ZStack 技术白皮书精选

架构篇：中册

微服务、通用插件系统、工作流引擎

扫一扫二维码，获取更多技术干货吧

版权声明

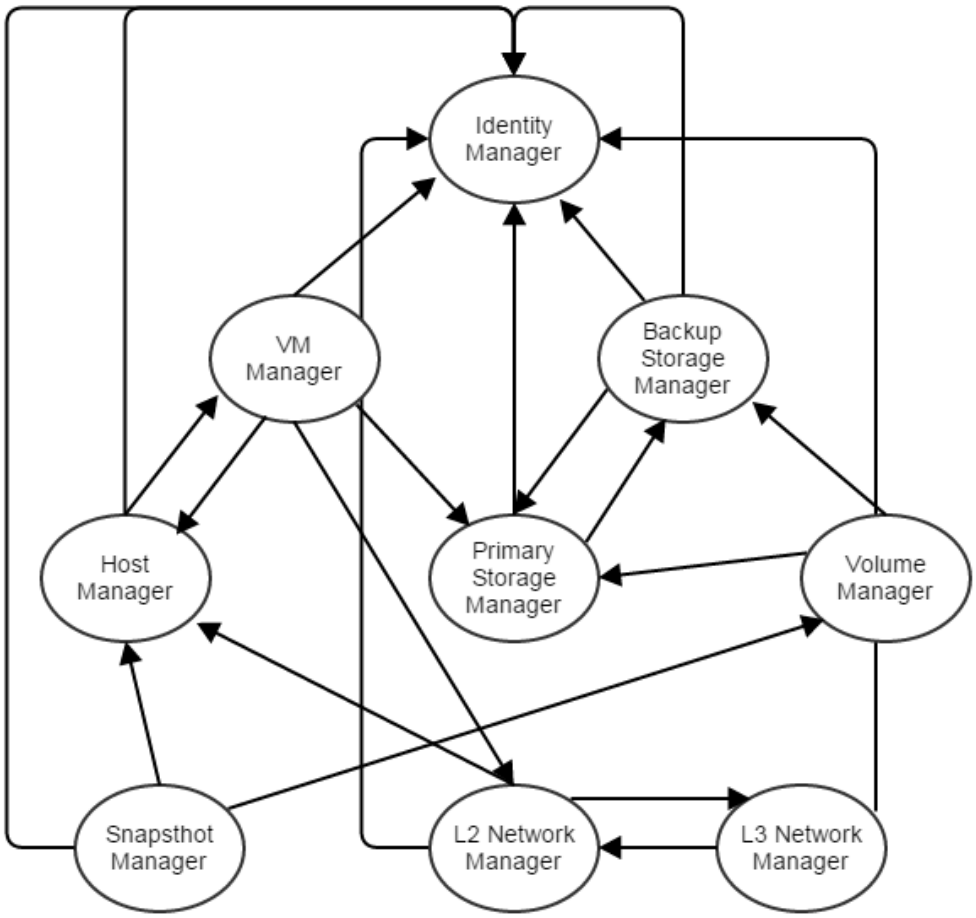
目录

ZSTACK--进程内的微服务架构

为了应对诸如惊人的操作开销、重复的努力、可测试性等微服务通常面临的挑战，以及获得诸如代码解耦，易于横向扩展等微服务带来的好处，ZStack 将所有服务包含在单个进程中，称为管理节点，构建一个进程内的微服务架构。

