ZStack 技术白皮书精选

自动化测试系统

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。

目录

ZSTACK--自动化测试系统 1：集成测试

测试，对于一个 IaaS 软件的可靠性、成熟度和可维护性而言，是一个重要的因素.测试在 ZStack 中是全自动的。这个自动化测试系统包括了三个部分：集成测试，系统测试，基于模块的测试。其中集成测试构建于 Junit 之上，使用了模拟器。通过这个集成测试系统提供的各种各样的功能，开发人员可以快速的写出测试用例，用于验证一个新特性或者一个缺陷修复。

概述

这个关键因素，在构建一个可靠的、成熟的和可维护的软件产品中，就是架构；这是我们自始自终相信的设计原则。ZStack 已经付出了大量的努力，以设计这么一个架构：始终保持软件稳定，无论是添加新特性，常规的操作错误，还是为特殊目的裁剪；我们之前的文章：ZStack—进程内微服务架构、ZStack—通用插件系统、ZStack—工作流引擎、ZStack—标签系统，已经表现了我们的一些尝试。然而，我们也充分理解测试在软件开发中的重要性。ZStack，从第一天开始，设定了这么一个目标：每一个特性都必须有测试用例保证，测试必须是全部自动化的，写单元测试应该是验证一个新特性或任何代码改变的唯一方式。

为了实现这个目标，我们把我们的测试系统分成了三个组件：集成测试，系统测试，模块测试。分类方式是通过它们的关注点和功能。

- **集成测试系统**构建于 Junit，全部使用模拟器；测试用例存放在 ZStack 的 Java 源代码中；开发人员可以轻松地使用常规的 Junit 命令来启动测试套件。
- **系统测试系统**是一个独立的 Python 项目，称之为 *zstack-woodpecker*，基于 ZStack 的 API；在一个真实的硬件环境中测试一切。

- **基于模块的测试系统**构建于基于模块的测试这么一个理论，是 *zstack-woodpecker* 中的一个子项目。这个系统中的测试用例将会持续地，以随机的方式，执行 API，直到一些预定义的条件被满足。

从这篇文章开始，我们将会有一系列的，共计三篇文章，来详细阐述我们的测试架构，以向你展示我们保证 ZStack 每一个特性的方式。

单元测试的几句话

好奇的读者可能已经在他们的心中问了这么一个问题，为什么我们没有提到单元测试，这么一个可能是最著名的，也是每一个冷静的测试驱动的开发人员会强调的测试概念。我们确实有单元测试。如果你看到了后续的章节：*测试框架*，你可能会困惑，为什么用在命令中的命名类似于：UnitTest balabala，但在这篇文章中被命名为集成测试。

一开始，我们认为我们的测试就是单元测试，因为每一个用例都是用于验证一个独立的组件，而不是整个软件；例如，这么一个用例：TestCreateZone，只测试 Zone 服务，其他的组件，像 VM 服务、存储服务将甚至不会被加载。然而，我们做测试的方式确实和传统的单元测试概念有所不同，传统的方式是测试一小段代码，通常是针对内部结构的白盒测试，使用 mock 和 stub 的方法论。当前的 ZStack 有大概 120 个测试用例满足这个定义，而剩下的 500 多个并不。大多数的测试用例，甚至关注于独立服务或组件的，都更像集成测试用例，因为会加载多个依赖的服务、组件用以执行一个测试活动。

另一方面，我们大多数的，基于模拟器的测试用例，都实际上在 API 层面进行测试，这对单元测试的定义而言，这就是倾向于集成测试的黑盒测试。基于这些事实，我们最终改变了我们的主意，我们将要做的是集成测试，不过保留了大量的旧的命名方式，类似 UnitTest balabla。

集成测试

从我们先前的经验中，我们深刻地意识到，开发人员持续忽视测试的一个主要原因是：***写测试太难了，有的时候甚至比实现一个特性还要难***。当我们设计这个集成测试系统的时候，一个反复考虑的地方便是尽可能地从开发人员那边卸下负担，让系统自身做绝大多数无聊、繁杂的工作。

对于几乎所有的测试用例而言，有两种重复性的工作。其中一个是准备一个最小的但是可以工作的软件；例如，为了测试一个 zone，你只需要核心的库和 zone 服务被加载，没有必要加载其他的服务，因为我们不需要它们。另一个是准备环境；例如，一个测试 VM 创建的用例，会需要这么一个环境，有一个 zone、一个 cluster、一个 host、存储、网络和所有的其他必须的资源准备就绪；开发人员不会想去重复无聊的事情，像创建一个 zone，添加一个 host，在他们能够真正开始测试自己的东西之前；理想的情况是，他们可以以最小的努力便获得一个准备就绪的环境，以集中精力与他们想测试的东西。

组件加载器

我们解决了所有的这些问题，通过一个构建于 JUnit 之上的框架。在一切开始之前，由于 ZStack 通过使用 Spring 管理着所有的组件，我们创建了一个 BeanConstruct，这样测试人员可以按需指定他们想要加载的组件：

```
public class TestCreateZone {

    Api api;

    ComponentLoader loader;

    DatabaseFacade dbf;
```

```
    @Before
    public void setUp() throws Exception {
        DBUtil.reDeployDB();
        BeanConstructor con = new BeanConstructor();
        loader =
con.addXml("PortalForUnitTest.xml").addXml("ZoneManager.xml").
addXml("AccountManager.xml").build();
        dbf = loader.getComponent(DatabaseFacade.class);
        api = new Api();
        api.startServer();
    }
```

在上面这个例子中，我们添加了三个 Spring 配置到 BeanConstructor，它们的名字暗示了将会为账户服务、zone 服务和其他包括在 PortalForUnitTest.xml 中的库加载组件。通过这种方式，测试人员可以把软件定制成一个最小的尺寸，仅包含需要的组件，以便加速测试过程和使东西易于调试。

环境部署器

为了帮助测试人员准备一个环境，包含将被测试的活动的所有必须依赖，我们创建了一个部署器，可以读取一个 XML 配置文件以部署一个完整的模拟器环境：

```
public class TestCreateVm {

    Deployer deployer;

    Api api;

    ComponentLoader loader;

    CloudBus bus;

    DatabaseFacade dbf;


    @Before

    public void setUp() throws Exception {

        DBUtil.reDeployDB();

        deployer = new
Deployer("deployerXml/vm/TestCreateVm.xml");

        deployer.build();

        api = deployer.getApi();

        loader = deployer.getComponentLoader();

        bus = loader.getComponent(CloudBus.class);

        dbf = loader.getComponent(DatabaseFacade.class);

    }


    @Test

    public void test() throws ApiSenderException,
InterruptedException {
```

```
        InstanceOfferingInventory ioinv =
api.listInstanceOffering(null).get(0);

        ImageInventory iminv = api.listImage(null).get(0);

        VmInstanceInventory inv =
api.listVmInstances(null).get(0);

        Assert.assertEquals(inv.getInstanceOfferingUuid(),
ioinv.getUuid());

        Assert.assertEquals(inv.getImageUuid(),
iminv.getUuid());

        Assert.assertEquals(VmInstanceState.Running.toString(),
inv.getState());

        Assert.assertEquals(3, inv.getVmNics().size());

        VmInstanceVO vm = dbf.findByUuid(inv.getUuid(),
VmInstanceVO.class);

        Assert.assertNotNull(vm);

        Assert.assertEquals(VmInstanceState.Running,
vm.getState());

        for (VmNicInventory nic : inv.getVmNics()) {

            VmNicVO nvo = dbf.findByUuid(nic.getUuid(),
VmNicVO.class);

            Assert.assertNotNull(nvo);

        }

        VolumeVO root = dbf.findByUuid(inv.getRootVolumeUuid(),
VolumeVO.class);
```

```
        Assert.assertNotNull(root);

        for (VolumeInventory v : inv.getAllVolumes()) {

            if (v.getType().equals(VolumeType.Data.toString()))
{

                VolumeVO data = dbf.findByUuid(v.getUuid(),
VolumeVO.class);

                Assert.assertNotNull(data);

            }

        }

    }

}
```

