AWS 迷你书

云计算

让数据管理变得更轻松

亚马逊 aws | InfoQ

目　　录

技术应用宏观分享：《亚马逊 CTO：9 种常见的云计算业务应用场景分析》

作者　Daniel Bryant，译者　朱　明

在 Amazon Web ServicesAWS 伦敦峰会的主题演讲中，Werner Vogels 分享了采用云的九种模式，这是在过去九年的运营中 AWS 在它的客户群中观察到的。Vogels 指出，云供应商提供的操作方便、低成本和其它增值业务服务，意味着企业必须拥抱云计算，才能保持竞争力。

Vogels，Amazon 的首席技术官，以“云计算已成为新常态”开始了主题演讲，并列举了越来越多的公司正在开发或迁移到基于云的平台。Vogels 表示，云平台的使用正在推动 IT 基础设施行业内的基本模式转变－－现在客户比供应商有更多的发言权，而通过传统的基础设施模式所提供的僵化和锁定供应商的现象正在逐渐消失。

> [云供应商] 必须提供最好的服务，否则客户就会离开。没有合同义务，没有束缚。[.....] 你的下一个项目很容易托管在其它地方

在演讲的其余部分 Vogels 指出并讨论了 Amazon Web Services（AWS）在自己的云平台基础架构的使用中看到的九种模式。

第一个模式，“创业公司在云上构建企业”，探讨了当科技创业

公司在云上构建自己的产品时，如何能够迅速地打乱长期存在的行业。 Vogels 讨论了几个案例，其中之一是 AirBnB，网上客房租赁网站。Airbnb 只有 5 名 IT 员工，却运行着上千个计算实例，处理数百 TB 的数据。

云基础设施的操作方便、低成本壁垒和避免长期合同，让企业可以用最少的预算建立平台，这与十年前的要求形成鲜明对比。迅速扩大基础设施规模的能力，也使得创业公司可以尝试不同的规模，从而能够最大限度地减少错误造成的财务影响。

Vogels 提出“速度不只限于创业公司”的第二个模式，利用云基础设施，各种规模的公司都可以比以往任何时候更快地行动。

> “当今，没有云无法保持竞争力。对固定硬件进行容量规划是非常棘手的——激增会引起问题，滚动硬件更新和升级是必须的。
>
> 而在云计算中完全不再需要进行容量规划。”

Vogels 提出，IT 往往不是企业内的差异化竞争优势，企业应专注于产品，而不是 IT。云供应商提供的不断创新，也可以是业务的推动力，而不是按照传统的观点把 IT 视为阻碍。例如，美国职业棒球大联盟（MLB）已经在棒球场内署了导弹雷达，来捕捉球员和球的运动。这样的移动数据传输到 Amazon Kinesis，并在 EC2 中处理，从而为家庭电视观众提供实时的“假定推测”分析。

所有的企业都应该象创业公司那样－－敏捷，解决客户的需求。

第三个模式，“企业更倾向于一站式解决方案”，探讨了客户在和云计算供应商合作时，如何想要解决所有的 IT 痛点。例如，AWS 产品，如 Amazon Workspaces 虚拟桌面和 Amazon Workmail 电子邮件和日历服务，在象计算和存储这样典型的云计算功能之上，提供额外的增值服务。Vogels 讨论道，在 25 年前，唯一不在公司内部构建的就是数据库。而现在，企业必须要非常小心地决定什么软件要自己编写、维护和运行。

第四个模式，“公司将更宽泛地使用数据”，包含对全球领先的在线集资平台 Just Giving 的案例研究。Richard Atkinson，Just Giving 的信息总管，讨论道，他的公司需要大量的分析处理，并指出，没有云基础架构，负荷的‘峰值’变化将是一个巨大的挑战：

位于 95%的分析负荷比中值需求大二十倍，而在 99%的负荷更是大了三倍。

Vogels 也讨论了新公布的 Amazon Machine Learning 服务，并提出该工具可以用来降低为充分利用预测分析所需的企业内部的机器学习的专业知识和操作知识的水平。

讨论的第五个模式是“创新是连续的”。Vogels 指出，通过使用云平台功能的更小的“积木”进行设计，应用程序可以更快地构建，

更容易适应，而且更灵活地部署。象 Amazon EC2 Container Service 和 Amazon lambda 这样的服务利用 LXC 容器技术，并允许创建微服务（microservice）架构和异步事件驱动系统。提供系统被设计成利用这些技术，提供的功能能够实现快速试验，并减少到达市场的时间。

讨论的其它模式包括：“云的安全性比内部部署强”，其中“共享责任（shared responsibility）”的工具可以用来控制粒度级别的访问，从而可以确保实现安全审计；在“迁移到云不是一个非此即彼的决定”中，Vogels 指出，用“混合的”内部部署 / 云解决方案迁移到云可以是一个渐进的过程；“客户和合作伙伴全部进入”探讨了象 Netflix 这样的天然的云公司把他们的整个 IT 资源建立在云中；“公共部门争取少花钱多办事”探讨了政府部门对云技术越来越多的使用。

> IT 市场正在发生根本性的变化 [.....]，而云正是帮助你更敏捷的地方。别对抗云，而是要利用云来推动你的业务。

Vogels 在主题演讲的总结陈词中说道，随着许多行业加快的变化步伐，抵制这种变化就好比是试图对抗重力。在他看来，在云上企业可以变得更敏捷，能够更好地适应这些行业变化。

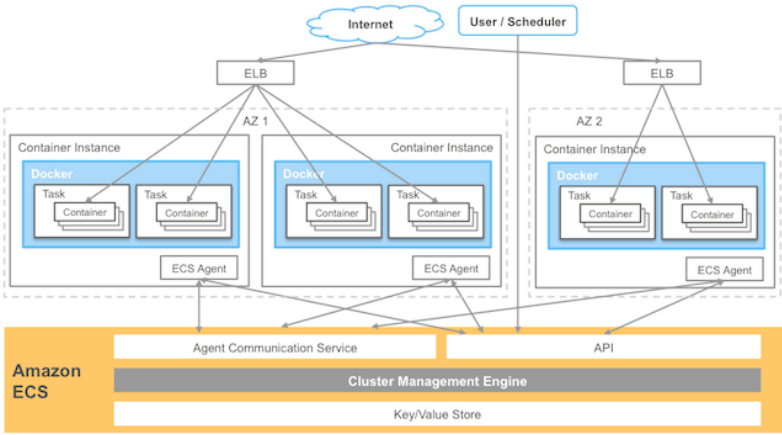
原文链接：http://www.infoq.com/cn/news/2015/05/vogels-patterns-cloud-adoption

英文原文链接：http://www.infoq.com/news/2015/04/vogels-patterns-cloud-adoption

EC2 服务技术解析：《揭秘 Amazon EC2 容器服务背后的技术》

作者　谢　丽

Amazon EC2 Container Service (ECS) 是一个高度可扩展的高性能软件容器管理服务，它支持 Docker，使用户可以轻松地在 Amazon EC2 实例集群上运行应用程序。近日，Amazon 首席技术官 Werner Vogels 撰文介绍了 Amazon ECS 的架构。下图是 Amazon ECS 包含的基本组件：



Amazon ECS 的核心是集群管理器，这是一个处理集群协调和状态管理任务的后台服务，它的上面是不同的调度器。集群管理和容器

调度相互分离，用户可以构建自己的调度器。集群是一个供用户应用程序使用的计算资源池，而所谓的资源是指由容器划分的 Amazon EC2 实例的 CPU、内存和网络资源。Amazon ECS 通过运行在每个实例上的 Amazon ECS 容器代理协调集群。该代理允许 Amazon ECS 与 EC2 实例通信，并在用户或调度器请求时启动、停止和监控容器。它是用 Go 编写的，在 GitHub 上遵循 Apache 许可协议开源。

为了协调集群，需要一个有关集群状态的唯一信息源，提供诸如集群包含的 EC2 实例、运行在实例上的任务、组成任务的容器以及可用资源或已占用的资源这样的信息。这样，才能成功地启停容器。为此，他们将状态存储在一个键 / 值存储中。在任何现代集群管理中，键 / 值存储都是一个核心。而且，为了实现持久性和高可用性，预防网络分区或硬件故障，该键 / 值存储需要采用分布式部署。但这又带来一个问题，就是数据一致性很难保证，并发修改也很难处理。这就需要有一种并发控制机制来确保多个状态修改不会冲突。

为了实现并发控制，他们在实现 Amazon ECS 时使用了 Amazon 的其中一个核心分布式系统组件：一个基于 Paxos 算法以事务日志为基础的数据存储。该组件记录了每个数据条目的每次修改。每次写入操作都会作为日志中的一个事务提交，并且有一个特定的有顺序的 ID。数据存储中的当前值是根据日志记录所做的所有事务操作的总和。它允许 Amazon ECS 采用乐观并发的方式存储集群状态信息，在一个共享数据不断变化的环境中，这是非常合适的。

有了键 / 值存储，就可以协调集群了。而为了使用户能够利用 Amazon ECS 的状态管理功能，他们通过一组 API 开放了 Amazon ECS 集群管理器。用户可以通过它们以一种结构化的方式访问存储在键 / 值存储中的所有集群的状态信息。这组 API 成为用户在 Amazon ECS 上构建自己的解决方案的基础。Vogels 举了两个例子。

