ZStack 技术白皮书精选

ZStack--工作流引擎

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--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。

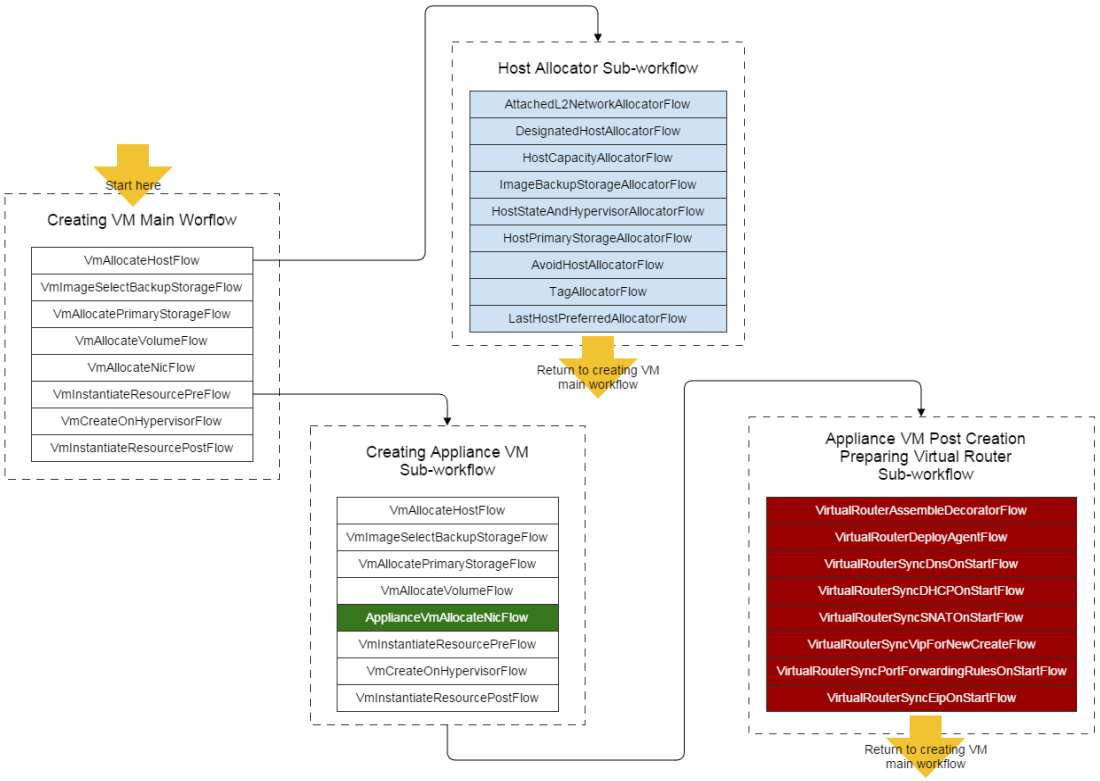
注意：在 ZStack 中，我们可以将工作流中的步骤（step）称为"流程（flow）"，在以下文章中，流程（flow）和步骤（step）是可以互换的。

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(createVmWorkFlowElements).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。