深入浅出Mesos

Profound And Simple Mesos

InfoQ

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系
editors@cn.infoq.com

ArchSummit 全球架构师峰会 2015

[北京站]

2015年12月18日-19日
北京·国际会议中心
www.archsummit.com

11月27日前 优惠立减680元 团购享受更多优惠

大会介绍

交易量逐年暴涨，双十一电商架构今年有何不同？经历了多起安全事件之后，互联网安全架构有何警示？云服务遍地开花，云架构如何能承受爆发式的增长？智能硬件产业方兴未艾，智能设备设计有何玄机？互联网银行蓄势待发，互联网架构跟银行业务又是如何结合？答案尽在ArchSummit北京2015全球架构师峰会。ArchSummit聚集了互联网行业各领域的一线知名架构师及高级技术管理者等，期待与您一起交流承载繁华业务的架构智慧。

主题演讲

梁胜
Rancher Labs创始人
兼全球CEO

袁泳
Uber软件工程师

部分演讲嘉宾（排名不分先后）

陈锐锋
学霸君研发副总裁
如何提供让人耳目一新的在线答疑服务的核心技术？

刘宁
腾讯云安全技术副总监
从甲方到乙方
--腾讯云安全实践之路

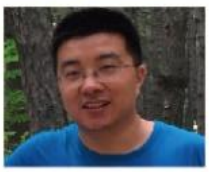
李靖
微众银行架构师
微众银行基于自主可控技术的分布式架构实践

黄正强
Marvell研发总监
Wi-Fi SoC 芯片在IoT智能设备中的应用

段念
宜人贷CTO
为行驶中的汽车升级——快速业务发展环境下的研发团队调优

沈剑
58赶集集团技术总监
58速运-百万单量用车O2O架构实践

陈浩然
携程移动开发总监
移动应用架构优化之旅



李庆丰
新浪微博高级技术经理
新浪微博高可用服务保障体系演进

垂询电话：010-89880682
QQ 咨询：2332883546
E-mail: arch@cn.infoq.com
更多精彩内容，请持续关注archsummit.com

Brought by Geekbang. 极客邦科技 InfoQ

卷 首 语

云计算和大数据是这个时代的主题。这个主题是由数据挖掘、算法、微服务等具体的活动或者业务组成的，进而它们可以分解为离线或实时的任务。这些任务可能是长时间运行的、定时执行的，亦或是短时间内一次执行即结束的。每个任务都需要 CPU 和内存资源来处理，这些资源很可能是跨主机的，甚至是跨数据中心的。那么，面对光鲜亮丽的时代主题，我们该如何设计和实现任务的调度和资源的分配呢？答案是不需要这样做，因为我们有 Mesos。

Mesos 是为软件定义数据中心而生的操作系统，跨数据中心的资源在这个系统中被统一管理。Mesos 非常智慧和优雅地以 Framework 的形式，提供了的两级调度机制，将任务的调度和执行分离。前面提到的各种类型的任务，在调度阶段，由 Framework 的调度器以资源邀约的形式向 Mesos Master 申请资源；在执行阶段，由 Framework 的执行器执行任务。Mesos 帮助我们解决了错综复杂的任务调度和资源管理的问题后，我们可以专注于实现自己的任务，进而完成云计算和大数据业务。

Mesos 的意义不止与此，它为我们带来了一个生机勃勃的生态环境。基于现有的 Framework，我们可以使用 Marathon 跑长时间运行的（微）服务，使用 Chronos 运行批量任务，使用 Spark 做大规模数据处理，使用 Cassandra 实现数据存储。这些 Mesos 生态中的系统都基于两级调度机制，实现了 Framework。Mesos 的这种插件式的 Framework，使其自身轻盈，使其生态蓬勃。

同 Borg 相比，Mesos 更亲民，我们摸得着看得见；同 Omega 相比，Mesos 更现实，曲调虽然不算高山流水，却能奏完每个篇章；与 Kubernates 相比，Mesos 更成熟；Mesos 之于 Hadoop，计算模型上使用 Spark 超越了 MapR，资源管理上，是 YARN 模仿的样板。

自 2009 年，诞生于加州大学伯克利分校的 AMPlab，Meosos 在不断地完善其生态系统。创始人 Benjamin Hindman 以首席架构师的身份，参与着 Mesos 背后的商业实体公司 Mesosphere 的建设，通过推出数据中心操作系统（DCOS）产品，不断降低 Mesos 的使用门槛，提高生产率；创始人 Andy Konwinski 和 Matei Zaharia 是 Spark 背后的商业实体公司 Databricks 的联合创始人。这样的生态体系和商业形式，为开发者带来了更多的信心和参与其中的兴趣。

目　录

深入浅出 Mesos（一）：
为软件定义数据中心而生的操作系统

作者 韩陆

【编者按】Mesos 是 Apache 下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos 最初是由加州大学伯克利分校的 AMPLab 开发的，后在 Twitter 得到广泛使用。InfoQ 接下来将会策划系列文章来为读者剖析 Mesos。本文是整个系列的第一篇，简单介绍了 Mesos 的背景、历史以及架构。

注：本文翻译自 Cloud Architect Musings，InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

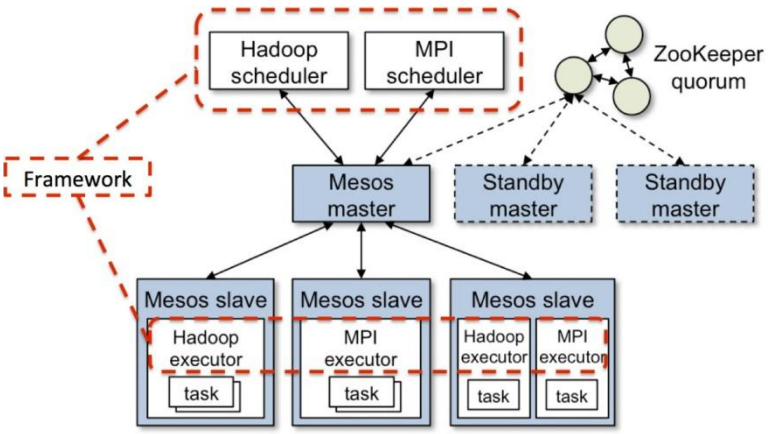
我讨厌“软件定义数据中心（SDDC）”这个词，并不是因为我质疑这个概念，而是我发现很多公司都对这个词有误用，他们甚至直接把这个词拿来套用，并急于把自己定位为下一代数据中心的创新者。具体来说，我认为，在商用 x86 硬件上运行软件（应用）并不是什么 SDDC 解决方案，它也不具备虚拟化硬件到资源池的能力。真正的 SDDC 底层基础架构应该可以从运行于其上的应用程序中抽象出来，并根据应用程序不断变化的需求，动态且自动地分配、重新分配应用程序，然后运行于数据中心的不同组件之中。

这就是为什么我一直兴奋地要在后面介绍 Mesos，一个 Apache 开源项目。为什么我对 Mesos 如此兴奋？回想 x86 虚拟化之初对数据中心曾经的承诺：通过增加服务器利用率使其更高效，通过从物理基础架构抽象应用使其更敏捷。虽然收获颇丰，但是以虚拟机为单位，粒度仍不够精细，如果应用程序都过于庞大，那就难以充分实现这一承诺。如今，飞速发展的容器技术、分布式应用程序和微服务技术正悄然改变着我们对数据中心的运行和管理方式。

试想，可否整合数据中心中的所有资源，并将它们放在一个大的虚拟池里，代替单独的物理服务器；然后开放诸如 CPU、内存和 I/O 这些基本资源而不是虚拟机？同样，可否把应用程序拆分成小的、隔离的任务单位，从而根据数据中心应用的需求，从虚拟数据中心池中动态分配任务资源？就像操作系统将 PC 的处理器和 RAM 放入资源池，使其可以为不同的进程协调分配和释放资源。进一步讲，我们可以把 Mesos 作为操作系统内核，然后将数据中心看为 PC。这也是正是我想说的：Mesos 正在改变数据中心，它让真正的 SDDC 成为现实。

接下来我先介绍下 Mesos 的历史。Mesos 的起源于 Google 的数据中心资源管理系统 Borg。你可以从 WIRED 杂志的这篇文章中了解更多关于 Borg 起源的信息及它对 Mesos 影响。Twitter 从 Google 的 Borg 系统中得到启发，然后就开发一个类似的资源管理系统来帮助他们摆脱可怕的“失败之鲸”（译者注：见上图）。后来他们注意到加州大学伯克利分校 AMPLab 正在开发的名为 Mesos 的项目，这个项目的负责人是 Ben Hindman，Ben 是加州大学伯克利分校的博士研究生。后来 Ben Hindman 加入了 Twitter，负责开发和部署 Mesos。现在 Mesos 管理着 Twitter 超过 30,0000 台服务器上的应用部署，“失败之鲸”已成往事。其他公司纷至沓来，也部署了 Mesos，比如 Airbnb（空中食宿网）、eBay（电子港湾）和 Netflix。

Mesos 是如何让 Twitter 和 Airbnb 这样的公司，通过数据中心资源更高效的管理系统，扩展应用的呢？我们从一个相当简单但很优雅的两级调度架构开始说起。



上图修改自 Apache Mesos 网站上的图片，如图所示，Mesos 实现了两级调度架构，它可以管理多种类型的应用程序。第一级调度是 Master 的守护进程，管理 Mesos 集群中所有节点上运行的 Slave 守护进程。集群由物理服务器或虚拟服务器组成，用于运行应用程序的任务，比如 Hadoop 和 MPI 作业。第二级调度由被称作 Framework 的“组件”组成。Framework 包括调度器（Scheduler）和执行器（Executor）进程，其中每个节点上都会运行执行器。Mesos 能和不同类型的 Framework 通信，每种 Framework 由相应的应用集群管理。上图中只展示了 Hadoop 和 MPI 两种类型，其它类型的应用程序也有相应的 Framework。

Mesos Master 协调全部的 Slave，并确定每个节点的可用资源，聚合计算跨节点的所有可用资源的报告，然后向注册到 Master 的 Framework（作为 Master 的客户端）发出资源邀约。Framework 可以根据应用程序的需求，选择接受或拒绝来自 master 的资源邀约。一旦接受邀约，Master 即协调 Framework 和 Slave，调度参与节点上任务，并在容器中执行，以使多种类型的任务，比如 Hadoop 和 Cassandra，可以在同一个节点上同时运行。

我将在接下来的文章中，详细介绍 Mesos 的体系结构和工作流。我认为，Mesos 使用的两级调度架构以及算法、隔离技术让在同一个节点上运行多种不同类型的应用成为了现实，这才是数据中心的未来。正如我之前所述，这是到目前为止我所见过的，履行 SDDC 承诺最好的现成技术。

我希望这篇介绍让你受用并吊起你了解 Mesos 的胃口。接下来，我将带你深入技术细节，教你一些上手方法，还会告诉你如何加入社区。

查看英文原文： APACHE MESOS: THE TRUE OS FOR THE SOFTWARE DEFINED DATA CENTER?

