OSGi技术框架选型研究

目录

[1. 目的 2](#_Toc358966918)

[2. 概念和术语 2](#_Toc358966919)

[一、 组件 2](#_Toc358966920)

[二、 组件模型与组件框架 2](#_Toc358966921)

[三、 模块与组件 3](#_Toc358966922)

[四、 OSGi与组件 3](#_Toc358966923)

[3. OSGi规范研究 3](#_Toc358966924)

[一、 Service Listener 4](#_Toc358966925)

[二、 Service Binder 5](#_Toc358966926)

[三、 Dependency Manager 6](#_Toc358966927)

[四、 Declarative Services 6](#_Toc358966928)

[五、 iPOJO 7](#_Toc358966929)

[六、 Blueprint 8](#_Toc358966930)

[4. 技术框架选型分析 9](#_Toc358966931)

[5. 结论 10](#_Toc358966932)

[6. 总结 10](#_Toc358966933)

[7. 附录：实例研究 11](#_Toc358966934)

# 目的

早期的J2EE规范以EJB为核心，对于很多应用显得太过重量级，后续发展起来的以Spring为核心的技术框架，对轻量级企业应用开发提供了很好的解决方案，对于提升开发效率起到了较大的促进作用，并逐步影响到了J2EE规范的定义，丰富了J2EE规范的内容。随着OSGi的引入，我们同样需要考虑基础的OSGi核心规范和实现，是否能够满足我们企业应用开发的需要，OSGi和Spring、Hibernate到底是什么关系？我们是否还能在OSGi环境下使用它们，有没有更好的技术框架为我们提供支撑，这些是本次预研的主要目的。

本文档主要针对企业级应用开发所需的“高级组件框架”进行选型预研，同时选择确定要在产品中使用的主要OSGi企业级规范（JPA、JNDI、Web Applications）的开源实现。

# 概念和术语

## 组件

组件（组合构件）是一种粗粒度的功能构建块，它们通过接口提供功能，一般组件具有某种显示方式声明它们所提供的接口和引用的外部接口，组件可通过其它组件的接口使用其它组件提供的功能。给定方法的组件通常根据组件模型定义的特定模式进行编程。组件框架用来执行组件。通常组件被打包为独立的部署但愿，例如JAR文件或DLL，但这不是必须的。

借助面向组件我们可以把已有的、可重用的组件搭建在一起，从而简单快捷地创建应用，这个乐观的目标的实际价值有待商榷，但遵守组件模型确实可以带来很多好处，提高了代码的可重用性。但是使用组件也会带来一些弊端：组件框架过于笨重、以来过于复杂、需要组件框架提供额外的功能、编译时或运行时问题会让调试变得很困难等等，当然随着开发和调试工具的丰富，这些问题也会不断改善。

## 组件模型与组件框架

组件模型描述了组件是什么样子，如何与其它组件交互以及有何功能（如生命周期或者配置管理）。组件框架实现支持组件模型和执行组件所需的运行时环境。两者并不是严格的一对一关系。一个组件模型可能由不同平台的不同组件框架分别实现。一个组件框架可能支持多个组件模型。组件框架不受其支持的组件模型的约束，可能提供其它功能。

## 模块与组件

模块和组件都会提供服务并使用服务。都被打包成独立的部署单元。它们可以被认为是同一个概念，但是组件和模块用于不同的目的，组件是由最终可以打包到模块中的代码组成的。

模块处理代码封装和代码之间的依赖关系。组件解决实现更高层次的功能和构件间的依赖关系。组件需要管理代码依赖，但是在技术上不需要模块化系统来处理（通常由程序员通过类路径来处理）。

总之，你可以认为模块用来处理静态代码和编译时的依赖关系，而组件解决实例和执行时的依赖关系。

## OSGi与组件

OSGi的核心规范定义了一个组件模型和框架用来运行相应的组件。OSGi核心规范所定义的组件模型是面向服务的组件模型。

# OSGi规范研究

随着OSGi的流行，很多企业应用也考虑引入OSGi，OSGi核心技术规范并未涉及企业应用，但在OSGi R4版本之后逐步引入并完善了OSGi企业级应用规范的内容，对J2EE相关的规范（JTA、JDBC、JMX、JPA、JNDI、Web Applications等）提供了支持。

