

智能时代的 新技术实战

AWS 迷你书

NEW
TECHNOLOGIES IN
THE SMART AGE



目录

卷首语.....	1
Werner Vogels 脑中的未来世界@AWS re:Invent 2017.....	2
为什么你应该关注 Amazon SageMaker.....	11
AWS 正式把 KVM 投入使用 你需要知道些什么?	16
深入 Serverless—让 Lambda 和 API Gateway 支持二进制数据.....	20
让你的数据库流动起来 – 利用 MySQL Binlog 实现流式实时分析架构.....	36
AWS Organizations 提供基于策略的集中化帐户管理能力.....	43
Netflix 打造容器管理基础架构: Titus.....	45

卷首语

早在过去的 5-6 年间，AWS 业务及产品的拓展就已经对机器学习产生了大量的需求，而这些需求主要受三大因素影响：算法、大型数据集以及效能计算。算法层面，它们相比 20 年前已变得越来越复杂，并结合了人工智能的算法；数据集层面，由于云计算的出现，使获得、应用丰富而海量的数据成为可能；效能计算层面，GPU 可以创造出高度复杂和规模化的计算能力，从而把算法与海量数据相结合。

所有的这些需求，都离不开以技术为基础的演进和优化。此外，为了给用户提供一个更加广阔和深度的服务平台，AWS 团队陆续开发和提供了越来越多的服务和功能。例如，作为 Serverless 架构主要的服务，AWS Lambda 和 AWS API Gateway 已开始支持二进制数据，让用户享受其带来的便利；AWS Organizations 也正实现对运维复杂度的降低，并能通过细化的访问权限应用服务控制策略。因而，很多开发者会迫切的想要知晓，在 Lambda、EC2、AWS Organizations、Amazon Alexa、Amazon Lex 这些技术架构、平台、软件或服务体系下，深藏着哪些创新和突破？AWS 又是如何设计这一系列的解决方案的呢？

在本期《AWS 迷你书：智能时代的新技术实战》中，我们就将揭开人工智能全面来临的时代下，AWS 在各领域的技术创新和实践经验，及其在 2017 re:Invent 大会上发布的一系列「超燃杀器」，在 Serverless、实时数据系统构建等热门技术上演进和突破等，让各位读者能更深入的了解 AWS 在技术路上的探索精神和开放精神。

诚如 Amazon CTO Werner Vogels 所说，“在 AWS 平台上，我们是不设守门人（gatekeeper），因此我们不会告诉我们的合作伙伴他们在 AWS 平台上什么可以做、什么不可以做。‘没有守门人’这一点能够激发更多、更好地创新。”

Werner Vogels 脑中的未来世界@AWS re:Invent 2017

by 杨赛

2017 年 11 月 30 日，拉斯维加斯。这是 Werner Vogels 第六次站在 AWS re:Invent 主题演讲的舞台上。作为 AWS 的 CTO，多年来他发起、参与并见证了无数的变化，但是有一件事情一直没有变过——

他一直在想象一件事：未来的开发会发生怎样的变化？

“虽然我无法预测五年后的世界会变成什么样子，正如同五年之前的我无法预测今天一样。但是有一点我们非常清楚：靠我们自己是无法把 AWS 建设成今天这样的。不是我们去教育开发者们应该如何开发软件，而是他们教会我们应该去开发怎样的软件。”

“举个例子，当年我们之所以开发 DynamoDB，是因为我们听到了开发者对于另一种数据库的需求。然而当我们发布 DynamoDB 的时候，我们并不知道开发者们会需要 IAM 层面的访问控制功能。是开发者们教会我们 DynamoDB 的服务应该要怎样去做。”

“我们的发展路线图掌握在你们手中。”

以后的 App 开发会是怎样的？

想象的能力建立在观察的积累之上，观察的积累建立在用心的基础上。技术提供方与技术需求方之间的交流，本质上是人与人之间的交流。当客户有意或无意间说出一句话的时候，身为希望去帮助客户解决问题的这一个人，他听到了什么？他听见了什么？

“一位 GE 客户曾经说过一句话：我们晚上睡觉的时候还是一家制造公司，结果一觉睡醒，变成一家数据分析公司了。”

Vogels 举了这个例子之后，说了两件事：

1、机器学习领域过去两三年的发展，最大的意义在于两个字：“实时”。实时在线数据和离线历史数据完全是两码事。TensorFlow 和 MXNet 这样的神经网络框架，有它们和没有它们最大的区别就在于是以天为单位出结果，还是以秒为单位出结果。以天为单位出结果，就限制了只能对那些离线的历史数据进行学习。以秒为单位出结果，才使得我们有能力对现在正在发生的事情进行学习。

Real time 这个词组在 Vogels 大叔的这段话里至少出现了三次。我想他应该是希望我们听见什么。

2、大家为什么要花费这么大精力来到我的 Keynote 现场呢？为什么你的爷爷奶奶除了 Skype 之外不会用别的 App？为什么菲律宾的贫困农民不会用智能手机 App 来改进他们的生产？为什么同样的人机交互方式，我们就可以无障碍的使用，他们却不行？

有的观点会说，如果他们真的需要某个 App，他们肯定还是能学会的。之所以学不会，是因为他们还并没有那么想要这个 App 的功能。

Vogels 绝对不认同这个观点。

“他们怎么可能会不想要？”



图：迄今为止的人机交互方式

“我们之所以具备了人机交互的能力，是因为我们去适应了机器的交互方式。我们学会了使用键盘、鼠标和屏幕，是因为我们具备了特定的学习条件，但这些并不是人类的自然交互方式。大家来到我的 Keynote 现场，难道会是为了来看我背后这块屏幕的吗？肯定不能！你们是来听我说话的。”

“未来的软件一定要去适应人类天然的交互方式。人类天然的交互方式就是说话、倾听。这是你的爷爷奶奶和菲律宾的农民们都会用的交互方式。”



图：以人类为中心的交互方式

的确，Amazon 是已经发布了 Echo 设备，发布了 Polly、Lex、Transcribe 和 Comprehend，以及基于其所构建的 Alexa for Home、Alexa for Business 服务，但这些只不过是微不足道的小小几步。这个领域需要更多开发者们的关怀，需要开发者们持续不断的进行更多的观察和想象。

以上是 Vogels 对于机器技术栈的最上层——人机交互接口层面的未来思考。

“语音能力一定会是下一代系统的构建核心，这也是深度学习技术将发挥重大意义的第一个阵地。”

其实从技术背景来说，Vogels 的关注点主要是在底层技术栈，应用层的工作算不上 AWS 的主业。如果是强调语音交互能力的战略重要性，完全可以由 AWS CEO Andy Jassy 而不是 Vogels 来讲。如果只是产品发布的话，这次连 GuardDuty 这个量级的安全服务都没上主题演讲（GuardDuty 这次是安排了一次晚场活动做的发布，该服务利用机器学习技术实现了一些威胁自动报警的功能），Transcribe 和 Comprehend 在 Andy 的演讲里都只是在中间小小的露了一个脸，Alexa for Business 身为一个应用层服务，为什么能排在 Vogels 演讲的第一部分、而且还占了这么大篇幅呢？InfoQ 编辑认为，Vogels 来讲语音交互这个事情是想表达他的一个态度：

他觉得开发者们现在对语音交互领域的关注力度实在是太不够了。

所以，Vogels 要动用自己技术领导者的影响力来号召开发者们赶紧往这个方向多跑跑。

回到底层

讲完语音交互，Vogels 把话题收回到 IT 架构的底层领域。

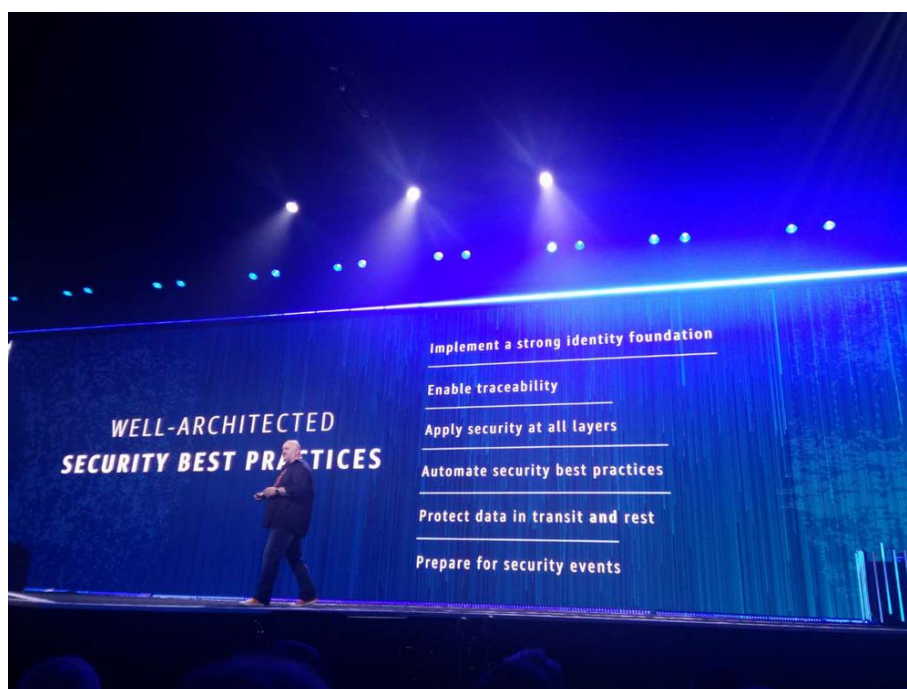
底层的几件事主要是在实践层面，即所谓的“最佳实践”推荐。“最佳实践”在 AWS re:Invent 期间有很多专门的课程，此类课程特别受到一线工程师们的欢迎。其实这些最佳实践，Vogels 想必已经说了不知道多少遍了，可是他还是继续说。也许他觉得听进去的人还不够多吧？

技术细节在主题演讲的有限时间也讲不了太深，InfoQ 编辑在这里给 Vogels 的分享做个简单的综述。

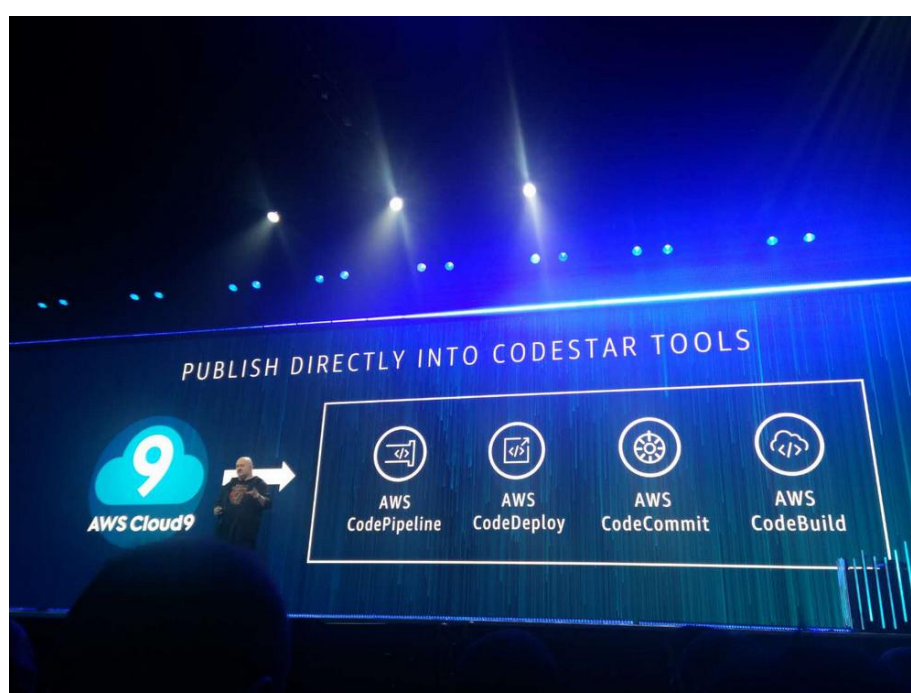
第一件事：别让你的架构被最初的计划限制死了。你还在预测“我两年后会需要多少容量”这件事吗？那已经是老黄历了，现在再这样做是要坑死自己的！请从一开始就以“可以伸展的方式”设计你的架构（*extensive architecture*）。



第二件事：安全必须在所有工作之前进行。安全的优先级高于任何一个特性开发！你的每一位开发者都应该是一个安全工程师。与此同时，如果你想确保你的数据只有你自己能访问，那么加密是唯一的保证（Vogels 在这里提到了本次 re:Invent 发布的 KMS 服务——*bring your own keys*）。当然，你一定会需要各种自动化工具来记录系统的一切变化、检测系统的任何异常——Macie 和 GuardDuty 就是做这事儿的。




