ZStack 技术白皮书精选

架构篇：上册

ZStack 可拓展性的秘密武器

扫一扫二维码，获取更多技术干货吧

版权声明

目录

ZSTACK--可拓展性的秘密武器 1：异步架构

ZStack 的架构使得其中 99%的任务能被异步执行。基于这点，ZStack 中单一的管理节点可以管理几千台物理服务器，上万台虚拟机，处理成千上万个并发任务。

动机

对于管理大量硬件和虚拟机的公有云而言，可拓展性是一个 IaaS 软件必须解决的关键问题之一。对于一个大概拥有 5 万台物理服务器的中型数据中心，预计可能有 150 万台虚拟机，1 万名用户。虽然用户开关虚拟机的频率不会像刷朋友圈一样频繁，但是在某一时刻，IaaS 系统可能有成千上万个任务要处理，这些任务可能来自 API 也可能来自内部组件。在糟糕的情况下，用户为了创建一台新的虚拟机可能需要等待一个小时，因为系统同时被 5000 个任务阻塞，然而线程池仅有 1000 条线程。

问题

首先，我们非常不赞同一些文章里面描写的关于**“一些基础配套设施，尤其是数据库和消息代理（message brokers）限制了 IaaS 的可拓展性”**的观点。首先，对于数据库而言，IaaS 软件的数据量相比 facebook 和 twitter 而言只能勉强算中小型，facebook 和 twitter 的数据量是万亿级别，IaaS 软件只处于百万级别（对于一些非常大型的数据中心），而 facebook 和 twitter 依旧坚强的使用 Mysql 作为他们主要的数据库。其次，对于消息代理而言，ZStack 使用的 rabbitmq 相对 Apache Kafka 或 ZeroMQ 是一个中型的消息代理，但是它依然可以维持平均每秒 5 万条消息的吞吐量，对于 IaaS 软件内部通信而言这不就足够了么？我们认为足够了。

限制 IaaS 可拓展性的主要原因在于：任务执行缓慢。IaaS 软件上的任务运行非常缓慢，通常一项任务完成需要花费几秒甚至几分钟。所以当整个系统被缓慢的任务填满的时候，新任务的延迟非常大是很正常的。执行缓慢的任务通常是由一个很长的任务路径组成的，比如，创建一个虚拟机，需要经过身份验证服务→调度器→镜像服务→存储服务→网络服务→虚拟机管理

程序，每一个服务可能会花费几秒甚至几分钟去引导外部硬件完成一些操作，这极大的延长了任务执行的时间。

同步和异步

传统的 IaaS 软件使用同步的方式执行任务，他们通常给每一个任务安排一个线程，这个线程只有在之前的任务执行完毕时才会开始执行下一个任务。因为任务执行缓慢，当达到一个任务并发的高峰时，系统会因为线程池容量不足，运行非常缓慢，新来的任务只能被放在队列中等待被执行。

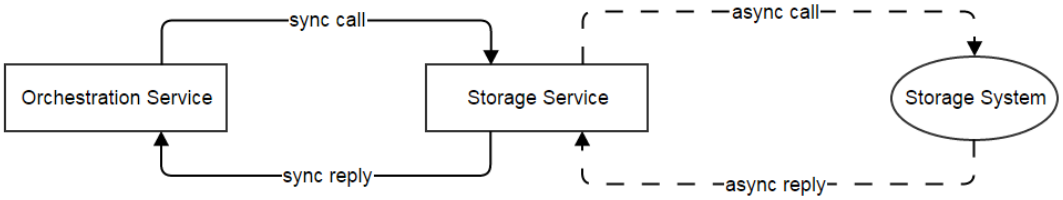
为了解决这个问题，一个直观的想法是提高线程池容量，但是这个想法在实际中是不可行的，即使现代操作系统允许一个应用程序拥有成千上万条线程，没有操作系统可以非常有效率的调度他们。随后有一个想法是把线程分发出去，让不同的操作系统上相似的软件分布式的处理线程，因为每一个软件都有它自己的线程池，这样最终增加了整个系统的线程容量。然而，分发会带来一定的开销，它增加了管理的复杂度，同时集群软件在软件设计层面依旧是一个挑战。最后，IaaS 软件自身变成云的瓶颈，而其他的基础设施包括数据库，消息代理和外部的系统（比如成千台物理服务器）都足够去处理更多的并发任务。

ZStack 通过异步架构来解决这个问题，如果我们把目光投向 IaaS 软件和数据中心的设备之间的关系，我们会发现 IaaS 软件实际上扮演着一个协调者的角色，它负责协调外部系统但并不做任何真正耗时的操作。举个例子，存储系统可以分配磁盘容量，镜像系统可以下载镜像模板，虚拟机管理程序可以创建虚拟机。**IaaS 软件所做的工作是做决策然后把子任务分配给不同的外部系统。**比如，对于 KVM，KVM 主机需要执行诸如准备磁盘、准备网络、创建虚拟机等子任务。创建一台虚拟机可能需要花费 5s，IaaS 软件花费时间为 0.5s，剩下的 4.5s 被 KVM 主机占用，ZStack 的异步架构使 IaaS 管理软件不用等待 4.5s，它只需要花费 0.5s 的时间选择让哪一台主机处理这个任务，然后把任务分派给那个主机。一旦主机完成了它的任务，它将结果通知给 IaaS 软件。通过异步架构，一个只有 100 条线程容量的线程池可以处理上千数的并发任务。

ZStack 的异步方法

异步操作在计算机科学中是非常常见的操作，异步 I/O,AJAX 等都是一些众所周知的例子。然而，把所有的业务逻辑都建立在异步操作的基础上，尤其是对于 IaaS 这种非常典型的集成软件，是存在很多挑战的。