在上面这个 TestCreateVm 的用例中，部署器读取了一个配置文件，存放在 deployerXml/vm/TestCreateVm.xml，然后部署了一个完整的，准备好创建新的 VM 的环境；更进一步，我们事实上让部署器创建了这个 VM，正如你并没有在 test 方法看到任何代码调用 api.createVmByFullConfig();测试人员真正做的事情是，验证这个 VM 是否按照我们在 deployerXml/vm/TestCreateVm.xml 中指定的条件正确地创建。现在你看到了这一切是多么的容易了，测试人员只写了大概 60 行代码，然后将一个 IaaS 软件中最重要的部分——创建 VM，测试好。

这个在上面例子中的配置文件 TestCreateVm.xml 看起来像：

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

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

```
    <instanceOfferings>
        <instanceOffering name="TestInstanceOffering"
            description="Test" memoryCapacity="3G" cpuNum="1"
cpuSpeed="3000" />
    </instanceOfferings>

    <backupStorages>
        <simulatorBackupStorage name="TestBackupStorage"
            description="Test" url="nfs://test" />
    </backupStorages>

    <images>
        <image name="TestImage" description="Test"
format="simulator">
<backupStorageRef>TestBackupStorage</backupStorageRef>
        </image>
    </images>

    <diskOffering name="TestRootDiskOffering"
description="Test"
        diskSize="50G" />
```

```
    <diskOffering name="TestDataDiskOffering"
description="Test"
        diskSize="120G" />

    <vm>
        <userVm name="TestVm" description="Test">

<rootDiskOfferingRef>TestRootDiskOffering</rootDiskOfferingRef
>
            <imageRef>TestImage</imageRef>

<instanceOfferingRef>TestInstanceOffering</instanceOfferingRef
>
            <l3NetworkRef>TestL3Network1</l3NetworkRef>
            <l3NetworkRef>TestL3Network2</l3NetworkRef>
            <l3NetworkRef>TestL3Network3</l3NetworkRef>

<defaultL3NetworkRef>TestL3Network1</defaultL3NetworkRef>

<diskOfferingRef>TestDataDiskOffering</diskOfferingRef>
        </userVm>
    </vm>
```

```
    <zones>
        <zone name="TestZone" description="Test">
            <clusters>
                <cluster name="TestCluster" description="Test">
                    <hosts>
                        <simulatorHost name="TestHost1"
description="Test"
                            managementIp="10.0.0.11"
memoryCapacity="8G" cpuNum="4" cpuSpeed="2600" />
                        <simulatorHost name="TestHost2"
description="Test"
                            managementIp="10.0.0.12"
memoryCapacity="4G" cpuNum="4" cpuSpeed="2600" />
                    </hosts>

<primaryStorageRef>TestPrimaryStorage</primaryStorageRef>
                    <l2NetworkRef>TestL2Network</l2NetworkRef>
                </cluster>
            </clusters>

            <l2Networks>
                <l2NoVlanNetwork name="TestL2Network"
description="Test"
```

```
                physicalInterface="eth0">
                <l3Networks>
                    <l3BasicNetwork name="TestL3Network1"
description="Test">
                        <ipRange name="TestIpRange1"
description="Test" startIp="10.0.0.100"
                            endIp="10.10.1.200"
gateway="10.0.0.1" netmask="255.0.0.0" />
                    </l3BasicNetwork>
                    <l3BasicNetwork name="TestL3Network2"
description="Test">
                        <ipRange name="TestIpRange2"
description="Test" startIp="10.10.2.100"
                            endIp="10.20.2.200"
gateway="10.10.2.1" netmask="255.0.0.0" />
                    </l3BasicNetwork>
                    <l3BasicNetwork name="TestL3Network3"
description="Test">
                        <ipRange name="TestIpRange3"
description="Test" startIp="10.20.3.100"
                            endIp="10.30.3.200"
gateway="10.20.3.1" netmask="255.0.0.0" />
                    </l3BasicNetwork>
                </l3Networks>
```

```
            </l2NoVlanNetwork>
        </l2Networks>

        <primaryStorages>
            <simulatorPrimaryStorage
name="TestPrimaryStorage"
                description="Test" totalCapacity="1T"
url="nfs://test" />
        </primaryStorages>

<backupStorageRef>TestBackupStorage</backupStorageRef>
    </zone>
  </zones>
</deployerConfig>
```

模拟器

大多数集成测试用例都构建于模拟器之上；每一个资源，只要它需要和后端设备通信，都有一个模拟器实现；例如，KVM 模拟器，虚拟路由虚拟机的模拟器，NFS 主存储的模拟器。因为现在的资源后端都是基于 Python 的 HTTP 服务器，大多数模拟器通过嵌入了 HTTP 服务器的 Apache Tomcat 被构建。KVM 模拟器的一小段代码看起来像：

```
@RequestMapping(value=KVMConstant.KVM_MERGE_SNAPSHOT_PATH,
method=RequestMethod.POST)
    public @ResponseBody String
mergeSnapshot(HttpServletRequest req) {
        HttpEntity<String> entity =
restf.httpServletRequestToHttpEntity(req);
        MergeSnapshotCmd cmd =
JSONObjectUtil.toObject(entity.getBody(),
MergeSnapshotCmd.class);
        MergeSnapshotRsp rsp = new MergeSnapshotRsp();
        if (!config.mergeSnapshotSuccess) {
            rsp.setError("on purpose");
            rsp.setSuccess(false);
        } else {
            snapshotKvmSimulator.merge(cmd.getSrcPath(),
cmd.getDestPath(), cmd.isFullRebase());
            config.mergeSnapshotCmds.add(cmd);
            logger.debug(entity.getBody());
        }

        replyer.reply(entity, rsp);
        return null;
    }
```

```
@RequestMapping(value=KVMConstant.KVM_TAKE_VOLUME_SNAPSHOT_PATH, method=RequestMethod.POST)
    public @ResponseBody String takeSnapshot(HttpServletRequest req) {
        HttpEntity<String> entity = restf.httpServletRequestToHttpEntity(req);
        TakeSnapshotCmd cmd = JSONObjectUtil.toObject(entity.getBody(), TakeSnapshotCmd.class);
        TakeSnapshotResponse rsp = new TakeSnapshotResponse();
        if (config.snapshotSuccess) {
            config.snapshotCmds.add(cmd);
            rsp = snapshotKvmSimulator.takeSnapshot(cmd);
        } else  {
            rsp.setError("on purpose");
            rsp.setSuccess(false);
        }
        replyer.reply(entity, rsp);
        return null;
    }
```

