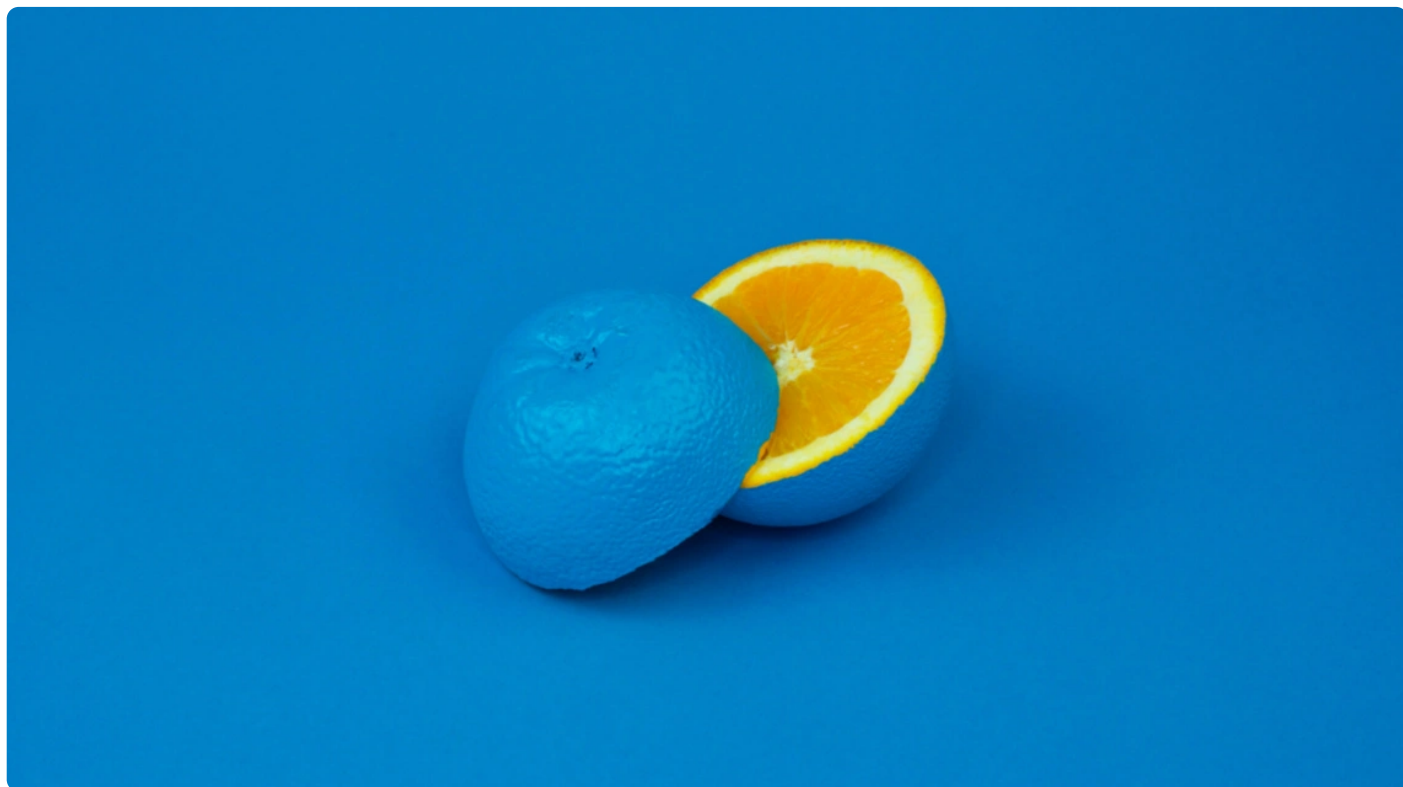


37 | 微内核架构详解

2018-07-21 李运华 来自北京

《从0开始学架构》



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，不同的保险品种可以将逻辑封装成插件。

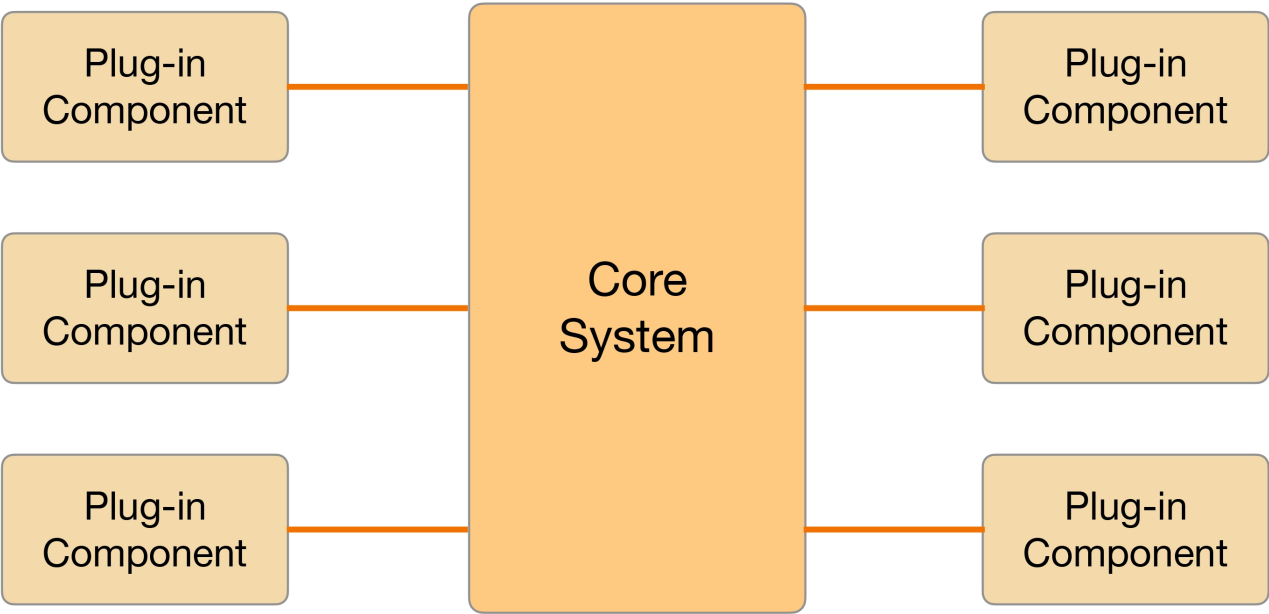
今天我将为你详细介绍[常见的微内核架构及其实现](#)。

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：



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 和内存、内存和硬盘肯定是有通信的，计算机通过主板上的总线提供了这些组件之间的通信功能。