

引领 OSGi 国内研究和普及

OSGi原理与最佳实践

精选版



林昊 曾宪杰 著

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

InfoQ中文站
www.infoq.com/cn

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

<http://infoq.com/cn/minibooks/osgi-best-practice>

QCon

旧金山 • 伦敦 • 北京 • 东京

QCon全球企业开发大会（北京站）
每年四月，我们一同期待！

大会整体评价

总结而言，这次大会是相当成功的，一次成功的大会不能缺少的两个要素：知名的嘉宾和精彩的Topics，无疑QCon北京很好地把握了这两个要素。

----- 淘宝网架构师、OSGi China User Group的负责人BlueDavy（林昊）

The first QCon Beijing conference concluded successfully, I was amazed by the overwhelming attendance. The Beijing debut is probably the biggest QCon gathering globally. I was also amazed by how efficient infoQ run the conference.

----- FreeWheel创始人/CTO、DoubleClick前CTO（于晶纯）

出乎很多人意料的是，本次大会门票竟然全部提前售罄。即便如此，到会议现场购票的人仍然络绎不绝。究其原因，前沿的技术话题和高水平的讲师是最大的吸引力。

----- Joyent中国区技术经理
《程序员》杂志前主编（郑柯）

往届讲师及评价



Martin Fowler

敏捷宣言缔造者，Thoughtworks首席科学家

来自CnBlogs网友Daniel的评价：Martin Fowler的两个演讲，关于DSL的和Ruby的，坚定了我以后的技术方向：Polyglot+Polyparadigm。



Randy Shoup

eBay高级架构师

来自手机之家架构师徐超前的评价：非常不错，都是实战的总结。InfoQ上发表过《可伸缩性最佳实践：来自eBay的经验》。Randy在不断用心的数据补充和完善该话题。



洪强宁

豆瓣网首席架构师

来自DBAnotes.net博主冯大辉的评价：第一时间看到豆瓣@hongqn大侠在QCon上做分享的PPT，击节赞叹！这是今年看到的最好的一份PPT。极有参考价值。如果非要形容一下的话，那就是：牛！



岳旭强

淘宝网资深架构师

来自网友Rocky_rup的评价：岳旭强在介绍淘宝网应用Java的历程汇总，激起我的共鸣，我在心中默默感叹：原来大家在实践中演化系统架构的经历是如此的相似！



Dylan

Dojo Tookit创始人



程立

支付宝首席架构师

.....



www.QConBeijing.com



QCon@cn.infoq.com

前言

Java 7 的发布日期临近，模块化是 Java 7 中最重要的特性之一。在 Java 语言级对模块化提供支持之前，OSGi 已经是业界中最知名的 Java 模块化规范。OSGi 联盟成立于 1999 年，发展到今天已经得到了众多企业、厂商、开源组织的支持，尤其当主流的 Java 应用服务器（Oracle 的 Weblogic、IBM 的 Websphere 及 Sun 的 Glassfish 等）都采用 OSGi 时，OSGi 作为 Java 模块化标准已成为事实。掌握 OSGi 是实现模块化 Java 应用的必备技能，在将来甚至会成为 Java 语言中必须学习的技能之一，就像现在 Java 中的泛型一样，而动态化也是 OSGi 的另一特性。OSGi 对于动态化的支持能够帮助开发者更好地实现“即插即用”、热部署及“即删即无”的系统。

《OSGi 原理与最佳实践》作为一本早于同类技术英文书而编写的 OSGi 中文书籍，旨在为希望实现模块化、动态化 Java 系统的架构师和开发工程师提供 OSGi 入门知识，同时也为希望深入掌握 OSGi 的架构师、开发工程师提供 OSGi 知识的深入讲解。原书内容从 OSGi 的简介开始，到 OSGi 框架的使用，再到 OSGi 规范的掌握，最后到 OSGi 框架的实现分析，阐述了基于 OSGi 编写模块化、动态化的 Java 系统须要掌握的知识体系，希望此书能给读者带来一次愉快的 OSGi 之旅。

本迷你书作为《OSGi 原理与最佳实践（精选版）》，节选了原书中的第二、三章，结合简单例子及经典的 PetStore 对 OSGi 框架的使用进行了介绍。

本迷你书中所包含的实例，由于篇幅关系，书中仅列出了代码的片断，如需完整代码，请到 <http://china.osgiusers.org> 中下载。

目录

前 言	I
第 1 章 OSGi 框架简介	1
1.1 Equinox	1
1.1.1 简介	1
1.1.2 环境搭建	1
1.1.3 HelloWorld	4
1.1.4 开发传统类型的应用	11
1.1.5 从外部启动 Equinox	27
1.2 Felix	29
1.2.1 简介	29
1.2.2 环境搭建	29
1.2.3 应用的部署	29
1.2.4 在 Eclipse 中调试 Felix	30
1.3 Spring-DM	35
1.3.1 简介	35
1.3.2 环境搭建	35
1.3.3 HelloWorld	39
1.3.4 Web 版 HelloWorld	42
第 2 章 基于 Spring-DM 实现 Petstore	47
2.1 “即插即用”的 Petstore	47
2.1.1 Petstore 的功能需求	47
2.1.2 OSGi 框架的功能和设计思想	48
2.1.3 Petstore 的设计	50
2.2 新一代 Petstore 的实现	60
2.2.1 环境准备	60
2.2.2 Utils 模块	61
2.2.3 Bootstrap 模块	63
2.2.4 ProductDal 模块	67
2.2.5 ShoppingCartDal 模块	68
2.2.6 ProductList 模块	69
2.2.7 ShoppingCart 模块	73
2.2.8 ProductManagement 模块	73
2.3 部署	73
2.4 Petstore 的扩展	73

第 1 章 OSGi 框架简介

前面我们对 OSGi 及其现状做了一个大体的介绍。下面一起来看我们常用的 OSGi 框架，也会结合具体的实例让大家来使用这些框架。

1.1 Equinox

1.1.1 简介

首先来看 Equinox。Equinox 是 Eclipse 中的项目，并作为 OSGi R4 RI 而知名，由于 Equinox 有 Eclipse IDE 这个成功案例，反映出了 Equinox 作为 OSGi 框架的优势。Equinox 目前是随着 Eclipse 版本而发布的，同时，它也提供独立的下载，在独立的下载页面中可以下载到 Equinox 对于 OSGi R4 的所有实现，以及 Equinox 扩展 OSGi R4 而提供的 Bundle。

Equinox 开发小组由 IBM 的 Jeff 领衔，开发状态非常活跃，从其开发者 maillist 可以看出，讨论非常热闹，大家感兴趣的话可以申请加入开发者 maillist：

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

想了解更多的 Equinox 信息请参看：

官方站 <http://www.eclipse.org/equinox>

中文站 <http://china.osgiusers.org>

下面介绍如何搭建开发环境。

1.1.2 环境搭建

既然是基于 Equinox 开发，我们首先要下载 Equinox。Equinox 是 Eclipse 的工程，Eclipse 3.1 之后的版本都是通过它来启动的，如果使用的是 Eclipse 3.1 之后的版本，Eclipse 本身就已经包含了 Equinox，可在 Eclipse 的 plugins 目录下看到类似 org.eclipse.osgi_3.4.3.R34x_v20081215-1030.jar 这样的文件（不同版本的 Eclipse 下对应的版本号和日期会有所不同），它其实就是 Equinox 的 OSGi R4 Core 的实现。如果采用的不是 Eclipse 3.1 之后的版本，建议下载一个 Eclipse 3.1 之后的版本。

下面我们来检查环境，首先启动 Eclipse。

- 第一步，打开 Run Configurations 对话框（见图 1-1）。

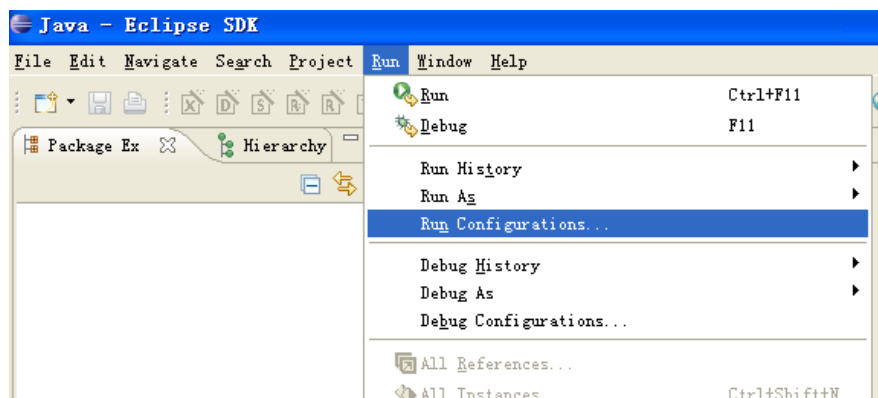


图 1-1 运行配置菜单

- 第二步，创建 OSGi Framework 类型的新的运行配置（见图 1-2）。

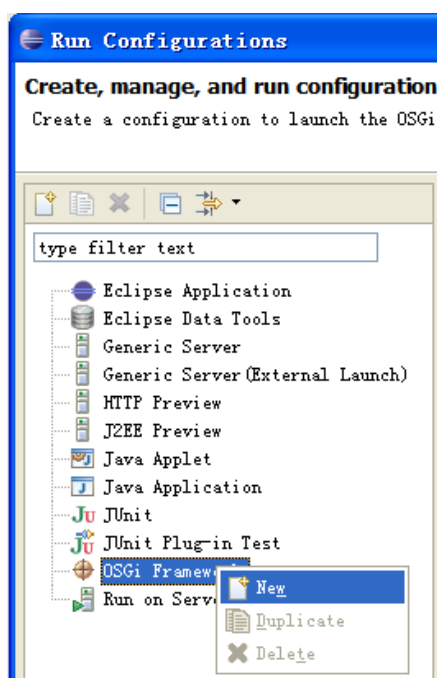


图 1-2 创建运行配置

- 第三步，显示所有的 Bundles（见图 1-3）。

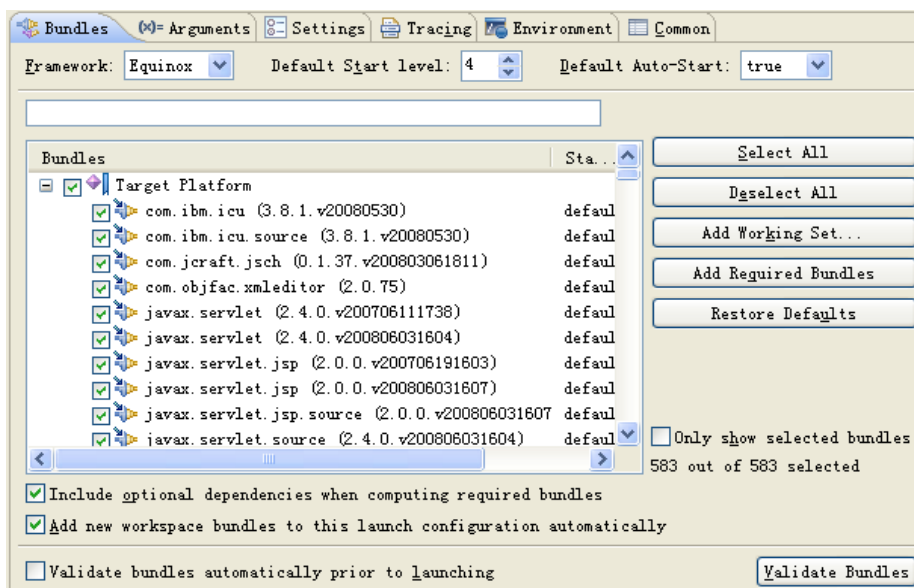


图 1-3 显示 Bundles

- 第四步，取消对 Bundles 的选择（见图 1-4）。

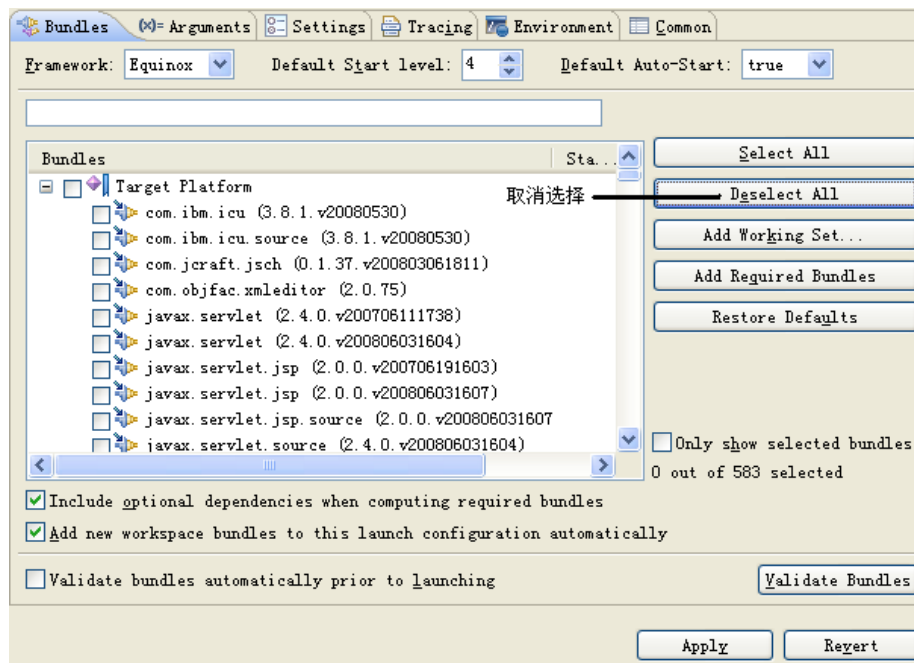


图 1-4 取消对 Bundles 的选择

- 第五步，选择 org.eclipse.osgi 这个 Bundle（见图 1-5）。

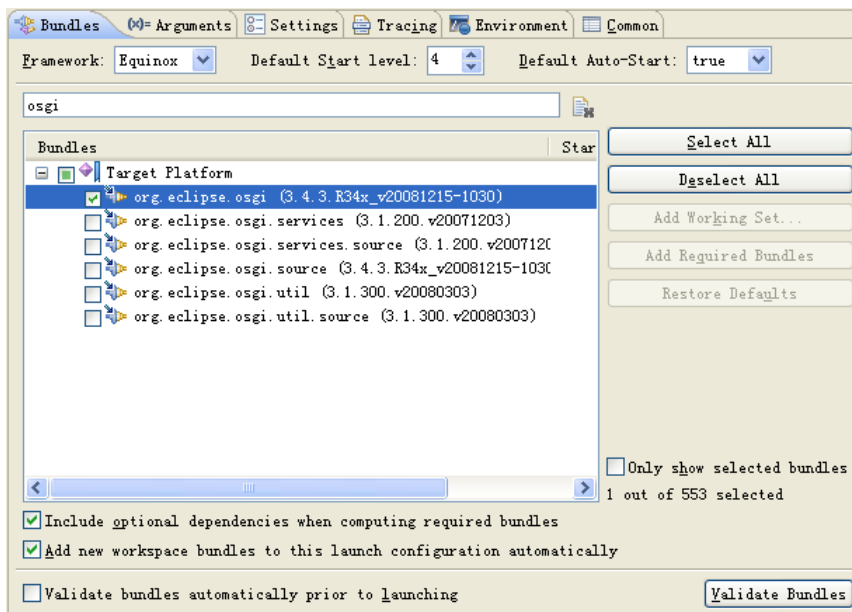


图 1-5 选中 Bundle 后的显示

➤ 第六步，运行。

点击 Run 按钮，如果 Console 中出现“osgi>”并且没有错误信息，说明环境已经正常了。我们可以在 osgi>提示符后输入 ss，然后回车。将会看到如图 1-6 所示的界面。

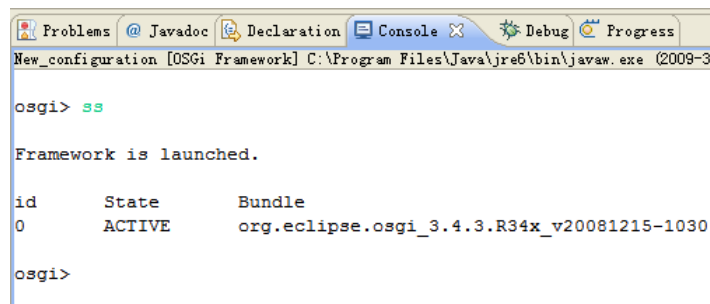


图 1-6 运行结果显示

环境已经准备好了。下面来开始我们的第一个 OSGi 的应用。

1.1.3 HelloWorld

在这一节中，我们将完成一个 HelloWorld 的例子。之前大家应该看到过 Java 语言、C 语言或 C++ 等语言中的 HelloWorld，那些 HelloWorld 程序都是在运行后输出一个“Hello World”，然后就结束了程序。我们今天的 HelloWorld 例子程序主要的功能是在启动和停止一个 Bundle 的时候来做些事情（输出信息）。下面我们就一步一步完成第一个 Bundle。

➤ 第一步，创建 Bundle 工程。

- 在 Eclipse 中创建一个 Plug-in 工程（见图 1-7）。

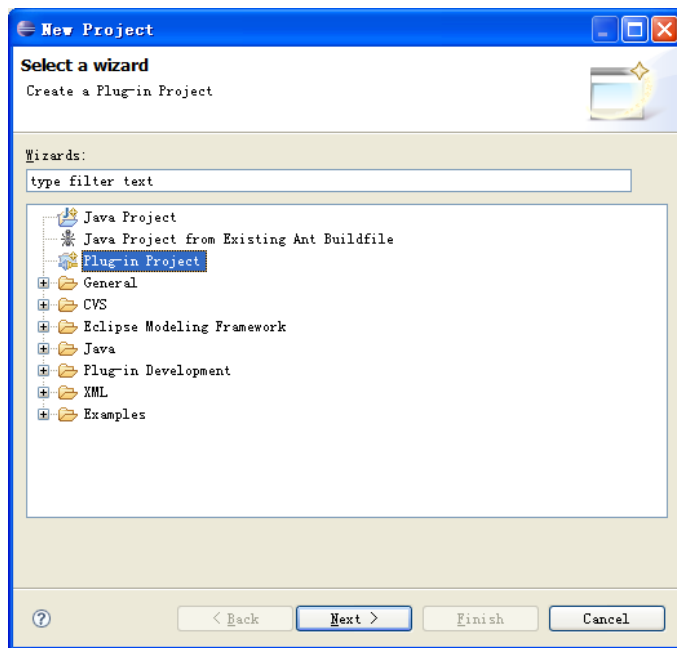


图 1-7 创建插件工程

- 输入工程相关的信息，这里和创建普通的 Java 工程唯一的不同点就是要选择 this plug-in is targeted to run with，在这里选择 an OSGi framework 的 standard 选项，也就是说建立一个标准的 OSGi Bundle 工程（见图 1-8）。
- 输入 Bundle 的相关元数据信息（见图 1-9）。
 - Plug-in ID 指的是 Bundle 的唯一标识，在实际的项目中可以采用类似 java 的包名组织策略来保证标识的唯一性；
 - Plug-in Version 指的是 Bundle 的版本；
 - Plug-in Name 指的是 Bundle 的更具有意义的名称；
 - Plug-in Provider 指的是 Bundle 的提供商；
 - Execution Environment 指的是 Bundle 运行需要的环境；
 - 剩下的最关键的就是 Activator 部分了，这里填入自己的一个类名就可以了，在工程建立时 Eclipse 会自动建立这个类。

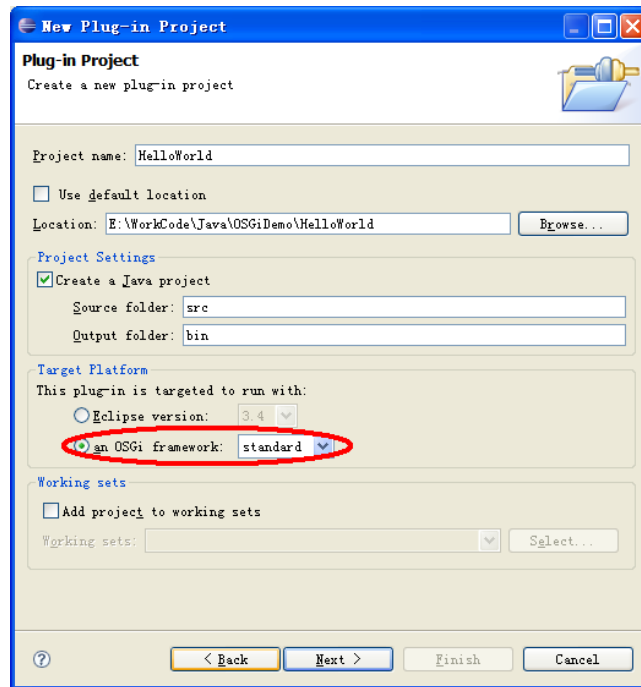


图 1-8 插件工程设置

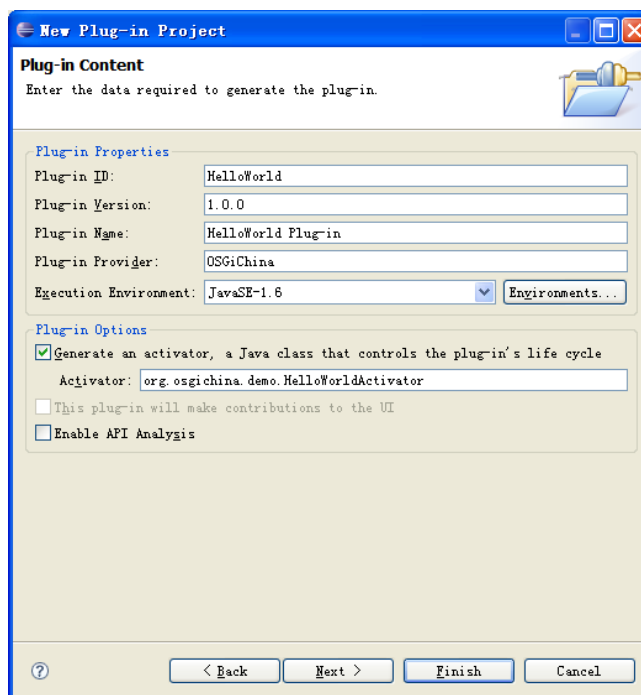


图 1-9 插件信息设置

- 完成了 Bundle 工程的创建后，在 Package 视图中就可以看到如图 1-10 这样的视图，表明工程创建成功了。

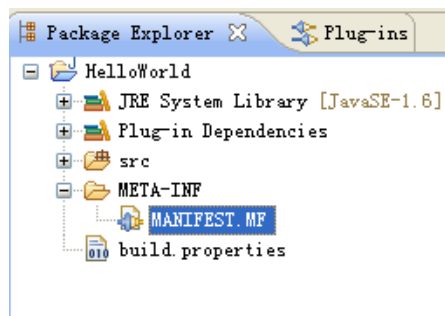


图 1-10 插件工程内容

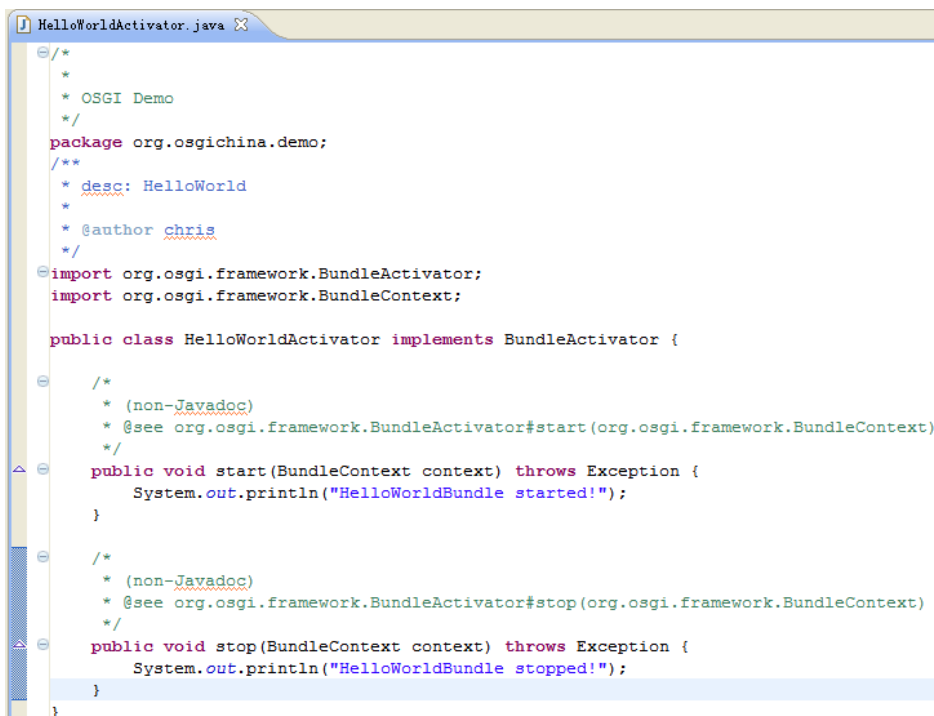
➤ 第二步，完成 Activator 的代码。

■ 打开默认的 HelloWorldActivator.java 代码（见图 1-11）。

```
1  /*
2  *
3  * OSGI Demo
4  */
5  package org.osgichina.demo;
6
7  /**
8   * desc: HelloWorld
9   *
10  * @author chris
11  */
12  import org.osgi.framework.BundleActivator;
13  import org.osgi.framework.BundleContext;
14
15  public class HelloWorldActivator implements BundleActivator {
16
17      /**
18       * (non-Javadoc)
19       * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
20       */
21      public void start(BundleContext context) throws Exception {
22      }
23
24      /**
25       * (non-Javadoc)
26       * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
27       */
28      public void stop(BundleContext context) throws Exception {
29      }
30  }
```

