亚马逊 aws 迷你书

设计模式

高可用性

运维

监控与自动化

高弹性

卷首语

云带来的改变显而易见，高可用、高弹性与高扩展性、减少运维成本，然而并不是任何随便一款应用都可以享受云带来的这些好处。应用的架构要针对云作出调整，高可用、高弹性也是有条件的，还要学习在云上运维监控和开发的技巧。作为全球最成功的云服务商，亚马逊 AWS 正在努力将云的优势发挥到最大化。本期迷你书《设计模式、高可用性、高弹性、运维、监控与自动化》将 InfoQ 过去一年中的亚马逊 AWS、高可用、监控与自动化等内容整理，重新编排呈现给大家。

迷你书主编 包研

目录

基于 AWS 的自动化部署实践

作者 徐桂林

1. 背景

在过去几年里，社交、移动和云计算深刻改变了整个互联网的格局。作为设计软件领域的全球领导厂商，Autodesk也与2009年正式开始从传统桌面设计软件提供商向在线服务、协作和移动端设计转型。在这次转型中，公司充分利用现代云计算的巨大优势给客户带来了大大超过传统桌面软件的处理能力、用户体验和性价比。其中AWS是目前公司服务的主要运行平台，每年在此投入千万美金级别。

1.1. 传统软件交付的挑战

在过去的30多年里，Autodesk拥有了非常多的桌面设计软件（如AutoCAD，Maya，3dsMax等）。由于设计软件经常需要处理超大的数据集合（如整个上海中心的设计模型）和极其复杂的运算逻辑（如阿凡达电影画面的绘制），其软件尺寸一般都比较大（GB级别）。以前，客户基本都是通过互联网下载、快递或者分销商获得软件安装包，整个过程比较耗时。另外，软件的升级、安装和维护也是一个非常大的工作量（一些大的设计公司要购买上千份软件拷贝）。尽管公司软件已经支持基于应用程序虚拟化的集中管理模式，但它还是有如前期基础设施建设成本大，服务能力缺乏弹性，仍需要专职运维人员等明显的缺点。所以，提升软件交付的用户体验成为我们必须要面对的一个问题。

如大多数人所猜测，SaaS成为我们的第一个尝试方向。在2006年，Autodesk实验室开始尝试以SaaS提供设计软件服务的可能性，并且取得了不错的成绩。但是，目前SaaS应用仍然面临着浏览器能力限制、大数据传输慢等诸多问题，无法给专业设计师提供传统桌面软件一样的体验和设计能力。

1.2. 云计算带来的新可能

随着云计算的兴起，以AWS为代表的基础设施云提供商让我们有可能以一种全新的方法去解决这个问题。我们可以利用基础设施云的强大后台来帮助用户运行虚拟化的软件实例。用户无需下载、安装和维护这些软件，只需要链接到互联网上就可以在线使用我们的软件。而且按需付费以降低使用成本。基于此，Autodesk实验室在2009年开始这个尝试（注：该服务已经于2013年成为公司云平台产品的一部分），并选择了AWS做作为我们的后台云服务提供商。选择AWS有下面的几个原因：

- 需要基础设施云（IaaS）。现在的平台云（PaaS）大部分都是为Web服务准备的，并不适合我们运行虚拟化实例的要求。
- 需要强大的弹性运算能力。Autodesk设计软件对于计算的需求都很大，而计算能力的成本不低。所以，弹性计算能力能让我们很好的控制成本。AWS EC2在这方面非常符合我们的需求。
- 需要丰富的服务。除了运算能力，我们还需要给用户提供海量设计数据的存储，快速的数据访问，安全的访问控制等。AWS云服务在这方便服务非常完备，而且相互集成很容易。
- 需要稳定的服务。AWS EC2能够提供超过99.5%的弹性计算可用率，能为我们建立可靠服务提供坚实的基础设施。
- 需要全球化部署。Autodesk是一个在全球提供软件、服务的公司，所以我们希望基础设施提供商也能全球布局。而AWS已经在全球建立多个数据中心。

2. 自动化部署

在完成服务的基本实现并上线服务后，整个后台的维护和部署成本在不断加大。尤其考虑到我们需要高频（每个月更新一次）、多地（多个AWS数据中心）部署服务并同时需要维持高的服务可用性，构建一个自动化的部署系统成为必须要做的事情。

2.1. 设定目标

作为一个运行在AWS上的服务，我们在设计之初就开始思考AWS给自动化部署带来的新可能和挑战。在我们看来，针对AWS上服务的自动化部署需要特别关注到下面的一些特点：

- 基础设施的服务化。在AWS中，你可以利用类似Cloud Formation服务在很短时间（几分钟）获取你所要的所有基础设施（包括运算、存储、网络、IO等），而这些基础设施已经按照你的要求自动配置、连接好。所以，我们可以让自动化部署把基础设施的管理也包括进来，而这在传统的数据中心模式下很难实现。
- 支持弹性资源。因为需要支持弹性，整个服务要在运行时动态加入新的基础设施，如计算单元等。自动化部署系统需要能让新加入的基础设施立马投入服务。
- 保护数据更为重要。在传统的数据中心，我们可以让部署完全在内部网络完成，在确认好所有的安全配置后再上线。但是AWS是一个公有云服务，你的所有部署其实是在公有网络上完成的。所以，我们必须在自动化部署中充分考虑到数据安全的问题。

结合上面的特点、DevOps的普遍实践和项目的实际情况，我们给整个自动化部署系统定下了下面的目标：

1. 一键式部署：必须尽可能的自动化所有部署过程，包括基础设施的创建和部署。
2. 多环境支撑：必须能够适应于Production、Staging和Development环境。
3. 无服务中断：必须能够无缝的进行服务升级、切换。

4. 随时可回滚：必须可以很容易的回滚到前面的版本以处理意外问题。
5. 安全性检测：必须在确认部署环境的安全设置已经满足条件后才开始做部署工作。

2.2. 整体架构

在确定目标后，我们进行了技术选型，希望能够找到既很好支持AWS相关自动化操作，又能集成DevOps优秀实践的方案（注：那时AWS还没有推出OpsWorks服务）。但最后并没有找到合适的方案，于是决定自己来实现整个自动化部署系统。经过几轮的改进和实现，现在的自动化部署系统整体结构如下：

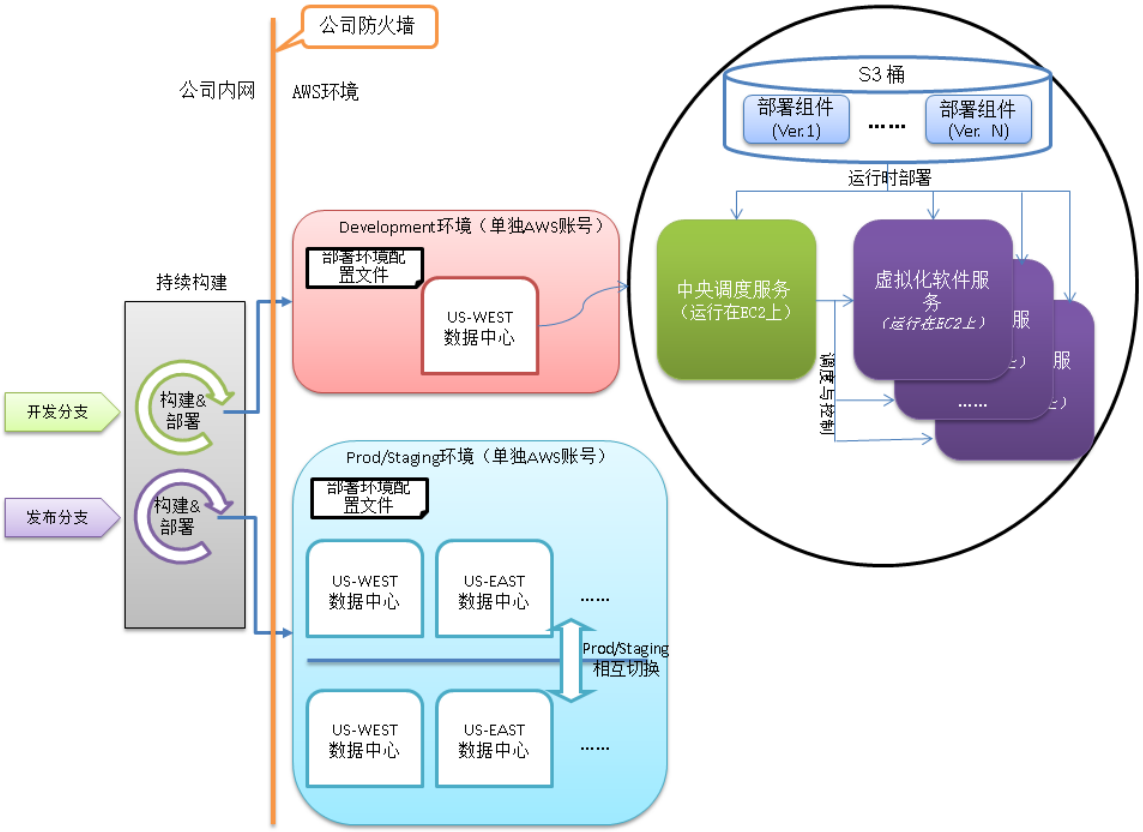

图1：自动化部署系统整体架构

类似于传统的自动化部署，我们也同样有开发分支和发布分支，并与持续构建系统对接。当开发人员提交代码后，相应的构建就会在代码所在的分支自动触发。在完成代码编译和集成的自动化测试（目前主要是单元测试）后，将产生相应部署组件并存放在构建系统中（目前，每个构建会产生所有的系统组件并拥有同样的版本号）。至此，整个构建过程完成。

为了支持多环境部署和安全隔离，我们使用两个独立的AWS账号来运行不同的环境。其中开发环境在一个AWS账户内，只部署开发分支的构建。而生产系统则在另外一个AWS账户中，其下运行Prod/Staging两组环境。发布分支永远只会向Staging环境部署并在完成最后验证测试后与当前Prod环境进行热切换，从而达到无服务中断的目的。

2.3. 一键部署流程

在完成自动化持续构建后，我们就可以部署其中的任意版本。当部署某一构建版本时（无论开发分支还是发布分支），整个流程如下：

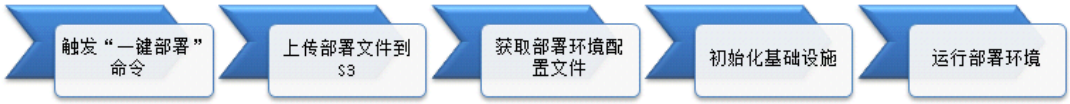

图2：一键式部署流程

- 触发“一键部署”命令：在构建系统中选择好要部署的分支和版本，直接一键点击触发命令。
- 上传部署文件到S3：如图1所示，部署组件是在运行时动态下载安装。AWS S3提供在线存储并能够方便的下载到EC2实例中，所以把部署组件上传到S3中最合适不过。为了支持多版本并存和部署回滚，所有上传到S3的部署组件都按版本号分文件夹存储。
- 获取当前部署环境的配置文件：部署环境配置文件存在相应AWS账户下的S3中。它定义了当前AWS账户下面的部署配置，包括需要部署的数据中心列表，每个数据中心下面的基础设施描述（Cloud Formation Stack）。由于Prod/Staging环境都在同一个AWS账户下，每个数据中心都会有两组Cloud Formation Stack配置（分别用于Prod和Staging环境）。部署系统会选择当前Staging环境的Cloud Formation Stack作为下一步的部署目标。
- 初始化部署目标基础设施：根据当前选中的Cloud Formation Stack初始化整个基础设施，包括启动相应的EC2实例，绑定Elastic IPs等。
- 运行部署环境：在整个基础设施运行起来后，EC2实例第一步就是自动做运行时部署（利用操作系统的启动脚本实现）。具体运行时部署细节如下：

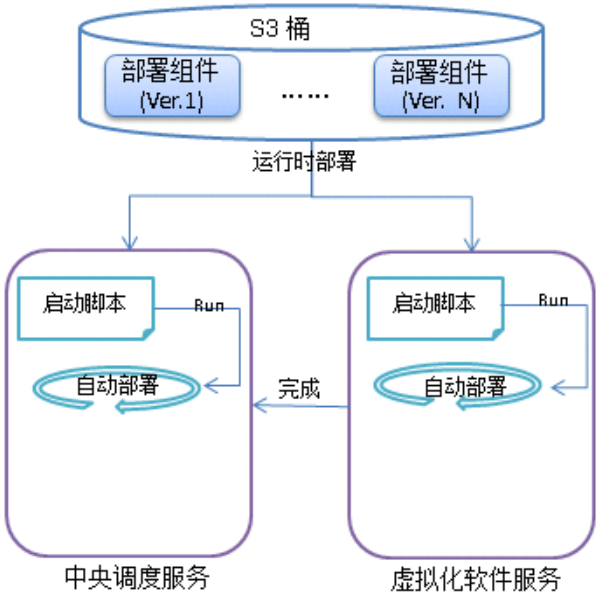

图3：运行时部署

- EC2实例在完成整个自动化部署中要及时进行有效性检测，如检测下载组件包是否完整正确，组件安装是否成功，期望的服务是否可以正确访问等。在完成所有部署和有效性检测后方加入的正式服务系统并处理用户请求，否则及时报告运维团队进行处理。

2.4. 为什么选择运行时部署

看到上面部署流程，相信很多人会问一个问题：AWS不是可以通过虚拟机镜像（AMI）来启动EC2实例，为什么不把上面的部署组件直接烧入AMI？这样在EC2实例启动后就直接使用而无需做运行时部署。其实，我们的最初解决方案就是把部署组件直接烧入AMI，但很快就发现了这种方案的局限：

部署组件的变化是非常频繁的，尤其是在发布前，一天都能够产生上十次的变化。这就意味着自动部署系统可能需要频繁产生AMI。而AMI的创建过程并不快（10分钟~1小时），并且过多的AMI也会造成管理问题和额外的存储成本。

作为一个平台项目，各个产品会在我们的平台上运行。我们希望提供给各个产品部门的基本AMI是平台版本无关的。这样产品部门在基本AMI基础上部署它们产品并重新生成的产品AMI也是平台版本无关的。于是产品AMI可以不做任何修改就能够快速采用最新的平台版本。具体如下图所示：

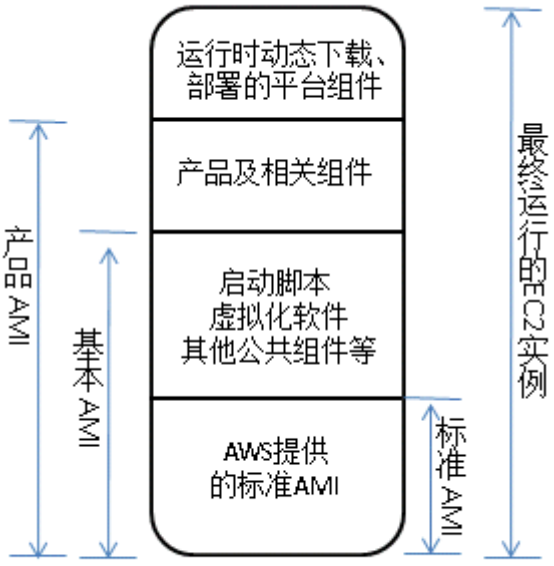

图4：自动化部署中的AMIs

为了保证图4中的基本AMI和产品AMI是平台版本无关，我们就需要保证烧入AMI的所有部分都是能够独立于平台版本的。但是，启动脚本需要做平台组件的运行时部署，显然这个运行时部署脚本也会随着平台更新不断变化，所以我们无法直接把整个运行时部

署脚本放到启动脚本中。针对这个问题，我们的解决方案就是把整个运行时部署脚本分成两部分。一部分做平台组件的安装、检测工作，并随安装组件存到S3中，而另外一部分只做基本不会改变的事情（下载平台组件包并调用自动化部署脚本）并设置为操作系统的启动脚本。这也是图3中启动脚本和自动化部署分开两部分的原因。

正是因为采取了上面的AMI生成、管理策略，我们的产品部门基本上不用太频繁的更新它们产品的AMIs（除非产品自身有更新），真正做到平台版本和产品版本的去耦合，极大的降低了运维成本。

