OSGi培训文档素材

# OSGi 清单选项

OSGi R4 框架核心目前的规范草案几乎有 PDF 格式的 300 页。介绍该规范的每个部分超出了本文范围，但我将讨论 Eclipse 插件开发人员特别感兴趣的 OSGi manifest.mf 选项：

**Bundle-Activator**

该类用于启动和停止绑定包。在上面的示例插件中，指定了 org.eclipse.pde.internal.ui.PDEPlugin 类。该类扩展org.eclipse.core.runtime.Plugin，实现了 BundleActivator 接口。

**Bundle-ClassPath**

该属性指定要用于绑定包的 CLASSPATH。该属性可以包含对绑定包 jar 文件中目录或 jar 文件的引用。可以使用句点指明绑定包的根。在示例 Eclipse PDE 绑定包中，指定了绑定包 jar 文件中的 org.eclipse.pde.ui\_3.1.0.jar。如果将插件的源版本导入工作区中，导入过程将更改绑定包 CLASSPATH 以显示为 Bundle-ClassPath:，这允许插件的开发版本挑选已编译的绑定包类。

**Bundle-Version**

该属性指定绑定包的版本号。包导入和必需的绑定包规范可以包括绑定包版本号。

**Export-Package**

该属性指定要公共暴露给其他插件的所有包。

**Import-Package**

该属性指定要从必需插件中显式导入的所有包。默认情况下，必须为要启动的绑定包解析所有包。还可以将包导入指定为可选项，以支持包不存在的情况。显式导入的类在 Require-Bundle 插件中的包之前解析。

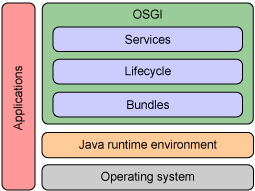
**Require-Bundle**

该属性指定要在给定绑定包中导入使用的绑定包及其已导出的包。指定的绑定包在显式包导入之后解析。

# 类加载器

OSGi 规范定义了绑定包生命周期的基础架构和绑定包的交互方式。这些规则通过使用特殊 Java 类加载器来强制执行。在一般 Java 应用程序中，CLASSPATH 中的所有类都对所有其他类可见。相反，OSGi 类加载器基于 OSGi 规范和每个绑定包的 manifest.mf 文件中指定的选项（稍后将详细介绍）来限制类交互。

# OSGi架构图



# 什么是 OSGi

OSGi 联盟成立于 1999 年 3 月，致力于制定管理本地网络设备服务的规范。OSGi 联盟是为家用设备、汽车、手机、桌面、小型办公环境以及其他环境制定下一代网络服务标准的领导者。OSGi 全名原为 Open Services Gateway initiative，但现在这个全名已经废弃。

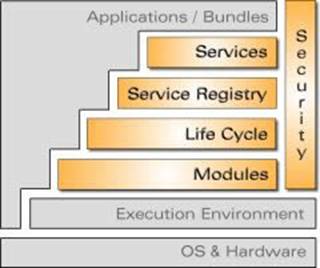
OSGi 联盟基于它成立之初的使命，推出了 OSGi 服务平台规范，用于提供开放和通用的架构，使得服务提供商、开发人员、软件提供商、网关操作者和设备提供商以统一的方式开发、部署和管理服务。

目前最广为认可和应用的是 OSGi 规范 4（R4，Release 4），其最新版本是 4.2，共由核心规范、标准服务（Standard Services）、框架服务（Framework Services）、系统服务（System Services）、协议服务（Protocol Services）、混合服务（Miscellaneous Services）等几部分共同组成。核心规范是 OSGi 规范中的核心部分，它通过一个分层的框架，实现了 OSGi 最为成功的动态插件机制，它主要提供了：1. OSGi Bundle 的运行环境；2. OSGi Bundle 间的依赖管理；3. OSGi Bundle 的生命周期管理；4. OSGi 服务的动态交互模型。