每一个模拟器都有一个配置对象，像 KVMSimulatorConfig，可以被测试人员用于控制模拟器的行为。

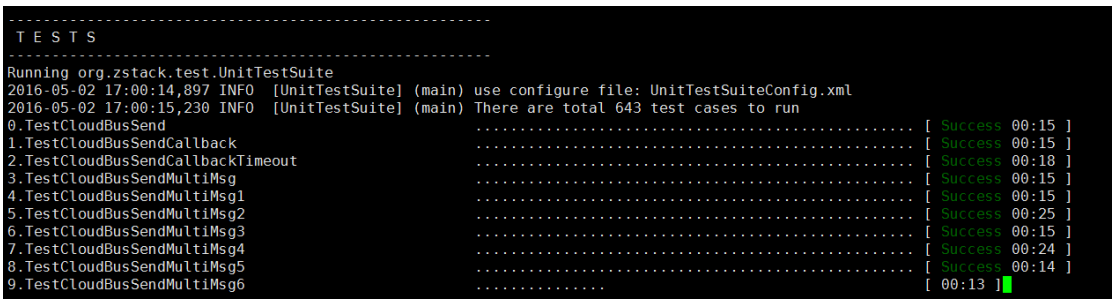
测试框架

由于所有的测试用例都事实上是 Junit 测试用例，测试人员可以使用通常的 Junit 命令单独地跑每一个测试用例，例如：

```
[root@localhost test]# mvn test -Dtest=TestAddImage
```

而且一个测试套件中的所有用例可以用一条命令执行，例如：

```
[root@localhost test]# mvn test -Dtest=UnitTestSuite
```



用例也可以在一个组里被执行，例如：

```
[root@localhost test]# mvn test -Dtest=UnitTestSuite -
Dconfig=unitTestSuiteXml/eip.xml
```

一个 XML 配置文件列出了一个组里的用例，比如，上面的 eip.xml 看起来像：

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

<UnitTestSuiteConfig xmlns="http://zstack.org/schema/zstack"
timeout="120">

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip1"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip2"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip3"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip4"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip5"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip6"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip7"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip8"/>
```

```
    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip9"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip10"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip11"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip12"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip13"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip14"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip15"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip16"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip17"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip18"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip19"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip20"/>
```

```
    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip21"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip22"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip23"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip24"/>

    <TestCase
class="org.zstack.test.eip.TestVirtualRouterEip25"/>

    <TestCase class="org.zstack.test.eip.TestQueryEip1"/>

    <TestCase
class="org.zstack.test.eip.TestEipPortForwardingAttachableNic"
/>

</UnitTestSuiteConfig>
```

多个用例也可以在一条命令中执行，只要填充它们的名字，例如：

```
[root@localhost test]# mvn test -Dtest=UnitTestSuite -
Dcases=TestAddImage,TestCreateTemplateFromRootVolume,TestCreat
eDataVolume
```

总结

在这篇文章中，我们引入了 ZStack 自动化测试系统的第一部分——集成测试。通过它，开发人员可以以 100%的信心写代码。而且写测试用例也不再是一个令人气馁和无聊的任务；开发人与可以以少于 100 行的代码来完成大多数的用例，这是非常容易和有效率的。

ZSTACK——自动化测试系统 2：系统测试

ZStack 的系统测试系统在真实的硬件环境中运行测试用例；像集成测试一样，这个系统测试也是全自动的，而且覆盖的层面包括：功能性测试、压力测试、性能测试。

概述

虽然集成测试系统，如我们在 ZStack—自动化测试系统 1：集成测试中所介绍的，强大到可以暴露开发过程中大多数的缺陷，也是有着固有的弱点的。首先，由于测试用例使用模拟器，它们不能测试真实场景，比如在一个物理的 KVM 主机上创建一个 VM。第二，集成测试用例主要关注一个简单的场景，在一个简单的人造的环境中；举个例子，还是创建 VM 的这个用例，它可能只部署一个最小的环境，包括一个主机和一个 L3 网络，仅仅用于满足创建一个 VM 的需求。这些弱点，然而也是深思熟虑过的，因为我们想要开发人员能够在他们开发新特性时快速和容易地写测试用例，这是一个我们必须采取的权衡。

系统测试，目标在于测试整个软件，在一个真实的、复杂的环境中，很自然地补充集成测试。ZStack 的系统测试系统被设计用于以下两个目标：

1. 复杂的场景：这些场景应该比真实世界的使用场景更复杂，以测试软件的极限。举个例子，挂载和卸载磁盘的测试用例应该持续地、重复地对虚拟机执行，以一种非常快，人类无法手动做到的方式。

2. 易于编写和维护测试用例：就像集成测试系统，系统测试系统接管了大多数无聊重复的任务，让测试人员有效率地写测试用例。

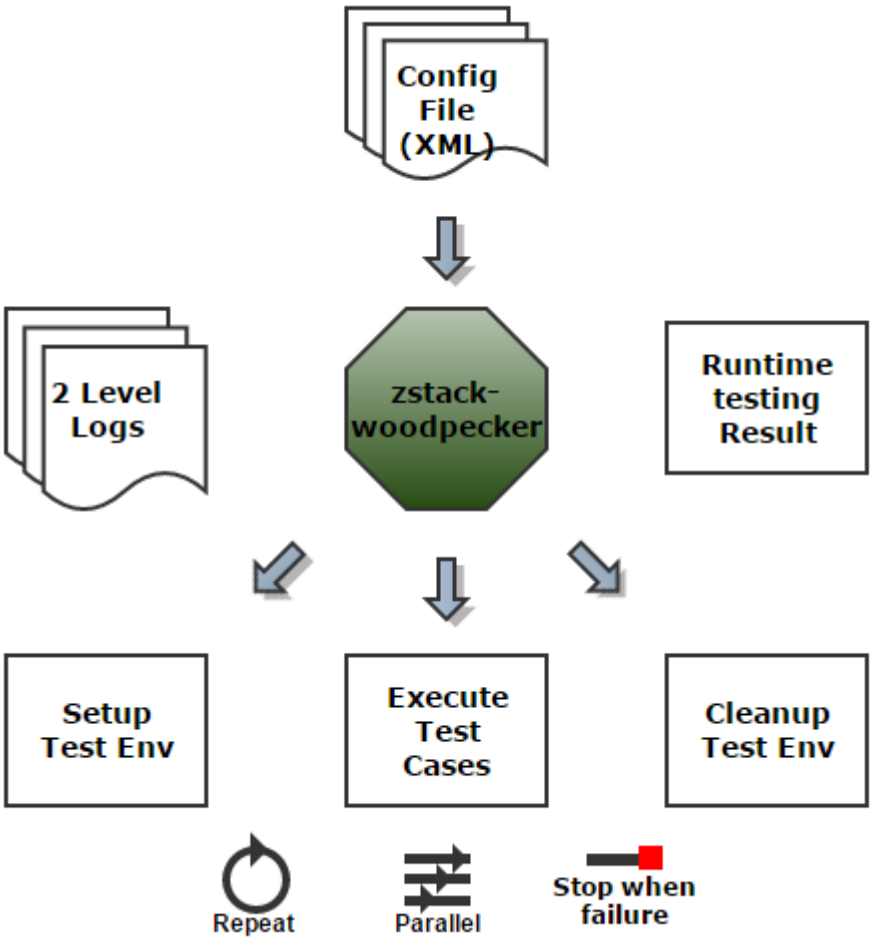
这个系统测试系统是一个 Python 项目，命名为 zstack-woodpecker，由以下三个部分组成：

1. 测试框架：一个测试框架，管理所有的测试用例，以及提供必须的库和工具。

2. 环境部署工具：一个工具，用于从 XML 配置文件部署一个环境；它非常类似于集成测试系统的部署器。

3. 模块化测试用例：测试用例是高度模块化的，而且覆盖了：功能测试、性能测试和压力测试。

系统测试



zstack-woodpecker 完全由我们自己创建；在决定重新造这个轮子之前，我们试过了流行的 Python 测试框架，像 nose，然后最终选择了创造一个新的工具，用以最大化地满足我们的目标。

套件配置

类似所有的其他测试框架，一个 zstack-woodpecker 中的测试套件是以 suite setup 开始，以 suite teardown 结束，在其中有一些测试用例。这里的 suite setup 和 suite teardown 是两个特殊的测试用例，suite setup 负责准备后续的测试用例所需的环境，

suite teardown 负责在所有测试用例结束之后清理这个环境。一个典型的测试套件配置文件看起来像：