图 1-11 默认 Activator 代码

可以看到 HelloWorldActivator 实现了 BundleActivator 接口，然后 HelloWorldActivator 中有两个空的方法——start 和 stop。其中，start 方法是在 Bundle 被启动的时候调用的，stop 是在 Bundle 被停止的时候调用的，下面我们在这两个方法中加入代码（见图 1-12）。



```

HelloWorldActivator.java
/*
 *
 * OSGi Demo
 */
package org.osgichina.demo;
/**
 * desc: HelloWorld
 *
 * @author chris
 */
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class HelloWorldActivator 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!");
    }
}

```

图 1-12 修改后的 Activator 代码

HelloWorld 的例子已经完成了，下面看这个例子的运行效果。

➤ 第三步，运行。

- 首先创建一个 HelloWorld 用的运行配置（见图 1-13）。

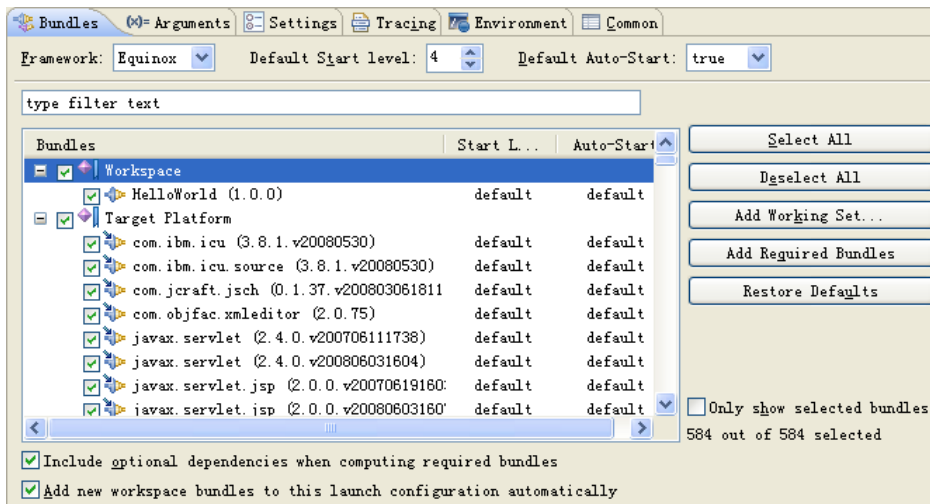


图 1-13 运行配置的 Bundles

- 然后我们设置 HelloWorld 的 Auto-Start 属性为 false，并且将 Target Platform 中不需要的 Bundle 去掉。

再直接点击 HelloWorld 这个 Bundle 配置行后面的 Auto-Start 列的 default，在下拉框中选择 false，并点击 Target Platform 前的钩，使得 Target Platform 下面的项都变为不选择状态。接着我们点击右侧的“Add Required Bundles”，这个时候发现 Target Platform 前的方框变为部分选择的状态，最后我们选中“Only show selected bundles”。这样就完成了运行配置的设置（见图 1-14）。

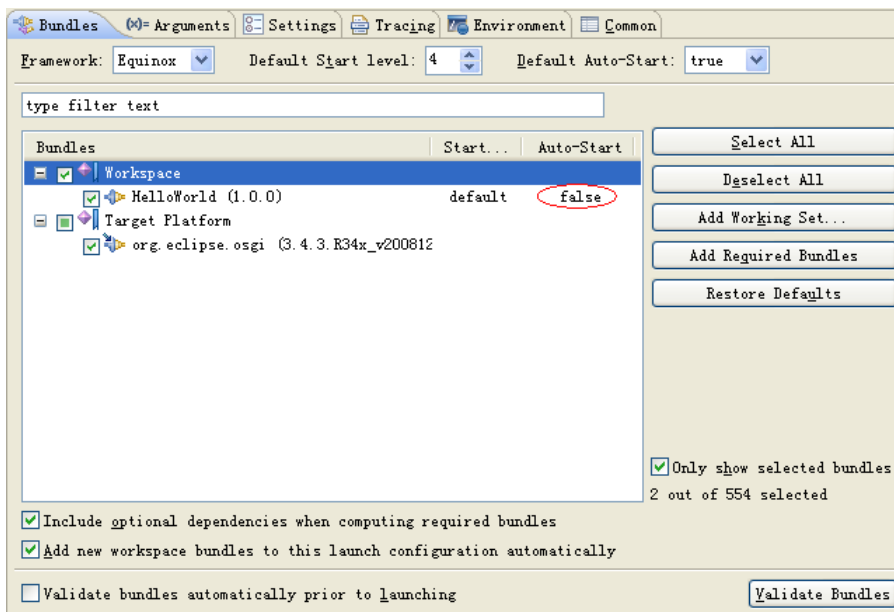


图 1-14 运行配置的 Bundles

- 最后点击“Run”，在 Console 中出现“osgi>”提示。这表明已经成功启动了我们的第一个 OSGi 应用。

在 osgi> 提示符下输入 ss，然后回车，我们可以看到如图 1-15 所示的显示。

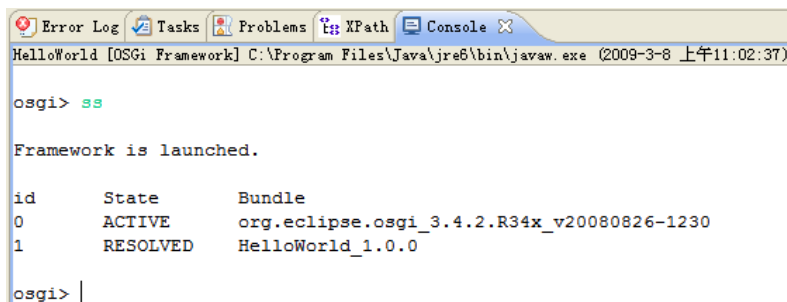


图 1-15 运行结果

可以看到，HelloWorld Bundle 已经被安装并且完成了解析，但是还没有启动。下面我们在 osgi> 提示符下输入 start 1，然后回车，看看会发生什么。这个时候再输入 ss。应该有如图 1-16 所示的显示。

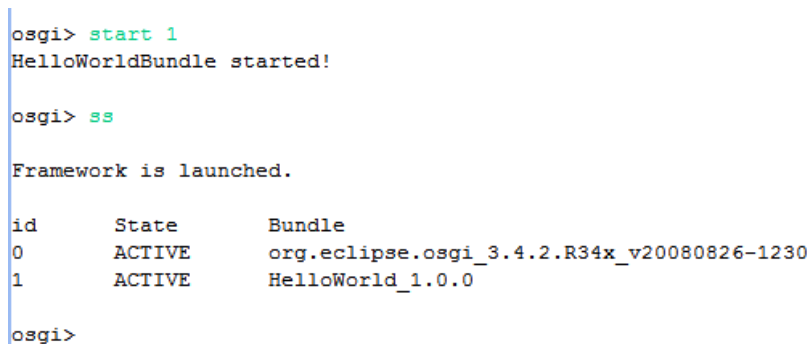


图 1-16 启动 Bundle 后的结果

我们可以看到输出了“HelloWorldBundle started!”，并且通过 ss 命令，看到 HelloWorld Bundle 的

状态从前面的 RESOVLED 变成了 ACTIVE，说明我们的 HelloWorld Bundle 已经成功启动了。并且在 Activator 的 start 方法中加入的输出信息也正确打印出来了。

接着，我们输入 stop 1，并且再用 ss 命令查看 Bundle 的状态，会得到如图 1-17 所示的反馈。

```
osgi> stop 1
HelloWorldBundle stopped!

osgi> ss
Framework is launched.

id      State      Bundle
0        ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230
1        RESOLVED     HelloWorld_1.0.0

osgi>
```

图 1-17 停止 Bundle 后的结果

这个时候 HelloWorld Bundle 已经被停止。我们在 Activator 中加入的输出信息正确地输出在了 Console，并且通过 ss 命令看到 HelloWorld Bundle 的状态从刚才的 ACTIVE 变为了 RESOLVED。

到这里我们完成了第一个 OSGi Bundle，也尝试运行了第一个 OSGi 的程序。在下面的章节，将会看到更加复杂一些的例子。

1.1.4 开发传统类型的应用

1.1.4.1 B/S

我们首先来看一下，如何基于 OSGi 来开发 B/S 结构的应用。B/S 结构应用程序的开发，可有两个选择：一个是在 OSGi 的框架中嵌入 Http 服务器，另外一个是在 Servlet 容器中嵌入 OSGi 框架。下面分别介绍这两种方式的实现。此外，本节还会给出 Declarative Service 的使用实例。

首先来构想一下这个 B/S 应用的功能。我们提供一个字典服务，用户在浏览器中输入一个单词，点击提交，给出这个单词的解释。当然，这个例子并不会真正细致地实现这个功能，比如不会真的做一个字典来提供服务，仅仅支持很少的单词的查询。有了这个例子，如果你有兴趣，可以来完善它。

在这个例子中，我们会有四个 Bundle，分别是字典查询响应 Bundle，字典查询接口 Bundle，本地字典服务 Bundle，远程字典服务 Bundle。设计示意图如图 1-18 所示。

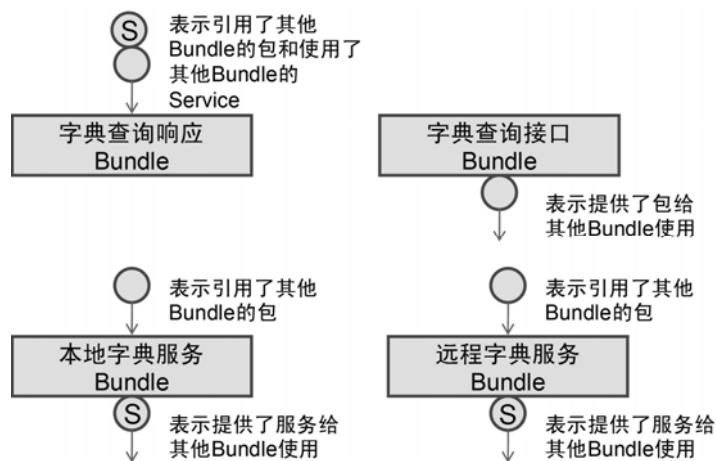


图 1-18 Bundle 设计图

下面分析介绍它们。

1. 字典查询响应 Bundle

提供输入要查询单词的页面，接受用户的查询请求，从 BundleContext 中获取字典服务的 Service，调用字典服务的查询接口得到查询结果，并返回结果到页面。

2. 字典查询接口 Bundle

对外提供字典查询的接口。

3. 本地字典服务Bundle

提供字典查询服务，是从本地的字典中查询结果。

4. 远程字典服务Bundle

提供字典查询服务，是从远程的字典中查询结果。

好了，我知道大家已经迫不及待了。下面我们就来动手实现第一个 B/S 结构的应用。

OSGi框架嵌入Http服务器

这一节我们采用把 Http 服务器嵌入到 OSGi 框架中的方法来完成这个字典查询系统的开发。首先我们要准备一下环境。回忆一下，我们在前面讲到 HelloWorld 例子的时候，介绍过环境的准备。在 HelloWorld 的例子中，我们只需要一个系统的 Bundle。现在我们的运行环境要比 HelloWorld 稍微复杂一些，我们需要更多的 Bundle，下面先来准备一下我们的环境。

我们在 Run Configurations 的对话框中创建一个新的 OSGi Framework 的运行配置，在这个配置的 Bundles 中选择下面几个 Bundle：javax.servlet、org.apache.commons.logging、org.eclipse.equinox.http.jetty、org.eclipse.equinox.http.servlet、org.eclipse.osgi、org.eclipse.osgi.services 和 org.mortbay.jetty，如图 1-19 所示。

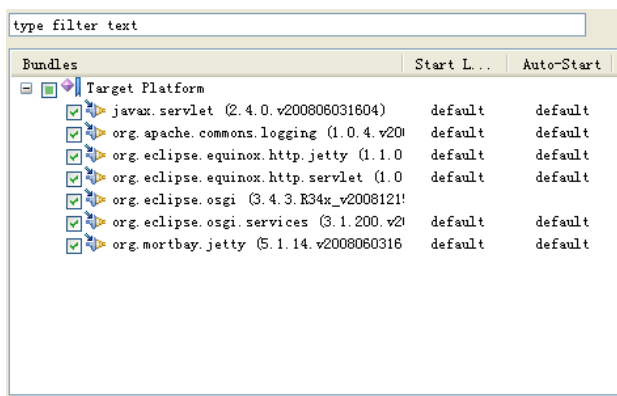


图 1-19 运行配置的 Bundles

点击 Run，可以看到有些启动的日志输出。如果看到 osgi>而且没有错误信息，则表明环境已经配置成功；而如果看到类似 Address already in use: JVM_Bind，则说明本机的 80 端口已被占用，由于 Equinox 的 Http Service 实现默认使用的是 80 端口，所以会报这个错，那么我们可以指定 Http Service 使用的端口。我们打开 Run Configurations 中运行配置里面的 Arguments 页签，在 VM arguments 中添加“-Dorg.osgi.service.http.port=8080”就可以设置使用 8080 作为 Http Service 的端口了，本例子中仍采用默认的 80 端口，在正常启动后，输入 ss，回车，可以看到如图 1-20 所示的显示。

说明我们的环境已经准备好了。我们可以打开浏览器，输入 <http://localhost/>，提示见图 1-21。

```
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE     org.eclipse.osgi_3.4.3.R34x_v20081215-1030
1       ACTIVE     org.eclipse.osgi.services_3.1.200.v20071203
2       ACTIVE     javax.servlet_2.4.0.v200806031604
3       ACTIVE     org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830
4       ACTIVE     org.apache.commons.logging_1.0.4.v20080605-1930
5       ACTIVE     org.mortbay.jetty_5.1.14.v200806031611
6       ACTIVE     org.eclipse.equinox.http.jetty_1.1.0.v20080425

osgi>
```

图 1-20 运行结果

HTTP ERROR: 404

ProxyServlet: /
RequestURI=/
[Powered by Jetty://](#)

图 1-21 浏览器结果显示

说明我们的服务器已经正常启动。下面就可以开始 Bundle 的开发工作了。

- 第一步，完成字典查询接口 Bundle 工程。

首先创建名为 DictQuery 的 Plug-in 工程（见图 1-22）。

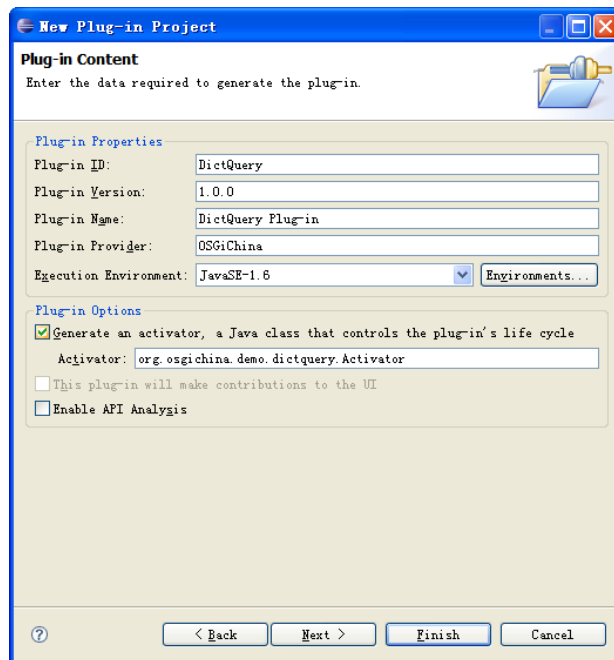


图 1-22 插件信息设置

然后我们在这个工程的 org.osgichina.demo.dictquery.query package 中创建一个接口（见图 1-23）。

```

QueryService.java
package org.osgichina.demo.dictquery.query;

/**
 * @desc 字典查询服务接口
 * @author chris
 */
public interface QueryService {
    /**
     * 根据输入的单词，查询结果
     * @param word 要查询的单词
     * @return 查询结果
     */
    String queryWord(String word);
}

```

图 1-23 QueryService 接口定义

因为这个 Bundle 是为了导出接口，所以这个 Bundle 的 BundleActivator 不做任何的改动。

双击这个 Bundle 工程中 META-INF 下的 MANIFEST.MF 文件，在新的窗口中会显示这个 MANIFEST 的信息（见图 1-24）。

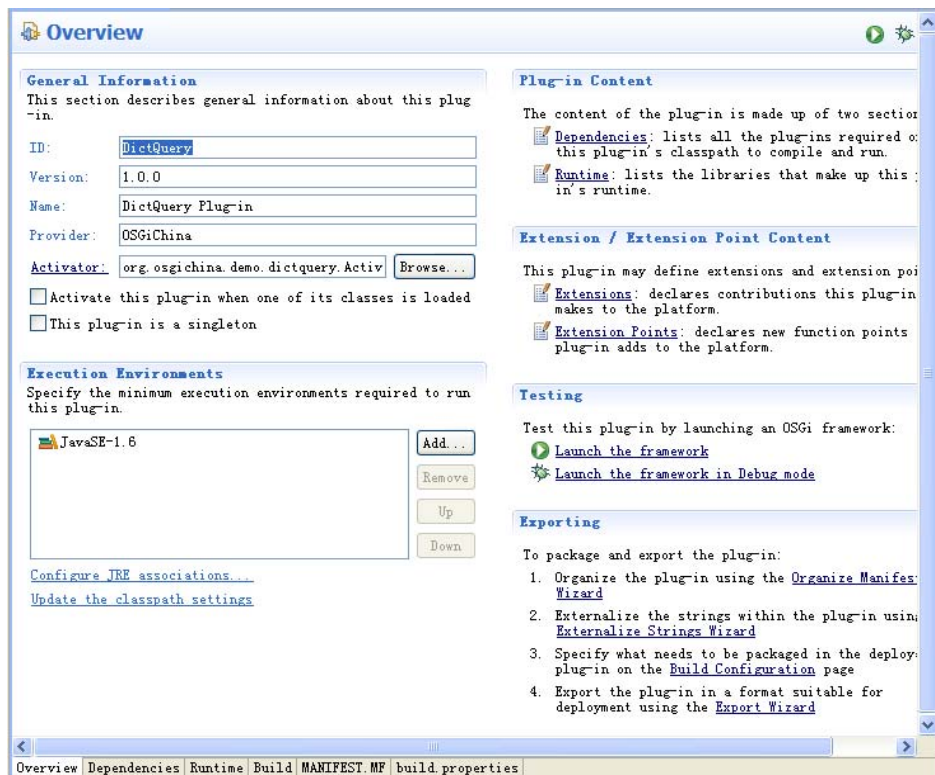


图 1-24 MANIFEST.MF 显示

可以看到这个窗口下有多个页签，其中 Overview、Dependencies、Runtime、Build 这四个页签是图形化的编辑页签，对应的修改会反映到 MANIFEST.MF 和 build.properties 文件中。MANIFEST.MF 和 build.properties 这两个页签分别直接显示了 MANIFEST.MF 和 build.properties 文件的内容。

下面来完成将接口的 package 从 Bundle 中导出，能够让其他的 Bundle 来使用这个接口的 package。

首先选择 Runtime 页签，然后点击 Exported Packages 中的 Add，在弹出的窗口中选择所创建的 QueryService 所在的 package: org.osgichina.demo.dictquery.query，点击保存就完成导出 Package 的操作了（见图 1-25）。

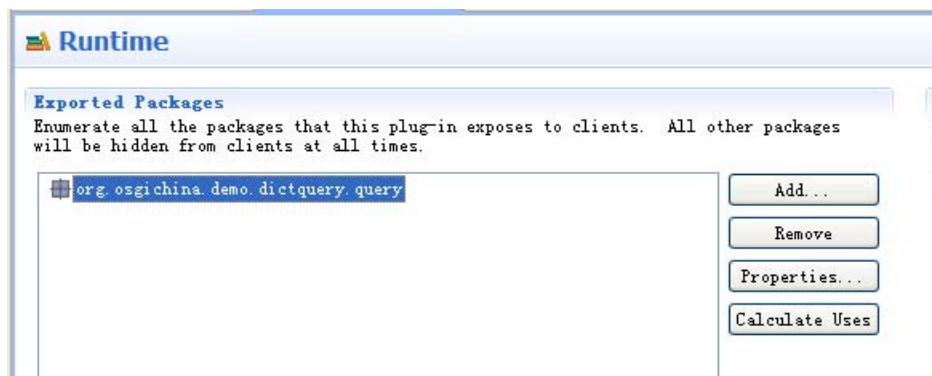


图 1-25 导出的 Package

这个时候去看 MANIFEST.MF 文件，其中多了如下一行：

```
Export-Package: org.osgi.china.demo.dictquery.query
```