一个是自创建第一天起就托管在 AWS 上的免费叫车应用 Hailo。在过去的几年里，该应用从一个运行在单个 AWS 区域中的单体应用程序演化成为一个运行在多个区域中的基于微服务的架构。起初，每个微服务运行在一个实例集群上。但实例为静态分区，导致每个分区的资源利用率都不高。为此，他们决定基于服务优先级和其它指标在一个弹性资源池上调度容器。他们选择了 Amazon ECS，因为后者通过 API 完全暴露了集群状态，使他们可以使用满足特定应用需求的逻辑构建一个自定义的调度器。

另一个是教育类通讯软件 Remind。它起初是一个运行在 Heroku 上的大型单体应用。但随着用户数的增长，他们希望具备水平扩展的能力。因此，大约在 2014 年底，其工程团队开始探索使用容器迁移到微服务架构。他们希望在 AWS 上构建一个兼容 Heroku API 的 PaaS（平台即服务）。为了管理集群和容器编排，他们首先考察了一些开源解决方案，如 CoreOS 和 Kubernetes。但考虑到团队规模较小，他们没有时间管理集群基础设施及保持集群高可用。经过简单的评估之后，他们决定在 Amazon ECS 上构建他们的 PaaS。这样，工程团

队就可以专注于应用开发和部署。在 6 月份的时候，Remind 开源了他们的 PaaS 解决方案“Empire”。在接下来的几个月中，他们将把核心基础设施的 90% 迁移到 Empire 上。

总之，Amazon ECS 的架构提供了一种高可扩展、高可用、低延迟的容器管理服务。它允许以乐观并发的方式访问共享的集群状态信息，并通过 API 赋予用户创建自定义容器管理解决方案的能力。另外，Vogels 还提到，集群中实例的数量并不会对 Amazon ECS 的延迟产生明显的影响。

原文链接：http://www.infoq.com/cn/news/2015/07/Amazon-ECS

IAM 服务技术实践：《深入了解 AWS IAM 及访问控制服务》

作者　陈　天

编者按：本文系 InfoQ 中文站向陈天的约稿，这是 AWS 系列文章的第一篇。以后会有更多文章刊出，但并无前后依赖的关系，每篇都自成一体。读者若要跟随文章来学习 AWS，应该至少注册了一个 AWS 账号，事先阅读过当期所介绍服务的简介，并在 AWS management console 中尝试使用过该服务。否则，阅读的效果不会太好。

写在前面：访问控制，换句话说，谁能在什么情况下访问哪些资源或者操作，是绝大部分应用程序需要仔细斟酌的问题。作为一个志存高远的云服务提供者，AWS 自然也在访问控制上下了很大的力气，一步步完善，才有了今日的 IAM：Identity and Access Management。如果你要想能够游刃有余地使用 AWS 的各种服务，在安全上的纰漏尽可能地少，那么，首先需要先深入了解 IAM。

基本概念

按照 AWS 的定义：

IAM enables you to controlwho can do what in your AWS account.

它提供了用户（users）管理、群组（groups）管理、角色（roles）管理和权限（permissions）管理等供 AWS 的客户来管理自己账号下面的资源。

1、首先说用户（users）。在 AWS 里，一个 IAM user 和 unix 下的一个用户几乎等价。你可以创建任意数量的用户，为其分配登录 AWS management console 所需要的密码，以及使用 AWS CLI（或其他使用 AWS SDK 的应用）所需要的密钥。你可以赋予用户管理员的权限，使其能够任意操作 AWS 的所有服务，也可以依照 Principle of least privilege，只授权合适的权限。下面是使用 AWS CLI 创建一个用户的示例：

```
saws> aws iam create-user --user-name tyrchen
{
"User":{
"CreateDate":"2015-11-03T23:05:05.353Z",
"Arn":"arn:aws:iam::<ACCOUNT-ID>:user/tyrchen",
"UserName":"tyrchen",
"UserId":"AIDAISBVIGXYRRQLDDC3A",
"Path":"/"
}
}
```

当然，这样创建的用户是没有任何权限的，甚至无法登录，你可以用下面的命令进一步为用户关联群组，设置密码和密钥：

```
    saws> aws iam add-user-to-group --user-name
--group-name
    saws> aws iam create-login-profile --user-name
--password
    saws> aws iam create-access-key --user-name
```

2、群组（groups）也等同于常见的 unix group。将一个用户添加到一个群组里，可以自动获得这个群组所具有的权限。在一家小的创业公司里，其 AWS 账号下可能会建立这些群组：

- Admins ：拥有全部资源的访问权限
- Devs ：拥有大部分资源的访问权限，但可能不具备一些关键性的权限，如创建用户
- Ops ：拥有部署的权限
- Stakeholders ：拥有只读权限，一般给 manager 查看信息之用

创建一个群组很简单：

```
    saws> aws iam create-group --group-name stakeholders
    {
    "Group":{
    "GroupName":"stakeholders",
    "GroupId":"AGPAIVGNNEGMEPLHXY6JU",
    "Arn":"arn:aws:iam::<ACCOUNT-ID>:group/stakeholders",
    "Path":"/",
    "CreateDate":"2015-11-03T23:15:47.021Z"
    }
    }
```

然而，这样的群组没有任何权限，我们还需要为其添加 policy ：

```
    saws> aws iam attach-group-policy --group-name
--policy-arn
```

在前面的例子和这个例子里，我们都看到了 ARN 这个关键字。ARN 是 Amazon Resource Names 的缩写，在 AWS 里，创建的任何资源有其全局唯一的 ARN。ARN 是一个很重要的概念，它是访问控制可以到达的最小粒度。在使用 AWS SDK 时，我们也需要 ARN 来操作对应的资源。

policy 是描述权限的一段 JSON 文本，比如 AdministratorAccess 这个 policy，其内容如下：

```
    {
    "Version":"2012-10-17",
    "Statement":[
    {
    "Effect":"Allow",
    "Action":"*",
    "Resource":"*"
    }
    ]
    }
```

用户或者群组只有添加了相关的 policy，才会有相应的权限。

3、角色（roles）类似于用户，但没有任何访问凭证（密码或者密钥），它一般被赋予某个资源（包括用户），使其临时具备某些权限。比如说一个 EC2 实例需要访问 DynamoDB，

我们可以创建一个具有访问 DynamoDB 权限的角色，允许其被 EC2 Service 代入（AssumeRule），然后创建 EC2 的 instance-profile 使用这个角色。这样，这个 EC2 实例就可以访问 DynamoDB 了。当然，这样的权限控制也可以通过在 EC2 的文件系统里添加 AWS 配置文件设置某个用户的密钥（AccessKey）来获得，但使用角色更安全更灵活。角色的密钥是动态创建的，更新和失效都无须特别处理。想象一下如果你有成百上千个 EC2 实例，如果使用某个用户的密钥来访问 AWS SDK，那么，只要某台机器的密钥泄漏，这个用户的密钥就不得不手动更新，进而手动更新所有机器的密钥。这是很多使用 AWS 多年的老手也会犯下的严重错误。

4、最后是权限（permissions）。AWS 下的权限都通过 policy document 描述，就是上面我们给出的那个例子。policy 是 IAM 的核心内容，我们稍后详细介绍。

每年的 AWS re:invent 大会，都会有一个 session ：Top 10 AWS IAM Best Practices，感兴趣的读者可以去 YouTube 搜索。2015 年的 top 10（top 11）如下：

1 users: create individual users

2 permissions: Grant least priviledge

3 groups: manage permissions with groups

4 conditions: restrict priviledged access further with conditions

5 auditing: enable cloudTrail to get logs of API calls

6 password: configure a strong password policy

7 rotate: rotate security credentials regularly.

8 MFA: enable MFA (Multi-Factor Authentication) for priviledged users

9 sharing: use IAM roles to share access

10 roles: use IAM roles for EC2 instances

11 root: reduce or remove use of root

这 11 条 best practices 很清晰，我就不详细解释了。

按照上面的原则，如果一个用户只需要访问 AWS management console，那么不要为其创建密钥；反之，如果一个用户只使用 AWS CLI，则不要为其创建密码。一切不必要的，都是多余的——这就是安全之道。

使用 policy 做访问控制

上述内容你若理顺，IAM 就算入了门。但真要把握好 IAM 的精髓，需要深入了解 policy，以及如何撰写 policy。

前面我们看到，policy 是用 JSON 来描述的，主要包含 Statement，也就是这个 policy 拥有的权限的陈述，一言以蔽之，即：谁在什么条件下能对哪些资源的哪些操作进行处理。也就是所谓的撰写 policy 的 PARCE 原则：

- Principal ：谁
- Action ：哪些操作
- Resource ：哪些资源
- Condition ：什么条件
- Effect ：怎么处理（Allow/Deny）

我们看一个允许对 S3 进行只读操作的 policy ：

```
{
"Version":"2012-10-17",
"Statement":[
{
"Effect":"Allow",
"Action":[
"s3:Get*",
"s3:List*"
],
"Resource":"*"
}
]
}
```

其中，Effect 是 Allow，允许 policy 中所有列出的权限，Resource 是 *，代表任意 S3 的资源，Action 有两个：s3:Get* 和 s3:List*，允许列出 S3 下的资源目录，及获取某个具体的 S3 Object。