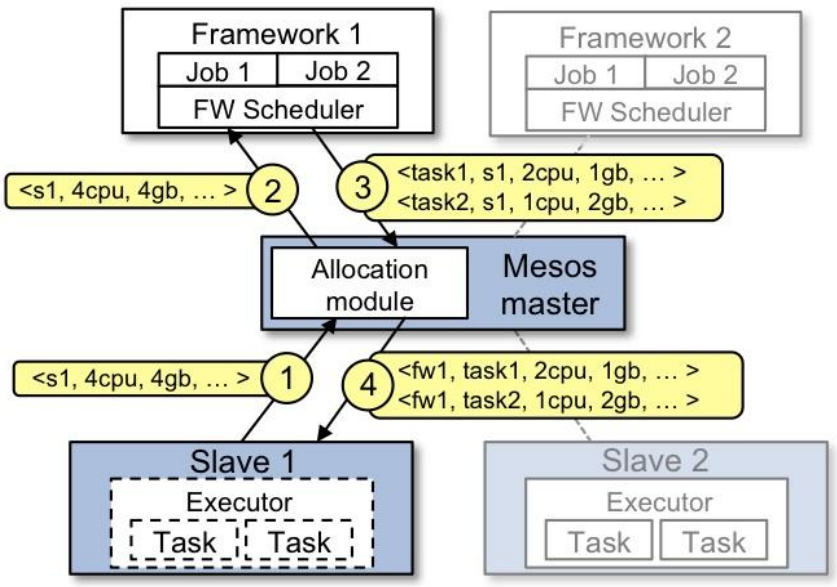
深入浅出 Mesos（二）：Mesos 的体系结构和工作流

作者 韩陆

在本系列的第一篇文章中，我简单介绍了 Apache Mesos 的背景、架构，以及它在数据中心资源管理中的价值。本篇文章将深入剖析 Mesos 的技术细节和组件间的流程，以便大家更好地理解为什么 Mesos 是数据中心操作系统内核的重要候选者。文中所述的大部分技术细节都来自 Ben Hindman 团队 2010 年在加州大学伯克利分校时发表的白皮书。顺便说一句，Hindman 已经离开 Twitter 去了 Mesosphere，着手建设并商业化以 Mesos 为核心的数据中心操作系统。在此，我将重点放在提炼白皮书的主要观点上，然后给出一些我对相关技术所产生的价值的思考。

Mesos 流程

接着上一篇文章说。并结合前述的加州大学伯克利分校的白皮书以及 Apache Mesos 网站，开始我们的讲述：



我们来研究下上图的事件流程。上一篇谈到，Slave 是运行在物理或虚拟服务器上的 Mesos 守护进程，是 Mesos 集群的一部分。Framework 由调度器（Scheduler）应用程序和任务执行器（Executor）组成，被注册到 Mesos 以使用 Mesos 集群中的资源。

- Slave 1 向 Master 汇报其空闲资源：4 个 CPU、4GB 内存。然后，Master 触发分配策略模块，得到的反馈是 Framework 1 要请求全部可用资源。
- Master 向 Framework 1 发送资源邀约，描述了 Slave 1 上的可用资源。
- Framework 的调度器（Scheduler）响应 Master，需要在 Slave 上运行两个任务，第一个任务分配<2 CPUs, 1 GB RAM>资源，第二个任务分配<1 CPUs, 2 GB RAM>资源。
- 最后，Master 向 Slave 下发任务，分配适当的资源给 Framework 的任务执行器（Executor）,接下来由执行器启动这两个任务（如图中虚线框所示）。此时，还有 1 个 CPU 和 1GB 的 RAM 尚未分配，因此分配模块可以将这些资源供给 Framework 2。

资源分配

为了实现在同一组 Slave 节点集合上运行多任务这一目标，Mesos 使用了隔离模块，该模块使用了一些应用和进程隔离机制来运行这些任务。不足为奇的是，虽然可以使用虚拟机隔离实现隔离模块，但是 Mesos 当前模块支持的是容器隔离。Mesos 早在 2009 年就用上了 Linux 的容器技术，如 cgroups 和 Solaris Zone，时至今日这些仍然是默认的。然而，Mesos 社区增加了 Docker 作为运行任务的隔离机制。不管使用哪种隔离模块，为运行特定应用程序的任务，都需要将执行器全部打包，并在已经为该任务分配资源的 Slave 服务器上启动。当任务执行完毕后，容器会被“销毁”，资源会被释放，以便可以执行其他任务。

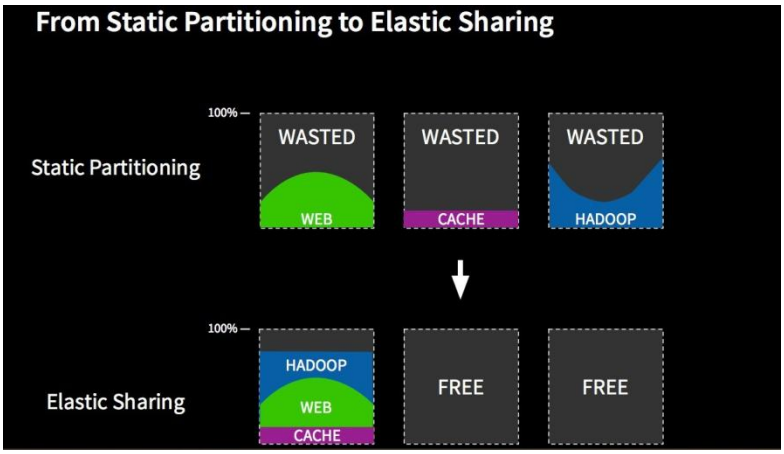
我们来更深入地研究一下资源邀约和分配策略，因为这对 Mesos 管理跨多个 Framework 和应用的资源，是不可或缺的。我们前面提到资源邀约的概念，即由 Master 向注册其上的 Framework 发送资源邀约。每次资源邀约包含一份 Slave 节点上可用的 CPU、RAM 等资源的列表。Master 提供这些资源给它的 Framework，是基于分配策略的。分配策略对所有的 Framework 普遍适用，同时适用于特定的 Framework。Framework 可以拒绝资源邀约，如果它不满足要求，若此，资源邀约随即可以发给其他 Framework。由 Mesos 管理的应用程序通常运行短周期的任务，因此这样可以快速释放资源，缓解 Framework 的资源饥饿；Slave 定期向 Master 报告其可用资源，以便 Master 能够不断产生新的资源邀约。另外，还可以使用诸如此类的技术，每个 Fraamework 过滤不满足要求的资源邀约、Master 主动废除给定周期内一直没有被接受的邀约。

分配策略有助于 Mesos Master 判断是否应该把当前可用资源提供给特定的 Framework，以及应该提供多少资源。关于 Mesos 中使用资源分配以及可插拔的分配模块，实现非常细粒度的资源共享，会单独写一篇文章。言归正传，Mesos 实现了公平共享和严格优先级（这两个概念我会在资源分配那篇讲）分配模块，确保大部分用例的最佳资源共享。已经实现的新分配模块可以处理大部分之外的用例。

集大成者

现在来回答谈及 Mesos 时，“那又怎样”的问题。对于我来说，令人兴奋的是 Mesos 集四大好处于一身（概述如下），正如我在前一篇文章中所述，我目测 Mesos 将为下一代数据中心的操作系统内核。

- 效率：这是最显而易见的好处，也是 Mesos 社区和 Mesosphere 经常津津乐道的。

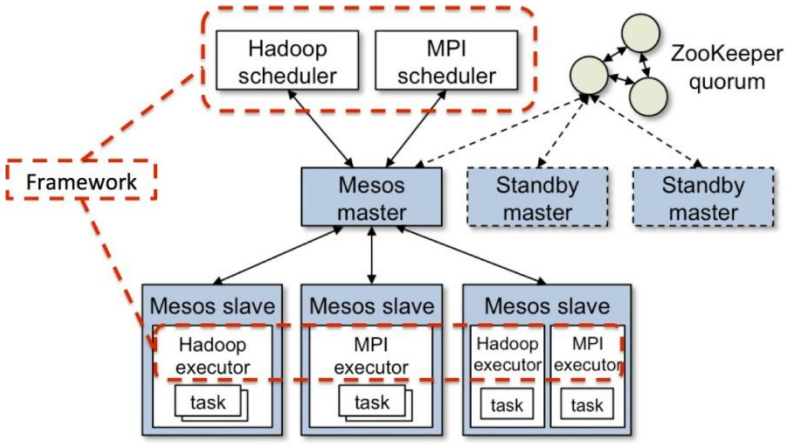


上图来自 Mesosphere 网站，描绘出 Mesos 为效率带来的好处。如今，在大多数数据中心中，服务器的静态分区是常态，即使使用最新的应用程序，如 Hadoop。这时常令人担忧的是，当不同的应用程序使用相同的节点时，调度相互冲突，可用资源互相争抢。静态分区本质上是低效的，因为经常会面临，其中一个分区已经资源耗尽，而另一个分区的资源却没有得到充分利用，而且没有什么简单的方法能跨分区集群重新分配资源。使用 Mesos 资源管理器仲裁不同的调度器，我们将进入动态分区/弹性共享的模式，所有应用程序都可以使用节点的公共池，安全地、最大化地利用资源。一个经常被引用的例子是 Slave 节点通常运行 Hadoop 作业，在 Slave 空闲阶段，动态分配给他们运行批处理作业，反之亦然。值得一提的是，这其中的某些环节可以通过虚拟化技术，如 VMware vSphere 的分

布式资源调度（DRS）来完成。然而，Mesos 具有更精细的粒度，因为 Mesos 在应用层而不是机器层分配资源，通过容器而不是整个虚拟机（VM）分配任务。前者能够为每个应用程序的特殊需求做考量，应用程序的调度器知道最有效地利用资源；后者能够更好地“装箱”，运行一个任务，没有必要实例化一整个虚拟机，其所需的进程和二进制文件足矣。

- 敏捷：与效率和利用率密切相关，这实际上是我认为最重要的好处。往往，效率解决的是“如何花最少的钱最大化数据中心的资源”，而敏捷解决的是“如何快速用上手头的资源。”正如我和我的同事 Tyler Britten 经常指出，IT 的存在是帮助企业赚钱和省钱的；那么如何通过技术帮助我们迅速创收，是我们要达到的重要指标。这意味着要确保关键应用程序不能耗尽所需资源，因为我们无法为应用提供足够的基础设施，特别是在数据中心的其他地方都的资源是收费情况下。
- 可扩展性：为可扩展而设计，这是我真心欣赏 Mesos 架构的地方。这一重要属性使数据可以指数级增长、分布式应用可以水平扩展。我们的发展已经远远超出了使用巨大的整体调度器或者限定群集节点数量为 64 的时代，足矣承载新形式的应用扩张。

  Mesos 可扩展设计的关键之处是采用两级调度架构。使用 Framework 代理任务的实际调度，Master 可以用非常轻量级的代码实现，更易于扩展集群发展的规模。 因为 Master 不必知道所支持的每种类型的应用程序背后复杂的调度逻辑。此外，由于 Master 不必为每个任务做调度，因此不会成为容量的性能瓶颈，而这在为每个任务或者虚拟机做调度的整体调度器中经常发生。