2.5. 运行时版本选择与回滚

由于产品AMI是平台版本独立的，以它为镜像运行起来的EC2实例无法知道应该去下载哪一个版本的平台组件。幸运的是AWS的EC2服务提供了“User Data”机制。简单来说，就是在启动EC2实例的时候可以传入一段用户自定义数据给AWS。当这个实例运行起来后，实例内部程序可以通过调用AWS EC2的元数据服务取得先前传入的用户自定义数据。于是，我们可以把当前需要运行的平台版本号以“User Data”的方式传给EC2实例，然后让启动脚本读取相应版本号并下载相应版本组件来部署。

同样，基于“User Data”机制和平台版本无关的AMI，我们非常容易得实现版本的回滚。只需要修改传入给“User Data”中的版本号并保证要回滚到的平台组件版本仍然存在于S3中即可。即使是从零开始回滚到以前版本也可以在几分钟内完成（主要时间花费在启动EC2实例上）。在实际运营中，因为已经有了Prod/Staging的热切换，我们一般会在切换上新的版本后保持原来的版本运行一段时间（这段时间一般是问题的高发期）以便做到秒级的回滚。

2. 6. 关于安全

如前面所说，我们需要格外关心在AWS上自动化部署的安全问题。在我们的实践中，时刻会遵循最小授权原则并利用AWS中的IAM服务实施，具体体现在下面的两个原则上：

1. 不要让管理员之外的任何人直接使用AWS根账户（包括它的Access Key和Access Security）。取而代之的是创建专门的IAM User并给于其必须的权限
2. 不要在公司防火墙之外使用任何账号的Access Key和Access Security。取而代之的是使用IAM Role来获取EC2实例运行时需要的资源访问权限。

例如，我们的构建系统需要访问多项AWS服务资源（如S3、EC2、Cloud Formation等），而构建系统又在防火墙内部，所以我们创建专用的IAM User来做自动化部署并给予构建系统需要的资源访问权限。另外，EC2实例需要访问S3并下载部署组件，所以EC2应该以专用的IAM Role来运行并在IAM Role中给予相应的S3桶只读属性。

2.7. 为什么不是AWS OpsWorks

熟悉AWS服务的人可能都知道Amazon已经在2013年发布了它的DevOps服务：OpsWorks。我们的自动化部署系统为什么没有选择这个服务呢？最直接的原因就是我们开始构建自动化部署系统时候，AWS还没有发布OpsWorks。在AWS发布OpsWorks后，我们对此做了仔细调研，并决定继续使用目前的系统。要了解其背后原因，就要完整理解AWS提供的一系列应用程序管理服务（Application Management Services）之间的关系。这里，让我们首先看看AWS文档中这张示意图：

图5：AWS中的应用程序管理服务

就如上图所示，AWS为各种应用场景提供了不同的解决方案。由于针对的应用场景不同，这些服务的关注点也就不同。例如，AWS Elastic Beanstalk主要是为Web应用程序提供快速部署服务（有点类似国内SAE、BAE等服务），它帮助开发和运维人员隐藏了大量的底层细节，非常方便上手部署应用。但是，该服务在管理底层基础设施架构上就基本没有多少灵活性，无法适应项目的个性化需求。就我们的服务而言，它开始于最右边的完全自己定制方案。在AWS推出Cloud Formation后，为了提高系统的效率和自动化程度，我们迁移到以Cloud Formation为基础的新方案。但是我们没有继续向AWS OpsWorks迁移。究其缘由，让我们先多方面比较一下AWS OpsWorks和AWS Cloud Formation：

特性	**AWS Cloud Formation**	**AWS OpsWorks**
基础设施架构	以Stack的方式管理整个基础设施，你可以以任何你想要的架构组织你的基础设施。	遵循常见的架构实践，以Stack、Layer和App为核心观念来管理整个服务架构，并利用Chef来把App自动化部署到各个Layer上。尽管它也有很大的灵活性，但是你需要把自己的基础设施架构映射到上面的这些概念中以实施该服务。
AWS资源管理	支持几乎所有的AWS资源。你可以在Cloud Formation的JSON模板	支持主要的AWS服务资源（如EC2、EBS、ElasticIP，ELB

	中方便地加入各种AWS资源。Cloud Formation会自动帮你管理各种资源之间的依赖关系。你也可以自定义一些资源之间的逻辑依赖关系。	等），并利用这些资源构建常见的Layers。当然你也可自己加入一些非内建的资源（如RDS服务）到OpsWorks中来，但是它无法达到Cloud Formation对于资源管理的广度。 另外，目前的OpsWorks仅仅支持Amazon Linux和Ubuntu LTS操作系统，你可以使用这些系统的默认AMI或者在这些默认AMI基础上创建的新AMI。
定制化	支持利用参数改变由资源模板定义的Stack默认行为。	支持利用自定义JSON改变Stack的默认行为。
自动部署	支持各种自动化部署工具（如Chef、Puppet等），但是你需要自己实现所有的自动化部署脚本。	支持基于Chef的自动化部署，并且提供了大量的内建自动化脚本。这些内建的自动化脚本会帮助开发和运维人员完成很多的常见工作（如自动部署Tomcat服务器等）并已经集成到整个服务中去。另外，它同样支持自定义的Chef脚本来实现项目相关的部署。
弹性支撑	支持完全自由控制的弹性策略。用户可以使用Auto-Scaling组加自定义的性能指标来确定Scale-up/down的条件。	提供基于负载（Load-Based）和基于时间（Time-Based）的弹性策略。其中基于负载的弹性策略主要考虑CPU、内存等几个主要因素。
系统监控	支持基于Cloud Watch的监控机制。用户可以监控大量的内建指标并添加自定义的监控指标到Cloud Watch中去。	提供以Stack为单元的整合监控界面，同样以Cloud Watch为基础提供监控服务。另外，它还提供了基于Ganglia的可选监控方案。
安全管理	支持IAM的完整功能。用户可以精细控制各个资源的访问权限。	支持IAM的完整功能。并能方便的把相应的安全策略批量实施。

表1：Cloud Formation与OpsWorks的比较

参考上面的比较和我们目前的自动化部署系统，OpsWorks对于我们仍然有一些限制：

1. 无法支持Windows操作系统。我们的很多服务仍然是运行在Windows系统上。
2. 无法提供自定义的弹性策略。我们的系统实现了自己的调度算法，目前把该调度算法映射到现在OpsWorks中还比较困难。

随着OpsWorks的发展，这些限制未来都可能会被解决。但是，作为一个已经运行的系统，我们仍然需要仔细评估OpsWorks带来的收益和成本以确定是否迁移。如果需要在AWS上面部署一个新的应用，推荐的实践就是如图5从左向右依次选择，并且在确认左面的方案有明确限制的情况下再考虑下一个选择。如果图5中左边服务已经能够解决问题，就不建议自己开发相应的系统。毕竟整个部署系统的开发和维护还是需要一个不小的成本，而AWS提供的这些应用程序管理服务都免费的。另外，在绝大多数情况下，Cloud Formation已经足够灵活以满足你在AWS上部署服务。但是，在某些情况下（例如，同时支持在多个云服务提供商上部署服务），你可能还得选择完全自己开发或采用第三方供应商的方案。

在过去的一年里，我们利用上面的自动化部署系统让整个部署过程的耗时从最初的几天快速下降到几分钟之内，大大降低了的开发、运维人员的负担。如此同时，自动化部署还把部署出错的可能性降到了一个极低的水平，提高了系统的整体可用性。

3. 总结

在上面的文章中，我仔细介绍了整个自动化部署系统，并讨论了怎样充分利用AWS服务的特点来提高自动化部署的效率。该系统运行高效、稳定，而且极大程度的降低了平台自身和运行于平台上产品的运维成本。接下来，我们仍然会持续改进整个系统，其中关注的重点有：

1. 解决各个平台组件之间的兼容问题。我们希望让自动化部署系统能够根据设置的规则自动检测各个组件内的兼容问题并选择合适版本自动化部署，这样，各个组件可以按照自动的节奏（和版本号）独立发布（而不是像现在所有的组件必须以一个版本一块部署）。
2. 开发整个自动化部署系统的控制台界面（Dashboard）。该界面可以让我们自己和产品部门看到系统的目前部署状态及部署历史，并且可以让一键部署在该控制台触发，从而让平台内部的构建系统彻底隐藏在背后。
3. 尝试和AWS OpsWorks的结合。OpsWorks的一个优势就是集成了大量DevOps实践精髓并提供了丰富的内建部署脚本，可以降低自动化部署系统自身的开发和维护成本。尽管如前所述，目前还没有办法向该系统迁移，但我们仍然会密切关注OpsWorks自身的发展，并会在平衡收益与成本的前提下积极尝试和AWS OpsWorks的结合。

查看原文：[基于AWS的自动化部署实践]

虚拟专家座谈会：迈向云开发

作者 Richard Seroter ，译者 王灵军

开发者正在不断体验多种不同的云环境。当在云中工作时，开发者应如何改变他们的思考方式？是否有某些云环境更适合于刚准备入门的开发者？而那些目前尚未涉及云开发的开发者们又如何在此领域获得相应技能呢？

为了回答这些问题，InfoQ就云开发的现状、推荐工具和反面模式与三位意见领袖进行了交流。我们的专家组成员是：

- Adron Hall，通晓多种语言的码农和开发人员布道师。
- Magnus Mårtensson，微软MVP，担任瑞典Active Solution顾问公司的云架构师。
- Andy Piper，热衷于推动Cloud Foundry的开发人员。

InfoQ：是不是一位云开发人员的工具箱相比于普通开发者会有很大不同？如果是这样的话，那么在你们看来，云开发者更依赖于哪些传统的web开发者不会使用的工具呢？

Hall：首先我会定义当听到“云开发人员”这个词时会想到什么。一名云开发人员就是负责这样的代码解决方案的人，解决方案是基于水平扩展的、分布式的、幂等的和异步处理，同时具有可伸缩、高度可用和弹性存储的特点。

我说在回答这个问题时当然应完全根据这个定义。一名普通的开发人员经常是在某个传统的RDBMS数据库的基础之上构建应用，在此过程中他会使用某个框架或是其它基于此框架之上的工具，并受到垂直扩展的限制。这并不是不好的开发方式，但是对于云或其它任何可水平扩展的环境来说，以这种方式来构建应用或服务效率会非常低。一旦达到最大物理扩展极限，开发者就完全无能为力了，因为他再也没有办法使用任何合理的方式来提升性能。

一位云开发人员会横跨广阔的资源范围来构建应用，他经常将某个应用的功能拆分成更为具体的服务或模块。云开发人员也常需要涉足于某些含有更多语言的工具包，这些语言包括从JavaScript到C#、Ruby或其它语言等。这样做的原因固然经常是出于必要，但在很大程度上也是为了在每个特定工作最适合使用的工具之间提供匹配。

所以，简而言之，云开发人员的工具绝对是不同的。

Mårtensson：云开发人员需要用很多与非云开发环境不同的思考方法来武装自己。较之于其他寄宿（host）选项而言，当处于云部署平台时，你可以很容易地获取到很多东西。良好的云平台会提供一个工具集，这个工具集与你可能使用过的相比显得非常不同。它可拥有“无限的”存储空间，该空间同时具有自动备份、内建的缓存功能、强有力的服务总线和其他特性。

公平来说，你完全没有必要仅为了获得此工具箱而去100%买入云计算。如果需要的话，你完全可以只是使用来自云平台的如服务总线这样的某一项服务。与任何非云环境的寄宿类型不同的是，工具箱中拥有更多“工具”，比如快速弹性、内建容错、故障转移，以及你可以在任何时候按需优化费用的完全可度量的服务等。这些云特点常被引为NIST云计算定义。我认为，云开发人员有非常多的工具待去了解和运用。合适地运用这些工具可以助你构建那些以前看起来近乎不可能的、或者即使可能但也是代价昂贵的解决方案。如果这些特性被滥用的话，那么云开发人员则会冒着失去获得强大力量的希望的风险，甚至于冒着构建了更为昂贵的解决方案的风险。因为能力越强大，责任也越多。

最后，如果我们再多谈论点传统开发人员工具，其实对于云开发人员和其他类型的开发人员来说，这些工具都是极其一致的。按照我的经验，虽然在Windows Azure平台上使用PowerShell可以在云环境中获得很多收益，并且也自动化了构建和部署。但对于大多数其他寄宿情形来说，这种做法也可以获得同样的效果。我只是觉得在一个像云这样的内在的分布式环境中这样做是很自然的。对于任何一个想使用云来让自己能力更强大的开发人员，我的建议是去学习和了解云计算的真正力量。它们是你的新工具，将会助力你实现从一名传统开发者成为云开发人员的目标，你曾经为此有过一段痛苦的时光甚至于以前这只能是一种奢望！

Piper：好问题，这也是我一直仔细考虑的问题之一。Adron在一件事上绝对正确，就是云带来的是规模——无论是就数据本身而言（同时到达的数据量，或存储的数据量）还是就你如何在云环境中为了获得可用性而扩展你的应用至跨地理位置的多个实例而言。也就是说，虽然很多专为这个新时代而创建的工具和技术（如Riak和其他的分布式数据库）纷纷出现，并且与以前比工具箱中出现了很多不同的工具，我总是倾向于认为，开发者使用新的云平台的最重要的一件事，就是忘掉他们过去的假设——换种方式来思考如何部署应用。

给开发人员提供一致的体验是构建能支持云应用的操作系统的目标之一。我们宁愿如此简单：用Ruby或node.js或随便某个工具来编写一个简单web应用，然后本地运行，或者运行在自己的单一服务器上，亦或运行在一个可缩放的弹性的云环境中。所以我们构建一个抽象层来隐藏不同的云基础设施（AWS、OpenStack或无论什么）之间的差异，让开发人员能够很轻松地部署应用。这确实意味着他们需要意识到不能依赖于自己应用的位置、硬编码的IP地址或其他配置，这些都是不好的做法，而应是在编码时应尽可能将代码与数据库（它们有可能会改变）之间解耦。所以思考方式和架构都是不同的，不过最起码很多工具和代码都可以保持不变。

InfoQ：你们认为哪些IDE最适合于云开发？开发者应为些IDE添加哪些东西来增强其云开发的能力？你们对基于云的IDE有兴趣吗？

Piper：很个人的说我是有潜在偏见的（作为一个Eclipse提交者），我很喜欢Eclipse，也是IntelliJ和RubyMine的粉丝。现在大部分成熟的IDE都拥有非常好的插件与云服务进行交互，比如有低级别的AWS Explorer，也有提供了平滑的构建/测试/部署体验Eclipse的Cloud Foundty集成插件。

我曾使用过像Eclipse Orin和Cloud9这样的基于云的IDE和编辑器，它们都非常方便。我很高兴地看到它们演进得也很快，从而可以利用最新的web特性。

当然很可能具有讽刺意味的是，我自己的开发者工具集根本不是以IDE为中心的。我将一天的大部分时间都花在Sublime Text以及命令行上，并且我也是Ubuntu的包管理和OS X的自制软件开源包管理器的大粉丝 :-)。

Mårtensson：听到Andy这么说很有趣，因为我有类似的背景，不过刚好相反。我是一个Microsoft/C#/Windows Azure的开发人员，这使得我在面对IDE环境和开发体验上极具偏见。最近Microsoft在Visual Studio中实现无缝的云开发体验上投入的努力是无与伦比的，而这是我在Visual Studio身上前所未见。它们确实做出了巨大的努力来支持快速和强有力的云开发。Visual Studio中的Server Explorer能够踏足任何云账户并和运行其上的任何服务进行交互。你可以管理它们，监控它们，并可以在云上运行类似于IntelliTrace和性能监视器这样的工具。然后你可以下载结果并在Visual Studio中进行分析。很自然地，你能在本地构建和测试任何云应用，包括最近加入的非超级管理员模式的计算模拟器（Express版本）。随后你可以将应用作为一个单元推送到位于Windows Azure上的自己的网站和云服务中。我在Visual Studio IDE所见到的用于云开发的（读作：Windows Azure开发）部分在Visual Studio历史上是无与伦比的。干了这杯Kool-Aid？是啊！大口地享受它吧！

