/\*\*

\*  转载请注明作者longdick    http://longdick.iteye.com

\*

\*/

OSGI(Open Services Gateway Initiative)，或者通俗点说JAVA动态模块系统，定义了一套模块应用开发的框架。OSGI容器实现方案如Knopflerfish, Equinox, and Apache Felix允许你把你的应用分成多个功能模块，这样通过依赖管理这些功能会更加方便。

和Servlet和EJB规范类似，OSGI规范包含两大块：一个OSGI容器需要实现的服务集合；一种OSGI容器和应用之间通信的机制。开发OSGI平台意味着你需要使用OSGI API编写你的应用，然后将其部署到OSGI容器中。从开发者的视角来看，OSGI提供以下优势：

1. 你可以动态地安装、卸载、启动、停止不同的应用模块，而不需要重启容器。
2. 你的应用可以在同一时刻跑多个同一个模块的实例。
3. OSGI在SOA领域提供成熟的解决方案，包括嵌入式，移动设备和富客户端应用等。

OK，你已经有个Servlet容器来做web 应用，有了EJB容器来做事务处理，你可能在想为什么你还需要一个新的容器？简单点说，OSGI容器被设计专门用来开发可分解为功能模块的复杂的Java应用。

**企业应用领域的OSGI**

OSGI规范最初是由OSGI联盟在1999年3月发起。它的主要目的是成为向网络设备传输服务管理的开放规范。核心思想是一旦你向网络设备中添加了一个OSGI服务平台，你可以在网络中的任意位置管理该设备上的服务组件。这些服务组件可以任意安装，更新或移除而不会对设备产生影响。

多年来，OSGI技术只出现在嵌入式系统和网络设备市场。现在，Eclipse使OSGI在企业开发领域焕发出新的光彩。

**OSGI受到越来越广泛的支持**

2003年，Eclipse开发团队开始寻找一种使eclipse成为一种功能更动态、工具更模块化的富客户端平台。最终，他们的目光锁定在OSGI框架上。Eclipse3.0，2004年6月发布，是基于OSGI技术搭建的首个Eclipse版本。

几乎所有企业应用服务提供商支持或计划支持OSGI。Spring框架同样支持OSGI，通过Spring DM（Spring Dynamic Modules for OSGI Service Platforms）项目，可以让我们在Spring上更方便的应用OSGI。

**开源OSGI容器**

从企业应用开发者的角度看，OSGI容器侵入性非常小，你可以方便地将其嵌入一个企业应用。举个例子来说，假设你在开发一个复杂的web应用。你希望将这个应用分解成多个功能模块。一个View层模块，一个Model层模块，一个DAO模块。使用嵌入式OSGI容器来跨依赖地管理这些模块可以让你随时更新你的DAO模块却不需要重启你的服务器。

只要你的应用完全符合OSGI规范，它就可以在所有符合OSGI规范的容器内运行。现在，有三种流行的开源OSGI容器：

1. Equinox是OSGI Service Platform Release 4的一个实现。是Eclipse 模块化运行时的核心。
2. Knopflerfish另一个选择。
3. Apache Felix是Apache软件基金会赞助的一个OSGI容器

在这篇文章里我们使用Equinox作为我们的OSGI容器。

**尝试开发一个Hello World bundle**

在OSGI的领域，发布的软件是以bundle的形式出现。bundle由java class类和资源文件组成，向设备所有者提供功能，同时可以为其他的bundles提供服务。Eclipse对开发bundles提供了强大的支持。Eclipse不仅仅提供创建bundles的功能，它还集成了Equinox这个OSGI容器，你可以在其上开发和调试OSGI组件。其实所有的Eclipse插件都是使用Eclipse规范代码写的OSGI bundle。接下来，你将可以学到如何使用Eclipse IDE开发一个Hello world osgi bundle。