到此，我们就完成了字典查询接口 Bundle 的开发了。接着，我们来完成本地字典查询 Bundle 的开发。

➤ 第二步，完成本地字典查询 Bundle。

- 首先，创建一个名字为 LocalDictQuery 的插件工程。
- 然后，导入字典查询接口 Bundle，并且实现一个真正的字典查询的类。

我们打开 LocalDictQuery 工程中 META-INF 下的 MANIFEST.MF 文件，选择 Dependencies 页签。点击右侧 Imported Packages 的 Add 按钮，在弹出的对话框中选择前面请求处理接口 Bundle 导出的那个 Package: org.osgi.china.demo.dictquery.query，保存后就完成了导入的操作。我们可以查看一下请求处理 Bundle 的 MANIFEST.MF 文件，在 Import-Package 项中，多了 org.osgi.china.demo.dictquery.query（见图 1-26）。

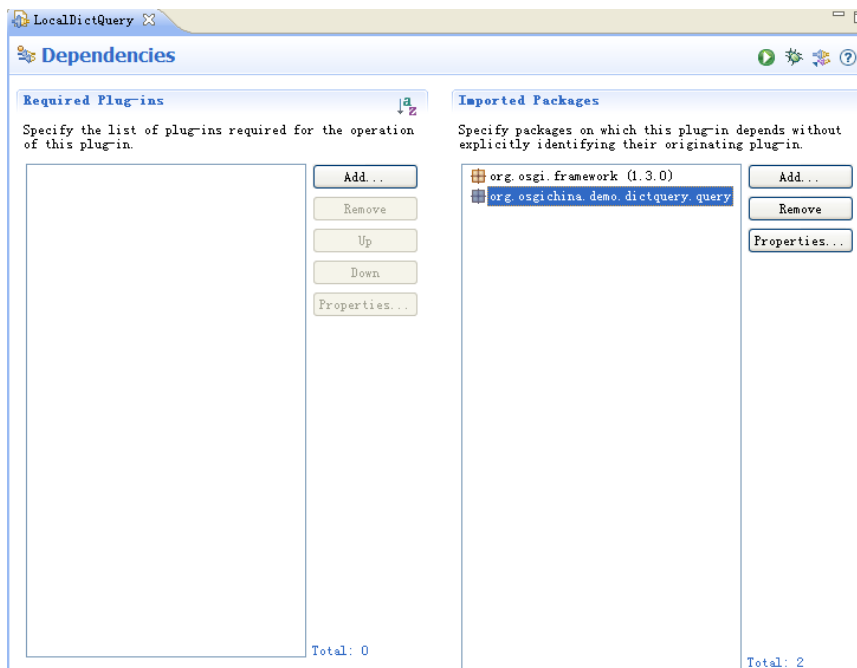


图 1-26 引入的 Package

- 接着，我们来编写 LocalDictQueryServiceImpl 的代码，这个是实现了 QueryService 接口的一个类。LocalDictQueryServiceImpl 中的 queryWord 方法，可从本地初始化的字典中查询结果；如果没有结果，返回一个“N/A”，代码如图 1-27 所示。

```
package org.osgichina.demo.localdictquery.impl;

import java.util.concurrent.ConcurrentHashMap;

/**
 * @desc 从本地字典中查询单词
 * @author chris
 */
public class LocalDictQueryServiceImpl implements QueryService {
    private static final ConcurrentHashMap<String, String> dict =
        new ConcurrentHashMap<String, String>();

    static {
        dict.put("test", "测试");
        dict.put("China", "中国");
    }

    @Override
    public String queryWord(String word) {
        System.out.println("LocalDictQueryServiceImpl.queryWord called!");
        String result = dict.get(word);
        if (null == result) {
            result = "N/A";
        }
        return result;
    }
}
```

图 1-27 LocalDictQueryServiceImpl 实现代码

- 最后，我们要编写 Activator 的代码（见图 1-28），在 Bundle 启动的时候注册我们提供的字典查询服务。从开发角度来看，Service 有点像虚拟的概念，因为在编写 Service 时和编写普通的 Java 类（POJO）没有任何区别，这个工程中的 LocalDictQueryServiceImpl 就是一个 Service 类。那么如何能够让其他的 Bundle 使用这个 Service？在 OSGi 的框架中，我们要通过 BundleContext 来进行服务的注册，并且后面会看到在其他的 Bundle 中如何拿到这个 Service 实例来使用。

```
package org.osgichina.demo.localdictquery;

import org.osgi.framework.BundleActivator;

public class Activator implements BundleActivator {
    private ServiceRegistration sr = null;

    /**
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        sr = context.registerService(QueryService.class.getName(),
            new LocalDictQueryServiceImpl(), null);
    }

    /**
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        sr.unregister();
    }
}
```

图 1-28 Activator 代码

可以看到，在自动生成的 Activator 中，我们改动了两处。一个是我们加入了一个类型是 ServiceRegistration 的成员变量 sr，一个是在 start 和 stop 方法中分别加入了一行代码，下面我们来看这两行代码的含义。

```
sr = context.registerService(QueryService.class.getName(),
    new LocalDictQueryServiceImpl(), null);
```

以上两行代码是用 `QueryService` 的全类名作为注册的 `Service` 的名称，把新创建的 `LocalDictQueryServiceImpl` 对象注册成为了一个 `Service`。而这个 `Service` 对象将在下面演示如何被使用。

```
sr.unregister();
```

以上这行代码是取消注册的 `Service`。到这里，我们就完成了请求处理 `Bundle` 的代码编写。

➤ 第三步，完成实现远程字典查询 `Bundle`。

`Bundle` 和 `LocalDictQuery Bundle` 非常地类似，只是工程名称为 `RemoteDictQuery`，另外为了能够显示区别，这个 `Bundle` 中提供服务的类的代码有所变化。

实现 `QueryService` 接口的类的代码如图 1-29 所示。

```
package org.osgichina.demo.remotedictquery.impl;

import java.util.concurrent.ConcurrentHashMap;

/**
 * @desc 远程查询单词
 * @author chris
 */
public class RemoteDictQueryServiceImpl implements QueryService {
    private static final ConcurrentHashMap<String, String> dict =
        new ConcurrentHashMap<String, String>();
    static {
        dict.put("sky", "天空");
        dict.put("computer", "计算机");
    }

    @Override
    public String queryWord(String word) {
        System.out.println("RemoteDictQueryServiceImpl.queryWord called!");
        String result = dict.get(word);
        if (null == result) {
            result = "N/A";
        }
        return result;
    }
}
```

图 1-29 RemoteDictQueryServiceImpl 实现代码

最后，我们完成字典查询响应 `Bundle`。

➤ 第四步，完成字典查询响应 `Bundle`。

和传统的 Web 开发方式不同，由于 OSGi 框架中并没有像 web 应用服务器那样的 `Bundle`，就不能像 web 应用直接部署到 web 服务器那样简单了，要通过 `HttpService` 将 `Servlet` 及资源文件（像图片、css、html 等）进行注册，这样才可以访问。这里只是一个简单的 Demo，提供一个字典查询的页面和对应的 `Servlet`。我们在 `src` 目录下建立一个 `page` 的目录，在其中编写 `dictquery.htm`，另外就是实现字典查询响应的 `Servlet`，由于 `Servlet` 要继承 `HttpServlet`，要引用 `Servlet API`。要引入 `javax.servlet` 和 `javax.servlet.http` 两个包，接着，除了要引入 `org.osgichina.demo.dictquery.query` 这个 package 外，还要一个 `org.osgi.service.http` package，如图 1-30 所示。

然后，我们可以编写 `Servlet` 的代码，和普通的 `Servlet` 的写法没有差别。要解释的是在 `doGet` 方法中的这么一段代码：

```
ServiceReference serviceRef =
    context.getServiceReference(QueryService.class.getName());
```

```

if(null != serviceRef){
    queryService = (QueryService) context.getService(serviceRef);
}

```

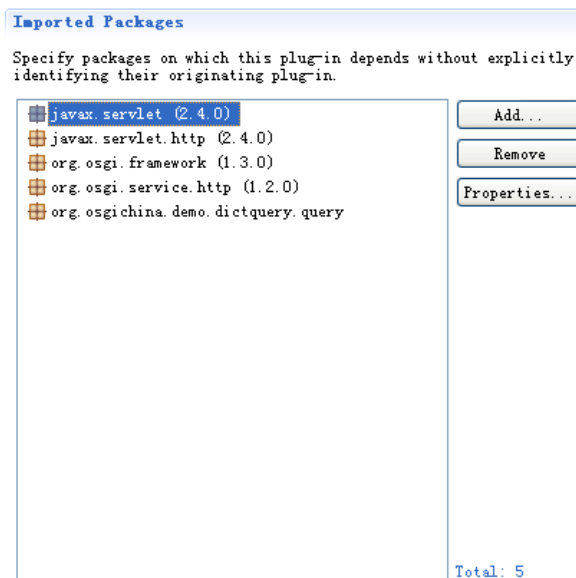


图 1-30 引入的 Package

context 是在创建 Servlet 的时候传入的 BundleContext，要通过这个 context 来获取提供字典查询功能的服务。首先通过 context 获取 Service 的引用，返回的是一个 ServiceReference 对象。然后再通过 ServiceReference 获取 Service 实例。拿到 Service 实例后，就可以调用 Service 的方法来完成字典查询功能了。

另外，在这个字典查询响应 Bundle 中还要做的一件事情就是在 Bundle 启动的时候，把 Servlet 注册到 Http 服务中去。这个代码是在 BundleActivator 中完成的。具体代码可以查看源码。

至此已经完成了字典查询系统的开发。下面来运行一下系统。

启动之后我们在 osgi>提示符下输入 ss，回车，可以看到类似图 1-31 的显示。

```

osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE      org.eclipse.osgi_3.4.3.R34x_v20081215-1030
3       ACTIVE      org.apache.commons.logging_1.0.4.v20080605-1930
4       ACTIVE      org.mortbay.jetty_5.1.14.v200806031611
5       ACTIVE      org.eclipse.osgi.services_3.1.200.v20071203
7       ACTIVE      org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830
8       ACTIVE      org.eclipse.equinox.http.jetty_1.1.0.v20080425
9       ACTIVE      DictQuery_1.0.0
10      ACTIVE      RemoteDictQuery_1.0.0
11      ACTIVE      LocalDictQuery_1.0.0
12      ACTIVE      javax.servlet_2.4.0.v200806031604
13      ACTIVE      DictQueryWeb_1.0.0

```

图 1-31 启动后的 Bundle 状态

打开浏览器，然后输入 <http://localhost/demo/page/dictquery.htm>，将会显示如图 1-32 所示的内容。



图 1-32 运行后的浏览器效果

下面，我们先执行 stop 10（停掉 RemoteDictQuery Bundle），然后输入 test，点击查询。会有如下显示：

```
Result is 测试
```

并且我们可以在 Eclipse 的 Console 中看到如下的输出：

```
osgi> LocalDictQueryServiceImpl.queryWord called!
```

接着我们执行 stop 11，回车，stop 10，回车。停止 LocalDictQuery Bundle，并且启动 RemoteDictQuery Bundle，然后在查询页面上输入 sky，点击查询。会有如下显示：

```
Result is 天空
```

到这里，我们已经完成了字典查询系统的开发。

1. 测试和调试

前面很顺利地完成了字典查询系统的开发和运行。但是在现实生活中，一个系统最终的完成和上线，始终离不开测试和调试。对于测试的支持无疑是现在对于框架的重要考评点。那么基于 OSGi 框架的系统怎么做测试呢？系统的集成测试方法只和系统的结构（B/S、C/S）有关，和框架没什么关系，所以这里就是来看看如何完成基于 OSGi 框架的系统单元测试。

编写单元测试时最重要的就两点：

- 设置测试类的依赖（通过 Mock、实例化等方法）；
- 构造输入场景，检验输出是否和预期的一致。

在做单元测试时最复杂的部分往往就是设置测试类的依赖，典型的就像 EJB 中的 session bean 的测试，由于它要依赖 EJB Container，所以是比较麻烦的。在我们的例子中，已经编写的几个 Service 是没有依赖的情况的，只须测试在某种输入的情况下方法执行的输出和预期的结果是否一致，这就很容易编写了，具体请参见源码中 DictQuery_Classic 目录里面的 LocalDictQueryServiceImplTest.java 和 RemoteDictQueryServiceImplTest.java 文件。而在字典查询响应 Bundle，最复杂的就是字典查询响应 Servlet 的测试。这个 Servlet 要用到 OSGi 框架的 BundleContext 获取字典查询服务，同时还要依赖 HttpServletRequest 和 HttpServletResponse，在没法实例化类所依赖的环境时，只能采用 Mock 的方法来实现，代码见源码中 DictQuery_Classic 目录下的 DictQueryServletTest.java。

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

调试也是系统开发关注的重点，由于可以在 Eclipse 中启动基于 Equinox 开发的系统，那么一切都不是问题。和普通 Java 工程进行调试的方法没有任何区别，设置断点，Debug，就 OK 了。在

运行到断点对应的代码时就进入熟悉的调试视图了（见图 1-33）。

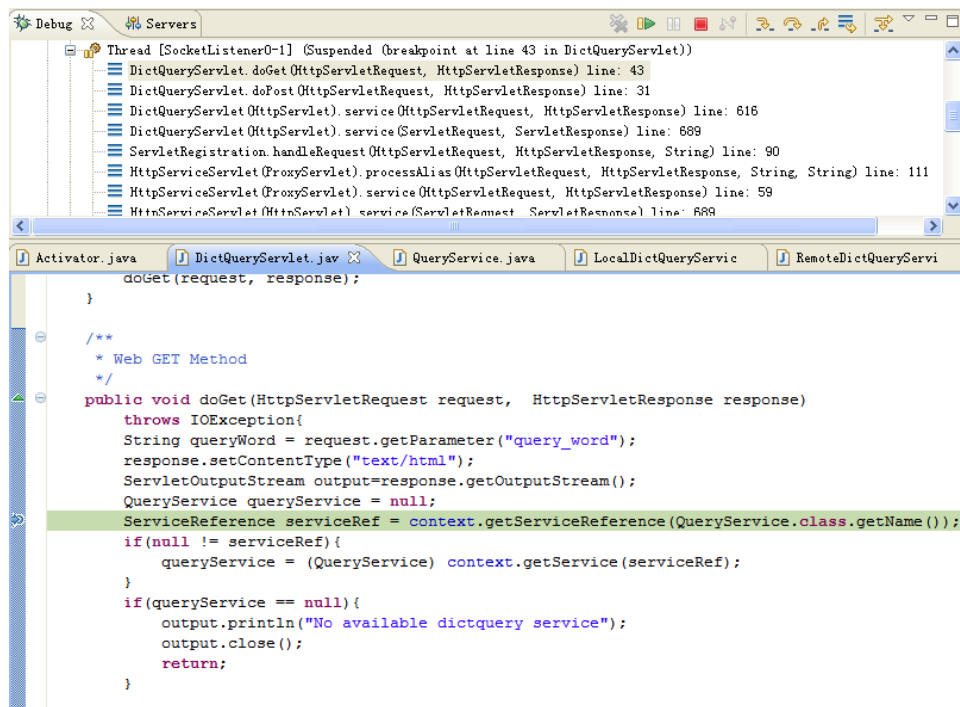


图 1-33 调试视图

2. 应用的部署

我们已经完成了字典查询整个应用的开发、运行、调试和测试工作。不过这些都是在 Eclipse 这个开发工具中完成的，我们可不能给客户一个要客户在 Eclipse 上运行的应用，所以下面来看一下如何部署 OSGi 应用。

- 第一步，创建独立的 Equinox 运行环境。

在硬盘上创建一个 `osgi_demo` 目录，从 Eclipse 的 `plugins` 目录复制 `org.eclipse.osgi_3.4.3.R34x_v20081215-1030.jar`（在不同版本的 Eclipse 中，这个 jar 包的 `org.eclipse.osgi_` 后面的部分会有所不同）到这个 `osgi_demo` 目录。修改这个 jar 包的名称为 `equinox.jar`，然后在相同目录下编写一个 `run.bat`，其内容如下：

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

双击 `run.bat`，如果出现 `osgi>` 的提示，则说明启动已经成功了。输入 `ss` 命令然后回车，这个时候会看到只有一个 `ACTIVE` 状态的 `system bundle`。

- 第二步，导出各个 Bundle 工程为 jar。

以最复杂的 DictQueryWeb 为例，首先打开 `MANIFEST.MF`，选择 `Runtime` 页签，设置 `Classpath`（见图 1-34）。

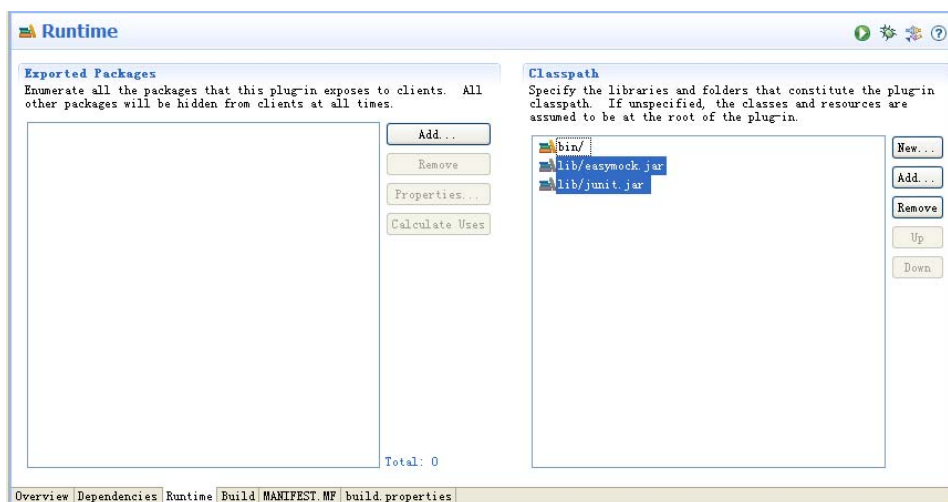


图 1-34 ClassPath 设置

然后选择 `Build` 页签，选中其中 `Binary Build` 里面的 `lib` 目录（见图 1-35）。

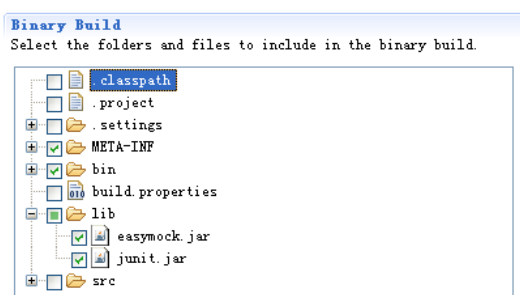


图 1-35 Build 设置

接着，选中 DictQueryWeb 工程，点右键，选择 `Export`，然后选中弹出对话框中的 `Deployable plug-ins and fragments`（见图 1-36）。

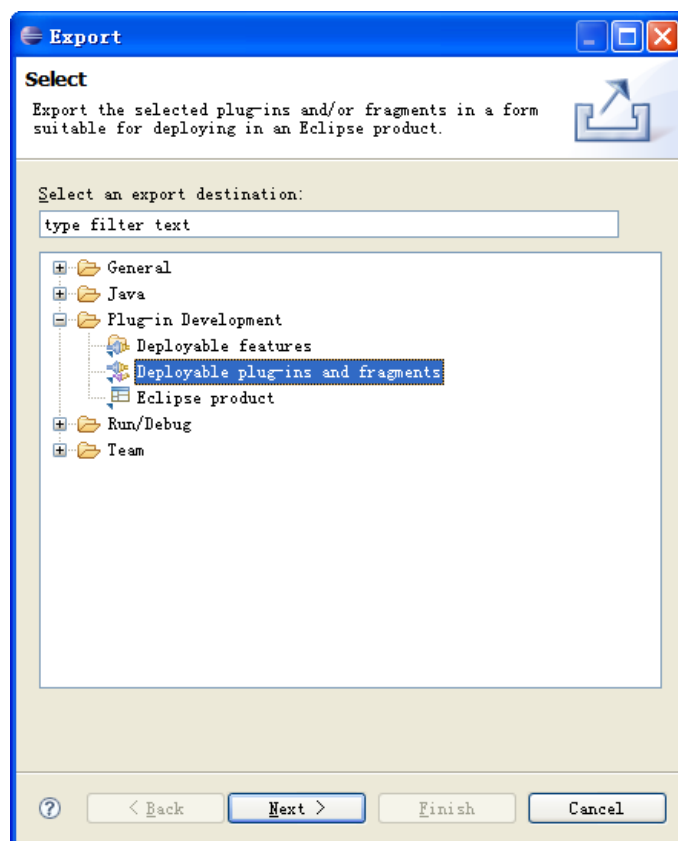


图 1-36 导出插件

在进入到 Deployable plug-ins and fragments 窗口中已经默认选择了 DictQueryWeb 这个 bundle，然后选择 Destination 标签页，设置一个有效的 Directory，然后点击 Finish，在设置的目录中可以看到一个 plugins 目录，在 plugins 目录中就有 DictQueryWeb_1.0.0.jar 这个 bundle 了。

按照相同的方法可以导出其他的几个 bundle，也可以一次性地把几个 bundle 都导出来（见图 1-37）。

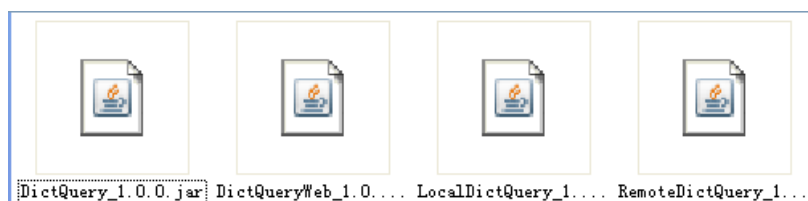


图 1-37 导出的 jar 包

➤ 第三步，安装各 Bundle 到 Equinox 中。

首先在 osgi_demo 目录下创建一个 bundles 目录，然后将第二步生成的三个 bundle 复制到 bundles 目录下，此外，我们要从 Eclipse 的 plugins 目录中把我们需要的 javax.servlet_2.4.0.v200806031604.jar、org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830.jar、org.eclipse.equinox.http.jetty_1.1.0.v20080425.jar、org.mortbay.jetty_5.1.14.v200806031611.jar、org.apache.commons.logging_1.0.4.v20080605-1930.jar、org.eclipse.osgi.services_3.1.200.v20071203.jar 这几个 Jar 文件复制到 Bundles 目录中。

有两种办法可将 Bundle 安装到 Equinox 中：

- 第一种：运行之前编写的 run.bat，在 osgi> 中输入 install reference:file:bundles/DictQuery_1.0.0.jar，回车，这样就完成了 DictQuery Bundle 的安装。我们可以用同样的方法完成 DictQueryWeb Bundle、LocalDictQueryBundle、RemoteDictQuery Bundle 及其他系统 Bundle 的安装。

然后在 osgi>提示符下输入 ss，回车，可以看到如图 1-38 所示的状态。

```
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE      org.eclipse.osgi_3.4.3.R34x_v20081215-1030
1       INSTALLED   DictQuery_1.0.0
2       INSTALLED   DictQueryWeb_1.0.0
3       INSTALLED   LocalDictQuery_1.0.0
4       INSTALLED   RemoteDictQuery_1.0.0
5       INSTALLED   javax.servlet_2.4.0.v200806031604
6       INSTALLED   org.apache.commons.logging_1.0.4.v20080605-1930
7       INSTALLED   org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830
8       INSTALLED   org.eclipse.equinox.http.jetty_1.1.0.v20080425
9       INSTALLED   org.eclipse.osgi.services_3.1.200.v20071203
10      INSTALLED   org.mortbay.jetty_5.1.14.v200806031611

osgi>
```

