

MESOS IN ACTION



MANNING

Table of Contents

前言	1.1
Mesos基础篇	1.2
Mesos概述	1.2.1
搭建一个Mesos集群	1.2.2
Mesos工作原理浅析	1.2.3
总结	1.2.4
Mesos框架篇	1.3
Mesos框架概览	1.3.1
Marathon	1.3.2
Marathon简介	1.3.2.1
搭建Marathon服务	1.3.2.2
运行Marathon任务	1.3.2.3
小结	1.3.2.4
Chronos	1.3.3
Chronos简介	1.3.3.1
搭建Chronos服务	1.3.3.2
运行Chronos任务	1.3.3.3
小结	1.3.3.4
Spark	1.3.4
Spark简介	1.3.4.1
搭建Spark服务	1.3.4.2
运行Spark任务	1.3.4.3
小结	1.3.4.4
总结	1.3.5
Mesos实战篇	1.4
Mesos搭建日志处理系统实战	1.4.1
Mesos搭建企业级容器云实战	1.4.2
Mesos搭建企业级容器云项目概述	1.4.2.1
docker基础知识介绍	1.4.2.2
Marathon Framework 介绍	1.4.2.3

容器云平台基础概述	1.4.2.4
容器云搭建	1.4.2.5
关于容器云平台的其他一些杂谈	1.4.2.6
Mesos搭建大数据平台Hadoop和深度机器学习平台Singa实战	1.4.3
Mesos搭建分布式生物信息算法计算系统实战	1.4.4
Mesos搭建视频压缩批处理系统实战	1.4.5
7 Mesos搭建持续集成系统实战	1.4.6
7.1 Mesos 搭建持续集成系统概述	1.4.6.1
7.2 持续集成概念介绍	1.4.6.2
7.3 Jenkins 开源软件介绍	1.4.6.3
7.4 持续集成系统搭建	1.4.6.4
7.5 持续集成系统的维护心得	1.4.6.5
7.6 参考	1.4.6.6

Mesos 实战

Apache Mesos 是 Apache 软件基金会下属的顶级开源项目，它是目前云计算开源领域关于分布式计算方面最具生产价值的项目。在它出师不久就已经在 Twitter、AirBnb、Apple、Mesosphere 等公司大显身手，目前被全球关注云计算的公司作为重点应用的对象之一。

计划和进度

- ~~2015-11-15 完成初稿~~ → 第一次延期
- ~~先暂定 2015-12-15 完成初稿~~ 第二次延期
- ~~2015-1-15 完成初稿~~
- ~~2016-3-23 到 2016-4-23 肖德时 统稿写~~
- 2016-4-23 到 2016-6-23 肖德时 涉及的细节都需要确认。计划使用 Google Doc 做协作编辑

作者列表

- 肖德时 - 数人科技(xiaods@gmail.com)
- 陈显鹭 - 灵雀云开发工程师(xianlubird@gmail.com)
- 徐磊 - 去哪儿系统开发工程师(49068995@qq.com)
- 赵英俊 - 城云科技（杭州）有限公司(zyj@citycloud.com.cn)
- 周伟涛 - 数人科技(zhouwtlord@gmail.com)
- 杨成伟 - 爱奇艺(me@chengweiyang.cn)

章节计划进度：

- 前言 - 责任人：肖德时 初稿-WIP
- 第1章 Mesos 概述 - 责任人：肖德时 初稿-WIP
- 第2章 Mesos 基础 - 责任人：杨成伟 初稿 - Working In Progress(70%)
- 第3章 Mesos 框架 - 责任人：杨成伟 初稿 - Working In Progress(60%)
- 第4章 Mesos 搭建日志处理系统实战 - 责任人：徐磊 初稿-DONE
- 第5章 Mesos 搭建企业级容器云实战 - 责任人：陈显鹭 初稿 -DONE
- 第6章 Mesos 搭建大数据平台 Hadoop 和深度机器学习平台 Singa - 责任人：赵英俊 初稿 -DONE
- 第7章 搭建持续集成 - 责任人：周伟涛 初稿-DONE

- 章节优化完毕

审核组

- 李颖杰@DockOne
- 张春雨-电子工业出版社 责任编辑
- 王璞-数人科技CEO

Mesos 基础

本章将对 Mesos 的方方面面进行介绍，以便读者对 Mesos 项目能有比较全面的了解，包括：Mesos 历史、生态、能够做什么、怎样搭建一个 Mesos 集群以及 Mesos 工作原理

Mesos概述

Mesos，一个 Apache 开源项目，致力于构建数据中心级别的集群管理系统。它通过虚拟化物理主机（或者虚拟主机）之上的 CPU、内存、磁盘以及其它计算资源，构建高容错的、可动态伸缩的分布式系统，让运行计算任务快捷有效。**Mesos** 帮助开发者可以在一个统一的资源池之上运行应用，简化了构建分布式系统的复杂度。为了应对企业应用的复杂需求，开发者可以把各种优秀的计算框架安装到 **Mesos** 集群中，通过统一的 API 接口来调用管理，从而让弹性扩展的能力在整个数据中心甚至云计算环境中可以被快速的利用起来。**Mesos** 就像 Linux 中的 Kernel，已经成为构建分布式系统的Kernel。

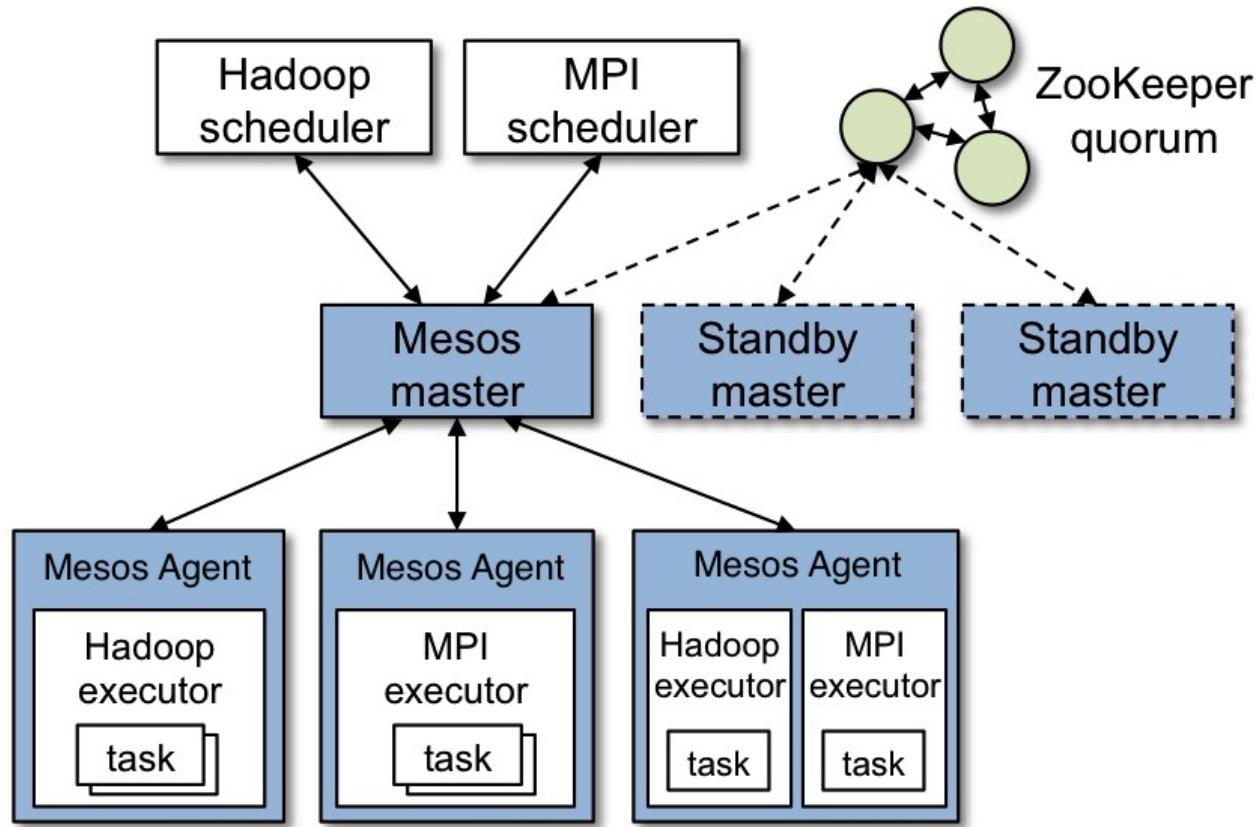
1.1 Mesos 简介

Apache Mesos是由美国伯克利大学(UCB)的AMPLab研发并贡献到 Apache 基金会的一款开源群集管理系统，支持 Hadoop、ElasticSearch、Spark、Storm 和 Kafka 等应用架构。

Mesos特性如下：

- 弹性扩展支持10,000个计算节点
- 使用ZooKeeper 实现 Master 和 Slave 的多容错副本技术
- 支持 Docker 容器技术
- 支持原生的 Linux 容器技术隔离任务
- 基于多资源调度，包括内存，CPU、磁盘、端口
- 提供 Java，Python，C++等多种语言 API接口开发新的分布式应用
- 提供 Web UI 界面查看集群状态

Apache Mesos 架构图如下：

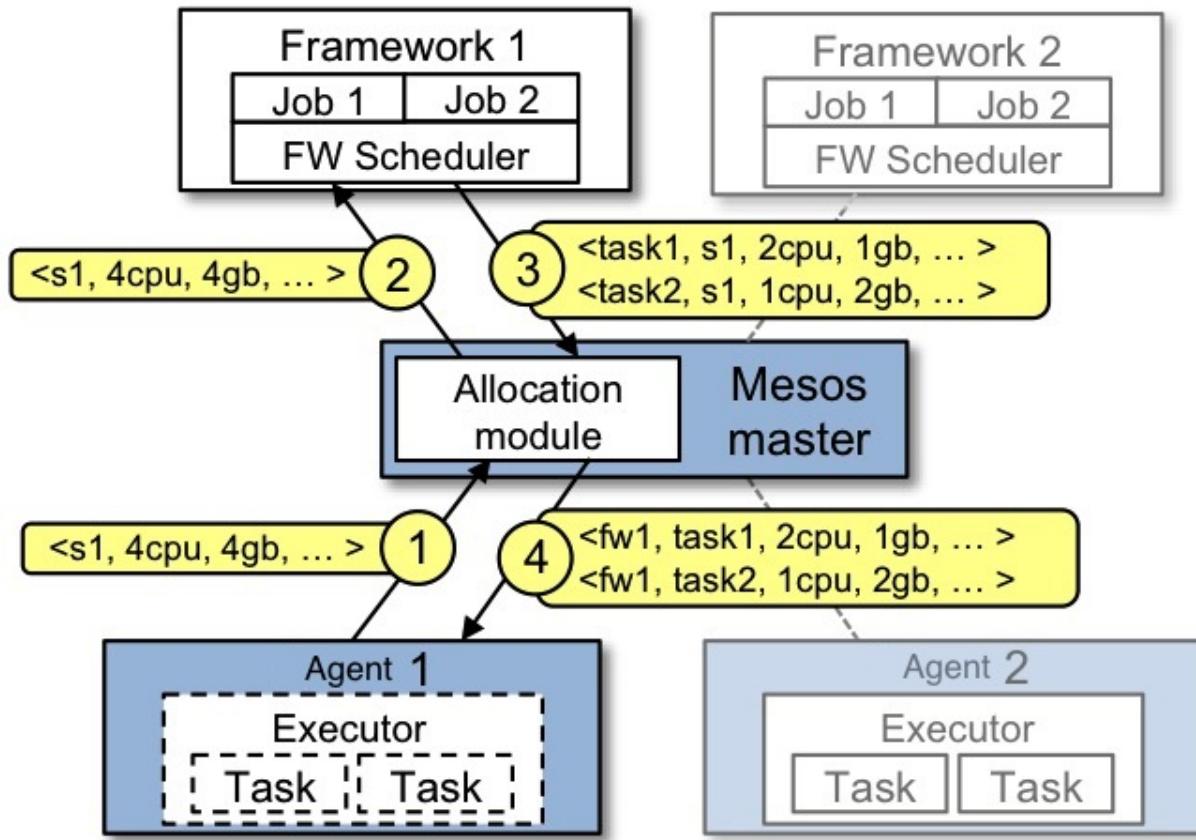


通过以上架构图，我们可以了解到：

- Mesos本身包含两个组件:Master Daemon和Slave Daemon。
 - Master Daemon
 - 管理所有的 Slave Daemon。
 - 用 [Resource Offers](#) 实现跨应用细粒度资源共享，如 cpu、内存、磁盘、网络等。
 - 限制和提供资源给应用框架使用。
 - 使用可拔插的模块化的架构，方便增加新的策略控制机制。
 - Slave Daemon
 - 负责接收和管理 Master 发来的请求 Task
 - 支持使用各种环境运行各种 Task，如 Docker、VM、进程调度(纯硬件)。
- Mesos上的task由2个组件管理:调度器(Scheduler)和执行进程(Executor Process)
 - 调度器(Scheduler)
 - 调度器通过注册 Mesos Master 获得集群资源调度权限
 - 调度器可通过 MesosSchedule Driver 接口和 Mesos Master 交互
 - 执行进程(Executor Process)
 - 用于启动框架内部的 Task
 - 不同的调度器使用不同的 Executor
- Mesos 集群为了避免单点故障，所以使用 Zookeeper 提供高容错的副本机制。

1.1.1 Mesos的运行方式

下图描述了一个 Framework 如何通过调度来运行一个 Task



事件流程:

1. Slave1 向 Master 报告，有4个CPU和4 GB内存可用
2. Master 发送一个 Resource Offer 给 Framework1 来描述 Slave1 有多少可用资源
3. Framework1 中的 FW Scheduler会答复 Master，我有两个 Task 需要运行在 Slave1，一个 Task 需要<2个cpu，1 gb内存="">，另外一个Task需要<1个cpu，2 gb内存="">
4. 最后，Master 发送这些 Tasks 给 Slave1。然后，Slave1还有1个CPU和1 GB内存没有使用，所以分配模块可以把这些资源提供给 Framework2

注意：当 Tasks 完成和有新的空闲资源时，Resource Offer会不断重复这一个过程。

1.1.2 Mesos与虚拟化、容器技术对比

1.1.3 Mesos的应用场景

- 容器编排

Mesos本身是一个分布式资源调度管理系统，而且是一个比较开放通用的资源调度管理系统。其北向提供了开放性的Framework框架平台，允许其他应用框架的友好接入（如spark、hadoop等），其南向定义了Executor执行器机制，允许容器、虚拟机等作为执行器进行任务

的处理等工作。容器技术随着Docker的出现，目前越来越受到热捧，关于如何更好更快的将容器技术应用到生产实践中全世界都在进行广泛的实践和探索。容器技术不可以质疑的是最佳的执行者，但其缺少一个上层的编排调度系统。应用框架+Mesos+容器的架构当前被大量的讨论，也有一些企业在对这一架构进行了实践，总体上来说是一个比较稳定可靠的架构。因为这个架构下，不管对于应用框架还是对于执行器来说都是统一通用的，这个样似乎更加符合DCOS的设想。

- 提升资源利用率

提升资源利用率这个词语一定会和虚拟化或云计算一起出现，的确目前在各个层面都在追逐提升资源利用率，因为在大规模服务器的数据中心中，资源利用率的提升代表着更少的资源投入。对于使用超过50台服务器的公司而言，一个通常的使用Mesos的动机就是提升资源利用率，并且减少运维成本。目前已经有许多这样的公司，比如各种公有云和私有云服务的提供商。在Ebay的案例中，它们曾经在Mesos上运行Jenkins这样可以减少虚拟机的使用。Mesosphere也发布了相关的文章对于HubSpot(运行在AWS上)的案例研究,文章中介绍了HubSpot是如何使用几十台大型的服务器来替代了几百台小型的服务器，使得硬件的利用率更高。

- 批处理服务和普通处理服务共存

所谓的批处理服务与普通服务共存可以在一个共享的Mesos集群中同时运行批处理任务以及其他的服务，这将对资源利用率的提升起到关键作用，同时这也是mesos一直所追求目标：统一的DCOS。如在一个Mesos集群中可以运行如M&R、Spark等批处理服务也可以运行如jenkins等普通服务。这样就不用在一个数据中心划分不同的服务运行区域，进行资源的隔离和精确匹配。

1.2 Mesos安装指南

1.2.1 Mesos集群组件介绍

1.2.2 Mesos生产环境配置介绍

1.3 安装Mesos和Zookeeper

1.3.1 标准包安装

1.3.2 源码包安装

1.4 Docker安装配置指南

1.4.1 安装方法

1.4.2 配置方法

1.4.3 配置Mesos Slave与Docker

1.5 Mesos升级

1.5.1 升级流程

- 按照官所升级版的要求编译和安装所有依赖模块，以保证更新后的版本不存在依赖包的问题。
- 在Master节点安装新版本的源码包，并重启master节点。
- 在slave节点安装新版本的源码包，并重启slave节点。
- 通过连接本地库/jar等来更新调度器
- 重启调度器。
- 如果有必要也可以通过连接本地库/jar来更新执行器（如docker的版本）。

1.5.2 注意事项

0.24.x 升级到 0.25.x

注意：在0.25.x版本中一些配置文件不需要包含json后缀，不过在0.25版本配置文件是否包含json后缀同样是有效的，包含json后缀的配置文件形式会在后续的版本中逐步的被取消。

Master节点：

- /state.json 变成 /state
- /tasks.json 变成 /tasks

在slave节点：

- /state.json 变成 /state
- /monitor/statistics.json 变成 /monitor/statistics

master和slave节点有变化的配置文件：

- /files/browse.json 变成 /files/browse

- /files/debug.json 变成 /files/debug
- /files/download.json 变成 /files/download
- /files/read.json 变成 /files/read

注意：在0.25.x版本中，C++，Java,Python的调度包也已经进行了更新，特别地，这些调度包的驱动可以通过生成一个suppressOffers()去直接停止receiving offers进程。

0.23.x 升级到 0.24.x

注意：在0.24.x版本，master节点在zookeeper中发布信息是通过JSON文件来进行，不在通过protobuf。

0.22.x 升级到 0.23.x

注意：

- 在master和slave节点上，配置文件stats.json已经被更新成metrics/snapshot。
- 配置文件/master/shutdown已经被弃用
- 为了在decorator模块可以移动元数据（环境变量或者标签），在0.24.x版本中改变了一些decoratorhooks返回值的意义，详细请见更新文档。
- Slave节点的ping超时时间现在可以在master节点进行配置，可以通过--slave_ping_timeout 和 --max_slave_ping_timeouts进行配置。
- 在0.23.x版本中新增了一个调度driverAPI：acceptOffers，这是launchTasks API的更完善版本。这个driverAPI的作用是允许调度器去接受一个提议并指定一个应用运行列表去进行资源的调度。目前支持的应用包括：LAUNCH (launching tasks), RESERVE (making dynamic reservations), UNRESERVE (releasing dynamic reservations), CREATE (creating persistent volumes) and DESTROY。
- protobuf源已经扩展成可以包含更多的metadata以便支持存储持久化、动态伸缩、资源超售。这样两个资源对象拥有不同的metadata时，你就不用必须把他们合并。

0.21.x 升级到 0.22.x

- 在这个版本中，slave检查点标签已经被移除，因为所有的slave节点都会启用这一功能，但是在Frameworks在利用checkpoint登记他们自己的任务时，还需要开启checkpointing。
- 在master和slave节点上，stats.json已经被弃用，需要使用metrics/snapshot。
- C++/Java/Python调度包已经被更新，尤其是调度driver里包含了一个附加参数可以指定是否去使用模糊的驱动确认。

- 验证API为支持第三方的验证机制在这个版本中有了一个轻微的变化
AuthenticationStartMessage.data 中的变量类型从string类型变成bytes类型并没有对C++或者over-the-wire表示法造成影响，所以这个一轻微的变化只影响了如java Python等这些语言包，因为在这些语言中UTF-8 sting和byte数组使用的是不同的类型。
- 所有的mesos参数可以使用file://将他们从文件中读取出来进行传递。包括白名单证书、所有JSON返回参数标签，尽管支持只传送一个绝对路径而不是一个file://，但是这种方式已经被弃用了，如果继续使用就会产生警告信息。

升级 0.20.x 到 0.21.x

关闭slave节点的检验信息已经被弃用，slave节点的检查点标签也被弃用，并在下一个版本中会被移除

1.6 总结

Mesos具备当前最流行的两层资源调度机制和最开放最稳定的部署框架，mesos社区也是当前最火热的社区之一，目前已经迭代了27个版本，即0.27版本已经发布。在社区和企业的全力探索和实践下，mesos正在向着统一的云数据中心操作系统这一伟大目标前进，我们可以乐观的预想或许不久之后mesos能够像openstack那样被全世界全面的接受，成为每个数据中心所必需的优秀资源调度系统的代名词。

搭建一个 Mesos 集群

实践出真知，学习 Mesos 也不例外，这里将介绍怎样搭建一个最小生产集群。由于 Mesos 目前仅提供对 Linux 版本，所以这里将介绍怎样在 Linux 环境下搭建 Mesos 集群；所以，读者需要有一定的 Linux 基础知识。另外，考虑到在生产环境中主要以 RHEL/CentOS 为主，所以本节内容都基于最新的 CentOS 7 稳定版本。

准备工作

在开始搭建 Mesos 集群之前，需要准备以下环境。

- 3 台 Intel x86_64 架构的计算机
- 计算机中安装了 CentOS 7.1 操作系统
- 计算机至少配备了 2core, 4GB 内存

如果准备 3 台物理机有难度，虚拟机也是可以的；Linux 环境中最常用的桌面虚拟化软件例如：VirtualBox 都非常好用。

注：介绍怎样准备虚拟机，怎样安装 CentOS 7.1 操作系统超出了本书的范围，不再赘述。

假设 3 台计算机分别为：

- A, IP 10.23.85.233
- B, IP 10.23.85.234
- C, IP 10.23.85.235

上面的 IP 地址其实不重要，只需要读者的 3 台计算机具有互相可访问的 IP 地址即可（当然，最好是局域网，在同一个子网更好，能够保证网络质量）。

下面我们将在这三台计算机中搭建一个 Mesos 集群，由于 Mesos 是一个分布式的集群服务，所以要提供稳定高可用的服务，3 个 mesos-master 结点是必不可少的，所以这里不再介绍只要一个节点的 mesos 服务；同时，为了降低演示的复杂度，3 台计算机节点将同时运行 mesos-master 和 mesos-slave 进程，也就是同时充当控制节点和计算节点的角色。

在开始之前，我们选定 A 作为操作机，大部分的操作都将在 A 上完成，在 A 上通过 SSH 连接到 B 和 C 来完成部署，所以下面将介绍下怎样搭建 SSH 环境。

搭建 SSH 环境

SSH(Secure Shell) 顾名思义是一种安全 Shell 技术，通过 SSH，可以拿到远程主机的 Shell，就像操作本机一样。

SSH 是一项非常常用的功能，所以大部分 Linux 发行版都默认提供了 SSH 客户端，CentOS 也不例外；所以，这里只介绍怎样在 CentOS 7 上搭建 SSH 服务，搭建了服务之后，就可以从其它主机上通过 SSH 客户端获取本机的安全 Shell 了。

还是以 A, B, C 三个节点为例，由于这里使用 A 作为控制机，所以需要在 B 和 C 上搭建 SSH 服务，并且配置好账号以便能够允许用户从 A 上获取 B 和 C 的安全 Shell。

这里以 B 为例介绍怎样配置 SSH 服务。

由于要执行一些特权操作，所以这里直接切换到 root 用户，所以，一定要小心操作，以免损坏系统。

```
$ su - root  
Password:
```

在输入 root 用户密码后，即获得了一个 root 用户登录 Shell，登录 Shell 和非登录 Shell (su root) 的不同在于 Shell 初始化的不同，具体细节这里不再介绍。

在获得了 root Shell 之后，就可以安装软件了，下面来安装 SSH 服务器，输入如下命令：

```
# yum clean all  
# yum makecache  
# yum install -y openssh-server
```

上面 3 条命令：

- 第一条命令删除所有 yum cache，以免用户修改过 yum repo 等信息，导致错误
- 第二条命令重新构建 yum cache
- 最后一条命令安装 openssh-server 包，`-y` 参数表示不用提示用户，直接安装

安装完成后，使用下面的命令启动 SSH 服务：

```
# systemctl start sshd.service
```

如果想要设置 SSH 服务开机自动启动，可以使用如下命令：

```
# systemctl enable sshd.service
```

如果读者使用过 CentOS 6，应该会发现，这里启动服务不再使用 `service` 或者 `initctl` 命令，而是使用 `systemctl`。

`systemctl` 是 `systemd` 的控制命令，CentOS 从 7 版本开始，使用 `systemd` 替换了原有的 `sysvinit` 及其它相关软件。

启动后，可以使用 `systemctl status` 命令来查看 SSH 服务（名字为 `sshd.service`）的状态。

```
# systemctl status sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Mon 2015-11-16 17:30:12 CST; 2 weeks 5 days ago
     Main PID: 986 (sshd)
      CGroup: /system.slice/sshd.service
              └─986 /usr/sbin/sshd -D
...
... 省略部分输出 ...
```

开启了 SSH 服务后，就可以从其它机器上连接 SSH 服务的专有 22 端口了。但是，要成功登陆，还需要配置好用户账号。

默认情况下，SSH 服务允许使用 root 账号通过密码登陆，所以，只需要设置了 root 账号密码即可通过 SSH 登陆，如下为 `/etc/ssh/sshd_config` 的相关配置：

```
#PermitRootLogin yes
```

为了简单，这里使用 root 账号密码登陆，如果用户需要了解怎样配置通过 SSH 密钥登陆，可以参考相关文档配置。

下面从计算机 A 登陆 B，在 A 上执行下面命令：

```
$ ssh -l root 10.23.85.234
```

在输入 B 机器中 root 账号的密码后，登陆成功了！接下来在这个会话中执行的任何命令都和在计算机 B 上执行的效果一样（当然，除了登出）。

上面的命令中 `-l` 参数表示要登陆的账号用户名，而 10.23.85.234 则是远程主机 IP 地址。

在配置好了计算机 B 的 SSH 服务后，以同样的方式配置好 C。搭建 SSH 服务只是为了搭建 Mesos 集群做准备，接下来将要介绍怎样搭建 ZooKeeper 集群服务。

搭建 ZooKeeper 集群

ZooKeeper 是一个分布式基础服务软件，最初作为 Hadoop 项目的一个子项目，目前已经成为 Apache 软件基金会的顶级项目。Mesos 依赖 ZooKeeper 提供的服务，所以在搭建 Mesos 集群之前，首先需要搭建 ZooKeeper 集群。

通过后面的章节可以了解到，不仅 Mesos，包括后面将要介绍的 Marathon, Chronos, Storm 等项目也都使用了 ZooKeeper 服务。

首先，我们需要到 ZooKeeper 项目主页 (<http://zookeeper.apache.org/>) 上下载 ZooKeeper，如下图所示：

这里我们下载目前最新的稳定版本：3.4.6，如下图所示：

Name	Last modified	Size	Description
Parent Directory		-	
zookeeper-3.4.6.tar.gz	2014-10-31 05:48	17M	

可以直接通过浏览器点击下载链接来下载，或者，也可以通过 wget 来下载，假设在 A 节点上：

```
$ cd ~/Downloads
$ wget http://apache.faye.com/zookeeper/current/zookeeper-3.4.6.tar.gz
```

要提供稳定可靠的 ZooKeeper 服务，至少需要 3 个 ZooKeeper 服务实例，这里将在 A, B 和 C 上搭建一个由 3 个 ZooKeeper 服务实例组成的高可用 ZooKeeper 服务。

下面的操作如非特别说明，都在节点 A 上完成。

首先，解压下载好的 ZooKeeper 压缩包。

```
$ cd ~/Downloads
$ tar -xzf zookeeper-3.4.6.tar.gz
```

上面的解压命令在正确解压时，不会输出任何内容，这也是 Linux 的一个设计哲学：没有消息即是好消息。所以，Linux 会尽量不去打扰用户。

解压完成后，首先来看一下怎样启动一个只有一个 ZooKeeper 实例的服务，只有一个实例也就意味着：一旦这个实例退出，服务就变得不可用了。

启动一个 ZooKeeper 服务

进入解压后的 ZooKeeper 目录，这里需要留意的有几个目录：

- `conf`，配置文件目录
- `bin`，可执行文件，脚本目录

在启动 ZooKeeper 之前，首先需要一个可用的配置文件，ZooKeeper 自带一个示例配置文件，我们可以在示例配置文件基础上修改自己的配置。

复制实例配置文件：

```
$ cd ~/Downloads/zookeeper-3.4.6
$ cp conf/zoo_sample.cfg conf/zoo.cfg
```

其实，无需任何修改，就可以启动单个实例 ZooKeeper 服务了：

```
$ ./bin/zkServer.sh start
JMX enabled by default
Using config: /home/chengwei/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Starting zookeeper ... STARTED
```

上面的命令很简单，最后会输出命令执行的结果，`STARTED` 表示启动 ZooKeeper 成功。

另外，可以使用 `zkServer.sh status` 命令来查看 ZooKeeper 服务的当前状态。

```
$ ./bin/zkServer.sh status
JMX enabled by default
Using config: /home/chengwei/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Mode: standalone
```

可见，现在 ZooKeeper 服务已经启动，并且可以提供服务了，在没有修改配置文件的情况下，默认 ZooKeeper 运行在 `standalone` 模式，也就是非高可用模式。

下面我们简单的测试一下，使用 `zkCli.sh` 可以连接 ZooKeeper 服务，不指定服务地址将默认连接本地服务。

如下所示（省略了部分输出）：

```
$ ./bin/zkCli.sh
Connecting to localhost:2181
2015-12-05 18:35:30,100 [myid:] - INFO  [main:Environment@100] - Client environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT
2015-12-05 18:35:30,102 [myid:] - INFO  [main:Environment@100] - Client environment:host.name=mesos-master-dev021-cqdx.qiyi.virtual
2015-12-05 18:35:30,103 [myid:] - INFO  [main:Environment@100] - Client environment:java.version=1.7.0_75
2015-12-05 18:35:30,104 [myid:] - INFO  [main:Environment@100] - Client environment:java.vendor=Oracle Corporation
2015-12-05 18:35:30,104 [myid:] - INFO  [main:Environment@100] - Client environment:java.home=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.75-2.5.4.2.el7_0.x86_64/jre
...
... 省略了部分输出 ...
2015-12-05 18:35:30,106 [myid:] - INFO  [main:ZooKeeper@438] - Initiating client connection, connectString=localhost:2181 sessionTimeout=30000 watcher=org.apache.zookeeper.ZooKeeperMain$MyWatcher@65297ad9
Welcome to ZooKeeper!
...
... 省略了部分输出 ...
[zk: localhost:2181(CONNECTED) 0]
```

连接到 ZooKeeper 服务后，会打开一个命令 shell，可以使用 `help` 命令查看支持的命令以及各个命令的用法。

```
[zk: localhost:2181(CONNECTED) 0] help
ZooKeeper -server host:port cmd args
    connect host:port
    get path [watch]
    ls path [watch]
    set path data [version]
    rmr path
    delquota [-n|-b] path
    quit
    printwatches on|off
    create [-s] [-e] path data acl
    stat path [watch]
    close
    ls2 path [watch]
    history
    listquota path
    setAcl path acl
    getAcl path
    sync path
    redo cmdno
    addauth scheme auth
    delete path [version]
    setquota -n|-b val path
```

下面我们测试一下使用 ZooKeeper 来存取简单的键值对。

```
[zk: localhost:2181(CONNECTED) 1] create /book mesos-in-action
Created /book
[zk: localhost:2181(CONNECTED) 2] ls /
[book, zookeeper]
[zk: localhost:2181(CONNECTED) 3] get /book
mesos-in-action
cZxid = 0xe
ctime = Sat Dec 05 18:51:37 CST 2015
mZxid = 0xe
mtime = Sat Dec 05 18:51:37 CST 2015
pZxid = 0xe
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
```

ZooKeeper 远远不是用来存取键值这么简单，它的目标是提供分布式软件开发中常见的问题，常用的功能，例如：一致性配置存储，领导选举，分布式锁等。

在基本了解了怎样搭建一个单实例的 ZooKeeper 服务之后，我们来看看怎样搭建由 3 个 ZooKeeper 实例组成的高可用服务。在生产环境中，推荐至少搭建 3 个 ZooKeeper 实例，由它们组成一个高可用的服务，避免因一个服务实例故障而导致服务不可用。

由于 ZooKeeper 作为分布式环境基础服务，所以一旦 ZooKeeper 服务不可用，那么基于它的服务也就不可用了，例如这里将要介绍的 Mesos，以及后续将要介绍的 Marathon，Chronos 等。

搭建高可用 ZooKeeper 服务

这里将搭建由 3 个 ZooKeeper 实例组成的高可用服务，前面我们已经搭建了一个单实例服务。简单的说，高可用服务首先需要 3 个实例，并且需要分别位于不同的机器上，如果是虚拟机，那么虚拟机也要分别位于不同物理机上，如果需要更严格的高可用，可以要求物理机不要位于同一机架，交换机。

这里我们以前面的 3 个 A, B, C 节点为例，分别在上面启动一个 ZooKeeper 实例，并且让它们组成一个服务。

我们首先在 A 上启动 ZooKeeper 实例，方法和上一节搭建单个 ZooKeeper 实例方法类似，需要有两点修改。

首先，需要修改配置文件 `conf/zoo.cfg`，将下面几行添加到 `conf/zoo.cfg` 末尾。

```
server.1=10.23.85.233:2888:3888
server.2=10.23.85.234:2888:3888
server.3=10.23.85.235:2888:3888
```

上面 3 行配置分别指定了 3 个 ZooKeeper 实例的地址，键的格式为：`server.<N>`，其中数字 N 表示第几个实例，合法值为从 1 到 255。值的格式为：`host:port1:port2`，host 表明这个实例运行的地址，port1(2888) 为 ZooKeeper leader 和其它 ZooKeeper 实例之间通信的端口，port2(3888) 为各个实例进行 leader 选举时所用端口。

由于上面的配置文件在每个 ZooKeeper 实例中都是一样的，ZooKeeper 并不会自己检测 IP 来确定自己是第几个 server，所以，还需要单独配置。

配置方法是在配置文件中指定的 `dataDir` 目录中创建一个 `myid` 文件，并且其中只包含一个数字，这个数字即是配置文件中 `server.<N>` 中的 `<N>`。

例如：A 机器 10.23.85.233 在配置文件中为 `server.1`，那么就需要在 `dataDir` 中创建一个 `myid` 文件，并且内容只包含数字 1。

默认的 `zoo.cfg` 中的 `dataDir` 目录为 `/tmp/zookeeper`。在 Linux FHS(`/tmp/zookeeper`) 标准中，`/tmp` 是一个挂载在内存设备中的临时目录，数据在机器关机后丢失，所以在生产环境中，记住修改此配置，这里我们修改 `dataDir` 为 `/var/zookeeper`。

在熟悉了配置方式后，在 A, B, C 节点上，在 `conf/zoo.cfg` 中都添加同样的 3 行配置。

```
server.1=10.23.85.233:2888:3888
server.2=10.23.85.234:2888:3888
server.3=10.23.85.235:2888:3888
```

然后，分别在 A, B, C 上创建一个文件 `/var/zookeeper/myid`，值分别为 1, 2, 3。以在 A 机器上操作为例：

```
$ mkdir /var/zookeeper
$ echo 1 > /tmp/zookeeper/myid
```

在 3 台机器上配置完成后，就可以启动 ZooKeeper 实例了，以 A 节点为例：

```
$ ./bin/zkServer.sh start
JMX enabled by default
Using config: /home/chengwei/Downloads/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Starting zookeeper ... STARTED
```

如果此时使用 `status` 命令查看。

```
[chengwei@mesos-master-dev021-cqdx zookeeper-3.4.6]$ ./bin/zkServer.sh status
JMX enabled by default
Using config: /home/chengwei/Downloads/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Error contacting service. It is probably not running.
```

会发现提示：`Error contacting service. It is probably not running.`，这是因为 `status` 命令会尝试连接服务，并且从连接信息中判断服务是否可用，而由于我们将 ZooKeeper 实例配置成了高可用服务，所以单单只启动一个 ZooKeeper 实例是不能提供服务的，所以也就显示了上面的错误提示。

接下来，在 B 节点上启动 ZooKeeper 实例：

```
$ ./bin/zkServer.sh start
JMX enabled by default
Using config: /home/chengwei/Downloads/zookeeper-3.4.6/bin/..../conf/zoo.cfg
Starting zookeeper ... STARTED
```

现在，再次使用 `status` 在 A 上查看 ZooKeeper 服务状态。

```
$ ./bin/zkServer.sh status
JMX enabled by default
Using config: /home/chengwei/Downloads/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: follower
```

可以看到它的状态已经正常了，并且处于 `follower` 模式（注意：在读者的实践中，模式也有可能为 `leader`），那么，现在再来看看 B 节点上的 ZooKeeper 实例状态。

```
$ ./bin/zkServer.sh status
JMX enabled by default
Using config: /home/chengwei/Downloads/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: leader
```

现在，ZooKeeper 服务已经可以提供服务了，但是，为了保证服务的高可用性，需要在 C 节点上也启动 ZooKeeper 服务实例；这样，A, B, C 中任意一个服务实例故障，都不会影响到 ZooKeeper 服务的可用性。

当然，一旦故障，必须尽快恢复，否则，另一个实例再发生故障，那么服务就不可用了，对于由 N 个实例组成的 ZooKeeper 服务，任意时间只要有大于 $N/2$ 个实例正常，则可提供服务。

在搭建了 ZooKeeper 服务之后，就可以搭建 Mesos 生产集群了，其实，如果不是生产集群，Mesos 也可以不使用 ZooKeeper 服务。

搭建 Mesos 集群

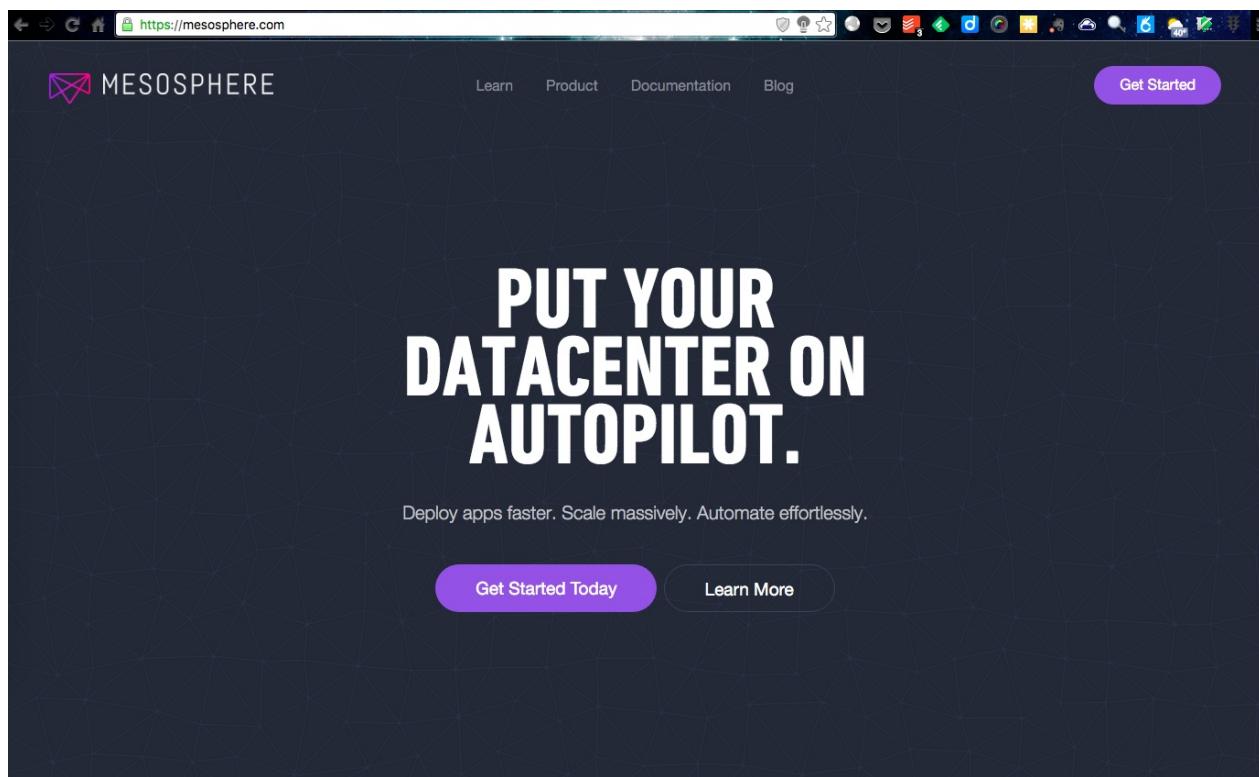
在介绍了怎样搭建 ZooKeeper 服务后，下面来看看怎样搭建一个 Mesos 集群，和搭建 ZooKeeper 服务类似，我们分两步进行：首先介绍搭建一个单 master/slave 的伪集群，然后搭建一个多 master（高可用）多 slave 的真集群。

搭建单 master/slave Mesos 集群

我们通过搭建单 master/slave 的 Mesos 集群，来了解一些 Mesos 基础知识，以便能更容易理解怎样搭建一个可用于生产环境的高可用 Mesos 集群。

Mesos 虽然是托管于 Apache 软件基金会，但主要的开发者大部分来自于 Mesosphere 公司，Mesosphere 是一家基于 Mesos 的创业公司，由 Mesos 的原始作者创办，mesosphere 同时为社区提供了一些非常好的文档，软件包，并且其主要产品 DCOS(Data Center Operating System) 也是基于 Mesos 构建。这里我们将从 mesosphere 官方网站上下载 mesos RPM 包用于安装，从而免去下载源码，编译，安装。

下面是一张 mesosphere 主页的截图。



从主页上，其实是不太方便找到 Mesos RPM 包下载地址的，将页面拉倒底部，或者直接访问 <https://mesosphere.com/downloads/>，就可以到达下载页面，如下图所示：

A screenshot of the "GET THE BUILDING BLOCKS" page from the Mesosphere website. The URL in the address bar is https://mesosphere.com/downloads/. The page has a dark background with a light gray grid. The Mesosphere logo is at the top left, and navigation links are at the top right. The main title "GET THE BUILDING BLOCKS" is in large, bold, white capital letters. Below it is a subtitle "Get started with some of Mesosphere's base services." Three service cards are shown in a grid: "APACHE MESOS" (blue icon), "CHRONOS" (green icon), and "MARATHON" (black icon). Each card contains a brief description and a "Get Started" button. The Apache Mesos card also includes a note about older releases.