**开始开发bundle**

我们一步步的开始：

1. 启动Eclipse，依次点 File --> New --> Project。
2. 选择Plug-in Project，next。
3. 输入Project Name项目名称，比如com.howard.sample.HelloWorld,Target Platform(目标平台)里的an OSGI framework，选择standard。
4. 剩下的保持默认，next。
5. 下个对话框也默认，next。
6. 然后选择Hello OSGI Bundle作为模版。Finish。

Eclipse会飞快的为你创建Hello world bundle的模版代码。主要包含两个文件：Activator.java和MANIFEST.MF。

Activator.java的代码如下所示：

**Java代码  [收藏代码](javascript:void())**

1. **import** org.osgi.framework.BundleActivator;
2. **import** org.osgi.framework.BundleContext;
3. **public** **class** Activator **implements** BundleActivator {
4. **public** **void** start(BundleContext context) **throws** Exception {
5. System.out.println("Hello world");
6. }
7. **public** **void** stop(BundleContext context) **throws** Exception {
8. System.out.println("Goodbye World");
9. }
10. }

如果你的bundle在启动和关闭的时候需要被通知，你可以考虑实现BundleActivator接口。以下是定义Activator的一些注意点：

1. 你的Activator类需要一个公有的无参数构造函数。OSGI框架会通过类反射的方式来实例化一个Activator类。
2. 容器启动bundle过程中负责调用你的Activator类的start方法。bundle可以在此初始化资源比如说初始化数据库连接。start方法需要一个参数，BundleContext对象。这个对象允许bundles以取得OSGI容器相关信息的方式和框架交互。如果某一个bundle有异常抛出，容器将对该bundle标记为stopped并不将其纳入service列表。
3. 容器关闭的时候会调用你的Activator类方法stop(),你可以利用这个机会做一些清理的操作。

**MANIFEST.MF**

这个文件是你的bundle的部署描述文件。格式和Jar里的MANIFEST.MF是一样的。包含的不少名值对，就像如下：

**Xml代码  [收藏代码](javascript:void())**

1. Manifest-Version: 1.0
2. Bundle-ManifestVersion: 2
3. Bundle-Name: HelloWorld Plug-in
4. Bundle-SymbolicName: com.howard.sample.HelloWorld
5. Bundle-Version: 1.0.0
6. Bundle-Activator: com.howard.sample.helloworld.Activator
7. Bundle-Vendor: HOWARD
8. Bundle-RequiredExecutionEnvironment: JavaSE-1.6
9. Import-Package: org.osgi.framework;version="1.3.0"