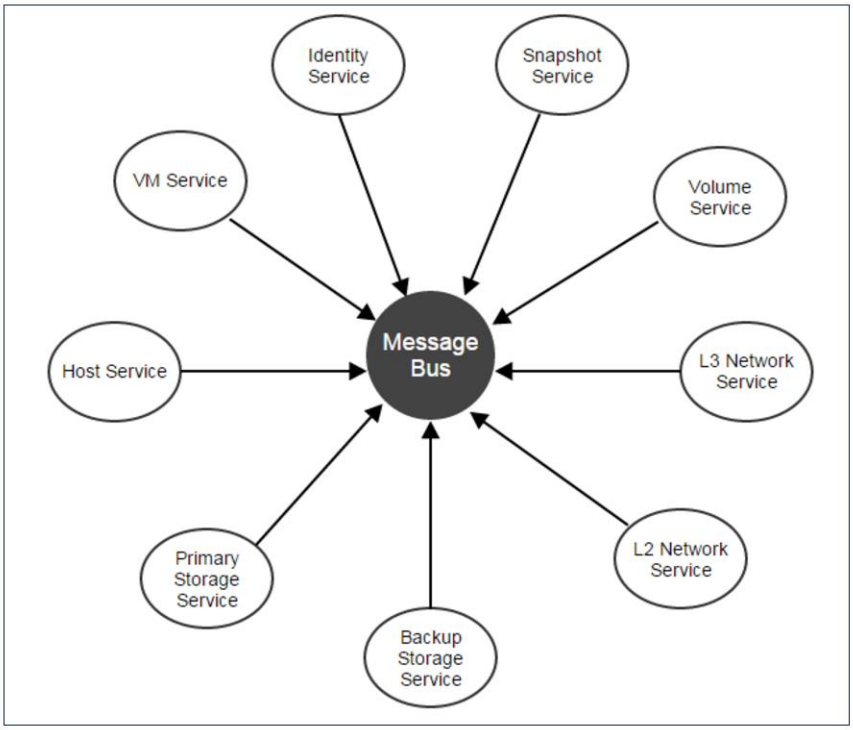
动机

构建一个 IaaS 软件是很难的，这是一个已经从市场上现存的 IaaS 软件获得的教训。作为一个集成软件，IaaS 软件通常需要去管理复杂的各种各样的子系统（如：虚拟机管理器 hypervisor，存储，网络，身份验证等）并且需要组织协调多个子系统间的交互。例如，创建虚拟机操作将涉及到虚拟机管理模块，存储模块，网络模块的合作。由于大多数 IaaS 软件通常对架构考虑不够全面就急于开始解决一个具体问题，它们的实现通常会演变成：



The organic growth of monolithic IaaS software

随着一个软件的不断成长，这个铁板一块的架构（monolithic architecture）将最终变为一团乱麻，以至于没有人可以修改这个系统的代码，除非把整个系统从头构建。这种铁板一块的编程问题是微服务可以介入的完美场合。通过划分整个系统的功能为一个个小的、专一的、独立的服务，并定义服务之间交互的规则，微服务可以帮助转换一个复杂笨重的软件，从紧耦合的、网状拓扑架构，变成一个松耦合的、星状拓扑的架构。



因为服务在微服务中是编译独立的，添加或者删除服务将不会影响整个系统的架构（当然，移除某些服务会导致功能的缺失）。

微服务远比我们已经讨论的内容更多：*微服务的确有很多引入注目的优点，尤其是在一个的开发运维 流程（DevOps process）中，当涉及到一个大机构的很多团队时。我们不打算讨论微服务的所有支持和反对意见，我们确定你可以在网上找到大量的相关文章，我们主要介绍一些我们认为对 IaaS 软件影响深远的特性。*

问题

虽然微服务可以解耦合架构，但这是有代价的。阅读 Microservices - Not A Free Lunch!和 Failing at Microservices 会对这句话有更深的理解。在这里，我们重点强调一些我们认为对 IaaS 软件影响重大的事情。

1. 难以定义服务的边界和重复做功

创建 Microservices 架构的挑战之一是决定应该把哪一部分的代码定义为服务，一些是非常明显的，比如说，处理主机部分的逻辑代码可以被定义为一个服务。然而，管理数据库交互的代码非常难以决定应不应该被定义为服务。数据库服务可以使得整个架构更加清晰明了，但是这样会导致严重的性能下降。通常，类似于这样的代码可以被定义为库，库可以被各个服务调

用。鉴于所有服务一般在互相隔离的目录下开发和维护，创建一个给不同的单一的软件提供接口的虚拟的库，要求开发者必须具有良好的和各个不同组的开发者沟通协调的能力。综上，服务很容易重复造轮子和导致不必要的重复做功。

2. 软件难以部署、升级和维护

服务，尤其是那些分散在不同进程和机器上的，是难以部署和升级的。用户通常必须去花费几天甚至几周去部署一个完整的可运行的系统，并害怕升级一个已经构建好的稳定的系统。尽管一些类似 puppet 的配置管理软件一定程度上缓解了这个问题，用户依旧需要克服陡峭的学习曲线去掌握这些配置工具，仅仅是为了部署或者升级一个软件。管理一个云是非常困难的，努力不应该被浪费在管理这些原本应该使生活更轻松的软件上。

服务的数量确实很重要：*IaaS 软件通常有许许多多的服务。拿著名的 openstack 举个例子，为了完成一个基础的安装你将需要：Nova, Cinder, Neutron, Horizon, Keystone, Glance。除了 nova 是在每台主机都需要部署的，如果你想要 4 个实例（instances），并且每个服务运行在不同机器上，你需要去操纵 20 台服务器。虽然这种人造的案例将不太可能真实地发生，它依旧揭示了管理相互隔离的服务的挑战。*