在这个 policy 里，Principal 和 Condition 都没有出现。如果对资源的访问没有任何附加条件，是不需要 Condition 的；而这条 policy 的使用者是用户相关的 principal（users, groups, roles），当其被添加

到某个用户身上时，自然获得了 principal 的属性，所以这里不必指明，也不能指明。

所有的 IAM managed policy 是不需要指明 Principal 的。这种 policy 可以单独创建，在需要的时候可以被添加到用户、群组或者角色身上。另一大类 policy 是 Resource policy，它们不能单独创建，只能依附在某个资源之上（所以也叫 inline policy），这时候，需要指明 Principal。比如，当我希望对一个 S3 bucket 使能 Web hosting 时，这个 bucket 里面的对象自然是要允许外界访问的，所以需要如下的 inline policy ：

```
{
"Version":"2012-10-17",
"Statement":[
{
"Effect":"Allow",
"Principal":"*",
"Action":"s3:GetObject",
"Resource":"arn:aws:s3:::corp-fs-web-bucket/*"
}
]
}
```

这里，我们对于 arn:aws:s3:::corp-fs-web-bucket/* 这个资源的 s3:GetObject，允许任何人访问（Principal: *）。

有时候，我们希望能更加精细地控制用户究竟能访问资源下的哪些数据，这个时候，可以使用 Condition。比如对一个叫 tyrchen 的

用户，只允许他访问 personal-files 这个 S3 bucket 下和他有关的目录：

```
{
"Version":"2012-10-17",
"Statement":[
{
"Action":[
"s3:ListBucket"
],
"Effect":"Allow",
"Resource":[
"arn:aws:s3:::personal-files"
],
"Condition":{
"StringLike":{
"s3:prefix":[
"tyrchen/*"
]
}
}
},
{
"Action":[
"s3:GetObject",
"s3:PutObject"
],
"Effect":"Allow",
"Resource":[
"arn:aws:s3:::personal-files/tyrchen/*"
]
}
]
}
```

这里我们用到了 StringLike 这个 Condition，只有当访问的 s3:prefix 满足 tyrchen/* 时，才为真。我们还可以使用非常复杂的 Condition ：

```
"Condition":{
"IPAddress":{"aws:SourceIP":["10.0.0.0/8","4.4.4.4/32"]},
"StringEquals":{"ec2:ResourceTag/department":"dev"}
}
```

在一条 Condition 下并列的若干个条件间是 and 的关系，这里 IPAddress 和 StringEquals 这两个条件必须同时成立；在某个条件内部则是 or 的关系，这里 10.0.0.0/8 和 4.4.4.4/32 任意一个源 IP 都可以访问。这个条件最终的意思是：对于一个 EC2 实例，如果其 department 标签是 dev，且访问的源 IP 是 10 网段的内网地址或者 4.4.4.4/32 这个外网地址，则 Condition 成立。

讲完了 Condition，我们再回到之前的 policy。

这条 policy 里有两个 statement，前一个允许列出 arn:aws:s3:::personal-files 下 prefix 是 tyrchen/* 里的任何对象；后一个允许读写 arn:aws:s3:::personal-files/tyrchen/* 里的对象。注意这个 policy 不能写成：

```
{
"Version":"2012-10-17",
"Statement":[
```

```
{
"Action":[
"s3:ListObject",
"s3:GetObject",
"s3:PutObject"
],
"Effect":"Allow",
"Resource":[
"arn:aws:s3:::personal-files/tyrchen/*"
]
}
]
}
```

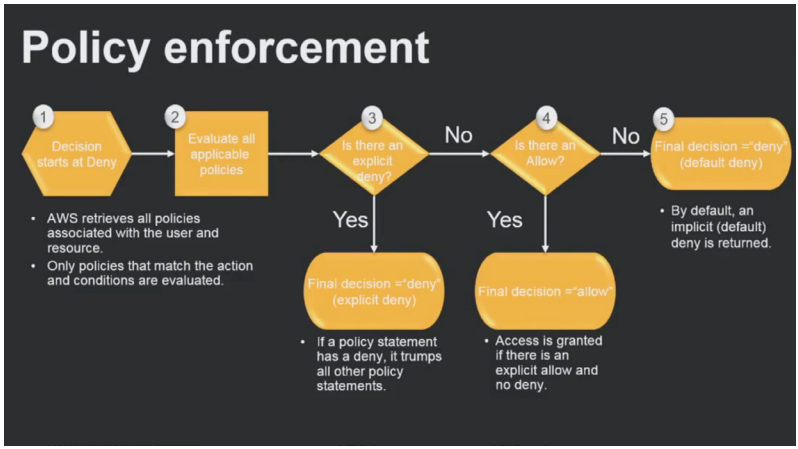
因为这样的话，用户在 AWS management console 连在 personal-files 下列出 /tyrchen 这个根目录的机会都没有了，自然也无法进行后续的操作。这样的 policy 其权限是不完备的。

上面的 policy 可以被添加到用户 tyrchen 身上，这样他就可以访问自己的私人目录；如果我们创建一个新用户叫 lindsey，也想做类似处理，则需要再创建一个几乎一样的 policy，非常不符合 DRY（Don’t Repeat Yourself）原则。好在 AWS 也考虑到了这一点，它支持 policy variable，允许用户在 policy 里使用 AWS 预置的一些变量，比如 ${aws:username}。上述的 policy 里，把 tyrchen 替换为 ${aws:username}，就变得通用多了。

以上是 policy 的一些基础用法，下面讲讲 policy 的执行规则，它也是几乎所有访问控制方案的通用规则：

- 默认情况下，一切资源的一切行为的访问都是 Deny
- 如果在 policy 里显式 Deny，则最终的结果是 Deny
- 否则，如果在 policy 里是 Allow，则最终结果是 Allow
- 否则，最终结果是 Deny

如下图：



什么情况下我们会用到显式 Deny 呢？请看下面的例子：

```
{
"Version":"2012-10-17",
"Statement":[
{
"Effect":"Allow",
"Action":[
"dynamodb:*",
"s3:*"
```

```
    ],
    "Resource":[
    "arn:aws:dynamodb:AWS-REGION-IDENTIFIER:ACCOUNT-
ID-WITHOUT-HYPHENS:table/EXAMPLE-TABLE-NAME",
    "arn:aws:s3:::EXAMPLE-BUCKET-NAME",
    "arn:aws:s3:::EXAMPLE-BUCKET-NAME/*"
    ]
    },
    {
    "Effect":"Deny",
    "NotAction":[
    "dynamodb:*",
    "s3:*"
    ],
    "NotResource":[
    "arn:aws:dynamodb:AWS-REGION-IDENTIFIER:ACCOUNT-
ID-WITHOUT-HYPHENS:table/EXAMPLE-TABLE-NAME",
    "arn:aws:s3:::EXAMPLE-BUCKET-NAME",
    "arn:aws:s3:::EXAMPLE-BUCKET-NAME/*"
    ]
    }
    ]
    }
```

在这个例子里，我们只允许用户访问 DynamoDB 和 S3 中的特定资源，除此之外一律不允许访问。我们知道一个用户可以有多重权限，属于多个群组。所以上述 policy 里的第一个 statement 虽然规定了用户只能访问的资源，但别的 policy 可能赋予用户其他资源的访问权限。所以，我们需要第二条 statement 封死其他的可能。按照之

前的 policy enforcement 的规则，只要看见 Deny，就是最终结果，不会考虑其他 policy 是否有 Allow，这样杜绝了一些隐性的后门，符合 Principle of least privilege。

我们再看一个生产环境中可能用得着的例子，来证明 IAM 不仅「攘内」，还能「安外」。假设我们是一个手游公司，使用 AWS Cognito 来管理游戏用户。每个游戏用户的私人数据放置于 S3 之中。我们希望 congito user 只能访问他们各自的目录，IAM policy 可以定义如下：

```
    {
    "Version":"2012-10-17",
    "Statement":[
    {
    "Effect":"Allow",
    "Action":["s3:ListBucket"],
    "Resource":["arn:aws:s3:::awesome-game"],
    "Condition":{"StringLike":{"s3:prefix":["cogn
ito/*"]}}
    },
    {
    "Effect":"Allow",
    "Action":[
    "s3:GetObject",
    "s3:PutObject",
    "s3:DeleteObject"
    ],
    "Resource":[
    "arn:aws:s3:::awesome-game/cognito/${cognito-
```

```
identity.amazonaws.com:sub}",
    "arn:aws:s3:::awesome-gamecognito/${cognito-
identity.amazonaws.com:sub}/*"
    ]
    }
    ]
    }
```

最后，讲一下如何创建 policy。很简单：

```
$ aws iam create-policy --policy-name
   --policy-document
```

其中 policy-document 就是如上所示的一个个 JSON 文件。

参考文档

- IAM policy overview

http://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/access_policies.html

- policy variables

http://docs.aws.amazon.com/zh_cn/IAM/latest/UserGuide/reference_policies_variables.html

- policy evaluation logic

http://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html

原文链接：http://www.infoq.com/cn/articles/aws-iam-dive-in

微服务架构实践：《Gilt ION-Roller：如何在 AWS 上部署微服务架构》

作者　Natalia Bartol, Gary Coady, Adrian Trenaman

经过七年的发展，gilt.com 已经从一个使用 Ruby on Rails 开发的创业公司成长为使用 Scala 微服务架构的主流电子商务平台。Gilt 的限时抢购商业模式的基础是：在短时间内会涌入大量的客户访问，以竞买某些限量的奢侈品。通过使用微服务架构，它为我们的服务提供了可伸缩性、性能以及可靠性的结合，同时也为我们的开发团队带来了自治性、自主性以及灵活性。团队可以自由地选择编程语言、框架、数据库以及构建系统，为核心的网站功能、移动应用、个性化算法、实时数据源以及通知功能创建服务。