微内核的核心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。

OSGi 架构简析

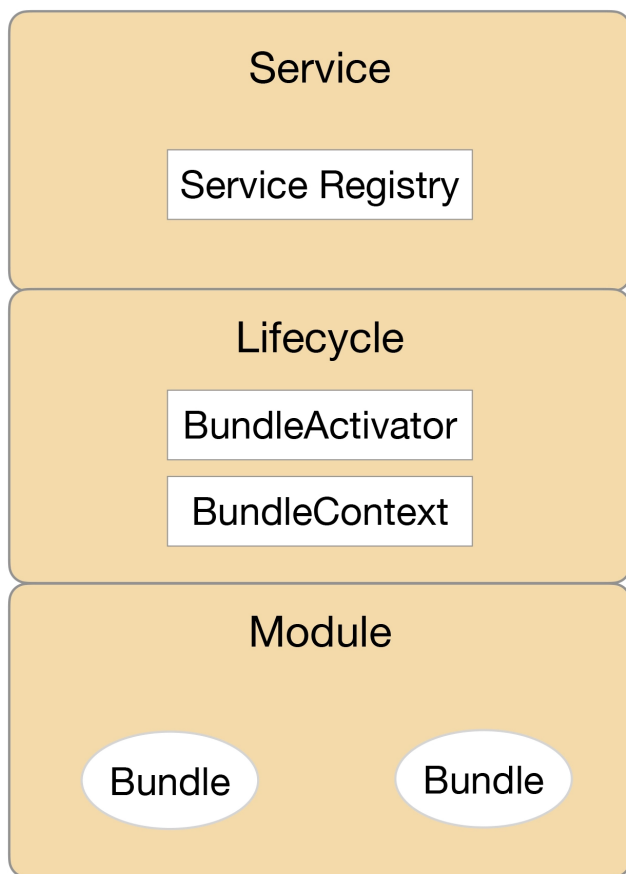
OSGi 的全称是 Open Services Gateway initiative，本身其实是指 OSGi Alliance。这个联盟是 Sun Microsystems、IBM、爱立信等公司于 1999 年 3 月成立的开放的标准化组织，最初名为 Connected Alliance。它是一个非盈利的国际组织，旨在建立一个开放的服务规范，为通过网络向设备提供服务建立开放的标准，这个标准就是 OSGi specification。现在我们谈到 OSGi，如果没有特别说明，一般都是指 OSGi 的规范。

