## 关于本教程

OSGi 是目前动态模块系统的事实上的工业标准，虽然一开始只是作为嵌入式设备和家庭网关的框架来使用，但是实际上它适用于任何需要模块化、面向服务、面向组件的应用程序。而 Equinox 则是的 Eclipse 所使用的 OSGi 框架，是 Eclipse 强大的插件体系的基础，Eclipse 的稳定可靠性也为该框架带来了声誉。

本教程就将演示如何在 Eclipse 环境下利用 Equinox 框架进行 OSGi 应用开发。首先解释了实现上述应用程序所必需了解的基本概念和基础知识，并结合示例代码演示 OSGi 开发的一些重要技术，最后探讨了基于 OSGi 应用程序一般所采用的架构，以及如何将 Equinox OSGi 应用程序脱离 Eclipse 而部署为一个标准的 Java 应用程序。

### 目标

在本教程中，您将学习：

* OSGi 及框架简介
* 编写第一个 OSGi 应用程序
* 重要的理论知识
* 开发一个真实的 OSGi 应用程序
* 探讨 OSGi 应用架构
* 部署 OSGi 应用程序

### 先决条件

本教程假设读者熟悉基本 Java 语言以及 Eclipse 开发环境的使用。

### 系统需求

本教程假设您有一个可以工作的 Eclipse 3.x 环境。如果还没有，请在 [Eclipse 网站](http://www.eclipse.org/) 上找到相关下载的链接，以帮助您在自己的系统上操作示例步骤以及运行示例代码。

#### OSGi 及框架简介

OSGi 简介

OSGi 是目前动态模块系统的事实上的工业标准，虽然一开始只是作为嵌入式设备和家庭网关的框架来使用，但是实际上它适用于任何需要模块化、面向服务、面向组件的应用程序。

目前 OSGi 规范已经发展到第四版（R4）, 由 OSGi 联合组织（OSGi Alliance）负责进行维护管理，相关的规范资料也可以从该网站获得。（[参考资料](http://www.ibm.com/developerworks/cn/education/opensource/os-eclipse-osgi/resources.html)）

#### OSGi 框架

开发基于 OSGi 的应用程序离不开实现了 OSGi 标准的框架，就好比是基于 J2EE 的开发离不开应用服务器一样。目前比较流行的基于 OSGi R4 标准实现的 OSGi 框架有三个：

1. **Equinox**：这是大名鼎鼎的 Eclipse 所使用的 OSGi 框架，Eclipse 强大的插件体系就是构建在 OSGi bundles 的基础之上，Eclipse 的稳定可靠性为该框架带来了声誉，而且由于有 IBM 公司的强力支持，其后续的开发和文档资料也有了一定的保障。一般情况下，我们推荐您使用该框架进行 OSGi 开发。本教程的后续部分也将演示如何使用 Equinox 框架来进行 OSGi 应用程序的开发。
2. **Makewave Knopflerfish**：这是另外一个比较知名的 OSGi 框架，目前的版本已经支持 R4 规范，其特点在于为应用程序的开发提供了大量的 bundle 。
3. **Apache Flex**：由 Apache 基金组织开发的面向社区的 OSGi 框架实现，提供了标准的服务和一些有趣的和 OSGi 相关的服务实现。

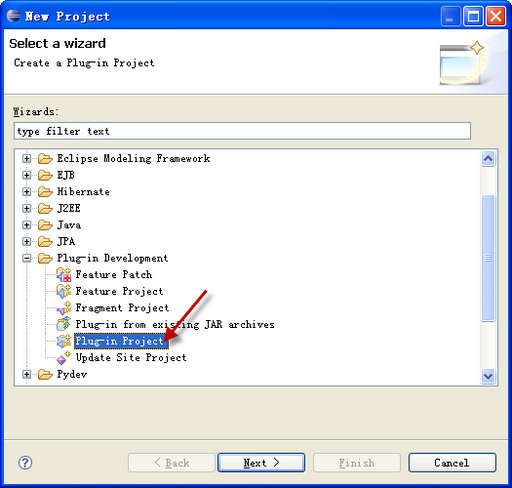
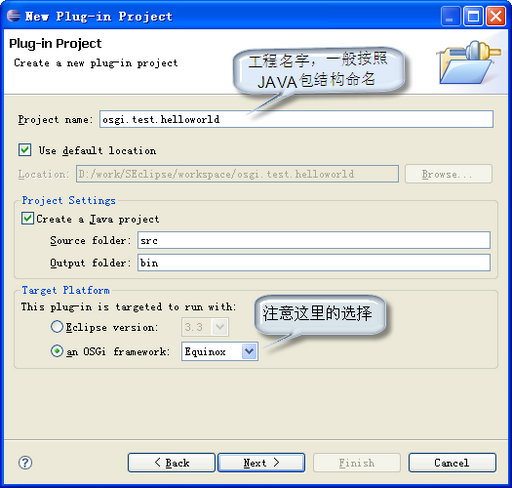
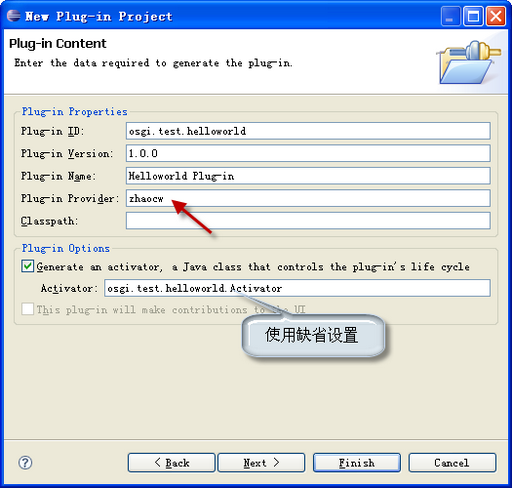
Hello World！编写第一个 OSGi 应用程序

#### 准备工作

1. 从附属资料中下载 Eclipse 3.x 版本，Eclipse 3.2+ 版本已经全面支持 OSGi R4 规范。目前最佳实践是下载 Eclipse 3.3.2 版本。（下载请见：[参考资料](http://www.ibm.com/developerworks/cn/education/opensource/os-eclipse-osgi/resources.html)）
2. 将 Eclipse 解压缩到 d:\work\seclipse 目录，开始我们的 OSGi 之旅。

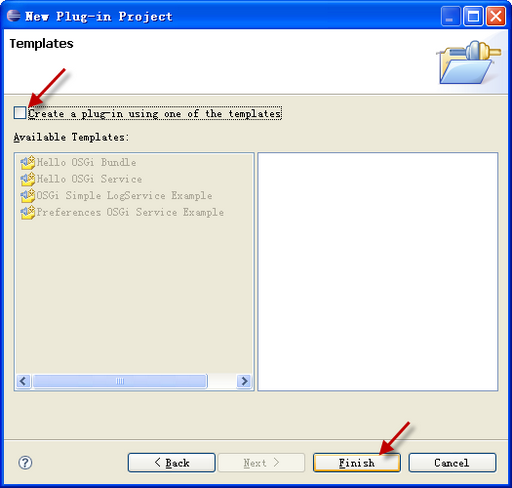
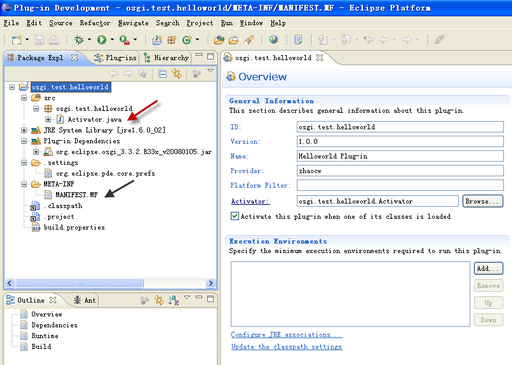
### Hello World