随着软件服务的大爆发，随之出现的问题是如何部署与运行我们所创建的代码。在 Gilt 发展的早期，它们的软件运行在“裸机”之上，部署在一个、随后升级为两个传统的数据中心之内。团队需要申请新的硬件设置，随后通过自定义脚本和 Capistrano 的混合使用进行部署。这种方式虽然能够运作，但也带来了很大的困难。对虚拟化技术的首次尝试表现为在每日的访问高峰时性能表现不佳，大量的服务最终在同一个物理环境内相互交织在一起，导致了性能方面的异常，并且有时会由于缺乏资源的分离性而导致停机。在访问高峰时，对于

某个服务进行大规模的部署会占用“每次请求的多个线程”，由此造成了整个网站的瘫痪。

我们曾经尝试了多种技术，并且创建了各种工具，通过在生产环境中进行测试和持续部署等手段试图减轻这方面的问题。但经过研究发现，我们用于在数据中心的硬件上分配并自动设置服务的方式已经被证实是一种非常耗时、并且非常困难的方式。

因此我们开始考虑将微服务基础设施大规模地转移到云环境中，主要是指 Amazon Web Services（AWS）环境，并且提炼出不可变部署（Immutable Deployment）这一概念，它是由我们之前在函数式编程中所使用的不可变变量的经验所启发的。我们将一套原本基于自身的数据中心打造的工具“ION-Cannon”升级为一套即将开源的工具“ION-Roller”，促使我们开发 ION-Roller 这套工具的动力在于：

- 允许团队通过声明式的方式指定如何将他们的服务部署到 AWS，例如可用的节点数的最小或最大值、每个实例的大小等等。
- 提供一条将服务与应用作为不可变的 Docker 镜像部署到生产环境中的管道，可支持分阶段式发布，并且能够分阶段地将访问从老版本的服务转移到新版本的服务上。
- 在大型发布的时候支持快速回滚，能够使一组已经处于“热”状态的实例仍然保持运行之前的版本。

在本文中，我们将谈论 ION-Roller 中的一些核心概念，以及相

关的技术和实现途径，并且简要地叙述它的使用方式。

ION-Roller，基于云的微服务王国

我们首先将微服务世界想象成为构建在独立的、不可变的环境之上的 HTTP 终结点，通过 REST API 的方式进行访问。

HTTP 终结点

Gilt 使用 HTTP 作为系统间通信的传输封装，对于用户来说，HTTP 终结点表现为一个主机名与端口的组合（在它之上还有一个发现层）。我们的想法是，当组织中的软件与配置随着时间而演变时，将这个终结点作为一个组织级的概念进行使用。

虽然这个概念很简单，但可以实现许多实用的目标。通过对代理与 / 或网络配置的合理应用，我们就能够提供以下特性：

12 逐步地发布新的软件版本

13 安全地回滚至之前版本的软件或配置

14 通过错误率以及延迟情况的监控，检测到异常情况的发生

15 能够了解哪些软件提供了终结点

16 能够了解软件或配置随时间进行变更的信息

在描述环境时配置胜于代码

在运维规模非常小的情况下，在部署过程中要求得到不受限制

的灵活性看起来是一种合理的诉求。但随着规模的增加，这种方式很快就变得难以管理、监控及理解了。

声明式的配置在描述及管理部署方面是一种实用的工具，它能够以一种结构化的风格进行管理。从配置的角度来说，它能完成的任务比起代码来说更为有限，但也因此能够对请求进行分析与报告。

在软件部署中进行风险管理

软件发布包括了各种权衡：灵活性与简便性、以及速度与安全性。

大多数生产环境的发展方向是转向新的部署流程，这种流程能够在特性开发或 bug 修复完成之后在更短的时间之内让用户看到这些变更。这也表示对生产环境进行变更的机率更高。由于每次发布都内含着风险，因此这一策略也有它的不利之处。为了抵消这种不利情况，团队需要采取一些能够缓解每次发布中的风险的发布流程，让每次发布都成为一个日常的、安全的操作。

风险体现在多个方面：异常的数量和严重度、受影响用户的数量及比例、以及从故障中恢复的时间长短。

发布流程对于异常的数量以及异常的严重度的影响并没有开发流程那么高，但频繁的发布通常能够使找到某个异常发生原因的速度更快（因为每个发布中的变量数量减少了）。

我们所能做的就是减少受影响的用户数量，以及减少修复的

时间。在生产环境上进行测试、部分发布以及金丝雀部署（canary release）等方式正适用于这一场景。如果能够在没有任何生产环境中访问的情况下对新的发布进行测试，那么就能够将影响降至最低（甚至为零），但这种方式能够找到的异常数量也是最少的。由于许多异常情况只在常规的访问中才会出现，因此一台金丝雀机器（运行新软件的一个单一实例）是很有价值的工具，通过它可以了解新的发布是否适于使用。目前为止，对用户的影响依然是很小的，这取决于你如何对服务进行分片。保持渐进式的发布过程也是通过将受影响的比例最小化，将 bug 对用户的影响降至最低。

从故障中恢复的时间是不可变软件方式的一个主要优点，因为旧版本的软件依然“可用”。一旦发现异常情况，可以以最小的风险在最短的时间内回到之前的旧版本。我们稍后将进一步对不可变部署展开讨论。

运行中软件的期望状态以及实际状态

我们希望持续地监控某个终结点背后系统的期望运行状态以及实际运行状态的差别。如果这种差别确实存在，那么我们就应当（逐渐地）让实际状态向期望状态靠拢。我们持续地对运行环境进行大量的测试（包括运行的软件、负载均衡器设置、DNS 等等），如果其中任何一项不能够满足需求，就将它替换掉。

举例来说，假设我们知道访问应当由某个特定版本的软件进行

处理，却发现没有任何一台运行这个版本的服务器是可用的（可以出于任何原因，包括某人无意中移除了这些服务器），那么就应当请求进行部署。

不可变部署

“不可变部署”的思想是通过一个已定义的、易于理解的配置部署软件，而且这个配置不会随着时间而改变。软件与配置都是发布中的自然组成，你不能通过修改数据中的某些地方随后重启的方式对软件进行变更，而是重新生成该软件的一个副本，让其中包含经过更新的细节内容。

这种思想能够产生更易于预测的环境，并且当某个新发布被证实“有问题”的情况下，能够回滚至软件的老版本而无需进行重启。（即使能够找到老版本的软件，在有些情况下要完全启动它也是一件费时费力的任务，因为它可能需要进行加载缓存等工作。）

在一个小规模的环境中，可能只存在少量的运行服务器，因此可能只有在发布完全结束之后，才能够发现新的发布中出现的异常情况。在传统的非不可变发布场景中，所有旧版本的软件都已经被替换了。

不可变部署方式在我们目前的服务中的实现还存在着一点不足之处。因为所有的环境都是按需搭建的，因此搭建时间取决于某些因素，例如启动新 EC2 实例的时间、以及下载适当的 Docker 镜像的时

间。与之相比，一些其它部署工具要求存在一个机器池，不要求在软件安装之前搭建机器。因此我们选择为可回滚性牺牲了一些低延迟特性，因此在软件发布后的初次启动会有较高的延迟。

Docker 镜像

在微服务环境中，将工具与编程语言的选择从部署环境中抽象出来是一种非常实用的概念。它允许个别团队在开发软件时具有更多的自治性。虽然这方面的选择已经有许多，从特定于操作系统的包管理系统——例如 RPM，到特定于云提供商的平台——例如 Packer，但我们最终还是决定用 Docker 实现这一目标。Docker 在选择部署环境与策略方面提供了很高的灵活性，并且十分符合我们的需求。

重新发明轮子？

当我们在寻找能够方便地管理基于不可变 web 服务器的基础设施时，所找到的选择是非常有限的。大多数现有的软件都是为了便于软件更新进行优化的，但这不是我们所感兴趣的更新方式。我们也希望能够管理 web 服务器流量的整个生命周期，但没有发现任何一种对这一任务进行优化的选择。

另一点需要考虑的是，到底是要求开发者学习并理解某种现有的部署工具集所带来的价值更高，还是说通过一种无冲突的、简化的工具，能够满足我们将某个特定版本的软件部署到生产环境中的方式

能够带来更高的生产力。我们相信，开发者应当能够简单地发布软件，而无需理解复杂的底层机制（同时也保留进行自定义的可能性，这在一些高级场景中是必需的）。

为什么不选择 Puppet、Chef 等工具

许多工具采取的都是以机器为中心的视角，配置是在每一台机器的基础上完成的。我们则采取了系统状态这一更广泛的视角（比如我们需要某个软件的四个拷贝以提供服务），而并不太关心个别机器的情况。我们利用了 AWS 所提供的高层概念，例如将机器运行情况检查时总是失败的那些机器进行替换，以及当 HTTP 终结点的访问量激增的情况下动态地扩展它的能力等等。

