**OSGi Service**

# OSGi 服务模型

一个OSGi 服务就是注册到OSGi框架[服务注册表]中的一个Java对象，向服务注册表注册自身的一个或多个接口。

那么“动态”究竟指的是什么，举例来说就是：当一个 Bundle 被 OSGi 框架发现并做为一个服务启动后，该服务可以在任何时候变的可用或不可用。造成服务不可用的情况有很多，或许是 Bundle被停止或者被拆卸，或许是系统的某一部分硬件出现问题，或许是 ... 不管是什么原因，我们必须能够恰当的处理这种突发的情况，以此来维持应用程序的正常运转。值得欣慰的是，OSGi 提供了这样的技术和一些实用的类库，可以帮我们实现动态服务。OSGi 框架包含一个中央服务注册表 (Centralized Service Registry)，遵循“发布 - 查找 - 绑定”的模型，该模型是实现动态服务的基础。如下图：

* 一个提供服务的 Bundle 可以将 POJOs 发布成服务
* 一个消费者 Bundle 可以查找并绑定和消费相应的服务

# Service的概念

简单的说，服务就是“为别人所做的工作”，比如两个对象互相调用方法，那么被调用者就是在为调用者做工作。 也许有的人会觉得服务是从网络上获取的，但是经典的看法其实并没有这个限制，有时候即使是完全本地的应用，也能从面向服务的编程中得到好处。

那么如何将服务和一次普通的方法调用区别开来呢？其实一个服务可以看作是在服务的提供者和使用者之间的一个契约。使用者一般不关心其实现的细节，甚至连谁提供的都不想知道，只要满足这个契约（服务应该提供什么功能，满足什么格式）就好了。使用服务的过程也包含了发现服务和达成协议的形式，也就是说我们需要通过服务的标志性特征来找到对应的服务。

其实，Java的接口可以说提供了一种契约的提供方式。

OSGi中的服务是一个很广的概念，从编程语言的角度来说，一个服务（service）就是一个实现了一个或多个Java接口的Java对象，它通常被注册到了服务注册表中。bundle可以注册服务，查找服务，接收注册服务的状态改变信息。

An OSGi service is a java object instance, registeredinto an OSGi framework with a set of properties. Any java object can be registered as a service, but typically it implements a well-known interface。

# Service Layer

OSGi 服务层定义了Bundle 动态协作的服务发布、查找和绑定模型，服务层基于模块层和生命周期层，使得 OSGi 形成了完整的动态模型。bundle可以注册服务，查找服务，接收注册服务的状态改变信息。

# OSGi服务要点

* 协作性 — 服务层必须为bundle提供一种机制来发布、查找和绑定bundle之间的服务，而无需事先知道bundle的信息。
* 动态性 — 服务层机制必须能够处理外界或者是子结构的变化。
* 安全 — 必须能够限定对服务的访问。
* 深入性 — 对服务层的内部状态可以进行完全控制。
* 版本控制 — 提供对bundle以及对它们服务更新的控制。
* 持久性 — 在框架重启过程中提供方法来跟踪服务。

# OSGi服务相关的概念

## 服务（Service）

一个服务（service）就是一个实现了一个或多个Java接口的Java对象，它通常被注册到了服务注册表中。bundle可以注册服务，查找服务，接收注册服务的状态改变信息。

在OSGi服务平台下，bundle建立在一系列的相互协作的可用服务之上，这些服务共享一个服务注册中心。这样一个OSGi服务在语义上通过它的服务接口来定义，并实现为一个服务对象。

在服务接口中应该尽可能少的指定实现细节。OSGi规定了很多常用的服务接口，以后还会增加。

服务对象是属于bundle的，而且在bundle之内运行。bundle必须要将服务对象注册到框架的服务注册中心，这样，才可以在框架的控制下来为其他bundle提供服务。

提供服务的bundle和使用服务的bundle之间的依赖关系由框架来进行管理。例如，当停止一个bundle后，这个bundle在框架中注册的服务必须要自动的取消注册。