直接点击 Apache Mesos 的 Get Started 链接，即可查看 mesosphere 关于安装 Mesos 软件的文档。这里以在 A 节点上搭建 Mesos 单 master/slave 集群为例。

安装 Mesos 软件

下载完成后，假设存放在节点 A 上的 /root/Downloads 目录下，现在就可以安装了：

```
# rpm -ivh /root/Downloads/mesos-0.25.0-0.2.70.centos701406.x86_64.rpm
warning: mesos-0.25.0-0.2.70.centos701406.x86_64.rpm: Header V4 RSA/SHA1 Signature, ke
y ID e56151bf: NOKEY
Preparing... ################################################ [100%]
Updating / installing...
1:mesos-0.25.0-0.2.70.centos701406 ##### [100%]
Created symlink from /etc/systemd/system/multi-user.target.wants/mesos-master.service
to /usr/lib/systemd/system/mesos-master.service.
Created symlink from /etc/systemd/system/multi-user.target.wants/mesos-slave.service t
o /usr/lib/systemd/system/mesos-slave.service.
```

安装完成后，可以看到，在 /etc/systemd/system/multi-user.target.wants/ 目录中创建了两个符号链接，分别指向 /usr/lib/systemd/system/mesos-master.service 和 mesos-slave.service。

细心的读者可能已经从配置 SSH 服务一节看到，这里的输出和 `systemctl enable sshd.service` 非常类似，是的，安装完 mesos RPM 包之后，自动设置了 mesos-master 和 mesos-slave 为开机启动。

读者也可以使用 `systemctl is-enabled` 命令来验证，如下：

```
# systemctl is-enabled mesos-master mesos-slave
enabled
enabled
```

由于我们将在 A, B, C 三个节点上都启动 mesos-master, mesos-slave, 所以不用禁止它们中的任何一个。但是，在实际生产环境中，节点往往只充当一个角色，要么运行 mesos-master，要么运行 mesos-slave，所以往往需要禁止一个服务的开机启动，以禁止 mesos-slave 为例：

```
# systemctl disable mesos-slave
Removed symlink /etc/systemd/system/multi-user.target.wants/mesos-slave.service.
```

配置 Mesos

安装完成后，在启动之前，我们需要进行一些必须的配置，mesos-master 和 mesos-slave 各自都有很多可配置的选项，这里不再详细介绍，读者可以参考 `--help` 或者官方文档了解。

安装完 mesos RPM 包后，可以通过下面命令查看安装的文件：

```
# rpm -ql mesos
```

这里需要注意的有几个文件：

- `/usr/lib/systemd/system/mesos-master.service`, `mesos-master` 进程的服务配置文件，用来控制服务的启停
- `/usr/lib/systemd/system/mesos-slave.service`, `mesos-slave` 进程的服务配置文件，用来控制服务的启停
- `/etc/default/mesos`, `mesos-master` 和 `mesos-slave` 共用的配置文件
- `/etc/default/mesos-master`, `mesos-master` 专用配置文件
- `/etc/default/mesos-slave`, `mesos-slave` 专用配置文件
- `/etc/mesos/zk`, ZooKeeper 地址配置文件, `mesos-master` 和 `mesos-slave` 共用

感兴趣的读者可以查看 `mesos-master.service` 和 `mesos-slave.service` 文件了解 `mesos-master`, `mesos-slave` 分别是被怎样启动的，然后怎样加载配置文件的，这里不再赘述。

我们知道，要启动一个高可用的 mesos 集群，需要至少 3 个 `mesos-master` 进程，以及 ZooKeeper 服务，接下来看下怎样配置：

/etc/default/mesos

`/etc/default/mesos` 为 `mesos-master` 和 `mesos-slave` 共用配置文件，默认配置内容如下：

```
# cat /etc/default/mesos
LOGS=/var/log/mesos
ULIMIT="-n 8192"
```

这里只有两项配置：

- `LOGS` 指定 `mesos` 存放日志的目录
- `ULIMIT` 指定 `mesos` 额外的 `ULIMIT` 配置，这里配置了可以打开的文件数为 8192

读者可以在这个文件中添加其它配置，例如：

- `GLOG_max_log_size=1024`，设置日志文件大小为 1024MB
- `GLOG_stop_logging_if_full_disk=1`，当磁盘满后停止写日志

需要注意的是，如果 `mesos-slave` 启动的进程打开的文件（Linux 中一切皆文件）太多，很可能超过 8192，此时就需要设置更大的值。

在这里，我们保持这个文件不修改即可。

mesos-master 配置

`mesos-master` 进程在启动时，会加载共用的配置文件，还会加载 `mesos-master` 专用的配置文件，所以非共用的配置文件，在 `/etc/default/mesos-master` 中配置即可。

首先来看看默认的配置都有什么：

```
# cat /etc/default/mesos-master
PORT=5050
ZK=`cat /etc/mesos/zk`
```

可以看到，这里只有两个配置项：

- PORT，mesos-master 监听的端口，默认为 5050，不用修改
- ZK，ZooKeeper 服务地址，默认为读取 /etc/mesos/zk 中的内容

可以看到，/etc/mesos/zk 默认配置为：

```
# cat /etc/mesos/zk
zk://localhost:2181/mesos
```

这是一个本地 ZooKeeper 服务，并且是一个 standalone 的 ZooKeeper 服务，这里简单解析下配置的格式：

- zk://，这是 ZooKeeper 服务地址的前缀，意义类似 HTTP 协议的 http://
- localhost:2181，这是一个 ZooKeeper 服务实例，如果有多个服务实例，则用逗号分隔，例如：ip1:port1,ip2:port2,ip3:port3
- /mesos，这是 mesos 将要使用的存储资源，ZooKeeper 提供了类似文件系统的存储资源

所以，这里将其配置为我们在上一节搭建的 ZooKeeper 服务：

```
# cat /etc/mesos/zk
zk://191.168.1.101:2181,191.168.1.102:2181,191.168.1.103:2181/mesos
```

配置完成后，就可以启动 mesos-master 进程了，所以，实际上必须的配置非常简单，只需要配置好 ZooKeeper 服务地址即可，甚至，因为我们在 A 结点上也启动了 ZooKeeper 服务，连默认的 /etc/mesos/zk 配置文件也无需修改。

mesos-slave 配置

mesos-slave 和 mesos-master 类似，只是具有不同的配置选项和专有的配置文件。

```
# cat /etc/default/mesos-slave
MASTER=`cat /etc/mesos/zk`
```

可以看到，mesos-slave 的专有配置文件里，只配置了一个选项：

- MASTER，指定 mesos-master 的服务地址

由于我们在同一个计算节点上启动 mesos-master 和 mesos-slave，所以这里不需要再配置 /etc/mesos/zk 了，已经在上面的 mesos-master 配置中配置好了。

启动 Mesos 集群

配置完成后，执行下面的命令分别启动 mesos-master 和 mesos-slave 进程，这两个进程分别作为 Mesos 集群的控制结点和计算结点。

```
# systemctl start mesos-master
# systemctl start mesos-slave
```

正常情况下，上面两行命令不会有任何输出，然后，可以使用下面的命令来查看 mesos-master 或 mesos-slave 进程状态，以 mesos-master 为例：

```
# systemctl status mesos-master
● mesos-master.service - Mesos Master
   Loaded: loaded (/usr/lib/systemd/system/mesos-master.service; enabled; vendor prese
t: disabled)
     Active: active (running) since Sun 2016-04-03 16:53:37 CST; 2s ago
       Main PID: 13489 (mesos-master)
          Memory: 5.2M
            CGrou... /system.slice/mesos-master.service
              └─13489 /usr/sbin/mesos-master --zk=zk://10.23.85.233:2181,10.23.85.234:218
1,10.23.85.235:2181/mesos --port=5050 --log_dir=/var/log/mesos...
                  ├─13499 logger -p user.info -t mesos-master[13489]
                  └─13500 logger -p user.err -t mesos-master[13489]

Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.084251 13504 group.cp
p:403] Trying to create path '/mesos/log_replicas' in ZooKeeper
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.086472 13508 contende
r.cpp:265] New candidate (id='3') has entered the cont...adership
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.092560 13507 detector
.cpp:156] Detected a new leader: (id='2')
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.092670 13507 network.
hpp:415] ZooKeeper group memberships changed
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.092680 13501 group.cp
p:674] Trying to get '/mesos/json.info_0000000002' in ZooKeeper
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.092877 13504 group.cp
p:674] Trying to get '/mesos/log_replicas/0000000001' ...ooKeeper
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.094472 13504 group.cp
p:674] Trying to get '/mesos/log_replicas/0000000002' ...ooKeeper
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.094674 13501 detector
.cpp:481] A new leading master (UPID=master@10.23.85.2...detected
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.094743 13501 master.c
pp:1603] The newly elected leader is master@10.23.85.2...c6141315
Apr 03 16:53:37 10.23.85.233 mesos-master[13500]: I0403 16:53:37.095158 13503 network.
hpp:463] ZooKeeper group PIDs: { log-replica(1)@10.23....3:5050 }
Hint: Some lines were ellipsized, use -l to show in full.
```

通过命令的输出可以看到，mesos-master 进程已经正常启动，现在，可以打开浏览器，指向 A 节点的 5050 端口，就可以查看到 mesos-master 的 WEB 管理界面了。

如下图所示：

The screenshot shows the Mesos master web interface at the URL 10.23.85.234:5050. The top navigation bar includes tabs for Mesos, Frameworks, Slaves, Offers, and a search bar. The title bar says "mesos-in-action".

Active Tasks:

ID	Name	State	Started ▾	Host
No active tasks.				

Completed Tasks:

ID	Name	State	Started ▾	Stopped	Host
No completed tasks.					

Left sidebar details:

- Cluster:** mesos-in-action
- Server:** 10.23.85.234:5050
- Version:** 0.25.0
- Built:** 5 months ago by root
- Started:** 29 minutes ago
- Elected:** 29 minutes ago

LOG:

Slaves:

Activated	3
Deactivated	0

Tasks:

Staged	0
Started	0
Finished	0
Killed	0
Failed	0
Lost	0

Resources:

CPU	Mem
Total	24 43.5 GB
Used	0 0 B
Offered	0 0 B
Idle	24 43.5 GB

Mesos WEB 管理界面主要由 5 部分组成：

- 首页：集群信息概览页
- 框架（Frameworks）概览页
- 计算结点（Slaves）页
- 资源（Offers）页
- 其它详细页

首页

在首页中，左边栏中显示了集群的一些关键信息，例如：

- 集群名称
- 本服务地址
- 集群版本号
- 计算节点统计信息
- 任务统计信息
- 资源统计信息

等信息。

页面中间由两部分信息组成：

- 当前活动的任务
- 最近已经完成的任务

框架概览页

框架概览页面如下图所示：

The screenshot shows the Mesos master's web interface. At the top, there are tabs: 'Mesos', 'Frameworks', 'Slaves', and 'Offers'. The 'Frameworks' tab is active. Below the tabs, a breadcrumb navigation shows 'Master / Frameworks'. The main content area has two sections: 'Active Frameworks' and 'Terminated Frameworks'. The 'Active Frameworks' section contains a table with columns: ID, Host, User, Name, Active Tasks, CPUs, Mem, Max Share, Registered, and Re-Registered. The 'Terminated Frameworks' section contains a table with columns: ID, Host, User, Name, Registered, and Unregistered.

Active Frameworks

ID ▼	Host	User	Name	Active Tasks	CPUs	Mem	Max Share	Registered	Re-Registered
------	------	------	------	--------------	------	-----	-----------	------------	---------------

Terminated Frameworks

ID ▼	Host	User	Name	Registered	Unregistered
------	------	------	------	------------	--------------

页面由两部分信息组成：

- 当前活动的框架
- 运行完成的框架

当前活动的框架并不一定指当前正在运行的框架，只是对于 Mesos 来说，该框架活着或者允许重新注册（已经不工作，但是还在 failover 超时时间内），所以，一定不能以此来判断框架是否正在正常运行。

运行完成的框架即表示已经断开连接，也意味着已经超过了 failover 超时时长，当该框架再次注册时，mesos 将会把它当做一个全新的框架来处理。

计算节点页

计算节点页面如下图所示：

The screenshot shows the Mesos master's web interface. At the top, there are tabs: 'Mesos', 'Frameworks', 'Slaves', and 'Offers'. The 'Slaves' tab is active. Below the tabs, a breadcrumb navigation shows 'Master / Slaves'. The main content area displays a table titled 'Slaves' with columns: ID, Host, CPUs, Mem, Disk, Registered, and Re-Registered. A search bar with a magnifying glass icon and a 'Find...' placeholder is located above the table.

Slaves

ID ▼	Host	CPUs	Mem	Disk	Registered	Re-Registered
...9b69-fa377aeacb69-S4	10.23.85.233	8	14.5 GB	45.0 GB	4 minutes ago	

该页面会展示所有当前在线的计算节点，每页显示最多 50 个节点，并且支持搜索，页面中包含了计算节点的基本信息，例如：计算节点配备的 CPU 核数，内存量，磁盘量等。

点击计算节点的 ID 字段，可以跳转到该计算节点的详情页面，如下图所示：

The screenshot shows the Mesos master's "Frameworks" tab. At the top, there are tabs for Mesos, Frameworks, Slaves, and Offers. The Frameworks tab is selected. Below the tabs, the URL is shown as Master / Slave c3cb8544-1d2e-476c-bdad-76bd16ded94a-S1. The main content area displays the following information:

- Cluster:** mesos-in-action
- Slave:** 10.23.85.233
- Version:** 0.25.0
- Built:** 5 months ago
- Started:** 8 minutes ago
- Master:** 10.23.85.234

LOG

Tasks	Staged	0
Started	0	
Finished	0	
Killed	0	
Failed	0	
Lost	0	

Resources

	Used	Allocated
CPU	0	8
Memory	0 B	14.5 GB
Disk	0 B	290.2 GB

Frameworks

ID ▾	User	Name	Active Tasks	CPU (Used / Allocated)	Mem (Used / Allocated)

Completed Frameworks

ID ▾	User	Name	Active Tasks	CPU	Mem

该页面和首页有点类似，但是不同的是，这里看到的信息不是整个集群的统计信息，而是单个计算节点上的信息，例如：使用了多少资源，运行的任务等等。

资源页

资源页一般是关注得最少的页面，因为资源是动态变化的，并且变化得很快，所以从这里基本上看不到多少有用的信息，如下图所示：

The screenshot shows the Mesos master's "Offers" tab. At the top, there are tabs for Mesos, Frameworks, Slaves, and Offers. The Offers tab is selected. Below the tabs, the URL is shown as Master / Offers. The main content area displays the following information:

Offers

ID ▾	Framework	Host	CPU	Mem

可以看到，搭建具有一个控制节点和一个计算节点的 Mesos 伪集群还是非常简单的；但是，单个控制结点如果失败，那么整个集群就不可用了，所以在生产环境中，至少需要由 3 个控制结点一起组成高可用的服务，则能够在有一个节点故障的时候继续提供服务。

搭建高可用 Mesos 集群

高可用的 Mesos 集群主要是指 mesos-master 高可用，这样，即使所有 mesos-slave 都故障，那么也是可以随时加入进来的。这里会介绍在 A, B, C 三台机器上分别启动 3 个 mesos-master 进程组成高可用的 Mesos 集群，然后，在 A, B, C 三台机器上分别启动 3 个 mesos-slave 进程作为集群的计算结点。

注意：在生产环境中，为了保证 mesos-master 运行稳定，通常不会将 mesos-master 所在的结点同时作为计算结点。

前面已经介绍过怎样搭建只有一个控制节点和计算节点的伪集群，搭建具有多个控制节点和计算节点的集群时安装 Mesos 方式是一样的，不同的是在配置上；所以，这里主要介绍怎样配置，安装 mesos 不再赘述。

配置控制结点

我们已经在 A, B, C 三台机器上都已经安装了 mesos RPM 包，现在开始配置 mesos-master，也就是控制结点。

由于 mesos 实现了 Paxos 分布式协议，所以，mesos 需要知道要写入几份数据才算成功，简单说就是要有超过一半的 mesos-master 都成功，整个动作才能算作成功。

这个参数在 mesos-master 中叫做 `--quorum`，例如：我们要搭建 3 个 mesos-master，那么超过一半就可以为 2, 3；但是，如果是 3 的话，任意一个 mesos-master 故障，那么服务就不可用了，所以这里只能为 2。

那么，读者可能会问，如果控制节点数量为 5 呢？那么超过一半可为：3, 4, 5；同样，如果指定 `quorum` 为 5，则意味着有一个控制结点故障，服务不可用，指定为 4 意味着有 2 个控制节点故障就不可用，而 3 意味着可以允许两个控制结点不可用。

所以，其实 `--quorum` 的选取就非常简单了，即：超过一半控制结点数量的最小值。这样，既能保证服务的最大高可用，同时能提高性能（写入的份数越多，性能越低）。

那么，为什么总是选择奇数个控制结点呢？偶数不可以吗？偶数也是可以的，但是效果不好，例如：4 个控制结点的高可用性实际上和 3 个控制结点一样的，都只能允许 1 个控制结点失效。所以，为什么要多搭建一个控制结点呢。

所以，这里我们需要配置 `quorum` 为 2，mesosphere 的 mesos RPM 安装包默认配置了 `quorum` 为 1，这也是为什么在上一节没有修改 `quorum` 配置就能启动集群的密码。

```
# cat /etc/mesos-master/quorum
1
```

由于现在要启动 3 个 mesos-master 进程，所以，修改该文件中的数字为 2，确保在 A, B, C 结点中的配置一致（包括前面修改过的 `/etc/mesos/zk`）！

```
# cat /etc/mesos-master/quorum
2
```

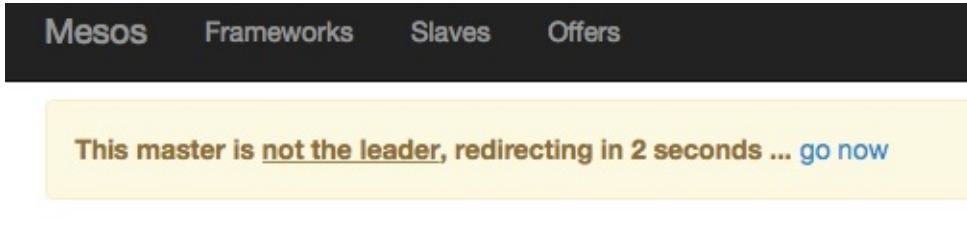
配置完成后，在启动之前，我们需要先删除之前单个集群的日志数据，否则，新启动的 mesos-master 会尝试去恢复之前的状态，从而失败，导致集群不能启动。

```
# rm -rf /var/lib/mesos/*
```

删除之后，就可以启动 mesos-master 进程了，分别在 A, B, C 3 个节点上启动 mesos-master：

```
# systemctl start mesos-master
```

启动完成后，可以通过浏览器分别访问 A, B, C 节点的 5050 端口，就可以打开 mesos-master WEB 页面了，由于只有其中 1 个节点是 leader 节点，所以在打开非 leader 节点的 WEB 页面时，mesos 会自动跳转到 leader 页面，如下图所示：



启动了高可用的 mesos 服务后，就可以向集群中添加计算节点了，在 A, B, C 3 个节点上启动 mesos-slave 进程：

```
# systemctl start mesos-slave
```

完成后，可以在 mesos-master WEB 页面看到有 3 个 slaves 在线，如下图所示：

ID	Host	CPUs	Mem	Disk	Registered	Re-Registered
...9b69-fa377aeacb69-S4	10.23.85.233	8	14.5 GB	45.0 GB	just now	
...9b69-fa377aeacb69-S3	10.23.85.234	8	14.5 GB	45.0 GB	just now	
...9b69-fa377aeacb69-S2	10.23.85.235	8	14.5 GB	45.0 GB	just now	

至此，搭建一个高可用的 Mesos 集群也已经完成了，总结起来，需要修改的配置有两处：

- /etc/mesos/zk，配置 ZooKeeper 服务地址
- /etc/mesos-master/quorum，配置集群 quorum

所以，搭建一个高可用的 Mesos 集群实际上是很简单的，但是，mesos-master, mesos-slave 还有许多配置项，默认的配置往往不能适应所有用户需求，需要读者去发现。

另外，由于我们使用的是 Mesosphere 提供的 RPM 包，这和从源码安装的配置方式完全不同，所以，如果读者想要更加深入的学习 mesos 配置及用法，建议通过源代码编译安装配置。

小结

本节详细介绍了怎样搭建单实例 ZooKeeper 服务，高可用 ZooKeeper 服务；单实例 Mesos 集群，高可用 Mesos 集群；以及一些 ZooKeeper 基础知识。在接下来一节内容里，我们将介绍更多的 Mesos 知识。

Mesos 工作原理

本节将简单介绍下 Mesos 的工作原理，以及通过简单的实例来展示 Mesos 的魔力。更多更具深度，难度的真实案例读者可以参考本书其它章节。

为什么是 Mesos

简单回顾下计算机界的发展历程，可以发现一些规律，例如：CPU 的发展首先是尝试不断提升单核 CPU 的计算能力，但是当单核计算能力的提升越来越困难时，多核 CPU 顺势推出，继续提高着单个计算结点的计算能力，但是当单个计算结点计算能力越来越难于提高，或者说成本越来越高时，分布式大规模的集群计算迎难而上。

而当各种分布式计算框架越来越流行时，问题出现了，各个框架独自占有各自的计算集群，即使在任务不多或者没有任务时，资源依然不能被其它框架共享，导致不能充分利用现有的计算资源。究其原因，大概主要有以下几方面。

集群搭建复杂，配置管理难度大

集群搭建复杂，配置难度大，主要在于如果将多个计算框架部署在一个集群上，一旦集群结点达到几十甚至上百个，那么配置管理的难度将非常大，并且呈指数上升，因为要维护多个不同的计算框架。

硬件可能定制化

由于各个计算框架都是针对一定使用场景设计的，所以其任务的类型也具有一定特点，例如：当前流行的 Hadoop, Spark, Storm 等计算框架，各自都有自己擅长的计算类型；相应地，资源需求类型也不同：Hadoop 需要高性能的磁盘 I/O 以便加速 MapReduce 读写的速度，Spark 需要大容量内存因为其将计算中间结果都存放在内存中。

所以，为了更加节省计算资源，充分利用资源，硬件可能定制化，例如：给 Hadoop 集群配备 SSD，给 Spark 集群配置大容量内存等。

所以，定制化的硬件并不能很好的适用于其它计算框架或者说收益成本比不高。

计算框架间资源抢占，影响稳定性

即使将多个框架搭建在同一个集群中，由于各个框架是独立的，互相间并不知道各自的状态，那么很可能出现资源抢占，甚至冲突，例如：框架 A 和 B 可能同时提交大量任务，导致集群资源不够用，发生 OOM 等严重错误，甚至同时访问相同的资源，例如：访问同一个

文件，这将导致更严重的数据损坏。

计算框架间任务隔离性差

单个框架独占集群时，由于整个集群的资源都归单个框架调度，所以，能够灵活的调度各个任务到各个结点运行，即使任务间采用最简单的进程隔离，问题也不大。

但是，如果是多个框架，那么隔离性差，很可能导致框架间的任务互相影响，例如：两个框架的两个任务依赖的库文件版本不一样，或者库之间有冲突等等，这时，简单的进程隔离完全不能解决问题，需要文件系统的隔离。

在细数了为什么不能很好的将多个分布式计算框架运行在同一个集群之上后，我们回到起点：为什么要将多个框架运行在一个集群中呢？

在提出这个疑问时，我们潜意识里就已经承认了多个分布式计算框架共享集群能够更好的利用资源。在著名的 Mesos 论文中，几位作者也通过详尽的建模，编码，测试，分析验证了这个结论。

所以这里我们不再论述这个问题，假设读者接受了这个事实：集群共享比独占集群更节省资源。

Mesos 怎样工作

首先，引用一下 Mesos 论文中的 Mesos 架构图，FIXME：如果不能直接引用，可以重画。

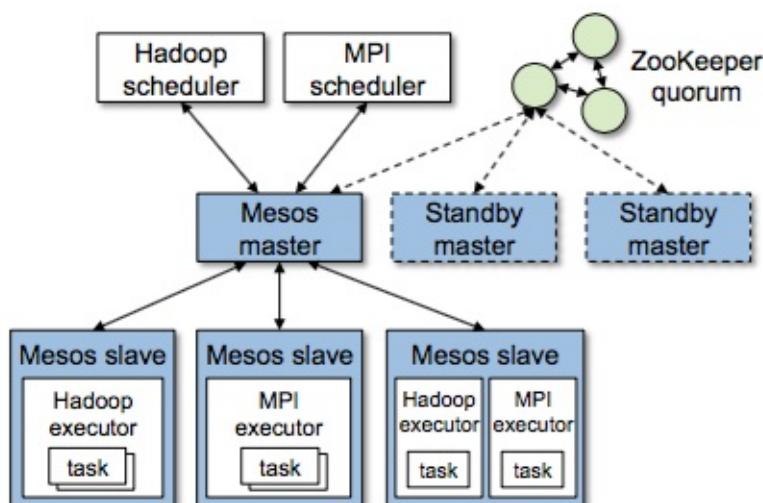


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

图中显示了 MPI 和 Hadoop 框架同时运行在 Mesos 之上的情景，图中所示 Mesos 集群由 3 个控制结点和 3 个计算结点组成，并且所依赖的 ZooKeeper 服务也由 3 个实例组成。

在某一个时刻，一个计算结点上可能会同时运行着 MPI 和 Hadoop 的任务。而 MPI 和 Hadoop 都通过 Mesos 控制结点来分发任务，而不是直接向计算结点分发任务。

在 Mesos 环境中，运行于 Mesos 之上的框架不再直接管理集群中的计算结点，以及直接分发任务，而是通过 Mesos 控制结点来获取当前可用的资源，然后将任务和资源组合发送给 Mesos 控制结点，由控制结点代为分发。

这样，Mesos 控制结点唯一控制整个集群的所有资源，具有全局资源试图，能够协调上层运行的多个计算框架，从而实现集群资源有序共享，而不会导致资源抢占甚至冲突。

在任务隔离上，基于 Linux Control Groups，Mesos 可以做到多方面的隔离，保证各个任务在一个良好的隔离环境中运行；值得一提的是：Mesos 支持当前最为流行的容器技术 Docker，Docker CTO 曾公开表示 Mesos 是 Docker 的黄金搭档。

在理解了 Mesos 架构图之后，我们再深入一点，看看 Mesos 是怎样管理资源的。

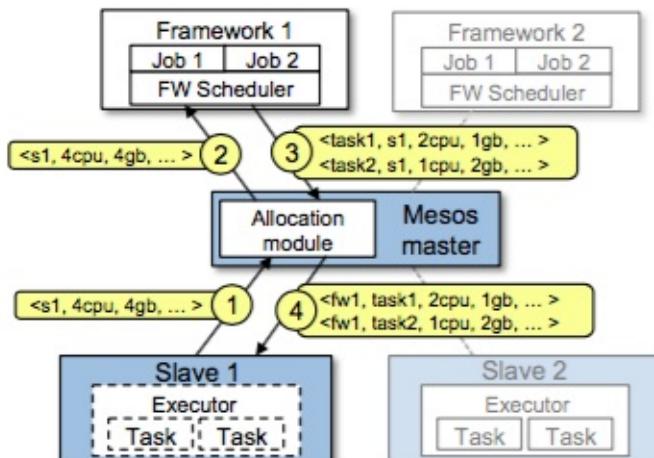


Figure 3: Resource offer example.

1. 计算节点向控制结点汇报可用资源
2. 控制结点汇集当前可用的资源，并以 `offer` 的方式发送给计算框架
3. 计算框架根据收到的资源，调度自己的任务
4. 控制结点解析计算框架发送的任务，并调度到计算节点中执行

看起来很简单，对吗？我们稍微深入一点实现细节。

汇报可用资源

如果读者观察过 `mesos-slave` 的日志文件，可以看到 `mesos-slave` 在启动时会自动获取当前机器可用的资源，包括：

- CPU 核数
- 内存大小
- 磁盘大小
- 网络端口

如下为 A 机器上 mesos-slave 启动时的相关日志示例。

```
... 省略部分日志 ...
I0403 18:14:33.822067 14747 main.cpp:272] Starting Mesos slave
I0403 18:14:33.822293 14747 slave.cpp:190] Slave started on 1)@10.23.85.234:5051
... 省略部分日志 ...
I0403 18:14:33.823786 14747 slave.cpp:354] Slave resources: cpus(*):8; mem(*):14862; disk(*):46055; ports(*):[31000-32000]
I0403 18:14:33.824726 14747 slave.cpp:390] Slave hostname: 10.23.85.234
I0403 18:14:33.824758 14747 slave.cpp:395] Slave checkpoint: true
... 省略部分日志 ...
```

从上面的日志中，我们可以知道，在 A 机器上，mesos-slave 表明自己可用的资源为

- CPU: 8 核
- 内存：14862 MB
- 磁盘：46055 MB
- 网络端口：[31000-32000]

每种资源的括号后都带有一个星号 (*)，表示这些资源是可共享的资源，在 Mesos 中，默认的资源都是共享的，也可以为计算节点指定为某些 role 预留资源。

指定计算资源通过配置 mesos-slave 的 `--resources` 参数实现，格式为：

```
name(role):value;name(role):value...
```

其中，`name` 是资源的名称，mesos 内置了以下几种资源：

- `cpus`，CPU
- `mem`，内存
- `disk`，磁盘
- `ports`，网络端口

CPU 和内存是必须资源，缺少其中任何一种都不能运行任务。例如：可以指定 A 机器的资源为：

```
cpus(foo):2;mem(foo):2048;cpus(*):2;mem(*):6144
```

这里我们只指定了两种必须的资源：CPU 和内存，其它资源将以自动检测的为准。除了 Mesos 内置的资源外，还可以指定用户自定义的资源，mesos 同样会将用户定义的资源发送给计算框架，从而调度任务。

注意：修改资源配置要在重启 mesos-slave 后才生效，不过，Mesos 在 0.23.0 版本中也引入了动态预留资源，这部分高级特性留给感兴趣的读者自行研究。

发送 Offer

计算节点在启动时，会向控制节点汇报可用资源，而控制节点则会将计算结点加入到注册表中，并且将可用资源记录在案，并且在适当的时候根据调度算法将资源以 Offer 的方式发送给上层运行的计算框架。

每个 Offer 代表一个计算节点上的可用资源，所以，为了提高效率，控制结点往往会一次性将多个 Offer 发送给一个框架，以便框架能够批量调度任务。

查看 mesos-master 的日志，我们可以看到这种事件，例如：

```
I0403 20:58:51.936903 1427 master.cpp:4967] Sending 2 offers to framework 20150714-18
1406-760551178-5050-27792-0000 (marathon) at scheduler-2ccd1a4e-0191-4c1d-be7d-b12e760
e98c5@10.23.85.218:26030
```

上面的日志显示 mesos-master 向框架 20150714-181406-760551178-5050-27792-0000 (marathon) 发送了 2 个 Offer。

在 Mesos 中，一个 Offer 就是一个计算结点在某一时刻对当前框架的所有可用资源。这里，需要注意一些限定：

- 一个计算结点，也就是说 Offer 和计算结点是一对一的关系
- 当前框架，也就是即使是在同一时刻，各个框架看到的 Offer 可能是不同的，因为计算结点可能为某些角色预留资源，配置不同的资源等
- 所有可用资源，即总是可以断定，mesos 不会将一个计算结点对某个框架可用的资源分成不同的 Offer 来发送，原因很明显，因为会导致碎片

调度任务

当计算框架收到 Offer 时，便可以根据 Offer 来调度任务了，例如：有两个任务需要运行，分别为：

- 任务 A: 2 个 CPU, 1GB 内存
- 任务 B: 1 个 CPU, 2GB 内存

收到的 2 个 Offer 为：

- Offer 1: 1.5 个 CPU, 5GB 内存
- Offer 2: 2 个 CPU, 2GB 内存

那么，调度框架如果先调度 A（可能 A 的优先级比 B 的高），会发现 Offer 1 不满足需求，所以可能首先拒绝掉 Offer 1，然后检查 Offer 2，发现可用，那么就会组装一个任务，并且声明这个任务使用了 Offer 2 的 2 个 CPU, 1GB 内存；然后，继续调度任务 B，发现 Offer 2 已经不够了，而且已经没有更多 Offer 了，所以这一轮调度就只能调度一个任务。

很显然，这里并没有最优调度，因为任务 B 是可以使用 Offer 1 的，但是 Offer 1 却被拒绝掉了。

这里只是一个简单的任务调度示例，真实的调度算法可以非常复杂，非常高效。

当计算框架完成了本轮调度后，就会把组装好的任务发送给控制结点。

启动任务

控制结点在收到任务后，会打开任务，看看这个任务使用的 Offer，并且更新相应的 Offer，减去将要被使用的资源，然后将任务发送到 Offer 中指定的计算结点，我们知道，Offer 是何计算结点一一对应的。

计算结点在收到任务后，就会按照指示，启动任务，并且为任务设置好资源限制，以免任务使用超标的资源。

接下来，控制节点需要向控制结点汇报任务的状态，如果任务完成，那么控制结点就会回收这个任务使用的资源，并且更新相应的 Offer。

这样，一个任务的生命周期就完成了，Mesos 就这样周而复始的动态的管理着整个集群的运转。

小结

本节简单的剖析了 Mesos 的工作原理，希望对读者在阅读后面的实战章节有些帮助，本节内容抛砖引玉，只揭开了 Mesos 神秘面纱的一角，更多的秘密需要读者更多实践，更多的学习去解开。

在下一章，我们将介绍几个 Mesos 生态圈中最常见的计算框架，以及一些基础用例，以便读者能更好的理解后续实战章节的内容以及创造自己的实战案例。

总结

本章简单介绍了 Mesos 的历史，以及怎样搭建一个可用于生产环境的 Mesos 集群，还介绍了 Mesos 依赖的 ZooKeeper 服务以及怎样搭建高可用的 ZooKeeper 服务。最后，分析了 Mesos 工作原理。

Mesos 框架

在 Mesos 出现之前，各个分布式计算框架都是以独占的方式使用集群资源，Mesos 的出现，使计算框架之间共享集群计算资源成为了可能，以便更好的利用集群资源，降低部署、运维成本。

目前有许多计算框架都能完美的运行在 Mesos 之上，比如：Hadoop, Spark, Storm, Chronos, Marathon, Cassandra 等等。

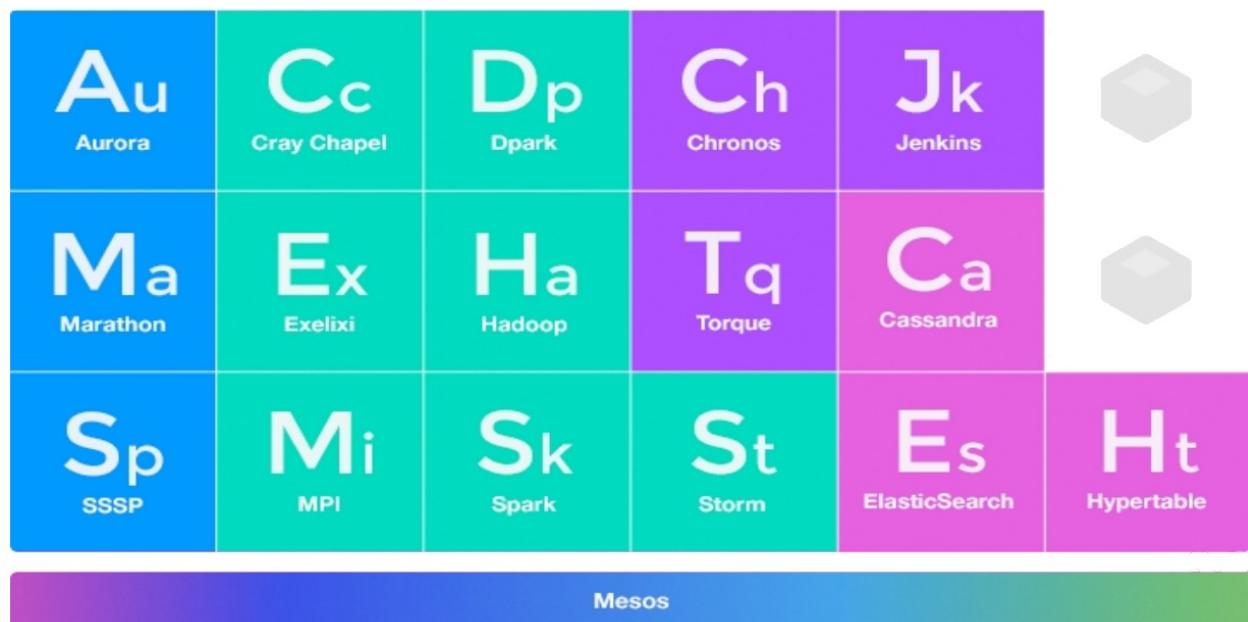
本章将首先对目前 Mesosphere 官方宣称可以完美运行在 Mesos 之上的所有框架作简要介绍，然后重点介绍几个使用比较广泛的计算框架：