图 1-38 启动后的 Bundle 状态

可见，目前我们安装后的 10 个 Bundle 都已经是 INSTALLED 的状态。下面让我们来启动这些 Bundle。首先来启动系统的 Bundle，依次启动 javax.servlet、org.apache.commons.logging、org.eclipse.osgi.services、org.mortbay.jetty、org.eclipse.equinox.http.servlet 和 org.eclipse.equinox.http.jetty，然后启动我们自己开发的 Bundle。可以在 osgi>提示符下输入 ss，会有如图 1-39 所示的输出。

这个时候，可以通过浏览器来测试我们的应用了。

最后输入 exit，就可以退出系统。以后只须双击 run.bat 就可以完成系统的启动。经过这样的步骤，就形成了单独运行的 OSGi 系统。

```
Framework is launched.

id      State      Bundle
0       ACTIVE      org.eclipse.osgi_3.4.3.R34x_v20081215-1030
1       ACTIVE      DictQuery_1.0.0
2       ACTIVE      DictQueryWeb_1.0.0
3       ACTIVE      LocalDictQuery_1.0.0
4       RESOLVED    RemoteDictQuery_1.0.0
5       ACTIVE      javax.servlet_2.4.0.v200806031604
6       ACTIVE      org.apache.commons.logging_1.0.4.v20080605-1930
7       ACTIVE      org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830
8       ACTIVE      org.eclipse.equinox.http.jetty_1.1.0.v20080425
9       ACTIVE      org.eclipse.osgi.services_3.1.200.v20071203
10      ACTIVE      org.mortbay.jetty_5.1.14.v200806031611

osgi>
```

图 1-39 启动了要用的 Bundle 后的 Bundle 状态

- 第二种：安装 Bundle 的方法。

在 osgi_demo 目录下创建 configuration 目录，在 configuration 目录下创建 config.ini 文件，这个文件内容如下：

```
osgi.noShutdown=true
osgi.bundles=reference\:file\:bundles/javax.servlet_2.4.0.v200806031604.jar@start,reference\:file\:bundles/org.apache.commons.logging_1.0.4.v20080605-1930.jar@start,reference\:file\:bundles/org.eclipse.osgi.services_3.1.200.v20071203.jar@start,reference\:file\:bundles/org.mortbay.jetty_5.1.14.v200806031611.jar@start,refere
```

```
nce\;file\;bundles/org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830.jar@start,reference\;file\;bundles/org.eclipse.equinox.http.jetty_1.1.0.v20080425.jar@start,reference\;file\;bundles/DictQuery_1.0.0.jar@start,reference\;file\;bundles/LocalDictQuery_1.0.0.jar@start,reference\;file\;bundles/RemoteDictQuery_1.0.0.jar@start,reference\;file\;bundles/DictQueryWeb_1.0.0.jar@start
osgi.bundles.defaultStartLevel=4
```

这样，双击 `run.bat` 就能够完成整个系统的启动。

接下来，介绍使用 Declarative Service 来实现的字典查询系统。

Declarative Service版本的实现

这一节，我们将采用 Declarative Service 的方式来实现前面的字典查询系统。在第 1 章，我们对 Declarative Service 已有一个初步的介绍。相比于通过 BundleContext 获取 Service 这样的方式，Declarative Service 更像是一种 Dependency Injection 的方式。下面我们来准备一下环境，首先下载 org.eclipse.equinox.ds 包 (https://www.osgi.org/repository/org.eclipse.equinox.ds_1.0.0.v20060828.jar)，下载完毕后放入 Eclipse 的 plugins 目录中，重新启动 Eclipse。下面来修改我们之前的字典查询系统的代码。

1. 重构服务发布实现

在我们的例子中，LocalDictQuery Bundle 和 RemoteDictQuery Bundle 是通过在 BundleActivator 中主动注册服务来提供服务的。这里要重构这两个 Bundle 的代码。

- 第一步，去掉 Activator。

我们无须通过 Activator 来注册和注销服务，可以删除工程中的 Activator 的类，并且修改 MANIFEST.MF 文件，去掉其中的 Bundle-Activator 属性。

- 第二步，创建 Component 配置。

在工程中创建 OSGI-INF 目录，在该目录下创建 component.xml 文件。文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="DictQueryService">
  <implementation class="org.osgichina.demo.localdictquery.impl.
    LocalDictQueryServiceImpl"/>
  <service>
    <provide interface="org.osgichina.demo.dictquery.query.
      QueryService"/>
  </service>
</component>
```

这个是 LocalDictQuery Bundle 的 component.xml，RemoteDictQuery Bundle 的 component.xml 是类似的写法，只是 implementation 的 class 有所不同。另外，我们不在 RemoteDictQuery Bundle 的 MANIFEST.MF 中加入 Component 的设置，我们只希望系统中有一个 DictQuery 服务。

- 第三步，在 MANIFEST.MF 引用 component 配置。

打开 MANIFEST.MF，在文件中加入对于 component 配置的引用：

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

这就完成了服务发布的重构，下面来看一下服务引用的重构。

2. 重构服务引用实现

我们的例子中只有字典查询响应 Bundle 引用了服务，只须修改这一个工程。

- 第一步，去掉 Activator

之前的实现是在 `BundleActivator` 中监听 `HttpService` 来注册 `Servlet` 的，在 `DS` 中不需要这样的做法了，所以我们先删除 `Activator` 和 `MANIFEST.MF` 中关于 `Activator` 的设置。

■ 第二步，修改 `Servlet`

我们不再通过 `BundleContext` 来获取 `Service` 了，所以删除 `Servlet` 中以 `BundleContext` 为参数的构造器，然后增加 `setHttpService`、`unsetHttpService`、`setQueryService`、`unsetQueryService` 方法，用来获取 `HttpService` 和 `QueryService`。

■ 第三步，加入 `component` 配置

在 `DictQueryWeb` 工程下创建 `OSGI-INF` 目录，并在该目录下创建 `component.xml` 文件，文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="DictQueryServlet">
  <implementation class="org.osgichina.demo.dictqueryweb.servlet.
    DictQueryServlet"/>
  <reference name="QueryService" interface="org.osgichina.demo.dictquery.
    query.QueryService" bind="setQueryService" unbind="
    unsetQueryService" policy="dynamic" cardinality="0..1"/>
  <reference name="HttpService" interface="org.osgi.service.http.
    HttpService" bind="setHttpService" unbind="unsetHttpService"
    policy="dynamic"/>
</component>
```

完成了重构，就可以启动系统了，须要注意的是，在 `Run Configurations` 中要引入 `org.eclipse.equinox.ds bundle`，并且设置这个 `bundle` 的 `start level` 为 2，让 `ds` 这个 `bundle` 先启动。然后我们可以通过浏览器来测试一下系统的功能。

现在 `Servlet` 重构后不须要再通过 `BundleContext` 获取 `QueryService` 了，对于 `Servlet` 的测试代码也无须去 `Mock ServiceReference` 和 `BundleContext` 对象了。新的测试代码请参看源码中的 `DictQuery_DS` 目录下的 `DictQueryServletTest.java`。

我们已经完成了 `OSGi` 框架中嵌入 `Http` 服务器的例子，也介绍了 `Declarative Service` 的用法。下面我们简单看一下如何通过把 `OSGi` 框架嵌入 `Servlet` 容器中的方式完成基于 `OSGi` 的 `Web` 应用的开发。

Servlet容器嵌入OSGi框架

在 `Equinox` 中提供了另外一种方式，是在 `Servlet` 容器中嵌入 `Equinox`。在 `Equinox` 的网站上，可以下载到 `bridge.war` (<http://www.eclipse.org/equinox/server/downloads/bridge.war>)，也可以自己从 `CVS` 下载代码来编译最新的版本。我们把这个 `bridge.war` 部署到 `Servlet` 容器中，就可以在控制台上使用 `Equinox` 的命令了。启动后的显示如图 1-40 所示。

从中可以看到熟悉的 `osgi>` 的提示，我们可以在 `osgi>` 提示符下安装使用 `BundleContext` 注册 `Service` 的版本系统的 `Bundle`，然后启动已经安装的 `Bundle`。我们可以从浏览器上使用我们的系统，但是在提交后会出现问题。原因是什么呢？因为现在我们的系统是在 `bridge` 这个 `Web` 应用下的，而我们的 `HTML` 页面却是提交到了 `/demo/dictquery`，这样是不行的，一个办法是修改提交的地址，另外一个办法是把 `bridge.war` 这个包解压后直接放到 `Tomcat` 安装目录的 `webapps` 下的 `ROOT` 目录中。

在这两种 `B/S` 的开发方式中，`Equinox` 是推荐使用在 `Equinox` 中嵌入 `HttpServer` 的方式来进行的。所以在 `Servlet` 容器中嵌入 `Equinox`，我们就不再花过多的篇幅来介绍了。

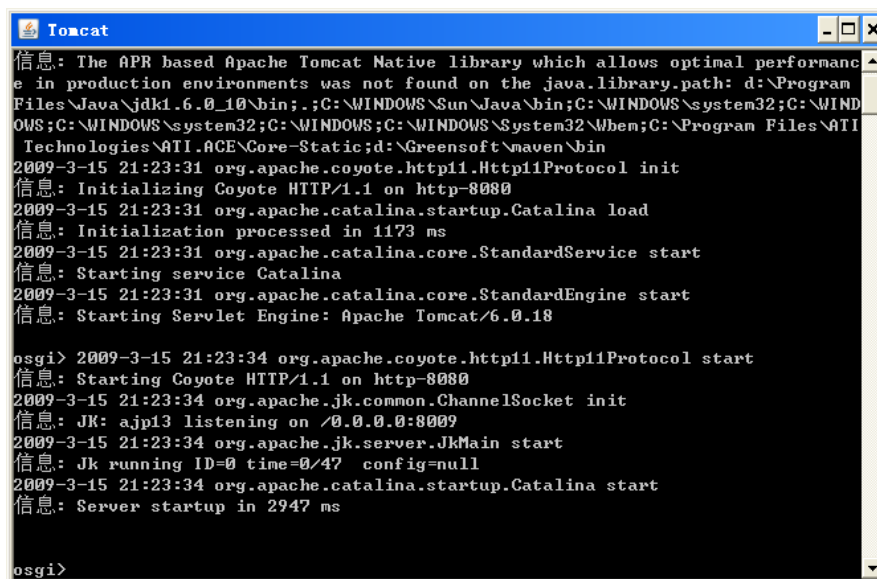


图 1-40 启动后的显示

1.1.4.2 Standalone的应用

我们的第一个 HelloWorld 的例子，就是一个 Standalone 类型的应用。当然 HelloWorld 中只有一个 Bundle，而实际的例子会由很多的 Bundle 组成，我们可以使用经典的 BundleActivator 注册服务，或者用 Declarative Service 来注入组件的方式完成功能。

1.1.4.3 C/S

C/S 结构的程序可以看成是两个 Standalone 的应用。它们遵循一定的协议进行通信。所以 C/S 程序的开发对我们来说也不存在什么困难。

1.1.4.4 嵌入式

提到嵌入式结构的系统开发，绝对是 OSGi 的强项，OSGi 的诞生就是为嵌入式系统提供支撑的，所以在嵌入式系统方面采用 OSGi 没有任何的问题。综合对于 B/S、Standalone、C/S、嵌入式这几种体系结构系统的支持，可以看出对于 Standalone、C/S、嵌入式系统而言采用 OSGi 框架没有什么太多的问题，对于 B/S 结构的系统而言其实大部分情况下是通过采用 HttpService 来实现的，实在不行的话可以采用 Equinox 提供的 server-side incubator 中的 bridge.war 来实现，相信随着 OSGi 在 Server-Side 应用和企业应用中的不断发展，对于 B/S 结构应用的支撑将会越来越优秀。

1.1.4.5 Equinox注意事项

在使用 Equinox 时，特别要注意的是使用 classloader 去加载资源文件的时候，由于每个 Bundle 拥有独立的 classloader。而有些开源框架会使用顶级 classloader 去加载文件，这个时候会导致在 Equinox 中运行时出错。如在 equinox 使用 spring 时，若配置文件中采用了 classpath:file 这样的方式去加载其他的配置文件，就会出现找不到文件的现象，这个错误就是由这个问题引起的。

另外要注意的就是包的命名问题，如果引用了其他 Bundle Export 的 package，那么在当前 Bundle 中就不能再建同样的包了，否则在运行时会出现找不到包中类的现象。

1.1.5 从外部启动Equinox

前面，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。

我们在前面演示了如何通过命令行来启动 Equinox，常见的一种脚本为：`java-jar plugins/org.eclipse.osgi_3.4.3.R34x_v20081215-1030.jar -configuration configuration -console`，然后在当前的 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`，那么要在程序中启动 Equinox 容器，其实是做差不多的事情。

通过查看 Equinox 的代码，会看到调用上面的 `org.eclipse.osgi.jar` 后执行的是 `EclipseStarter` 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 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 `getSystemBundleContext` 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 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：

```

/**
 * 启动并运行 equinox 容器
 */
public static void start() throws Exception{
    // 根据要加载的 bundle 组装出类似 a.jar@start,b.jar@3:start 这样格式的
    // osgibundles 字符串来
    String osgiBundles="";
    // 配置 Equinox 的启动
    FrameworkProperties.setProperty("osgi.noShutdown", "true");
    FrameworkProperties.setProperty("eclipse.ignoreApp", "true");
    FrameworkProperties.setProperty("osgi.bundles.defaultStartLevel", "4");
    FrameworkProperties.setProperty("osgi.bundles",
        osgiBundlesBuilder.toString());
    // 根据需要设置 bundle 所在的路径
    String bundlePath="";
    // 指定要加载的 plugins 所在的目录
    FrameworkProperties.setProperty("osgi.syspath", bundlePath);
    // 调用 EclipseStarter，完成容器的启动，指定 configuration 目录
    EclipseStarter.run(new String[]{"-configuration","configuration",
        "-console"}, null);
    // 通过 EclipseStarter 获得 BundleContext
    context=EclipseStarter.getSystemBundleContext();
}

/**
 * 停止 equinox 容器
 */
public static void stop(){
    try {
        EclipseStarter.shutdown();
        context=null;
    }
    catch (Exception e) {
        System.err.println("停止 equinox 容器时出现错误:"+e);
        e.printStackTrace();
    }
}

/**
 * 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情况
 *
 * @param serviceName 服务名称（完整接口类名）
 *
 * @return Object 当找不到对应的服务时返回 null
 */
public static Object getOSGiService(String serviceName){
    ServiceReference serviceRef=context.getServiceReference
        (serviceName);
    if(serviceRef==null)
        return null;
    return context.getService(serviceRef);
}

/**

```

```

    * 获取 OSGi 容器中插件的类
    */
    public static Class<?> getBundleClass(String bundleName,
        String className) throws Exception{
        Bundle[] bundles=context.getBundles();
        for (int i = 0; i < bundles.length; i++) {
            if(bundleName.equalsIgnoreCase(bundles[i].getSymbolicName())){
                return bundles[i].loadClass(className);
            }
        }
    }
}

```

在实现了 OSGi 容器外与 OSGi 交互之后，通常会同时产生一个需求，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：

```

public class ClassLoaderService{
    public void setClassLoader(ClassLoader classloader);
}

```

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。到这里，我们基本上完成了对于 Equinox 的介绍，下面来看另外一个应用也较广泛的 OSGi 的容器——Felix。

1.2 Felix

1.2.1 简介

Felix 是 Apache Foundation 关于 OSGi R4 的一个实现。包括了 OSGi 框架和标准的服务，同时也提供并且支持其他的 OSGi 相关技术。Felix 最终的目标是要提供和 OSGi 框架和标准服务完全兼容的一个实现。目前 Felix 已经实现了 OSGi R4 规范中的大部分内容。具体的情况可以参考 <http://felix.apache.org/site/index.html>。

1.2.2 环境搭建

在 Eclipse 中是能够创建 OSGi 标准插件工程的，我们只要在 Eclipse 中创建插件工程的时候选择标准的 OSGi 插件工程即可。而前面在 Equinox 的例子中的工程都是标准的 OSGi 插件工程。所以前面开发的系统都是可以直接在 Felix 上运行的。我们在这节中就不再去开发新的应用了。下面主要介绍有关 Felix 下应用如何部署，以及如何在 Ecilpse 中调试的知识。

1.2.3 应用的部署

Bundle 的打包方式和前面在 Equinox 节介绍的是一样的，我们来看一下如何在 Felix 中启动这些

Bundle。

从 Felix 网站上下载二进制发行版后，把压缩包解开，可以看到如图 1-41 所示的目录结构。

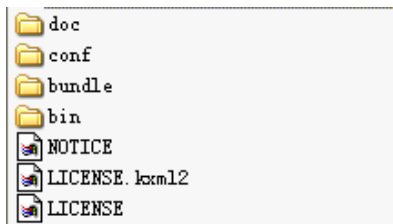


图 1-41 Felix 包含的内容

在 conf 目录下有一个 config.properties 的配置文件。可以在这个文件中设置自动启动的 Bundle。在 config.properties 中有 felix.auto.start.1= 这个配置，这个配置中写入的 bundle 就是在 Felix 启动的时候自动启动的 Bundle。

如果我们没有在 config.properties 中设置应用的 Bundle，那也没有关系，我们仍然可以在 Felix 的命令行窗口通过命令来安装我们应用的 Bundle。

1.2.4 在Eclipse中调试Felix

下面我们来看一下如何在 Eclipse 中启动 Felix 的环境来调试应用。有两个办法可以帮助我们做到这一点。一个是使用 Pax Cursor 插件。这个插件的 update site 地址是 <http://www.ops4j.org/pax/eclipse/update/>，在安装了这个插件后，我们在 Run Configurations 中的 OSGi Framework 配置的 Bundles 页签中，可以发现在 Framework 中除了 Equinox 外还多了其他的选择（见图 1-42）。

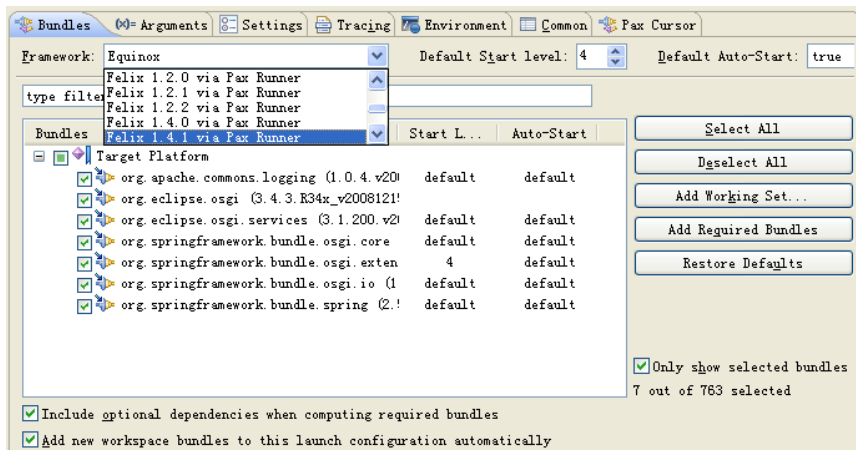


图 1-42 Framework 的选择

我们可以选择 Felix，这样我们的 OSGi 框架就是 Felix 而不是 Equinox 了（见图 1-43）。

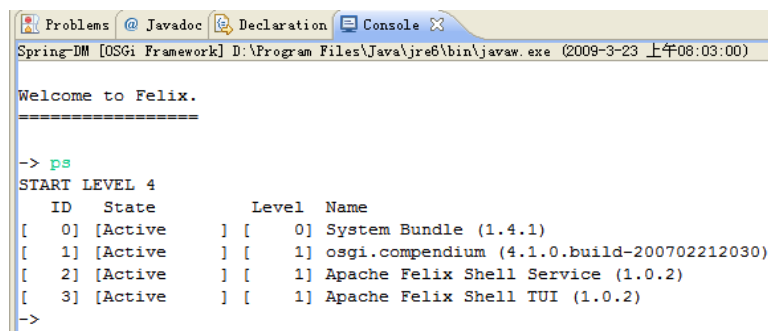


图 1-43 启动后的 Bundle 及状态

另外一个办法是，可以从 Felix 的网站下载一个 Felix 的二进制发行版，把这个发行版解压到一个临时目录中，或者从 Felix 的 trunk 上下载最新的代码自行编译 Felix。然后启动 Eclipse，按照下面的步骤来进行操作。