3. 零散的配置

运行在不同服务器上的服务，分别维护着它们散乱在系统各个角落的配置副本。在系统范围更新配置的操作通常由临时特定的脚本完成，这会导致由不一致的配置产生的令人费解的失败。

4. 额外的监控努力

为了跟踪系统的健康状况，用户必须付出额外的努力去监控每一个服务实例。这些监控软件，要么由第三方工具搭建，要么服务自身维护，仍然受到和微服务面临的问题所类似的问题的困扰，因为它们仍然是以分布式的方式工作的软件。

5. 插件杀手

插件这个词在微服务的世界中很少被听到，因为每个服务都是运行在不同进程中一个很小的功能单元（function unit）；传统的插件模式（参考 The Versatile Plugin System）目标是把不同的功能单元相互挂在一起，这在微服务看来是不可能的，甚至是反设计模式的。然而，对于一些很自然的，要在功能单元间强加紧密依赖的业务逻辑，微服务可能会让事情变得非常糟糕，因为缺乏插件支持，修改业务逻辑可能引发一连串服务的修改。

所有的服务都在一个进程

意识到上述的所有问题，以及这么一个事实，即一个可以正常工作的 IaaS 软件必须和所有的编排服务一起运行之后，ZStack 把所有服务封装在单一进程中，称之为管理节点。除去一些微服务已经带来的如解耦架构的优点外，进程内的微服务还给了我们很多额外的好处：

1. 简洁的依赖

因为所有服务都运行在同一进程内，软件只需要一份支持软件（如：database library, message library）的拷贝；升级或改变支持库跟我们对一个单独的二进制应用程序所做的一样简单。

2. 高可用，负载均衡和监控

服务可以专注于它们的业务逻辑，而不受各种来自于高可用、负载均衡、监控的干扰，这一切只由管理节点关心；更进一步，状态可以从服务中分离以创建无状态服务，详见 ZStack's Scalability Secrets Part 2: Stateless Services。

3. 中心化的配置

由于在一个进程中，所有的服务共享一份配置文件——zstack.properties；用户不需要去管理各种各样的分散在不同机器上的配置文件。

4. 易于部署、升级、维护和横向扩展

部署，升级或者维护一个单一的管理节点跟部署升级一个单一的应用程序一样容易。横向扩展服务只需要简单的增加管理节点。

5. 允许插件

因为运行在一个单一的进程中，插件可以很容易地被创建，和给传统的单进程应用程序添加插件一样。

进程内的微服务并不是一个新发明：*早在 90 年代，微软在 COM（Component Object Model）中把 server 定义为远程、本地和进程内三种。这些进程内的 server 是一些 DLLs，被应用程序在同一进程空间内加载，属于进程内的微服务。Peter Kriens 在四年前就声称已经定义了一种总是在同一进程内通信的服务，OSGi µservices。*

服务样例

在微服务中，一个服务通常是一个可重复的业务活动的逻辑表示，是无关联的、松耦合的、自包含的，而且对服务的消费者而言是一个“黑盒子”。简单来说，一个传统的微服务通常只关心特定的业务逻辑，有自己的 API 和配置方法，并能像一个独立的应用程序一样运行。尽管 ZStack 的服务共享同一块进程空间，它们拥有这些特点中的绝大多数。ZStack 很大程度上是一个使用强类型语言 java 编写的项目，但是在各个编排服务之间没有编译依赖性，例如：计算服务（包含 VM 服务、主机服务、区域服务、集群服务）并不依赖于存储服务（包含磁盘服务、基础存储服务、备份存储服务、磁盘快照服务等），虽然这些服务在业务流程中是紧密耦合的。

在源代码中，一个 ZStack 的服务并不比一个作为一个独立的 jar 文件构建的 maven 模块多任何东西。每一个服务可以定义自己的 APIs、错误码、全局配置，全局属性和系统标签。例如 KVM 的主机服务拥有自己的 APIs（如下所示）和各种各样的允许用户自己定义配置的方式。

