# OSGI

OSGi(Open Service Gateway Initiative)技术是Java动态化模块化系统的一系列规范。OSGi一方面是指维护OSGi规范的OSGI官方联盟，另一方面指的是该组织维护的基于Java语言的服务（业务）规范。简单来说，OSGi可以认为是Java平台的模块层。

OSGi服务平台向Java提供服务，这些服务使Java成为软件集成和软件开发的首选环境。Java提供在多个平台支持产品的可移植性。OSGi技术提供允许应用程序使用精炼、可重用和可协作的组件构建的标准化原语，这些组件能够组装进一个应用和部署中。

# OSGI案例

**Eclipse**作为Java业界成功的IDEproject，在3.0以前的版本它采用的是自己设计的一套插件体系结构，而Eclipse的插件体系结构在整个业界都是非常知名的，也是被认为非常成功的一种设计，但Eclipse在3.0版本时却做了一个重大决度，就是推翻它自己以前的插件体系结构，采用OSGI作为其插件体系结构。Eclipse之所以要抛弃自己那套已经比较成熟的插件体系结构而转而采用OSGI，就是因为OSGI的规范性以及OSGI对于插件体系结构更为完整的定义，Eclipse采用OSGI作为其插件体系结构的成功是很明显的，在Eclipse 3.1版本以后大家可以明显的感觉到启动速度的提升，同时也使得可以在运行时对插件进行管理，更明显的提升是插件的开发更加的规范，从而可以使用很多已有的OSGI插件。

**BMW汽车的应用控制系统**采用OSGI作为其底层架构，很多人都认为基于java的系统低效，不可能用于汽车这样的应用控制系统上。这套系统主要用来控制汽车上的音箱、灯光等等设备，总共由1000多个Bundle构成，但BMW汽车的应用控制系统启动时间却只需要3.5秒，这也从很大程度上反应了采用OSGI的系统的效率并不会低。

# OSGI 规范

# OSGI 框架

## Equinox

最知名，也是更新最频繁的，由于Eclipse 基金的支持，其功能越来越完善，实现了OSGi R4 规范，并提供很多平台性质的服务，包括：常用功能模块、日志模块、Web服务器模块、Servlet 模块、JSP解析模块等等。由于其与Eclipse 的天然联系，使得开发基于Equinox的应用程序变得很简单。Eclipse 3.1 之后的版本，Eclipse 本身就已经包含了Equinox。

遵循EPL (The Eclipse Public License)，任何扩展自Eclipse源码的代码也必须是开源的,商业软件可以使用，也可以修改代码，但要承担代码产生的侵权责任。

## Knopflerfish

很早的，也很优秀的一个OSGI框架，也实现了OSGI R4 标准。该项目的宗旨在于创建一个易于开发的OSGI平台，与Equinox 不同之处在于它本身提供一些小应用实例，包括一个可视化控制台等，也提供基于Eclipse 的插件。

遵循Knopflerfish License (BSD-esque)

<http://www.knopflerfish.org/license.html>

商业软件可以使用，也可以修改使用BSD协议的代码

## Felix

很新的一个OSGi 框架，社区很活跃，更新频率高，是Apache 的开源项目。该项目2007年8月才出1.0 版，也实现了OSGiR4 规范，也提供相关的基础服务和扩展服务功能。

Felix也有了Eclipse集成支持，开发者可以在Eclipse IDE里运行Felix。

Felix组件按照Apache软件许可证2.0（Apache Software License Version 2.0）来发布许可。

Apache Licence也是对商业应用友好的许可。使用者也可以在需要的时候修改代码来满足需要并作为开源或商业产品发布/销售。商业软件可以使用，也可以修改使用Apache协议的代码。

# 与流行框架结合

Spring-DM 指的是Spring Dynamic Modules。Spring-DM 的主要目的是能够方便地将Spring 框架和OSGi 框架结合在一起，使得使用Spring 的应用程序可以方便简单地部署在OSGi 环境中，利用OSGi框架提供的服务， 将应用变得更加模块化。

