ZStack 技术白皮书精选

架构篇：下册

标签系统、级联框架、查询 API

扫一扫二维码，获取更多技术干货吧

版权声明

目录

ZSTACK--标签系统

ZStack 中的标签不仅帮助用户聚集资源，也帮助控制软件行为。ZStack 有一套完整的规范，用以定义标签的类别、形式和用法。除了用户外，插件也可以创建自己的标签，以记录元数据和拓展现有的资源属性；通过这些手段，标签可以帮助插件引入新的特性，而不改变 ZStack 的数据库结构，消除了在软件升级对数据库迁移的需求。

动机

随着云中资源的不断增长，用户可能会想要有一种方式，使用人类可读的标签，去分组相似的资源。举个例子，所有 Web 服务器的虚拟机都可以有一个标签*'web-tier-vm'*，这样可以从 UI 和 CLI 把它们作为一个组来获取。对于 IaaS 本身，预先定义的业务逻辑也许从来都不能满足用户的需求。以创建虚拟机为例，默认的选择目标主机的算法是，从主机池中随机选择一个，但用户可能需要各种各样的算法来满足它们的使用情景。比如说选择内存超过 8G 的主机，选择拥有 SR-IOV 硬件的主机，或选择一个有当前用户的运行中虚拟机的主机。IaaS 软件几乎不能为所有无止境的、不可预知的需求提供单独的 API，必须有一种机制允许基础 API（如 APICreateVmInstanceMsg）携带额外信息。

根据各自的业务逻辑，插件可以选择是否创建数据库表。比如，Open vSwitch L2 Network 插件，由于需要创建一种新的类型的资源，可能需要添加一张新表；然而，一个允许主机保留内存的插件可能不需要添加一张新表，而仅需在主机上附加一点数据。如果 IaaS 软件没有为插件提供一种附加数据，它们将开始创造新的、琐碎的模式或添加现有模式的列从而修改现有的模式，导致软件升级时数据库迁移的难处理的情况。

最后，对于建立在 ZStack 上的第三方软件，允许它们将信息存储到 ZStack 的数据库可以避免数据完整性问题，并使得它们可以使用 ZStack 的全部查询 API（详见“查询 API”）。

问题

大多数 IaaS 软件都有着标签的概念。然而，它们并不是都为不同场景定义了一个详尽的标签规范。例如，一些 IaaS 使用标签是为了用户聚合资源，一些 IaaS 是为了内部业务逻辑。ZStack 则为不同场景的标签的每一个层面都精心设计了标签规范。

标签系统

在 ZStack 中，标签本质上是携带了少量资源相关信息的字符串。一个标签通常由以下几个字段组成：

FIELD	DESCRIPTION
uuid	标签的 UUID
resourceUuid	标签所关联的资源的 UUID
resourceType	标签所关联的资源的类型
Tag	一个包含了有意义信息的字符串
Type	标签类型：System 或者 User

在标签方面，ZStack 和其他 IaaS 软件的本质区别是 ZStack 将标签分为两类：用户（User）和系统（System）。

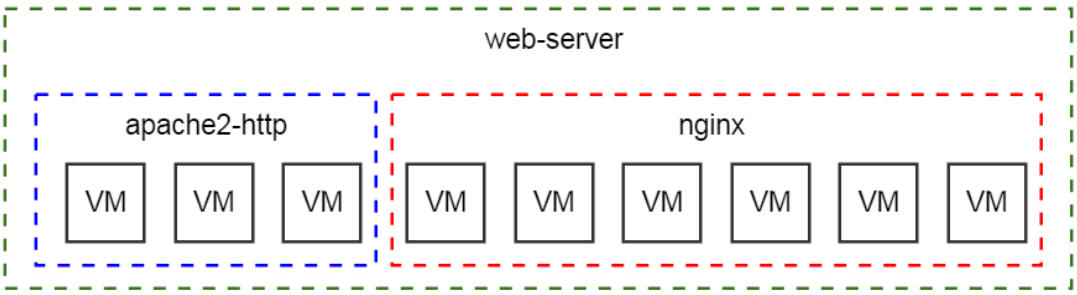
1.用户标签

用户标签，顾名思义，是用户为资源分组而创建的标签。例如，通过标签'apache2-http'将安装了 Apache2 HTTP 服务器的虚拟机分组，这样用户就可以通过查询 API 获取这些虚拟机，使用标签'apache2-http'作为查询条件即可：

```
QueryVmInstance __userTag__=apache2-http
```

备注：详见“查询 API”

这是最常见的标签使用方法，一个资源可以和多个标签相关联，并且被不同的逻辑组划分。



用户标签也可以通过和系统的标签一起使用来控制 ZStack 的行为。例如，如果一个用户标签 `SSD` 已经在主存储系统上创建好，那么一个系统标签可以指导 ZStack 在有用户标签 `SSD` 的主存储上去创建 VM 的根目录。在这种情况下，用户标签更像是用户输入的资源元数据。我们很快就会看到，插件也可以使用系统标签创建资源元数据。