最大的挑战是必须让所有组件都异步，并不只是一部分组件异步。举个例子，如果你在其他服务都是同步的条件下，建立一个异步的存储服务，整个系统性能并不会提升。因为在异步的调用存储服务时，调用的服务自身如果是同步的，那么调用的服务必须等待存储服务完成，才能进行下一步操作，这会使得整个工作流依旧是处于同步状态。



The thread that orchestration service calls storage service has to block until the storage service returns, though the storage service communicates to the external storage system asynchronously

（一个执行业务流程服务的线程同步调用了存储服务，然而存储服务和外部存储系统是异步通信的，因此它依旧需要等到存储服务返回之后，才能往下执行，这里的异步就等同于同步了。）

ZStack 的异步架构包含了三个模块：异步消息，异步方法，异步 HTTP 调用。

1. 异步消息

ZStack 使用 rabbitmq 作为一个消息总线连接各类服务，当一个服务调用另一个服务时，源服务发送一条消息给目标服务并注册一个回调函数，然后立即返回。一旦目标服务完成了任务，它返回一条消息触发源服务注册的回调函数。代码如下：

```
AttachNicToVmOnHypervisorMsg amsg = new AttachNicToVmOnHypervisorMsg();

amsg.setVmUuid(self.getUuid());

amsg.setHostUuid(self.getHostUuid());

amsg.setNics(msg.getNics());

bus.makeTargetServiceIdByResourceUuid(amsg, HostConstant.SERVICE_ID, self.getHostUuid());
```

```
bus.send(amsg, new CloudBusCallBack(msg) {
    @Override
    public void run(MessageReply reply) {
        AttachNicToVmReply r = new AttachNicToVmReply();
        if (!reply.isSuccess()) {
            r.setError(errf.instantiateErrorCode(VmErrors.ATTACH_NETWORK_ERROR, r.getError()));
        }
        bus.reply(msg, r);
    }
});
```

一个服务也可以同时发送一列消息给其它服务，然后异步的等待回复。

```
final ImageInventory inv = ImageInventory.valueOf(ivo);
final List<DownloadImageMsg> dmsgs = CollectionUtils.transformToList(msg.getBackupStorageUuids
(), new Function<DownloadImageMsg, String>() {
    @Override
    public DownloadImageMsg call(String arg) {
        DownloadImageMsg dmsg = new DownloadImageMsg(inv);
        dmsg.setBackupStorageUuid(arg);
        bus.makeTargetServiceIdByResourceUuid(dmsg, BackupStorageConstant.SERVICE_ID, arg);
        return dmsg;
    }
});

bus.send(dmsgs, new CloudBusListCallBack(msg) {
    @Override
```

```
    public void run(List<MessageReply> replies) {
        /* do something */
    }
}
```

更甚，以设定的并行度发送一列消息也是可以实现的。也就是说，含有 10 条消息的消息列表，可以每次发送 2 条消息，也就是说第 3、4 条消息可以在第 1、2 条消息的回复收到后被发送。

```
     final List<ConnectHostMsg> msgs = new ArrayList<ConnectHostMsg>(hostsToLoad.size());
     for (String uuid : hostsToLoad) {
         ConnectHostMsg connectMsg = new ConnectHostMsg(uuid);
         connectMsg.setNewAdd(false);
         connectMsg.setServiceId(serviceId);
         connectMsg.setStartPingTaskOnFailure(true);
         msgs.add(connectMsg);
}

bus.send(msgs, HostGlobalConfig.HOST_LOAD_PARALLELISM_DEGREE.value(Integer.class), new CloudBu
sSteppingCallback() {
         @Override
         public void run(NeedReplyMessage msg, MessageReply reply) {
            /* do something */
         }
});
```

2.异步的方法

服务在 ZStack 中是一等公民，他们通过异步消息进行通信。在服务的内部，有非常多的组件、插件使用方法调用的方式来进行交互，这种方式也是异步的。

```
protected void startVm(final APIStartVmInstanceMsg msg, final SyncTaskChain taskChain) {

    startVm(msg, new Completion(taskChain) {

        @Override

        public void success() {

            VmInstanceInventory inv = VmInstanceInventory.valueOf(self);

            APIStartVmInstanceEvent evt = new APIStartVmInstanceEvent(msg.getId());

            evt.setInventory(inv);

            bus.publish(evt);

            taskChain.next();

        }



        @Override

        public void fail(ErrorCode errorCode) {

            APIStartVmInstanceEvent evt = new APIStartVmInstanceEvent(msg.getId());

            evt.setErrorCode(errf.instantiateErrorCode(VmErrors.START_ERROR, errorCode));

            bus.publish(evt);

            taskChain.next();

        }

    });

}
```

回调函数也可以有返回值：

```
public void createApplianceVm(ApplianceVmSpec spec, final ReturnValueCompletion<ApplianceVmInv
entory> completion) {

          CreateApplianceVmJob job = new CreateApplianceVmJob();

          job.setSpec(spec);

          if (!spec.isSyncCreate()) {

            job.run(new ReturnValueCompletion<Object>(completion) {

                @Override

                public void success(Object returnValue) {

                  completion.success((ApplianceVmInventory) returnValue);

                }



                @Override

                public void fail(ErrorCode errorCode) {

                  completion.fail(errorCode);

                }

            });

          } else {

             jobf.execute(spec.getName(), OWNER, job, completion, ApplianceVmInventory.class);

          }

}
```