对于高动态高可扩展的应用，OSGi是一个非常好的平台。但是，也因此增加了复杂性，开发中对服务的依赖变得复杂。服务的关系管理通过组件框架来实现，目前OSGi相关的组件框架有以下6种，Service Listener是原生的OSGi组件框架，后续逐步改进衍生出Service Binder、Dependency Manager、Declarative Services，其中Declarative Services是最新的进化版本，Declarative Services 、iPOJO、Blueprint是三种高级组件框架，Declarative Services 、Blueprint是OSGi企业级规范中的内容，iPOJO是Felix的OSGi扩展子项目，不属于OSGi规范的内容，以下是详细介绍：

1. Service Listener
2. Service Binder
3. Dependency Manager
4. Declarative Services
5. iPOJO
6. Blueprint

后面分别简要介绍下。

## Service Listener

这是OSGi中原生的Service依赖管理机制，是最简单直接的方式，其基本原理非常简单，标准的注册/查找：

1. 被依赖的Bundle通过BundleContext.registerService()方法注册服务到系统中。
2. 使用依赖的Bundle在Start时通过BundleContext的getServiceReferences()/getService()来查找依赖的Service。
3. 使用依赖Bundle通过BundleContext.addServiceListener()来分别注册ServiceListener。
4. 在被依赖的Bundle/Service状态发生变化时, 使用依赖Bundle通过ServiceListener的serviceChanged()得到通知并作出调整。

在这种方法中，使用依赖的Service必须进行大量的编码工作来完成对依赖的Service的关系管理，需要处理琐碎细节如各个Service的运行时状态变化。为了减少工作量，OSGi设计了ServiceTracker来简化对依赖Service的编码工作，即ServiceTracker将负责处理上述步骤中的2/3/4。

经过ServiceTracker优化后的Service Listener机制，还是存在一些缺点：

1. 编码量还是不小，尤其对于依赖较多的场景；
2. Activator 还是太复杂了，尽管已经很努力的试图简化；

对于一些业务逻辑简单的Service，如果依赖的Service比较多，那么Activator的逻辑和代码实现远比Service本身的逻辑和实现要复杂，这违背了我们使用框架简化开发的初衷。

1. Activator对测试不利；

这个是Activator的复杂性造成的，由于Activator中存在大量的依赖处理逻辑，理所当然的会增加测试的复杂性。

总之，Service Listener 机制下，管理Service依赖对于开发者来说完全是个重体力活: 很重要，经常要做，容易出现错误, 出错时不容易测试。而且，这些工作都不是Service业务逻辑的组成部分，不能带来直接收益。简言之，吃力不讨好，一不小心就搬石头砸自己的脚。

更重要的，从分工的角度上将，开发人员应该将更多的精力投入与应用的逻辑，而不是OSGi的底层实现机制。因此，从2000之后，就陆续有人开始考虑对此改进。

## Service Binder

Service Binder OSGi针对这个问题的一个尝试，基本出发点非常明确：希望找到一个通用的方法来简化OSGi下的动态Service依赖管理。

最初开始这个工作的是两位大牛，Richard S. Hall和Humberto，大概在2002年的时候开发完成。

看我们看Service Binder是怎么做的，基本步骤为：

1. 提供一个org.apache.felix.servicebinder.GenericActivator；

现在Bundle的Activator只要简单的继承GenericActivator就可以了，所有的代码实现在GenericActivator提供，减少代码的目标顺利达成。这个可以说足够简单到没有办法再简单了。

1. 通过metadata.xml 来实现Service的依赖注入
2. 具体的Service的实现类基本就是一个干净的POJO了

当然还是需要实现bind-method/unbind-method 两个方法，好在这个是通过metadata.xml来做映射，不是另外提供接口定义，因此勉强避开了"侵入"的问题。