一般情况下，学习一门新的技术，程序员都习惯于首先开发一个 hello world 应用程序，这似乎也是一种“工业标准”。好的，让我们开始吧，开发一个简单的 OSGi 应用程序并不难，步骤如下：

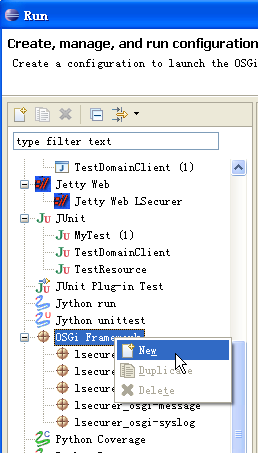
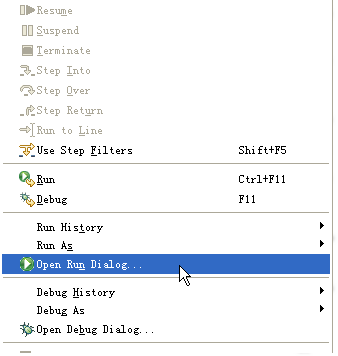
1. 建立一个 plug-in 工程，**File > New > Project**，选择 **Plug-in development > Plug-in Project**   
     
   **图 1. 新建 plug-in 工程**  
   
2. 在建立工程的第一个向导，填入工程的名称：osgi.test.helloworld，使用缺省的工程路径。注意目标平台的选择，由于我们的项目是一个通用的 OSGi bundle，所以选择 **equinox** 。   
     
   **图 2. 填入工程名及选择目标平台**  
   
3. 在下一个向导界面中，填入需要的一些插件信息（注意 Eclipse 中的插件概念基本类似于 OSGi 中的 bundle 的概念），这里需要填入的是 OSGi 的 provider（供应商）和 classpath 。如果没有特别的设计，一般可以忽略这两个字段 。最后是关于 activator 的部分，如果不是一个 fragment bundle 则需要填入，除非您的 bundle 自己实现框架的事件监听，这个似乎也没有必要。因此，建议使用缺省的设置，如图 3：  
     
     
   **图 3. 使用缺省设置**  
   

**Activator：**这是 bundle 启动时首先调用的程序入口，相当于 Java 模块中的 main 函数。不同的是，main 需要通过命令行调用，而 OSGi 的 Activator 是被动的接受 OSGi 框架的调用，收到消息后才开始启动。

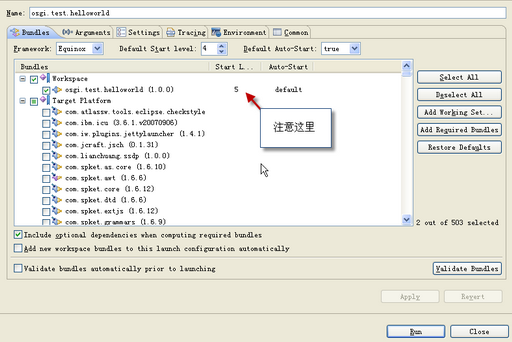
最佳实践：不要在 Activator 中写太多的启动代码，否则会影响 bundle 启动速度，相关的服务启动可以放到服务的监听器中。

1. 最后一步，不使用任何的模板，所以勾掉缺省的选项，点击完成，如图 4：   
     
   **图 4. 勾掉缺省的选项**  
   
2. 完成，基本的插件视图如图 5，Eclipse 会在工程名下建立相同路径的 Java Package，其中包含了 Activator 类，插件的配置信息也都放在 MANIFEST.MF 文件中，将来我们相当多的工作都是在其中完成。   
     
   **图 5. 基本的插件视图**  
   
3. 编辑 Activator.java，输入 hello world 语句，代码如下：   
     
   **清单 1. 编辑 Activator.java**

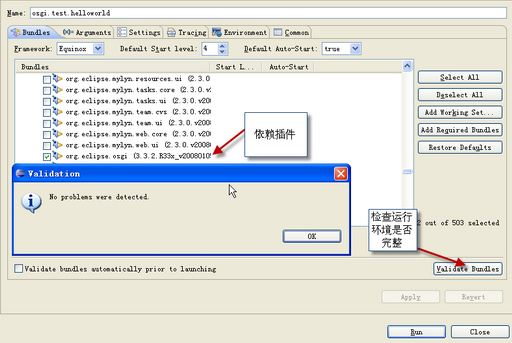
|  |
| --- |
| package osgi.test.helloworld;  import org.osgi.framework.BundleActivator;  import org.osgi.framework.BundleContext;  public class Activator implements BundleActivator {  /\*  \* (non-Javadoc)  \* @see org.osgi.framework.BundleActivator  \* #start(org.osgi.framework.BundleContext)  \*/  public void start(BundleContext context) throws Exception {  **System.out.println("hello world");**  }  /\*  \* (non-Javadoc)  \* @see org.osgi.framework.BundleActivator  \* #stop(org.osgi.framework.BundleContext)  \*/  public void stop(BundleContext context) throws Exception {  }  } |

1. 我们可以看到每个 Activator 实际都是实现了 BundleActivator 接口，此接口使 Activator 能够接受框架的调用。在框架启动后，启动每个 bundle 的时候都会调用每个 bundle 的 Activator 。
2. 注意：bundle 的 Activator 必须含有无参数构造函数，这样框架才能使用 Class.newInstance() 方式反射构造 bundle 的 Activator 实例。
3. 这里我们在 start 方法中填入了我们希望输出的 hello world 字符串。那么，怎么才能启动这个 bundle 呢？
4. 执行：选择 **Run > Open Run Dialog**，进入运行菜单，在 OSGi framework 中右键点击选择 **new** 一个新的 OSGi 运行环境，如图：   
     
   **图 6. 新建 OSGi 运行环境**  
   

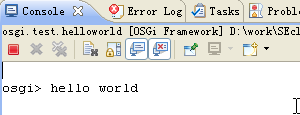
在右边的运行环境对话框中，输入运行环境的名字、start level 和依赖的插件，由于我们目前不需要其它的第三方插件，因此只需要勾上系统的 org.eclipse.osgi 插件，如果不选择此插件，hello world 将无法运行。如图 7，只有当您点击了 **validate bundles** 按钮 ，并且提示无问题之后，才表明您的运行环境基本 OK 了。

**图 7. 选择 org.eclipse.osgi插件**  


依赖插件的选择：

**图 8. 依赖插件的选择**  


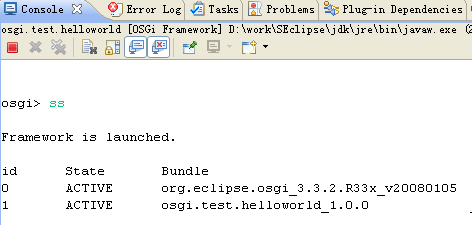
好的，如果您的运行环境已经 OK，那么就点击 **Run** 吧。

**图 9. 运行 OSGi 项目**  


恭喜您，成功了！

### OSGi 控制台

OSGi 控制台对于习惯开发普通 Java 应用程序的开发人员来说，还是比较新鲜的。一般来说，通过 OSGi 控制台，您可以对系统中所有的 bundle 进行生命周期的管理，另外也可以查看系统环境，启动、停止整个框架，设置启动级别等等操作。如图 10，键入 SS 就可以查看所有 bundle 的状态：