2.系统标签

不像用户标签可以被用户在任意时间、以任意值创建，系统标签有固定的格式，并且是被 ZStack 的业务服务和插件提前定义好的，可以在以下场景被使用：

2.1 元数据

插件可以使用系统标签来记录资源的元数据。例如，主机的数据库表中没有列去记录如 hypervisor 版本，hypervisor SDK 版本这样的元数据；然而，衍生的主机插件，例如，KVM 主机插件，可能需要这些元数据来确定当前虚拟机管理程序是否有某些特征；例如，是否能对 KVM

在线快照是由 libvirt 和 QEMU 版本决定的。在 ZStack 中，当连接到后端主机时，KVM 主机插件将 OS 的版本，libvirt 版本，QEMU 的版本和 qemu-img 工具的版本作为系统标签保存。

```
QuerySystemTag fields=tag resourceUuid=d07066c4de02404a948772e131139eb4



{

  "inventories": [

      {

          "tag": "capability:liveSnapshot"

      },

      {

          "tag": "qemu-img::version::2.0.0"

      },

      {

          "tag": "os::version::14.04"

      },

      {

          "tag": "libvirt::version::1.2.2"

      },

      {

          "tag": "os::release::trusty"

      },

      {

          "tag": "os::distribution::Ubuntu"

      }

  ],
```

```
  "success": true

}
```

2.2 资源属性

插件也可以使用系统标签将新属性添加到资源中。例如，虚拟机的数据库模式中没有列来记录该使用什么 IP 分配算法，什么时候分配虚拟机网卡。这种额外的属性可以用系统标签实现。插件可以创建的系统标签的数量没有限制，附加的插件可以利用这点，并避免干扰数据库模式。

> *数据库表和系统标签：因为数据库表和系统标签都可以定义资源属性，有时会难以决定属性是应该为数据库模式中的一列，还是应该为一个单独的表中的系统标签。添加新列来修改一个现有的数据库模式，通常需要进行数据库迁移，这是 IaaS 软件升级的一个主要痛点。所以开发者可能更倾向使用系统标签来代表新属性。然而，滥用系统标签是一种错误的编程方式。按照 ZStack 的约定，只应该使用系统标签形式引入非固有的资源属性；系统的标签并不能拯救设计的很烂的数据库表。例如，如果 VM 的数据库表缺失集群 UUID（虽然不会），即使需要进行数据库迁移也必须补充回来；但为了私人使用而被用户创建的插件引入的部门 ID 应该作为一个系统标签实现。这种权衡有时候并不容易，我们会严格控制任何数据库结构的变化。*

2.3 元编程

系统标签也可以标注资源以影响 ZStack 的执行流，它在某种程度上类似于 Metaprogramming（https://en.wikipedia.org/wiki/Metaprogramming）。

例如，管理员可以在 KVM 主机上创建一个系统标签 `reservedMemory::1G`，提示 ZStack 主机分配器从主机的可用内存保留 1G 内存；如果管理员改变心意，他可以通过删除标签来回收这 1G 内存。有很多类似的系统标签。

例如，在用户标签这一节中，我们提到了同时使用*用户标签* `SSD` 和*系统标签*来为 VM 的根云盘指定主存储。系统标签叫 `primaryStorage::allocator::userTag::{tag}::required`，如果一个*虚拟机实例规格*上有 `primaryStorage::allocator::userTag::SSD::required`，从该*虚拟机实例规格*上创建的虚拟机根云盘的任务，将只被分配到拥有*用户标签*为 `SSD` 的主存储上。有许

多称之为解释点 `interpreting points` 的代码，将在执行过程中寻找特定的系统标签，可以改变代码的默认行为。

2.4 第三方软件集成

建立在 ZStack 上的第三方软件可以使用系统标签在 ZStack 的数据库中存储和资源关联的信息，这能有效避免第三方软件数据库和 ZStack 数据库的数据不一致性。

例如，一个私有软件可能需要记录虚拟机的部门 ID 来审计每个部门 IT 资源的使用情况，这个功能通常由一个私有的数据库完成，并迫使私有软件跟踪虚拟机的生命周期，因为它需要在数据库创建或销毁时，去更新自己的数据库。否则，数据将不会正确反映真实情况。

有了系统标签的帮助，私有软件可以使用系统标签，例如 `audit::departmentId::{id}`将信息存储在 ZStack 的数据库，将管理*部门 ID* 生命周期的责任转移给 ZStack。当一个虚拟机被销毁，它的*部门 ID*（例如 `audit::departmentId::1`）将在删除该虚拟机记录的数据库事务中被自动删除。此外，私有软件可以用它们的部门 ID 调用常规查询 API 检索虚拟机：

```
QueryVmInstance fields=uuid __sysTag__=audit::departmentId::1
```