- 模块化：对我来说，预测任何开源技术的健康发展，很大程度上取决于围绕该项目的生态系统。我认为 Mesos 项目前景很好，因为其

设计具有包容性，可以将功能插件化，比如分配策略、隔离机制和 Framework。将容器技术，比如 Docker 和 Rocket 插件化的好处是显而易见。但是我想在此强调的是围绕 Framework 建设的生态系统。将任务调度委托给 Framework 应用程序，以及采用插件架构，通过 Mesos 这样的设计，社区创造了能够让 Mesos 问鼎数据中心资源管理的生态系统。因为每接入一种新的 Framework，Master 无需为此编码，Slave 模块可以复用，使得在 Mesos 所支持的宽泛领域中，业务迅速增长。相反，开发者可以专注于他们的应用和 Framework 的选择。当前而且还在不断地增长着的 Mesos Framework 列表参见[此处](#)以及下图：

总结

在接下来的文章中，我将更深入到资源分配模块，并解释如何在 Mesos 栈的各级上实现容错。同时，我很期待读者的反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。 我也会在 Twitter 响应你的反馈，请关注 @hui_kenneth。

下一篇是关于 Mesos 的持久性存储和容错的。

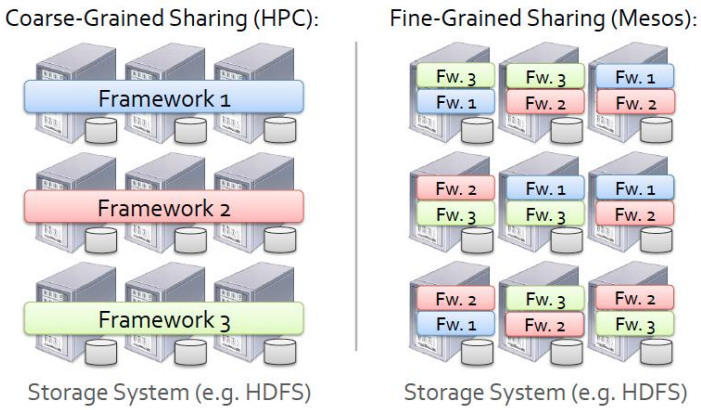
查看英文原文： DIGGING DEEPER INTO APACHE MESOS

深入浅出 Mesos（三）：持久化存储和容错

作者 韩陆

在深入浅出 Mesos 系列的第一篇文章中，我对相关的技术做了简要概述，在第二篇文章中，我深入介绍了 Mesos 的架构。完成第二篇文章之后，我本想开始着手写一篇 Mesos 如何处理资源分配的文章。不过，我收到一些读者的反馈，于是决定在谈资源分配之前，先完成这篇关于 Mesos 持久化存储和容错的文章。

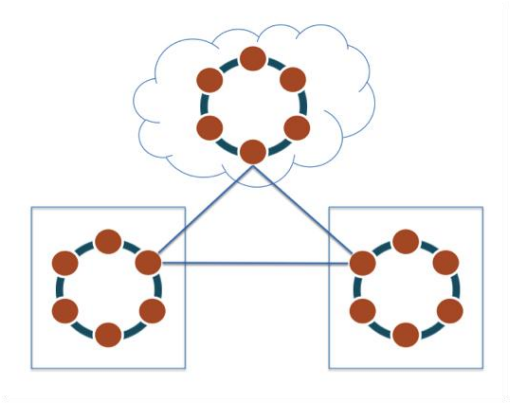
持久化存储的问题



正如我在前文中讨论过的，使用 Mesos 的主要好处是可以在同一组计算节点集合上运行多种类型的应用程序（调度以及通过 Framework 初始化任务）。这些任务使用隔离模块（目前是某些类型的容器技术）从实际节点中抽象出来，以便它们可以根据需要在不同的节点上移动和重新启动。

由此我们会思考一个问题，Mesos 是如何处理持久化存储的呢？如果我在运行一个数据库作业，Mesos 如何确保当任务被调度时，分配的节点可以访问其所需的数据？如图所示，在 Hindman 的示例中，使用 Hadoop 文件系统（HDFS）作为 Mesos 的持久层，这是 HDFS 常见的使用方式，也是 Mesos 的执行器传递分配指定任务的配置数据给 Slave 经常使用的方式。实际上，Mesos 的持久化存储可以使用多种类型的文件系统，HDFS 只是其中之一，但也是 Mesos 最经常使用的，它使得 Mesos 具备了与高性能计算的亲缘关系。其实 Mesos 可以有多种选择来处理持久化存储的问题：

- **分布式文件系统**。如上所述，Mesos 可以使用 DFS（比如 HDFS 或者 Lustre）来保证数据可以被 Mesos 集群中的每个节点访问。这种方式的缺点是会有网络延迟，对于某些应用程序来说，这样的网络文件系统或许并不适合。
- **使用数据存储复制的本地文件系统**。另一种方法是利用应用程序级别的复制来确保数据可被多个节点访问。提供数据存储复制的应用程序可以是 NoSQL 数据库，比如 Cassandra 和 MongoDB。这种方式的优点是不再需要考虑网络延迟问题。缺点是必须配置 Mesos，使特定的任务只运行在持有复制数据的节点上，因为你不会希望数据中心的所有节点都复制相同的数据。为此，可以使用一个 Framework，静态地为其预留特定的节点作为复制数据的存储。



- **不使用复制的本地文件系统**。也可以将持久化数据存储在指定节点的文件系统上，并且将该节点预留给指定的应用程序。和前面的选择一样，可以静态地为指定应用程序预留节点，但此时只能预留给单个节点而不是节点集合。后面两种显然不是理想的选择，因为实质上都需要创建静态分区。然而，在不允许延时或者应用程序不能复制它的数据存储等特殊情况下，我们需要这样的选择。

Mesos 项目还在发展中，它会定期增加新功能。现在我已经发现了两个可以帮助解决持久化存储问题的新特性：

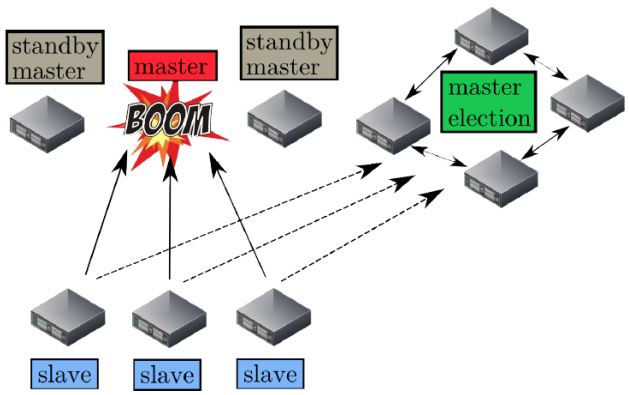
- **动态预留**。Framework 可以使用这个功能框架保留指定的资源，比如持久化存储，以便在需要启动另一个任务时，资源邀约只会发送给那个 Framework。这可以在单节点和节点集合中结合使用 Framework 配置，访问永久化数据存储。关于这个建议的功能的更多信息可以从此处获得。

- **持久化卷**。该功能可以创建一个卷，作为 Slave 节点上任务的一部分被启动，即使在任务完成后其持久化依然存在。Mesos 为需要访问相同的数据后续任务，提供在可以访问该持久化卷的节点集合上相同的 Framework 来初始化。关于这个建议的功能的更多信息可以从此处获得。

容错

接下来，我们来谈谈 Mesos 在其协议栈上是如何提供容错能力的。恕我直言，Mesos 的优势之一便是将容错设计到架构之中，并以可扩展的分布式系统的方式来实现。

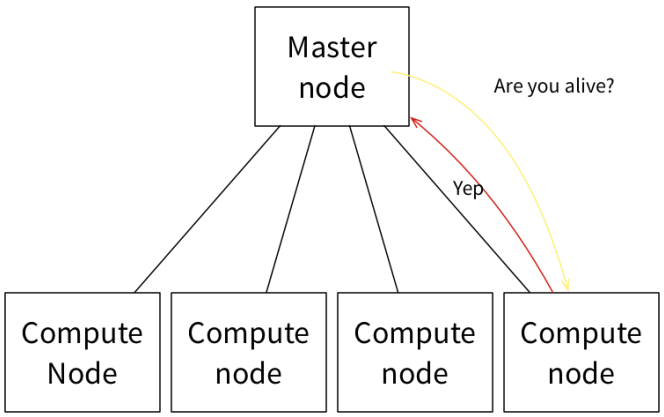
- **Master**。故障处理机制和特定的架构设计实现了 Master 的容错。



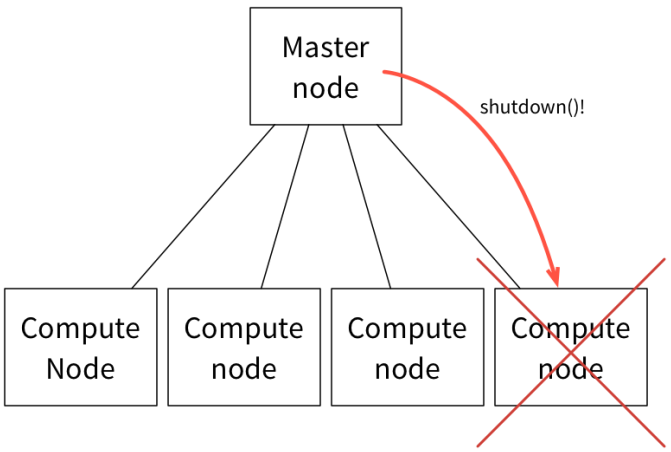
首先，Mesos 决定使用热备份（hot-standby）设计来实现 Master 节点集合。正如 Tomas Barton 对上图的说明，一个 Master 节点与多个备用（standby）节点运行在同一集群中，并由开源软件 Zookeeper 来监控。Zookeeper 会监控 Master 集群中所有的节点，并在 Master 节点发生故障时管理新 Master 的选举。建议的节点总数是 5 个，实际上，生产环境至少需要 3 个 Master 节点。 Mesos 决定将 Master 设计为持有软件状态，这意味着当 Master 节点发生故障时，其状态可以很快地在新选举的 Master 节点上重建。 Mesos 的状态信息实际上驻留在 Framework 调度器和 Slave 节点集合之中。当一个新的 Master 当选后，Zookeeper 会通知 Framework 和选举后的 Slave 节点集合，以便使其在新的 Master 上注册。彼时，新的 Master 可以根据 Framework 和 Slave 节点集合发送过来的信息，重建内部状态。

- **Framework 调度器**。Framework 调度器的容错是通过 Framework 将调度器注册 2 份或者更多份到 Master 来实现。当一个调度器发生故障时，Master 会通知另一个调度来接管。需要注意的是 Framework 自身负责实现调度器之间共享状态的机制。
- **Slave**。Mesos 实现了 Slave 的恢复功能，当 Slave 节点上的进程失败时，可以让执行器/任务继续运行，并为那个 Slave 进程重新连接那台 Slave 节点上运行的执行器/任务。当任务执行时，Slave 会将任务的监测点元数据存入本地磁盘。如果 Slave 进程失败，任务会继续运行，当 Master 重新启动 Slave 进程后，因为此时没有可以响应的消息，所以重新启动的 Slave 进程会使用检查点数据来恢复状态，并重新与执行器/任务连接。