分别来看下：

Bundle-ManifestVersion

 数值为2意味着本bundle支持OSGI规范第四版；如果是1那就是支持OSGI规范第三版。

Bundle-Name

 给bundle定义一个短名，方便人员阅读

Bundle-SymbolicName

 给bundle定义一个唯一的非局部名。方便分辨。

Bundle-Activator

 声明在start和stop事件发生时会被通知的监听类的名字。

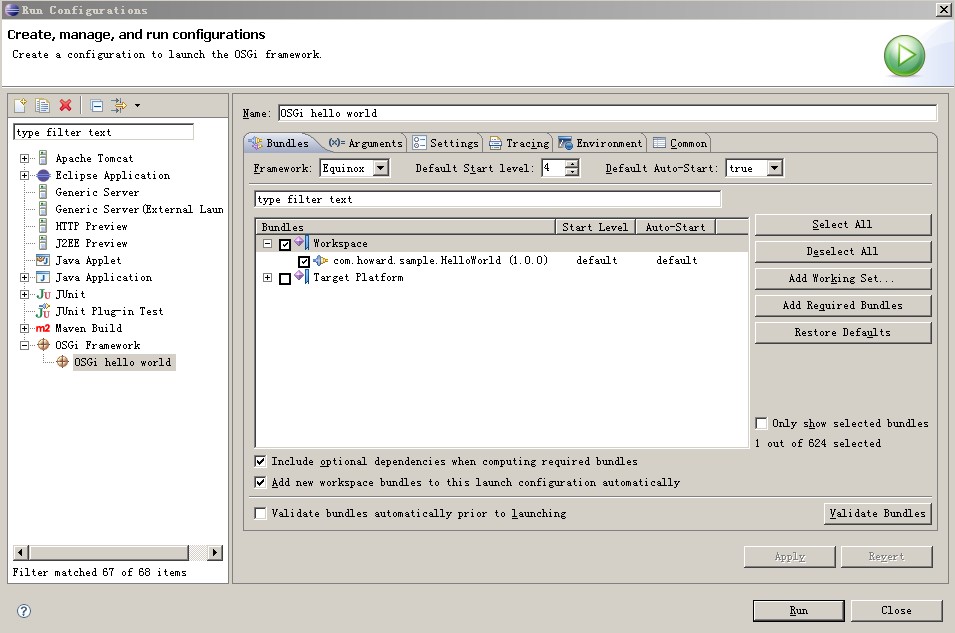
Import-Package

 定义bundle的导入包。

Hello World bundle完成了，接下来我们运行一下。

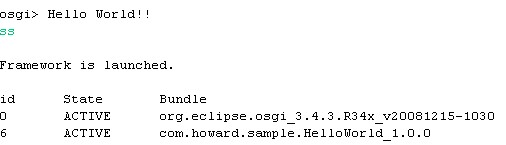
**执行bundle**

1. 点击Run --> Run Configuration
2. 在左边的OSGI Framework选项里右键 new ，创建一个新的OSGI Run Configuration
3. 名字随便取好了，我们取个OSGi hello world。
4. 你会注意到中间的窗口里Workspace项目里有一子项 com.howard.sample.HelloWorld,将其勾选上，其他的不用管。这时的状态应该如下图。
5. 点击Run按钮。在控制台你应该可以看见点东西了。那是叫做OSGI控制台的东东。与子相伴，还有一个"Hello world"。



**OSGI控制台**

OSGI控制台是一个OSGI容器的命令行界面。你可以利用它做些诸如启动，关闭，安装bundles，更新和删除bundles等操作。现在，点击OSGI控制台所在的位置，回车，你就会发现可以输入命令了。这时的OSGI控制台应该如下图：



下面列出一些常用的OSGI命令，你可以试着和OSGI容器交互。

ss 显示已安装的bundles的状态信息，信息包括bundle ID，短名，状态等等。

start 启动一个bundle

stop  关闭一个bundle

update  载入一个新的JAR文件更新一个bundle

install  安装一个新的bundle到容器中

uninstall  卸载一个已在容器中的bundle

**依赖管理**

OSGI规范允许你把你的应用分解成多个模块然后管理各个模块间的依赖关系。

这需要通过bundle scope来完成。默认情况下，一个bundle内的class对其他bundle来说是不可见的。那么，如果要让一个bundle访问另一个bundle里的class要怎么做？解决的方案就是从源bundle导出包，然后在目标bundle里导入。

接下来我们对此做一个例子。

首先，我们需要先创建一个com.howard.sample.HelloService bundle,我们将通过它导出一个包。

然后，我们在com.howard.sample.HelloWorld 这个bundle里导入包。

**导出包**

1、创建名为com.howard.sample.HelloService的bundle，创建步骤和前面一样。

2、在这个bundle内，添加一个com.howard.sample.service.HelloService.java 接口，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** HelloService {
2. **public** String sayHello();
3. }

3、创建一个com.howard.sample.service.impl.HelloServiceImpl.java类实现刚才的接口：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloServiceImpl **implements** HelloService{
2. **public** String sayHello() {
3. System.out.println("Inside HelloServiceImple.sayHello()");
4. **return** "Say Hello";
5. }
6. }

4、打开MANIFEST.MF,选择Runtime标签项，在Exported Packages选项栏，点击Add并且选择com.howard.sample.service这个包。然后MANIFEST.MF的代码应该如下：

