The Workflow Engine

by Frank Zhang on Apr 4, 2015

(/) (/) (/) (/) (/) (/) (/) |
Share (https://www.addtoany.com/share_save#url=http%3A%2F%2Fzstack.org%2Fblog%2Fworkflow.html&title=ZStack%20-%20The%20Workflow%20Engine&description=ZStack%20%3A%20ZStack%20is%20open%20source%20IaaS%20software%20managing%20resources%20of%20compute%2C%20storage%2C%20networking%20throughout%20a%20datacenter%20all%20by%20APIs.)

Tasks in IaaS software normally have long execution paths; an error may happen at any given step. In order to keep the integrity of the system, an IaaS software must provide a mechanism that can roll back prior executed steps. With a workflow engine, every step in ZStack wrapped in individual flow can be rolled back on error. Besides, key execution paths can be configured by assembling workflows in configuration files, which decouples the architecture further.

The motivation

A datacenter is made up of massive, various resources that can be both physical(e.g. storage, servers) and virtual(e.g. virtual machines). The IaaS software is essentially managing states of different resources; for example, creating a VM will normally change state of storage(a new volume is created on the storage), state of network(DHCP/DNS/SNAT related information are set on the network), and state of hypervisor(a new VM is created on the hypervisor). Unlike ordinary applications which largely manage own states stored in memory or database, IaaS software have to manage states scattered on every device in order to reflect the overall state of the datacenter, leading to long execution paths. A task of IaaS software will typically involve changing states on multiple devices, an error may happen at any step then leave the system in an intermediate stage where some devices have changed states while some have not. For example, when creating a VM, an IaaS software usually configures VM's network in order of DHCP --> DNS --> SNAT, if an error happens when creating SNAT, DHCP and DNS configured before will most likely remain in the system because they have been successfully applied, even the VM fails to be created at last. This kind of state inconsistent problem usually entails an unstable cloud.

On the other side, the hard-coded business logic in traditional IaaS software is inflexible for changes; developers often have to rewrite or modify existing code in order to change some established behaviors, which hurts the stability of the software.

The cure for those issues is to introduce the concept of workflow that can break monolithic business logic into fine-grained, rollbackable steps, allowing the software to clean up applied states on error and making the software configurable.

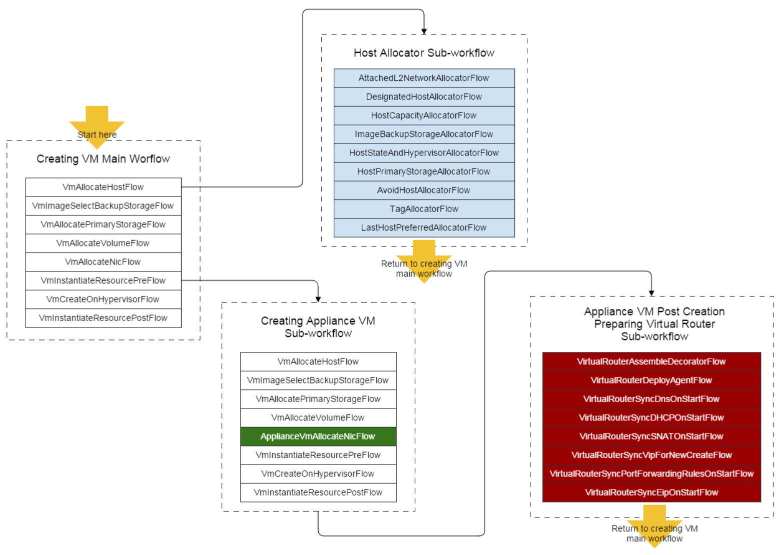
***Note**: In ZStack, we call each step in the workflow as a 'flow'; in following context, steps and flows are used interchangeable.*

The problem

Error handling is always a headache in software design. Though nearly every software engineer knows the importance of error handling, but in reality, they keep finding excuses for ignoring it. Elegant error handling is hard, especially in a system that a task may span separate components. Even if skilled engineers may take care of errors in their own code, they are not able to spread the effort to components not written by themselves if the architecture doesn't enforce a unified mechanism that can consolidate error handling globally. Ignoring error handling is particularly harmful in an IaaS software; unlike consuming applications that can be rebooted to recover all states, an IaaS software typically has no way to recover states itself, and will need administrators to manually correct states both in database and external devices. A solo state inconsistency may not cause any big problem and may not even be observed, but accumulated state inconsistencies will finally bring down the cloud at some point.