  如下情况则截然不同，计算节点上 Slave 正常运行而任务执行失败。在此，Master 负责监控所有 Slave 节点的状态。



当计算节点/Slave 节点无法响应多个连续的消息后，Master 会从可用资源的列表中删除该节点，并会尝试关闭该节点。



然后，Master 会向分配任务的 Framework 调度器汇报执行器/任务失败，并允许调度器根据其配置策略做任务失败处理。通常情况下，Framework 会重新启动任务到新的 Slave 节点，假设它接收并接受来自 Master 的相应的资源邀约。

- **执行器/任务**。与计算节点/Slave 节点故障类似，Master 会向分配任务的 Framework 调度器汇报执行器/任务失败，并允许调度器根据其配置策略在任务失败时做出相应的处理。通常情况下，Framework 在接收并接受来自 Master 的相应的资源邀约后，会在新的 Slave 节点上重新启动任务。

结论

在接下来的文章中，我将更深入到资源分配模块。同时，我非常期待读者的反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我非全知，虚心求教，所以期待读者的校正和启示。我也会在 twitter 响应你的反馈，请关注 @hui_kenneth。

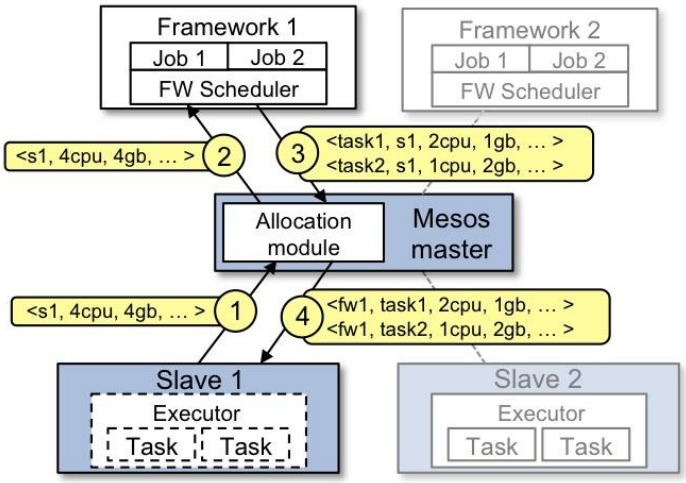
查看英文原文： DEALING WITH PERSISTENT STORAGE AND FAULT TOLERANCE IN APACHE MESOS

深入浅出 Mesos（四）：Mesos 的资源分配

作者 韩陆

Apache Mesos 能够成为最优秀的数据中心资源管理器的一个重要功能是面对各种类型的应用，它具备像交警一样的疏导能力。本文将深入 Mesos 的资源分配内部，探讨 Mesos 是如何根据客户应用需求，平衡公平资源共享的。在开始之前，如果读者还没有阅读这个系列的前序文章，建议首先阅读它们。第一篇是 Mesos 的概述，第二篇是两级架构的说明，第三篇是数据存储和容错。

我们将探讨 Mesos 的资源分配模块，看看它是如何确定将什么样的资源邀约发送给具体哪个 Framework，以及在必要时如何回收资源。让我们先来回顾一下 Mesos 的任务调度过程：



从前面提到的两级架构的说明一文中我们知道，Mesos Master 代理任务的调度首先从 Slave 节点收集有关可用资源的信息，然后以资源邀约的形式，将这些资源提供给注册其上的 Framework。

Framework 可以根据是否符合任务对资源的约束，选择接受或拒绝资源邀约。一旦资源邀约被接受，Framework 将与 Master 协作调度任务，并在数据中心的相应 Slave 节点上运行任务。

如何作出资源邀约的决定是由资源分配模块实现的，该模块存在于 Master 之中。资源分配模块确定 Framework 接受资源邀约的顺序，与此同时，确保在本性贪婪的 Framework 之间公平地共享资源。在同质环境中，比如 Hadoop 集群，使用最多的公平份额分配算法之一是最大最小公平算法（max-min fairness）。最大最小公平算法算法将最小的资源分配最大化，并将其提供给用户，确保每个用户都能获得公平的资源份额，以满足其需求所需的资源；一个简单的例子能够说明其工作原理，请参考最大最小公平份额算法页面的示例 1。如前所述，在同质环境下，这通常能够很好地运行。同质环境下的资源需求几乎没有波动，所涉及的资源类型包括 CPU、内存、网络带宽和 I/O。然而，在跨数据中心调度资源并且是异构的资源需求时，资源分配将会更加困难。例如，当用户 A 的每个任务需要 1 核 CPU、4GB 内存，而用户 B 的每个任务需要 3 核 CPU、1GB 内存时，如何提供合适的公平份额分配策略？当用户 A 的任务是内存密集型，而用户 B 的任务是 CPU 密集型时，如何公平地为其分配一揽子资源？

因为 Mesos 是专门管理异构环境中的资源，所以它实现了一个可插拔的资源分配模块架构，将特定部署最适合的分配策略和算法交给用户去实现。例如，用户可以实现加权的最大最小公平性算法，让指定的 Framework 相对于其它的 Framework 获得更多的资源。默认情况下，Mesos 包括一个严格优先级的资源分配模块和一个改良的公平份额资源分配模块。严格优先级模块实现的算法给定 Framework 的优先级，使其总是接收并接受足以满足其任务要求的资源邀约。这保证了关键应用在 Mesos 中限制动态资源份额上的开销，但是会潜在其他 Framework 饥饿的情况。

由于这些原因，大多数用户默认使用 DRF（主导资源公平算法 Dominant Resource Fairness），这是 Mesos 中更适合异质环境的改良公平份额算法。

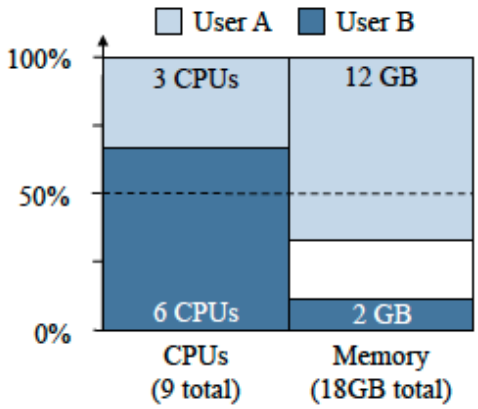
DRF 和 Mesos 一样出自 Berkeley AMPLab 团队，并且作为 Mesos 的默认资源分配策略实现编码。

读者可以从此处和此处阅读 DRF 的原始论文。在本文中，我将总结其中要点并提供一些例子，相信这样会更清晰地解读 DRF。让我们开始揭秘之旅。

DRF 的目标是确保每一个用户，即 Mesos 中的 Framework，在异质环境中能够接收到其最需资源的公平份额。为了掌握 DRF，我们需要了解主导资源（dominant resource）和主导份额（dominant share）的概念。Framework 的主导资源是其最需的资源类型（CPU、内存等），在资源邀约中以可用资源百分比的形式展示。例如，对于计算密集型的任务，它的 Framework 的主导资源是 CPU，而依赖于在内存中计算的任务，它的 Framework 的主导资源是内存。因为资源是分配给 Framework 的，所以 DRF 会跟踪每个 Framework 拥有的资源类型的份额百分比；Framework 拥有的全部资源类型份额中占最高百分比的就是 Framework 的主导份额。DRF 算法会使用所有已注册的 Framework 来计算主导份额，以确保每个 Framework 能接收到其主导资源的公平份额。

概念过于抽象了吧？让我们用一个例子来说明。假设我们有一个资源邀约，包含 9 核 CPU 和 18GB 的内存。Framework 1 运行任务需要（1 核 CPU、4GB 内存），Framework 2 运行任务需要（3 核 CPU、1GB 内存）Framework 1 的每个任务会消耗 CPU 总数的 1/9、内存总数的 2/9，因此 Framework 1 的主导资源是内存。同样，Framework 2 的每个任务会 CPU 总数的 1/3、内存总数的 1/18，因此 Framework 2 的主导资源是 CPU。DRF 会尝试为每个 Framework 提供等量的主导资源，作为他们的主导份额。在这个例子中，DRF 将协同 Framework 做如下分配：Framework 1 有三个任务，总分配为（3 核 CPU、12GB 内存），Framework 2 有两个任务，总分配为（6 核 CPU、2GB 内存）。

此时，每个 Framework 的主导资源（Framework 1 的内存和 Framework 2 的 CPU）最终得到相同的主导份额（2/3 或 67％），这样提供给两个 Framework 后，将没有足够的可用资源运行其他任务。需要注意的是，如果 Framework 1 中仅有两个任务需要被运行，那么 Framework 2 以及其他已注册的 Framework 将收到的所有剩余的资源。



那么，DRF 是怎样计算而产生上述结果的呢？如前所述，DRF 分配模块跟踪分配给每个 Framework 的资源和每个框架的主导份额。每次，DRF 以所有 Framework 中运行的任务中最低的主导份额作为资源邀约发送给 Framework。如果有足够的可用资源来运行它的任务，Framework 将接受这个邀约。通过前面引述的 DRF 论文中的示例，我们来贯穿 DRF 算法的每个步骤。为了简单起见，示例将不考虑短任务完成后，资源被释放回资源池中这一因素，我们假设每个 Framework 会有无限数量的任务要运行，并认为每个资源邀约都会被接受。

回顾上述示例，假设有一个资源邀约包含 9 核 CPU 和 18GB 内存。Framework 1 运行的任务需要（1 核 CPU、4GB 内存），Framework 2 运行的任务需要（3 核 CPU、2GB 内存）。Framework 1 的任务会消耗 CPU 总数的 1/9、内存总数的 2/9，Framework 1 的主导资源是内存。同样，Framework 2 的每个任务会 CPU 总数的 1/3、内存总数的 1/18，Framework 2 的主导资源是 CPU。

	Framwork 1			Framework 2			CPU	RAM
Framework Chosen	Resoure Shares	Dominant Share	Dominant Share %	Resoure Shares	Dominant Share	Dominant Share %	Total Allocation	Total Allocation
	0/9, 0/18	0	0%	0/9, 0/18	**0**	**0%**	0/9	0/18
Framework 2	0/9, 0/18	**0**	**0%**	3/9, 1/18	**1/3**	**33%**	3/9	1/18
Framework 1	1/9, 4/18	**2/9**	**22%**	3/9, 1/18	**1/3**	**33%**	4/9	5/18
Framework 1	2/9, 8/18	4/9	44%	3/9, 1/18	**1/3**	**33%**	5/9	9/18
Framework 2	2/9, 8/18	**4/9**	**44%**	6/9, 2/18	**2/3**	**67%**	8/9	10/18
Framework 1	3/9, 12/18	**2/3**	**67%**	6/9, 2/18	**2/3**	**67%**	9/9	14/18