Service Binder 机制在减少Activator方面成效显著，但是引入的metadata.xml文件似乎又带来了新的问题，xml配置文件的可维护性个人感觉值得怀疑。用过EJB的同学都明白，EJB的部署文件有多令人烦恼。

当然Service Binder 的思路非常的正确：通过引入了自动化的Service依赖关系管理，简化开发，允许开发人员可以集中精力在Service的实现上，而不是疲于处理依赖关系管理。

Service Binder的实现似乎并没有被推广开，因为很快OSGi就在2004年的OSGI R4规范中引入了Declarative Services。因此Felix也就终止了对Service Binder的后续支持。

## Dependency Manager

继Service Binder之后，Felix又提供了名为Dependency Manager 的Service依赖管理方式，对比Service binder，个人感觉这个Dependency Manager 只是针对Service Binder的一个改进：将metadata.xml 文件取消，由新引入的DependencyManager来实现metadata.xml 文件的功能。原来在metadata.xml 文件中的配置转变为在Activator中通过代码调用DependencyManager来实现。

Dependency Manager其实现的方式为：

1. 提供org.apache.felix.dependencymanager

.DependencyActivatorBase Bundle的Activator需要继承DependencyActivatorBase，并实现DependencyActivatorBase要求的init()/destroy()方法

1. 在init()中，可以通过DependencyManager 对象来注册服务，并注明依赖。
2. 具体的Service类可以是POJO，DependencyManager 通过反射来注入依赖的Service。

## Declarative Services

2004年发布的OSGi的4.0版本中，加入了Declarative Services（声明式服务），据说是从Service Binder进化而来。

Declarative Services的实现方式和Service Binder的确非常相似：

1. 同样是需要一个xml文件来配置

在 bundle manifest中增加Service-Component 的header

提供的功能和Service Binder很类似，配置方法也很接近。

1. 同样的提供bind/unbind 方法的配置

对于每个依赖的service，都可以在配置文件中通过指定bind/unbind 方法来处理依赖的状态变化。

此外，Declarative Services 提供两个特殊的lifecircle方法：

protected void activate(ComponentContext context)

protected void deactivate(ComponentContext context)

如果Service实现类提供了这两个方法，则Declarative Services 会自动识别并调用这两个方法。注意这两个方法没有接口定义进行强约束，只是一个约定而已，估计是为了避免OSGi对service的侵入。

Declarative Services 是OSGi规范的一部分，因此Declarative Services的支持自然是各个OSGi实现都提供的。

从功能上将，Declarative Services 基本已经不错了，但是大牛们的脚步并未就此停住。

## iPOJO

2005年，Richard 开始考虑使用字节码生成技术来进行创建组合服务的改进，另外一个牛人Peter Kriens也同样的工作并实现了一个原型。

2006年，Clement Escoffier 和 Richard 开始开发iPOJO，合并了Peter Kriens之前的工作内容，这就是iPOJO的由来。

对于iPOJO的定义，Felix的iPOJO页面有如下说明：iPOJO是一个服务器组件运行时，目标在于简化OSGi应用开发。原生支持所有的OSGi活动。给予POJO的概念，应用逻辑开发简单。非功能性的属性在运行时被注入到组件中。

同样看看Felix对iPOJO优点的说明：

1. 组件被作为POJO开发，不需要其他任何东西；

2. 组件模块是可扩展的，因此可以自由的适应需要；

3. 标准组件模型管理Service 供应和Service 依赖，所以可以要求其他任何OSGi服务来创建组合服务；

4. iPOJO管理组件实例的生命周期和环境动态；

5. iPOJO提供一个强力的组合系统来创建高度动态的应用；

6. iPOJO支持注解，xml或者基于Java的API来定义组件；

可以看到iPOJO的功能远比之前的几个解决方案要强大，除了支持Declarative Services已经实现的功能外，还提供了强大的注解支持，而且实现了组合系统，这些对于开发大型的复杂应用时非常有用的。