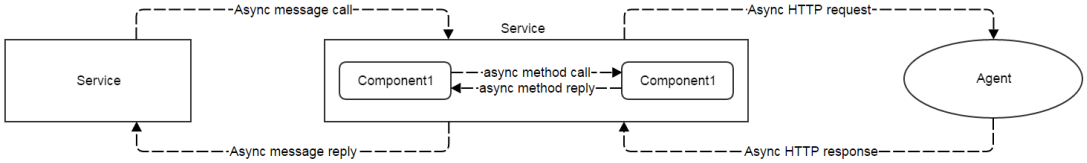
3.HTTP 异步调用

ZStack 使用一组 agent 去管理外部系统，比如：管理 KVM 主机的 agent，管理控制台代理的 agent，管理虚拟路由的 agent 等，这些 agents 全部都是搭建在 Python CherryPy 上的轻量级

web 服务器。因为如果没有 HTML5 技术，如 websockets 技术，是没有办法进行双向通信的，ZStack 每个请求的 HTTP 头部嵌入一个回调的 URL，因此，在任务完成后，agents 可以发送回复给调用者的 URL。

```
RefreshFirewallCmd cmd = new RefreshFirewallCmd();

List<ApplianceVmFirewallRuleTO> tos = new RuleCombiner().merge();

cmd.setRules(tos);



resf.asyncJsonPost(buildUrl(ApplianceVmConstant.REFRESH_FIREWALL_PATH), cmd, new JsonAsyncREST
Callback<RefreshFirewallRsp>(msg, completion) {

    @Override

    public void fail(ErrorCode err) {

        /* handle failures */

    }



    @Override

    public void success(RefreshFirewallRsp ret) {

        /* do something */

    }



    @Override

    public Class<RefreshFirewallRsp> getReturnClass() {

        return RefreshFirewallRsp.class;

    }

});
```

通过以上三种方法，ZStack 已经建立了一个可以使所有组件都异步进行操作的全局架构。



总结

为了解决由缓慢且并发的任务引起的 IaaS 软件可拓展性受限的问题，我们演示了 ZStack 的异步架构。我们使用模拟器进行测试后发现，一个具有 1000 条线程的 ZStack 管理节点可以轻松处理创建 100 万台虚拟机时产生的 10000 个并发任务。虽然单一管理节点的拓展性已经可以满足大多数云的负载需要，考虑到系统需要高可用性以及承受巨大的负载量（10 万个并发任务），我们需要一组管理节点来满足这些需求，如需了解 ZStack 的无状态服务，请阅读下一篇“ZStack 可拓展性秘密武器 2：无状态服务”。

ZSTACK—可拓展性秘密武器 2：无状态的服务

每一个 ZStack 服务都是无状态的，简单的开启一个富余的服务实例然后使之负载均衡，就能实现服务的高可用和可横向拓展；此外，ZStack 把所有服务封装进一个称为管理节点的进程中，使得部署和管理服务变得尤其简单。

动机

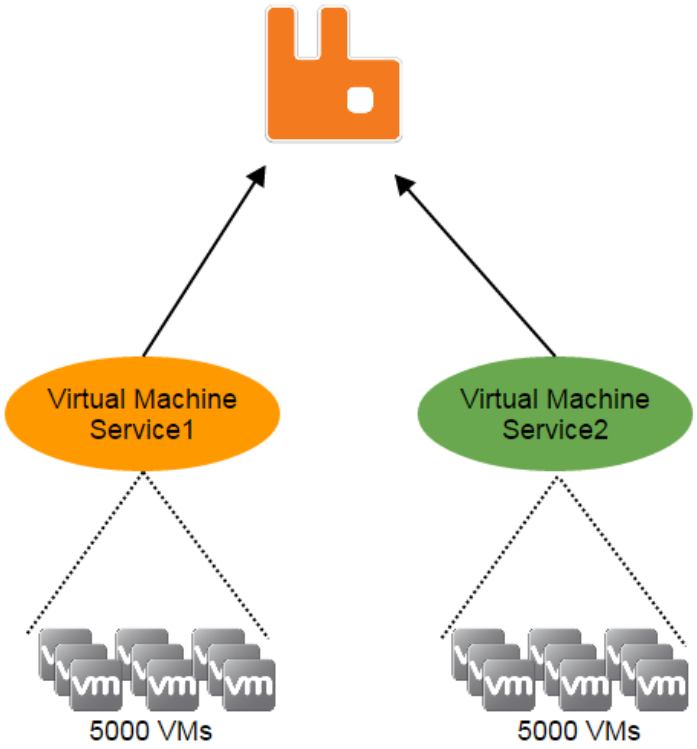
在“ZStack 的可拓展性秘密武器 1：全异步架构”中，我们论述了异步的架构使得单一的 ZStack 管理节点足以承担大多数云的负载量；然而当用户想要去创建一个高可用的生产环境或处理非常大的并发工作负载，一个管理节点是不够的。解决方案是建立一个负载均衡的分布式系统，这种通过添加新节点来拓展整个系统的能力的方法被称为横向拓展。

问题

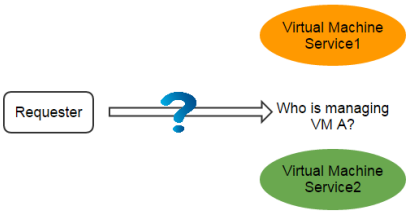
设计一个分布式系统不是一件简单的事情。一个分布式系统，特别是一个有状态的系统，必须处理一致性（consistency）、可用性（availability）和分区容忍性（partition tolerance）（CAP 理论）的问题，每一个问题都是非常复杂的。与之相反，一个无状态的分布式系统一定程度上降低了复杂度。第一，因为节点不用分享状态，整个系统的一致性是可以保证的。第二，因为节点都是相似的，对于分区问题系统通常是可以容忍的。因此，通常把一个分布式系统设计为无状态的而不是有状态的。但是设计一个无状态的分布式系统通常比设计一个有状态

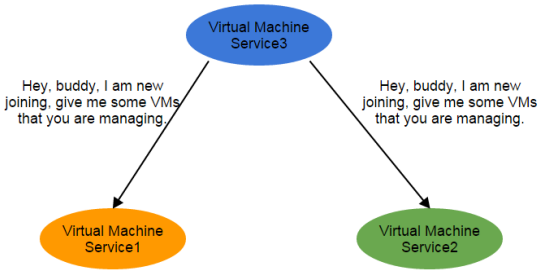
的分布式系统难得多。利用消息代理和数据库的优点，ZStack 建立了一个包含各种无状态服务的无状态分布式系统。