注：在 ZStack 版本（0.6）中，我们还没开放允许定义任意系统标签的接口，所有的系统标签都是预先定义的。我们计划在下一个版本中开放这个接口，用户定义的系统标签可以在创建的时候添加一些系统允许的前缀，例如，`3rd::`。

和其他组件的关系

标签系统是 ZStack 核心组件之一；它不仅具有单独的 API 和服务，而且还和其他核心组件无缝集成。用户可以在资源创建时或在资源创建后创建标签。ZStack 所有创造型的 API 支持两个固有参数：`userTags` 和 `systemTags`，通过它们传递的标签将随着资源一起创建。例如:

```
CreateVmInstance name=testTag systemTags=hostname::web-server-1 l3NetworkUuids=6572ce44c3f6422
d8063b0fb262cbc62

instanceOfferingUuid=04b5419ca3134885be90a48e372d3895 imageUuid=f1205825ec405cd3f2d259730d47d1
d8
```

如果资源已经存在，用户可以使用*标签API*来创建或删除标签：

```
CreateUserTag resourceType=VmInstanceVO resourceUuid=613af3fe005914c1643a15c36fd578c6 tag=web

DeleteTag uuid=596070a6276746edbf0f54ef721f654e
```

资源被删除时，与资源相关联的标签将被自动删除。

资源可以通过使用两个特殊的查询条件进行查询：`__userTag__`and`__systemTag__`标签：

```
QueryVmInstance __userTag__=web zoneUuid=04b5419ca3134885be90a48e372d3895

QueryHost __systemTag__=capability:liveSnapshot
```

也有查询 API 专门用于分类：

```
QueryUserTag resourceUuid=0cd1ef8c9b9e0ba82e0cc9cc17226a26 tag~=web-server-%

QuerySystemTag resourceUuid=50fcc61947f7494db69436ebbbefda34
```

总结

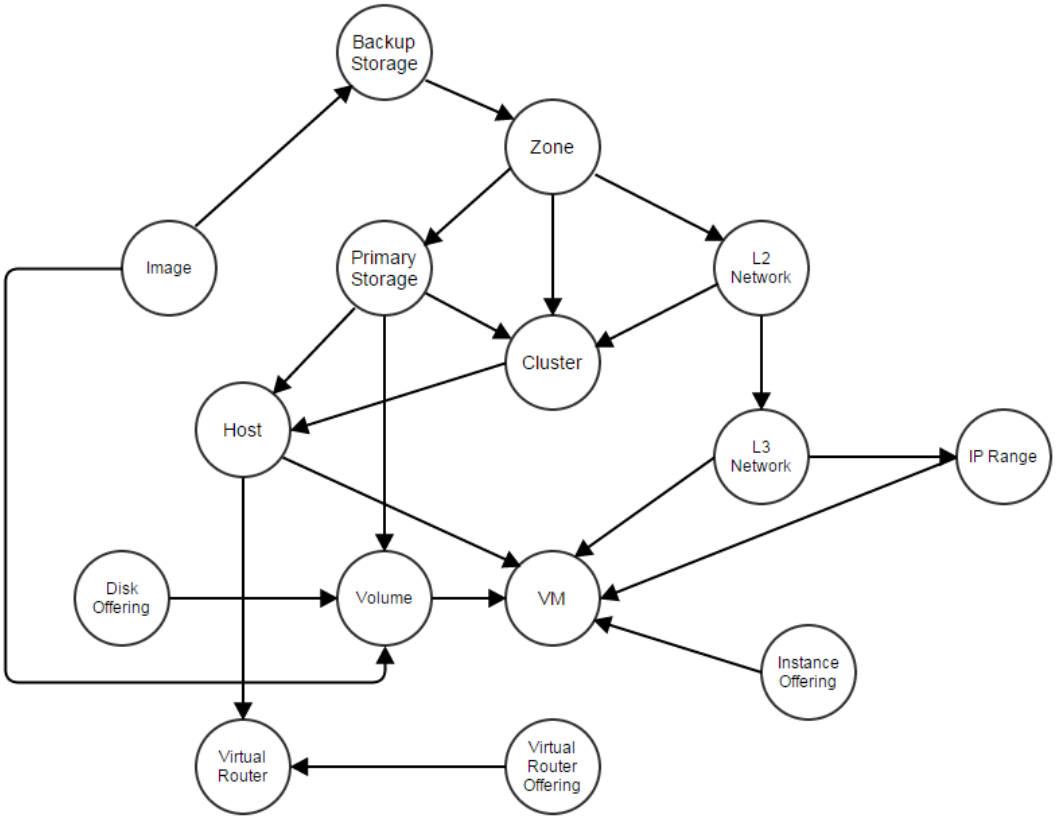
在这篇文章中，我们展示了 ZStack 的标签系统。通过这个系统，用户、插件和第三方软件可以通过各种各样的方式使用标签，而不用改变代码和数据库表结构。这是又一个基本点，激发了一种潜力，使得 ZStack 在快速进化为一个成熟的、完整的云计算解决方案的同时，又能保持核心架构强壮稳定。