**图 10. 查看所有 bundle 的状态**  


下面列出了主要的控制台命令：

**表 1. Equinox OSGi 主要的控制台命令表**

|  |  |  |
| --- | --- | --- |
| **类别** | **命令** | **含义** |
| 控制框架 | launch | 启动框架 |
| shutdown | 停止框架 |
| close | 关闭、退出框架 |
| exit | 立即退出，相当于 System.exit |
| init | 卸载所有 bundle（前提是已经 shutdown） |
| setprop | 设置属性，在运行时进行 |
| 控制 bundle | Install | 安装 |
| uninstall | 卸载 |
| Start | 启动 |
| Stop | 停止 |
| Refresh | 刷新 |
| Update | 更新 |
| 展示状态 | Status | 展示安装的 bundle 和注册的服务 |
| Ss | 展示所有 bundle 的简单状态 |
| Services | 展示注册服务的详细信息 |
| Packages | 展示导入、导出包的状态 |
| Bundles | 展示所有已经安装的 bundles 的状态 |
| Headers | 展示 bundles 的头信息，即 MANIFEST.MF 中的内容 |
| Log | 展示 LOG 入口信息 |
| 其它 | Exec | 在另外一个进程中执行一个命令（阻塞状态） |
| Fork | 和 EXEC 不同的是不会引起阻塞 |
| Gc | 促使垃圾回收 |
| Getprop | 得到属性，或者某个属性 |
| 控制启动级别 | Sl | 得到某个 bundle 或者整个框架的 start level 信息 |
| Setfwsl | 设置框架的 start level |
| Setbsl | 设置 bundle 的 start level |
| setibsl | 设置初始化 bundle 的 start level |

### MANIFEST.MF

MANIFEST.MF 可能出现在任何包括主类信息的 Jar 包中，一般位于 META-INF 目录中，所以此文件并不是一个 OSGi 特有的东西，而仅仅是增加了一些属性，这样也正好保持了 OSGi 环境和普通 Java 环境的一致性，便于在老的系统中部署。表 2 列出此文件中的重要属性及其含义：

**表 2. MANIFEST.MF 文件属性**

|  |  |
| --- | --- |
| **属性名字** | **含义** |
| Bundle-Activator | Bundle 的启动器 |
| Bundle-SymbolicName | 名称，一般使用类似于 JAVA 包路径的名字命名 |
| Bundle-Version | 版本，注意不同版本的同名 bundle 可以同时上线部署 |
| Export-Package | 导出的 package 声明，其它的 bundle 可以直接引用 |
| Import-Package | 导入的 package |
| Eclipse-LazyStart | 是否只有当被引用了才启动 |
| Require-Bundle | 全依赖的 bundle，不推荐 |
| Bundle-ClassPath | 本 bundle 的 class path，可以包含其它一些资源路径 |
| Bundle-RequiredExecutionEnvironment | 本 bundle 必须的执行环境，例如 jdk 版本声明 |

重要的理论知识

好的，刚才我们已经从头到尾开发了一个基于 Equinox 框架的 Hello world 应用程序。我们发现似乎并不是很困难，很多工作 Eclipse 已经帮我们做好了，例如 Activator 代码框架和 MANIFEST.MF 文件，我们也学会了如何控制 OSGi 的控制台和编写 MANIFEST.MF 文件，但是，您真的明白它们是如何运行的么？下面我们将重点介绍一些 OSGi 运行必备的基础知识。

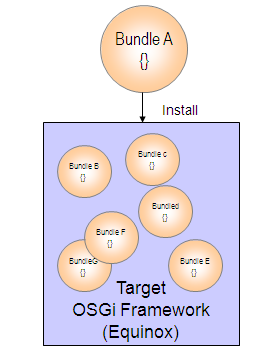
什么是 bundle？

我们已经看到，编写一个很普通的 Hello world 应用，必须首先创建一个 plug-in 工程，然后编辑其 Activator 类的 start 方法，实际我们这样做的本质是为 OSGi 运行环境添加了一个 bundle，那么一个 bundle 必须的构成元素是哪些呢？

1. MANIFEST.MF：描述了 bundle 的所有特征，包括名字、输出的类或者包，导入的类或者包，版本号等等，具体可以参考 表 2. MANIFEST.MF 文件属性。
2. 代码：包括 Activator 类和其它一些接口以及实现，这个和普通的 Java 应用程序没有什么特殊的区别。
3. 资源：当然，一个应用程序不可能没有资源文件，比如图片、properties 文件、XML 文件等等，这些资源可以随 bundle 一起存在，也可以以 fragment bundle 的方式加入。
4. 启动级别的定义：可以在启动前使用命令行参数指定，也可以在运行中指定，具体的 start level 的解释，请参考 后面的说明。

#### 框架做了些什么？

好了，我们已经明白 bundle 是什么了，也知道如何开发一个基本的 bundle 了，那么我们还必须要明白，我的 bundle 放在 Equinox 框架中，它对我们的 bundle 做了些什么？

**图 11. Equinox 框架架构**  


实际上，目标平台已经为我们准备了 N 个 bundle，它们提供各种各样的服务，OSGi 中，这些 bundle 的名字叫 system bundle，就好比精装修的房子，您只需要拎包入住，不再需要自己铺地板，装吊顶了。

我们的 bundle 进入 Equinox 环境后，OSGi 框架对其做的事情如下：

1. 读入 bundle 的 headers 信息，即 MANIFEST.MF 文件；
2. 装载相关的类和资源；
3. 解析依赖的包；
4. 调用其 Activator 的 start 方法，启动它；
5. 为其提供框架事件、服务事件等服务；
6. 调用其 Activator 的 stop 方法，停止它；

#### Bundle 的状态变更

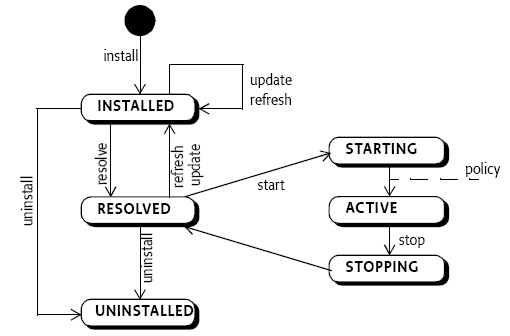
OK, 现在我们大概明白了一个 bundle 的定义和其在 OSGi 框架中的生命周期，前面我们看到控制台可以通过 ss 命令查看所有装载的 bundle 的状态，那么 bundle 到底具有哪些状态，这些状态之间是如何变换呢？我们知道了这些状态信息，对我们有何益处？

首先，了解一下一个 bundle 到底有哪些状态：

**表 3. Bundle 状态表**

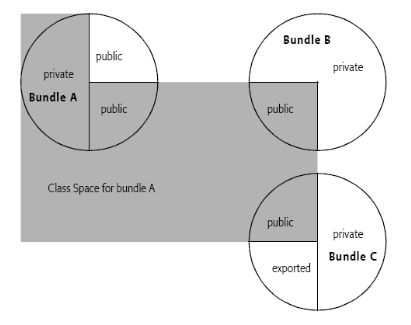
|  |  |
| --- | --- |
| **状态名字** | **含义** |
| INSTALLED | 就是字面意思，表示这个 bundle 已经被成功的安装了 |
|  |  |
| RESOLVED | 很常见的一个状态，表示这个 bundle 已经成功的被解析（即所有依赖的类、资源都找到了），通常出现在启动前或者停止后 |
| STARTING | 字面意思，正在启动，但是还没有返回，所以您的 Activator 不要搞的太复杂 |
| ACTIVE | 活动的，这是我们最希望看到的状态，通常表示这个 bundle 已经启动成功，但是不意味着您的 bundle 提供的服务也是 OK 的 |
| STOPPING | 字面意思，正在停止，还没有返回 |
| UNINSTALLED | 卸载了，状态不能再发生变更了 |