Richard 在他的Presentation谈到iPOJO 的设计思路：

1. Make things simple / 让事情简单

2. Follow POJO philosophy / 遵循POJO的哲学

3. Employ byte code manipulation techniques / 使用字节码操纵技术

4. Be as lazy as possible / 尽可能的偷懒

目前的iPOJO还在继续发展中，最新的一个版本为1.10.0。

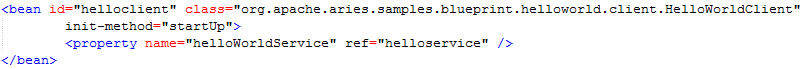
## Blueprint

Blueprint是OSGi为了解决上述问题的最新尝试，在发布的OSGi v4.2规范中新加入了 Blueprint Container 的规范内容。

提到Blueprint 就不能不提到Spring Dynamic Modules，Blueprint 可以认为是Spring Dynamic Modules版本的改进和标准化。SpringDM 1.x版本实现了Spring Dynamic Modules for OSGi，在Spring Dynamic Modules被标准化为Blueprint Container 规范后，新的SpringDM 2.x 则成为Blueprint的参考实现（目前SpringDM已转化为Eclipse virgo项目）。

Blueprint的使用实行非常类似标准的Spring IOC容器，比如同样的使用xml配置文件，只是从ioc 的applictionContext xml变成了Blueprint XML。而Blueprint XML的配置方式和Spring 有惊人的相似。

举例，最简单的bean：



基本就是照搬Spring IOC的方式，对于熟悉spring的开发人员来说无疑是个好消息，起码学习曲线平缓了。

由于是OSGi的标准规范，Blueprint 目前的支持还是不错的，除了上面说的SpringDM外，还有Geronimo Blueprint Container 和 Apache Felix Karaf 都提供了对Blueprint的支持。

从上述的发展历程上看，OSGI中的service依赖关系管理方式，经历了从简单原始到逐渐成熟强大的过程，前后经历了大概10年的时间。

# 技术框架选型分析

之所以要研究选择OSGi技术框架（OSGi高级组件框架），是因为组件框架可以简化创建OSGi应用的任务，并且添加了一些有用的功能，包括延迟启动、复杂的服务依赖管理，以及配置外部化。否则，你需要自己去做这些工作，组件框架可以避免这些工作。

1. 声明式服务（Declarative Services）是OSGi规范之一，并且是最简单的框架，它提供了对服务依赖和组件配置进行管理的功能。
2. Blueprint也是OSGi规范之一，并且其特性与声明式服务类似，但是拥有更丰富的配置模式。对于有Spring开发背景的开发人员来说，框架并不陌生。
3. iPOJO是一个开源的解决方案，它使用组件的字节码插装为构建具有动态的、以服务为基础的应用，提供了一个全面而复杂的框架。

使用其中任何一个组件框架，都可以构建出丰富、动态、以OSGi为基础的应用，另外，这些框架都可以通过OSGi服务注册的方式进行集成和协作。

以下表格对三者的特性进行了详细总结：

|  |  |  |  |
| --- | --- | --- | --- |
| **特性** | **声明式服务** | **Blueprint** | **iPOJO** |
| **依赖注入** |  |  |  |
| 回调注入 | 支持 | 但方法必须为公有 | 支持 |
| 构造函数注入 | 不支持 | 支持 | 不支持 |
| 字段注入 | 不支持 | 不支持 | 支持 |
| 赋值注入 | 支持 | 支持 | 支持 |
| 代理注入 | 不支持 | 支持 | 支持 |
| 列表注入 | 不支持 | 支持 | 支持 |
| 可空注入 | 不支持 | 不支持 | 支持 |
| **生命周期** |  |  |  |
| 回调（激活/停止） | 支持 | 支持 | 支持 |
| 工厂模式 | 支持 | 支持 | 支持 |
| 延迟初始化 | 支持 | 支持 | 支持 |
| 阻塞 | 不支持 | 支持 | 支持 |
| 字段同步 | 不支持 | 不支持 | 支持 |
| 组件生命周期控制 | 支持 | 部分支持 | 支持 |
| 服务生命周期控制 | 不支持 | 不支持 | 支持 |
| **配置** |  |  |  |
| 属性配置 | 不支持 | 支持 | 支持 |
| 字段配置 | 不支持 | 不支持 | 支持 |
| 配置管理 | 支持 | 不支持 | 支持 |
| **服务** |  |  |  |
| 自定义属性类型 | 不支持 | 支持 | 支持 |
| 延迟初始化 | 支持 | 支持 | 支持 |
| 组成服务 | 不支持 | 不支持 | 支持 |
| **描述方式** |  |  |  |
| XML | 支持 | 支持 | 支持 |
| Java注解 | 不支持 | 不支持 | 支持 |
| API | 不支持 | 不支持 | 支持 |
| **非功能** |  |  |  |
| 多提供者 | 支持 | 支持 | 不支持 |
| **支持项统计** | **10** | **14** | **23** |

