ZStack 技术白皮书精选

ZStack--通用插件系统

扫一扫二维码，获取更多技术干货吧

版权声明

摘要

大道至简·极速部署，ZStack 致力于产品化私有云和混合云。

ZStack 是新一代创新开源的云计算 IaaS 软件，由英特尔、微软、CloudStack 等世界上最早一批虚拟化工程师创建，拥有 KVM、Xen、Hyper-V 等成熟的技术背景。

ZStack 创新提出了云计算 4S 理念，即 Simple （简单）、Strong （健壮）、Smart （智能）、Scalable （弹性），通过全异步架构，无状态服务架构，无锁架构等核心技术，完美解决云计算执行效率低，系统不稳定，不能支撑高并发等问题，实现 HA 和轻量化管理。

ZStack 发起并维护着国内最大的自主开源 IaaS 社区——zstack.io，吸引了 6000 多名社区用户，对外公开的 API 超过 1000 个。基于这 1000 多个 API，用户可以自由组装出自己的私有云、混合云，甚至利用 ZStack 搭建公有云对外提供服务。

ZStack 拥有充足的知识产权储备，积极申报多项软著和专利，参与业内标准、白皮书的撰写，入选云计算行业方案目录，还通过了工信部云服务能力认证和信通院可信云认证。ZStack 面向企业用户提供基于 IaaS 的私有云和混合云，是业内唯一一家实现产品化，并领先业内首家推出同时打通数据面和控制面无缝混合云的云服务商。选择 ZStack，用户可以官网直接下载、1 台 PC 也可上云、30 分钟完成从裸机的安装部署。

目前已有 1000 多家企业用户选择了 ZStack 云平台。

ZSTACK--通用插件系统

当前 IaaS 软件更像云控制器软件，要成为一个完整的云解决方案还缺少很多特性（features）。作为一个正在发展中的技术，预测一个完整的解决方案的必备的所有特性是非常困难的，所以一个 IaaS 软件不可能在一开始就完成它所有的特性。基于以上事实，一个 IaaS 软件的架构必须有能力，在添加新特性的同时保持核心结构稳定。ZStack 的通用插件系统，使得特性可以像插件一样实现（在线程内或在线程外），这样不只能使 ZStack 的功能得到了拓展，也可以注入业务逻辑内部去改变默认的行为。

动机

eBay 管理 OpenStack 私有云的首席工程师，Subbu Allamaraju 曾说过：

然而，OpenStack 是一个云控制器软件。尽管社区为 OpenStack 的组建做出了巨大贡献，但是安装好的一个 OpenStack 实例并不能算一个云。作为一个操纵者你必须处理许许多多的用户不一定知道的附加的操作。这些包括基础的员工培训、初始化、维护、配置管理、补丁、打包、升级、高可用性、监控、度量、用户支持、容量预测和管理、计费或退款、资源回收、安全、防火墙、DNS、与其他内部的基础设施和工具的集成，等等，等等。这些活动将花费大量的时间和精力。OpenStack 给出了一些创建云必备的成分，但并没有把云打包好。

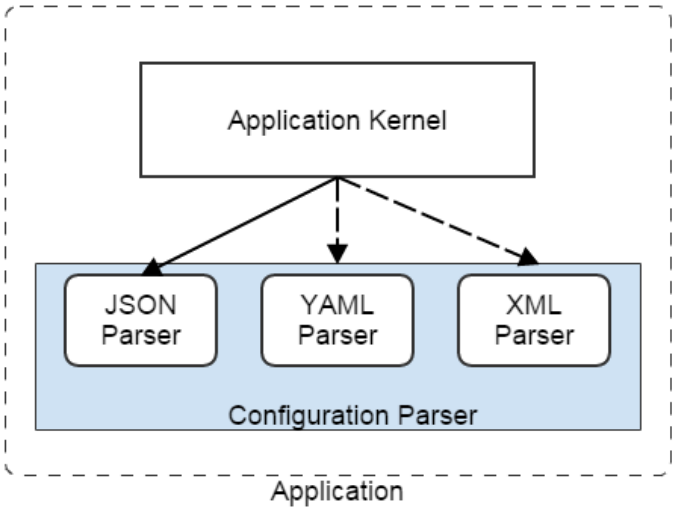
这表达当前 IaaS 软件的处境，除了一些构建的非常好的 IaaS（像 AWS），大多数 IaaS（包括我们的 ZStack）仍然不是一个完整的云解决方案。由于像 Amazon 这样的已经探索多年的先驱，公有云已经有一个更成熟的模型，对于公共云解决方案应该是什么样子而言。由于仍然处在开发阶段，私有云目前还没有经过验证的完整的解决方案。不像专用的公有云软件，可以专门为制造商的基础设施和服务定制；开源的 IaaS 软件必须同时考虑公有云和私有云的需求，使得创建一个完整的解决方案变得更加困难。因为我们没有办法预测一个完整的解决方案应该是什么样，我们唯一的办法是提供一个插件式的架构，它能在添加插件的同时，不影响核心业务稳定性。

