Controller 类进行实现，并注册为扩展点实现即可。

  留言列表模块同时需要对外提供右上角功能链接的扩展点、留言列表功能链接的扩展点，右上角功能链接的扩展点对外提供 html 片段的扩展属性，留言列表功能链接的扩展点对外提供 html 片段以及角色的扩展属性。

  留言列表模块的类图如下所示：

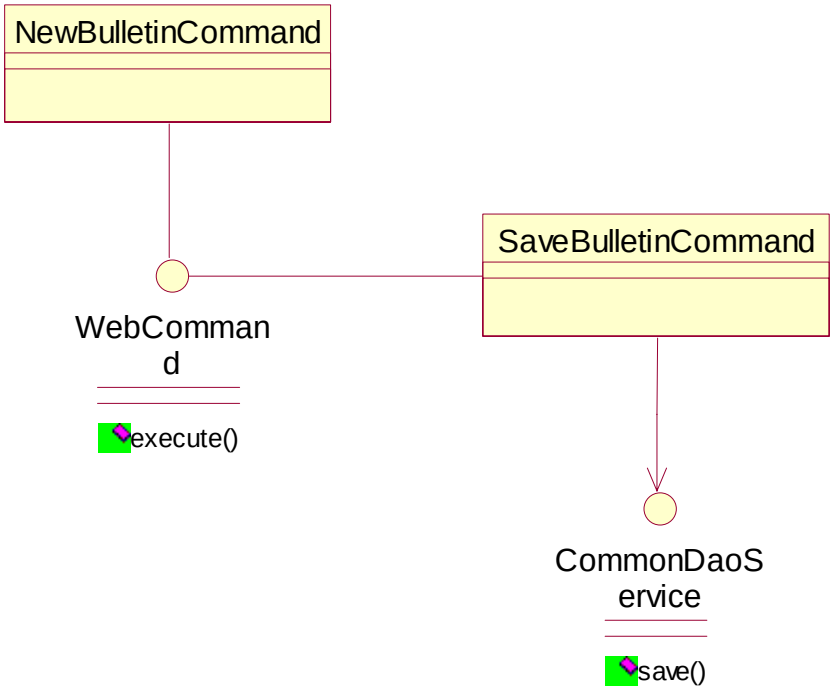


- 新增留言模块

  按照系统分层方法以及新增留言的功能需求，新增留言模块分为新增留言页面、保存留言 Command 服务和保存留言服务。

  在功能的基础上需增加一个留言列表右上角扩展点的实现，以将功能挂接到留言列表上。

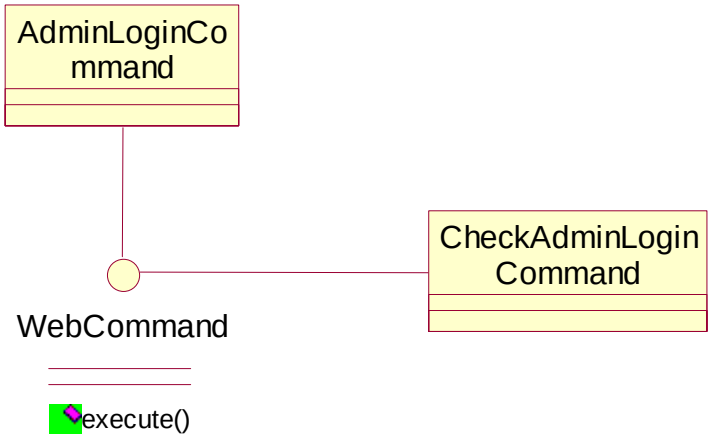
  新增留言模块的类图如下所示：



- 管理员登录模块

  按照系统分层方法以及的管理员登录功能需求，管理员登录模块分为管理员登录页面、登录 Command 服务。

  管理员登录模块的类图如下所示：

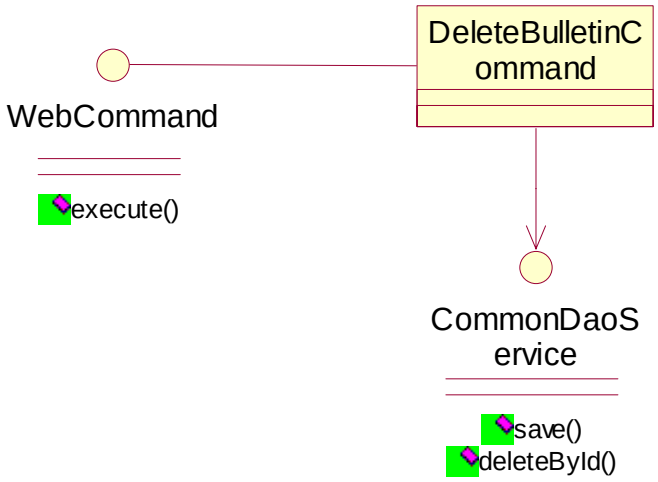


- 删除留言模块

  按照系统分层方法以及删除留言的功能需求，删除留言模块分为删除留言 Command 服务以及删除留言服务类。

  在功能的基础上需增加一个留言列表功能链接扩展点接口的实现，以将功能挂接到留言列表的每条留言的功能链接上。

删除留言模块的类图如下所示：



- 留言板所需的扩展功能的支持

  基于之上的留言板设计，已可支持回复留言和编辑留言的功能的扩展，搜索留言方面则需留言列表模块增加左上角功能的扩展点定义。

2.3. 基于 OSGi 的留言板的实现

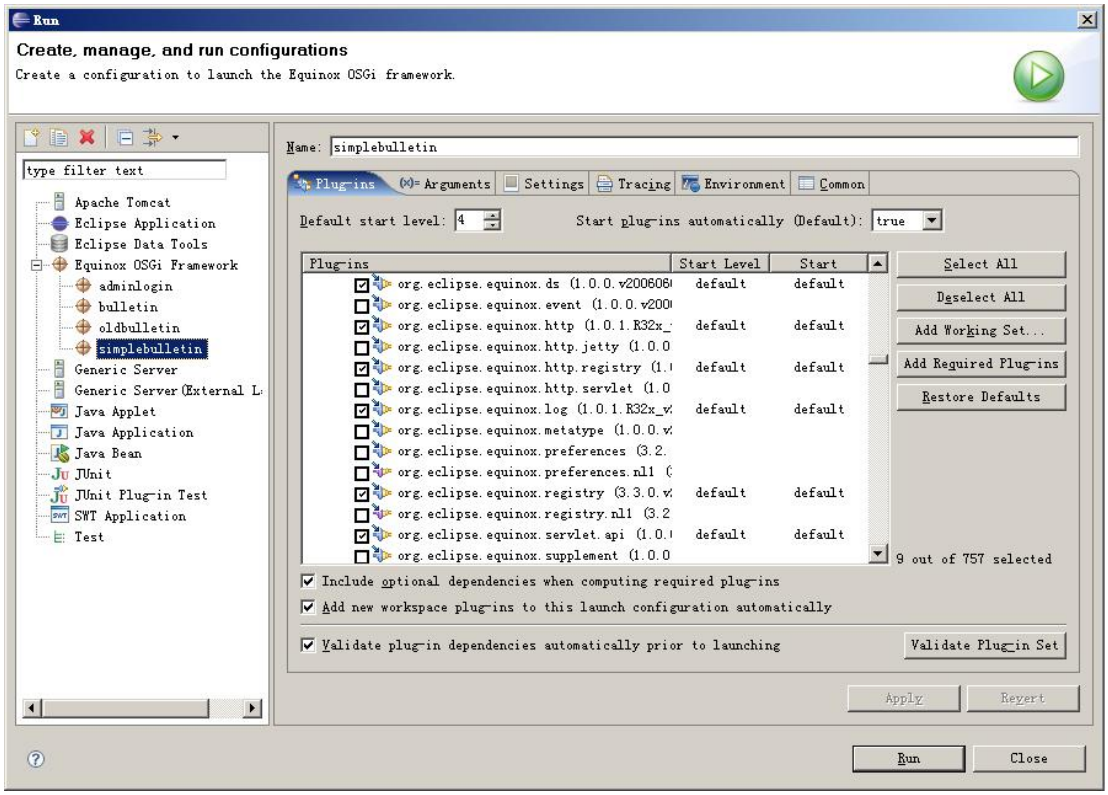
按照之上的设计来完成留言板的实现：

2.3.1. 环境准备

在《OSGi 实战》Opendoc 中详细的介绍了开发 OSGi 系统的工具箱的准备方法，因此在此篇 Opendoc 中就不再详细介绍了，实现留言板系统所需的 Bundle 有：

- org.eclipse.osgi
- org.eclipse.osgi.services
- org.eclipse.equinox.ds
- org.eclipse.equinox.http
- org.eclipse.equinox.log
- org.eclipse.equinox.servlet.api
- org.eclipse.equinox.registry
- org.eclipse.equinox.http.registry
- org.eclipse.equinox.common
- HibernateModule

点击 Run，在 Equinox OSGi Framework 新建一个 bulletin 的应用，在 plugins 中选择上述的 bundle，点击 Apply 即可，效果类似如下：



之后编写完模块时可通过启动此 bulletin 应用来启动留言板系统。

2.3.2. 简单的 MVC 框架模块

在准备好环境后我们就可以开始第一个模块的实现了，由于留言板系统是以简单的 MVC 框架模块为树根，慢慢的生长而形成，首先来实现简单 MVC 框架模块。

- 以 Plugin-Project 的方式创建简单 MVC 框架模块工程；
- 在 Import Packages 中增加 javax.servlet、javax.servlet.http 以及 org.osgi.framework；
- 新建 WebCommand 接口：

```
/**
 * 响应页面请求
 *
 * @param request
 * @param response
 * @return String HTML片段
 * @throws Exception
 */
public String execute(HttpServletRequest
request,HttpServletResponse response) throws Exception;
```

- 新建 SimpleMVCFrameworkActivator 类，为 AbstractController 提供获取 BundleContext 的方法；
- 新建 AbstractController 类，该类中最重要的部分就是根据请求参数中的

command 来确定调用相应的 COMMAND 服务，调用方法如下所示：

```
BundleContext bc=SimpleMVCFrameworkActivator.getContext();
if(bc==null)
    throw new Exception("BundleContext不可用");
ServiceReference[]
serviceRefs=bc.getAllServiceReferences(WebCommand.class.getName(),
"(command="+action+")");
if(serviceRefs.length==0){
throw new Exception("系统中没有相应的Command服务或服务不可用");
}
if(serviceRefs.length>1){
throw new Exception("系统中对应此Command的服务超过1个");
}
command=(WebCommand) bc.getService(serviceRefs[0]);
```

- 编写完上述类和接口后，在 MANIFEST.MF 中 Runtime 的 Export Packages 中增加 cn.org.osgi.opendoc.bulletin.mvc 和 cn.org.osgi.opendoc.bulletin.service，以供其他 Bundle 调用 AbstractController 类和 WebCommand 接口。

2.3.3. 留言列表模块

留言列表模块作为留言板系统的入口，需要完成留言板应用 Controller 的编写和注册，首先来完成此部分：

- 在 MANIFEST.MF 的 Import Packages 中导入 cn.org.osgi.opendoc.bulletin.mvc、cn.org.osgi.opendoc.bulletin.service 、 javax.servlet 、 javax.servlet.http 、 org.eclipse.equinox.http.registry；
- 编写 DefaultController 类作为留言板系统的 Controller，继承 AbstractController 抽象类，实现其中的抽象方法；
- 注册 DefaultController 到 Equinox HttpService 提供的扩展点；

  注册此扩展的方法比较简单，在留言列表模块的工程下新增 plugin.xml 文件，在 plugin.xml 文件中增加如下内容：

```
<plugin>
  <extension point="org.eclipse.equinox.http.registry.servlets">
    <servlet
      alias="/bulletin"
      class="cn.org.osgi.bulletin.mvc.controller.DefaultController"/>
  </extension>
</plugin>
```

这样就完成了将 DefaultController 注册到 HttpService，启动留言板应用系统后

HttpService 就会调用 register 方法将此 Controller 进行注册，当访问/bulletin 时，请求就会转发到 DefaultController 类处理。

接下来需要完成的为留言列表 Command 服务和留言列表服务：

- 要实现留言列表服务，首先需要完成留言对象的编写，按照 Hibernate 的格式要求编写留言对象，并生成 hbm 映射文件；
- 将留言对象注册到 Hibernate 封装模块提供的扩展点，起到的效果相当于在 hibernate.cfg.xml 中加了：

  <mapping resource=”cn/org/osgi/bulletin/po/Bulletin.hbm.xml”/>

  模块化的特点就是不会侵入到其他的模块，只通过扩展等方法来扩展其他的模块的资源和功能等。

  注册留言对象只需在 plugin.xml 中增加以下内容：

```
<extension point="HibernateModule.HibernateExtension">
    <po class="cn.org.osgi.bulletin.po.Bulletin"/>
</extension>
```

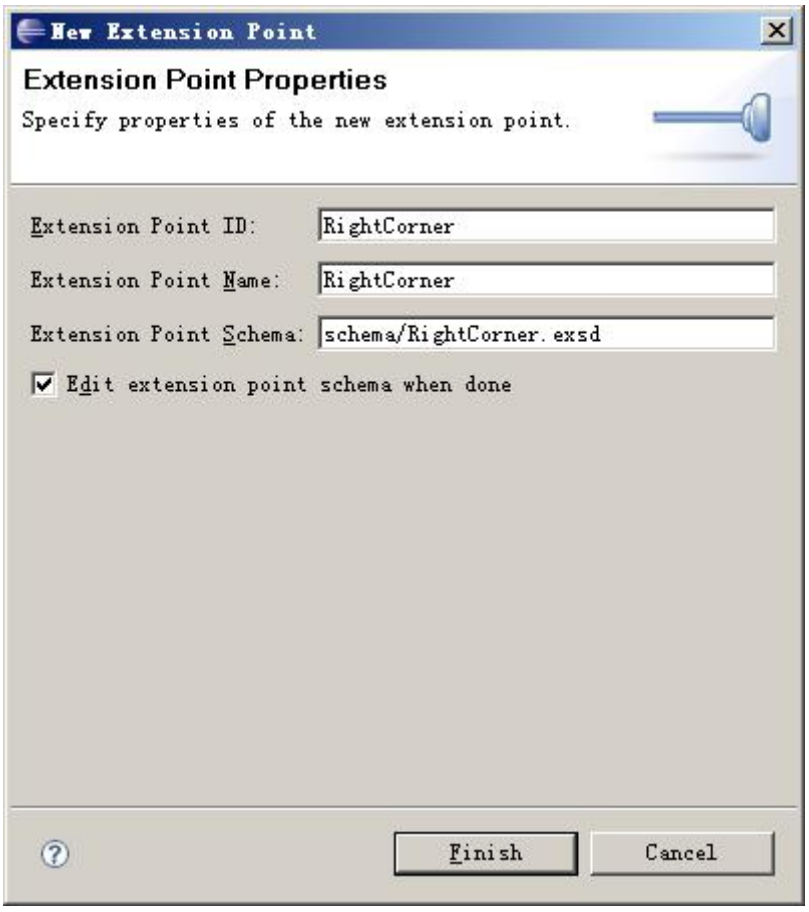
  编写完毕后在 MANIFEST.MF 的 Runtime 的 Export Packages 中增加 cn.org.osgi.bulletin.po 包的导出。

  经过上面的这两步后启动留言板应用时 Hibernate 封装模块即可加载留言对象到 Hibernate 的管理范围内。
- 编写留言列表服务，留言列表服务基于 Hibernate 封装模块 Export 的 CommonDaoService 实现；
- 编写留言列表 Command 服务，采用 Declarative Services 支持的 bind 的方式注入留言列表服务，实现分页获取留言列表数据，并将列表数据填入 VelocityContext 中，以供页面进行显示；
- 基于 Velocity 方式编写留言列表模板页面；
- 编写 Declarative Services 所需的服务组件的描述文件，将留言列表服务和留言列表 Command 服务列入 Declarative Services 的管辖范围，在编写留言列表 Command 服务的描述文件时，需要加上 Controller 调用 Command 服务时使用的标识符，如：<property name="command" value="LIST"/>；
- 将服务组件描述文件加入到 MANIFEST.MF 的 Service-Component 中。