上面表中的每一行提供了以下信息：

- Framework chosen——收到最新资源邀约的 Framework。
- Resource Shares——给定时间内 Framework 接受的资源总数，包括 CPU 和内存，以占资源总量的比例表示。
- Dominant Share（主导份额）——给定时间内 Framework 主导资源占总份额的比例，以占资源总量的比例表示。
- Dominant Share %（主导份额百分比）——给定时间内 Framework 主导资源占总份额的百分比，以占资源总量的百分比表示。
- CPU Total Allocation——给定时间内接受的所有 Framework 的总 CPU 资源。
- RAM Total Allocation——给定时间内接受的所有 Framework 的总内存资源。

注意，每个行中的最低主导份额以粗体字显示，以便查找。

最初，两个 Framework 的主导份额是 0%，我们假设 DRF 首先选择的是 Framework 2，当然我们也可以假设 Framework 1，但是最终的结果是一样的。

1. Framework 2 接收份额并运行任务，使其主导资源成为 CPU，主导份额增加至 33%。
2. 由于 Framework 1 的主导份额维持在 0%，它接收共享并运行任务，主导份额的主导资源（内存）增加至 22%。
3. 由于 Framework 1 仍具有较低的主导份额，它接收下一个共享并运行任务，增加其主导份额至 44%。
4. 然后 DRF 将资源邀约发送给 Framework 2，因为它现在拥有更低的主导份额。
5. 该过程继续进行，直到由于缺乏可用资源，不能运行新的任务。在这种情况下，CPU 资源已经饱和。
6. 然后该过程将使用一组新的资源邀约重复进行。

需要注意的是，可以创建一个资源分配模块，使用加权的 DRF 使其偏向某个 Framework 或某组 Framework。如前面所提到的，也可以创建一些自定义模块来提供组织特定的分配策略。

一般情况下，现在大多数的任务是短暂的，Mesos 能够等待任务完成并重新分配资源。然而，集群上也可以跑长时间运行的任务，这些任务用于处理挂起作业或行为不当的 Framework。

值得注意的是，在当资源释放的速度不够快的情况下，资源分配模块具有撤销任务的能力。Mesos 尝试如此撤销任务：向执行器发送请求结束指定的任务，并给出一个宽限期让执行器清理该任务。如果执行器不响应请求，分配模块就结束该执行器及其上的所有任务。

分配策略可以实现为，通过提供与 Framework 相关的保证配置，来阻止对指定任务的撤销。如果 Framework 低于保证配置，Mesos 将不能结束该 Framework 的任务。

我们还需了解更多关于 Mesos 资源分配的知识，但是我将戛然而止。接下来，我要说点不同的东西，是关于 Mesos 社区的。我相信这是一个值得考虑的重要话题，因为开源不仅包括技术，还包括社区。

说完社区，我将会写一些关于 Mesos 的安装和 Framework 的创建和使用的，逐步指导的教程。在一番实操教学的文章之后，我会回来做一些更深入的话题，比如 Framework 与 Master 是如何互动的，Mesos 如何跨多个数据中心工作等。

与往常一样，我鼓励读者提供反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我非全知，虚心求教，所以非常期待读者的校正和启示。我们也可以在 twitter 上沟通，请关注 @hui_kenneth。

查看英文原文： PLAYING TRAFFIC COP: RESOURCE ALLOCATION IN APACHE MESOS

深入浅出 Mesos（五）：
成功的开源社区

作者 韩陆

最近我一直在写 Apache Mesos 的系列文章，目前已经完成的内容如下：

- 深入浅出 Mesos（一）：为软件定义数据中心而生的操作系统
- 深入浅出 Mesos（二）：Mesos 的体系结构和工作流
- 深入浅出 Mesos（三）：持久化存储和容错
- 深入浅出 Mesos（四）：Mesos 的资源分配

包括技术考量在内，我同样对 Mesos 项目本身的进展颇为兴奋。所以，我想从以技术为重点的文章中走出，做些关于 Mesos 项目的总体观察。正如我此前在推文中所说的，我对 Mesos 一直颇具深刻印象的是它的三个特点：

1. 让人清楚地理解它的好处
2. 易于管控的作用域
3. 没有第二家厂商的实现

借此机会，我要说下近来大家对 Mesos 的认识，我发现人们已经非常容易掌握 Mesos 的概念，并了解其技术的价值。这对于正在发展并寻求扩大其覆盖面的项目来说是至关重要的。一个项目中的技术所带来的切实利益是非常重要的，它能让人心生向往并积极参与在社区中。正如本系列第二篇文章中所述，我看到了在效率、商业敏捷性和可扩展性等方面，Mesos 带给数据中心的很清晰的好处。随着分布式应用程序和微服务的流行，越来越多的用户正在寻找一种技术，以帮助他们管理这些复杂的应用程序。因此，我们看到越来越多的人在关注着 Mesos 项目和 Mesosphere，Mesosphere 是一家基于 Mesos 来构建商业产品的公司。

Mesos 项目的另一个重要优势是对其作用域的限制。Mesos 被设计成一个数据中心资源管理系统，Mesos 具备其主要功能，并避免超越设计理念的诱惑，至少在这之前，已经建立了一个坚实的基础。相信 Mesos 项目已完成了两件重要的事情，使 Mesos 不会过早迷失于作用域之外。

- 建立了坚实的基础——诱惑是永远存在的，新的技术总是会不断地增加新的功能。当功能驱动开发并以代码的稳定性为代价时，问题随之而来，特别是疏于确保新增加的模块不会破坏已有模块的时候。Mesos 项目已经为此做出了很好的工作，Mesos 关注于修复社区中报出的缺陷并加强现有功能，并不鼓励人们不断地追逐闪亮的新事物。
- 构建了强大的生态系统——为了专注于资源管理和控制 Mesos 架构的规模，该项目启用了插件化的 Framework 生态系统。在大多数情况下，Mesos 项目避免了为每个应用程序建立一个调度器或者严格限定一个隔离模块。这使得不同的社区可以参与其中，例如 Hadoop 社区和 Docker 社区都可以为 Mesos 开发插件。可以预见 Mesos 项目的好兆头，因为拥有一个强大的生态系统是其在软件领域成功的必要条件。

在做好培养一个强大生态系统的同时，Mesos 项目做到了避免让太多的厂商太早介入。相反，似乎有一个最终用户和厂商合作的极佳组合。这其中的主要原因是因为 Mesos 是为特定问题，提供解决方案的，而不是像 AWS 那样针对通用的问题。不管是什么原因，阻止大量厂商的介入以及该项目日趋成熟，使得 Mesos 社区的成长没有厂商政治干预、利益斗争，以及过度的商业诉求等包袱。我不是说这些挑战就没有，但 Mesos 至少不是一个基本上由厂商控制的项目，Mesos 可以以一个自然的步伐去成长。就像 Linux 项目，厂商的参与是以匹配客户的兴趣和使用，自然而然地发生的。

正如你所知道的，我很期待 Mesos 项目的未来，当更多的最终用户走进分布式系统的世界之时，希望可以看到 Mesos 在数据中心操作系统内核中发挥的价值。同时，我鼓励大家学习和参与进来。David Lester 在这篇采访中讲述了一些与此相关的方法，David Lester 是 Twitter 的工程师和开源倡导者。

本系列的后续文章将讲述如何搭建 Mesos 集群、如何为部署和管理应用程序，集成和编写 Framework。同时，我鼓励读者提供反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我非全知，虚心求教，所以期待读者的校正和启示。我也会在 twitter 响应你的反馈，请关注 @hui_kenneth。

查看英文原文： APACHE MESOS: OPEN SOURCE COMMUNITY DONE RIGHT

深入浅出 Mesos（六）：
亲身体会 Apache Mesos

作者 韩陆

关于下一代数据中心操作系统 Apache Mesos 的系列文章，已经完成的内容如下：

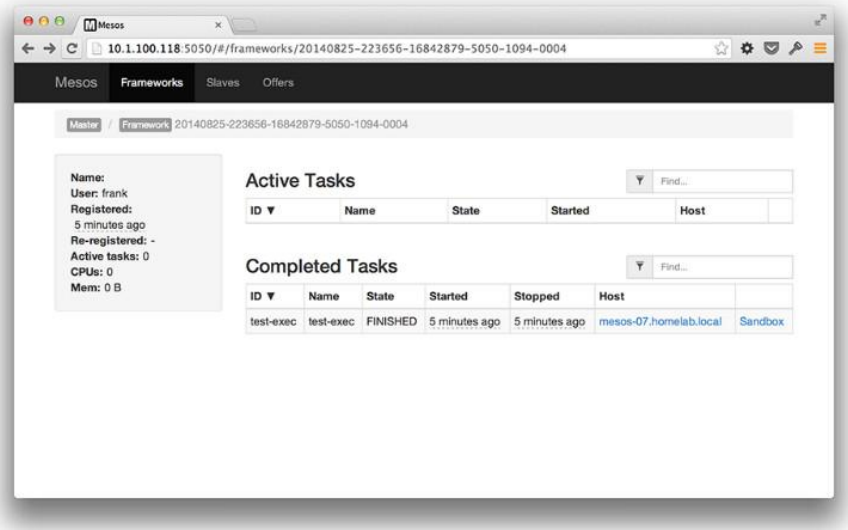
- 深入浅出 Mesos（一）：为软件定义数据中心而生的操作系统
- 深入浅出 Mesos（二）：Mesos 的体系结构和工作流
- 深入浅出 Mesos（三）：持久化存储和容错
- 深入浅出 Mesos（四）：Mesos 的资源分配
- 深入浅出 Mesos（五）：成功的开源社区

与本系列的前序文章相比，本文是一个短篇，因为这背后有充分的理由。我原本打算写篇更长的文章来详细说明如何在单个节点和多个节点上搭建 Mesos。不过，我很快就意识到，已经有一些非常聪明的家伙完成了相关的文章。因此，为了不重新发明轮子，我将在本文中描述并链接这些免费资源。

正在努力创建基于 Mesos 的商业化产品的 Mesosphere 同行们，制作了一个精简的 Mesos 在线课程，值得大家去研究一番。他们的网站也介绍了如何在物理服务器、个人笔记本电脑或 PC，以及一些公有 IaaS 云计算平台上安装 Mesos，请访问相关的链接。

到目前为止，我发现的讲述搭建 Mesos 测试环境最清楚的是，来自 Frank Hinek 的一系列文章。在 Frank 的博客中，我们可以获得搭建单节点 Mesos 环境的操作指南、搭建多节点 Mesos 环境的操作指南，以及在各种配置中搭建高可用的多节点环境的操作指南。需要说明的是，Frank 的这些操作指南是运行于 VMware 的 ESXi 虚拟机之上的；因此，他所述的安装工作流程，运行在 VirtualBox 上时，可能会遇到的一些奇怪的现象。还有其他一些类似的资源，很容易在互联网上搜索到。

上述的操作指南可以帮助我们创建测试环境，其中包括 Apache Mesos 和至少一种 Framework，比如 Apache Marathon。大多数的博客文章都会提供任务的示例，我们可以运行一下，来体验 Mesos 是如何运转的。



最后，正如我在本系列前序的文章中提到的，Mesos 的设计是成为真正的数据中心操作系统内核，Mesosphere 正在努力尝试封装一套围绕着 Mesos 的工具，构建一个全面的数据中心操作系统。Mesosphere 已经公布了他们的旗舰产品——数据中心操作系统（DCOS）的公开测试版。如果你有兴趣一试身手的话，可以注册一个试用帐号。