```
<integrationTest>
    <suite name="basic test" setupCase="suite_setup.py" teardownCase="suite_teardown.py" parallel="8">
        <case timeout="120" repeat="10">test_create_vm.py</case>
        <case timeout="220">test_reboot_vm.py</case>
        <case timeout="200">test_add_volume.py</case>
        <case timeout="200">test_add_volume_reboot_vm.py</case>
        <case timeout="400">test_add_multi_volumes.py</case>
        <case timeout='600' repeat='2' noparallel='True'>resource/test_delete_l2.py</case>
    </suite>
</integrationTest>
```

敏锐的读者可能会注意到一些参数是在其他的测试框架中看不到的。

第一个是 timeout；每一个测试用例可以定义自己的超时时间，如果在这段时间内不能完成，它将被在最终的结果里被标记成超时。

第二个是 repeat，允许你在测试套件中指定这个用例应该被执行多少次。

第三个，也是杀手级的参数是 parallel，允许测试人员设定这个套件的并行级别；这是一个使得 zstack-woodpecker 运行测试用例非常快的关键特性；在上面这个例子中，parallel 被设置成 8，这意味着将有至多 8 个用例在同时运行；这不只是加速运行测试用例，也创造了一个复杂的场景，模拟许多用户在共享同一个环境时执行不同的任务。然而，不是所有的用例都可以被同时执行；在我们的例子中，用例 test_delete_l2.py 将会删除被其他用例依赖的 L2 网络，所以在其他用例执行时，它不能被执行；这就是第四个参数 noparallel 发挥作用的地方；一旦它被设置成 true，这个用例将会单独被执行，不会有其他用例可以同时运行。

命令行工具

zstest.py 是一个命令行工具，用于帮助测试人员控制测试框架，执行任务，像启动测试套件，列出测试用例，等等。zstest.py 提供了丰富的选项帮助测试人员简化他们的工作。这些选项中的一些，用于在我们的日常测试中，特别有用，列在了下面。

测试人员可以通过选项-l 获取可用的测试用例，例如：./zstest.py -l

它将会展示如下的结果：

```
[root@centos61 dailytest]# ./zstest.py -l -s basic

 - Available ZStack Integration Test Cases:

================================================================================
  No.   |       Test Case
--------------------------------------------------------------------------------
  [1]   | basic/suite_setup.py
  [2]   | basic/suite_teardown.py
  [3]   | basic/test_add_multi_volumes.py
  [4]   | basic/test_add_vol_to_stopvm.py
  [5]   | basic/test_add_volume_negative.py
--------------------------------------------------------------------------------
  [6]   | basic/test_add_volume.py
  [7]   | basic/test_add_volume_reboot_vm.py
  [8]   | basic/test_create_vm.py
  [9]   | basic/test_crt_temp_from_volume.py
  [10]  | basic/test_crt_vm_with_volume.py
--------------------------------------------------------------------------------
  [11]  | basic/test_reboot_vm.py
  [12]  | basic/test_start_vm.py
  [13]  | basic/test_stop_vm.py
  [14]  | basic/test_vm_securitygroup.py
  [15]  | basic/zstack_restart.py
================================================================================

 - List ZStack Integration Test Case Completed
```

测试套件名，是测试用例的第一级文件夹的名称；例如，在上图中你看到了大量的用例以 basic 开头（例如：basic/test_reboot_vm.py），是的，basic 就是这个测试套件的名字。测试人员可以通过选项-s 启动一个套件，使用套件名的全称或者部分都行，只要它是独一无二的，例如：./zstest.py -s basic 或 ./zstest.py -s ba

```
[root@centos61 dailytest]# ./zstest.py -s basic
export woodpecker_http_proxy=$http_proxy; export woodpecker_https_proxy=$https_p
roxy; unset http_proxy; unset https_proxy; zstack-woodpecker -f /root/zstack-woo
dpecker/dailytest/config_xml/test_suite.xml
Woodpecker is correctly installed. Test begins ...

[Begin suite parsing]

discovering test cases in /root/zstack-woodpecker/dailytest/config_xml/test_suit
e.xml ...
discovering test cases in /root/zstack-woodpecker/dailytest/config_xml/basic_int
egration.xml ...
        Suite [basic_test] will run [1] times:
                Run [1] times for Case: [basic/test_create_vm]
                Run [1] times for Case: [basic/test_stop_vm]
                Run [1] times for Case: [basic/test_start_vm]
                Run [1] times for Case: [basic/test_reboot_vm]
                Run [1] times for Case: [basic/test_add_volume]
                Run [1] times for Case: [basic/test_add_vol_to_stopvm]
                Run [1] times for Case: [basic/test_add_volume_reboot_vm]
                Run [1] times for Case: [basic/test_add_multi_volumes]
                Run [1] times for Case: [basic/test_add_volume_negative]
                Run [1] times for Case: [basic/test_crt_temp_from_volume]
                Run [1] times for Case: [basic/test_crt_vm_with_volume]
                Run [1] times for Case: [basic/test_vm_securitygroup]

[Suite parsing finished]: 1 test suites discovered, total 14 cases

[Test Begin]

Test suite: [basic_test], 14 cases, 2 execution threads, repeat 1 times:
suite_setup                    ..........................  [ success 0:03:19 ]
basic/test_create_vm           ..........................  [ success 0:00:13 ]
basic/test_stop_vm             ..........................  [ success 0:00:18 ]
basic/test_reboot_vm           .....                          [ 0:00:04 ]
```

测试人员也可以选择性地执行测试用例，通过使用它们的名字或者 ID，已经选项-c；例如：./zstest.py -c 1,6 或 ./zstest.py -c suite_setup,test_add_volume.py

记住，你需要运行 suite setup 的用例：suite_setup.py 作为第一个用例，除非你已经这么做了。

由于一个测试套件将会执行所有的测试用例，清理环境，发出一个结果报告，测试人员有时可能想要停止测试套件，并在一个用例失败时保持环境，这样他们就可以深入查看失败结果并调试；选项-n 和-S 就是为此准备的；-n 指示测试框架不要清理环境，-S 要求跳过没有被执行的用例；例如：./zstest.py -s virtualrouter -n -S

另外，选项-b 可以拉取最新的源代码并构建一个全新的 zstack.war，这在 Nightly 测试中特别有用，这种测试被假定为测试最新的代码：./zstest.py -s virtualrouter -b

一旦所有的测试用例完成，一个报告将会被生成并被打印到屏幕上：