**Xml代码  [收藏代码](javascript:void())**

1. Manifest-Version: 1.0
2. Bundle-ManifestVersion: 2
3. Bundle-Name: HelloService Plug-in
4. Bundle-SymbolicName: com.howard.sample.HelloService
5. Bundle-Version: 1.0.0
6. Bundle-Activator: com.howard.sample.helloservice.Activator
7. Bundle-Vendor: HOWARD
8. Bundle-RequiredExecutionEnvironment: JavaSE-1.6
9. Import-Package: org.osgi.framework;version="1.3.0"
10. Export-Package: com.howard.sample.service

你可以看到，MANIFEST.MF文件和刚才的HelloWorld的那份很类似。唯一的区别就是这个多了Export-Package这个标记，对应的值就是我们刚才选择的com.howard.sample.service。

Export-Package标记告诉OSGI容器在com.howard.sample.service包内的classes可以被外部访问。

注意，我们仅仅暴露了HelloService接口，而不是直接暴露HelloServiceImpl实现。

**导入包**

接下来我们要更新原来的HelloWorld bundle以导入com.howard.sample.service包。步骤如下：

1、进入HelloWorld bundle，打开MANIFEST.MF，进入Dependencies标签页，在Imported Packages里添加com.howard.sample.service。MANIFEST.MF文件应该如下所示：

**Xml代码  [收藏代码](javascript:void())**

1. Manifest-Version: 1.0
2. Bundle-ManifestVersion: 2
3. Bundle-Name: HelloWorld Plug-in
4. Bundle-SymbolicName: com.howard.sample.HelloWorld
5. Bundle-Version: 1.0.0
6. Bundle-Activator: com.howard.sample.helloworld.Activator
7. Bundle-Vendor: HOWARD
8. Bundle-RequiredExecutionEnvironment: JavaSE-1.6
9. Import-Package: com.howard.sample.service,
10. org.osgi.framework;version="1.3.0"

没错，Import-package标记的值也就是导入的包名之间是用逗号隔开的。在这里导入了两个包om.howard.sample.service和org.osgi.framework。后者是使用Activator类时必须导入的包。

2、接下来，打开HelloWorld项目下的Activator.java文件，这时候你会发现可以使用HelloService这个接口了。但还是不能使用HelloServiceImpl实现类。Eclipse会告诉你：Access restriction（立入禁止）。

**Class级别可见域**

为什么OSGI容器可以做到让jar包中的一些classes可见而另一些又不可见呢。

答案其实就是OSGI容器自定义了java class loader来有选择的加载类。OSGI容器为每一个bundle都创建了不同的class loader。因此，bundle可以访问的classes包括

* Boot classpath：所有的java基础类。
* Framework classpath：OSGI框架级别的classloader加载的类
* Bundle classpath：Bundle本身引用的关系紧密的JAR的路径
* Imported packages：就是在MANIFEST.MF里声明的导入包，一旦声明，在bundle内就可见了。

bundle级别的可见域允许你可以随时放心的更改HelloServiceImpl实现类而不需要去担心依赖关系会被破坏。

**OSGI服务**

OSGI框架是实现SOA的绝佳土壤。通过它可以实现bundles暴露服务接口给其他bundles消费而不需要让细节暴露。消费bundles甚至可以完全不知道提供服务的bundles。凭着可以良好的隐藏具体实现的能力，OSGI当之无愧是SOA的一种较完美的实现方案。

OSGI中，提供服务的bundle在OSGI容器上将一个POJO注册成一个service。消费者bundle请求OSGI容器中基于某个特殊接口的注册service。一旦找到，消费者bundle就会绑定它，然后就可以调用service中的方法了。举个例子会更容易说明。

**导出services**

1、确保com.howard.sample.HelloService里的MANIFEST.MF导入org.osgi.framework包

2、创建com.howard.sample.service.impl.HelloServiceActivator.java,代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloServiceActivator **implements** BundleActivator {
2. ServiceRegistration helloServiceRegistration;
3. @Override
4. **public** **void** start(BundleContext context) **throws** Exception {
5. HelloService helloService = **new** HelloServiceImpl();
6. helloServiceRegistration = context.registerService(HelloService.**class**
7. .getName(), helloService, **null**);
8. }
10. @Override
11. **public** **void** stop(BundleContext context) **throws** Exception {
12. helloServiceRegistration.unregister();
13. }
15. }

