# ZStack 技术白皮书精选

## ZStack 可拓展性的秘密武器 3：

## 无锁架构

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--可拓展性秘密武器 3：无锁架构

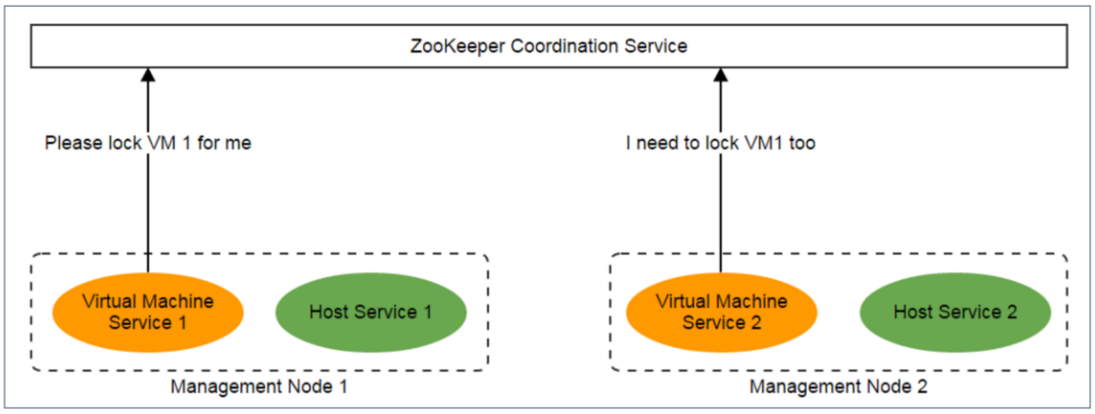
很多 IaaS 软件中的任务需要按照顺序执行，例如，当一个启动虚拟机的任务正在运行时，关闭同一台虚拟机的任务必须等候前面开启虚拟机的任务完成。另一方面，一些任务应该支持并行完成；例如，20 个在同一台主机上创建虚拟机的任务可以同时运行。在一个分布式系统中同时实现串行化和细粒度的并行化并不容易，这通常需要借助分布式调度软件。面对挑战，ZStack 提出基于队列的无锁架构，使得任务本身可以简单的控制他们的并行化等级为 1（串行的）...N（并行的）。

## 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：

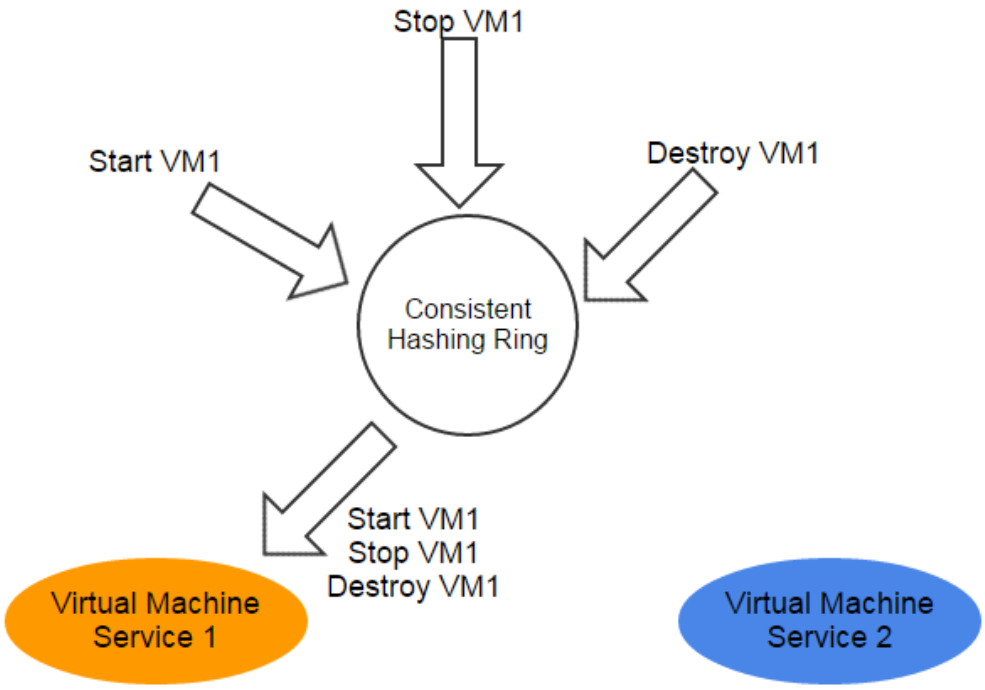



问题在于，当使用锁和信号量的时候，**一个线程需要去等待其他线程释放他们所持有的锁或者信号量。**在“ZStack 可拓展性秘密武器 1：异步架构”中我们阐述了 ZStack 是一个异步的软件，没有线程将因为等待其他线程的完成而被阻塞，所以使用锁和信号量并不是一个可行的方案。我们也关注使用分布式调度软件后，系统的复杂度和可拓展性。假设一个系统充满了 10000 个需要锁的任务，这并不方便处理、拓展性也不强。