下面请看一张经典的 OSGi bundle 变更状态的图：

**图 12. OSGi bundle 变更状态图**  


Bundle 导入导出 package

OK，到现在为止，似乎一切都是新鲜的，但是您似乎在考虑，OSGi 到底有什么优势，下面介绍一下其中的一个特点，几乎所有的面向组件的框架都需要这一点来实现其目的： ***面向服务、封装实现*** 。这一点在普通的 Java 应用是很难做到的，所有的类都暴露在 classpath 中，人们可以随意的查看您的实现，甚至变更您的实现。这一点，对于希望发布组件的公司来说是致命的。

**图 13. OSGi bundle 原理**  


OSGi 很好的解决了这个问题，就像上面的图显示的，每个 bundle 都可以有自己公共的部分和隐藏的部分，每个 bundle 也只能看见自己的公共部分、隐藏部分和其它 bundle 的公共部分。

bundle 的 MANIFEST.MF 文件提供了 EXPORT/IMPORT package 的关键字，这样您可以仅仅 export 出您希望别人看到的包，而隐藏实现的包。并且您可以为它们编上版本号，这样可以同时发布不同版本的包。

Bundle class path

这一点比较难理解，一般情况下您不需要关心这个事情，除非事情出现了问题，您发现明明这个类就在这里，怎么就是报告 ClassNotFoundException/NoClassDefExcpetion 呢？在您垂头丧气、准备砸掉电脑显示器之前，请看一下 bundle 中的类是如何查找的：

1. 首先，它会找 JRE，这个很明显，这个实际是通过系统环境的 JAVA\_HOME 中找到的，路径一般是 JAVA\_HOME/lib/rt.jar、tools.jar 和 ext 目录，endorsed 目录。
2. 其次，它会找 system bundle 导出的包。
3. 然后，它会找您的 import 的包，这个实际包含两种：一种是直接通过 require-bundle 的方式全部导入的，还有一种就是前面讲的通过 import package 方式导入的包。
4. 查找它的 fragment bundle，如果有的话。
5. 如果还没有找到，则会找自己的 classpath 路径（每个 bundle 都有自己的类路径）。
6. 最后它会尝试根据 DynamicImport-Package 属性查找的引用。

启动级别 Start level

在 Equinox 环境中，我们在配置 hello world 应用的时候，看到我们将 framework start level 保持为 4，将 Hello world bundle 的 start level 设置为 5 。 start level 越大，表示启动的顺序越靠后。在实际的应用环境中，我们的 bundle 互相有一定的依赖关系，所以在启动的顺序上要有所区别，好比盖楼，要从打地基开始。

实际上，OSGi 框架最初的 start level 是 0，启动顺序如下：

1. 将启动级别加一，如果发现有匹配的 bundle（即 bundle 的启动级别和目前的启动级别相等），则启动这个 bundle；
2. 继续第一步，直到发现已经启动了所有的 bundle，且活动启动级别和最后的启动的 bundle 启动级别相同。

停止顺序，也是首先将系统的 start level 设置为 0：

1. 由于系统当前活动启动级别大于请求的 start level，所以系统首先停止等于当前活动启动级别的 bundle；
2. 将活动启动级别减一，继续第一步，直到发现活动启动级别和请求级别相等，都是 0。

开发一个真实的 OSGi 应用程序

我们不能只停留在 hello world 的层面，虽然那曾经对我们很重要 ，但是现实需要我们能够使用 OSGi 写出激动人心的应用程序，它能够被客户接受，被架构师认可，被程序员肯定。好的，那我们开始吧。下面将会着重介绍一些现实的应用程序可能需要的一些 OSGi 应用场景。

发布和使用服务

由于 OSGi 框架能够方便的隐藏实现类，所以对外提供接口是很自然的事情，OSGi 框架提供了服务的注册和查询功能。好的，那么我们实际操作一下，就在 Hello world 工程的基础上进行。

我们需要进行下列的步骤：

1. 定义一个服务接口，并且 export 出去供其它 bundle 使用；
2. 定义一个缺省的服务实现，并且隐藏它的实现；
3. Bundle 启动后，需要将服务注册到 Equinox 框架；
4. 从框架查询这个服务，并且测试可用性。

好的，为了达到上述要求，我们实际操作如下：

1. 定义一个新的包 osgi.test.helloworld.service ，用来存放接口。单独一个 package 的好处是，您可以仅仅 export 这个 package 给其它 bundle 而隐藏所有的实现类
2. 在上述的包中新建接口 IHello，提供一个简单的字符串服务，代码如下：   
     
   **清单 2. IHello**

|  |
| --- |
| package osgi.test.helloworld.service;  public interface IHello {  /\*\*  \* 得到 hello 信息的接口 .  \* @return the hello string.  \*/  String getHello();  } |

1. 再新建一个新的包 osgi.test.helloworld.impl，用来存放实现类。
2. 在上述包中新建 DefaultHelloServiceImpl 类，实现上述接口：   
     
   **清单 3. IHello 接口实现**

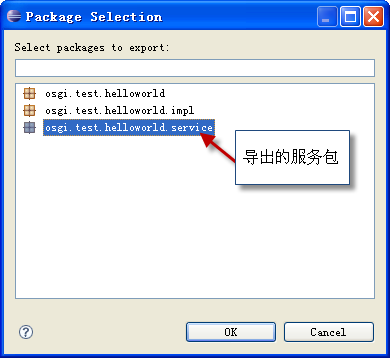
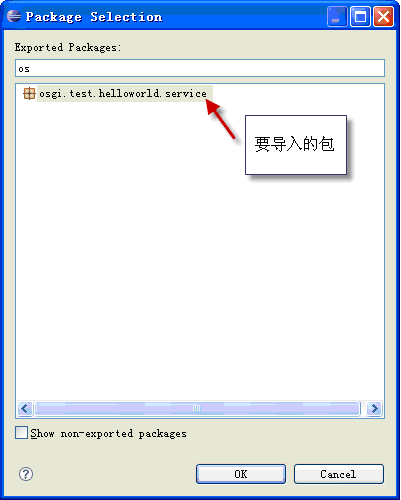
|  |
| --- |
| public class DefaultHelloServiceImpl implements IHello {  @Override  public String getHello() {  return "Hello osgi,service";  }  } |

1. 注册服务，OSGi 框架提供了两种注册方式，都是通过 BundleContext 类实现的：
   1. registerService(String,Object,Dictionary) 注册服务对象 object 到接口名 String 下，可以携带一个属性字典 Dictionary；
   2. registerService(String[],Object,Dictionary) 注册服务对象 object 到接口名数组 String[] 下，可以携带一个属性字典 Dictionary，即一个服务对象可以按照多个接口名字注册，因为类可以实现多个接口；

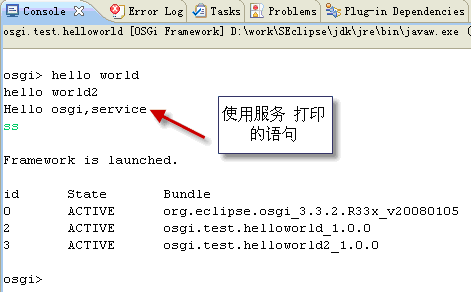
我们使用第一种注册方式，修改 Activator 类的 start 方法，加入注册代码：

**清单 4. 加入注册代码**

|  |
| --- |
| public void start(BundleContext context) throws Exception {    System.out.println("hello world");  context.registerService(  IHello.class.getName(),  new DefaultHelloServiceImpl(),  null);    } |

1. 为了让我们的服务能够被其它 bundle 使用，必须在 MANIFEST.MF 中对其进行导出声明，双击 MANIFEST.MF，找到 **runtime > exported packages > 点击 add**，如图，选择 **service** 包即可：   
     