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：

 复制代码

```
1 // MANIFEST.MF
2 Bundle-ManifestVersion: 2
3 Bundle-Name:UserRegister
4 Bundle-SymbolicName: com.test.userregister
5 Bundle-Version: 1.0
6 Bundle-Activator: com.test.UserRegisterActivator
7
8 Import-Package: org.log4j;version="2.0",
9 .....
10 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，提供了运行时模块管理、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：


 复制代码

```
1 public class UserRegisterActivator implements BundleActivator {
2
3     public void start(BundleContext context) {
4         UserRegister.instance = new UserRegister ();
5     }
6
7     public void stop(BundleContext context) {
8         UserRegister.instance = null;
9     }
10 }
```

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：

 复制代码

```
1 // 注册服务
2 public class UserRegisterActivator implements BundleActivator {
3 //在start()中用BundleContext.registerService()注册服务
4 public void start(BundleContext context) {
5 context.registerService(UserRegister.class.getName(), new UserRegisterImpl(), null
6 }
7 //无须在stop()中注销服务，因为Bundle停止时会自动注销该Bundle中已注册的服务
8 public void stop(BundleContext context) {}
9 }
10 // 检索服务
11 public class Client implements BundleActivator {
12 public void start(BundleContext context) {
13 // 1. 从服务注册表中检索间接的“服务引用”
14 ServiceReference ref = context.getServiceReference(UserRegister.class.getName());
15 // 2. 使用“服务引用”去访问服务对象的实例
16 ((UserRegister) context.getService(ref)).register();
17 }
18 public void stop(BundleContext context) {}
19 }
```

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。

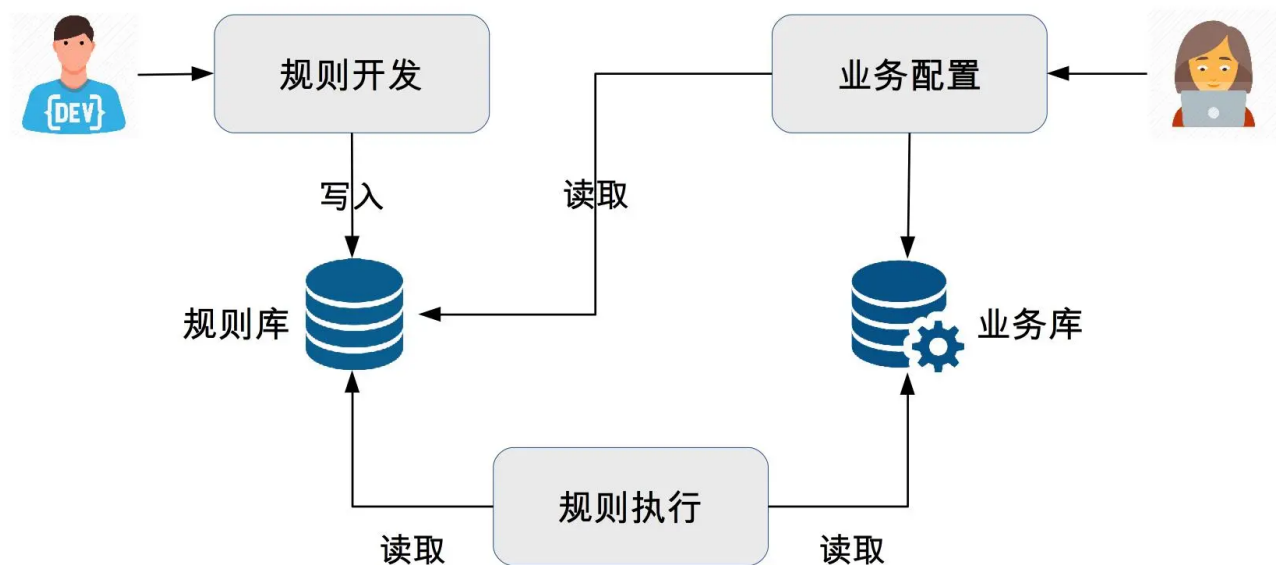
2. 易理解

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：



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具体是如何实现的。

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>）：