Hall：很明显有至上之选：Cloud9IDE。

虽然如此，在现在很多情况下IDE看起来有所妨碍。当某个人需要在具有很多种语言和工具的不同环境之间切换时，最容易的就是抓到Sublime或类似的某个东西并运行。

在重量级有像Visual Studio、Eclipse、IntelliJ、WebStorm和其他应用工具等这样的IDE。这些都很好，并且为一种或某几种语言提供了钩子以方便日常的运维编码工作。但是有可能当某一天快要结束时，某个不在那个IDE范围之内的语言突然冒出来，那就毁了你这一天。

另外一方面，如果一个团队能够专注于某个特定IDE，使用它来加载任何开发所需要的一切，并且IDE能够和首选的云环境协同工作，那么Visual Studio和其他几个IDE就会脱颖而出。一个很好的例子就是用于Visual Studio的AWS的.NET插件。这个特殊的插件是我所见过的仅有的没有将开发者绑定于实际云的工具集。它只是极大地简化了部署和查看云服务，这些都不用离开Visual Studio。对于.NET开发者来说，这是极大的好处，因为他们总是被鼓励并习惯于在云开发过程中一直呆在一种IDE中。

然而，在大多数web开发和云环境中，你会推送一些东西并使用像SSH这样的工具。在这种情形中Visual Studio和Windows通常都不是首选者（non-starter：比赛中不是首发的队员）。让Windows和Visual Studio与SSH和其他Linux或相关环境一起工作所花费的时间会非常让人沮丧。此时，非常戏剧性的是，像下面这样做反而容易得多，抓到一个终端，学会如何摆弄它，然后SSH连接到在这些终端，实际上就可以工作了。即使在使用Windows Azure，如果你并不打算在Windows上寄宿或也不想使用Visual Studio的至

Azure的私有钩子，那么完全可以甩开基于Windows的开发工具转而使用运行于Linux或OS-X之上的来自于JetBrains的工具。从这些工具上获取好处，你的开发团队最终会感谢你。云毕竟是源自于Unix并将在多年来已成为Unix技术世界一部分的很多理念的基础上继续演化着。

InfoQ：当开发者在构建云规模的应用时，应避免哪些反面模式呢？云提供商又如何避免你们犯这些错误，或反过来说，让你们做些错误的事情？

Hall：在开发云分布式应用时，开发者会掉入很多巨大的陷阱之中。

我曾一次又一次看到的最大问题是，他们更愿意搭建单一的数据中心（比如AWS，一个区域等）。使用局限于某个数据中心的框架栈和故障转移所构建的某个应用仍然趋向于引起很多停机情况，比如“East 1 AWS Failure”，绰号为复活节故障。

另外一个很大的问题就是很多提供商——好吧，也许是所有的提供商——仍然还有很多SPOFs（单点故障）。Windows Azure在几个月前由于其证书问题而打破了大数据中心的故障记录。AWS也出现过由于数据中心的某个网络设备而导致的故障。Rackspace同样也有类似的问题，也鼓励了人们使用机架设备，这又在已有的其他故障点之上生成了更多的SPOFs。整个想法是想让云架构具有弹性，而这些情况适得其反。

从纯粹的开发者角度来看，最大的错误在于很容易在云的计算和存储环境中只是简单地构建一个传统的垂直堆叠在一起的应用。在云环境中使用按照传统架构信条所构建的应用会付出高昂的代价昂贵，并且效率很低。然而一次又一次，我看到人们只是重新实现某个垂直设计的应用；Sharepoint，WebSphere和其他所能想到的。他们通常只有单一的RDBMS，在此之上有个数据层，以及一个或者可能是几个应用节点，而这几个应用节点却位于某个有着SPOF的缓存层之上。

总的来看，云开发中有很多反面模式，而运维和开发总是很容易实现它们。

Mårtensson：一个真正而又常见的反面模式就是将老的思考方式应用到新的范式上。比如认证；“我们想让顾客能够使用他们自己已有的Google，Yahoo！，Facebook和Microsoft帐号来认证我们的服务。当然我们也需要标准的用户名/密码登录方式。”你来找出其中的缺陷！基本上就是他们说想要设法变得很时髦，然后又想通过将自己的祖母带到聚会上来回归保守。如果这确实是你的开发和维护工作所要关注的，当然你可以运行并处理自己的用户名/密码认证服务。但如果你正在采用所有的方式来外部化自己的应用的认证，那么就需要通过将自己寄宿的认证服务（术语也称为安全令牌服务 STS）从你的应用中分离出来，然后才能真正实现这些方式。简而言之，你的应用不应关心任何认证。然而它应拥有一个可信任的、为你处理这个单独的关注点的标识符提供者。如果你不保存密码，则你根本不用保留此职责。最开初的表述是害怕由于不提供的标准的用户名/密码认证选项从而失去顾客。但这只会增加你的工作负荷，并且你将会失去使用云服务进行外部化认证的好处。如果你不能公正地对待这种新的模式，除了在开始就以完全错误的方式来使用它这种坏处之外，你最终还将会损害自己的业务。

Piper：好吧，说到现在我已经很难在Adron和Magnus所阐述的常见反面模式之外再添加其他内容了——这些我都见到过。有个很明确的倾向就是以传统的方式思考并构建垂直而不是水平扩展的应用。在这样的情况下，当一个开发者“发现”像RabbitMQ这样的异步消息传递和像Redis这样的内存中的数据结构，然后想到，哦，这是一种“全新”的做事方法时，我总是很吃惊。不，完全不是，这些概念已经出现了很长一段时间，而云平台比曾经任何时候都提供了一个幂等的、最终一致的服务模式。

在云提供商如何帮助你避免失误方面，就我自己来说，Cloud Foundry尽了其最大努力尽可能地来提供作为其环境一部分的有用配置信息，所以你可以编码来从配置信息中去查找值而不用硬编码端口、数据库设置等。这并不妨碍你硬编码，但是这样做确实不是一件好想法。

InfoQ：Redmonk的Stephen O’Grady曾经说过“云计算的最重要特征：进入的低门槛。”你们认为哪些云对于开发人员来说学习会更容易一些？不仅只是让开发者注册，而且还向开发者展示了该如何开始部署应用这些方面，谁工作做得好？对于开发人员仍然还存在哪些进入门槛？

Mårtensson：在瑞典我们说“"tala i egen sak”，意思就是说你是偏颇地来表达自己关于事情的看法。再次说明，我是一名虔诚的.NET/Visual Studio/C#开发人员。当然就会偏向Windows Azure云。仍然……哦！不用暴露我的年纪，我在这个持续的消防比赛中已经有很好的数十年的经历，而且我还从来没有在Microsoft看见过像这样的事情。也许巧合的是，微软的云的气息到来适逢其时？也许只是因为这就是应当做的？但我还从来没有在Microsoft身上见过如此致力于推动技术变迁的情况。并不仅是VB和C#，还有很多。Microsoft为现在的Windows Azure的多种平台和IDE构建和维护了工具集和SDK。PHP、Java、Node.js、Ruby、Python & Visual Studio、Eclipse和开发一次并可给很多不同类型设备发送消息的能力；这些设备有iOS、Android、Window Phone和Windows 8。挑选你的毒药吧！今天谁是对Linux最大的开源贡献者？是的，这就对了，但是谁又曾知道这些呢？对于那些泡在这个空间的人以及那些知道Microsoft的人来说，这是一笔非常大交易！这是一个拥抱多元化的全新的Microsoft，并且它意识到许多公司为了日常运营将会使用很多服务和平台来构建自己的现实世界。这就是我们现在所处的情形，而Microsoft就在这儿。有哪个其他的栈/平台能匹配这个呢？

Hall：对于任何PaaS（平台即服务），进入门槛都尽可能如你所能达到的一样低。当我们深入探讨这些时，这会得到一些奇怪的比较结果。Window Azure据称首先是PaaS，然后才是IaaS（基础设施即服务），它以这种方式开始，然后努力推广它。AWS甚至都没有提到过PaaS这个词，即使他们拥有很多提供了PaaS风格的单一命令行（某些时候要点击）部署方式的服务和特征。而对于其他那些没有一个明确的PaaS说法的环境，门槛过多，而且很笨重，并且我觉得不太值得提及。所以对于那些没有一个合理的PaaS说法的我将会在其他时候再讨论。

这带来了另外一个关键之处，在AWS上实际运行的所有PaaS服务都是怎么样的。Heroku、EngineYard、AppHarbor、AppFpg和几乎每个Cloud Foundry或OpenShift PaaS服务都安装在AWS上。所以很明显，如果某个应用包含了AWS的服务和AWS提供了PaaS

服务的客户，那么它在可用性方面就大幅领先于其他人。即使我们回溯并只是看看首批AWS客户的安装和使用，这些客户在使用Beanstalk、EC2或S3，其安装和使用都极其简单。签约、检查认证机制或类似于通过邮件发送的编码令牌，然后安装好上面所提到的项目之一并启动，你就已经在运行一个应用了。

可以说最严重的障碍都在基于Heroku、EngineYard、Cloud Foundry和OpenShift的PaaS中被移除了。在上面这样PaaS环境中可以很容易地安装、使用和部署应用，这些恰好都是位于AWS中。

但是对于Windows Azure，Windows Azure团队和努力将这些云选项从一个“绝对不”移动到“嘿，这是非常容易的且功能丰富的”。Windows Azure，我不会再为回答这个问题而谈论其过去，它已经戏剧性地重新定义了如何更贴近部署，并积极地比几乎其他每一个PaaS或IaaS服务提供商提供了更多的部署选项和部署能力。另外其已转变为多语言的，在这场由AWS扮演兔子的龟兔赛跑中获胜。比如AWS节点Beanstalk功能。在Windows Azure能够极为容易地、优雅地和极好地在AWS上部署基于Node.js的应用差不多一年之后，AWS才实现。加上它可以围绕Node.js来定价，对于小型的共享的云寄宿模型来说，.NET、PHP和其他的应用已经为零。这对于开发者来说当然非常棒，他们可以在运营之前测试、调试大多数的应用。

总的来说，上面几乎涵盖了我个人所使用和一直关注的主要提供商的基本内容。Windows Azure、AWS、Cloud Foundry、OpenShlft、Heroku、EngineYard都是现在值得关注的主要公司，它们正在这个空间内做着繁重的创新工作，并正在逐渐地前进以移除更多的进入门槛。

Piper：我喜欢Redmonk的这些家伙！他们都是非常聪明的人，并且我推荐那些任何不熟悉他们意见的人去寻找他们。开发者都是新的拥护者。

所以，是的，我完全同意进入的低门槛对于开发人员快速采用我们所讨论的云技术来说非常重要。实际上，这就是某个“啊哈时刻”，这可以让我从自己以前在IBM的角色转换到VMware的Cloud Foundry上来——本地编码应用的能力，无需任何改变地将其推送至云上，然后在几秒钟内将其扩展。鉴于我的角色，不用想就我可以说，很明显，Cloud Foundry进入门槛很低……但就如Adron所说，我想很多相似的PaaS提供商都是这样，如Heroku。我同样对为所见过的在Visual Studio和Azure环境中开发者所展示的工具集成所惊叹，在这个环境中开发者会觉得走向云的旅程十分愉快。我应该指出的是，很多云提供商，特别是PaaS提供商也拥有工作得很好的IDE插件。然而对于真正地低进入门槛，我仍然喜欢源自于Github命令行方式的克隆，本地构建和测试，然后从命令行推送至云的体验，Cloud Foundry和几个其他云提供商都提供了这种——这比需要一个IDE更轻量级。

InfoQ：对于开发者来说哪些事情在过去曾经是很难的，而现在由于云变得简单多了？而且，有没有某些事情在过去是简单地忽略掉或者根本不做，而现在变得简单明了的？

Mårtensson：这是一个大局观的问题。当我们开发时什么是重要的？我们会在某一天回忆起云之前的“黑暗时代”并问自己在那个时代我们是曾经如何让一切运转起来的吗？构建一个全球可伸缩的供几十万甚至上亿用户使用的解决方案确实不是我们大部分人都曾经做过的。但是我们确实可以做到这一点。如果我们业务比像Facebook那样构建自己的数据中心要小得多的话，那么能有机会看到过自己能做到哪一步吗？如果有一个了解了云平台的力量的像样的架构师的话，我就敢说再开发这样的解决方案甚至都不再是什么难事了。现在没有新成立的公司会说“好吧，我们现在获得一笔风险投资，让我们出去采购一些服务器回来。”如果我们展望未来并设想我们开发将会所使用的环境，我确信CPU的能力、内存的大小、存储容量、甚至互联网的速度对于我们构建应用的方式的重要性会越来越小。相反我们会开始依赖于总会有足够的容量给我们正在做的无论什么东西，以及能满足我们的服务所要求的无论什么样的使用模式的需要。当然事情在未来仍然会出现中断，并且某些时候服务会出现故障。但不应会出现这样的情形，即我们不得不恐慌地冲到市中心去买一个全新的硬盘驱动器。我们的服务将会是自愈的。在这个大局观中我们会使用新的模式来开发，这些模式从开始就会把所有这些问题都考虑进来，而且我们从来也不会再关心是否拥有足够的计算能力。

Hall：我将建议的是排在前三名的事物：

- 已经影响到开发人员能如何来开发的最大的影响是能力，即仅需在这儿或那儿点击一些选项或某个脚本就可以让整个运行开发环境跑起来。服务器、测试服务器、UAT服务器等。在以前，即使在简单的虚拟化下这些在很多方式上都会受到限制。但是现在，在拥有AWS、Azure以及其他类似云环境所提供的云计算能力的情形下前面那些都完全不再是问题。以前需要耗时几小时或几天甚至几个月的事情现在几分钟之内就可以完成，并且以一种能向前移动并保留全部努力的方式工作，而这种方式在6-7年前完全无法想象可以这样做的。
- 跨地理边界的分发系统的能力，这在6-7年前，即便不用花费数百万美元资本投资，也会需要数十万。现在每个月仅只需要几百或几千美元，一个庞大的、跨越广为分散的多个国家的分布式系统在几分钟之内就可以安装并运转起来，并准备好投入使用。
- 与垂直叠高的方法相比，分布式系统正在成为普遍的做法。随着这种改变，隐藏在幂等、弹性、自愈、异步、可伸缩、高度可用性系统背后的思想在大大小小公司的绝大多数程序员中出现。随着越来越多的开发者转向水平扩展的做法，垂直扩展背后的极端受限的设计逐渐地被扔到路旁。随后一系列很多用来增强利用这些功能的能力的语言和框架应运而生。这种观念模式和方法的变化一直在持续进行着，并日益增强着云计算的能力。

……这就是在我脑袋中立刻能想到的排在前三的最重要的事物。现在已经发生那么多的变化，获得了很多的进展，以至于关于这个话题有人能写一本书了。

Piper：对于开发人员来说，云让哪些变得容易呢？

- 按需的、潜在的可自由支配的环境。实际上，像Vagrant这样的本地工具还一如既往是开发者的朋友，但是能快速提供和千篇一律的克隆环境以及能以通常很小的代价运行这些环境的能力也极具价值。这对于开发和运维活动一直是巨大的推动力——开发者不用再面对运营维护者的突发奇想，开发者曾经得去为他们提供新的环

境。这并不是为了敲打运营团队——新的云工具和环境也为他们提供更多的敏捷性。“作为服务”是*aaS缩写的关键部分。

- 可伸缩性、可用性、弹性。我想Magnus和Adron对这部分已经说得很好。在这里除了将它们归结为这三原则之一之外，我无法再补充什么了。
- 我认为有两件事——“大数据”和“物联网”——因为云的可用性在很大程度上变得更有能力。垂直扩展的大数据库这许多年来已经成为可能，但是具有海量存储容量、复制、MapReduce等特性的分布式数据库更倾向于与云联系在一起。传感器的连通性、数据的采集和对数据的响应一直以来都是只以单点为基础，但是现在的开发者已经可以构建复杂的能实现自己想法的系统，使用云结构的灵活性构建的系统只有零或很少几个故障点。
- 上面只是一个快速总结。这是一个技术演变具有深远影响的领域。

InfoQ：虽然很多开发人员都开始花费大量的时间来开发云应用，现实是还有很大一部分开发者在其日常工作中都没有理由去接触云。假定这部分开发者没有充足的自由时间来体验云环境，那么你们会推荐他们做什么以便与最新的云服务和战略与时俱进？你们自己又是怎样跟上这种不停向前流动的云空间的呢？