ZSTACK--级联框架

云中的资源相互都有关系。操作一个资源通常会引发连锁反应；例如，当删除一个集群的时候，是非常合理地去删除属于该集群的所有主机并停止所有在这些主机上运行的虚拟机。传统的 IaaS 软件要么硬编码连锁反应，要么简单地禁止这些操作，例如，禁止用户删除有虚拟机运行的集群。ZStack 提供一个级联框架，用以散布本来只对一个资源的操作到所有相关的资源。资源可以通过实现一个简单的扩展点以加入级联框架，使得资源的业务逻辑与框架解耦。

动机

云中的资源多多少少都彼此依赖；例如，一个主机是一个集群的子资源，一个主存储是一个集群的兄弟资源，L3 网络是一个区域的后裔资源。资源之间的关系可以被描述为一个有向图：



上图，我们展示了 ZStack 的主要资源；不同的 IaaS 软件可能使用不同的术语，上图主要是想让你有一个粗略的概念。由上图所暗示的，当对资源进行操作时，不仅仅是目标资源，相关资源也将受到影响；例如，当删除一个区域时，比较理想的是属于区域的集群、主机、主存储、L2 网络等资源也同时被删除。为了处理这个问题，IaaS 软件必须满足级联（cascading）操作的需求。

问题

大多数 IaaS 软件很少考虑级联操作。它们要么硬编码业务逻辑，例如，你需要显式删除一个将要被删除帐户的所有资源；要么直接不允许这种操作，例如，当你试图删除一个 IP 地址范围时，抛出一个错误信息“仍有 VM 使用在这个 IP 范围中的 IP”。这两种方法都会带来很多麻烦。对硬编码而言，它使软件不能灵活的添加新的资源，因为你必须修改现有的代码来添加级联操作，例如，修改删除帐户的代码使得账户删除时，新资源也被删除。对于完全没有责任感的错误信息，用户要么去做无聊的工作，例如，在删除一个 IP 范围之前，手动删除 100 个虚拟机；要么摧毁现有的一切，然后从零开始，例如，重新部署整个云。

> ***避免误操作不是借口：****有些人可能会声称不允许级联删除是慎重考虑的结果，因为用户可能会误操作，误操作可能带来灾难性的后果；例如，错误地删除区域会导致损失掉所有虚拟机。然而，这种说法只是一个错误的借口，并且是一种为用户做决定的自作聪明。你能想象吗，当你为了删除一个区域必须手动删除 10,000 个虚拟机，因为软件认为你可能会做错事，所以迫使你枯燥的重复 10,000 次任务确认？一个好的软件应该为用户提供选择，并让他们做出决定。在我们的例子中，IaaS 软件应该在进行到最后删除之前警告用户，还有 10,000 台虚拟机在运行；但一旦用户承认他们需要这么做，软件就应该这么做。*