- 第一步，创建一个新的 Java 工程（见图 1-44）。

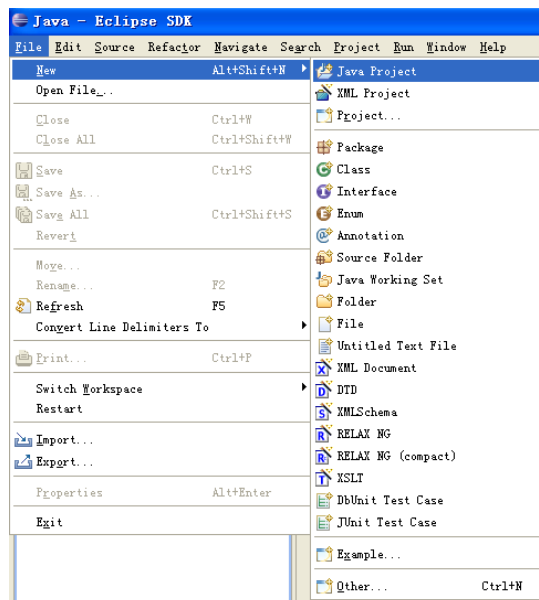


图 1-44 创建 Java 工程

- 第二步，工程设置（见图 1-45）。

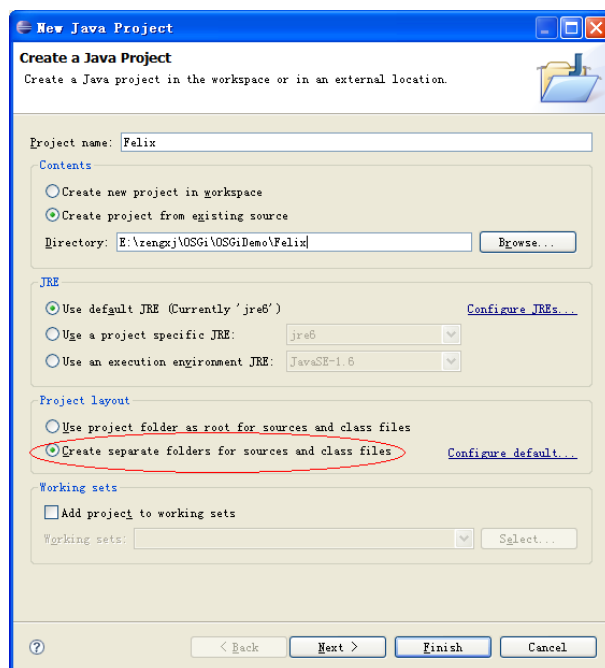


图 1-45 工程设置

完成后的工程如图 1-46 所示。

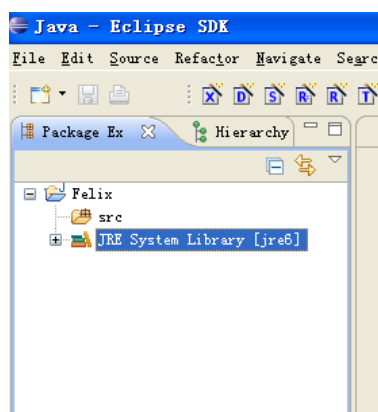


图 1-46 工程显示

- 第三步，修改默认 Output 文件夹（见图 1-47）。

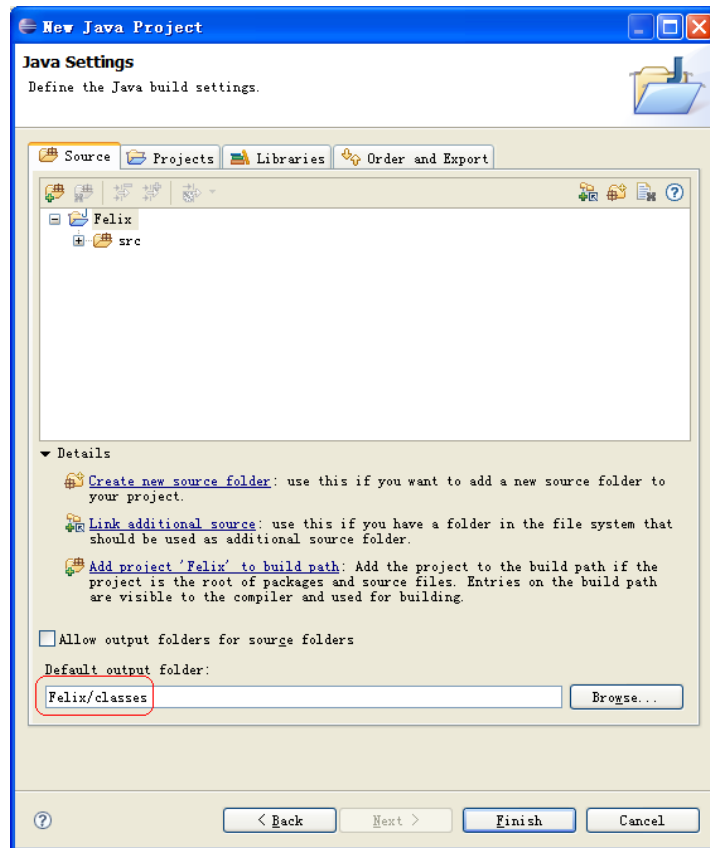


图 1-47 设置输出文件夹

- 第四步，复制 Felix 文件。

将刚才解压的 Felix 的发行版文件或自己编译的 Felix 文件复制到新创建的工程目录中。

完成后的工程显示如图 1-48 所示。

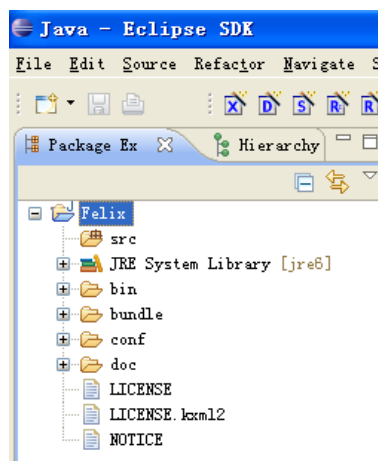


图 1-48 工程显示

- 第五步，将 felix.jar 加入到 Build path 中（见图 1-49）。

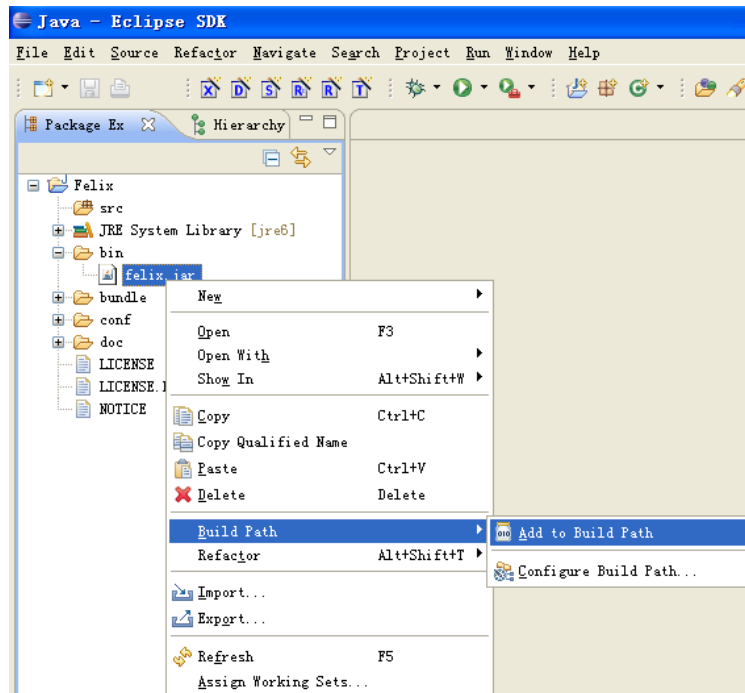


图 1-49 BuildPath 中加入 felix.jar

- 第六步，配置 Run Configurations（见图 1-50）。

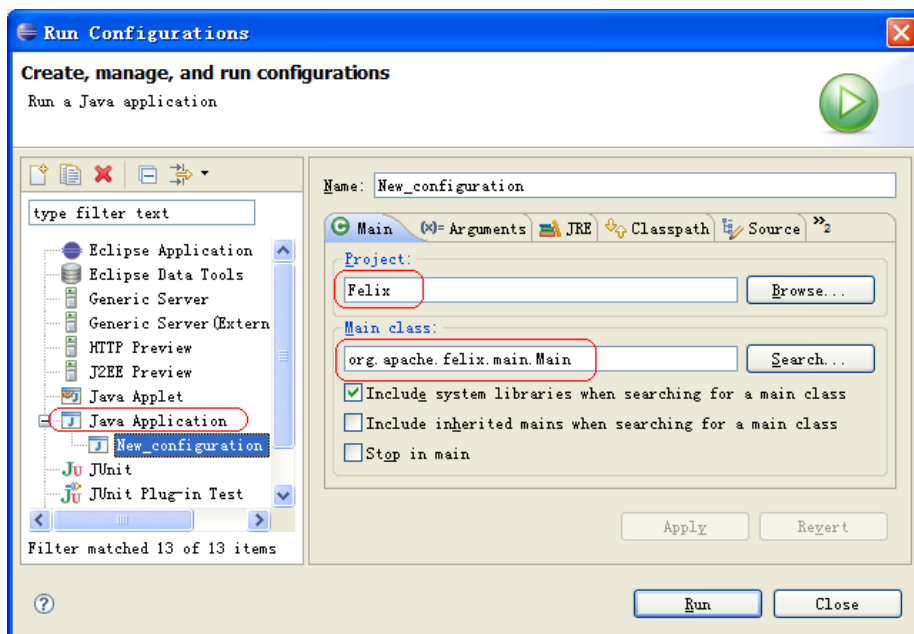


图 1-50 增加运行配置

然后就可以启动 Felix 了，运行后的结果见图 1-51。

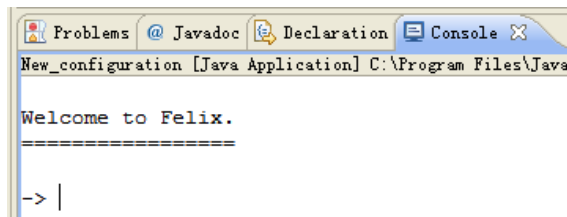


图 1-51 运行后的显示

如果我们要调试 Bundle，可以在 Bundle 的代码上打上断点，然后用 Debug 模式启动 Felix，再把 Bundle 部署到 Felix 中，运行到 Bundle 断点代码的时候，就会打开调试窗口了。

1.3 Spring-DM

1.3.1 简介

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。具体的信息可以查看 <http://static.springframework.org/osgi/docs/1.2.0-rc1/reference/html/>。

1.3.2 环境搭建

可以从 Spring 的网站下载最新的 Spring-DM 包。目前最新的版本为 1.2.0 RC1，下载地址在 <http://www.springsource.com/download/community?project=Spring%20Dynamic%20Modules&version=1.2.0.RC1> 这个页面上。它提供了 with-dependencies 和没有 dependencies 两个包的下载。我们建议大家下载 with-dependencies 的这个包，这个包里面包含了 Spring-DM 依赖的其他的包。下载了这个 Spring-DM 包后，我们把压缩包中的 dist 和 lib 目录解压到硬盘上的某个目录，比如解压到 C 盘根目录下的 spring-dm 目录中。那么我们会在 C:\Spring-dm\dist 目录下看到如图 1-52 所示的内容。在 C:\spring-dm\lib 中看到如图 1-53 所示的内容。

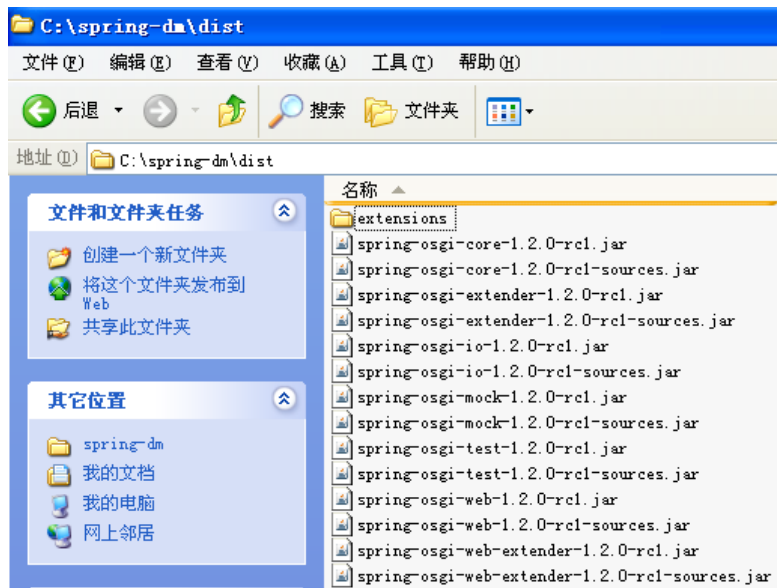


图 1-52 Spring-DM 中 Dist 的内容

下面把 spring-dm 的 bundle 导入到 Eclipse 中。

我们首先导入 spring-dm 的包。打开 Eclipse, 选择 Import..., 在 Import 的对话框中选择 Plug-ins and Fragments (见图 1-54)。

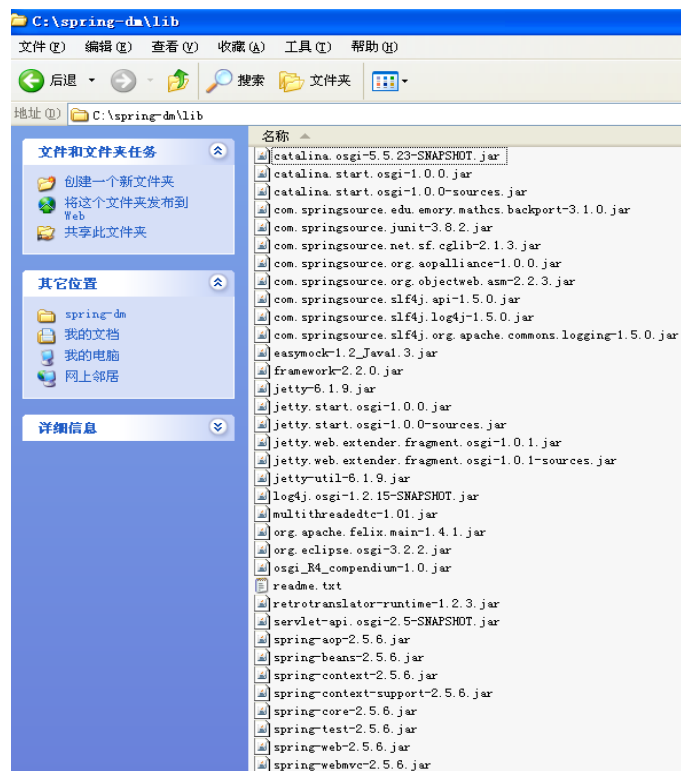


图 1-53 Spring-DM 中 lib 的内容

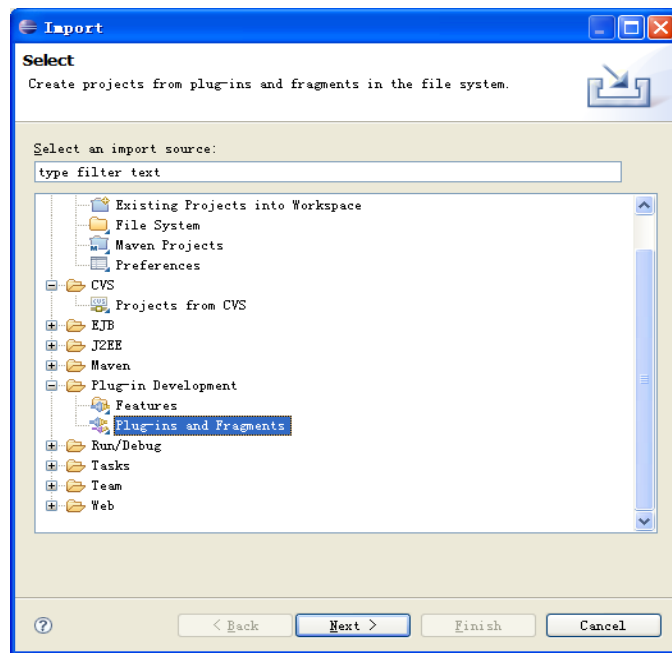


图 1-54 导入插件

然后在 Import Plug-ins and Fragments 对话框中做如图 1-55 所示的设置。

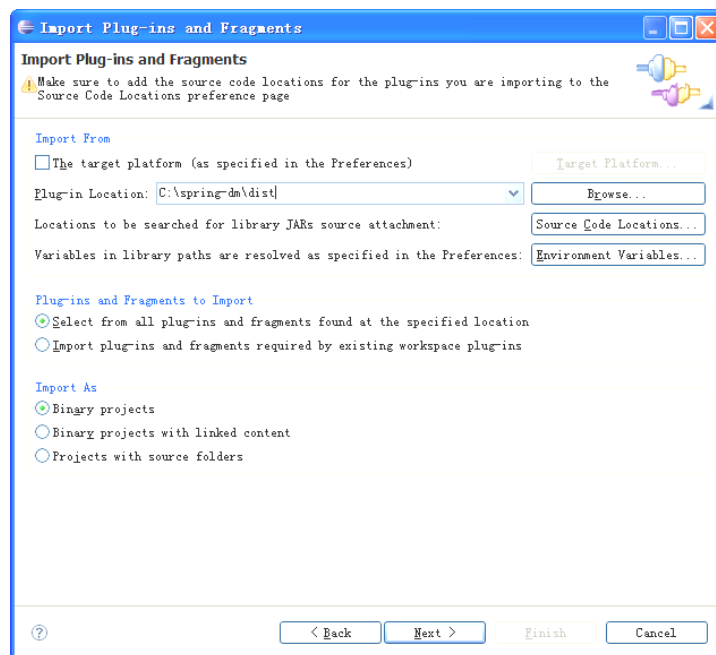


图 1-55 设置要导入插件的目录

我们要设置 Plug-in Location，先设置为 c:\spring-dm\dist，导入 spring-dm 的包。点击 Next 后，出现了让我们选择导入的 Plugin 界面（见图 1-56）。

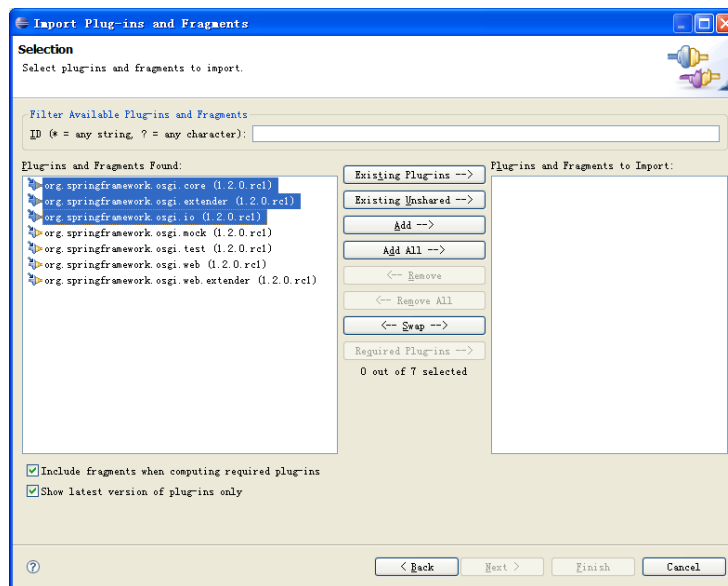


图 1-56 选择要导入的插件

我们可以导入 core、extender、io 三个 Bundle。完成后，会在 Eclipse 的工作区看到如图 1-57 的显示。

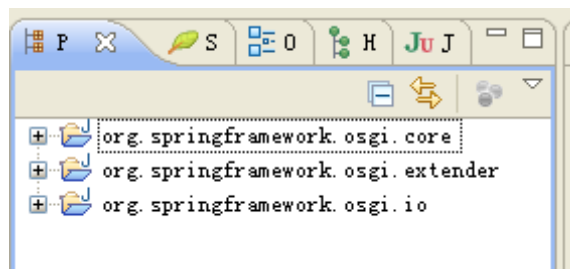


图 1-57 导入后的显示

我们直接在 Run Configurations 中选择这三个 Bundle，并执行，会发现三个 Bundle 都是 INSTALLED 状态，如果我们启动 Bundle，会报错，原因是我们没有加入这三个 Bundle 所依赖的 Bundle。而这些 Bundle，也就在 c:\spring-dm\lib 目录下。我们用和前面一样的方式来导入 lib 中所需要的 Bundle。要导入的 Bundle 是 com.springsource.org.aopalliance、org.springframework.aop、org.springframework.beans、org.springframework.context、org.springframework.context.support、org.springframework.core。

这个时候，在 Eclipse 的工作区看到的应该是如图 1-58 的显示。

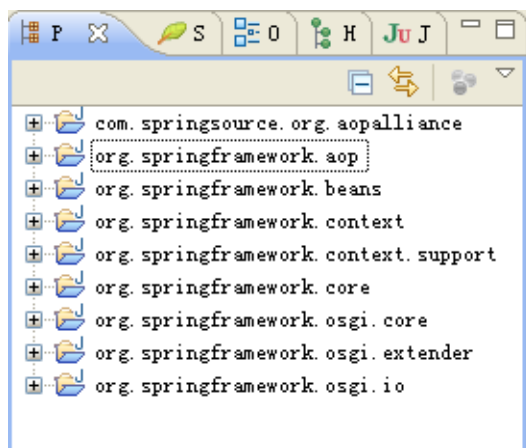


图 1-58 导入其他需要的插件

在 Run Configurations 中选择目前导入的所有 Bundle，然后再选择 Target Platform 中的 org.apache.commons.logging 这个 Bundle，运行。可以看到，所有的 Bundle 都是 ACTIVE 状态了（见图 1-59）。

到这里就完成了环境的搭建。下面就来做一个基于 Spring-DM 的应用。

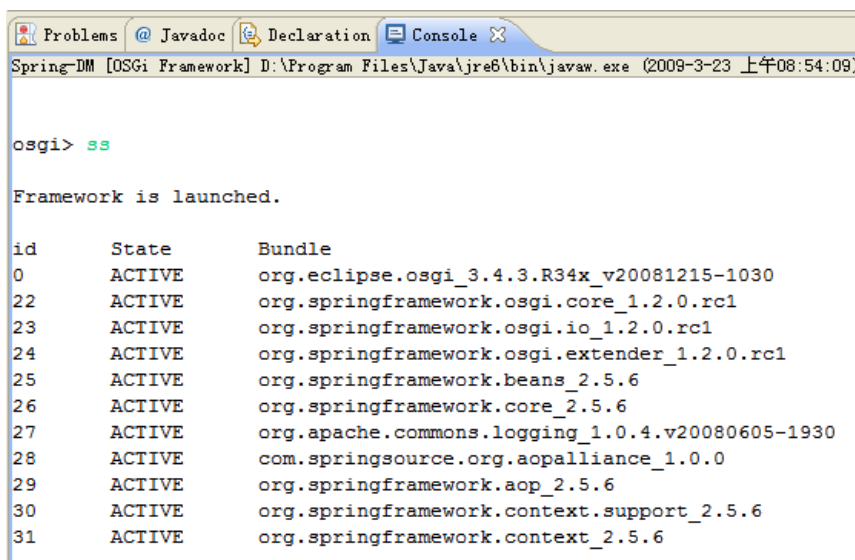


图 1-59 启动后的 Bundle 状态

1.3.3 HelloWorld

先看一个简单的例子。在这个例子中将展示如何把一个 Spring Bean 展示成服务，以及如何把 OSGi 的服务注入到 Spring 的 Bean 中。在这个例子中，有两个 Bundle，分别是 HelloWorld Bundle 和 TimeService Bundle。例子代码见源码 Spring-DM 中的 HelloWorld 工程和 TimeService 工程。

这两个工程中都没有 BundleActivator。在 Bundle 启动和停止的时候不用去做什么事情。先来看一下 TimeService 工程（见图 1-60）。

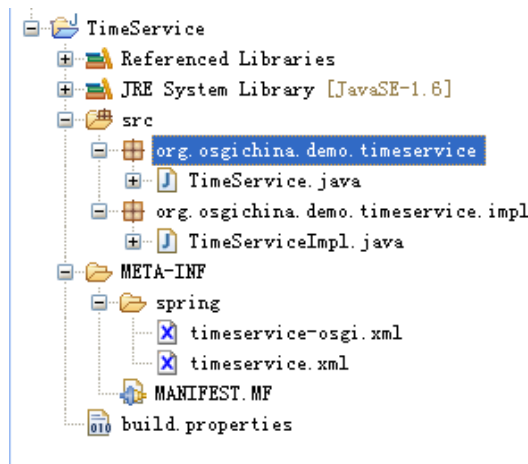


图 1-60 TimeService 工程

可以看到，我们在 TimeService 工程中定义了 TimeService 接口和 TimeService 的实现（TimeServiceImpl）。在这里，这样定义和实现并不是一个很好的做法，接口的定义应该放到一个单独的 Bundle 中，这里不过是为了示意，选择了一个简单的做法。TimeService 接口和 TimeServiceImpl 类的代码很简单，我们不再介绍。这个工程最大的不同是在 META-INF 目录下有一个 spring 目录。这个目录中有两个文件，分别是 timeservice.xml 和 timeservice-osgi.xml。我们分别看一下这两个文件的内容。

timeservice.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean name="timeService"
class="org.osgichina.demo.timeservice.impl.TimeServiceImpl">
    </bean>
</beans>
```

这个 xml 文件中定义了一个 spring bean。这个和通常的 spring 的配置文件没有任何区别。

timeservice-osgi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">
  <osgi:service id="osgiTimeService" ref="timeService"
    interface="org.osgichina.demo.timeservice.TimeService">
    </osgi:service>
</beans>
```

