

37 | 微内核架构详解
2018-07-21 李运华

更多一手资源请添加QQ/微信1182316662



37 | 微内核架构详解

李运华



- 00:06 / 09:53

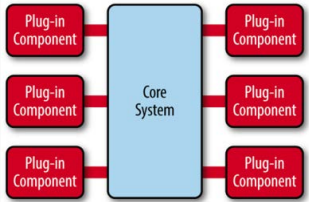
微内核架构（Microkernel Architecture），也被称为插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于产品（原文为product-based，指存在多个版本、需要下载安装才能使用，与web-based相对应）的应用。例如Eclipse这类IDE软件、UNIX这类操作系统、淘宝App这类客户端软件等，也有一些企业将自己的业务系统设计成微内核的架构，例如保险公司的保险核算逻辑系统，不同的保险产品可以将逻辑封装成插件。

今天我将为你详细介绍[常见的微内核架构及其实现](#)。

基本架构

微内核架构包含两类组件：核心系统（core system）和插件模块（plug-in modules）。核心系统负责和具体业务功能无关的通用功能，例如模块加载、模块间通信等；插件模块负责实现具体的业务逻辑，例如专栏前面经常提到的“学生信息管理”系统中的“手机号注册”功能。

微内核的基本架构示意图如下：



(<http://images2015.cnblogs.com/blog/834562/201512/834562-20151209224435824-385608713.png>)

上面这张图中核心系统Core System功能比较稳定，不会因为业务功能扩展而不断修改，插件模块可以根据业务功能的需要不断地扩展。微内核的架构本质就是将变化部分封装在插件里面，从而达到快速灵活扩展的目的，而又不影响整体系统的稳定。

设计关键点

微内核的核心系统设计的关键技术有：插件管理、插件连接和插件通信。

1. 插件管理

核心系统需要知道当前有哪些插件可用，如何加载这些插件，什么时候加载插件。常见的实现方法是插件注册表机制。

核心系统提供插件注册表（可以是配置文件，也可以是代码，还可以是数据库），插件注册表含有每个插件模块的信息，包括它的名字、位置、加载时机（启动就加载，还是按需加载）等。

2. 插件连接

插件连接指插件如何连接到核心系统。通常来说，核心系统必须制定插件和核心系统的连接规范，然后插件按照规范实现，核心系统按照规范加载即可。

常见的连接机制有OSGI（Eclipse使用）、消息模式、依赖注入（Spring使用），甚至使用分布式的协议都是可以的，比如RPC或者HTTP Web的方式。

3. 插件通信

插件通信指插件间的通信。虽然设计的时候插件间是完全解耦的，但实际业务运行过程中，必然会出现某个业务流程需要多个插件协作，这就要求两个插件间进行通信。由于插件之

更多一手资源请添加QQ/微信1182316662

间没有直接联系，通信必须通过核心系统，因此核心系统需要提供插件通信机制。这种情况和计算机类似，计算机的CPU、硬盘、内存、网卡是独立设计的配件，但计算机运行过程中，CPU必须和内存和硬盘有稳定的通信机制，才能和磁盘交互，CPU总线提供CPU和内存之间的通信功能。同样，网络操作系统也必须提供类似的通信机制，各个插件之间才能进行正常的通信。

更多一手资源请添加QQ/微信1182316662

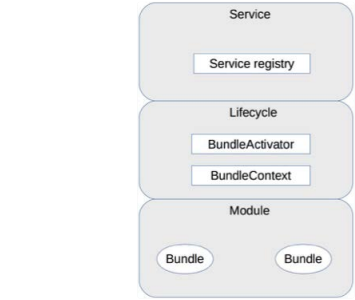
OSGi架构简析

OSGi的全称是Open Services Gateway initiative，本身其实是指OSGi Alliance。这个联盟是Sun Microsystems、IBM、爱立信等公司于1999年3月成立的开放的标准化组织，最初名为Connected Alliance。它是一个非盈利的国际组织，旨在建立一个开放的服务规范，为通过网络向设备提供服务建立开放的标准，这个标准就是OSGi specification。现在我们谈到OSGi，如果没有特别说明，一般都是指OSGi的规范。