- **Marathon**, 长时任务处理框架
- **Chronos**, 批处理任务处理框架
- **Spark**, 大数据处理框架

如果读者需要用到本书中未介绍的框架，可以参考项目文档动手实践，以及和开源社区交流。

Mesos 框架概览

从上一章我们知道，移植一个现有的分布式计算框架到 Mesos 平台难度并不高，所以 Mesos 生态迅速得到了其它已有分布式计算框架的支持，下图是 Mesosphere 社区文档中发布的一张框架元素周期表，大概概括了目前能够完美运行在 Mesos 之上的计算框架。



Mesosphere 官方宣称有超过 40 个计算框架支持 Mesos。

在上图中，按照颜色将框架分为了 4 类，从左到右：

- PaaS 或者长时任务框架
- 大数据框架
- 短任务或者批处理框架
- 数据存储框架

同时，Apache Mesos 官方文档也有对支持 Mesos 的计算框架的简单介绍。

PaaS 或者长时任务框架

上图中的左边第一列既是 PaaS 或者长时任务框架，包括 Aurora, Marathon 和 SSSP。

Aurora

Aurora 最初由 Twitter 开发，后来贡献给 Apache 软件基金会，目前已经成功 Apache 顶级项目。

Aurora 是专门为运行在 Mesos 之上开发的，而非移植到 Mesos 的框架，它实际上支持两类任务：长时任务和批处理任务，虽然这里将其归类为 PaaS 或者长时任务框架。

Aurora 的核心特性包括：

- 平滑的升级及回滚
- 资源配额以及多用户支持
- 强大的 DSL 描述语言
- 服务发现

Aurora 的任务配置非常灵活，强大，但也稍显复杂，所以目前在国内的用户并不多。

Marathon

Marathon 是 Mesosphere 专门为 Mesos 开发的长时任务框架，根正苗红，目前是 Mesos 平台上使用最广泛的 PaaS 框架。Marathon 目标是成为 Mesos 集群中的类似 init 进程之于操作系统的功能，能够启动长时任务，并且保证长时任务总是在线。

根据 Marathon 的特点，Mesosphere 也推荐将其它框架使用 Marathon 来运行，这样其它框架就能够更简单的实现高可用性，失败转移等特性，降低运维成本。本章将会有专门的篇幅来介绍 Marathon，所以这里就此打住。

SSSP

SSSP 即 S3 Proxy Mesos Framework 是一个 Amazon S3 存储的 Proxy，由 Mesosphere 开发并开源，由于 S3 在国内并不可用，所以这里不对 SSSP 做介绍；另外，此项目也已经停止开发近两年，所以没有必要再对其进行介绍。

大数据框架

Cray Chapel

Cray Chapel 是一种并行编程语言，最初为高性能并行计算开发，后来逐步改善兼容性，现在也能运行在常见的支持类 Linux 系统上。

Cray Chapel on Mesos 调度框架则实现了将使用 Chapel 编写的应用运行在 Mesos 集群中。

Exelixi

Exelixi 是一个将基因算法应用运行在 Mesos 集群中的框架。

MPI

MPI (Message Passing Interface) 是一种并行计算消息通信接口，主要面向高性能并行计算，MPI on Mesos 框架主要是能够让 MPI 运行在 Mesos 集群中。

Dpark

Dpark 是国内公司豆瓣开发的，使用 Python 语言重新实现了 Spark，支持 Map-Reduce 任务，并且在 GitHub 上开源。

Hadoop

Hadoop 应该是当前大数据领域炙手可热的计算框架了，开发灵感来自于 Google 公布的 Map-Reduce 论文，Hadoop 项目包含了许多子项目，是一个完整的大数据解决方案生态，最基础的莫过于分布式文件系统 HDFS 和 Map-Reduce 计算框架，Hadoop 也是一个 Apache 软件基金会的顶级项目，并且其生态圈中，有非常多的子项目同时也是 Apache 软件基金会顶级项目。

Hadoop on Mesos 项目能够将 Hadoop 运行在 Mesos 集群中，但是随着 YARN(Map-Reduce v2) 的发布，Hadoop on Mesos 的进度就转移到了怎样将 YARN 运行在 Mesos 之上，并且在 Apache 软件基金会孵化了一个项目：Apache Myriad，目前发布了 0.1 版本。

Spark

Spark 是一个 Apache 软件基金会顶级项目，是另一个非常活跃的大数据计算框架，支持 Map-Reduce 任务，得益于基于内存的计算模型，Spark 宣称运行速度在 Hadoop Map-Reduce 百倍以上；即使基于磁盘计算，速度也在 Hadoop Map-Reduce 十倍以上。

值得一提的是，Spark 可谓是 Mesos 的同门，最初都有伯克利 AMPLab 创建，Spark 发展到现在，已经不仅仅支持 Map-Reduce 任务了，还支持 Spark SQL, Spark Streaming 实时流式计算，MLlib 机器学习以及 GraphX 图计算。

Concord

Concord 是一个基于 Mesos 的实时流计算框架，和 Apache Storm 类似。

Storm

Storm 是 Apache 软件基金会的顶级项目，面向实时流计算。

短任务或者批处理框架

Chronos

Chronos 是一个目前比较活跃的 Mesos 批处理任务框架，最初由 Airbnb 开发，目前由 Mesosphere 维护，Chronos 类似于分布式的 Cron，它支持 ISO8601 标准的定时任务，以及支持优先级，同步、异步任务，任务依赖等等。

Jenkins

Jenkins 是老牌 Continuous Integration(CI) 开源软件，使用非常广泛，本书也有专门的实战案例介绍怎样使用 Jenkins + Mesos 搭建可扩展的可持续集成服务。

Torque

Torque 虽然出现在了元素周期表中，但是由于基本上处于无维护状态，所以已经不推荐使用了。

数据存储框架

ElasticSearch

ElasticSearch 更为出名的是作为 ELK(ElasticSearch, Logstash, Kibana) 的一员，ELK 被广泛应用在日志采集、处理、分析、检索、报表领域。

Cassandra

Cassandra 同样是一个 Apache 软件基金会的顶级项目，提供分布式，可先行扩展的数据存储服务，并且支持跨数据中心的冗余。

Hypertable

Hypertable 是一个开源的分布式数据库软件，设计理念来源于 Google Bigtable，着眼于可扩展性，可扩展性也是目前 NoSQL 相对于 SQL 数据库的主要优点之一，但随之妥协的往往是稍差的性能和一致性。

Hypertable 号称在扩展性，性能以及一致性方面都做得非常出色。

Marathon

Marathon 是 Mesosphere 开发的支持长时任务（long running service）的框架，只能运行在 Mesos 上。Mesosphere 是一家初创公司，由 Mesos 作者等人创办，所以 Marathon 也算是 Mesos 生态中标准的长时任务处理框架了。

首先，我们来了解一下这里的长时任务是什么，简单的说：长时任务在这里是指一旦任务运行起来，就不期望其结束。如果读者了解 Linux 系统，那么 Linux 系统中的守护进程（daemon）就是一种长时任务，总是被期望在系统运行期间都在运行。

那么，在实际生产环境中，哪些任务是我们说的长时任务呢？最常见的莫过于 web 服务了。除了 web 服务外，也可以是一些 RPC 服务；另外，长时任务不仅可以是服务，也可以是一些 worker。

Marathon 为什么适合运行长时任务？因为它为长时任务提供了许多辅助特性，包括但不限于：

- 失败自动重启
- 健康检查
- 横向扩展
- 服务发现

所以，一旦将长时任务运行在 Marathon 上，你总是可以期望你的任务总是在线，而不用担心计算结点故障导致任务失败。

本节将简单介绍 Marathon 的特性，用法以及运行一个网页版的 2048 游戏，更多的高级特性需要读者继续挖掘。

Marathon 简介

Marathon 是 Mesosphere 专门为 Mesos 开发的长时任务处理框架，并且在 GitHub 上开源，利用 Marathon 以及一些其它开源软件，加上一些本地开发，结合当前流行的容器解决方案 Docker，就可以搭建一个基于 Mesos 的 PaaS(Platform as a Service) 平台。

当然，Docker 并不是必须的，但是由于 Docker 能够很好的解决构建、打包、分发和运行的问题，所以加入 Docker 也就成了不二之选，Mesos, Marathon 都原生支持 Docker，所以使用起来非常方便。

截止本书写作之时，Marathon 的最新版本为 0.13.0，支持以下特性：

- HA(High Availability)
- Constraints
- 服务发现和负载均衡
- 健康检查
- 事件机制
- Web UI
- RESTful API
- Basic Auth 以及支持 SSL
- Metrics

下面将对以上特性逐一介绍。

HA(High Availability)

Marathon 的 HA 实现非常简单，Marathon 将所有数据持久化到 ZooKeeper 服务中，自身是一个无状态的服务，而各个服务实例之间通过 ZooKeeper 实现 Leader 选举，任意时刻，只有最多一个服务实例作为 Leader，其它实例则作为 Follower，任何到达 Follower 的请求都将被转发到 Leader。所以，任意时刻，只要有一个 Marathon 服务实例存活，整个 Marathon 服务都是可用的。

所以，在生产环境中，要实现 Marathon 服务的高可用性，至少需要搭建 2 个 Marathon 服务实例组成一个服务。

Constraints

在生产环境中，保持集群中所有结点的一致性显然是不可能的事，总是有一些特性能够将 Mesos 计算结点区分开来，例如：

- 结点所在的机架
- 结点的网络带宽
- 结点是否可以访问外网，通过何种方式
- 结点所在机房的运营商
- 结点是物理机还是虚拟机
- 等等

所以，Marathon 使用 **Constraints** 特性来支持将任务运行在具有指定特性的计算结点上。

服务发现和负载均衡

想象一下在传统的 Web 服务部署中，通常是将服务直接部署在固定的一台或多台物理机或者虚拟机上，然后，将这些服务器的 IP 地址绑定到负载均衡器中，对外提供服务。但是，当服务器故障时怎么办呢？需要故障处理和恢复，如果故障的服务器不能恢复，需要部署新的服务器并加入到负载均衡器中，避免影响服务的稳定性。

想象一下，将这样的 Web 服务通过 Marathon 运行在 Mesos 集群中会是什么样？Marathon 会自动将服务的多个实例运行在一些计算结点上，当结点或者实例故障时，会自动重启一个新的实例，那么这里存在的问题是在任意时刻怎样知道服务实例运行的地址呢？

服务发现和负载均衡特性就是为了解决这个问题，Marathon 能够为长时任务生成 HAProxy 配置文件，如果使用 HAProxy 作为负载均衡器，那么服务实例的故障是完全不用运维的。虽然 Marathon 没有提供其它负载均衡器的支持，但是已经不乏这样的开源软件，例如：Bamboo, Consule 等。

健康检查

由于 Marathon 专为长时任务设计，并且支持自动重启故障的任务，任务的故障有很多种，计算结点导致的故障可以由 Mesos 通知，但是如果是任务内部的异常，Mesos 则无能为力了，例如：Web 服务正在运行，但是已经不能响应服务请求了，那么有什么机制能够发现并处理这种异常吗？

熟悉 HAProxy, Nginx 的读者应该知道，它们可以通过检测后端服务器的服务接口来自动屏蔽异常的服务器，Marathon 也实现了类似的机制，叫做健康检查，能够自动检查服务实例并且在发现异常后做出响应，例如：杀掉异常的服务实例并且重新启动一个新的服务实例。

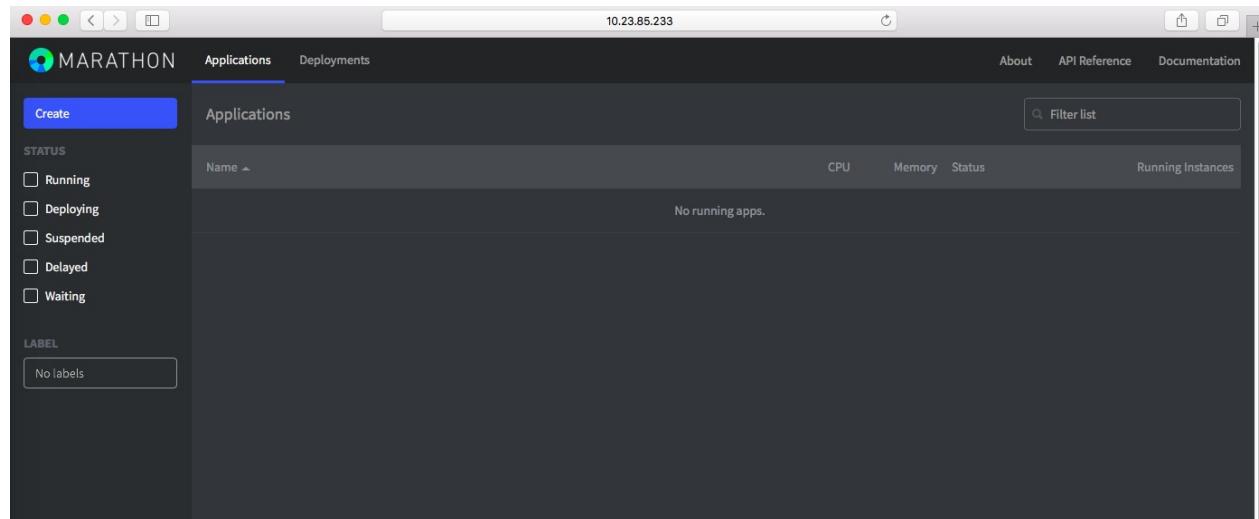
事件机制

事件机制为开发者提供了无限的可能，开发者可以通过注册到 Marathon 的事件总线，及时知道自己感兴趣的事件，从而做出相应。

例如：通过事件机制，可以监听到服务实例的变化，自动配置负载均衡器。

Web UI

Marathon 提供了实用的 Web UI，能够方便用户管理长时任务，实现任务的增、删、查、改操作。下图是一个 Marathon Web UI 截图。



RESTful API

Marathon 不仅提供了 Web UI，还提供了 RESTful API，对于开发者来说，良好设计的 API 更值得拥有，你可以通过 Marathon API 实现通过 Web UI 能够实现的所有功能，甚至一些 Web UI 不支持的功能。例如：强制重新部署。

Basic Auth 以及支持 SSL

Marathon 实现了 Basic Auth，Basic Auth 是一种简单的用户名密码认证方式，目前 Marathon 只支持单用户，所有任务都只能属于这个用户，所以本质上，如果有多个用户在一个 Marathon 服务上创建了任务，那么是有可能误操作的。

在通信安全方面，Marathon 支持 SSL 加密。

Metrics

通过 Metrics，用户可以很方便的知道 Marathon 的运行情况，例如：任务数量，使用的资源等等。

搭建Marathon服务

在编写本书时，Marathon 最新的稳定版本是 0.13.0，所以这里将以 Marathon 0.13.0 版本为例来搭建 Marathon 服务。

准备环境

Marathon 是运行在 Mesos 之上的长时任务处理框架，并且依赖 ZooKeeper 服务来持久化数据，所以在开始搭建 Marathon 服务之前，首先需要有可用的 Mesos 集群以及 ZooKeeper 服务。

这里我们将使用在前一章中搭建的 Mesos 集群以及 ZooKeeper 服务，虽然 Mesos 也是基于此 ZooKeeper 服务，但是这里纯属巧合，Marathon 可以使用任何其它可用的 ZooKeeper 服务。

所以，Mesos 集群的地址为：

```
zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos ; ZooKeeper 服务地址  
为： zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181 。
```

这里，我们选择在 10.23.85.233 上搭建 Marathon 服务，也就是 Mesos 集群中的其中一台，我们在上一章中称为 A 的机器，这里我们继续称之为 A。

之所以选择 A，并没有什么特殊的原因，读者也可以选择在 B 或者 C 上安装 Marathon，方法和这里相同，只是需要注意的是：安装 Marathon 首先要安装 Mesos，因为 Marathon 依赖于 Mesos 库，所以不能在一台没有安装 Mesos 的机器上安装 Marathon。

下载 Marathon

首先，到 Github 上下载 Marathon，下载地址为：

<https://github.com/mesosphere/marathon/releases/tag/v0.13.0>。

假设将 Marathon 下载到了 ~/Downloads 目录下，或者执行下面的命令进行下载：

```
$ cd ~/Downloads  
$ curl -O http://downloads.mesosphere.com/marathon/v0.13.0/marathon-0.13.0.tgz
```

使用下面的命令解压下载好的压缩包

```
$ tar xzf marathon-0.13.0.tgz
$ cd marathon-0.13.0
$ ls
bin Dockerfile docs examples LICENSE README.md target
```

执行解压后的 `bin/start` 脚本即可启动 marathon，如下所示：

```
$ ./bin/start --master zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos \
> --zk zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/marathon
```

上面命令中有两个参数：

- `--master`，指定 Mesos 集群控制结点服务地址
- `--zk`，指定 ZooKeeper 服务地址

上面的命令可能会报如下错误：

```
MESOS_NATIVE_JAVA_LIBRARY is not set. Searching in /usr/lib /usr/local/lib.
MESOS_NATIVE_LIBRARY, MESOS_NATIVE_JAVA_LIBRARY set to ''
Exception in thread "main" java.lang.UnsupportedClassVersionError: mesosphere/marathon
/Marshaller : Unsupported major.minor version 52.0
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:803)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:482)
```

`Unsupported major.minor version 52.0` 这个错误表示 Marathon 是使用 Java 1.8 编译的，并且不兼容老版本，而本机上安装的 Java 版本为 1.7.0，所以执行时才会出现错误。

解决办法就是升级本机的 Java 版本，或者从源码编译安装 Marathon，升级 Java 版本很简单，直接通过 YUM 从 CentOS 软件源中安装即可。

```
# yum install -y java-1.8.0-openjdk
```

安装完成后，再次运行上面的命令，可以看到，已经不再报这个错误了，但是，可能会报下面的错误。

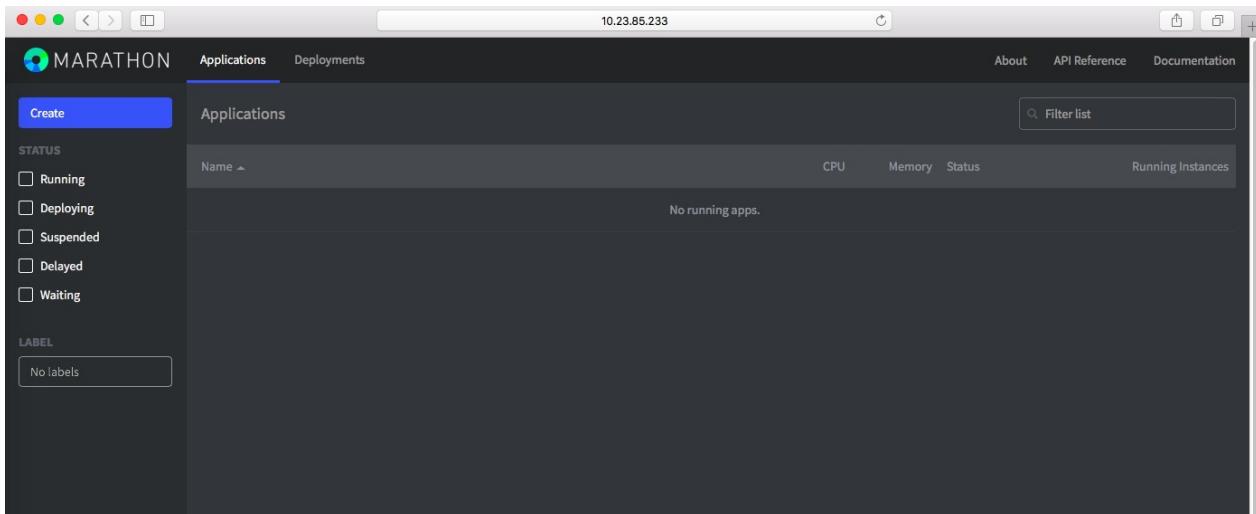
```
Failed to load native Mesos library from
Exception in thread "pool-1-thread-2" java.lang.UnsatisfiedLinkError: Expecting an absolute path of the library:
    at java.lang.Runtime.load0(Runtime.java:806)
    at java.lang.System.load(System.java:1086)
    at org.apache.mesos.MesosNativeLibrary.load(MesosNativeLibrary.java:159)
    at org.apache.mesos.MesosNativeLibrary.load(MesosNativeLibrary.java:188)
```

这是因为 Marathon 没有找到 Mesos 库导致的，修复这个问题也很简单，设置一个环境变量，再启动 Marathon 即可，如下：

```
$ MESOS_NATIVE_JAVA_LIBRARY=/usr/lib64/libmesos.so ./bin/start --master zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos --zk zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/marathon
```

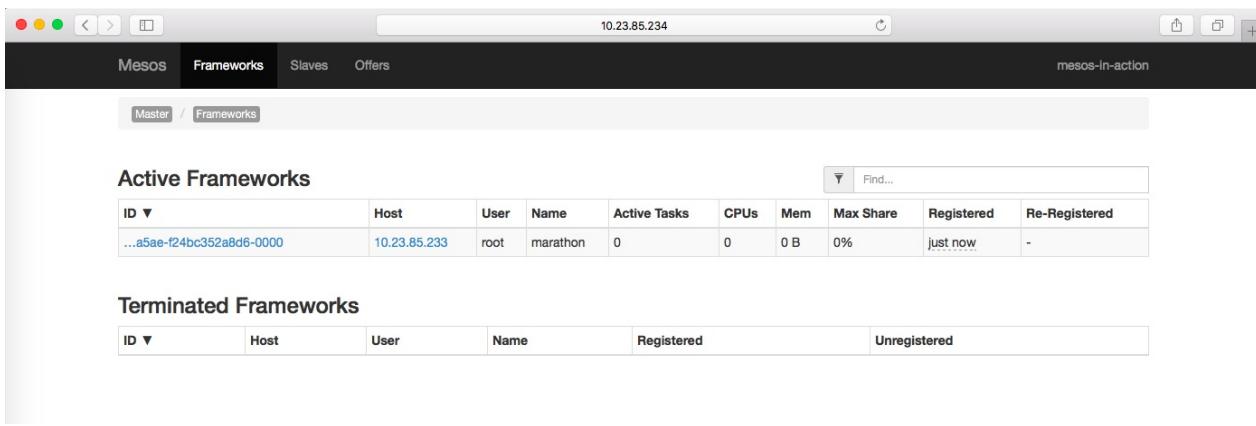
如果用户直接从 Mesosphere 软件源中安装 Mesos 包，则不会发生这个错误，因为 Mesosphere 的安装包将 Mesos 库安装到了 /usr/local/lib 下。

启动完成后，打开浏览器访问本机的 8080 端口，就可以看到 Marathon UI 了，如下图所示：



Marathon 启动后，会将自己注册到 Mesos 集群中，并且将数据持久化到 --zk 指定的 ZooKeeper 服务中的指定地址。

现在，打开 Mesos 服务的 Web UI，就可以看到刚刚注册的 Marathon 服务了，如下图所示：



到现在为止，一个可用的 Marathon 服务就搭建起来了，非常简单，Marathon 还有一些可配置的参数，这里介绍一些读者可能会用到的参数。

Marathon 参数

marathon 只有一个必须的参数，即 `--master`，以便知道 Mesos 的服务地址。其它一些比较常用的可选参数如下表所示：

参数	默认值	示例
<code>--zk</code>	无	<code>--zk=zk://host1:port1,host2:port2,host3:port3</code>
<code>--[disable_]checkpoint</code>	<code>--checkpoint</code>	<code>--checkpoint</code>
<code>--failover_timeout</code>	604800	<code>--failover_timeout=86400</code>
<code>--hostname</code>	主机的 hostname	<code>--hostname=10.23.85.233</code>
<code>--mesos_role</code>	无	<code>--mesos_role=marathon</code>

--default_accepted_resource_roles	所有资源	--default_accepted_resource_roles=mara
--task_launch_timeout	300000, 5分钟	--task_launch_timeout=1800000
--event_subscriber	无	--event_subscriber=http_callback
--http_address	所有网络地址	--http_address=10.23.85.233
--http_credentials	无	--http_credentials=admin:adminpass
--http_port	8080	--http_port=80
--http_max_concurrent_requests	无	--http_max_concurrent_requests=100

Marathon 的参数还可以通过环境变量来配置，这和 Mesos 类似，只需要将参数全部大写，并且加上 MARATHON_ 前缀，例如：

- --zk 参数可以通过环境变量 MARATHON_ZK 来指定
- --mesos_role 可以通过环境变量 MARATHON_MESOS_ROLE 来指定

需要注意的是，如果一个参数同时通过环境变量指定，又通过命令行参数指定，那么命令行参数将会覆盖环境变量。

服务的高可用性

在上一节中，我们了解了 Marathon 高可用性实现方案，Marathon 将所有需要持久化的数据都存储在 ZooKeeper 服务中，并且多个实例之间由 ZooKeeper 来实现 Leader Election。所以，实现高可用性不需要任何配置即可完成。

但是，对于 Marathon 用户来说，Marathon 这种高可用性却不是透明的，因为，当 Marathon Leader 故障时，新的 Leader 提供的服务地址和以前的 Leader 服务地址不一样，所以用户需要更改访问的地址。

所以，对用户透明的高可用性就非常有需要了，特别是对于通过 HTTP 协议来访问的用户来说。

下面将介绍两种实现透明高可用性的方案：

- 虚拟 IP 方案
- 负载均衡方案

虚拟 IP 方案

虚拟 IP 方案基于 VRRP (Virtual Router Redundancy Protocol) 协议，VRRP 的工作原理如下：



简单的说，有两个设备同时提供一个虚拟 IP，两个设备采用主备的方式工作，任意时间只有一个设备提供虚拟 IP，当主机点故障时，备用结点提供虚拟 IP，当主节点恢复时，主节点提供虚拟 IP。所以，主节点总是优先。

Linux 下常见的实现虚拟 IP HA 的软件有：

- keepalived
- ucarp
- heartbeat

回顾一下 Marathon HA 的工作原理可以知道，这种主备工作方式的 HA 并不适合 Marathon，包括后面将要介绍的 Chronos，它具有和 Marathon 相同的 HA 实现方式。

原因是在 Marathon 服务中，所有跟随者实例都需要将请求转发给 Leader，所以，当 Marathon Leader 运行在 keepalived 或者 ucarp 中的备用结点上时，所有的请求都需要经过转发才能到达 Marathon Leader，降低了效率。

举个例子：假设有 A, B 两台服务器，使用 keepalived 实现了 IP HA，并且配置了 A 服务器作为主结点，B 作为备结点。同时在 A, B 两台服务器上搭建了 Marathon 服务，服务启动时，A 结点上的 Marathon 作为 Leader，所以所有通过虚拟 IP 到达的请求都直接由 Marathon

Leader 处理，但是，假设某一时刻服务器 A 故障宕机，显然，服务器 B 上的 Marathon 会作为新的 Marathon Leader 并且所有通过虚拟 IP 到达的请求都将直接到达 B，此时所有的请求也是直接由 Marathon Leader 处理的，看起来一切工作的非常好。

但是，过了一段时间后，服务器 A 恢复了，重新上线，由于 A 被配置成了虚拟 IP 的主结点，所以当 A 在线时，所有对虚拟 IP 的访问都将发往 A，但是，此时服务器 A 上的 Marathon 并非 Leader，所以，Marathon 收到请求后，需要将请求转发给服务器 B 上的 Marathon Leader，直到下次 B 上的 Marathon 故障，将服务器 A 上的 Marathon 重新选举为 Leader。

简单的说，只有当虚拟 IP 主结点和 Marathon Leader 是同一个结点时，才能避免服务转发。

负载均衡方案

负载均衡器顾名思义是用来实现各个服务器之间负载均衡的设备，包括软硬件设备，在软件定义以及开源软件大行其道的今天，可选的开源负载均衡软件也不少，最常见的有：

- LVS(Linux Virtual Server)
- HAProxy
- Nginx

负载均衡器由于可以通过心跳来检查后端服务器的健康状况，所以，也能够在实现负载均衡的同时，也实现服务的高可用性，负载均衡器能够根据配置的心跳检测策略检查后端服务器，并且避免将流量继续发往故障的后端服务器。

使用负载均衡器和使用 IP HA 方式相比，几乎总是有一半的流量会通过 Marathon 跟随结点转发到 Marathon Leader 结点上，不会太坏，也不会太好。

当然，这里介绍的是最简单的配置情况下，读者可以通过一些编程，实现动态配置，从而避免转发，这里不再赘述，留给读者自行研究。

运行 Marathon 任务

在搭建了 Marathon 框架后，本节将介绍怎样提交一个任务到 Marathon，通常，我们将 Marathon 任务称作为 App，Application 的缩写。

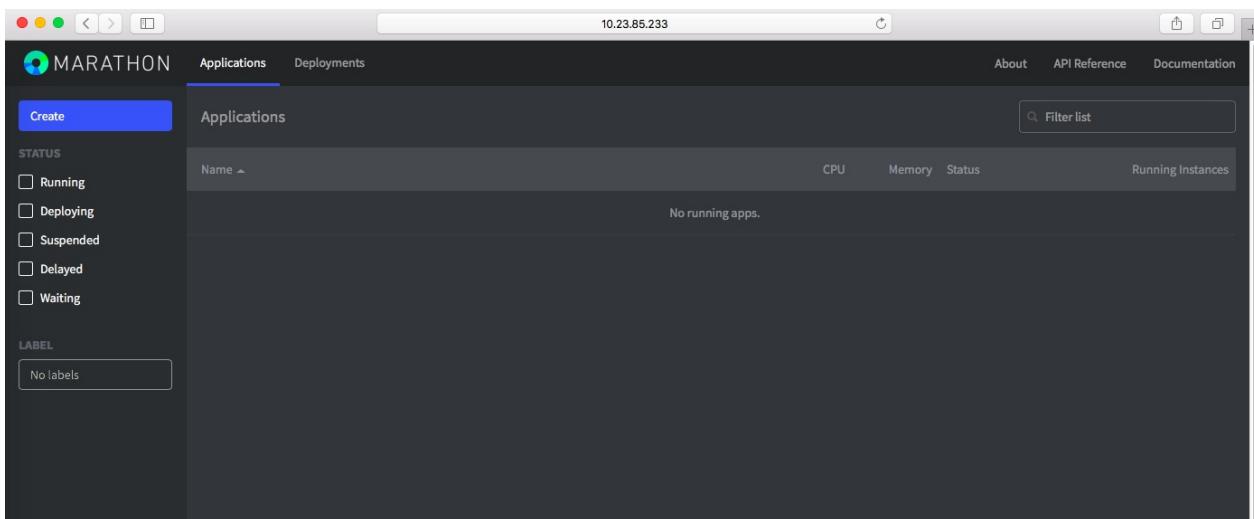
本节将首先介绍怎样使用 Marathon 启动一个 Hello Marathon 任务，然后介绍怎样运行基于 Docker 的任务：怎样搭建一个 Web 版 2048 游戏。

Hello Marathon

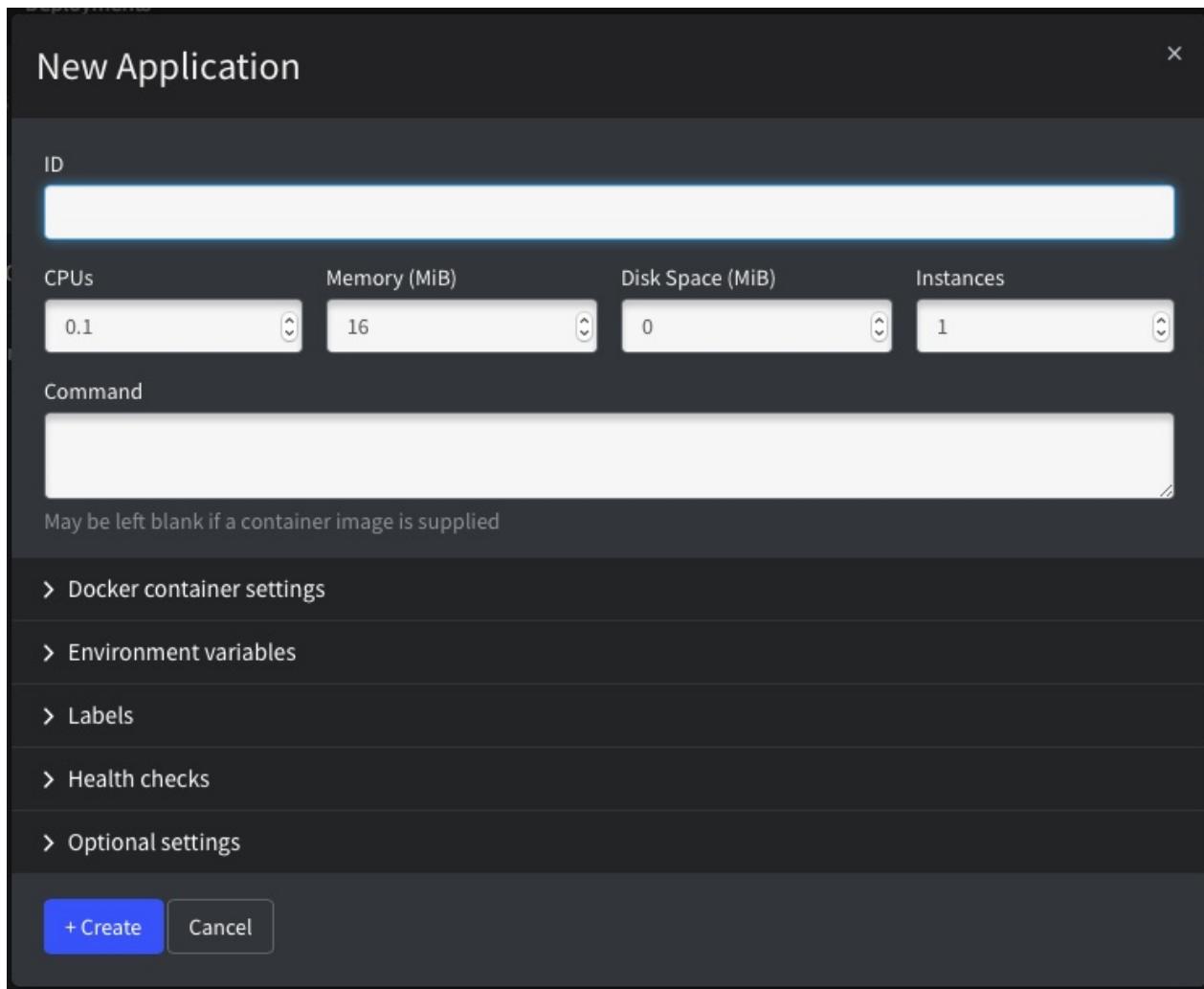
这里将创建一个 hello-marathon App 来演示怎样通过 Marathon Web 界面创建 App。

Marathon Web 顾名思义就是 Marathon 的 Web 界面，通过 Marathon Web 界面，可以了解很多有用的信息，例如：有多少 App 正在运行，各个 App 的实例列表，运行状态等等。不仅如此，还可以通过 Web 界面实现一些对 App 的控制，例如：调整实例数，暂停 App，删除 App，创建 App 等。

用浏览器打开已经启动的 Marathon 服务 (<http://10.23.85.233:8080>)，如下图所示：



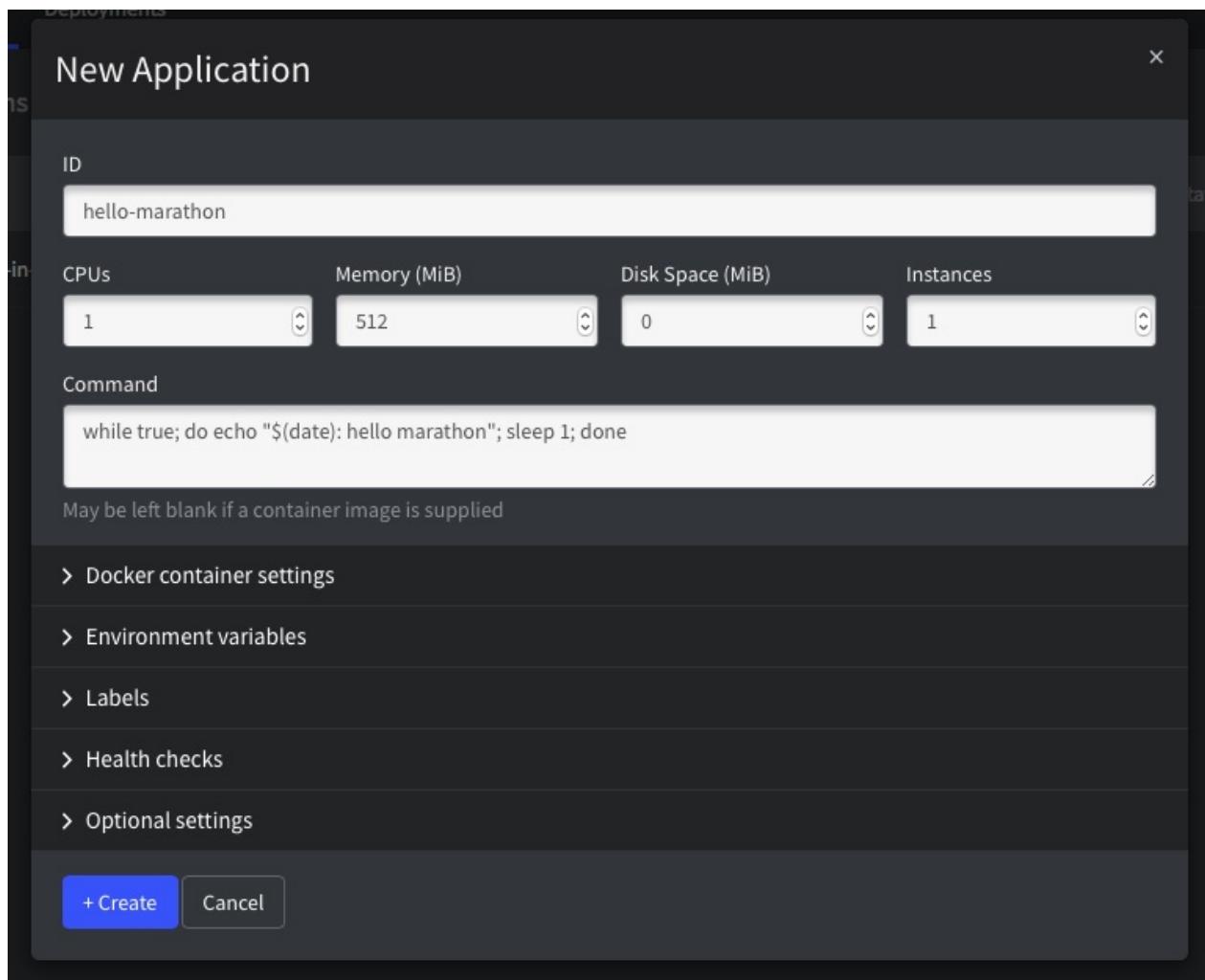
点击左上角的 **Create** 按钮创建一个新的 App，如下图所示：



在新建页面，可以配置很多应用的选项，主要包括以下几部分：

- 基本信息，容器资源配置等
- Docker 容器配置，如果使用了 Docker 来封装应用的话
- 环境变量，会被传递给应用
- 标签，方便过滤，归类等
- 健康检查
- 其它可选配置

为了创建 `hello-marathon`，我们只需要输入几个必须的配置即可，如下图所示：



输入完成后，点击 **+Create** 完成 App 的创建，现在，在 Marathon Web 上，可以看到刚才创建的 `hello-marathon` 已经处于 `Running` 状态了，如果用户操作足够快，应该还能看到在 `Running` 之前，还有一个 `Deploying` 状态；其对应于 Mesos 中的 `Staging` 状态，表示正在启动任务。

