企业 OSGi 应用开发教程

企业 OSGi 应用开发教程

在 JavaOne 2011 上，Peter Kriens 关于 OSGi 做了两个介绍。Kriens 的演讲解释了为什么尽管 OSGi 表现的很难，用 OSGi 实现模块化对于今天的应用开发者来说是很有价值的。他也解释了如何进入这个领域，同时澄清了一些关于 OSGi 和模块化应用的错误概念。那么对于模块化应用开发的未来是怎么样的？企业中 OSGi 应用开发如何实现？在这本教程中我们将为您详细介绍。

OSGi 新主张

OSGi 联盟对于模块化应用开发的未来怎么看呢？他们认为应用开发总体而言包括一个基于组件的应用开发社区，足够支持这种蒸蒸日上的组件市场。就像消费者现在下载应用，使其手机能够执行最新最棒的效果，应用开发者也许某天能够下载模块化组件，将其插入到现有的企业应用中去。

- OSGi 成应用开发未来引路人
- OSGi 主张：封装是组件化安全途径

应用 OSGi 准备

OSGi 是 Java 领域里无可辩驳的最成熟的模块系统，它与 Java 几乎是如影相随，最早出现于 JSR 8，但是最新规范是 JSR 291。在决定采用 OSGi 框架开发项目之前，我们需要考虑哪些问题，是否为企业级 OSGi 做足了准备呢？如何用 OSGi 实现快乐共享？

- 采用 OSGi 框架开发项目的十个问题

- 为使用企业级 OSGi 做好准备了吗？
- OSGI：让共享变得快乐

OSGi 模块化

规范的模块化开发是需要 OSGi 的重要理由之一，模块化的开发方式一直就是现在的主流开发方式，但业界却一直缺乏这样的标准，当然，如果 Java 本身具备这样的标准自然就更好了。但由于大多数应用程序和系统的目的不是为模块化，或被设计并建造为土生土长的模块化设计，采用的 OSGi 通常包含某种程度的困难。

- OSGi 是所有模块化应用的框架
- 用 OSGi 架构实现模块化

OSGi 成应用开发未来引路人

OSGi 联盟对于模块化应用开发的未来怎么看呢？他们认为应用开发总体而言包括一个基于组件的应用开发社区，足够支持这种蒸蒸日上的组件市场。就像消费者现在下载应用，使其手机能够执行最新最棒的效果，应用开发者也许某天能够下载模块化组件，将其插入到现有的企业应用中去。

在 JavaOne 2011 上，Peter Kriens 关于 OSGi 做了两个有益的介绍。Kriens 的演讲解释了为什么尽管 OSGi 表现的很难，用 OSGi 实现模块化对于今天的应用开发者来说是很有价值的。他也解释了如何进入这个领域，同时澄清了一些关于 OSGi 和模块化应用的错误概念。

一种假设是局外人可能让开发服务网关（Open Services Gateway）计划中的“服务”代表面向服务架构（SOA）中的 Web 服务。结果正如 Kreins 指出的，OSGi 服务实际上是本地服务，谈不上什么冗长，所以相当快速。但是，很多的面向服务想法适用于 OSGi 模块化。

在这两种情况下，服务是系统间模块化（SOA 中称之为“松耦合”）的代理人。不同之处在于 Web 服务通常可以通过互联网进行分布式系统通信，而在 OSGi 本地服务连接子系统（模块）中交互形成复合应用。

这些本地服务在 OSGi 模块化模型中极其重要。这些服务正确实行时，就可以实现模块之间真正的松耦合。根据 Kriens 所说，确保每一个模块看不到所有其他模块的内部操作十分重要。也就是说，它们没有对任何事物有依赖性，除了产出表示。也意味着这些产出如何得到也无所谓。这样模块就成为可交换零件。

如果你已经模块化产出数据 X、Y 和 Z，而且正在构建一个模块，需要输入数据 X、Y 和 Z，它可以很轻易地将你的新模块抓取到现有模块中。Kriens 强调这并不是实现它的最佳方法。你所有的新模块实际需要的是 X、Y 和 Z，所以那也正是依赖所在。

如果在彻底一点，另一个开发者需要取代现有模块，他们所要做的就是确保替换模块仍旧提供 X、Y 和 Z 即可。他们甚至可以将模块分离成两部分，只要在二者之间即可，它们仍旧提供 X、Y 和 Z。这对于未来的开发者来说更具有灵活性。

这也是为什么 Kriens 建议首先通过服务转移到 OSGi 的原因。构建真正的模块化模块确实很难，尤其是如果剩余的子系统还没有模块化。OSGi 框架感觉有点想只许成功，不许失败。但是你可以用模块化的方式设置服务，无论模块实际上是否实现了模块化。合适位置上定义良好的服务，能够以一种迭代的形式更轻松地转移到 OSGi。

让模块化盈利并不会立即将一个完全模块化的系统的所有好处呈现给你，但是如果实施的恰当，就会获得好处。这些好处可以用于证明模块化的价值，并鼓励其他开发者加入，构建现有的模块化，同时创造已近实现模块化的新的组件。

（作者：James Denman　译者：张培颖　来源：TechTarget 中国）

原文链接：http://www.searchsoa.com.cn/showcontent_53861.htm

OSGi 主张：封装是组件化安全途径

最近，Java 社区内，围绕的组件化的探讨主要是关于 OSGi 和 Jigsaw 之间谁才是实现组件化的正确途径进行争辩。Jigsaw 阵营组件基于 Java 类，而 OSGi 组件基于封装。Peter Kriens 强烈支持用 OSGi 系统来编写组件化代码。在这篇文章中，Kriens 通过解释 Java 组件封装接口到类：是最小化组件间耦合关系的安全途径。

关于 Jigsaw 读的越多，我越发感觉 Jigsaw 只是想解决相对次要的类路径装配（class path assembly）问题。应用本身就有数以百计的依赖装配类路径，这已经很困难了，Jigsaw 试图要自动化这些，期望组件路径调用类路径的时候，可以让痛点消失。然而，如果问题是类路径地狱（JAR hell、DLL hell）的话，简化类路径就像用吗啡治疗癌症一样。可能临时去痛，但是还是不能解决根本问题。