留言列表模块还需提供对于留言应用而言的新功能以及对于留言本身操作的新功能

的扩展点，以便对此两部分的功能进行扩展，在这里就需要学习如何编写 Equinox 的扩展点了：

- 定义扩展点的方法并不复杂，进入 MANIFEST.MF 中的 Extension Points 标签项，点击其中的 Add 按钮，输入扩展点的 ID、Name 以及 Schema，Schema 如何定义也就决定了其他模块要进行扩展时如何描述了，在这里以对于留言应用而言的新功能的扩展点为例，将其 ID 设置为 RightCorner、Name 设置为 RightCorner、Schema 设置为 schema/RightCorner.exsd：

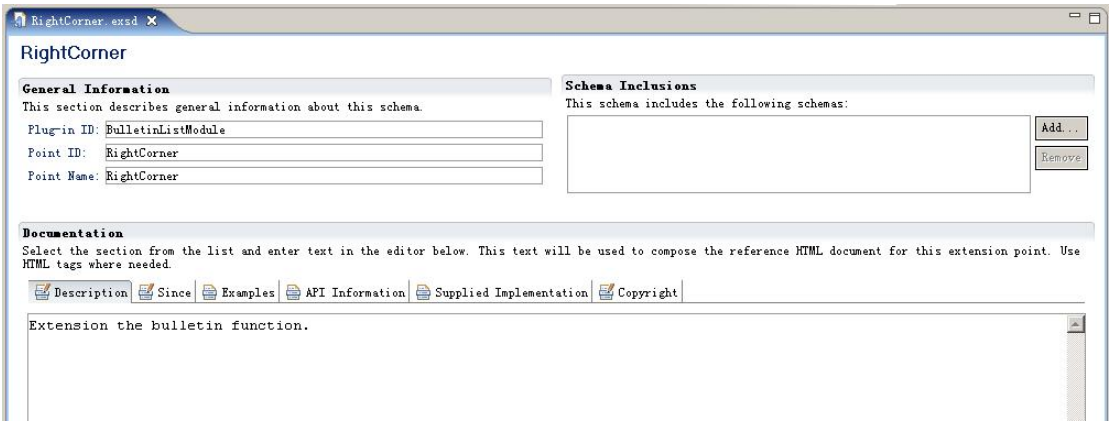


设置完毕后在 plugin.xml 可看到如下信息：

<extension-point id="RightCorner" name="RightCorner"

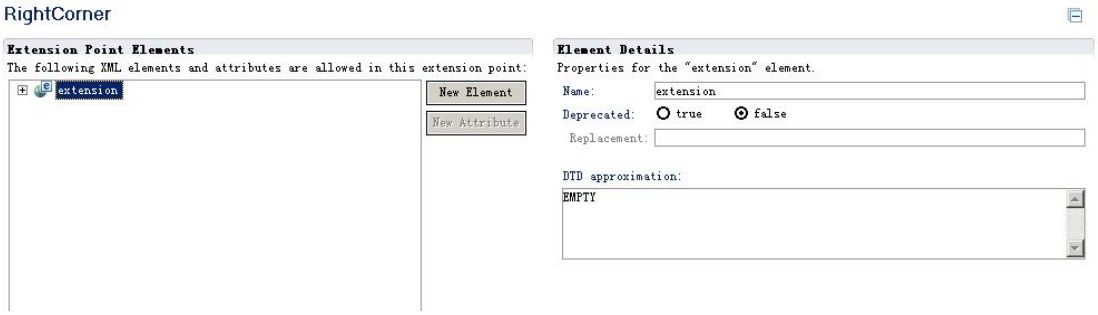
schema="schema/RightCorner.exsd"/>

- 在留言列表模块工程的目录下建立 Schema 目录，在此目录下新建 RightCorner.exsd，双击打开进入如下视图：



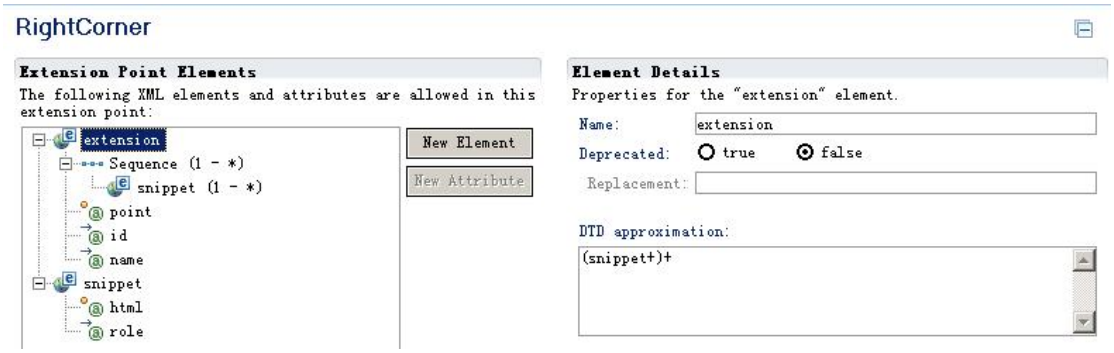
填入相应的信息；

- 点击标签项中的 Definition 标签项，进入如下视图：



在上面的视图中点击 New Element，将此 element 的名称修改为 snippet；

- 选中 snippet 元素，点击 New Attribute，在其中增加 html 和 role 两项 attribute，将 html 的 Use 选项设置为 required，Type 设置为 string；role 的 Use 选项设置为 optional，Type 设置为 string；
- 选中 extension 元素，右键选择其中的 New—Compositor—sequence，将 sequnece 中的 Max Occurences 设置为 Unbounded；
- 选中 sequence，右键选择其中的 New—Reference—snippet，将 snippet 中的 Max Occurences 设置为 Unbounded；
- 在配置完上面的步骤后，最后形成的 Definition 视图如下所示：



经过上面的步骤后，即完成了 RightCorner 扩展点的配置工作，按照类似的方法配置 Bulletin 扩展点，在该扩展点下增加 snippet 元素，其中的 Attribute 分别为 title、command 和 role，其中 title 和 command 为 required 项。

在配置了 RightCorner 和 Bulletin 扩展点之后，就是如何去获取这两个扩展点的扩展了，根据 Equinox 对于扩展点的使用方法，可通过 Equinox 提供的 IExtensionRegistry 服务来获取，同时通过实现 IRegistryChangeListener 来注册为扩展注册变化的监听器，这样当实现扩展点的扩展发生变化时可动态的获取到这些变化的信息，以便相应的做出处理，结合设计编写获取留言板列表扩展的服务，在编写该服务时，最重要的是掌握如何获取扩展点对应的扩展以及如何监听扩展点对应的扩展发生的变化，以保证系统的动态性：

- 获取相应扩展点的扩展；

  通过 IExtensionRegistry 服务获取：

  IExtensionRegistry.getExtensionPoint(扩展点所在插件的 ID+"."+扩展点 ID).getExtensions()；

  通过这样的方法即可获取到相应扩展点的扩展，例如对于 RightCorner 扩展点，获取其扩展实现的方法为：

  IExtensionRegistry.getExtensionPoint("BulletinListModule.RightCorner").getExtensions()；

- 监听相应扩展点的扩展发生的变化；

  实现 IRegistryChangeListener 接口的

  public void registryChanged(IRegistryChangeEvent event)方法，通过 IExtensionRegistry 将当前类注册为扩展点变化的监听器，在注册时使用的 NAMESPACE 参数为扩展点所在的插件的插件 ID，例如对于 RightCorner 扩展点而言即为：

  IExtensionRegistry.addRegistryChangeListener(this,"BulletinListModule")

  在 registryChanged 方法中通过 event.getExtensionDeltas 来获取所感兴趣的扩展点的扩展的变化情况，如 RightCorner 扩展点的扩展的变化的监听：

  event.getExtensionDeltas("BulletinListModule","RightCorner");

  在获取到 IExtensionDelta 后，通过其 getKind 方法来判断扩展发生了何种变化，如 getKind()==IExtensionDelta.*ADDED*，那也就意味着系统中增加了一个此扩展

点的实现；如 getKind()==IExtensionDelta.*REMOVED*，那也就意味着之前的一个扩展点的实现已经从系统中删除了。

通过这样的方法即可动态的获取和处理系统中扩展点的扩展实现，在留言表列表模块中编写了一个 BulletinListExtensionComponent 来专门处理扩展点扩展实现的获取：

- 在激活组件时获取到当前扩展点扩展实现的集合，并注册监听器；

```
public void activate(ComponentContext context){
   IExtension[]
   extensions=registry.getExtensionPoint(NAMESPACE+"."+RIGH
   TCORNER_EXTENSIONPOINT).getExtensions();
        for (int i = 0; i < extensions.length; i++) {
            rightCornerExtensions.add(extensions[i]);
     }
    registry.addRegistryChangeListener(this,NAMESPACE);
 }
```

- 在扩展点扩展实现发生变化时，相应的处理扩展点实现的集合；

```
public void registryChanged(IRegistryChangeEvent event) {
     IExtensionDelta[]
     deltas=event.getExtensionDeltas(NAMESPACE,RIGHTCORNER_E
     XTENSIONPOINT);
     for (int i = 0; i < deltas.length; i++) {
         switch (deltas[i].getKind()) {
             case IExtensionDelta.ADDED:
     rightCornerExtensions.add(deltas[i].getExtension());
             break;
             case IExtensionDelta.REMOVED:
     rightCornerExtensions.remove(deltas[i].getExtension());
             break;
             default:
             break;
         }
     }
}
```

在编写完扩展点的管理服务后，就需要把这些扩展点的扩展表现到留言列表页面上了，这步较为简单，在留言板列表模块的 Command 服务中调用留言板列表扩展点管理服务，获取扩展点的扩展实现填充至 VelocityContext 中，页面获取到扩展中的

相关属性配置完成显示即可，如右上角功能链接的扩展的页面显示：

```
#if($rightCornerExtensionsHtml.size()>0)
<table id="normaltable" cellspacing="0" align="center">
    <tr height="23">
        <td align="right">
            #foreach($html in $rightCornerExtensionsHtml)
                 $html
            #end
        </td>
    </tr>
</table>
#end
```

通过上述步骤，留言列表模块即宣告完成，这个时候启动留言列表应用，通过 http://localhost:Web端口/bulletin访问到此应用(此处的Web端口默认为 80，可通过在 vm arguments中增加-Dorg.osgi.service.http.port=8080 来自定义web端口)。

2.3.4. 新增留言模块

新增留言作为对于留言板功能的扩展，需要作为扩展实现挂接到留言列表页面中，新增留言模块的实现较为简单，在这里主要讲解下如何使用留言列表模块中提供的 RightCorner 扩展点：

- 要使用 RightCorner 扩展点，只需在 plugin.xml 中加入以下内容：

```
<extension point="BulletinListModule.RightCorner">
 <snippet
        html="&lt;a
        href=&apos;/bulletin?command=NEWBULLETIN&apos;&gt;新增留言&lt;/a&gt;"/>
</extension>
```

在编写完新增留言模块需要的新增留言 Command 服务、保存留言 Command 服务、新增留言 Velocity 模板页面后，再次启动留言板应用，这个时候在留言板列表页面的右上角，可看到新增留言链接，通过卸载新增留言模块来测试下扩展点的动态效果，在 console 中 uninstall 新增留言模块，刷新留言板列表页面，此时新增留言链接也从右上角删除了，真正的做到了“即插即用、即删即无”的效果。

2.3.5. 管理员登录模块

作为演示程序，在此处管理员登录的校验直接写为了固定值，管理员登录模块和新增留言模块的编写相似，在此不多描述。

2.3.6. 删除留言模块

删除留言模块和之前的新增留言模块不同的地方仅在于使用的扩展点不一样，删除留言模块使用的为 Bulletin 这个扩展点，在 plugin.xml 中增加以下内容即可实现对此扩展点的扩展：

```
<extension point="BulletinListModule.Bulletin">
 <snippet
       command="DELETEBULLETIN"
       role="admin"
       title="删除"/>
</extension>
```

在编写完上面的模块后，通过http://localhost:web端口/bulletin访问即可使用到上述的功能，管理员通过http://localhost:web端口/bulletin?command=ADMINLOG来登录，大家可尝试将管理员登录这个功能挂接到留言板列表上去，就像新增留言的功能链接一样。

2.4. 小结

- **内容总结**

  本章节以一个简单的留言板系统为例，介绍了 OSGi 框架的功能、设计思想，遵循 OSGi 的设计思想完成留言板系统的设计，并基于设计实现了此留言板系统。

  本章中所展示的模块化的实现方法也不一定是最佳的方法，本章中所展示的模块化的方法适合于分模块进行开发的团队，但并不适合分层开发的团队，如果团队采用分层开发的方法，那么在模块的划分上最好是在现有基础上再次划分（即把模块按层次再划分为几个 Bundle，而不是统一的一个 Bundle），以提高团队协作的效率。

  如果你有想使用 OSGi 的想法了，估计看完本章后还是会有不少的疑问，例如基于 OSGi 怎么来构建基于 webwork+spring 2+hibernate 的系统、能否把现有系统改造为基于 OSGi 的系统呢，很高兴后面的章节将解答你的这些疑问。

- **知识点**

  从这个章节的内容中我们学习到了基于 OSGi 如何设计/实现一个规范的模块化，动态化以及可扩展的系统，希望这些实践方法能为大家在基于 OSGi 设计/实现 OSGi 系统提供一些指导，以更好的实现规范的模块化、动态化以及可扩展的系统，大家可以去想想，如果不基于 OSGi，要实现一个符合这样需求的系统要怎么去做呢，如果觉得比 OSGi 复杂的话，那么不妨试试基于 OSGi 来实现你的系

统。

在本章中还提及到了树状设计实践方法，这个设计实践方法有利于保证 OSGi 应用保持模块化、动态化以及可扩展性，并形象化的将系统的模块化、扩展性以及耦合情况描述出来。

- **衍生发展**

本章节主要是为了展示 OSGi 的设计思想以及基于 OSGi 的设计思想如何设计实际的系统，因此在留言板的功能等方面还有很多可改进的余地，大家可以基于源码来进行重构、扩展（例如增加编辑留言功能、回复留言功能等等）或编写一个新的基于 OSGi 的更好用的留言板，希望这个留言板系统能成为 OSGi 的一个很好的 Demo，充分的展示出 OSGi 的优势，就像 java ee 的 petstore 一样。

同时在本章节所产生的 MVC 框架只是一个非常简单的 MVC 框架，大家可以进行改造来提升这个 MVC 框架的功能，如支持将 HttpServletRequest 绑定到对象上，使得 WebCommand 服务能脱离 Servlet 环境等。

3. 与流行的 Java B/S 体系架构进行集成

在看完了上面的章节后，大家可能会想：OK，OSGi 是还不错，也大概知道了怎么样基于它去设计实际的项目，但对于 Java B/S 系统而言，OSGi 怎么和目前流行的体系架构集成呢，这恐怕是大部分读者最为关注的问题。如果能解决这个问题的话，我想大部分人都会考虑在新的项目/产品中去使用 OSGi，毕竟 OSGi 带来的好处是非常明显的，在本章节中，将和大家来一起搭建一个基于 OSGi 的 Hibernate+Spring+Webwork 的脚手架。首先来谈谈愿景，看看想象中的这个脚手架应该是什么样的，脚手架是为了支撑应用开发的，所以要从基于此脚手架开发 Web 应用的方式，才能看出脚手架需要支持些什么。如果具备一个这样的脚手架，之前留言板系统的留言列表模块就会这么开发：