# Equinox框架

基于equinox框架开发，需要下载equinox，开发的项目其实是eclipse项目。另加之eclipse ide 本身在3.1版本之后就是基于osgi开发的，换言之eclipse ide 就是基于equinox开发的，因此可在eclipse 的plugins 目录中找到类似org.eclipse.osgi\_3.4.3.R34x\_20081215-1030.jar文件（不同版本的eclipse对应的版本和日期会有所不同），jar文件org.eclipse.osgi\_3.4.3.R34x\_20081215-1030.jar就是equinox的OSGI R4 Core的实现。

## 检查equinox环境

### 启动eclipse

没啥可说的。

### 打开run configuration 窗口

|  |
| --- |
| http://image52.360doc.com/DownloadImg/2012/05/3015/24480591_1.jpg |

### 创建osgi framework类型的运行配置

|  |
| --- |
| http://image52.360doc.com/DownloadImg/2012/05/3015/24480591_2.jpg |

### 显示所有的bundles

|  |
| --- |
| http://image52.360doc.com/DownloadImg/2012/05/3015/24480591_3.jpg |

### 取消bundles的选择

|  |
| --- |
| http://image52.360doc.com/DownloadImg/2012/05/3015/24480591_4.jpg |

### 选择org.eclipse.osgi bundle

|  |
| --- |
| http://image52.360doc.com/DownloadImg/2012/05/3015/24480591_5.jpg |

### 运行

点击run 按钮，如果无误，则console会出现“osgi>”字符串，输入 “ss“ 回车，出现以下界面：

|  |
| --- |
|  |

## 第一个bundle

### 创建bundle工程（其实就是plugin-in 工程）

|  |
| --- |
|  |

输入工程相关信息，这里和创建普通的java项目唯一的不同点就是要选择 this plugin-in is targeted to run with，在这里我们暂选择 an osgi framework 的standard选项，standard意为建立一个标准的osgi bundle工程。

Plgun-in ID 指的是bundle的唯一标识，在实际项目中可以采用类似java包名组织策略来保证唯一性（对应manifest.mf中的Bundle-SymbolicName）；

Plugin-in version 指的是bundle的版本（对应manifest.mf中的Bundle-version）；

Plugin-in name 指的是bundle更具意义的名称（对应manifest.mf中的Bundle-Name）；

Plgun-in provider 指的是bundle的提供商（对应manifest.mf中的Bundle-Vendor）；

Execution enviroment指的是bundle运行所需环境（对应manifest.mf中的Bundle-RequiredExecutionEnvironment）；

Acitivator部分是可选项，如果勾选并输入类名，在工程建立时eclipse会自动创建一个实现BundleActivator接口的类。

|  |
| --- |
|  |

### 完成Activator类

|  |
| --- |
| **package** 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("HelloWorldBundle started!!!");  }  /\*  \* (non-Javadoc)  \* @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)  \*/  **public** **void** stop(BundleContext context) **throws** Exception {  System.*out*.println("HelloWorldBundle stopped!!!");  }  } |

### 运行

创建一个HelloWorld运行的osgi配置（参照5.1）

将HelloWorld Bundle的Auto-start 属性设置为false，

选中target plateform，依次选择deselect All 、add required bundles

最后点击run 并输入”ss”结果如下：

|  |
| --- |
|  |

可以看到，HelloWorld Bundle已经被安装并且完成了解析，但是没有启动。

我们在”osgi>“输入符下输入”start 1“启动 HelloWorld Bundle，并输入”ss“查看bundle状态，如下：

|  |
| --- |
| osgi> start 1  HelloWorldBundle started!!!  osgi> ss  Framework is launched.  id State Bundle  0 ACTIVE org.eclipse.osgi\_3.5.1.R35x\_v20090827  1 ACTIVE HelloWorld\_1.0.0  osgi> |

可以看出 HelloWorld bundle 的状态由lazy状态转为active状态。

依次输入”stop 1“ “ss”结果如下

|  |
| --- |
| osgi> stop 1  HelloWorldBundle stopped!!!  osgi> ss  Framework is launched.  id State Bundle  0 ACTIVE org.eclipse.osgi\_3.5.1.R35x\_v20090827  1 RESOLVED HelloWorld\_1.0.0  osgi> |