```
Test Summary:
================================================================================
 Test Case                                      Pass    Fail    TMO     Skip
--------------------------------------------------------------------------------
 basic_test:
    suite_setup                                 1       0       0       0
    basic/test_create_vm                        1       0       0       0
    basic/test_stop_vm                          1       0       0       0
    basic/test_start_vm                         1       0       0       0
    basic/test_reboot_vm                        1       0       0       0
    basic/test_add_volume                       1       0       0       0
    basic/test_add_vol_to_stopvm                1       0       0       0
    basic/test_add_volume_reboot_vm             1       0       0       0
    basic/test_add_multi_volumes                1       0       0       0
    basic/test_add_volume_negative              1       0       0       0
    basic/test_crt_temp_from_volume             1       0       0       0
    basic/test_crt_vm_with_volume               1       0       0       0
    basic/test_vm_securitygroup                 1       0       0       0
    suite_teardown                              1       0       0       0
--------------------------------------------------------------------------------
 Total: 14                                      14      0       0       0
================================================================================

The detailed test results are in /root/zstack-woodpecker/dailytest/config_xml/te
st-result/151231-064358

Total test time: 00:06:06.849 (366.84996295)


 - ZStack Auto Test Completed
```

测试框架将会保存所有的日志，并直接输出每一个失败日志的绝对路径，如果存在的话。为了在一般的日志中记录更多的细节，有一种特殊的日志 action log，用于记录每一个 API 调用；因为这是一个完全纯粹关于 API 的日志，我们可以容易地找到一个失败的根本来源，而不用被测试框架的日志分散注意力。另外，它是一种重要的工具，可以自动地生成一个新的用例用于重现失败，这是一个我们所使用的魔法武器，用于在基于模型的测试（每个用例都随机地执行各种 API）中调试失败。你可以在 ZStack--自动化测试系统 3：基于模型的测试中找到细节。Action log 的片段如下：

```
  <<Action>> Create VM: test_vm_default_name with [image:] 07f7c252cdc14c4da5d3590f293b5852 and [l3_network:] ['6c7af862b03
44e32840e998aa5b49d52'] [14:56:28]

     <Log> [vm:] 2f5b5956418b46b2a017fcf5e7d31823 is created. [14:56:29]
     <Log> Add checker for [ZstackTestVm:] 2f5b5956418b46b2a017fcf5e7d31823. Checkers are: CheckerChain: [zstack_kvm_vm_set
_host_vlan_ip] [zstack_vm_db_checker] [zstack_kvm_vm_running_checker] [14:56:29]
     <Log> L2: 89e46a02c6c74570a32304a7a794edc2 did not have vlan.  [14:56:30]
     <Log> eth2 might be manually created vlan dev. [14:56:30]

    !!WARN!! L3 name: public l3 network uuid: 6c7af862b0344e32840e998aa5b49d52 network [mask:] 255.255.255.0 is not 255.255
.0.0 . Will not assign IP to host. Please change test configuration to make sure L3 network mask is 255.255.0.0. [14:56:30]
     <Log> Checker: [zstack_kvm_vm_set_host_vlan_ip] PASS. Expected result: True. Test result: True. [14:56:30]
     <Log> Checker: [zstack_vm_db_checker] begins. [14:56:30]
     <Log> Checker: [zstack_vm_db_checker] PASS. Expected result: True. Test result: True. [14:56:30]
     <Log> Checker: [zstack_kvm_vm_running_checker] begins. [14:56:30]
     <Log> Testagent is running on Host: 192.168.0.202 . Skip testagent installation. [14:56:30]
     <Log> L2: 89e46a02c6c74570a32304a7a794edc2 did not have vlan.  [14:56:30]
     <Log> eth2 might be manually created vlan dev. [14:56:30]

    !!WARN!! L3 name: public l3 network uuid: 6c7af862b0344e32840e998aa5b49d52 network [mask:] 255.255.255.0 is not 255.255
.0.0 . Will not assign IP to host. Please change test configuration to make sure L3 network mask is 255.255.0.0. [14:56:30]
     <Log> Check [vm:] 2f5b5956418b46b2a017fcf5e7d31823 running status on host [name:] 192.168.0.202 [uuid:] 546cf42092e249
428f0ebe65dbe0c625. [14:56:30]
     <Log> Check result: [vm:] 2f5b5956418b46b2a017fcf5e7d31823 is RUNNING on [host:] 192.168.0.202 . [14:56:30]
     <Log> Checker: [zstack_kvm_vm_running_checker] PASS. Expected result: True. Test result: True. [14:56:30]

  <<Action>> Create [Volume:] test_volume with [disk offering:] dbb8f3065efc463eaeac2bc0716d9234  [14:56:31]

     <Log> [volume:] 7db8d70d2eea44a8aec0235933cfedad is created. [14:56:31]

  <<Action>> Create [Volume:] test_volume with [disk offering:] dbb8f3065efc463eaeac2bc0716d9234  [14:56:31]

     <Log> [volume:] ada7968f3c7a4c0c90be47b23d0487d5 is created. [14:56:32]

  <<Action>> Create [Volume:] test_volume with [disk offering:] dbb8f3065efc463eaeac2bc0716d9234  [14:56:32]

     <Log> [volume:] 59e3d0e236ba49ce8d8ed0950d8449a7 is created. [14:56:32]

  <<Action>> Create [Volume:] test_volume with [disk offering:] dbb8f3065efc463eaeac2bc0716d9234  [14:56:33]

     <Log> [volume:] 11d2f2d9bf8c49c7ba5c744d79887b51 is created. [14:56:33]
```

环境部署工具

类似于集成测试，对每一个测试用例来说，准备环境是频繁且重复的任务；例如，用于测试创建虚拟机的用例需要去配置独立的资源，像 zone，cluster，host 等等。Zstack-woodpecker 调用 zstack-cli，这个 ZStack 的命令行工具去从一个 XML 配置文件部署测试环境。例如：zstack-cli -d zstack-env.xml

这里的 XML 配置文件的格式类似于集成测试所用的，一个片段看起来像这样：

```
...
  <zones>
    <zone name="$zoneName" description="Test">
      <clusters>
        <cluster name="$clusterName" description="Test"
          hypervisorType="$clusterHypervisorType">
          <hosts>
            <host name="$hostName" description="Test" managementIp="$hostIp"
              username="$hostUsername" password="$hostPassword" />
          </hosts>
          <primaryStorageRef>$nfsPrimaryStorageName</primaryStorageRef>
          <l2NetworkRef>$l2PublicNetworkName</l2NetworkRef>
          <l2NetworkRef>$l2ManagementNetworkName</l2NetworkRef>
          <l2NetworkRef>$l2NoVlanNetworkName1</l2NetworkRef>
          <l2NetworkRef>$l2NoVlanNetworkName2</l2NetworkRef>
          <l2NetworkRef>$l2VlanNetworkName1</l2NetworkRef>
          <l2NetworkRef>$l2VlanNetworkName2</l2NetworkRef>
        </cluster>
      </clusters>
...
      <l2Networks>
        <l2VlanNetwork name="$l2VlanNetworkName1" description="guest l2 vlan network"
          physicalInterface="$l2NetworkPhysicalInterface" vlan="$l2Vlan1">
          <l3Networks>
            <l3BasicNetwork name="$l3VlanNetworkName1" description = "guest test vlan network with DHCP DNS SNAT PortForwarding
 EIP and SecurityGroup" domain_name="$L3VlanNetworkDomainName1">
              <ipRange name="$vlanIpRangeName1" startIp="$vlanIpRangeStart1" endIp="$vlanIpRangeEnd1"
               gateway="$vlanIpRangeGateway1" netmask="$vlanIpRangeNetmask1"/>
              <dns>$DNSServer</dns>
              <networkService provider="VirtualRouter">
                <serviceType>DHCP</serviceType>
                <serviceType>DNS</serviceType>
                <serviceType>SNAT</serviceType>
                <serviceType>PortForwarding</serviceType>
                <serviceType>Eip</serviceType>
              </networkService>
              <networkService provider="SecurityGroup">
                <serviceType>SecurityGroup</serviceType>
              </networkService>
            </l3BasicNetwork>
          </l3Networks>
        </l2VlanNetwork>
...
```

部署工具通常在运行任何用例前被 suite setup 调用，测试人员可以在 XML 配置文件中通过以$符号开头来定义变量，然后在一个独立的配置文件中解析。通过这种方式，这个 XML 配置文件像模板一个工作，可以产生不同的环境。配置文件的例子如下：