- 编写留言 PO，根据 PO 生成映射文件，并将此 PO 注册到 Hibernate 模块中，这个和之前的开发方式没多大区别；
- 编写留言列表服务，并将其定义为 Spring bean，并通过 spring 注入 Hibernate 模块提供的通用 Dao 服务；
- 编写留言 Command 服务，但不是基于之前的简单 MVC 框架了，而是归入 Webwork，在这里只需要以 POJO 的方式进行编写即可了，将此 Command 的配置注册到 Webwork 模块中，将此 Command 服务定义为 Spring bean，以通过 spring 将留言列表服务注入，并需要以 OSGi 服务的方式对外提供；
- 注册留言板应用。

按照之上的步骤即可完成留言列表模块的开发，来看看这种开发模式和传统的基于 Hibernate+Spring+Webwork 开发模式的不同：

	传统开发方式	**基于 OSGi 的新的开发方式**
模块开发方式	多模块放入同一工程中开发或模块分为多工程，通过工程依赖的方式来开发。	每个模块做为一个单独的工程，并且不需要通过工程依赖的方式。
模块部署方式	启动时加载所有模块，不可动态部署、更新和管理模块的状态。	模块可动态部署、更新和管理。
Command 服务注册方式	以 Action File 的方式写入统一的 xml 文件中，多人共同	编写独立的描述 Command 服务的 xml 文件，通过扩展点的方式注册

	维护。	到 Webwork 中，按模块单独维护。
PO 注册方式	以 mapping recource 的方式写入统一的 hibernate.cfg.xml 文件中，多人共同维护。	通过扩展点的方式注册到 Hibernate 中，按模块单独维护。

为什么说基于 OSGi 后就不好使用这些开源框架了呢，原因其实非常简单，就在于这些开源框架都是基于统一管理和唯一的 classloader 来设计开发的，但基于 OSGi 后，各个模块分工程开发，资源分布在不同的模块中，并且这些资源都位于不同的 classloader 中，这就是基于 OSGi 后造成的冲突，下面我们就来解决这些冲突，使得 OSGi 可以和 Hibernate+Spring+Webwork 很好的集成在一起，在此处使用的 Hibernate 的版本为：3.1.3；Webwork 的版本为：2.2.6。

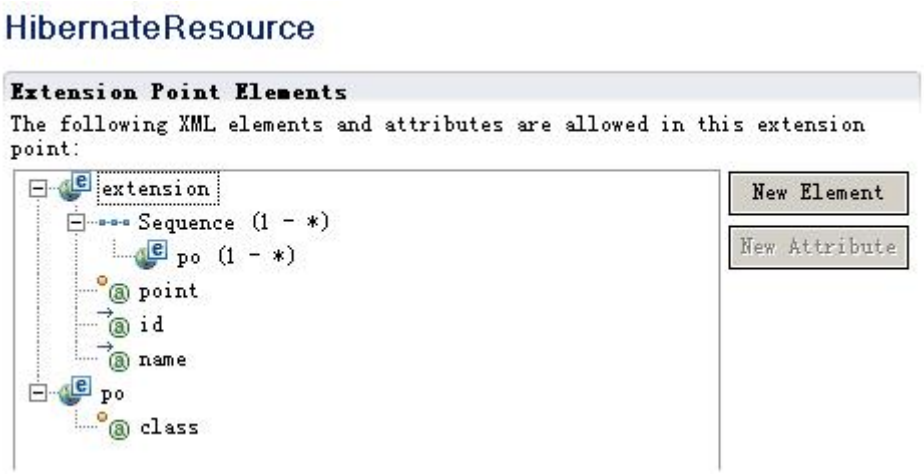
3.1. 解决和 Hibernate 的集成

默认情况下 Hibernate 通过 hibernate.cfg.xml 中的 mapping resource 来加载归入 SessionFactory 中管理的 PO，在加载时 Hibernate 通过 Hibernate 类所在的 classloader 加载 Mapping resource 中配置的 hbm 映射文件，并通过 cglib 将 hbm 映射文件中指定的 class 生成 proxy。

改为基于 OSGi 后产生冲突的地方就在于 PO 分散到不同的工程中去了，同时要保持模块化的封装性，就不能所有人都来修改封装 Hibernate 模块里的 hibernate.cfg.xml；另外一个冲突点在于封装 Hibernate 的模块的 classloader 无法加载到位于其他模块的 PO 对象，只要解决了这两个冲突，基于 OSGi 使用 Hibernate 的问题就解决了。

要解决这两个冲突，需要解决的就是如何将其他模块的 PO 注册到封装 Hibernate 的模块中，以及 Hibernate 的 Configuration 如何加载其他模块的 PO 和映射文件。

基于 OSGi 的可扩展性，要将其他模块的 PO 注册到封装 Hibernate 的模块中，通过扩展点方式即可实现，首先来设计此扩展点，我们注意到 Hibernate 的 Configuration.addClass 加载 class 的方式可达到和 mapping resource 完全一样的效果，那么就很简单了，扩展点中需要的仅为 po 的 classname，此扩展点设计完毕后如下所示：



图表 4 Hibernate PO 扩展点

然后就要考虑如何获取扩展点的实现了，在第二个章节中已经演示了通过 IExtensionRegistry 获取扩展点的实现的方法。

获取到扩展点的实现后，需要的就是把 class 加入到 Hibernate 的 Configuration 中，OSGi 中每个 Bundle 都是独立的 ClassLoader，那也就是说 Hibernate 所在的这个 Bundle 是无法加载到扩展点中 className 对应的类的，所幸的是 IExtension 的 IConfigurationElement 提供了初始化扩展中 class 的方法，也就是说可以实现在扩展点的模块中直接创建提供扩展的模块中的类，实现的方法是：

```
IExtension[]
extensions=registry.getExtensionPoint("HibernateModule.Hiber
nateExtension").getExtensions();
 for (int i = 0; i < extensions.length; i++) {
      IConfigurationElement[]
      elements=extensions[i].getConfigurationElements();
      for (int j = 0; j < elements.length; j++) {
   poClass=elements[j].createExecutableExtension("class").get
   Class();
```

通过这样的方法就在封装 Hibernate 的模块中创建出了扩展的 PO 的 Class 实例，创建出此实例后即可通过 Hibernate 的 Configuration.addClass 将此 PO 加入到 SessionFactory 的管理范围内，这样就解决了那两个冲突，实现了基于 OSGi 的 Hibernate 的扩展。

上面的实现方式解决了静态时 Hibernate 加载其他模块中的 PO 初始化 SessionFactory 的问题，但记住要保持好基于 OSGi 的系统的动态性的特征，要保持动态化，就得监听 Hibernate 模块这个扩展点的扩展实现的变化情况，当系统中

增加了新的需要归入 SessionFactory 管理的 PO 时，需要动态的加入到目前的 SessionFactory 中，当已有的 PO 从系统中删除时，也需要动态的将其从目前的 SessionFactory 中删除，不过 Hibernate 的 SessionFactory 是不支持动态的删除和增加其中管理的 PO 的，当发生变化时，只能重新初始化 SessionFactory 了。

根据上面的这些分析，最终我们将 Hibernate 的封装模块设计为两个服务和一个扩展点构成：

- Hibernate PO 扩展点

  该扩展点定义如下：

  <extension-point id="HibernateExtension"

  name="cn.org.osgi.hibernate.extension"

  schema="schema/hibernateExtension.exsd"/>

  hibernateExtension.xsd的定义如图表 4所示。

- SessionService

  SessionService 负责 SessionFactory 的管理和对外提供 Session 的管理(获取、关闭 Session)，在 SessionService 被激活时初始化 SessionFactory，初始化 SessionFactory 的代码如下所示：

```
private void initSessionFactory(){
  Class poClass=null;
  try{
      if(_sf!=null){
          _sf.close();
      }
      Configuration config=(new
      Configuration()).configure(_configFile);
      for (Iterator iter = poExtensions.iterator();
      iter.hasNext();) {
          IExtension extension = (IExtension) iter.next();
          IConfigurationElement[]
          elements=extension.getConfigurationElements();
          for (int j = 0; j < elements.length; j++) {

    poClass=elements[j].createExecutableExtension("class").getClass()
;
                config.addClass(poClass);
          }
      }
```

```
            _sf=config.buildSessionFactory();
            info("已初始化SessionFactory");
        }
        catch(Throwable t){
            error("初始化Hibernate SessionFactory时出现错误",t);
            throw new RuntimeException("初始化Hibernate
            SessionFactory错误："+t);
        }
    }
```

其中的 poExtensions 在激活 SessionService 组件时填充：

```
IExtension[]
extensions=registry.getExtensionPoint("HibernateModule.HibernateExtension").g
etExtensions();
        for (int i = 0; i < extensions.length; i++) {
            poExtensions.add(extensions[i]);
        }
```

SessionService 组件还需实现 IRegistryChangeListener，以保证 SessionFactory 能动态的管理需要归入其管理的 PO：

```
public class SessionComponent implements
SessionService,IRegistryChangeListener{
    public void registryChanged(IRegistryChangeEvent event) {
        IExtensionDelta[]  deltas=event.getExtensionDeltas("HibernateModule",
"HibernateExtension");
        if(deltas.length==0)
            return;
        for (int i = 0; i < deltas.length; i++) {
            switch (deltas[i].getKind()) {
            case IExtensionDelta.ADDED:
                poExtensions.add(deltas[i].getExtension());
                break;
            case IExtensionDelta.REMOVED:
                poExtensions.remove(deltas[i].getExtension());
```

```
                    break;
                default:
                    break;
                }
            }
            initSessionFactory();
        }
    }
```

注册 SessionService 组件为扩展变化的监听器：

```
registry.addRegistryChangeListener(this,"HibernateModule");
```

通过以上步骤即实现了动态的加载和卸载其他模块的 PO。

- CommonDaoService

  CommonDaoService 对外提供了一些通用的数据库操作的接口，如 save(Object obj)、update(Object obj)、getListByPage 等。

按照上面的步骤完成后，可编写一个 Bundle 来测试下 Hibernate 封装模块是否可用，启动后将会出现如下错误：

```
entity class not found: cn.org.osgi.bulletin.po.Bulletin
```

这是为什么呢？

根据错误信息的堆栈可看到有这么一行：

```
ReflectHelper.classForName
```

打开 `ReflectHelper` 类可以看到 `Hibernate` 会根据映射文件中的 `classname` 去实例化该 `Class`，而不是直接使用我们通过 `Configuration.addClass` 时传递给 `Hibernate` 的 `Class`，直接使用 `Class.forName` 的方法实例化扩展出的 `PO`，自然会出现上面的错误，因为 `Hibernate` 类所在的 `ClassLoader` 和 `Bulletin` 类所在的 `ClassLoader` 并不同，碰到这样的情况，就得使用 `OSGi` 提供给 `Bundle` 的 `DynamicImport-Package` 了，使用此属性 `OSGi` 就可以在运行期动态的为这个 `Bundle` 获取其他 `Bundle Export` 的 `Package`，使用这种方法的话要求 `PO` 所在的 `Bundle` 将 `PO` 的 Package Export，打开封装 Hibernate 的 Bundle 的 MANIFEST.MF 文件，在其中加上这一行：

```
DynamicImport-Package: *
```

然后在 cn.org.osgi.bulletin.po.Bulletin 的 Bundle 的 MANIFEST.MF 中加上 Export-Package: cn.org.osgi.bulletin.po；

加上这些后再次启动，发现在控制台中没有了之前的那个错误，但又出现了一个新的错误：

```
java.lang.NoClassDefFoundError:org/hibernate/proxy/HibernateProxy
```

这个错误对于延迟加载而言会有不小的影响，因此我们需要追查此错误的原因，根据错误堆栈信息，可追查出是 cglib 中的 `AbstractClassGenerator.create` 方法执行时出现了错误，找出 `AbstractClassGenerator` 的源码，跟踪后发现是在此类中无法找到 `HibernateProxy`，是不是有些奇怪呢？

仔细来分析下，为什么明明在同一个 Bundle 下的两个 jar，却加载不到呢？问题必然还是出在 classloader 上，仔细看 create 方法，会看到其中有个 getClassLoader()，跟踪调试看 getClassLoader()就会发现，当为 Bulletin 生成 Proxy 时，这个时候的 ClassLoader 变成了 Bulletin 所在 Bundle 的 ClassLoader 了，不用说，用这个 ClassLoader 去加载 HibernateProxy 自然是加载不到了，找到了问题的根源，很明显需要 hack 下 AbstractClassGenerator 这个类了，把它的 ClassLoader 改为使用当前类所在的 ClassLoader，即：

```
// FIXME: Hack For OSGi
if(t==null){
   t=this.getClass().getClassLoader();
}
```

OK，改完后重启，一切都好了，到这为止，可以说基于 OSGi 的 Hibernate 的扩展就封装好了。

相信细心的朋友们会发现这个 Bundle 还有不少需要改进指出：例如支持对于 PO Class 使用缓存的配置、支持多个连接不同库的 SessionFactory 等，这一切读者们都可以自己来完成，或者共同参与到 OSGi.org.cn 的基于 OSGi 的 Hibernate+Spring+Webwork 的脚手架项目中来。

3.2. 解决和 Spring 的集成

和 Spring 的集成是 OSGi 必须解决的问题，毕竟 Spring 占据了目前大部分的 Java 应用领域，幸运的是 Spring 也发现了 OSGi 的好处，Spring-OSGi 的推出无疑让 OSGi 的忠实 fans 们更加的开心，同时也给 Spring 的用户体验 OSGi 所带来的好处的机会，

Spring 和 OSGi 可以很好的结合在一起来满足企业应用的需求，这对于 OSGi 进军企业应用领域无疑是非常好的事。

对于目前版本的 OSGi（R4）而言，Spring 能带来的好处主要有这么两点：

- 不需要对外 Export 的服务可控制在 Bundle 范围内进行注入使用；

  在不使用 Spring 时，只能将这些服务也通过 DS 以 OSGi Service 的方式对外提供，否则 Bundle 内的其他 Component 会无法通过注入的方式使用这个 Service，而在使用 Spring 的情况下则可将 Bundle 内的服务定义成 bean，并通过 Spring DI 注入到其他的 Component 中，例如留言列表模块中的留言列表服务、留言列表扩展点服务，这些都是不需要归入 DS 管理的。

  这个带来的另外一个好处就是 Component 可以统一到一个描述文件里编写了，而不用像在 DS 里一样，每个 Component 编写一个新的描述文件，那样管理起来会比较的麻烦。

- 获取到 Spring 提供的 POJO Enhanced 的众多功能；

  这个好处非常的明显，Spring 之所以成功其中最重要的原因就是它的 POJO Enhanced 的支持，通过 Spring 的 POJO Enhanced 的支持，POJO 很容易就变成实现了企业应用复杂需求的功能类，例如远程调用、事务控制等，Spring 和 OSGi 的集成也就使得基于 OSGi 的应用很容易获取到 Spring 提供的这些好处，从而快速的实现企业应用的开发。

对于留言板这个简单的应用而言，在这里我们就只改进其中各个服务控制在 Bundle 范围内进行注册使用这一点，从而来说明 Spring-OSGi 的环境的搭建和使用的方法。

3.2.1. 搭建开发环境

首先来搭建 Spring-OSGi 的开发环境，根据对 OSGi 的了解，很容易知道，要实现 Spring 与 OSGi 的结合，必然是 Spring 会提供一些 Bundle 部署至 OSGi 框架中，有了这些 Bundle 后就可以使用到 Spring 提供的功能了，首先我们来了解需要部署哪些 Bundle 至 OSGi 的框架中：

- 从 Spring-OSGi 的官方网站：http://www.springframework.org/osgi 上下载 Spring-OSGi，在编写此篇Opendoc时的版本为 1.0 M3；