谈到 Maven，它以一种类似 Jigsaw 依赖架构的方式，基本上已经解决了类路径装配问题。在 JavaOne 的 Demo 中，Jigsaw 实际上使用 Maven 来自动化创建 Jigsaw 组件，但是发现他们必须修正依赖性，因为这些都是错的，或者指向组件路径不兼容的版本。

并不是说 Maven 不好或者手工编程开发者很傻，只是这种具体的组件到组件的组件依赖由于其聚合和传递属性，很大程度上增加了耦合。这种依赖架构是很难预防从网上下载 maven 的根本原因。这种模型依赖比吗啡还糟糕，这个痛点杀手实际上让根本的耦合问题更糟糕。

有趣的是，这种聚合/传递依赖问题是一个伪装的软件老问题。1996 年，Java 出现之前，类实际上遭遇了同样的问题。试图包含来自大型应用的类，常常拉进来大量其他的

类，这些类又拉进来更多的类，以至于无穷无尽。这个问题可以用 Brian Foote 的大泥球理论（越滚越大）来描述。

根本原因是类的实现聚合了大量的依赖来简化结构；使用库来让结构变得容易。然而，这些实现依赖，在类成为传递依赖链的一部分时就变成了问题，然后总是通过原始类重用将比实际需要多得多的类拖进来。

每一个 Java 程序员都知道解决方案，但是不系统：接口。通过打破实现类和客户端类之间的依赖，接口可以最小化耦合。接口确实让我们的类图看起来更加的解耦，而且尽可能利用了当今流行的反转控制系统。基于程序的接口成为可论证的 Java 最佳实践之一。

所以如果 Jigsaw 组件类似地类实现，那么接口的组件类似物是什么呢？目前的 Jigsaw 文档有组件为这个角色提供了计划陈述。其用例是组件可以提供任何的事情。在提议中，这只是一个名字和一个版本，所有的语义都是隐式名字，没有任何契约实施。

提供的主要用例是 IBM 可以说他们在其 VM 版本中提供了 JDK 7。只是用一个名字来模拟接口，而忽略了 Java 类型系统；不能在实现和客户端通过编译器和 VM 来证实。如果这样制造了接口，又没有包含任何方法，就是一个名字而已。例如 Runnable 不能指定 the run()方法。如果客户端调用 run()同一个编译器只是认为客户端想做什么事情。尽管在某种程度上可以减少键入，那还是 Java 吗？

对于 Jigsaw 组件，我们能不只是命名一个契约吗？没有一个使用现有 Java 类型系统的解决方案来核对一个组件是否提供了它所做的承诺吗？就像为类而生的接口？

那么，什么才是个真正的接口呢？一个接口的关键在于抽象描述了类实现的一部分，所以类实现随后就可以免费使用他所需要的任何东西来简化其实现工作。其外部契约因此

不是隐含实现的聚合和传递依赖，多重实现可以同不同的依赖共存。接口是方法的命名集，用于合适实现和用户之间的契约。

我们关注组件的时候，接口本身粒度很低，因为组件的粒度要比类大。取代一套方法（接口），一套类型是组件更为固有的契约，因为可以在软件契约中组件化不同角色（服务器、事件监听器、域类型等）。类型命名集本身是组件间更为安全的具体契约粒度。

Java 已经具有类型命名集的概念，称之为封装。封装本身就是适合组件契约的自然属性，因为在很多 JSR 和开源项目中，它已经作为具体的契约来使用了。最好的是，封装已经和 Java 类型系统完全集成。间接地满足了当今软件系统架构解耦和灵活性的需求。

基于模型依赖的组件封装可以提供很多封装，并且可以消费很多封装。就像类可以实现很多接口并使用很多类型一样。封装可以为组件提供类似接口组件的好处；允许组件优化实现选择，而不影响其提供的契约。OSGi 提供了 Jigsaw 所使用的直接的组件到组件的依赖（Require-Bundle），以及类似接口的封装所实现的（Import-Package）轻量型耦合。长期 OSGi 用户一关于 Import-Package 的优势，所以害怕 Require-Bundle 的长期破坏。

经过多年的经验，我们知道封装导入工作比在系统中请求 bundles 好得多。不幸的是，我们缺少语言来向人们解释，而人们对于这种类型的模块化缺少经验。在 C（++）之后开始学习 Java 的人，可能记得他们第一次看到接口的时候觉得多奇怪，新概念很难解释。也是最近我才明白封装实际上为类执行了接口起到的解耦作用；这个现在提供给了一种语言可以用来向 Java 用户解释 OSGi。所以，如果太多的狂热分子加入到 Java 模块化的谈论，我理解。但是说实在的，如果有人建议从 Java 中移除接口，你感觉怎么样？

（作者：Peter Kriens　译者：张培颖　来源：TechTarget 中国）

原文链接：http://www.searchsoa.com.cn/showcontent_55173.htm

采用 OSGi 框架开发项目的十个问题

OSGi 是 Java 领域里无可辩驳的最成熟的模块系统，它与 Java 几乎是如影相随，最早出现于 JSR 8，但是最新规范是 JSR 291。OSGi 在 JAR 的 MANIFEST.MF 文件中定义了额外的元数据，用来指明每个包所要求的依赖。这就让模块能够（在运行时）检查其依赖是否满足要求， 另外，可以让每个模块有自己的私有 classpath（因为每个模块都有一个 ClassLoader）。这可以让 dependency hell 尽早被发现，但是不能完全避免。

不过，Java 社区领袖 Adam Bien 最近在其博客中认为，从技术角度讲，OSGi 的确是实现模块化的可行办法，但 OSGi 的主要挑战不是技术，而是模块和 bundle 的管理。他建议在决定采用 OSGi 框架开发项目之前，考虑以下问题：