框架将服务映射到框架下的服务对象，而且，框架提供了一种简单而强大的查询机制，通过使用这种机制，bundle就可以请求它需要的服务。框架也提供了一种事件机制，这样，bundle就可以接收到服务注册、更改和取消注册的消息。

## 服务引用（Service Reference）

对一个服务的引用。提供对一个服务属性的访问但不是对实际的服务对象。对服务对象的访问必须要通过bundle的BundleContext来进行。

通常，注册的服务是通过一个ServiceReference对象来引用的。这样，在bundle只需要知道一个服务而不是一个服务对象的情况下，就可以避免创建bundle之间的不必要的动态服务依赖关系

可以把一个ServiceReference对象保存并发送给其他bundle，而不会带来依赖关系。当bundle需要使用服务时，就可以通过将ServiceReference对象作为方法getService的参数来获取服务，即为BundleContext. getService(ServiceReference)。详情参阅服务定位一节。

在ServiceReference对象中封装了属性和服务对象的其他元数据信息。其他bundle可以通过查找元数据信息来选择最合适的服务。 当bundle在框架的服务注册中心查找服务时，框架必须将请求服务目标的ServiceReference对象发送给请求者bundle，而不是直接发送服务对象本身。也可以通过ServiceRegistration对象来得到一个ServiceReference对象。 只有当服务对象注册之后，ServiceReference对象才是有效的。而只要ServiceReference对象存在，那么它的属性应该是可以使用的。

## 服务注册（Service Registration）

当一个服务注册之后提供凭据。服务注册允许对服务进行更新和取消注册。

## 服务权限（Service Permission）

当注册或者使用一个服务的时候，使用一个接口名称来进行权限控制。

## 服务工厂（Service Factory）

由提供服务的bundle来为每一个使用服务的bundle定制一个服务对象的便捷方法。

## 服务监听器（Service Listener）

服务事件监听器。

## 服务事件（Service Event）

个服务对象的注册、更改、取消注册的事件。

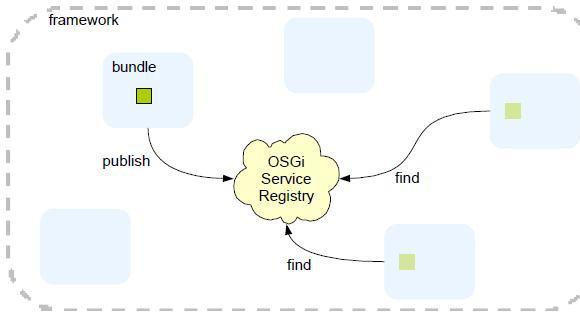
## 过滤器（Filter）

实现了一个简单但是功能强大的过滤器语言的对象。可以用于属性选择。

## 语法错误异常（Invalid Syntax Exception）

当一个过滤器表达式包含错误的时候抛出的异常。

## **OSGi Service Registry**



# Service的作用

Serivice是OSGi环境中模块间用来进行对象级交互的。

# 实战Serivce

## 服务注册表

## 服务引用

## 服务注册

# 单例服务

# 多例服务

# service factory

## 发布服务

## 使用引用

## 服务属性

属性信息保存在键值对中。键值为一个字符串对象，值则为Filter对象可以识别的类型。如果一个属性有多个值，则可以使用数组和Vector对象。

属性值应该限定为标准或原始的Java类型，以防止对bundle内部的依赖。框架不能检测到bundle之间通过服务属性产生的依赖关系。

属性键名称是不区分大小写的。ObjectClass、OBJECTCLASS和objectclass都是指同样的含义。如果调用方法ServiceReference.getPropertyKeys，那么框架必须返回它上次设置的属性值。如果Dictionary对象中存在只是大小写不同的键，那么框架必须要抛出一个异常。

服务属性提供了服务对象的信息，不应该将这些属性值用于服务的实际功能中。对注册服务的属性值更改时一种潜在的代价颇高的操作。例如，框架会提前将属性值加入到索引中以加快以后的查找速度。