级联框架

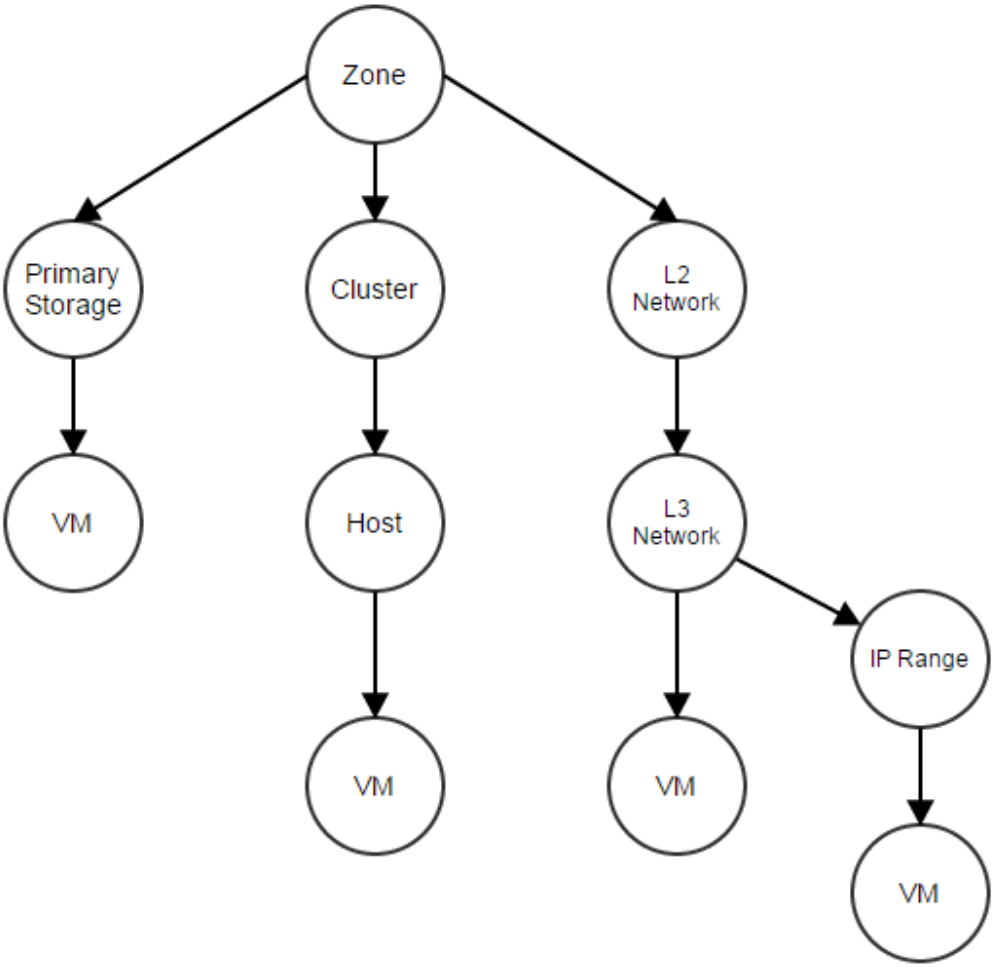
ZStack 通过一个级联框架解决这一问题；顾名思义，级联框架允许一个操作能从一个资源级联到其他资源。为了解耦整个架构，这个级联框架被作为一个单独的组件创造出来，资源可以按意愿加入框架。要加入框架，资源所需要做的全部事情就是实现一个扩展点 `CascadeExtensionPoint`（在我们的例子中 `AbstractAsyncCascadeExtension` 是一个实现 `CascadeExtensionPoint` 的类）：

```
class VmCascadeExtension extends AbstractAsyncCascadeExtension {

    @Override
```

```
public void asyncCascade(CascadeAction action, Completion completion) {

    if (/* this is from deleting Primary Storage*/) {

        /* delete VMs that have root volumes on the primary storage*/

    } else if (/*this is from deleting L3 Network*/) {

        /* stop VMs that have nics on the L3 network, and remove those nics */

    } else if (/* this is from deleting IP range*/) {

        /* stop VMs that have nics whose IP is in the IP range */

    } else if (/* this is from deleting host*/) {

        /* stop VMs that run on the host */

    }


    completion.success();

}


@Override

public List<String> getEdgeNames() {

    return Arrays.asList(

            PrimaryStorageVO.class.getSimpleName(),

            L3NetworkVO.class.getSimpleName(),

            IpRangeVO.class.getSimpleName(),

            HostVO.class.getSimpleName()

    );

}


@Override
```

```
    public String getCascadeResourceName() {
        return VmInstanceVO.class.getSimpleName();
    }

    @Override
    public CascadeAction createActionForChildResource(CascadeAction action) {
        return convertContextToVmRelatedContext(action);
    }
}
```

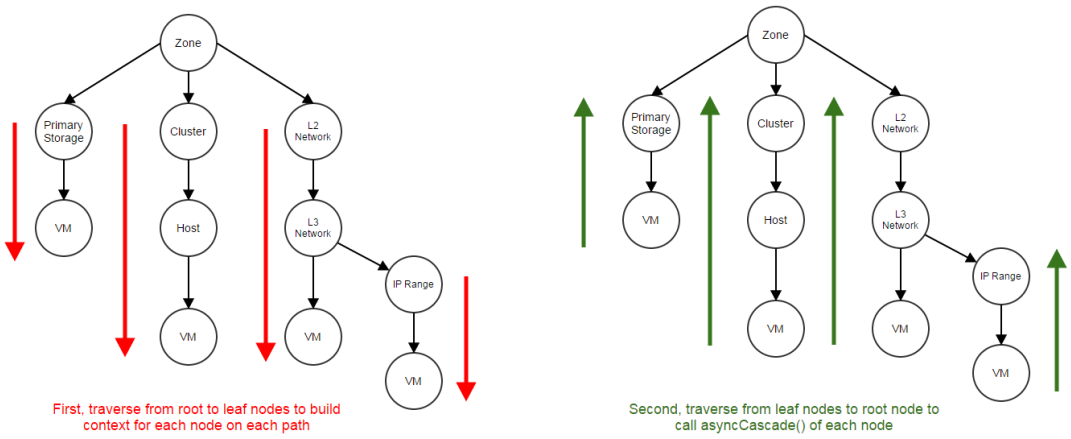
`getCascadeResourceName()`方法返回该资源的名称（VmInstance）；`getEdgeNames()`方法返回一个和资源直接关联的资源名列表，在我们的例子中返回主存储、L3 网络、IpRange 和主机；所以如果删除操作在这些 `edge resources` 或其上游资源（如区域）上发生时，该操作将被级联至在 `getEdgeNames()`方法中声明了这些资源的扩展。级联扩展可以在 `asyncCascade()` 中采取行动，并获取必须的信息比如操作码（如删除），根发起者（如区域，下文将很快给出解释），作为操作来源的父发起者（如主机，将很快给出解释）和操作上下文（例如，哪台主机正在被删除）。由于资源的关系是一个可能有环路的有向图，级联框架将把图压扁成一棵树，并把环路变为分支。例如，删除区域的操作将最终创建以下树（一部分）：