  下载解压后可看到 dist 和 lib 两个目录，这两个目录中提供了我们搭建 Spring-OSGi 开发环境所需的 Spring 相关的 Bundle。

- 部署 Spring-OSGi 所需的 Bundle；

Spring-OSGi 运行环境所需的 Bundle 为：

- org.springframework.osgi.aopalliance.osgi

  对应的 jar 文件为：aopalliance.osgi.jar

- org.springframework.osgi.aspectjrt.osgi

  对应的 jar 文件为：aspectjrt.osgi.jar

- org.springframework.osgi.backport-util-concurrent

  对应的 jar 文件为：backport-util-concurrent

- org.springframework.osgi.spring-aop

  对应的 jar 文件为：spring-aop.jar

- org.springframework.osgi.spring-aspects

  对应的 jar 文件为：spring-aspects.jar

- org.springframework.osgi.spring-beans

  对应的 jar 文件为：spring-beans.jar

- org.springframework.osgi.spring-context

  对应的 jar 文件为：spring-context.jar

- org.springframework.osgi.spring-core

  对应的 jar 文件为：spring-core.jar

- org.springframework.osgi.spring-dao

  对应的 jar 文件为：spring-dao.jar

- org.springframework.osgi.spring-osgi-core

  对应的 jar 文件为：spring-osgi-core.jar

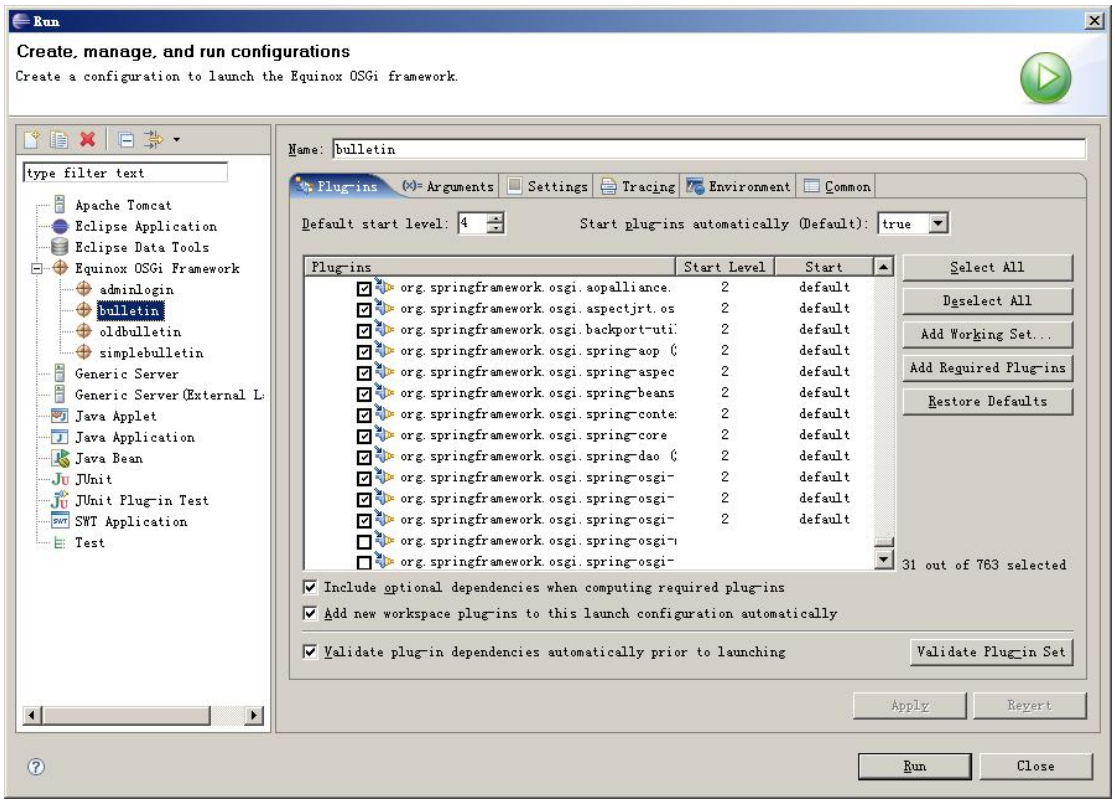
- org.springframework.osgi.spring-osgi-extender

  对应的 jar 文件为：spring-osgi-extender.jar

- org.springframework.osgi.spring-osgi-io

  对应的 jar 文件为：spring-osgi-io.jar

在之前的 dist 和 lib 中找到类似上面所写的文件名的 jar 文件，并将其复制到 eclipse 的 plugins 目录(或通过外部 link 插件的方法；或直接 install 到 OSGi 框架中)，启动 eclipse，此时在 eclipse 的 equinox osgi framework 中即可看到有如上插件名称的插件了，这样就完成了 Spring-OSGi 的开发环境的准备工作。



3.2.2. 在 Spring bean xml 中发布和引用 OSGi Service

在搭建好开发环境后，需要来学习下在 Spring-OSGi 中如何将 bean 发布为 OSGi Service，以及如何引用 OSGi Service，在发布和引用 OSGi Service 时希望做到的是和使用 DS 时相同的效果。

要把一个 Spring bean 发布为 OSGi Service，只需在 spring bean 的 xml 中这么配置就行了：

<osgi:service id=”OSGi 服务 ID” ref=”需要发布为 OSGi 服务的 spring bean” interface=”服务的接口”/>

在发布 OSGi Service 时，除了上面这些基本信息，还有需要经常用到的就是 Service 的 Properties 的配置了，通过在上面的 xml 元素下增加 osgi:service-properties 的节点来实现，例如：

```
<osgi:service>
  <osgi:service-properties>
        <prop key=”属性名”>属性值</prop>
  </osgi:service-properties>
  </osgi:service>
```

在发布 OSGi Service 的配置上 Spring-OSGi 还支持 lazy-init、depends-on 以及 context-classloader 这些属性配置，当 lazy-init 配置为 true 时此服务只有在被调用时才会激活；depends-on 可用于配置此服务所依赖的服务；context-classloader 用于配置使用第三方的 classloader 来加载服务，例如 SpringWebApplicationContext 等。

要在 Spring bean 中引用 OSGi Service，只需在 spring bean 的 xml 中这么配置：

<osgi:reference id=”相当于 Spring bean name” interface=”服务接口” cardinality=”0..n”/>

可以看出和 ds 中的配置基本是一致的，不过目前尚缺少对于 policy 这样的属性的配置，另外如要实现 bind 和 unbind 的设置，需要在 osgi:reference 的元素下增加 osgi:listener 的节点，例如：

```
<osgi:reference>
  <osgi:listener ref=”引用了此 service 的 spring bean”
                 bind-method=”服务可用时调用的方法”
                 unbind-method=”服务不可用时的调用的方法”/>
</osgi:reference>
```

这里 bind-method 和 unbind-method 的方法签名和在使用 DS 时稍有不同，这里 bind-method 和 unbind-method 的方法签名如下所示：

public void some-method-name(服务接口，Dictionary)

在引用 OSGi Service 的配置上 Spring-OSGi 还支持 filter、timeout、depends-on 和 context-classloader 这些属性配置，filter 属性用于配置查找的目标服务的过滤条件（例如版本、名称、属性等）；timeout 属性用于配置在服务不可用时等待的重试时间；depends-on 用于配置此引用的服务所依赖的服务；context-classloader 配置使用第三方的 classloader（如 spring 的 WebcontextClassloader 等）来加载此 OSGi 服务。

3.2.3. 重构留言板列表模块

在学习了怎么发布和引用 OSGi 服务后，可以开始动手来重构之前的留言板系统了，在此以留言板列表模块为例来说明具体如何使用 Spring-OSGi，同时体会 Spring-OSGi 所带来的优缺点。

Spring-OSGi 默认加载解析 META-INF/spring 目录下的 xml 文件作为 spring bean 配置文件，也可通过在 MANIFEST.MF 中增加 Spring-Context 来指定 spring bean 配置文件。

按照留言板列表模块的设计，其中留言板列表扩展点服务和留言板列表服务是不需要列入 DS 范畴管理的，因此基于 Spring-OSGi 重构后将把其重构为 Spring bean，留言板列表 Command 服务则可基于 Spring 的 DI 方式注入所需的服务。

- 定义 Spring bean；

  这步对于使用过 Spring 的用户而言是一件比较简单的事，在 META-INF 目录下建立 spring 目录，在此目录下新建 springbeans.xml；

  在此 xml 中定义留言板列表模块中的 bean，定义完毕后如下所示：

```
<bean name="bulletinListExtensionService"
class="cn.org.osgi.bulletin.service.impl.BulletinListExten
sionComponent"/>
  <bean name="bulletinListService"
class="cn.org.osgi.bulletin.service.impl.BulletinListCompo
nent"/>
  <bean name="bulletinListCommand"
class="cn.org.osgi.bulletin.mvc.command.BulletinListComman
d">
    <property name="bulletinListService"
ref="bulletinListService"/>
    <property name="extensionService"
ref="bulletinListExtensionService"/>
 </bean>
```

  在上面的定义中可能会引起大家疑问的地方是 bulletinListExtensionService 明明是依赖 IExtensionRegistry 服务的，为什么没定义呢，还有 bulletinListService 明明是依赖 CommonDaoService 的，原因就在于这两个 bean 所依赖的都是其他 bundle 所提供的服务，对于其他 bundle 所提供的服务仍然遵照 OSGi 的方式使用，也就是要通过引用 OSGi 服务的方式来配置。

- 定义所需引用的 OSGi 服务；

  对于所需引用的 OSGi 服务，推荐采用另外的单独的文件来配置，因此在 spring 目录下新建一个 osgiservces.xml，配置 bulletinListService 和 bulletinListExtensionService 所需的服务，配置完毕后如下所示：

```
<osgi:reference id="commonDaoService"
interface="cn.org.osgi.module.hibernate.service.CommonDaoS
ervice">
    <osgi:listener ref="bulletinListService"
```

```
    bind-method="setCommonDaoService"
    unbind-method="unsetCommonDaoService"/>
  </osgi:reference>
  <osgi:reference id="extensionRegistry"
interface="org.eclipse.core.runtime.IExtensionRegistry">
    <osgi:listener ref="bulletinListExtensionService"
    bind-method="setRegistry"
    unbind-method="unsetRegistry"/>
 </osgi:reference>
```

可以看到，对于需要引用 OSGi 服务的 bean 在这里采用了 osgi:listener 的方式，而不是直接在 bean 中采用<property name="" ref=""/>的方式，这是为了维持 OSGi 系统的动态性，以便引用此 OSGi 服务的 bean 能够动态的获取服务的状态。

- 定义所需发布的 OSGi 服务；

  留言板列表模块中的留言列表 Command 服务需要发布为 OSGi 服务，仍然在 osgiservices.xml 中进行配置，配置完毕后如下所示：

```
<osgi:service id="BulletinListCommandService"
 ref="bulletinListCommand"
 interface="cn.org.osgi.opendoc.bulletin.service.WebCommand
 " lazy-init="true">
 <osgi:service-properties>
    <prop key="command">LIST</prop>
 </osgi:service-properties>
 </osgi:service>
```

- 运行重构后的留言板系统；

  在运行前，需要首先将之前以 DS 方式定义的 OSGi 服务的部分删除，这步非常简单，直接把留言板列表模块中的 OSGi-INF 目录删除；

  打开 MANIFEST.MF，将其中的 Service-Component 项删除；

  在之前的运行的留言板系统的插件中增加 Spring-OSGi 所需的插件，点击运行；

  运行后访问http://localhost/bulletin，会出现Command【LIST】服务不可用的错误，造成这个错误的原因是留言板列表模块和Spring-OSGi的那些插件是同时启动的，而根据Spring-OSGi的要求，留言板列表模块是要晚于Spring-OSGi启动的；调整Spring-OSGi的那些插件的StartLevel为 2，重新启动留言板系统，通过 http://localhost/bulletin访问，可看到一切和之前的留言板系统保持一样了。

按照以上的步骤对其他的 Bundle 进行重构，重构完毕后即将这套留言板系统改造为

基于 OSGi+Spring+Hibernate 了。

3.2.4. 小结

通过本章节学习了如何基于 Spring-OSGi 来实现 OSGi 与 Spring 的集成，与 Spring 的集成使得基于 OSGi 的应用能够很容易的实现企业应用的一些常见需求，如事务的控制、远程调用等。

在本章节中主要介绍的是 Spring 与 OSGi 的衔接部分，至于如何增强 Spring bean 的功能（如让 bean 具备事务功能等），这些可以参见 Spring 的 Reference，另外 Spring-OSGi 除了本章节中所写的 OSGi 服务的配置外，还支持 osgi:bundle、osgi:config 等等配置，具体可参见：

http://blog.csdn.net/shuyaji/archive/2006/11/17/1393272.aspx Spring-OSGi 规范 V0.7 中文版

http://www.springframework.org/osgi/specification Spring-OSGi规范V0.7 英文官方版

Spring-OSGi 带来的优势有：

- Bundle 内服务的注入支持，这样就不需要把内部的服务发布为 OSGi Service 了；
- 企业应用所需的技术的支撑，例如事务管理、远程调用，Spring 的这些功能支撑增强了 OSGi 进军企业应用领域的资本，避免了基于 OSGi 去做企业应用时需要做 N 多的重复性的工作；
- 支持以第三方的 classloader 去加载 OSGi Service，如 WebApplicationContextClassLoader 等。

Spring-OSGi 值得改进的地方：

- 形成 Spring microKernel，目前的 Spring kernel 仍然是太大了，例如在留言板应用上需要的其实只是 spring core 的功能，但却需要引用到非常多的 bundle，也许侧面反应了 Spring 的模块化做的也不够的好；
- Spring 本身所支持的功能的模块化的支撑，例如 Spring-DAO、Spring-MVC 这些，应该做到能够切实的支撑模块化的开发；
- 无需设置启动顺序，需要设置启动顺序来启动基于 OSGi 的应用被认为是 OSGi Bad Smell 中非常典型的一种，在这点上 Spring-OSGi 也许可以考虑使用扩展点的方式来实现；
- 动态性方面，目前的 Spring-OSGi 在动态性方面还没有达到 OSGi 所支持的效果，例如 Bundle 更新时，并没有同时更新其所引用和发布的 OSGi Service 的信息。

鉴于 Spring 强大的 POJO Enhanced 功能，当 Spring-OSGi 提供的功能完全和 DS 匹配时，对于企业应用而言 DS 就可以隐藏在后台了，而 osgi:bundle 这些可为不熟悉 OSGi 的人来继续采用 Spring bean xml 配置的方式来定义 bundle，但在 bundle 这部分我还是更推荐继续采用 OSGi 的方式：分工程，通过 MANIFEST.MF 方式定义 Bundle 的基本信息、import、export 的 Package 等，Spring 和 OSGi 的完美结合是非常值得期待的；但对于非企业应用的一些简单应用而言，则不一定需要 Spring 提供的那些复杂的技术功能的支持，在这种情况下也许 OSGi 在 R5 时应该考虑为 DS 增加 scope 的概念。

3.3. 解决和 Webwork 的集成

要集成 Webwork，要解决的问题和与 Hibernate 的集成基本类似，集中在如下三个方面：

- Webwork 要求所有的 Action 配置文件都需要在其统一的 xwork.xml 做 include 配置，而基于 OSGi 后需要达到的目标是：各模块维护自己的 xwork.xml，以某种方式绑定到 webwork 上去；
- Webwork 加载 Action Class 要求这些 Action Class 必须处于同一 Classloader 下，而基于 OSGi 后 Action Class 的 Classloader 与 Webwork 的则不同，还要解决的一个问题是如何让这些 Action Class 能够注入 Spring bean 或者 OSGi 服务；
- Velocity 加载页面模板文件时可采用 classloader 和文件路径两种方式进行加载，基于 OSGi 后各模块的页面位于不同的 classloader 和不同的文件路径下；

要解决这三个问题，就得分析 webwork 是如何装载、解析 Action 配置文件；如何加载 Action Class 以及相关的 Interceptor Class：