可看出HelloWorld状态由active🡪resolved。

## 从外部启动bundle

前面(5.1、5.2)，我们都是通过Eclipse启动我们的Bundle。但是，在有些时候，我们希望自己来控制OSGi的容器的启动，并且在OSGi的容器外部获取OSGi的服务，甚至是把OSGi的容器内嵌到我们的应用之中。下面我们就来看一下如何把Equinox嵌入到应用中，由应用来启动Equinox、获取OSGi的服务，以及加载OSGi容器中的其他插件的类。并且也会演示OSGi容器中的插件如何加载OSGi容器外的类的方法。

### 通过命令来启动equinox

常见的脚本：

|  |
| --- |
| java-jar plugins/org.eclipse.osgi\_3.4.3.R34x\_v20081215-1030.jar -configuration configuration -console |

configuration目录下放置一个config.ini，在此config.ini中通过 osgi.bundles=来配置要加载和启动的插件，例如osgi.bundles=example.jar@start

### 程序中启动equinox

通过查看Equinox的代码，会看到调用上面的org.eclipse.osgi.jar后执行的其实是EclipeStarter类中的静态run方法，因此只须在外部传入合适的参数，并调用此run方法即可完成Equinox的启动，在程序中启动Equinox，通常希望做到的是能够指定config.ini 的配置信息及插件（bundel）的位置，而不是由Equinox去决定。如果不进行设置，默认情况下EclipseStarter将会在工作路径下产生 configuration，并以该configuration目录下的config.ini作为Equinox启动的配置。对于 osgi.bundles配置的bundle的路径，默认则为当前EclipseStarter代码所在的目录，例如上面的命令行，Equinox在启动时就会从plugins目录中去加载插件，这通常是无法满足在程序中启动Equinox的需求的。如果想自定义Equinox启动的配置信息，而不是通过加载指定的configuration中的config.ini，那么可以在程序中调用 FrameworkProperties.setProperty来设置启动Equinox的配置信息。如希望指定osgi.bundles中指定加载的bundle的相对路径，那么可以在Equinox启动的配置信息中增加osgi.syspath的设定：FrameworkProperties.setProperty（"osgi.syspath"，你希望指定的bundle所在的路径）。Equinox启动的配置信息还有很多种，有具体需要的话可以查看EclipseStarter中processCommandLine的方法。通过这样的方式，就可以启动Equinox：EclipseStarter.run（new String[]{"-console"},null）；按照上面的方式就可以实现在外部程序中启动equinox了。

### OSGI容器外获取获取OSGi服务

OSGi通过BundleContext来获取OSGi服务，因此想在OSGi容器外获取OSGi服务，首要的问题就是要在OSGi容器外获取到BundleContext，EclipseStarter中提供了一个getSystemBundle- Context的方法，通过这个方法可以轻松拿到BundleContext，而通过BundleContext则可以轻易拿到OSGi服务的实例。不过这个时候要注意的是，如果想执行这个OSGi 服务实例的方法，还是不太好做的，因为容器外的classloader和OSGi服务实例的class所在的classloader可能并不相同，因此不太好按照java对象的方式直接去调用，更靠谱的是通过反射去调用。

如果想在容器外获取到OSGi容器里插件的class，一个可选的做法是通过BundleContext获取到Bundle，然后通过Bundle来加载 class，采用这样的方法加载的class就可以保证是相同的。否则会出现容器外的一个A.class不等于容器里插件的A.class，其中原因对于稍微知道java classloader机制的人都是理解的。