问题

许多软件声称自己是插件式的，但是很多并不是真的插件式的，或至少不是完全插件式的。在解释原因之前，让我们看两种主要的插件式架构的形式。虽然有很多文章讨论过这个话题，以我们的经验来看，我们把所有插件归纳成两种结构，可以被准确的描述为 GoF design patterns 一书中的策略模式和观察者模式。

源自策略者模式的插件

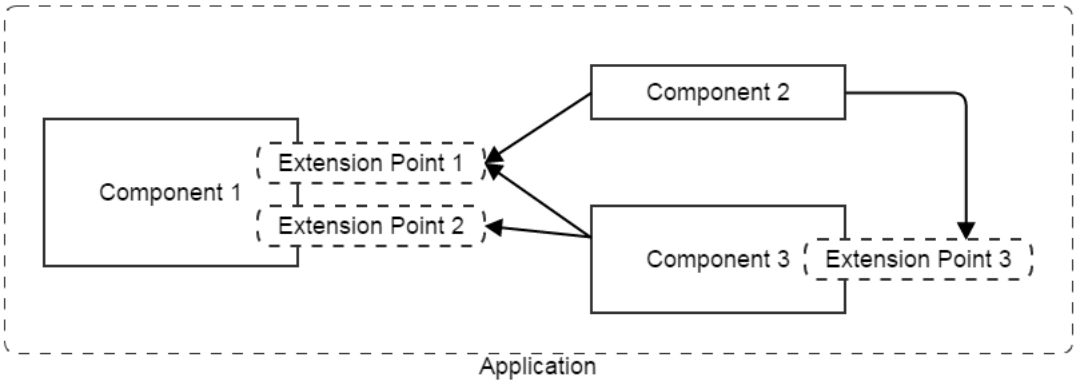
这种形式的插件通常是通过提供不同的实现，拓展软件特定的功能；或者通过添加插件 APIs 去添加新的功能。我们熟悉的很多软件都是通过这种模式搭建的，比如，操作系统的驱动，网页浏览器的插件。这种插件组成的工作方式是，**允许应用程序通过定义良好的协议去访问插件**。



源自观察者模式的插件

这种形式的插件通常注入应用程序的业务逻辑，针对特定的事件。一旦一个事件发生，挂在上面的插件将被调用，以执行一段甚至可能改变执行流的代码，比如，当事件满足某些条件，抛出异常去停止执行流。基于这种模式的插件通常对最终用户是透明的、纯内部实现

的，例如，一个监听器监听数据库插入事件。这种插件的工作方式是，**允许插件通过定义良好的扩展点去访问应用程序**。



大多数软件声称它们是插件式的，要么实现了这些组成方式中的一种，要么有一部分代码实现这些组成方式。为了变得完全插件化，软件必须设想到这么一个想法，即所有的业务逻辑都使用这两种方式实现。这意味着整个软件是由大量的小插件组成的，就像乐高玩具一样。

插件系统

一个重要的设计原则贯穿了所有 ZStack 的组件：**每一个组件都应该被这么设计，信息最少、自包含、无关其他组件。**比如，为了创建一个虚拟机，分配磁盘、提供 DHCP、建立 SNAT 都是非常必要的步骤，管理创建 VM 的组件应该非常清楚。但是它真的需要知道这么多吗？为什么这个组件不能简化为，分配 VM 的 CPU/内存，然后给主机发送启动请求，让其他组件，像存储、网络来关心它们自己的事情。你可能已经猜到了这个答案：不，在 ZStack 中，组件并不需要知道那么多，没错！可以是那么简单。我们充分意识到这么一个事实，**你的组件知道的信息越多，你的应用程序耦合越紧密，最终你得到一个复杂的难以修改的软件。**所以我们提供以下插件形式来保证我们的架构是松耦合的，并且使我们容易添加新特性，最终形成一个完整的云解决方案。