虽然 OSGi 在嵌入式应用方面有成功的案例，比如宝马汽车的嵌入式应用，但真正让 OSGi 得到认可和关注的则是 Eclipse 项目。Eclipse 是目前著名的一个 IDE，其最大的两个特点是插件和扩展，而其实插件机制正是通过实现 OSGi 规范实现的。下面我们了解下 Eclipse 的核心 Equinox 项目。

# OSGi 体系结构

OSGi 的体系架构是基于插件式的软件结构，包括一个 OSGi 框架和一系列插件，在 OSGi中，插件称为 Bundle，其中，OSGi 框架规范是 OSGi 规范的核心部分，它提供了一个通用的、安全可管理的 Java 框架，通过这个框架，可以支持 Bundle 服务应用的部署和扩展。Bundle 之间可以通过 Import Package 和 Require-Bundle 来共享 Java 类，在 OSGi 服务平台中，用户通过开发 Bundle 来提供需要的功能，这些 Bundle 可以动态加载和卸载，或者根据需要远程下载和升级。OSGi 体系结构图如图 1 所示：

**图示1 OSGi 体系结构**  
 

其中：

Execution Environment：

Bundle 应用所倚赖运行的 Java 执行环境，如 J2SE-1.4、CDC-1.0 等都是可用的执行环境。

Modules：

模块层定义了 Bundle 应用的加载策略。OSGi 框架是一个健壮并且严格定义的类加载模型。在大多数 Java 应用中，通常只有一个单独的 ClassPath，它包含了所有的 Java 类文件和资源文件，OSGi基于Java技术，对于每个实现了 BundleActivator 接口的 Bundle 应用，为它生成一个单独的 ClassLoader，使得 Bundle 应用的组织更加模块化。

Life Cycle：

生命周期层可以动态地对 Bundle 进行安装、启动、停止、升级和卸载等操作。该层基于模块层，提供了一组 API 来控制 Bundle 应用的运行时操作。

Service Registry 和 Services：

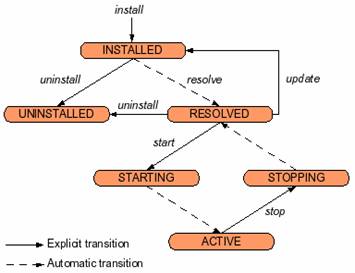
OSGi 服务层定义了一个集成在生命周期层中的动态协作模型，是一个发布、动态寻找、绑定的服务模型。一个服务通常是一个 Java 对象实现了特定的服务接口，并且通过服务注册，被绑定到 OSGi 的运行环境中。Bundle 应用可以注册发布服务，动态绑定服务，并且在服务注册状态改变时，可以接受到事件消息等。

Security：

OSGi 的安全管理是基于 Java2 安全体系的，贯穿在 OSGi 平台的所有层中，它能够对部署在 OSGi 运行环境中的 Bundle 应用进行详细的管理控制。

# Bundle 生命周期的状态

在一个动态扩展的 OSGi 环境中，OSGi 框架管理 Bundle 的安装和更新，同时也管理 Bundle 和服务之间的依赖关系。一个 Bundle 可能处于以下六个状态，如图 2 所示：

**图示 2 Bundle 状态图**  
 

INSTALLED：安装完成，本地资源成功加载。

RESOLVED：依赖关系满足，这个状态意味该Bundle要么已经准备好运行，要么是被停止了。

STARTING：Bundle正在被启动，BundleActivator的start()方法已经被调用但是还没有返回。

STOPPING：Bundle正在被停止，BundleActivator的stop()方法已经被调用但是还没有返回。

ACTIVE：Bundle 被成功启动并且在运行。

UNINSTALLED：bundle被卸载并且无法进入其他状态。

Bundle接口定义了getState()方法来返回Bundle的状态。

# OSGi 标准服务

在 OSGi 平台之上，OSGi 联盟定义了很多服务。服务是由一个 Java Interface 来定义的，Bundle 可以实现这个接口并且把服务注册到服务注册表中去，用户可以从注册表中找到需要的服务来使用，并且可以响应特定服务的状态改变，如服务注册和服务取消。下面简单介绍一下 OSGi Release 4 的一些主要服务。OSGi 框架提供了权限管理服务，包管理服务和最初加载系统服务。这些服务是 OSGi 框架的一部分并且管理着 OSGi 框架的运作。