   **图 14. 选择导出的服务包**  
   
2. 另外新建一个类似于 hello world 的 bundle 叫：osgi.test.helloworld2，用于测试 osgi.test.helloworld bundle 提供的服务的可用性；
3. 添加 import package：在第二个 bundle 的 MANIFEST.MF 文件中，找到 **dependencies > Imported packages > Add …**，选择我们刚才 export 出去的 **osgi.test.helloworld.service** 包：   
     
   **图 15. 选择刚才 export 出去的 osgi.test.helloworld.service 包**  
   
4. 查询服务：同样，OSGi 框架提供了两种查询服务的引用 ServiceReference 的方法：
   1. getServiceReference(String)：根据接口的名字得到服务的引用；
   2. getServiceReferences(String,String)：根据接口名和另外一个过滤器名字对应的过滤器得到服务的引用；
5. 这里我们使用第一种查询的方法，在 osgi.test.helloworld2 bundle 的 Activator 的 start 方法加入查询和测试语句：   
     
   **清单 5. 加入查询和测试语句**

|  |
| --- |
| public void start(BundleContext context) throws Exception {  System.out.println("hello world2");    /\*\*  \* Test hello service from bundle1.  \*/  IHello hello1 =  (IHello) context.getService(  context.getServiceReference(IHello.class.getName()));  System.out.println(hello1.getHello());  } |

1. 修改运行环境，因为我们增加了一个 bundle，所以说也需要在运行配置中加入对新的 bundle 的配置信息，如下图所示：   
     
   **图 16. 加入对新的 bundle 的配置信息**  
   图 16. 加入对新的 bundle 的配置信息
2. 执行，得到下列结果：   
     
   **图 17. 执行结果**  
   

恭喜您，成功了！

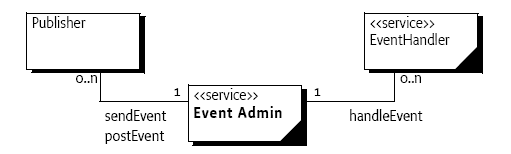
使用事件管理服务 EventAdmin

前面讲过，OSGi 规范定义了很多可用的 bundle，您尽管使用它们完成您的工作，而不必另外再发明轮子，OSGi 框架定义的事件管理服务，类似于 JMS，但是使用上比 JMS 简单。

OSGi 整个框架都离不开这个服务 ，因为框架里面全都依靠事件机制进行通信，例如 bundle 的启动、停止，框架的启动、停止，服务的注册、注销等等等等都是会发布事件给监听者，同时也在监听其它模块发来的自己关心的事件。 OSGi 框架的事件机制主要核心思想是：

1. 用户（程序员）可以自己按照接口定义自己的事件类型
2. 用户可以监听自己关心的事件或者所有事件
3. 用户可以将事件同步的或者异步的提交给框架，由框架负责同步的或者异步的分发给监听者

说明：框架提供的事件服务、事件提供者、事件监听者之间的关系如下：

**图 18. 事件服务、事件提供者、事件监听者之间的关系**  


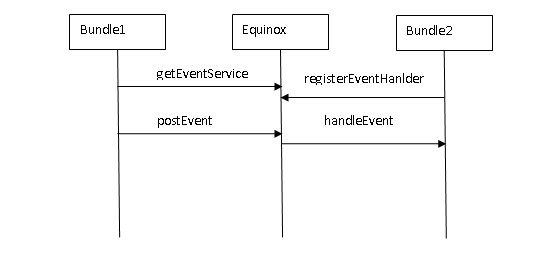
事件提供者 Publisher 可以获取 EventAdmin 服务，通过 sendEvent 同步（postEvent 异步）方式提交事件，EventAdmin 服务负责分发给相关的监听者 EventHandler，调用它们的 handleEvent 方法。

这里要介绍一个新的概念 Topics，其实在 JMS 里面也有用，也就是说一个事件一般都有一个主题，这样我们的事件接收者才能按照一定的主题进行过滤处理，例如只处理自己关心的主题的事件，一般情况下主题是用类似于 Java Package 的命名方式命名的。

同步提交（sendEvent）和异步提交（postEvent） 事件的区别是，同步事件提交后，等框架分发事件给所有事件接收者之后才返回给事件提交者，而异步事件则一经提交就返回了，分发在另外的线程进行处理。

下面的程序演示了事件的定义、事件的发布、事件处理，同时还演示了同步和异步处理的效果，以及运行环境的配置。

（约定 osgi.test.helloworld 为 bundle1，osgi.test.helloworld2 为 bundle2）

**图 19. 同步和异步处理演示**  


1. 在 bundle1 中的 MANIFEST.MF 的 dependency 页面中定义引入新的包：org.osgi.service.event。
2. 在 bundle1 中的 osgi.test.helloworld.event 包中定义新的类 MyEvent，如下（注意其中的 topic 定义的命名方式）：   
     
   **清单 6. 定义新的类 MyEvent**

|  |
| --- |
| import java.util.Dictionary;  import org.osgi.service.event.Event;  public class MyEvent extends Event {  public static final String MY\_TOPIC = "osgi/test/helloworld/MyEvent";  public MyEvent(String arg0, Dictionary arg1) {  super(MY\_TOPIC, arg1);  }  public MyEvent() {  super(MY\_TOPIC, null);  }  public String toString() {  return "MyEvent";  }  } |

1. 在 bundle1 的 DefaultHelloServiceHandler 类的 getHello 方法中，加入提交事件的部分，这样 bundle2 在调用这个服务的时候，将触发一个事件，由于采用了 Post 方式，应该是立刻返回的，所以在 postEvent 前后打印了语句进行验证。   
     
   **清单 7. getHello 方法**

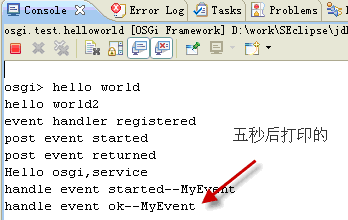
|  |
| --- |
| import org.osgi.framework.BundleContext;  import org.osgi.framework.ServiceReference;  import org.osgi.service.event.EventAdmin;  @Override  public String getHello() {    //post a event  ServiceReference ref =  context.getServiceReference(EventAdmin.class.getName());  if(ref!=null) {  eventAdmin = (EventAdmin)context.getService(ref);  if(eventAdmin!=null) {  System.out.println("post event started");  eventAdmin.postEvent(new MyEvent());  System.out.println("post event returned");  }  }    return "Hello osgi,service";  } |

1. 定义监听者，在 bundle2 中，也引入 osgi 的事件包，然后定义一个新的类：MyEventHandler 类，用来处理事件，这里故意加入了一个延迟，是为了测试异步事件的调用，实现如下：   
     
   **清单 8. MyEventHandler 类**

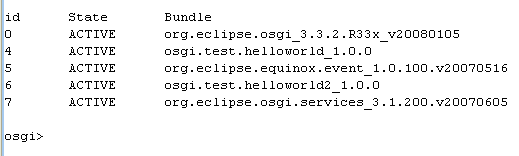
|  |
| --- |
| import org.osgi.service.event.Event;  import org.osgi.service.event.EventHandler;  public class MyEventHandler implements EventHandler {  @Override  public void handleEvent(Event event) {  System.out.println("handle event started--"+event);  try {  Thread.currentThread().sleep(5\*1000);  } catch (InterruptedException e) {    }  System.out.println("handle event ok--"+event);  }  } |

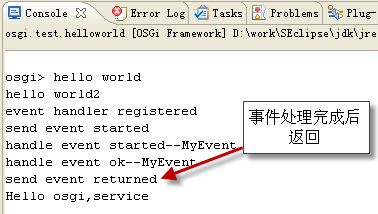
1. 注册监听器，有了事件处理器，还需要注册到监听器中，这里在 bundle2 的 Activator 类中加入此监听器，也就是调用 context.registerService 方法注册这个监听服务，和普通服务的区别是要带一个监听事件类型的 topic，这里列出 Activator 类的 start 方法：   
     