OSGi联盟的初始目标是构建一个在广域网和局域网或设备上展开业务的基础平台，所以OSGi的最早设计也是针对嵌入式应用的，诸如机顶盒、服务网关、手机、汽车等都是其应用的主要环境。然而，无心插柳柳成荫，由于OSGi具备动态化、热插拔、高可复用性、高效性、扩展方便等优点，它被应用到了PC上的应用开发。尤其是Eclipse这个流行软件采用OSGi标准后，OSGi更是成为了首选的插件化标准。现在我们谈论OSGi，已经和嵌入式应用关联不大了，更多是将OSGi当作一个微内核的架构模式。

Eclipse从3.0版本开始，抛弃了原来自己实现的插件化框架，改用了OSGi框架。需要注意的是，OSGi是一个插件化的标准，而不是一个可运行的框架，Eclipse采用的OSGi框架称为Equinox，类似的实现还有Apache的Felix、Spring的Spring DM。

OSGi框架的逻辑架构图如下：



1. 模块层 (Module层)

模块层实现插件管理功能。OSGi中，插件被称为Bundle，每个Bundle是一个Java的JAR文件，每个Bundle里面都包含一个元数据文件MANIFEST.MF，这个文件包含了Bundle的基本信息。例如，Bundle的名称、描述、开发商、classpath，以及需要导入的包和输出的包等，OSGi核心系统会将这些信息加载到系统中用于后续使用。

一个简单的MANIFEST.MF 样例如下：

```
// MANIFEST.MF
Bundle-ManifestVersion: 2
Bundle-Name:UserRegister
Bundle-SymbolicName: com.test.userregister
Bundle-Version: 1.0
Bundle-Activator: com.test.UserRegisterActivator

Import-Package: org.log4j;version="2.0",
.....
Export-Package: com.test.userregister;version="1.0",
```

2. 生命周期层 (Lifecycle层)

生命周期层实现插件连接功能，提供了执行时模块管理、模块对底层OSGi框架的访问。生命周期层精确地定义了Bundle生命周期的操作（安装、更新、启动、停止、卸载），Bundle必须按照规范实现各个操作。例如：

```
public class UserRegisterActivator implements BundleActivator {

    public void $start(BundleContext context) {
        UserRegister.instance = new UserRegister ();
    }

    public void $stop(BundleContext context) {
        UserRegister.instance = null;
    }
}
```

3. 服务层 (Service层)

服务层实现插件通信的功能。OSGi提供了一个服务注册的功能，用于各个插件将自己能提供的服务注册到OSGi核心的服务注册中心，如果某个服务想用其他服务，则直接在服务注册中心搜索可用服务中心就可以了。

例如：

```
// 注册服务
public class UserRegisterActivator implements BundleActivator {
//在$start()中用BundleContext.registerService()注册服务
public void $start(BundleContext context) {
```

更多一手资源请添加QQ/微信1182316662

context.regiderService(UserRegider.class.getName(), new UserRegiderImpl(), null);
}
//无依赖的stop()中注册服务, 因为Bundle停止的时候会调用init函数,Bundle中已注册的服务
public void stop(BundleContext context) {}
}
// 检索服务
public class Client implements BundleActivator {
public void start(BundleContext context) {
// 1. 从服务注册表中检测问题的“服务引用”
ServiceReference ref = context.getServiceReference(UserRegider.class.getName());
// 2. 使用“服务引用”去访问服务对象实例的实例
(UserRegider) context.getService(ref).regider();
}
public void stop(BundleContext context) {}
}

注意： 这里的服务注册不是插件管理功能中的插件注册，实际上是插件间通信的机制。

规则引擎架构简析

规则引擎从结构上来看也属于微内核架构的一种具体实现，其中执行引擎可以看作是微内核，执行引擎解析配置好的业务流，执行其中的条件和规则，通过这种方式来支持业务的灵活多变。

规则引擎在计费、保险、促销等业务领域应用较多。例如电商促销，常见的促销规则有：

- 满100送50
- 3件立减50
- 3件8折
- 第3件免费
- 跨店满200减100
- 新用户立减50
-

以上仅仅列出来常见的几种，实际上完整列下来可能有几十上百种，再加上排列组合，促销方案可能有几百上千种，这样的业务如果完全靠代码来实现，开发效率远远跟不上业务的变化速度，而规则引擎却能够很灵活的应对这种需求，主要原因在于：