```

1 package com.sample
2
3 import com.sample.DroolsTest.Message;
4
5 rule "Hello World"
6     when
7         m : Message( status == Message.HELLO, myMessage : message )
8     then
9         System.out.println( myMessage );
10        m.setMessage( "Goodbye cruel world" );
11        m.setStatus( Message.GOODBYE );
12        update( m );
13    end
14
15
16 rule "GoodBye"
17     when
18         Message( status == Message.GOODBYE, myMessage : message )
19     then
20         System.out.println( myMessage );
21    end

```

因此，通常情况下需要基于 Drools 进行封装，将规则配置做成可视化的操作，例如下面电商反欺诈的一个示例（<https://cloud.tencent.com/developer/article/1031839>）：

纸尿裤黑规则1

WHEN

存在 用户维度 [uv]，它带有：

所有::

最近半年该产品线订单.平均.客单价. 选择...

大于

350

最近半年该产品线订单.平均.每次交易间隔天数. 选择...

小于

3

最近半年该产品线订单.汇总.父单或子单数. 选择...

大于

10

最近半年该产品线订单.汇总.交易天数. 选择...

小于

8

最近半年所有订单.最频繁二级类目.名称. 选择...

等于

尿裤/湿巾

最近半年该产品线订单.比率.交易订单占总订单的比重. 选择...

大于

0.9

最近半年该产品线订单.比率.促销总额占总交易额的比重. 选择...

大于

0.08

最近半年所有订单.优惠折扣减免最多的一级类目.名称. 选择...

等于

母婴

THEN

1. 监控该订单

小结

今天我为你讲了微内核架构设计的关键点以及常见的两种微内核具体实现：OSGi 和规则引擎，希望对你有所帮助。

这就是今天的全部内容，留一道思考题给你吧，结合今天所学内容，尝试分析一下手淘 Atlas 容器化框架是如何实现微内核架构的设计关键点的，分享一下你的理解。

欢迎你把答案写到留言区，和我一起讨论。相信经过深度思考的回答，也会让你对知识的理解更加深刻。（编辑乱入：精彩的留言有机会获得丰厚福利哦！）

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (50)



性能

2018-08-01

突然发现我们用了很久的框架，是微内核架构

作者回复：惊不惊喜，意不意外，醍醐灌顶 😊



👍 32



Johnny.Z

2018-07-27

规则引擎Esper 可以使用类sql编写规则，入门比较简单，不过功能应该没Drools强大

共 1 条评论 >

👍 17



Boh

2019-07-28

请教老师一个问题，我们目前使用的规则引擎就是这种架构，不过目前存在一个比较头疼的问题，就是对内开放了一个引擎调用接口，可是由于规则非常多，所以接口的请求参数和响应参数是动态的，所以接口调用传参比较头疼。目前我们的做法是将规则根据业务整合成起来，然后将每个业务的输入参数输出参数写入配置表，调用方先根据业务类型读取配置表查询输入输出参数的定义和类型，然后调用方自己生成这份数据调用接口，这样导致接入很不友好同时每次变动改动的地方非常多，而且配置表也越来越臃肿，请问老师这种情况有什么比较好的方案吗

作者回复：调用方自己管理各自的配置，不要由规则引擎来管理



👍 12



小喵喵

2018-07-22

1 规则引擎，我简单的理解为开发人员把规则提前写到数据库，这个是基础数据，然后用户选择规则，具体选择了那些规则可以通过外键关联。

2 那么多规则，就比如促销的例子，用策略模式实现是否更妥当？