   **清单 9. start 方法**

|  |
| --- |
| import java.rmi.registry.LocateRegistry;  import java.rmi.registry.Registry;  import java.util.Hashtable;  import org.osgi.framework.BundleActivator;  import org.osgi.framework.BundleContext;  import org.osgi.service.event.EventConstants;  import org.osgi.service.event.EventHandler;  import osgi.test.helloworld.event.MyEvent;  import osgi.test.helloworld.service.IAppService;  import osgi.test.helloworld.service.IHello;  public void start(BundleContext context) throws Exception {    System.out.println("hello world2");    /\*\*  \* 添加事件处理器 .  \*/  String[] topics = new String[] {MyEvent.MY\_TOPIC};  Hashtable<String,String[]> ht = new Hashtable<String,String[]>();  ht.put(EventConstants.EVENT\_TOPIC, topics);  EventHandler myHandler = new MyEventHandler();  context.registerService(  EventHandler.class.getName(),  myHandler,  ht);  System.out.println("event handler registered");    /\*\*  \* Test hello service from bundle1.  \*/  IHello hello1 =  (IHello) context.getService(  context.getServiceReference(IHello.class.getName()));  System.out.println(hello1.getHello());  } |

1. 为了使用框架的事件服务，需要修改运行环境，加入两个系统 bundle，分别是：
   1. org.eclipse.osgi.services
   2. org.eclipse.equinox.event
2. 好了一切准备好了，执行：   
     
   **图 20. 执行**  
   

可以看到，post 事件后，不等事件真的被处理完成，就返回了，事件处理在另外的线程执行，最后才打印处理完成的语句。然后 ss 看一下，目前我们已经有五个 bundle 在运行了：

**图 21. ss 查询**  


1. OK，修改代码以测试同步调用的情况，我们只需要把提交事件的代码由 postEvent 修改为 sendEvent 即可。其它不变，测试结果如下：   
     
   **图 22. 同步调用测试结果**  
   

使用 Http 服务 HttpService

OSGi 的 HTTP 服务为我们提供了展示 OSGi 的另外一个途径，即我们可以专门提供一个 bundle 用来作为我们应用的 UI，当然这个还比较简单，只能提供基本的 HTML 服务和基本的 Servlet 服务。如果想提供复杂的 Jsp/Struts/WebWorks 等等，或者想用现有的 Web 中间件服务器例如 Tomcat/Resin/WebSphere Application Server 等，都需要另外的途径来实现，目前我提供一些基本的使用 HTTP 服务的方式。

要使用 HTTP 服务，必然有三个步骤

1. 获取 HttpService，可以像 上述方式 那样通过 context 的 getService 方法获得引用；
2. 使用 HttpService 的引用注册资源或者注册 Servlet：
   1. registerResources：注册资源，提供本地路径、虚拟访问路径和相关属性即可完成注册，客户可以通过虚拟访问路径 + 资源名称访问到资源
   2. registerServlet：注册 Servlet，提供标准 Servlet 实例、虚拟访问路径、相关属性以及 HttpContext（可以为 null）后即可完成注册，客户可以直接通过虚拟访问路径获取该 Servlet 的访问
3. 修改运行环境，加入支持 http 服务的 bundle

那么，接下来我们实际操作一下：

1. 首先，在 bundle1 的 src 中建立一个新的 package，名字叫 pages，用来存放一些 HTML 的资源文件，为了提供一个基本的 HTTP 服务，我们需要提供一个 index.html，内容如下：

|  |
| --- |
| <html>  <h1>hello osgi http service</h1>  </html> |

1. 第二步，注册资源服务，首先我们要为 bundle1 加入 HTTP 服务的 package 引用，即修改 MANIFEST.MF 文件的 dependencies，加入包：org.osgi.service.http;version="1.2.0"，然后在 Activator 类的 start 方法中加入 HTTP 资源的注册：   
     
   **清单 10. 加入 HTTP 资源的注册代码**

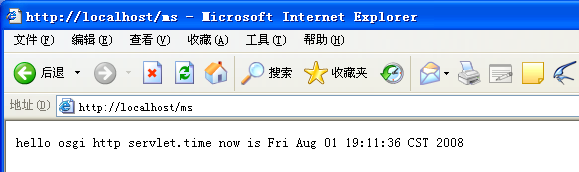
|  |
| --- |
| httpService = (HttpService)context.getService  (context.getServiceReference(HttpService.class.getName()));  httpService.registerResources("/", "/pages", null); |

1. 修改运行环境，在 **target platform** 的 bundle 列表中加入：**org.eclipse.equinox.http** 和 **javax.servlet** 这两个 bundle 保证了 HttpService 的可用性：   
     
   **图 23. 加入 HttpService bundle**  
   图 23. 加入 HttpService bundle
2. 运行，然后打开 IE 访问本机 http://localhost/index.html：   
     
   **图 24. 运行结果**  
   
3. 加入 servlet，首先在 bundle1 建立一个包：osgi.test.hellworld.servlet，建立一个新的类：MyServlet，要从 HttpServlet 基类继承，实现其 doGet 方法，如下：   
     
   **清单 11. MyServlet 代码**

|  |
| --- |
| import java.io.IOException;  import java.util.Date;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  public class MyServlet extends HttpServlet {  /\*\*  \* 实现测试 .  \* @param request the req.  \* @param response the res.  \* @throws IOException io exception.  \*/  public void doGet(  HttpServletRequest request,  HttpServletResponse response  ) throws IOException {  response.getWriter()  .write("hello osgi http servlet.time now is "+new Date());  }  } |

1. 注册 servlet，在 Activator 类的 start 方法中加入注册 servlet 的代码，如下：   
     
   **清单 12. 注册 servlet 的代码**

|  |
| --- |
| MyServlet ms = new MyServlet();  httpService.registerServlet("/ms", ms, null, null); |

1. 运行，打开 IE 访问 http://localhost/ms 后得到结果：   
     
   **图 25. 运行结果**  
   

分布式部署的实现

分布式部署的实现方式一般可以通过 Web 服务、RMI 等方式，这里简单介绍一下基于 RMI 方式的分布式实现。

在 OSGi 环境中，并没有直接提供分布式部署的支持，我们可以采用 J2SE 提供的 RMI 方式来实现，但是要考虑 OSGi 的因素，即如果您希望您的服务既可以本地使用，也可以被远程访问，那么您应该这样定义接口和类：

**图 26. 以被远程访问需要定义的接口和类**  


说明：

1. Remote 接口是 J2SE 定义的远程对象必须实现的接口；
2. IAppService 接口是 OSGi 服务接口，继承了 Remote 接口，即定义方式为：

|  |
| --- |
| public interface IAppService extends Remote |

1. AppServiceImpl 实现了 IAppService 接口，此外注意里面的方法都抛出 RemoteException 异常；

实际操作如下：

1. 在 bundle1 的 service 包中加入 IAppService 接口的定义，继承自 Remote 接口，定义个方法：   
     
   **清单 13. IAppService 接口定义**

|  |
| --- |
| public interface IAppService extends Remote {  /\*\*  \* 得到一个远程服务的名称 .  \* @return .  \* @throws RemoteException .  \*/  String getAppName() throws RemoteException;  } |

1. 把这个接口注册为 OSGi 标准服务以及一个 RMI 服务对象如下：

注册为标准服务：

**清单 14. 注册为标准服务**

|  |
| --- |
| IAppService appService = new DefaultAppServiceImpl(context);  context.registerService(  IAppService.class.getName(),  appService,  null); |