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

<service xmlns="http://zstack.org/schema/zstack">

    <id>host</id>

    <message>

        <name>org.zstack.kvm.APIAddKVMHostMsg</name>

        <interceptor>HostApiInterceptor</interceptor>

        <interceptor>KVMApiInterceptor</interceptor>

    </message>

</service>
```

通过全局配置来配置

备注：*这里只简单展示一小部分，用户可以使用 API 去更新/获取全局配置，在这里展示一下全局配置的视图。*

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

      <globalConfig xmlns="http://zstack.org/schema/zstack">
```

```
    <config>
        <category>kvm</category>
        <name>vm.migrationQuantity</name>
        <description>A value that defines how many vm can be migrated in parallel when putting a KVM h
ost into maintenance mode.（当一个 KVM 主机变成维护模式的时候，这里的值定义了可以被并发迁移的虚拟机的数量）<
/description>
        <defaultValue>2</defaultValue>
        <type>java.lang.Integer</type>
    </config>

    <config>
        <category>kvm</category>
        <name>reservedMemory</name>
        <description>The memory capacity reserved on all KVM hosts. ZStack KVM agent is a python web s
erver that needs some memory capacity to run. this value reserves a portion of memory for the agent as
 well as other host applications. The value can be overridden by system tag on individual host, cluste
r and zone level（所有的 KVM 主机预留的内存容量。ZStack 中的 KVM 代理运行时是一个需要一部分内存容量去运行的 py
thon 的 web 服务器，这个值为代理和其他主机应用程序预留了一部分内存，在单一主机上的、集群上的、区域上的系统标签
可以覆盖这个值）</description>
        <defaultValue>512M</defaultValue>
    </config>
      </globalConfig>
```

通过全局属性配置

备注：*以下代码对应 zstack.properties 文件夹中相应的属性*

```
@GlobalPropertyDefinition
public class KVMGlobalProperty {
```

```
    @GlobalProperty(name="KvmAgent.agentPackageName", defaultValue = "kvmagent-0.6.tar.gz")
    public static String AGENT_PACKAGE_NAME;
    @GlobalProperty(name="KvmAgent.agentUrlRootPath", defaultValue = "")
    public static String AGENT_URL_ROOT_PATH;
    @GlobalProperty(name="KvmAgent.agentUrlScheme", defaultValue = "http")
    public static String AGENT_URL_SCHEME;
}
```

通过系统标签配置

备注：*以下代码对应数据库中相应的系统标签。*

```
      @TagDefinition
      public class KVMSystemTags {
    public static final String QEMU_IMG_VERSION_TOKEN = "version";
    public static PatternedSystemTag QEMU_IMG_VERSION = new PatternedSystemTag(String.format("qemu-im
g::version::%s", QEMU_IMG_VERSION_TOKEN), HostVO.class);

    public static final String LIBVIRT_VERSION_TOKEN = "version";
    public static PatternedSystemTag LIBVIRT_VERSION = new PatternedSystemTag(String.format("libvir
t::version::%s", LIBVIRT_VERSION_TOKEN), HostVO.class);

    public static final String HVM_CPU_FLAG_TOKEN = "flag";
    public static PatternedSystemTag HVM_CPU_FLAG = new PatternedSystemTag(String.format("hvm::%s", H
VM_CPU_FLAG_TOKEN), HostVO.class);
}
```

载入服务

服务在 Spring 的 bean 的 xml 文件中声明自身，例如，kvm 的部分声明类似于：

```
      <?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="KvmHostReserveExtension" class="org.zstack.kvm.KvmHostReserveExtension">

        <zstack:plugin>

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

            <zstack:extension interface="org.zstack.header.allocator.HostReservedCapacityExtensionPoint" />

        </zstack:plugin>

    </bean>


    <bean id="KVMHostFactory" class="org.zstack.kvm.KVMHostFactory">
```

```
        <zstack:plugin>
            <zstack:extension interface="org.zstack.header.host.HypervisorFactory" />
            <zstack:extension interface="org.zstack.header.Component" />
            <zstack:extension interface="org.zstack.header.managementnode.ManagementNodeChangeListene
r" />
            <zstack:extension interface="org.zstack.header.volume.MaxDataVolumeNumberExtensionPoint"
/>
        </zstack:plugin>
    </bean>

    <bean id="KVMSecurityGroupBackend" class="org.zstack.kvm.KVMSecurityGroupBackend">
        <zstack:plugin>
            <zstack:extension interface="org.zstack.network.securitygroup.SecurityGroupHypervisorBack
end" />
            <zstack:extension interface="org.zstack.kvm.KVMHostConnectExtensionPoint" />
        </zstack:plugin>
    </bean>

    <bean id="KVMConsoleHypervisorBackend" class="org.zstack.kvm.KVMConsoleHypervisorBackend">
        <zstack:plugin>
            <zstack:extension interface="org.zstack.header.console.ConsoleHypervisorBackend"/>
        </zstack:plugin>
    </bean>

    <bean id="KVMApiInterceptor" class="org.zstack.kvm.KVMApiInterceptor">
        <zstack:plugin>
```

```
        <zstack:extension interface="org.zstack.header.apimediator.ApiMessageInterceptor"/>
    </zstack:plugin>
</bean>
  </beans>
```