- 如何装载、解析 Action 配置文件

  Webwork 基于 Xwork 而构建，Action 配置文件的装载、解析均由 Xwork 来完成，跟踪 webwork 的启动流程后，在 Xwork 的 DefaultActionProxy 的构造器中可看到这么一行代码：

  config = ConfigurationManager.getConfiguration().getRuntimeConfiguration().getActionConfig(namespace, actionName);

  继续跟踪 ConfigurationManager，在 getConfiguration 方法中可看到实例化了 DefaultConfiguration，并调用了它的 reload 方法；

  跟踪 DefaultConfiguration.reload 方法，可看到如下方法：

```
for (Iterator iterator = ConfigurationManager.getConfigurationProviders().iterator();
                iterator.hasNext();) {
            ConfigurationProvider provider = (ConfigurationProvider)
iterator.next();
            provider.init(this);
        }
```

回到 ConfigurationManager 的 getConfigurationProviders 方法，默认情况下 Xwork 仅使用了 XmlConfigurationProvider，跟踪查看 XmlConfigurationProvider 的 init 方法，在其中可找到如下代码：

```
loadConfigurationFile(configFileName, null);
```

跳至此方法，在这个方法中，Xwork 完成了加载和解析 Action 配置文件的工作，在此方法中首先读取位于 classpath 下的 xwork.xml，并解析此 xml 中的内容，如所碰到的子元素节点为 package，则直接调用 addPackage 来加载 Package 中的元素信息；如所碰到的子元素节点为 include，则再次调用 loadConfigurationFile 来加载该文件中的元素，这个方法中最为关键的代码是这行：

```
is = getInputStream(fileName);
```

这行表现了 XmlConfigurationProvider 是如何寻找 Action 配置文件的，getInputStream 方法通过这么一行来将指定名称的 Action 配置文件解析为 InputStream：

```
return FileManager.loadFile(fileName, this.getClass());
```

查看 FileManager 的 loadFile 方法，可发现最后是通过 com.opensymphony.util 包中的 ClassLoaderUtil.getResource 来获取文件流的，查看 ClassLoaderUtil 的 getResource 方法：

```
public static URL getResource(String resourceName, Class
callingClass)
    {
        URL url =
Thread.currentThread().getContextClassLoader().getResource(reso
urceName);
        if(url == null)
            url =
(com.opensymphony.util.ClassLoaderUtil.class).getClassLoader().g
etResource(resourceName);
```

```
            if(url == null)
            {
                ClassLoader cl = callingClass.getClassLoader();
                if(cl != null)
                    url = cl.getResource(resourceName);
            }
            if(url == null && resourceName != null &&
 resourceName.charAt(0) != '/')
                return getResource('/' + resourceName, callingClass);
            else
                return url;
        }
```

根据这个方法可看出，Xwork 在加载 Action 配置文件的顺序为：

- 使用当前线程的 classloader 加载文件；
- 使用 ClassLoaderUtil 类的 classloader 加载文件；
- 使用调用类的 classloader 加载文件；
- 尝试在文件名前加上“/”，再次调用 getResource 进行尝试。

经过上面的分析，可看出 Xwork 通过 XmlConfigurationProvider 来完成 Action 配置文件的装载和解析。

- 如何加载 Action Class 以及相关的 Interceptor Class

跟踪 Webwork 的执行流程，可看到是通过 DefaultActionInvocation.invoke 来执行 Action 的，DefaultActionInvocation 通过 createAction 方法来创建 Action 的实例，在这个方法中最重要的代码如下所示：

```
action = ObjectFactory.getObjectFactory().buildAction(proxy.getActionName(),
proxy.getNamespace(), proxy.getConfig(), contextMap);
```

跟踪 ObjectFactory 的执行，发现无论是 Action Class 还是 Interceptor Class，最终都是通过 getClassInstance 方法来加载类的，这个方法的代码如下：

```
if (ccl != null) {
    return ccl.loadClass(className);
}

return ClassLoaderUtil.loadClass(className, this.getClass());
```

对于 Continuation 方式的 package 而言，通过 ContinuationsClassLoader 来加载 class，而普通方式的 package，则通过 ClassLoaderUtil 来加载 class，ClassLoaderUtil

在加载 class 时采用如下顺序：

- 使用当前线程的 classloader 加载；
- 直接使用 Class.forName 的方式加载；
- 使用 ClassLoaderUtil 类所在的 classloader 加载；
- 使用调用 class 的 classloader 进行加载。

如何让 Action Class 能够注入 Spring bean 或 OSGi Service 是我们最为关注的问题，所幸的是 Webwork 本身就提供了和 Spring 的集成的支持，加上之前已经完成了 OSGi 和 Spring 的集成，因此这里我们只需要考虑如何让 Webwork 能够和 Spring-OSGi 来进行集成了，跟踪 Webwork 和 Xwork 的代码可发现 Webwork 实现和 Spring 的无缝集成是通过 Webwork 中的 WebworkSpringObjectFactory 和 Xwork 的 SpringObjectFactory 来实现的，也就是说如果你将 webwork 的 objectfactory 设置为 spring 时，Xwork 在加载 Action Class 时就不会使用之前的 ObjectFactory 了，而是使用 WebworkSpringObjectFactory 来加载，要实现与 Spring-OSGi 的集成，可通过编写一个新的 SpringOSGiObjectFactory 来实现了。

在了解了 webwork 如何加载、解析 Action 配置文件和如何加载 Action Class 后，来写一个封装 webwork 的 Bundle，从而支持 Action 配置文件和 Action 类在各个模块中独立编写。

要支持各模块中编写的 Action 配置文件和 Action 类能够绑定到 webwork 上，同时又不修改 webwork 模块的东西，可以通过扩展点的方式将 Action 配置文件注册到 webwork 模块，并同时编写一个支持动态加载 Action 配置文件的对象，至于 Action 配置文件的解析方法可以完全照抄 XmlConfiguration 的写法；另外要解决的就是 webwork 模块加载 Action/Interceptor 类的问题了，如果只是加载 Action/Interceptor 类的话，可以通过 DynamicImport-Package 的方式来实现，但现在我们还得实现 Action Class 注入 Spring bean 和 OSGi 服务的支持，就得参照 WebworkSpringObjectFactory 来编写一个 SpringOSGiObjectFactory 来实现了。

- 定义绑定 Action 配置文件的扩展点

  在定义扩展点之前，要首先创建封装 Webwork 的 Bundle，此 Bundle 需要放入 webwork 编译和运行所依赖的 jar 包，在完成此步工作后，开始定义扩展点的工作；

  根据上面的分析可知，在扩展点中需要定义的为需要加载的 Action 配置文件，在

封装 webwork 的 Bundle 中增加一个 ActionExtension 的扩展点配置，在此扩展点中增加一个 action 的节点元素，该节点元素的属性仅需要一个，即 Action 配置文件的路径和名称。

定义好扩展点后就可以来编写加载和解析 Action 配置文件的类了，此类和默认的 XmlConfiguration 不同的地方不是通过 xwork.xml 来加载 action 的配置文件，而是通过监控扩展点的方式来进行加载，在扩展点发生变化时需要动态的进行 action 配置的加载和卸载，由于其他的功能和 XmlConfiguration 相同，复制 XmlConfiguration 进行修改来完成此类的编写，并重新命名为 XmlExtensionConfigurationProvider，首先在 XmlExtensionConfigurationProvider 初始化时调用 IExtensionRegistry 获取到系统中的 Action 的扩展，并调用 loadConfigurationFile 方法来完成对于 Action 配置文件的加载；接着实现 IRegistryChangeListener 接口，以实现动态的根据 Action 扩展点实现的变化来做出相应的处理，此接口方法实现的代码示例如下：

```
    /*
     * (non-Javadoc)
     * @see
org.eclipse.core.runtime.IRegistryChangeListener#registryCha
nged(org.eclipse.core.runtime.IRegistryChangeEvent)
     */
    public void registryChanged(IRegistryChangeEvent event) {
      IExtensionDelta[]
deltas=event.getExtensionDeltas(NAMESPACE,EXTENSION_POINT);
      for (int i = 0; i < deltas.length; i++) {
          IConfigurationElement[]
elements=deltas[i].getExtension().getConfigurationElements()
;
          switch (deltas[i].getKind()) {
          case IExtensionDelta.ADDED:
              for (int j = 0; j < elements.length; j++) {

  loadConfigurationFile(elements[i].getAttribute("configFile
"), null);
              }
              break;
          case IExtensionDelta.REMOVED:
              for (int j = 0; j < elements.length; j++) {
                  includedFileNames.remove(elements[j]);
                  String
```

```
fileName=elements[j].getAttribute("configFile");

  if(configFilePackages.containsKey(fileName)){
                    List packages=(List)
configFilePackages.get(fileName);
                    for (Iterator iter = packages.iterator();
iter
                            .hasNext();) {
                        String packageName = (String)
iter.next();

  configuration.removePackageConfig(packageName);
                    }
                }
            }
            break;
        default:
            break;
        }
    }
}
```

另外一个问题就是这些配置文件的加载，在分析 webwork 对于 Action 配置文件的加载时 xwork 是使用 ClassLoaderUtils 来获取的，要在 OSGi 中让 Webwork 模块能够加载到其他模块中的 Action 配置文件，就得将这些 Action 配置文件放入对外提供的 package 中，同时在 webwork 模块中加上 DynamicImport-Package:*，这样就可以在 webwork 中通过 ClassLoaderUtils 来加载到其他模块中的 Action 配置文件了。

- 支持加载其他模块中的 Action/Interceptor 类，并与 Spring-OSGi 进行集成

  在加载 Action/Interceptor 类时，Webwork 是通过 ObjectFactory 来完成的，在此编写一个新的 SpringOSGiObjectFactory 的类，以便加载的 Action 能和 Spring-OSGi 进行集成，这个 SpringOSGiObjectFactory 对于两个方法的实现如下：

```
/*
   * (non-Javadoc)
   * @see
com.opensymphony.xwork.ObjectFactory#buildBean(java.lang.Str
ing, java.util.Map)
   */
  public Object buildBean(String className, Map extraContext)
throws Exception {
```

```
            BundleContext context=WebworkActivator.getContext();
            ServiceReference[]
        serviceRefs=context.getAllServiceReferences(IAction.class.ge
        tName(), "(command="+className+")");
            if((serviceRefs==null)||(serviceRefs.length==0)){
                Class clazz=getClassInstance(className);
                return super.buildBean(clazz, extraContext);
            }
            else if(serviceRefs.length>1){
                throw new Exception("可用的Action服务：【"+className+"】
        大于1");
            }
            return context.getService(serviceRefs[0]);
          }
    /*
     * (non-Javadoc)
     * @see
com.opensymphony.xwork.ObjectFactory#getClassInstance(java.lang.Strin
g)
     */
    public Class getClassInstance(String className) throws
ClassNotFoundException {
       Class clazz=null;
       if(useClassCache)
           clazz=(Class) classes.get(className);
       if(clazz==null){
           BundleContext context=WebworkActivator.getContext();
           try{
               ServiceReference[]
serviceRefs=context.getAllServiceReferences(IAction.class.getName(),
"(command="+className+")");
               if((serviceRefs==null)||(serviceRefs.length==0)){
                   clazz=super.getClassInstance(className);
                   if(useClassCache)
                       classes.put(className, clazz);
               }
               else if(serviceRefs.length>1){
                   throw new Exception("可用的Action服务：【"+className+"】
大于1");
               }
               else{

    clazz=context.getService(serviceRefs[0]).getClass();
               }
```

```
            }
            catch(Exception e){
                throw new ClassNotFoundException(e.toString());
            }
        }
        return clazz;
            }
```

从上面的方法中可以看出，对于 Action 类的加载以及和 Spring-OSGi 的集成采用的方法是将 Action 以 OSGi 服务的方式对外 Export，这种方法和之前 SimpleMVCFramework 的方法是基本相同的，在加载 Action 上和 Webwork 本身的加载方式还有一个不同的地方在于 Webwork 采用的是通过 Action 的 ClassName 进行加载，而 SpringOSGiObjectFactory 则是将这个 className 作为 OSGi 服务的 filter 来使用的，也就是在 action 的配置文件中对于 action 的 class 中应使用的是和 Action OSGi 服务中的 command 属性相同的值。

在解决了上面两个问题后，需要将 Webwork 改为使用上面两个类来加载 Action 配置文件信息和加载 Action 类：

- 使用 XmlExtensionConfigurationProvider 来加载 Action 配置文件信息

  由于在 ConfigurationManager 中使用的默认的 XmlConfigurationProvider 不是配置出来的，因此只能复制 ConfigurationManager 的代码到封装 webwork 的 bundle 中，并将其中的 getConfigurationProviders 方法中的 *configurationProviders*.add(**new** XmlConfigurationProvider()); 修改为 *configurationProviders*.add(**new** XmlExtensionConfigurationProvider ());。

- 使用 SpringOSGiObjectFactory 来加载 Action 类

  参照 Webwork 在集成 Spring 时的做法，需要修改 DispatcherUtils 类 init(ServletContext)方法，在 **if** (className.equals("spring")) {这段中再加上

```
else if(className.equalsIgnoreCase("springosgi")){
            className =
"cn.org.osgi.xwork.springosgi.SpringOSGiObjectFactory";
}
```

  在完成上面的修改后，只需将 webwork.properties 中的 webwork.objectFactory 修改为 springosgi，在以后的运行中就会使用 SpringOSGiObjectFactory 来加载 Action 类。

做好了这些准备后，以留言板列表模块为例来将以前基于 SimpleMVCFramework+Spring+Hibernate 的结构重构为基于 Webwork+Spring+Hibernate 的结构。

- 去除留言板列表模块中使用到 SimpleMVCFramework 的部分

  在留言板列表模块中使用到 SimpleMVCFramework 有 Controller 的注册和 WebCommand 的部分。

  Controller 的注册部分是用于完成 Servlet 的注册的，Webwork 采用 ServletDispatcher 作为 Controller，因此需要通过 HttpService 来注册 ServletDispatcher，在封装 Webwork 的 Bundle 需要将注册 Controller 部分作为扩展点对外提供，这样基于 webwork 的应用才可以注册自己的 web context，这里要注意的是 HttpService 目前尚不支持*这种通配符的注册方式，也就是说不能像通常使用 webwork 时采用*.action 这样的方式来做 servlet-mapping，另外目前 HttpService 暂时不支持 filter，因此也没办法直接使用 webwork 新版本的 filter 的处理方式。

  在封装 webwork 的 bundle 中编写 webwork controller 的扩展点，该扩展点对外提供的为 web context 的注册，在此扩展点管理的组件中要注意的是在初始化 ServletDispatcher 后，在 httpService 注册此 Servlet 后需要显式的调用 ServletDispatcher 的 init 方法：

```
  ~~ServletDispatcher~~ controller=new ServletDispatcher();
  try {
    httpService.registerServlet(webcontext, controller, null,
null);
     controller.init();
     info("已注册："+webcontext);
  }
  catch (Exception e) {
     error("注册扩展的："+webcontext+"失败",e);
  }
```

  在留言板列表模块中通过修改 plugin.xml 来注册 controller 的扩展，将其映射到 /bulletin 的 web context 上：

```
<extension point="WebworkModule.ControllerExtension">
  <controller webcontext="/bulletin"/>
 </extension>
```

打开留言板列表模块的 MANIFEST.MF，去除其中对于 SimpleMVCFramework Export Package 的 Import。

- 重构 Action

  在去除了对 SimpleMVCFramework 的依赖后，开始重构 Action 部分的代码，对于留言板列表模块而言也就是 BulletinListCommand 类，将 BulletinListCommand 重构为符合 webwork 的 BulletinListAction，对于使用 request 提取的参数改为使用注入的方式获取，在重构完毕后相应的修改 springbeans.xml 和 osgiservices.xml 中关于此 Action 的描述。