使整个系统无状态的基础是无状态的服务，在讨论什么是无状态的服务之前，我们首先理解什么是“状态”。在 ZStack 中，主机、虚拟机、镜像和用户等资源是被一个个服务管理的。当整个系统的服务实例不止一个的时候，资源会被分发到不同的服务实例中。假设有 10000 台虚拟机和两个虚拟机服务实例，理想状态下每个实例会管理 5000 台虚拟机。



因为有两个服务实例，在向一个虚拟机发出请求前，请求者必须知道哪个实例管理哪个虚拟机，否则，他将不知道向哪个实例发出请求。类似“**哪个服务实例管理哪个资源**”的信息就是我们所说的状态。如果一个服务是有状态的，每个服务维护自己的状态。请求者必须可以获取到当前的状态信息。当服务实例的数量改变的时候，服务需要去改变状态。



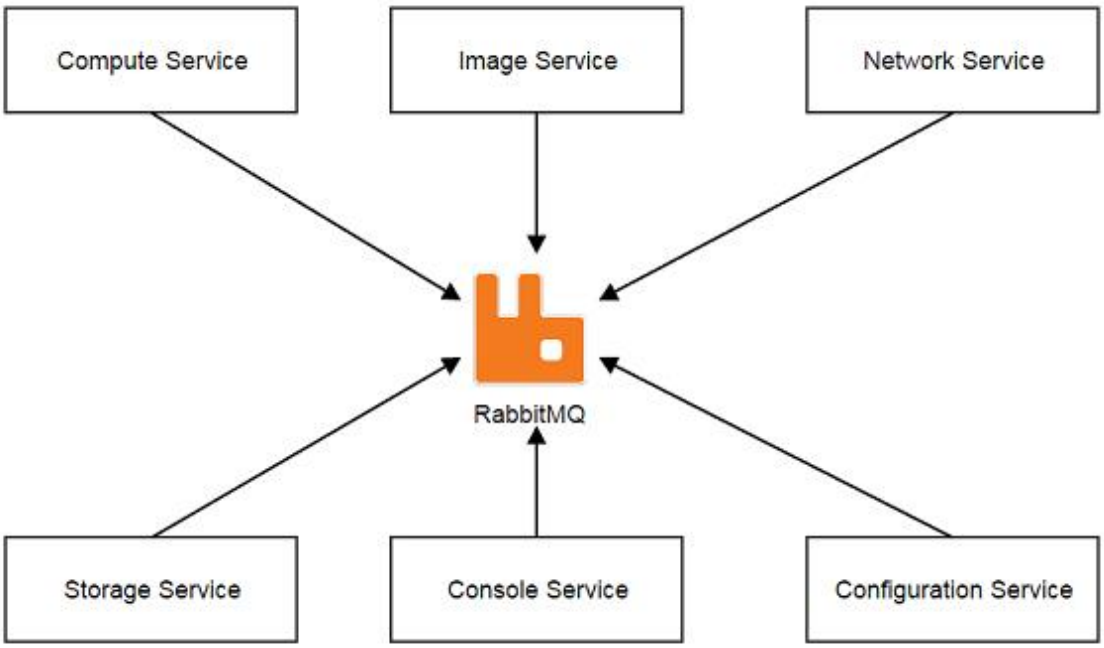


状态的改变是危险且易错的，这通常限制了整个系统的可拓展性。为了使整个系统的可靠性和横向拓展性增强，把状态和服务分离开，使得服务无状态是比较理想的解决办法（参考 Service Statelessness Principle）。**无状态的服务使请求者不需要询问向哪里发送请求，当新添一个新的服务实例或者删除一个旧的服务实例的时候，服务之间也不用交换状态。**

备注：在以下文档中，为了简便，“服务”和“服务实例”是可以互换的。

服务和管理节点

通过中心消息代理-- RabbitMQ 彼此通信的服务，在 ZStack 中是一等公民。



与典型的 microservice 架构不同，典型的 microservice 架构中每个服务通常运行在不同的进程或者不同的机器上，ZStack 则把所有服务封装在一个被称为管理节点的进程中。文章“进程内的 microservice 架构”解释了我们这么做的原因。

每个管理节点都是一个功能齐全的 ZStack 软件。因为包含的服务是无状态的，管理节点不共享任何状态，但仍然需要维护和其他节点的心跳记录，和一个一致性哈希环，我们接下来将详细介绍哈希环。心跳用来监视管理节点是否正常运行，一旦一个管理节点停止更新自己的心跳一段时间后，其他的管理节点将会驱逐它然后接管他所管理的资源。

无状态的服务

针对 ZStack 的业务逻辑，实现无状态服务的核心技术，是一致性哈希算法。当系统启动的时候，每一个管理节点将被分配一个 version 4 UUID（管理节点 UUID），这个 UUID 将和服务名称拼在一起在消息代理上注册一个服务队列。例如，一个管理节点可能有类似下面的服务队列：