第三件事：你需要更好的开发工具。也许你可以试试我们的 Cloud9 IDE？



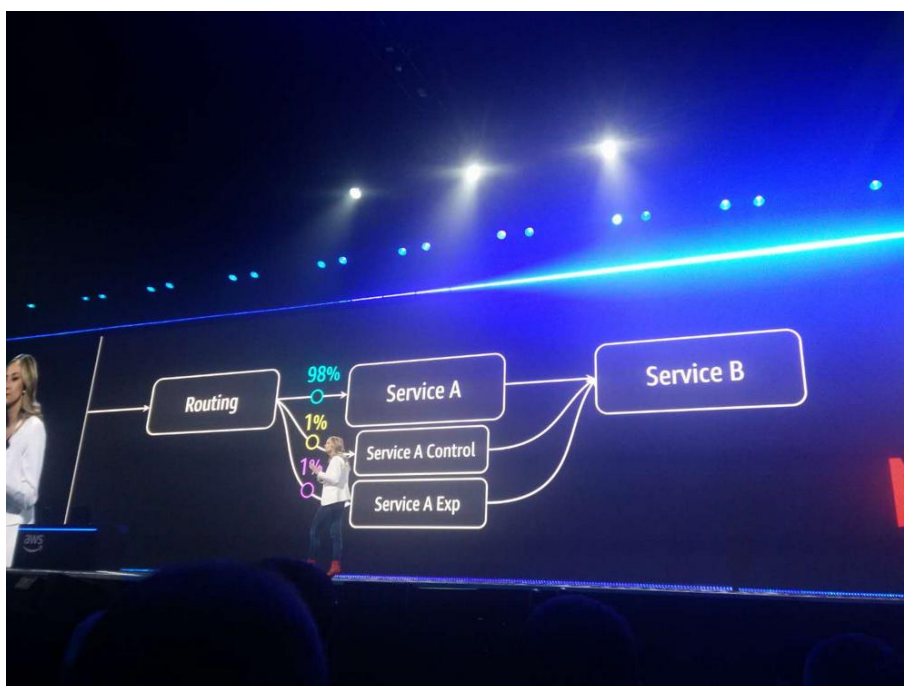
图：与 AWS 各服务深度集成的 Cloud9 IDE

第四件事：关于可用性的一些常识。如果你在一个可用性区域（AZ）上部署，其基础架构的理论可用性是 99%。如果你想要四个九，你可能会需要三个 AZ，这意味着三倍的成本。如果你想要五个九，你还需要跨区域部署（region），这意味着六倍的成本。所以在讨论可用性之前，请搞清楚你想要什么。比如，在 AWS 的工程师们就很清楚的知道 Route 53 服务是绝对不能挂的，所以这个服务的可用性做到了绝对的 100%，不骗人。而其他的服务，则根据各自的情况来进行各自的决策。



SERVICE	CONTROL PLANE	SINGLE AZ DP	MULTI AZ DP	SERVICE
EC2	99.95	99.95	99.99	
RDS	99.95	99.95	99.99	
ECS	99.95			99.99
EBS	99.95			99.999
Kinesis				99.99
DynamoDB				99.99
Cloudfront	99.9			99.99
Route 53	99.95			100
IAM	99.9			99.995
KMS	99.99			99.995

第五件事：关于测试的一些常识。你还在“测试环境”里做测试吗？别逗啦。这年头，生产环境才应该是你的测试场地。强烈推荐大家都学习一些“混乱工程学”（Chaos Engineering），并在自己的生产系统上赶紧用起来。这一段邀请了 Netflix 的美女工程师、《Chaos Engineering》的作者之一 Nora Jones 为大家上台做了一次科普。这本书的英文电子版可以在他们的网站 [Principles of Chaos](https://principlesofchaos.com/) 上查看。



图：Chaos Engineering 是怎样做的

第六件事：Gall's Law——可用的复杂系统在一开始都是可用的简单系统。如果你的起点是一个复杂系统，那么最可能得到的结果是一个不可用的复杂系统。你问我怎么才能让系统

不那么复杂？我的第一个建议是微服务——不仅仅是容器，同时还需要正确的微服务设计思路、以及好用的容器管理工具。欢迎早点来试试我们的 EKS（Kubernetes）和 Fargate！第二个建议是无服务。Vogels 在这里介绍了 Lambda 最近的一些新功能，包括 API Gateway VPC 集成、并发控制、3GB 的内存、以及 .NET core 2.0 语言的支持，同时还发布了 [AWS Serverless Application Repository](#)。



图：Vogels 举例说明无服务（Lambda）的一些实际用法

第七件事：SageMaker！

当 Andy Jassy 正式把 SageMaker 的消息对外发布的时候，Vogels 在 Twitter 上发了一条推：

“SageMaker 有多么重大的意义呢？那就是无论我描述它有多大的意义都不算过分。”

按照 AWS 的惯例，新产品正式对外发布之前都是已经有内部选定的客户秘密使用过一段时间的，对于 SageMaker 而言，DigitalGlobe 就是其中的一个客户。他们是做卫星地图数据的，数据采集了 17 年下来攒了有 100PB。因为这个数据量太大，所以他们首先是成了 AWS Snowmobile 的用户——对，你没想错，就是在 2016 年 re:Invent 上台的那辆卡车：



100PB 的图片在 AWS 上放着，成本实在不低，所以很自然的当作冷数据放进了 Glacier。但是这些图片数据还需要拿来做分析，怎么办？所以 Glacier Select 相当于也是 AWS 配合他们家的需求做的。分析、预测的需求场景那就比较多了，比如从他们自己的层面，需要提升图片缓存命中率；从他们客户的层面，比如某国运营商要部署 5G 网络，会问他们要全国树木的分布情况；某国森林大火要做紧急疏散，会问他们要疏散方案等等。现在，他们在 SageMaker 上每天分析的图片量有 80TB。



图：在某城市卫星地图上标记所有的树木

SageMaker 的意义还不止如此。如果你已经试用过 SageMaker（如果还没有，建议去注册一个 AWS 账号试试，这个服务现在可以免费试用），你大概已经知道 SageMaker 上的训练建模代码是写在 Jupyter Notebook 里面的。然后，Jupyter Notebook 是可以分享的！

DigitalGlobe 就把他们的一个 [GBDX Notebooks](#) 分享了出来，这套代码是用来从图片里抽取特征的。我想，他们会很希望看到有人能对这套代码进行一些改善。

总结

看完了 Vogels 的分享，你的收获如何？关于最佳实践部分的内容，除了本次 re:Invent 大会上的现场课程之外，有条件的同学也可以了解他们在各地的 [AWSome Day](#)、[培训课程](#)，或者是[白皮书](#)。

为什么你应该关注 Amazon SageMaker

by 杨赛

2014 年，AWS 在 re:Invent 上发布 Lambda 服务的时候，很多开发者尚未意识到这个服务能给他们带来怎样的好处。时隔三年，Lambda 服务已经成为了“Serverless”和“Function as a Service”开发范式的代表，并且已经承担起为数不少的企业关键业务。可以说，Lambda 服务为应用开发领域开辟了一片全新的游戏场地。

在 InfoQ 编辑看来，AWS CEO Andy Jassy 在今年的 re:Invent 大会上发布的 [Amazon SageMaker](#) 也是一个改写游戏方式的关键标志，而这一次的游戏场地是在人工智能领域（2017 年 AWS re:Invent 的产品发布列表[见之前的这篇报道](#)）。

在 2017 年的今天，阻碍人工智能技术被广泛应用的最主要的门槛是什么？

芯片、框架、算法、数据量？这些领域当然还有很大成长空间，但很难说是当前的最大门槛。有人说，最大的门槛是准确率还不够，尤其是 toC 领域的自然语言处理这样的场景，机器对人的表达方式的理解还远远达不到可以正常交流的水平。对于这个准确率的问题，每个体验过相关服务的读者应该都有自己的判断。但如果这个问题为真，那么下一个问题是：为什么在这些领域，准确率提高得还不够快？

用机器学习来解决一个问题是建立在一个假设之上：针对任何一个问题，我们是能够根据固定数量的变量，得到一个足够好的模型，从而对其进行有效的解释与预测的。所以，基于这样的一个假设，那么准确率低的问题，就是一个模型不够好的问题。

模型不够好有很多原因，可能是因为训练数据不够，也可能是因为算法不合适，或者算法的参数没选择到最合适的组合。训练数据不够，就要多采集数据、多存储数据、多清洗数据。算法不合适，可能是因为参数的调试还没做到位，可能因为参数的调试需要耗费太多时间和其他成本所以在一个局部最优解就停下了，也可能是因为使用的框架（引擎）不合适。至于哪个框架更合适，可能需要多试几个框架，乃至可能当前市面上流行的框架都不合适，需要一个新的框架。

对于以上各种可能的问题来源，Andy Jassy 提出了一个也许是更根本的问题：

也许，我们在建模这个层面的人手太少了？

根据 InfoQ 编辑所了解，目前市面上有一些业务做得很好的大数据公司，他们的技术团队/数据团队有 80% 的精力都用来做什么工作呢？

数据准备。

剩下的 20% 精力，又有很大部分是用在部署集群、安装框架、调优这些“杂活儿”上面了。

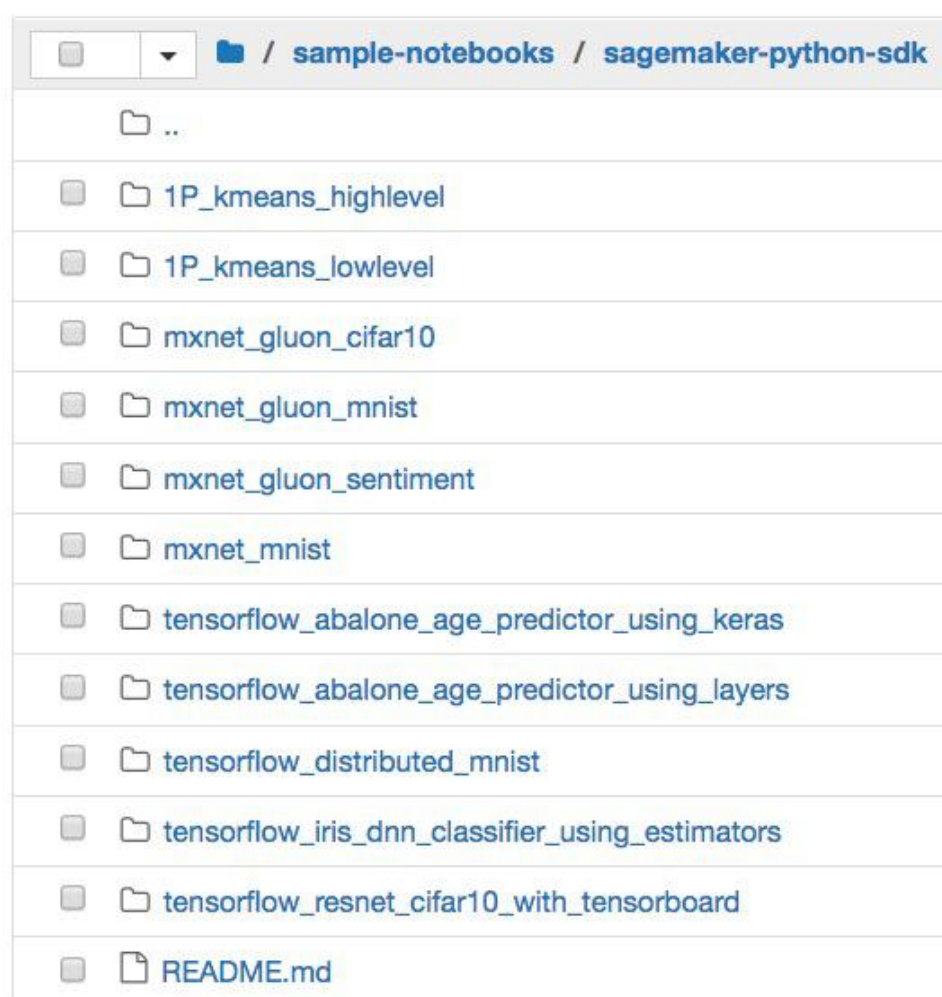
从当前的客观情况来看，这是把数据分析业务做好的一个基本功。然而从理想世界的角度来看，这样的现状是相当惊人的浪费。AI 领域大牛李飞飞，她有多少时间是用在改进框架和算法上，有多少时间是投入到标记那 320 万张图片上？

这就好比在云计算出现之前，一个开发者想要开发一个应用，可能 80%的时间是用在买服务器、安装操作系统、调试数据库、部署扩容备份等等跟开发应用无关的事情上了。现在的数据领域，跟那个年代的应用开发领域其实是非常相似的——一半以上的时间都不是用来产出的。

有些开发者可能还没动手就被这些麻烦事儿吓跑了。坚持在这个领域深耕的数据科学家们，他们的精力也用不到刀刃上。这就是 AWS 发布 SageMaker 的立场：让有能力去改进框架和算法的开发者，尽可能少花费精力在那些跟主业无关的事情上。

简单的看一下 SageMaker 的基本用法——如何训练一个模型。以下内容来自 [AWS 布道师 Randall Hunt 发布的博客](#)。

首先，在 AWS 的后台新建一个 Notebook 用来放置你的代码和一些配置信息。新建好的 Notebook 下面会自带一个叫做 sample-notebooks 的目录，里面包含了一些预置的算法：



无论是哪种算法，代码大致上是下面这个结构：

```
def train(
    channel_input_dirs, hyperparameters, output_data_dir,
    model_dir, num_gpus, hosts, current_host):
    #给训练代码占座
    pass

def save(model):
    #找个地方保存模型文件
    pass
```

首先，channel input dirs：训练数据从哪儿来？可以是本地的一个目录，也可以是云端的。自从 S3 支持对象级别的查询之后，S3 在这几年已经变成一个挺流行的 Data Lake 选项。现在有了新发布的 [S3 Select](#)，这个 Data Lake 的查询变得更高效了。而且现在还有了 [Glacier Select](#)，于是那些冷数据也可以直接被拿来当作训练数据（题外话：Andy Jassy 说这两个 Select 功能是研发了一年的时间做出来的。你怎么看？我真心觉得还挺快的。）

然后，hyperparameters：这一个步骤就是 SageMaker 的第一个魔法所在了。看看教程中是怎么定义这个 hyperparameter 的：

```
hyperparameters={
    'batch_size': 128,
    'epochs': 50,
    'learning_rate': 0.1,
    'momentum': 0.9
}
```

传统来说，这些 hyperparameter 的初始设置可能对最终的模型质量有很大的影响。比如这个 learning rate，如果设置太低，则学习得太慢；如果设置太高，可能压根跑不出来结果。于是乎，数据科学家们不得不一次又一次的试错，一直试错到他们觉得足够好，或者他们觉得足够烦了为止。