Mesos 的任务具有以下几个状态：

- `Staging`，表示任务已经收到，正在准备启动
- `Running`，任务已经启动
- `Failed`，任务失败
- `Lost`，任务丢失
- `Killed`，被杀死

在 Marathon 界面，点击 `hello-marathon` 可以看到这个 App 的详情，在 `Tasks` 卡片页可以看到当前 App 的实例；在 `Configuration` 卡片页可以看到任务配置的详情，历史版本等信息，如下图所示：

ID	Status	Version	Updated
hello-marathon.52583386-1b7e-11e6-98da-525400fd9fbc 10.23.85.234:31276	Started	8 minutes ago	May 16, 2016 at 23:53:24 GMT+8

另外，在上方还有几个控制按钮：

- Scale Application, 调整 App 运行实例数量
- Restart, 重启 App 所有实例
- Suspend, 挂起 App, 相当于杀死所有正在运行的实例，但是保留 App 的配置
- Destroy App, 彻底删除 App

从 Tasks 中，可以看到，hello-marathon 启动了一个任务，运行在 10.23.85.234 上，那么，怎样看到 hello-marathon 的输出呢？hello-marathon 任务执行了如下任务：

```
while true; do echo "$(date): hello marathon"; sleep 1; done
```

任务的效果是每隔 1 秒钟，输出 hello marathon 字符串，所以我们应该能够在任务进程的 stdout 看到输出。

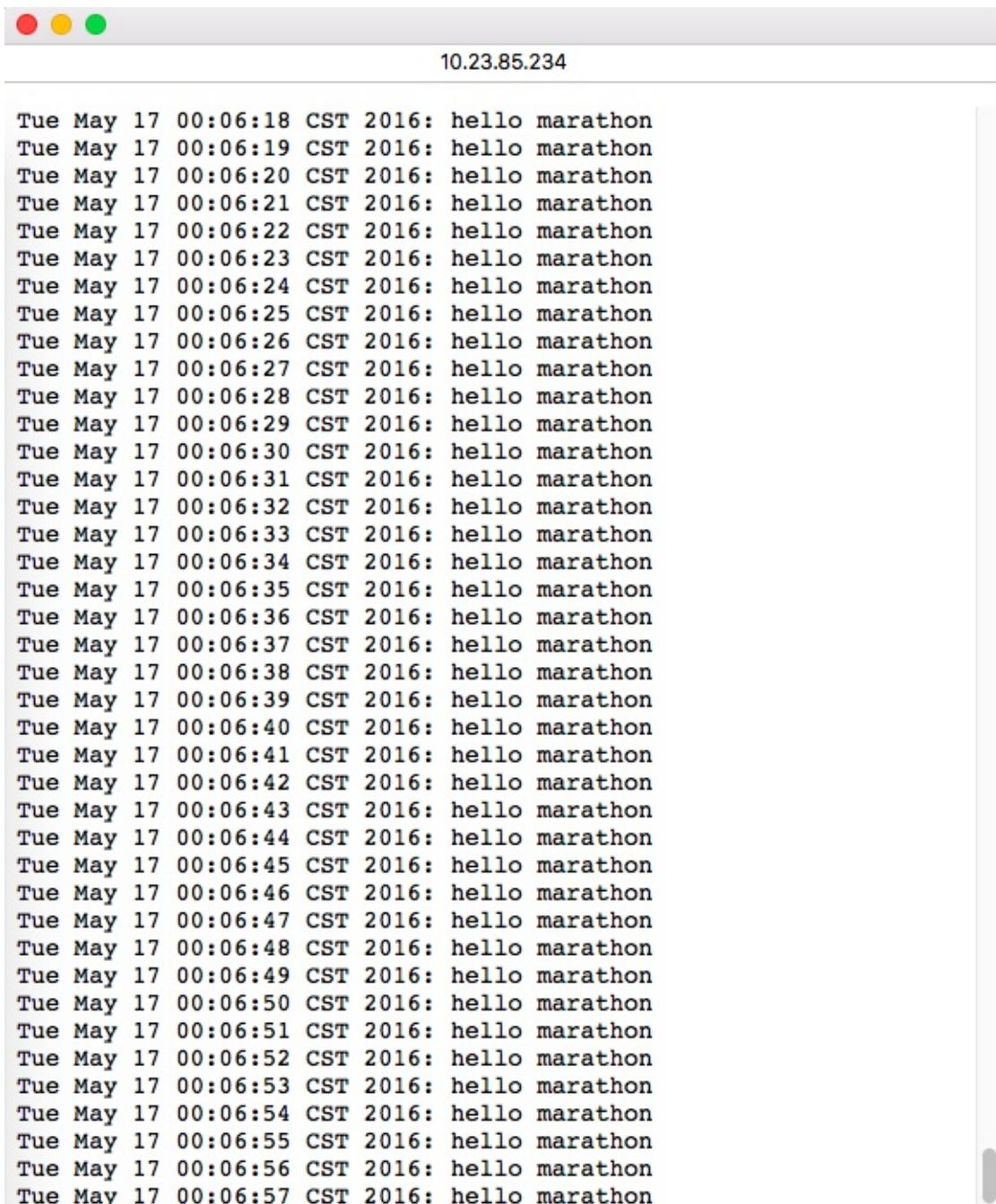
现在我们打开 Mesos 集群页面，在 Slaves 页面，找到这个计算节点，然后进入结点页面，找到 hello-marathon 这个任务，如下图所示：

ID	Name	Source	Active Tasks	Queued Tasks	CPUs (Used / Allocated)	Mem (Used / Allocated)
hello-marathon.52583386-1b7e-11e6-98da-525400fd9fbc	Command Executor (Task: hello-marathon.52583386-1b7e-11e6-98da-525400fd9fbc (Command: sh -c 'while true; ...'))	hello-marathon.52583386-1b7e-11e6-98da-525400fd9fbc	1	0	0 / 1.1	3 MB / 544 MB

点击任务后面的 Sandbox 链接，可以看到当前任务运行的目录，这里有两个文件：

- stdout, 任务的 stdout 输出会被重定向到这个文件
- stderr, 任务的 stderr 输出会被重定向到这个文件

所以，打开 stdout 文件的内容，就可以看到 hello-marathon 输出了，如下图所示：

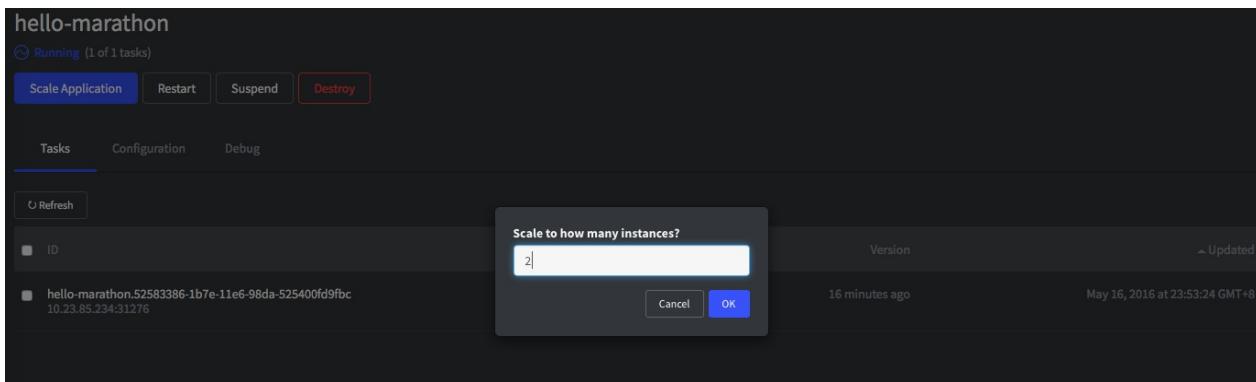


The screenshot shows a terminal window with three colored window control buttons (red, yellow, green) at the top. The title bar displays the IP address "10.23.85.234". The main content area contains a log of "hello marathon" tasks running from May 17, 2016, at 00:06:18 CST to 00:06:57 CST. Each log entry consists of a timestamp followed by the message "hello marathon".

```
Tue May 17 00:06:18 CST 2016: hello marathon
Tue May 17 00:06:19 CST 2016: hello marathon
Tue May 17 00:06:20 CST 2016: hello marathon
Tue May 17 00:06:21 CST 2016: hello marathon
Tue May 17 00:06:22 CST 2016: hello marathon
Tue May 17 00:06:23 CST 2016: hello marathon
Tue May 17 00:06:24 CST 2016: hello marathon
Tue May 17 00:06:25 CST 2016: hello marathon
Tue May 17 00:06:26 CST 2016: hello marathon
Tue May 17 00:06:27 CST 2016: hello marathon
Tue May 17 00:06:28 CST 2016: hello marathon
Tue May 17 00:06:29 CST 2016: hello marathon
Tue May 17 00:06:30 CST 2016: hello marathon
Tue May 17 00:06:31 CST 2016: hello marathon
Tue May 17 00:06:32 CST 2016: hello marathon
Tue May 17 00:06:33 CST 2016: hello marathon
Tue May 17 00:06:34 CST 2016: hello marathon
Tue May 17 00:06:35 CST 2016: hello marathon
Tue May 17 00:06:36 CST 2016: hello marathon
Tue May 17 00:06:37 CST 2016: hello marathon
Tue May 17 00:06:38 CST 2016: hello marathon
Tue May 17 00:06:39 CST 2016: hello marathon
Tue May 17 00:06:40 CST 2016: hello marathon
Tue May 17 00:06:41 CST 2016: hello marathon
Tue May 17 00:06:42 CST 2016: hello marathon
Tue May 17 00:06:43 CST 2016: hello marathon
Tue May 17 00:06:44 CST 2016: hello marathon
Tue May 17 00:06:45 CST 2016: hello marathon
Tue May 17 00:06:46 CST 2016: hello marathon
Tue May 17 00:06:47 CST 2016: hello marathon
Tue May 17 00:06:48 CST 2016: hello marathon
Tue May 17 00:06:49 CST 2016: hello marathon
Tue May 17 00:06:50 CST 2016: hello marathon
Tue May 17 00:06:51 CST 2016: hello marathon
Tue May 17 00:06:52 CST 2016: hello marathon
Tue May 17 00:06:53 CST 2016: hello marathon
Tue May 17 00:06:54 CST 2016: hello marathon
Tue May 17 00:06:55 CST 2016: hello marathon
Tue May 17 00:06:56 CST 2016: hello marathon
Tue May 17 00:06:57 CST 2016: hello marathon
```

并且，保持这个文件打开，文件内容会自动更新，实时查看到任务的 stdout，非常棒！

好了，现在已经通过 Marathon Web 界面创建了一个 App，并且这个 App 只有一个实例在运行，那么，试试运行两个实例呢，只需要点击在 Marathon Web 界面上点击 Scale Application，然后输入数字 2 即可，可以看到，很快就会有两个 hello-marathon 实例在运行了，如下图所示：



同样，可以通过 `Suspend` 来暂停 App，这里的暂停不是让已经在运行的两个实例进程暂停，而是整个 App 意义上的暂停，即停止所有正在运行的实例，所以任务会被杀死；但是，可以很方便的恢复运行，`Destroy` 不仅会杀死所有正在运行的实例，还会将 App 配置一起删除。

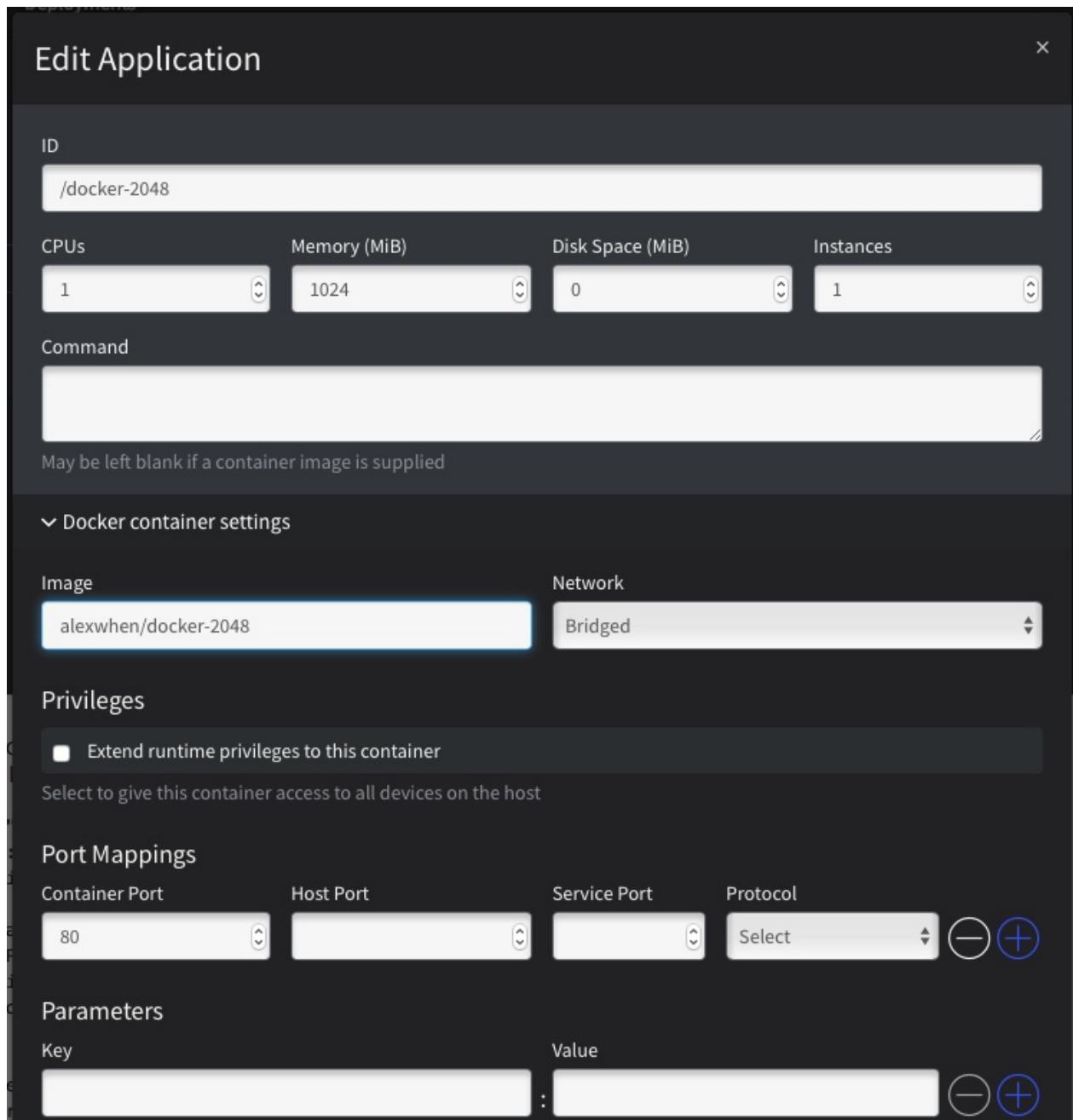
Hello Docker-2048

`hello-marathon` 实际上是将任务直接运行在 Mesos 计算结点上的，运行环境需要由计算结点来保证，例如：`hello-marathon` 运行了 shell 脚本，那么计算结点上必须有 `shell` 环境；如果 App 要依赖 `python` 环境，`perl` 环境，`PHP` 环境，`Go` 环境等等，那么可以想见，为了运行任务，计算节点上必须同时支持所有需要的环境；计算结点将非常臃肿，甚至，如果任务使用的环境有冲突，那么这时计算结点就无能为力了。

Docker 一问世，就掀起了一场云计算革命，Docker 能够将任务运行在隔离的环境中，各自任务都将自己的依赖环境全部打包到 Docker 镜像中，所以不再对底层的计算结点强依赖；相反，只需要底层计算结点支持运行 Docker 即可。

Mesos 和 Marathon 也原生对 Docker 进行了支持，所以，用户可以很方便的将 Docker 任务通过 Marathon 调度到 Mesos 集群中执行。这里将创建一个基于 Docker 的 Marathon App：`docker-2048`。

同样，在 Marathon WEB 界面中，点击 **Create** 按钮来创建新的 App，这里输入如下图所示参数：



虽然这里还有很多其它配置项，但是只需要输入上图中的几项即可，非常简单，和 `hello-marathon` 相比，这里有以下几点不同：

- 没有输入 **Command** 参数
- 输入了一个 **Docker Image**，表示将启动这个 Docker 镜像
- 选择了使用 **Bridged** 模式的网络，默认为无
- 一个容器端口参数 80

以上参数表示：在启动这个应用时，从 Docker 镜像启动，并且使用 Bridged 网络模式将容器内部的 80 端口随机映射到主机上。

输入完成后，点击 **+Create** 按钮，创建应用，稍等一段时间后，可以看到，任务已经启动了，如下图所示：

The screenshot shows the Marathon web interface. On the left, there's a sidebar with 'Create' and filters for 'STATUS' (Running, Deploying, Suspended, Delayed, Waiting) and 'LABEL'. The main area is titled 'Applications' and lists two entries:

Name	CPU	Memory	Status	Running Instances
docker-2048	1.0	1 GiB	Running	1 of 1
hello-marathon	2.0	1 GiB	Running	2 of 2

打开 `docker-2048` 应用详情页面，找到 Task 所在的主机，如下图所示：

A screenshot of a Docker task details page. It shows a single task named `docker-2048.2bec0654-1b84-11e6-98da-525400fd9fbc` with IP `10.23.85.234:31046`. The task was started 22 minutes ago and is running since May 17, 2016 at 00:35:17 GMT+8.

点击任务下面的主机端口号，将在新的浏览器页面中打开该地址，如下图所示：

A screenshot of a browser window showing a 2048 game. The URL in the address bar is `10.23.85.234`. The game board has two `2` tiles. The top right corner of the browser window shows the port number `2048`.

可以看到，一个基于 Web 的 2048 游戏已经运行起来了，酷！

注意：考虑到国内特殊情况，如果不能访问 **alexwhen/docker-2048** 镜像，那么任务将不能启动

使用 API 来创建 App

Marathon 提供了比 Web 界面功能更强大的 RESTful API，相应地，Marathon Web 界面实际上也通过 Marathon API 来管理 App。

Marathon API 采用 JSON 格式来封装数据，如下为 `docker-2048` 的 JSON 数据，我们将其保存在一个名为 `docker-2048.json` 的文件中。

```
{
  "id": "docker-2048-created-by-api",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "alexwhen/docker-2048",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "hostPort": 0 }
      ]
    }
  },
  "cpus": 0.5,
  "mem": 512.0,
  "instances": 1
}
```

可以看到，`docker-2048.json` 内容非常简单，除了之前已经在 `hello-marathon` 中使用过的几个参数：

- `id`, App 唯一标识
- `cpus`, 申请使用的 CPU
- `mem`, 申请使用的内存
- `instances`, 实例数

这里多了另一个参数：`container`，它又有几个参数组成：

- `type`, 指定 `container` 的类型，这里是 `DOCKER`
- `docker`, 指定 `docker` 的一些参数
 - `image`，指明使用的 Docker 镜像
 - `portMappings`，指明端口映射规则，`containerPort` 表示容器内部端口，`hostPort` 表示映射到计算结点的端口，0 表示随机分配

Marathon API 中关于 App 的定义还有许多参数，这里只介绍了最简单的运行 Docker 任务的几个必须参数，其它参数后面将分类做简单介绍。

现在，使用任何支持传递 JSON 数据的客户端都可以向 Marathon 提交这个 App 了，只需要遵守 Marathon 定义的 API 格式。

下面使用 `curl` 来提交 `docker-2048.json` 到 Marathon 以便创建 `docker-2048` 这个 App，创建 App 的 API 格式如下：

DESCRIPTION

Create and start a new application.

Note: This operation will create a deployment. The operation finishes, if the deployment succeeds. You can query the deployments endpoint to see the status of the deployment.

SECURITY SCHEMES

Anonymous

BODY

application/json

Example:

```
{
  "id": "/tools/docker/registry",
  "instances": 1,
  "cpus": 0.5,
  "mem": 4096,
  "disk": 0,
  "executor": ""
}
```

Try it

AUTHENTICATION

Security Scheme

Anonymous

HEADERS

QUERY PARAMETERS

BODY

application/json

```

1 | 
2 |   "id": "/tools/docker/registry",
3 |   "instances": 1,
4 |   "cpus": 0.5,
5 |   "mem": 4096,
6 |   "disk": 0,
7 |   "executor": "",
8 |   "constraints": [],
9 |   "uris": []

```

所以，只需要将 `docker-2048.json` 使用 POST 方法提交到 Marathon 服务的 `/v2/apps` 即可。这里我们将使用 `curl` 作为客户端来提交 `docker-2048.json`，如下所示：

```
$ curl -H "Content-type: application/json" -d @docker-2048.json -X POST http://10.23.8.5.233:8080/v2/apps
```

`curl` 是一个在 Linux 环境下广泛使用的 HTTP 客户端，支持 JSON 格式的数据，上面的命令中，使用了 3 个选项：

- `-H`，添加一个 HTTP Header
- `-d`，数据，`@` 符号表示使用该文件的内容
- `-X`，指定使用的方法，常用的方法还有 `GET`，`POST`，`DELETE` 等

提交完成后，可以看到 Marathon Web 界面上新建了一个叫做 `docker-2048-created-by-api` 的 App，如下图所示：

Name	CPU	Memory	Status	Running Instances
docker-2048	1.0	1 GiB	Running	1 of 1
docker-2048-created-by-api	0.5	512 MiB	Deploying	0 of 1
hello-marathon	2.0	1 GiB	Running	2 of 2

Marathon 任务配置

在前面两个例子中，我们只使用非常少的几个配置参数就创建了 Marathon App，其实，Marathon App 有非常多配置，可以配置 App 的方方面面，下面介绍两类最常用的配置：

- 健康检查
- Constraints

健康检查能够根据 App 配置来检查 App 是否健康，并且在不健康的时候采取相应的动作；而 Constraints 则能够控制 App 在制定的结点上运行。

健康检查

作为长时任务，我们总是期望任务在运行的时候都是正常的，而不是表面上在运行，实际上却不正常了。以 Web 服务来说，许多反向代理或者负载均衡器都提供健康检查机制，例如：Nginx, HAProxy 等；类似地，Marathon 对任务也提供了健康检查机制。

Marathon 健康检查机制分为 3 类：

- HTTP, 定时访问指定 HTTP 接口，根据返回状态码来判断任务是否健康
- TCP, 定时建立到任务的 TCP 连接，根据是否可连接来判断任务是否健康
- COMMAND, 定时执行一个命令，根据命令的返回值来判断任务是否健康

Marathon 还提供了可以配置的策略，当检查到任务不健康时，可以采取相应的动作，例如：杀死当前实例并且启动一个新的实例。

以 `docker-2048` 为例，我们可以添加一个 HTTP 机制的健康检查，添加后，`docker-2048-v2.json` 将变成下面这个样子：

```
{
  "id": "docker-2048-created-by-api-v2",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "alexwhen/docker-2048",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "hostPort": 0 }
      ]
    }
  },
  "cpus": 0.5,
  "mem": 512.0,
  "instances": 1,
  "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/",
      "gracePeriodSeconds": 30,
      "intervalSeconds": 10,
      "portIndex": 0,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 3
    }
  ]
}
```

我们在 `docker-2048-v2.json` 中添加了一个 `healthChecks` 配置，其中包含一个配置，使用的是 HTTP 方式，各参数的含义如下：

- `protocol`, 检查方式，可选的为 : HTTP, TCP, COMMAND
- `path`, uri 路径，可选的，默认为 '/'，只对 HTTP 有用
- `gracePeriodSeconds`, 任务变成 Running 后多长时间内忽略健康检查失败事件，默认为 15 秒，如果任务启动后需要很长一段时间才能提供服务，例如：可能进行内部初始化需要很长事件，那么最好设置一个较大值，或者不要使用该参数，则表示直到第一次任务变成 health 未知
- `portIndex`, 第几个内部端口，默认值为 0，docker-2048 在 `portMappings` 中只配置了一个端口，所以也只能指定为 0
- `timeoutSeconds`, 单次健康检查的超时，默认为 10 秒
- `maxConsecutiveFailures`, 最大连续失败次数，达到指定次数后，任务将被杀死，默认为 3

在更新了 `docker-2048-v2.json` 后，我们可以通过 `curl` 来提交更新后的 App，如下所示：

```
$ curl -H "Content-type: application/json" -d @docker-2048-v2.json -X PUT http://10.23.85.233:8080/v2/apps/docker-2048-created-by-api
```

更新已有 App 使用的是 PUT 方法，并且需要在 url 中指定原有 App 的 id，这里为 `docker-2048-created-by-api`。

更新完成后，打开 Marathon Web 界面，可以看到 `docker-2048-created-by-api` 的 **Running Instances** 状态处显示了一条绿色的进度条，如下图所示：

Name	CPU	Memory	Status	Running Instances
<code>docker-2048</code>	1.0	1 GiB	<code>Running</code>	1 of 1
<code>docker-2048-created-by-api</code>	0.5	512 MiB	<code>Running</code>	1 of 1
<code>hello-marathon</code>	2.0	1 GiB	<code>Running</code>	2 of 2

绿色表示该 App 所有运行实例都正常。

Constraints

在一个实际生产环境中，Mesos 集群的计算结点可能具有差异性，例如：

- 从机器类型看：结点可能虚机或者是物理机
- 从网络带宽看：结点可能配备了千兆网卡，万兆网卡，40G 网卡等
- 从是否可访问英特网看：结点可能是外网直连，通过代理，NAT，或者不能访问英特网
- 从网络运营商看：结点可能使用电信，联通，移动等网络出口

以及还有很多其它方面，所以，如果 App 对结点配置有要求，那么就需要使用 `constraints` 来限制只能在制定的结点上启动 App 运行实例，要使用 `constraints` 来过滤计算结点，需要为结点配置适当的属性（`attributes`），这些属性可以在启动 `mesos-slave` 进程的时候配置。

除了 `mesos-slave` 配置的属性外，Marathon 还支持按照计算节点名字过滤。

除了用来过滤计算结点外，`constraints` 还可以用来控制 App 运行实例的分布特性，例如：

- 平均分布：尽量在启动实例的结点上均匀分布实例
- 唯一分布：在一个计算节点上只能启动一个运行实例

这里，我们为 `docker-2048` 添加一个基于主机名的限制条件，使其只能运行在 `10.23.85.234` 上，修改后的 `docker-2048-v3.json` 内容如下：

```
{
  "id": "docker-2048-created-by-api",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "alexwhen/docker-2048",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "hostPort": 0 }
      ]
    }
  },
  "cpus": 0.5,
  "mem": 512.0,
  "instances": 1,
  "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/",
      "gracePeriodSeconds": 30,
      "intervalSeconds": 10,
      "portIndex": 0,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 3
    }
  ],
  "constraints": [["hostname", "LIKE", "10.23.85.234"]]
}
```

然后，使用 `curl` 命令提交修改后的 `docker-2048-v3.json` 到 Maraton，更新已经存在的 `docker-2048` App。

```
$ curl -H "Content-type: application/json" -d @docker-2048-v3.json -X PUT http://10.23.85.233:8080/v2/apps/docker-2048-created-by-api
```

App 更新后，可以看到 App 的一个实例在 `10.23.85.234` 上运行了，限制，可以测试将 App 运行实例数设置为 **4**，可以看到，随后又在 `10.23.85.234` 上启动了 3 个运行实例，却没有在其它结点上启动任何实例。

运行Marathon任务

docker-2048-created-by-api

Running (4 of 4 tasks)

Scale Application Restart Suspend Destroy

Tasks Configuration Debug

Refresh

ID	Status	Version	Updated	Health
docker-2048-created-by-api.321b9a9c-1b8c-11e6-98da-525400fd9fbc 10.23.85.234:31898	Started	a minute ago	May 17, 2016 at 01:32:44 GMT+8	green
docker-2048-created-by-api.4762d6cd-1b8c-11e6-98da-525400fd9fbc 10.23.85.234:31497	Started	a minute ago	May 17, 2016 at 01:33:19 GMT+8	green
docker-2048-created-by-api.476324ef-1b8c-11e6-98da-525400fd9fbc 10.23.85.234:31124	Started	a minute ago	May 17, 2016 at 01:33:19 GMT+8	green
docker-2048-created-by-api.4762fdde-1b8c-11e6-98da-525400fd9fbc 10.23.85.234:31315	Started	a minute ago	May 17, 2016 at 01:33:19 GMT+8	green

读者甚至可以停止掉 10.23.85.234 上的 mesos-slave 进程来验证，看看 marathon 会不会在其它结点上启动 docker-2048 的运行实例。

小结

本节介绍了 Marathon 计算框架，包括适用的场景，特性；怎样搭建 Marathon 生产环境；最后介绍了一个实例。Marathon 作为基础的长时任务计算框架，读者可以基于 Marathon 做出更好更酷的产品，例如：编写一个集成业务逻辑的控制器，根据业务情况来自动调节 App 实例数的数量，达到 Auto Scaling 的目的；业务逻辑可以是 QPS, 活跃连接数等等。

Chronos

Chronos 是一个基于 Mesos 的批处理任务处理框架，最初由 Airbnb 开发，目前由 Mesosphere 维护，属于 Mesos 生态中较为活跃的项目之一。

本节将从以下几个方面介绍 Chronos：

- Chronos 简介
- 怎样搭建 Chronos 服务
- 运行 Chronos 任务

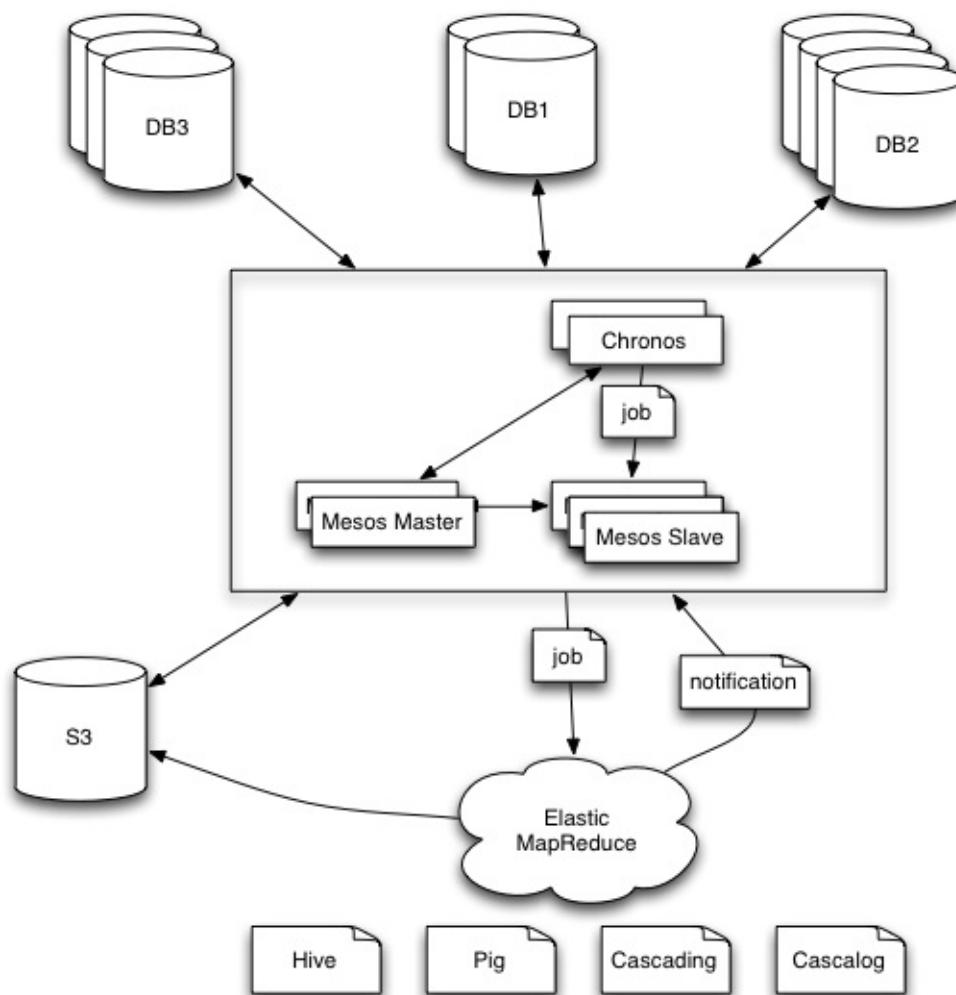
Chronos 简介

Chronos 是一个基于 Mesos 的分布式的 Cron 任务处理框架，最初由 Airbnb 公司开发，目前由 Mesosphere 公司负责维护。作为短任务框架，Chronos 几乎可以用于任何地方，只要用户有批处理任务需求即可，特别是原生支持 Docker 以后，Chronos 任务的运行环境将不再受到 Mesos 计算结点的限制。

Chronos 架构

Chronos 在架构上和 Marathon 非常相似，只是二者面向的业务类型不同，Chronos 主要面向批处理任务，定时重复执行的任务，而 Marathon 面向长时运行任务。

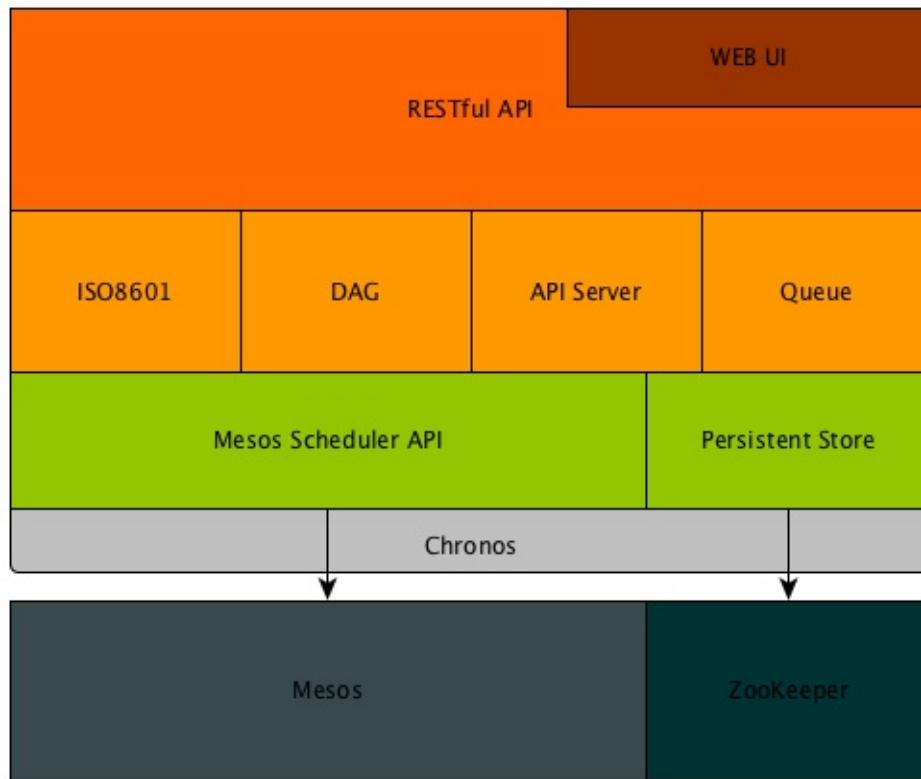
下图是 Chronos 官方的一个示例架构图：



其中矩形框内是 Chronos 架构，而框外部是其它服务。所以，Chronos 本身是一个运行在 Mesos 之上的框架。

Chronos 工作原理

下图展示了 Chronos 内部功能模块：



主要包含了以下几个功能模块：

- Mesos Scheduler API，主要负责实现 Mesos 定义的 Scheduler API 接口，以便和 Mesos 通信，获取资源，提交任务，获得任务状态更新等事件
- Persistent Store，Chronos 使用 ZooKeeper 作为数据持久化存储后端，所有需要存储的数据都存储在 ZooKeeper 中；所以，除了依赖于 Mesos 外，Chronos 还依赖于 ZooKeeper。
- ISO8601 模块负责解析 ISO8601 标准定义的重复任务属性
- DAG，有向无环图实现了对任务间依赖关系的支持，任务间依赖也是 Chronos 的一大亮点，当所有前置任务完成后，后置任务会被自动触发
- API Server，一个 Web API Server 负责接收处理收到的 API 请求，Chronos 实现了高可用性，在任意时刻，同一个服务中只有一个 Chronos 实例作为 Leader 负责处理请求，其它 Chronos 实例的 API Server 此时会将请求转发给 Leader
- Queue，任务队列，以便将可运行的任务调度到队列中，在有合适资源时将任务调度到 Mesos 集群中运行
- RESTful API，Chronos 提供了完善的 RESTful API 来使用 Chronos
- WEB UI，Chronos 也提供了 Web 用户界面，能够实现对任务的大部分操作

在了解了 Chronos 各个功能模块之后，也就大概了解了 Chronos 工作原理，总的来说：作为一个 Mesos 计算框架，计算资源的获取，计算任务状态更新都需要由 Mesos 来辅助完成，所以 Chronos 主要实现了对任务的存储，调度，搜索等任务逻辑层的功能。

Chronos 特性

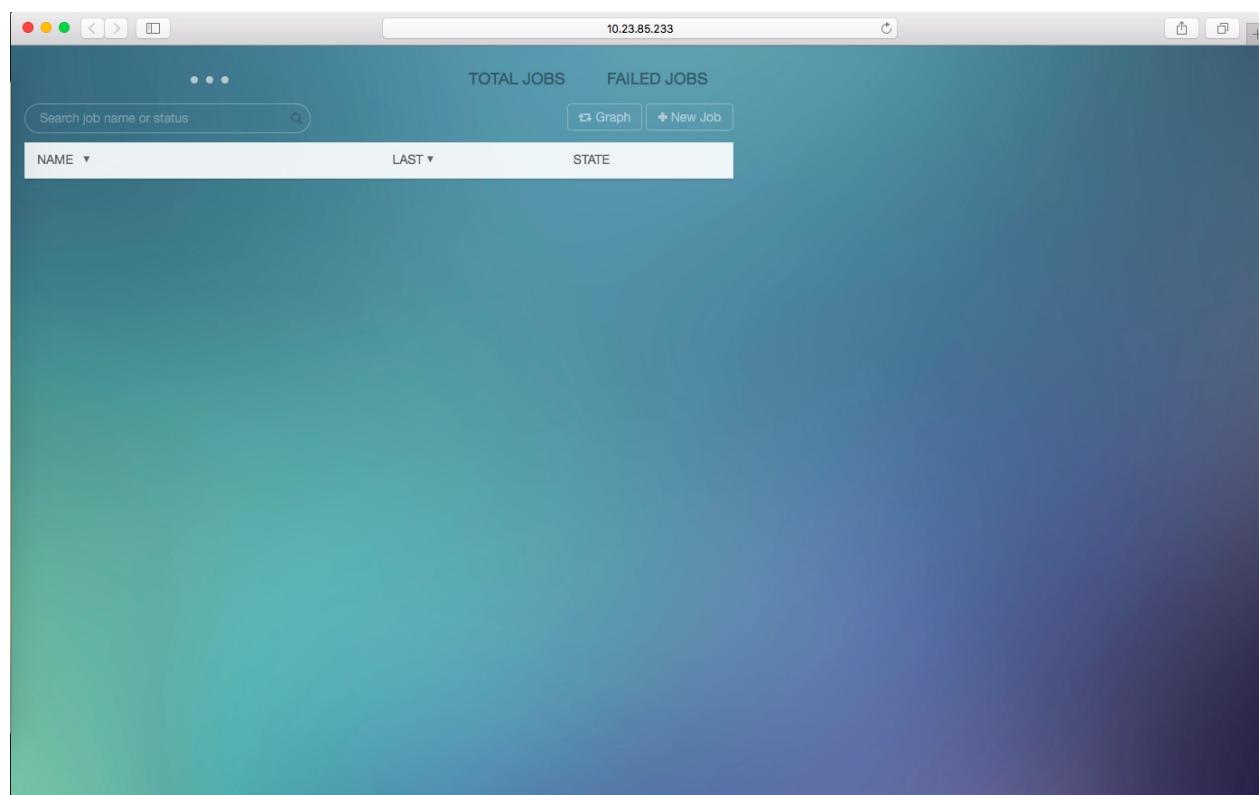
Chronos 作为一个分布式的 Cron 批处理任务计算框架，除了支持定时任务外，还有许多高级特性：