Hall：对于那些没有接触过云/公共云或刚出现的私有云的开发者来说，我发现两种很有用的方式非常重要。

- 学习一般的分布式系统。这些包括分布式数据库、分布式计算（网格计算等）、通过自动化或大量其他选项进行的跨分布式环境的网络管理。这段时间也出版了很多书籍，这些都能在这上面给予他们很多帮助，因为很多工作都已经极大地学术化了（对那些实际上正试图利用分布式系统的编码人员或运营人员并不是很有用），但其中很多东西在日常的开发和运营中基本上没用。
- 当这样做有用时候，尽量尝试在日常的开发过程中小规模地引入它们。即使没有用到云，从分布式角度而不是垂直式角度出发来构建某个系统也可以拥有强有力、有用性和健壮性这样的特性。下面就是我这段时间来一直坚决鼓励的一个做法的要旨：除非应用只是临时性的，否则请不要开发纯粹的垂直式应用。如果某个应用的期望生命周期超过一年的话，那么请按水平扩展的、可伸缩的架构来构建该应用，这样它在一个分布式系统环境之中也能很好地工作。

Mårtensson：向云迈出第一步很容易。首先我想提醒的是最大的问题是，从广泛和普遍采用云计算方面来看，我们目前处于哪个位置？我是一名云计算的倡导者和忠实信徒，并且我真的很想相信现在我们正准备促进云的大爆炸。这就是说很多即使不是所有的信号都在指着这个方向：培训公司在这上面的兴趣在显著增加，顾问们注意到更多关于云的喋喋不休，更多的客户正在激起兴趣而提供商的市场份额也在持续增长。我实在看不到一些关于正在快速增加地采用云技术的反面迹象。传统寄宿选项仍会继续挣扎求生并尽一切能力来显示自己仍是可替代的选项，但在我看来这只是延缓了它们无法避免的命运的到来。特别如果你认为混合场景也是一种真正的云场景的话，我想我们应也将看到一个快速应用需求。如果我们谈论技术采用生命周期，我认为我们在甲板上已拥有了早先的采用者，我们正站在深渊的边缘，深渊将早先的采用者们与其它分割开来。

云平台的提供商们正在尽力做到很容易就能采用并平滑地过渡到各自产品。比如Microsoft最近对于某些试用场合就取消了信用卡的要求。为了回答这个问题，开始与云

计算同行将会实际上已经是很容易的了。你不需要花费大量时间来上手。你可以仅仅需用几分钟时间来注册并获得一个试用的可运行版本。实际上拥有一个已分配给自己的MSDN订阅的每个人都已在云中有一个个人的开发/测试环境。所需要的就是激活MSDN订阅上的Windows Azure，这大概需要2分钟左右。大量的在线指南可以帮助你将自己的第一个应用部署到平台。例如我博客上的视频演示。确实地如果你有15分钟的话，我敢打赌你肯定能将自己的第一个应用在Windows Azure云上正式运行起来。这可能是一件简单得也许微不足道的事情，但它真的很酷很让人耳目一新。

Piper：我想PaaS所提供的任何东西都是瞄着提供一个无阻力的部署表面——AppEngine、Cloud Foundry、Heroku、OpenShift以及Azure的涉及PaaS的很多方面等等确实都是这样。如果在自己所选的平台上没有尽力提供一种在其上部署应用的简单方式，那么很有可能你开始就没有做对。当你开始寻找自己应用中的可伸缩性和数据访问方面某些问题时，学习曲线上仍然还有很多要去学习。

个人来说虽然我发现像O'Reilly这样的出版社所出版的很多不错的语言和编程指南书籍常常都有好几年的生存期，但由于云领域的快速创新，"云"相关的书籍显然还没有老到有这种火候。只要等待一个月，AWS就会已引入一个全新的API或调降了价格，某个PaaS提供商就会宣布有了新的合作者、插件或功能！这就是说在博客中挖来挖去并跟随Twitter上的那些能推荐很好的链接的家伙会更有用。我发现两个特别的来源是很有价值的。Github活动feed让我能跟随自己的联系人所评级或创建的新的项目、apps和库。DevOps每周简讯（云和开发空间的每周汇总邮件）也是一个获取关于最新进展的摘要信息的有用途径。

作者简介

Adron Hall：软件架构师、工程师、程序猿、码农、分布式系统的拥护者。他是位多产的开源贡献者，积极使用Github来贡献项目。你可以在CompositeCode.Com上了解他的思想，还可以在Twitter上的@adron跟随他。

Magnus Mårtensson：就职于瑞典的Active Solution顾问公司，担任云架构师/开发者。他是Windows Azure MVP、Windows Azure的业内人士、Windows Azure顾问。你可以在MagnusMartensson.com上阅读其著述，还可以在Twitter上的@noopman跟随他。

AndyPiper：倡导Cloud Foundry的开发者。他的日常工作是包含以下内容的有趣混合：技术市场、业务开发、与开发者交谈、各种会议上的公开演讲、撰写文档和示例、向工程师抱怨其中断了事情、博客和推特、还有就是组织活动。从这儿可以了解他更多东西，还可以在Twitter上的@andypiper跟随他。

感谢侯伯薇对本文的审校。

查看英文原文：Virtual Panel: Adjusting to Development in the Cloud

查看原文：虚拟专家座谈会：迈向云开发

云端监控和移动测试释疑——对话 Just.me 工程副总裁 Kevin Nilson

作者 Sai Yang ，译者 刘君

云端监控和服务器管理区别甚远。在2013上海JavaOne大会上，just.me工程部副总裁Kevin Nilson讲述了云端部署的必要措施，其中涵盖一些AWS平台管理操作时的注意事项。

InfoQ就云端监控和服务器管理的具体差异、差异相关的应对方案以及app监控和移动测试的必备知识等方面内容采访了Kevin。

InfoQ: Kevin你好，很高兴你能来。你今早就云端监控做了讲演。那么就你看，从现有服务器向AWS迁移时，监控方面的最大不同是什么？

Kevin: 云端部署会面临很多挑战，其中之一是确知特定时刻正在运行的服务器数量。系统规模会自适应调整，因而无从知晓到底有多少服务器在运行，亦无从知晓这些服务器的性能如何。这会是一种挑战。

另一个特定的挑战在于，云服务建立在商用硬件之上。云中一切共享，但每台服务器并非完全相同。很可能新启动的实例会比你先前使用的实例慢很多。

你要学会以不同的监控粒度查看API性能，一些情况下视整个服务为一体，另一些时候以服务器为性能查看的基本单元。有时，你大致了解API调用的执行流程，却发现本次调用的性能远低于预期。遇到这种情况，你可以试试删除当前实例并重新申请。毕竟，没有人会愿意为远低于应得的性能付款，而这种情况虽不常发生但确实存在。

许多人 – 我曾和Pinterest的首席工程师谈过，他们也曾遇到过同样的问题。我还知道Netflix的那群人在这一研究领域做了不少调研，而我们会继续保持对该领域的关注。

还有一件有意思的事情在于，使用云服务时，你常常会终止实例，而终止意味着完全终结。你会失去一切，包括各种日志以及其他类似物。这是另一种挑战。

云端部署的其他挑战源于服务器管理。在just.me我们使用Graphite，一种类似Ganglia的工具。配置特定数量的服务器实例并不困难，你可以用Graphite对这些服务器进行监控，并以拉模式收集运行时信息。这也是此类工具最常见的用途。不过，在云端使用拉模式意味着每次新实例添加都必须重启Graphite，重新配置并重启服务器。这显然不合适。

或许你可以试着换个角度。不是让服务器监管所有运行时实例，而是打破常规，让全部运行时实例了解到监视服务器的存在。这样，数据会被运行实例推送给负责汇报的进程，而不是汇报进程从服务器上主动获取。

我认为这是云服务的极大进步。行动前确定方向并考虑清楚，将很大程度上简化我们想做的事。

InfoQ: 你在演讲中曾提及AWS CloudWatch会有5分钟的时延。

Kevin: 诚然，AWS为监控提供了很多基础技术。CloudWatch十分出众，它提供了一些基本的警告和通知，并帮你监测一些基础属性，像CPU和网络流入/流出。CPU是我很关注的一点。

CloudWatch的问题在于，借助Amazon提供的基础监控技术，你无从获取应用相关的更深入细节。它只能让你查看诸如服务器状态、网络流入/流出等基础属性。

这是一个好的开端，但实用价值不大。试问，CPU0%代表什么？一切运转正常，还是应用服务器崩溃？可能为最好情况，也可能是最坏结果，而你，对实际情形一无所知。

CloudWatch本身存在不足。试着对其中的部分加以完善，你确实很想监测特定服务及其运转性能，对么？Yammer Metrics是我力荐的一种工具。它可以为你提供定时器，表和直方图，更重要的是，它能让你注释代码，让你监测任何API的运转性能及调用频率（每分或每秒的调用次数），这些信息唾手可得。

这在一定程度上迈出了提供服务相关信息的第一步。但问题在于，你真正想了解的是运行趋势。知道服务运行了400毫秒意义并不大。昨天是仅仅用了20毫秒，还是800毫秒？是服务已步入困境，还是性能有了飞跃？诚然，你无从知晓。

因此，我在just.me借助Graphite获得所有执行信息相关的图表，并与昨天、上月的情况进行比对，进而分析其运行趋势。这样，在软件版本大更新时，我就能从商业运作和系统运维两个维度出发，分析更新是否会影响性能和访问人数。Graphite带有一项十分便捷的设置服务，用于帮你迅速上手并熟悉Graphite。我们对这项服务非常满意。

其后的挑战在于监测各服务性能，并在故障发生时推送电子邮件或SMS以示提醒、通知。我用一款名为Nagios的工具，它非常出众，你可以通过对配置文件的简单修改来定义什么情况下你愿意得到“嘿，问题将至”的警告，什么情况下真正遇到了问题。

挑战的关键在于，你想在问题发生前未卜先知。先知往往会让分析变得简单。但毫无疑问，对顾客而言，故障永不发生更值得期待。

关于just.me的预测分析和全面监测，我所做的最后一件事是，Graphite向你提供了每次监测一项服务的一系列工具。也许你能一次观测五个服务，或是一次观测所有服务，但我更希望看到行为和趋势展望。

Square团队开发了开源软件Cubism。它能在屏幕上实时显示大量数据并推断出运行性能。它最吸引我的地方在于，很多情况下单个服务带来的问题会蔓延至整个应用。可能你会先收到某项服务持续10分钟低效运转的警告，再一段时间后，整个系统非正常运转。警告能帮你剥离出问题根源，关注最初发现问题的服务并在某种程度上引导你更快地接近问题根源。而尽快找到根源才能尽快恢复运转。

InfoQ: 在那么多的监测量度中，你如何选定适合你的量度？

Kevin: 的确有很多重要量度值得关注。你想监测CPU，若要同时关注成千上万事件，可以考虑创建仪表盘，有了它，各类状态一目了然。有CPU运转状态，也有实例运行情况。

监控中最难实现却又必须完成的功能是主动请求。获取哪些API发起主动请求和请求的确切数目的相关信息是最具挑战性也最具价值的事之一。若是访问量很大，当某项服务开始出现问题时，完成相关任务会更耗时，从而带来主动请求数量的上升。总体看，这非常非常有趣，特别是使用Java时，线程池大小固定且每项服务独占线程，一次API调用挂起就能引起整个服务崩溃。避免单个线程带来的服务崩溃十分重要。

很多服务并不关注性能。其中，主要区别在于信息取送时机。当你向服务器传送信息时，几乎可以肯定，后台进程是异步的。如果用户对究竟发生了什么一无所知，那么势必无从得知服务时耗。获取信息或是下载时，用户等待几乎是必然。下行量度中，用户性能体验更重要。

上传，发布信息，就商业角度而言更重要。我们监测各类社交行为，他们何时发布信息，发布多少信息，赞、评论和其他行为，关注人群。我们对这些事件的数目感兴趣，而商家并不关心用户刷过多少次屏。这就像一种博弈。你要了解获取时性能，而商家更关注提交。

Cubism很有用。决定哪些量度应一目了然时，它会是一个好的选择。通过展示顶部紧紧相靠的三色标注，Cubism在最小空间内让一切一目了然。我能以此在屏幕上监测尽可能多的量度并使之在更远处可见。

这更多是从操控角度而非商业角度。从商业的角度讲，你只需紧盯着你想要达成的核心目标就好。

我的仪表盘比起登录时约有30%的变动。我们将那些以往认为会发生问题但实际并未发生的量度移除。而另一些预料之外的量度被排放在办公仪表盘显眼的位置。

InfoQ: 仪表盘保留了哪些量度?

Kevin: 我会关注CPU。CPU信息与监控密切相关，我把所有MySQL CPU，EC2服务器CPU，Lucene CPU以及Neo4j CPU放在一起。它们都显示一个0到100间的数值，超过

75%到80%时，会发生灾难。当我一眼看去，发现没有超过50%的，就会很放心。我并不关心每个CPU的具体数值，这种压缩可以使屏幕容纳更多信息。

我关注负载均衡器，确定顺利运转的实例数量不为0。我也关注主动请求并确定没有出现API峰值。

我关注DynamoDB，对于它，我想谈两点，我并不喜欢为DynamoDB读单元设定的收费方式。在我看，这种方式是非云的。它不会自动调整。基本上，你说，“我愿为100读单元支付”。一旦这100读单元用掉了，系统就开始拒绝返回结果，抛出异常。即便只需要10个，你也要为100单元支付。若是有一个白天活跃但夜晚清静的网站，你会面对不间断的重新配置。我常会惴惴不安于是否有任何一个服务达到了读单元的阈值，届时它将停止显示数据，而你将陷入大麻烦。这真的非常糟糕。

他们为Dynamo配置了CloudWatch，但使用不便。在just.me我们有很多很多量度，要立即打开浏览器监测CloudWatch的所有数据几乎不现实。这也是为什么我要在Graphite的基础上构建自己的工具来集成信息——那些需要上百浏览器方能显示的信息，我像对待CPU一样把他们整合在一张表中。监测读写单元时，我会划定一条基线作为阈值，超出阈值的就是有问题的。

仪表盘中还有什么？我曾排放过很多安全层相关的东西。每次请求都需要通过JAAS安全机制并确认是否授权，而早期的授权机制有过一些问题，那些曾被排放在仪表盘的顶部、前部还有中心，若通过安全验证耗费了10s，那其余部分的网站性能将不再重要，你必须先修正这一部分，不过现在这些都已解决。那些量度已经过时，并从仪表盘中销声匿迹。

我还把公开的工单数目加入报告中，如果被公开的工单数目很大，也许意味着你在这方面的监控是有漏洞的，所以我在仪表盘中安排了监测公开的工单数目的地方。

为了激发职员的士气，我也放入了一些市场量度。我能监测到系统新注册了多少设备以及这些设备发送了多少消息又收到了多少回复。这些信息主要是出于激励团队的目的。我希望仪表盘能吸引他们的目光，让他们对此好奇进而注意到更多其他方面。我们将仪表盘安置在办公室，放在一块很大的屏幕上，所有开发者都能看到。休息时，人们站在监视器下方，好奇而兴奋地谈论这些信息。

InfoQ：就iOS和Android应用的性能而言，你会重点监测哪些量度？

Kevin：当开发者同时开发维护Android，iOS或是移动HTML5三个版本时。作为一款响应式应用，移动HTML5常常是我们的选择，HTML版在最初向用户介绍应用时可以避免安装。也许有人会和他们分享信息，而他们会得到一种很棒的本地化体验，并宣布“哇，我想安装完全版并继续使用”。

得到高度关注的三种客户端后，你会希望找出其间的差别，从商业维度观测其运行，并确定哪一版本带来更多的流量。最大的挑战之一是找到安装来源。那些来到应用商店并

开始安装的用户，是从哪来？又如何得知？也许我们做过大范围的广告推动，为广告投入了很多资金，但这真的有效？投资回报又如何？或许是我们博客上的文章带来了很多访问量，又或许我们应该更贴近那些潜在商机者，以期达成目标。

在just.me，我们并不直接将引导用户去应用商店，我们会向用户展示HTML5绘制的网页，即just.me/gettheapp。让他们去gettheapp，那儿Google Analytics会告诉我们用户如何得知该产品。我们常在尾部添加查询字段，just.me/gettheapp?src=TechCrunchArticle或是just.me/gettheapp?src=emailcampaign。从而根据量度确定最成功的推广途径。HTTP头部中的URL引用也非常管用。我们试图尽量平衡应用商店和gettheapp间的访问。毫无疑问这很有效。

