ZStack 技术白皮书精选

自动化测试系统 1：集成测试

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——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_PAT
H, 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。

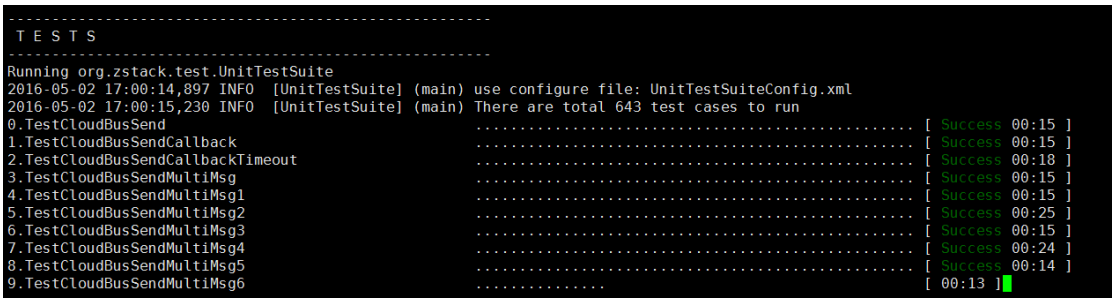
测试框架

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。