按照上面的说法，一个简单的启动Equinox及与OSGi容器交互的类可以这么写：

1. /\*\*
2. \* 启动并运行equinox容器
3. \*/
4. **public** **static** **void** start() **throws** Exception{
5. // 根据要加载的bundle组装出类似a.jar@start,b.jar@3:start这样格式的
6. osgibundles字符串来
7. String osgiBundles="";
8. // 配置Equinox的启动
9. FrameworkProperties.setProperty("osgi.noShutdown", "true");
10. FrameworkProperties.setProperty("eclipse.ignoreApp", "true");
11. FrameworkProperties.setProperty("osgi.bundles.defaultStartLevel", "4");
12. FrameworkProperties.setProperty("osgi.bundles",
13. osgiBundlesBuilder.toString());
14. // 根据需要设置bundle所在的路径
15. String bundlePath="";
16. // 指定要加载的plugins所在的目录
17. FrameworkProperties.setProperty("osgi.syspath", bundlePath);
18. // 调用EclipseStarter，完成容器的启动，指定configuration目录
19. EclipseStarter.run(**new** String[]{"-configuration","configuration",
20. "-console"}, **null**);
21. // 通过EclipeStarter获得BundleContext
22. context=EclipseStarter.getSystemBundleContext();
23. }
25. /\*\*
26. \* 停止equinox容器
27. \*/
28. **public** **static** **void** stop(){
29. **try** {
30. EclipseStarter.shutdown();
31. context=**null**;
32. }
33. **catch** (Exception e) {
34. System.err.println("停止equinox容器时出现错误:"+e);
35. e.printStackTrace();
36. }
37. }
39. /\*\*
40. \* 从equinox容器中获取OSGi服务instance   还可以基于此进一步处理多服务接口实现的状况
41. \*
42. \* @param serviceName 服务名称（完整接口类名）
43. \*
44. \* @return Object 当找不到对应的服务时返回null
45. \*/
46. **public** **static** Object getOSGiService(String serviceName){
47. ServiceReference serviceRef=context.getServiceReference
48. (serviceName);
49. **if**(serviceRef==**null**)
50. **return** **null**;
51. **return** context.getService(serviceRef);
52. }
54. /\*\*
55. \* 获取OSGi容器中插件的类
56. \*/
57. **public** **static** Class<?> getBundleClass(String bundleName,
58. String className) **throws** Exception{
59. Bundle[] bundles=context.getBundles();
60. **for** (**int** i = 0; i < bundles.length; i++) {
61. **if**(bundleName.equalsIgnoreCase(bundles[i].getSymbolicName())){
62. **return** bundles[i].loadClass(className);
63. }
64. }

### OSGi容器内的插件要加载OSGi容器外的类

在实现了OSGi容器外与OSGi交互之后，通常会同时产生另外一个需求，就是在OSGi容器内的插件要加载OSGi容器外的类，例如OSGi容器内提供了一个mvc框架，而Action类则在OSGi容器外由其他的容器负责加载，那么这个时候就会产生这个需求了，为了做到这点，有一个比较简单的解决方法，就是编写一个Bundle，在该Bundle中放置一个允许设置外部ClassLoader的OSGi服务，例如：

1. **public** **class** ClassLoaderService{
2. **public** **void** setClassLoader(ClassLoader classloader);
3. }

基于上面的方法，在外部启动Equinox的类中去反射执行ClassLoaderService这个OSGi服务的setClassLoader方法，将外部的classloader设置进来，然后在OSGi容器的插件中要加载OSGi容器外的类的时候就调用下面这个ClassLoaderService去完成类的加载。

基于以上说的这些方法，基本上可以较好地实现OSGi容器与其他容器的结合，例如在tomcat中启动OSGi等，或者在我们自身的应用中来控制OSGi的容器。

# OSGI 与 servlet 容器结合

参见：<http://www.cnblogs.com/skyme/archive/2012/07/10/2583952.html>

<http://www.ibm.com/developerworks/cn/web/0907_osgiweb_liuqing/>

## 在OSGI框架中嵌入Http服务器/web应用/servlet容器

其实就是把servlet 容器做成一个bundle，这种方式不常使用。

嵌入servlet容器至Osgi框架实现web应用有三种方式

准备  
首先从从Equinox下载完整的[eclipse-equinox-<buildno>.zip](http://archive.eclipse.org/eclipse/equinox/drops/R-3.3-200706251500/index.php) 二进制文件框架。本文中的示例代码在版本3.6下进行测试。解压缩下载的文件拷贝以下文件：  
      org.eclipse.osgi\_3.6.1.R36x\_v20100806  
      org.eclipse.osgi.services\_3.1.200.v20070605  
      org.eclipse.equinox.http\_1.0.100.v20070423  
      javax.servlet\_2.4.0.v200706111738  
      org.mortbay.jetty\_5.1.11.v200706111724  
      org.eclipse.equinox.http.servlet\_1.0.0.v2007060  
      org.apache.commons.logging\_1.0.4.v201005080501  
  