1.可扩展

通过引入规则引擎，业务逻辑实现与业务系统分离，可以在不改动业务系统的情况下扩展新的业务功能。

2.易理解

规则通过自然语言描述，业务人员易于理解和操作，而不像代码那样只有程序员才能理解和开发。

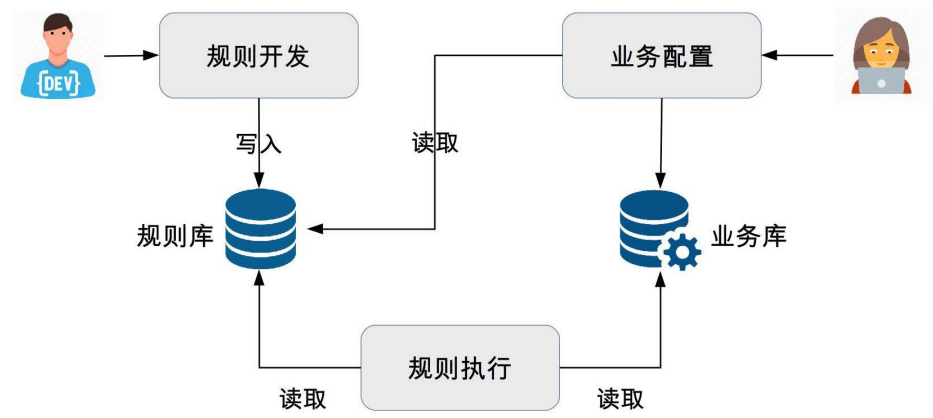
3.高效率

规则引擎系统一般提供可视化的规则定制、审批、查询及管理，方便业务人员快速配置新的业务。

规则引擎的基本架构如下：

更多一手资源请添加QQ/ 微信1182316662

更多一手资源请添加QQ/微信1182316662



我来简单介绍一下：

- 开发人员将业务功能分解提炼为多个规则，将规则保存在规则库中。
- 业务人员根据业务需要，通过将规则排列组合，配置成业务流程，保存在业务库中。
- 规则引擎执行业务流程实现业务功能。

对照微内核架构的设计关键点，我们来看看规则引擎是如何实现的。

1.插件管理

规则引擎中的规则就是微内核架构的插件，引擎就是微内核架构的内核。规则可以被引擎加载和执行。规则引擎架构中，规则一般保存在规则库中，通常使用数据库来存储。

2.插件连接

类似于程序员开发的时候需要采用Java、C++等语言，规则引擎也规定了规则开发的语言，业务人员需要基于规则语言来编写规则文件，然后由规则引擎加载执行规则文件来完成业务功能，因此，规则引擎的插件连接实现机制其实就是规则语言。

3.插件通信

规则引擎的规则之间进行通信的方式就是数据流和事件流，由于单个规则并不需要依赖其他规则，因此规则之间没有主动的通信，规则只需要输出数据或者事件，由引擎将数据或者事件传递到下一个规则。

目前最常用的规则引擎是开源的JBoss Drools，采用Java语言编写，基于Rete算法（参考https://en.wikipedia.org/wiki/Rete_algorithm）。Drools具有下面这些优点：

- 非常活跃的社区支持，以及广泛的应用。
- 快速的执行速度。
- 与Java Rule Engine API（JSR-94）兼容。
- 提供了基于Web的BRMS——Guvnor，Guvnor提供了规则管理的知识库，通过它可以实现规则的版本控制，以及规则的在线修改与编译，使得开发人员和系统管理人员可以在线管理业务规则。

虽然Drools号称简单易用，但实际上其规则语言还是和编程语言比较类似，在实际应用的时候普通业务人员面对这样的规则语言，学习成本和理解成本还是比较高的，例如下面这个样例（<https://blog.csdn.net/ouyangshixiong/article/details/46315273>）：

更多一手资源请添加QQ/微信1182316662

空档滑行

Atlas通过Bundle Framework，提供Bundle管理、加载、生命周期、安全等一些最基本的能力。

通过运行期管理层，提供功能清单，在调用时方便查找；提供版本管理，代理系统的运行环境

2018-07-21

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662

更多一手资源请添加QQ/微信1182316662