1. 策略模式插件

通常 IaaS 软件中的插件是整合不同物理资源的驱动。例如，NFS 主存储，ISCSI 主存储，

基于 VLAN 的 L2 网络，基于 Open vSwitch 的 L2 网络；这些插件都是我们刚刚提到的策略模式的形式。ZStack 已经将云资源抽象成：虚拟机管理器、主存储、备份存储、L2 网络、L3 网络等等。每个资源都有一个相关的驱动程序，作为一个单独的插件。要添加一个新的驱动程序，开发人员只需要实现三个组件：一个类型，一个工厂，和一个具体的资源实现，这些全部都被封装在单一的插件中，通常被构建成一个 jar 文件。引用 Open vSwitch 举一个例子，让我们假定我们将创建一个使用 Open vSwitch 作为后台的新 L2 网络，然后开发者需要：

1.1 定义一个 Open vSwitch 类型的 L2 网络，它将自动注册到 ZStack L2 网络类型系统中。

```
public static L2NetworkType type = new L2NetworkType("Openvswitch");



/* once the type is declared as above, there will be a new L2 network type called

'Openvswitch' that can be retrieved by API */（一旦类型被声明，一个新的叫做“Openvswitch”的 L2 网络

类型可以被 API 检索）
```

1.2 创建一个 L2 网络工厂，负责将一个具体的实现返回给 L2 网络服务。

```
public class OpenvswitchL2NetworkFactory implements L2NetworkFactory {

    @Override

    public L2NetworkType getType() {

        /* return type defined in 1.1 */

        return type;

    }



    @Override

    public L2NetworkInventory createL2Network(L2NetworkVO vo, APICreateL2NetworkMsg msg) {

        /*
```

```
        * new resource will normally have own creational API APICreateOpenvswitchL2NetworkMsg that
        * usually inherits APICreateL2NetworkMsg, and own database object OpenvswitchL2NetworkVO
that
        * usually inherits L2NetworkVO, and a java bean OpenvswitchL2NetworkInventory that usually
inherits
        * L2NetworkInventory representing all properties of Openvswitch L2 network.
        */（新的资源将通常有一个创建的API APICreateOpenvswitchL2NetworkMsg，通常继承自
APICreateL2NetworkMsg，还将有自己的数据库对象，OpenvswitchL2NetworkVO，通常继承自L2NetworkVO，和一个
java bean OpenvswitchL2NetworkInventory，通常在L2NetworkInventory中表示Openvswitch L2网络的所有属性）

        APICreateOpenvswitchL2NetworkMsg cmsg =
(APICreateOpenvswitchL2NetworkMsg)APICreateL2NetworkMsg;
        OpenvswitchL2NetworkVO cvo = new OpenvswitchL2NetworkVO(vo);
        evaluate_OpenvswitchL2NetworkVO_with_parameters_in_API(cvo, cmsg);
        save_to_database(cvo);
        return OpenvswitchL2NetworkInventory.valueOf(cvo);
    }

    @Override
    public L2Network getL2Network(L2NetworkVO vo) {
        /* return the concrete implementation defined in 1.3 */
        return new OpenvswitchL2Network(vo);
    }
```

```
}
```