针对模块（bundle），采取何种版本 控制方案？大、小版本如何定义？

采用何种软件配置管理策略？允许开放和维 护模块所有版本的分支吗？预计要维护多少个分支？通过 SVN 吗？

- 在生产环境中，同时存在多少不同版本的模块？
- 针对模块和模块组合，如何进行测试？每一个版本都会显著增加复杂度。
- 采用何种发布管理策略？提供客户专属的模 块组合吗？缺陷修补/补丁策略是什么？

- 需要在系统运行中替换模块吗？如何处理正 在进行的事务？

- 对于 Eclipse RCP 应用，是否应该开放插件给最终用户？

- 采用何种软件分发系统？很多公司已经有了一套软件分发系统。应用和 JVM 经常打包到一个二进制文件中整体安装。增量更新几乎是不可能的。

- 模块之间如何交互？只通过 Java 接口吗？如果是，那么 JPA 实体的直接关联如何处理？

- 是否采用 Maven 描述模块和 OSGi？Maven 模块版本会在 OSGi bundle 版本中得到体现吗？

Adam Bien 认为，在深入思考这十个问题之后，OSGi 可能就不再是项目的最佳选择。

读者朋友是否同意 Adam Bien 的观点，针对他的十个问题，有何看法？在实际开发中又是如何解决的？欢迎踊跃发表意见。

（来源：TechTarget 中国）

原文链接：http://www.searchsoa.com.cn/showcontent_51347.htm

为使用企业级 OSGi 做好准备了吗？

为什么是企业级 OSGi？在介绍 OSGi 功能包的重要内容之前，值得花些时间概述一下它有助于解决的问题。

值得一提的是，组织使用 WebSphere Application Server 的功能部署和管理大量应用程序。这些应用程序通常包含公共库。在这种情况下，应用程序开发人员在其应用程序中包含相同的库就很正常。尽管这是每个应用程序获得预期库的一种安全方法，但是此策略可能会占用过量内存并使应用程序更新变得比较困难。软件供应商提供了解决此问题的解决方案，但他们是针对供应商的，并且会导致管理时间成本增加。

有时会在应用程序中使用供应商库，通常开发人员无法控制这些库的依赖关系。所以当一个应用程序使用的两个库具有不同且不兼容的内部依赖关系时，就会出现问题。例如，一个库可能需要 ASM 3.0，而同一应用程序使用的另一个库可能需要 ASM 2.0。要解决这种问题，通常需要更改代码。

总的说来，这两个问题在 Java EE 中通常都可以解决，那么为什么要忍受 OSGi 可以解决的问题呢？

OSGi 联盟的 EEG（企业专家组）成立于 2007 年，并且制定了 OSGi 功能包实现规范。EEG 的目的是查看最常用的 Java EE 技术（例如，JPA、JTA、JNDI 和 JMX 等），并构建使用部署到 OSGi 框架的应用程序绑定技术的标准方法。

开源和 OSGi

去年启动了提供 OSGi 企业规范实现的两个开源项目。Apache Aries 孵化器项目在 2009 年 9 月启动，并具有由 Apache Geronimo 社区开发的 Blueprint 实现。2009 年启动的企业模块项目（Enterprise Modules Project）由 Eclipse Foundation 主办，该项目与 Aries 项目的目标大致相同。

在很短的时间内，就建立了许多与 Apache Aries 相关的活跃团体。参与者包括来自 IBM、Progress、RedHat、Ericsson、SAP、Prosyst 和 LinkedIn 等的个人。由于 Blueprint 的最初贡献，JTA、 JPA、JNDI 和 JMX 的组件都已添加。演示如何使用这些技术（包括 Blog Sample 和 Aries Trader 应用程序）的示例是项目的重要部分。

Aries 积极开发的另一个领域是 “应用程序” 概念，这是将 bundle 分组为单个应用程序或 Enterprise Bundle Archive (EBA)的一种方式。

OSGi 功能包

OSGi 功能包提供了一个整合应用程序框架，有助于 Java EE 应用程序开发人员利用 OSGi 企业架构。该功能包整合了 Apache Aries 开发的组件和 WebSphere Application Server 运行时与管理。（JPA 2.0 支持也是功能包的一部分，但在本文中不会详细介绍它。）

具体来说，功能包交付 OSGi Blueprint Container 规范的开放社区和基于标准的实现，并且具有将应用程序组装、部署和管理为 OSGi bundle 版本集合的能力。常见 Web 应用程序的模块设计、简单的基于 POJO 的组件和高效数据访问需求，都可以使用 OSGi 应用程序和功能包的 JPA 2.0 组件解决。功能包的一些主要功能如下所述。下面将详细介绍这些功能以及其他一些功能。

Enterprise Bundle Archives

组成 OSGi 应用程序的 bundle 经过合理组装，可形成可部署的 Enterprise Bundle Archive 或 EBA。开发 Java EE 应用程序时，通常会将所有模块库（应用程序内容）放在 EAR 文件中。相反，尽管 EBA 中的应用程序元数据描述应用程序内容，但是无需在 EBA 中包含二进制内容。EBA 可以仅指从 bundle 库的相应阶段获得的 bundle。最常见的情形可能包括在 EBA 中放置其他 bundle 并包含其他内容。例如，Web 模块是应用程序的主要部分，所以它们自然位于 EBA 文件中；而非应用程序特定内容（比如共享库）从 bundle 库获得可能会更好。后续文章将探讨 bundle 库，以及如何且何时提供 bundle。

Web 应用程序

正如 OSGi Web Container 规范定义的那样，OSGi 应用程序的 Web 内容仅是具有其他 OSGi 元数据的 Web 模块。该规范定义了 Web 应用程序 bundle 所需的元数据。部署时，WebSphere Application Server 将 EBA 中的 WAR 文件转换为 OSGi bundle。将 Web 模块作为 bundle 部署的优势是，可以将其依靠的库从 WAR 移动到集中式、托管的、版本化的 bundle 库中，在 WebSphere Application Server 部署流程中使用该库。

Blueprint

OSGi 服务平台企业规范 介绍了 Java EE 中一种称为 Blueprint 的主要技术，这是 Spring Dependency Injection 模型的标准化。尽管企业级 OSGi 应用程序不需要使用 Blueprint，但是 WebSphere Application Server 实现提供了许多功能，这些功能使 Blueprint 成为开发人员乐于使用的一种技术。