```
TEST_ROOT=/usr/local/zstack/root/
zstackPath = $TEST_ROOT/sanitytest/zstack.war
apachePath = $TEST_ROOT/apache-tomcat
zstackPropertiesPath = $TEST_ROOT/sanitytest/conf/zstack.properties
zstackTestAgentPkgPath = $TEST_ROOT/sanitytest/zstacktestagent.tar.gz
masterName = 192.168.0.201
DBUserName = root

node2Name = centos5
node2Ip = 192.168.0.209
node2UserName = root
node2Password = password

node1Name = 192.168.0.201
node1Ip = 192.168.0.201
node1UserName = root
node1Password = password


instanceOfferingName_s = small-vm
instanceOfferingMemory_s = 128M
instanceOfferingCpuNum_s = 1
instanceOfferingCpuSpeed_s = 512

virtualRouterOfferingName_s = virtual-router-vm
virtualRouterOfferingMemory_s = 512M
virtualRouterOfferingCpuNum_s = 2
virtualRouterOfferingCpuSpeed_s = 512

sftpBackupStorageName = sftp
sftpBackupStorageUrl = /export/backupStorage/sftp/
sftpBackupStorageUsername = root
sftpBackupStoragePassword = password
sftpBackupStorageHostname = 192.168.0.220
```

注意:正如你可能会猜测的，这个工具可以被管理员用于从一个 XML 配置文件部署一个云环境；更进一步，管理员们做相反的事情，将一个云环境写入到一个 XML 配置文件，通过 zstack-cli -D xml-file-name.

对于性能和压力测试，环境通常需要大量的资源，例如 100 个 zone，1000 个 cluster。为了避免手动在配置文件中重复 1000 行，我们引入了一个属性 duplication，用于帮助创建重复的资源。例如：

```
...
    <zones>
      <zone name="$zoneName" description="10 same zones" duplication="100">
        <clusters>
          <cluster name="$clusterName_sim" description="10 same Simulator Clusters" duplication="10"
            hypervisorType="$clusterSimHypervisorType">
            <hosts>
              <host name="$hostName_sim" description="100 same simulator Test Host"
                managementIp="$hostIp_sim"
                cpuCapacity="$cpuCapacity" memoryCapacity="$memoryCapacity"
                duplication="100"/>
            </hosts>
            <primaryStorageRef>$simulatorPrimaryStorageName</primaryStorageRef>
            <l2NetworkRef>$l2PublicNetworkName</l2NetworkRef>
            <l2NetworkRef>$l2ManagementNetworkName</l2NetworkRef>
            <l2NetworkRef>$l2VlanNetworkName1</l2NetworkRef>
          </cluster>
...
```

备注：这段不翻译了。

模块化的测试用例

在系统测试中测试用例可以被高度模块化。每一个用例本质上执行以下三步：

1. 创建要被测试的资源
2. 验证结果
3. 清理环境

Zstack-woodpecker 本身提供一个完整的库用于帮助测试人员调度这些活动。API 也很好地被封装在一个，从 zstack 源代码自动生成的库中。测试人员不需要去写任何的原生的 API 调用。检查器，用于验证测试结果，也已为每一个资源创建；例如，VM 检查器，云盘检查器。测试人员可以很容易地调用这些检查器去验证他们创建的资源，而不需写成顿成吨的代码。如果当前检查器不能满足某些场景，测试人员也能创建自己的检查器，并作为插件放入测试框架。

一段测试用例看起来像：