注：如你所见，删除区域操作将多次级联到虚拟机的级联扩展；这是刻意的，因为级联扩展通常依赖于父发起者去决定该采取什么行动；在这个例子中，虚拟机的父发起者为主存储、主机、L3 网络和 IP 范围；然而，对于不同的父发起者，扩展可能会采取不同的行动；例如，如果父发起者为主存储并且操作码为`delete`，该扩展将摧毁所有根云盘在该主存储的虚拟机；但如果父发起者是主机，扩展将会只停止在那台主机上的虚拟机，因为这些虚拟机稍后就可以在其他主机上启动。考虑到 ZStack 没有产生冲突的级联操作，例如，不会有一个操作导致虚拟机在路径 A 启动而在路径 B 停止，所以级联操作从不同路径进行多次延伸是没有问题的。

当级联一个操作时，该框架从该操作被应用的`root issuer`开始；在上述删除区域的示例中，`zone`是根发起者；那么框架将从根发起者遍历树，并调用扩展的`createActionForChildResource()`方法为每一条路径上的每一个扩展创建上下文；一旦所有上下文创建成功，该框架将再次遍历树，不过是从叶子节点到根，并调用每个扩展的`asyncCascade()`方法；一个扩展可以依靠父发起者去决定应该做哪些操作，父发起者在

`getEdgeNames()`方法中以资源名的方式声明；例如，如果父发行者是主机，则停止虚拟机；如果父发行者是主存储，则删除虚拟机。



这两个阶段的遍历保证，一个操作（例如删除）将只会被应用到根发起者，在所有下游资源都做完一些合适的操作后。例如，一个区域只在所有子孙资源都被删除后才能被删除。

由于并不是所有的操作都需要级联，一个资源可以在它需要的时候直接调用`CascadeFacade.asyncCascade()`。

总结

在这篇文章中，我们演示了 ZStack 的级联框架，这是一个强大的工具，用于扩散操作而不需要硬编码。ZStack 用很多方式使用了它，除了我们在文中提到的以外，一些操作，如卸载主存储（这将停止将被卸载的集群中的所有虚拟机），卸载 L2 网络（这将停止将被卸载的集群中的所有虚拟机）都是以这种方式实现的。有了它的帮助，管理员可以快速尝试不同的云部署而无需担心不方便；你可以只删除你的部署的一部分并重新创建一个新的，而不需要仅因为你在一个设计错误的 L2 网络上创建了许多虚拟机，就重新部署整个云（举个例子）。

ZSTACK--查询 API

IaaS 软件用户面临的共同挑战是如何快速、准确地找到一个想要的资源；例如，从 10,000 台虚拟机中发现有 EIP（16.16.16.16）的虚拟机。大多数 IaaS 软件通过 API 中的特定查询逻辑解决这个问题。ZStack 不用特定查询，而是配备了一个框架，这个框架可以自动为每个资源的每个字段生成查询，并联合跨越了多个资源的查询，帮助用户管理云端数量庞大的资源。

动机

一个中型的云可以管理几百台物理主机和成千上万台虚拟机，因为 IaaS 软件很少有全部的查询 API，导致寻找想要的资源成为挑战。大多数 IaaS 软件只允许用户使用少量条件（如 name，UUID）查询资源，这些条件硬编码在查询 API 中。如果用户想要使用硬编码之外的条件做一个查询，例如，通过创建日期查询虚拟机，他们可能不得不最终列出所有虚拟机，然后用 `for..loop` 来过滤结果。使用任意字段查询资源至今在大多数 IaaS 软件都不被完全支持，更不用说联合查询；例如，如果用户想要找到一个虚拟机，这个虚拟机的网卡应用了特定的安全组规则，他们可能不得不列出所有资源（虚拟机，安全组），然后做两次 `for..loop`。