这里，我们看到了一个新的标签“<osgi:service>”，这个标签的含义是来定义一个 osgi 的 service。这个 service 实现的接口是“org.osgichina.demo.timeservice.TimeService”，另外这个 service 引用的对象是“timeService”，也就是刚才我们看到的上一个配置文件中定义的 Spring Bean。这样的一个配置，就可以把 spring bean 发布成一个 osgi 的 service 了。是不是很简单呢。

下面我们看一下 HelloWorld 工程（见图 1-61）。

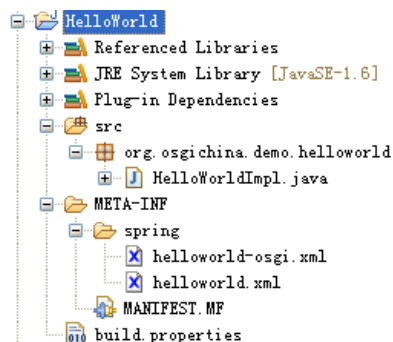


图 1-61 HelloWorld 工程

可以看到，HelloWorld 文件中只有 HelloWorldImpl 这样一个 Java 文件，并且在 META-INF 下也躲了一个 spring 的目录，目录中有 helloworld-osgi.xml 和 helloworld.xml 文件。我们简单看下 HelloWorld 的代码。

```
public class HelloWorldImpl{
    private TimeService timeService;

    public TimeService getTimeService() {
        return timeService;
    }

    public void setTimeService(TimeService timeService) {
        this.timeService = timeService;
    }

    public void start(){
        System.out.println("started at " +
            timeService.getCurrentTime());
    }

    public void stop(){
        System.out.println("stopped at " +
            timeService.getCurrentTime());
    }
}
```

可以看到，HelloWorld 是一个简单的 bean，在调用 start 和 stop 的时候会有日志的输出。

好，接下来看 helloworld.xml 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="hello" class="org.osgichina.demo.helloworld.HelloWorldImpl"
        init-method="start" destroy-method="stop" >
        <property name="timeService" ref="osgiTimeService"/>
    </bean>
</beans>
```

可以看到，这个配置文件和普通的 spring 配置文件是一样的，定义了一个 bean，以及 init 和 destroy 的方法。并且注入了一个对象给 timeService 这个属性。下面，我们来看一下 helloworld-osgi.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <osgi:reference id="osgiTimeService"
interface="org.osgichina.demo.timeservice.TimeService"/>
</beans>
```

注意，这里也多了一个新的标签“osgi:reference”，这个是对 OSGi 服务的一个引用。这里要定义一下接口名称，因为这是用来表示引用的服务的。

现在，我想大家已经明白了如何用 spring-DM 来把一个 spring bean 发布为服务，以及如何引用一个 OSGi 服务作为 bean 的属性了吧。这里，可能大家有一个疑问是，如果 timeService 不是注入到 spring bean 里面，而是要在代码中获取这个服务，那该怎么办呢？很简单，我们只要通过 BundleContext 来获取这个服务就好了。服务的名字就是接口的名字，换句话说，对于这个例子中的 timeService，我们通过下面的代码就能获得这个服务：

```
ServiceReference sf = context.getServiceReference(TimeService.
class.getName());
TimeService ts = (TimeService)context.getService(sf);
```

到这里，例子已经完成了。在这里简单介绍一下这背后的实现。其实是 spring 的 extender 这个 bundle 帮我们完成了服务的发布和注入的工作。这个 bundle 会寻找 META-INF 下 spring 目录中的所有 xml，完成配置工作。当然，我们也可以在 MANIFEST.MF 文件中配置 spring-dm 用到的配置。而 spring 目录则是一个默认的约定。完成了这个例子，下面我们要完成一个 Spring-DM 用在 Web 场景下的例子。

1.3.4 Web版HelloWorld

前面一节已经看到了 Spring-DM 的 HelloWorld 的例子。这里，来看一下 Web 版的 HelloWorld。这里主要是想介绍在 Spring-DM 中的另外两个 Bundle，一个是 org.springframework.osgi.web，另外一个 org.springframework.osgi.web.extender。大家还记得，在上一节的例子中，我们用到了 org.springframework.osgi.extender 这个 Bundle。这里的 org.springframework.osgi.web.extender 这个 Bundle 是用在 Web 环境中的一个 Extender。

其实构造 Web 版的 HelloWorld 有多种方法。一种方式是可以在前面的 HelloWorld 的例子中通过嵌入 HttpServer 来完成一个 Web 应用。而这一节不采用这样的方案，我们采用 Spring-DM 的 WebExtender 来部署一个 Web 程序。

在 Eclipse 中有两个方案来创建我们的工程：一个是创建一个标准的 Web 工程，这样的做法有一个不方便的地方是调试的问题；另外一个办法是创建一个插件工程，而这个工程打包后是拥有 Web 应用结构的 jar 包。下面，我们就来动手完成这个例子。

1.3.4.1 环境搭建

我们在完成 Spring-DM 的 HelloWorld 的时候，搭建了一个环境。现在，要在那个环境的基础上

再引入几个 Bundle，分别是 `org.springframework.osgi.catalina.osgi`、`org.springframework.osgi.catalina.start.osgi`、`org.springframework.osgi.servlet-api.osgi`、`org.springframework.osgi.web`、`org.springframework.osgi.web.extender`。其中后面两个 Bundle 是从 `dist` 目录下导入的，前面三个是从 `lib` 目录下导入的。导入这些 Bundle 后，我们的 Eclipse 看起来应该是如图 1-62 所示的样子。

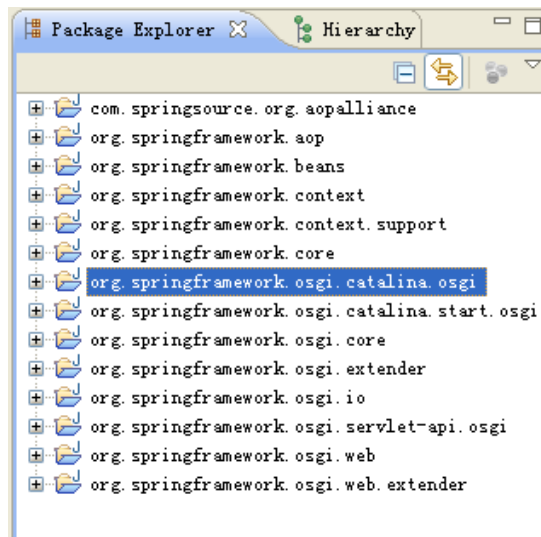


图 1-62 加入新插件后的显示

接着，我们要修改 Run Configurations，设置 `org.springframework.osgi.web.extender` 的 Start Level，需要把这个 Bundle 的 Start Level 修改得比别的 Bundle 大。另外就是在 Target Platform 中要选择 `javax.servlet` 和 `org.apache.commons.logging` 这两个 Bundle。如图 1-63 所示。

然后点击“Run”，可以在 Eclipse 的 Console 中看到类似如图 1-64 所示的显示。

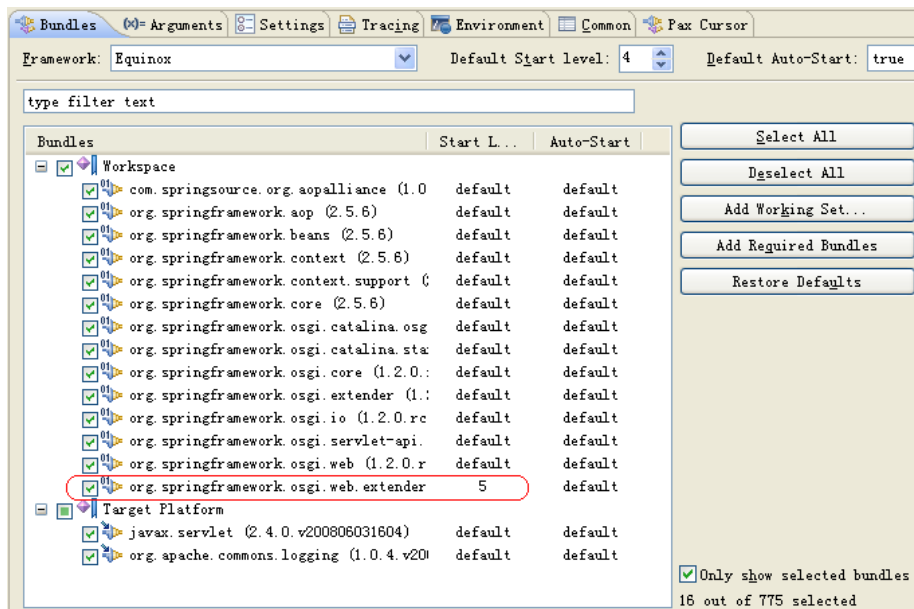


图 1-63 运行配置的 Bundles


```

2009-3-26 7:58:46 org.apache.coyote.http11.Http11BaseProtocol init
信息: Initializing Coyote HTTP/1.1 on http-8080
2009-3-26 7:58:46 org.apache.catalina.startup.Catalina load
信息: Initialization processed in 265 ms
2009-3-26 7:58:46 org.apache.catalina.core.StandardService start
信息: Starting service Catalina
2009-3-26 7:58:46 org.apache.catalina.core.StandardEngine start
信息: Starting Servlet Engine: Apache Tomcat/5.5.23
2009-3-26 7:58:46 org.apache.catalina.core.StandardHost start
信息: XML validation disabled
2009-3-26 7:58:46 org.apache.coyote.http11.Http11BaseProtocol start
信息: Starting Coyote HTTP/1.1 on http-8080
2009-3-26 7:58:46 org.springframework.osgi.web.tomcat.internal.Activator$1 run
信息: Successfully started Apache Tomcat/5.5.23 @ Catalina:8080
2009-3-26 7:58:46 org.springframework.osgi.service.importer.support.internal.aop.
信息: Found mandatory OSGi service for bean []
2009-3-26 7:58:46 org.springframework.osgi.web.tomcat.internal.Activator$1 run
信息: Published Apache Tomcat/5.5.23 as an OSGi service
2009-3-26 7:58:46 org.springframework.osgi.web.deployer.tomcat.TomcatWarDeployer
信息: Found service Catalina

```

图 1-64 运行后的显示

这个时候，也可以查看到 8080 端口已经是在 LISTEN 的状态。我们的 Web 版环境准备结束，下面就来进行开发。

1.3.4.2 Web应用开发

Web 应用开发步骤如下。

- 第一步，创建插件工程。

我们创建一个名为 HelloWorldWeb 的插件工程。

- 第二步，添加 WEB-INF 目录和 index.html 文件，并且在 WEB-INF 目录中加入 web.xml 文件。

这样，我们的工程看起来应该如图 1-65 所示。

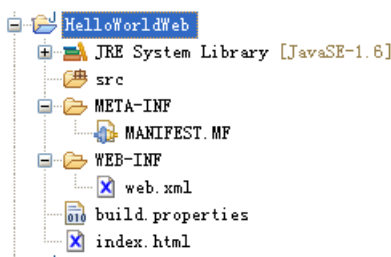


图 1-65 HelloWorldWeb 的显示

- 第三步，引入 Package（见图 1-66）。

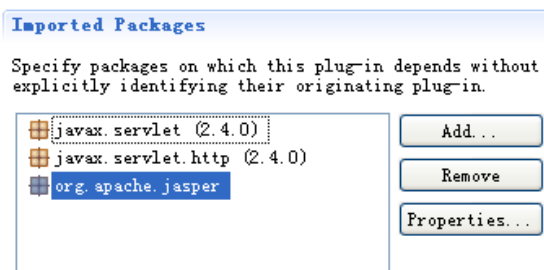


图 1-66 引入的 Package

- 第四步，创建 HelloWorldServlet。代码请见源码中 HelloWorld-SpringDM 目录下的 HelloWorldWeb 工程。并且，我们在这里也要引入前面 HelloWorld 例子中的 TimeService。

- 第五步，编辑 index.html。

这里，我们可以随便在 index.html 中搞点儿什么你喜欢的。

- 第六步，编辑 web.xml。

这个 web.xml 基本就是普通的 Web 应用中的配置，在这个 web.xml 中，我们加入了 Spring 的 ApplicationContext 的配置：

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>org.springframework.osgi.web.context.support.
    OsgiBundleXmlWebApplicationContext</param-value>
</context-param>
```

完整的内容可以参考源码中 HelloWorld-SpringDM 目录下 HelloWorldWeb 工程中的 web.xml。

- 第七步，添加 applicationContext.xml。我们在 WEB-INF 目录下添加 applicationContext.xml 文件，这里的配置是为了引入 TimeService，内容如下：

```
<!--引入OSGi Service的定义-->
<osgi:reference id="osgiTimeService"
interface="org.osgichina.demo.timeservice.TimeService"/>
```

- 第八步，配置 Run Configurations。

我们要在 Run Configurations 的 Target Platform 中加入三个 Bundle，分别是 javax.servlet.jsp、org.apache.jasper、org.apache.commons.el。

- 第九步，启动，运行。

我们会看到 Console 中会输入类似下面的信息：

```
信息: Successfully deployed bundle [HelloWorldWeb Plug-in (HelloWorldWeb)] at
[/HelloWorldWeb] on server org.apache.catalina.core.StandardService/1.0
```

至此，可以在浏览器上测试一下我们的应用了。

打开浏览器，输入 `http://localhost:8080/HelloWorldWeb/`。

会看到如图 1-67 所示的显示。

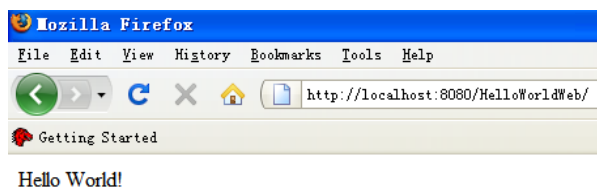


图 1-67 浏览器结果显示

然后，再输入 `http://localhost:8080/HelloWorldWeb/hello`。

将会看到如图 1-68 所示的内容。

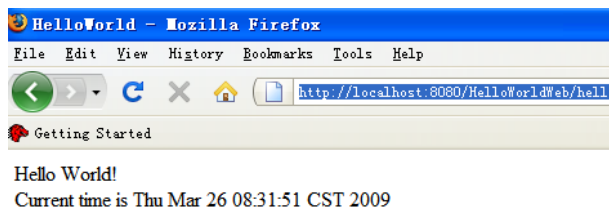


图 1-68 浏览器结果显示

好了，我们的 Web 应用已经可以正常工作了，并且也可以方便地在 Eclipse 的集成环境中调试我们的程序。

第 2 章 基于Spring-DM实现Petstore

在前面一章，我们分别介绍了 Equinox、Felix 和 Spring Dynamic Modules，并完成了简单的 HelloWorld 程序。接下来将通过 Spring Dynamic Modules 完成一个简化的 PetStore 例子。

2.1 “即插即用”的Petstore

2.1.1 Petstore的功能需求

Petstore 是 J2EE 的 BluePrints 中提到的一个例子，里面涉及了很多 J2EE 的规范及使用，最新的 Petstore 实现中也引入了 Ajax 的技术。也有其他的语言实现了 Petstore 并和 J2EE 的实现进行比较。在本章中，我们并不想去完整地做一个 Petstore 的实现，而是想去实现一个从功能上说简化版的 Petstore。而我们的重点是希望给大家展现一个全新的“即插即用”的 Petstore。

首先来看这次要实现的 Petstore 包含的功能。

- 类目产品展示
 - 列出所有的类目，并且配有类目的图片。
 - 列出类目下所有产品的名称。
 - 在页面上提供点击某个产品后进入到产品详细信息页面的功能。
 - 产品详细信息中列出产品所属的类目、产品名称和属于该产品的具体条目。
 - 每个条目后可以选择将该条目放入到购物车中。
 - 提供返回到类目列表的按钮。
- 购物车
 - 显示目前购物车中的条目列表。
 - 每个条目显示类目图片，条目的 ID、名称、描述、单价、数量和小计。
 - 显示总计的价格。
 - 提供回到类目列表继续购物的按钮。
 - 能够删除购物车中的某个条目。
 - 能够修改购物车中的某个条目的数量。
- 库存管理
 - 列出类目。
 - 列出类目中包含的产品。

- 点击产品进入到产品的详细信息页面。
- 列出产品包含的具体条目的列表。
- 显示产品中每个条目的 ID、单价、描述、库存数量等信息。

在这个例子中没有包含用户的管理和权限的管理及支付的功能，这几个部分可以作为课后的作业，给广大读者在已有的 Petstore 的基础上去扩展完成。

所谓“即插即用”的 Petstore，就是要我们实现的 Petstore 中的功能模块能够动态地加载和卸载。也就是希望模块加载了，这个功能就可用，而模块卸载了，这个功能就停用。这就要求功能是模块化的，并且是可动态插拔的模块化而不仅仅是静态设计上的模块化。

2.1.2 OSGi 框架的功能和设计思想

下面，我们先来看 OSGi 框架能提供的基础功能。

1. 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）动态地部署至框架中，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，那就是将工程创建为符合规范的 Bundle 工程（就是 Eclipse 中的插件工程），或者使用工具将工程打包成符合规范的 Jar 文件。

2. 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。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。

基于 OSGi 的 Event 服务也是实现模块交互的一种可选方法，模块对外发布事件，订阅了此事件的模块就会相应地接收到消息，从而做出处理。

3. 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间进行增加、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。

4. 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。

那么，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：

5. 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体验到何谓模块化，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得还是很不足。

基于 OSGi 进行模块化设计和传统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，相信大家在这点上很容易适应，在 OSGi 中只是更为规范，更为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 及 OSGi Service 的方式实现。

6. 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、对外提供的服务及服务的属性。

7. 动态化的设计

动态化的设计是指系统中所有的模块均须支持动态的插拔和修改，系统的模块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地依赖任何服务，避免服务不可用时造成系统的崩溃，从而保证系统的“即插即用，即删即无”。

8. 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，它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 组件相应地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，若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，这要结合实际情况来进行设计。

2.1.3 Petstore的设计

2.1.3.1 数据库表设计

从前面列出的 Petstore 的功能看，我们要保存的数据有：类目数据，产品数据，产品的具体条目数据，购物车数据。在这个 Petstore 的例子中，我们购物车的数据采用非持久的方式来保存（有兴趣的读者可以改造这部分，完成购物车数据的持久保存）。然后剩下的三种数据，我们采用三张表来存储。我们先来看以下三种数据的关系，如图 2-1 所示。

数据库表设计 Category 和 Product 是 1 对多的关系，而 Product 和 Item 也是 1 对多的关系（见表 2-1~表 2-3）。

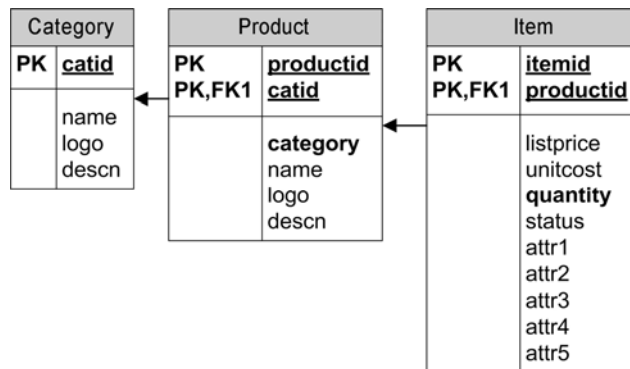


图 2-1 Category、Product 与 Item 三种数据的关系

表 2-1 Category 表字段设计

字段名称	类型	是否允许为空	是否主键	是否外键	描述
catid	varchar(10)	N	Y	N	类目标识
name	varchar(80)	Y	N	N	类目名称
logo	varchar(80)	Y	N	N	类目图片
descn	varchar(255)	Y	N	N	类目描述

表 2-2 Product 表字段设计

字段名称	类型	是否允许为空	是否主键	是否外键	描述
productid	varchar(10)	N	Y	N	产品标识
category	varchar(10)	N	N	Y	所属类目
name	varchar(80)	Y	N	N	产品名称
logo	varchar(80)	Y	N	N	产品图片
descn	varchar(255)	Y	N	N	产品描述

表 2-3 Item 表字段设计

字段名称	类型	是否允许为空	是否主键	是否外键	描述
itemid	varchar(10)	N	Y	N	条目标识
productid	varchar(10)	N	N	Y	所属产品
listprice	decimal(10,2)	Y	N	N	价格
unitcost	decimal(10,2)	Y	N	N	成本

字段名称	类型	是否允许为空	是否主键	是否外键	描述
quantity	int	N	N	N	数量
status	varchar(2)	Y	N	N	状态
attr1	varchar(80)	Y	N	N	属性

续表

字段名称	类型	是否允许为空	是否主键	是否外键	描述
attr2	varchar(80)	Y	N	N	属性
attr3	varchar(80)	Y	N	N	属性
attr4	varchar(80)	Y	N	N	属性
attr5	varchar(80)	Y	N	N	属性

2.1.3.2 模块层次划分

完成了数据库表的结构设计，我们接下来对模块的层次做一个划分。

Petstore 是一个 B/S 系统，一般来说，B/S 系统可以分为三个层次：展现层、业务逻辑层、数据访问层。因为在我们的需求中没有很复杂的业务逻辑，所以我们可以省去业务逻辑层，因此我们把系统的模块分为展现层和数据访问层两个部分，展现层用于和用户进行交互，实现对用户请求的处理和页面的展现，而数据访问层提供数据访问服务。

2.1.3.3 模块划分

根据需求分析，我们将数据的访问分为两个部分。

➤ 产品数据

对类目数据、产品数据、条目数据的存取。

➤ 购物车数据

购物车信息的访问的存取。

而展现层我们可以分为三个部分。

➤ 产品展示

- 显示类目列表。
- 显示类目中包含的产品。
- 显示产品中包含的条目。
- 购买产品。

➤ 产品维护

- 显示类目列表。
- 显示类目中包含的产品。
- 显示产品中包含的条目。

➤ 购物车展示

- 显示购物车中的条目。
- 修改已经购买的产品的数量。
- 删除已经购买的产品。

按照上面的分析，我们把展现层和数据访问层的功能分为 5 个模块。

在传统的 B/S 应用中，我们是将整个应用打成一个 war 包，部署在应用服务器下。在 OSGi 中，可以通过 `HttpService` 或通过 `Equinox` 的 `Bridge` 方式和 Web 应用服务器集成来部署 Web 应用。我们在 `Petstore` 的例子中采用直接通过 OSGi 的 `HttpService` 来注册 Web 应用，提供服务。

OSGi 通过 `Http Service` 将相应的 `Servlet` 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，在前面的章节，我们已经看到过这样的例子。在 `Petstore` 的例子中，我们有一个专门的 `ControllerServlet`，统一响应用户的请求，然后从请求中获取 URI，通过 URI 查找对应的处理类（`ActionHandler`）来进行后续的处理。并且，`ControllerServlet` 本身也控制了整个页面的布局，我们把页面从上至下分为三个部分：`Header`、`PageContent`、`Footer`。`ActionHandler` 要对用户请求进行处理，然后返回 `PageContent` 的内容。`Header` 中，我们将会显示菜单式的链接入口，以切换不同的页面显示，`Footer` 主要显示一些版权信息。

对于页面的展示，因为我们展现的功能并不是非常的复杂，所以我们没有选择使用模板引擎，而是在 `ActionHandler` 中完成对页面的构建。为了简化处理，我们增加了一个 `Bootstrap` 模块，这个模块向 `HttpService` 中注册了 `ControllerServlet`，并且处理了 `Header` 和 `Footer` 的展现。

另外，在 `Bootstrap` 模块中，除了前面提到的功能外，还要处理 `Header` 中的模块菜单显示以及系统默认页面的显示。我们的 `Petstore` 是一个即插即用、即拔即停的系统，所以系统默认的页面以及菜单是动态的。这个功能的实现也放到了 `Bootstrap` 中。另外，我们还添加了一个 `utils` 模块，把模块中通用的功能部分放到了 `utils` 这个模块儿中。

通过我们对系统的分析，我们一共要实现 7 个功能模块。分别是：

- 启动和控制模块——`Bootstrap`
- 产品展示——`ProductList`
- 库存管理——`ProductManagement`
- 购物车——`ShoppingCart`
- 产品数据管理——`ProductDal`
- 购物车数据管理——`ShoppingCartDal`
- 工具——`Utils`

在下一节，我们来完成模块的设计。

2.1.3.4 模块设计

`Petstore` 是一个 Web 应用，我们知道，构建基于 OSGi 的 Web 应用有两种方式，一种是采用 `Bridge` 方式进行部署，也就是把 OSGi 框架嵌入到 Web 服务器中，另外一种是把 Web 服务器嵌入到 OSGi 容器中，我们这次采用的是把 Web 服务器嵌入到 OSGi 容器中的方式。

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

- 部署 Bootstrap 模块，这个时候，可以看到应用的页面，但是页面只有一个简单的 Header 和 Footer，没有内容的显示。
- 部署 Utils 模块，这是一个基础模块，在外部展示我们看不到变化。
- 部署 ProductDal 模块，这是一个基础模块，在外部展示我们看不到变化。
- 部署 ShoppingCartDal 模块，这是一个基础模块，在外部展示我们看不到变化。
- 部署购物车模块，这个时候我们可以在 Header 上看到“我的购物车”的链接。

并且可以进入到展示购物车的页面。

- 部署 ProductManagement 模块，这个时候我们可以在 Header 上看到“库存维护”的链接。并且可以进入到库存维护的页面。

可以进入到具体产品的页面。

- 部署 ProductList 模块，这个时候我们可以在 Header 上看到“宠物类目”的链接。并且可以进入到宠物类目的页面。

可以进入到具体产品的页面。

可以将产品条目加入到购物车中。

好，这个时候，我们整个的系统是部署完毕了。我们看一下模块之间的关系（见图 2-2）。

分析完模块之间的关系后，我们就要来结合各个模块具体要实现的功能进行模块自身的设计。

1. Bootstrap模块

这个模块是 Petstore 中的一个核心的控制模块。在 Bootstrap 模块中，我们做了如下几个事情：

- 总控的 Servlet。
- 定义 PageHeader, PageFooter 接口，用于扩展页面的头部和底部的实现。
- 定义 DefaultPage 接口，用于进行默认页面的处理。
- 定义 MenuItem 接口，用于其他模块注册自己模块需要的菜单。

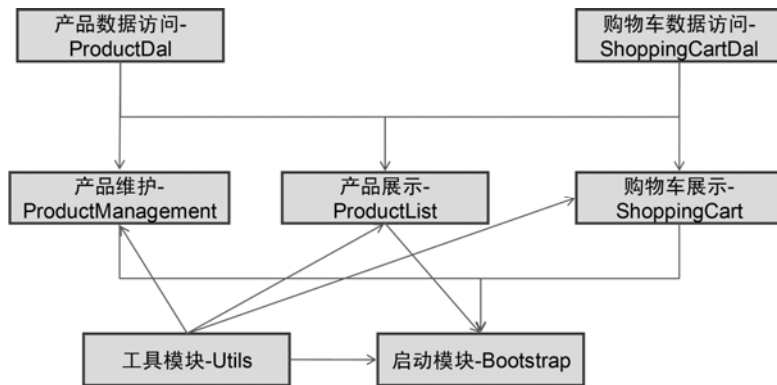


图 2-2 系统模块设计

- 定义 ActionHandler 接口，用于其他模块注册自身提供服务的 URI，以及对应的 ActionHandler。
- 启动了 HSQLDB 数据库，并且进行了数据初始化工作。
- 提供了 PageHeader 的实现。
- 提供了 PageFooter 的实现。

- 注册了自身管理的静态资源。

看到上面的列表，可以看到 Bootstrap 模块做的事情还是比较多的，下面就来分别看一下这些功能的设计。

■ ControllerServlet

在整个应用中（包括其他的模块），我们只有这么一个 Servlet，用户的请求（除了资源的请求）都会进入到这个 Servlet。在这个 Servlet 中，我们首先是向 `HttpService` 注册了自身。并且管理了 `PageHeader`、`PageFooter` 的实现，各个模块提供的 `DefaultPage`，以及各个模块提供的 `ActionHandler` 和 `URI` 的映射。

`ControllerServlet` 定义了页面显示的格式，分为上中下三个部分，上面是 `Header`，下面是 `Footer`，而中间就是根据请求来显示的不同内容，而如果没有找到 `URI` 对应的 `ActionHandler`，则会显示默认的面。

■ PageHeader 接口

`PageHeader` 接口中定义了两个方法：