```
def test():
    test_util.test_dsc('Create test vm and check')
    vm = test_stub.create_vlan_vm()
    test_util.test_dsc('Create volume and check')
    disk_offering = test_lib.lib_get_disk_offering_by_name(os.environ.get('rootDiskOfferingName'))
    volume_creation_option = test_util.VolumeOption()
    volume_creation_option.set_disk_offering_uuid(disk_offering.uuid)
    volume = test_stub.create_volume(volume_creation_option)
    volume.check()
    vm.check()
    test_util.test_dsc('Attach volume and check')
    volume.attach(vm)
    volume.check()
    test_util.test_dsc('Detach volume and check')
    volume.detach()
    volume.check()
    test_util.test_dsc('Delete volume and check')
    volume.delete()
    volume.check()
    vm.destroy()
    vm.check()
    test_util.test_pass('Create Data Volume for VM Test Success')
```

像集成测试一样，测试人员可以仅以十几行便写出一个测试用例。模块化不只是帮助简化测试用例的编写，也为基于模型的测试构建了一个坚实的基础，下篇文章我们会详细讨论。

总结

在这篇文章中，我们引入了我们的系统测试系统。通过执行比现实世界的用例更复杂的测试，系统测试可以给我们更多的自信，关于 ZStack 在真实的硬件环境中的表现。使得我们可以快速进化成一个成熟的产品。

ZSTACK——自动化测试系统 3：基于模型的测试

模型测试系统是 zstack-woodpecker 中的一个子项目。通过有限状态机和行为选择策略，它可以产生随机的 API 操作，一直运行下去，直到遇到一个缺陷或者预定义的退出条件。ZStack 依赖模型测试去测试真实世界中难以遇到的边界用例，在测试覆盖度方面补充集成测试和系统测试。

概述

测试覆盖率是一个判断一个测试系统品质的重要指示器。常规测试方法论，例如单元测试，集成测试，系统测试，都是由人类逻辑思考构建的，难以覆盖软件中的边界场景。这个问题在 IaaS 软件中变得更加明显，因为管理不同的子系统会导致极为复杂的场景。

ZStack 通过引入基于模型的测试来解决这个问题。它可以产生由随机 API 组合构成的场景，会持续运行知道遇到预定义的退出条件或者找到一个缺陷。作为机器驱动的测试，基于模型的测试可以克服人类逻辑思考的缺陷来执行一些，看起来反人类逻辑，但是 API 完全正确的测试，帮助发现难以被人类主导的测试发现的边界问题。

一个例子可以帮助理解这个思想。基于模型的测试系统通常在执行大约 200 个 API 后暴露一个 bug，在调试后，我们找到最小重现这个问题的序列是：

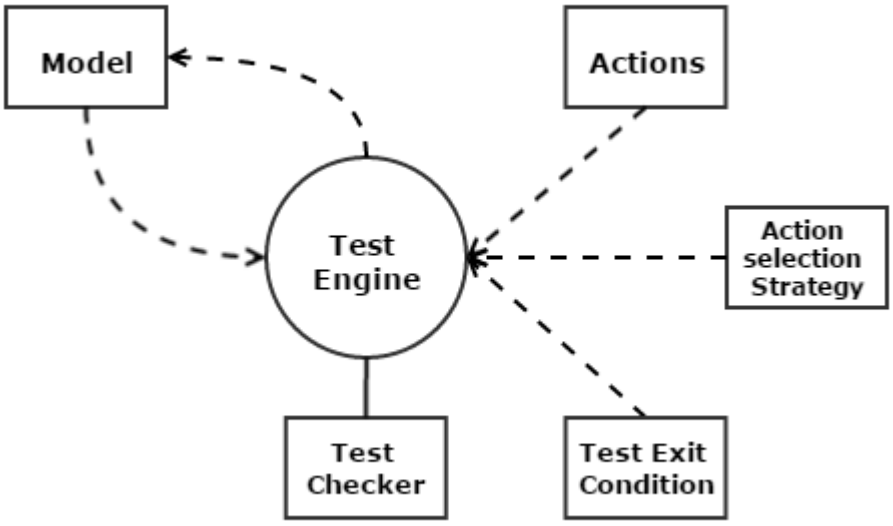
1. 创建一个 VM
2. 关闭这个 VM
3. 为这个 VM 的根云盘创建一个云盘快照
4. 从这个 VM 的根云盘创建一个新的数据云盘快照
5. 销毁这个 VM

6. 创建一个新的数据云盘，使用 4 中的模板
7. 从 6 中的数据云盘创建一个新的云盘快照

这个操作序列显然是反逻辑的，我们相信没有测试者会写一个集成测试用例或者系统测试用例这么做。这就是机器思考闪光的地方，因为它没有人类的感情，会做人类感觉不合理的事情。在找到这个 bug 之后，我们生成了一个回归测试为以后保障这个问题。

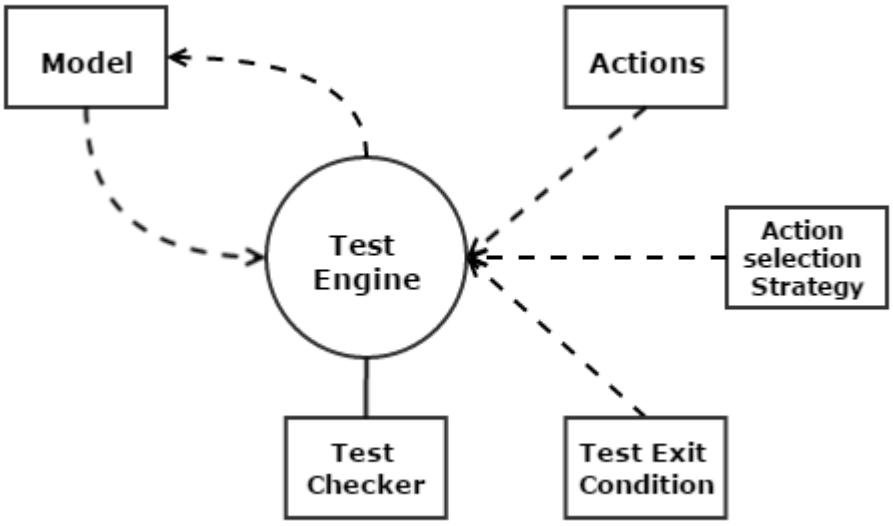
基于模型的测试系统

基于模型的测试系统，因为由机器驱动，也被称为机器人测试。当这个系统运行时，它从一个模型（在下面几节中也被称为阶段）移动到另一个模型，通过执行被动作选择策略选出的动作（也被称为操作）。在每一个模型完成之后，检查器将会验证测试结果，测试退出条件。如果任何失败被发现，或者退出条件被满足，系统将会退出。否则，它将会移动到下一个模型，然后重复。



有限状态机

在基于模型的测试的理论中，有许许多多生成测试操作的方式。例如：有限状态机，自动推导，模型检验。我们选择使用有限状态机，因为它自然地适合 IaaS 软件，其中每一个资源都由状态驱动。例如，从用户角度看，VM 的状态像这样：



在基于模型的测试系统中，每一个资源的每一个状态都预先定义在 test_state.py 中，看起来像：

```
vm_state_dict = {
            Any: 1 ,
            vm_header.RUNNING: 2,
            vm_header.STOPPED: 3,
            vm_header.DESTROYED: 4
            }

    vm_volume_state_dict = {
            Any: 10,
            vm_no_volume_att: 20,
            vm_volume_att_not_full: 30,
            vm_volume_att_full: 40
            }

    volume_state_dict = {
            Any: 100,
            free_volume: 200,
            no_free_volume:300
            }

    image_state_dict = {
            Any: 1000,
            no_new_template_image: 2000,
            new_template_image: 3000
            }
```

系统中所有资源的所有状态构成一个阶段（模型），系统可以从一个阶段转移到下一个阶段，通过执行维护在转换表中操作。一个阶段被定义成类似这样：

```
class TestStage(object):
    '''
        Test states definition and Test state transition matrix.
    '''
    def __init__(self):
        self.vm_current_state = 0
        self.vm_volume_current_state = 0
        self.volume_current_state = 0
        self.image_current_state = 0
        self.sg_current_state = 0
        self.vip_current_state = 0
        self.sp_current_state = 0
        self.snapshot_live_cap = 0
        self.volume_vm_current_state = 0
...
```

一个阶段可以被表示成一个整数，即由这个阶段的所有状态的和。通过这个整数，我们可以在转换表中查找到下一个后选的操作。转换表的一个例子如下：