SageMaker 的第一个魔法就在于这一个步骤的自动化：把试错这件事交给机器来做，直到机器认为足够好了为止——机器可是从来都不会厌烦的。

其余的训练参数定义比较直观，就不多介绍了。要跑训练的时候，假设你要用 MXNet 这个框架配合一个你写好的 cifar10 算法，那么你就这么写（[点击这里查看 AWS 提供的 sample](#)

[notebook - cifar10.py](#) 文件的详细内容，以及 AWS Github 账号上的 [SageMaker Python SDK](#)。这个文件使用了 [Gluon API](#)。):

```
import sagemaker
from sagemaker.mxnet import MXNet
m = MXNet("cifar10.py", role=role,
          train_instance_count=4, train_instance_type="ml.p2.xlarge",
          hyperparameters={'batch_size': 128, 'epochs': 50,
                           'learning_rate': 0.1, 'momentum': 0.9})
```

现在，可以把训练数据丢给它了：

```
m.fit("s3://randall-likes-sagemaker/data/gluon-cifar10")
```

于是你就有了四个 P2 实例开始帮你跑训练了！当训练跑完的时候，你就有了一个模型 `m`，可以拿来预测了。至于这四个 P2 实例，它们会被自动释放，不用去管它们。这是 SageMaker 的第二个魔法：忘记你的服务器，也不用折腾各种框架的安装部署这些底层工作。SageMaker 上的这些框架都是已经优化好的，你自己部署的话，效果多半不会比它更好。

预测的执行也仅仅需要下面这一个指令，不需要在主机层面搞东搞西：

```
predictor = m.deploy(initial_instance_count=1, instance_type='ml.c4.xlarge')
```

比如，可以把想要做预测的在线实时数据——比如一张 Twitter 上的新照片——通过一个 Flask App 丢过去：

```
@app.route('/invocations', methods=["POST"])
def invoke():
    data = request.get_json(force=True)
    return jsonify(predict.download_and_predict(data['url']))
```

然后你就可以得到你的预测结果——比如说，这张照片的拍摄地点。

那么你要问了，SageMaker 对于数据准备这个最大的“杂活儿”有啥帮助呢？

SageMaker 有一个 `preprocess` 数据的功能。至于这个功能能做到什么程度，还得试试才知道，详细情况可以参考[这个文档](#)。我想，恐怕暂时还不能跟那些专门提供数据准备服务的质量相比，因为这个文档里面同时也推荐了 [Marketplace 上的相关服务](#)。

不过，能够节省下创建实例、安装框架、训练试错这些方面的工作，已经是一个很大的进步。现在，开发者只要有一个 **AWS** 账号就可以体验 **SageMaker**，而且现在是有两个月的免费额度的，正是尝鲜的最佳时机。**AWS** 的开发者文档网站已经更新了 **SageMaker** 的详细说明，可以[在此查看](#)。期待看到 **SageMaker** 之后的发展情况。

AWS 正式把 KVM 投入使用 你需要知道些什么？

by 杨赛

如之前 InfoQ 中文站所报道，[AWS 在 11 月 6 日推出新的实例类型 C5](#)，其中采用了新的虚拟化引擎——一款 AWS 自家定制的 KVM。这可能意味着 AWS 从 2006 年启动时就开始使用并持续优化至今的 Xen 技术栈，将逐渐淡出这一体量庞大的云计算平台。

关于 Xen 与 KVM

Xen 最早是剑桥大学的一个研究项目。该项目在 2003 年以开源协议发布后，先后经历了 XenSource 公司、Citrix 公司、Linux 基金会等组织的领导，其技术阵营包含了 Citrix、IBM、Intel、HP、Novel、红帽、Sun、Oracle、Amazon、AMD、Bromium、CA Technologies、Calxeda、Cisco、Google、三星、以及 Verizon 等业界巨头。

KVM (Kernel-based Virtual Machine，直译为“基于内核的虚拟机”)，最早是以色列初创企业 Qumranet 发布的开源项目。该项目在 2007 年被合并入 Linux 内核代码——这对 KVM 而言是非常重要的一个节点，该公司则在 2008 年被红帽收购。KVM 技术差不多到 2010 年之后进入成熟阶段，该技术阵营目前包括红帽、SUSE、Linaro (ARM)、IBM、Intel、Google、Oracle 等业界巨头，国内的华为、阿里巴巴、腾讯等也均有参与。

在 AWS 启动的 2006 年，Xen 还是当时最成熟的虚拟化引擎技术（对于 Linux 操作系统而言），而 KVM 项目还没有出现在大家的视野中。因此，AWS 在早期技术选型当中采用了 Xen，成为其弹性计算的底层基础。（此外，2009 年启动的阿里云也因为当时 KVM 还不成熟的原因采用了 Xen，不过两家一直未停止过对 KVM 的关注与投入，阿里云更是在数年前就已经推出了基于 KVM 的主机。另外，2011 年启动的 Google Cloud 从一开始就采用了 KVM 引擎。）

AWS 的用户需要知道什么？

首先，AWS 的用户现在已经可以将自己跑在 C4 实例上的主机切换到 C5，前提是：

- 现在已经推出 C5 的区域有 US East (Northern Virginia)、US West (Oregon)、EU (Ireland) Regions，目前要用 C5 的话只能在这三个区，其他区还得等等。
- AMI 镜像的操作系统必须包含 ENA 和 NVMe 的驱动，因为 C5 的网络和存储功能是在硬件层面实施的。AWS 平台上提供的最新版 AMI 镜像，包括 Amazon Linux、Microsoft Windows (Windows Server 2012 R2 以及 Windows Server 2016)、Ubuntu、RHEL、CentOS、

SLES、Debian、FreeBSD，现在都支持。同理，如果你制作自己的 AMI，也是一样的要求。

[Reddit 网站](#)上已经有用户在分享自己的切换过程，感兴趣的读者可以前往查看或提问。Jeff Barr 也会在那里回复一些问题。

C5 实例可以选择的实例尺寸见下表：

Instance Name	vCPUs	RAM	EBS Bandwidth	Network Bandwidth
c5.large	2	4 GiB	Up to 2.25 Gbps	Up to 10 Gbps
c5.xlarge	4	8 GiB	Up to 2.25 Gbps	Up to 10 Gbps
c5.2xlarge	8	16 GiB	Up to 2.25 Gbps	Up to 10 Gbps
c5.4xlarge	16	32 GiB	2.25 Gbps	Up to 10 Gbps
c5.9xlarge	36	72 GiB	4.5 Gbps	10 Gbps
c5.18xlarge	72	144 GiB	9 Gbps	25 Gbps

其次，如果你已经用上了 C5，想用它跑一些机器学习的推断（inferencing）任务或者类似的计算任务，可以看一下这个 [Intel Math Kernel Library](#)。C5 的处理器用的是 [3.0 GHz Intel Xeon Platinum 8000 系列](#)，这款专为 EC2 做过优化的处理器配合上面那个数学库可能会有很好的性能。

此外，C5 还增加了每个 vCPU 的内存。对于兼容 AVX-512 指令集的代码而言，矢量操作和浮点操作的性能甚至可以翻倍。

切换到 C5 之后，由于运行在新虚拟化引擎上的实例是通过 NVMe 接口从 EBS 卷上启动的，而运行在 Xen 上的实例是从一个模拟 IDE 硬盘上启动、再切换到 Xen 的半虚拟块存储驱动上的，所以虽然操作系统（OS）能够识别自己正运行在哪个虚拟化引擎上，但如果软件本身假设底层的虚拟化引擎是 Xen 并依赖于该假设，则可能会引发一些问题。

但总体来说，AWS 表示只要 OS 层面能够支持 ENA 网络和 NVMe 存储，则大部分软件都能正常工作。AWS 还表示，其他的 EC2 功能并不会受到影响。

如果你是 AWS EC2 API 的重度用户，担心这次虚拟化引擎的变更会对 API 造成什么变化，则不用担心了：[AWS 在其 FAQs 页面中表示](#)其对外公开的 API 完全不会因为引入新的虚拟化引擎而有任何改变。

KVM 在 EC2 上的正式启用对 AWS 意味着什么？

作为计算资源服务的提供方，提升性能、降低成本是永恒的话题。

Xen 最初设计时，x86 架构尚未引入虚拟化扩展功能，所以 Xen 为了实现 Linux 系统的虚拟化，就把 Linux 内核给改了——这就相当于在之后的十几年里，Xen 一直维护着一套自己的 Linux 内核版本，所以上游 Linux 内核社区的很多新的好东西，它要费一番功夫才能移植进来，这就造成了很大的维护成本。

而另一方面，因为 KVM 项目是合并到 Linux 内核代码中的，维护起来就非常容易。Linux 内核上游社区的研发势能是非常大的，在这种情况下，KVM 的发展速度迅猛，在稳定性、性能方面的提升很快赶超了 Xen，受到很多技术人与企业的青睐。

EC2 是 AWS 的基石，虚拟化引擎又是 EC2 的基石。由于 AWS 是一套构建多年的、庞大而复杂的系统，很多功能会对 Xen 有所依赖，要让这套系统同时稳定的支持 Xen 与 KVM，是一项非常复杂的工作。所以 C5 的推出，意味着 AWS 这套系统已经脱离了对 Xen 的完全依赖。

对 AWS 而言，基于 KVM 的系统要比基于 Xen 的系统的维护成本更低，这是一方面。

另一方面，可能也与性能有关。按 AWS 首席布道师 Jeff Barr [在博客中所说](#)，C5 在性价比方面相比 C4 提升了 25%，针对有些任务甚至可以达到 50%——这是针对用户而言。要知道对于 AWS 今天这样的体量，哪怕是 1% 的节省都是巨大的；如果有 25% 这样比例的性价比提升，则绝对是势在必行，无论花费多大代价也要上。当然这其中的性能提升有多少是来自新的硬件，有多少是来自 KVM，这就不一定了。

此外，AWS 首先在“计算密集型实例”（compute-intensive）上正式采用 KVM，而不是从通用型、内存密集型等其他类型上开始，可能是因为考虑到计算密集型业务的 I/O 操作较少，比较独立，耦合性比较小，因此更容易替换的原因。

这次改变对于业界有什么影响？

从市场占有率的角度来看，就是 Xen 的最后一个大体量的用户对 Xen 说，以后我不再需要你了。

如果说以前还有业界传言说 AWS 在 Xen 上投入太过巨大、依赖太重，所以 AWS 将一直支持 Xen 的话，那么现在这个传言已经宣告终结。毕竟，如果连 AWS 都以实际行动表示切换到 KVM 的收益大于成本了，那么那些体量更小的 Xen 用户还能对 Xen 有什么更多的期待呢？

所以对于未来的市场而言，这次改变也许象征着 Xen 的使命已经结束了。对于过去的市场而言，Xen 的使命就是让那些仍然跑在 Xen 上的业务们能够继续平稳运行。不过这里面还有一个变数就是 Unikernel，那就是另一个话题了。

从上游社区的角度，影响可能不大，因为 Amazon 一直不是个活跃的开源社区参与者。即使作为最大的 Xen 用户那么多年，Amazon 对 Xen 上游社区的代码贡献数量却是屈指可数。所以对于 KVM 项目，可能我们未来也不大会看到很多来自 Amazon 的代码贡献。

从技术发展的角度，这是个大趋势下技术更新换代的必然——也许从 KVM 被合并到 Linux 内核代码的那时起就已经种下了种子吧。

深入 Serverless—让 Lambda 和 API Gateway 支持二进制数据

by AWS Team

1. 概述

Serverless 即无服务器架构正在迅速兴起，AWS Lambda 和 AWS API Gateway 作为 Serverless 架构主要的服务，正受到广泛关注，也有越来越多用户使用它们，享受其带来的便利。传统上来说，Lambda 和 API Gateway 主要用以实现 RESTful 接口，其响应输出结果是 JSON 数据，而实际业务场景还有需要输出二进制数据流的情况，比如输出图片内容。本文以触发式图片处理服务为例，深入挖掘 Lambda 和 API Gateway 的最新功能，让它们支持二进制数据，展示无服务器架构更全面的服务能力。

先看一个经典架构的案例——响应式主动图片处理服务。

Lambda 配合 S3 文件上传事件触发在后台进行图片处理，比如生成缩略图，然后再上传到 S3，这是 Lambda 用于事件触发的一个经典场景。

<http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html>

在实际生产环境中这套架构还有一些局限，比如：

- 后台运行的图片处理可能无法保证及时完成，用户上传完原图后需要立即查看缩略图时还没有生成。
- 很多图片都是刚上传后使用频繁，一段时间以后就使用很少了，但是缩略图还不能删，因为也可能有少量使用，比如查看历史订单时。
- 客户端设备类型繁多，一次性生成所有尺寸的缩略图，会消耗较多 Lambda 运算时间和 S3 存储。
- 如果增加了新的尺寸类型，旧图片要再生成新的缩略图就比较麻烦了。

我们使用用户触发的架构来实现实时图片处理服务，即当用户请求某个缩略图时实时生成该尺寸的缩略图，然后通过 CloudFront 缓存在 CDN 上。这其实还是事件触发执行 Lambda，只是由文件上传的事件主动触发，变成了用户访问的被动触发。但是只有原图存储在 S3，任何尺寸的缩图都不生成文件不存储到 S3。要实现此架构方案，核心技术点就是让 Lambda 和 API Gateway 可以响应输出二进制的图片数据流。