另一方面，相比类似 JIRA 的软件，大多数 IaaS 软件的 UI 是最粗糙简陋的。许多开发人员可能并没有意识到糟糕 UI 的根源不是因为 UI 开发人员缺乏 CSS/HTML/JavaScript 的技能，而是软件本身并不能提供强大的 API 来支持复杂的 UI；例如，为了实现一个类似 JIRA 过滤器的功能，即只显示满足指定条件的资源，UI 可能需要做很多的需要 `listing all then filtering by for..loop` 的后置处理工作，这些后置处理工作将把大量的资源拧在一起。

IaaS 软件因此饱受折磨了一段时间；对此的解药就是要提供一种机制，这种机制可以自动为每个资源的每个字段都生成查询，而且可以处理 join 查询。

问题

大多数 IaaS 软件使用关系型数据库（如 MySQL）作为后台数据库，在这种数据库中资源通常被安排在单独的表中，比如虚拟机表，主机表，云盘表。对于每一个资源，都有一个 API 用来获取该资源的单独的一条，它可能被命名为 `describe API`、`list API` 或 `query API`；这些 API 通常有硬编码的参数，用来暴露一部分数据库表的列，允许用户通过少量的查询条件查询资源；这些参数是精心选择的，通常是对 API 设计者自身非常重要的列，例如，name、UUID。然而，由于并不是所有的列都被暴露，用户经常遇到他们想查询的列不存在的情况，这样他们就必须检索所有的资源，然后使用一个或多个 `for..loop` 进行后置处理。

一个复杂的查询场景，可能需要使用联合查询，这种查询通常跨越多个数据库表；例如，找到一个 EIP 为 16.16.16.16 的虚拟机，它可能涉及虚拟表、网卡表和 EIP 表。一些 IaaS 软件使用数据库视图解决这个问题，这是另一种硬编码方式，只能以固定的格式 join 选中的表，而在现实中表是能以非常复杂的方式被 join 的。在软件升级过程中，如果一个视图指向的任一表已经改变了的话，视图也需要进行数据库迁移操作。

查询 API

为了避免在 API 中手动编码查询逻辑，并给用户提供能在任何地方查询任何东西的灵活的查询，ZStack 创建了一个框架，这个框架可以自动为所有资源生成查询，并且不需要开发者写代码去实现查询逻辑；更进一步，该框架还可以生成各种 join 查询，只要所需的表已经通过外键连接。

在以下篇幅中，我们将用 zone 作为一个例子来阐述这个令人惊叹的框架。一个 zone 在数据库中有下面的这些列：

FIELD	DESCRIPTION
uuid	zone UUID
name	zone name

description	zone description
state	zone state
type	zone type
createDate	the time the zone was created
lastOpDate	the last time the zone was operated

用户可以通过任何一个字段或字段组合来查询 zone，并采用常规的 SQL 比较运算符如'=', '!=', '>', '>=', '<', '<=', 'in', 'not in', 'is null', 'is not null', 'like', 'not like'。

注意：在命令行工具中，一些运算符有不同的格式：'in'(?=), 'not in'(!?=), 'is null'(=null), 'is not null'(!=null), 'like'(~=), 'not like'(!~=).

```
QueryZone name=west-coast-zone

QueryZone name=west-coast-zone state=Enabled
```

因为 zone 是 ZStack 中主要资源的祖先，很多资源都或多或少和它有关系；例如，一个运行中的虚拟机总是在一个 zone 内。像这种关系可以生成联合查询，如：

```
QueryZone vmInstance.name=web-vm1
```

如上表格所示，一个 zone 不会暴露任何叫 `vmInstance` 的字段，但在上述查询中有一个条件是由'vmInstance'开始的。这种查询在 ZStack 中称为*扩展查询*。这里 `vmInstance` 代表 VM 表，VM 表有一个字段为 `zoneUuid`（外键）指向 zone 表，因此查询框架可以理解它们的关系并生成联合查询。上面的例子可以被解释为“*寻找运行着名字为 web-vm1 的虚拟机的 zone*”。进一步扩展这个例子，因为虚拟机网卡表有外键指向 VM 表，并且 EIP 表有外键指向虚拟机网卡表，查询 zone 也可以使用 EIP 作为条件：

```
QueryZone vmInstance.vmNics.eip.vipIp=16.16.16.16
```

查询被解释为*“查找一个区域，它上面的虚拟机的网卡的 EIP 为 16.16.16.16”*。现在您知道了查询接口的强大之处了！我们甚至可以创建一些非常复杂的查询：

```
QueryVolumeSnapshot volume.vmInstance.vmNics.l3Network.l2Network.attachedClusterUuid
s=13238c8e0591444e9160df4d3636be82
```