因此，尝试一下 Apache Mesos 还是很有乐趣的。接下来的一篇，我将会深入探讨 Mesos 中 Framework 的概念，并解释各种 Framework 是如何适应生

态系统的。随着 OpenStack 峰会即将到来，我也打算写些关于 Mesos 是如何与 IaaS 云计算平台，比如 OpenStack，以及 AW 一起玩的文章.....亦或作罢。

与往常一样，欢迎大家的反馈。

查看英文原文：TRYING OUT APACHE MESOS

Apple 使用 Apache Mesos 重建 Siri 后端服务

作者 Daniel Bryant　译者 韩陆

苹果公司宣布，将使用开源的集群管理软件 Apache Mesos，作为该公司广受欢迎的、基于 iOS 的智能个人助理软件 Siri 的后端服务。Mesosphere 的博客指出，苹果已经创建了一个命名为 J.A.R.V.I.S.，类似 PaaS 的专有调度 Framework，由此，开发者可以部署可伸缩和高可用的的 Siri 服务。

集群管理软件 Apache Mesos 将 CPU、内存、存储介质以及其它计算机资源从物理机或者虚拟机中抽象出来，构建支持容错和弹性的分布式系统，并提供高效的运行能力。Mesos 使用与 Linux 内核相同的系统构建原则，只是他们处在不同的抽象层次上。Mesos 内核运行在每台机器上，通过应用程序 Framework，提供跨整个数据中心和云环境进行资源管理和调度的 API。苹果已经创建了自己专有的调度 Framework 以运行 Siri 的后端服务，将其命名为 J.A.R.V.I.S.。

J.A.R.V.I.S.是“一种相当智能的调度器（Just A Rather Very Intelligent Scheduler）”的缩写，这个名字的灵感来自《钢铁侠》电影中的智能化计算机助手。苹果公司使用 J.A.R.V.I.S.作为内部的平台即服务（PaaS）系统，使开发者编写的 Siri 后端应用程序可以部署为可伸缩性和弹性的服务，用于响应 iOS 用户通过个人助理应用程序请求的语音查询。

据 Mesosphere 的博客报道，在苹果公司总部加州库比蒂诺的聚会上，苹果的开发者表示，他们的 Mesos 集群有数千个节点。支持 Siri 应用程序的后台系统包括约 100 种不同类型的服务，应用程序的数据存储在 Hadoop 分布式文件系统（HDFS）中。从基础设施的角度来看，使用 Mesos 有助于使 Siri 具备可伸缩性和可用性，并且还改善了 iOS 应用程序自身的延迟。

Mesos 后端是第三代 Siri 平台，告别了之前部署在“传统的”基础设施的历史。Mesosphere 博客认为，从概念上讲，苹果公司与 Mesos 的合作以及 J.A.R.V.I.S.类似于 Google 的 Borg 项目，领先于其他支持长时间运行应用服务的类 PaaS Framework，比如 Mesospere 数据中心操作系统（DCOS）的相关组件 Mesosphere Marathon 和出自 Twitter 基础设施团队的 Apache Aurora。

Mesosphere 高级研究分析师 Derrick Harris 在 Mesosphere 的博客中表示，关于 Siri 由 Apache Mesos 集群管理软件支撑的公告是对 Mesos 成熟度的证明：

> 苹果公司能够信任使用 Mesos 支撑 Siri——这是一个复杂的应用程序，用以处理只有苹果知道每天会有多少数量的、来自数以亿计的 iPhone 和 iPad 用户的语音查询—— 这足以说明 Mesos 的成熟度，Mesos 已经为各种类型的企业带来巨大影响做好了准备。

InfoQ 采访了 Mesosphere 高级副总裁 Matt Trifiro，并询问了这项公告对正在考虑部署应用到 Mesos 的企业和软件开发者会有什么影响：

InfoQ：为什么苹果的这项公告对 Mesos 和 Mesosphere 很重要？

> **Trifiro：**苹果公司宣布，他们完全重建了 Siri，以运行于 Mesos 之上。这再次表明，Mesosphere DCOS 中的分布式内核 Mesos，是编排大规模容器和构建新的分布式系统的黄金标准。

InfoQ：不是每家企业都能达到苹果公司的规模，那么传统企业怎样应用 Mesos 呢？

> **Trifiro：**像苹果和 Twitter 这样的公司，几乎全部的基础设施都使用了这项技术。因为 Siri 和 Twitter 都依赖于 Mesos，可想而知，它必须是可靠的。但是，开源的 Apache Mesos 是一项非常尖端的技术，通过开源工具手工装配，并将 Mesos 用于生产环境是非常困难的。这正是 Mesosphere 产生的原因。任何公司都能使用这项久经考验的技术，构建完整的数据中心操作系统（DCOS），并具备和 Twitter 或者苹果公司同等的能力和自动化效果，而不必成为 Twitter 或者苹果那样大规模的公司。

InfoQ：苹果公司从 Mesos API 直接实现了一套调度器（J.A.R.V.I.S.），这意味着什么呢？

Trifiro：Mesos 最强大的方面其一就是，它提供了用于构建新的分布式系统的基本功能。如果你去看其它的分布式系统，比如早于 Mesos 出现的 Hadoop，它有几十万行代码，很多地方是在重复制造轮子。所有的失败处理、网络实现、消息传递和资源分配的代码，开发者不应重写这些功能。而为程序员提供了内置这些功能的 Mesos 内核的话，他们就可以快速构建新的高可用性和弹性分布式系统，而无需重复所有基本的功能。他们可以专注于业务逻辑的实现上。

InfoQ：Mesos 和 Mesosphere DCOS 之间是什么关系？

Trifiro：Mesosphere DCOS 是一种新型的操作系统，跨越数据中心或云环境中的所有机器，将他们的资源放到一个资源池中，使他们的行为整体上像一个大的计算机。Apache 的开源项目 Mesos 是这个操作系统里面的内核。我们将其和其他组件包装到一起，包括初始化系统（marathon）、文件系统（HDFS）、应用打包和部署系统、图形用户界面和命令行界面（CLI）。所有这些组件一起构成了 DCOS。这就像苹果公司的 Yosemite 操作系统或者像 Android，他们各有一个内核（分别是 BSD 和 Linux），他们为内核添加了系统服务和工具，使内核成为值得笔记本电脑或者智能手机使用的产品。我们为数据中心所做的工作也是相同的。

更多关于苹果公司宣布使用 Mesos 作为 Siri 后端服务的消息，详见 Mesosphere 的博客。

查看英文原文：Apple Rebuilds Siri Backend Services Using Apache Mesos

Singularity：

基于 Apache Mesos 构建的服务部署和作业调度平台

作者 马德奎

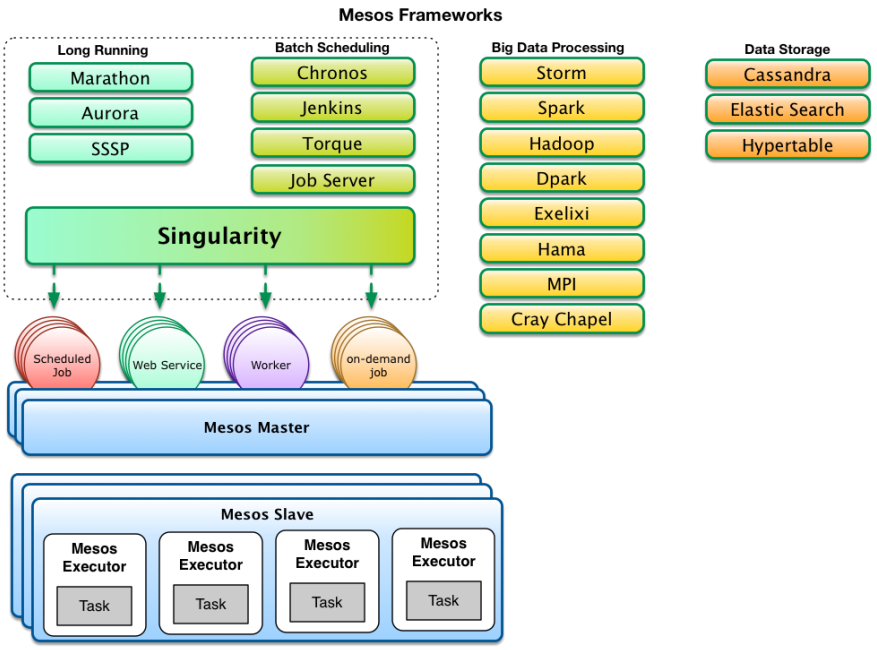
Singularity 是一个在云基础设施中部署和运行服务和计划作业的平台，同时也是 HubSpot PaaS 的核心组件。它能够高效地管理底层进程的生命周期，并有效地利用集群资源。它可以作为持续部署基础设施的基本组成部分，而且是微服务部署的理想选择。它不仅能够管理数以百计的服务器上运行着的数以千计的进程，而且还提供了如下开箱即用的特性：

- 丰富的 REST API，既有用于部署的，也有用于获取活动部署和历史部署信息的；
- Web 应用客户端（Singularity UI）使用上述 API 向用户提供所有可获得信息的友好视图；
- 部署失败时自动回滚；
- 服务本身或者服务器故障时自动实现服务的故障转移；
- 自动暂停反复失败的服务；
- 进程和服务端点级别的健康检查；
- 多实例服务的负载均衡；
- 日志循环和归档；
- 针对每个服务实例的资源限制和资源隔离，并能杀死超过限制的实例；
- “机架（Rack）”/可用区域识别。

在 Apache Mesos 的术语中，使用 Mesos API 在集群中调度任务的 Mesos 应用程序称为框架。Singularity 就是一个 Apache Mesos 框架，它作为一个任务调度器运行在 Mesos 集群之上，如下图所示。

从中可以看出，Singularity 在一个框架中融合了长期运行任务的调度功能和批处理作业的调度功能，可以支持开发人员需要每天部署的许多常见进程类型，包括：

- Web 服务：长期运行的进程；
- 工作进程：长期运行的进程，类似 Web 服务，但不暴露 API，比如 Queue Consumer 就是一种常见的工作进程类型；
- 计划作业：周期性运行的任务；
- 按需执行的进程：需要手动执行的进程。



Mesos 框架有两个主要组件：调度器组件和执行器组件，前者注册到 Mesos 主进程用于分配资源，后者由 Mesos 从属进程在集群从节点上启动并运行框架任务。Mesos 主进程决定为每个框架分配多少资源，框架调度器选取提供的部分资源用于运行所需的任务。Mesos 从属进程并不直接运行任务，而是委派给合适的执行器来运行。Singularity 实现了这两个基本的框架组件，并且还提供了日志查看器、S3 上传器、执行器清理、OOM 进程清除、Singularity UI、Singularity Java Client 等组件。

此外，借助请求对象和部署对象，Singularity 在 Mesos 任务之上提供了一个面向部署的层。其中，请求对象定义一个可部署项，而部署对象定义一个可部署项的执行参数。要想了解所有 Singularity 端点及相应请求和响应对象的完整描述，请查看 Singularity API 参考。