- Web 用户界面
- RESTful API
- ISO8601 重复任务支持
- 任务依赖关系
- 任务状态查看及任务历史
- 可配置的失败重试次数
- 屏蔽底层结点故障
- 原生 Docker 支持

下面对每个特性分别做简要介绍。

Web 用户界面

Chronos 的 Web 用户界面如下图所示：



在这个 Web 用户界面，用户可以完成大部分操作，包括对任务的增删查改，搜索，查看状态，运行历史等等。

但是，需要注意的是，Chronos Web 用户界面的性能并不高，当 Chronos 上积累的任务过多时，会导致加载速度变慢。所以，如果对 Chronos 使用量较大，推荐使用 API 方式管理任务，更高效。

RESTful API

Chronos 提供了完善的 RESTful API，完善的 API 对于批处理任务框架来说比 Web 用户界面更重要，因为批处理任务往往意味着运行时间短，量大，所以通过 Web 用户界面来控制批处理任务显得非常低效，这时，通过 API 控制任务显然更加高效。

ISO8601 重复任务支持

和 Cron 标准类似，ISO8601 定义了一种表示重复任务的标准，包括对：日期，时间，间隔，重复的定义，例如：`R3/2015-12-20T13:44:00Z/P1Y2M10DT2H30M10S` 表示从 UTC 时间 2015 年 12 月 20 日 13 点 44 分 00 秒开始，以间隔 1 年 2 个月 10 天 2 小时 30 分 10 秒发生 3 次；所以，事件的第一次发生在 2015 年 12 月 20 日 13 点 44 分 00 秒，第二次发生则在 1 年 2 个月 10 天 2 小时 30 分 10 秒之后。

任务依赖关系

Chronos 通过 DAG（有向无环图）实现了任务依赖关系的支持，依赖任务只有在其所有被依赖任务成功完成之后才会被触发，所以用户可以定义非常复杂的任务依赖关系；显然，互相依赖或者闭环的依赖是不被允许的。

任务状态查看及任务历史

Chronos 会在 Web 用户界面实时显示当前任务的状态，并且用户可以手动激活任务。由于任务可能会被重复执行，Chronos 还支持对任务历史状态的查看。

可配置的失败重试次数

用户可以针对每个任务配置失败重试任务，当任务在失败时自动进行重试，非常方便。

屏蔽底层结点故障

由于 Mesos 能够区分任务的失败是由于任务自身运行的失败还是由于底层计算节点故障导致的任务失败，所以 Chronos 能够自动屏蔽由计算结点导致的任务失败，并对用户屏蔽这种失败，自动发起重试。

原生 Docker 支持

原生 Docker 支持的引入，使 Chronos 任务不再依赖于 Mesos 计算结点的环境，例如：Chronos 的任务可能需要调用某一个外部命令，而这个外部命令如果不存在，显然会导致任务的失败。

另外，使用 shell executor 执行 Chronos 任务也非常危险，因为有可能任务中包含了破坏性的命令，一旦以不当的权限运行，那么将会对 Mesos 计算节点造成破坏，例如：以 root 用户权限执行了 `rm -rf /` 显然会破坏整个 Mesos 计算结点。

所以，对 Docker 的原生支持极大的提高了 Chronos 的可用性。

搭建Chronos服务

在了解了 Chronos 的工作原理和特性后，这里将介绍怎样搭建一个生产环境可用的 Chronos 服务，生产环境可用意味着要实现服务的高可用性。

我们知道 Chronos 将所有需要持久化的数据都存储到了 ZooKeeper 服务中，所以 Chronos 数据的可靠性由 ZooKeeper 服务来保障，由于 Chronos 是单一 Leader 模型，所以本身不存在数据一致性的问题，但是由于 ZooKeeper 服务是分布式的，所以需要保障数据的一致性。

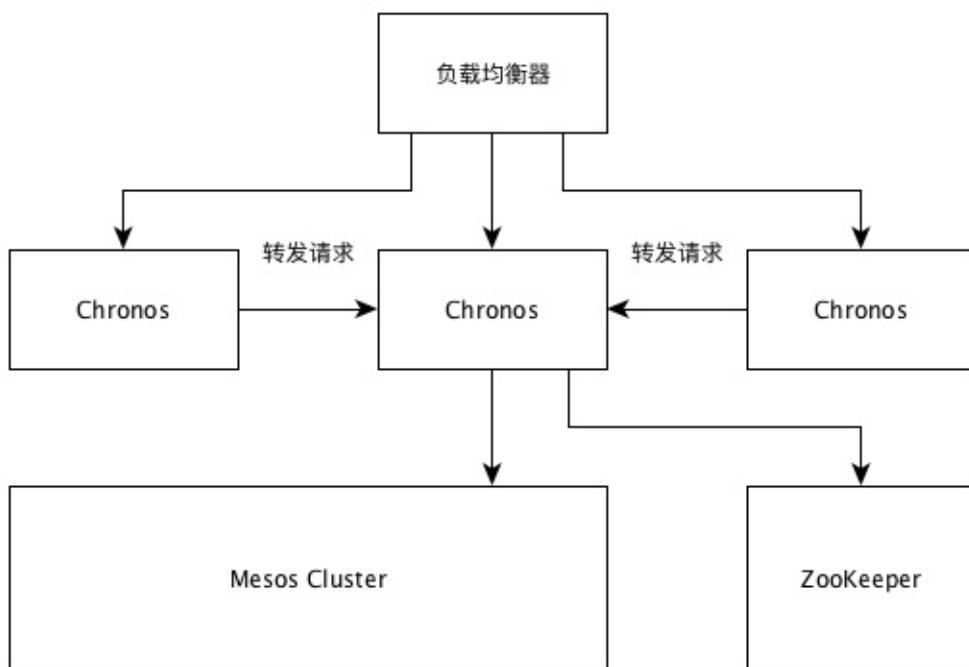
本节首先介绍 Chronos 高可用性模型，然后动手搭建一个高可用的 Chronos 服务，更进一步，我们将 Chronos 作为一个 Marathon App 运行在 Marathon 之上，从而进一步减少对 Chronos 的运维。

Chronos 高可用性

得益于将所有数据持久化在 ZooKeeper 服务中，Chronos 的高可用性模型非常简单。

Chronos 使用 ZooKeeper 来实现服务内部实例之间的 Leader 选举，任意时刻只有一个服务实例作为 Leader 提供服务，其它实例均作为跟随者，虽然可以接受请求，但是并不对请求进行处理，而是简单的转发，Chronos 会监听 Leader 改变事件，实时知道当前的 Leader 实例。

下图是一个高可用性的 Chronos 服务在生产环境中运行的示例：



首先，上图中部署了 3 个 Chronos 实例，但是在任意时刻，最多只有一个实例作为 leader 来真正的处理请求，其它 Chronos 则会将收到的请求转发到 leader 实例。

而为了实现 Chronos 服务故障对用户不可见，在 Chronos 前端搭建了一个负载均衡器，同时也作为一个反向代理，这样，当 Chronos leader 故障，切换 leader 后，用户不需要任何修改。

Chronos 的高可用性非常简单，在任意时刻只需要有一个 Chronos 实例在运行即可提供服务，所以，由两个实例组成的 Chronos 服务即可提供高可用服务。

搭建 Chronos 服务

在编写本书之时，Chronos 最新发布的稳定版本是 2.4.0，并且支持 Mesos 0.24.0，而当前 Mesos 最新的稳定版本是 0.25.0，所以这里我们将搭建 Chronos-2.4.0_mesos-0.25.0，所以需要重新基于 Mesos 0.25.0 编译 Chronos。

编译 Chronos

首先，从 mesosphere 或者 GitHub 上下载 Chronos-0.24.0_mesos-0.24.0 源码包，这里以从 GitHub 下载为例，如下图所示：

The screenshot shows the GitHub repository page for `mesos/chronos`. At the top, there's a header with the URL `Hub, Inc. [US] https://github.com/mesos/chronos/releases` and various GitHub navigation icons. Below the header, there's a search bar and links for `Pull requests`, `Issues`, and `Gist`. The repository name `mesos / chronos` is displayed, along with a `Watch` button (304), an `Unstar` button, and a star count of 2,967. A navigation bar below the repository name includes `Code`, `Issues 116`, `Pull requests 11`, `Pulse`, and `Graphs`. The `Releases` tab is currently selected. A list of releases is shown, with the latest one being `v2.4.0`, released on Oct 7 by `gkleiman`. The release notes mention `45 commits to master since this release`. Below the release notes, there's a section titled `Changes from 2.3.4 to 2.4.0`.

这里我们下载 tar.gz 格式的包，假设下载到了本地的 `~/Downloads` 目录下。

接下来，执行下面的命令解压下载好的代码包。

```
$ tar xzf chronos-2.4.0_mesos-0.24.tar.gz
```

然后，修改目录的名字为 `chronos-2.4.0_mesos-0.25`

```
$ mv chronos-2.4.0_mesos-0.2{4,5}
$ cd chronos-2.4.0_mesos-0.25
```

修改 `pom.xml` 文件中的第 34 行

```
<mesos-utils.version>0.24.0</mesos-utils.version>

修改为

<mesos-utils.version>0.25.0</mesos-utils.version>
```

`mesos-utils` 是 Mesosphere 为 Scala 提供的编译包，感兴趣的读者可以前往其项目主页：<https://github.com/mesosphere/mesos-utils> 了解更多。

现在，可以编译 Chronos-0.24.0_mesos-0.25 了，这里我们介绍两种编译 Chronos 的方法，一种是传统的将其编译成 jar 包的方式，一种是直接构建出 Chronos Docker 镜像。

编译 Chronos Jar

Chronos 使用 Maven 来编译，所以首先需要安装配置好 Maven 以及 JDK，这里不再赘述，读者可以参考 Maven 及 JDK 官方文档。

设置好 Maven 和 JDK 之后，运行 mvn 命令编译项目，如下：

```
$ mvn clean package -Dmaven.test.skip=true
```

上面的命令表示执行两个 mvn task:

- `clean`，清除之前构建的文件
- `package`，重新执行一次编译并打包

命令中的 `-Dmaven.test.skip=true` 表示忽略构建测试用例，这样能够加尽快完成打包；由于我们是在使用 `release` 版本而非参与开发，所以可以忽略构建测试用例。

编译完成后，会在当前目录生成一个 `target` 目录，并且在这里可以找到构建好的 Chronos jar 包，如下所示：

```
$ ls -l target
total 38728
drwxr-xr-x 2 chengwei chengwei 4096 May 19 15:54 antrun
-rw-r--r-- 1 chengwei chengwei 36836527 May 19 15:56 chronos-2.4.0.jar
drwxr-xr-x 4 chengwei chengwei 4096 May 19 15:55 classes
-rw-r--r-- 1 chengwei chengwei 1 May 19 15:55 classes.-837014839.timestamp
drwxr-xr-x 2 chengwei chengwei 4096 May 19 15:56 maven-archiver
-rw-r--r-- 1 chengwei chengwei 2797728 May 19 15:56 original-chronos-2.4.0.jar
```

chronos-2.4.0.jar 既是我们需要使用的 chronos jar 文件。

编译 Chronos Docker 镜像

另一种编译 Chronos 的方式，是将 Chronos 直接编译成可启动的 Docker 镜像，在 chronos-0.24.0_mesos-0.25 目录中，可以找到一个 Dockerfile 文件，直接在本目录下运行 docker build 命令即可直接构建出 Chronos 的镜像，例如：

```
$ docker build -t chronos:0.24.0_mesos-0.25 .
```

不过，读者首先需要在次机器上安装 Docker 环境，构建完成后，即生成了一个镜像，可以使用 docker images 查看，输入如下：

\$ docker images			
REPOSITORY	IMAGE ID	CREATED	TAG
E			
chronos			0.24.0_mesos
-0.25	30f102b4345b	17 hours ago	1.2
03 GB			

不管以何种方式编译 Chronos，在完成之后，就可以启动 Chronos 了。

运行单个 Chronos 服务

本地运行 Chronos

在编译好 Chronos 之后，就可以启动 Chronos 服务了，执行 chronos-0.24.0_mesos-0.25 目录下的 bin/start-chronos.sh 即可，首先查看帮助文档，如下：

```
[root@10 chronos-2.4.0_mesos-0.25]# ./bin/start-chronos.bash --help
Chronos home set to /root/chronos-2.4.0_mesos-0.25
Using jar file: /root/chronos-2.4.0_mesos-0.25/target/chronos-2.4.0.jar[0]
[2016-05-22 14:49:50,468] INFO ----- (org.apache.mesos.chronos.scheduler.Main$:26)
[2016-05-22 14:49:50,469] INFO Initializing chronos. (org.apache.mesos.chronos.scheduler.Main$:27)
[2016-05-22 14:49:50,471] INFO ----- (org.apache.mesos.chronos.scheduler.Main$:28)
    --assets_path <arg>                                Set a local file system path to
                                                        load assets from, instead of
                                                        loading them from the packaged
                                                        jar.
    --cassandra_consistency <arg>                      Consistency to use for Cassandra
                                                        (default = ANY)
    -c, --cassandra_contact_points <arg>              Comma separated list of contact
                                                        points for Cassandra
    --cassandra_keyspace <arg>                          Keyspace to use for Cassandra
                                                        (default = metrics)
    --cassandra_port <arg>                            Port for Cassandra
                                                        (default = 9042)
...
省略若干行 ...

```

Chronos 有很多可以配置的选项，但是为了启动一个 Chronos 服务，只需要配置少数几个必须的项即可：

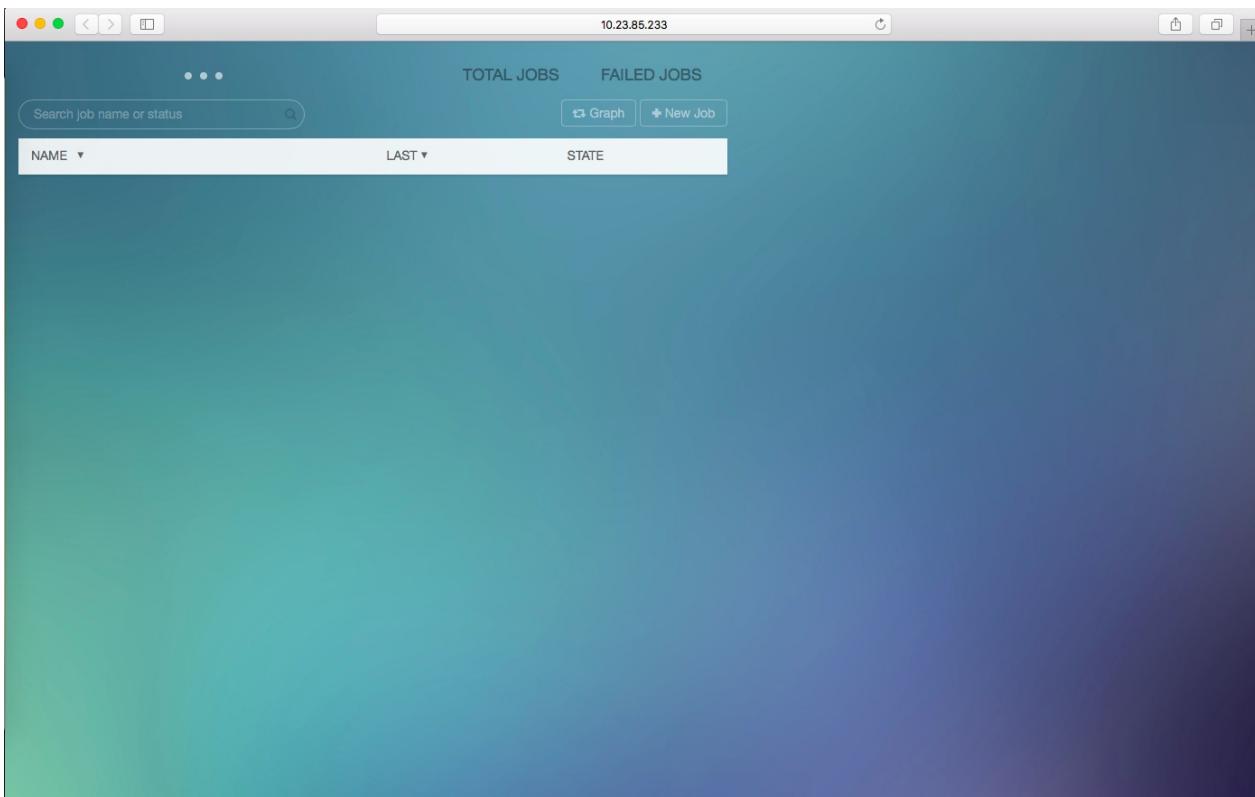
- `--master`，配置 mesos 集群地址
- `--zk_hosts`，配置 ZooKeeper 服务地址

从前面 Chronos 介绍中我们知道，Chronos 依赖于 mesos 集群以及 ZooKeeper 服务，所以这两个参数是必须的。

这里以前面搭建的 mesos 集群和 ZooKeeper 服务为例来启动 Chronos 服务，如下：

```
# [root@10 chronos-2.4.0_mesos-0.25]# ./bin/start-chronos.bash --master zk://10.23.85
.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos --zk_hosts zk://10.23.85.233:2181,
10.23.85.234:2181,10.23.85.235:2181
```

启动后，Chronos 会默认监听在 8080 端口接收服务请求，使用浏览器打开本地的 8080 端口地址，例如：<http://10.23.85.233:8080>，可以看到 Chronos Web 用户界面，如下图所示：



同时，在 Mesos master WEB 界面上也可以看到新注册的 Chronos 框架，如下图所示：

A screenshot of the Mesos master's "Frameworks" section. At the top, there are tabs for "Mesos", "Frameworks", "Slaves", and "Offers", with "Frameworks" being the active tab. On the far right, it says "mesos-in-action". Below the tabs, there are two tables. The first table, titled "Active Frameworks", has columns: ID, Host, User, Name, Active Tasks, CPUs, Mem, Max Share, Registered, and Re-Registered. It shows two entries: one for "chronos-2.4.0" registered 2 hours ago and another for "marathon" registered 7 days ago. The second table, titled "Terminated Frameworks", has columns: ID, Host, User, Name, Registered, and Unregistered. It is currently empty.

以 Docker 的方式

除了直接在主机上启动 Chronos 外，还可以将 Chronos 运行在 Docker 中，使用 `docker run` 命令来启动在前面构建的 Chronos 镜像，例如：

```
$ docker run -p 8080:8080 --name=chronos chronos:0.24.0_mesos-0.25 --master zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos --zk_hosts zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181
```

这里使用了两个 `docker run` 的参数：

- `-p`，映射容器内部指定端口到主机指定端口
- `--name`，制定容器名称，这里为 `chronos`

Chronos 默认会监听 8080 端口，这里我们将其映射到主机端口 8080，所以在启动 Chronos 容器之前，首先需要确保主机的 8080 端口没有被占用，否则将失败。

Chronos 配置参数

Chronos 有许多配置参数可以用来修改其默认行为，这里介绍一些比较常用的参数。

参数	默认值	含义
--decline_offer_duration <arg>	5 秒	设置过滤被拒绝的 Mesos Offer 的时间，单位为毫秒，如果不设置，则采用 Mesos 默认的过滤时间 5 秒
--disable_after_failures <arg>	0	设置在任务失败多少次之后，禁止任务，默认为不禁止
--failover_timeout <arg>	604800 秒，即一周	设置 Chronos 框架 failover 的时间超时；在 Mesos 中，如果框架异常失败，则会设置一个 failover 超时，如果在次时间内恢复，则还可以重新注册并且获取到失败之前的任务状态
--failure_retry <arg>	60000 毫秒	设置任务失败重试的时间间隔，默认为 60000 毫秒，即 60 秒
--graphite_group_prefix <arg>	空	设置 Chronos metrics 导出到 Graphite 中的路径前缀，Graphite 是一个基于时间的数据存储，非常适合存储运行状态数据，Chronos 也支持将运行数据发送到 Graphite，以便查看历史运行状态
--graphite_host_port <arg>	空	设置 Graphite 主机和端口，格式为 host:port
--graphite_reporting_interval <arg>	默认值 60 秒	设置汇报运行数据的时间间隔，单位为秒
--hostname <arg>	本机 hostname	设置可访问的主机名，一定要能够被其它 Chronos 实例访问，否则当本实例作为 Leader 时，其它实例不能转发请求到本实例
--http_credentials <arg>	空	设置 HTTP basic 认证使用的用户名和密码，格式为： user:password
--job_history_limit <arg>	5	设置显示任务的历史数量
--master <arg>	local	Mesos 服务地址
--mesos_checkpoint	否	设置是否开启框架的 checkpoint，推荐开启
--mesos_framework_name <arg>	空	设置本框架的名称，默认值将有 chronos 加版本号组成，例如： chronos-2.4.0

--mesos_role <arg>	*	设置框架的 role，role 是 Mesos 的属性，用来将框架归类，从而实现资源的隔离，默认的 role 为 *，即不使用特殊 role，即共享资源的 role
--mesos_task_cpu <arg>	0.1	任务申请的 CPU 量，可以在每个任务中设置
--mesos_task_disk <arg>	256	任务申请的磁盘量，可以在每个任务中设置，单位为 MB
--mesos_task_mem <arg>	128	任务申请的内存，可以在每个任务中设置，单位为 MB
--task_epsilon <arg>	60	设置任务错过时间，单位为秒，在调度时，如果任务应该启动的时间早于当前时间超过这里设置的时间，将忽略任务的本次调度
--user <arg>	root	Chronos 运行该任务是的权限，可见，默认的 root 权限非常开放，如果 mesos-slave 以 root 权限运行，那么一不小心可能执行了错误的任务，导致计算结点被破坏，或者如果 mesos-slave 以非 root 权限运行，显然任务启动会失败，所以，推荐使用 Chronos 时，只以 Docker 的方式提交任务
--zk_hosts <arg>	localhost:2181	ZooKeeper 服务的地址，Chronos 使用 ZooKeeper 来存储所有持久化数据
--zk_path <arg>	/chronos/state	Chronos 在 ZooKeeper 中使用的存储路径，Chronos 将把所有数据存储在该目录下
--zk_timeout <arg>	10000，即 10 秒	ZooKeeper 操作的超时时间

在了解了上表中常用的 Chronos 参数后，我们知道，上一小节中启动的 Chronos 没有做任何配置，所以它会以本地方式运行，所以实际上没有任何用处，接下来我们将对 Chronos 进行一些配置，并且结合上一章介绍的 Mesos 生产环境和 ZooKeeper 生产环境，搭建高可用，可用于生产环境的 Chronos 服务。

搭建高可用 Chronos 服务

在启动了一个 Chronos 实例后，Chronos 就可以提供服务了，Chronos 在高可用方面实现原理和 Marathon 类似，都使用了 ZooKeeper 作为持久化存储服务，只要在任意时刻有一个实例在运行即可提供服务。

所以，要搭建高可用的 Chronos 服务，非常简单，在其它机器上以同样的配置，启动多个 Chronos 即可。

这里假设在 10.23.85.234 上再启动一个 Chronos，这样就实现了高可用。

但是，由于 Chronos 暴露服务的方式是直接通过 IP 和端口暴露，所以，当 Leader 故障后，虽然存活的 Chronos 能够自动选举为 Leader，继续服务，但是服务地址和端口却改变了。这对于使用方来说并不是透明的，因为用户需要修改访问地址才能继续访问 Chronos。

为了能够对用户透明，需要为 Chronos 服务提供一个统一的服务地址和端口，而后端由多个 Chronos 实例来支撑，这样，当后端的 Chronos 实例故障之后，用户并不需要做任何修改。

关于实现透明的高可用服务，在上一节的 Marathon 配置中已经介绍过，这里不再重复。

将 Chronos 运行在 Marathon 上

将 Chronos 运行在 Marathon 上将非常有趣。它们二者都是基于 Mesos 的计算框架，Marathon 是一个长时任务框架，而 Chronos 是一个短时批处理任务框架。

通过将 Chronos 运行在 Marathon 上，能够实现服务实例故障自动转移、恢复，结合 Marathon 提供的健康检查机制以及服务发现机制，能够让我们的 Chronos 服务做到对用户透明的高可用性服务，并且实现服务实例故障自动转移，恢复，极大降低运维成本。

首先，假设我们使用在上一节中搭建的 Marathon 服务，并且使用本节中介绍的以 Docker 运行 Chronos 的方式将 Chronos 运行在 Marathon 上。

启动 Chronos

在 Marathon 一节中，曾介绍过怎样创建一个基于 Docker 的 2048 网页版游戏，这里我们使用同样的方式，启动一个 Chronos 应用，由于启动 Chronos 镜像需要传递额外的参数 `--master` 和 `--zk_hosts`，而 Marathon 的 WEB 界面目前还不支持，所以这里介绍怎样用命令行来创建。

首先，准备一个 `chronos-on-marathon.json` 文件，内容如下：

```
{
  "id": "chronos",
  "container": {
    "docker": {
      "image": "docker-registry.qiyi.virtual/yangchengwei/chronos:0.24.0_mesos-0.25",
      "network": "BRIDGE",
      "portMappings": [{ "containerPort": 8080, "hostPort": 0, "protocol": "tcp" }]
    },
    "type": "DOCKER"
  },
  "args": [
    "--master", "zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos"
  ],
  "cpus": 1.0,
  "mem": 1024.0,
  "instances": 2
}
```

然后使用 curl 命令来提交，如下：

```
$ curl -H "Content-type: application/json" -d @chronos-on-marathon.json -X POST http://10.23.85.233:8080/v2/apps
{
  "id": "/chronos",
  "cmd": null,
  "args": [
    "--master", "zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos",
    "--zk_hosts", "zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181"
  ],
  "use_r": null,
  "env": {},
  "instances": 2,
  "cpus": 1,
  "mem": 1024,
  "disk": 0,
  "executor": "",
  "constraints": [],
  "uris": [],
  "storeUrls": [],
  "ports": [0],
  "requirePorts": false,
  "backoffSeconds": 1,
  "backoffFactor": 1.15,
  "maxLaunchDelaySeconds": 3600,
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "docker-registry.qiyi.virtual/yangchengwei/chronos:0.24.0_mesos-0.25",
      "network": "BRIDGE",
      "privileged": false,
      "parameters": [],
      "forcePullImage": false
    },
    "healthChecks": [],
    "dependencies": [],
    "upgradeStrategy": {
      "minimumHealthCapacity": 1,
      "maximumOverCapacity": 1
    },
    "labels": {},
    "acceptedResourceRoles": null,
    "version": "2016-05-22T08:27:11.469Z",
    "tasksStaged": 0,
    "tasksRunning": 0,
    "tasksHealthy": 0,
    "tasksUnhealthy": 0,
    "deployments": [
      {"id": "26552208-6ab7-4dbe-8c98-758e4adfa8cb"}
    ],
    "tasks": []
  }
}
```

提交完成后，可以看到有 2 个任务处于 Staged 状态，如下图所示：

ID	Status	Version	Updated
chronos.34c577a0-1fef-11e6-98da-525400fd9fbc 10.23.85.233:31772	Staged	3 minutes ago	May 22, 2016 at 15:31:32 GMT+8
chronos.34c74c61-1fef-11e6-98da-525400fd9fbc 10.23.85.234:31704	Staged	3 minutes ago	May 22, 2016 at 15:31:32 GMT+8

这是因为 **Mesos** 计算节点需要下载 Chronos 镜像到本地，之后才能启动容器，所以根据下载时间的长短，可能需要等几分钟到几十分钟。

注意：这里假设使用了共有的 **docker hub** 服务，所以 **mesos** 集群的计算节点需要能够从 **docker hub** 服务上下载镜像，否则将不能启动，另一种较复杂的方式是读者可以搭建私有的 **docker-registry** 服务来存储镜像

当任务处于运行状态后，使用浏览器打开任意一个 Chronos 实例，可以看到 Chronos 服务已经在正常运行了。

服务发现和负载均衡

我们知道，当运行在 Marathon 上的 Chronos 任务出现故障时，Marathon 会自动启动新的 Chronos 实例，几乎可以肯定的是，这个新实例运行的地址和端口和原来的不一样。所以，在真实的生产环境中，为了对外提供对用户透明的高可用 Chronos 服务，需要在 Chronos 实例前端添加一层负载均衡器，同时也充当反向代理的作用。

对于 Marathon 来说，marathon 提供了两种服务发现和负载均衡的方式：

- mesos-dns
- marathon-lb

二者的配置稍显复杂，这里留给感兴趣的读者自行研究。

小结

本节介绍了 Chronos 高可用模型，怎样搭建高可用的 Chronos 服务，以及将 Chronos 运行在 Marathon 之上，进一步提高可用性，降低对 Chronos 的维护成本。

运行Chronos任务

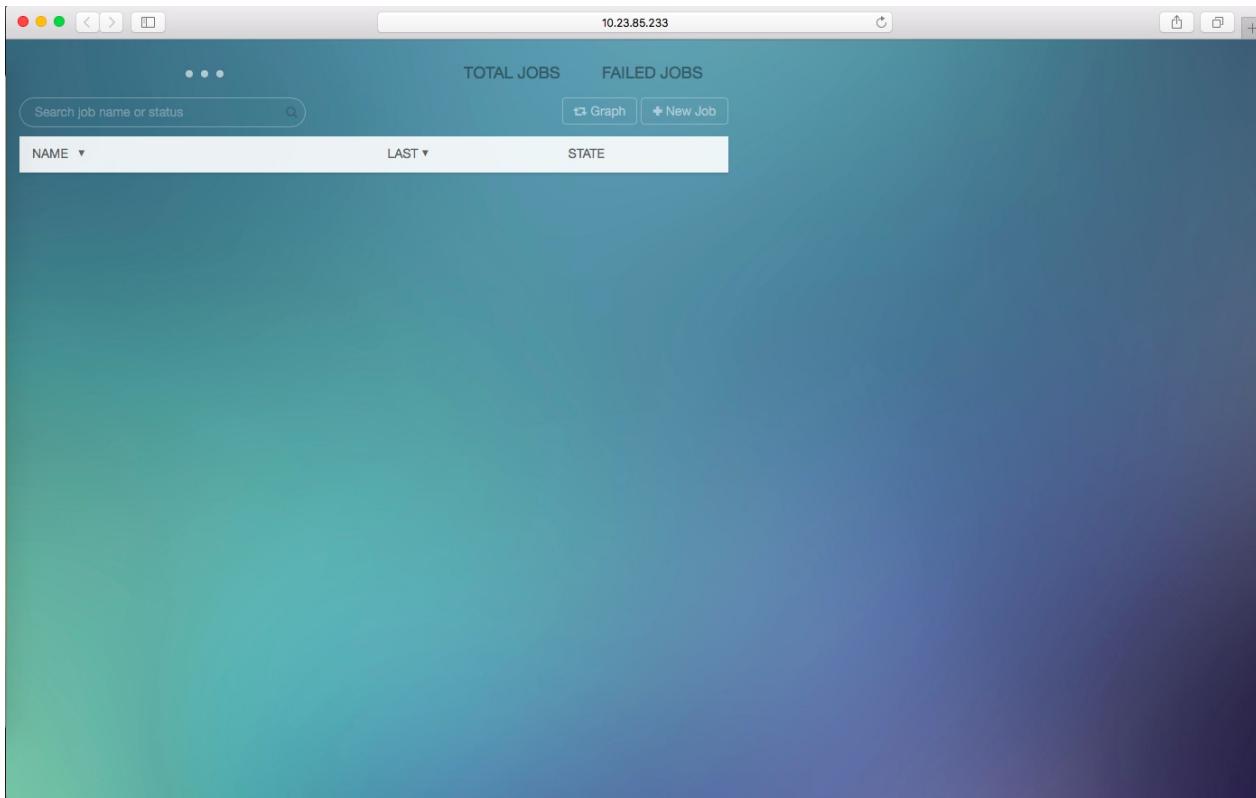
在搭建了 Chronos 服务后，就可以使用 Chronos 来运行任务了，Chronos 面向的是基于时间的可重复的短时运行任务。

- 基于时间：任务在指定的时间点开始执行，也可以是即时任务，提交后立即执行
- 可重复：任务可以配置重复执行的策略
- 短时运行：任务在运行一段时间后会结束
- 任务间依赖：任务间可以有依赖关系
- 同步、异步任务

Chronos 同时提供了 RESTful API 和 Web 用户界面来管理任务，这里分别对其进行介绍。

Web 用户界面

Chronos Web 用户界面的主页如下图所示：



通过 Web 用户界面可以提交任务，查看任务状态，历史运行情况等，还可以搜索任务。下面提交一个任务输出 "Hello Chronos!"。如下图所示：

✓ Create ✘ Cancel X

NAME
hello-chronos

DESCRIPTION
Hello Chronos

COMMAND
echo "Hello Chronos"

PARENTS
Choose parents...

OWNER(S)
yangchengwei@qiyi.com

OWNER NAME

SCHEDULE
R 1 / 2016-05-22 T 08:50:41 Z/ P T24H

NO STATS AVAILABLE FOR JOB.

[Other settings](#)

这里我们提交的是一个即时任务，所以任务一旦提交就会被立即调度，如果 Mesos 集群有资源，那么任务将立即启动，否则将在 Chronos 调度队列中等待。

很快，可以发现任务已经完成了，如下图所示：

Completed Tasks

ID	Name	State	Started ▾	Stopped	Host	
ct:1463906534980:0:hello-chronos:	ChronosTask:hello-chronos	FINISHED	just now	just now	10.23.85.235	Sandbox

从 Chronos web UI 也可以看到任务的状态，如下图所示：

The screenshot shows the Chronos Web interface. At the top, it displays 'TOTAL JOBS 1' and 'FAILED JOBS 0'. Below this is a search bar and a toolbar with icons for Graph and New Job. The main area shows a table with columns: NAME, LAST, and STATE. A single row is selected for 'hello-chronos', showing 'success' under LAST and 'idle' under STATE. To the right of the table, detailed information about the job is provided:

- NAME:** hello-chronos
- DESCRIPTION:** Hello Chronos
- COMMAND:** echo "Hello Chronos"
- OWNER(S):** yangchengwei@qiyi.com
- LAST SUCCESS:** 2016-05-22T08:50:45.187Z
- OWNER NAME:**
- LAST ERROR:**
- # SUCCESS:** 1
- # ERROR:** 0
- SCHEDULE:** R0/2016-05-23T08:50:41.000Z/PT24H
- JOB RUNTIME (PERCENTILES):**
 - 50th: 4.85 seconds
 - 75th: 4.85 seconds
 - 95th: 4.85 seconds
 - 99th: 4.85 seconds
- Other settings:**

但是，任务的输出从哪里查看呢？可以从 Mesos Web 用户界面找到这个任务，并且进入 Sandbox 中，查看 `stdout` 文件，即可找到。如下图所示：

```

10.23.85.234

Registered executor on 10.23.85.235
Starting task ct:1463906534980:0:hello-chronos:
Forked command at 27111
sh -c 'echo "Hello Chronos"'
Hello Chronos
Command exited with status 0 (pid: 27111)

```

可见，Mesos 启动了 shell 执行器来执行 Chronos 任务的命令，由于这里只执行了一条 `echo` 命令，所以可以在 Web 用户界面上输入，但是，如果任务要执行的是一个脚本呢？再从 Chronos Web 用户界面输入显然不合适：

- 每次输入比较麻烦，容易输入错误等
- Web UI 使用起来比较低效

下面来看看怎样使用 Chronos API 来管理任务。

RESTful API

创建一个叫 `hello-chronos.sh` 的脚本，内容如下：

```

#!/bin/sh
set -e
echo "Hello Chronos!"
echo "This is a complicated script"
sleep 10
echo "Mission complete"

```

并且将这个文件放到一个在 Mesos 计算节点上可以访问的地方，例如：HDFS, HTTP 服务器，甚至计算结点上。这里将这个脚本部署在了所有计算结点上，存放在 `/usr/local/mesos/chronos/hello-chronos.sh`，并且赋予可执行权限：

```
# chmod +x /usr/local/mesos/chronos/hello-chronos.sh
```

在这里，读者可能在好奇为什么不把脚本放在例如：HDFS 或者 HTTP 服务器上，而是将其部署在 Mesos 计算节点上。在实际生产环境中，为了加快任务启动速度，往往会选择将需要下载的文件都部署在本地，这样能够避免网络下载时间，避免网络抖动带来的影响等。特别是当需要下载的文件特别大时，部署在本地就更能加快任务启动速度。

另外，对于短时任务来说，启动速度更加重要，因为短时任务运行的时间一般不长。

对于批处理任务来说，通过 API 提交的方式更加高效实用，Chronos 提供了完善的 RESTful API，这里将介绍怎样实用 API 来提交任务，完整的 API 读者可以参考 Chronos 官方文档。

以提交 `hello-chronos.sh` 为例，创建一个 `hello-chronos.json`，内容如下：

```
{
  "name": "hello-chronos-script",
  "command": "./hello-chronos.sh",
  "uris": [
    "/usr/local/mesos/chronos/hello-chronos.sh"
  ]
}
```

然后，使用 curl 命令提交这个任务：

```
$ curl -H 'Content-Type: application/json' -X POST -d @hello-chronos.json http://10.23
.85.233:8080/scheduler/iso8601
```

这里使用了 `curl` 作为 HTTP 客户端提交任务，各个参数的意义如下：

- `-H`，指定 HTTP header
- `-X`，指定方法，HTTP 支持多种方法，常见的有：GET, POST, PUT, DELETE
- `-d`，设置负载数据，这里的数据是 JSON 内容

最后指定的是 Chronos 服务地址及其 API 地址。

提交完成后，前往 Chronos Web 用户界面，可以看到刚才提交的任务已经完成了，如下图所示：

这是因为上面的命令提交的是一个即时任务。在 Chronos 中，省略 `schedule` 表示即时任务，即提交后立即执行。

现在，对上面的 `hello-chronos.json` 稍作修改，变成一个重复执行的定时任务。

```
{
  "schedule": "R3/2016-05-23T09:30:00.000Z/PT5M",
  "name": "hello-chronos-script-cron",
  "command": "./hello-chronos.sh",
  "uris": [
    "/usr/local/mesos/chronos/hello-chronos.sh"
  ]
}
```

上面的命令，相较于上一次的提交有两个改变：

- `name` 为 `hello-chronos-script-cron`，这是因为 Chronos 任务中的 `name` 必须唯一
- `schedule` 为 `R3/2016-05-23T09:30:00.000Z/PT5M`，`R3` 表示任务要执行 3 次，`2016-05-23T09:30:00.000Z` 表示在 UTC 时间 2016 年 05 月 23 日 09 点 30 分开始第一次执行，也就是北京时间的 17 点 30 分，`PT5M` 表示重复执行的间隔为 5 分钟

基于 Docker 的任务

前面介绍的任务使用的都是 `shell` 执行器来执行的，所以会直接在 Mesos 计算节点上以配置的用户执行，如果执行的脚本中有误操作，例如：删除系统上的文件，那么将非常危险。用户不恰当的操作可能会破坏平台的计算环境甚至稳定性。

而使用 Docker，可以将任务的运行环境更好的隔离，包括文件系统方面的隔离，所以更加安全，同时，任务也将不再依赖于 Mesos 计算结点上的环境，例如：任务需要 Python 3 环境但是计算结点只有 python 2.x 环境，用户需要 Go 运行环境，但是计算结点却没有安装 Go 等。

Chronos 和 Mesos 都原生支持 Docker 任务，所以这里将提交一个 Docker 任务，并且同样的，将执行 `hello-chronos.sh` 脚本。