总体架构图如下：



主要技术点：

- 涉及服务都是 AWS 完全托管的，自动扩容，无需运维，尤其是 Lambda，按运算时间付费，省去 EC2 部署的繁琐。
- 原图存在 S3 上，只开放给 Lambda 的读取权限，禁止其它人访问原图，保护原图数据安全。
- Lambda 实时生成缩略图，尽管 Lambda 目前还不支持直接输出二进制数据，我们可以设置让它输出 base64 编码后的文本，并且不再使用 JSON 结构。配合 API Gateway 可以把 base64 编码后的文本再转换回二进制数据，最终就可以实现输出二进制数据流了。
- 用 API Gateway 实现 图片访问的 URL。我们常见的 API Gateway 用来做 RESTful 的 API 接口，接口的 URL 形式通常是 /resource?parameter=value，其实还可以配置成不用 GET 参数，而把 URL 中的路径部分作参数映射成后端的参数。
- 回源 API Gateway，缓存时间可以用户自定义，建议为 24 小时。直接支持 HTTPS，支持享用 AWS 全球边缘节点。
- CloudFront 上还可使用 Route 53 配置域名，支持用户自己的域名。

相比前述的主动生成，被动触发生成有以下便利或优势：

- 缩略图都不存储在 S3 上，节省存储空间和成本。
- 方便给旧图增加新尺寸的缩略图。

2. 部署与配置

本例中使用的 Region 是 Oregon(us-west-2)，有关文件从以下链接下载：

<https://s3.amazonaws.com/snowpeak-share/lambda/awslogo.png>

2.1 使用 IAM 设置权限

打开控制台：

<https://console.aws.amazon.com/iam/home?region=us-west-2>

创建一个 Policy，名叫 CloudWatchLogsWrite，用于确保 Lambda 运行的日志可以写到 CloudWatch Logs。内容是


```
{
  "Version":
    "2012-10-17",
  "Statement": [
    {
      "Effect":
        "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Resource": [
        "arn:aws:logs:*:*:*"
      ]
    }
  ]
}
```

创建一个 Role，名叫 LambdaReadS3，用于 Lambda 访问 S3。Attach Policy 选 AmazonS3ReadOnlyAccess 和刚刚创建的 CloudWatchLogsWrite。

记下它的 ARN，比如 arn:aws:iam::111122223333:role/LambdaReadS3

2.2 使用 S3 配置原图存储

打开控制台

<https://console.aws.amazon.com/s3/home?region=us-west-2>

创建 Bucket，Bucket Name 需要填写全局唯一的，比如 img201703，Region 选 US Standard。通常图片的原图禁止直接访问，这里我们设置权限，仅允许 Lambda 访问。

Permissions 下点 Add bucket policy，使用 AWS Policy Generator：

Select Type of Policy 选 S3 bucket policy，

Principal 填写前述创建的 LambdaReadS3 的 ARN

`arn:aws:iam::111122223333:role/LambdaReadS3`，

Actions 下拉选中 GetObject,

Amazon Resource Name (ARN) 填写刚刚创建的 bucket 的 ARN, 比如

```
arn:aws:s3:::image201703/*
```

然后点 Add Statement, 最后再点 Generate Policy, 生成类似

```
{
  "Id": "Policy1411122223333",
  "Version":
    "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1411122223333",
      "Action": [
        "s3:GetObject"
      ],
      "Effect":
        "Allow",
      "Resource":
        "arn:aws:s3:::img201703/*",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111122223333:role/LambdaReadS3"
        ]
      }
    }
  ]
}
```

复制粘贴到 Bucket Policy Editor 里 save 即可。

验证 S3 bucket 配置效果。把前下载图片文件 awslogo.png 下载到自己电脑, 然后把它上传到这个 bucket 里, 测试一下。直接访问链接不能下载, 需要右键菜单点 “Download” 才能下载, 说明权限配置已经成功。

2.3 创建 Lambda 函数

AWS Lambda 管理控制台：

[https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2#/](https://us-west-2.console.aws.amazon.com/lambda/home?region=us-west-2/)

点击 Create a Lambda function 按钮

Select runtime 菜单选 Node.js 4.3，然后点 Blank Function。

Configure triggers 页，点 Next。

Configure function 页，在 Name 栏输入 ImageMagick，Description 栏输入

Uses ImageMagick to perform simple image processing operations, such as resizing.

Lambda function code 里填写以下代码：

```
'use strict';

var AWS = require('aws-sdk');
const im = require('imagemagick');
const fs = require('fs');

const defaultFilePath = 'awslogo_w300.png';
// 样式名配置，把宽高尺寸组合定义成样式名。
const config = {
  'w500': {'width': 500,
  'height': 300},
  'w300': {'width': 300,
  'height': 150},
  'w50': {'width': 50,
  'height': 40}
};
// 默认样式名
const defaultStyle = 'w50';
// 完成处理后把临时文件删除的方法。
const postProcessResource = (resource, fn) => {

let ret = null;
```

```
if (resource) {
  if (fn) {
    ret = fn(resource);
  }
  try {

    fs.unlinkSync(resource);
  } catch (err) {
    // Ignore
  }
}
return ret;
};

// 生成缩略图的主方法
const resize = (filePathResize, style, data, callback) => {
  // Lambda 本地写文件，必须是 /tmp/ 下
  var filePathResize =
    '/tmp/' + filePathResize;
  // 直接用 Buffer 操作图片转换，源文件不写到本地磁盘，但是转换成的文件要写盘，所以最后
  // 再用 postProcessResource 把临时文件删了。
  var resizeReq = {
    srcData: data.Body,
    dstPath: filePathResize,
    width: style.width,
    height: style.height
  };
  try {
    im.resize(resizeReq,
      (err) => {
        if (err) {
          throw err;
        } else {

          console.log('Resize ok: ' + filePathResize);
          // 注意这里不使用 JSON 结构，直接输出图片内容数据，并且要进行 base64 转码。
          callback(null,
            postProcessResource(filePathResize, (file) => new
```

```
Buffer(fs.readFileSync(file)).toString('base64')));
}
});
} catch (err) {
  console.log('Resize
operation failed:', err);
  callback(err);
}
};

exports.handler = (event, context, callback) => {
  var s3 = new AWS.S3();
  //改成刚刚创建的 bucket 名字, 如 img201703
  var bucketName = 'image201702';
  // 从文件 URI 中截取出 S3 上的 key 和尺寸信息。
  // 稳妥起见, 尺寸信息应该规定成样式名字, 而不是直接把宽高参数化, 因为后者会被人滥用。
  // 使用样式还有个好处, 样式名字如果写错, 可以有个默认的风格。
  var filepath = (undefined ===
event.filePath ? defaultFilePath:event.filePath);
  var tmp =
filepath.split('.');
  var fileExt = tmp[1];
  tmp = tmp[0].split('_');
  var fileName = tmp[0];
  var style = tmp.pop();
  console.log(style);
  var validStyle = false;
  for (var i in config)
  {
    if (style == i)
    {
      validStyle = true;
      break;
    }
  }
}
```



```
style = validStyle ? style :
defaultStyle;
console.log(style);
var fileKey =
fileName+'.'+fileExt;
var params = {Bucket:
bucketName, Key: fileKey};

// 从 S3 下载文件，成功后再回调缩图
s3.getObject(params,
function(err, data) {
if (err)
{
console.log(err,
err.stack);
}
else
{
resize(filepath,
config[style], data, callback);
}
});
};
```

注意一定要把

```
varbucketName = 'image201702';
```

改成刚刚创建的 **bucket** 名字，如

```
varbucketName = 'img201703';
```

这个 **Lambda** 函数就可以运行了。

Lambda function handler and role 部分的 **Role** 选择 **Choose an existing role**，然后 **Existing role** 选择之前创建的 **LambdaReadS3**。

Advanced settings: Memory (MB)* 选 **512**，**Timeout** 选 **30 sec**。

其它保持默认，点 **Next**；最后一页确认一下，点 **Create Function**。

提示创建成功。

点击 **Test** 按钮，测试一下。第一次测试时，会弹出测试使用的参数值，这些参数其实我们都不用，也不用管它，点击 **Save and test** 按钮测试即可。以后再测试就不会弹出了。

显示“Execution result: succeeded”表示测试成功了，右边的 **Logs** 链接可以点击，前往 CloudWatch Logs，查看详细日志。右下方的 **Log output** 是当前测试执行的输出。

Execution result: succeeded (logs)

The area below shows the result returned by your function execution. [Learn more](#) about returning results from your function.

```
"iVBORw0KGgoAAAANSUhEUgAAADIAAAAVCAyAAAE1r0/AAAF/01EQVRYw81XaVMiSRD1f27sfN6ftLHjeIyoqNxyCgKC3ChyCQICzSWXCF6cztvMQ1F"
```

Summary		Log output	
Code	KpBI3zksDJOG7nAvtBA4vCAMi5X0jmaGpXne36E1es3	The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.	
SHA-256			
Request ID	b2539628-3790-11e7-83a1-c323c328542a		
Duration	567.14 ms		
Billed duration	600 ms		
Resources configured	512 MB		
Max memory used	37 MB		

```
START RequestId: b2539628-3790-11e7-83a1-c323c328542a Version: $LATEST
2017-05-13T04:00:26.976Z b2539628-3790-11e7-83a1-c323c328542a fai
2017-05-13T04:00:26.977Z b2539628-3790-11e7-83a1-c323c328542a w50
2017-05-13T04:00:27.458Z b2539628-3790-11e7-83a1-c323c328542a Resi
END RequestId: b2539628-3790-11e7-83a1-c323c328542a
REPORT RequestId: b2539628-3790-11e7-83a1-c323c328542a Duration: 567.14 ms
```

可以看到这里 **Execution result** 下面显示的结果是一个长字符串，已经不是我们以往普通 Lambda 函数返回的 JSON 结构了。想做进一步验证的，可以把这个长字符串 **base64** 解码，会看到一个尺寸变小的图片，那样可以进一步验证我们运行成功。

2.4 配置 API Gateway

管理控制台

<https://us-west-2.console.aws.amazon.com/apigateway/home?region=us-west-2#>

2.4.1 配置

点击 **Create API**

API name 填写 ImageMagick。

Description 填写 Endpoint for Lambda using ImageMagick to perform simple image processing operations, such as resizing.

这时左侧导航链接会显示成 **APIs > ImageMagick > Resources**。点击 **Actions** 下拉菜单，选择 **Create Resource**。

Resource Name* 填写 filepath

Resource Path* 填写 {filepath}，注意要包括大括号。然后点击 Create Resource 按钮。

这时刚刚创建的{filepath}应该是选中状态，再点击 Actions 下拉菜单，选择 Create Method，在当时出现的方法菜单里选择 GET，然后点后面的对号符确定。

然后在/{filepath} - GET - Setup 页，Integration type 保持 Lambda Function 不变，Lambda Region 选 us-west-2，在 Lambda Function 格输入 ImageMagick，下拉的备选菜单中选中 ImageMagick，点击 Save。弹出赋权限提示，点击“OK”。

这时会显示出完整的“/{filepath} - GET - Method Execution”配置页。

点击右上角“Integration Request”链接，进入配置页，点击“Body Mapping Templates”左边的三角形展开之。

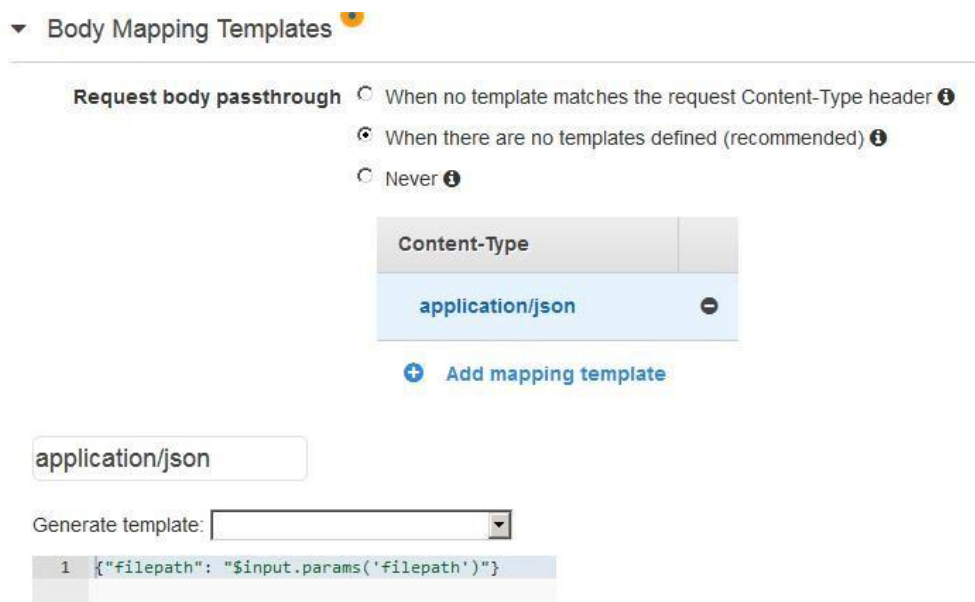
Request body passthrough 选择 When there are no templates defined (recommended)。

点击最下面“add mapping template”链接，“Content-Type”格，注意即使这里已经有提示文字 application/json，还是要自己输入 application/json，然后点击右边的对勾链接，下面会弹出模板编辑输入框，输入

```
{"filepath": "$input.params('filepath')"}

```

完成的效果如下图所示：



最后点击“Save”按钮。点击左上角“Method Execution”链接返回。

点击左下角“Method Response”链接，HTTP Status 下点击第一行 200 左边的三角形展开之，“Response Headers for 200”下点击 add header 链接，Name 格输入 Content-Type，点

点击右边的对勾链接保存。Response Body for 200 下已有一行 application/json，点击其右边的笔图标编辑，把值改成 image/png，点击右边的对勾链接保存。点击左上角 “Method Execution” 链接返回。

点击右下角 “Integration Response” 链接，点击第一行 “- 200 Yes” 左边的三角形展开之，“Content handling” 选择 Convert to binary(if needed)，然后点击 “Save” 按钮。这项配置是把 Lambda 返回的 base64 编码的图片数据转换成二进制的图片数据，是此架构的另一个技术重点。

Header Mappings 下已有一行 Content-Type，点击其右边的笔图标编辑，在 “Mapping value” 格输入 'image/png'，注意要带上单引号，点击右边的对勾链接保存。

点击 “Body Mapping Templates” 左边三角形展开之，点击 “application/json” 右边的减号，把它删除掉。点击左上角 “Method Execution” 链接返回。

点击最左边的竖条 Test 链接，来到 “/{filepath} - GET - Method Test” 页，“{filepath}” 格输入 awslogo_w300.png，点击 Test 按钮。右侧显示类似下面的结果

Request: /awslogo_w300.png

Status: 200

Latency: 247 ms

Response Body 是乱码是正常的，因为我们的返回内容就是图片文件本身。可以查看右下角 Logs 部分显示的详细执行情况，显示类似以下的日志表示执行成功。

Thu Mar 09 03:40:11 UTC 2017 : Method response body
 after transformations: [Binary Data]

Thu Mar 09 03:40:11 UTC 2017 : Method response
 headers:

{X-Amzn-Trace-Id=Root=1-12345678-1234567890abcdefghijkln,

Content-Type=image/png}

Thu Mar 09 03:40:11 UTC 2017 : Successfully completed
 execution

Thu Mar 09 03:40:11 UTC 2017 : Method completed with
 status: 200

2.4.2 部署 API

点击 Actions 按钮，在下拉菜单中点选 Deploy API，Deployment stage 选择 [New Stage]，Stage name 输入 test，注意这里都是小写。

Stage description 输入 test stage

Deployment description 输入 initial deploy to test.