至于其他移动监控的用具，最初我们选了Flurry。这种工具天生适合移动领域。我们把它用于iPhone和Android。它能告诉我们人们的手持设备型号。如果马上关注Android版just.me，你会发现，我们支持1500多种设备。我不可能命令QA，“去测试1500种设备”或是“随机挑选一种进行测试”。我们希望发现并购置用户真正用到的那种设备。

我们支持1500种设备，但公司里只有10到15种独立设备。我确实对开发者施加过不少压力来让他们购置和市场使用者接轨的个人手机。是否流行对我们而言很重要，因为，只有这样QA才能侧重关注人们真正在用的设备。

几周前索尼设备上出过问题，应用上架后我们发现索尼用户无法发布信息。之前我们从未购置过索尼设备。我们曾在百余人中做过beta测试，有意思的是，他们中没有人用索尼。反正就是没有人报告问题。不过，产品发布的第一天，我拨通了索尼热线，让他们尽快送来一台Xperia。我们拿到了机器，并在20分钟内做出了修正，然后上架应用商店。

Android的碎片化问题着实烦心，但能在一个小时内修正并上架，这很神奇。苹果应用商店上架前经常需要近五天来审核新版本。

Android带来的另一好处是风险转移。Google Play在今年的I/O大会上宣称有alpha和beta版本面市。在just.me应用发布经历三阶段：alpha，beta和成品。Alpha版本面向那些我很熟悉的人，比如我的酒友们，他们也能在手机上调试应用。如果我想上市新产品，并觉得存在风险，我会在alpha阶段停留几日。让熟悉的人试用，看看运行状况。这样，即便发生了可怕的错误，也没什么可担心的。

接下来是beta。这一版本面向那些登录过网址，并称“嘿，我对你的产品很有兴趣，我很期待上市的那天”的参与者，也许会有几百人，我们通过Google+小组进行管理。我把应用分发给他们，这一阶段发生错误也许不太好，但并非灾难。由于产品尚处于beta阶段，应用商店中无法评分。你肯定不想因为发布过早而获得1星评论。这一阶段Android已经做出很多真正值得开发者关注的工作。

就前面提到的索尼案例而言，我们采取的措施是马上登录应用商店使其不支持所有索尼设备。若是有手持索尼设备的人进入商店，它会被告之“你的设备还未被支持。稍后再

来”。我们将索尼设备下线了四天来等官方承诺的送货日期。然后，我们完成修复，并将索尼设备重设为支持态。应用商店当真带给你很多便利。

在Apple平台上进行发布时很有意思的一件事在于如何进入编辑推荐区。Apple希望推荐时能看到你为产品引入很多特色。他们并不提倡每星期提交代码。不被推荐的最好方法是每周都向顾客推出新功能。而理想情况是一年发布三到四次。

当你完成一项特性的开发，你会很想立刻将其发布出去，认为这会挽救你的公司，带来商机，闭合下行困境而使相关指数上扬。但我们需要忍，因为要多攒点新特性一起发布才能上编辑推荐区。这很痛苦，却无疑是移动领域成功的关键。

回到Flurry的使用。最近Google Analytics发布了新更新Universal Access。Universal Access回归到Android和iPhone的本质，让你能做移动分析。此前，他们为JavaScript提供API。

我们用Google Analytics替代Flurry来获取量度，但获取数据前你必须真正理解Google Analytics。一般职员并不会去获取Google Analytics的各类数据。市场上只有真正想找到数据的人才能得到这些，而普通开发者并不会闲逛并宣布“噢，我洞察到了”。这不会发生，没有人会愿意让他们耗费那么久，因为标签，动作或是值集的获取并不容易，甚至看上去设计得非常不人性化。

我们开始使用另一样工具Mixpanel。这是一家初创企业发布的付费工具，我在JavaOne讲演中并未提到。它相当友好。API有点像Flurry，但更擅长同期群分析和路径分析。你能观察到是谁做了这些，是否还有其他人在做，你能找到那些使用产品一周以上的长期用户。基本上，它更关注长期的客户分层而非单纯统计值。

举一个简单的例子：如果有人以每周一次，每天五次的频度注册并登录应用，你能得到一个多少人仍在使用的大概估计。第一周，可能高达100%或90%，第二周，也许80%，下周70%，你希望在50%或其他目标上保持稳定。若半数的人还在，你想知道最终是否会下降到0。是否有忠实用户。一般而言，新人进入时，只关注统计器会让你很难分辨这是新人还是老用户；Mixpanel在这方面做得很好，它允许你关注关联事件，如消息回复或类似事件。

InfoQ: 好的，最后一个问题，在面临很多工具（自己动手、参与开源项目、采用可信的第三方服务，如graphite, nagios, jmeter, yammer metrics, New Relic等等）可选时，你会如何抉择？最主要的影响因素是什么？

Kevin: 我常关注是否已有现成的解决方案。我会先从开源工具中选取。开源工具的一大好处在于，若是存在痛点，将会是很多人共同的痛点，他们会尝试修复，若尚未修复，我会亲手完成。曾发生过很多次我或同事修复痛点的情况，我们编码回馈发起者。真的很像一块踏脚石，你打下桩，其他人帮你完善。

Picasso是源于Square的一款我非常喜欢的程序库。在just.me我用Picasso来载入图像。我已经向Square提交了一些功能请求。我还拥有为功能或其他东西投票的权利。对just.me而言，这很实用。我真的不愿意为缓存和图像载入编写所有需要的代码。

我常常关注开源，但开源并非次次有效。有时会没有足够的人对这一问题感兴趣。有时会需要你启动自己的开源项目。以我在E*TRADE工作时为例，我想测试应用如何在所有浏览器中运行，我想用Jenkins做持续集成。没有人那样做。我发起并为Jenkins完成了具备相关功能的插件。很多人开始贡献补丁、修正和各种很棒的功能需求。

很多次，开源方案并不完全适合你的需求。就像我对Graphite所做的，我需要一个仪表盘，他们的仪表盘做得很用心，但很难做出契合需求的修改。我并不认为在他们的仪表盘上再进行改进来契合需求可行，所以我在他们所提供原始图表基础上创建了自己的仪表盘。Graphite图表的自由度令我叹为观止，但我发现仪表盘有不足。所以我用自己的工具开发出一个更灵活的仪表盘。

我也会关注一些商用软件。商用产品在满足需求又不需过多个性化定制的情形下能帮你节约时间和支出。涉及业务核心时从零开始构建的确是正确的选择。你想和其他产品有足够的区分度，你想完完全全控制。具备资本时，你可以力挺开源项目。然而，当那些东西作为你的业务核心时，你不该总从开源入手。

如果你需要比较高级的功能，比如JVM调优，这时候你会发现在整个开源界里找不到能够同时做分析和调优的项目。人们并没有足够的时间和精力将之投放到开源世界。我认为New Relic已经做得很好，所以我们选择了跟New Relic合作。

Kevin Nilson 是一位Java Champion，曾三次获得JavaOne Rock Star称号。Kevin在JavaOne, Devoxx, JAX, O'Reilly Fluent, Silicon Valley Code Camp, JAX, HTML5DevConf, On Android, NFJS SpringOne and AjaxWorld等会议上做过讲演。Kevin是Web 2.0 Fundamentals的作者之一。 他曾在San Mateo大学当助教。 Kevin拥有Southern Illinois大学的计算机硕士和学士学位。Kevin是Silicon Valley Java User Group, Silicon Valley Google Developer Group和Silicon Valley JavaScript Meetup的领跑者。

查看英文原文：Interview with Kevin Nilson on Cloud Monitoring and Mobile Testing

查看原文：云端监控和移动测试释疑——对话Just.me工程副总裁Kevin Nilson

基于 AWS 技术实现发布/订阅服务

作者 Boris Lublinsky ，译者 王丽娟

AWS提供两种服务——Amazon简单通知服务（Simple Notification Service）和Amazon简单队列服务（Simple Queue Service），两者结合起来可以为完整的发布/订阅服务提供支撑。

现有的AWS功能

Amazon简单通知服务（Amazon SNS）是一个Web服务，能让应用、最终用户和设备立即从云端发送和接收通知。简化的SNS架构如下图所示（图1）：

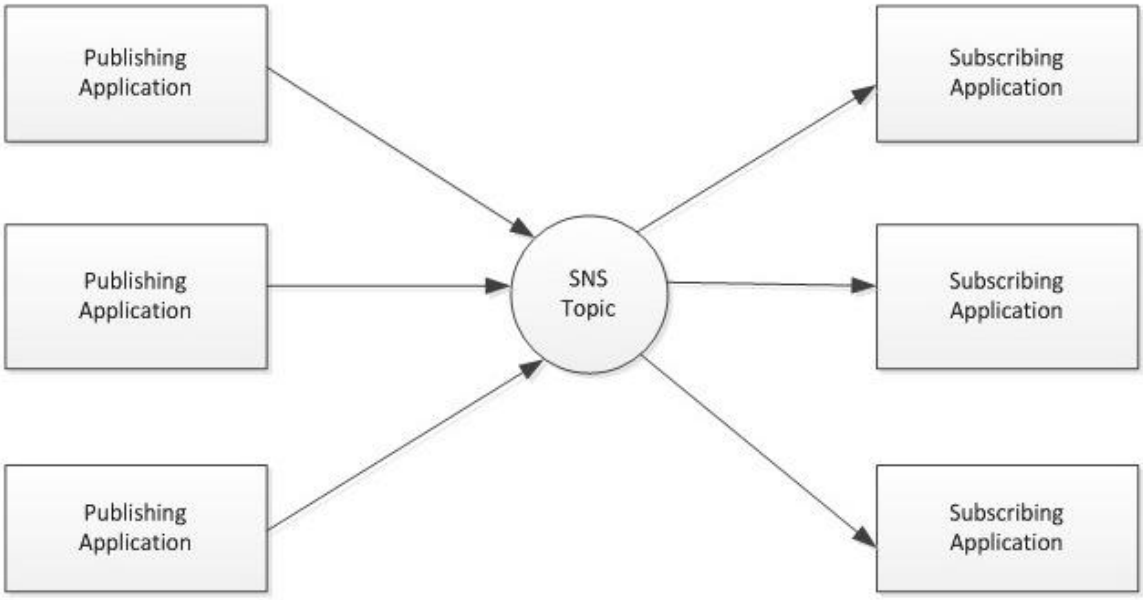

图1：Amazon SNS的基础架构

多个发布应用和多个订阅应用可以将SNS主题作为中介互相通讯。这样实现的优点是发布者和订阅者不需要知道对方，因此，应用可以完全动态地进行集成。SNS支持用多种传输协议传递通知，包括HTTP、HTTPS、Email、SMS和Amazon简单队列（Simple Queue）。

Amazon简单队列服务（Amazon SQS）提供可靠、可伸缩的托管队列，用来存储计算机之间传输的消息。使用Amazon SQS，你可以在执行不同任务的应用分布式组件之间移动数据，而不会丢失消息，也不必要求每个组件始终都是可用的。SQS和SNS结合起来会带来两个额外的优势——解除时间上的耦合度，根据消费应用特定的情况提供负载均

衡——这是SNS无法单独提供的。要做到第二个附加优势，需要同一个应用的多个实例从同一个队列里读取消息。下图展示了SNS和SQS结合的总体架构（图2）。其中的一个订阅应用显示为负载均衡的。

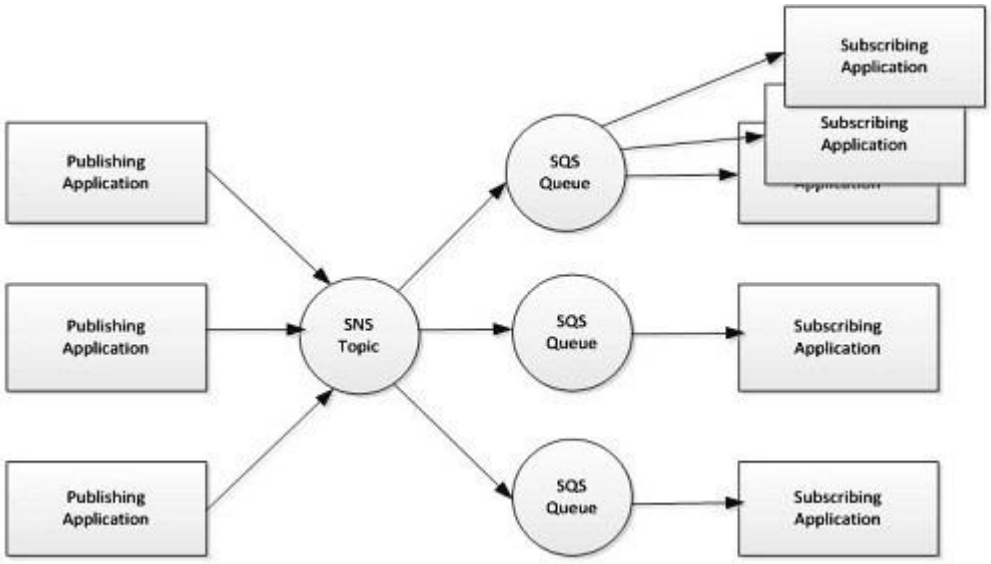

图2：结合SNS和SQS

这个实现的主要缺点是，发布者和订阅者需要明确统一SNS主题的名称。此外，如果一个特定的消费者想从多个主题获取信息，那他需要把队列注册到多个主题上。

期望中的发布/订阅实现

这个问题的典型解决方案是采用基于树的主题组织，大部分发布/订阅引擎都是这样实现的。OASIS规范的Web Services Topics 1.3概述了这种组织的主要原则。

这个规范将主题定义为：

“……主题是一组通知的组织和分类方式。主题机制为订阅者推断出感兴趣的通知提供了便捷的方式……发布者可以将通知发布和一或多个主题关联起来。当订阅者创建订阅的时候，可以提供一个主题的过滤器表达式，将订阅和一或多个主题关联起来……每个主题都可以有零或多个**子主题**，子主题本身也可以进一步包含子主题。没有**“父亲”**的主题叫**根主题**。特定的根主题和它所有的后代会形成一个层次结构（称为**主题树**）。”

下面是手机销售的一个主题树例子（图3）。

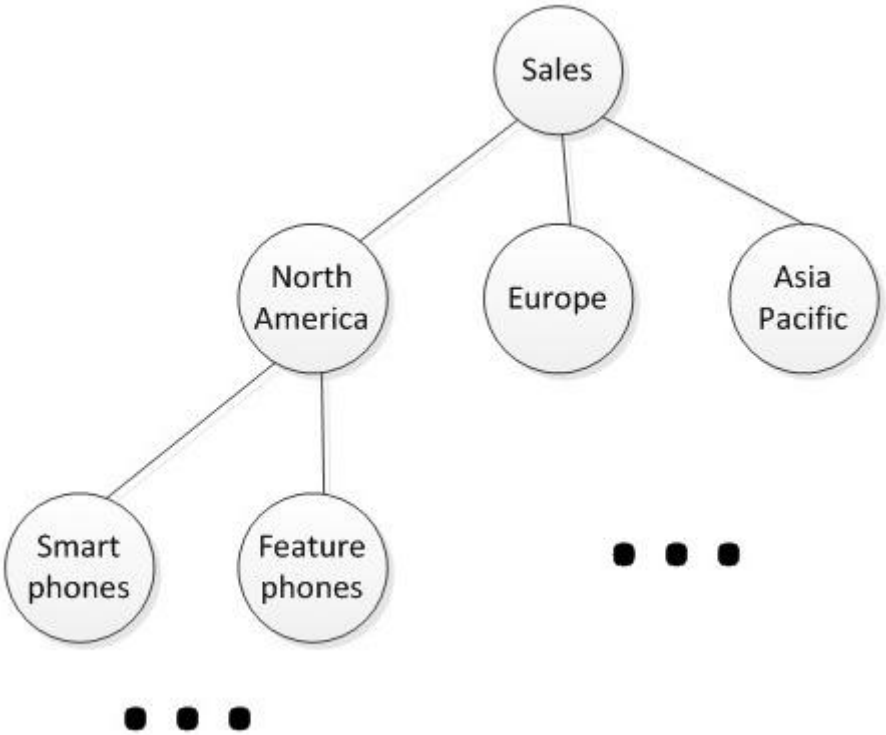

图3：主题树示例

主题树的根表示销售。销售可以按区域细分（在我们的例子中有北美、欧洲和亚太地区）。特定区域的销售还可以按照手机类型进一步细分，依此类推。

在发布/订阅系统中，这样的结构之所以重要是因为树反映了数据的组织。如果消费者对北美的智能手机销售感兴趣，他可以监听这个特定的主题。如果他对北美所有的销售都感兴趣，那他就可以监听北美的主题，从子主题获取所有的通知。