管理节点，作为所有服务的容器，将在启动阶段读取它们的 XML 配置文件，载入每一个服务。

总结

在这篇文章中，我们演示了 ZStack 的进程内微服务架构。通过使用它，ZStack 拥有一个非常干净的，松耦合的代码结构，这是创建一个强壮 IaaS 软件的基础。

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。

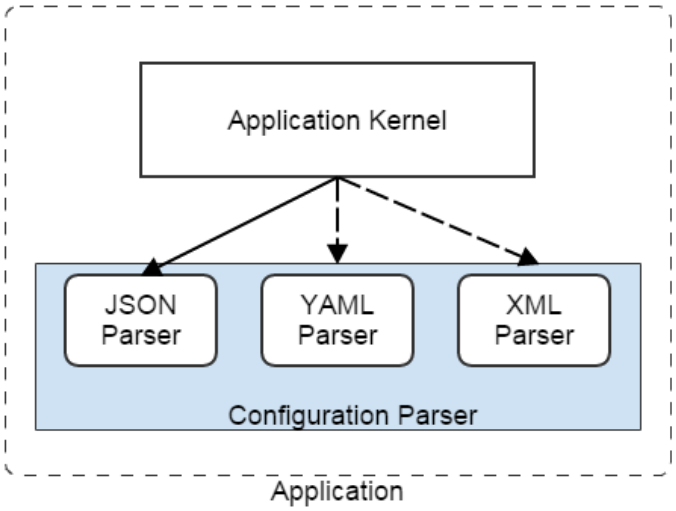
这表达当前 IaaS 软件的处境，除了一些构建的非常好的 IaaS（像 AWS），大多数 IaaS（包括我们的 ZStack）仍然不是一个完整的云解决方案。由于像 Amazon 这样的已经探索多年的先驱，公有云已经有一个更成熟的模型，对于公共云解决方案应该是什么样子而言。由于仍然处在开发阶段，私有云目前还没有经过验证的完整的解决方案。不像专用的公有云软件，可以专门为制造商的基础设施和服务定制；开源的 IaaS 软件必须同时考虑公有云和私有云的需求，使得创建一个完整的解决方案变得更加困难。因为我们没有办法预测一个完整的解决方案应该是什么样，我们唯一的办法是提供一个插件式的架构，它能在添加插件的同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

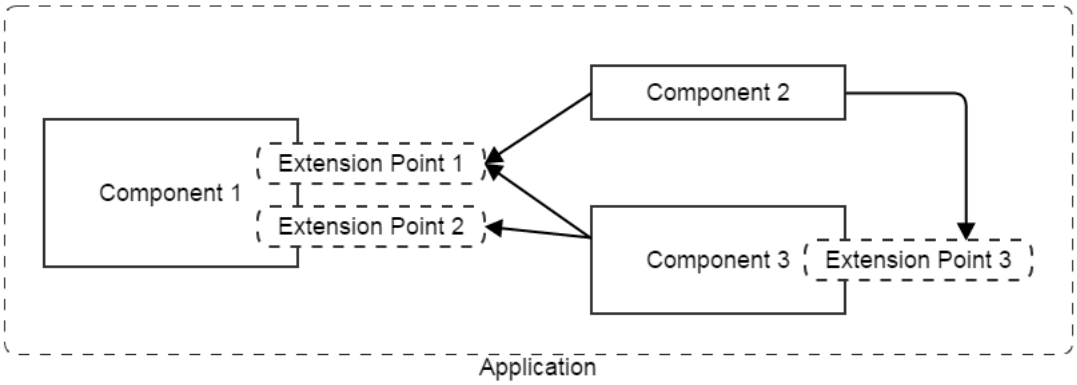
源自策略者模式的插件

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 in
herits
         * 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 PreVmInstantiateResourceExten
sionPoint {

    @Override

    public void preBeforeInstantiateVmResource(VmInstanceSpec spec) throws VmInstantiateR
esourceException {

        /*

   * you can do some check here; if any condition makes you think the VM should not be creat
ed/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 V
mInstanceSpec */

        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 KVMHostConnectExten
sionPoint {
        @Override
        public void kvmHostConnected(KVMHostConnectedContext context) throws KVMHostConnectEx
ception {
            /*
        * you can use various methods like SSH login, HTTP call to KVM agent to check the Openv
switch 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KVMHostConnectedContex
t 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 KVMHostConnectExte
nsionPoint 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。