点击 Deploy 按钮。然后会跳转到 Test Stage Editor 页。

复制 Invoke URL: 后面的链接, 比如

`https://1234567890.execute-api.us-west-2.amazonaws.com/test`

然后在后面接上 `awslogo_w300.png`, 组成形如以下的链接

`https://1234567890.execute-api.us-west-2.amazonaws.com/test/awslogo_w300.png`

输入浏览器地址栏里访问, 可以得到一张图片, 表示 API Gateway 已经配置成功。

2.5 配置 CloudFront 分发

我们在 API Gateway 前再加上 CloudFront, 通过 CDN 缓存生成好的图片, 就可以实现不需要把缩略图额外存储, 而又不用每次都为了图片处理进行计算。这里使用了 CDN 和其它使用 CDN 的思路一样, 如果更新图片, 不建议调用清除 CloudFront 的 API, 而是从应用程序生成新的图片标识字符串, 从而生成新的 URL 让 CloudFront 成为无缓存状态从而回源重新计算。

由于 API Gateway 仅支持 HTTPS 访问, 而 CloudFront 同时支持 HTTP 和 HTTPS, 所以我们可以配置成 CloudFront 前端同时支持 HTTP 和 HTTPS, 但是实测发现 CloudFront 前端使用 HTTP 而回源使用 HTTPS 时性能不如前端和回源同为 HTTPS。所以这里我们也采用同时 HTTPS 的方式。

我们打开 CloudFront 的管理控制台

<https://console.aws.amazon.com/cloudfront/home?region=us-west-2#>

点击 Create Distribution 按钮, 在 Web 下点击 Get Started。

Origin Domain Name, 输入上述部署出来的 API Gateway 的域名, 比如 `1234567890.execute-api.us-west-2.amazonaws.com`

Origin Path, 输入上述 API Gateway 的 Stage 名, 如 `/test`

Origin Protocol Policy 选择 HTTPS Only

Object Caching 点选 Customize, 然后 Maximum TTL 输入 86400

Alternate Domain Names (CNAMEs) 栏本例使用自己的域名, 比如 `img.myexample.com`。SSL Certificate 选择 Custom SSL Certificate (example.com), 并从下面的证书菜单中选择一个已经通过 ACM 管理的证书。

注意，如果填写了自己的域名，那么下面的 SSL Certificate 就不建议使用默认的 Default CloudFront Certificate (*.cloudfront.net)，因为很多浏览器和客户端会发现证书的域名和图片 CDN 的域名不一致会报警告。

其它项保持默认，点击 Create Distribution 按钮，然后回到 CloudFront Distributions 列表，这里刚刚创建的记录 Status 会显示为 In Progress，我们点击 ID 的链接，前进到详情页，可以看到 Domain Name 显示一个 CloudFront 分发的 URL，比如 cloudfronttest.cloudfront.net。大约 10 多分钟后，等待 Distribution Status 变成 Deployed，我们可以用上述域名来测试一下。注意测试用的 URL 不要包含 API Gateway 的 Stage 名，比如

https://1234567890.execute-api.us-west-2.amazonaws.com/test/awslogo_w300.png

那么 CloudFront 的 URL 应该是

https://cloudfronttest.cloudfront.net/awslogo_w300.png

尽管我们已经配置了自己的域名，但是这时自己的域名还未生效。我们还需要到 Route 53 去添加域名解析。

2.6 Route 53

最后我们使用 Route 53 实现自定义域名关联 CloudFront 分发。访问 Route 53 管制台

<https://console.aws.amazon.com/route53/home?region=us-west-2>

在 Hosted Zone 下点击一个域名，在域名列表页，点击上方 Create Record Set 按钮，页面右侧会弹出创建记录集的面板。

Name 栏输入 img。

Type 保持默认 A - IP4 Address 不变。

Alias 点选 Yes，在 Alias Target 输入前述创建的 CloudFront 分发的 URL cloudfronttest.cloudfront.net。

点击右下角 Create 按钮完成创建。

3. 效果验证

现在我们回到 CloudFront 控制台，等到我们的 Distribution 的 Status 变成 Deployed，先用 CloudFront 自身的域名访问一下。

https://cloudfronttest.cloudfront.net/awslogo_w300.png

顺利的话，会看到咱们的范例图片。再以自定义域名访问一下。

http://img.myexample.com/awslogo_w300.png

还是输出这张图片，那么到此就全部部署成功了。现在可以在 S3 的 bucket 里上传更多其它图片，比如 `abc.png`，然后访问时使用的 URL 就是

http://img.myexample.com/abc_w300.png

用浏览器打开调试工具，可以看到响应头里已经有

```
X-Cache: Hit from cloudfront
```

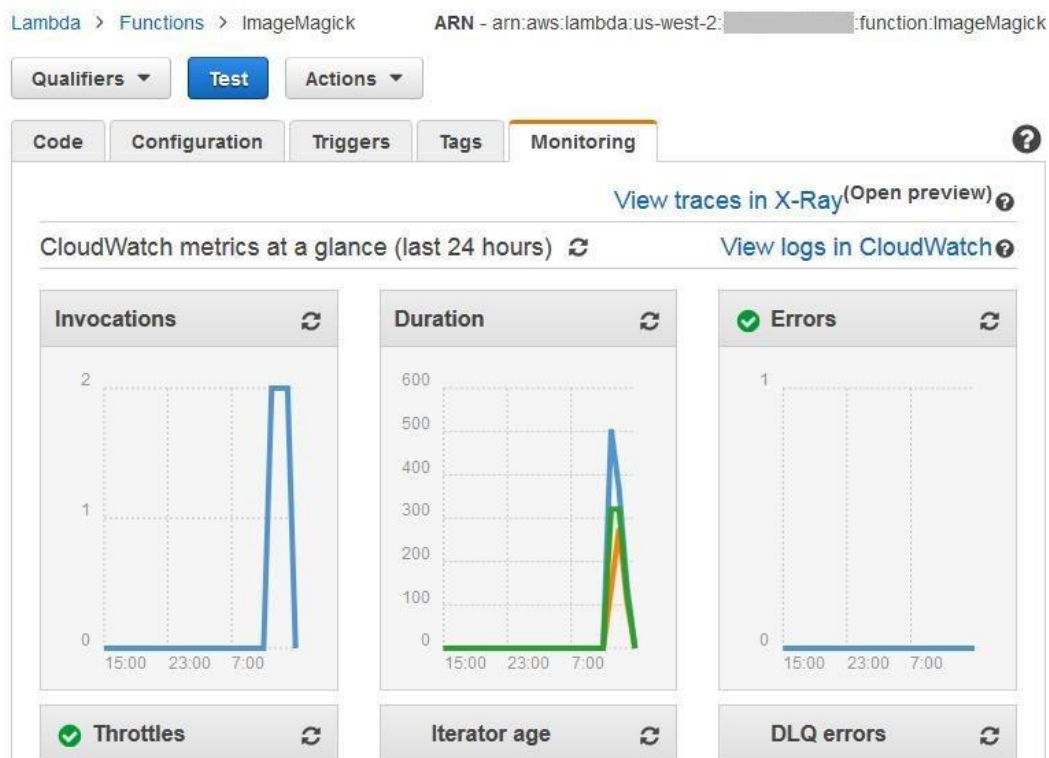
表示已经经过 CloudFront 缓存。

4. 监控

这个架构方案使用的服务都可以通过 CloudTrail 记录管理行为，使用 CloudWatch 记录用户访问情况。

4.1 Lambda 监控

在 Lambda 控制台点击我们的 ImageMagick 函数，然后点击选项卡最末一个 Monitoring，可以看到常用指标的简易图表。

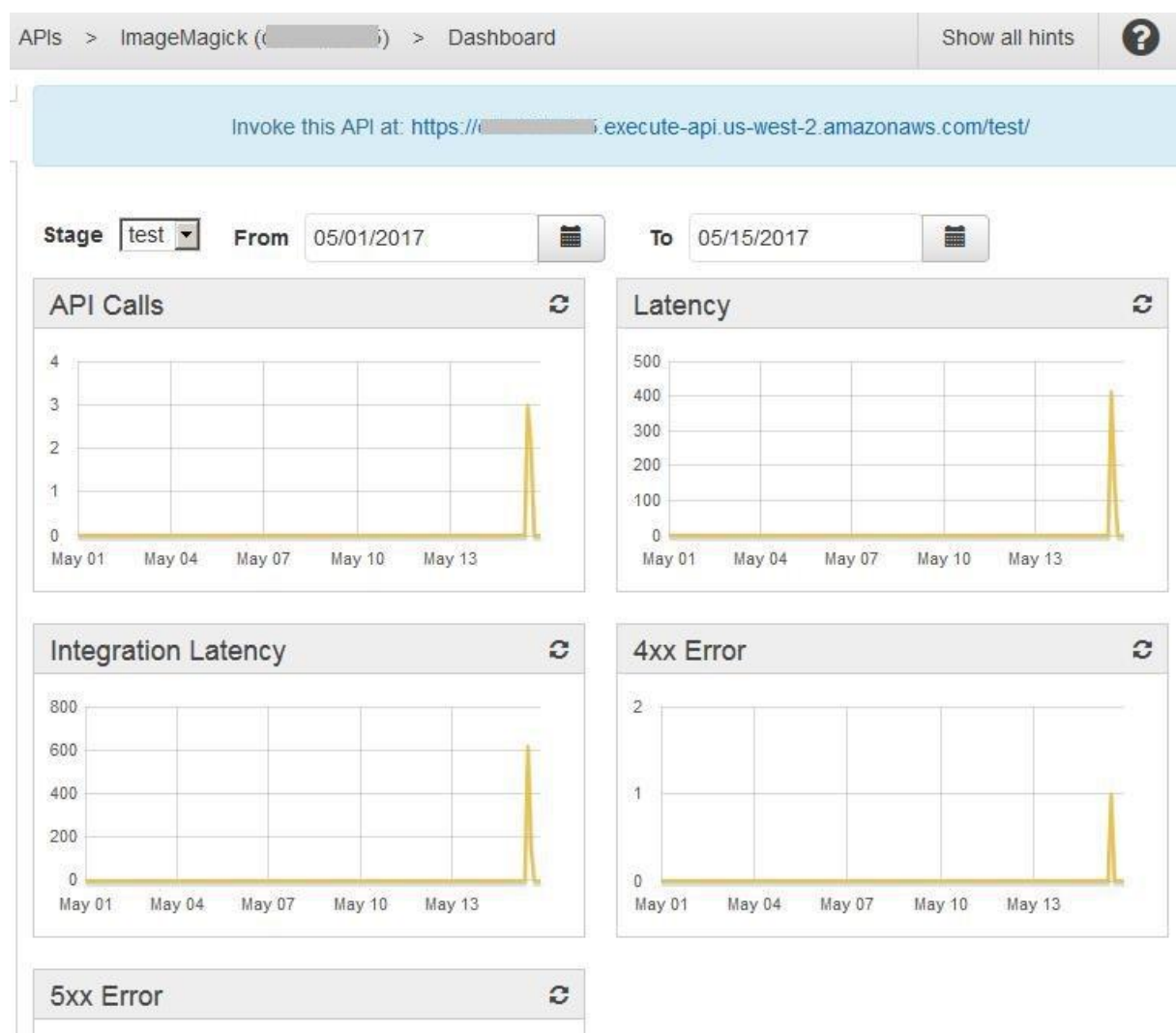


点击任何一个图表，都可以前进到 CloudWatch 相关指标的指标详细页。然后我们还可以为各个指标配置相关的 CloudWatch Alarm，用以监控和报警。

点击 [View logs in CloudWatch](#) 链接，可以前往 CloudWatch Log，这里记录了这个 Lambda 函数每次执行的详细信息，包括我们的函数中自己输出的调试信息，方便我们排查问题。