为什么不选择 CodeDeploy

CodeDeploy 是一套由 Amazon 提供的软件发布管理系统，但它无法满足我们支持不可变基础设施的需求。虽然很容易搭建脚本，使用 CodeDeploy 发布软件，但你需要预先搭建你的环境。此外，CodeDeploy 本身没有内置的部署 Docker 镜像的功能。

为什么不选择简单的 Elastic Beanstalk

Elastic Beanstalk 提供了创建环境的各种功能，它允许你运行 Docker 镜像，并且创建大量的支持性系统，例如 EC2 实例、负载均

衡器、自动扩展组（根据访问量改变服务器的数量）。它还允许对日志文件的访问，并且可以对这些系统进行一定程度的管理工作。

但是，它对于“不可变部署”概念的支持非常有限，对于某部分软件的多次发布将为同一个用户可见的终结点带来大量的访问量，然后逐渐地转移访问量。对于这一点，它唯一支持的能力就是“切换CNAMEs”，这是一种非常粗糙的转移访问量的方式，因为所有的访问量都会瞬间转移到新的环境中。此外，出于 DNS 的本质，它在可靠性方面也存在问题，因为 DNS 查找结果在 DNS 进行变更后依然会被缓存一段时间，并且某些糟糕的客户端会忽略 DNS TTL 值，导致访问量依然在很长的一段时间内会被发送给旧的环境。

此外，Elastic Beanstalk 没有提供某种高层次的结构，以帮助你理解在某个终结点的背后有哪些东西运行在生产环境上。“有哪些运行正在运行，它们的配置情况又是怎样的”这一问题并不容易解答，需要某些高层次的系统辅助。

我们决定利用 Elastic Beanstalk 作为一种实用的方法以部署 Docker 软件，但需要有层次地进行适当的管理，并且对它的层次进行控制，以便为用户提供一个完整的工作流。

介绍 ION-Roller

ION-Roller 是一套服务（包含 API、web 应用和命令行界面），它利用了 Amazon 的 Elastic Beanstalk 及其底层的 CloudFormation 框

架的功能，将 Docker 镜像部署到 EC2 实例中。

开始使用 ION-Roller，你需要进行一些简单的操作

启动部署软件过程的全部工序如下：

- 准备一个 AWS 帐号，并在 ION-Roller 中为其授权，它能够为你启动实例并访问资源（如果你的组织在运行软件时使用了多个 AWS 帐户运行，ION-Roller 能够为所有这些帐号提供一个单一的部署视角，只需具有足够的权限即可）。
- 由某个 Docker registry（可以选择 hub.docker.com，或使用私有 Docker registry 服务）提供的 Docker 镜像
- 准备软件的部署规格说明，包括 HTTP 终结点、EC2 实例的数量与类型、Docker 镜像的运行时参数、环境变量、安全设置等等。它们将作为你的服务配置的一部分提供，并通过 REST API 提交至 ION-Roller。

如果想了解使用 AWS 帐户安装 ION-Roller 的细节与完整的文档，敬请期待即将开源的这个项目 https://github.com/gilt/ionroller。

使用 ION-Roller 部署软件

你可以通过内置的命令行工具简单地启动部署过程：

```
ionroller release <SERVICE_NAME><VERSION>
```

这个工具能够在发布过程中提供即时的反馈信息：

```
    [INFO] NewRollout(ReleaseVersion(0.0.17))
    [INFO] Deployment started.
    [INFO] Added environment: e-k3bybwxy2f
    [INFO] createEnvironment is starting.
    [INFO] Using elasticbeanstalk-us-east-1-830967614603
as Amazon S3 storage bucket for environment data.
    [INFO] Waiting for environment to become healthy.
    [INFO] Created security group named: sg-682b430c
    [INFO] Created load balancer named: awseb-e-k-
AWSEBLoa-A4GOD7JFELTF
```

你也可以选择以编程的方式触发一次部署过程，ION-Roller 提供了一套 REST API，可以对你的配置和发布过程进行完全控制。

在系统的背后，ION-Roller 会触发一个 Elastic Beanstalk 部署流程，充分地利用了它的功能，包括创建一个负载均衡器、安全以及自动扩展组、设置 CloudWatch 监控，并从某个 Docker registry 中获取特定的 Docker 镜像。

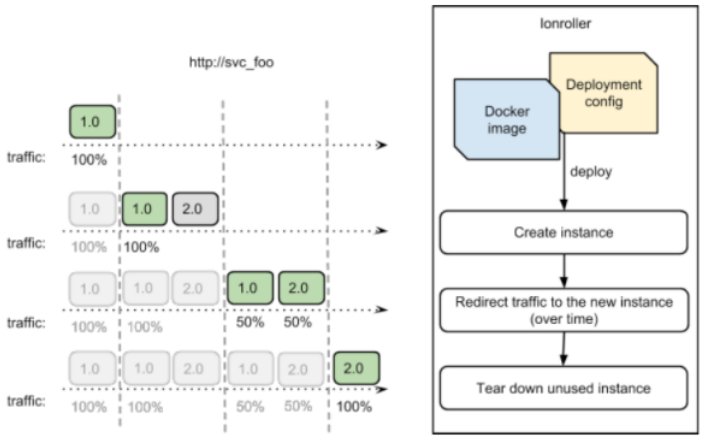
访问重定向

一旦 ION-Roller 检测到部署成功之后，它就能够安全地逐渐将访问从老版本转移到新版本的服务。

访问的重定向是通过对 EC2 实例的修改实现的，它能够通过负载均衡器对 HTTP 请求进行响应。随着发布的流程推进，新部署的实例会逐渐加入负载均衡器的配置中，而原先部署的实例会逐步被删

除。这一流程的启动时机是可配置的。

渐进式的访问重定向使你能够对最近的发布进行监控、快速地检测到失败、并在必要时进行回滚。



软件回滚

由于在发布过程中，老的环境依然可访问，老版本的软件也依然在运行中，因此我们可以安全地将软件回滚到之前的某个老版本。无用的旧实例在一段时间（可配置）之后才会被移除，由于这段延迟的存在，因此我们在发布全部完成之后的一段时间内仍然可以选择回滚。

我们的目标是持续地监控终结点的运行情况，并且在检测

到异常的情况下自动地回滚到老的版本。我们将使用 Amazon 的 CloudWatch 警告功能发出一个即将进行回滚操作的信号。

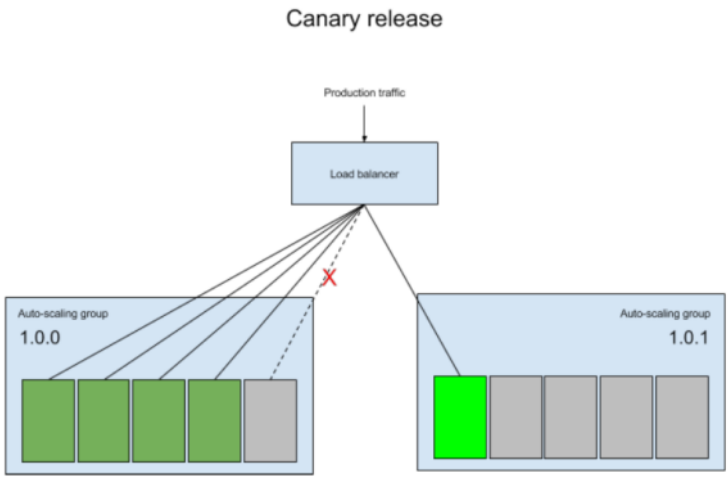
进行手动回滚非常简单，运行以下脚本即可：

```
ionroller release <SERVICE_NAME><PREVIOUS_VERSION>
```

如果旧的实例仍然可用，那么只需几秒钟的时间即可将访问全部转回老版本上。当然，前提是你没有进行任何使老版本的软件不可用的操作（例如更新数据存储系统的 schema 等等）。如果你希望依靠这种能力以回滚软件，那么牢记这一点是非常重要的，无论你使用的是哪一种部署系统。

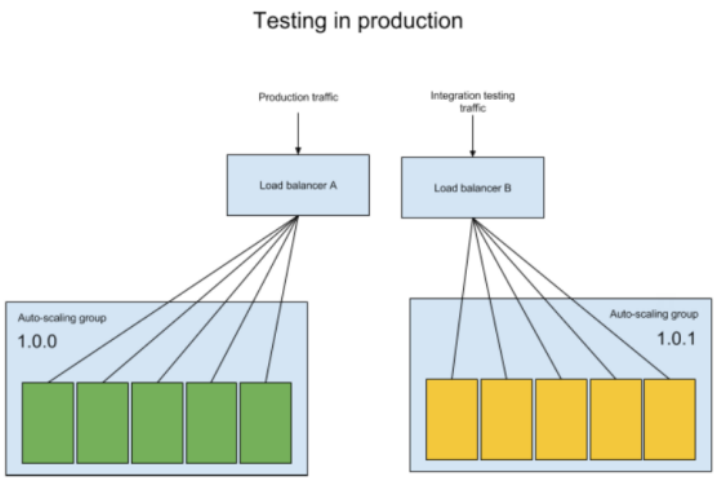
金丝雀发布以及在生产环境中测试

ION-Roller 通过对访问转移流程的配置，支持金丝雀发布的概念。当有新的版本部署到一些初始的实例之后，该流程就会停止，可以对生产环境中的访问进行一些发布测试。在一段可配置的时间之后，发布流程将会继续推进。