OK，我们就是用BundleContext的registerService方法注册service的。这个方法需要三个参数。

* service的接口名。如果service实现了多个接口，那样你需要传入一个包含所有接口名的String数组。在这里我们传入的是HelloService这个接口。
* 真正的service实现。在例子中我们传了一个HelloServiceImpl实现。
* service属性。这个参数可以在有多个service实现同一个接口的情况下，消费者用来区分真正感兴趣的service。

3、最后一步就是修改HelloService的MANIFEST.MF文件，将Bundle-Activator改成com.howard.sample.service.impl.HelloServiceActivator

现在HelloService bundle已经随时准备将HelloServiceImpl服务发布了。OSGI容器启动HelloServie bundle的时候会让HelloServiceActivator运作，在那个时候将HelloServiceImpl注册到容器中，接下来就是创建消费者的问题了。

**导入service**

我们的消费者就是HelloWorld bundle，主要修改的就是其中的Activator.java,修改代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** Activator **implements** BundleActivator {
2. ServiceReference helloServiceReference;
3. **public** **void** start(BundleContext context) **throws** Exception {
4. System.out.println("Hello World!!");
5. helloServiceReference=context.getServiceReference(HelloService.**class**.getName());
6. HelloService helloService=(HelloService)context.getService(helloServiceReference);
7. System.out.println(helloService.sayHello());
8. }
9. **public** **void** stop(BundleContext context) **throws** Exception {
10. System.out.println("Goodbye World!!");
11. context.ungetService(helloServiceReference);
12. }
13. }

代码很简单，就不多说了。

在运行之前我们在Run-->Run Configurations对话框里，把HelloWorld和HelloService这两个bundle前面的钩都打上。然后运行时你会发现HelloServiceImpl.sayHello()方法已经被调用了。

在OSGI控制台输入ss并回车，所有容器内的bundle状态一目了然。其中id为0的bundle是OSGI框架基础bundle，另两个就是HelloService和HelloWorld了，它俩的id是随机的，状态是ACTIVE也就是已启动状态。假设HelloService的id为7，HelloWorld为8。

输入stop 8就可以暂停bundle的运行，容器内这个bundle还是存在的，只是状态变成了RESOLVED。再次启动使用start 8，然后就会看见HelloWorld bundle消费了HelloService的服务。

**创建服务工厂**

刚才例子所示，我们会在HelloService bundle启动时初始化并注册service。然后不管存不存在消费端，这个service都会存在，而且消费端取得的service 实例其实都是同一个。OK，某些servie是比较耗费资源的主，我们不希望它一直占用资源，最好是在真正用它的时候创建不用的时候销毁就最好了。

解决如上问题的方案就是使用ServiceFactory接口的实现来代替原先service具体的实现到OSGI容器去注册。这样，以后只有当其他bundle请求该服务时，才会由ServiceFactory实现类来处理请求并返回一个新的service实例。

实例步骤如下：

1、在HelloService bundle创建一个实现ServiceFactory接口的类HelloServiceFactory类，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloServiceFactory **implements** ServiceFactory {
2. **private** **int** usageCounter = 0;
3. @Override
4. **public** Object getService(Bundle bundle, ServiceRegistration registration) {
5. System.out.println("Create object of HelloService for " + bundle.getSymbolicName());
6. usageCounter++;
7. System.out.println("Number of bundles using service " + usageCounter);
8. HelloService helloService = **new** HelloServiceImpl();
9. **return** helloService;
10. }
11. @Override
12. **public** **void** ungetService(Bundle bundle, ServiceRegistration registration, Object service) {
13. System.out.println("Release object of HelloService for " + bundle.getSymbolicName());
14. usageCounter--;
15. System.out.println("Number of bundles using service " + usageCounter);
16. }
17. }