4.2 API Gateway 监控

在 API Gateway 控制台找到我们的 API ImageMagick，点击它下面的 Dashboard。



如果部署了多个 Stage，注意左上角 Stage 菜单要选择相应的 Stage。同样下面展示的是常用图表，点击每个图表也可以前往 CloudWatch 显示指标监控详情。

4.3 CloudFront 日志

我们刚刚配置 CloudFront 时没有启用日志。如果需要日志，可以来到 CloudFront 控制台，点击我们刚刚创建的分发，在 General 选项页点击 Edit 按钮。在 Edit Distribution 页找到 Logging 项，选择 On，然后再填写 Bucket for Logs 和 Log Prefix，这样 CloudFront 的访问日志就会以文件形式存储在相应的 S3 的 bucket 里了。

5. 小结

我们这样一个例子使用了 **Lambda** 和 **API Gateway** 的一些高级功能，并串联了一系列 **AWS** 全托管的服务，演示了一个无服务器架构的典型场景。虽然实现的功能比较简单，但是 **Lambda** 函数可以继续扩展，提供更丰富功能，比如截图、增加水印、定制文本等，几乎满足任何的业务需求。相比传统的的计算能力部署，不论是用 **EC2** 还是 **ECS** 容器，都要自己管理扩容，而使用 **Lambda** 无需管理扩容，只管运行代码。能够让我们从繁琐的重复工作中解脱，而把业务集中到业务开发上，这正是无服务器架构的真正理念和优势。

作者介绍：



薛峰

AWS 解决方案架构师，**AWS** 的云计算方案架构的咨询和设计，同时致力于 **AWS** 云服务在国内和全球的应用和推广，在大规模并发应用架构、移动应用以及无服务器架构等方面有丰富的实践经验。在加入 **AWS** 之前曾长期从事互联网应用开发，先后在新浪、唯品会等公司担任架构师、技术总监等职位。对跨平台多终端的互联网应用架构和方案有深入的研究。

让你的数据库流动起来 – 利用 MySQL Binlog 实现流式实时分析架构

by AWS Team

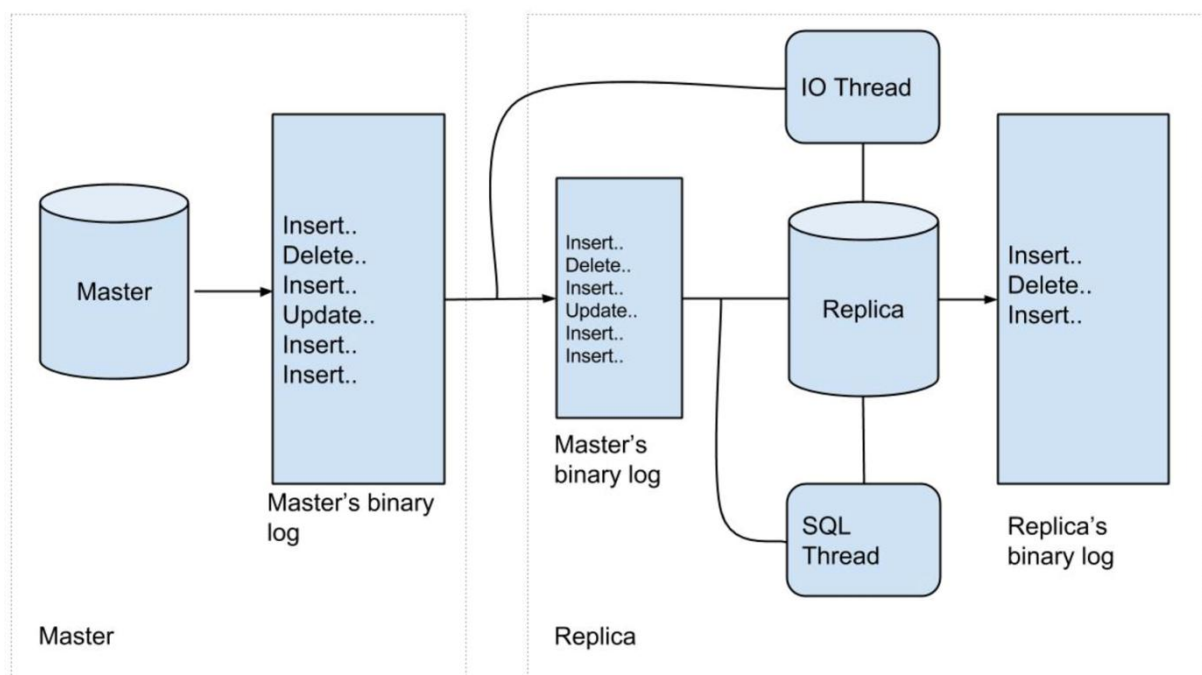
数据分析特别是实时数据分析，已经越来越多的成为各行各业的分析要求与标准 – 例如，（新）零售行业可能希望通过线下 POS 数据与实时门店客流流量的进行实时结合与分析，实现商品销售，销量，总类等等的实时预测； 在线广告平台期望通过广告(Impression)总类，数据量以及基于时间的点击（Click）量，计算实时的广告转化率(Conversion Rate)； 物联网的用户想通过实时分析线下的状态设备与设备采集的数据，进行后台的计算与预判 – 例如做一些设备维修的提前预警(Predicative Failure Analysis)与线下用户的使用习惯； 电商平台或者是在线媒体需要给终端用户提供个性化的实时推荐等等。

纵观这些业务系统，从数据流的角度看，往往数据架构可以分为前后端两个部分 – 前端的业务数据与日志收集系统（其中业务数据系统一般都是利用关系型数据库实现 – 例如 MySQL, PostgreSQL）与后端的数据分析与处理系统（例如 ElasticSearch 搜索引擎, Redshift 数据仓库，基于 S3 的 Hadoop 系统 等等，或者基于 Spark Stream 的实时分析后端）。

“巧妇难为无米炊”，实时数据分析的首要条件是实现实时数据同步，即从上述前端系统到后端系统的数据同步。具体来讲包含两个要求（根据业务场景的不同，实时性会有差异） - 1) 实时 2) 异构数据源的增量同步。实时的要求容易理解 – 无非是前后端系统的实时数据 ETL 的过程，需要根据业务需求，越快越好。所谓异构数据源的增量同步是指，前端产生的增量数据（例如新增数据，删除数据，更新数据 – 主要是基于业务数据库的场景，日志相对简单，主要是随时间的增量数据）可以无缝的同步到后端的数据系统 – 例如 ElasticSearch, S3 或者 Redshift 等。显然，这里的挑战主要是来自于异构数据源的数据 ETL – 直白一点，就是怎么把 MySQL（或者其他 RDBMS）实时的同步到后端的各类异构数据系统。因为，MySQL 的表结构的存储不能简单的通过复制操作实现数据同步。 业界典型的做法大概可以归纳为两类 – 1) 通过应用程序双写的架构 (application dual-writes) 2) 利用流式架构实现数据同步，即基于流式数据的 Change Data Capture (CDC) 。 双写架构实现简单，利用应用逻辑实现，但是要保证数据一致性相对复杂（需要通过二阶段提交实现 – two phase commit），而且，架构扩展相对比较困难 – 例如增加新的数据源，数据库等。 利用流式数据重构数据，越来越成为很多用户与公司的实时数据处理的架构演化方向。 MySQL 的 Binlog，以日志方式记录数据变化，使这种异构数据源的实时同步成为可能。 今天，我们主要讨论的是如何利用 MySQL 的 binlog 实现流式数据同步。

MySQL Binlog 数据同步原理

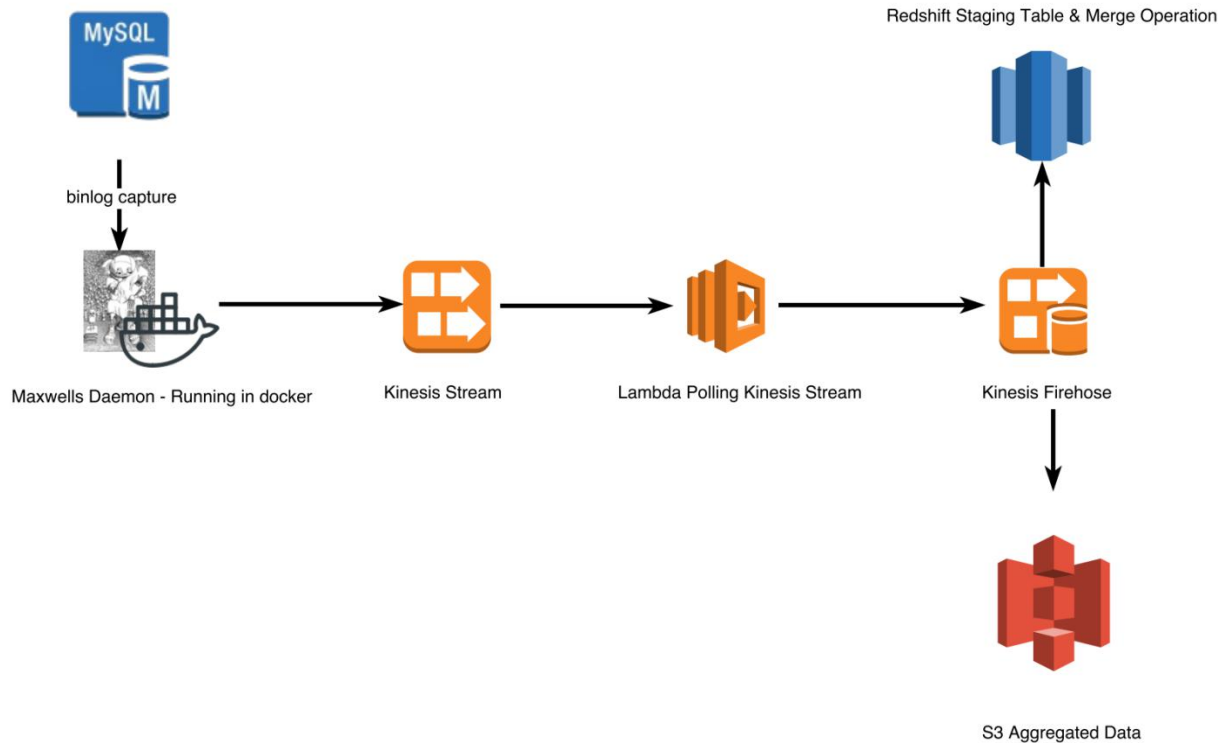
讲了这么多，大家看张图。我们先了解一下 MySQL Binlog 的基本原理。MySQL 的主库（Master）对数据库的任何变化（创建表，更新数据库，对行数据进行增删改），都以二进制文件的方式记录与主库的 Binary Log（即 binlog）日志文件中。从库的 IO Thread 异步地同步 Binlog 文件并写入到本地的 Replay 文件。SQL Thread 再抽取 Replay 文件中的 SQL 语句在从库进行执行，实现数据更新。需要注意的是，MySQL Binlog 支持多种数据更新格式 - 包括 Row，Statement，或者 mix（Row 和 Statement 的混合）。我们建议使用 Row 这种 Binlog 格式（MySQL5.7 之后的默认支持版本），可以更方便更加实时的反映行级别的数据变化。



如前所述，MySQL Binlog 是 MySQL 主备库数据同步的基础，因为 Binlog 以日志文件的方式，记录了数据库的实时变化，所以我们可以考虑类似的方法 - 利用一些客户端工具，把它们伪装成为 MySQL 的 Slave（备库）进行同步。

基于 Binlog 的流式日志抽取的架构与原理

在我们这个场景中，我们需要利用一些客户端工具“佯装”成 MySQL Slave，抽取出 Binlog 的日志文件，并把数据变化注入到实时的流式数据管道中。我们在管道后端对 Binlog 的变化日志，进行消费与必要的数据处理（例如利用 AWS 的 Lambda 服务实现无服务器的代码部署），同步到多种异构数据源中 - 例如 Redshift, ElasticSearch, S3 (EMR) 等等。具体的架构如下图所示。



这里需要给大家介绍一个比较好的 MySQL 的 Binlog 的抽取工具 Maxwell's Daemon。这款由 Zendesk 开发的开源免费(<http://maxwells-daemon.io/>) Binlog 抽取工具可以方便的抽取出 MySQL (包括 AWS RDS)的变化数据，方便的把变化数据以 JSON 的格式注入到后端的 Kafka 或者 Amazon Kinesis Stream 中。我们把 RDS MySQL 中的 Binlog 输出到控制台如下图所示 - 下图表示从 employees 数据库的 employees 数据表中，删除对应的一行数据。

```

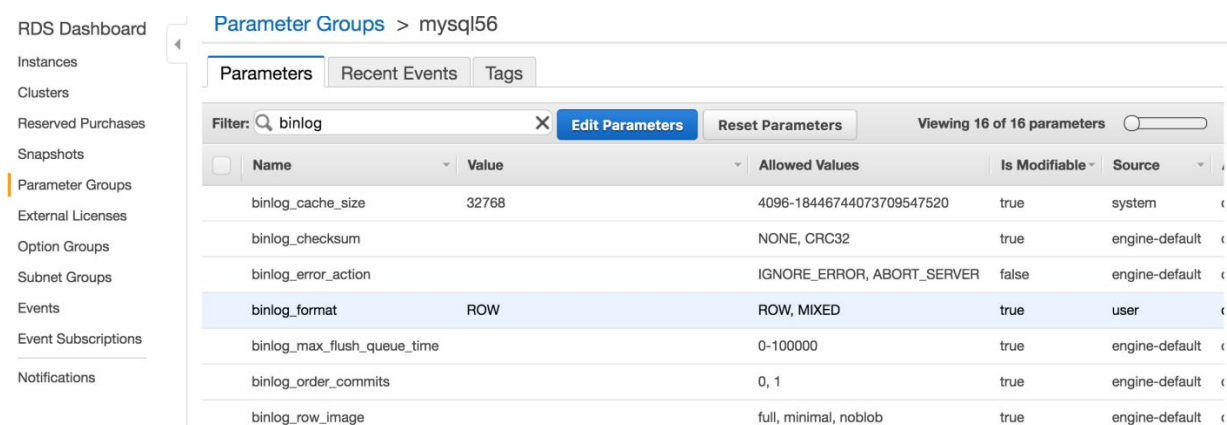
08:44:09,090 INFO Maxwell - Maxwell v1.8.1 is booting (StdoutProducer), starting at BinlogPosition[mysql-bin-changelog.012426:1192]
08:44:09,298 INFO MysqlSavedSchema - Restoring schema id 8 (last modified at BinlogPosition[mysql-bin-changelog.010975:1894])
08:44:09,447 INFO MysqlSavedSchema - Restoring schema id 1 (last modified at BinlogPosition[mysql-bin-changelog.007817:3823])
08:44:09,515 INFO MysqlSavedSchema - beginning to play deltas...
08:44:09,539 INFO MysqlSavedSchema - played 7 deltas in 23ms
08:44:09,648 INFO OpenReplicator - starting replication at mysql-bin-changelog.012426:1192
{"database":"employees","table":"employees","type":"delete","ts":1493282592,"xid":626594,"commit":true,"data":{"emp_no":10022,"birth_date":"1952-07-08","first_name":"Shahaf","last_name":"Famili","gender":"M","hire_date":"1995-08-22"}}
  