Permission Admin Service：权限管理是指 Bundle 是否许可其他的 Bundle 的代码。当前的或者其他的 Bundle 的权限可以通过这个服务来操作，一旦被设定权限，马上就生效。 Package Admin Service：Bundle 之间可以共享包内的 Java 类和资源，bundle 的更新可能需要 OSGi 框架重新解析 Bundle 之间的依赖关系，这个服务提供了 OSGi 服务平台中包的共享状态信息。

Start Level Service：Start Level是指一些在特定Bundle起动之前必须运行或者初始化的一系列 bundle。Start Lever Service 可以设置当前OSGi服务框架初始的Start Level，并且可以指定和查询特定Bundle的Start Level。

# Java 类装入器原理

类装入器是 JVM 用来装入类的类，它对于 Java 编程是非常重要的一个概念。一般情况下，程序员在编写程序的时候都可以忽略类装入器的存在性。但是对于服务器端编程或者是一些特殊情况下时候，深入了解类装入器的机制以及其在不同情况下的实现还是非常必要的。

首先，当一个 JVM 启动的时候，Java 缺省开始使用三个类装入器。它们分别是：

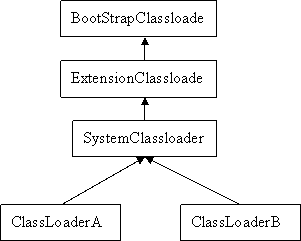
* 引导（Bootstrap）类装入器；
* 扩展（Extension）类装入器；
* 系统（System）类装入器；

它们分别实现如下的功能：

* 引导类装入器是用本地代码实现的类装入器。它负责将 <Java\_Runtime\_Home>/lib 下面的类库加载到内存中。
* 扩展类装入器是由 Sun 的 ExtClassLoader 实现的。它负责将 < Java\_Runtime\_Home >/lib/ext 或者由系统变量 java.ext.dir 指定位置中的类库加载到内存中。
* 系统类装入器又叫应用程序类装入器，是由 Sun 的 AppClassLoader 实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。

当应用程序需要加载某个类到内存中的时候，类装入器是如何工作的呢？这就设计到类装入器的一个重要方面：代理机制。每一个类装入器，除了引导类装入器以外，都有一个父类装入器。对于系统缺省定义的三个类装入器，引导类装入器是扩展类装入器的父类装入器，而扩展类装入器是系统类装入器的父类装入器。当然，应用程序也可以使用自己的类装入器来使用特定的方法来装载类，因此，整个系统中的类装入器就形成一个树状结构。

当使用某个类装入器来试图装载某个类的时候，该类装入器会首先使用其父类装入器来试图装载该类。对于每一个装载进来的类，JVM 都会给其分配一个唯一的 ID。因此，不同类装入器可以装载同一个类到 JVM 中。例如，对于如下图结构的 ClassLoaderA 和ClassLoaderB：

**图 1 类装入器的结构**  
 

假设类 C 在系统类装入器指定的类路径中，则无论是使用 ClassLoaderA 还是使用 ClassLoaderB，都只会得到同样一个类 C。

但是如果类 C 分别在 ClassLoaderA 以及 ClassLoaderB 指定的类库中，则使用 ClassLoaderA 得到到类 C 实例会不同于 ClassLoaderB得到的类 C 实例。尽管两个类装入器在同一个 JVM 中。

上面的类装入器的向上代理结构看上去很完美了，但是，当系统变得复杂的时候，就还是显得不够用了。

例如，当 Java 引入了 JNDI 以后，JNDI 核心部分是通过引导 类装入器在 JVM 启动的时候装载进入 JVM 的。而 JDNI 核心部分是通过配置信息来在运行时候装载定义在用户的类路径中的特定类来完成特定需要。而这是上面定义的类装入器的向上代理模式所不能支持的。