# 结论

OSGi原生的组件框架（Service Listener）和高级组件框架相比，OSGi原生组件框架更适合基础OSGi服务和底层技术组件的实现，高级组件框架更适合企业应用开发。

高级组件框架有三种：声明式服务、iPOJO、Blueprint，上一节已经做了一些分析，虽然iPOJO特性上优势明显，但是考虑到Blueprint是OSGi的标准，能够得到更加广泛的开源产品的支持，并且由于配置文件源自于Spring，学习曲线较低，而iPOJO非OSGi标准，学习曲线相对较高，因此建议选择Blueprint作为应用开发的标准组件框架。

由于Apache Aries项目提供了较为全面的OSGi企业级规范的实现，包括Blueprint、JPA、JNDI等J2EE规范的支持，相对一些需要考虑向前兼容而研发的产品，Apache Aries是全新的基于OSGi规范的实现，没有历史负担（相对于Eclipse virgo、Spring、Hibernate而言），因此建议选择Apache Aries项目相关的Bundle集合**为主**提供OSGi框架实现，直接替代Spring、Hibernate这些传统的开源产品，作为企业级规范实现的首选。

# 总结

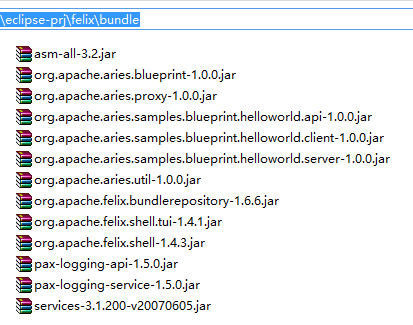
通过对OSGi技术框架选型预研，了解了OSGi相关的规范的内容、概念模型、框架实现等相关内容。在此之前虽然对OSGi有一些研究和了解，但是事隔1年多的时间，OSGi规范已经发生了很多变化，企业级规范和实现框架已经逐步成熟，具备了更多的可用性特征。通过对Eclipse Equinox、Apahce Felix、Apache Aries等项目的了解，以及实例尝试，对OSGi的应用有了更多的信心。坚持基于OSGi的统一的规范体系（核心规范是必须的，主要是选择合适的企业级规范），可以保证现有组件资产最大限度的复用，避免非标准环境下产生的迁移成本。Apache Felix中的iPOJO组件框架已经提供了较为全面完整的特性实现，Apache Aries项目已经提供了OSGi企业级规范较为完整的实现。但短时间掌握OSGi相关的众多规范和技术，还是有比较大的技术挑战的，需要投入较多的工时，减缓相关的技术风险和工期风险。

# 附录：实例研究

由于当前重点研究的OSGi技术框架主要是针对Apache Felix、Apache Aries相关的子项目：iPOJO、Blueprint，因此实例研究也主要针对这两种框架。Apache Felix、Apache Aries官方提供几个实例，这样我们可以更快的实现一个可运行的实例，当然由于环境的差异，文档的缺失，涉及的技术点比较多，实际运行也通常是困难重重，但尽快运行一个完整的实例对于直观的理解和评估该产品有很多好处，有些风险也可以提前暴漏下，便于后续的预研规划更有针对性。由于预研时间相对可能遇到的问题还是比较短的，实例研究内容比较少，部分实例未完全测试，当前文档仅基于一个Apache Aries官方完整实例作验证介绍。