然后在当前目录下创建configuration目录，在该目录下创建文件config.ini，内容为：  
osgi.noShutdown=true  
eclipse.ignoreApp=true  
osgi.clean=true  
osgi.bundles.defaultStartLevel=4  
osgi.bundles=reference\:file\:org.eclipse.osgi.services\_3.1.200.v20070605.jar@start,reference\:file\:org.eclipse.equinox.http\_1.0.100.v20070423.jar@start,reference\:file\:javax.servlet\_2.4.0.v200706111738.jar@start,reference\:file\:org.mortbay.jetty\_5.1.11.v200706111724.jar@start,reference\:file\:org.eclipse.equinox.http.servlet\_1.0.0.v20070606.jar@start,reference\:file\:org.apache.commons.logging\_1.0.4.v201005080501.jar@start

### 服务注册方式

为了演示，这里创建一个bundle，Activator 定义为  
public class Activator implements BundleActivator, ServiceListener {  
    private ServiceReference ref;  
    private Servlet servlet;  
    private static BundleContext context;  
public void start(BundleContext bundleContext) throws Exception {  
        context = bundleContext;  
        servlet = new LoginServlet(context);  
        registerServlet();  
        context.addServiceListener(this,  
                "(objectClass=" + HttpService.class.getName() + ")");  
    }  
public void stop(BundleContext bundleContext) throws Exception {  
        try {  
            unregisterServlet();  
        } catch (Throwable t) {  
            t.printStackTrace();  
        }  
  
        servlet = null;  
        context = null;  
        ref = null;  
    }  
     
    private void registerServlet() {  
        if (ref == null) {  
            ref = context.getServiceReference(HttpService.class.getName());  
        }  
  
        if (ref != null) {  
            try {  
                HttpService http = (HttpService) context.getService(ref);  
                http.registerServlet("/login", servlet, null, null);  
                http.registerResources("/page", "page", null);  
                System.out.println("已启动用户登录验证web模块，请通过/page/login.htm访问");  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
     
    private void unregisterServlet() {  
        if (ref != null) {  
            try {  
                HttpService http = (HttpService) context.getService(ref);  
                http.unregister("/login");  
                http.unregister("/page");  
                System.out.println("已卸载用户登录验证web模块！");  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    @Override  
    public void serviceChanged(ServiceEvent event) {  
        switch (event.getType()) {  
        case ServiceEvent.REGISTERED:  
            registerServlet();  
            break;  
  
        case ServiceEvent.UNREGISTERING:  
            unregisterServlet();  
            break;  
        }  
    }  
}  
  
用命令启动osgi：java -jar org.eclipse.osgi\_3.6.1.R36x\_v20100806.jar -console，  
然后install file:bundle安装编写的bundle，最后通过浏览器输入<http://localhost/page/login.htm>即可访问编写的login.htm文件  
以上使用的默认的80端口，可以使用这个参数改变端口：-Dorg.osgi.service.http.port=8080

### 声明服务DS方式

参见：

服务声明是对osgi框架做的优化旨在降级启动速度和减少内存占用，能够让框架自动完成服务的发布和引用以及生命周期管理，简化以上操作，真正做到Component+Service的开发方式。Component和Service从定义上来看是差不多的，任何一个普通的Java对象都可以通过在配置文件中定义成Component。  
  
同上假如要用DS方式实现一个用户登录验证的服务：  
public class DBValidatorImpl implements Validator{  
   public boolean validate(String username, String password) throws Exception {  
    
   }  
}  
只要去掉原来的BundleActivator的实现类Activator即可。

**服务发布**  
  
创建目录OSGI-INF，然后在该目录下创建文件component.xml文件，内容为：  
<component name="DBValidator">  
    <implementation class="org.riawork.demo.service.user.impl.DBValidatorImpl"/>  
    <service>  
        <provide interface="org.riawork.demo.service.user.Validator"/>  
    </service>  
</component>  
然后在MANIFEST.MF文件中引用这个文件就可以了：  
Service-Component: OSGI-INF/components.xml