OSGI 功能包的 Blueprint 实现提供了用户期望 Dependency Injection (DI)容器拥有的功能；例如，使用 POJO 进行构建的能力，以及使容器控制那些 POJO 生命周期的能力。通过 DI 容器（比如 Spring Framework）进行 Blueprint 实现的一个好处是 OSGi 整合。这意味着发布服务的 bundle 稍后可以被注入其他组件甚至是其他 bundle 中。此依赖性注入模型还支持在 Java EE 或 OSGi 运行时以外的时间进行单元测试。在 OSGi 功能包中实现的

Blueprint 容器是中间件的一部分，而不再是应用程序的一部分，这就为应用程序员消除了一个令人头疼的问题。

JPA 持久化

OpenJPA 是 WebSphere Application Server 默认的持久化提供者。OSGi 功能包包括 JPA 2.0 支持，所以开发人员可以使用 WebSphere Application Server V7 中现有的 JPA 1.0 支持或 OSGi 应用程序中的新 JPA 2.0 支持。

除了 OSGi 服务平台企业规范中介绍的 JPA 模型，功能包还包括扩展的 JPA 支持，向 Blueprint 组件提供由容器完全托管的 JPA，从而向目前正在使用 Java EE 的 JPA 的应用程序开发人员提供一种熟悉的开发体验。此功能包还包括对 Blueprint 的扩展，所以容器可以将持久化单元和上下文注入到 Blueprint bean 中。EJB 中应用程序开发人员熟悉的事务性行为同样存在于 OSGi 功能包中。

结束语

寻找可靠和已验证的模块化技术的企业 Web 应用程序开发人员，已经有很多都转向了使用 OSGi。IBM WebSphere Application Server V7 Feature Pack for OSGi Applications and Java Persistence API (JPA) 2.0 提供了一种环境，该环境支持 Java EE 开发人员使用 OSGi 功能构建应用程序。

用户对 OSGi 功能包的反馈、对 Apache Aries 的兴趣，以及对现有 WebSphere Application Server 的兴趣表明，Java EE 开发人员乐于使用 OSGi 技术，并期待它提供的许多好处。

(作者：Zoe Slattery 来源：developerWorks 中国)

原文链接：http://www.searchsoa.com.cn/showcontent_54563.htm

OSGI：让共享变得快乐

Windows 刚出现时，应用程序都是静态链接的，方式非常简单，应用程序运载着自己的链接，从没发生过混乱。然而，这种简单是有代价的。

很明显，每一个应用程序都要携带相同的库，是有存储成本的。当时人们认为 500M 的磁盘就很大了，所以存储成本非常昂贵。当然，也修复了一些故障。如果发现了安全故障，必须要重新构建每一个应用程序，确保故障在所有的地方都被解决了。如果想为已经安装的应用做一次更新，简单是一场噩梦，通常是不可能的。曾经有个问题，某些库文件必须从平台得到，例如，设备驱动程序和操作系统库文件。这些库文件不能静态链接。

微软提出的解决方案是动态链接库（DLL），这个概念，就是当时 Unix 用户非常熟悉的共享库。它是这样工作的：加载应用程序时，加载器会自己加载 DLL 到内存中，在 DLL 中链接应用程序的代码和数据。DLL 还可以信赖其他的 DLL，所以加载器可能会需要递归的链接 DLL，结果常常是一个复杂的信赖关系图。开发者接受了它，认为它不错。也就是说，开始大量的使用这个功能。然后就乱套了，为什么呢？因为共享是很难的，就像我们第一次在幼儿园学习一样。

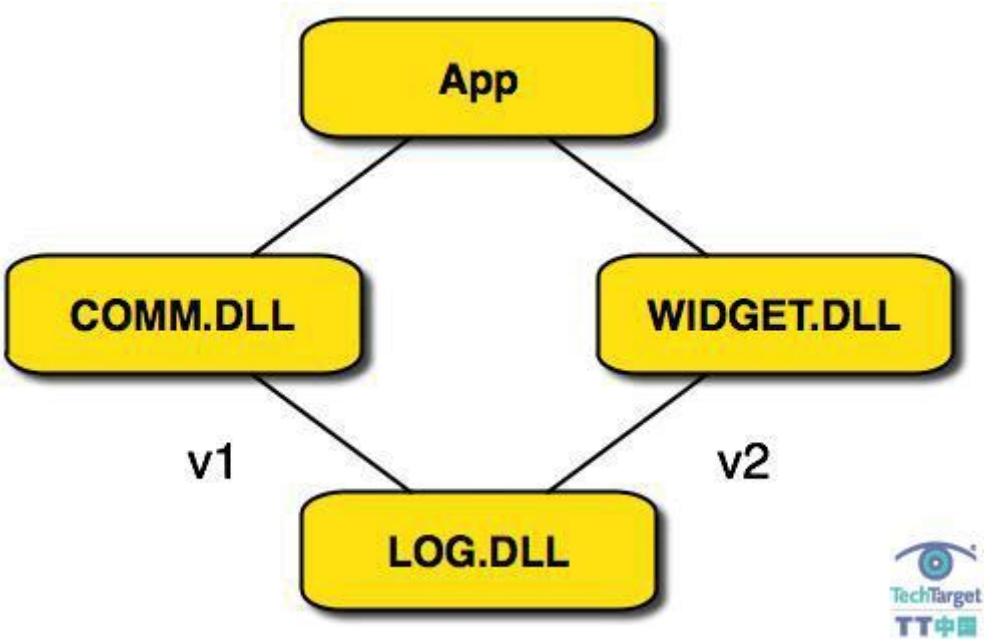
共享

软件共享问题是所有的参与者必须同意一直都要共享。应用程序开发者熟知捷径，侵入库文件，以获得额外的特性，或者修补一些另人讨厌的故障，这将永远流行。不幸的是，一个开发者的修复办法让另一开发者感到恐慌，尤其是，其他开发者修复的问题。所以，修改过的 DLL 文件总是不能和所有使用它的应用程序兼容。在 C++中，在类中添加一