注册为远程对象：

**清单 15. 注册为远程对象**

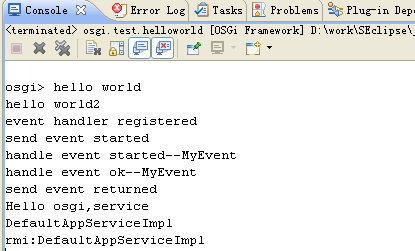
|  |
| --- |
| /\*\*  \* 启动 rmi server .  \* @param service the service.  \* @throws RemoteException re.  \*/  private void startRmiServer(IAppService service) throws RemoteException {  if(registry == null) {  registry = LocateRegistry.createRegistry(1099);  }  // 注册 appService 远程服务 .  IAppService theService =  (IAppService)UnicastRemoteObject.exportObject(service,0);  registry.rebind("appService", theService);  } |

1. 在 bundle2 中通过 OSGi 方式使用这个服务：   
     
   **清单 16. 使用服务**

|  |
| --- |
| IAppService appService =  (IAppService)context.getService(  context.getServiceReference(IAppService.class.getName()));  System.out.println(appService.getAppName()); |

1. 通过 RMI 方式使用这个服务：   
     
   **清单 17. 通过 RMI 方式使用服务**

|  |
| --- |
| String host = "127.0.0.1";  int port = 1099;  try {  Registry registry = LocateRegistry.getRegistry(host,port);  appServiceStub = (IAppService) registry.lookup("appService");  } catch (Exception e) {  e.printStackTrace();  }  System.out.println("rmi:"+appServiceStub.getAppName()); |

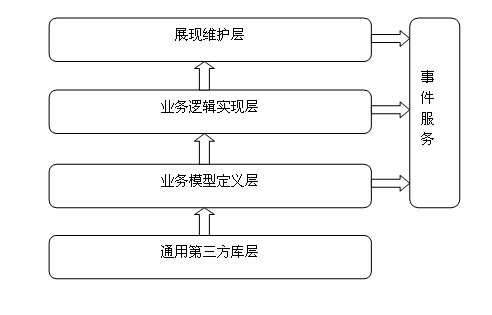
1. 最终的运行结果如下：   
     
   **图 27. 运行结果**  
   

探讨 OSGi 应用架构

设计思路

到目前为止，我们已经涉及到了 OSGi 的诸多方面，那么在实际进行应用程序的架构设计的时候我们要考虑哪些因素呢，这一节我们详细讨论一下这个问题。

应用架构的设计应该充分考虑到可靠性、可扩展性、可维护性等因素，使用了 OSGi 框架后，我们可以更加容易的实现系统分层，组件化的设计方式。通过使用 HTTP 服务我们可以设计出一个基于 HTTP 服务的程序维护平台。架构如下：

**图 28. 基于 HTTP 服务的程序维护平台**  


说明：

1. 通用第三方库层：这一层包括了常用的第三方库，例如 apache commons，jfreechart，xml 包等等，这一层需要将这些包全部 export 出去，这样上层就可以直接通过 require bundle 的方式使用这些包。
2. 业务模型定义层：这一层依赖于（require-bundle）通用第三方库层，定义了应用的业务模型，例如各种 JavaBeans，也可以在这一层提供额外的应用统一配置服务，即为上层应用提供配置文件的管理服务。
3. 业务逻辑实现层：这一层依赖于（require-bundle）通用第三方库层，还依赖于 (import package) 业务模型定义层提供的业务模型，定义了应用的业务逻辑，这一层可以细分为：
   1. DAO（Database Access Object）服务层：即为上层应用提供数据库存取服务的层，其 export 出去的接口全部都是和数据库操作相关的；
   2. Service 层：为 UI 层提供的业务逻辑的封装层，这样 UI 层只需要执行 service 层的接口即可，可以将更多的精力放在 UI 的设计；
4. 展现维护层：这一层依赖于（require-bundle）通用第三方库层，和下面各层提供的管理服务接口，基于 HTTP 服务的方式提供应用的维护，例如配置文件的在线修改、服务的管理，bundle 的管理，日志的管理，内存的管理等等，这些都可以以“ RUNTIME ”的方式展现，管理员或者维护人员操作的就是 Equinox 运行环境。还可以实现大部分的操作不需要重启 JVM，这一点类似于 JMX。
5. 事件服务：事件服务层是 OSGi 框架提供的标准服务之一，为除了通用第三方库层以外的各层提供事件服务，包括同步、异步的通知各种事件、发布各种事件等。通过事件服务，可以实现各层之间的联动。

这种架构的优势在于：

1. 各层只用关心自己的业务，例如通用第三方库层只需要 export，其它事情不用管，它也没有自己的 Activator 类，业务模型定义层只需要关心业务模型，而不必关心业务的流程，业务逻辑层中的 DAO 层则只需要关心数据库操作，service 层则负责组合业务流程。各司其职，这样才能精于自己的模块；
2. 较好的可维护性：最上层的维护展现层，为管理员提供了一个 OSGi 应用的管理窗口，提供在线重启服务、管理各个 bundle 和服务的能力，提供了类似于 JMX 的能力；
3. 统一的事件管理框架：为各层定义了统一的事件管理接口，基于 TOPIC 方式的事件监听机制能够有效的过滤事件，而且提供了异步、同步两种方式对事件进行处理，可以说有相当大的灵活性。

可维护性的考虑

一般的应用架构可能都比较多的考虑可靠性、灵活性、可扩展性等，对可维护性却没有提供太多的关注，使用 OSGi 后，将对可维护性提供类似于 JMX 的支持，当然这不需要您实现 MBEAN，就像上述介绍的架构设计，我们在最上层可以设计一个基于 HTTP 的维护层，这样，提供了一个小的 Web 控制台，供管理员进行维护。

维护的方面包括：

1. 系统维护
   1. Bundle 的管理：包括每个 bundle 的更新、停止、启动；
   2. 服务的管理：包括运行环境注册的服务的列表、停止、启动；
   3. 系统所有服务的重启、停止、启动；
2. 系统状态的监控
   1. 对各个业务层提供的服务的状态进行实时监视、统计；
   2. 对各个业务层提供服务的状态进行控制，通过 OSGi 事件的方式进行通知；
3. 系统日志的管理
   1. 对系统中各个层的日志进行统一列表、查看；
   2. 对系统中所有操作进行统一日志记录、管理；
4. 配置管理
   1. 对各个业务模块需要的配置文件进行统一展示；
   2. 对各个业务模块提供的配置文件提供在线编辑、提交功能；
   3. 对修改后的配置文件提供实时上线的功能；
5. 其它
   1. 维护系统的登录、登出；
   2. 维护系统自审计；
   3. 维护系统权限控制；

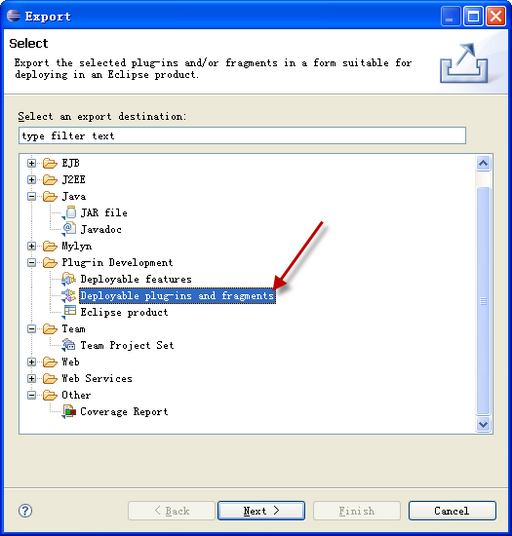
部署 OSGi 应用程序

我们的 bundle 不会只能在 Eclipse 环境运行，我们需要能够将 bundle 部署到实际的操作系统中，可能是 Windows/Linux/Unix 等环境，这要求我们按照下列步骤进行：