首先，构建一个 Docker 镜像，将 `hello-chronos.sh` 放在 Docker 镜像中，并且设置为启动命令，构建镜像的 Dockerfile 如下所示：

```
FROM alpine
MAINTAINER Chengwei Yang <me@chengweiyang.cn>

ADD hello-chronos.sh /
RUN chmod +x /hello-chronos.sh

CMD ["/hello-chronos.sh"]
```

将 `hello-chronos.sh` 和 `Dockerfile` 都保存在同一个目录中，如下所示：

```
$ tree .
.
├── Dockerfile
└── hello-chronos.sh

0 directories, 2 files
```

然后构建 Docker 镜像，如下所示：

```
$ docker build -t mesos-in-action/hello-chronos:v1 .
$ docker push mesos-in-action/hello-chronos:v1
```

将 Docker 镜像上传到 Docker Hub 之后，就可以提交任务了。创建一个 `hello-chronos-docker.json` 文件，内容如下：

```
{  
  "name": "hello-chronos-docker",  
  "container": {  
    "type": "DOCKER",  
    "image": "mesos-in-action/hello-chronos:v1"  
  },  
  "cpus": "0.5",  
  "mem": "512",  
  "command": ""  
}
```

然后，使用 curl 命令提交这个任务，命令行和前面的一样：

```
$ curl -H 'Content-Type: application/json' -X POST -d @hello-chronos-docker.json http://10.23.85.233:8080/scheduler/iso8601
```

任务提交后，当任务被初次调度到计算结点时，计算结点需要下载 Docker 镜像，这需要花一定的时间，视网络情况而定。由于 Docker Hub 在国内非常慢，甚至有时不可访问，所以读者也可以配置 Mesos 计算节点使用一些国内的 Docker Hub 镜像服务，例如：道客云提供永久免费的 Docker Hub 镜像服务。

另外，在实际生产环境中，往往选择搭建私有的 Docker 镜像服务，这部分内容这里不再介绍。

小结

本节介绍了 Mesos 上的短时任务框架 Chronos，包括 Chronos 特性，怎样搭建 Chronos 服务，搭建高可用的 Chronos 服务，以及怎样使用 Chronos。

Spark

Spark 是一个大数据处理框架，近几年社区发展非常快，从最初的 Map-Reduce 计算模型扩展到 SQL 结构化数据，机器学习，图计算，流计算等。

Spark 和 Mesos 都出自加州大学伯克利分校的 AMPLab，Spark 也出现在 Mesos 原型论文中，和 Mesos 一样，Spark 也贡献给了 Apache 软件基金会，并且很快成长为顶级项目。

本节将介绍 Spark 基础，怎样搭建 Spark 集群，Spark on Mesos 集群以及怎样使用 Spark。

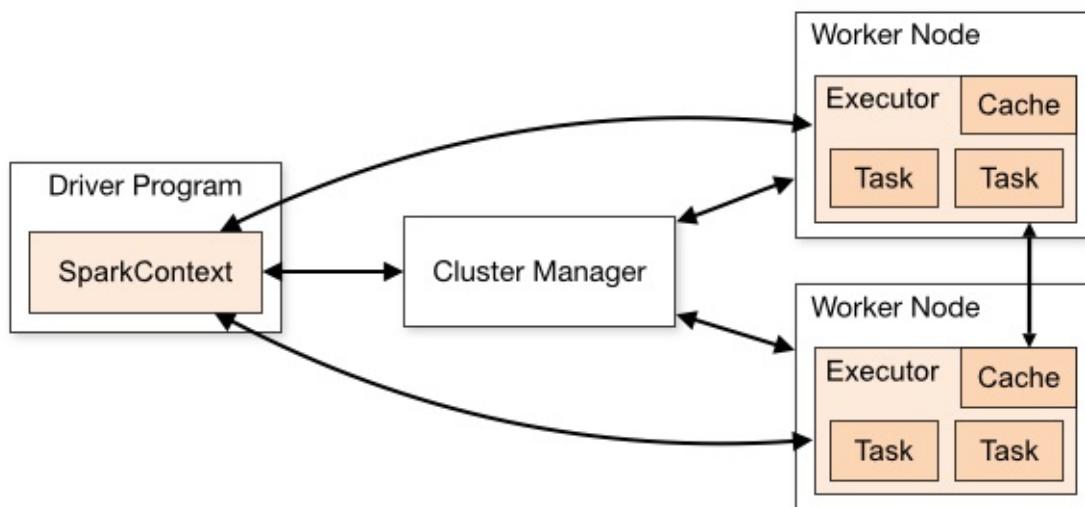
Spark 简介

Spark 是一个快速、通用的大数据处理引擎。在速度上，Spark 宣称和 Hadoop 相比，有百倍的提升，得益于其基于内存的计算模型；但是，即使使用磁盘，也有十倍的速度提升。

Spark 支持多种高级编程语言，如：Java, Scala, Python, R 等；并且提供了许多高级操作符，大大降低用户编写 Spark 任务的难度。

Spark 集群模型

Spark 可以单独以集群的方式运行，也可以运行在 Mesos, YARN 之上。Spark 包含以下几种组件。



Cluster Manager 扮演集群资源管理者，例如：Mesos, YARN，或者是 Spark 本身；**Driver Program** 负责和 **Cluster Manager** 协调，申请资源，并且调度根据资源调度任务到相应的 **Worker Node** 上。**Executor** 负责启动具体的任务进程，并且实现和 **Driver Program** 之间的协调，完成任务的管理。

这一点和前面介绍的 Marathon 和 Chronos 不一样，Marathon 和 Chronos 对任务都是不可知的，它们实际上只对任务的状态感兴趣，完全不介入任务的逻辑。而 Spark 是一个大数据处理框架，它需要切割，组装任务，提供任务间缓存等。

搭建Spark服务

Spark 可以以多种方式运行，最简单的为本地运行方式，适合于开发测试；还可以以 Spark 集群的方式运行；运行在 Mesos 或者 YARN 上。

这里将以两种方式搭建 Spark 服务：集群方式和 Mesos 方式。

Spark 集群

在生产环境中搭建 Spark 服务，最简单的应该就是以独立 Spark 集群的方式提供 Spark 服务，因为这种方式更简单，不用处理与其它框架整合，例如：Mesos 和 YARN，然后不可避免的还需要和其它计算框架共享集群。

在开始之前，首先下载 Spark，在编写本书之际，Spark 最新的稳定版本是 1.5.2，可以从 Spark 官网上下载，如下图所示：

The screenshot shows the Apache Spark homepage at spark.apache.org. The main header features the Spark logo and the tagline "Lightning-fast cluster computing". Below the header is a navigation bar with links for Download, Libraries, Documentation, Examples, Community, and FAQ. A prominent callout box highlights that "Apache Spark™ is a fast and general engine for large-scale data processing." To the right, there's a "Latest News" sidebar with recent announcements. In the center, a chart compares the running time of logistic regression between Hadoop and Spark. The chart shows that Spark runs significantly faster than Hadoop.

System	Running time (s)
Hadoop	110
Spark	0.9

Logistic regression in Hadoop and Spark

[Download Spark](#)

点击页面中的 `Download Spark` 后，跳转到下载页面，如下图所示：

The screenshot shows the official Apache Spark website at spark.apache.org. The main navigation bar includes links for Download, Libraries, Documentation, Examples, Community, and FAQ. The 'Download Spark' section is highlighted, showing the latest release (Spark 1.5.2) and download options. A sidebar on the right lists 'Latest News' items such as 'CFP for Spark Summit East 2016 is closing soon!' and 'Spark 1.5.2 released'.

由于 Spark 使用了 Hadoop client 来访问 HDFS，所以 Spark 需要基于 Hadoop 来构建，Spark 下载页面提供了多种下载选择：

- 源代码
- 不包含 Hadoop 的可执行文件
- 针对各个特性 Hadoop 版本的可执行文件，例如：Hadoop 2.6 及更新版本

这里下载 Pre-built for Hadoop 2.6 and later，下载的文件名为：spark-1.5.2-bin-hadoop2.6.tgz，

为了启动集群，需要将 Spark 部署到各个结点中，Spark 集群中只有一个控制结点，其余均为计算结点。

这里依然假设有 3 台服务器 A, B, C 用来搭建 Spark 集群，其中 A 将作为 Spark 控制结点，运行 Spark master 进程，而 B 和 C 将作为计算节点，运行 Spark worker 进程。

将下载好的 spark-1.5.2-bin-hadoop2.6.tgz 复制到 3 台服务器的 /opt/spark 目录下，并且解压。

```
# mkdir -p /opt/spark
# cd /opt/spark
# tar xzf spark-1.5.2-bin-hadoop2.6.tgz
# cd spark-1.5.2-bin-hadoop2.6
# ls
bin  CHANGES.txt  conf  data  ec2  examples  lib  LICENSE  licenses  NOTICE  python  R
  README.md  RELEASE  sbin
```

Spark master

在服务器 A 上启动 Spark master 进程，如下：

```
# cd /opt/spark/spark-1.5.2-bin-hadoop2.6
# ./sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /opt/spark/spark-1.5.2-bin-
hadoop2.6/sbin/../logs/spark-root-org.apache.spark.deploy.master.Master-1-10.23.85.233
.out
```

Spark master 进程默认会监听本地地址的以下端口：

- 8080 , Spark master web 用户界面，可以用浏览器查看
- 7077 , Spark master 服务端口，worker 进程将和 master 进程通过此端口建立连接

注意：如果读者在这台机器上启动了 Marathon 或者 Chronos 占用了 8080 端口，需要在启动前使用环境变量修改端口号

```
# EXPORT SPARK_MASTER_WEBUI_PORT=8081
# ./sbin/start-master.sh
```

同样，也可以使用环境变量 SPARK_MASTER_PORT 修改默认的监听端口 7077。

现在，打开浏览器，指向本地地址的 8080 端口，将可以看到 Spark 集群的运行情况，如下图所示：

The screenshot shows the Spark Master web interface at `spark://10.23.85.233:7077`. The top section displays basic cluster statistics: URL (spark://10.23.85.233:7077), REST URL (spark://10.23.85.233:6066), Alive Workers (0), Cores in use (0 Total, 0 Used), Memory in use (0.0 B Total, 0.0 B Used), Applications (0 Running, 0 Completed), Drivers (0 Running, 0 Completed), and Status (ALIVE). Below this are three tables: 'Workers' (empty), 'Running Applications' (empty), and 'Completed Applications' (empty).

从 Spark master web 用户界面，可以看到以下几方面的信息：

- 基本信息：URL，REST URL，Alive Workers 等
- Works : 当前的 work
- Running Applications : 正在运行的应用
- Completed Applications : 已经完成的应用

可以看到，目前集群中还没有任何可用的计算结点，所以也不能执行任何任务。现在，让我们向集群中添加两个计算结点。

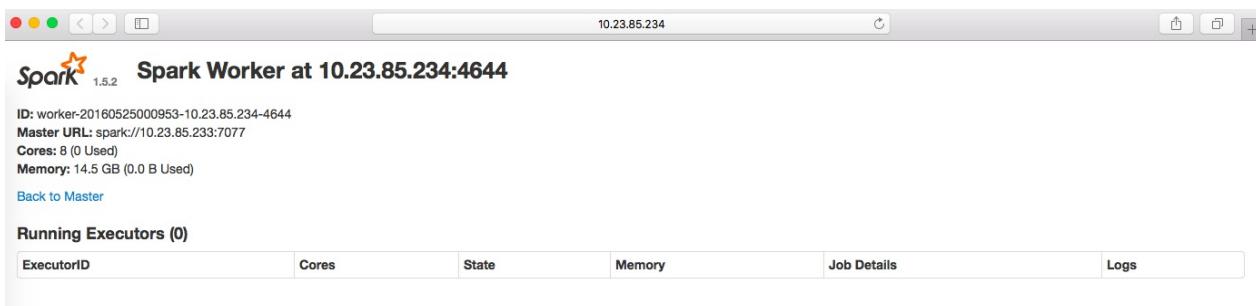
Spark worker

Spark worker 结点充当 Spark 的计算节点，负责和 Spark 控制节点通信并且运行任务。在启动了 Spark 控制节点后，现在分别在服务器 B 和 C 上执行下面的操作来启动 Spark worker 进程。

```
$ cd /opt/spark/spark-1.5.2-bin-hadoop2.6
$ ./sbin/start-slave.sh spark://10.23.85.233:7077
starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/spark-1.5.2-bin-
hadoop2.6/sbin/.../logs/spark-root-org.apache.spark.deploy.worker.Worker-1-10.23.85.235
.out
```

`start-slave.sh` 接收一个必需的参数，指定 Spark master 运行的地址。Spark worker 在启动之后，会向 Spark master 注册自己，并且汇报可用的计算资源，默认情况下，将汇报所有可用的 CPU 资源以及所有内存减去 1GB，例如：如果计算结点配备了 8GB 的内存，那么 Spark worker 将汇报有 7GB 可用内存，目的是为计算节点自身稳定运行预留 1GB 内存。

Spark worker 默认会监听在本地地址的 8081 端口，提供 web 用户界面，所以，用浏览器打开 B 结点的 8081 地址，如下图所示：



从 Spark worker web 用户界面，可以了解到以下几方面信息：

- 基本信息：ID，Master URL，核数，内存大小
- Running Executors：正在运行的 Executors

由于这里没有运行任何任务，Running Executors 中为空。

另外，再次打开 Spark master 页面，可以看到刚才添加的 2 个 Spark worker，如下图所示：

The screenshot shows the Spark Master UI at `spark://10.23.85.233:7077`. It displays the following information:

- Cluster Summary:**
 - URL: `spark://10.23.85.233:7077`
 - REST URL: `spark://10.23.85.233:6066 (cluster mode)`
 - Alive Workers: 2
 - Cores in use: 16 Total, 0 Used
 - Memory in use: 29.0 GB Total, 0.0 B Used
 - Applications: 0 Running, 0 Completed
 - Drivers: 0 Running, 0 Completed
 - Status: ALIVE
- Workers:** A table showing two workers:

Worker Id	Address	State	Cores	Memory
worker-20160525000953-10.23.85.234-4644	10.23.85.234:4644	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)
worker-20160525001030-10.23.85.235-57654	10.23.85.235:57654	ALIVE	8 (0 Used)	14.5 GB (0.0 B Used)
- Running Applications:** An empty table.
- Completed Applications:** An empty table.

启动参数配置

在上面启动 Spark master 和 worker 进程时，对于可选参数，都使用了默认值，在大多数情况下，这都工作得非常好，但是有些时候可能需要调整一些默认参数。下面分别是 `start-master.sh` 和 `start-slave.sh` 接受的可选参数。

公共参数

`start-master.sh` 和 `start-slave.sh` 接受以下公共参数。

参数	默认值	含义
<code>-h HOST, --host HOST</code>	本地所有地址	监听的地址，例如： 10.23.85.233
<code>-i HOST, --ip HOST</code>	本地所有地址	不再建议使用，建议使用 <code>-h</code> 或者 <code>--host</code>
<code>-p PORT, --port PORT</code>	对于 master 来说是 7077，slave 为随机端口	master 或 slave 和对方通信的端口
<code>--webui-port PORT</code>	对于 master 来说是 8080, slave 是 8081	master 或 slave web 用户界面监听的端口
<code>--properties-file FILE</code>	<code>conf/spark-defaults.conf</code>	属性配置文件

start-master.sh 参数

除了公共参数外，`start-master.sh` 并没有其它特有的参数。

start-slave.sh 参数

除了公共参数外，`start-slave.sh` 还有一些特有的参数：

参数	默认值	含义
-c CORES, --cores CORES	本机所有的 CPU	向 master 注册的可用 CPU 核数
-m MEM, --memory MEM	本机所有的内存减去 1GB	向 master 注册的可用内存
-d DIR, --work-dir DIR	SPARK_HOME/work	本地工作目录

例如，我们可以修改 `--cores` , `--memory` 为本机预留更多的资源，以便计算结点自身更加稳定。只需要在启动 `start-slave.sh` 时指定即可，例如：

```
$ ./sbin/start-slave.sh --cores 7 --memory 14G spark://10.23.85.233:7077
```

集群辅助脚本

除了以 `start-master.sh` , `start-slave.sh` 分别启动 `master` 和 `slave` 进程外，Spark 还提供了一次性启动或停止集群的方式，包括以下脚本，它们都位于 Spark 的 `sbin` 目录下。

- `start-slaves.sh`, 在所有 `conf/slaves` 文件中指定的计算节点上启动 `slave` 进程
- `start-all.sh`, 启动 `master` 和所有 `slave` 进程
- `stop-slaves.sh` - 在所有 `conf/slaves` 文件中指定的计算节点上停止 `slave` 进程
- `stop-all.sh` - 停止整个集群，包括 `master` 和 `slave` 进程

这些脚本都通过 SSH 的方式来工作，所以需要事先在其它结点上配置好基于公钥认证的 SSH 登陆配置。详细配置方法可以参考本章开始。

高可用性

Spark 集群的高可用性包含两方面：Spark master 的高可用性和 Spark worker 的高可用性。

Spark worker 的高可用性通过将运行在其上的任务转移到其它结点来实现，所以 Spark 本身即实现了 worker 的高可用性。

而 Spark master 在整个集群中只有一个，所以是一个单点故障，从而需要借助其它方式来实现 master 的高可用性。Spark 提供了两种方式：

- 基于 ZooKeeper
- 基于本地文件

基于 ZooKeeper

在前面的章节已经介绍过 ZooKeeper 的基础知识，以及怎样搭建 ZooKeeper 服务。而且包括：Mesos, Marathon, Chronos 都无一例外的使用了 ZooKeeper，Spark 也不例外，使用 ZooKeeper 作为其 master 高可用性方案也是首选。

和 Marathon, Chronos 类似，Spark master 也可以将集群元数据存储在 ZooKeeper 之中，以便在重新启动，或者其它 master 被选举为 Leader 时重新加载数据，从而继续提供服务。

所以，基于 ZooKeeper 实现 Spark master 的高可用性非常容易理解：在不同机器上以相同的配置启动多个 Spark master 进程，都注册到同一个 ZooKeeper 服务中即可。相关配置如下：

- spark.deploy.recoveryMode, 设置为 ZOOKEEPER
- spark.deploy.zookeeper.url, 设置 ZooKeeper 集群地址
- spark.deploy.zookeeper.dir, ZooKeeper 中用来存储数据的目录，默认为 /spark

这些参数都通过 SPARK_DAEMON_JAVA_OPTS 来设置，这个变量可以在 conf/spark-env.sh 中设置。例如：

```
$ cd /opt/spark/spark-1.5.2-bin-hadoop2.6
$ cp conf/spark-env.sh.template conf/spark-env.sh
```

默认这个文件中没有开启任何配置，所以在这个文件中添加如下一行来配置上面的参数。

```
SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER -Dspark.deploy.zookeeper.url=10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181
```

注意，需要修改所有 Spark 控制结点上的配置。同时，由于现在启动了多个 Spark master 进程，当当前作为 Leader 的 master 故障时，为了能够让 worker 能够自动注册到新的 Leader 结点，worker 在启动时需要同时指定所有 Spark master。如下：

```
./sbin/start-slave.sh spark://host1:port1,host2:port2
```

如果我们在服务器 A 和 B 上启动了 Spark master，那么可以以下面的方式启动 worker

```
./sbin/start-slave.sh spark://10.23.85.233:7077,10.23.85.234:7077
```

基于本地文件

Spark master 除了能够将集群元数据存储在 ZooKeeper 之外，还可以将数据存储在本地文件，只需要配置以下两个参数：

- spark.deploy.recoveryMode, 设置为 FILESYSTEM

- spark.deploy.recoveryDirectory, 设置为一个本地目录

和基于 ZooKeeper 方式一样，通过设置 conf/spark-env.sh 文件中的 SPARK_DAEMON_JAVA_OPTS 变量即可。

基于本地文件方式的高可用性假设本地文件系统足够可靠，并且能够及时重新启动 Spark master 进程。所以，这种方式本质上不能称作高可用性，也不推荐在生产环境中使用。因为首先本地文件并不是可靠的，硬件可能损坏，文件系统可能损坏，操作系统也可能损坏，所以很可能需要花很长一段时间来恢复服务。

当然，为了避免本地文件系统的不可用性，读者也可以使用网络文件系统，但这或许会引入性能问题或者其它复杂度的问题。所以，总是推荐在生产环境中使用基于 ZooKeeper 的高可用性实现方式。

Spark on Mesos

上面介绍了 Spark 以独立的方式搭建集群，Spark 还可以和 Mesos 结合，运行在 Mesos 集群之上。

Spark 最早出现是在 Mesos 的论文里，作为一个运行在 Mesos 之上的框架，用来证明 Mesos 是一种可以有效管理集群，且提高集群综合利用率的方案。所以，不言而喻，Spark 可以运行在 Mesos 之上。

Spark 可以以两种方式运行在 Mesos 之上：

- Client 模式
- Cluster 模式

在 Client 模式下，Spark 以 mesos scheduler 的方式直接和任务耦合在一起。直接运行在提交任务的本地节点上。这种方式比较适合提交任务的节点和 Mesos 集群的计算节点相邻，或者位于同一个子网中，以便 spark driver 和 executor 之间能够保证高速通信。

而在 Cluster 模式下，Spark scheduler 不再和任务耦合在一起，任务是从客户端提交的，spark scheduler(driver) 运行在 Mesos 集群中。

这里只介绍 Cluster 模式。

启动 Spark Scheduler

假设 Mesos 线上服务为

```
zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos °
```

将 spark-1.5.2-bin-hadoop2.6.tgz 复制到 10.23.85.234 结点上，然后启动 Spark Scheduler，命令如下：

```
$ tar xzf spark-1.5.2-bin-hadoop2.6.tgz
$ cd spark-1.5.2-bin-hadoop2.6
$ ./sbin/start-mesos-dispatcher.sh --master mesos://zk://10.23.85.233:2181,10.23.85.234:2181,10.23.85.235:2181/mesos
```

`start-mesos-dispatcher.sh` 脚本接受一个参数 `--master`，指明 `mesos` 集群的地址，这里的格式为 `mesos://<path>`，其中 `mesos://` 为地址前缀，`<path>` 为 `mesos` 集群地址，这里可以是单个 `mesos master` 地址也可以是 `mesos` 集群在 `ZooKeeper` 中的地址，这里我们使用了在上一章中搭建了 `mesos` 生产集群。

启动了 `Spark Scheduler` 之后，可以从 `Mesos master web` 用户页面看到其已经注册了，如下图所示：

The screenshot shows the Mesos master web interface at `http://10.23.85.233`. The top navigation bar includes tabs for Mesos, Frameworks, Slaves, and Offers. The current view is under the 'Frameworks' tab, specifically the 'Master / Frameworks' section. It displays two sections: 'Active Frameworks' and 'Terminated Frameworks'. In the 'Active Frameworks' section, there is one entry:

ID	Host	User	Name	Active Tasks	CPUs	Mem	Max Share	Registered	Re-Registered
...ae3c-d05b17cf5f0-0000	10.23.85.234	root	Spark Cluster	0	0	0 B	0%	3 minutes ago	-

In the 'Terminated Frameworks' section, there are no entries.

用浏览器打开 `http://10.23.85.234:8081`，可以看到如下 spark 页面：

The screenshot shows the Spark Driver for Mesos cluster interface at `http://10.23.85.234`. The page title is "Spark Drivers for Mesos cluster". It displays four sections: "Queued Drivers", "Launched Drivers", "Finished Drivers", and "Supervise drivers waiting for retry". Each section has a table with columns for Driver ID, Submit Date, Main Class, and other relevant details.

现在，用户可以通过 Spark 客户端向 Spark Scheduler 提交任务了，只需要指明 Spark Scheduler 的服务地址即可，这里为：`mesos://10.23.85.234:7077`。

运行模式

Spark 在 Mesos 上可以有两种运行模式：

- `fine-grained`
- `coarse-grained`

`fine-grained` 为默认模式，在该模式下，每个 Spark 提交在 Mesos 上都会注册一个框架，而该次提交内部的每个任务则会通过 Mesos 来调度，调度每个任务都需要获取资源、调度任务、执行、完成任务释放资源。

这种调度方式的优点是 Spark 各个提交中之间动态共享资源，并且和其它 Mesos 框架也是动态共享资源，缺点是单次任务都需要经过完整的 Mesos 调度周期，调度较慢；所以，不太适合用于低延时的任务，例如，交互式查询等。

`coarse-grained` 则是另一种粒度更粗的方式，这种方式只会在有资源的 Mesos 计算结点上启动 Worker，而后续任务的调度则由 Spark Scheduler 直接将任务分发给 Worker，从而避免完整的 Mesos 调度周期。所以，`coarse-grained` 方式调度更快，但是会长期占用资源。

运行模式可以通过 `SparkConf` 的键 `spark.mesos.coarse` 来配置，值为 `true|false`。需要注意的是：`coarse-grained` 模式默认会使用所有 Mesos 分配的 Offer，所以为了避免占用过多的资源，总是应该设置 `SparkConf` 的 `spark.cores.max` 为一个合理值。

使用 Docker

Spark 支持启动 Docker 容器来运行 Spark 任务，这要求 Mesos 版本等于或高于 0.21.0，因为 Mesos 在 0.21.0 中引入了对 Docker 的支持。如果要使用 Docker 容器来执行 Spark 任务，需要再 Docker 镜像中配置好 Spark 执行环境。可以通过 `SparkConf` 的 `spark.mesos.executor.docker.image` 来配置要使用的镜像。

Spark on Mesos 配置

Spark 具有非常高的可配置性，所以也有非常多的配置项，这里不再介绍 Spark 通用的配置项，读者可以参考官方文档或者其它 Spark 书籍。

这里将介绍特定于 Spark on Mesos 的配置，如下表所示：

配置项	默认值	含义
<code>spark.mesos.coarse</code>	<code>false</code>	是否使用 <code>coarse-grained</code> 模式运行
<code>spark.mesos.extra.cores</code>	0	只在 <code>coarse-grained</code> 模式下使用，表示使用额外的多少核 CPU 来启动 worker，但是所有 worker 的 CPU 之和不能超过 <code>spark.cores.max</code>
<code>spark.mesos.mesosExecutor.cores</code>	1.0	只在 <code>fine-grained</code> 模式下使用，表示 Spark Executor 额外使用的 CPU 核

<code>spark.mesos.executor.docker.image</code>	空	启动 Spark Executor 使用的 Docker 镜像
<code>spark.mesos.executor.docker.volumes</code>	空	启动 Spark Executor 容器时挂载的卷
<code>spark.mesos.executor.docker.portmaps</code>	空	启动 Spark Executor 容器时映射的端口
<code>spark.mesos.executor.home</code>	Spark Driver 中设置的 SPARK_HOME	当没有设置 <code>spark.executor.uri</code> 时，Spark 需要从该值中找到 Executor 目录并启动 Executor
<code>spark.mesos.executor.memoryOverhead</code>	<code>spark.executor.memory</code> 的十分之一，最小为 384MB	每个 Executor 额外的内存，为了保证不占用 task 的内存
<code>spark.mesos.uris</code>	空	uris 是 Mesos 提供的初始化运行环境的功能，在启动任务之前，会将 uris 指定的资源下载到任务启动目录中
<code>spark.mesos.principal</code>	空	Mesos 认证需要的 principal
<code>spark.mesos.secret</code>	空	Mesos 认证需要的 secret
<code>spark.mesos.role</code>	*	Spark Scheduler 使用的角色，角色是 Mesos 用来分配计算资源的一种方法
<code>spark.mesos.constraints</code>	空	根据 Mesos 计算节点的标签来设置过滤规则，任务将只被调度到符合条件的计算节点上

运行 Spark 任务

在上一节中我们介绍了怎样搭建一个运行在 Mesos 集群之上的 Spark 服务，并且可以通过 ZooKeeper 实现 Spark 服务的高可用性。

本节将介绍怎样提交 Spark 任务，以及怎样将 Spark 任务运行在 Docker 中。

Spark 提供了多种编程语言绑定，包括：Java, Scala, Python, R 等。介于目前 Java 在大数据领域的绝对优势，这里介绍怎样提交 Java 任务，另外，由于 Scala 也是一种 JVM 语言，提交 Scala 任务和提交 Java 任务的方式类似。

spark-submit

Spark 提供了一个工具用来提交任务到 Spark 服务中，即：`spark-submit`，在 `bin` 目录下。`spark-submit` 支持许多参数来定制提交任务的行为，包括一些必要的参数和可选的参数。

`spark-submit --help` 能够打印出 `spark-submit` 的帮助文档，下面介绍一些常用的参数。

参数	含义
<code>--master MASTER_URL</code>	指定 spark 服务地址，例如： <code>mesos://10.23.85.234:7077</code>
<code>--deploy-mode DEPLOY_MODE</code>	指定提交任务的方式，合法值为 <code>client</code> 或者 <code>cluster</code> ，默认为 <code>client</code>
<code>--class CLASS_NAME</code>	指定将要被运行的 Java/Scala class 名称
<code>--name NAME</code>	任务的名称
<code>--conf PROP=VALUE</code>	设置 Spark 属性
<code>--properties-file FILE</code>	指定 Spark 配置文件，默认为 <code>conf/spark-defaults.conf</code>
<code>--driver-memory MEM</code>	指定 spark driver 使用的内存，默认为 1024M
<code>--executor-memory MEM</code>	指定 spark executor 使用的内存，默认为 1G

以下参数在 `standalone` 和 `Mesos` 集群方式下有效。

参数	含义
<code>--total-executor-cores NUM</code>	所有 executor 占用的 cpu 核数

以下参数只有在 `standalone` 和 `Mesos` 集群方式，并且在 `--deploy-mode` 为 `cluster` 时才有效。

参数	含义
--supervise	如果指定，当 spark driver 退出时会被自动重启
--kill SUBMISSION_ID	停止任务
--status SUBMISSION_ID	查看任务状态

提交任务

在学习了 `spark-submit` 常用参数后，下面使用 `spark-submit` 来提交任务。提交一个 Java/Scala 任务，需要指定以下参数：

- `--master`
- `--class`

以提交 `SparkPi` 这个示例应用为例，在 10.23.85.235 上 /home/spark/spark-1.5.2-bin-hadoop2.6/bin 目录中，执行下面的命令。

```
$ ./spark-submit --master mesos://10.23.85.234:7077 --class org.apache.spark.examples.SparkPi --name "Spark PI example" ../lib/spark-examples-1.5.2-hadoop2.6.0.jar 100
FIXME: output
```

上面的命令首先指定了 spark 服务地址，这里是上节我们搭建的 `spark on mesos` 服务地址，然后指定了要运行的类名称，然后还指定了本次任务的名称，最后指定了包含任务内容的 jar 文件。

`spark-submit` 默认会以 `client` 的方式提交任务，所以 `spark driver` 会在本地前台执行，任务的输出会直接通过终端打印出来。

如果修改上面的命令，添加 `--deploy-mode cluster` 参数，那么任务将会以 `cluster` 的方式运行，如下所示：

```
$ ./spark-submit --master mesos://10.23.85.234:7077 --class org.apache.spark.examples.SparkPi --name "Spark PI example" --deploy-mode cluster ../lib/spark-examples-1.5.2-hadoop2.6.0.jar 100
FIXME: output
```

通常来说，对于运行在 Mesos 上的 Spark 任务，我们都可以指定 `executor` 能够使用的资源上限，避免 spark 任务占用过多的资源，例如：

```
$ ./spark-submit --master mesos://10.23.85.234:7077 --class org.apache.spark.examples.SparkPi --name "Spark PI example" --deploy-mode cluster --total-executor-cores 10 ./lib/spark-examples-1.5.2-hadoop2.6.0.jar 100
FIXME: output
```

由于 executor 默认会占用 1GB 内存，1 核 CPU，所以本次提交最多占用 10 核 CPU，10GB 内存。

对于 Spark 任务来说，通常需要指定较大的 executor 内存，因为 Spark 专门为内存计算设计，在较大内存中更容易获得更快的执行速度，例如：为每个 executor 指定 20GB 内存大小，甚至更大，由于本书的实验环境虚拟机内存大小为 16GB，所以不宜指定太大的 executor 内存。

还需要注意的是，spark 任务的 jar 应该要包含除 spark 之外的其它所有依赖，以免集群中的节点并没有配置所有的依赖包。

集成 Docker

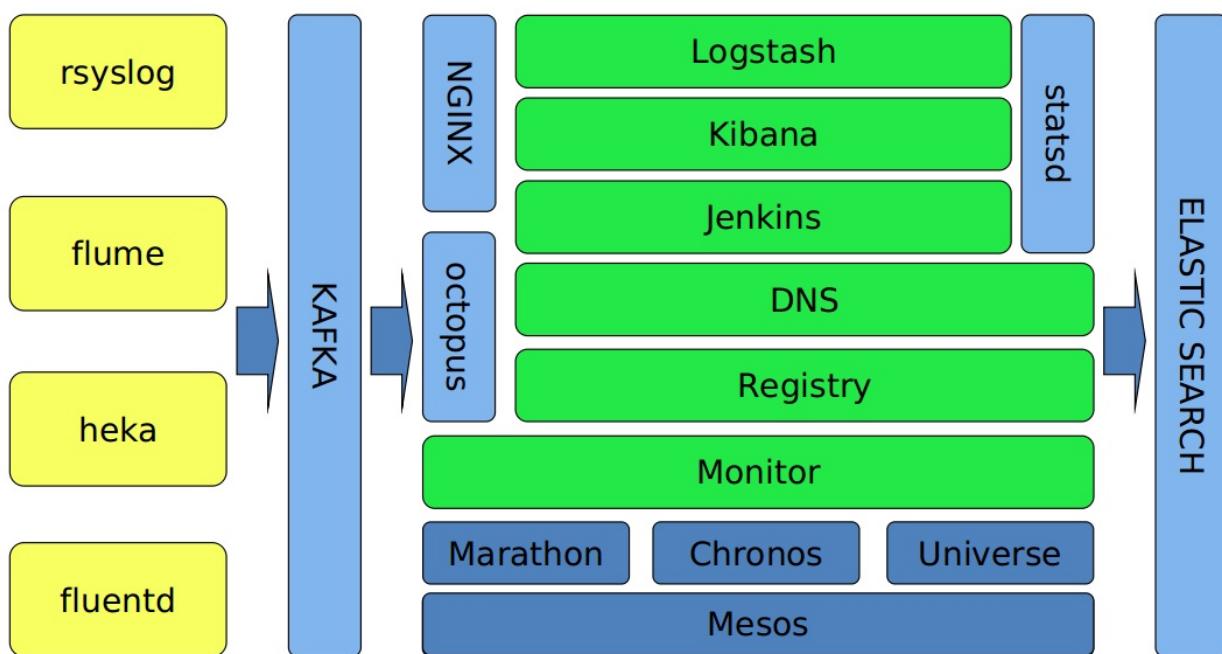
小结

总结

Mesos搭建日志处理系统实战

平台介绍

我们是在今年的5月份开始调研并尝试使用Mesos，第一个试点就是我们的日志平台，我们将日志分析全部托管在Mesos平台上。日志平台面向业务线开发、测试、运营人员，方便定位、追溯线上问题和运营报表。



这个是我们平台的结构概览。

日志分析我们使用ELK（Elasticsearch、Logstash、Kibana），这三个应该说是目前非常常见的工具了。而且方案成熟，文档丰富，社区活跃（以上几点可以作为开源选型的重要参考点）。稍微定制了下Kibana和Logstash，主要是为了接入公司的监控和认证体系。

日志的入口有很多，如kernel、mail、cron、dmesg等日志通过rsyslog收集。业务日志通过flume收集，容器日志则使用mozilla的heka和fluentd收集。

这里稍稍给heka和fluentd打个广告，两者配合收集Mesos平台内的容器日志非常方便，可以直接利用MESOS_TASK_ID区分容器（此环境变量由Mesos启动容器时注入）。而且我们也有打算用heka替换logstash。

Mesos技术栈

下面主要分享一下Mesos这块，我们使用了两个框架：Marathon和Chronos，另外自己开发了一个监控框架Universe。

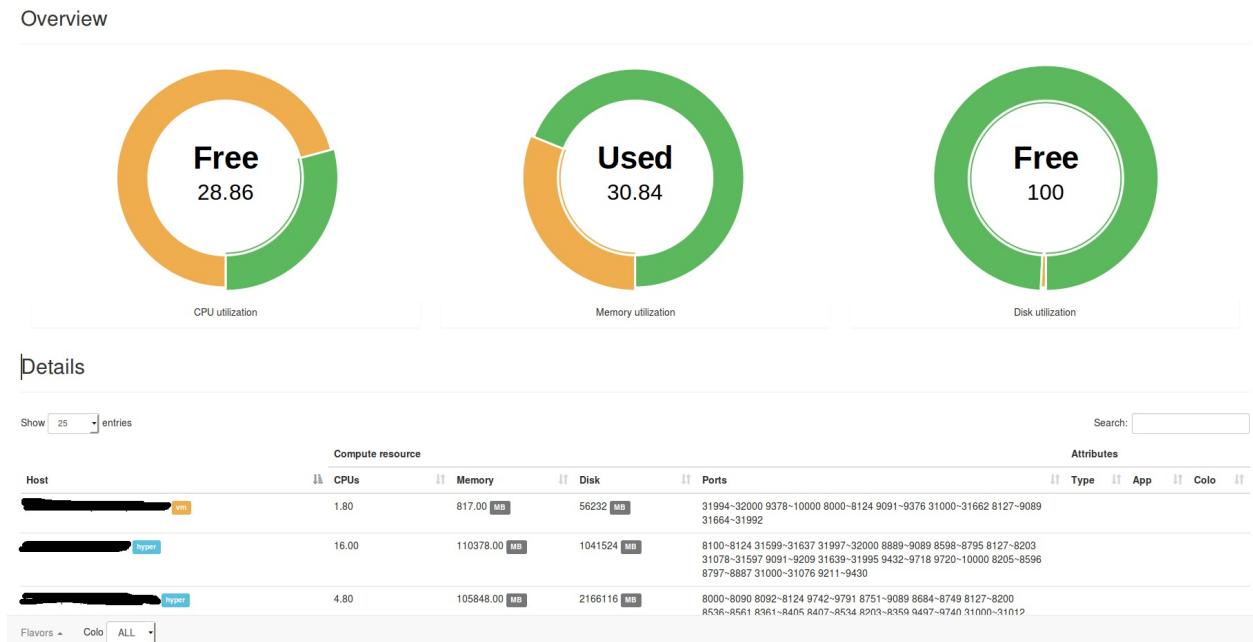
先说Marathon，eventSubscriptions是个好功能，通过它的httpcallback可以有很多玩法，群里经常提到的bamboo就是利用这个功能做的。利用好这个功能，做容器监控就非常简单了。

接着是Marathon的重启（更新），推荐设置一下minimumHealthCapacity，这样可以减少重启（更新）时的资源占用，防止同时启动多个运行实例时消耗过多集群资源。

服务发现，Marathon提供了servicerouter.py导出haproxy配置，或者是bamboo，但是我们现在没有使用这两个。而是按协议分成了两部分，HTTP协议的服务是使用OpenResty开发了一个插件，动态加载Marathon（Mesos）内的应用信息，外部访问的时候proxy_pass到Mesos集群内的一个应用，支持upstream的配置。

非HTTP的应用，比如集群内部的statsd的UDP消息，我们就直接用Mesos DNS + 固定端口来做了。随即端口的应用依赖entrypoint拉取域名+端口动态替换。

带有UNIQUE attribute的应用，官方目前还无法做到自动扩容，从我们的使用情况来看，基于UNIQUE方式发布的应用全部是基础服务，比如statsd、heka（收集本机的Docker日志）、cAdvisor（监控容器）等，集群新加机器的时候Marathon不会自动scale UNIQUE实例的数量，这块功能社区正在考虑加进去。我们自己写了一个daemon，专门用来监控UNIQUE的服务，发现有新机器就自动scale，省的自己上去点了。