```
String getHeadInfo(String resourcePath);
```

这个方法返回的是放在 `HTML` 页面中的 `<head></head>` 之间的内容，一般我们是可以放 `css` 文件的连接等信息。这个方法有一个参数 `-resourcePath`，这个参数的意思是资源文件的完整路径的前缀。方便 `PageHeader` 的实现者能够生成正确的 `URI`。

```
String getPageHeader(String servletPath, String resourcePath);
```

这个方法返回的就是 `Header` 本身内容对应的 `HTML`。两个参数，其中第二个 `resourcePath` 和前一个方法中的 `resourcePath` 含义相同。第一个 `servletPath` 的含义是 `Petstore` 应用的 `URI` 的前缀，方便应用生成正确的链接。

■ PageFooter 接口

`PageFooter` 接口中定义了两个方法：

```
String getHeadInfo(String resourcePath);
```

这个方法返回的是放在 `HTML` 页面中的 `<head></head>` 之间的内容，一般我们是可以放 `css` 文件的连接等信息。这个方法有一个参数 `-resourcePath`，这个参数的意思是资源文件的完整路径的前缀。方便 `PageHeader` 的实现者能够生成正确的 `URI`。

```
String getPageFooter(String servletPath, String resourcePath);
```

这个方法返回的就是 `Footer` 本身内容对应的 `HTML`。两个参数，其中第二个 `resourcePath` 和前一个方法中的 `resourcePath` 含义相同。第一个 `servletPath` 的含义是 `Petstore` 应用的 `URI` 的前缀，方便应用生成正确的链接。

■ DefaultPage 接口

`DefaultPage` 接口中定义了两个方法：

```
String getUri();
```

这个方法返回了一个 `Uri`，也就是如果当前请求的 `Uri` 没有对应的 `ActionHandler`，则会使用这个 `Uri` 作为替代的 `Uri` 进行处理。

```
int getPriority();
```

这个方法返回了一个整型值，越小表明这个 `DefaultPage` 的优先级越高，在不同模块注册的 `DefaultPage` 之间，是通过 `Priority` 的值来进行比较的。

■ MenuItem 接口

这个接口只包含了一个方法：

```
MenuItemInfo getMenuItemInfo();
```

这个方法返回了一个 `MenuItemInfo` 对象，代表了一个菜单项。`MenuItemInfo` 中主要包含了如表 2-4 所示的内容。

表 2-4 MenuItemInfo 内容

名称	描述
caption	菜单显示的名称
imgURL	菜单的图片 URL，如果不设置，则表示菜单没有图片
position	菜单的位置，越小的越靠左
url	菜单指向的 URL

一个模块可以实现一个或多个的 `MenuItem`。

■ ActionHandler 接口

这个接口包含了两个方法：

```
String handleRequest(HttpServletRequest request, String servletPath, String resourcePath);
```

这个方法用于生成响应的 HTML 内容，有三个传入的参数，`request` 包含了请求的内容，`servletPath` 同前面我们看到的 `servletPath`，是 `petstore` 应用的 URI 的前缀。`resourcePath` 是静态资源请求的前缀，用于生成正确的静态资源的链接。

```
String getHeadInfo(String resourcePath);
```

这个方法和 `PageHeader` 及 `PageFooter` 中的那个同名方法的意义是一样的，用于获取这个页面要在 HTML 的 `<head></head>` 加入的内容。模块通过实现一个或多个 `Action-Handler` 来提供对 URI 的响应。

■ HSQLDB 数据库

为了简化 `Petstore` 的环境，我们没有使用 `Mysql`、`Oracle` 这样类型的数据库，我们使用的是 `HSQLDB`。在应用启动的时候，我们会启动 `HSQLDB`，并且完成数据库的初始化工作。

■ PageHeader 的实现

在 `Bootstrap` 中，我们定义了 `PageHeader` 接口，并且也提供了一个默认的实现。在这个默认的 `PageHeader` 实现中，我们主要实现了模块注册的菜单展示。

■ PageFooter 的实现

相对于 `PageHeader` 的实现，`PageFooter` 的实现非常简单，只是输出了一行版权的信息。

■ 注册自身用到的静态资源

在 `Bootstrap` 模块中的 `PageHeader` 实现里面，要用到静态的图片及 `css` 的文件，所以我们要注册静态资源到 `HttpService` 中。这个工作是通过 `Utils` 工程中的一个辅助类来完成的。在 `Utils` 工程中，我们会介绍这个类，并且这个类也被其他要注册静态资源的模块所使用。

2. Utils 模块

`Utils` 模块主要提供了多个模块都需要的功能的实现，以方便其他功能模块的功能编写。`Utils` 主要实现了两个功能：

- 模块 Web 静态资源注册管理。完成模块内部的 Web 静态资源向 `HttpService` 的注册和注销。注册管理器实现了 `ServiceListener` 接口，注册了对 `HttpService` 服务变化的感知。在服务发生变化的时候，根据变化的类型来进行相应的动作。

- Web 应用的配置管理。这里面的管理主要是 Web 应用的 Servlet 的 URI 前缀和静态资源的 URI 前缀的配置。这个配置管理器在整个应用中是同一个对象，由 Bootstrap 模块提供。不同模块使用同样的 Web 应用配置管理，能够保证模块生成的链接和资源链接的一致和可移植。

3. ProductDal模块

这个模块是数据访问的模块，提供了数据的存取服务。在这里，我们没有采用 Hibernate 或 iBatis 这样的框架，而是简单地使用 JDBC 的方式直接和数据库交互。并且定义了供其他模块使用的和持久数据相关的数据对象（DataObject）。ProductDal 是一个底层的模块，对外提供服务，而自身并不依赖于其他的模块。

ProductDal 中的一个示意的类图如图 2-3 所示。

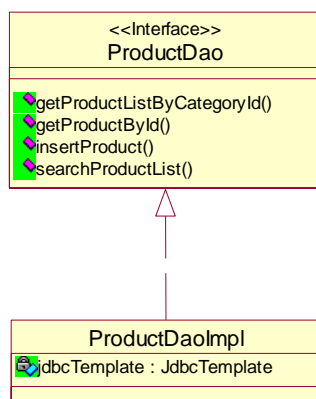


图 2-3 ProductDao 接口和其实现

其他的几个 Dao 接口和实现的类图，我们这里就不再一一列出了。

4. ShoppingCartDal模块

ShoppingCartDal 提供了用户的购物车数据管理的功能。思路和 ProductDal 模块是类似的，只是在实现上，我们实现了一个基于内存的 ShoppingCart 数据管理，也提供了外部使用的数据对象。感兴趣的读者，可以改写 ShoppingCartDal 模块，可以考虑使用 iBatis 或者 Hibernate 来实现 ShoppingCartDal 中的 Dao 接口，提供持久的数据存取服务。

5. ProductList模块

ProductList 模块提供了类目列表页面，页面上显示了所有的类目，以及类目中包含的产品。点击产品的链接，会进入到产品信息的面，列出了该产品下包含的具体条目，也显示了产品和条目相关的信息，用户可以在这个页面中把选中的条目加入到购物车中。这里，ProductList 模块要用到 ShoppingCartDal 和 ProductDal 中的一些 Dao 接口的实现，用于访问相关的数据。在 ProductList 中，要实现多个 ActionHandler，来提供对页面请求的处理。并且提供了菜单给 Bootstrap 中的 PageHeader 模块。用于展示菜单。我们来看下 ProductList 模块中的类的类图。

DefaultPage 的实现，如图 2-4 所示。

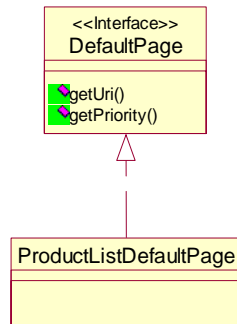


图 2-4 DefaultPage 接口及实现

MenuItem 的实现，如图 2-5 所示。

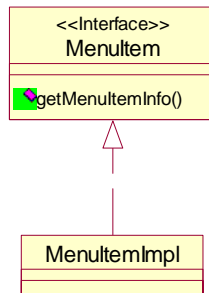


图 2-5 MenuItem 接口及实现

ActionHandler 的实现，如图 2-6 所示。

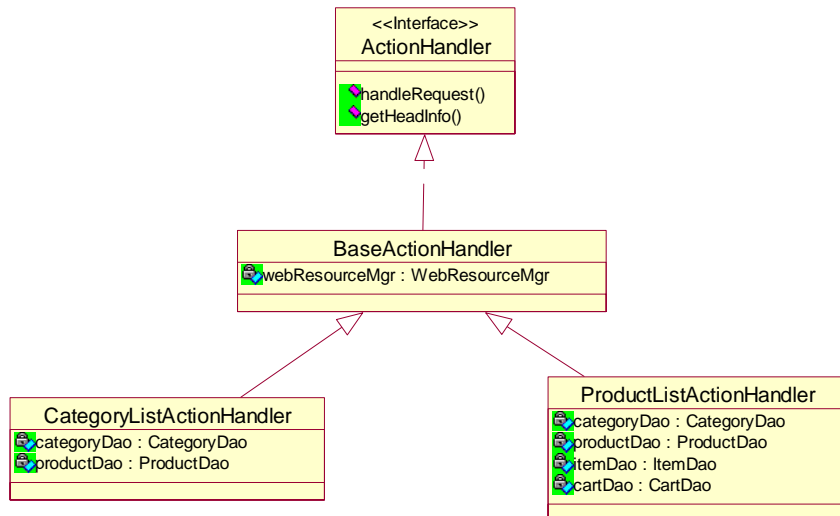


图 2-6 ActionHandler 接口及实现

`BaseActionHandler` 主要实现了对 Web 静态资源的配置管理，而从 `BaseActionHandler` 派生出的 `CategoryListActionHandler` 和 `ProductListActionHandler` 是实际处理 URI 的 `ActionHandler`。

6. ProductManagement模块

这个模块提供了产品维护的功能，目前主要实现的是库存的修改功能。和之前提到的 `ProductList` 模块类似，`ProductManagement` 模块也要响应 `Dal` 模块提供的 `Dao` 接口的实现。并且要提供 `MenuItem` 和 `ActionHandler` 的实现。整体思路同 `ProductList` 模块。

7. ShoppingCart模块

这同样也是一个展现用的模块。功能是展现用户当前购物车的状况，以及对购物车中的产品条目

进行删除和修改数量的操作。从设计上，ShoppingCart 和之前的 ProductList、ProductManagement 模块是一样的。

完成了模块的设计，下面就让我们来实现这些模块，完成 Petstore 的编码。

2.2 新一代Petstore的实现

在前面一节，我们完成了 Petstore 功能模块的设计，这在一节中，我们就动手来实现这个即插即用、即卸即停的 Petstore。

2.2.1 环境准备

首先，我们要准备开发环境。在前面一章，我们已经详细介绍了在 Eclipse 环境下如何准备开发环境，也介绍了如何基于 Spring-DM 进行开发，这里就不再赘述。下面就列出 Petstore 需要用到的 Bundle。

```
javax.servlet
org.apache.commons.logging
org.eclipse.equinox.http.jetty
org.eclipse.equinox.http.servlet
org.eclipse.osgi
org.eclipse.osgi.services
org.mortbay.jetty
org.springframework.bundle.spring
org.springframework.osgi.core
org.springframework.osgi.extender
org.springframework.osgi.io
```

其中 org.springframework.osgi.core, org.springframework.osgi.extender 和 org.springframework.osgi.io 这三个 bundle 是在 Spring-dm 的下载压缩文件中的，org.springframework.bundle.spring 是 SpringIDE 插件中的一个 bundle，我们这里使用这个 Bundle 是为了方便，这个 Bundle 包含了很多的内容，比如 spring-aop, spring-context, spring-beans, spring-dao 等内容。

我们为每个模块创建一个插件工程，然后把这些工程放在一个 Workspace 中，并且把 Spring-DM 中我们要用到的包导入到 Eclipse 中，具体显示见图 2-7。

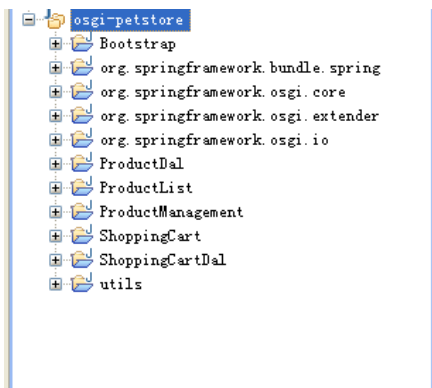


图 2-7 工程显示

然后在 Run Configurations 中进行配置，加入要用到的 Bundle（见图 2-8）。

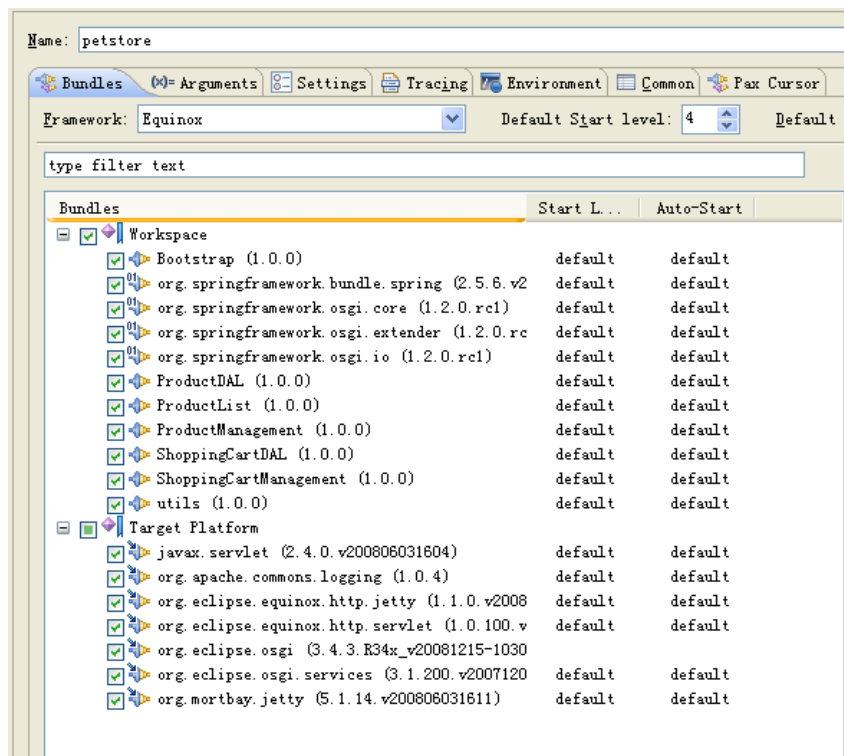


图 2-8 运行配置的 Bundles

之后，我们就可以开始编码了。

2.2.2 Utils模块

首先我们从 Utils 模块开始，这是一个只依赖于外部 Bundle 的模块。

- 在 Import Packages 中增加下面的 Package:

```
org.osgi.framework
org.osgi.service.http
org.springframework.osgi.context
```

- 新建 WebConfigMgr 接口，代码如下:

```
public interface WebConfigMgr {
    String getResourcePath();
}
```

- 增加对 WebConfigMgr 的实现——WebConfigMgrImpl，代码如下:

```
public class WebConfigMgrImpl implements WebConfigMgr {
    private String resourcePath;

    public String getResourcePath() {
        return resourcePath;
    }

    public void setResourcePath(String resourcePath) {
        this.resourcePath = resourcePath;
    }
}
```

- 增加 WebResourceMgr 类。这个类的功能是方便我们的模块注册自己的静态 Web 资源。这个类

实现了 `BundleContextAware` 接口，在启动的时候能够获取 `BundleContext`；另外，这个类也实现了 `ServiceListener` 接口，并且在这个类 `start` 方法被调用的时候，通过 `BundleContext` 注册了对 `HttpService` 的 `Listener`，在 `HttpService` 发生变化时候去进行相应的注册和注销的工作。

我们看下面这个类中比较重要的代码：

```
private HttpService getHttpService(){
    if (ref == null){
        ref = bundleContext.getServiceReference(HttpService.
            class.getName());
    }

    if (ref != null){
        return (HttpService) bundleContext.getService(ref);
    }
}
```

上面是获取 `HttpService` 的代码，我们通过 `BundleContext` 获得 `HttpService` 的 `Reference`，然后再通过 `BundleContext` 获取到 `HttpService` 这个服务。

```
public void start() throws InvalidSyntaxException{
    registerWebResource();
    this.bundleContext.addServiceListener(this, "(objectClass=" +
        HttpService.class.getName() +
        ")");
}

return null;
}
```

上面的代码是 `WebResourceMgr` 这个类的 `start` 方法，`WebResourceMgr` 是会被配置为一个 `Spring` 的 `Bean`，`start` 方法是在 `WebResourceMgr` 这个 `Bean` 被初始化的时候调用的。可以看到，我们进行了 `Web` 资源注册，并且添加了 `ServiceListener`。

```
private void registerWebResource(){
    try {
        HttpService httpService = getHttpService();
        if(null != httpService){
            httpService.registerResources(getResourceAlias(), getName(), null);
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

这段是注册 `Web` 资源的代码，其中 `getResourceAlias()` 和 `getName()` 返回的值都是用户在 `Spring` 中为 `WebResourceMgr` 配置的值，我们在不同的模块中会配置独立的 `WebResourceMgr` 的实例。最后，我们再简单看一下注销 `Web` 资源的代码：

```
private void unregisterWebResource(){
    try {
        HttpService httpService = getHttpService();
        if(null != httpService){
            httpService.unregister(getResourceAlias());
        }
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
```

```
}
}
```

从以上代码可以看出：

- 在 Exported Packages 中增加 org.osgichina.petstore.util 包。
- 增加 Spring-DM 配置文件。

我们在 Petstore 中，在每个模块的 META-INF 下，都有一个 spring 的目录，然后在 spring 目录下都有两个 xml，类似如图 2-9 所示的情况。

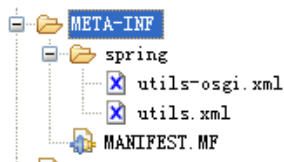


图 2-9 Spring-DM 配置目录

utils.xml 是一个普通的 Spring 的配置文件。utils-osgi.xml 是一个 Spring-DM 扩展的和 OSGi 服务相关的配置，我们下面简单地看下这两个配置文件的内容。

utils.xml

```
<bean name="_webConfigMgr" class="org.osgichina.petstore.#
    util.WebConfigMgrImpl">
    <property name="resourcePath" value="/petstore/resource" />
</bean>
```