1. 发布 bundle，即将我们的 plug-in 工程发布为可以执行的 Jar 文件或者其它格式；
2. 配置 config.ini，指出 bundle 的运行环境，启动顺序等；
3. 启动脚本编写，编写能够运行在各种操作系统的脚本；

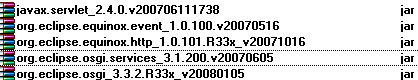
发布 Bundle

发布 bundle 的工作其实很简单，通过 eclipse 平台即可完成：

1. 选择 Eclipse 的 **plug-in** 视图的 File -> Export，从弹出的窗口中选择 **Deployable plug-ins and fragments**：   
     
   **图 29. 选择 Deployable plug-ins and fragments**  
   
2. 在下一个窗口中，选择想要发布的 bundle，这里我们选择 **osgi.test.helloworld**，**osgi.test.helloworld2** 工程，下面的 options 里面选择“打包为一个 Jar ”，目标目录选择为 osgi.test.deploy 目录（在当前的 workspace 下面）；
3. 选择确定，发布后的目录结构如下图所示，eclipse 帮我们在部署根目录下建立了一个新的子目录 plugins（类似于 eclipse，因为 eclipse 就是基于 OSGi 的）：   
     
   **图 30. 发布后的目录结构**  
   
4. 好了，到这里，发布工作完成。

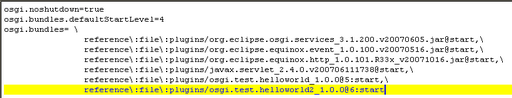
Config.ini

为了让我们的 Jar 文件跑起来，需要 OSGi 的运行环境支持，所以我们需要拷贝一些 system bundle 到 plugins 目录中，包括：

**图 31. OSGi 的运行环境支持**  


然后，把 eclipse 目录的 org.eclipse.osgi\_3.3.2.R33x\_v20080105 文件拷贝到 osgi.test.deploy 根目录，重命名为 equinox.jar 文件。

在 osgi.test.deploy 目录新建子目录 configuration，新建一个文本文件 config.ini，用来配置 bundle 的启动环境，配置如下：

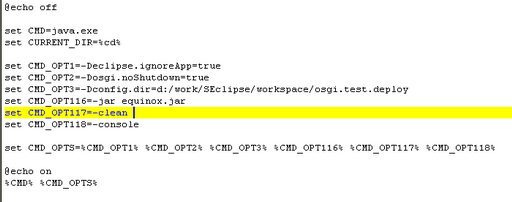
**图 32. config.ini 配置文件**  


注意最后两个 bundle 的启动顺序配置格式为：bundle@start\_leve:start。

好了，config.ini 也已经准备好了。

启动脚本

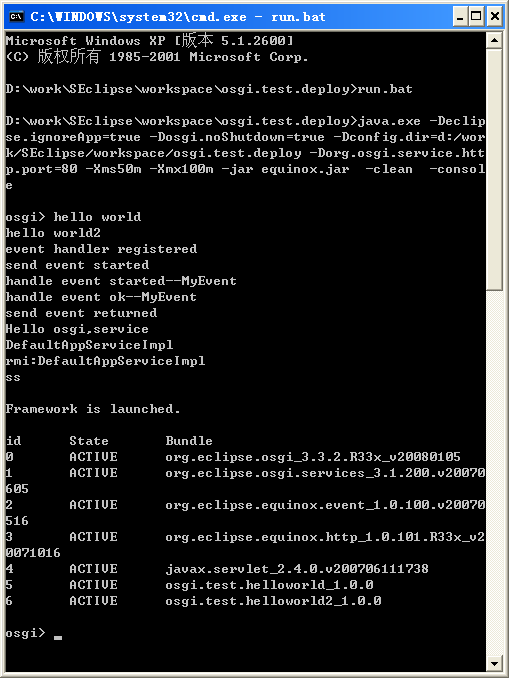
下面进行启动脚本编写，这个和普通的 Java 程序没有什么大的区别，都是调用 Java 程序执行一个 jar 文件，关键是其中的一些参数定义：

**图 33. 启动脚本**  


注意 1/2/3/117/118 参数都是 OSGi 环境特有的。

运行

双击 run.bat，可以看到如下结果：

**图 34. 运行结果**  
  
总结

通过阅读本文您应该已经掌握了使用 Equinox 开发基于 OSGi 的应用程序的方法，了解了其关键的理论知识，还学习了如何开发分层的，模块化的、分布式的应用程序，掌握了在 Windows 平台部署基于 Equinox 平台的 OSGi 应用程序的方法。总体上看，OSGi 能够有效的降低模块之间的耦合程度，将软件设计的开闭原则（Open-Close Principle）提高到一个新的水平，另外 OSGi 也为系统架构设计提供了更大的灵活性，使得我们开发出像 Eclipse 那样插件化的平台系统不再遥不可及。

参考资料

**学习**

* [Eclipse.org](http://www.eclipse.org/)：获得有关 Eclipse 的更多详细资料。
* [Equinox](http://www.eclipse.org/equinox)：获得有关 Equinox 框架的更多详细资料。
* [OSGi Alliance Service Platform](http://www.osgi.org/)：了解更多关于 OSGi 的信息，包括 OSGi Release 4 规范等信息。
* [Help – Eclipse SDK](http://help.eclipse.org/help32/index.jsp)：获得在 Eclipse 下进行开发的详细帮助文档。
* “[Eclipse 平台入门 -- 使用 Eclipse 插件来编辑、编译和调试应用程序](http://www.ibm.com/developerworks/cn/linux/opensource/os-ecov/index.shtml)”（developerWorks，2004 年 2 月）：本文为您提供关于 Eclipse 平台的概述，包括其起源和体系结构。
* “[了解 Eclipse 插件如何使用 OSGi](http://www.ibm.com/developerworks/cn/opensource/os-ecl-osgi/)”（developerWorks，2006 年 9 月）：阐明了 Eclipse 与 OSGi 的关系，还解释了 OSGi manifest.mf 文件选项以及通过 Eclipse 提供的添加项。
* “[基于 OSGi 的面向服务的组件编程](http://www.ibm.com/developerworks/cn/opensource/os-cn-ecl-osgi/)”（developerWorks，2007 年 8 月）：本文介绍了基于 OSGi 开发一个应用程序的过程，读者可以学习如何基于 OSGi 开发自己的应用。
* “[探索 OSGi 框架的组件运行机制](http://www.ibm.com/developerworks/cn/java/j-lo-osgi/)”（developerWorks，2008 年 7 月）：本文介绍了 OSGi 框架中的组件（Bundle）的运行机制，并结合实际示例加以说明。
* [developerWorks Eclipse 技术资源中心](http://www.ibm.com/developerworks/cn/eclipse/)：这里汇集了大量和 Eclipse 开发平台相关的技术文章和教程。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：这里有数百篇关于 Java 编程方方面面的文章。
* 浏览 [技术书店](http://www.ibm.com/developerworks/apps/SendTo?bookstore=safari)，阅读关于这些技术主题和其它技术主题的书籍。

**获得产品和技术**

* 下载 [Eclipse](http://www.eclipse.org/downloads/)。
* 下载 [Equinox OSGi 框架](http://download.eclipse.org/eclipse/equinox/)。

**讨论**

* 访问 [Eclipse 新闻组](http://www.eclipse.org/newsgroups/)。
* 访问 [Eclipse 邮件列表](http://www.eclipse.org/mail/)。
* 访问 [Equinox 邮件列表](https://dev.eclipse.org/mailman/listinfo/equinox-dev)。
* 通过参与 [developerWorks blogs](http://www.ibm.com/developerworks/blogs/) 加入 [developerWorks 社区](http://www.ibm.com/developerworks/community)。