ServiceFactory接口定义了两个方法：

* getService方法：特定的bundle在第一次调用BundleContext的getService方法时由OSGI框架调用，在实例代码中，我们用这个方法来返回一个新的HelloService的实现。OSGI框架会缓存这个返回的对象，如果同一个bundle在未来再次调用BundleContext的getService方法的话，会直接返回这个缓存中的对象。
* ungetService方法：bundle释放service的时候由OSGI容器调用。

2、修改HelloServiceActivator.java的start方法，将ServiceFactory作为服务注册，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloServiceActivator **implements** BundleActivator {
2. ServiceRegistration helloServiceRegistration;
3. @Override
4. **public** **void** start(BundleContext context) **throws** Exception {
5. HelloServiceFactory helloServiceFactory = **new** HelloServiceFactory();
6. helloServiceRegistration = context.registerService(HelloService.**class**
7. .getName(), helloServiceFactory, **null**);
8. }
10. @Override
11. **public** **void** stop(BundleContext context) **throws** Exception {
12. helloServiceRegistration.unregister();
13. }
14. }

现在运行下试试看，你会发现HelloWorld bundle启动时才会初始化HelloService，控制台会打印出"Number of bundles using service 1"，当HelloWorld bundle暂停时会打印出"Number of bundles using service 0"。

**services跟踪**

某种情形下，我们可能需要在某个特殊的接口有新的服务注册或取消注册时通知消费端。这时我们可以使用ServiceTracker类。如下步骤所示：

1、在HelloWorld bundle里的MANIFEST.MF导入org.util.tracker包。

2、创建HelloServiceTracker类，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloServiceTracker **extends** ServiceTracker {
2. **public** HelloServiceTracker(BundleContext context) {
3. **super**(context, HelloService.**class**.getName(),**null**);
4. }
5. **public** Object addingService(ServiceReference reference) {
6. System.out.println("Inside HelloServiceTracker.addingService " + reference.getBundle());
7. **return** **super**.addingService(reference);
8. }
9. **public** **void** removedService(ServiceReference reference, Object service) {
10. System.out.println("Inside HelloServiceTracker.removedService " + reference.getBundle());
11. **super**.removedService(reference, service);
12. }
13. }

我们在HelloServiceTracker的构造函数里将HelloService接口名传进去，ServiceTracker会跟踪实现这个接口的所有的注册services。ServiceTracker主要有两个重要方法：

* addingService方法：bundle注册一个基于给定接口的service时调用。
* removeService方法：bundle取消注册一个基于给定接口的service时调用。

3、修改Activator类，使用刚刚创建的HelloServiceTracker来获取service：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** Activator **implements** BundleActivator {
2. HelloServiceTracker helloServiceTracker;
3. **public** **void** start(BundleContext context) **throws** Exception {
4. System.out.println("Hello World!!");
5. helloServiceTracker= **new** HelloServiceTracker(context);
6. helloServiceTracker.open();
7. HelloService helloService=(HelloService)helloServiceTracker.getService();
8. System.out.println(helloService.sayHello());
9. }
11. **public** **void** stop(BundleContext context) **throws** Exception {
12. System.out.println("Goodbye World!!");
13. helloServiceTracker.close();
14. }
15. }

现在运行一下，可以发现只要HelloService bundle启动或是暂停都会导致HelloServiceTracker的对addingService或removedService方法的调用。

ServiceTracker不仅仅能跟踪Service的动向，它还能通过getService方法取得Service实例并返回。但是如果同一个接口下有多个service注册，这时返回哪个service呢？这时候就需要看service的等级哪个高了。这个等级是service注册时的property属性里的一项：SERVICE\_RANKING。谁的SERVICE\_RANKING高，就返回谁。

如果有两个一样高的呢？这时再看这两个service谁的PID更低了。

如果对OSGI的ClassLoader机制有疑问，可以看看这篇解释ClassLoader机制和自定义ClassLoader相关的文章：

<http://longdick.iteye.com/blog/442213>

OSGI的基本原理和入门开发就到这里了。希望同学们能对OSGI开发有个简单而清晰的认识。