1. 实例介绍

本地测试的是官方提供的Blueprint Helloworld实例，通过完整[实例下载](http://www.apache.org/dyn/closer.cgi/aries/samples-1.0.0-source-release.zip)包可以获得，该实例包括以下三个OSGi构件包：



详细代码和配置内容请参见具体代码，不再详细介绍。

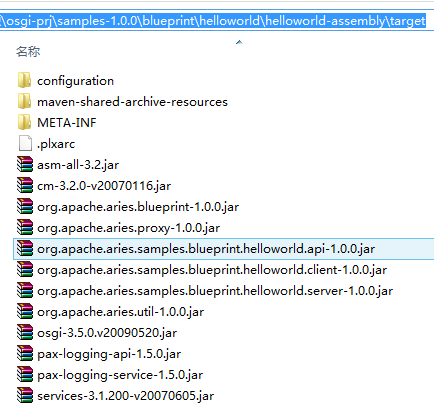
1. 部署步骤

本实例介绍通过Maven来完成构建过程，Maven的使用请参考相关资料。

首先需要安装设置Maven运行环境，然后打开命令行，进入“samples-1.0.0\blueprint\helloworld\helloworld-assembly”路径，执行Maven命令：

mvn assembly:assembly

编译完成后，可以看到当前target路径下已经生成了如图所示的OSGi构件包。



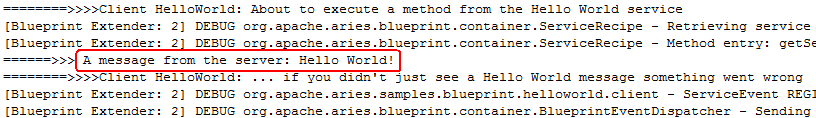
**图1：Helloworld实例bundle集合**

默认实例使用的Equinox OSGi内核，我们暂且先运行下，然后再切换成Felix内核试下。

在命令行执行：

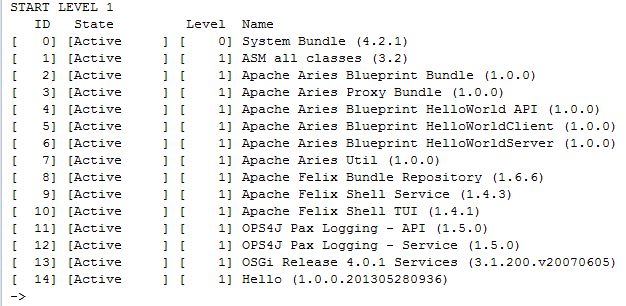
java -jar osgi-3.5.0.v20090520.jar –console

运行结果如下：



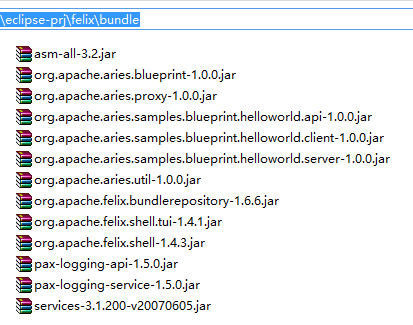
如上所示，表示实例已经正常运行。

在命令行执行“ps”命令，可以看到如下结果：



上图显示了所有当前运行的OSGi Bundle，以及对应的版本和状态。

在Felix的Bundle路径下部署如下图中的Bundle，



在Felix根路径下执行：

java –jar bin\felix.jar

执行成功后，控制台信息如图1所示。

1. 总结分析

通过上述实例主要验证下基础的Blueprint运行环境的Bundle依赖，运行简单的Helloworld实例，并验证Bundle引用，并且可以实现Equinox OSGi内核和Felix OSGi内核的替换（注意：Equinox、Felix项目不仅包含OSGi内核实现，还有很多扩展子项目），另外所用的Bundle版本都不是最新的，实际项目过程中需要基于最新的Bundle集合开发。