这个复杂的查询目的是找到磁盘快照，目标磁盘快照是由虚拟机磁盘创建的，而该虚拟机有网卡在 L3 网络上，这个 L3 网络的父 L2 网络则是附加在一个集群上的，这个集群的 uuid 是 13238c8e0591444e9160df4d3636be82。不要惊慌，你很少需要这么复杂的查询，但它确实证明了框架的能力。此外，SQL 的一些特性例如选择字段、排序、计数和分页也是支持的：

```
QueryL3Network name=L3-SYSTEM-PUBLIC count=true

QueryL3Network l2NetworkUuid=33107835aee84c449ac04c9622892dec limit=10

QueryL3Network l2NetworkUuid=33107835aee84c449ac04c9622892dec start=10 limit=100

QueryL3Network fields=name,uuid l2NetworkUuid=33107835aee84c449ac04c9622892dec

QueryL3Network l2NetworkUuid=33107835aee84c449ac04c9622892dec sortBy=createDate sort
Direction=desc
```

实现

尽管查询 API 功能是如此强大，实现却是非常简洁的。当添加一个新的资源时，开发人员不需要写任何关于查询逻辑的代码，除了定义查询 API 和资源本身。要实现 zone 的查询 API，开发人员需要：

1. 使用查询元数据注解 zone 的 inventory

```
@Inventory(mappingVOClass = ZoneVO.class)

@PythonClassInventory
```

```
@ExpandedQueries({
        @ExpandedQuery(expandedField = "vmInstance", inventoryClass = VmInstanceInventory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "cluster", inventoryClass = ClusterInventory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "host", inventoryClass = HostInventory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "primaryStorage", inventoryClass = PrimaryStorageInventory.cla
ss,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "l2Network", inventoryClass = L2NetworkInventory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "l3Network", inventoryClass = L3NetworkInventory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid"),
        @ExpandedQuery(expandedField = "backupStorageRef", inventoryClass = BackupStorageZoneRefInven
tory.class,
                foreignKey = "uuid", expandedInventoryKey = "zoneUuid", hidden = true),
      })
@ExpandedQueryAliases({
        @ExpandedQueryAlias(alias = "backupStorage", expandedField = "backupStorageRef.backupStorage
")
      })
     public class ZoneInventory implements Serializable{
    private String uuid;
    private String name;
    private String description;
```

```
    private String state;

    private String type;

  private Timestamp createDate;

  private Timestamp lastOpDate;

      }
```

上面的注解声明了 zone 和其他资源之间的关系，这是 zone 扩展查询的基础。

2.定义一个查询 API

```
@AutoQuery(replyClass = APIQueryZoneReply.class, inventoryClass = ZoneInventory.class)

     public class APIQueryZoneMsg extends APIQueryMessage {

      }
```

3.在区域的 API 配置文件中声明查询 API

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

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

    <id>zone</id>

    <interceptor>ZoneApiInterceptor</interceptor>



    <message>

        <name>org.zstack.header.zone.APIQueryZoneMsg</name>

        <serviceId>query</serviceId>

    </message>

      </service>
```

API `APIQueryZoneMsg` 通过指定服务 ID `query` 被路由到查询服务。就是这样了，查询逻辑不需要一行代码；查询服务会把其余部分自动完成。所有的 ZStack 查询 API 都像这样定义，添加新资源的查询 API 是非常容易的。

当前限制

主要的限制是在查询条件中，只有逻辑 `AND` 是被支持的，`OR` 是不被支持的。例如：

```
QueryZone name=west-coast-zone state=Enabled
```

上述查询语句可以被解释为“寻找区域名字为 west-coast 且 state 是 Enabled 的区域”我们这么做的原因是我们由 ZStack 源代码中 SQL 的使用分析得出 99%的组合的查询条件都是基于 `AND` 逻辑的。另一方面，如果逻辑 `OR` 在没有创建 DSL 的情况下就被引入，要保持代码简洁是非常困难的。然而，在很多情况下，`OR` 可以使用比较操作 `in(?=)`实现：

```
QueryZone name=west-coast-zone state?=Enabled,Disabled
```

上述例子表述的是“寻找名字为 west-coast 的区域，并且它的状态是 Enabled 或 Disabled”，将来，我们将引入 DSL 风格的查询语言，例如：

```
QueryZone name=west-coast-zone AND (state=Enabled OR state=Disabled)
```

总结

这篇文章中，我们演示了 ZStack 的查询 API。通过使用这个强大的工具，用户能以类似关系型数据库的方式查询任何资源。将来，ZStack 将建立一套高级的 UI，它可以使用查询 API 创建各种各样的视图（过滤器），例如，展示所有运行在同一 L3 网络的虚拟机，为 IaaS UI 的用户体验带来革命性的改变。