1.3 创建一个具体的 Open vSwitch L2 网络实现，跟后台 Open vSwitch 控制器进行交互。

```
public class OpenvswitchL2Network extends L2NoVlanNetwork {

    public OpenvswitchL2Network(L2NetworkVO self) {

        super(self);

    }



    @Override

    public void handleMessage(Message msg) {

        /* handle Openvswitch L2 network specific messages(both API and non API) and delegate

    * others to the base class L2NoVlanNetwork; so the implementation can focus on own business

    * logic and let the base class handle things like attaching cluster, detaching cluster;

    * of course, the implementation can override any message handler if it wants, for example,

    * override L2NetworkDeletionMsg to do some cleanup work before being deleted.

（处理和Openvswitch L2 网络相关的特定消息（API 消息或不是 API 的消息），并把其他消息交给 L2NoVlanNetwork 的基础类处理；所以它的实现可以关注它自身的业务逻辑，并让基础类处理一些如绑定集群，解绑集群，的事情。当然，实现方法也可以覆盖任何消息处理器，例如，覆盖 L2NetworkDeletionMsg 在删除前做一些清理工作。）

    */

        if (msg instanceof OpenvswitchL2NetworkSpecificMsg1) {

            handle((OpenvswitchL2NetworkSpecificMsg1)msg);

        } else if (msg instanceof OpenvswitchL2NetworkSpecificMsg2) {

            handle((OpenvswitchL2NetworkSpecificMsg2)msg);
```

```
            } else {
                super.handleMessage(msg);
            }
        }
}
```

让三个组件一起放到一个 Maven 模块中，添加一些必须的 Spring 配置文件，并编译为 JAR 文件，你就在 ZStack 中创建了一个新的 L2 网络类型。所有的 Zstack 资源驱动程序都是通过以上步骤实现的（类型，工厂，具体实现）。一旦你已经学会了怎么为一个资源创建驱动程序，你就学会了怎么为所有的资源这么做。正如我们在“ZStack--进程内的微服务架构”中提到的一样，驱动可以有自己的 API 和配置方法。

2. 观察者模式插件

策略模式的插件（驱动）允许你扩展现有的 ZStack 的功能；然而，为了使架构松耦合，插件必须能注入应用程序的业务逻辑，甚至是其他插件的业务逻辑；观察模式插件的关键是*拓展点*，拓展点允许一段插件的代码在一个代码流运行的时候被调用。目前 Zstack 定义了大约 100 个扩展点，暴露了大量让插件去接收事件或改变代码流行为的场景。创建一个新的扩展点就是定义一个 java 接口，组件可以很容易地创建扩展点，以允许其他组件注入自己的业务逻辑。为了了解它是如何工作的，让我们继续我们的 Open vSwitch 的例子；假设 Open vSwitch L2 网络需要钩入创建 VM 的过程，以在 VM 创建之前准备好 GRE 隧道，该插件实现如下：

PreVmInstantiateResourceExtensionPoint:

```
public class OpenvswitchL2NetworkCreateGRETunnel implements
PreVmInstantiateResourceExtensionPoint {

    @Override
```

```
        public void preBeforeInstantiateVmResource(VmInstanceSpec spec) throws
VmInstantiateResourceException {
            /*
      * you can do some check here; if any condition makes you think the VM should not be
created/started,
      * you can throw VmInstantiateResourceException to stop it
      */
        }

        @Override
        public void preInstantiateVmResource(VmInstanceSpec spec, Completion completion) {
            /* create the GRE tunnel, you can get all necessary information about the VM from
VmInstanceSpec */
            completion.success();
        }

        @Override
        public void preReleaseVmResource(VmInstanceSpec spec, Completion completion) {
            /*
      *in case VM fails to create/start for some reason, for cleanup, you can delete the prior
created GRE tunnel here
      */
            completion.success();
```

```
        }

}
```

当 ZStack 连接到 KVM 主机，Open vSwitch L2 网络想要在主机上检查并启动 Open vSwitch 的守护进程，那么它实现 KVMHostConnectExtensionPoint：

```
      public class OpenvswitchL2NetworkKVMHostConnectedExtension implements
KVMHostConnectExtensionPoint {

         @Override

         public void kvmHostConnected(KVMHostConnectedContext context) throws
KVMHostConnectException {

            /*

        * you can use various methods like SSH login, HTTP call to KVM agent to check the
Openvswitch daemon 你可以使用很多方式（如SSH 登录，KVM 代理的HTTP 调用）通过使用在
KVMHostConnectedContext 上的信息，去检查在主机上的Openvswitch 的守护进程。如果任意状态让你认为主机不
能提供Openvswitch L2 网络方法，你可以抛出KVMHostConnectExtensionPoint 去阻止主机连接。

        * on the host, using information in KVMHostConnectedContext. If any condition makes
you think the

        * host cannot provide Openvswitch L2 network function, you can throw
KVMHostConnectExtensionPoint to

        * stop the host from being connected.

        */

         }

}
```