**服务引用**  
<component name="LoginServlet">  
    <implementation class="org.riawork.demo.web.servlet.LoginServlet"/>  
    <reference name="ValidatorService" interface="org.riawork.demo.service.user.Validator" bind="setValidator" unbind="unsetValidator" policy="dynamic" cardinality="0..1"/>  
</component>  
同时install file:org.eclipse.equinox.ds\_1.2.1.R36x\_v20100803.jar 服务声明的jar和方法  
public void setValidator(Validator validator){  
        this.validator=validator;  
    }  
     
    public void unsetValidator(Validator validator){  
        if(this.validator!=validator)  
            return;  
        this.validator=null;  
    }  
这样当服务可用时DS会自动注入。  
Service的引用策略

Cardinality

由于在OSGI框架中服务的提供通常都是提供接口的方式，那么就会产生系统中提供多个实现接口的服务的问题，例如在系统中可能同时有三个验证服务的实现，那么在引用服务时其实是可以同时引用这三个服务的，在DS中可以通过Cardinality的定义来只引用一个或多个服务。  
Cardinality有四种策略：  
􀂄 0..1 最多只引用其中的一个服务，但系统也可以不存在可用的服务，象Log服务就很适合这种情况，即使系统中不存在Log服务组件应该也同样保持可用。

引用策略

引用策略是指在引用服务时可采用static和dynamic两种策略方式，static策略是默认的方式，在Component启动后，如果引用的service发生变动时，整个Component会重启；而采用dynamic策略的情况下，在Component启动后，如果引用的service发生变动，并不会导致整个Component重启，而只会调用其中的unbind和bind方法。  
要使用引用策略只需要在之前的service引用配置中增加policy属性的配置。  
􀂄 1..1 引用的服务必须至少有一个可用的（默认）。  
􀂄 0..n 引用0到多个可用的服务。  
􀂄 1..n 引用的服务必须存在，使用时可使用多个服务。  
设置Cardinality策略只需要在之前的service引用的配置上增加Cardinality属性的配置即可。

属性过滤

在Service Layer中是通过填充属性到Dictionary对象中来实现根据属性过滤获取服务，而在DS中同样可以通过在配置中定义需要过滤的属性：  
<reference name=”HTTPService”  
interface=”org.osgi.service.http.HttpService”  
bind=”setHttpService”  
unbind=”unsetHttpService”  
cardinality=”0..n”  
policy=”dynamic”  
target=”(component.version=1.0)”/>  
</component>  
  
在DS中可以通过property和properties两种方式来配置component的属性：  
<?xml version="1.0" encoding="UTF-8"?>  
<component name="osgi.DemoComponent">  
<implementation class="org.riawork.opendoc.osgi.DemoComponent"/>  
<property name=”component.version”>1.0</property>  
<property name=”component.canuse” type=”Boolean”>true</property>  
<properties entry=”OSGI-INF/config.properties”/>  
</component>  
property用于配置单个属性，在property中可以指定property值的类型，而properties则用于引用bundle中的配置文件。

### 基于osgi扩展点方式

参见：<http://www.ibm.com/developerworks/cn/web/0907_osgiweb_liuqing/> ()

同上创建bundle，修改plugin.xml为  
<plugin>  
   <extension-point id="servlets" name="HttpService servlets" schema="schema/servlets.exsd"/>  
   <extension-point id="resources" name="HttpService resources" schema="schema/resources.exsd"/>  
<extension  
id="loginServlet" point="org.eclipse.equinox.http.registry.servlets">  
<servlet alias="/login" class="com.javaworld.sample.osgi.web.declarative.webServlet">  
</servlet>  
</extension>  
<extension id="resource" point="org.eclipse.equinox.http.registry.resources">  
<resource alias="/login.html" base-name="/page/login.htm" />  
</extension>  
</plugin>  
这里使用了两个扩展点：一个是servlet，扩展org.eclipse.equinox.http.registry.servlets；另一个是资源，扩展org.eclipse.equinox.http.registry.resources。