在生产环境进行测试

对于某些用例，你希望能够对新的软件进行测试（或演示），而又不希望产生实际的访问，那么我们需要 ION-Roller 搭建一个分离的 HTTP 终结点，在生产环境的终结点更新之前，可以通过它处理请求。



留意系统状况 - ION-Trail

ION-Roller 能够查看某个终结点背后的环境与软件随着时间推移产生的变化，在对环境进行监控或进行变更的过程中，它将记录与环境生命周期或部署活动相关的事件。可以通过这一功能实现系统的审计、监控以及报表功能。ION-Trail 是一个支持性的服务，为所有被记录的部署活动提供了一个事件源。

结论——去中央化的 DevOps

希望读者在阅读本文后能够对于我们将微服务架构部署到 AWS 的思想有所了解。ION-Roller 允许我们将整个 DevOps 组织实现去中

央化，并且通过声明式的部署让工程师更易于掌握。ION-Roller 允许我们进行分阶段的发布与热回滚，并且支持诸如“金丝雀发布”与“在生产环境中测试”等特性。如果想了解更多的信息，请关注 Gilt Tech 博客（http://tech.gilt.com），我们很快就会在博客上宣布 ION-Roller 开源项目的公开发布。

关于作者

Natalia Bartol 曾在 IBM 担任 Eclipse 支持工程师，从事改善开发者工具的工作，随后在 Zend Technologies 担任 Eclipse 开发人员及团队主管。如今她在 Gilt 担任软件工程师，专注于微服务的部署以及提高开发者的生产力。Natalia 拥有波兹南科技大学软件工程专业的理科硕士证书。

Gary Coady 是来自 Gilt Groupe 的一位高级软件工程师，他的工作是自动化部署、编写构建工具以及传授 Scala 技术。他之前曾在 Google 担任网站可靠性工程师，将大量时间用于剖析、管理以及处理大规模环境中的运维异常。Gary 拥有都柏林大学圣三一学院计算机科学专业的学士学位（Mod）。

Adrian Trenaman 在位于爱尔兰都柏林的 gilt.com 担任工程师高级副总裁。他拥有爱尔兰国立梅努斯大学计算机科学专业的博士学位，和爱尔兰管理学院的商业开发专科文凭，以及都柏林大学圣三一学院计算机科学专业的学士学位（Mod. Hons）。

原文链接：http://www.infoq.com/cn/articles/gilt-deploying-microservices-aws

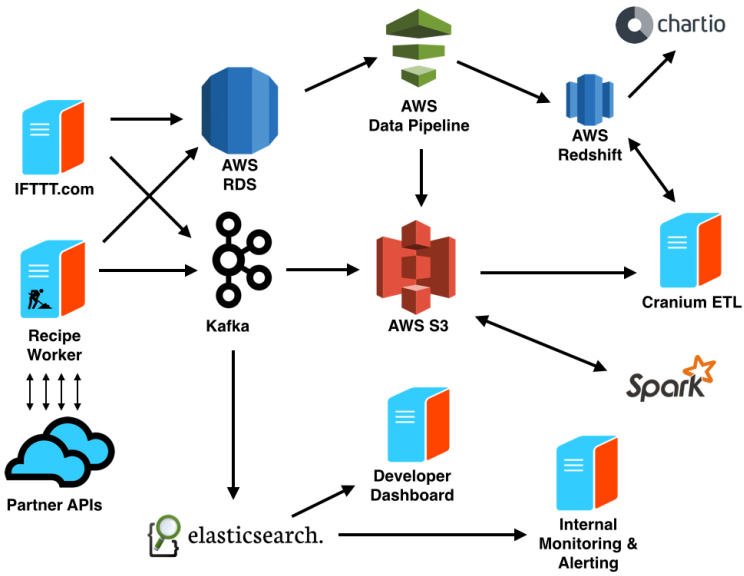
英文原文链接：http://www.infoq.com/articles/gilt-deploying-microservices-aws

数据管理服务应用案例：《IFTTT ：借助 AWS 数据管理服务构建高性能数据架构》

作者　张天雷

随着信息技术的发展，人们在日常生活和工作中都不可避免的要用到邮箱、聊天工具、云存储等网络服务。然而，这些服务很多时候都是单独运行的，不能很好的实现资源共享。针对该问题，IFTTT 提出了“让互联网为你服务”的概念，利用各网站和应用的开放 API，实现了不同服务间的信息关联。例如，IFTTT 可以把指定号码发送的短信自动转发邮箱等。为了实现这些功能，IFTTT 搭建了高性能的数据架构。近期，IFTTT 的工程师 Anuj Goyal 对数据架构的概况进行了介绍，并分享了在操作数据时的一些经验和教训。

在 IFTTT，数据非常重要——业务研发和营销团队依赖数据进行关键性业务决策；产品团队依赖数据运行测试 / 了解产品的使用情况，从而进行产品决策；数据团队本身也依赖数据来构建类似 Recipe 推荐系统和探测垃圾邮件的工具等；甚至合作伙伴也需要依赖数据来实时了解 Channel 的性能。鉴于数据如此重要，而 IFTTT 的服务每天又会产生超过数十亿个事件，IFTTT 的数据框架具备了高度可扩展性、稳定性和灵活性等特点。接下来，本文就对数据架构进行详细分析。



数据源

在 IFTTT，共有三种数据源对于理解用户行为和 Channel 性能非常重要。首先，AWS RDS 中的 MySQL 集群负责维护用户、Channel、Recipe 及其相互之间的关系等核心应用。运行在其 Rails 应用中的 IFTTT.com 和移动应用所产生的数据就通过 AWS Data Pipeline，导出到 S3 和 Redshift 中。其次，用户和 IFTTT 产品交互时，通过 Rails 应用所产生的时间数据流入到 Kafka 集群中。最后，为了帮助监控上百个合作 API 的行为，IFTTT 收集在运行 Recipe 时所产生的 API 请求的信息。这些包括反应时间和 HTTP 状态代码的信息

同样流入到了 Kafka 集群中。

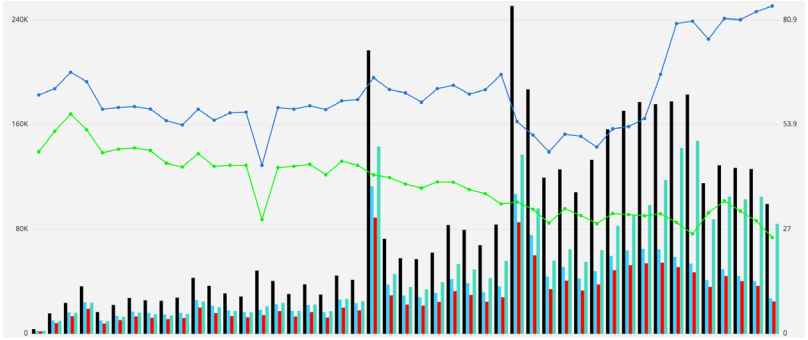
IFTTT 的 Kafka

IFTTT 利用 Kafka 作为数据传输层来取得数据产生者和消费者之间的松耦合。数据产生者首先把数据发送给 Kafka。然后，数据消费者再从 Kafka 读取数据。因此，数据架构可以很方便的添加新的数据消费者。

由于 Kafka 扮演着基于日志的事件流的角色，数据消费者在事件流中保留着自己位置的轨迹。这使得消费者可以以实时和批处理的方式来操作数据。例如，批处理的消费者可以利用 Secor 将每个小时的数据拷贝发送到 S3 中；而实时消费者则利用即将开源的库将数据发送到 Elasticsearch 集群中。而且，在出现错误时，消费者还可以对数据进行重新处理。

商务智能

S3 中的数据经过 ETL 平台 Cranium 的转换和归一化后，输出到 AWS Redshift 中。Cranium 允许利用 SQL 和 Ruby 编写 ETL 任务、定义这些任务之间的依赖性以及调度这些任务的执行。Cranium 支持利用 Ruby 和 D3 进行的即席报告。但是，绝大部分的可视化工作还是发生在 Chartio 中。



而且，Chartio 对于只了解很少 SQL 的用户也非常友好。在这些工具的帮助下，从工程人员到业务研发人员和社区人员都可以对数据进行挖掘。

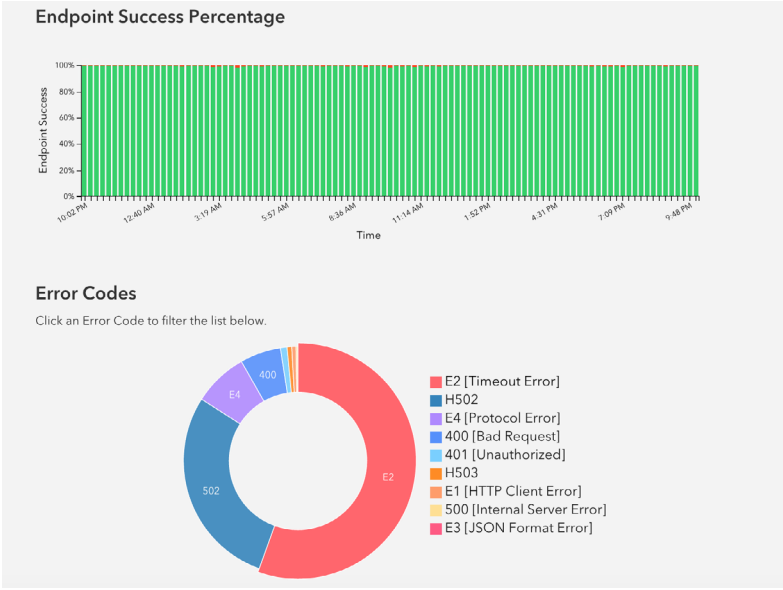
机器学习

IFTTT 的研发团队利用了很多机器学习技术来保证用户体验。对于 Recipe 推荐和问题探测，IFTTT 使用了运行在 EC2 上的 Apache Spark，并将 S3 当作其数据存储。