个方法，总是使编译后的代码与 DLL 不兼容。应用开发者始终没有考虑共享的后果，践踏（覆盖）已经安装的 DLL 文件，引起其他应用程序崩溃，使很多系统管理员产生阴影。

最初，DLL 甚至是没有版本。后来才添加上的，但是，一些开发人员不太关心这些，包括微软自己也没有正确的 DLL 版本，所以，版本不能解决这个烂摊子。然而，即使版本、所有组件的标记都正确，人们也会遇到无法解决的情况，在信赖中造成不一样性。

我来解释一下。如果你有信赖关系图，你可以通过不同路径依赖相同的 DLL，但是，每一个系统规定的参数可以是不同的。例如，一个颇为流行的 LOG DLL。WIDGET DLL 以及 COMM DLL 都使用这个库。见相邻的图片。



如果 COMM.DLL 需要 LOG.DLL V1，WIDGET.DLL 需要 V2，加载器必须做一个选择，LOG。DLL 只能在 V1 或者 V2 中链接一次。这就是所谓的“原子”问题，图片看起来像个钻石。在大的依赖关系中，这些不一致的问题是不可避免的，除非有单一的代码，所有组件在同一时间变化。

原子问题的解决方案是要始终向后兼容。如果 LOG.DLL V2 向后兼容 V1，那么聪明的加载器会选择最新的版本。然而，迫使后代总是保持向后兼容现在的特性不是最佳方案，对开发者来说是残酷的。

共享的一个解决方案是隔离开来：停止共享。在 Windows 中，解决 DLL 痛苦的情况，应用程序因此允许他们自己的 DLL 搜索路径，使他们能有一个私有的 DLL 复本。这又回到了原点，只是增加了一些额外的复杂性：浪费内存，没有单独的故障修复的地方等。

为什么共享呢？

共享的所有问题，它值得吗？在 Java 中，WAR 概念非常流行，清楚的隔离开来：每一个 WAR 包含所有的依赖，除了 Java 环境和应用程序服务器提供的库文件。是的，部署时有很多严重的问题，因为大公司在许多不同的 WAR 中复制了相同的库文件，引起了上传延迟等问题。在一些网站，虚拟机试图处理数量庞大，往往又不必要的字节码时会爆裂。然而，这些问题都有解决方案，应该明确公平的共享等问题。

所以，共不共享是一个问题。

我相信共享是必然发生的，从八十年代我工作的时候就有了这个信念。软件应该通过明确的接口，从组件上建立一对一的合作。这是解决这个大泥球问题的唯一方式，代码的维护和修改变得越来越难。如果你的软件依赖关系图看起来像 Jackson Pollock，你就会有这样一个大泥球。

在组件模型中，我们必须把开发者（开发组件）和汇编者（把组件组装到应用程序中）的角色分开。在我们行业中，许多工作都朝不同软件子系统的更多的自主权方向发展。开源项目确实接近组件，但是，不幸地是它不是一个稳定的模块系统，该系统使这些组件更容易地使用。

更高的自主性需要共享接口，并让所有的组件联系起来，尽可能的保持最大的灵活性。组件开发人员甚至不喜欢看到组合组件，以防止不必要的依赖关系。组件越少，出现的错误越少。在组件开发中，只有契约是可见的。在这个模型中，这些组件组合到一起，形成了一个应用程序。

在 1998 年，考虑到了这种合作模式，我们开始开发 OSGi。我们需要一个模型，不同厂商的组件可以组合成一个点对点的方式，这些组件事先都不了解。在一个受限的环境中，但是组件可以彼此找到并一起工作，而不用受限。为了实现这个目标，我们知道，我们必须解决共享问题，在这 13 年中，我们在模块化中学到了什么？

实施

没有实施的模块化不是模块化。你违反模块的边界，除非你受到惩罚，否则你就不能模块化工作。迁移到 OSGi 最另人失望的是：找出与架构不一样的模块化代码，和依赖关系图，使人们相信。不幸地是，人们往往会生气、并指责使者。

隐藏

迄今为止，共享问题最好的解决方案是不共享。任何共享都会在组件生命周期的后期阶段有一个隐性的开销。随着时间的推移，任何共享方面的内容都必须谨慎地修改版本，在他的演化过程中是有限制的。共享非常昂贵！出于这个原因，OSGi 在 bundle 里为你提供了一个私有的 Java 命名空间，尽可能多地隐藏命名空间外的代码。无论你做什么，都不要影响 bundle 外的人。不同的 bundle 实际上可以包含相同的类，而不会引起命名问题。共享很好、隐藏更好、一开始就没有事情最好。

依赖的花销

每一个依赖都是有代价的，但是你会感到惊讶，到底有多少人不知道他们的依赖关系是来自哪里呢。一旦我发现超过 30M 的依赖，就会把他们放到一个单线程中，参照 apache

公共资源集合。很多 WAR 和 Maven 项目都有很多模糊的依赖链，因为很难估计出在现今的 Java 中实际有多少依赖。

包共享

我们发现，对于 JAVA，包是共享的正确粒度。类对于共享来说粒度过细。在同一个包中，它们过多地链接到与其相邻的类中。输出类在 Java 包中也重叠了。JAR 对于共享来说，太难处理，因为是一个在代码中共享的包聚合。这种聚合在版本控制和依赖管理上有让人不爽的副作用。

基于 API 编程的意外结果

基于 API 编程，指的是，一个 API 可以通过不同的实体实现，要求每一个再访问版本为我们所知道。在 Unix 包管理系统中，它是软件依赖管理模型的鼻祖，版本控制基于请求提供者的消费者。在 Java 中倾向于盲目追随这些模型，但是模型是错误的。对于白费力气重复工作而言是个很好的战略，但是使用基于 API 编程模型，我们实际上得到了一个第三方：API/接口/契约。结果是这个三元组对于兼容性规则具有重大意义，这就是依赖模型。

