ZStack 技术白皮书精选

自动化测试系统 3：基于模型的测试

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 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：

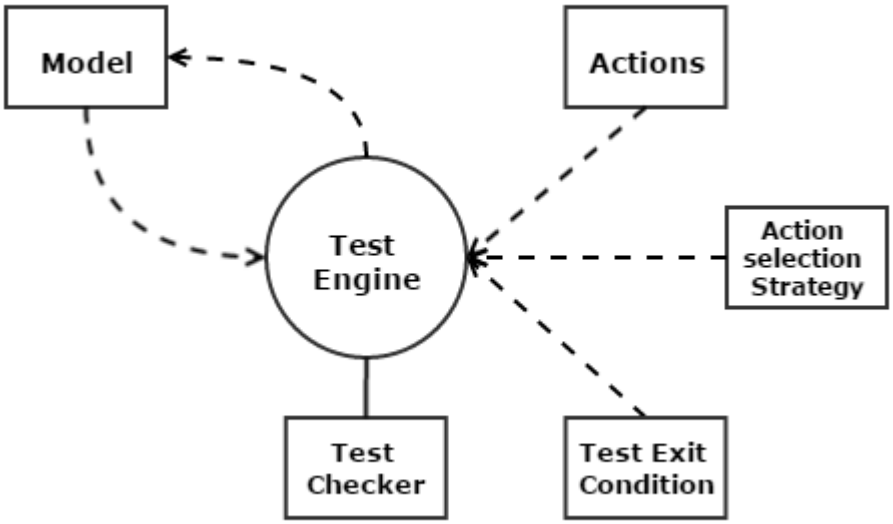
1. 创建一个 VM

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。

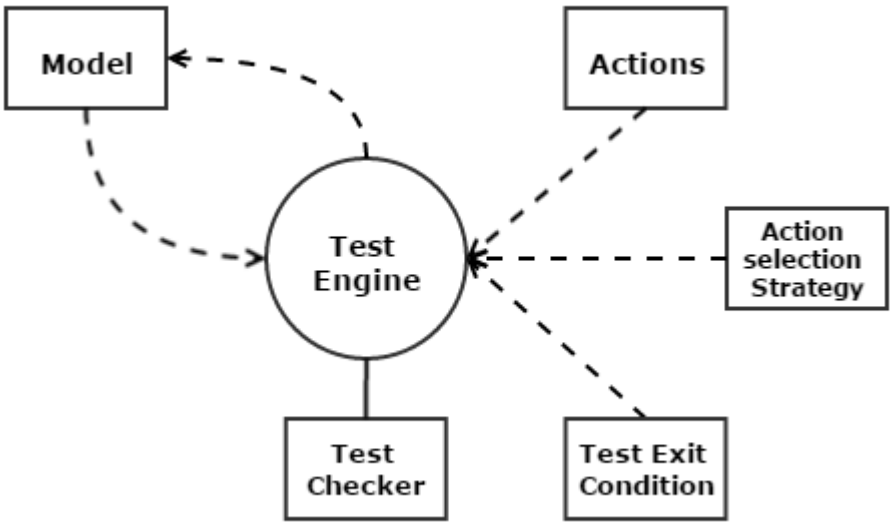
基于模型的测试系统

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]
    }
```

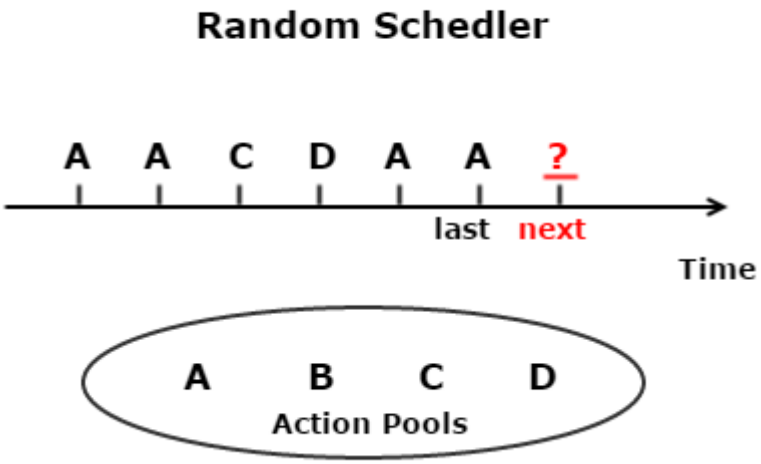
通过这种方式，基于模型的测试系统可以保持运行，从一个阶段到另一个阶段，直到遇到预先定义的退出条件或者发现一些缺陷，它可以持续地跑很多天，数以万次地调用 API。

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。

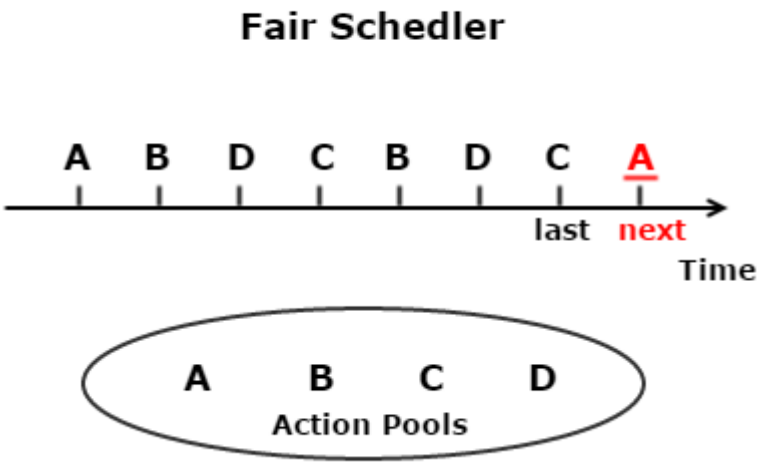
当前系统有三种策略：

- 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。

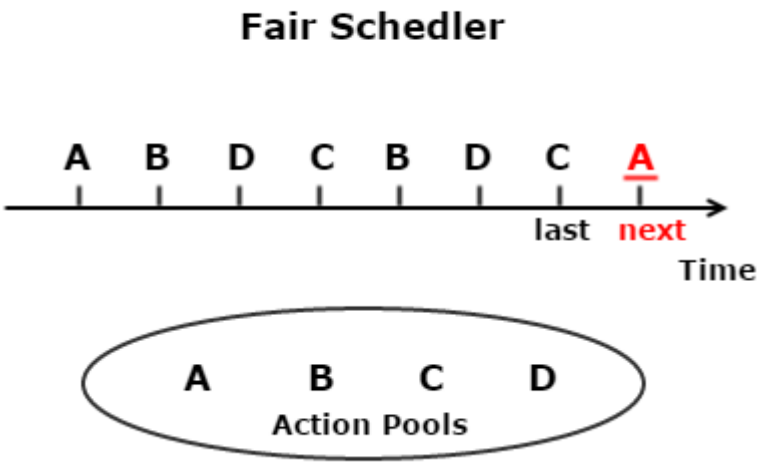


- 公平调度器：一种对待每个操作都完全平等的策略，以这样一种方式补充随机调度器：每个操作都有平等的机会被执行，保证只

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。