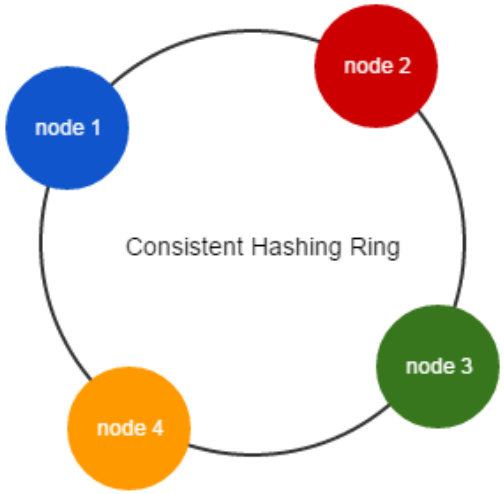
```
zstack.message.ansible.3694776ab31a45709259254a018913ca

zstack.message.api.portal
```

```
zstack.message.applianceVm.3694776ab31a45709259254a018913ca

zstack.message.cloudbus.3694776ab31a45709259254a018913ca

zstack.message.cluster.3694776ab31a45709259254a018913ca

zstack.message.configuration.3694776ab31a45709259254a018913ca

zstack.message.console.3694776ab31a45709259254a018913ca

zstack.message.eip.3694776ab31a45709259254a018913ca

zstack.message.globalConfig.3694776ab31a45709259254a018913ca

zstack.message.host.3694776ab31a45709259254a018913ca

zstack.message.host.allocator.3694776ab31a45709259254a018913ca

zstack.message.identity.3694776ab31a45709259254a018913ca

zstack.message.image.3694776ab31a45709259254a018913ca

zstack.message.managementNode.3694776ab31a45709259254a018913ca

zstack.message.network.l2.3694776ab31a45709259254a018913ca

zstack.message.network.l2.vlan.3694776ab31a45709259254a018913ca

zstack.message.network.l3.3694776ab31a45709259254a018913ca

zstack.message.network.service.3694776ab31a45709259254a018913ca

zstack.message.portForwarding.3694776ab31a45709259254a018913ca

zstack.message.query.3694776ab31a45709259254a018913ca

zstack.message.securityGroup.3694776ab31a45709259254a018913ca

zstack.message.snapshot.volume.3694776ab31a45709259254a018913ca

zstack.message.storage.backup.3694776ab31a45709259254a018913ca
```

备注：你应该已经注意到所有的队列都是以一个相同的管理节点的 UUID 结尾的。

主机，磁盘，虚拟机等资源也有特定的 UUID。和资源相关的消息通常在服务之间传递，在发送一个消息之前，发送者必须基于资源的 UUID 选择一个接收服务，一致性哈希算法这时候就发挥作用了。

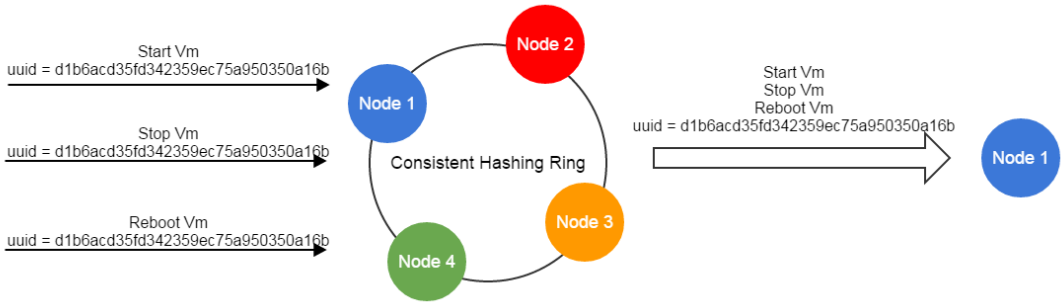
一致性哈希是一种较特别的哈希，当一个哈希表的大小发生变化时，只有一部分键需要被重新映射。深入了解一致性哈希，请阅读 http://www.tom-e-white.com/2007/11/consistent-hashing.html，在 ZStack 中，管理节点组成了一个一致性哈希如下：



每一个管理节点维护了一份包含系统中所有管理节点的 UUID 的环拷贝，当一个管理节点添加或删除的时候，一个生命周期事件将通过消息代理广播到其他的管理节点，这将导致这些节点拓展或者收缩环去描述当前系统的状态。当发送一条消息时，发送者将使用资源的 UUID 哈希得出目标管理节点的 UUID。例如，当 VM 的 UUID 是 932763162d054c04adaab6ab498c9139 时发送一个 StartVmInstanceMsg，伪代码如下所示：

```
    msg = new StartVmInstanceMsg();

    destinationManagementNodeUUID = consistent_hashing_algorithm("932763162d054c04adaab6ab498c
9139");

    msg.setServiceId("vmInstance." + destinationManagementNodeUUID);

    cloudBus.send(msg)
```

有了哈希环，资源 UUID 相同的消息将被映射到特定管理节点的相同服务中，这点是 ZStack 的无锁架构的基础（参考 ZStack 的可拓展性秘密武器 3：无锁架构）。



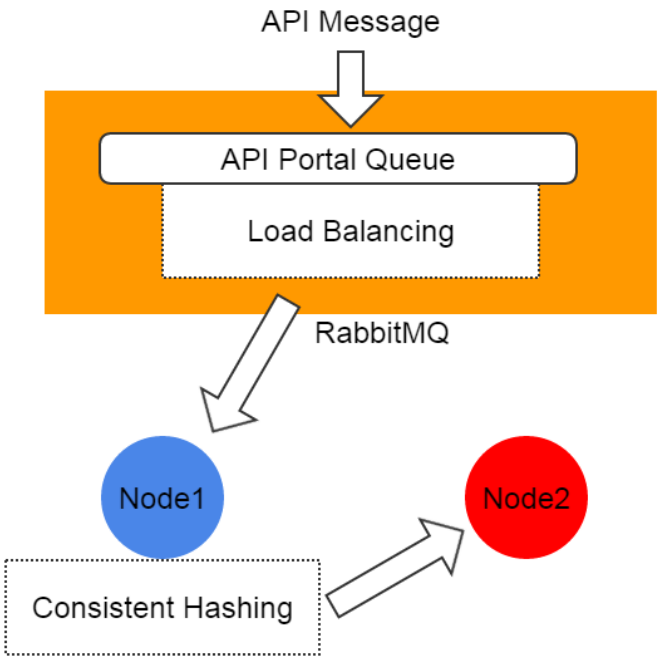
当环收缩或者拓展的时候，因为哈希环的固有特性，仅有小部分节点将被影响。

因为使用一致性哈希环，发送者不需要知道哪个服务实例将处理这条消息，因为服务实例将被哈希计算出来。服务也不用维护、交换他们管理的资源信息，并且因为选择正确的服务实例可以由哈希环完成，服务只需要单纯的处理消息。因此，服务变得极其简单且无状态。