```

在上述架构中，我们利用 Lambda 实时读取 Amazon Kinesis Stream 中的 MySQL Binlog 日志，通过 Kinesis Firehose 实时地把 MySQL binlog 的结构数据自动化地同步到 S3 和 Redshift 当中。值得注意的是，整个架构基于高可用和自动扩展的理念 - Kinesis Stream (高可用)，Lambda (Serverless 与自动扩展)，Kinesis Firehose(兼具高可用与自动扩展)。Kinesis Stream 作为统一的一个数据管道，可以通过 Lambda 把数据分发到更多的数据终点 - 例如，ElasticSearch 或者 DynamoDB 中。

动手构建实时数据系统

好了，搞清楚上面的架构，我们开始动手搭建一个 RDS MySQL 的实时数据同步系统吧。这里我们将把 MySQL 的数据变化（包括具体的行操作 - 增删改）以行记录的方式同步到 Redshift 的一张临时表，之后 Redshift 会利用这种临时表与真正的目标表进行合并操作 (Merge) 实现数据同步。

1) 配置 AWS RDS MySQL 的 Binlog 同步为 Row-based 的更新方式：在 RDS 的参数组中，设置 `binlog_format` 为 Row 的格式。如下图所示。



Parameter Groups > mysql56

Parameters Recent Events Tags

Filter: binlog X Edit Parameters Reset Parameters Viewing 16 of 16 parameters

Name	Value	Allowed Values	Is Modifiable	Source
binlog_cache_size	32768	4096-18446744073709547520	true	system
binlog_checksum		NONE, CRC32	true	engine-default
binlog_error_action		IGNORE_ERROR, ABORT_SERVER	false	engine-default
binlog_format	ROW	ROW, MIXED	true	user
binlog_max_flush_queue_time		0-100000	true	engine-default
binlog_order_commits		0, 1	true	engine-default
binlog_row_image		full, minimal, noblob	true	engine-default

2) 另外，我们可以利用 AWS RDS 提供的存储过程，实现调整 Binlog 在 RDS 的存储时间为 24 个小时。我们在 SQL 的客户端输入如下命令：

```
call mysql.rds_set_configuration('binlog retention hours', 24)
```

3) 在这里我们通过如下的 AWS CLI，快速启动一个 stream（配置 CLI 的过程可以参考 <http://docs.aws.amazon.com/cli/latest/userguide/installing.html>，并且需要配置 AWS 的用户具有相应的权限，为了方便起见，我们在这个测试中配置 CLI 具有 Administrator 权限）：这里我们创建了一个名为 `mysql-binlog` 的 kinesis stream 同时配置对应的 `shard count` 为 1。

```
aws kinesis create-stream --stream-name mysql-binlog --shard-count 1
```

4) Maxwell's Daemon 提供了 Docker 的封装方式，在 EC2 运行如下 Docker command 就可以方便的启动一个 Maxwell's Daemon 的客户端。其中，蓝色字体部分代表对应的数据库，`region` 与 `kinesis stream` 等。

```
docker run -it --rm --name maxwell-kinesis
-v `cd && pwd`/.aws:/root/.aws saidimu/maxwell sh -c 'cp
/app/kinesis-producer-library.properties.example
/app/kinesis-producer-library.properties && echo "Region=AWS-Region-ID" >>
/app/kinesis-producer-library.properties && /app/bin/maxwell
--user=DB_USERNAME --password=DB_PASSWORD --host=MYSQL_RDS_URI
--producer=kinesis --kinesis_stream=KINESIS_NAME '
```

5) 有了数据注入到 Kinesis 之后，可以利用 Lambda 对 kinesis stream 内部的数据进行消费了。Python 代码示例如下。需要注意的是，我们这里设置 Lambda 的 trigger 是 Kinesis Stream（这里我们对应的 Kinesis Stream 的名字是 mysql-binlog），并且配置对应的 Lambda 访问 Kinesis Stream 的 batch size 为 1。这样，对应的数据实时性可能更快，当然也可以根据需要适度调整 Batch Size 大小。具体的配置过程可以参考 -

<http://docs.aws.amazon.com/lambda/latest/dg/get-started-create-function.html>

Code
Configuration
Triggers
Tags
Monitoring


Code entry type Edit code inline

```

8
9
10 def lambda_handler(event, context):
11     for record in event['Records']:
12         #Kinesis data is base64 encoded so decode here
13         payload = base64.b64decode(record["kinesis"]["data"])
14         json_str = json.loads(payload)
15         print("Binlog Payload: " + json.dumps(json_str) + '\n')
16
17         # put the record into Kinesis delivery string
18         firehoseString = convertToFirehoseString(payload)
19         print('Firehose string: ' + firehoseString)
20         firehose.put_record(DeliveryStreamName=deliveryStreamName, Record={ 'Data': firehoseString})
21
22     return 'Successfully processed {} records.'.format(len(event['Records']))
23

```

Code
Configuration
Triggers
Tags
Monitoring


Kinesis: mysql-binlog
arn:aws:kinesis:us-east-1:460508756665:stream/mysql-binlog
lastProcessingResult: **PROBLEM: Function call failed** batchSize: 1

Add trigger
Refresh triggers

View function policy

上述代码（Python 2.7 runtime）的主要功能实现从 Kinesis Stream 内部读取 base64 编码的默认 binlog data。之后遍历 Kinesis Stream 中的 data record，并直接写入 Kinesis Firehose 中。

6) 接着，我们可以通过配置 Kinesis Firehose 把 lambda 写入的数据同步复制到 S3/Redshift 中。具体的配置细节如下。配置细节可以参考

<http://docs.aws.amazon.com/firehose/latest/dev/basic-create.html#console-to-redshift>

Use the tabs below to view, edit and monitor your delivery stream.

Details Monitoring S3 Logs Redshift Logs Delete Delivery Stream Edit

Delivery stream name*	kinesis-firehose-redshift	Redshift cluster*	redshift-demo
S3 bucket*	iad-s3-bigdata	Redshift database*	test
S3 prefix	maxwell	Redshift table*	maxwell_employee_staging
IAM role*	firehose_delivery_role	Redshift table columns	none
Data transformation*	Disabled	Redshift username*	admin
Source record backup*	Disabled	Redshift COPY options	DELIMITER as ','
S3 buffer size (MB)*	5	Retry duration (sec)*	3600
S3 buffer interval (sec)*	60	COPY command	COPY maxwell_employee_staging FROM 's3://iad-s3-bigdata/<manifest>' CREDENTIALS 'aws_iam_role=arn:aws:iam::<aws-account-id>:role/<role-name>' MANIFEST DELIMITER as ',';
S3 Compression	UNCOMPRESSED		
S3 Encryption	No Encryption		
Status	ACTIVE		
Error logging	Enabled		

View instructions

其中，通过 S3 buffer interval 来指定往 S3/Redshift 中注入数据的频率，同时，Copy Command 用来指定具体的 Redshift 的 Copy 的操作与对应的 Options，例如（我们指定逗号作为原始数据的分隔符 - 在 lambda 内部实现）。

至此，我们已经可以实现自动化地从 MySQL Binlog 同步变量数据到 Redshift 内部的临时表内。好了，可以试着从 MySQL 里面 delete 一行数据，看看你的 Redshift 临时表会发生什么变化？

还有问题？手把手按照 github 上面的 code 走一遍吧 -

<https://github.com/bobshaw1912/cdc-kinesis-demo>

作者介绍



肖凌

AWS 解决方案架构师，负责基于 AWS 的云计算方案架构的咨询和设计，同时致力于 AWS 云服务在国内和全球的应用和推广，在大规模并发后台架构、跨境电商应用、社交媒体分享、Hadoop 大数据架构以及数据仓库等方面有着广泛的设计和实践经验。在加入 AWS 之前曾长期从事移动端嵌入式系统开发，IBM 服务器开发工程师。并负责 IBM 亚太地区企业级高端存储产品支持团队，对基于企业存储应用的高可用存储架构和方案有深入的研究。

AWS Organizations 提供基于策略的集中化帐户管理能力

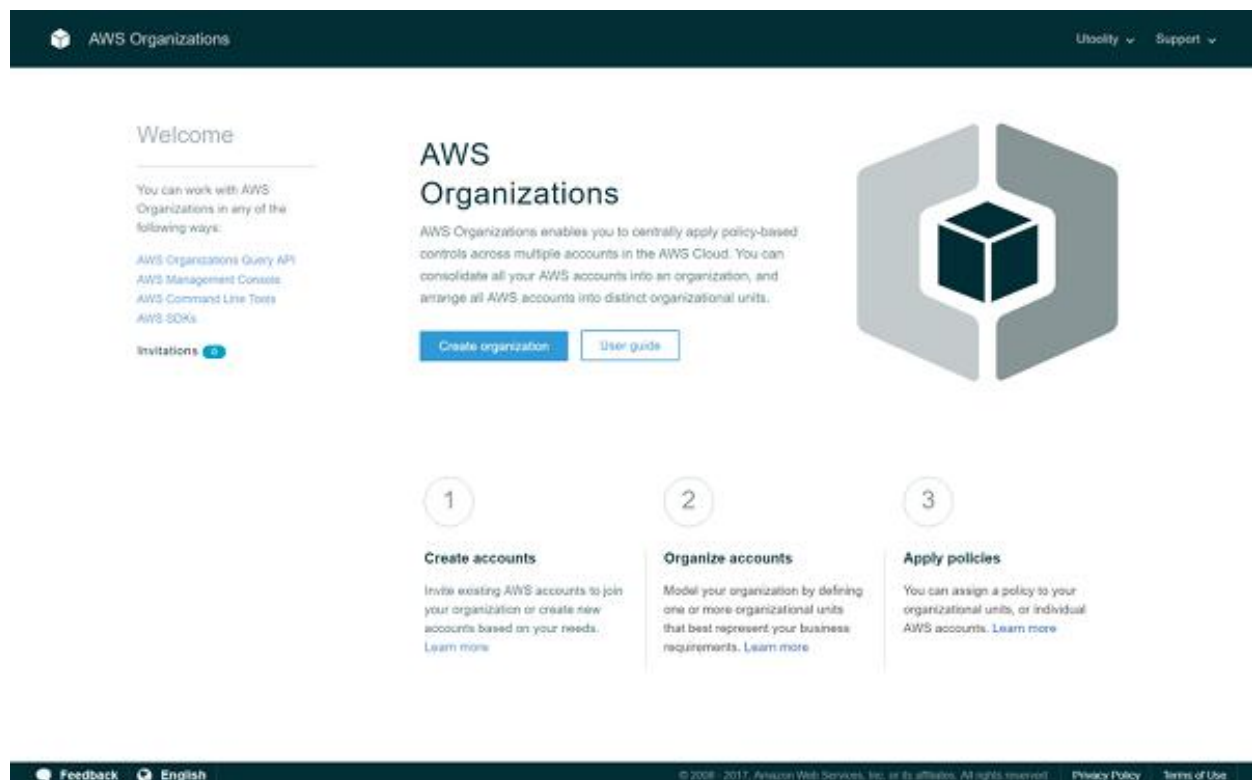
by [Steffen Opel](#), 译者 [大愚若智](#)

自从在 re:Invent 2016 大会上发布[预览版](#)三个月后，Amazon Web Services [最近](#)正式发布了 AWS Organizations。这个新服务可在由组织单位（Organizational unit）组成的层次结构中集中管理多个 AWS 帐户，并能通过细化的访问权限应用服务控制策略。

根据 Amazon Web Service 首席传道士 [Jeff Barr](#) 的介绍，很多 AWS 用户出于不同原因正在同时使用多个帐户，例如需要循序渐进地在不同的部门和团队中采用云计算，或仅仅是为了“满足合规性方面的严格要求，或创建更强大的隔离壁垒”，例如创建相互严格隔离的开发、测试，以及生产环境。

在 VPC peering，EC2 镜像、EBS 和 RDS 快照共享，以及通过 IAM 角色实现的跨帐户控制台访问等跨帐户功能帮助下，AWS 早已支持相同或不同组织的帐户相互协作。然而为了对这些跨帐户功能的依赖性进行一致的管理，运维方面将很快面临挑战。