当然，这种方法并不能解决所有的问题。比如说，如果消费者想监听所有智能手机销售的事件，他就需要明确订阅所有地区的智能手机销售事件。这种情况通常是主题树设计的问题。树的设计基于信息的组织和典型的使用模式。在某些情况下，会设计多个主题来满足不同的内部需求（参见Web Services Topics 1.3里的主题命名空间）。发布/订阅架构的另一个重要特性就是基于内容的消息过滤：

“在基于内容的系统中，如果消息的属性或内容与订阅者定义的约束相匹配，消息就只会传递给这个订阅者。订阅者负责消息的分类。”

换句话说，订阅者在这种情况下可以使用正则表达式列表，明确指定他们感兴趣的消息内容。

把这种过滤和结构化的主题结构结合起来，可以创建出非常灵活和强大的发布/订阅实现。

我们将在本文中展示如何用AWS组件轻松构建这类系统。

发布/订阅架构建议

建议给大家的架构如下图所示（图4）。在这个架构中，发布/订阅服务器的实现是一个Tomcat容器里运行的Web应用。我们还充分利用了AWS的弹性负载均衡器（Elastic Load Balancer），它可以根据当前的负载动态扩展或缩减发布/订阅服务器集群的大小。此外，架构还用关系型数据服务（Relational Data Service）存储当前的配置，以便动态新增发布/订阅实例。为了提高整体性能，我们在内存里保留了当前的拓扑结构，尽量减少数据库访问的次数。这样的话，实际的消息路由会非常迅速。这个解决方案需要一种机制，能在拓扑结构发生变化的时候去通知所有的服务器（因为任何服务器都能处理负载均衡器）。Amazon SNS能轻而易举地做到这一点。最后，我们用Amazon SQS将通知分发给消费者。需要注意的是，一个消费者可以监听多个队列。

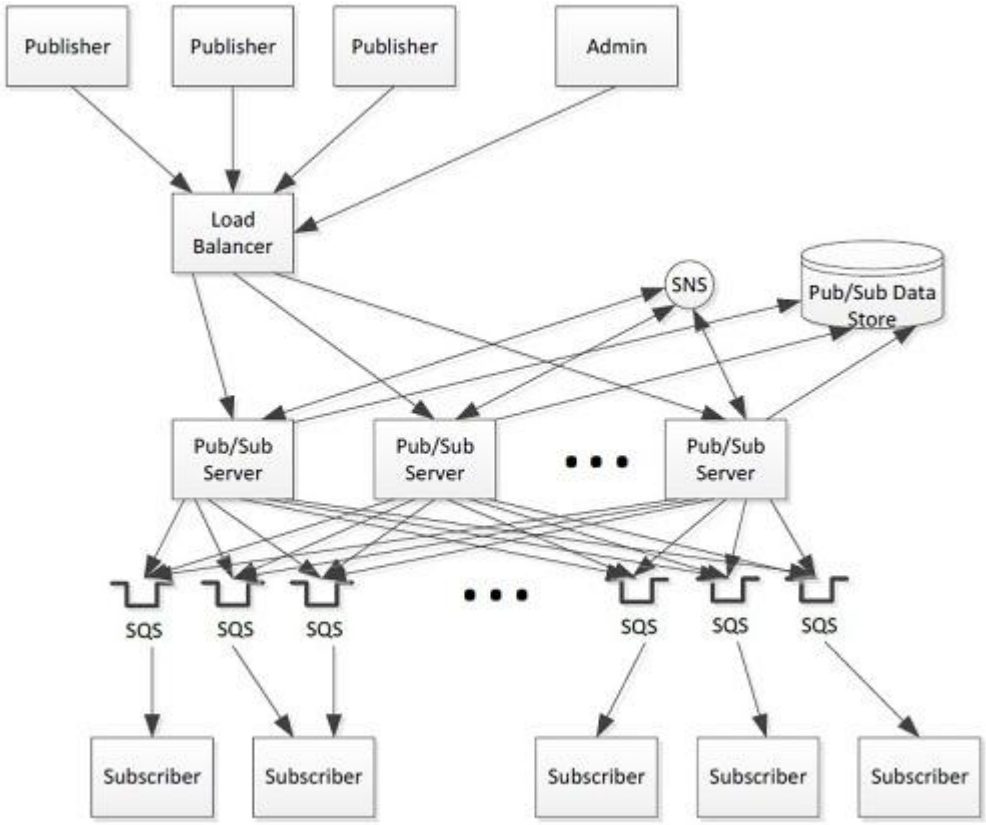

图4：整体架构建议

发布/订阅服务器

这个实现的核心是一个自定义的发布/订阅服务器。服务器实现包括三个主要的层——持久化、域和服务。

持久化

服务器持久化层采用JPA 2.0实现，定义了三个主要的实体——主题、订阅和语义过滤器。

主题实体（清单1）描述了特定主题要存储的相关信息，包括主题ID（数据库的内部ID）、主题名称（标识主题的字符串）、一个布尔变量（定义该主题是否是个根主题）、到父主题和孩子主题的引用（以便对主题层次结构进行遍历），以及与给定主题关联的订阅列表。

```
@Entity
@NamedQueries({
    @NamedQuery(name="Topic.RootTopics",
                query="SELECT t FROM Topic t where t.root='true'"),
    @NamedQuery(name="Topic.AllTopics",
                query="SELECT t FROM Topic t")
})
@Table(name = "Topic")
public class Topic {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;      // 自动生成的ID

    @Column(name = "name",nullable = false, length = 32)
    private String name;                        // 主题名称

    @Column(name = "root",nullable = false)
    private Boolean root = false;               // 根主题标识

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="TOPIC_ID")
    private Topic parent;

    @OneToMany(mappedBy="parent",cascade=CascadeType.ALL,orphanRemoval=true)
    private List<Topic> children;

    @OneToMany(mappedBy="topic",cascade=CascadeType.ALL,orphanRemoval=true)
    private List<Subscription> subscriptions;
    ……
```

清单1：主题实体

我们定义了两个命名的查询，用来访问主题：RootTopics获取从根开始的主题结构，AllTopics获取所有现有的主题。

这个实体提供了一个完整的主题定义，也可以支持多个主题树（而不是实现示例的一部分）。

订阅实体（清单2）描述了订阅相关的信息，包括订阅ID（数据库的内部ID）、队列名称（SQS队列的ARN，ARN即Amazon Resource Name）、对订阅关联主题的引用，还有一个语义过滤器列表。只有所有的过滤器都接受消息（见下文），通知才会分发给给定的队列（客户端）。如果通知不包含语义过滤器，那来自于关联主题的所有消息都会直接传递给队列。

```
@Entity
@NamedQueries({
    @NamedQuery(name="Subscription.AllSubscriptions",
                query="SELECT s FROM Subscription s")
})
@Table(name = "Subscription")
public class Subscription {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;     // 自动生成的ID

    @Column(name = "queue",nullable = false, length = 128)
    private String queue;

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="TOPIC_ID")
    private Topic topic;

    @OneToMany(mappedBy="subscription",
                cascade=CascadeType.ALL,orphanRemoval=true)
    private List<SemanticFilter> filters;
    ……
```

清单2：订阅实体

我们还定义了一个命名的查询，获得所有存在的订阅。

最后，语义过滤器实体（清单3）描述了特定语义过滤器的信息，包括语义过滤器ID（数据库的内部ID）、该语义过滤器测试的属性名称、使用的正则表达式，以及对语义过滤器关联订阅的引用。

```
@Entity
@NamedQueries({
    @NamedQuery(name="SemanticFilter.AllSemanticFilters",
                query="SELECT sf FROM SemanticFilter sf")
```

```
})
@Table(name = "Filter")
public class SemanticFilter {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;      // 自动生成的ID

    @Column(name = "attribute",nullable = false, length = 32)
    private String attribute;                    // 属性名称

    @Column(name = "filter",nullable = false, length = 128)
    private String filter;                       // 正则表达式过滤器

    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="SUBSCRIPTION_ID")
    private Subscription subscription;
    ……
```

清单3：语义过滤器实体

我们一样定义一个命名的查询，用来获取所有现有的语义过滤器。

除了实体，持久化层还包含一个持久化管理类，负责：

管理数据库访问和事务

从数据库读取、写入对象

对域对象（见下文）和持久化实体进行相互转换

发送拓扑结构变化的通知

域模型

域模型对象的主要职责是支持服务操作，包括数据的订阅和发布，并把通知真正发布到订阅的队列上。在这个简单的实现里，域模型和持久化模型是合在一起的，但为了阐述得更清楚，我们分开介绍。这两层的数据模型是一样的，但域对象会多一些明确支持发布/订阅实现的方法。

过滤器处理的实现（清单4）利用了Java String里对正则表达式处理的内置支持。

```
public boolean accept(String value){
```

```
            if(value == null)
                return false;
            return value.matches(_pattern);
    }
```

清单4：过滤器处理方法

发布实现（清单5）是订阅类的一个方法。请注意，这个方法对语义过滤器进行了或操作。如果给定的客户端能有多个订阅，或者对订阅实现进行扩展、让它支持Boolean函数，那就可以突破这个限制了。

```
public void publish(Map<String, String> attributes, String message){
    if((_filters != null) && (_filters.size() > 0)){
        for(DomainSemanticFilter f : _filters){
            String av = attributes.get(f.getField());
            if(av == null)
                return;
            if(!f.accept(av))
                return;
        }
    }
    SQSPublisher.getPublisher().sendMessage(_queue, message);
}
```

清单5：发布实现

这个实现利用了基于现有AWS Java API的SQSPublisher类（清单6）。

```
import java.io.IOException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.sqs.AmazonSQSClient;
import com.amazonaws.services.sqs.model.CreateQueueRequest;
import com.amazonaws.services.sqs.model.DeleteQueueRequest;
import com.amazonaws.services.sqs.model.SendMessageRequest;

public class SQSPublisher {

    private static SQSPublisher _publisher;
```

```
    private AmazonSQSClient _sqs;

    private SQSPublisher()throws IOException {
             AWSCredentials credentials = new PropertiesCredentials(
                  this.getClass().getClassLoader().
getResourceAsStream("AwsCredentials.properties"));
              _sqs = new AmazonSQSClient(credentials);
    }

    public String createQueue(String name){
             CreateQueueRequest request = new CreateQueueRequest(name);
             return _sqs.createQueue(request).getQueueUrl();
    }

    public void sendMessage(String queueURL, String message){
             SendMessageRequest request = new SendMessageRequest(queueURL,
message);
              _sqs.sendMessage(request);

    }

       public void deleteQueue(String queueURL){
             DeleteQueueRequest request = new DeleteQueueRequest(queueURL);
              _sqs.deleteQueue(request);
    }

    public static synchronized SQSPublisher getPublisher(){
             if(_publisher == null)
                      try {
                               _publisher = new SQSPublisher();
                      }catch (IOException e) {
                              e.printStackTrace();
                      }
             return _publisher;
    }

}
```

清单6：SQS发布者

订阅者可以利用这个类的其他方法创建/销毁SQS队列。

除了SQS队列，我们的实现还利用SNS进行数据库变化的同步。与SNS的交互由SNSPubSub类实现（清单7），这个实现也利用了AWS SNS Java API。

```
import java.io.IOException;
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.sns.AmazonSNSClient;
import com.amazonaws.services.sns.model.PublishRequest;
import com.amazonaws.services.sns.model.SubscribeRequest;
import com.amazonaws.services.sns.model.SubscribeResult;
import com.amazonaws.services.sns.model.UnsubscribeRequest;

public class SNSPubSub {

	private static SNSPubSub _topicPublisher;
	private static String _topicARN;
	private static String _endpoint;

	private AmazonSNSClient _sns;
	private String _protocol = "http";
	private String _subscriptionARN;

	private SNSPubSub()throws IOException {
		AWSCredentials credentials = new PropertiesCredentials(
		    this.getClass().getClassLoader().
getResourceAsStream("AwsCredentials.properties"));
		_sns = new AmazonSNSClient(credentials);
	}

	public void publish(String message){
		PublishRequest request = new PublishRequest(_topicARN,
message);
		_sns.publish(request);
	}

	public void subscribe(){
		SubscribeRequest request = new SubscribeRequest
(_topicARN, _protocol, _endpoint);
		_sns.subscribe(request);
	}

	public void confirmSubscription(String token){
		ConfirmSubscriptionRequest request = new
 ConfirmSubscriptionRequest(_topicARN, token);
```

```
                ConfirmSubscriptionResult result = _sns
.confirmSubscription(request);
                _subscriptionARN = result.getSubscriptionArn();
        }

        public void unSubscribe(){
                if(_subscribed){
                        UnsubscribeRequest request = new
UnsubscribeRequest(_subscriptionARN);
                        _sns
.unsubscribe(request);
                }
        }

        public static void configureSNS(String topicARN, String endpoint){
                _topicARN = topicARN;
                _endpoint = endpoint;
        }

        public static synchronized SNSPubSub getSNS(){
                if(_topicPublisher == null){
                        try{
                                _topicPublisher = new SNSPubSub();
                        }
                        catch(Exception e){
                                e.printStackTrace();
                        }
                }
                return _topicPublisher;
        }
}
```

清单7：SNS Pub/Sub

使用SNS

使用SNS的时候要谨记：订阅主题并不意味着你已经准备好监听主题。SNS订阅的过程包含两个步骤。向SNS发送订阅请求时，SNS返回的响应表明确认订阅的必要性。这正是清单8既有subscribe方法又有confirmSubscription方法的原因。

```
<xsd:complextype name="NotificationType">
   <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" />
      <xsd:element name="MessageId" type="xsd:string" />
```

```
        <xsd:element name="Token" type="xsd:string" minoccurs="0" />
        <xsd:element name="TopicArn" type="xsd:string" />
        <xsd:element name="Message" type="xsd:string" />
        <xsd:element name="SubscribeURL" type="xsd:string" minoccurs="0" />
        <xsd:element name="Timestamp" type="xsd:string" />
        <xsd:element name="SignatureVersion" type="xsd:string" />
        <xsd:element name="Signature" type="xsd:string" />
        <xsd:element name="SigningCertURL" type="xsd:string" />
        <xsd:element name="UnsubscribeURL" type="xsd:string" minoccurs="0" />
    </xsd:sequence>
</xsd:complextype>
```

上面的Schema描述了两种消息类型——确认请求和实际的通知。两种类型通过Type元素进行区分。如果元素值是“SubscriptionConfirmation”，那它就是订阅确认的请求，如果是“Notification”，就表明是个真正的通知。

主题类实现了两个方法（清单8），以便支持发布。

```
public void publish(Map<String, String> attributes, String message){

    if(_subscriptions == null)
        return;
    for(DomainSubscription ds : _subscriptions)
        ds.publish(attributes, message);
}

public void processPublications(List<DomainTopic> tList, StringTokenizer st)
throws PublicationException{

    tList.add(this);
    if(!st.hasMoreTokens())
        return;
    String topic = st.nextToken();
    for(DomainTopic dt : _children){
        if(topic.equalsIgnoreCase(dt.getName())){
            dt.processPublications(tList, st);
            return;
        }
    }
    throw new PublicationException("Subtopic " + topic + " is not found in topic " +
_name);
}
```

清单8：主题对发布的支持

processPublications方法创建了一个主题列表，这些主题与给定的消息相关联。这个方法有一个标记过的主题树字符串，如果标记和主题名称相对应，就会把当前的主题添加到列表中。主题的publish方法维护一个消息属性的映射，对主题相关的每个订阅来说，publish方法还会尝试着去发布一条消息。