另外一个问题，资源碎片化，Marathon只是个框架，关注点也不在这里。Mesos的UI里虽然有统计，但是很难反应真实的情况，于是我们就自己写了一个Mesos的框架，专门来计算资源碎片和真实的余量，模拟发布情况，这样我们发布新应用或者扩容的时候，就知道集群内真实的资源余量能否支持本次发布，这些数据会抄送一份给我们的监控/报警系统。Chronos我们主要是跑一些定时清理和监控的脚本。

Docker这块，我们没有做什么改动，网络都使用host模式。Docker的监控和日志上面也提到了，我们用的是cAdvisor和heka，很好很强大，美中不足的是cAdvisor接入我们自己的监控系统要做定制。

我们也捣鼓了一个Docker SSH Proxy，可能是我们更习惯用虚拟机的缘故吧，有时候还是喜欢进入到容器里去干点啥的（其实业务线对这个需求更强烈），就是第一张图里的octopus，模拟docker exec -it的工作原理，对接Mesos和Marathon抓取容器信息。这样开发人员在自己机器上就能SSH到容器内部debug了，也省去了申请机器账号的时间了。

应用方案

接着说说我们的日志平台。这个平台的日志解析部分全部跑在Mesos上，平台自身与业务线整合度比较深，对接了一些内部系统，主要是为了考虑兼容性和业务线资源复用的问题，我尽量省略与内部系统关联的部分，毕竟这块不是通用性的。

平台目前跑了有600+的容器，网络是Docker自带的host模式，每天给业务线处理51亿+日志，延时控制在60~100ms以内。

最先遇到的问题是镜像，是把镜像做成代码库，还是一个运行环境？或者更极端点，做一个通用的base image？结合Logstash、heka、statsd等应用特点后，我们发现运行环境更适合，这些应用变化最大的经常是配置文件。所以我们先剥离配置文件到GitLab，版本控制交给GitLab，镜像启动后再根据tag拉取。

另外，Logstash的监控比较少，能用的也就一个metrics filter，写Ruby代码调试不太方便。索性就直接改了Logstash源码，加了一些监控项进去，主要是监控两个Queue的状态，顺便也监控了下EPS和解析延时。

Kafka的partition lag统计跑在了Chronos上，配合我们每个机房专门用来引流的Logstash，监控业务线日志的流量变得轻松多了。



容器监控最开始是自己开发的，从Mesos的接口里获取的数据，后来发现hostname：UNIQUE的应用Mesos经常取不到数据，就转而使用cAdvisor了，对于Mesos/Marathon发布应用，cAdvisor需要通过libcontainer读取容器的config.json文件，获取ENV列表，拿到MESOS_TASK_ID和MARATHON_APP_ID，根据这两个值做聚合后再发到statsd里（上面提到的定制思路）。

发布这块我们围绕这Jenkins做了一个串接。业务线的开发同学写filter并提交到GitLab，打tag就发布了。发布的时候会根据集群规划替换input和output，并验证配置，发布到线上。本地也提供了一个sandbox，模拟线上的环境给开发人员debug自己的filter用。

任务列表							
序号	构建名称	状态	索引大小	索引分片	马拉松配置	数据库配置	操作
1	[REDACTED]	运行	5.39 TB	Count:6 Shards:5	LCPU:1.5 LMEM:2048.0 LINS:17 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:17 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
2	[REDACTED]	运行	2.67 TB	Count:6 Shards:5	LCPU:1.5 LMEM:2048.0 LINS:12 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:12 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
3	[REDACTED]	运行	2.40 TB	Count:6 Shards:5	LCPU:1.5 LMEM:2048.0 LINS:28 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:10 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
4	[REDACTED]	运行	2.40 TB	Count:13 Shards:1	LCPU:1.5 LMEM:2048.0 LINS:10 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:2 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
5	[REDACTED]	运行	2.22 TB	Count:6 Shards:5	LCPU:1.5 LMEM:2048.0 LINS:16 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:15 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
6	[REDACTED]	运行	840.15 GB	Count:6 Shards:1	LCPU:1.5 LMEM:2048.0 LINS:8 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:1.5 LMEM:2048.0 LINS:8 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
7	[REDACTED]	运行	649.22 GB	Count:12 Shards:1	LCPU:2.0 LMEM:2048.0 LINS:5 KCPU:0.2 KMEM:512.0 KINS:1	LCPU:2.0 LMEM:2048.0 LINS:5 KCPU:0.2 KMEM:512.0 KINS:1	<button>同步</button> <button>修改</button> <button>删除</button> <button>备份</button>
				Count:6	LCPU:1.5 LMEM:2048.0 LINS:2	LCPU:1.5 LMEM:2048.0 LINS:2	<button>同步</button> <button>修改</button>

同时发布过程中我们还会做一些小动作，比如Kibana索引的自动创建，Dashboard的导入导出，尽最大可能减少业务线配置Kibana的时间。每个应用都会启动独立的Kibana实例，这样不同业务线间的ACL也省略了，简单粗暴，方便管理。没人使用的时候自动回收Kibana容器，有访问了再重新发一个。

除了ELK，我们也在尝试Storm on Mesos，感觉这个坑还挺多的，正在努力的趟坑中。扫清后再与大家一起交流。

- Mesos 搭建企业级容器云项目概述
- Docker 基础知识介绍
- Marathon Framework 介绍
- 容器云平台基础概述
- 容器云平台搭建
- 企业级容器云的一些坑

Mesos搭建企业级容器云项目概述

随着docker的兴起，容器一词变得十分的火热，到处都可以看到容器的身影。身在云计算时代，不讨论一下容器，都不好意思说自己是做云计算的。

引领这一浪潮的，莫过于docker这一项技术。docker使用Linux的namespace和cgroups实现了容器级别的隔离，然后又加入了image这一概念，使得部署变得十分简单。现在只需要开发代码，build一个镜像，然后就可以借助docker的东风将它无障碍的部署到各种linux发行版中，甚至是Windows。很好的解决了线上环境和开发环境不同带来的问题，让程序的部署变得更加简单。

更加让人激动的是，项目被docker容器化后，就具有了可以扩容的能力。你可以简单的将项目由一个实例变成两个或者更多个而无需复杂的线上配置。而且容器化后的项目可以很好的在分布式环境中运行，自由在多个主机之间迁移。这是传统项目所不具备的。

分布式应用在传统项目里门槛很高，会有各种各样的问题。但是现在借助mesos，我们可以非常容易的管理众多主机组成的集群，就像操作一台大型机器一样。

对于长时间运行的任务，marathon framework很好的帮助我们解决了调度和监控的问题。使用它可以很方便的在mesos集群中启动容器，然后对他进行健康监控。

在集群中运行的容器随时都会被扩容或者迁移到其他机器，那么对于他们的访问，也就是服务发现，一般使用反向代理技术来解决，我们使用开源项目bamboo来实现这样的功能。

高可用也是企业级云平台一个非常重要的问题。mesos默认使用zookeeper来完成主从节点的选择和高可用，后面我们也会详细介绍。

综上是搭建一个企业级容器云平台需要的基础构建，可以看到，几乎清一色的开源项目。后面我们会以此为例子，一步步搭建一个分布式容器云平台。

FIXME: private docker-registry。

docker基础知识介绍

什么是docker，引用官方的一句话

Build, Ship and Run Any Application, Anywhere

docker就是一个这样的工具。它可以帮助开发者很方便的去构建，部署，运行自己的程序。它可以让你非常迅速的测试和部署你的项目到生产环境中。

对于docker的具体实现和原理我们不多讲，让我们直接来做一个简单的例子来体验一下docker的魅力。

首先你需要在你自己的机器上安装docker，详细的安装文档请参考 [Docker 官方文档](#)(FIXME: hard copy book isn't a browser, URL is meaningless)。

这里以在 Ubuntu 14.04 系统上安装 Docker 为例。

```
curl -sSL https://get.docker.com | sh
```

一段美妙的小脚本就被安装到了你的机器上，他完成了你安装docker需要的所有内容。下面我们就开始使用它吧。

如果我们以一个简单的小应用来演示肯定激发不了你的兴趣，那么我们以安装一个wordpress为例，看看docker是如何快速安装一个wordpress 的。

以前安装wordpress,你可能需要去了解PHP，mysql，然后还有你的服务器的系统，最后才是去安装wordpress。非常的麻烦，但是如果我们换一种方式，使用docker来安装呢。

```
docker run -d -p 80:80 --name wordpress index.alauda.cn/alauda/wordpress
```

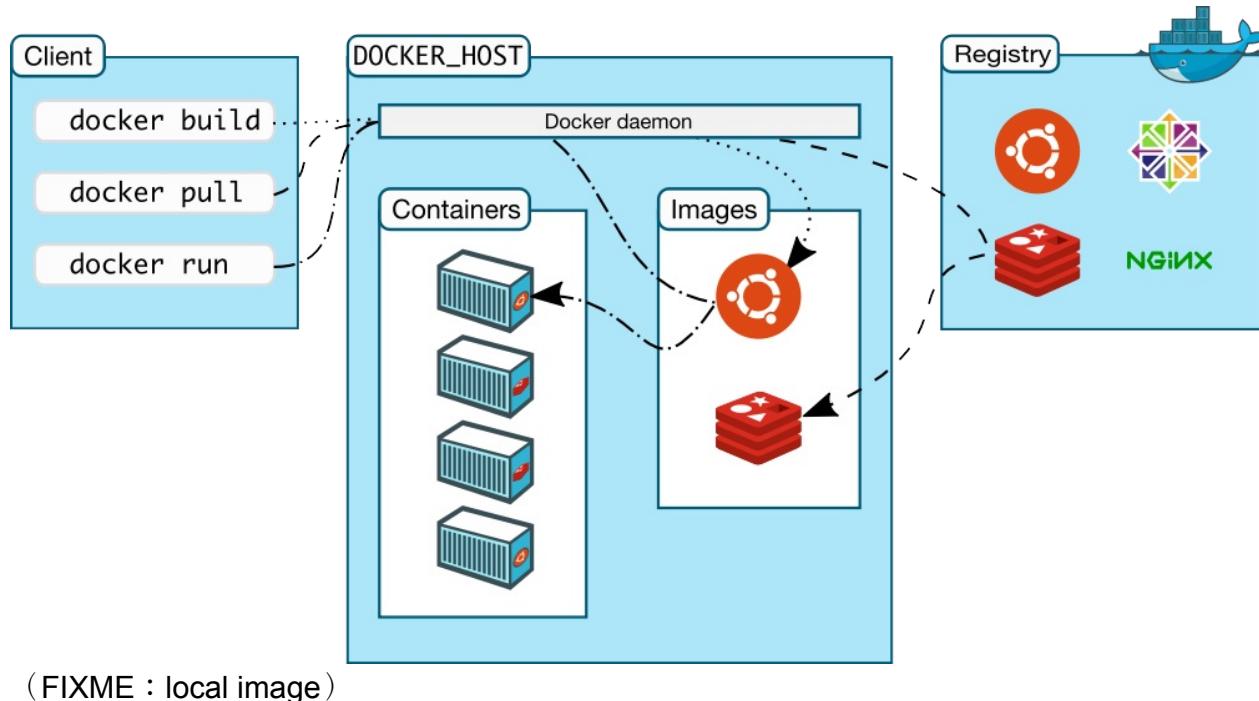
运行以上命令，docker就会自动从灵雀云平台拉取wordpress镜像，这个镜像是已经被build好的，包含了PHP，mysql和wordpress，你所做的工作就是等待docker帮你启动起来以后，在浏览器上访问你服务器的IP就可以看到wordpress的安装页面，然后一步步的点击页面安装即可。对于你的mysql密码

```
echo $(docker logs wordpress | grep password)
```

这个命令就可以获得mysql密码，填写到网页中，这样你就得到了一个可以运行的wordpress，然后开始愉快的使用他吧。

是不是感受到了docker的威力。其实这只是docker强大功能的冰山一角。快速部署是docker其中一个特性。你不需要去登录到服务器，将运行环境一个一个的安装好，最后再部署你自己的代码。docker像集装箱一样，帮助你打包好了一切，你只需要开箱使用即可。就像我们刚才的例子，我们还可以非常简单的再次运行刚才的命令，只需要换一下映射的端口，就可以再启动一个wordpress，这是安装原生应用所不敢想象的。

docker由client，daemon，registry组成。下面图就列出了docker的基本结构。

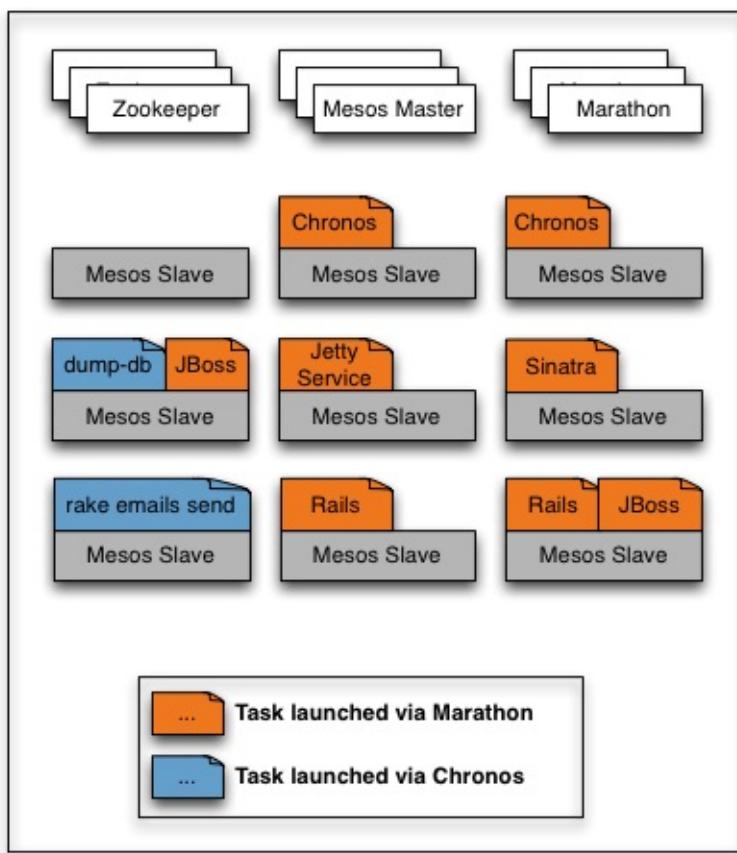


更加细节的docker介绍和讲解请参考docker文档。

Marathon Framework 介绍

前面章节已经介绍了Mesos，我们就不再冗余。对于在Mesos上面长时间运行的服务，Mesos提供了Framework来帮助解决调度，健康监控的功能。

每一个运行在Mesos上的容器，都会运行一段时间，对外或者对其他应用提供服务。当启动一个容器的时候，到底这个容器该被分配到Mesos集群上的哪台机器上，运行过程中如果容器异常退出了，能否报告健康状态或者重启这个容器，有些容器在运行中需要公开端口，但是用户没有显示指定，需要系统来动态的决定分配给一个什么端口。上面的这些功能，marathon framework都帮我们做了，下面我们来介绍一下这个framework。



这张图是Marathon官方用来描述Marathon工作状态的一个例子。由于加入了chronos可能显的有些复杂，我们只关心Marathon的部分。(FIXME: re-draw a graph)

首先看到，由于Marathon是Mesos的一个framework，因此他需要运行在Mesos上。当然，运行Mesos需要依赖zookeeper去做选举和一些数据一致性的问题。可以看到，橘黄色部分的任务都是被Marathon启动的，他们被分配到了一个个Mesos slave上面，然后在运行过程中，如果服务宕掉了，Marathon会去重新启动它，保证你服务在一直运行的过程。用户所需要操作的就是向Marathon rest API 发送创建服务请求，剩下的事情Marathon就会帮你做好。

更加具体的信息请参考[Marathon文档](#).下面我们就动手搭建一个简单的Marathon例子来体验一下。

官方文档里面介绍了安装方式是原生安装，我们将会使用docker安装方式，更加快捷方便。

Mesos容器化安装

在运行Marathon之前，我们需要有一个Mesos环境，作为例子，我们先搭建一个简单的单节点master。Mesos在运行的时候需要master来分配资源和管理slave，而真正干活的则是slave节点。

首先是运行zookeeper，因为Mesos需要使用zookeeper来管理集群。

```
docker run -d -e MYID=1 -e SERVERS=172.31.35.175 --name=zookeeper --net=host --restart=always mesoscloud/zookeeper:3.4.6-ubuntu-14.04
```

我们使用zookeeper 3.4.6版本。`MYID` 为当前zookeeper的ID用来在zookeeper集群中标识，`SERVERS` 为机器IP，`--net` 为网络方式，我们使用host共享宿主机网络方式。`--restart` 是指定当容器异常退出的时候由docker daemon帮助你重启，最后是镜像的名称。

下面来部署Mesos master。

```
docker run -d -e MESOS_HOSTNAME=172.31.35.175 -e MESOS_IP=172.31.35.175 -e MESOS_QUORUM=1 -e MESOS_ZK=zk://172.31.35.175:2181/mesos --name mesos-master --net host --restart always mesoscloud/mesos-master:0.23.0-ubuntu-14.04
```

这个参数比较多，我们来一一解释一下。

- `MESOS_HOSTNAME` 是用来指定当前Mesos master的主机名
- `MESOS_IP` 是当前机器的IP
- `MESOS_QUORUM` 为mesos master的数量，当前为单节点，后面我们会使用高可用模式。
- `MESOS_ZK` 是zookeeper的地址，mesos用来向其中写入数据来保证一致性。

后面的参数前面已经说过了，都是类似的。

Mesos的master有了，那么下面就是干活的slave了。你可以将slave部署到一台新的机器上，也可以部署在和master的同一台机器上，由于我们只是使用一下Marathon的例子，不需要特别的复杂，因此可以将slave部署在一台机器上。

```
docker run -d -e MESOS_HOSTNAME=172.31.35.175 -e MESOS_IP=172.31.35.175 -e MESOS_MASTER=zk://172.31.35.175:2181/mesos -v /sys/fs/cgroup:/sys/fs/cgroup -v /var/run/docker.sock:/var/run/docker.sock --name mesos-slave --net host --privileged --restart always mesoscloud/mesos-slave:0.23.0-ubuntu-14.04
```

slave的参数也不少，前面两个MESOS_HOSTNAME和MESOS_IP是指你部署这个slave所在机器的IP，MESOS_MASTER是指zookeeper所在服务器的地址，mesos通过这个节点去寻找master通信，zookeeper可以保证master的可用性。（FIXME：这里就一个master，zookeeper无能为力，实战最好还是介绍下multiple node的情况）

后面的参数就有些复杂。-v是docker挂载volume的命令，可以将宿主机的磁盘内容挂载到容器内部的指定位置。这里挂载了cgroup和docker.sock。原因是docker需要使用cgroup来实现容器隔离，而docker.sock是docker daemon通信的通道。将这两个目录挂载到容器里面，这样运行在容器里面的mesos slave就可以通过他们来管理宿主机的docker daemon从而实现在宿主机上启动和管理容器。

这里面有一个新的参数，privileged。默认情况下，docker的privileged是关闭的。目的是为了限制容器内部去访问宿主机的设备。比如你想在容器内部运行一个docker daemon默认情况下就是不支持的。如果你设置了privileged，那么容器就能去接触到宿主机的所有设备，这里设置privileged选项是因为mesos-slave需要执行一些特权操作，例如：控制cgroups。

这样一个mesos环境就搭建好了，可以访问一下IP:5050看一下效果。mesos默认开启5050端口提供浏览器访问。

The screenshot shows the Mesos master interface. On the left, there's a sidebar with navigation links: Mesos, Frameworks, Slaves, Offers, and a dropdown menu. Below these are sections for Cluster information (Cluster: (Unnamed), Server: 172.31.35.175:5050, Version: 0.23.0, Built: 3 months ago by root, Started: 23 hours ago, Elected: 23 hours ago), LOG, Slaves (2 activated, 0 deactivated), Tasks (Staged, Started, Finished, Killed, Failed, Lost), and Resources (CPUs: 2, Mem: 992 MB). The main content area has two tables: 'Active Tasks' (empty) and 'Completed Tasks'. The 'Completed Tasks' table lists 10 entries, each with an ID (e.g., php-001.0f2be97d-7d36-11e5-9784-56847afe9799), Name (php-001), State (KILLED), Started (22 hours ago), Stopped (22 hours ago), Host (172.31.35.175), and Hostname (Sandbox).

ID	Name	State	Started	Stopped	Host	Hostname
php-001.0f2be97d-7d36-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.1c795e5f-7d36-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.1c798570-7d36-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.0f2be97d-7d36-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.0f3fe6ae-7d36-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.f67724dc-7d35-11e5-9784-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.de38b3bd-7d33-11e5-8ee0-56847afe9799	php-001	KILLED	22 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.bf6b47c3-7d2c-11e5-b025-56847afe9799	php-001	KILLED	23 hours ago	22 hours ago	172.31.35.175	Sandbox
php-001.e474bed2-7d2b-11e5-b025-56847afe9799	php-001	KILLED	23 hours ago	23 hours ago	172.31.35.175	Sandbox

你的页面应该和这个类似，左侧是mesos集群的一些信息，右侧为目前正在运行的任务和已经运行完毕的任务，如果你没有运行过，那么就不会有记录。

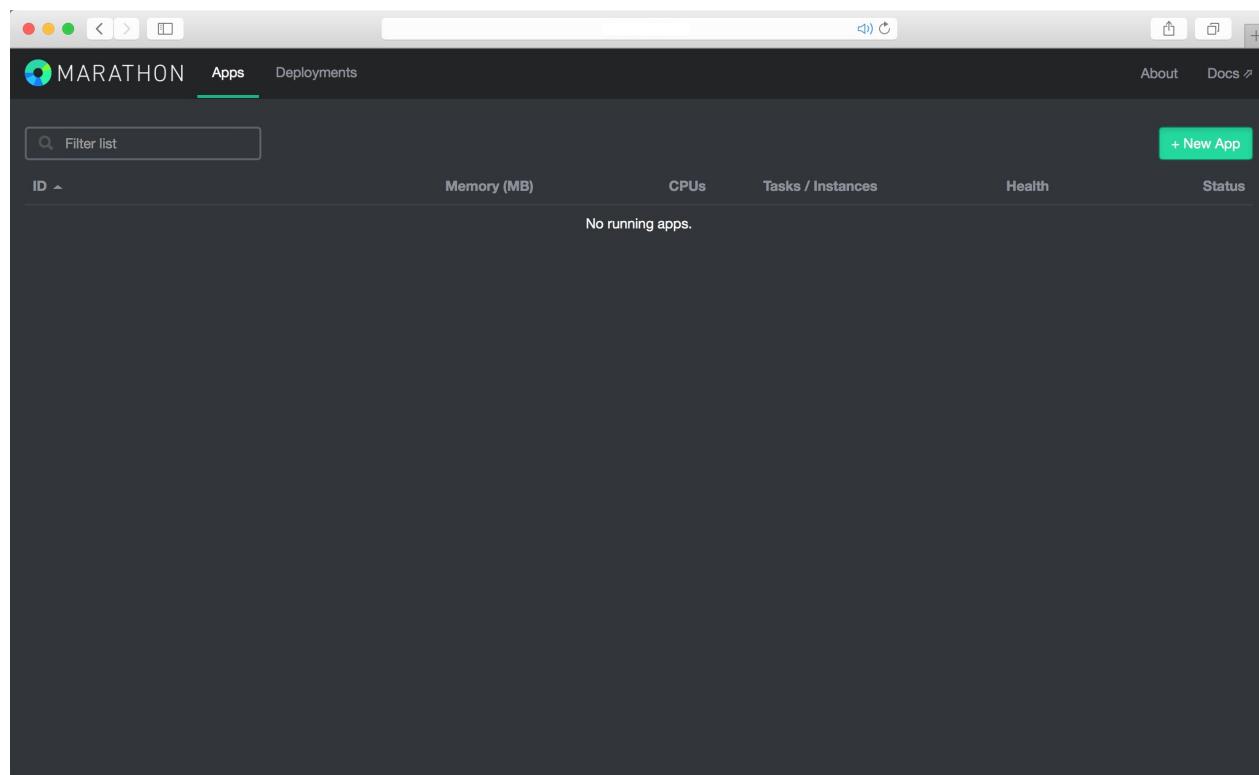
Marathon环境搭建

通过前面的步骤，我们已经有了一个可用的Mesos集群环境，那么下面我们就可以在这个环境下运行Marathon framework了。

```
docker run -d -e MARATHON_HOSTNAME=172.31.35.175 -e MARATHON_HTTPS_ADDRESS=172.31.35.175 -e MARATHON_HTTP_ADDRESS=172.31.35.175 -e MARATHON_MASTER=zk://172.31.35.175:2181/mesos -e MARATHON_ZK=zk://172.31.35.175:2181 marathon --name marathonv0.11.1 --net host --restart always mesosphere/marathon:v0.11.1
```

Marathon的参数也是比较多的。`MARATHON_HOSTNAME` 是部署Marathon本机的IP。`MARATHON_HTTPS_ADDRESS` 是部署Marathon的机器IP。`MARATHON_MASTER` 为mesos所在zookeeper的节点，因为Marathon作为一个framework需要注册到mesos上。`MARATHON_ZK` 为Marathon所在zookeeper的节点。

这样我们就安装好了Marathon环境，使用浏览器请求一下Marathon所在IP：8080看一下效果。



这样Marathon的环境就搭建好了。我们可以使用右上角的New App来创建一个简单的应用。

```
{
  "id": "basic-0",
  "cmd": "while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done",
  "cpus": 0.1,
  "mem": 10.0,
  "instances": 1
}
```

只需要在弹出框里面填写上对应的信息即可。

New Application

ID

basic-0

CPUs	Memory (MB)	Disk Space (MB)	Instances
0.1	10	0	1

Command

```
while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done
```

May be left blank if a container image is supplied

- ▶ Docker container settings
- ▶ Environment variables
- ▶ Optional settings

+ Create **Cancel**

MARATHON Apps Deployments About Docs ↗

Apps > /basic-0

/basic-0 Running

Suspend Scale **Restart App** **Destroy App**

Tasks Configuration Debug

↻ Refresh

ID	Status	Version	Updated
basic-0.218ae1e2-7e3e-11e5-9784-56847afe9799 172.31.23.17:31123	Started	a minute ago	10/29/2015, 9:08:22 PM

在这里你就可以看到运行的效果，这样就代表运行成功，Marathon给这个实例分配了机器资源，他就成功的运行在了mesos集群中。

我们也可以访问5050端口，在mesos控制台信息里面，也可以看到正在运行的任务。

The screenshot shows the Mesos master interface. At the top, there's a navigation bar with tabs: Mesos, Frameworks, Slaves, and Offers. Below the navigation bar, a message says "Master 20151028-042311-2938314668-5050-1".

Active Tasks:

ID	Name	State	Started ▾	Host	
basic-0.9d4b7c45-7e3e-11e5-9784-56847afe9799	basic-0	RUNNING	just now	172.31.23.17	Sandbox

Completed Tasks:

ID	Name	State	Started ▾	Stopped	Host	
basic-0.2b6eed04-7e3e-11e5-9784-56847afe9799	basic-0	KILLED	3 minutes ago	3 minutes ago	172.31.35.175	Sandbox
basic-0.2b6cca23-7e3e-11e5-9784-56847afe9799	basic-0	KILLED	3 minutes ago	3 minutes ago	172.31.23.17	Sandbox
basic-0.218ae1e2-7e3e-11e5-9784-56847afe9799	basic-0	KILLED	4 minutes ago	2 minutes ago	172.31.23.17	Sandbox
php-001.1c7a6fd1-7d36-11e5-9784-56847afe9799	php-001	KILLED	yesterday	yesterday	172.31.35.175	Sandbox
php-001.1c795e5f-7d36-11e5-9784-56847afe9799	php-001	KILLED	yesterday	yesterday	172.31.35.175	Sandbox
php-001.1c798570-7d36-11e5-9784-56847afe9799	php-001	KILLED	yesterday	yesterday	172.31.35.175	Sandbox
php-001.0f2be97d-7d36-11e5-9784-56847afe9799	php-001	KILLED	yesterday	yesterday	172.31.35.175	Sandbox
php-001.0f3fe6ae-7d36-11e5-9784-56847afe9799	php-001	KILLED	yesterday	yesterday	172.31.35.175	Sandbox

容器云平台基础概述

经过前面的搭建，我们已经拥有了一个mesos集群和一个Marathon framework。而对于企业级容器云平台来说，拥有这些还是不足的。

设想一下企业级容器云实际使用情况。我们需要一个高可用的环境。因此需要zookeeper多节点，mesos master 多节点以做备份。然后Marathon 也有ha模式，可以有效的防止master失效导致环境不可用。

实际使用中，我们应该会使用Marathon的rest api来发起部署请求，将我们需要部署的镜像和相关的环境变量交给Marathon，这样的功能一定是集成到我们自己的一个平台里面的。在Marathon部署的时候，我们的容器可能会被分配到一个随机的节点，因此我们需要一种服务发现机制。一般来说，mesos 和Marathon都是部署在内网环境中，一般不允许外部直接访问。我们应该有一个节点专门负责对外访问的负载均衡的功能。Marathon本身内置了服务发现的功能。我们可以通过开启他来让Marathon帮助你自动生成haproxy的配置文件，然后reload来实现服务发现。但是这样可定制性不是很强，而且因为和Marathon绑定，我们不容易迁移和扩展。因此我们这里使用bamboo这个开源项目来做服务发现。bamboo是一个使go开发的能够自动生成haproxy的配置文件然后reload的项目。他的信息来源是通过注册了Marathon的事件订阅机制，Marathon再有变化的时候会自动通知它，然后bamboo就会完成一系列的生成配置文件然后reload haproxy的功能。关于bamboo的详细介绍可以看[这里](#)。

由于bamboo也提供了rest api，因此我们可以很方便的使用代码去完成服务发现，到最终部署的服务可以被外网访问到这一个完整的流程，下面我们就开始搭建。

容器云搭建

本章我们将开始真正的环境搭建。

提前声明，在整个项目中，我们使用四台服务器。

```
Node1 : 172.31.35.175  
Node2 : 172.31.23.17  
Node3 : 172.31.40.200  
Node4 : 172.31.37.173
```

zookeeper 集群

前面章节我们搭建的zookeeper是单点的，这里我们需要搭建一个zookeeper集群，这里我们先搭建一个拥有三个节点的zookeeper集群。

首先在Node1上：

```
docker run -d -e MYID=1 -e SERVERS=172.31.35.175,172.31.23.17,172.31.40.200 --name=zoo  
keeper --net=host --restart=always mesoscloud/zookeeper:3.4.6-ubuntu-14.04
```

其中的参数，`MYID` 为zookeeper集群中的唯一值，用来确定当前节点在集群中的 ID。`SERVERS` 为指定当前集群每个zookeeper节点所在服务器的IP。

然后在Node2上：

```
docker run -d -e MYID=2 -e SERVERS=172.31.35.175,172.31.23.17,172.31.40.200 --name=zoo  
keeper --net=host --restart=always mesoscloud/zookeeper:3.4.6-ubuntu-14.04
```

Node3:

```
docker run -d -e MYID=3 -e SERVERS=172.31.35.175,172.31.23.17,172.31.40.200 --name=zoo  
keeper --net=host --restart=always mesoscloud/zookeeper:3.4.6-ubuntu-14.04
```

启动完毕后，我们进入各个机器的容器查看zookeeper启动情况。

```

root@ip-172-31-23-17:/opt/zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/bin/..../conf/zoo.cfg
Mode: leader

root@ip-172-31-35-175:/opt/zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/bin/..../conf/zoo.cfg
Mode: follower

root@ip-172-31-40-200:/opt/zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /opt/zookeeper/bin/..../conf/zoo.cfg
Mode: follower

```

可以看到，172.31.23.17 为leader，其他的为follower，这样zookeeper集群就搭建完毕了。

Mesos 集群搭建

前面章节我们搭建的mesos集群都是单master节点，生产环境下一定是需要HA的。因此我们这里搭建一个三个master节点的mesos集群。

```

docker run -d -e MESOS_HOSTNAME=172.31.35.175 -e MESOS_IP=172.31.35.175 -e MESOS_QUORUM=2 -e MESOS_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos --name mesos-master --net host --restart always mesoscloud/mesos-master:0.23.0-ubuntu-14.04

```

这是在机器Node1上执行的命令。其中的参数没什么变化，只是zk那里加了三个zookeeper节点。同样的在Node2，和Node3上执行类似的命令。

```

docker run -d -e MESOS_HOSTNAME=172.31.23.17 -e MESOS_IP=172.31.23.17 -e MESOS_QUORUM=2 -e MESOS_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos --name mesos-master --net host --restart always mesoscloud/mesos-master:0.23.0-ubuntu-14.04

docker run -d -e MESOS_HOSTNAME=172.31.40.200 -e MESOS_IP=172.31.40.200 -e MESOS_QUORUM=2 -e MESOS_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos --name mesos-master --net host --restart always mesoscloud/mesos-master:0.23.0-ubuntu-14.04

```

这样我们尝试访问其中一个Node的页面。

可以看到当前访问的节点并不是master节点，说明在这三个节点的选举中，他没有被选举上。mesos会自动帮你跳转到当前master节点所在的服务器。

这样一个具有三个master节点的mesos集群就配置好了，如果其中一个master宕机，另外两个master就会选举出来一个新的master，保证当前集群不会因为没有master而宕掉。

我们在zookeeper的mesos znode上也可以看到选举的结果信息。

```
[zk: 127.0.0.1:2181(CONNECTED) 5] get /mesos/info_0000000003
!20151103-021835-3358072748-5050-1?????
' "master@172.31.40.200:5050*
cZxid = 0x100000016
ctime = Tue Nov 03 02:18:22 UTC 2015
mZxid = 0x100000016
mtime = Tue Nov 03 02:18:22 UTC 2015
pZxid = 0x100000016
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x350cb21c20f0004
dataLength = 94
numChildren = 0
```

我们向其中加入一个slave实验一下。

```
docker run -d -e MESOS_HOSTNAME=172.31.35.175 -e MESOS_IP=172.31.35.175 -e MESOS_MASTER_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos -v /sys/fs/cgroup:/sys/fs/cgroup -v /var/run/docker.sock:/var/run/docker.sock --name mesos-slave --net host --privileged --restart always mesoscloud/mesos-slave:0.23.0-ubuntu-14.04
```

ID	Name	State	Started	Host
No active tasks.				

ID	Name	State	Started	Stopped	Host
No completed tasks.					