最后，你需要宣传你有两个组件实现了这些扩展点，ZStack 的插件系统将确保所有者在

一个适当的时间调用你的组件。该通知是在插件的 Spring 配置文件中完成的：

```
      <?xml version="1.0" encoding="UTF-8"?>

     <beans xmlns="http://www.springframework.org/schema/beans"

          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"

          xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:zstack="http://zstack.org/schema/zstack"

          xsi:schemaLocation="http://www.springframework.org/schema/beans

    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

    http://www.springframework.org/schema/aop

    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

    http://www.springframework.org/schema/tx

    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

    http://zstack.org/schema/zstack

    http://zstack.org/schema/zstack/plugin.xsd"

          default-init-method="init" default-destroy-method="destroy">



          <bean id="OpenvswitchL2NetworkCreateGRETunnel"
class="org.zstack.network.l2.ovs.OpenvswitchL2NetworkCreateGRETunnel">

              <zstack:plugin>

                  <zstack:extension
interface="org.zstack.header.vm.PreVmInstantiateResourceExtensionPoint" />

              </zstack:plugin>
```

```
        </bean>


        <bean id="OpenvswitchL2NetworkKVMHostConnectedExtension"
              class="org.zstack.network.l2.ovs.OpenvswitchL2NetworkKVMHostConnectedExtension">
            <zstack:plugin>
                <zstack:extension interface="org.zstack.kvm.KVMHostConnectExtensionPoint" />
            </zstack:plugin>
        </bean>


    </beans>
```

这就是你所需要做的一切。创建一个新类型的 L2 网络,却不需要更改其他任意一个 ZStack 组件的甚至一行代码。这是 ZStack 保持其核心业务流程稳定的基础。

不要 OSGI：*熟悉 Eclipse 和 OSGI 的人可能已经注意到，我们的插件系统和 eclipse、OSGI 的非常相似。可能有人会问，为什么我们不直接使用 OSGI，它可是为 Java 应用程序创建插件系统而专门设计的。实际上，我们花费了相当多的时间尝试 OSGI；然而，我们感觉它是用力过猛。我们不喜欢在我们的应用程序有另一个容器，不喜欢单独的类装载器，不喜欢它创建插件的复杂性。看起来 OSGI 正付出大量努力使插件相互隔离，但 ZStack 想让插件扁平化。我们已经注意到，许多项目在代码中引入了不必要的限制，以使整体架构明显是分层的、隔离的，但由于设计糟糕的接口，插件必须写很多丑陋的代码来克服这些限制，反而打乱了真正的架构。ZStack 将所有的插件作为自己核心的一部分来考虑，针对核心业务流程拥有一样的特权。我们不是构建一个类似浏览器的消费级应用程序，用户可能会错误地安装恶意插件；我们是在构建一个企业级软件，每一个角落都需要经过严格的测试。一个扁平的插件系统使我们的代码变得简单和健壮。*

3. 进程外的服务（插件）

除了以上两种方式外，开发人员确实有第三种方式扩展 ZStack--进程外服务。虽然 ZStack 把所有的编排服务包装成一个单一的进程，独立于业务流程服务的功能可以被实现为独立的服务，这些服务运行在不同的进程甚至不同的机器上。ZStack Web UI，一个通过 RabbitMQ 和 ZStack 编排服务进行交互的 Python 应用程序，是一个很好的例子。ZStack 有一个定义良好的消息规范，进程外的服务可以用任何语言编写，只要它们能通过 RabbitMQ 进行交互。ZStack 也有称为 canonical event 的机制，用于暴露一些内部事件给总线，比如 VM 创建，VM 停止，磁盘创建。诸如计费系统的软件完全可以通过监听这些事件，建立一个进程外的服务。如果一个服务想要在进程外，但仍需要访问一些还没有暴露的核心业务流程的数据结构，或需要访问数据库，它可以使用一种混合的方式，即在管理节点上的一块小插件负责采集数据并将它们发送给消息代理，在进程外的服务接受这些数据并完成自己的事情。

总结

在这篇文章中，我们展示了 ZStack 的插件架构。虽然 ZStack 并没有成为一个完整的云解决方案，但是它提供了一个架构，可以将任何未来所需要的特性构建成插件（进程内或进程外），在保持核心业务流程稳定的同时，使得快速发展成为一个成熟的、完整的云解决方案变得可能。