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

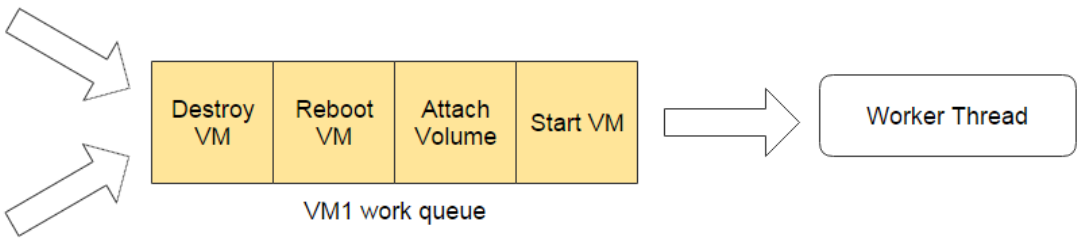
无锁架构的基础是一致性哈希算法，因为一致性哈希算法保证了对应同一资源的所有消息，总是被同一服务实例处理。这种聚合消息到特定节点的做法，降低了同步与并行化的复杂度，因为处理环境从分布式系统变为多线程。（详见“ZStack 可拓展性秘密武器 2：无状态服务”）。



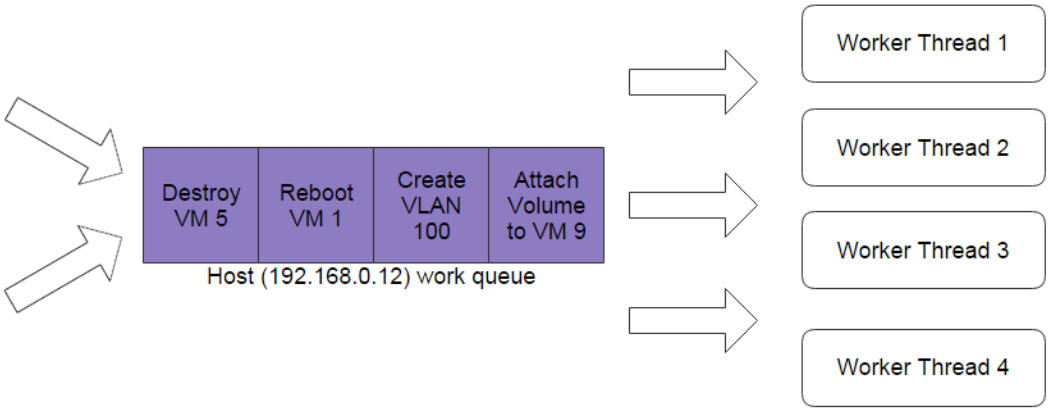

## 工作流：传统而又清楚的解决方案

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。

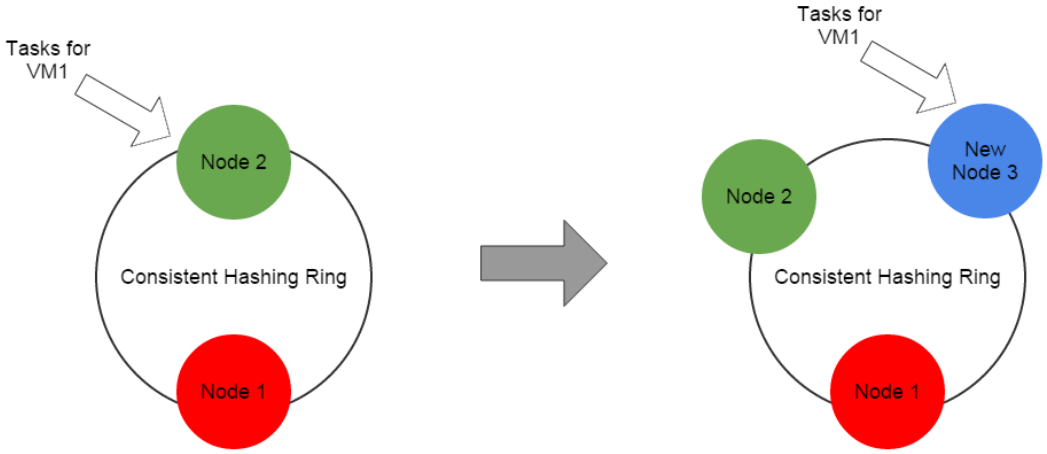
```
thdf.chainSubmit(new ChainTask(msg) {

    @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。