最后，对 Singularity 感兴趣的读者可以查看针对测试和开发的本地安装文档以及 Singularity 部署示例，以了解更多的细节。还有一点不得不提一下，就是 Singularity 的路线上图上有支持 Docker 容器部署一项。

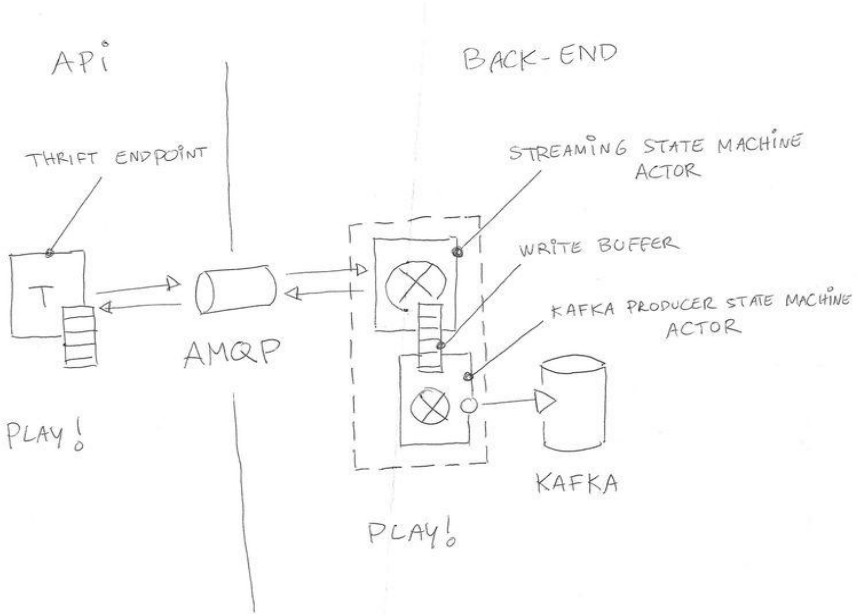
Autodesk 基于 Mesos 的可扩展事件系统

作者 Olivier Paugam，译者 韩陆

几个月前，我接到一个任务，要拿出一个集中式的事件系统，可以让我们的各种后端组件相互通信。我们讨论了后端活动流、渲染、数据转换、建筑信息模型（BIM）、个性身份（identity）、日志报表、分析等等。找寻其中是否有真正通用的可变负载、使用模式和扩展性配置。或许是其他的某样东西，能使我们的开发团队轻松接口。当然，系统的每个部分都应该具备自我扩展的能力。

由于我没有时间写太多的代码，所以我选择了 Kafka 作为我们的存储核心，因为它稳定且被广泛使用，而且表现良好（请注意，我没有说非得用它不可，可以用其他东西替代）。当然，现在我还不能直接将 Kafka 暴露出去，得在前端用一些 API 去实现这些。没想太多，我就放弃了在我的后端管理偏移量的想法，因为这在为实例处理失败时，会给人带来太多的约束。

所以我的结论就是，将系统分为独立的两层：**第一层是 API 层，负责处理请求流（incoming traffic）；第二层是后端层，负责处理长期的、有状态的、与 Kafka 交互的流处理进程**（比如，实现生产者和消费者）。这两层都可以独立扩展，只要求它们之间保持一致的路由，以保证客户端可以与相同的后端流处理进程，保持通信。



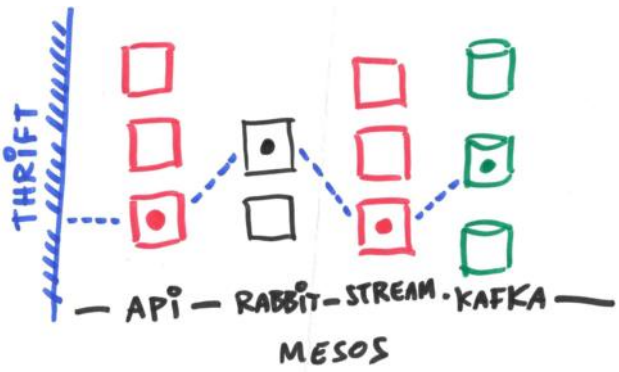
在此，哥文艺地展示下系统发布后的模样。

两层代码均 100%使用 **Scala** 实现，并使用 Play!框架。同时，它们重度依赖 **Akka actor 系统**（每个节点通常跑数百个 actor）。后端层实现了一组自定义的 Kafka 生产者和消费者，并使用一组专用的 actor 来管理预读与写缓冲区。一切都被实现为嵌套的有限状态机（我超爱这个概念）。分析数据存储到 **Splunk**，同时，度量数据存储到 **Librato**（collectd 是在容器中运行的）。

如何实现两层之间的路由呢？只需使用 RabbitMQ 即可，它有很好的持久性和弹性，甚至没什么趣味可言。AMQP 队列是实现这一简单的“电话交换机”模式的伟大方法。通过使用一些逻辑分片技术（比如对一些存在于每个事务中的 cookie 值进行散列或者类似的技术）将一组固定的后端节点关联到某个 RabbitMQ broker，就能够让路由支持扩展，这简直是小事一桩。

为什么我没有对 RabbitMQ 的 broker 做集群？哇哦，我能说我懒吗，关键是我真心觉得没啥必要。**在每个 broker 之间分流**实际上是为了高效，而在我看来，更重要的是以更轻松的方式去控制。为此付出的额外工作相比收成毫无意义。

所以，总之，给定容器拓扑，我们的请求会保持一条特定路径，这条路径依赖于后端节点所处理的流会话。**我们所需的每一层可独立扩展远比整体可扩展重要。**唯一的实际限制将是虚拟网络适配器和带宽。



虚线表示给定的会话将保持的特定路径。

现在，我们开始有意思的部分吧：**如何确保可靠的通信，避免拜占庭式的失败呢？**我要说，这非常简单，只需使用简单的两阶段提交协议，在客户端和后端同时建立状态机镜像模型（例如，他们总是同步的）。读写操作需要显式的确认请求。当你尝试读取并且失败的时候，你只需重新尝试，直到获得确认，然后这个确认将改变后端的状态（比如，向前移动 Kafka 的偏移量或者下发定时任务将事件发布）。所以我的客户端和后端之间的传输流实际上是这样的：“allocate session”、“read”、“ack”、“read”、“ack”.....“dispose”。

这一系统的巨大优势是有效地实现了操作幂等，而且没有任何烦人的声明语句，就可以编码状态机中的所有逻辑（我告诫自己：我要提供一个纯粹的功能实现，没别的，只为了炫技）。应对任何网络故障，当然是优雅地重新尝试。通过这种方式，还可以得到自由的控制流和背压技术。

这个系统的 API 会暴露为 Apache Thrift（当前通过 HTTPS 协议使用压缩技术工作，计划在某个时间点迁移到普通的 TCP 层）。我提供了 Python、Scala、.NET 和 Ruby 语言的客户端 SDK，以应对我们在 Autodesk 所使用的、花样百出的技术）。请注意，Kafka 偏移量是由客户端管理的（虽然这样不透明），这使得后端控制更简单。

稍等，我听见你说，**如何处理后端节点宕掉的情况？**哇哦，因为我们有两阶段协议，所以我们在读取数据的时候并不会真的获取：如果客户端不断地失败，将使用当前的偏移量，重新分配一个新的流会话。当向 Kafka 写入数据时，麻烦来了，因为这个过程是异步并且潜在地受下行流背压（比如你的节点失败了并且 Kafka broker 也出现问题了）。我会将后端节点配置为正常关机，在等待任何写挂起时，传入的任何请求都会快速失败。最后一招，我们甚至可以将任何待处理的数据刷新到磁盘上（之后，重新将数据注入）。

稍等，我听见你又说，**如果基础设施的一部分挂掉会怎样？**同样的。与流会话处理的实际后端节点之间的通信中断，当然会慢下来，但不会再有什么讨厌的影响，这又要归功于两阶段协议喽。

哦，我忘了说，在将数据写入 Kafka 日志前是**自动加密**的（使用 AES 256）。Kafka 的生产者和消费者使用同一套密钥共享。在涉密的主题上，我可以让流会话通过 OAUTH2 来验证，对每个请求验证 MD5-HMAC，走 TLS 到后端集群。

现在你又问，那么，我们如何完整地部署这个时尚最时尚的系统呢？嗯，我们将整套系统 100%地运行在一个普通的 Mesos/Marathon 集群上（现在用的不是 DCOS 哦，但我们可以切过去，因为这样可以用上他们超赞的仪表盘）集群**托管在 AWS EC2 上**，我们基本上整体复用了一些 c3.2xlarge 实例（在给定的区域内，10 到 20 个小型部署足够了）。请注意，我们可以在 Kubernetes 上（无论是 EC2 还是 GCE）做同样的事情。

小秀一下我们技术栈的结构。

每处部署都使用了我们的开源技术 Ochopod（自集群容器）。操作被降低到绝对最小值。例如，我们要做一次优雅的构建推送，就拿 API 层来说，无非是分配一些新的容器，等待他们启动好，然后逐步替换旧部署，这样就可以了。所有这一切都是通过集群中的一个专用的 Jenkins slave（它本身也是一个 Ochopod 容器）完成的！