作者回复: 纯代码实现可以用策略模式，但每次新来一种促销都要写代码，虽然代码写的少，但还是要测试部署



👍 13



xiao皮孩。。

2019-04-04

作者对规则引擎情有独钟啊

作者回复: 业务需要，不是我情有独钟😄



👍 11



醉爱星巴克

2018-07-22

想问问流程引擎算不算规则引擎的一种呢？

作者回复: 流程引擎应该是粗粒度的规则引擎

共 2 条评论 >

👍 11



飘宝

2019-04-23

李老师，我已经找到了基于产品这种说法的基本出处了，是在Mark Richards的软件架构模式中描述的。其中也对比了几种常见的架构，写的还是很实用的。

我也买了您的从零开始学架构的书，是一本非常适合反复翻阅的书。

作者回复: 谢谢，加油😊



👍 6



无聊夫斯基

2018-09-14

微内核我还真没接触过，又get到灵活的方案

作者回复: 银行，保险，电商用的比较多

共 2 条评论 >



6



奋斗心

2018-09-28

那能不能说规则引擎用了策略模式

作者回复: 你可以理解为道理是相通的，但不要这么等同，规则引擎是面向功能的一个架构设计模式，策略模式是一个面向对象的类设计模式



4



Geek_b6e6bc

2021-05-15

你好 老师。读了这节感觉收益很多，尤其是加深对spring的认识，有点体会，想到了其他的mysql是不是也是个微内核架构 server层是公用核心部分，存储引擎层属于插件，可以替换，还有dubbo框架也算是个微内核架构

作者回复: MySQL不能算微内核，MySQL只是存储引擎做成了插件，微内核的内核一定要“微”，只负责插件管理和插件通信等基础功能，而MySQL的server层是核心功能，包括SQL引擎、查询优化、binlog等



3



汉斯·冯·拉特

2018-07-25

老师，能不能写个基于微内核的代码例子在github上，之前也听过微内核，但是到目前为止，接触到的都是分层结构的架构

作者回复: 你可以对照Drools的文档写个促销的方案



3



Snway

2018-07-23

采用过esper实现复杂事件流处理引擎，看完此文，对原有架构设计有更深层次理解，感谢华仔

作者回复: 我去学习esper 😊



👍 3



丁丁历险记

2020-05-14

说个简单场景，指定时间内，累计满5000



👍 2



刘工的一号马由

2019-07-15

微内核架构和微服务架构区别就在于粒度吗？

作者回复: 形式也不同，微内核是一个应用，微服务是多个应用



👍 2



噗_/_ 噗

2018-08-21

REngine，回答楼上，.net规则引擎



👍 2



空档滑行

2018-07-21

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



👍 2



张昌

2020-06-30

这篇文章分析中有谈到老师的问题

https://mp.weixin.qq.com/s?__biz=MzAxNDEwNjk5OQ==&mid=2650400348&idx=1&sn=99bc1bce932c5b9000d5b54afa2de70e

bundle framework实现插件管理

runtime 中清单实现插件发现，代理插件通信



1



柳树

2020-04-28

Dubbo也是一个很好的例子。



1



jason

2019-11-24

尝试了一下用规则引擎实现各种活动的业务逻辑，感觉很复杂，比如首先第一步你需要通过运营配置的原子规则构造规则模板，这一步少不了代码中写ifelse或者模板中写when then ,第2步需要把业务逻辑封装成各种规则规则，第3步组装规则为新业务，感觉还不如策略模式好维护和效率高，唯一的优点貌似只是动态化，并且这种规则运营会不会使用

作者回复: 可能是你们的规则抽象太细，基本等同于代码块了，所以觉得比较难用，不过确实规则引擎里面各种if和when也是比较复杂的

共 2 条评论 >



1



loveluckystar

2019-07-01

之前一直觉得插件间通信是个很高深莫测的东西，后来真正实操一把发现，直接通过统一的上下文透传，这种数据流不也是一种插件通信方式🐼 真的要善于总结思考才行



1