## 在Servlet容器中嵌入OSGI框架

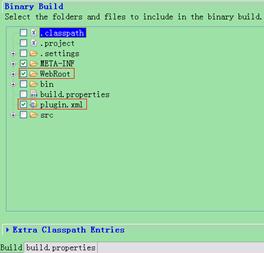
这种开发方式和刚才不同，这种开发方式是将 Equinox 框架以及项目的 bundle 做成一个 War 文件部署到 Servlet 容器中，和以前我们做普通的 Web 开发一样，当做一个普通的 Web 应用运行。在这种方式下，使用了一种叫做 Servlet Bridge(Servlet 桥 ) 的技术，用户对该 Web 应用的 http 请求被 servlet 容器转发给该 Web 应用，Equinox 框架运行在该 Web 应用中，统一处理对于该应用的请求，并将请求转发给对应的 bundle。

采用这种方式开发，一般情况下首先需要到 Equinox 站点上 [下载](http://www.eclipse.org/equinox/server/downloads/bridge.war)bridge.war 文件，这个文件是 2007 年 4 月的版本。我们在前面创建工程的时候没有让大家选择创建 activator, 原因就是在于 activator 引用的有些类在这个版本的 bridge.war 中不存在。此外：在我们的 sample 项目中，需要支持 JSP，所以需要几个额外的 bundle。为了大家的方便，我另外做了一个 bridge.war, 里面包含这些需要的 bundle。大家可以直接源代码下载部分 [下载](http://www.ibm.com/developerworks/cn/web/0907_osgiweb_liuqing/#download)。

下载该 war 文件成功后，将其导入到我们的 Eclipse 工作空间中来。

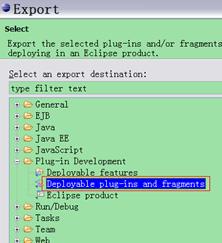
接下来，我们将刚才创建的 sample 项目导出为一个 bundle：打开我们的 sample 项目的 build.proerties 文件，将我们需要的 WebRoot 文件夹及其子文件夹以及 plugin.xml 都选中。

### 图 1. 修改 build 属性



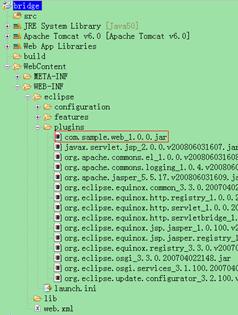
然后将我们的 sample 项目右键导出为一个 Deployable plug-ins and fragments

### 图 2. 导出项目为 bundle



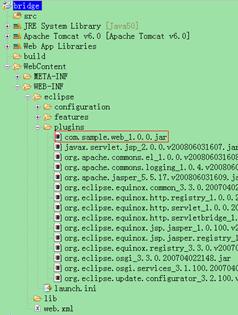
其他按照缺省设置即可，这时候大家会看到 Eclipse 会在你指定的目录下生成 plugins /com.sample.web\_1.0.0.jar 文件，将该 jar 文件放到刚才导入的 bridge 项目的 plugins 文件夹下。

### 图 3. 工作空间示意图



在这种方式下，我们不需要去运行我们的 sample 项目 , 而是运行我们的 bridge 项目，选中 bridge 项目 , 右键 ->run as->run on server 将 bridge 项目运行到我们配置的 Tomcat 中去。由于我们并没有设置让我们的 sample 项目默认自动启动，所以大家在此处需要启动我们的 sample 项目的 bundle.

### 图 4. 启动 sample bundle



启动完成后，大家来访问一下如下的 URL，看看有什么现象。考虑一下在这种开发方式下，同样的 bundle，URL 和刚才第一种方式下的有什么不同？

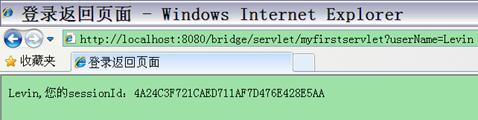
http://localhost:8080/bridge/servlet/myfirstservlet?userName=Levin

http://localhost:8080/bridge/jsp/index.jsp

http://localhost:8080/bridge/images/1.jpg

如果一切正常的话，在访问 Servlet 的时候您将会看到类似下图的运行效果。Tocmat 的默认端口是 8080，而图 9 中的 Jetty 默认端口是 80。并且，由于所有的 bundle 都是部署在 bridge 项目中，所以访问的 URL 也变成了以 http://localhost:8080/bridge/ 为基路径了。