除了包含资源 UUID 的消息（例如 StartVmInstanceMsg, DownloadImageMsg）以外，有一种不包含资源 UUID 的消息，这种消息通常是创造性的消息（例如 CreateVolumeMsg）和不进行资源操作的消息（例如 AllocateHostMsg），因为这些消息可以被发送到任意管理节点的服务中，他们就被发送到本地的管理节点，因为发送者和接收者在同一个节点上，接收者在发送者发送消息时一定是可用的。

对于 API 消息（如 APIStartVmInstanceMsg），有一个特别的处理方法是他们经常和一个重要的服务 ID api.portal 绑在一起发送。在消息代理中，一个称为 zstack.message.api.portal 的全局的队列被所有管理节点的 API 服务共享，带有 api.portal 的消息将通过一致性哈希环把消息映射到正确的服务中，从而实现负载均衡。通过上面这种方式，ZStack 隐藏了 API 客户端消息选路的实现，减少了 ZStack API 客户端代码。

```
msg = new APICreateVmInstanceMsg()

msg.setServiceId("api.portal")

cloudBus.send(msg)
```



总结

本文演示了 ZStack 是如何通过构建一个无状态的分布式系统来进行横向拓展的。因为管理节点共享的信息非常少，建立一个庞大的拥有几十或上百个管理节点的集群是非常容易的。然而，在现实中，对于私有云，两个管理节点足够满足高可用性和可拓展性的需求。对于公有云，管理者可以依据负载量大量创建管理节点。因为异步架构和无状态架构，ZStack 能够处理现有的 IaaS 软件处理不了的非常大的并发任务。

ZSTACK--可拓展性秘密武器 3：无锁架构

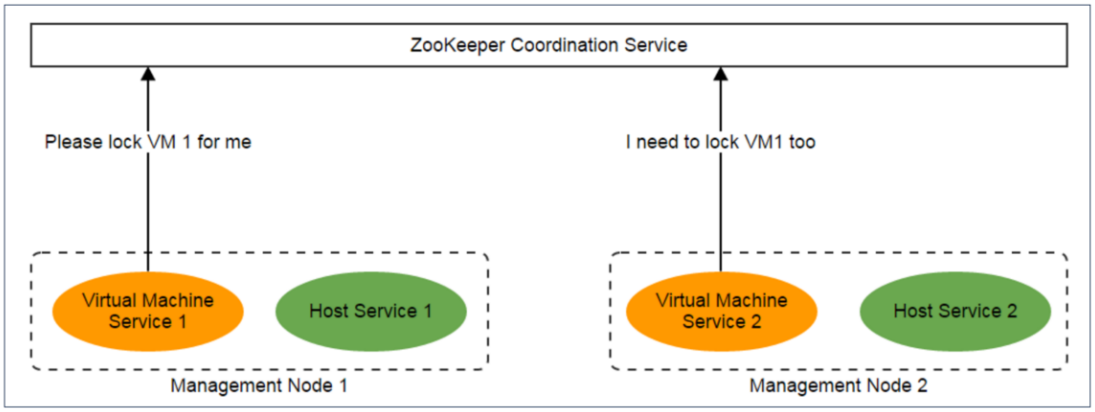
很多 IaaS 软件中的任务需要按照顺序执行，例如，当一个启动虚拟机的任务正在运行时，关闭同一台虚拟机的任务必须等候前面开启虚拟机的任务完成。另一方面，一些任务应该支持并行完成；例如，20 个在同一台主机上创建虚拟机的任务可以同时运行。在一个分布式系统中同时实现串行化和细粒度的并行化并不容易，这通常需要借助分布式调度软件。面对挑战，ZStack 提出基于队列的无锁架构，使得任务本身可以简单的控制他们的并行化等级为 1（串行的）...N（并行的）。

动机

好的 IaaS 软件应该能对任务的串行化和并行化进行细粒度的控制。通常，因为任务之间有依赖关系，任务希望以特定的序列被执行。比如，如果当一个对某个磁盘进行快照的任务正在运行时，那么删除该磁盘的操作不能被执行。有些时候，为了提升性能，任务应该被并行地执行；比如，在同一台主机上有十个创建虚拟机的任务，这些任务可以同时运行并不会产生任何问题。然而，如果不进行合理的控制，并行化会对系统造成一定的伤害；比如，1000 个在一台主机上创建虚拟机的并发任务，将毫无疑问的摧毁整个系统，或者导致整个系统长时间没有任何回应。这种并发编程问题在多线程的环境中是复杂的，并且在分布式系统环境中将更加复杂。

问题

教科书教导我们解决同步和并行问题的答案锁（lock）和信号量（semaphore）。在分布式系统中为了解决此问题，一个直接的想法是使用一些类似 Apache ZooKeeper 的分布式调度软件或一些基于 Redis 的类似软件。拿 ZooKeeper 来举例子，使用分布式调度软件的概览如下：