ZSTACK--工作流引擎

在 IaaS 软件中的任务通常有很长的执行路径，一个错误可能发生在任意一个给定的步骤。为了保持系统的完整性，一个 IaaS 软件必须提供一套机制用于回滚先前的操作步骤。通过一个工作流引擎，ZStack 的每一个步骤，包裹在独立的工作流中，可以在出错的时候回滚。此外，通过在配置文件中组装工作流的方式，关键的执行路径可以被配置，这使得架构的耦合度进一步降低。

动机

数据中心是由大量的、各种各样的包括物理的（比如：存储，服务器）和虚拟的（比如：虚拟机）在内的资源组成的。IaaS 软件本质就是管理*各种资源*的状态；例如，创建一个虚拟机通常会改变存储的状态（在存储上创建了一个新的磁盘），网络的状态（在网络上设置 DHCP/DNS/NAT 等相关信息），和虚拟机管理程序的状态（在虚拟机管理程序上创建一个新的虚拟机）。不同于普通的应用程序，它们绝大多数时候都在管理存储在内存或数据库的状态。为了反映出数据中心的整体状态，IaaS 软件必须管理分散在各个设备的状态，导致执行路径很长。一个 IaaS 软件任务通常会涉及在多个设备上的状态改变，错误可能在任何步骤发生，然后让系统处在一个中间状态，即一些设备已经改变了状态而一些没有。例如，创建一个虚拟机时，IaaS 软件配置 VM 网络的常规步骤为 DHCP→DNS→SNAT，如果在创建 SNAT 时发生错误，之前配置的 DHCP 和 DNS 很有可能还留在系统内，因为它们已经成功地被应用，即使虚拟机最后无法成功创建。这种状态不一致的问题通常使云不稳定。

另一方面，硬编码的业务逻辑在传统的 IaaS 软件内对于改变来说是不灵活的；开发人员往往要重写或修改现有的代码来改变一些既定的行为，这些影响了软件的稳定性。

这些问题的解决方法是引入工作流的概念，将整块的业务逻辑分解成细粒度的、可回滚的步骤，使软件可以清理已经生成的错误的状态，使软件变得可以配置。

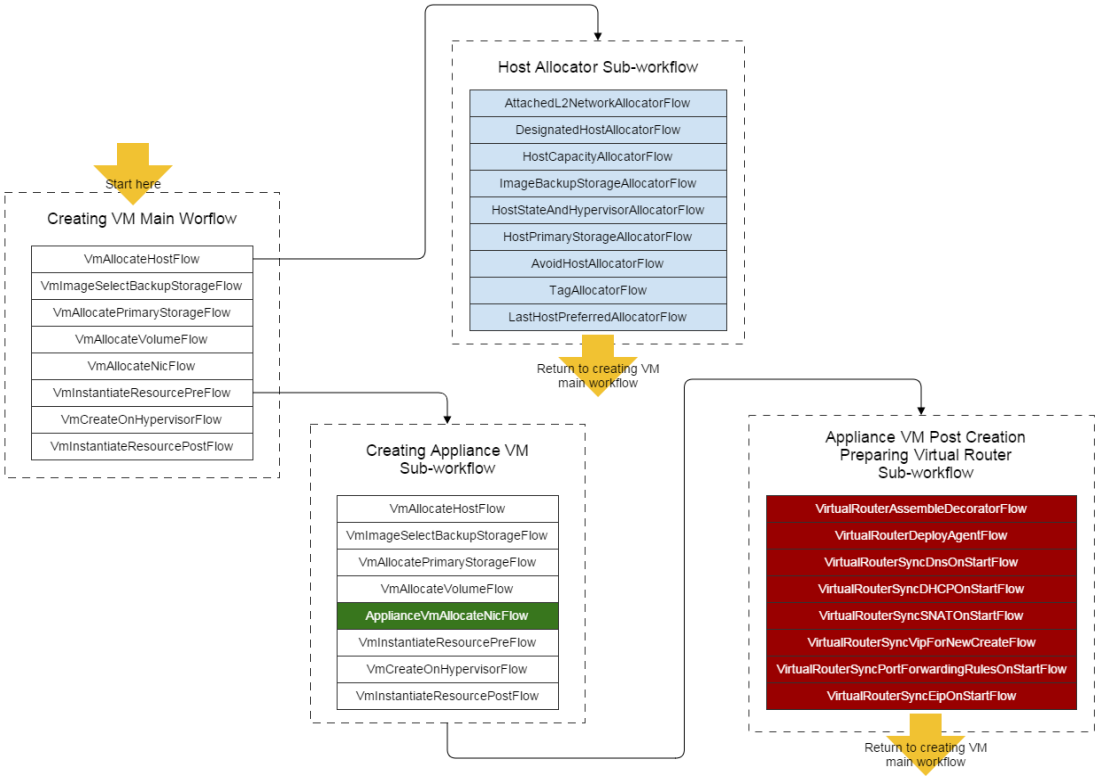
注意：在 ZStack 中，我们可以将工作流中的步骤（step）称为"流程（flow）"，在以下文章中，流程（flow）和步骤（step）是可以互换的。