  在重构完 Action 后，要做的事就是编写 Action 配置文件，编写完毕后示例如下：

```
<package name="bulletin-list" extends="webwork-default"
namespace="/bulletin">
      <action name="list" class="LIST">
          <interceptor-ref name="defaultStack" />
          <result name="list"
type="velocity">/cn/org/osgi/bulletin/list/pages/list.vm</re
sult>
      </action>
 </package>
```

  这个配置文件中之前没交代的部分为 result 页面跳转的部分，在留言板中采用的是 velocity 的方式，velocity 在加载其模板页面时支持以文件路径的加载方式和 classpath 的加载方式，在这里采用 classpath 方式进行加载，这样各模块只需将其模板页面所在的 package 对外 export，velocity 即可加载到相应的模板页面，否则会出现 ResourceNotFound 的错误。

  在重构完 Action 后，将 Action、Action 配置文件以及 Action 所用到的模板页面所在的 package 对外 Export。

- 注册 Action

  按照封装 webwork bundle 提供的扩展点注册 Action 配置文件，在 plugin.xml 中加上如下内容：

```
<extension
          point="WebworkModule.ActionExtension">
       <action
   configFile="cn/org/osgi/bulletin/list/action/xwork.xml"/>
</extension>
```

完成了上面的步骤后，重新启动留言板系统，此时暂时先不安装其他的留言板模块和

SimpleMVCFramework模块，通过http://IP:PORT/bulletin/list.action这样的方式来访问留言板列表，运行后会出现如下错误：

HTTP ERROR: 404

Not Found

这是为什么呢？跟踪 Webwork 的代码发现是由于 DefaultActionMapper 在处理 uri 时是按照*.action 这样的 servlet-mapping 的方式来获取 Action 名称的，而由于 HttpService 暂时不支持，因此要修改 DefaultActionMapper 对于 getUri 这个地方的处理，查看 webwork.properties 发现这个是可以配置的，因此编写一个新的 DefaultActionMapper，对 getUri 的部分进行修改，将 getUri 中的这行注释掉：

//uri = RequestUtils.getServletPath(request);

修改 webwork.properties，将 webwork.mapper.class 修改为使用新的 DefaultActionMapper。

重新启动留言板系统，再次访问留言板列表页面，OK，成功看到和之前一样的页面了，到此为止，宣告重构完成，可按照同样的方法对其他的留言板模块进行重构。

3.4. 小结

经过上面的步骤后，完成了 OSGi 与 Hibernate、Spring 和 Webwork 的集成，建立起了 OSGi+Hibernate+Spring+Webwork（OHSW）的脚手架，来全面的看看基于这个脚手架的环境的搭建、开发的方式以及部署的方式，以在项目/产品中能够使用 OSGi+Hibernate+Spring+Webwork（OHSW）这样的结构体系：

3.4.1. 开发环境的搭建

开发环境的搭建其实就是完成 OSGi+Hibernate+Spring+Webwork 脚手架的搭建，在经过了上面的分析和封装的过程后，整个脚手架的搭建还是比较容易的。

将Spring-OSGi所需的bundle（具体请参见解决和Spring的集成）、HibernateModule以及WebworkModule部署至OSGi框架中即可，最终形成的OHSW脚手架由以下Bundle构成：

- org.springframework.osgi.aopalliance.osgi
- org.springframework.osgi.aspectjrt.osgi
- org.springframework.osgi.backport-util-concurrent
- org.springframework.osgi.spring-aop
- org.springframework.osgi.spring-aspects

- org.springframework.osgi.spring-beans
- org.springframework.osgi.spring-context
- org.springframework.osgi.spring-core
- org.springframework.osgi.spring-dao
- org.springframework.osgi.spring-osgi-core
- org.springframework.osgi.spring-osgi-extender
- org.springframework.osgi.spring-osgi-io
- HibernateModule
- WebworkModule
- javax.servlet
- org.apache.commons.logging
- org.eclipse.equinox.common
- org.eclipse.equinox.ds
- org.eclipse.equinox.http.jetty
- org.eclipse.equinox.http.registry
- org.eclipse.equinox.http.servlet
- org.eclipse.equinox.log
- org.eclipse.equinox.registry
- org.eclipse.equinox.servlet.api
- org.eclipse.osgi
- org.eclipse.osgi.services
- org.mortbay.jetty

3.4.2. 开发方式

- **持久层的开发（基于 Hibernate）**

  持久层的开发方式和传统的基于 Hibernate 的开发不会有什么太大的区别，仅在于将以前修改 hibernate.cfg.xml 来绑定 po 的方式改为了通过编写 plugin.xml 来绑定 PO，并将 PO 所在的 package 对外 export。

- **业务层的开发（基于 Spring）**

  业务层的开发上对于熟练使用 Spring 的开发人员也没有太多的变化，只是要掌握什么情况下需要发布和引用 OSGi 服务（通常来讲就是要在模块外使用的服务就

发布为 OSGi 服务，需要在模块内引用其他模块的服务就通过 OSGi 服务的方式来引用），并充分保持使用 OSGi 服务的 bean 的动态性，记住 OSGi 服务是随时来、随时走的。

而对于不熟悉 Spring 的开发人员而言则应根据自己的需求来学习 Spring 的相关部分，例如学习 Spring-DAO 等；另外还需要学会将不需要作为 OSGi 服务的组件通过编写 Spring bean 配置文件来实现依赖注入。

- **Web 响应层的开发（基于 Webwork）**

  Web 响应层方面的开发方式和传统的基于 Webwork 的开发方式基本上没有什么区别，不同之处在于不再通过在 xwork.xml 中增加 include 元素来绑定 Action 配置文件，而是修改 plugin.xml 的方式来绑定 Action 配置文件，同时要记得将 Action 类、Action 配置文件以及页面模板文件所在的 package 对外 Export。

3.4.3. 部署方式

和传统的 Hibernate+Spring+Webwork 的部署方式不同的地方在于，现在各个模块是以独立的 project 的方式存在和部署的，并且所有的一切都可以动态的进行，部署的方法遵照 OSGi Bundle 的部署方式。

基于 OSGi+Hibernate+Spring+Webwork 这样的脚手架使得熟悉 Hibernate+Spring+Webwork 的人员很容易就能获得 OSGi 所带来的好处，当然目前的这个脚手架还有很多值得完善的地方，也欢迎大家来根据自己项目/产品的情况共同完善这个脚手架。

本章节较为详细的分析了如何实现 OSGi 与 Hibernate、Spring、Webwork 这些流行开源框架的集成，按照文中类似的方法相信大家能够很好的实现 OSGi 与 struts2、tapestry、iBatis 等等集成的脚手架，当然，最好还是大家用到的这些开源框架都能提供和 OSGi 集成的版本，就像 Spring-OSGi 一样，目前有的好消息是 Apache 的 Struts 2 准备开始实现和 OSGi 的集成，鉴于 struts 的巨大用户群，这对于 OSGi 的推广使用无疑会起到很大的作用。

在此推荐几篇网上的关于 OSGi 与流行 Java 框架集成的例子的文章：

Developing Equinox/Spring-OSGi/Spring Framework Web Application：

http://www.blogjava.net/Phrancol/articles/143084.html

DWR+OSGi 整合：

http://www.cnblogs.com/ycoe/archive/2007/09/19/898092.html

OSGi 上部署 Hibernate 的四种方式：

http://www.javaeye.com/topic/40364

Wicket 如何通过 OSGi 框架注入 Jetty：

http://www.javaeye.com/topic/116150

让 OSGi 支持 JSF Web 开发：

http://www.blogjava.net/Andyluo/archive/2007/10/08/jsf-support-in-osgi.html

4. 基于 OSGi 搭建分布式系统

OSGi 规范是基于同一 VM 而设计的，因此并无对于分布式通讯的支持，分布式的通讯有很多种实现方法，如采用 RMI、基于 MQ、SOA 等方式，SOA 无疑是目前分布式领域中最为流行的架构体系，在本章节中就以一个实例来分析基于 OSGi 如何搭建一个实现像 SOA 这样的脚手架。

4.1. 实例需求

仍然以留言板系统为例，现在要求管理员登录的校验服务部署至另外的一台机器上，这也就意味着需要能够分布式的调用其他 OSGi 应用中的服务。

4.2. 搭建基于 OSGi 的分布式系统的脚手架

根据需求，要求能够远程调用其他 OSGi 应用中的服务，SOA 是一个典型的分布式架构体系，在 SOA 的架构体系中，要求系统以服务的方式对外提供功能，同时以发起服务请求的标准方式获取其他系统提供的服务，OSGi 是按照面向服务的思想而设计的，因此其和 SOA 架构体系可以很好的融合在一起，区别就在于 OSGi 是针对同一 VM 环境下的，并不支持分布式，为了让 OSGi 应用能够互相通讯，可采用 web service 来实现，在这里我们来搭建一个这样的脚手架，使得 OSGi 应用之间可通过 web service 来进行通讯。

- 封装 Axis

  在此我们选择 Axis 做为脚手架中支持 web service 方式调用的框架，参照 Axis 部署至应用服务器的方法，建立一个如下目录结构的插件工程：

  AxisBundle

  | -- src

  | -- WEB-INF

  | -- server-config.wsdd

  | -- lib

  | -- META-INF

  | -- MANIFEST.MF

  将 Axis 所需的 jar 放入 lib 目录中，编写一个 Component 调用 HttpService 注册 Axis 中的 AxisServlet 和 AdminServlet。

  在注册完毕 AxisServlet 和 AdminServlet 后，要显示的调用下这两个 servlet 的 init

方法：

```
AxisServlet axisServlet=new AxisServlet();
     AdminServlet adminServlet=new AdminServlet();
     if(http!=null){
         try {

  http.registerServlet("/"+AXIS_WEB_CONTEXT+"/services",(Ser
vlet)axisServlet, null, null);

  http.registerServlet("/"+AXIS_WEB_CONTEXT+"/servlet/AdminS
ervlet",adminServlet, null, null);
             axisServlet.init();
             adminServlet.init();
             info("成功注册Axis相关Servlet!");
         }
         catch (Exception e) {
             error("注册Axis相关Servlet时出现错误", e);
         }
```

- 提供调用远程 OSGi 服务的接口

  此插件工程需要对其他 Bundle 提供调用远程 OSGi 服务的接口，这个接口的代码如下所示：

```
/**
   * 调用远程OSGi服务
   *
   * @param serviceName 服务接口名
   * @param methodName 方法名
   * @param target OSGi服务所在机器IP
   * @param port Web服务端口
   * @param args 传入参数
   * @return Object
   */
  public Object invokeService(String serviceName,String
methodName,String target,String port,Object[] args);
```

  此接口按照 Axis 调用 webservice 的方法实现即可，具体请参见 RemoteOSGiServiceCallerComponent 代码。

- 提供执行同一 VM 中的 OSGi 服务的 WebService

  此 WebService 根据请求的服务名获取同一 VM 中的 OSGi 服务，基于 java 的反射机制传入参数执行并返回相应的值。

  接口代码如下所示：

public Object invoke(String serviceName,String methodName,Object[] args);

在编写完此类后在 server-config.wsdd 中加入如下内容即可将此类以 webservice 的方式对外提供：

```
<service name="OSGiServiceCaller" provider="java:RPC">
    <parameter name="className"
value="cn.org.osgi.module.axis.service.impl.OSGiServiceCalle
r"/>
    <parameter name="allowedMethods" value="*"/>
 </service>
```

经过以上三步即完成了支持远程调用 OSGi 服务的脚手架的搭建，下面基于此脚手架来实现管理员登录的校验的需求。

4.3. 实例的实现

根据需求，管理员登录的校验需要调用位于远程或另一 VM 下的登录校验 OSGi 服务，首先需要建立一个新的插件工程来提供登录校验服务：

这个工程只需要对外 export 登录校验服务的接口即可，此接口的代码示例如下：

```
/**
 * 登录校验服务
 *
 * @param username 用户名
 * @param userpass 密码
 * @return boolean
 */
public boolean check(String username,String userpass);
```

接着建立一个插件工程来实现登录校验服务，并将此实现作为 OSGi 服务对外提供；

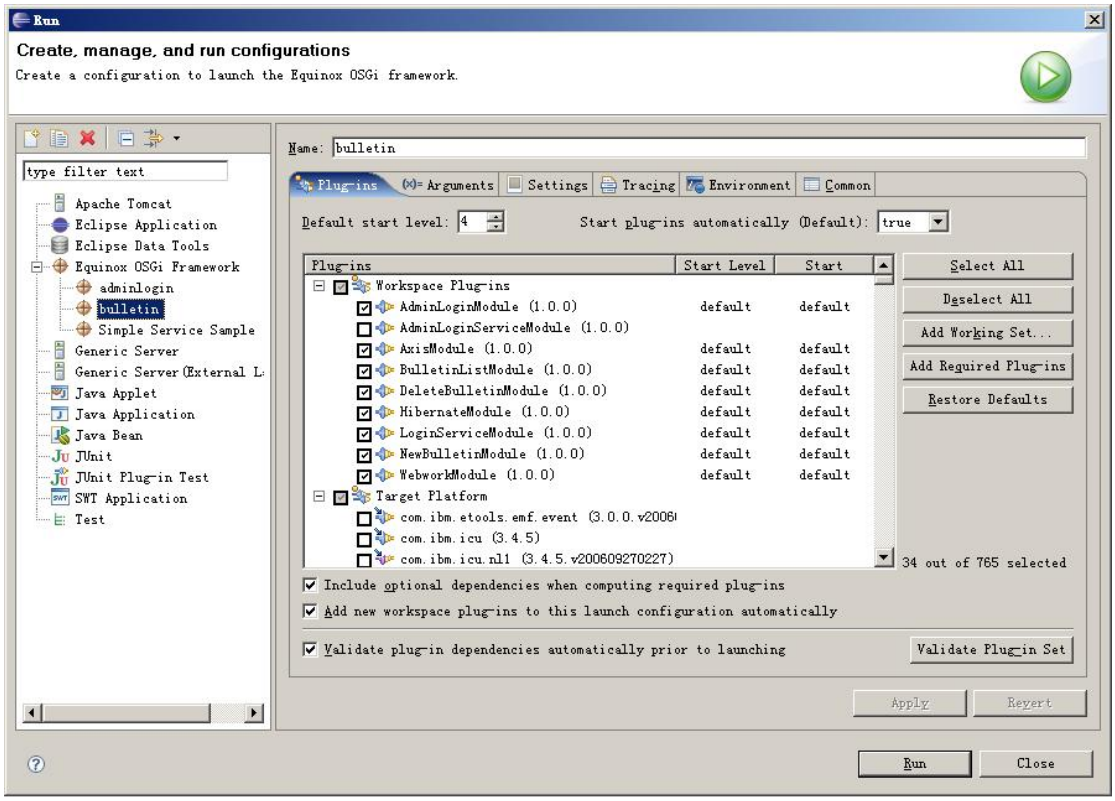
最后就是重构之前的管理员登录模块，将登录校验改为调用远程的 OSGi 服务来实现：

在管理员登录模块中 import 登录服务接口的 package，重构之前的 CheckAdminLoginAction，将其中校验的部分改为调用远程的 OSGi 服务来实现：