```
#state transition table for vm_state, volume_state and image_state
    normal_action_transition_table = {
        Any: [ta.create_vm, ta.create_volume, ta.idel],
        2: [ta.stop_vm, ta.reboot_vm, ta.destroy_vm, ta.migrate_vm],
        3: [ta.start_vm, ta.destroy_vm, ta.create_image_from_volume, ta.create_data_vol
_template_from_volume],
        4: [],
      211: [ta.delete_volume],
      222: [ta.attach_volume, ta.delete_volume],
      223: [ta.attach_volume, ta.delete_volume],
      224: [ta.delete_volume],
      232: [ta.attach_volume, ta.detach_volume, ta.delete_volume],
      233: [ta.attach_volume, ta.detach_volume, ta.delete_volume],
      234: [ta.delete_volume], 244: [ta.delete_volume], 321: [],
      332: [ta.detach_volume, ta.delete_volume],
      333: [ta.detach_volume, ta.delete_volume], 334: [],
      342: [ta.detach_volume, ta.delete_volume],
      343: [ta.detach_volume, ta.delete_volume], 344: [],
     3000: [ta.delete_image, ta.create_data_volume_from_image]
    }
```

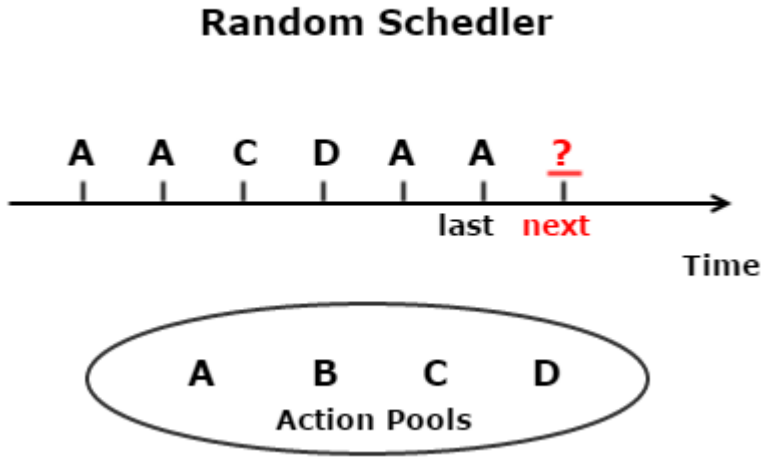
通过这种方式，基于模型的测试系统可以保持运行，从一个阶段到另一个阶段，直到遇到预先定义的退出条件或者发现一些缺陷，它可以持续地跑很多天，数以万次地调用API。

动作选择策略

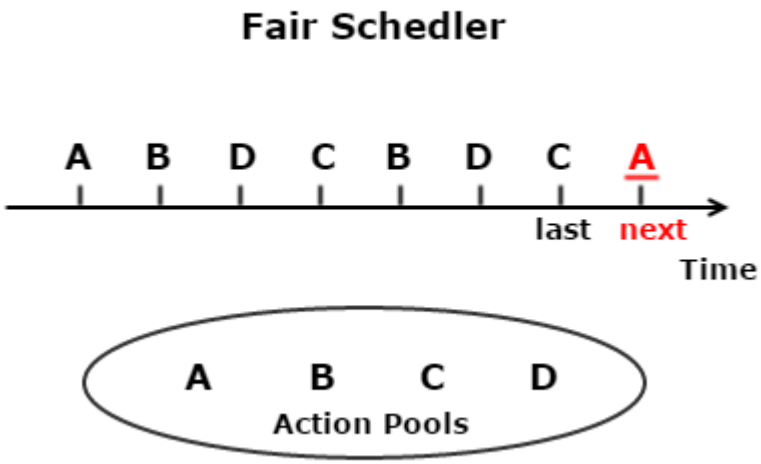
当在阶段间移动时，基于模型的测试系统需要决定下一个需要执行的操作是什么。决定制定器被称为动作选择策略，一个可扩展插件的引擎，不同的选择算法可以以不同的目的被实现。

当前系统有三种策略：

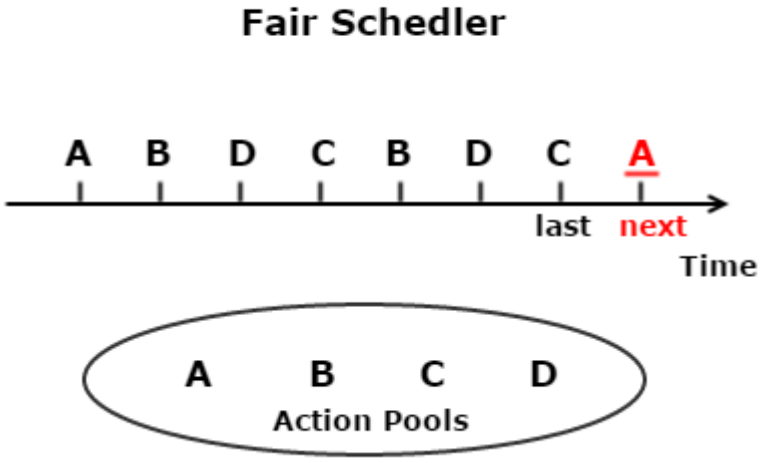
随机调度器：最简单的策略，为当前的阶段，从候选动作中随机地选择下一个操作。作为一种很直接的算法，随机调度器可能会重复一项操作，而使得其他操作等待。为了缓解这个问题，我们为每一个操作都增加了一个权重，这样测试人员可以为他们想多测试的操作赋予更高的权重。



公平调度器：一种对待每个操作都完全平等的策略，以这样一种方式补充随机调度器：每个操作都有平等的机会被执行，保证只要测试周期足够长，每个操作都会被测试到.



路径覆盖调度器：通过历史数据决定下一步操作的策略。这个策略会记住已经被执行过的所有操作路径，然后尝试选择一个可以形成新的操作路径的操作。例如，给定候选操作 A,B,C,D，如果前一个操作时 B 且路径 BA,BB,BC 都已经被执行，策略将会选取 D 作为下一个操作，这样路径 BD 将会被测试到。



如上面提及到的，动作选择策略是一个可扩展插件的引擎，每一个策略实际上由类 ActionSelector 派生来：

```
class ActionSelector(object):
    def __init__(self, action_list, history_actions, priority_actions):
        self.history_actions = history_actions
        self.action_list = action_list
        self.priority_actions = priority_actions

    def select(self):
        '''
        New Action Selector need to implement own select() function.
        '''
        pass

    def get_action_list(self):
        return self.action_list

    def get_priority_actions(self):
        return self.priority_actions

    def get_history_actions(self):
        return self.history_actions
```

一个随机调度器的实现例子像这样：

```
class RandomActionSelector(ActionSelector):
    '''
    Base on the priority action list, just randomly pickup action.

    If need to set higher priority for some action, it just needs to put them
    more times in priority_actions list.
    '''
    def __init__(self, action_list, history_actions, priority_actions):
        super(RandomActionSelector, self).__init__(action_list, \
                history_actions, priority_actions)

    def select(self):
        priority_actions = self.priority_actions.get_priority_action_list()
        for action in priority_actions:
            if action in self.get_action_list():
                self.action_list.append(action)

        return random.choice(self.get_action_list())
```

退出条件

在启动基于模型的测试系统之前，退出条件必须被设定好，否则系统将会保持运行，直到一个缺陷被发现，或者日志文件撑爆了测试机器的硬盘。退出条件可以是任何形式的，例如，在运行 24 小时后退出，在系统有 100 个 EIP 被创建后退出，在有 2 个停止的 VM、8 个运行中的 VM 时退出。一切都取决于测试者去定义条件，尽可能地增加发现缺陷的机会。

失败回放

调试一个被基于模型的测试系统发现的失败是很难而且令人沮丧的，大多数的失败都由大量的操作序列暴露，而且它们通常缺乏逻辑并有着大量的日志。我们通常手动重现失败，在痛苦地依照大约 500，000 行日志,使用 zstack-cli 调用 API 200 次后，我们最终意识到这个悲惨的任务不是人类可以做到的。然后我们发明了一个工具用于重现一个失败，通过回放动作日志（纯粹只记录了关于 API 的测试信息）。

一个动作日志像这样：

```
Robot Action: create_vm
 Robot Action Result: create_vm; new VM: fc2c0221be72423ea303a522fd6570e9
 Robot Action: stop_vm; on VM: fc2c0221be72423ea303a522fd6570e9
 Robot Action: create_volume_snapshot; on Root Volume: fe839dcb305f471a852a1f5e21d4fed
a; on VM: fc2c0221be72423ea303a522fd6570e9
 Robot Action Result: create_volume_snapshot; new SP: 497ac6abaf984f5a825ae4fb2c585a88
 Robot Action: create_data_volume_template_from_volume; on Volume: fe839dcb305f471a852a
1f5e21d4feda;  on VM: fc2c0221be72423ea303a522fd6570e9
 Robot Action Result: create_data_volume_template_from_volume; new DataVolume Image: fb
23cdfce4b54072847a3cfe8ae45d35
 Robot Action: destroy_vm; on VM: fc2c0221be72423ea303a522fd6570e9
 Robot Action: create_data_volume_from_image; on Image: fb23cdfce4b54072847a3cfe8ae45d3
5
 Robot Action Result: create_data_volume_from_image; new Volume: 20dee895d68b428a88e5ec
3d3ef634d8
 Robot Action: create_volume_snapshot; on Volume: 20dee895d68b428a88e5ec3d3ef634d8
```

测试人员可以通过调用回放工具重建失败的环境：

```
robot_replay.py -f path_to_action_log
```

总结

在这篇文章中，我们引入了基于模型的测试系统。由于善于暴露边界用例中的问题，基于模型的测试系统和集成测试系统、系统测试系统共同作为保卫 ZStack 质量的基础，使得我们可以以骄傲的自信发布产品。