实时监控和提醒

API 事件存储在 Elasticsearch 中，用于监控和提醒。IFTTT 使用 Kibana 来实时显示工作进程和合作 API 的性能。在 API 出现问题时，IFTTT 的合作者可以访问专门的 Developer Channel，创建 Recipe，从而提醒实际行动（SMS、Email 和 Slack 等）的进行。

在开发者视图内，合作者可以在 Elasticsearch 的帮助的帮助下访问 Channel 健康相关的实时日志和可视化图表。开发者也可以通过这些有力的分析来了解 Channel 的使用情况。



经验与教训

最后，Anuj 表示，IFTTT 从数据架构中得到的教训主要包括以下几点：

- 通过 Kafka 这样的数据传输层实现的生产者和消费者的隔离非常有用，且使得 Data Pipeline 的适应性更强。例如，一些比较慢的消费者也不会影响其他消费者或者生产者的性能。
- 从一开始就要使用集群，方便以后的扩展！但是，在因为性能问题投入更多节点之前，一定要先认定系统的性能瓶颈。例如，在 Elasticsearch 中，如果碎片太多，添加更多的节点或许并不会加速查询。最好先减少碎片大小来观察性能是否改善。
- 在类似的复杂架构中，设置合适的警告来保证系统工作正常是非常关键的！ IFTTT 使用 Sematext 来监控 Kafka 集群和消费者，并分别使用 Pingdom 和 Pagerduty 进行监控和提醒。
- 为了完全信任数据，在处理流中加入若干自动化的数据验证步骤非常重要！例如，IFTTT 开发了一个服务来比较产品表和 Redshift 表中的行数，并在出现异常情况时发出提醒。
- 在长期存储中使用基于日期的文件夹结构（YYYY/MM/DD）来存储事件数据。这样存储的事件数据可以很方便的进行处理。例如，如果想读取某一天的数据，只需要从一个文件夹

中获取数据即可。

- 在 Elasticsearch 中创建基于时间（例如，以小时为单位）的索引。这样，如果试图在 Elasticsearch 中寻找过去一小时中的所有 API 错误，只需要根据单个索引进行查询即可。
- 不要把单个数据马上发送到 Elasticsearch 中，最好成批进行处理。这样可以提高 IO 的效率。
- 根据数据和查询的类型，优化节点数、碎片数以及每个碎片和重复因子的最大尺寸都非常重要。

原 文 链 接：http://www.infoq.com/cn/news/2015/11/ifttt-data-infrastructure

数据存储服务应用案例：《纳斯达克首席架构师：利用 Amazon EMR 与 S3 建立统一化数据仓库平台》

作者　Nate Sammons

这是一篇由纳斯达克首席架构师 Nate Sammons 撰写的文章。

纳斯达克集团公司在全球范围内负责金融交易运营工作，且每天处理的数量总量极为庞大。我们运行着种类繁多且数量可观的分析及监控系统，而且这些系统全部需要访问同样的整体数据集。

纳斯达克集团自 Amazon Redshift 发布之日起就开始将其引入自身业务体系，我们也对这一决定感到由衷赞赏。我们此前已经在 re:Invent 大会上多次探讨过该系统的客户使用情况，最近的一次来自《FIN401 地震式转变：纳斯达克向 Amazon Redshift 迁移》一文。目前，我们的系统每天将平均 55 亿行数据移动至 Amazon Redshift 当中（2014 年 10 月的单日峰值移动量为 140 亿行）。

除了我们的 Amazon Redshift 数据仓库之外，我们还拥有一套庞大的历史数据设施体系，旨在将其作为单一、巨型数据集加以访问。目前，这套历史数据归依体系分布在大量不同系统当中，这使相关内容变得难于使用。我们的目标是建立起一套新的双重统一化数据仓库平台，其一方面需要为纳斯达克各内部部门带来更为理想的历史数

据集访问能力，另一方面则希望在流程当中实现更突出的成本效益。对于这套平台，Hadoop 是一项明确的选项：它支持多种不同类型的 SQL 及其它用于数据访问的接口，同时拥有一整套活跃且持续发展的工具及项目生态系统。

为什么选择 Amazon S3 与 Amazon EMR？

我们新型数据仓库平台的主要构建宗旨是为了将存储与计算资源加以分离。在传统 Hadoop 部署机制当中，存储容量的伸缩通常也要求我们对计算容量进行伸缩，而且计算与存储资源二者之间的任何比例变更都要求使用者对硬件进行修改。在我们的长期历史归档用例当中，HDFS 当中访问频率较低的数据仍然需要保持始终可用，并一直占用集群当中每个节点的计算资源。

除此之外，HDFS 的默认复制因子为 3，这意味着每个数据块需要占用集群中的三个节点。这虽然能够在一定程度上保障耐久性水平，但也同时代表着大家必须购买必要的存储容量、从而在对集群进行容量扩展时将磁盘数量增加至预期原始容量的三倍。除此之外焦点问题也需要得到关注：如果某个给定数据块在集群中只存在于三个节点当中，但却需要经受频率极高的访问操作，那么大家必须对其进行复制或者将特定计算节点设置为访问焦点。

不过在 Amazon S3 与 Amazon EMR 的帮助下，我们能够顺利摆脱上述难题。二者允许大家将用于数据仓库的计算与存储资源进行分

离，并根据需要单独对其进行规模伸缩。其中不再存在焦点，因为S3 中的每个对象都能够以无需焦点机制的支持下与任意其它对象一下接受访问。Amazon S3 还拥有几乎无限的可扩展能力、11 个 9 的卓越耐久性（即 99.999999999%）、自动跨数据中心复制、简洁直观的跨区域复制以及令人心动的低廉成本。随着与 IAM 政策的进一步集成，S3 还带来一套拥有经过深入细化调整且涵盖多个 AWS 账户访问控制机制的存储解决方案。考虑到上述理由，Netflix 公司以及其它众多企业已经证明了 Amazon S3 是一套极具可行性的数据仓库平台。此外，AWS Lambda 等新型服务的出现能够针对 S3 事件执行任意代码作为响应，而这将进一步拓展用户可资挖掘的使用潜力。

利用 EMR 启用集群

EMR 的出现令使用者能够更轻松地部署并管理 Hadoop 集群。我们能够根据实际需要对集群规模与提升与削减，并在周末或者节假日期间将其关闭。一切元素都运行在虚拟私有云（简称 VPC）环境之下，因此我们能够对网络访问进行严格控制。IAM 角色集成则让使用者能够轻松实现泛用性访问控制机制。最重要的是，EMRFS——这是一套 HDFS 兼容性文件系统，能够访问 S3 当中的存储对象——允许大家利用访问控制　顺 S3 当中的数据存储层。

我们能够利用多套 EMR 集群在同样的存储系统基础之上运行多个数据访问层，而且这些访问层能够实现彼此隔离且不需要占用计算

资源。如果某项特定任务需要在少数情况下使用每个节点中的大量内存资源，我们可以直接为其运行一套专用的内存优化型节点集群，而不必再像过去那样纠结于集群内 CPU 与内存的具体搭配比例。我们能够在无需针对特定内部客户修改“主”生产集群的前提下运行实验性集群。成本分配同时易于实现，因为我们能够为 EMR 集群添加标签、从而进行成本追踪或者根据需要将其运行在各个独立 AWS 账户当中。

由于它能够极大降低集群体系的管理难度，现在我们得以建立新的实验性数据访问层并运行大规模 POC，同时无需为了运行相关 Hadoop 集群所带来的新硬件采购支出或者将大量时间用于摆弄各种类型的移动部件。这与我们当初启用 Amazon Redshift 时的状况非常相似：在 Amazon Redshift 出现之前，我们甚至根本无法想象自己能够对容量超过 100TB 的生产数据库进行复制以完成实验性目标。如果使用传统处理方式，整个过程可能需要耗时数个月，其中包括订购硬件、设置以及进行数据迁移等等——现在，我们只需要一个周末就能搞定全部工作，并在实验完成后直接将其“回炉再造”。总而言之，Amazon Redshift 为我们开启了一扇新的可能性大门，而在此前这一切完全不可能实现。

随着我们全新数据仓库的推出，我们的目标旨在建立起一套极具实用性且能够为一系列类型广泛、技能储备有异的用户提供访问能力的平台方案。其主要接口通过 SQL 实现；我们曾经对 Spark SQL、

Presto 以及 Drill 等进行过评估，但我们同时也认真考量过其它候选方案以及非 SQL 机制。Hadoop 社区的发展速度极快，因此根本不存在一款能够通吃一切的万试万灵型查询工具。我们的目标是为客户希望使用的一切 Hadoop 生态系统分析及数据访问应用程序提供必要支持。EMR 与 EMRFS 与 S3 相对接让上述目标成为了可能。