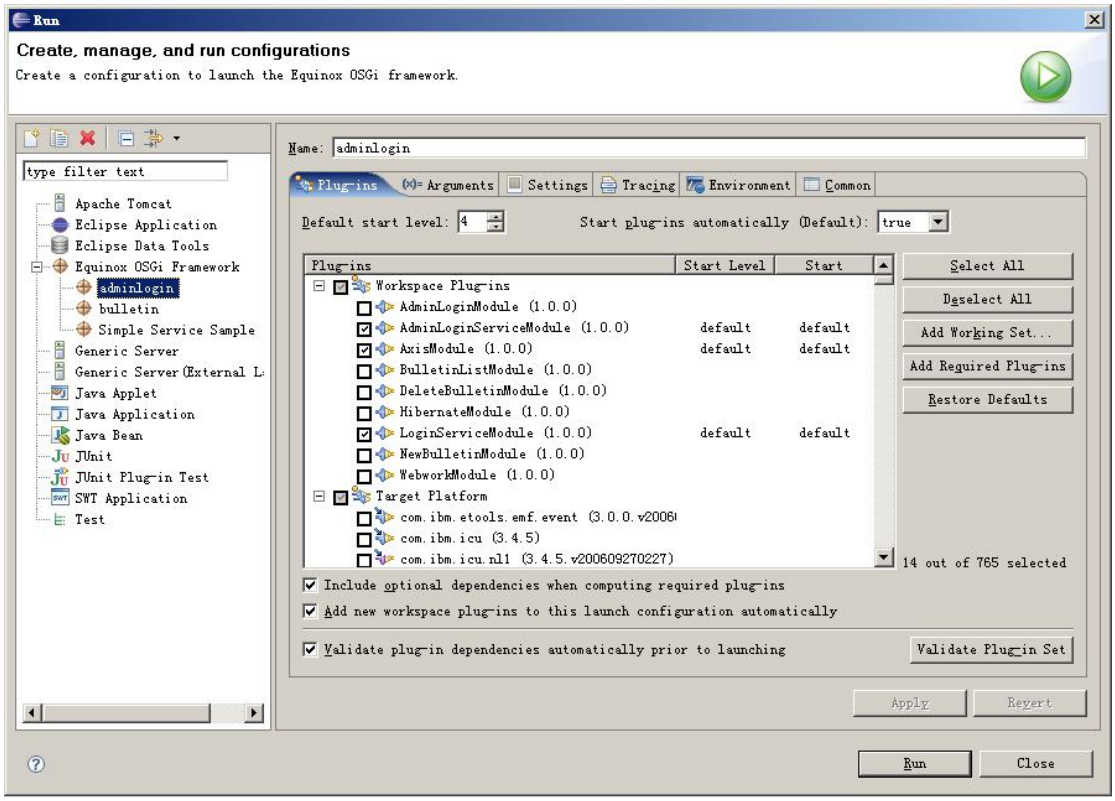
Boolean returnValue=(Boolean) caller.invokeService(LoginService.**class**.getName(), "check", "127.0.0.1", "1220", **new** Object[]{txtUser,txtPass});

在完成以上步骤后，部署新的留言板应用：

根据需求，需要将留言板应用部署为两个 OSGi 应用，其中一个 OSGi 应用中包含了之前留言板应用的所有模块：



另一个 OSGi 应用中仅包含登录校验服务模块、登录校验服务实现模块以及 Axis 封装模块：



将这个 OSGi 应用的 web 端口设置为 1220：-Dorg.osgi.service.http.port=1220

启动这两个 OSGi 应用，在登录时可看到在登录校验的 OSGi 应用的 console 中打出了调用的信息，经过以上的步骤就完成了一个演示分布式调用 OSGi 服务的留言板。

4.4. 小结

在这个章节中实现了一个简单的基于 webservice 的分布式的 OSGi 应用通讯的脚手架，并基于此重构了留言板应用以演示脚手架的作用。

编写这个章节主要是为了说明基于 OSGi 来实现分布式的系统是可行的，尽管这里面还有诸如分布式的事务等等复杂的企业应用领域的问题没有展示如何解决。

不过这个章节中产生的脚手架和SOA架构的实现还是有很大的差距的，SOA需要有服务的集中注册和统一智能寻找的场所，这个可参考或直接采用SCA[3]的实现：Newton[4]或Tuscany[5]，或者大家可以根据自己项目/产品的需求来重构这个脚手架，让它成为一个更加实用的脚手架。

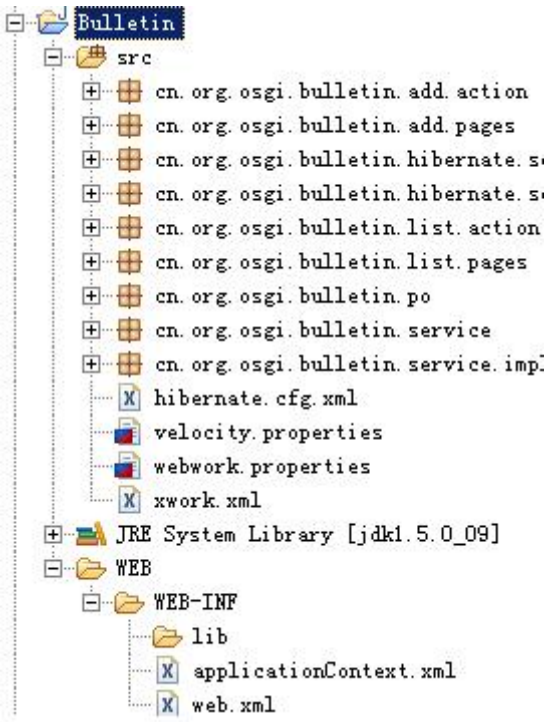
[3] http://www.osoa.org
[4] http://newton.codecauldron.org
[5] http://incubator.apache.org/tuscany

5. 将原系统重构为基于 OSGi 的系统

在看完以上的章节后，希望能让大家在新的项目/产品无所顾忌的使用 OSGi，但对于已存在的项目/产品而言，怎么办呢，因为像产品而言的话，通常都是经历过一段很长时间的发展的，不可能抛弃以前所有的努力，因此还得有将原系统重构为基于 OSGi 的系统的方法。

在这个章节中将把一个传统的基于hibernate+spring+webwork搭建的留言板系统重构为可部署至OSGi框架中的系统，重构时的原则是尽量不去修改原代码，基于BND（Bundle Tool）[6]直接创建符合OSGi规范的Bundle进行部署，从而实现重构。

先来看看传统的留言板系统的工程，工程目录示意如下：



此留言板系统实现了之前例子中的留言板的列表显示和新增留言的功能，重构需要达到的目的就是区分出留言列表显示模块和新增留言模块，并部署至(OHSW)OSGi+Hibernate+Spring+Webwork 的脚手架中，最终的效果则需和传统的单工程部署至 tomcat 等应用服务器的效果一致。

为了尽量不改变现有工程的结构和代码，而又要达到部署到(OHSW)的脚手架中去，需要做到的就是结合现有的代码来创建符合 OSGi 规范的 jar 文件，BND 可以起到这个作用，因此选择了 BND 来实现重构的需求。

[6] 关于BND具体请参见：http://www.aqute.biz/Code/BndCn

5.1. 重构

BND 根据一个 bnd 的描述文件来生成 bundle 的 jar 文件，根据之前对于留言板系统的设计，我们需要将留言列表模块和新增留言模块作为分开的两个 bundle 进行打包生成 jar，同时还需根据(OHSW)的脚手架中的要求来部署 po、action 和 spring bean。

- 留言列表模块

  首先根据设计确定留言列表模块需要包含目前工程中哪些 package：

  cn.org.osgi.bulletin.list.action

  cn.org.osgi.bulletin.list.pages

  cn.org.osgi.bulletin.po

  cn.org.osgi.bulletin.service

  cn.org.osgi.bulletin.service.impl

  接下来分析这些 package 哪些是需要对外 export 的，根据设计，留言列表模块中没有需要对外 export 的 package。

  在对 package 分析完毕后，来分析有哪些资源文件是此模块需要的，Action 的配置文件、Spring bean 的配置文件、Hibernate.cfg.xml 这些是需要根据 OHSW 脚手架的要求来稍作修改，因此没有需要直接绑定到留言列表模块 jar 文件的资源文件。

  在完成了上述的步骤后，需要根据 OHSW 脚手架的要求来注册留言列表应用、注册 webwork action、注册 PO、引用 Hibernate 模块的 CommonDaoService 实现持久化操作以及对外发布 Action 服务。

  - 注册留言板列表应用

    按照 OHSW 脚手架的要求，注册留言板列表应用需通过 WebworkModule.ControllerExtension 扩展点来实现，在现工程上增加一个 bulletinlist 目录，在该目录下新建 plugin.xml 文件，在其中加上如下内容：

```
<plugin>
<extension
       point="WebworkModule.ControllerExtension">
    <controller webcontext="/bulletin"/>
 </extension>
</plugin>
```

    这样也就完成了留言板列表应用的注册了。

- 注册 webwork action

  根据 OHSW 脚手架的要求，webwork Action 配置文件的编写方法稍有不同，主要是在 action 的 class 这个属性值上，需要改为 Action 服务对应的 command 属性值（如之前 Action 配置文件不是按模块编写的，现在则需改为按模块来编写为独立的 Action 配置文件）：

```
<package name="bulletin-list" extends="webwork-default"
namespace="/bulletin">
      <action name="list" class="LIST">
          <interceptor-ref name="defaultStack" />
          <result name="list"
type="velocity">/cn/org/osgi/bulletin/list/pages/list.vm
</result>
      </action>
  </package>
```

  Action 的注册通过 WebworkModule.ActionExtension 扩展点来实现，打开之前的 plugin.xml，在其中加入如下内容：

```
<extension
       point="WebworkModule.ActionExtension">
    <action
configFile="cn/org/osgi/bulletin/list/action/xwork.xml"/
>
 </extension>
```

- 注册 PO

  根据 OHSW 脚手架的要求，PO 的注册通过 HibernateModule.HibernateExtension 扩展点来实现，打开之前的 plugin.xml，在其中加入如下内容：

```
<extension point="HibernateModule.HibernateExtension">
    <po class="cn.org.osgi.bulletin.po.Bulletin"/>
  </extension>
```

- 引用 Hibernate 模块的 CommonDaoService 实现持久化操作

  持久化操作可引用 Hibernate 模块的 CommonDaoService 来实现，如要改为引用 CommonDaoService，需在现工程的 build path 中增加对于 Hibernate 模块的引用，如之前是直接引用的封装 Hibernate 模块的 jar，现在则需要改为通过 Spring-OSGi 的配置方式来引用 Hibernate 模块提供的 CommonDaoService 的 OSGi 服务：

```
<osgi:reference id="commonDaoService"
interface="cn.org.osgi.module.hibernate.service.CommonDa
oService">
  <osgi:listener ref="bulletinListService"
  bind-method="setCommonDaoService"
  unbind-method="unsetCommonDaoService"/>
 </osgi:reference>
```

同时需要删除 bulletinListService bean 配置中通用 Dao 服务的注入。

这样的改动会引起需要修改 bulletinListService，需要在其中加上 setCommonDaoService 和 unsetCommonDaoService 这两个方法，在改成这种情况下，需要注意 CommonDaoService 可能会随时变为 null，所以要做好防止调用 CommonDaoService 出现 NullPointerException 的处理。

- 对外发布 Action 服务

  Action 服务需要对外发布，由于 Action 的定义之前是在 spring bean 配置，而现在这个 Action 还需要对外提供接口为 cn.org.osgi.xwork.action.IAction 的 OSGi 服务，在此借助 Spring-OSGi 来实现，在 spring bean 的配置文件中加上如下内容：

```
<osgi:service id="BulletinListCommandService"
ref="bulletinListAction"
  interface="cn.org.osgi.xwork.action.IAction"
lazy-init="true">
  <osgi:service-properties>
      <prop key="command">LIST</prop>
  </osgi:service-properties>
 </osgi:service>
```

  这同样会引起 bulletinListAction 的修改，需要在该类上加上对于 IAction 接口的实现。

在完成了上面这些配置文件的编写和代码的部分修改后，就可以开始编写留言列表模块 BND 文件的编写了。

在 BND 文件中需要描述模块的信息，如 Bundle 名、版本、私有的 Package、对外 Export 的 Package、需要包含的资源文件、需要的 Bundle 等等。

按照留言列表模块的设计以及 OHSW 脚手架的要求，最后形成的留言列表模块的 bnd 文件内容如下所示：

```
Bundle-Name: BulletinList Module
```

```
Bundle-SymbolicName: BulletinListModule;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: OSGi.org.cn
# 需要包含至此Bundle的package，但不对外Export
Private-Package:
cn.org.osgi.bulletin.list.*,cn.org.osgi.bulletin.po,cn.org.o
sgi.bulletin.service.*
Require-Bundle:
org.eclipse.equinox.registry,org.eclipse.equinox.common,Webw
orkModule
# 多数需要Import的package bnd都会自动的识别出来，确实有需要的话也可以
# 手工在这里指定
Import-Package: !org.eclipse.core.runtime,org.eclipse.equino
x.http.registry,org.osgi.framework,*
Export-Package:
cn.org.osgi.bulletin.list.action;-noimport:=true,cn.org.osgi
.bulletin.list.pages;-noimport:=true,cn.org.osgi.bulletin.po
;-noimport:=true
```

需要包含的资源文件，plugin.xml 中有扩展点的描述，spring 目录下放置了 spring bean 的配置文件，下面的配置的意思是把 spring 目录下的文件打包到 META-INF/spring 目录下

Include-Resource: plugin.xml=bulletinlist/plugin.xml, META-INF/spring=spring

在编写完上面的 bnd 文件后，就可以在 Eclipse 中选中这个文件，右键，选择其中的 Make Bundle（前提是先安装 bnd 的 eclipse 插件），就可以生成留言列表模块的 jar 文件了。

在生成了这个jar文件后，就可以启动OSHW的脚手架了，启动完毕后在console通过install file:///这样的方式把jar文件安装到脚手架中，安装完毕后通过start的方式启动留言列表Bundle，通过web方式访问即可看到和传统的留言板系统同样的留言列表了。

- 新增留言模块

  新增留言模块和留言列表模块的 bnd 打包过程基本一致，不同的地方在于留言列表模块采用 Spring-OSGi 的方式实现注入和 OSGi 服务的引用、发布，新增留言模块则直接使用 OSGi 的 DS 方式实现，在 bnd 描述文件中可采用以下的方式来描述 OSGi Service 和 Component：

  Service-Component:

cn.org.osgi.bulletin.add.action.NewBulletinAction;properties:="command=NEWBULLETIN";provide:="cn.org.osgi.xwork.action.IAction",cn.org.osgi.bulletin.add.action.SaveBulletinAction;properties:="command=SAVEBULLETIN";service=cn.org.osgi.module.hibernate.service.CommonDaoService;provide:="cn.org.osgi.xwork.action.IAction";dynamic:="service"