Filter接口支持复杂的过滤规则；可以用于查找匹配的服务对象。因此，框架的服务注册中心中所有属性共享一个名称空间。导致的结果就是为了防止产生名称冲突，需要使用描述性名称或者是形式化定义的更短的名称来区别不同的属性，所有的属性名称前缀为service，同时在OSGi规范中保留了objectClass属性。

下表描述了预定义的标准服务属性：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 类型 | 常量 | 描述 |
| objectClass | String[] | OBJECTCLASS | objectClass属性包含了注册到框架中的服务对象所有实现的接口的集合。这个属性必须由框架自动设置。当使用BundleContext的getService方法获取服务对象时，框架必须要保证这时候服务对象可以声明为属性中指定的任何接口名称。 |
| service.description | String | SERVICE\_DESCRIPTION | service.description属性用于文档性的描述，这个属性是可选的。在框架和bundle中可以利用这个属性来对其提供的服务注册对象作一个简短的描述。这个属性的主要目的在于进行调试，这是由于没有对这个属性的本地化支持。 |
| service.id | Long | SERVICE\_ID | 每一个注册了的服务对象都由框架分配了一个惟一的service.id。将这个标志数字添加到服务对象的属性中。框架给每一个注册的服务对象分配一个惟一的标志值，这个值要比原来分配的任何值要大，也就是说是递增分配的。 |
| service.pid | String | SERVICE\_PID | service.pid属性是可选的，标记了服务对象的持久惟一标记。 |
| service.ranking | Integer | SERVICE\_RANKING | 当注册一个服务对象时，bundle也可以可选指定一个service.ranking数字作为服务对象的属性值。如果存在多个可用的服务接口，那么如果一个服务对象具有最大的SERVICE\_RANKING值或者是它的值等于最小的SERVICE\_ID值，那么就由框架返回这个服务对象。 |
| service.vendor | String | SERVICE\_VENDOR | 这是一个可选属性，描述服务对象的开发商信息。 |

### 内置属性

### 自定义属性

# OSGi　Serivce的进化

如下章节将介绍几种 OSGi 服务的发布和获取方式，这些方式各有利弊，在 OSGi 的发展过程中起到了非常重要的作用。

## 在 Activator 中注册和引用服务

此种方式是最原始的服务发布和获取方式，适用于不频繁调用服务的情况，例如只调用服务一次。该方法的缺点是过度依赖 Bundle 的启动顺序，每次访问都需要手动释放服务。

1. 在BundleActivator发布服务

|  |
| --- |
| public class DemoActivator implements BundleActivator {  private ServiceRegistration serviceRegistration = null;  //发布服务  public void start(BundleContext context) throws Exception {  ServiceRegistration = context.registerService(DemoService.class.getName(),  new DemoServiceImpl(), null);  }  //卸载服务  public void stop(BundleContext context) throws Exception {  if(serviceRegistration != null){  serviceRegistration.unregister();  }  } |

1. **使用 BundleContext 获取服务**

|  |
| --- |
| public void start(BundleContext context) throws Exception {  ServiceReference reference = context.getServiceReference(DemoService.class.getName());  if(reference != null){  DemoService demo = (DemoService)context.getService(reference);  }  } |

通过 BundleContext 来获取目标服务。首先通过 context 获取服务的引用，返回的是一个 ServiceReference 对象。然后再通过 ServiceReference 获取服务实例。拿到实例后，就可以调用服务的方法来完成所需的功能了。其中的 reference 如果为 null，在调用 context.getService(reference) 时会抛出异常。