为了解决这个问题，Java 2 中引入了线程上下文（Thread Content）类装入器的概念，每一个线程有一个 Context 类装入器。这个 Context 类装入器是通过方法 Thread.setContextClassLoader() 设置的，如果当前线程在创建后没有调用这个方法设置 Context 类装入器，则当前线程从他的父线程继承 Context 类装入器。如果整个应用都没有设置 Context 类装入器，则系统类装入器被设置为所有线程的 Context 类装入器。

对于我们上面所说 JNDI 的情况，引导 类装入器装载进入的 JNDI 核心类会使用 Context 类装入器来装载其所需要的 JNDI 实现类，而不是将该装载任务代理给其父类装入器来完成。这样，就解决了上面的问题。可以认为 Context 类装入器在传统的 Java 向上代理机制上打开了一个后门。Context 类装入器在 J2EE 中使用的很广泛，比如 Java 命名服务（JNDI），Java API for XML Parsing(JAXP)（注：在 Java1.4 中 JAXP 才作为 Java 的核心类的一部分，它才开始使用 Context 类装入器来加载不同的实现类）等。

简单而言，Java 中的类装入器就是上面几种，但是，在具体使用中，还是有很多变化，我们下面分别对于一些情况进行说明。

# Eclipse 类装入器原理

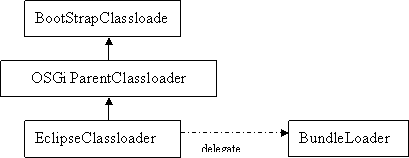
在典型的 Java 应用程序中，我们要加载一个类或资源时，通常可以使用三种类装入器：

* 当前类的类装入器；
* 当前线程的 Context 类装入器；
* 系统类装入器；

在实际的应用开发中比较常用到前两个类装入器是，下边重点介绍在 Eclipse 插件运行环境中前两种类装入器的结构和原理以及和典型的 Java 应用的不同之处。

**当前插件类的类装入器**

下图是 Eclipse 插件在运行时当前插件类的类装入器的体系结构图。

**图 2 Eclipse 插件类装入器的结构**  
 

其中 EclipseClassloader 类装入器实现了 OSGi 规范的 BundleClassLoader，它用来装载 Bundle （也就是插件）中的类 OSGi ParentClassloader 类装入器是 Eclipse 插件类装入器的父类装入器，可以通过在启动 Eclipse 时设置系统属性 osgi.parentClassloader 来改变它，这个改变类会影响所有的 Eclipse 插件。如果在启动时没有设置系统属性 osgi.parentClassloader, Eclipse 使用一个默认的空的类装入器。

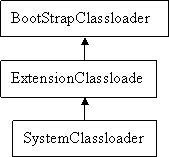
BundleLoader 是 EclipseClassloader 类装入器的代理，它是用来加载插件相关的资源的。

Eclipse 插件类装入器加载类或资源的过程如下：

1. 首先试图从父类装入器加载类，其过程是先从 OSGi ParentClassloader 类装入器加载类，OSGi ParentClassloader 类装入器使用传统的 Java 装入器的委托模式依次从父类装入器加载类。
2. 如果无法从 OSGi ParentClassloader 类装入器加载类，则试图通过代理 BundleLoader 来加载类。
3. BundleLoader 首先试图从此插件的需求依赖插件（"require"指定的插件）中去加载需要的类，如果找不到，则通过 EclipseClassloader 类装入器来从插件本地加载需要的类。
4. 如果还是找不到要加载的类，就会抛出类找不到异常。

**当前线程的 Context 类装入器**

在 Eclipse 中并没有设置 Context 类装入器，所以默认情况下当前线程的 Context 类装入器为系统的类装入器，其体系结构如 图 3 所示。

**图 3 Eclipse 插件当前线程 Context 类装入器的结构**  
 

在 Eclipse 中，每个插件都有自己的类装入器，每个线程有自己的类装入器。插件和线程之间没有统一的映射关系，所以 Eclipse 框架没有将线程的 Context 类装入器设置成有意义类装入器。