可以看到，这个和普通的 spring 的配置文件是一样的。

Utils-osgi.xml

```
<osgi:service id="webConfigMgr" ref="_webConfigMgr"
    interface="org.osgichina.petstore.util.WebConfigMgr">
</osgi:service>
```

这里，我们把在 utils.xml 中定义的 _webConfigMgr bean 发布为了一个 OSGi 的 service，供其他的模块来使用。

到此，我们已经完成了 Utils 模块。下面来看一下 Bootstrap 模块。

2.2.3 Bootstrap 模块

1. Import Packages

代码如下：

```
javax.servlet
javax.servlet.http
org.apache.commons.logging
org.osgi.framework
org.osgi.service.http
org.osgichina.petstore.util
org.springframework.beans.factory
```

```
org.springframework.core.io
org.springframework.osgi.context
```

2. Required Plug-ins

代码如下：

```
org.springframework.bundle.spring
```

3. 增加接口定义

代码如下：

```
org.osgichina.petstore.bootstrap.actionhandler.ActionHandler
org.osgichina.petstore.bootstrap.actionhandler.ActionHandlerMap
org.osgichina.petstore.bootstrap.menu.MenuItem
org.osgichina.petstore.bootstrap.pagetemplate.DefaultPage
org.osgichina.petstore.bootstrap.pagetemplate.PageFooter
org.osgichina.petstore.bootstrap.pagetemplate.PageHeader
```

代码请参见源码中 Petstore/Bootstrap 目录中的代码。这里就不再列出。

4. 辅助的Bean的实现

在 Bootstrap 中，我们实现了两个辅助的 Bean：

- MenuItemInfo，用户存储菜单项相关的信息。
- DefaultActionHandlerMap。实现了 ActionHandlerMap，用于实现 URI 到 ActionHandler 的映射关系的管理。

5. PageHeader的实现-PageHeaderImpl

在 PageHeaderImpl 中，我们管理了其他模块的菜单和展示。从代码上来说主要分为两个部分：一个是菜单对象的管理，一个是展示。那么我们先来看一下菜单对象的管理。

我们在 PageHeaderImpl 中有两个方法：onBind 和 onUnbind，分别是一个菜单对象绑定和接触绑定会被调用的。代码如下：

```
public synchronized void onBind(MenuItem menuItem, Map<?,?>
serviceProps){
    List<MenuItem> list = new LinkedList<MenuItem>();
    if(menuItems != null){
        list.addAll(menuItems);
    }
    list.add(menuItem);
    menuInfoMap.put(menuItem, menuItem.getMenuItemInfo());
    Collections.sort(list, new Comparator<MenuItem>(){
        public int compare(MenuItem o1, MenuItem o2) {
            return menuInfoMap.get(o1).getPosition() - menuInfoMap.
                get(o2).getPosition();
        }
    });
    menuItems = list;
}
```

可以看到这个方法做了四件事情：

- 复制目前的菜单项。
- 加入新的菜单项。
- 对菜单项列表排序。

- 更新菜单项列表对象。

而 onUnbind 做的就是相反的事情了。代码如下：

```
public synchronized void onUnbind(MenuItem menuItem, Map<?,?> serviceProps){
    List<MenuItem> list = new LinkedList<MenuItem>();
    if(menuItems != null){
        list.addAll(menuItems);
    }

    menuInfoMap.remove(menuItem);
    list.remove(menuItem);
    menuItems = list;
}
```

在 onUnbind 中做了三件事情：

- 复制目前的菜单项。
- 删除接触绑定的菜单项。
- 更新菜单项列表对象。

可以看到在 onUnbind 中，相比 onBind，少了排序的动作。那么，onBind 和 onUnbind 为什么会被调用到呢？这是因为我们 spring-dm 配置文件中的一个配置，我们看一下是如何配置的。

```
<osgi:list id="menuItem" interface="org.osgichina.petstore.bootstrap.
    menu.MenuItem" cardinality="0..N" >
    <osgi:listener bind-method="onBind"
        unbind-method="onUnbind"
        ref="pageHeader" />
</osgi:list>
```

可以看到，我们在引用 menuItem 的时候，定义了一个 listener，指定了这个 listener 的 bean 以及在 bind 和 unbind 的时候要调用的方法。这样，我们的 PageHeaderImpl 的 onBind 和 onUnbind 方法就可以在 menuItem 的服务启动或者停止的时候被调用了。

在解决了 MenuItem 服务的引用问题后，就是展示的问题了。我们是在 PageHeaderImpl 的 getPageHeader 方法中生成了展示的 HTML。这里就不列出详细的代码了。

1. PageFooter的实现-PageFooterImpl

相对于 PageHeaderImpl，PageFooterImpl 的实现就非常简单了。在 PageFooterImpl 这个实现中，我们只是在 getPageFooter 中返回了一个版权的信息。

```
public String getPageFooter(String servletPath, String resourcePath) {
    return "OSGi原理与最佳实践";
}
```

2. ControllerServlet实现

ControllerServlet 是我们 Petstore 中的控制 Servlet。在 ControllerServlet 中，负责响应用户的请求，并根据请求的 URI 把请求的处理派发给相应的 ActionHandler，负责渲染请求的响应结果。在 ControllerServlet 中，管理了 PageHeader、PageFooter，以及各个模块提供的 DefaultPage 的实现，最重要的是一个 ActionHandlerMap，用于进行请求派发的处理。我们简单看下 ControllerServlet 中对于 HTTP 请求的处理部分。

```
String pathInfo = request.getPathInfo();
if(pathInfo != null && pathInfo.length() > 0){
    String tempString = null;
```

```

while(true){
    tempString = pathInfo.replaceAll("//", "/");
    if(tempString.equals(pathInfo)){
        break;
    }
}
pathInfo = pathInfo.substring(1);
int index = pathInfo.indexOf("?");
if(index > 0){
    pathInfo = pathInfo.substring(0, index - 1);
}
}

```

上面这段代码对 HTTP 请求的 `pathInfo` 进行分析，得到去掉请求的参数后的路径信息。

```

if(null == pathInfo || pathInfo.length() == 0){
    if(this.defaultPages.size() > 0){
        pathInfo = this.defaultPages.get(0).getUri();
    }
}

```

而这部分的代码是在没有路径信息的情况下使用默认页面。类似 `http://localhost/petstore/app` 这样的请求，得到的 `pathInfo` 是 `null`。

```

if(pathInfo != null){
    actionHandler = this.actionHandlerMap.get(pathInfo.toLowerCase());
}

```

如果获取了路径信息，则从 `actionHandlerMap` 中查询对应的 `ActionHandler`。

根据路径信息得到了对应的 `ActionHandler` 后，我们就开始渲染页面了，加入在返回 HTML 中的 `<head></head>` 信息，并且按照先后顺序加入 Header，页面内容和 Footer 信息。然后返回给用户。

3. 数据初始化实现

为了简化例子，我们使用了 `HSQLDB` 作为数据库，在 `Bootstrap` 启动的时候，自动加载数据。主要通过两个类来实现。

■ HsqldbServerBean

这个类的功能是以 `Server` 方式启动 `HSQLDB`。主要的启动代码在 `afterPropertiesSet()` 这个方法中。

```

public void afterPropertiesSet() throws Exception {
    if ((params == null) || params.isEmpty()) {
        throw new IllegalArgumentException("missing hsqldb params");
    }
    server = new Server();
    HsqlProperties props = new HsqlProperties(params);
    server.setProperties(props);
    server.setLogWriter(null);
    server.setErrWriter(null);
    server.start();
}

```

■ DataLoader

这个类用于创建表并加载数据。主要的工作是从配置文件中读取 SQL，并执行 SQL（包括建表的 SQL 和初始化数据的 SQL）。具体的代码可以参考源码中的 `PetStore\Bootstrap` 下的 `DataLoader.java`。

4. Exported Packages

```
org.osgi.china.petstore.bootstrap.actionhandler
org.osgi.china.petstore.bootstrap.menu
org.osgi.china.petstore.bootstrap.pagetemplate
```

5. ClassPath

在 Bootstrap 中，我们要用到 commons-dbcp、commons-pool 和 hsqldb 这几个 jar 包。我们的方案是在 Bootstrap 中增加一个 lib 目录，把这三个 jar 包放到 lib 目录下，并且在 Bundle 的 ClassPath 中加入这三个 jar 包。

这个时候，我们已经可以启动了，下面来看目前的成果。

启动我们的工程，然后在浏览器中输入 `http://localhost/petstore/app`，将会看到类似如图 2-10 的现实。



图 2-10 当前 PetStore 浏览器上的展示

恭喜你，我们的 Petstore 已经可以工作了。

2.2.4 ProductDal模块

在 Petstore 中，我们有两个数据访问的模块，一个就是当前要实现的 ProductDal 模块，另外一个就是 ShoppingCarDal 模块。数据访问模块也是基础的模块，不依赖内部的其他模块，而本身是提供服务给其他模块使用。下面就来看一下 ProductDal 模块的实现。

1. Imported Packages

代码如下：

```
org.springframework.core
org.springframework.dao
org.springframework.jdbc.core
```

2. 添加Dao接口

代码如下：

```
org.osgi.china.petstore.productdal.dao.CategoryDao 处理和类目相关的数据
org.osgi.china.petstore.productdal.dao.ProductDao 处理和产品相关的数据
org.osgi.china.petstore.productdal.dao.ItemDao 处理和条目相关的数据
```

3. 添加数据对象（DataObject）

代码如下：


```
org.osgichina.petstore.productdal.dataobject.Category
org.osgichina.petstore.productdal.dataobject.Product
org.osgichina.petstore.productdal.dataobject.Item
```

4. 实现Dao接口

在 ProductDal 例子中，我们对于 Dao 的实现是通过 JDBC 来操作数据完成的。代码相应的在

```
org.osgichina.petstore.productdal.dao.impl.CategoryDaoImpl
org.osgichina.petstore.productdal.dao.impl.ProductDaoImpl
org.osgichina.petstore.productdal.dao.impl.ItemDaoImpl
```

这三个类中。

5. Exported Packages

代码如下：

```
org.osgichina.petstore.productdal.dao
org.osgichina.petstore.productdal.dataobject
```

6. Spring-dm配置

Bean 的定义——productdal.xml

```
<bean name="_productDao" class="org.osgichina.petstore.productdal.dao.
    impl.ProductDaoImpl"/>

<bean name="_categoryDao" class="org.osgichina.petstore.productdal.dao.
    impl.CategoryDaoImpl"/>

<bean name="_itemDao" class="org.osgichina.petstore.productdal.dao.
    impl.ItemDaoImpl" />

<bean id="jdbcTemplate" class="org.springframework.jdbc.
    core.JdbcTemplate"/>
```

引用的服务和暴露的服务的定义——productdal-osgi.xml

```
<osgi:reference id="dataSource" interface="javax.sql.DataSource" />

<osgi:service id="productDao" ref="_productDao"
    interface="org.osgichina.petstore.productdal.dao.ProductDao">
</osgi:service>

<osgi:service id="categoryDao" ref="_categoryDao"
    interface="org.osgichina.petstore.productdal.dao.CategoryDao">
</osgi:service>

<osgi:service id="itemDao" ref="_itemDao"
    interface="org.osgichina.petstore.productdal.dao.ItemDao">
</osgi:service>
```

好，到这里，我们完成了 ProductDal 模块，下面我们再接再厉，完成另外一个数据访问模块——ShoppingCartDal 模块。

2.2.5 ShoppingCartDal模块

ShoppingCartDal 模块和 ProductDal 模块比较类似，都是提供数据访问服务，不过 ShoppingCartDal 的实现更加简单，数据是非持久的，也就是数据只是放到内存当中。在实现的过程中，和 ProductDal 是基本一样的，所以，我们在这里也不再详细列出 ShoppingCartDal 模块的具体实现步骤了，大家可

以参考源码中的 Petstore\ShoppingCartDal 下的代码。

2.2.6 ProductList模块

介绍了基础的模块后，我们接下来看到的三个模块都是展现层的模块。我们先来看下 ProductList 模块。

1. Imported Packages

代码如下：

```
org.osgi.service.http
org.osgichina.petstore.bootstrap.actionhandler
org.osgichina.petstore.bootstrap.menu
org.osgichina.petstore.bootstrap.pagetemplate
org.osgichina.petstore.productdal.dao
org.osgichina.petstore.productdal.dataobject
org.osgichina.petstore.shoppingcartdal.dao
org.osgichina.petstore.util
```

2. 添加Required Plug-ins

代码如下：

```
javax.servlet
```

3. MenuItemImpl

这个类实现了 MenuItem 接口，提供了 ProductList 模块要注册的一个菜单项。

4. ProductListDefaultPage

这个类实现了 DefaultPage 接口，设置了 ProductList 模块提供的默认的 URI。

5. ActionHandler相关实现

■ BaseActionHandler

BaseActionHandler 实现了 ActionHandler 接口，是 ProductList 模块中其他 ActionHandler 实现类的父类。主要功能是管理 WebResourceMgr 对象，实现了统一的 getHeadInfo 方法，提供了生成图片 URL 的方法，方便了 ActionHandler 中处理图片 URL 的代码的编写。

■ CategoryListActionHandler

这是针对类目显示页面的一个 ActionHandler，获取类目列表，循环显示类目列表中的产品信息。

■ ProductListActionHandler

这是针对某一个产品显示详细信息页面的 ActionHandler，能获取这个产品的信息，以及属于这个产品的条目，并进行显示。

6. Spring-DM配置

Bean 的定义——productlist.xml

```
<bean name="_menuItem" class="org.osgichina.petstore.productlist.
    menuItem.MenuItemImpl"/>

<bean name="_defaultPage" class="org.osgichina.petstore.productlist.
    pagetemplate.ProductListDefaultPage" />
```

```
<bean name="categoryListActionHandler" class="org.osgichina.petstore.
    productlist.actionhandler.CategoryListActionHandler" />

<bean name="productListActionHandler" class="org.osgichina.petstore.
    productlist.actionhandler.ProductListActionHandler" />

<bean name="_actionHandlerMap" class="org.osgichina.petstore.bootstrap.
    actionhandler.DefaultActionHandlerMap">
    <property name="actionHandlerMap">
        <map>
            <entry key="productlist" >
                <ref bean="categoryListActionHandler"/>
            </entry>
            <entry key="productlist/product_list" >
                <ref bean="productListActionHandler"/>
            </entry>
        </map>
    </property>
</bean>

<bean name="webResourceMgr" class="org.osgichina.petstore.util.
    WebResourceMgr"
    init-method="start">
    <property name="relativeResourcePath" value="productlist"/>
    <property name="name" value="productlist"/>
</bean>
```

引用的服务和暴露的服务的定义——productlist-osgi.xml

```
<osgi:service id="menuItem" ref="_menuItem"
    interface="org.osgichina.petstore.bootstrap.menu.MenuItem" />

<osgi:service id="actionHandlerMap" ref="_actionHandlerMap"
    interface="org.osgichina.petstore.bootstrap.actionhandler.
        ActionHandlerMap" />

<osgi:service id="defaultPage" ref="_defaultPage"
    interface="org.osgichina.petstore.bootstrap.pagetemplate.
        DefaultPage" />

<osgi:reference id="productDao" interface="org.osgichina.petstore.
    productdal.dao.ProductDao" />

<osgi:reference id="categoryDao" interface="org.osgichina.petstore.
    productdal.dao.CategoryDao" />

<osgi:reference id="itemDao" interface="org.osgichina.petstore.
    productdal.dao.ItemDao" />

<osgi:reference id="cartDao" interface="org.osgichina.petstore.
    shoppingcartdal.dao.CartDao" />

<osgi:reference id="webConfigMgr" interface="org.osgichina.
    petstore.util.WebConfigMgr" />
```

好，到这里，我们完成了 ProductList 模块，我们再次运行一下我们的 Petstore，然后在浏览器中输入 <http://localhost/petstore/app>，则会看到类似图 2-11 的页面显示。

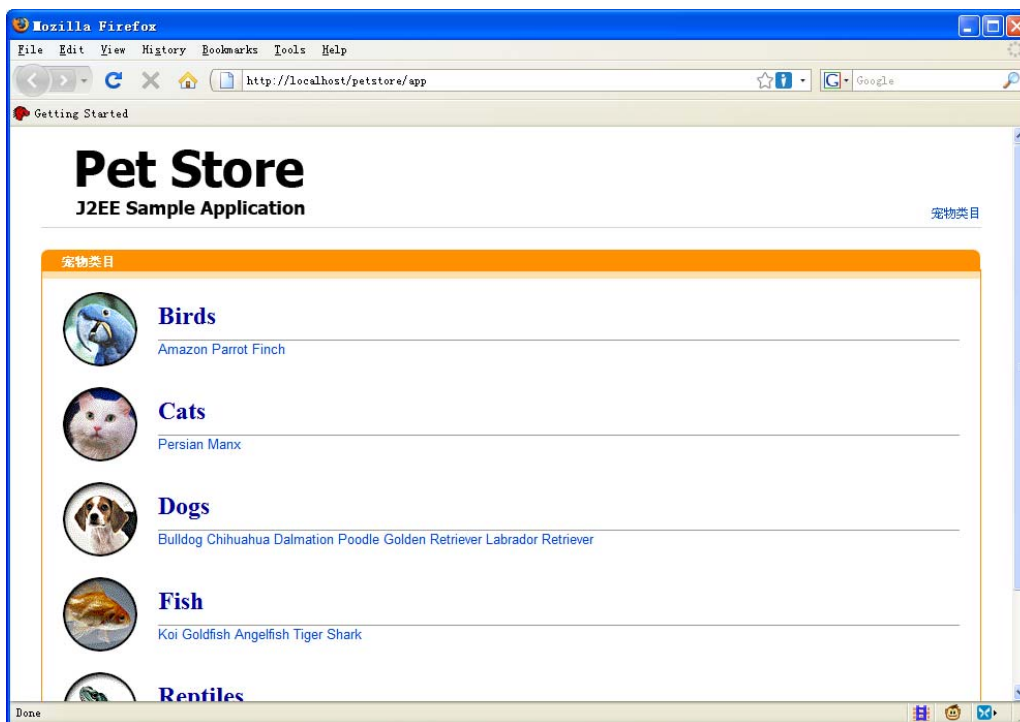


图 2-11 产品浏览的展现

我们可以点击某个分类下的一个产品，比如“Amazon Parrot”，然后会看到如图 2-12 的显示。

可以看到，有一个“放到购物车”的按钮，我们会在接下来实现购物车的功能。

这个时候，我们在 OSGi 的控制台上停止 ProductList 模块，然后刷新一下页面，可以看到类似 2-13 的页面。

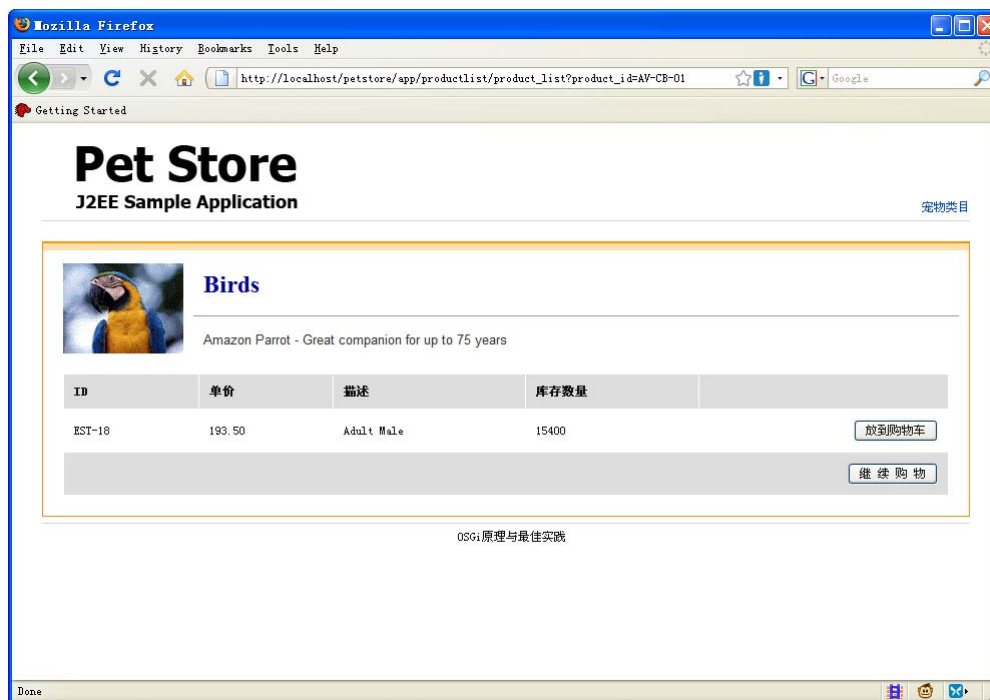


图 2-12 展品详细信息的展现



图 2-13 卸载模块后的展现

大家会发现，这个页面和开始没有 `ProductList` 模块时的显示是一样的，而我们这个时候再次启动 `ProductList` 模块，会发现又回到了刚才产品信息的页面了。

好，下面我们继续来完成剩下的两个展现层的模块。

2.2.7 ShoppingCart模块

ShoppingCart 模块主要展示了目前用户购物车中的条目列表信息，并且可以删除某个条目或者修改某个条目的数量。整体上的实现步骤和 ProductList 是类似的，这里就不再列出具体的步骤了。

2.2.8 ProductManagement模块

ProductManagement 模块是进行产品维护的模块。在例子程序中，我们在 ProductManagement 模块中，完成了对现有类目的显示，以及对类目下每个产品具体信息的显示。读者可以扩充这个模块，增加添加、删除类目和产品的功能，以及修改条目数量的功能。实现这个模块的步骤这里就不再详细列出了。

2.3 部署

到这里，我们已经在 Eclipse 的开发环境里完成了 Petstore 的例子，我们对于 Petstore 的实现是在 OSGi 的容器中嵌入了 Web 服务器。在前面一章，我们提到了如何对 OSGi 应用进行打包部署。因此我们参照之前章节的内容，即可完成 Petstore 例子的打包、部署，脱离 Eclipse 环境来启动 Petstore 应用。

2.4 Petstore的扩展

在这一章中，我们实现了一个很简单的 Petstore。这个例子主要向读者展现的是如何用 Spring-DM 来构建一个 Petstore，以及如何做到即插即用。在目前这个 Petstore 的基础上，有几个方面是可以再扩展完善的，比如在展现的时候，我们没有用任何的框架，读者可以使用自己熟悉的 Web 框架或一些模板引擎来进行扩展，而 Dal 的部分，我们使用的是 JDBC 直接访问数据库的方式，读者同样可以根据自己的情况采用一些框架来完成这个工作，比如 Hibernate 或 iBatis，另外，我们 Dal 的接口和实现是定义在一个工程中，这里这么做是因为是演示程序，为了少创建工程而放在一起的，在实际产品中，我们是会把接口的定义独立出来放到一个工程中的。

线下定期举行的技术讨论俱乐部

Qclub

分享

交流

我们影响有影响力的人



主题：围绕InfoQ关注的领域设定话题

形式：1个主题、1+个嘉宾、N个参会人员

参会者：有决策权的中高端技术人员

人数：限制人数，保证每个人的发言权利

频率：每月一次