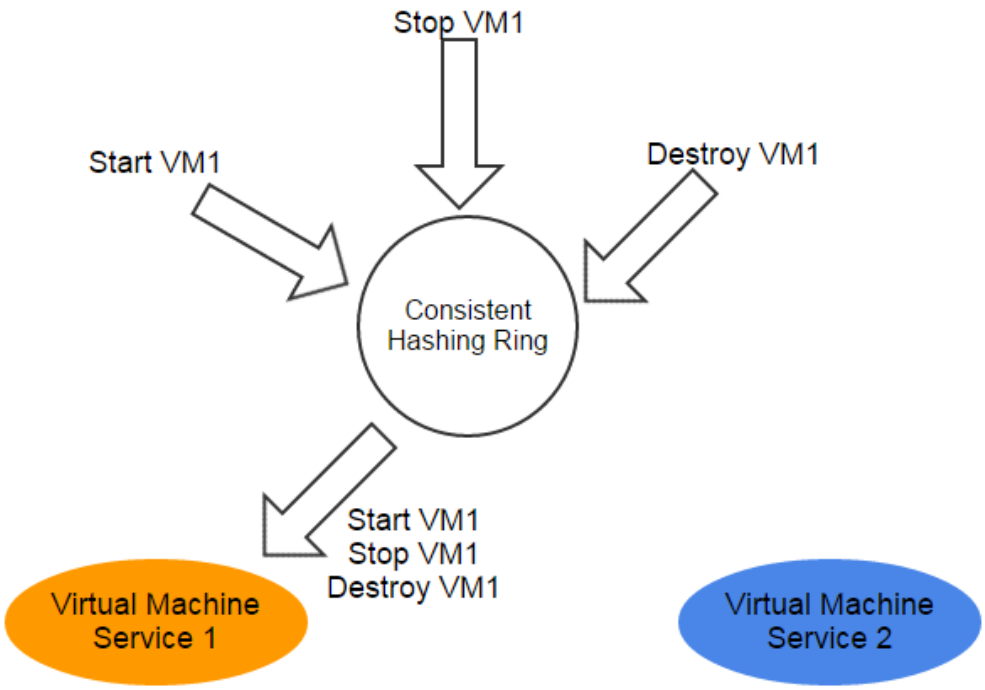


问题在于，当使用锁和信号量的时候，**一个线程需要去等待其他线程释放他们所持有的锁或者信号量。**在“ZStack 可拓展性秘密武器 1：异步架构”中我们阐述了 ZStack 是一个异步的软件，没有线程将因为等待其他线程的完成而被阻塞，所以使用锁和信号量并不是一个可行的方案。我们也关注使用分布式调度软件后，系统的复杂度和可拓展性。假设一个系统充满了 10000 个需要锁的任务，这并不方便处理、拓展性也不强。

同步（Synchronous）与串行化（Synchronized）：在“ZStack 可拓展性秘密武器 1：异步架构”中，我们讨论同步（Synchronous）和异步，在这篇文章中，我们将讨论串行化（Synchronized）和并行化，同步（Synchronous）与串行化（Synchronized）某些时候是可以互换的，但是他们是不同的，在我们的文档中同步（Synchronous）是指一个任务将会在运行时阻塞线程，而串行化（Synchronized）是指一个任务执行时是互斥的。如果一个任务一直占用一个线程直到任务完成，它是同步的（Synchronous）任务。如果一个任务在其他任务执行时不能被同时执行，它是串行的（Synchronized）任务。

无锁架构的基础

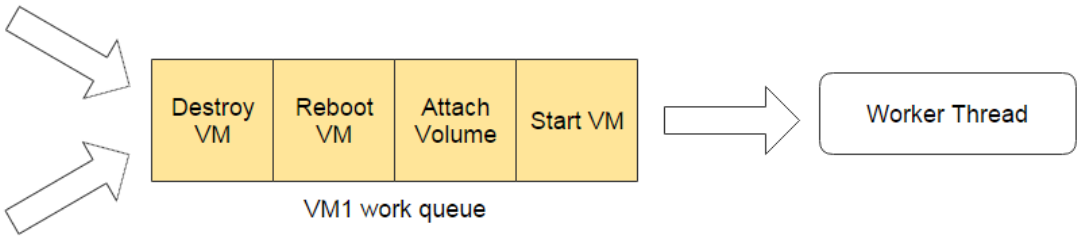
无锁架构的基础是一致性哈希算法，因为一致性哈希算法保证了对应同一资源的所有消息，总是被同一服务实例处理。这种聚合消息到特定节点的做法，降低了同步与并行化的复杂度，因为处理环境从分布式系统变为多线程。（详见“ZStack 可拓展性秘密武器 2：无状态服务”）。



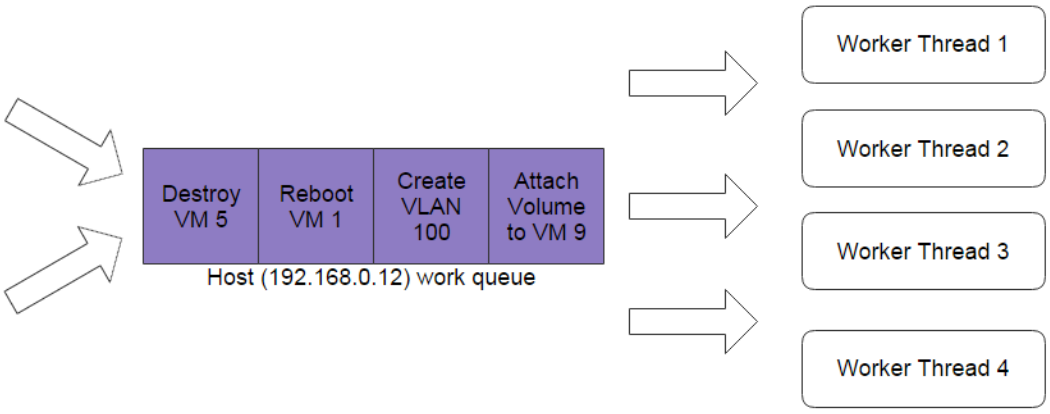
工作流：传统而又清楚的解决方案

备注：在深入讲解细节之前，请注意我们将要讲的队列和“ZStack 可拓展性秘密武器 2：无状态服务”中提到的 RabbitMQ 的消息队列是没有任何关联的。消息队列就是 RabbitMQ 的一个术语，ZStack 的队列指的是内部的数据结构。

因为 ZStack 是消息驱动的，聚合消息使得相关联的任务在同一节点执行，避免了需要使用线程池进行并发编程的经典问题。为了避免竞争锁，ZStack 将使用工作队列而不是锁和信号量的方式。串行的（synchronized）任务以工作队列的方式保存在内存中，最终一个个的被执行。