针对这种需求打造的 [AWS Organizations](#) 服务意在降低运维复杂度，提供“集中管理多个 AWS 帐户，创建由组织单位（OU）组成的层次结构，将不同帐户分配到不同 OU，定义策略，随后将策略应用给整个层次结构，或也可应用给所选 OU，甚至特定帐户”。



最重要的 [AWS Organizations](#) 概念包括：

组织 (Organization) - 一个组织可以有一个“主帐户 (Master)”，以及通过层次结构整理在一起的零个或多个成员帐户，借此可组成一种树状结构。

帐户 (Account) - 包含 AWS 资源的常规 AWS 帐户。

根 (Root) - 一个组织中所包含所有帐户的父容器。

组织单位 (Organizational unit, OU) - 一种帐户容器，也可通过内嵌其他 OU 的方式创建层次结构，应用给一个 OU 的策略可影响层次结构内的所有帐户。

服务控制策略 (Service Control Policy, SCP) - 一种 JSON 格式的策略，类似于身份和访问管理 (IAM) 策略，决定了受到该 SCP 影响的用戶可执行的操作，以及帐户所具备的角色。

通过选择一个主帐户 [创建](#)了组织后，可通过[邀请](#)的方式添加成员帐户，或直接在组织内部[创建](#)成员帐户，此外还可以编程的方式通过 [CLI](#) 或 [API](#) 添加，这是一个大家期待已久的功能。然而虽然邀请加入组织的帐户也可以[离开](#)组织，但在组织内部直接创建的帐户无法离开，也无法删除，提到这一点是因为该服务默认限制最多可以创建 20 个组织帐户，如果不够用必须通过[申请](#)的方式提高限制。

曾经使用过[汇总帐单 \(Consolidated Billing\)](#) 功能的 AWS 客户在[迁移](#)至 AWS Organization 之后可以继续保留原本的功能，包括通过[预留的 EC2 和 RDS 实例用量](#)节约成本的权益以及[大用量折扣](#)。这种情况下，新增的基于策略的访问控制机制需要由主帐户选择性开启，随后需要得到所有成员帐户的确认。仅使用访问控制机制但不使用汇总帐单的做法目前[尚不支持](#)。

一个组织中的每个帐户均可分配给一个组织单位 (OU)，组织单位最多可支持五级层次结构。该层次结构从根容器开始，根容器是独立于组织存在的，这种设计主要是为了在未来实现对更多层次结构的支持，而这种方式正是最近[刚刚发布的 Amazon Cloud Directory](#) 服务的基础。

阅读英文原文: [AWS Organizations Offers Centralized Policy-Based Account Management](#)

Netflix 打造容器管理基础架构：Titus

by [Andrew Spyker](#) 等，译者 [大愚若智](#)

容器技术在 Netflix 基于 AWS EC2 虚拟机打造的全球云平台中大放异彩，并产生了深远影响。之前我们曾分享过不少有关 Netflix 使用容器的故事（[视频](#)、[幻灯片](#)），本文将**深入讨论 Netflix 对容器的使用情况**，以及 Netflix 为基于容器的应用程序打造的底层基础架构 Titus。Titus 为 Netflix 提供了可伸缩的集群和资源管理能力，增强了容器执行与 Amazon EC2 之间的集成度，提供了 Netflix 所需的通用基础架构功能。

本月，Netflix 的容器技术迎来两个重大里程碑。首先，我们的平台规模更上一层楼，在一周内顺利扩展至上百万个容器。其次，Titus 现已可为提供流播服务客户体验的功能提供支持。我们将深入介绍 Docker 容器的具体使用方法，以及使得该容器运行时显得如此独一无二的原因。

容器的发展历史

基于 Amazon 虚拟机（EC2）的基础架构早已成为 Netflix 实现各项创新的重要推动力。除了虚拟机，我们还大量使用基于容器的工作负载，这类工作负载为我们提供了独特的价值。容器技术带给我们不菲的收益，对此我们倍感激动，同时容器的用量也经历了爆发式增长，甚至我们自己的开发人员都对这些因素感到吃惊。

虽然 EC2 支持对服务进行高级调度，但我们的批处理用户无法从中获益。Netflix 有很多用户需要同时运行多个作业，甚至要通过基于事件的触发器分析数据，执行运算，并将结果发送给其他 Netflix 服务、用户，以及报告。我们需要每天多次运行诸如机器学习模型训练、媒体编码、持续集成测试、大数据 Notebook，以及 CDN 部署分析作业等工作负载。因此我们希望为基于容器的应用程序提供不依赖具体工作负载类型的通用资源调度器，通过更高级的工作流调度器对所有内容进行集中控制。Titus 可同时承担通用部署单位（Docker 镜像）和通用批处理作业调度系统的任务，它的顺利上线帮助 Netflix 通过扩展为批处理用例提供了更好的支持。

借助 Titus，我们的批处理用户只需要指定自己对资源的需求，即可快速获得成熟的基础架构环境。用户无须选择并维护在规模方面无法与自己的工作负载完美匹配的 AWS EC2 实例，而是可以安心地通过 Titus 将更大规模的实例“打包”在一起，高效运用于不同类型的工作负载。批处理用户可在本地编写代码，随后立即通过调度通过 Titus 进行大规模执行。在容器的帮助下，Titus 可以运行任何批处理应用程序，并且可以明确指定自己需要的应用程序代码和依赖项。例如，在机器学习训练过程中，用户将能配合运行 Python、R、Java 和 Bash 脚本应用程序。

除了批处理，我们发现其他工作负载也能从更简单的资源管理和本地开发体验等方面获益。通过与 Edge、UI 和设备工程团队合作，我们还将这些收益扩展到了服务的使用者群体。目前，我们正在通过重建将[面向具体设备的服务器端逻辑部署到 API 层](#)，这样即可更充分地利用面向单内核优化的 NodeJS 服务器。我们的 UI 和设备工程团队还希望获得更好的开发体验，例如与生产环境完全一致，更简单的本地测试环境。

除了一致的环境，开发者还可以在容器技术的帮助下，通过 Docker 层镜像和面对容器部署进行优化的预供应虚拟机，更快速地推送新版应用程序。现在，使用 Titus 进行的部署可在 1-2 分钟内完成，而以往单纯使用虚拟机时往往需要数十分钟。

所有这些改进都源自开发者更快速的创新。现在，批处理用户和服务用户都可以更快速地在本地进行实验和测试，他们还可以更自信地将代码部署到生产环境。这样的速度也可以让 Netflix 客户更快获得各类新功能。可以说，容器技术对我们的业务非常重要。

深入了解 Titus

上文介绍了我们构建 Titus 的原因。下文将深入介绍 Titus 到底是如何提供这些价值的。我们将简要介绍 Titus 的调度功能和容器的执行如何为服务和批处理作业需求提供支持，具体情况如下图所示。



在处理应用程序的调度工作时，Titus 可对所需资源和可用计算资源进行匹配。Titus 可支持“永久”运行的服务作业，以及“持续运行直到完成”的批处理作业。服务作业可重启失败实例，并通过自动伸缩与负载的变化相匹配。批处理作业可根据策略进行重试并一直运行直到完成。

Titus 为资源调度提供了多种 SLA。通过 EC2 的自动伸缩功能，Titus 可根据当前需求为即席（Ad hoc）批处理和非关键内部服务提供按需获取的计算容量。它还能为用户需要执行的工作负载和更关键的批处理任务提供预配置、有保障的容量。调度器可通过集装优化（Bin

packing) 提高大规模虚拟机的效能, 并能通过反关联性 (Anti-affinity) 实现跨越虚拟机和可用性集的可靠性。这种调度功能源自 Netflix 开发的一个名为 [Fenzo](#) 的开源库。

Titus 容器运行在 EC2 虚拟机基础上, 并能与 AWS 和 Netflix 基础架构进行集成。我们预测到用户会在很长一段时间内同时使用虚拟机和容器, 因此决定尽可能让云平台 and 运维体验保持简单。使用 AWS 的过程中, 我们选择尽可能利用现有的 EC2 服务。例如, 我们会使用 Virtual Private Cloud (VPC) 获得可路由的 IP, 而没有使用单独的网络覆盖 (Network overlay)。我们使用 Elastic Network Interface (ENI) 以确保所有容器都包含特定应用程序所需的安全组。Titus 提供的元数据代理使得容器可以了解自己运行环境特定的容器视图以及 IAM 凭据。容器无法看到宿主机的元数据 (例如 IP、主机名、实例 ID)。我们还配合使用 Linux、Docker, 以及自有的隔离技术实施了多租户隔离 (CPU、内存、磁盘、网络、安全)。

为了成功地在 Netflix 的环境中应用容器技术, 我们还要将其与现有开发工具和运维基础架构无缝集成。例如, Netflix 已经具备持续交付解决方案 [Spinnaker](#)。虽然也可以通过调度器进行滚动更新并实现其他 CI/CD 概念, 但通过 Spinnaker 提供这样的功能使得我们的用户可以跨越虚拟机和容器使用一套一致的部署工具。另外还有个例子: 服务之间的通信。我们会尽可能避免重新实现服务发现和服务负载均衡工作, 取而代之的是提供一个完整的 IP 栈, 借此容器将能与现有的 Netflix [服务发现](#) 和基于 DNS (Route 53) 的负载均衡机制配合使用。在这些例子中, Titus 成功与否的一个关键原因在于: 需要确定那些事情不该通过 Titus 来做, 而是应该利用其他基础架构团队所提供的相关功能来实现。

继续使用现有系统的代价是: 我们需要对这些系统进行扩充, 以便能与虚拟机和容器技术配合使用。除了上文列举的例子, 我们还需要对遥测、性能自动调优、健康度检查系统、混沌自动化 (Chaos automation)、流量控制、区域性故障转移支持、密文管理, 以及交互式系统访问等机制进行扩充。另一个代价是: 由于需要与 Netflix 的这些系统进行深入绑定, 这使得我们很难利用其他开源容器解决方案提供容器运行时平台之外的其他功能。

以我们的规模 (以及工作负载密度) 来说, 容器平台的运行还需要密切关注可靠性, 并且我们还会在系统的所有层面上遭遇不小的挑战。我们已经顺利解决了 Titus 软件, 以及所依赖的其他开源软件 (Docker Engine、Docker Distribution、Apache Mesos、Snap 和 Linux) 相关的伸缩性和可靠性问题。我们针对整个系统的所有层面设计了失败处理机制, 包括通过协调促进资源管理层和容器运行时之间分布式状态的一致性。通过清晰衡量的服务级别目标 (容器发布启动延迟、由于 Titus 中问题导致容器崩溃的百分率, 以及系统 API 整体可用性), 我们可以更好地对可靠性和功能进行权衡。

容器技术帮助工程师提高效率的一个关键原因在于开发工具。开发者生产力工具团队构建了一个名为 [Newt](#) (Netflix Workflow Toolkit) 的本地开发工具。通过本地迭代并借助 Titus 进行部署, Newt 可简化容器开发工作。通过在 Newt 和 Titus 之间实现一致的容器环境, 也可以帮助开发者更自信地编写代码。

目前 Titus 的使用情况

为了驱动 Netflix 的服务，我们通过三个 Amazon 区域，用多个测试和生产账户运行了多个 Titus 栈。

2015 年 12 月发布 Titus 后，当时我们每周需要启动数千个容器，用来处理少量工作负载。上周，我们启动了上百万个容器，这些容器中运行了数百个工作负载。一年多时间里，容器的用量增加了 1000 倍，而增长势头依然没有任何放缓的迹象。

为了向批处理用户提供支持，我们在峰值时段运行了 500 个 r3.8xl 实例，这意味着 16,000 个处理器内核，以及 120TB 内存的规模。我们还通过 p2.8xl 实例获得了所需的 GPU 资源，借此支撑基于神经网络和微批（Mini-batch）的深度学习功能。

2017 年上半年，流处理即服务团队决定通过 Titus 为自己基于 Flink 的系统提供更简单快速的集群管理能力。这一决定导致新增了超过 10,000 个长期运行的服务作业容器，并且随着流处理作业的变化，这些容器都需要重新部署。各种服务共使用了数千个 m4.4xl 实例。

虽然上述用例对我们的业务很重要，但这些用例都不会对 Netflix 的客户产生直接影响。不过这种情况最近有了些变化，最近我们开始通过 Titus 容器运行某些直接面向 Netflix 客户的服务。

用于支撑直接面向客户的服务，这个决定必须慎重。过去六个月来，我们一直在对虚拟机和容器之间的实时通信进行复制，并使用复制的流量了解如何更好地运维容器，以验证该平台是否已经面向生产环境做好了准备。通过大量的前期工作，我们已经可以自信地在基础架构中推行如此重大的变动。

Titus 团队

Netflix 得以成功运用 Titus 的原因之一在于，Titus 开发团队所积累的宝贵经验和不断的成长。我们的容器用户坚信不疑地认为，Titus 团队可以通过 Titus 的运维和创新满足自己的需求。

这个团队的后续成长还有很长的路要走，我们也希望能将这个容器运行时和开发者体验扩展到更多领域。

[Andrew Spyker](#)、[Andrew Leung](#) 和 [Tim Bozarth](#) 代表整个 Titus 开发团队撰文并发布。

阅读英文原文：[The Evolution of Container Usage at Netflix](#)

感谢郭蕾对本文的审校。

给 InfoQ 中文站投稿或者参与内容翻译工作，请邮件至 editors@cn.infoq.com。也欢迎大家通过新浪微博（[@InfoQ](#)，[@丁晓昀](#)），微信（微信号：[InfoQChina](#)）关注我们。



扫描关注InfoQ中文站



扫描关注AWS公众号