版本控制

为了让组件可以是当地连接，需要独立于其他组件的功能分解其需求。在这样一个线路模型中，包的正确描述是最重要的。OSGi 中，一个输出可以声明版本，输入可以根据兼容性声明一个版本范围。OSGi 规范指出了语义版本的规则，模型是版本的一部分，意味着有一个良好顶合一的兼容性。很多其他的模式只是执行反向兼容性，没有认识到消费者/接口/提供者这个三元组。OSGi 版本模式通过自动化工具很好地实现了这个。OSGi 中，你可以清晰无疑地声明，你不会和优先版本和谐相处。能表达你不支持旧版本的反向兼容性是组件系统中必不可少的内容。

多版本依赖

依赖于同一包的多版本不是每个人都渴求的，但是当应用使用了足够的外部依赖，就称为不可避免的了。在 JAVA 社区中，有一种倾向于忽略这种麻烦多版本的趋势，就好象他们没有这个问题。例如，Maven 中第一个版本在依赖中还在使用，尽管随后的依赖有了更高的版本。在大型类路径上发现相同库的多版本很正常，之所以奇怪是因为我们希望花更多的时间在语言的类型安全上，但由于忽略版本，远离了运行时的诸多好处。OSGi 中，依赖需要小心管理，它是一项精密科学，没有模糊性和启发性。OSGi 框架通过链接 bundle 到期正确的版本解决了域问题，但是之允许没有冲突的 bundle 之间的协作。在 DLL 例子中，OSGi 可以轻松处理 App 和 COMM 以及 App 和 WIDGET 之间的协作，而不用交换 LOG 对象。实际上，OSGi 支持协作和共享，但是当需求的关键原因之一是应用服务器厂商过多地使用 OSGi，它也是孤立的。

总结

OSGi 核心规范的大部分是在模块层，这一层中我们定义了共享规则。这些规则通常是复杂的，难以理解，而且需要很多关于命名域、类型系统以及 Java 内部的知识。然而，这些规则只需要少量 OSGi 框架开发者理解。作为回馈，OSGi 为模型开发者提供了稳固的共享模型，这个模型强大，而且具有惊人的易用性。

看到了 Unix 系统（如 Debian）如何指导 Jigsaw 的设计，是一种悲哀，这里没有使用 OSGi。然而，这些系统包强大之处是，他们解决的问题，表面上仅仅是类似于一个 Java 模块。就 Jigsaw 而言，Java 社区面临一个漫长而又曲折的道路。看完这篇文章之后，可以清晰的知道，OSGi 会让这条路变得通畅，它只是一个信使，告诉你必须运行无模块化的代码。

（译者：刘志超　来源：TechTarget 中国）

原文链接：http://www.searchsoa.com.cn/showcontent_51485.htm

http://www.searchsoa.com.cn/showcontent_51486.htm

OSGi 是所有模块化应用的框架

OSGi（Open Service Gateway Initiative）是关于通过在框架中安装一套（可重用的）模块，如何创建一个有用的应用。基本上，我们在 1998 年开始着手解决组件编程模型；令人惊奇的是，我觉得我们主要的成功来源于很多成功的 OSGi 项目。那么，任务已经完成了吗？

不见得，看看企业世界中关于 OSGi 的探讨：Spring、Jigsaw 等等。很多人喜欢它，一些人似乎极度讨厌它（伙计们，别烦了，它就是个技术而已）。我的解释是 OSGi 违反了一些企业开发者的基本期望值，OSGi 应该就是类路径上的一个库，提供一些有用的且便捷的功能，但是在别的方面要置身事外。

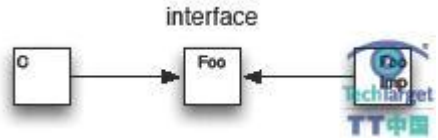
模块化是关于你的代码基的；OSGi 仅仅是一个微小层，为性能良好的模块提供运行时环境。OSGi 并不是什么会让应用瞬间模块化的“神秘酱汁”。就像面向对象强制结构化程序员差异思考，模块化迫使现在的开发者摆脱框框（和代码基），开始收获好处。模块化是架构！

正如我们长久以来在 OSGi 内部进行模块化，我们发现很多现有的软件模式本质上是非模块的，而且已经开发了模块替代物。此外，我们发现这些模块化解决方案仅仅被开发者所理解，这些开发者是实际工作在模块化环境中的。对于其他人来说，它就像是官方文件。

就像所有的 next-next 问题一样（在你解决下一个问题的时候就会遇到这个问题），模块化 OSGi 解决方案常常会让人觉得过于复杂且不必要，如果一个人还没有体会道 next-next 问题的痛苦，这样像也合乎逻辑。本文中我为目前的一些最佳实践来尝试解释稳健的模块化所带来的意想不到的结果。

Java 中的模块化是关于（简言之）将你的 JAR 转到一个模块中。模块依赖于其他模块，而且它们可以隐藏实施细节。模块之间的领域就仅仅是精确的共享包。正如实施细节是保密的，我们可以在未来的版本中改变它们，这也正是模块化的主要好处。在我们的编程模型中，这种隐藏会有什么影响呢？

行业中一个完全的最佳实践就是接口基于编程。取代使用实施类，我们使用接口。接口消费者不会同任何提供者的实施细节耦合。这种解耦为我们提供了重用和替换；测试变得更容易等等。我还没有遇到哪个企业开发者不理解这些好处的。下图展示的就是这种类型的耦合模型：



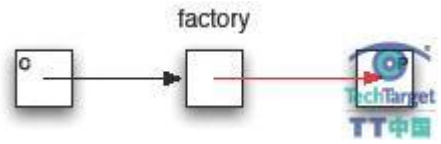
耦合模型

接口允许我们描绘出我们系统中耦合的一种乐观观点，在 UML 中，消费者和提供者之间没有耦合、不幸的是，这个 UML 并不是真实世界的实际反映；在系统的某个深处，不可避免的潜藏着耦合的实施类。业内已经同这个问题斗争了很长时间，通过大量著名的模式来解决这个问题。坏消息就是几乎所有的模式都破坏了模块性。