问题

错误处理在软件设计中总是一个很头疼的问题。即使现在每一个软件工程师都知道了错误处理的重要性，但是实际上，他们仍然在找借口忽略它。精巧的错误处理是很难的，尤其是在一个任务可能跨越多个组件的系统中。即使富有经验的工程师可以关注自己代码中的错误，他们也不可能为不是他们所写的组件付出类似的努力，如果整个架构中没有强制一种统一的，可以全局加强错误处理的机制。忽略错误处理在一个 IaaS 软件中是特别有害的。不像消费级程序可以通过重启来恢复所有的状态，一个 IaaS 软件通常没有办法自己恢复状态，将会需要管理员们去手动更正在数据库和外部设备中的错误。一个单一的状态不一致可能不会导致任何大的问题，而且也可能甚至不会被注意到，但是这种状态不一致性的不断积累将会在某个时刻最终摧毁整个云系统。

工作流引擎

工作流是一种方法，把一些繁琐的方法调用分解为一个个专注于一件事情的、细粒度的步骤，它由序列或状态机驱动，最终完成一个完整的任务。配置好回滚处理程序后，当错误或未处理的异常在某一步骤发生时，一个工作流可以中止执行并回滚所有之前的执行步骤。以创建虚拟机为例，主要工作流程看起来像：



顺序工作流，来源于*链式设计模式（Chain Pattern）*，有着可以预见的执行顺序，这是ZStack 工作流的基础。一个流程（flow），本质上是一个 java 接口，可以包含子流程，并只在前面所有流程完成后才可以执行。

```
public interface Flow {

    void run(FlowTrigger trigger, Map data);



    void rollback(FlowTrigger trigger, Map data);

}
```

在 Flow 接口中，工作流前进到这个流程（flow）的时候，run(FlowTrigger trigger, Map data) 方法会被调用；参数 Map data 可以被用于从先前的流程（flow）中获取数据并把数据传递给后续的流程（flow）。当自身完成时，这个流程（flow）调用 trigger.next()引导工作流（workflow）去执行下一个流程（flow）；如果一个错误发生了，这个流程（flow）应该调用 trigger.fail(ErrorCode error)方法中止执行，并通知工作流（workflow）回滚已经完成的流程（包括失败的流程自身）调用各自的 rollback()方法。

在 FlowChain 接口中被组建好的流程代表了一个完整的工作流程。有两种方法来创建一个 FlowChain：

1. 声明式

流程可以在一个组件的 Spring 配置文件中被配置，一个 FlowChain 可以通过填写一个*流程的类*的名字的列表到 FlowChainBuilder 中以被创建。

```
<bean id="VmInstanceManager" class="org.zstack.compute.vm.VmInstanceManagerImpl">
    <property name="createVmWorkFlowElements">
        <list>
            <value>org.zstack.compute.vm.VmAllocateHostFlow</value>
            <value>org.zstack.compute.vm.VmImageSelectBackupStorageFlow</value>
            <value>org.zstack.compute.vm.VmAllocatePrimaryStorageFlow</value>
            <value>org.zstack.compute.vm.VmAllocateVolumeFlow</value>
            <value>org.zstack.compute.vm.VmAllocateNicFlow</value>
            <value>org.zstack.compute.vm.VmInstantiateResourcePreFlow</value>
            <value>org.zstack.compute.vm.VmCreateOnHypervisorFlow</value>
            <value>org.zstack.compute.vm.VmInstantiateResourcePostFlow</value>
        </list>
    </property>

    <!--only a part of configuration is showed -->
</bean>
```

```
FlowChainBuilder createVmFlowBuilder = FlowChainBuilder.newBuilder().setFlowClassNames(createVmWorkFlowEleme
nts).construct();

FlowChain chain = createVmFlowBuilder.build();
```

这是创建一个严肃的、可配置的、包含可复用流程的工作流程的典型方式。在上面的例子中，那个工作流的目的是创建用户 VM；一个所谓的*应用 VM* 具有除分配虚拟机网卡