优点：逻辑够简单，适合一次性操作

缺点：

1、OSGi的动态性无法保证获取的ref长期有效

2、每次访问都需要手动释放service

3、过于依赖bundle的启动顺序

使用场合：不频繁的调用service，如只调用一次的情况

## Listener方式

优点：只在Service变更时产生一次service获取开销，动态感知service的注册和注销。

缺点：

1、在listener起来之前的服务无法监听

2、需要维护service的获取和释放

3、监听多个 Service实例时，使用并不方便

使用场合：不推荐使用

## ServiceTracker方式

 ServiceTracker，即服务跟踪器

## 通过DS(Declarative Service)方式使用服务

Declarative Services 是一个面向服务的组件模型，它制订的目的是更方便地在 OSGi 服务平台上发布、查找、绑定服务，对服务进行动态管理，如监控服务状态以及解决服务之间的复杂的依赖关系等问题。Declarative Services 采用服务组件的延迟加载以及组件生命周期管理的方式来控制对于内存的占用以及启动的快速，很好的解决了传统的 OSGi 服务模型在开发和部署比较复杂应用时内存占用大、启动慢等问题，并且对服务组件的描述采用XML来实现，十分便于用户理解和使用。

优点：

1、通过配置发布获取服务

2、采用服务组件的延迟加载以及组件生命周期管理的方式来控制对于内存的占用以及启动的快速

缺点：与其他用spring开发项目不好集成

Declarative Service（DS）是 OSGi 核心框架的一个标准服务。它是 OSGi R4 中的一个重大改进，对 OSGi 框架是一个改善和提升。在没有 DS 之前，基于 OSGi 的系统采用的是 Module+Service 的方式，就像我们上面介绍的那样。而 DS 让我们可以在 Bundle 中定义 Component，通过配置的方式发布服务和获取服务。

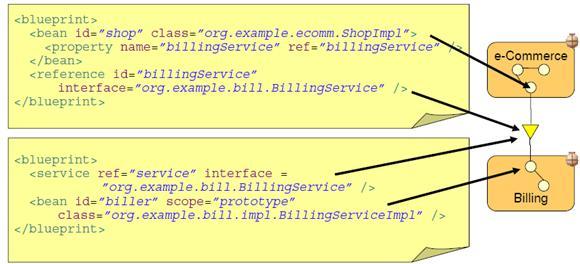
在 META-INF 的同级目录建立 OSGI-INF 目录，通过开发工具（如 RAD）建立 Plug-in Development/Component Definition 文件，取名 component.xml，然后配置发布服务，当然也可以使用图形界面。

## 使用Blueprint方式

最新的 OSGi Service Platform Release 4 V4.2 规范包括了一个名为 Blueprint Container 的规范。OSGi 企业专家组把 Spring DM 引入的依赖注入 (Dependency Injection) 技术在 OSGi 服务平台的应用方法进行了定义，为 Bundle 提供一个基于 XML 编写的配置文件，位于路径“OSGI-INF/blueprint/”下。在这个配置文件中，定义了 Bundle 中 Bean 对象的初始化方式以及它们之间的关系，同时，也声明了其需要引用的服务对象和打算发布的服务。拥有这样一个配置文件的 Bundle 可以称之为 Blueprint Bundle，它可以被 Blueprint Container 发现并进行相关的操作。Blueprint Container 规范为 OSGi 在企业级的发展提供了更大的便利。该方式也是 WAS 所推荐的完全支持的 OSGi 服务的发布和获取方式。使用 Blueprint Container 发布 OSGi 服务有以下好处：

* 以 spring 的方式管理对象 (Dependency Injection 方式 )
* 通过配置发布获取服务，发布获取方式更加灵活
* 与其他 spring 项目容易集成

我们来看一个应用 Blueprint Container 规范的例子，如下图：



在上面的例子中，e-Commerce Bundle 有一个 shop bean，它需要引用另外一个 Bundle 提供的 BillingService 服务。Blueprint Container 负责查找这个服务，并将查找到的服务注入到 shop bean 中。

在服务提供者这边，BillingServiceImpl 是一个 POJO，实现了 BillService 接口，在本例中，其所在的 Bundle 在启动后，Blueprint Container 负责将其注册为 OSGi 服务。

优点：

1、 spring的方式管理对象(Dependency Injection方式)

2、通过配置发布获取服务

3、与其他spring项目容易集成

4、发布获取方式更加灵活

缺点：

## 远程服务