例如，我们有一个消费者（C）和一个提供者（P）模块，提供者提供 FooImpl 类用以实施接口 Foo。传统方式下，消费者仅仅是“new FooImpl()”来为 Foo 接口创建一个实现：

FooImpl 类

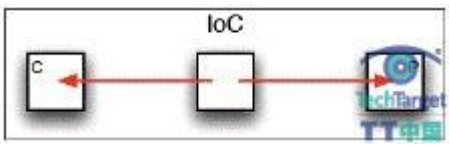
如红箭头所指出的，随着消费者请求访问提供者的一个实施细节类 FooImpl，这个就触犯了模块的边界。即使没有模块，这个对于使用也是不舒服的，而且导致在四人帮（Gang of Four）的设计模式书中大多数成功的模式：Factory pattern（工厂模式）。取代消费者抓取一个实施类，称之为 FooFactory.create()方法，然后知道如何创建一个期望的实施。



工厂模式

这个迂回对于我们的消费者来说是好消息，因为真正从提供者这里解耦了。然而，正如图片所示，Factory 仍旧不得不违背提供者的模块边界，也就是 FooImpl 类。是的，它可以被隐藏在文本/XML 源中，但是这并不意味着耦合就不存在了。这也是一种有点尴尬的模式，因为它也具有所有全局和单独的问题。

在 2000 年初，Rod Johnson 写了一本非常成功的书，主要关于控制反转（IoC），并演示了对于企业应用来说，这个模式是多么有用。随后，他用开源 Spring 框架普及了这个模型。IoC 或者好莱坞原则（Hollywood principle）：“不要呼叫我们，我们来呼叫你，”意味着消费者失去了其所拥有的任何控制。取而代之的是在需要的时候创建一个对象，在时间注入前它通过一套方法或者构造函数获得这个对象，a.k.a.依赖注入（Dependency Injection）。在 IoC 图中，箭头展示了控制的方向。



IoC 图

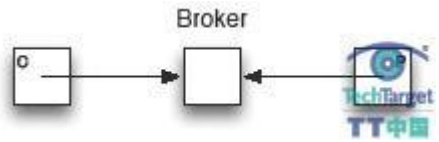
不幸的是，IoC 容器让实施细节的耦合更糟糕。IoC 容器需要知道消费者（在哪里注入）以及提供者（创建了什么）的实施细节。因而，在我们的 UML 图表中，令人难以置信地它好像解耦了。罪魁祸首就是 IoC 配置，浓缩了（因此耦合）实施细节，例如 Spring 中的 XML。

问题是 IoC 要求我们放弃在代码中的控制，我们实际上完全了解的唯一的地方就是内部细节。没有一种模式我们可以拥有所有优势，但是也不会违反模块边界吗？消费者和提供者就没有哪里都没有边界贯穿其中吗？

正如我们不愿意知道制造了多少香肠，但是仍旧希望吃掉它们一样，这是个比喻。我们能否不用知道其任何生产过程而得到一个德式小香肠（Bratwurst）呢？当然，在实际世界中，我们只要去当地的商店，在冰箱里拿出来就好了。

那么，商店知道它们是怎么做出来的吗？也不知道，它不是工厂，屠夫每天来提供新鲜的香肠，只有他们知道那个“残酷的细节”。这里的秘密就是中介：那个商店。消费者和提供者拥有控制，商店是一个被动的中间人。这就是典型的发布、查找、捆绑模式。

如果一个提供者模块可以发布其实例，一个消费者可以查找到这些实例，然后中介将这一切结合在一起。在 OSGi 中，这个模式被作为“μservices”来实现，同时它也支持 Java Service Loader。



μservices

因此，我们都在涅槃，除了现在的令人讨厌的动态依赖问题。正如消费者和提供着都拥有独立控制，我们可能不再去上商店，也不用找到香肠。使用 Java 代码来控制这种依赖性很棘手，会导致很多样板代码（boiler plate code）。

早期 OSGi 版本强迫你或多或少的用手来操作；做起来有难又讨厌。然而，对于依赖注射来说是个完美的工作。每一个模块能够制定其依赖性；中心控制器可以监控这些依赖性，并按需激活/停用组件。

在 OSGi 中，μservices 的 DI 由 Spring DM、Blueprint、iPojo、Declarative Services 以及其他完全彼此协作的机制来提供。中间件模式是我知道的唯一一个为消费者和提供者提供控制的模式，不会强迫他们来无偿地暴露其私有部分。

讲了这么多，我希望对于为什么模块化破坏了如此多的代码有个清晰的思路：我们的产业正在使用很多模式，这些模式并不能同模块世界和谐相处。在模块边界收到尊重的前提下，OSGi 出人意外地灵敏，但是当你的代码试图触犯它们的时候，它就有些残忍了。这也正是它的本质所在，因为如果没有这个强制执行，模块化就是个空壳。

（作者： Peter Kriens 译者：张培颖　来源：TechTarget 中国）

原文链接： http://www.searchsoa.com.cn/showcontent_50311.htm
http://www.searchsoa.com.cn/showcontent_50313.htm

用 OSGi 架构实现模块化

我认为代码共享是不可避免的，而且这种信仰趋势我从八十年代就开始从事自己的工作。软件应该从组件创建，这个组件在点对点的基础上，通过显示接口协作。这是解决这个大泥球问题（big ball of mud）的唯一途径，这里的大泥球指维护和修改代码基变得越来越难，由于这些代码相互纠缠。如果你的软件依赖范式看起来像 Jackson Pollock（20 世纪美国抽象绘画的奠基人之一。）的绘画，你就知道什么是大泥球了。

为一个模糊的基础架构塑型的方法之一就是模块化，和更加基于组件的模型。在一个组件模型中，我们将开发者（组件制造者）的角色和组装者（将组件组合成一个应用）的角色分离开。组件开发者应该不了解其他组件（即便是协作组件）。这种盲区可以组织实施中不必要的依赖。假设组件少，BUG 也就少。只有契约做组件开发中应该是可视的。