根据这样的描述，bnd 在进行打包生成 jar 文件时将会自动生成 ds 的描述 xml 文件。

新增留言模块最终形成的 bnd 文件内容如下所示：

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: NewBulletin Module
Bundle-SymbolicName: NewBulletinModule;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: OSGi.org.cn
Include-Resource: plugin.xml=newbulletin/plugin.xml
Service-Component:
cn.org.osgi.bulletin.add.action.NewBulletinAction;properties
:="command=NEWBULLETIN";provide:="cn.org.osgi.xwork.action.I
Action",cn.org.osgi.bulletin.add.action.SaveBulletinAction;p
roperties:="command=SAVEBULLETIN";service=cn.org.osgi.module
.hibernate.service.CommonDaoService;provide:="cn.org.osgi.xw
ork.action.IAction";dynamic:="service"
Export-Package:
cn.org.osgi.bulletin.add.action;-noimport:=true,cn.org.osgi.
bulletin.add.pages;-noimport:=true
```

选择此 BND 文件，运行 Make Bundle，生成新增留言模块的 jar 文件。

按照留言列表模块同样的方式 install 到 OHSW 脚手架中，此时留言列表显示以及新增留言功能均可正常使用了，效果和传统的基于 Hibernate+Spring+Webwork 的留言板系统完全一致，并且工程和代码结构也基本相同。

经过以上这些步骤，就实现了将传统的基于 Hibernate+Spring+Webwork 的留言板系统部署到基于 OHSW 的脚手架中了。

5.2. 小结

总结整个重构的过程，在重构时需要做的主要是：

- 找出模块所需的 package

- 确定模块对外 Export 的 package 以及 OSGi 服务、需要引用的 OSGi 服务
- 确定模块所需包含的资源文件
- 按照脚手架的要求编写必要的文件，对原有代码进行一定的修改

从章节中示例的留言板系统的重构来看，似乎没碰到多大的阻力，这是因为这个原有的留言板系统在实现时是比较严格的遵守了模块化的，因此在重构时定义出模块的 package、对外 export 的 package 以及 OSGi 服务时都能很容易的完成。

总的来说，要将原有系统重构为可部署至 OSGi 的系统，还是比较的困难和麻烦的，决定性的要素主要是以下三个方面：

- 模块化

  只有在重构进行的时候才会确定以前的系统是否真正的做到了模块化，如果之前的系统做的不够模块化的话，在重构的过程中将会比较的麻烦，会涉及到修改原工程的结构和原代码。

  模块化的部署能够改善原系统的设计，使得每个模块的闭合性和耦合性变得更加的清晰。

  如果想确定原有的系统是否足够的模块化的话，将它重构部署至 OSGi 是个不错的判断方法。

- 保持动态性

  这点呢，是由于部署至 OSGi 框架而带来的，例如之前通过 spring 注入的 bean，现在这些 bean 有可能是位于其他模块中的，这就需要通过以 OSGi 服务的方式进行注入，同时需要注意 OSGi 服务是动态性的，也就是说注入的服务可能随时变得可用，也有可能变得不可用，因此在代码中需要注意这种动态性，避免由于服务的动态性造成系统的错误。

  当然，这步重构不做也是可以，不过这一定程度上就发挥不出部署至 OSGi 的优势了。

  改为使用 OSGi 服务的方式来引用其他模块提供的功能，能够很大程度的提升系统的灵活性，并且一定程度上能够强制面向接口的执行。

- 即插即用

  即插即用是部署至 OSGi 的应用应追求的目标，像上面传统的留言板系统中的留言列表页面上的新增留言链接，就是个典型的非即插即用的例子，因为这个新增留言的链接应该是跟随新增留言模块而产生的，可以想像，当新增留言模块不存

在时，那么新增留言链接存在是没有意义的，基于 OSGi 的应用应保证当新增留言模块安装了后才会相应的出现新增留言的链接等，也就是要切实的保证模块的独立性和即插即用，不能像新增留言链接这种造成模块的强耦合。

而这点呢，也是将原系统重构为 OSGi 应用中最麻烦的一步，这通常来讲会需要重构之前的设计来实现。

只有在上面三点都做到的情况下，才能说原有系统成功的重构为了 OSGi 应用。

6. OSGi 的设计模式

6.1. 树状设计模式

所谓树状设计模式就是以树的方式来设计整个系统，由于 OSGi 在模块的规范化、动态性、扩展性上的良好的支持，使得以树的方式来设计整个系统变得可行。

在设计留言板系统时我们已经展示了如何以树状的方式来进行设计，树状设计模式强调系统像树一样慢慢的长大，在设计时涉及的元素主要是根、枝、叶三要素：

- 根

  整个系统的基础，对于基于 OSGi 框架而构建的系统，根可认为是 Equinox、Felix 等等这些 OSGi 实现框架。

- 枝

  根据系统的组织情况来划分系统的枝，通常来说模块可被划分为枝，模块中用于扩展的部分可划分为枝叶点，以便其他的枝叶能够挂接到此枝上。

- 叶

  叶作为系统功能的终止点，不对外提供扩展的功能。

系统按照根、枝、叶这样的方式来完成设计，依照这样的设计为开发团队人员的分工、系统的耦合、扩展等的考评提供了形象的依据，另外按照这样的设计思想形成的设计潜在的说明了系统的即插即用的特性，当某根枝从树上被折断时，相应的其上的枝和叶也就不再可用了，同样，当在树上增加新的枝叶时，相应的功能也就可以使用了。

这样的一个设计模式和以往的设计方法不同在哪呢？

树状设计模式能够更好的体现和保持 OSGi 系统的模块化、动态性以及可扩展性，来看看在遵照树状设计模式的情况下其系统的模块化、动态性以及可扩展性的体现：

- 模块化的体现

  根据树状设计模式形成的树状系统图，很容易就可以区分出系统的模块，并且还可看出模块的耦合程度等。

- 动态性的体现

  树都是允许折断枝、叶这些的，同时也可以生长出新的枝叶，而这呢都是不用重新从根上开始长起的，系统的动态性得以保证。

- 可扩展性的体现

  枝上提供的枝叶点表明了该模块的可扩展性，无枝叶点的则表明该模块是不可扩

展的。

6.2. 面向服务设计模式

面向服务本身已经是如今最为红火的名词之一，而由于 OSGi 本身最重要的设计思想就是面向服务的组件模型，因此基于 OSGi 而构建的系统自然就应该是面向服务式的。在基于纯粹的 OSGi 构建系统时，Component 之间的通讯应均以 OSGi Service 的方式来进行通讯，在基于 Spring-OSGi 构建系统时，Bundle 内部的 Component 的通讯采用 Spring Bean 的方式进行通讯，推荐这些 Spring bean 以类似 OSGi Service 的方式（即服务接口与实现组件的方式）进行编写，而 Bundle 之间的 Component 的通讯则必须采用 OSGi Service 的方式来实现。

7. OSGi 最佳实践

7.1. 接口和实现分离为不同的 Bundle

当接口和实现均处于同一个 Bundle 时，容易出现的问题就是如果要切换提供的 OSGi 服务的实现的话，就有些麻烦了，只能在调用该 OSGi 服务的组件中指定引用的服务的 target 来实现了，但如果接口和实现分离为不同的 Bundle 的话，对于提供的单一的 OSGi 服务（就是系统中只存在一个对于此 OSGi 服务的实现的组件）的切换就很方便了，对于存在多个实现同一 OSGi 服务的组件的情况也一样，在那种情况下尽管引用该 OSGi 服务的组件基本都会通过 target 过滤来获取自己想要的服务，但如果接口和实现在同一个 Bundle 的话，就必须保证新的实现的 Bundle 中的 OSGi 服务的 target 属性必须采用不同的值，而不能使用同一个值。

举例来理解下上面的场景：

系统中有一个登录校验的服务，在该登录校验服务的 Bundle 中同时存在了一个登录校验服务实现的组件，该组件基于文件的方式实现登录的校验，并对外提供了登录校验服务，此时登录校验 Action 类位于另外的 Bundle 中，它引用了登录校验服务，现在要将登录校验改为使用数据库的方式来进行校验，实现的方法有以下两种：

- 修改目前的登录校验服务组件

  这方法无疑不是很好的选择，如果将来又切换为基于文件的方式来校验呢，那岂不是又得修改这个组件的代码，另外的一个问题就是有些时候原有的代码是不能修改的。

- 新建登录校验服务组件，基于数据库方式实现

  新建一个登录校验服务组件，该组件基于数据库方式来实现，此组件在对外提供 OSGi 服务时增加一个标识为数据库方式的属性值；

  修改登录校验 Action，在其引用 OSGi 服务的部分增加 target 属性指定获取数据库方式的 OSGi 服务。

  可见这种方式也不是好的选择，同样的涉及到了对原有代码的修改。

从上面的例子中可以看出，当 OSGi 服务的接口和实现位于同一 Bundle，对于系统的扩展和灵活性会造成很大的限制。

接下来看看当 OSGi 服务的接口和实现分离为不同的 Bundle 时上面场景的实现方法：

此时只需要新建一个数据库方式校验的登录校验 Bundle，在其中编写一个基于数据

库方式实现登录校验的 OSGi Component，并对外提供登陆校验 OSGi 服务。

部署这个新的 Bundle 到 OSGi 应用中，同时停止原文件方式登录校验的 Bundle。

通过上面的方法就实现了登录校验服务的切换，可见当 OSGi 服务的接口和实现分离为不同的 Bundle 的时候，系统的灵活性和扩展性都得到了保证，因此其被列入 OSGi 的最佳实践之一。

7.2. 保持系统动态性

动态性是 OSGi 系统中应该具备的，动态性的原则有两点，其实就是“即插即用、即删即无”原则：

- 不因为 Bundle 的卸载或不可用导致应用出现崩溃性的错误或无意义的入口

  对于引用了的 OSGi 服务的 Component，应注意处理当服务不可用时的情况，同时应保证当 bundle 卸载时，其相应的功能入口（如网页上的链接、按钮或应用程序上的菜单等）也应相应的删除，避免出现无意义甚至会导致错误的功能入口。

  就像例子中的留言板系统，在基于 OHSW 脚手架中的留言板系统当卸载新增留言模块时，相应的其功能入口也被卸载，而同样的如果卸载掉 Hibernate 封装模块，并不会造成留言列表模块的崩溃性错误，代替的是友好的错误提示。

  而在重构原有留言板系统为 OSGi 应用时，即使卸载了新增留言的模块，留言列表的页面上新增留言的功能入口仍然存在，这就没有很好的保证系统的动态性。

- 当有新的 Bundle 安装时相应的功能入口或挂接应自动的完成

  当新的 Bundle 安装到 OSGi 应用时，相应的其 OSGi 的服务等功能应立即生效，例如其功能的入口应自动的挂接到相应的地方等。

以上两点原则是在编写 OSGi Component 时应特别注意的，否则使用 OSGi 带来的好处也就大幅度的降低了，按照树状设计模式而言，就是当叶被折断时，树上就没有了这叶了，如果是枝被折断时，枝上的其他的枝和叶也相应的不在树上了，当树上长出了新的枝叶的时候，那么树也就自然的变的茂盛了，系统的功能也应变得更为强大。保持系统的动态性是基于 OSGi 的初级实践者们很容易忽略的部分，因为原来实现系统的习惯都是静态化的实现，而实现的思维则要转化为实现的是动态化的系统，这需要一定的适应过程。

要实现“即插即用、即删即无”，推荐采用的实践方法是：

- 不强依赖其他 Bundle 的资源

  意思是不在当前的 Bundle 中显示的直接使用其他 Bundle 的资源，如不在留言列

表页面上显示的编写新增留言的功能链接等，这个可通过扩展点的方式来实现，避免出现强依赖其他 Bundle 资源或功能的现象。

- 不强依赖任何 OSGi 服务

  意思是不假设在当前 Bundle 的 Component 激活后，其中所引用的 OSGi 服务一直可用，这个可在代码中调用服务前先判断服务是否为 null 等方法来实现。

  对于某些一定需要有依赖的 OSGi 服务存在的 Component，则可在 component 的描述文件中通过 cardinality 属性来控制，这样可以保证当必须依赖的 OSGi 均不存在时，component 进入 deactive 状态，如；对于不是必须依赖的 OSGi 服务，则可将其 cardinality 设置为'0..1'或'0..n'这样的方式，对于引用的这种 OSGi 服务，就必须在调用其前判断下此 OSGi 服务是否可用，如 Component 中对于 LogService 的引用。

- 监听系统的动态的变化

  意思是应动态的去监听可能会动态变化的部分，在具体的实现时可采用监听器、Event、CM 以及 OSGi Declarative Services 的 bind、unbind 方法来实现，这样在当系统发生动态的变化时，Component 或其他类都可以做出相应的动作，保持系统可动态的做出响应。

7.3. 搭建公司级的 Bundle Resposiotory

使用 OSGi 一个很大的意义就是它很大程度的提升了系统的可重用性，由于 OSGi 提供了规范的模块化和规范的模块交互机制，并具备了充分的语义，这样的话使得模块化级别的重用甚至是系统级的重用成为了可能。

OSGi 下一版本规范中很可能会纳入 Bundle Respository，Bundle Respository 和 Maven Respository 功能基本是一样的，只是 Bundle Respository 存放的是 Bundle，Bundle 通过标准的描述发布至 Bundle Respository 中，使得其他需要使用 Bundle 的同仁们可通过 Bundle Respository 查找相应功能的 Bundle，目前已有的 Bundle Respository 有：

http://www2.osgi.org/Respository/HomePage

关于此 Respository 的使用，Peter Kriens 提供了一个很经典的录屏：

http://www.aqute.biz/Blog/20070703

建立公司级的Bundle Respository对于公司而言具备很大的意义，建立起Bundle Respository后，公司在进行每个项目之前都可通过Bundle Respository来寻找是否有相似功能的Bundle，这样就可以快速的搭建起项目的脚手架了，这对于公司项目的积累

和共享是非常有帮助的，而这也是在使用OSGi后能给公司带来的最明显的帮助，同时还可借助互联网上的各种公开的Bundle Respository，鼓励大家贡献Bundle给Bundle Respository[7]，增强业界的分享，提升系统的开发速度，减少重复劳动。

7.4. 创建共享 library Bundle

在一个项目中，每个模块都会需要用到一些共同的第三方的 jar 包，按照正常的方式去搭建 Bundle 的话，需要在每个 Bundle 的 lib 中都放入其所需要的 jar 包文件，这样会造成的问题有：

- 所依赖的第三方库重复出现，造成最后的 Bundle 文件过大；
- 各 Bundle 维护各自的第三方库，有可能会造成引用的第三方库版本不同；

解决这个问题的最佳实践的方法是：

- 创建一个共享的第三方库的 Bundle；
- 将需要共享的第三方库的 jar 文件放入此 Bundle；
- 将其他 Bundle 需要的第三方库的 jar 的 package 对外 Export；
- 需要使用第三方库的 Bundle 通过 import package 的方式引用所需的 package。

7.5. 最小化依赖（Minimize Dependencies）

这个最佳实践来源于Peter和BJ的OSGi best practices文档[8]。

最小化依赖的目的是要尽量的减少 Bundle 之间的依赖，避免出现安装一个 Bundle 时要安装 N 多其依赖的 Bundle 的现象。

为实现最小化依赖，Peter 和 BJ 提出了以下几点最佳的实践：

- 使用 Import-Package 代替 Require-Bundle

  只有在需要使用其他 Bundle 提供的资源文件时才采用 Require-Bundle 的方式，而在其他情况下应该都使用 Import-Package 去引用其他 Bundle 提供的 package，从而使用其他 Bundle 的功能。

- 使用版本范围控制

  在引用的 package 中尽量指定版本范围，以减少运行时 OSGi 过多的判断。

- 设计 Bundle

  在设计 Bundle 应遵循低耦合，高内聚的原则，不要把和 Bundle 不相关的功能放入到 Bundle 中。

[7] 关于如何贡献Bundle给OSGi.org的Respository，可参见此篇blog：
http://www.osgi.org/blog/2007/07/osgi-bundle-repository-indexer-open.html
[8] 此篇文档可从http://www2.osgi.org/wiki/uploads/Conference/OSGiBestPractices.pdf下载。

7.6. 避免启动顺序依赖（Avoid Start Ordering Dependencies）

这个最佳实践来源于 Peter 和 BJ 的 OSGi best practices 文档。

启动顺序依赖是指系统中 bundle 的启动依赖于其他 bundle 的启动，也就是说如依赖的 bundle 未启动时，启动该 bundle 的话会造成错误，这是 OSGi 初学者很容易犯的错误。

在任何 Bundle 的启动中都不应该对别的 Bundle 造成强依赖，应该遵循动态性原则，尤其是引用了其他 Bundle 提供的服务的时候，例如下面的例子：

A Bundle 的 Activator 中的 activate 方法是这么写的：

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

像上面这样的写法就使得提供 LoginService 实现的 Bundle 必须在 A Bundle 之前启动，否则在启动时就会造成 NullPointerException 错误。

启动顺序依赖除了会造成上面的问题，还有可能造成的问题就是启动时间过长，因为启动顺序依赖也就意味着在启动的时候就要做很多的工作了。

Peter 和 BJ 提出了以下的一些最佳实践来避免启动顺序依赖的问题：

- 不要在初始化 Bundle 时引用 OSGi 服务；
- 使用 ServiceTracker 去动态的获取所需引用的 OSGi 服务的状态；
- 使用 DS 或 Spring-OSGi 自动的获取 OSGi 服务的状态。