Workflow engine

Workflow is a way that breaks cumbersome method calls into small granularity steps that each step only focuses on one thing, and that are driven by either sequence or state-machine to complete a whole task. With rollback handlers installed, a workflow can rollback all prior executed steps and abort the execution when an error or a unhandled exception happens in a step. Cite creating VM as an example, the major workflow looks like:


Start here
Creating VM Main Worflow
VmAllocateHostFlow
VmImageSelectBackupStorageFlow
VmAllocatePrimaryStorageFlow
VmAllocateVolumeFlow
VmAllocateNicFlow
VmInstantiateResourcePreFlow
VmCreateOnHypervisorFlow
VmInstantiateResourcePostFlow
Host Allocator Sub-workflow
AttachedL2NetworkAllocatorFlow
DesignatedHostAllocatorFlow
HostCapacityAllocatorFlow
ImageBackupStorageAllocatorFlow
HostStateAndHypervisorAllocatorFlow
HostPrimaryStorageAllocatorFlow
AvoidHostAllocatorFlow
TagAllocatorFlow
LastHostPreferredAllocatorFlow
Return to creating VM main workflow
Creating Appliance VM Sub-workflow
VmAllocateHostFlow
VmImageSelectBackupStorageFlow
VmAllocatePrimaryStorageFlow
VmAllocateVolumeFlow
ApplianceVmAllocateNicFlow
VmInstantiateResourcePreFlow
VmCreateOnHypervisorFlow
VmInstantiateResourcePostFlow
Appliance VM Post Creation Preparing Virtual Router Sub-workflow
VirtualRouterAssembleDecoratorFlow
VirtualRouterDeployAgentFlow
VirtualRouterSyncDnsOnStartFlow
VirtualRouterSyncDHCPOnStartFlow
VirtualRouterSyncSNATOnStartFlow
VirtualRouterSyncVipForNewCreateFlow
VirtualRouterSyncPortForwardingRulesOnStartFlow
VirtualRouterSyncEipOnStartFlow
Return to creating VM main workflow

Sequential workflow, which is originated from Chain pattern (http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern) and has predictable execution order, is the basis of ZStack workflow. A flow, essentially a Java interface, can contain sub-workflow and can execute only after all ahead flows finish.

```
public interface Flow {
    void run(FlowTrigger trigger, Map data);

    void rollback(FlowTrigger trigger, Map data);
}
```

In the `Flow` interface, the `run(FlowTrigger trigger, Map data)` method is called when the workflow has proceeded to this flow; the parameter `Map data` can be used to retrieve data from previous flows and pass data to later flows. When finishing, a flow calls `trigger.next()` to instruct the workflow to move to the next; if an error happens, the flow should call `trigger.fail(ErrorCode error)` to abort the execution, notifying the workflow to rollback accomplished flows(including the failure flow itself) invoking `rollback()` method of them.

Flows are organized in `FlowChain` interface that represents an entire workflow. There are two ways to create a `FlowChain` :

1. Declarative way

Flows can be configured in a component's Spring configuration file, a `FlowChain` can be created by feeding a list of flow class names to `FlowChainBuilder` .

```
<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();
```

This is a typical way to create serious, configurable workflow containing reusable flows. In above example, the workflow is for creating user VM; a so-called appliance VM has basically the same flows except the one allocating VM nics, so a separate flow configuration for appliance VM can have most flows mutual to user VM's:

```
<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>
```

> ***Note**: In the former picture, we have highlighted the flow `ApplianceVmAllocateNicFlow` in green; that is the only flow that differentiates creational workflow of user VM and appliance VM.*

2. Programmatic way

A `FlowChain` can also be created in a programmatic way. Usually when a workflow is trivial, and flows are nonreusable.

```
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();
```

The above form is not handy because the data interchange amid flows is through a map `data` , every flow has to call `data.get()` and `data.put()` , which is very verbose. With a way DSL (http://en.wikipedia.org/wiki/Domain-specific_language) alike, flows can share data through variables:

```
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();
```

Summary

In this article, we demonstrated ZStack's workflow engine. With it, ZStack can elegantly keep states of the system consistent when errors happen, in 99% time. Note we say 99% time; though workflow is a great tool to handle most errors, there are still some cases it can't handle, for example, failures in rollback handlers. ZStack is also equipped with a garbage collection system that we will introduce in another article in future.

0 Comments zstack.org 1 Login

Recommend Share Sort by Best

Start the discussion…

Be the first to comment.

Subscribe Add Disqus to your site Privacy

Community

Mailing List (https://groups.google.com/forum/#!forum/zstack)
Community (http://www.zstack.org/community)
Gitter

Resources

Intallation (http://www.zstack.org/installation)
Tutorials (http://www.zstack.org/tutorials)
Blog (http://www.zstack.org/blog)
Documentation (http://www.zstack.org/documentation)

Connect Us

(https://twitter.com/zstack_org) (https://www.facebook.com/zstackorg) (https://github.com/zstackorg/zstack)
(../misc/wechat.html) (http://weibo.com/zstack)

Your feedback is invaluable, please let us know your thoughts. (mailto:info@zstack.org)