工作队列在保存任务的同时也保存并行度，使得并行的任务能以一个定义好的并行度执行。下面的例子展示了一个并行度为 4 的队列。



备注：串行的（synchronized）和并行的任务都可以被工作队列执行，当并行度等于 1 的时候，队列是串行的（synchronized）；并行度大于 1 的时候，队列是并行的；并行度等于 0 的时候，队列的并行度不被限制。

基于内存的同步队列

在 ZStack 中有两种类型的工作队列；一种是同步的（synchronous）工作队列，即一个任务（通常是一个 Java 可执行程序）在它返回后才被认为已经完成。

```
thdf.syncSubmit(new SyncTask<Object>() {

    @Override

    public String getSyncSignature() {
```

```
    return "api.worker";
}

@Override
public int getSyncLevel() {
    return apiWorkerNum;
}

@Override
public String getName() {
    return "api.worker";
}

@Override
public Object call() throws Exception {
    if (msg.getClass() == APIIsReadyToGoMsg.class) {
        handle((APIIsReadyToGoMsg) msg);
    } else {
        try {
            dispatchMessage((APIMessage) msg);
        } catch (Throwable t) {
            bus.logExceptionWithMessageDump(msg, t);
            bus.replyErrorByMessageType(msg, errf.throwableToInternalError(t));
        }
    }
```

```
        /* When method call() returns, the next task will be proceeded immediately */
        return null;
    }
});
```

强调：在同步（synchronous）队列中，一个 Runnable.run()返回后，工作线程将继续抓取下一个可执行程序，一直到整个队列为空时，该线程将被放回线程池。因为任务将一直占用一个工作线程，所以队列是同步的（synchronous）。

基于内存的异步队列

另一种异步工作队列的方式指的是，当任务发布了一个完整的通知，就被认为已经完成。

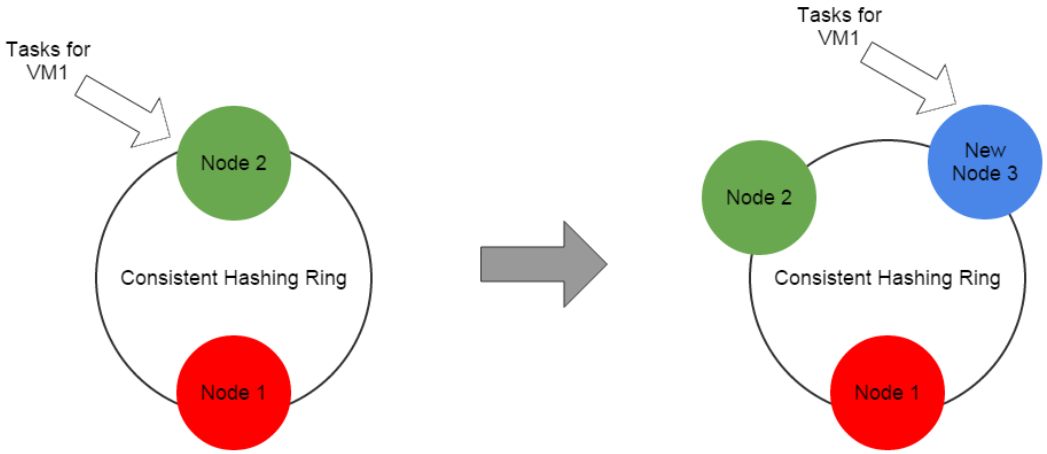
```
thdf.chainSubmit(new ChainTask(msg) {
    @Override
    public String getName() {
        return String.format("start-vm-%s", self.getUuid());
    }

    @Override
    public String getSyncSignature() {
        return syncThreadName;
    }

    @Override
    public void run(SyncTaskChain chain) {
```

```
        startVm(msg, chain);


        /* the next task will be proceeded only after startVm() method calls chain.next() */
    }
});
```

强调：因为使用异步队列，ChainTask.run(SyncTaskChain chain)方法可能在做完一些异步操作后立即返回，比如说，发送了一个注册回调函数的消息。在 run()方法返回后，工作线程立即返回到线程池；然而，工作线程返回的时候，任务并没有被完成，在队列中的任务直到前一个任务发出了一个通知（比如说调用 SyncTaskChain.next()函数）后才可以被运行，因为任务本身不会阻塞工作线程，不会让它等到任务本身完成，所以队列是异步的。

限制

虽然基于无锁架构的队列能 99.99%的解决一个管理节点内的串行和并行问题，一致性哈希算法会导致系统进入混乱的状态：一个新加入的节点将因为一致性哈希算法接管临近节点的一部分工作。



在这个样例中，node 3 新加入后，之前在 node2 上的一部分任务将转移到 node3 上，在这个时候，如果一个与某资源相关的旧任务仍然运行在 node2 上，但是与同一资源相关的新任务被提交到 node3 中，这种情况下就会发生混乱。然而，情景并没有想象的那么糟糕。首先，互

相冲突的任务在一个正常的系统中很少存在，比如，一个健壮的 UI 应该不允许你在启动一台虚拟机的同时关闭这台虚拟机。第二，每一个 ZStack 的资源有他们的状况（state）/状态（status），如果任务开始的时候，资源处于错误的状况/状态，这将会导致报错。比如，如果一个虚拟机在停止状态，给该虚拟机绑定磁盘的任务将立即返回错误。第三，处理很多任务传输的代理（agents），有一个附加的串行机制；例如：尽管我们在管理节点上已经有了云路由的工作队列，云路由代理（agent）将串行所有修改 DHCP 配置文件的请求。最后，提前规划好操作是持续管理云的关键，操作组可以在启动云之前启动足够的管理节点，如果不需要动态添加一个新的节点，当云在运行的时候添加管理节点的几率是较小的。

总结

本文主要介绍了无锁架构，它是建立在基于内存的工作队列的基础上的。ZStack 没有使用复杂的分布式调度软件，尽可能的提升了自己的性能，同时也预防了任务异常的产生。