业界很多的工作对于不同软件子系统更好的自主权具有指导意义。SOA 很明显是基于点到点的模型，但是大型软件库也有自己的线程和控制模型，而这些都是独立于主应用的。开源项目更接近组件，但是不幸的是，我们没有选定一个模块化的系统，这个系统可以让这些组件更易于使用。

这种更高层次的自主权的逻辑结论就是需要共享接口，还要允许组件尽可能在循环的最后链接在一起，以便最大化灵活性。一套组件可以组装在一起形成一个统一的应用。

1998 年，因为这种协作模型的想法，我们开始开发 OSGi。我们希望这个模型可以再没有优先知道其他组件的情况下，以点到点的形式，同来自不同厂商的组件进行协作。我们希望这个组件可以同其他组件集合起来共同工作，取代约束孤岛。为了实现这个目标，我们知道我们必须解决共享问题：DLL Hell。

在这十三年中，我们从模块化中学到了什么

执行

如果模块化没有执行，就不用存在了。在你违反模块边界的时候，除非有人打你的手掌，否则你就不能模块化。这也是人们转向 OSGi 时最沮丧的一点：找出代码基并不像模块化架构和依赖范式那样让人信服。

隐藏

有时，你必须保守秘密。共享问题最好的解决方案有时候不是简单的共享一切。任何共享的事物在晚期维护组件的阶段都有一个隐藏成本。任何共享方面必须小心描述，限制其发展。共享是昂贵的。

因为这个原因，OSGi 在你的 bundle 中为你提供一个私有 Java 命名域，用于尽可能多地隐藏其他地方的代码。在这里无论你做什么都有保证不会影响到 bundle 外的其他任何人。不同的 bundle 实际包含同样的类，而不会导致命名问题。

依赖成本

每一个依赖都有一个成本，但是你可能会吃惊于，如此多的人不能精确知道自己的依赖来自哪里。一旦我发现超过 30M 的依赖，就会把他们放到一个单线程中，参照 apache 公共资源集合。很多 WAR 和 Maven 项目都有很多模糊的依赖链，因为很难估计出在现今的 Java 中实际有多少依赖。

包共享

我们发现，对于 JAVA，包是共享的正确粒度。类对于共享来说粒度过细。在同一个包中，它们过多地链接到与其相邻的类中。输出类在 Java 包中也重叠了。JAR 对于共享来说，太难处理，因为是一个在代码中共享的包聚合。这种聚合在版本控制和依赖管理上有让人不爽的副作用。

基于 API 编程的意外结果

基于 API 编程，指的是，一个 API 可以通过不同的实体实现，要求每一个再访问版本为我们所知道。在 Unix 包管理系统中，它是软件依赖管理模型的鼻祖，版本控制基于请求提供者的消费者。在 Java 中倾向于盲目追随这些模型，但是模型是错误的。对于白费力气重复工作而言是个很好的战略，但是使用基于 API 编程模型，我们实际上得到了一个第三方：API/接口/契约。结果是这个三元组对于兼容性规则具有重大意义，这就是依赖模型。

版本控制

为了让组件可以是当地连接，需要独立于其他组件的功能分解其需求。在这样一个线路模型中，包的正确描述是最重要的。

OSGi 中，一个输出可以声明版本，输入可以根据兼容性声明一个版本范围。OSGi 规范指出了语义版本的规则，模型是版本的一部分，意味着有一个良好顶合一的兼容性。很多其他的模式只是执行反向兼容性，没有认识到消费者/接口/提供者这个三元组。OSGi 版本模式通过自动化工具很好地实现了这个。

OSGi 中，你可以清晰无疑地声明，你不会和优先版本和谐相处。能表达你不支持旧版本的反向兼容性是组件系统中必不可少的内容。

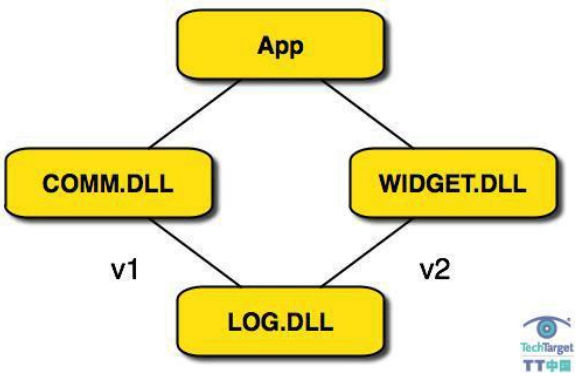
多版本依赖

依赖于同一包的多版本不是每个人都渴求的，但是当应用使用了足够的外部依赖，就称为不可避免的了。在 JAVA 社区中，有一种倾向于忽略这种麻烦多版本的趋势，就好象他们没有这个问题。

例如，Maven 中第一个版本在依赖中还在使用，尽管随后的依赖有了更高的版本。在大型类路径上发现相同库的多版本很正常，之所以奇怪是因为我们希望花更多的时间在语言的类型安全上，但由于忽略版本，远离了运行时的诸多好处。

OSGi 中，依赖需要小心管理，它是一项精密科学，没有模糊性和启发性。OSGi 框架通过链接 bundle 到期正确的版本解决了域问题，但是之允许没有冲突的 bundle 之间的协作。

例如，通用的 LOG DLL 很有欢迎。WIDGET DLL 将该库作为 COMM DLL 很好地运用，看下图：



在这个 DLL 例子中，OSGi 可以轻松处理 App 和 COMM 以及 App 和 WIDGET 之间的协作，而不用交换 LOG 对象。实际上，OSGi 支持协作和共享，但是当需求的关键原因之一是应用服务器厂商过多地使用 OSGi，它也是孤立的。

总结

OSGi 核心规范的大部分是在模块层，这一层中我们定义了共享规则。这些规则通常是复杂的，难以理解，而且需要很多关于命名域、类型系统以及 Java 内部的知识。然而，这些规则只需要少量 OSGi 框架开发者理解。作为回馈，OSGi 为模型开发者提供了稳固的共享模型，这个模型强大，而且具有惊人的易用性。

（作者： Peter Kriens 译者：张培颖 来源：TechTarget 中国）

原文链接：http://www.searchsoa.com.cn/showcontent_51061.htm