安全要求与 Amazon S3 客户端加密机制

纳斯达克集团拥有一套业务涵盖范围广且士气高昂的内部信息安全团队，其在数据及应用程序保护方面遵循着严格的管理政策及标准。考虑到数据的敏感性水平，这部分内容必须在存储及传输过程中得到确切加密。除此之外，加密密钥必须被保存在位于纳斯达克基础设施内部的 HSM（即硬件安全模块）集群当中。我们为 HSM 选择了与 AWS CloudHSM 服务相同的设备品牌与机型（即 SafeNet Luna SA），这也使得我们能够将 Amazon Redshift 集群主密钥保存在自己的 HSM 当中。为了简单起见，我们将其称为纳斯达克 KMS，其功能类似于原有 AWS 密钥管理服务（简称 AWS KMS）。

最近，EMR 在 EMRFS 中推出了一项新功能，允许客户利用自有密钥实现 S3 客户端加密，其利用 S3 加密客户端的封闭加密机制。EMRFS 允许大家通过实现由 AWS SDK 提供的 EncryptionMaterialsProvider 接口编写自己的瘦适配机制，这样当 EMRFS 在 S3 内读取或者写入某个对象时，我们就将获得一项回调

以为其提供加密密钥。这是一种简洁的低级钩子，允许大家在无需添加任何加密逻辑的前提下运行大多数应用程序。

由于 EMRFS 是一种 HDFS 接口实现机制（当大家在 EMR 当中使用‘s3://’时即可进行调用），因此前面提到的各层无需在加密过程中进行识别。EMRFS 在堆栈中还属于低级机制，因此其运作方式可谓包罗万象。换言之，它几乎能够应对一切集成加密任务。需要强调的是，seek() 函数同样适用于保存在 S3 当中的加密文件，因此其对于多数 Hadoop 生态系统内的文件格式而言都是一种极为重要的性能保障功能。

使用定制化加密素材提供程序

在素材提供程序方面，我们的实现方案能够与纳斯达克 KMS 进行通信（具体情况请参阅下文中的‘代码示例’章节）。EMR 能够从 S3 当中提取出自己的 .jar 实现文件，将其放置在集群中的每个节点当中，并通过 EMRFS 配置在集群上的 emrfs-site.xml 配置文件当中使用我们的素材提供程序。我们能够在创建新集群时指定下列参数，从而轻松通过 AWS 命令行实现上述目标：

```
    --emrfs Encryption=ClientSide,ProviderType=Custom,
CustomProviderLocation=
    s3://mybucket/myfolder/myprovider.jar,CustomProvid
erClass=providerclassname
```

利用上述代码，一个 s3get 与 configure-hadoop 引导操作将以自动化方式实现添加并分别被配置为复制提供程序 jar 文件及配置 emrfs-site.sml。其中 s3get 引导操作在获取到来自 S3 的请求后将使用被分配至该集群的 Instance Profile，这样一来大家就能够对自己的提供程序 jar 文件进行访问限制。另外，如果大家的提供程序类通过来自 Hadoop API 的 Configurable 实现，那么也能够从运行时中的 Hadoop 配置 XML 文件处获取配置数值。

S3 对象上的“x-amz.matdesc”用户元数据字段被用于存储“素材描述”，这样我们的提供程序就能了解到需要接收哪些密钥。该字段中包含一套 JSON 版本的 Map<String.String>，其会在针对现有目标进行密钥请求时被交付至提供程序实现处。当某密钥被请求编写一个新的对象时，同样的映射将被交付至 EncryptionMaterials 对象，后者当中也拥有一条 Map<String.String> 作为描述。我们会经常对加密密钥进行轮换，所以我们会利用该映射为 S3 对象所使用的加密密钥保存惟一一条标识符。密钥标识符本身不会被作为敏感信息处理，因此我们可以将其存储在 S3 当中的对象元数据之内。

数据输入任务流

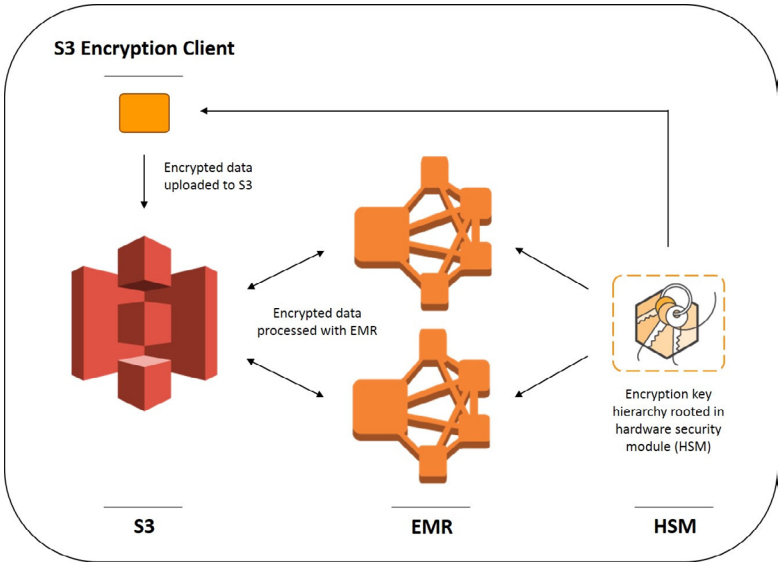
纳斯达克已经拥有一套成熟的数据采集系统，其开发目标在于支持我们对 Amazon Redshift 的采用与推广。这套系统建立在一套任务流引擎之上，该引擎每天能够执行约 30000 次编排操作，同时使用

MySQL 作为状态信息的持久性存储机制。

大部分此类操作需要执行多个步骤，且遵循以下几种常见模式：

- 通过 JDBC、SMB、FTP 以及 SFTP 等对其它系统进行数据检索。
- 验证数据语法及正确性，以确保既定模式未出现变更或者数据未出现丢失等等。
- 将数据转化为 Parquet 或者 ORC 文件。
- 利用 S3 客户端加密机制将文件上传至 S3 当中。

我们的数据采集系统本身并不具备直接针对 HDFS 的能力。我们利用类似的 EncryptionMaterialsProvider 实现机制将数据直接写入至 S3 当中，而无需借助 Hadoop Configuration 接口钩子。这一点非常重要，即避免在对 EMRFS 进行数据读取或者写入时采用任何特殊方式。而且只要大家的素材提供程序能够检测到需要使用的正确加密密钥，EMR 就能以无缝化方式在 S3 当中操作加密对象。



代码示例

我们的加密素材提供程序可参照以下代码示例：

```
    import java.util.Map;
    import javax.crypto.spec.SecretKeySpec;
    import com.amazonaws.services.s3.model.
EncryptionMaterials;
    import com.amazonaws.services.s3.model.
EncryptionMaterialsProvider;
    import org.apache.hadoop.conf.Configurable;
    import org.apache.hadoop.conf.Configuration;
    public class NasdaqKMSEncryptionMaterialsProvider
    implements EncryptionMaterialsProvider, Configurable
```

```
    {
    private static final String TOKEN_PROPERTY = "token";
      private Configuration config = null; // Hadoop 配置
      private NasdaqKMSClient kms = null;  // 面向 KMS 的
客户端
      @Override
    public void setConf(Configuration config) {
    this.config = config;
        // 读取任意需要初始化的配置值
    this.kms = new NasdaqKMSClient(config);
      }
      @Override
    public Configuration getConf() { return this.
config; }
      @Override
    public void refresh() { /* nothing to do here */ }
      @Override
    public EncryptionMaterials getEncryptionMaterials(Map
desc) {
        // 从素材描述中获取密钥标识
        String token = desc.get(TOKEN_PROPERTY);
        // 从 KMS 中检索加密密钥
    byte[] key = kms.retrieveKey(token);
        // 利用该密钥创建一个新的加密素材对象
       SecretKeySpec secretKey = new SecretKeySpec(key,
"AES");
        EncryptionMaterials materials = new EncryptionMaterials
(secretKey);
        // 利用密钥标识（identifier）对相关素材进行标记
    materials.addDescription(TOKEN_PROPERTY, token);
    return materials;
```

```
    }
    @Override
  public EncryptionMaterials getEncryptionMaterials() {
      // 生成一个新的密钥，利用其进行密钥分配，
      // 同时利用该标识返回新的加密素材
      // 并将其存储在素材描述当中。
  return kms.generateNewEncryptionMaterials();
    }
  }
```

总结

在本篇文章中，我们对新的数据仓库项目进行了宏观概述。由于我们现在能够将 Amazon S3 客户端加密机制与 EMRFS 加以配合，因此我们得以在 Amazon S3 当中针对闲置数据实现安全保护要求，并充分享受由 Amazon EMR 所带来的可扩展能力以及应用程序生态系统。

原文链接：http://www.infoq.com/cn/articles/nasdaq-using-aws-for-ad-hoc-access-to-a-mass

英文原文链接：http://blogs.aws.amazon.com/bigdata/post/TxXTJA064IEJ8E/Nasdaq-s-Architecture-using-Amazon-EMR-and-Amazon-S3-for-Ad-Hoc-Access-to-a-Mass?adbsc=social_blogs_20150410_43607246&adbid=UPDATE-c2382910-5992423737142562816&adbpl=li&adbpr=2382910