上面的方法都由Domain管理器类的publish方法调用（清单9）。这个方法首先标记主题字符串，然后用processPublications方法创建一个订阅者感兴趣的主题列表。列表一旦被创建好，就会构建一个消息属性的映射（我们假设是一个XML消息），并把这个映射发布给列表里的所有主题。

```
    public void publish (String topic, String message){
        StringTokenizer st = new StringTokenizer(topic, ".");
        List<DomainTopic> topics = new LinkedList<Domaintopic>();
        DomainTopic root =
PersistenceManager.getPersistenceManager().getRoot();
        try {
                if(!st.hasMoreTokens())
                        return;
                String t = st.nextToken();
                if(!t.equalsIgnoreCase(root.getName()))
                        throw new PublicationException("Unrecognized subtopic
name " + topic);
                root.processPublications(topics, st);
        }catch (PublicationException e) {
                e.printStackTrace();
                return;
        }
        MessageType msg = null;
        try {
                JAXBElement<MessageType> msgEl =
(JAXBElement<MessageType>)
                        _unmarshaller.unmarshal(new
ByteArrayInputStream(message.getBytes()));
                msg = msgEl.getValue();
        } catch (JAXBException e) {
                e.printStackTrace();
                return;
        }
        Map<String, String> attributes = new HashMap<String, String>();
        MessageEnvelopeType envelope = msg.getEnvelope();
        if(envelope != null){
                for(MessageAttributeType attribute :
envelope.getAttribute()){
```

```
                        attributes.put(attribute.getName(),
attribute.getValue());
                    }
            }
            for(DomainTopic t : topics)
                    t.publish(attributes, message);
    }
```

清单9：发布方法实现

服务模型

我们用一组REST服务对发布/订阅功能进行访问（清单10）。

```
@Path("/")

public class PubSubServiceImplementation {

      // 功能方法
      @POST
      @Path("publish")
      @Consumes("application/text")
      public void publish (@QueryParam("topic")String topic, String message)
throws PublicationException{
              DomainManager.getDomainManager().publish(topic, message);
      }

      @GET
      @Path("publish")
      public void publishGet (@QueryParam("topic")String topic,
@QueryParam("message")String message)  throws
PublicationException{
              DomainManager.getDomainManager().publish(topic, message);
      }

      @POST
      @Path("synch")
      @Consumes("text/plain")
      public void getSynchNotification (Object message){
              PersistenceManager.setUpdated();
      }

      // 配置方法
```

```
	@GET
	@Path("root")
	@Produces("application/json")
	public TopicType getRoot()throws PublicationException {
		return DomainManager.getDomainManager().getRoot();
	}

	@GET
	@Path("filters")
	@Produces("application/json")
	public FiltersType getFilters() throws PublicationException {
		return DomainManager.getDomainManager().getFilters();
	}

	@POST
	@Path("filter")
	@Consumes("application/json")
	public long addFilter(FilterType filter) throws PublicationException {
		return DomainManager.getDomainManager().addFilter(filter);
	}

	@DELETE
	@Path("filter/{id}")
	public void deleteFilter(@PathParam("id")long id) throws
PublicationException {
		DomainManager.getDomainManager().removeFilter(id);
	}

	@GET
	@Path("subscriptions")
	@Produces("application/json")
	public SubscriptionsType getSubscriptions() throws PublicationException
{
		return DomainManager.getDomainManager().getSubscriptions();
	}

	@POST
	@Path("subscription")
	@Consumes("application/json")
	public long addSubscription(SubscriptionType s) throws
PublicationException {
		return DomainManager.getDomainManager().addSubscription(s,
null);
```

```
	}

	@DELETE
	@Path("subscription/{id}")
	public void deleteSubscription(@PathParam("id") long id) throws
PublicationException {
		DomainManager.getDomainManager().removeSubscription(id);
	}

	@POST
	@Path("subscriptionFilters/{sid}")
	@Consumes("application/json")
	public long assignFilersToSubscription(@PathParam("sid") long sid, IDsType
ids) throws PublicationException{
		return
DomainManager.getDomainManager().assignFilersToSubscription(sid, ids);

	}

	@POST
	@Path("topic")
	@Consumes("application/json")
	public long addTopic(TopicType t) throws PublicationException {
		return DomainManager.getDomainManager().addTopic(t, null);
	}

	@DELETE
	@Path("topic/{id}")
	public void deleteTopic(@PathParam("id") long id) throws
PublicationException {
		DomainManager.getDomainManager().removeTopic(id);
	}

	@POST
	@Path("topicsubscription/{tid}")
	@Consumes("application/json")
	public void assignTopicHierarchy(@PathParam("tid") long tid, IDsType ids)
throws PublicationException{
		DomainManager.getDomainManager().assignTopicHierarchy(tid,
ids);
	}

	@POST
	@Path("topicsubscription/{tid}")
```

```
        @Consumes("application/json")
        public long assignTopicSubscriptions(@PathParam("tid") long tid, IDsType
ids) throws PublicationException{
                return
DomainManager.getDomainManager().assignTopicSubscriptions(tid, ids);
        }
```

清单10：发布/订阅服务

这些服务的使用者有消息发布者（publish方法）、服务订阅者（创建/删除语义过滤器，订阅，还有订阅和主题订阅相关的过滤器）、内部的发布/订阅实现（获取同步的服务）和管理应用。

结论

这个实现虽然简单，但创建了一个非常强大、可扩展的发布/订阅实现，同时利用了很多现有的AWS功能和少量的Java定制代码。另外它还充分利用了现有AWS部署功能对负载均衡和容错的支持。

作者简介

Boris Lublinsky博士是Nokia的主要架构师，参与大数据、SOA、BPM和中间件实现的相关工作。Boris去Nokia前是Herzum软件的主要架构师，负责为客户设计大型、可伸缩的SOA系统；在此之前，他是CNA保险的企业架构师，参与CNA集成和SOA策略的设计及实现，构建应用框架，实现面向服务的架构。Boris在企业技术架构和软件工程方面有二十五年多的经验。他是OASIS SOA RM委员会的活跃成员，和他人一起编著了《Applied SOA: Service-Oriented Architecture and Design Strategies》一书，另外他还写了很多关于架构、编程、大数据、SOA和BPM的文章。

查看英文原文：基于AWS技术实现发布/订阅服务

查看原文：基于AWS技术实现发布/订阅服务

如何在 AWS 云平台上构建千万级用户应用

作者 方国伟

AWS服务概述

高扩展性应用建设并非把应用直接迁移到云平台上就能轻易实现，相反我们需要根据云平台的特性进行专门的设计，这包括选择合适的云服务类型并进行良好的应用架构设计。对于希望基于AWS构建千万级用户应用的开发者而言，不仅需要对区域（Region）、可用区（AZ）和边缘站点等基础设施的分布有所了解，更需要了解不同的AWS服务各自的特点和最佳实践。

AWS的服务可大致按照其所处层面分为三类，从下到上依次是基础服务层、应用服务层、部署和管理层。基础服务层也有两层，下层是计算（EC2、WorkSpaces）、存储（S3、EBS、Glacier、Storage Gateway）、网络（VPC、Direct Connect、ELB、Route53），上层是数据库（RDS、Dynamo、ElastiCache、RedShift）、数据分析（EMR、Data Pipeline、Kinesis）、内容分发（CloudFront）。应用服务层主要是把邮件服务、消息队列服务等通用的功能单独抽离出来。部署和管理层则有用于监控的CloudWatch，用于部署运维工作的BeanStalk、OpsWorks、CloudFormation和CloudTrail等，以及IAM、Federation等身份管理服务。

单机到多实例

传统的单机服务，到AWS上面就是跑在一个EC2实例上，这个实例上跟以前的服务器一样上面安装所有的Web应用、数据库等，搭配一个EIP，外部用Route53做DNS。遇到瓶颈后，简单的扩展就是将小的实例换成大的实例，比如small换成2xlarge、8xlarge，服务结构不变，可以快速实现，但是最终都会遇到极限。

到了这一步，就要从单实例服务变成多实例。这一步骤涉及到Web实例和数据库实例的拆分，数据库可以开始考虑选择SQL或者NoSQL。SQL大家比较熟悉，优点很明显，缺点主要在规模变大之后呈现，不过一般对于百万级用户量内的应用，SQL是能够满足需求的；但如果数据量增长速度很快，数据是非结构化或者半结构化的，应用要求的延时低、写入的速度要求快，那考虑NoSQL会更合适一些。

几百个用户的情况，一个RDS实例+一个Web实例即可满足需求，前端直接用一个EIP，即单机的情况；用户上千的情况，建议启动两个RDS实例+Web实例并将实例部署在不同的可用区，前端用ELB做负载均衡。

对于百万级以下用户的规模，每一个可用区内会有多个Web实例和RDS实例组成的集群，其中Active RDS实例和Standby RDS实例要放在不同的可用区，其他RDS实例均为只读。

到了这个规模之后，再要往上扩展到百万级，就需要改变部分工作负载的设计方式了。

改变部分工作负载的设计方式

第一步可以引入S3和CloudFront。把静态内容从Web实例中迁移到S3上，适合的文件类型包括静态数据（CSS、JS、图片、视频）、日志、备份等。S3具备11个9的持久性，本身是海量存储，可以支撑大量的并发访问，而且成本很低。CDN方面，CloudFront以Web Service接口的方式提供服务，支持动态和静态内容、流式视频，支持根域，支持客户化SSL证书。

第二步可以引入ElastiCache和DynamoDB。ElastiCache是托管的Memcached和Redis服务，API是一样的，两者都是非常快的缓存服务（毫秒级别），区别在于Memcached使用一个AZ，Redis可以跨AZ复制。DynamoDB是NoSQL服务，后台存储基于SSD，平均延时在毫秒级别。

这时候我们可以开始考虑弹性的问题，即应用的自动扩展。弹性的实现有四个前提：

- 完善的、基于指标的监控体系
- 自动化构建
- 自动化部署
- 集中化日志管理

在AWS上实现自动构建部署，可以选择Beanstalk、OpsWorks或CloudFormation，也可以完全自己写脚本配合定制AMI来实现。Elastic Beanstalk是全自动化的，基于容器实现，适合常规的Web应用；OpsWorks是半自动化的，适合较为复杂的应用开发流程，可以对资源配给、配置管理、应用部署、软件升级、监控、身份控制进行定制化；CloudFormation是基于模板的管理模式，可定制的范围更大。

如果以上都做到，那么一个百万级用户量的应用基本上可以比较好的管理起来。进一步到千万级用户量的规模，我们需要更多的引入面向服务的架构设计，即SOA。

SOA、SOA、SOA

SOA在04、05年讲得比较多，到现在基本上已经是大家都认可的做法，非常适合大规模应用的场景，其核心在于松耦合。

比如消息队列服务SQS，加在模块A和模块B之间，这样即使模块A宕掉了，模块B也仍然可以正常运行一段时间。美国大选网站就是采用了这样的思路，在SQL实例压力大的时候把实例关掉，换上一个更大的实例，因为前面有SQS顶着才可以这样做。

而AWS上的通知服务（SNS）、邮件服务（SES），也建议大家多多采用，而不要自己搭建Web实例来做，因为此类服务在处理海量请求方面的能力要远远超过一般的实现。

千万级规模对数据库的性能挑战是很大的，对于SQL，联邦（federation）、分片（sharding）都是常用的方法，将“热”表、快速写数据迁移到NoSQL也是一种思路。应用的性能挑战方面，重点则在于即时获得反馈（完善实时的监控+报警），以及持续的调优各个模块。

参考资料

- AWS官网
- AWS参考架构
- AWS白皮书
- AWS英文博客
- 在线动手实验

查看原文：如何在AWS云平台上构建千万级用户应用

基于 AWS 云平台的高可用应用设计

作者 包研

2014年5月21日，AWS中国首席云技术顾问方国伟在InfoQ在线课堂介绍了AWS高可用和非高可用的服务分类，从高可用角度对典型服务进行介绍，以及高可用设计的5大常见设计原则，并结合AWS的相关服务依次进行了架构设计分析。期间，方国伟回答了许多网友的问题，InfoQ对其进行了整理和补充，一并提供给大家。

你也可以重复收看本次在线课堂。

关于存储

Q：instance down掉之后，这个instance上的数据会保留吗？能找回来吗？以前看到文档说要用EBS才能保障instance down掉之后的数据保留，是这样吗？是否有其他方案？

方国伟：有好几个问题就讲instance storage与EBS数据存储的问题，我统一讲一下。因为EC2里边的数据存储一般分两大类，一个是instance storage，另一个是EBS。instance storage是放在EC2里的硬盘里边的，是通过虚拟化实现的。如果这个EC2所在的物理机或instance出问题，那这个EC2的数据就不能访问了。为了解决这个问题，我们提供了EBS，EBS是通过网络访问的，如果你的instance出问题了，数据还在EBS里边。那可能有人会问，EBS的数据会不会丢呢？EBS的数据一般可靠性比较高，我们有个数据叫annual failure rate（AFR）会比一般的硬盘低好多，一般是普通硬盘的1/5到1/10。那EBS还是有可能要出问题，那怎么办？一种方式是定期对EBS做snapshot，snapshot的数据放在S3里，S3的数据持久性是11个9，非常高了。这是可以解决EBS丢数据的一个解决方案。

Q：instance down掉后EBS会自动卸载吗？

方国伟：一般会自动卸载。

Q：每个instance都有对应的EBS，（instance）销毁后EBS上存储的数据怎么处理？

方国伟：EBS上数据如何处理完全由用户自己决定。用户可以保留EBS卷，从而保留数据，也可以把EBS删除从而删除其上的数据。另外，用户也可以通过先制作EBS卷快照然后再删除EBS卷的方式来保留数据，因为以后还可以根据这个快照来重新生成一个包含这些旧数据的新EBS卷。

Q：instance storage怎么使用？哪些数据可以放在instance storage里面？

方国伟：这就有不同的场景，主要看你的数据有哪些对持久性要求，如果不是特别高，就放到instance storage里。或者在instance的数据在别的instance里有备份的，这样相当于一份丢了，从软件架构上自己可以从别的instance中恢复。

Q：AWS自己的DR是怎么设计的？如果一个region出现问题，怎么恢复呢？用户数据有sunshine吗？

方国伟：我们在设计这个平台的时候，region和region之间基本上是相互独立的，在默认情况下，你在region里面部署了应用程序，如果这个region宕掉了，当然你的应用就宕掉了。这里首先要指出一点，region本身不是一个数据中心，只要是商用region，我们至少要保证有两个AZ以上。这里强调一点，AZ本身也不是一个数据中心，有可能是多个数据中心组成一个AZ，所以整个region宕掉的可能性非常小。如果你有一个对可靠性要求特别高的应用，你可以自己在部署的时候跨region部署，但这个需要用户来做架构上面跨region的应用架构设计、数据拷贝等等。AWS是不会主动帮你来做的，最重要的原因是很多国家和地区对数据本身有一定的独立性要求，比如欧盟要求数据放在欧盟的region里边，AWS是不能把数据从region里主动让它离开的，这个数据拷贝的动作一定要用户自己操作才能进行。

关于数据库

Q：RDS的Multi-AZ部署支持SQL Server吗？

方国伟：我们最新宣布支持SQL Server，这样四个数据库类型（My SQL、Oracle、PostgreSQL和SQL Server）都支持Multi-AZ Deployment了。

Q：原来部署到Windows+MySQL上的BS应用，如果要部署到AWS，应该做哪些大的改动？

方国伟：不需要大的改动，Windows用EC2，MySQL可用EC2也可以用RDS服务。AWS的兼容性很好的。

Q：DynamoDB怎么进行备份？

方国伟：一般情况下是不需要用户做备份，因为我们已经帮你备了三个拷贝，甚至更多。不过多个拷贝是在一个region里面的，如果你对数据可靠性要求非常高，可以跨region做拷贝，这个备份也非常简单。

Q：当数据库master出现问题后，切换到slave，是否支持可写？如果支持可写，是否会出现master和slave互为双写的情况？

方国伟：是支持可写的，因为Multi-AZ部署是完全切换，这个跟Read Replica不太一样。在RDS服务里，对数据库的两个相关的服务，一个就是Multi-AZ部署，Multi-AZ部