### 图 5. Servlet 运行图



在实际的开发过程中，如果每次都需要将我们的项目导出为 jar 文件放置到 bridge 中，开发效率是非常低下的。上述的步骤其实只适合在项目开发完成以后，实际部署到服务器上的时候才需要。在我们在开发过程中我们可以采取以下办法来简化开发：

打开 bridge 项目的 WebContent/Web-INF/eclipse/configuration/config.ini 文件，在其 osgi.bundles 的配置中增加对我们 sample project 的引用，例如：

reference\:file\:D:/eclipse-jee-ganymede-SR2-win32/workspace/com.sample.web@start

这时如果我们选中 bridge 项目 , 右键 ->run as->run on server 将 bridge 项目运行到我们配置的 Tomcat 中去的话，bridge 会根据 config.ini 中的配置，自动将我们的 sample project 也当做一个 bundle 启动。

# 补充

## MANIFIS.MF的作用

### 一般的MANIFIS.MF

|  |
| --- |
| Manifest-Version: 1.0  Main-Class: windows.VideoWindow  Class-Path: lib\jetty-6.1.26.jar lib\aspectjweaver-1.6.12.jar  Manifest-Version 版本号  Main-Class 包含main函数的类，程序的入口  Class-Path 用到的一些包  要点：  要点1：每行的“:”(冒号)用来分隔键值对，冒号后边一定要跟一个空格！！！  也就是说务必写为:  Main-Class:(空格)YouClassName  Class-Path:(空格)Class1.Jar  要点2：Class-Path里边的内容用空格分隔而不是逗号或者分号  也就是说务必写成:  Class-Path:(空格)Class1.jar(空格)Class2.jar(空格)Class3.Jar  要点3:有的文章说MF文件每行不能超过七十多的字符，  另外：  关于java -classpath a.jar;b.jar; class test.Test 执行时. a.jar;b.jar; 后面都必须有分号.  cmd -> java -jar your.jar ，使用-jar参数后, 系统的Classpath 环境变量不再起作用. 虚拟机将去MANIFEST.MF中的Class-Path 下找相关的包. |

MANIFEST.MF 文件中奇怪的分行和空格要求：

### Bundle的MANIFIS.MF

待补充

## Java中的private、protected、public和default

（1）对于public修饰符，它具有最大的访问权限，可以访问任何一个在CLASSPATH下的类、接口、异常等。它往往用于对外的情况，也就是对象或类对外的一种接口的形式。

（2）对于protected修饰符，它主要的作用就是用来保护子类的。它的含义在于子类可以用它修饰的成员，其他的不可以，它相当于传递给子类的一种继承的东西。

（3）对于default来说，有点的时候也成为friendly（友员），它是针对本包访问而设计的，任何处于本包下的类、接口、异常等，都可以相互访问，即使是父类没有用protected修饰的成员也可以。

（4）对于private来说，它的访问权限仅限于类的内部，是一种封装的体现，例如，大多数的成员变量都是修饰符为private的，它们不希望被其他任何外部的类访问。

下表为Java访问控制符的含义和使用情况

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 类内部 | 本包 | 子类 | 外部包 |
| public | √ | √ | √ | √ |
| protected | √ | √ | √ | × |
| default | √ | √ | × | × |
| private | √ | × | × | × |

**注意：Java的访问控制是停留在编译层的，也就是它不会在.class文件中留下任何的痕迹，只在编译的时候进行访问控制的检查。其实，通过反射的手段，是可以访问任何包下任何类中的成员，例如，访问类的私有成员也是可能的。**

区别：

（1）public：可以被所有其他类所访问。

（2）private：只能被自己访问和修改。

（3）protected：自身，子类及同一个包中类可以访问。

（4）default（默认）：同一包中的类可以访问，声明时没有加修饰符，认为是friendly。

# 标题1

## 标题1.1

### 标题1.1.1

#### 标题1.1.1.1

##### 标题1.1.1.1.1