外基本相同的流程，所以 appliance VM 的单一的流程配置和用户 VM 的流程配置大多数是可以共享的：

```
<bean id="ApplianceVmFacade"
    class="org.zstack.appliancevm.ApplianceVmFacadeImpl">
    <property name="createApplianceVmWorkFlow">
        <list>
            <value>org.zstack.compute.vm.VmAllocateHostFlow</value>
            <value>org.zstack.compute.vm.VmImageSelectBackupStorageFlow</value>
            <value>org.zstack.compute.vm.VmAllocatePrimaryStorageFlow</value>
            <value>org.zstack.compute.vm.VmAllocateVolumeFlow</value>
            <value>org.zstack.appliancevm.ApplianceVmAllocateNicFlow</value>
            <value>org.zstack.compute.vm.VmInstantiateResourcePreFlow</value>
            <value>org.zstack.compute.vm.VmCreateOnHypervisorFlow</value>
            <value>org.zstack.compute.vm.VmInstantiateResourcePostFlow</value>
        </list>
    </property>

    <zstack:plugin>
        <zstack:extension interface="org.zstack.header.Component" />
        <zstack:extension interface="org.zstack.header.Service" />
    </zstack:plugin>
</bean>
```

***备注**：在之前的图片中，我们把 `ApplianceVmAllocateNicFlow` 流程高亮为绿色，这是创建用户 VM 和应用 VM 的工作流步骤中唯一不同的地方。*

2.编程的方式

一个 FlowChain 还可以通过编程方式创建。通常当要创建的工作流是琐碎的、流程不可复用的时候，使用这种方法。

```
FlowChain chain = FlowChainBuilder.newSimpleFlowChain();

chain.setName("test");

chain.setData(new HashMap());

chain.then(new Flow() {

    String __name__ = "flow1";

    @Override

    public void run(FlowTrigger trigger, Map data) {

        /* do some business */

        trigger.next();

    }



    @Override

    public void rollback(FlowTrigger trigger, Map data) {

        /* rollback something */

        trigger.rollback();

    }

}).then(new Flow() {

    String __name__ = "flow2";

    @Override

    public void run(FlowTrigger trigger, Map data) {

        /* do some business */

        trigger.next();
```

```
        }


        @Override
        public void rollback(FlowTrigger trigger, Map data) {
            /* rollback something */
            trigger.rollback();
        }
    }).done(new FlowDoneHandler() {
        @Override
        public void handle(Map data) {
            /* the workflow has successfully done */
        }
    }).error(new FlowErrorHandler() {
        @Override
        public void handle(ErrorCode errCode, Map data) {
            /* the workflow has failed with error */
        }
    }).start();
```

以上形式使用不方便，因为在流中通过一个 map data 交换数据，每一个流程必须冗余地调用 data.get()和 data.put()函数。使用一种类似 DSL 的方式，流可以通过变量共享数据：

```
    FlowChain chain = FlowChainBuilder.newShareFlowChain();
    chain.setName("test");
    chain.then(new ShareFlow() {
        String data1 = "data can be defined as class variables";
```

```
    {
        data1 = "data can be iintialized in object initializer";
    }

    @Override
    public void setup() {
        final String data2 = "data can also be defined in method scope, but it has to be final";

        flow(new Flow() {
            String __name__ = "flow1";

            @Override
            public void run(FlowTrigger trigger, Map data) {
                data1 = "we can change data here";
                String useData2 = data2;

                /* do something */
                trigger.next();
            }

            @Override
            public void rollback(FlowTrigger trigger, Map data) {
                /* do some rollback */
                trigger.rollback();
            }
```

```
        });

        flow(new NoRollbackFlow() {
            String __name__ = "flow2";

            @Override
            public void run(FlowTrigger trigger, Map data) {
                /* data1 is the value of what we have changed in flow1 */
                String useData1 = data1;

                /* do something */
                trigger.next();
            }
        });

        done(new FlowDoneHandler() {
            @Override
            public void handle(Map data) {
                /* the workflow has successfully done */
            }
        });

        error(new FlowErrorHandler() {
            @Override
            public void handle(ErrorCode errCode, Map data) {
```

```
                    /*the workflow has failed with error */

                }

            });

        }

    }).start();
```

总结

在这篇文章中，我们展示了 ZStack 的工作流引擎。通过使用它，在错误发生的时候，ZStack 在 99%的时间里可以很好地保持系统状态一致，注意是 99%的时间里，虽然工作流大多数时候是一个不错的处理错误的工具，但仍然有一些情况它不能处理，例如，回滚处理程序运行失败的时候。ZStack 还配备了垃圾收集系统，我们将在以后的文章对它进行介绍。