署完全是一个主拷贝和一个副拷贝，也就说这两个是同步复制的，一般所有的访问都是针对那个master拷贝的，当master拷贝出问题的时候，它会自动切换，这个访问不是通过IP的，而是通过Domain Name，所以对应用程序来讲，它是透明的。如果应用Read Replica，那当然可以把一些读操作放到Read Replica上面，那这个跟Multi-AZ又不太一样。

master和slave互为双写？不会的，因为它始终同时只有一个是写的，要保持数据的一致性。

Q：SQS最高是多少并发，需不需要做HA？

方国伟：一般来讲这个不是问题，你不需要做HA，SQS本身就是高可用的，而且它是Auto Scaling，我们有非常大批量的用户在用都没有问题。

关于Auto Scaling

Q：自动修复的功能，如果启动新的虚机，怎么部署代码啊？

方国伟：一般来讲有两大类方法。第一类，你把应用的代码做在AMI里面。Auto Scaling也好，还是你自己启动一个Instance也好，我们肯定会用到一个AMI，这样你新建了那个Instance里面就已经包含了你要部署的代码，你要部署的运行时环境，中间件等等。但是有的同学问，我的应用可能会有不同版本，或者改变比较多，那这个部署太不灵活了，所以我们另外还有办法，你在部署以后可以通过EC2有一个特性user data来实现，当应用部署启动Instance后，user data会帮你执行一段脚本，或者可以传进去一些参数给脚本，你他指定他从某个地方，比如S3上面下载一个新的程序的代码，然后在Instance启动之后去执行。将这两种方式结合起来，一般可以满足绝大部分用户关于动态软件和代码的需求。

Q：Auto Scaling策略需要编程实现吗？

方国伟：一般不需要。Auto Scaling实现需要用户做三个事情：

- 配置launch configuration，launch configuration存储了一个Instance以后，它应该是包含什么样的信息，用哪个AMI，放在哪个region里面，用哪几个AZ。
- 建个Auto Scaling Group，设置最少有多少个Instance，最大多少个Instance。
- 最后一个Scaling policy，它定义了如果CloudWatch监控到超过预值后，要做个什么操作，这个操作大部分情况下是不需要编码来做的。当然除非特别复杂，你可以定制来做一些事情。

Q：这个健康监测是AWS提供的吗？

方国伟：是通过CloudWatch提供的，用户也可以自己配置一些监控指标。

Q：启动新的EC2实例，需要加载Image，AMI。Image可以定制吗？加载自己的(custom)吗？

方国伟：当然可以的。AMI的来源有几种方式，你可以用AWS官方提供的，或者社区提供的。当然社区提供的AMI要要做安全方面检查。当然你可以自己定制AMI，我们有专门的工具和API来帮你定制。对于绝大部分用户，比较常用的方式是使用官方提供的AMI，然后在此基础上自己做镜像，而不是所有操作系统，所有环境从头开始自己来做。

Q：EC2实例的镜像（AMI）是不是可以事先准备好，使用的时候直接使用？

方国伟：当然，实例的镜像都是事先准备好的。如果你根据AMI新建一个实例，你可以通过EC2 create AMI功能根据当前运行的实例生成一个AMI，你可以把AMI存在你自己的地方。生成以后，可以用这个自动生成的AMI，每个AMI都有id，所以你在创建Auto Scaling的launch configuration时可以根据id调用需要的AMI。

Q：Auto Scaling对windows和Linux的AMI有区别吗？

方国伟：Auto Scaling使用过程在Windows、Linux上没有区别，在我们平台上面都有支持的，没有问题。

Q：SQS是顺序保证的吗？

方国伟：不保存，我们不保证first in first out，SQS能保证不丢。但是顺序保证，通常需要通过应用程序层来实现。

Q：我们需要考虑CDN吗？

方国伟：我们全球有CDN服务Cloud Front，我们全球现在应该有51个点。国内我们会用合作伙伴的CDN设施，我们现在官方推荐的是网宿。

关于AWS中国的服务

Q：AWS什么时候在中国商用？

方国伟：现阶段还处于有限预览阶段，只针对邀请客户开放。如果需求，建议联系AWS的销售人员。

Q：AWS有没有中文版？

方国伟：我们国内的版本就有中文版，如果你是有限预览用户，你可能已经看到了。如果你需要使用，你可以在www.amazonaws.cn上面申请。当然目前并不是申请了就能

用，原因很多了。申请之后，我们销售可能会跟你联系，我们内部会有个排队的过程。基本上，我们会根据用户的情况来筛选，哪些用户来进入有限预览名单。

Q：（AWS）有中文版试用吗？AWS有没有针对个人开发者的一些优惠政策？例如流量或CPU消耗一定程度下是免费的？

方国伟：关于中国和国外的帐号或私有帐号的问题，我来解释下。首先，AWS没有所谓的试用帐号，任何人申请的帐号都是一样的。用户可以自己区分，这个帐号是用来做测试的，另外一个帐号做生产。对AWS来讲，对所有的帐号是一视同仁，都是生产帐号。对开发人员来讲，我们有一个免费试用套餐，英文叫free usage tier，目前这个套餐针对海外全球服务系统，那free usage tier的意思是在一定的额度下面，所有注册AWS帐号的用户，在一年之内免费使用这些资源。如果你想熟悉一下AWS的功能，你就可以用这个free usage tier。当然，free usage tier是给所有用户的，只要你注册了，新用户都享受这个服务，哪怕你一上来就是做生产，我们也会把free user tier那部分的费用，帮你从实际的帐单里面扣除。

Q：有没有渠道来申请试用呢？

方国伟：请访问www.amazonaws.cn网站，有个提交申请的链接。

Q：AWS现在还是必须使用外币信用卡才能注册使用吗？

方国伟：AWS全球账户是需要支持外币信用卡，但国内服务开放后会公布国内的支付方式。

Q：AWS在国内有没有国内建AZ？

方国伟：那当然了，我们商用的时候肯定会有多个AZ。第一个region在北京，所以整个数据中心都会在北京地区，我们是跟合作伙伴合作的。根据我们去年12月份发布的计划，第二个region会在宁夏。

Q：有没有联系方式？

方国伟：我的微博是 @方国伟_云端，邮箱是guowfang@amazon.com 。

AWS能部署应用

Q：AWS对应用有要求吗，什么样的应用架构适合部署到AWS上？除了使用AWS的组件之外，对应用本身有没有什么要求，才能实现云应用的高可用性。

方国伟：如果你的应用只是要部署在AWS上，这个要求是非常低的，你的应用只要可以部署在虚拟环境，理论上差不多就可以运行在AWS上面。但是，如果你的应用要充分利用云计算平台的一些特性，尤其是高弹性、高可靠性、高可用性这些云的特点，那你对应用本身就需要做一些在架构上的调整。今天的讲座更多侧重在高可用性方面，6月17日还有个讲座“如何通过架构设计来体现应用本身的弹性”，这是两个不同的维度来看应用。所以说，如果你要充分利用云平台一些特性，那在应用架构上需要做些调整。（**InfoQ注：**6月7日在上海IC咖啡QClub技术沙龙上，方国伟与Autodesk高级软件工程师丁建将分享AWS云平台上建立规模应用的实践，你可以与两位专家当面交流，参会者将获得AWS纪念T恤及25美金AWS抵扣券，欢迎免费报名。）

Q：哪些AWS服务是可以跨region的呢？

方国伟：绝大部分AWS服务都是在一个region里面的，只有很少的服务可以跨region的。身份认证和访问控制（IAM），Route53，CloudFront（CDN），web console是跨region的。

案例资料

Q：海外的AWS案例，有一些介绍么？

方国伟：请访问aws.amazon.com上的case study资料，上面有许多案例。

Q：讲座的PPT可以提供吗？

方国伟：你回头关注一下我们的博客，我们博客会提供Link，大家可以去下载的。我们国内会有一个中文的博客，应该是blog.csdn.net/awschina。（InfoQ注：在所有注册本期在线课程的用户都会受到答谢EDM，EDM中同样有PPT下载地址）。

查看原文：InfoQ在线课堂：基于AWS云平台的高可用应用设计

AWS 云的设计模式与实践

作者 丁建

本文根据欧特克中国研究院（ACRD）高级软件工程师丁建在2014年6月7日的QClub活动分享内容整理而成。

过去一些年来，云计算逐渐发展起来，有人开始为云计算的设计总结可复用的模式，目前形成了不同的流派。

所有流派基本都遵循六条原则：

1. 无状态，包括无状态的镜像和无状态的实例。云上的实例是虚拟化的，很容易被搞坏，所以把需要持久化的数据和实例分离，无论是故障迁移、应用扩展还是升级都很容易。
2. 松耦合，包括地缘、平台、时间、数据格式方面的解耦
3. 弹性。层、服务和组件都应该是有弹性的
4. 自动化。一方面可适应频繁的扩展，另一方面也是避免人工失误导致的问题
5. 全面的监控/日志。线上做debug很困难，需要全面依赖详细的日志
6. Design for failure。容忍错误，快速恢复，将failure当做普通事件处理

设计模式这一概念源自于建筑架构师Christopher Alexander，其目的是通过为特定专业领域的设计问题的解决方案形成文档，以形成某一类问题的通用解决方案。这一概念被引入计算机领域后，就有了软件设计模式这一概念，即针对软件设计的常见问题提供通用的、可复用的解决方案。

云计算设计模式，即为云计算系统设计领域归纳出一套通用的、可复用的解决方案。目前流传较为广泛的云计算设计模式有四个流派，分别来自微软高级总监Simon Guest，cloudpatterns.org在线社区，德国斯图加特大学的Frank Leymann教授等人所写的Cloud Computing Patterns一书，以及AWS的三位日本同仁Ninja of Three。

Simon Guest是2010年微软关键技术人才之一，他所描述的设计模式强调五个方面：可伸缩、多租户、计算、存储、通信（网络）。其他方面大家比较熟悉，这里提到的多租户主要是指一个应用实例服务多个用户的场景，每个用户被授权对应用做一些定制，如界面或业务流程，但不能动应用代码。Salesforce就是典型的多租户模式。

CloudPatterns.org提出的模式是一种通过提问来进展的模式，问题涉及到基础架构、可扩展的资源池、可靠性与灾难恢复、安全、监控等多个方面。一个典型的问答流程为：

共享资源的场景

疑问：如何将物理资源的使用率最大化？

问题：将每台IT资源分配给一个独立用户会造成IT资源的浪费。

解决方案：将物理IT资源虚拟化，拆分成更小的IT资源分配给多个用户使用。

应用：在物理机上创建虚拟机实例，将每一个虚拟机实例指定给用户，而底层的物理IT资源是多用户共享的。

涉及的机制：审计监控、云存储设备、云资源使用监控、虚拟机系统、逻辑网络设备、资源复制、虚拟服务器

复合模式：Burst In，Burst Out至私有云/公共云，弹性环境，IaaS，多租户环境，PaaS，私有云，公共云，弹性环境，SaaS

Frank Leymann教授提出的模式跟上述模式类似，也是以问题作为起点，然后描述背景，最后提出解决方案。比如对于松耦合，提出的问题是：

如何降低分布式应用之间的依赖性以及应用的各个组件之间的依赖性？

背景描述大意是降低依赖性的好处，即可伸缩性、故障处理与升级管理的简化。解决方案是引入broker作为交互的中转。

Ninja of Three（简称NoT）在2012年提出的AWS云系统设计模式列出了九大类48个模式，九大类分别是基本模式、静态内容、批处理、高可用、动态内容、数据上传、关系数据库、运维、网络。

AutoDesk主要使用AWS，因此本次分享主要会介绍NoT的设计模式。目前AutoCAD 360、在线渲染、以及Autodesk 360都在AWS上运行。AutoDesk使用AWS的方式是所有的开发在内部域网络中进行，源代码不出域网络；所有的server都是EC2实例，包括负载均衡、度量、授权验证都是通过EC2实例完成；监控采用CloudWatch；SNS用于做状态异常邮件的推送；SQS在进行Scaling的时候启用；EBS用来存储Server依赖库与中间数据，比如激光扫描文件，每一个文件都有几十个G；数据库采用SDB而不是DynamoDB，因为用户数量不多，主要是单用户资源量很大；S3用来做各种各样的事情，比如客户端安装包、补丁、配置文件、日志信息都存在上面，因为它很便宜、带宽很稳定、对并发访问的支持很好，这个要好好利用；IAM用于管理资源的可访问权限，以保证安全性。

VPC目前还没有使用，因为Autodesk开始用AWS的时候还没有这个服务（2011年之前），而从非VPC环境迁移到VPC环境需要花很多功夫。不过VPC的好处很多，提供了一个简化、可靠的安全性，所以终归考虑要上的。

回到NoT的云系统设计模式。首先是基本模式，其中覆盖了纵向扩展（Scale Up）、横向扩展（Scale Out）、计划的横向扩展、按需扩容、Snapshot和Stamp这几种。对于每一种模式，NoT列出了该模式的优点及相关注意事项。

纵向扩展，即将small的实例换成large（增大）或micro（减少），优点是容易操作，缺点是扩展上限受限于该平台最大能提供的实例资源，而且调整时间在30秒到数分钟不等。目前对于应用而言，Scale Up已经不是主流的扩展模式，主要是关系数据库由于难以多机并行访问而不得不主要从该模式下手。不过有时候，我们可以通过Scale Up获得更好的经济性，比如1个large + 1个x-large的组合就要比3个large便宜，效果却差不多。

横向扩展，即增加服务节点数，好处是无需中断服务即可完成扩展，而且可以扩展的上限高于Scale Up方案。理想的情况下，CloudWatch可以实现autoscaling，但事实是启动新的实例是有延时的，这导致该模式对突发增长起不到什么作用。

相比之下，计划的横向扩展更加适合容易预测的服务。比如我们做推广活动，到了那个时间之前就可以提前把机器都起来warm up，活动开始的时候实例就已经在线上ready了。

我们之所以要把负载均衡放在EC2上自己跑，就是因为我们使用了多种scaling模式：Scale Out、Scale Up和Scheduled Scale，用自己的负载均衡就可以支持多种AMI类型，还可以精准的控制扩展开始和结束的时间——因为AWS是按小时收费的，如果autoscale，结果可能是用了2小时1分钟，我们不得不花3小时的钱，而如果采用计划模式就可以避免花这种冤枉钱。另外我们会时不时的把运行了太久的实例踢出去，因为实例运行时间长了之后稳定性会变差。

对于静态文件模式，NoT列出了Web Storage、Direct Hosting、Private Distribution、Cache Distribution和Rename Distribution等五种模式。我们首先用到Direct hosting，正如上面所说，我们的客户端安装文件和补丁文件、用户配置文件、新版本说明、大文件都是存储在S3上的。不过除此之外，我们还有一些特别的用法，就是用S3来生成日志数据：我们将参数放在URL当中，以此来传递度量数据。

查看原文：AWS云的设计模式与实践

亚马逊 Web 服务 2013 年推荐技术内容列表

作者 杨赛

亚马逊Web服务（AWS）官方博客上介绍了2013年重要的技术内容更新，包括白皮书、文章和视频。该清单列举的资料涉及关系数据库、数据仓库、安全/审计、参考架构、高可用性、大数据处理等多个方面。

安全相关的白皮书包括：

- 用加密确保静态数据的安全
- 规模化的安全：AWS上的管理
- 规模化的安全：AWS上的日志
- AWS安全最佳实践白皮书
- AWS用户的安全审计检查清单

参考实施架构方面的白皮书包括：

- Oracle on AWS（EC2和RDS），Oracle on AWS（EC2）
- Alfresco Enterprise on AWS（包含相应的CloudFormation模板）
- Microsoft Exchange Server 2010 in the AWS Cloud（包含相应的CloudFormation模板）
- Microsoft Windows Server Failover Clustering (WSFC)与SQL Server 2012 AlwaysOn Availability Groups（包含相应的CloudFormation模板）

跟MapReduce相关的白皮书和文章：

- Amazon Elastic MapReduce最佳实践
- Node.js Streaming MapReduce
- 在Amazon Elastic MapReduce上运行Spark和Shark
- Apache Accumulo与Amazon Elastic MapReduce
- 在EMR上使用Hive
- 使用Hive、Powershell和EMR分析大数据

由于文档众多，此处不做完整列举，感兴趣的朋友可以到AWS英文官网上查看。此外，博客中还推荐了2013年AWS re:Invent大会的视频资料。

查看原文：亚马逊Web服务发布2013年推荐技术内容列表