实际上，我开发了一个叫 Ochothon 的迷你 PaaS 系统，只是为了能够快速开发/运维所有的容器。Ochothon CLI 显示了我的一个预发布集群的情况。

```
[redacted].150 > grep
<*> -> 100% replies (38 pods total) ->

pod                                                 | pod IP          | node        | process | state
                                                    |                 |             |         |
cse.dev.us-west-2.haproxy #0                        | 10[redacted]38  | i-a[redacted]59 | running | leader
cse.dev.us-west-2.kafka #0                          | 10[redacted]38  | i-a[redacted]59 | running | leader
cse.dev.us-west-2.play-api #0                       | 10[redacted]26  | i-e[redacted]25 | running | leader
cse.dev.us-west-2.play-plugs #0                     | 10[redacted]254 | i-1[redacted]1f | running | leader
cse.dev.us-west-2.rabbitmq #0                       | 10[redacted]9   | i-0[redacted]ce | running | leader
cse.dev.us-west-2.redis #0                          | 10[redacted]5   | i-b[redacted]7f | running | leader
cse.dev.us-west-2.replication.coordinator #1        | 10[redacted]5   | i-b[redacted]7f | running | leader
cse.dev.us-west-2.replication.haproxy #0            | 10[redacted]9   | i-0[redacted]ce | running | leader
cse.dev.us-west-2.replication.redis #0              | 10[redacted]203 | i-a[redacted]57 | running | leader
cse.dev.us-west-2.replication.replicator #1         | 10[redacted]7   | i-0[redacted]c0 | running | leader
cse.dev.us-west-2.zookeeper #0                      | 10[redacted]70  | i-9[redacted]56 | running | leader
cse.dev.us-west-2.zookeeper #1                      | 10[redacted]208 | i-8[redacted]42 | running | follower
cse.dev.us-west-2.zookeeper #2                      | 10[redacted]38  | i-a[redacted]59 | running | follower
cse.staging.us-west-2.haproxy #0                    | 10[redacted]77  | i-b[redacted]71 | running | leader
cse.staging.us-west-2.kafka #0                      | 10[redacted]5   | i-b[redacted]7f | running | leader
cse.staging.us-west-2.kafka #1                      | 10[redacted]9   | i-0[redacted]ce | running | follower
cse.staging.us-west-2.kafka #2                      | 10[redacted]77  | i-b[redacted]71 | running | follower
cse.staging.us-west-2.kafka #3                      | 10[redacted]26  | i-e[redacted]25 | running | follower
cse.staging.us-west-2.kafka #4                      | 10[redacted]254 | i-1[redacted]1f | running | follower
cse.staging.us-west-2.kafka #5                      | 10[redacted]21  | i-d[redacted]15 | running | follower
cse.staging.us-west-2.play-api #0                   | 10[redacted]7   | i-0[redacted]c0 | running | leader
cse.staging.us-west-2.play-api #1                   | 10[redacted]203 | i-a[redacted]57 | running | follower
cse.staging.us-west-2.play-api #2                   | 10[redacted]21  | i-d[redacted]15 | running | follower
cse.staging.us-west-2.play-plugs #0                 | 10[redacted]7   | i-0[redacted]c0 | running | leader
cse.staging.us-west-2.play-plugs #1                 | 10[redacted]48  | i-3[redacted]f6 | running | follower
cse.staging.us-west-2.play-plugs #2                 | 10[redacted]203 | i-a[redacted]57 | running | follower
cse.staging.us-west-2.rabbitmq #0                   | 10[redacted]5   | i-b[redacted]7f | running | leader
cse.staging.us-west-2.redis #0                      | 10[redacted]77  | i-b[redacted]71 | running | leader
cse.staging.us-west-2.replication.coordinator #2    | 10[redacted]7   | i-0[redacted]c0 | running | leader
cse.staging.us-west-2.replication.haproxy #0        | 10[redacted]70  | i-9[redacted]56 | running | leader
cse.staging.us-west-2.replication.redis #0          | 10[redacted]21  | i-d[redacted]15 | running | leader
cse.staging.us-west-2.replication.replicator #2     | 10[redacted]7   | i-0[redacted]c0 | running | leader
cse.staging.us-west-2.replication.replicator #3     | 10[redacted]26  | i-e[redacted]25 | running | follower
cse.staging.us-west-2.zookeeper #0                  | 10[redacted]5   | i-b[redacted]7f | running | leader
cse.staging.us-west-2.zookeeper #1                  | 10[redacted]9   | i-0[redacted]ce | running | follower
cse.staging.us-west-2.zookeeper #2                  | 10[redacted]77  | i-b[redacted]71 | running | follower
```

为了能让你感受 Ocho-*平台是多么好用，我这样说吧，**一个人（比如我）就能够管理跨 2 个区域的 5 个部署系统，包括所有的基础设施副本**……此外，还能有时间写写博客、码码程序！

所以，总体来说，此间的设计和编码，整件事情都充满乐趣，再加上它现在已经在生产运行，是我们云基础架构任务的关键部分（这是一个还不错的打赏）。如果你想对这个奇葩的流系统有更多的了解，告诉我们哦！

相关文章

- Deploying our eventing infrastructure over Mesos/Marathon & Ochothon !
- Ochopod + Kubernetes = Ochonetes
- Return of the finite state machine!

查看英文原文：How Autodesk Implemented Scalable Eventing Over Mesos

Myriad 项目：Mesos 和 YARN 协同工作

作者 Boris Lublinsky，译者 楚晗

Jim Scott 发表了新文章《两个集群的故事：Mesos 和 YARN》，它从当今许多 IT 采购中一个相当常见的情况开始讲述，即多资源孤岛：

> 第一个集群是 Apache Hadoop，它相当于一个岛，它的资源全部用于 Hadoop 平台和它的进程。第二个集群就是除了 Hadoop 之外其它所有资源所在的集群。

之所以产生这种情况是由于 Hadoop 是用 Apache YARN 来管理自己的资源，而尽管 YARN 在 Hadoop 集群中工作的不错，但对于非大数据的应用 YARN 就显得不那么适用了。

就像 Scott 在他文章中说得那样，这里的问题在于 YARN 所实现的调度方法：

> 当一个作业请求提交到 YARN 的资源管理器，YARN 会对可用的资源进行评估，并放置作业到相应的位置。这是一个作业应该去哪儿的决定……，YARN 针对 Hadoop 中的作业调度进行了优化，从历史或者典型场景的角度来说，这种优化都是针对长时间运行的批处理作业。这就意味着 YARN 并不是为长时间服务，或者短生命周期的交互式查询来设计的……，虽然有可能让 YARN 去调度其它这些工作负载，但显然这不是个理想的模型。

另一个不同的调度模型就是 Apache Mesos，它：

> ……利用了两级调度机制，即资源的请求和提供是针对框架（framework）而不是作业，可以把框架视为在 Mesos 上面运行的应用。Mesos 的主节点决策提供给每个框架多少资源，每个框架接着决策它能接受的资源申请以及哪种应用可以在这些资源上运行。当集群中的节点共享多个框架时，这种资源分配方法可以获得近似最佳的数据本地化（data locality）。

现实情况中，Mesos 和 YARN 在 IT 基础设施中都占有重要的位置。但是像 Scott 所讲述的，当你把它们两个背靠背使用时就会导致资源分裂。

> 在相同的数据中心同时使用 Mesos 和 YARN 两个资源管理器会带来益处，但目前需要你创建两个不同的静态分区。这也意味着某些资源会专属于 Hadoop，需要用 YARN 来管理，而其余的则是用 Mesos。

如同 Scott 所说，eBay、MapR 和 Mesosphere 合作了一个新项目，被称作 Myriad，它可以让 YARN 和 Mesos 和谐的工作，而这会给企业和数据中心带来好处。

> 这个开源项目是 Mesos 框架和 YARN 调度器的结合，它使得 Mesos 可以管理 YARN 的资源请求。当 YARN 中有作业请求资源时，YARN 的资源管理器会先通过 Myriad 的调度器来调度，这样就可以和 Mesos 的资源申请和提供匹配起来。Mesos Master 接下来会把调度请求发给 Mesos 的工作节点（Mesos Slave）。Mesos 的工作节点会和 Myriad 的执行器（executor）进行通信并发送请求，Myriad 执行器的作用是运行 YARN 的节点管理器（Node Manager）。当 Myriad 在 Mesos 分配的资源上加载 YARN 节点管理器后，YARN 节点管理器就会和 YARN 的资源管理器通信来确定作业可用的资源。YARN 可以以自己认为适合的方法来使用资源，Myriad 则在 Mesos 可用的资源池和 YARN 的有资源需求的任务间提供了无缝的桥梁。

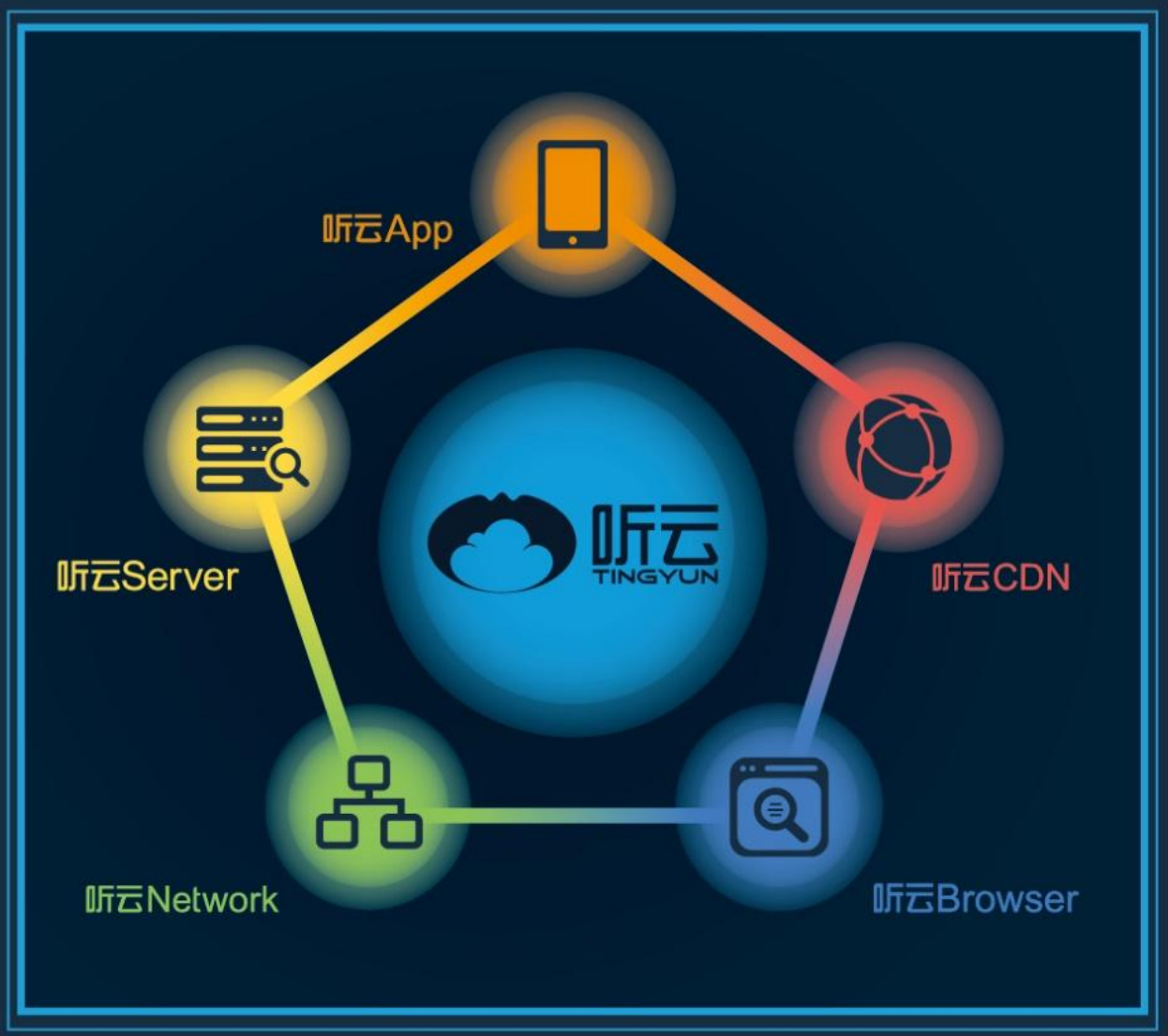
Myriad 使得在使用 Mesos 时，资源利用和跨数据中心的资源管理得以统一。在这种情况下，YARN 的工作负载是运行在共享的集群上，相比独立的 YARN 集群来说，更加动态和弹性。这个方法也使得数据中心维护团队可以扩展其资源以供给 YARN（或者，从 YARN 拿走）而无须去重新配置集群。

查看英文原文：Project Myriad: Mesos and YARN Working Together

AWS Summit

AWS技术峰会2015 · 上海

全业务应用性能管理解决方案



12大行业
72个子行业

30,000+ SaaS平台用户

2,800+ 企业级客户

每款应用都需要听云

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com