可以看到，slave已经注册成功，这样我们就搭建了一个具有三个master节点的mesos集群。

Marathon 搭建

有了Mesos集群，搭建Marathon 就变得非常的简单。Marathon默认支持高可用模式。只要多个运行的Marathon 实例使用同一个zookeeper集群即可，zookeeper来保证Marathon的leader失效时的选举等问题。

```
docker run -d -e MARATHON_HOSTNAME=172.31.35.175 -e MARATHON_HTTPS_ADDRESS=172.31.35.1
75 -e MARATHON_HTTP_ADDRESS=172.31.35.175 -e MARATHON_MASTER=zk://172.31.35.175:2181,1
72.31.23.17:2181,172.31.40.200:2181/mesos -e MARATHON_ZK=zk://172.31.35.175:2181,172.3
1.23.17:2181,172.31.40.200:2181/marathon -e MARATHON_EVENT_SUBSCRIBER=http_callback --
name marathonv0.11.1 --net host --restart always mesosphere/marathon:v0.11.1
```

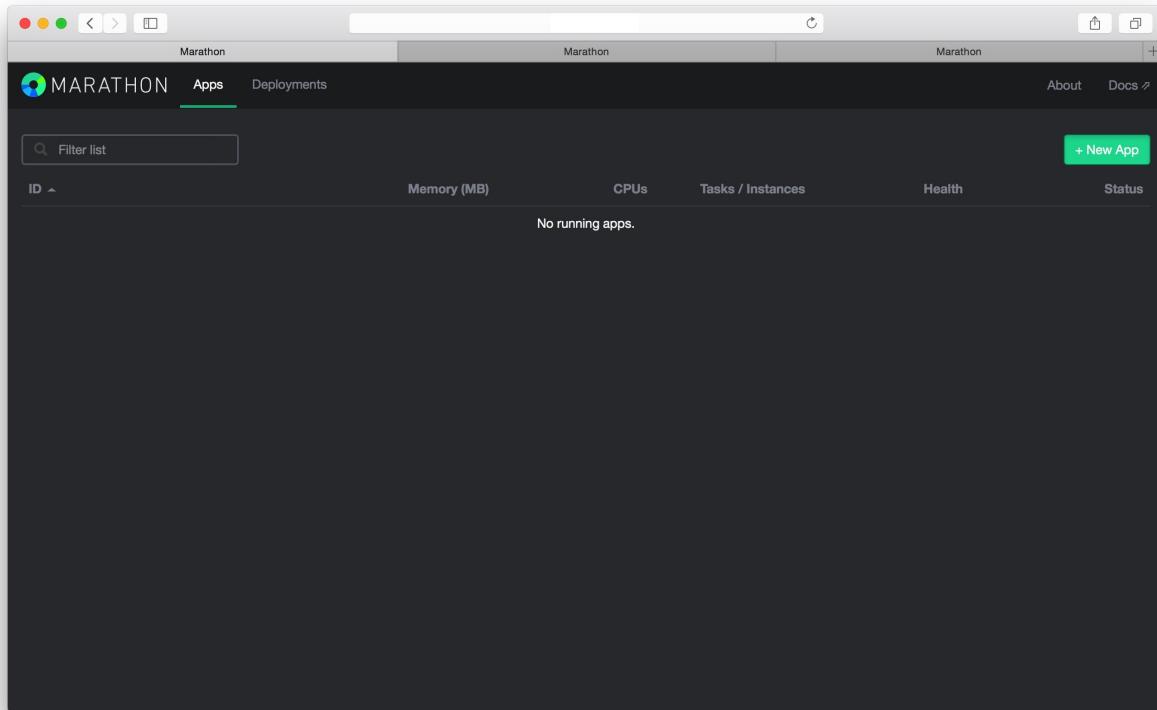
首先我们在Node1上起了一个Marathon实例。这里面的参数前面都讲过，主要的改变就是由原来的一个zookeeper节点变成了一个zookeeper集群。`MARATHON_EVENT_SUBSCRIBER=http_callback` 这里多了一个参数。这个参数是开启Marathon的事件订阅模式，我们使用了 `http_callback`，这样我们可以通过注册一个http回调事件，当Marathon启动，关闭，或者扩容某个实例的时候，我们都可以接收到通知，这为我们下一步做服务发现提供了数据源。

为了防止单点问题，我们启动三个Marathon实例，下面我们分别在Node2和Node3上面再启动两个Marathon实例。

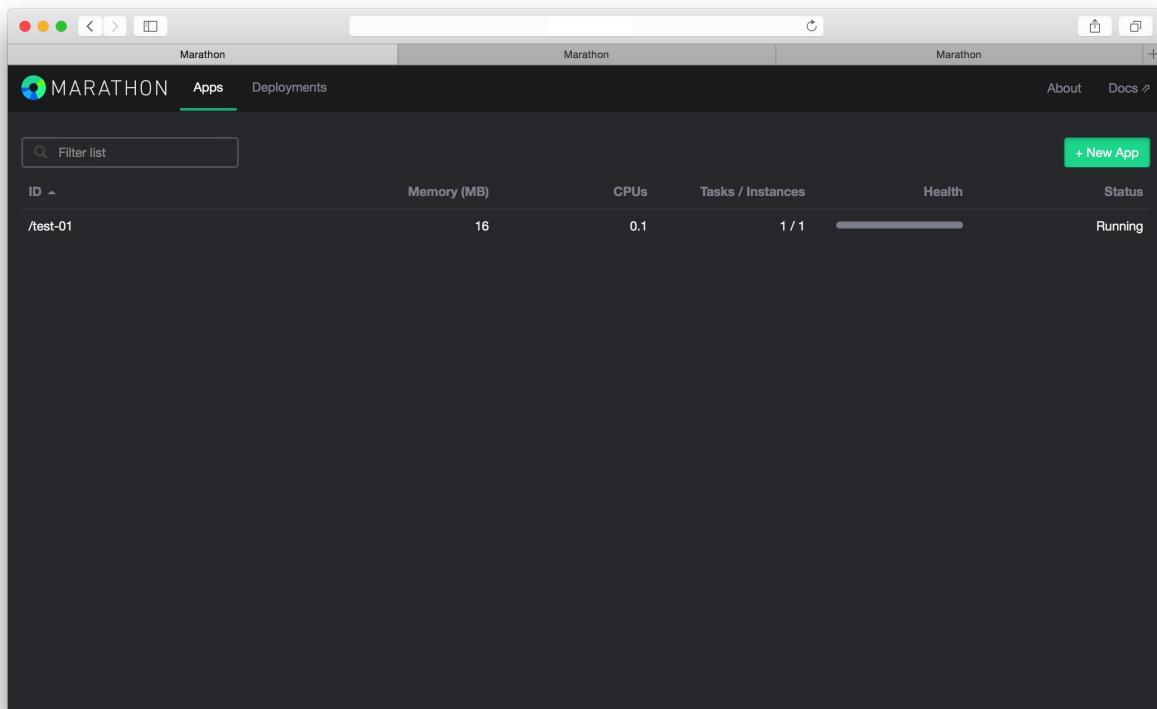
```
docker run -d -e MARATHON_HOSTNAME=172.31.23.17 -e MARATHON_HTTPS_ADDRESS=172.31.23.17
-e MARATHON_HTTP_ADDRESS=172.31.23.17 -e MARATHON_MASTER=zk://172.31.35.175:2181,172.
31.23.17:2181,172.31.40.200:2181/mesos -e MARATHON_ZK=zk://172.31.35.175:2181,172.31.2
3.17:2181,172.31.40.200:2181/marathon -e MARATHON_EVENT_SUBSCRIBER=http_callback --nam
e marathonv0.11.1 --net host --restart always mesosphere/marathon:v0.11.1

docker run -d -e MARATHON_HOSTNAME=172.31.40.200 -e MARATHON_HTTPS_ADDRESS=172.31.40.2
00 -e MARATHON_HTTP_ADDRESS=172.31.40.200 -e MARATHON_MASTER=zk://172.31.35.175:2181,1
72.31.23.17:2181,172.31.40.200:2181/mesos -e MARATHON_ZK=zk://172.31.35.175:2181,172.3
1.23.17:2181,172.31.40.200:2181/marathon -e MARATHON_EVENT_SUBSCRIBER=http_callback --
name marathonv0.11.1 --net host --restart always mesosphere/marathon:v0.11.1
```

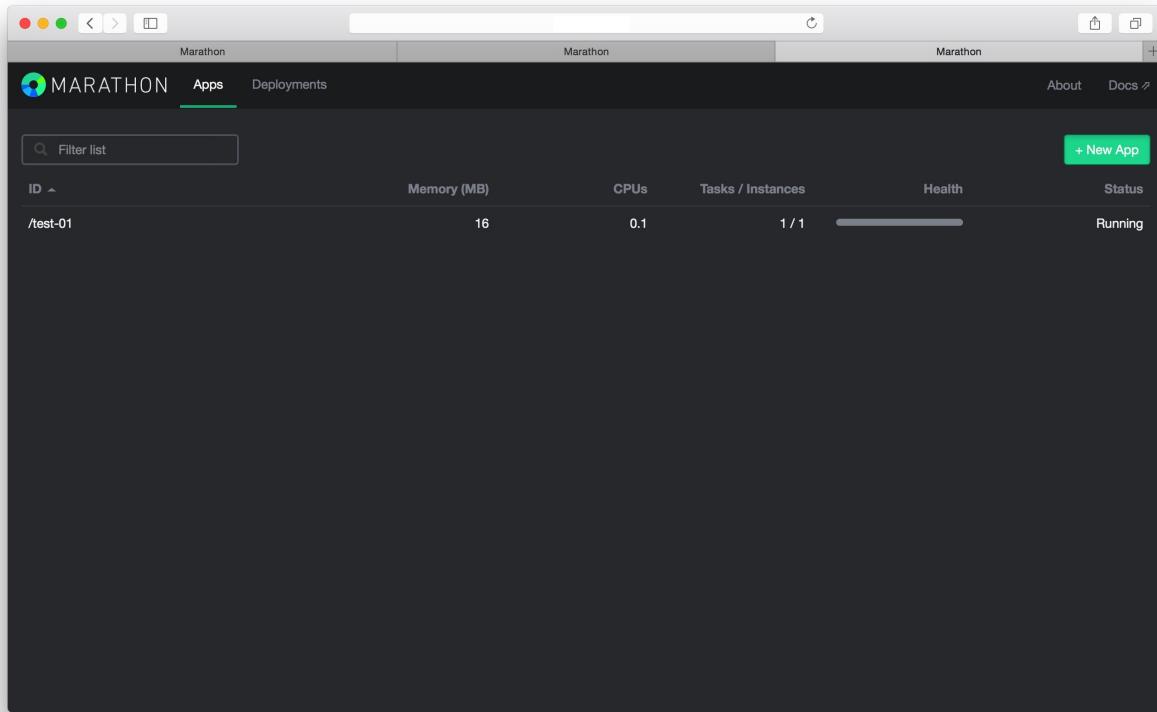
这样我们就启动了含有三个实例的Marathon集群。由于这三个Marathon实例都是共享一个zookeeper集群，因此他们的数据也是同步的。你在其中任何一个节点创建的应用在其他两个Marathon实例上也可以看到。



可以看到，三个tab页面分别打开的的三个服务器上的Marathon页面。我们在请求第一台机器的Marathon创建一个简单的服务。



在第一个tab上我们已经可以看到了这个被创建的服务，现在我们切换到第三个tab看一下。



可以看到，在第三个tab页面上，看到了和第一个tab一样的效果。这就说明了，目前的Marathon集群，他们之间的数据是共享的，在其中任何一个实例上创建服务，其他的实例都可以看到。我们进入zookeeper里面看一下Marathon集群的状态信息。

```
[zk: 127.0.0.1:2181(CONNECTED) 3] ls /marathon/leader
[member_0000000001, member_0000000002, member_0000000000]
```

可以看到leader节点下有三个member，对应着我们启动的三个实例。每个member节点里面记录的信息就是当前这个实例的一些具体的状态。

```
[zk: 127.0.0.1:2181(CONNECTED) 6] get /marathon/leader/member_0000000000
172.31.35.175:8080
cZxid = 0x100000020
ctime = Wed Nov 04 07:11:13 UTC 2015
mZxid = 0x100000020
mtime = Wed Nov 04 07:11:13 UTC 2015
pZxid = 0x100000020
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x350cb21c20f0006
dataLength = 18
numChildren = 0
```

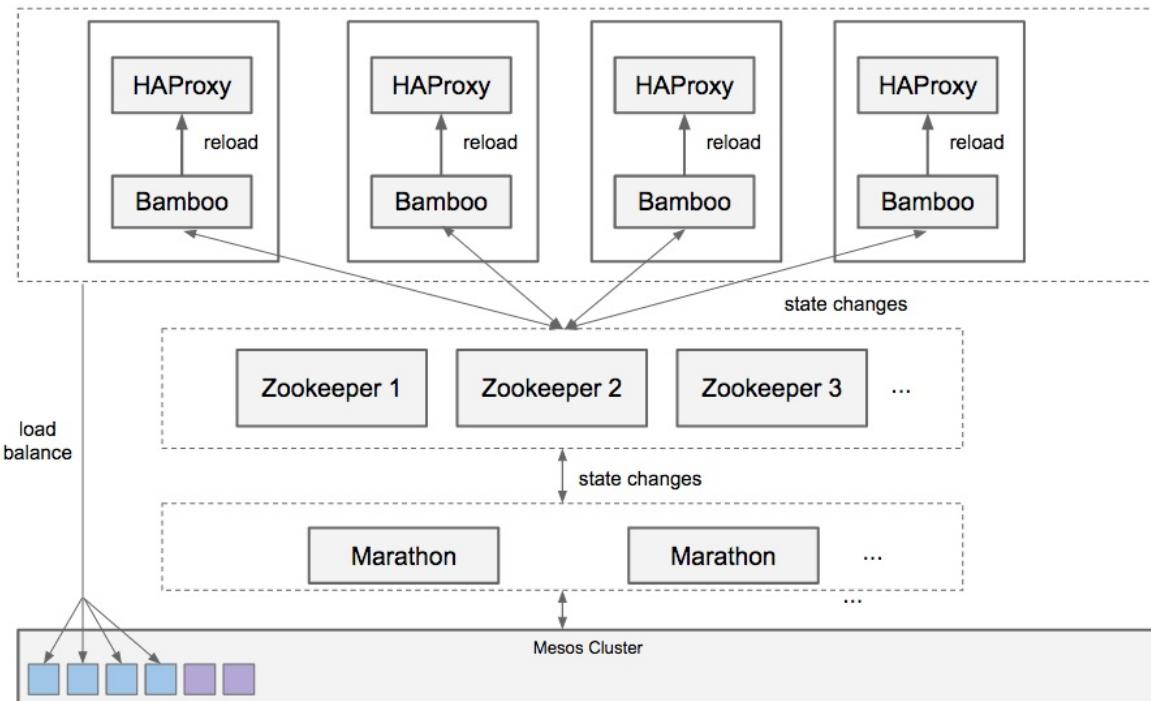
这样我们就搭建起来了一个高可用模式的Marathon集群。

Bamboo 搭建

经过前面的步骤，我们已经有了一个可以运行和管理docker容器的集群环境。我们现在可以通过Marathon启动一个或多个容器，他会被分配到我们集群slave上运行。在运行过程中，Marathon可以保证我们容器运行状态的监控，宕掉重启等活动。但是一般来说，mesos和marathon集群都是搭建在内网或者一个不可以被外部直接访问的网络环境里面，我们需要一个出口，通过这个出口外部可以访问我们容器内提供的服务。而且一般来说，Marathon管理的容器，被分配的slave机器不一定是同一台，在scale和update的时候，会出现IP的变换，这就需要我们有一套服务发现的机制。

Marathon其实提供了服务发现的功能。通过在运行Marathon的机器上跑一个haproxy，Marathon在服务有变动的时候，自动生成haproxy配置文件，然后重启。这种方式对于简单的应用来说应该足够，但是对于需要多租户，自定义ACL规则等功能，这个就显得不太足够。还好Marathon提供了Event_Callback功能，我们可以通过注册事件回调，获取Marathon上运行容器的信息，然后根据某些haproxy模板来自动生成haproxy配置文件，reload后就可以访问。

上面说的这些功能已经被一个名为bamboo的开源项目实现。[地址](#)。他不仅提供了上面提到的服务发现等功能，还可以自定义ACL规则，这样就给我们做服务发现提供了很大的空间。bamboo提供了rest api，我们可以很方便的把它集成到我们自己的项目中。



这是bamboo的部署图。在每个slave上部署一个haproxy加bamboo，然后他们之间可以负载均衡，通过zookeeper同步数据。当Marathon运行的容器有变化的时候，会通过http_call_back通知bamboo，然后bamboo就可以感知变化，我们就可以通过api或者bamboo的页面设置这个容器的acl访问规则，这样就完成了外部访问容器提供的服务的功能。下面我们来搭建Haproxy和bamboo。

首先我们在Node3,Node4上分别启动一个mesos_slave。

```
docker run -d -e MESOS_HOSTNAME=172.31.40.200 -e MESOS_IP=172.31.40.200 -e MESOS_MASTER_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos -v /sys/fs/cgroup:/sys/fs/cgroup -v /var/run/docker.sock:/var/run/docker.sock --name mesos-slave --net host --privileged --restart always mesoscloud/mesos-slave:0.23.0-ubuntu-14.04

docker run -d -e MESOS_HOSTNAME=172.31.37.173 -e MESOS_IP=172.31.37.173 -e MESOS_MASTER_ZK=zk://172.31.35.175:2181,172.31.23.17:2181,172.31.40.200:2181/mesos -v /sys/fs/cgroup:/sys/fs/cgroup -v /var/run/docker.sock:/var/run/docker.sock --name mesos-slave --net host --privileged --restart always mesoscloud/mesos-slave:0.23.0-ubuntu-14.04
```

如果你有更多机器，可以按照上面的命令，更改一下MESOS_HOSTNAME和MESOS_IP就可以非常简单的继续向我们现在的集群增加节点。

The screenshot shows the Mesos UI interface. On the left, there's a sidebar with navigation links: Mesos, Frameworks, Slaves, and Offers. The 'Slaves' section is currently selected. It displays cluster information: Cluster: (Unnamed), Server: 172.31.35.175:5050, Version: 0.23.0, Built: 3 months ago by root, Started: 2 days ago, and Elected: 2 days ago. Below this is a 'LOG' section with a 'Slaves' table showing 2 Activated and 0 Deactivated slaves. Further down are sections for 'Tasks' (Staged, Started, Finished, Killed, Failed, Lost) and 'Resources' (CPUs: Total 2, Used 0; Mem: Total 992 MB, Used 0 B). The main right panel is divided into two tabs: 'Active Tasks' and 'Completed Tasks'. The 'Active Tasks' tab shows a table with columns ID, Name, State, Started, and Host. The 'Completed Tasks' tab shows a table with columns ID, Name, State, Started, Stopped, Host, and a search bar labeled 'Find...'. Both tables show three completed tasks named test-01, each with a KILLED state and specific timestamps.

ID	Name	State	Started	Host
test-01.7bf70600-82c6-11e5-bf41-56847afe9799	test-01	KILLED	20 hours ago	172.31.40.200
test-01.1f9ce72f-82c6-11e5-bf41-56847afe9799	test-01	KILLED	20 hours ago	172.31.40.200
test-01.f4c5745e-82c4-11e5-bf41-56847afe9799	test-01	KILLED	21 hours ago	172.31.40.200

ID	Name	State	Started	Stopped	Host
test-01.7bf70600-82c6-11e5-bf41-56847afe9799	test-01	KILLED	20 hours ago	20 hours ago	172.31.40.200
test-01.1f9ce72f-82c6-11e5-bf41-56847afe9799	test-01	KILLED	20 hours ago	20 hours ago	172.31.40.200
test-01.f4c5745e-82c4-11e5-bf41-56847afe9799	test-01	KILLED	21 hours ago	21 hours ago	172.31.40.200

通过查看mesos页面，我们可以看到目前我们有两个slave提供服务。现在我们向这两个slave部署bamboo。

首先向Node4部署一个bamboo。

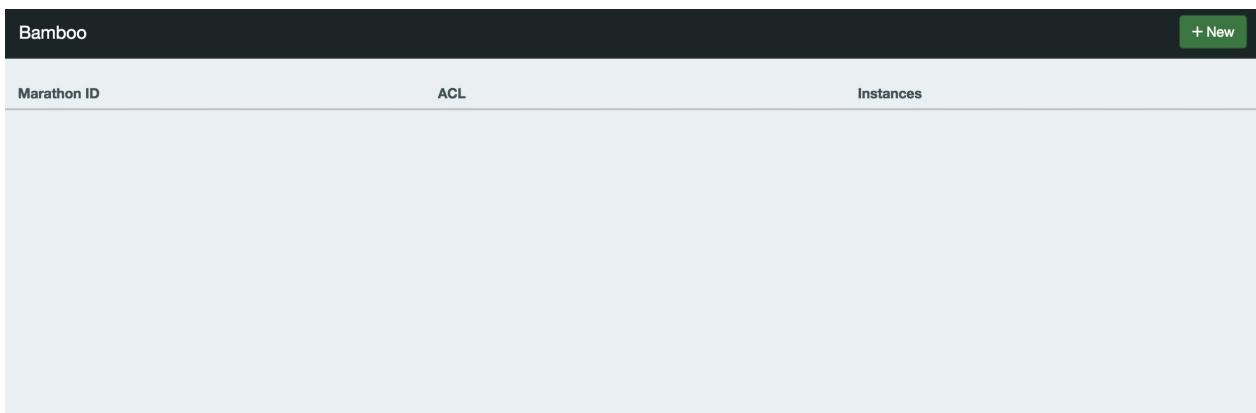
```
docker run -d -p 8000:8000 -p 80:80 -e MARATHON_ENDPOINT=http://172.31.35.175:8080,http://172.31.23.17:8080,http://172.31.40.200:8080 -e BAMBOO_ENDPOINT=http://公网IP:8000 -e BAMBOO_ZK_HOST=172.31.23.17:2181,172.31.40.200:2181,172.31.35.175:2181 -e BAMBOO_ZK_PATH=/bamboo -e BIND=:8000 -e CONFIG_PATH="config/production.example.json" -e BAMBOO_DOCKER_AUTO_HOST=true xianlubird/bamboo
```

这里面的参数，其中8000是bamboo公开的端口，我们可以通过这个端口访问他的控制页面，或者通过这个端口请求他的rest api。80端口是公开给haproxy使用，这个镜像里面内置了haproxy，你不需要自己再安装haproxy。MARATHON_ENDPOINT 是Marathon集群的地址，

bamboo通过这个地址向Marathon注册回调事件通知函数。BAMBOO_ENDPOINT 为bamboo的公开访问的地址，你应该填充你自己的可以被外访问的公网IP地址。BAMBOO_ZK_HOST 为zookeeper集群的地址，bamboo通过这个同步各个节点的数据。BAMBOO_ZK_PATH 为bamboo使用的znode名称。CONFIG_PATH 为bamboo使用的配置文件，虽然已经有一些配置通过环境变量的方式传进去了，但是像haproxy的模板格式，重启haproxy的命令等还是需要配置文件导入的。这里直接使用的官方默认的部署配置文件，你也可以针对这个配置文件按做自己的定制。

```
{  
  "Marathon": {  
    "Endpoint": "http://marathon1:8080,http://marathon2:8080,http://marathon3:8080"  
  },  
  
  "Bamboo": {  
    "Endpoint": "http://haproxy-ip-address:8000",  
    "Zookeeper": {  
      "Host": "zk01.example.com:2181,zk02.example.com:2181",  
      "Path": "/marathon-haproxy/state",  
      "ReportingDelay": 5  
    }  
  },  
  
  "HAProxy": {  
    "TemplatePath": "config/haproxy_template.cfg",  
    "OutputPath": "/etc/haproxy/haproxy.cfg",  
    "ReloadCommand": "haproxy -f /etc/haproxy/haproxy.cfg -p /var/run/haproxy.pid -D -sf $(cat /var/run/haproxy.pid)",  
    "ReloadValidationCommand": "haproxy -c -f "  
  },  
  
  "StatsD": {  
    "Enabled": false,  
    "Host": "localhost:8125",  
    "Prefix": "bamboo-server.development."  
  }  
}
```

部署完毕后，我们可以访问bamboo所在的机器的8000端口。



这里就可以看到bamboo的界面，说明我们部署成功。下面我们向Marathon提交一个容器，我们使用 `tutum/hello-world` 这个image，来测试一下bamboo的效果。

向Marathon提交运行容器，你可以使用Marathon的网页，但是更通用的方式是通过Marathon的restapi来提交，这样更加容易的集成到开发环境中，我们使用Marathon的python库来提交请求。

```
def create_docker_app():
    url = 'http://172.31.23.17:8080'
    c = MarathonClient(url)
    app = MarathonApp(
        id='docker-01',
        cmd='',
        cpus=0.3,
        mem=30,
        container={
            'type': 'DOCKER',
            'docker': {
                'image': 'tutum/hello-world',
                'network': 'BRIDGE',
                'portMappings': [
                    {
                        'containerPort': 80,
                        'hostPort': 0,
                    }
                ]
            }
        }
    )
    c.create_app('hello-001', app)
```

你可以向你Marathon集群的任何一个节点发起部署请求。

ID	Status	Version	Updated
hello-001.09553aa2-8379-11e5-bf41-56847afe9799 172.31.37.173:31162	Started	2 minutes ago	11/5/2015, 12:52:43 PM

可以看到Marathon已经运行起来了这个容器，他被部署到Node4机器上，被分配了一个31162端口。由于这些都是内网IP，我们无法通过外网访问，现在我们再来看一下bamboo的页面。

Marathon ID	ACL	Instances
/hello-001		1 Using default proxy rule +

可以看到bamboo已经检测到了新启动的容器，现在我们通过bamboo给他设置ACL规则。

现在我们访问一下bamboo所在机器的<http://IP/hello>

Hello world!

My hostname is 93ce2b3039b2

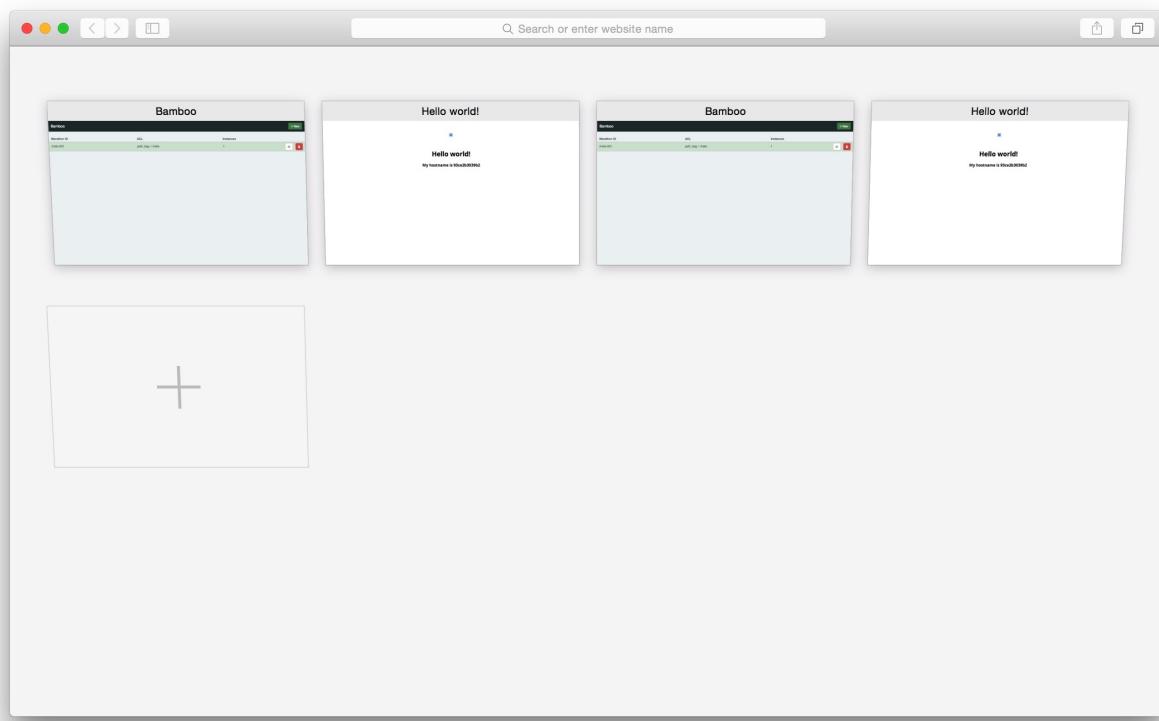
可以看到，我们部署的容器已经可通过haproxy解析访问到了。

Bamboo Ha Mode

下面我们在Node3上面再部署一个Bamboo实例。

```
docker run -d -p 8000:8000 -p 80:80 -e MARATHON_ENDPOINT=http://172.31.35.175:8080,http://172.31.23.17:8080,http://172.31.40.200:8080 -e BAMBOO_ENDPOINT=http://52.32.37.21:8000 -e BAMBOO_ZK_HOST=172.31.23.17:2181,172.31.40.200:2181,172.31.35.175:2181 -e BAMBOO_ZK_PATH=/bamboo -e BIND=:8000" -e CONFIG_PATH="config/production.example.json" -e BAMBOO_DOCKER_AUTO_HOST=true xianlubird/bamboo
```

下面我们分别访问Node3的bamboo和Node4的bamboo，并且同时访问Node3/hello和Node4/hello，看一下效果。



可以看到，无论通过哪一个bamboo实例，我们都可以访问到刚才创建的容器。我们进入bamboo容器内部可以看到他生成的haproxy.cfg

```
# Template Customization
frontend http-in
    bind *:80

    acl ::hello-001-aclrule path_beg -i /hello
    use_backend ::hello-001-cluster if ::hello-001-aclrule

    stats enable
    # CHANGE: Your stats credentials
    stats auth admin:admin
    stats uri /haproxy_stats

backend ::hello-001-cluster
    balance leastconn
    option httpclose
    option forwardfor

    server ::hello-001-172.31.37.173-31162 172.31.37.173:31162
```

下面我们将当前的一个实例scale到三个。

The screenshot shows the Marathon UI for the application /hello-001. The 'Scale' button is highlighted in red, indicating it's being used to increase the number of instances. The table below shows three instances of the application, each with a unique ID and status.

ID	Status	Version	Updated
hello-001.09553aa2-8379-11e5-bf41-56847afe9799 172.31.37.173:31162	Started	26 minutes ago	11/5/2015, 12:52:43 PM
hello-001.bc23e394-837c-11e5-bf41-56847afe9799 172.31.37.173:31195	Started	in a few seconds	11/5/2015, 1:19:06 PM
hello-001.bc1f10193-837c-11e5-bf41-56847afe9799 172.31.40.200:31770	Started	in a few seconds	11/5/2015, 1:19:07 PM

可以看到，他们中，有一个实例被分配到了Node3，另外两个实例在Node4。我们现在什么都不需要操作，继续访问我们刚才访问容器的路径，Node4 IP/hello。可以发现容器是可以正常访问的，而且打开bamboo的控制页面可以看到。

The screenshot shows the Bamboo UI for the application /hello-001. It displays the Marathon ID, ACL, and Instances for the application. The Instances column shows a count of 3, indicating that the application is now scaled to three instances.

Marathon ID	ACL	Instances
/hello-001	path_beg -i /hello	3

bamboo自动检测到了目前实例已经变成了3个，而且帮助我们使用haproxy做了负载均衡。当我们的容器扩容缩容，或者迁移的时候，IP地址发生了变化，我们不需要去关心，我们只需要继续访问我们刚才设置好的路径，bamboo会帮我们做好这些。我们再来看一下现在haproxy的配置文件。

```
#Template Customization
frontend http-in
    bind *:80

    acl ::hello-001-aclrule path_beg -i /hello
    use_backend ::hello-001-cluster if ::hello-001-aclrule


    stats enable
    # CHANGE: Your stats credentials
    stats auth admin:admin
    stats uri /haproxy_stats


backend ::hello-001-cluster
    balance leastconn
    option httpclose
    option forwardfor

    server ::hello-001-172.31.37.173-31162 172.31.37.173:31162
    server ::hello-001-172.31.40.200-31770 172.31.40.200:31770
    server ::hello-001-172.31.37.173-31195 172.31.37.173:31195
```

可以看到，bamboo已经帮我们发现了新创建的服务的ip和端口，并且生成了haproxy的配置文件，并在这三台服务器中做了负载，以后我们不管是scale到0还是scale到10个，bamboo都可以帮助我们自动生成haproxy的配置文件并生效，我们需要做的就是继续访问以前的域名就可以继续使用我们的服务。这样就做到了服务发现和自动的负载均衡。

这样我们就完成了一个基本的高可用的容器云平台的搭建。当然，如果要想把这个流程自动化起来，可能还需要再这个基础上增加一些功能，必须使用他们的restapi 提交请求，而不是使用网页等等。对于数据的持久化，volumn的挂载，我们没有做过多的讨论。后面我们会提及一下实现的思路。

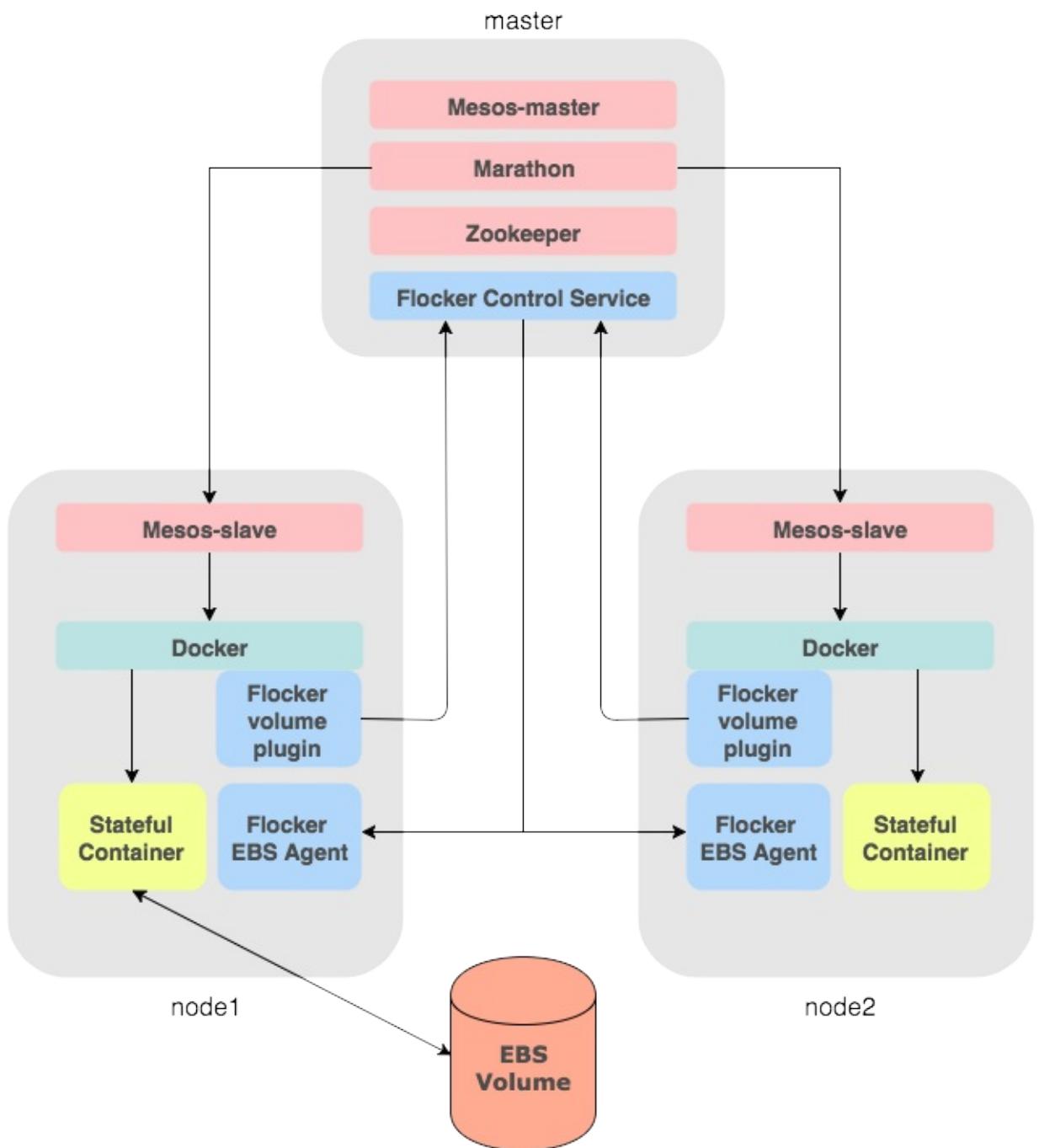
关于容器云平台的其他一些杂谈

前面我们搭建的这个平台，大部分操作都是通过使用框架提供的网页来操作，实际生产环境中，肯定需要我们自己来实现一套自动化的系统来帮助我们完成这些事情。我们可以根据前面章节提供的这些开源组件以及他们的功能来开发自己的容器云平台。

对于数据持久化

前面章节我们搭建的mesos集群和Marathon集群都没提及数据持久化问题。其实docker对于数据持久化做的很方便，只需要使用volume将需要存储的路径挂载到宿主机的磁盘上即可。我们可以根据需要，将mesos的日志文件或者zookeeper的存储文件挂载到自己指定的地方即可。其实我们在运行的时候，这些container在dockerfile中声明了volume，docker会在自己的/var/lib/docker/volume文件夹中存储容器中挂载出来的目录。

这是对于基础平台的数据持久化。但是如果是运行在我们平台上的容器，如果他被scale到别的机器或者被重启后迁移，他挂载在宿主机的磁盘数据如何跟随他自己迁移呢。这里推荐使用flocker。flocker可以将docker的volume在不同主机之间迁移，保证用户的数据不被丢失。这里有一个官方demo的架构图。



当左侧的node1失效的时候，slave宕机，flocker可以自动将存储在EBS volume上的数据同步到node2上，这样就可以实现docker volume的迁移，对于数据库等应用非常的适合。具体的搭建步骤参考[flocker-marathon](#)。

流程的优化

如何将我们前面搭建的架构使用代码串联起来，也是实际搭建容器云平台的一个问题。首先对于这些image的获取，docker pull 官方的image会比较慢，可以通过加速器来加快流程。可以使用[灵雀云](#)提供的镜像加速服务来快速pull镜像，如果有能力也可以自己搭建一个registry，自己管理images。

Marathon的client可以根据自己的语言喜好，有各种版本的client。整个平台创建一个容器的流程大概如下。根据用户选择的image和填写的参数以及环境变量等，使用client向Marathon发送请求，然后Marathon接受请求后开始部署。部署完毕后，bamboo就发现了这个服务的IP和port等信息，然后调用bamboo的rest api，给这个服务添加acl规则，这个时候bamboo就会根据你的规则生成对应的haproxy配置文件，然后haproxy会进行reload来使新的服务生效，这个时候就可以根据IP或者域名访问到部署上去的服务。

如果用户需要扩容，client可以直接向Marathon发送扩容的请求，Marathon就会完成扩容的动作，并且bamboo会自动发现实例数量的变化，reload haproxy，用户这边没有任何感知就可以继续使用原来的域名访问他们的服务，但是这时候后台已经有多个服务在通过haproxy的负载均衡策略提供服务。这样一个基本的容器云服务平台就搭建完成，也可以基本的使用，我们可以很方便的通过增加mesos slave来向集群添加机器，mesos和Marathon会帮我们做好机器之间的选择与部署。

Mesos搭建分布式生物信息算法计算系统实战

Refs: <https://github.com/rabix/rabix>

Mesos搭建视频压缩批处理系统实战

如何使用Chronos来压缩视频

- Mesos 搭建持续集成系统概述
- 持续集成简介
- Jenkins 开源软件介绍
- 持续集成系统搭建
- 持续集成系统的维护心得
- 参考

7.1 summary

软件的开发及部署，早已形成了一套标准流程，其中非常重要的组成部分就是持续集成（Continuous integration，简称CI）。

持续集成(CI) 是一种软件开发实践，其目的是通过频繁的将代码 merge 到主分支来实现产品的快速迭代。在团队合作开发过程中，持续集成不仅可以帮助团队成员及时发现集成错误，而且也避免了个人的分支工作大幅偏离主分支而导致的代码冲突。使用得当，持续集成会极大的提高软件开发效率并保障软件开发质量。

Jenkins 是基于 Java 开发的一种持续集成 (CI) 工具，它提供了一种易于使用的持续集成系统。

Mesos 是 Apache 下的一个开源的统一资源管理与调度平台，它被称为是分布式系统的内核。

Marathon 是注册到 Apache Mesos 上的管理长时应用(long-running applications)的 Framework，如果把 Mesos 比作数据中心 kernel 的话，那么 Marathon 就是 init 或者 upstart 的 daemon。

本章节旨在探讨如何利用 Jenkins，Apache Mesos 和 Marathon 搭建一套弹性的，高可用的持续集成环境。

7.2 持续集成简介

大师 Martin Fowler 对持续集成是这样定义的：持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽快地发现集成错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快的开发内聚的软件。

本章主要结合实践经验介绍持续集成的价值，涉及到的主要工具以及持续集成的大致流程等。

7.2.1 持续集成能为团队带来什么

我们一致认为，持续集成至少可以为团队带来下述几点价值：

- 有效的控制或降低风险

软件一天中进行多次的集成，并触发了相应的测试，这样有利于检查缺陷，了解软件的健康状况，减少假定，从而有效的控制或降低了风险。

- 提高团队效率

减少重复的过程可以节省团队的时间、费用和工作量。在软件开发过程中，浪费时间的重复劳动可能在我们的项目活动的任何一个环节发生，包括代码编译、数据库集成、测试、审查、部署及反馈。通过持续集成可以将这些重复的动作都变成自动化的，无需太多人工干预，让团队成员将时间更多的投入到动脑筋的、更高价值的事情上。

- 提高软件交付速度

持续集成可以让团队在任何时间发布可以部署的软件。从外界来看，这是持续集成最明显的好处，我们可以对改进软件品质和减少风险说起来滔滔不绝，但对于客户来说，可以部署的软件产品是最实际的资产。利用持续集成，团队任何成员都可以经常对源代码进行一些小改动，并将这些改动和其他的代码进行集成。如果出现问题，项目成员马上就会被通知到，问题会第一时间被修复。不采用持续集成的情况下，这些问题有可能到交付前的集成测试的时候才发现，有可能会导致延迟发布产品，而在急于修复这些缺陷的时候又有可能引入新的缺陷，最终可能导致项目失败。

- 提高客户及团队对项目的把控能力

持续集成让客户及团队能够注意到趋势并进行有效的决策。如果没有真实或最新的数据提供支持，项目就会遇到麻烦，每个人都会提出他最好的猜测。通常，项目成员通过手工收集这些信息，增加了负担，也很耗时。持续集成可以带来两点积极效果：

- 有效决策：持续集成系统为项目构建状态和品质指标提供了及时的信息，有些持续集成系统可以报告功能完成度和缺陷率。
- 注意到趋势：由于经常集成，我们可以看到一些趋势，如构建成功或失败、总体品质以及其它的项目信息。
- 建立团队对开发产品的信心

持续集成可以建立开发团队对开发产品的信心，因为他们清楚的知道每一次构建的结果，他们知道他们对软件的改动造成了哪些影响，结果怎么样。

7.2.2 持续集成涉及到的主要工具

- 版本控制/配置管理工具

版本控制与配置管理工具主要负责源代码，配置文件的管理。常见的工具如 git，svn 等，其中 git 是当下最流行的版本控制工具。

- 构建工具

实现主要负责自动化地编译、测试、部署等，这是持续集成的核心工具；构建工具是编程语言依赖型的，不同编程语言使用不同的构建工具。

- CI 服务器

CI 服务器主要负责将版本控制仓库和构建工具有机整合起来，并通过设置一种或多种构建触发条件来触发构建。常见的CI工具有

- jenkins
- travis
- codeship
- stridercd

7.2.3 持续集成的流程

关于CI的流程，首先我们需要明确的是，基于不同的项目规模，不同的项目阶段，不同的团队大小以及不同的团队经验，CI 的流程是相应调整的。CI 流程本身也是一个逐步迭代，逐步完善的过程。

下面列出的是一个基本的，泛化的持续集成过程：

- 团队开发人员频繁的从源代码仓库下载同步代码
- 团队开发编写代码、测试用例，并提交更新结果给版本控制仓库
- CI服务器根据触发条件，从版本控制仓库提取最新代码，交给构建工具的工作空间
- 构建工具对代码进行编译、测试，并进行打包。
- 通过构建工具与版本控制工具的配合，实现产品版本控制与管理
- 建立、管理项目开发的工作网站

另外下面的图片(图7-2-1)是我在[atlassian wiki](#)下载的持续交付成熟度矩阵。这里我们不需要要考慮持续集成与持续交付的差别，或者说，在本文中，持续集成与持续交付是一样的。

Continuous Delivery maturity matrix

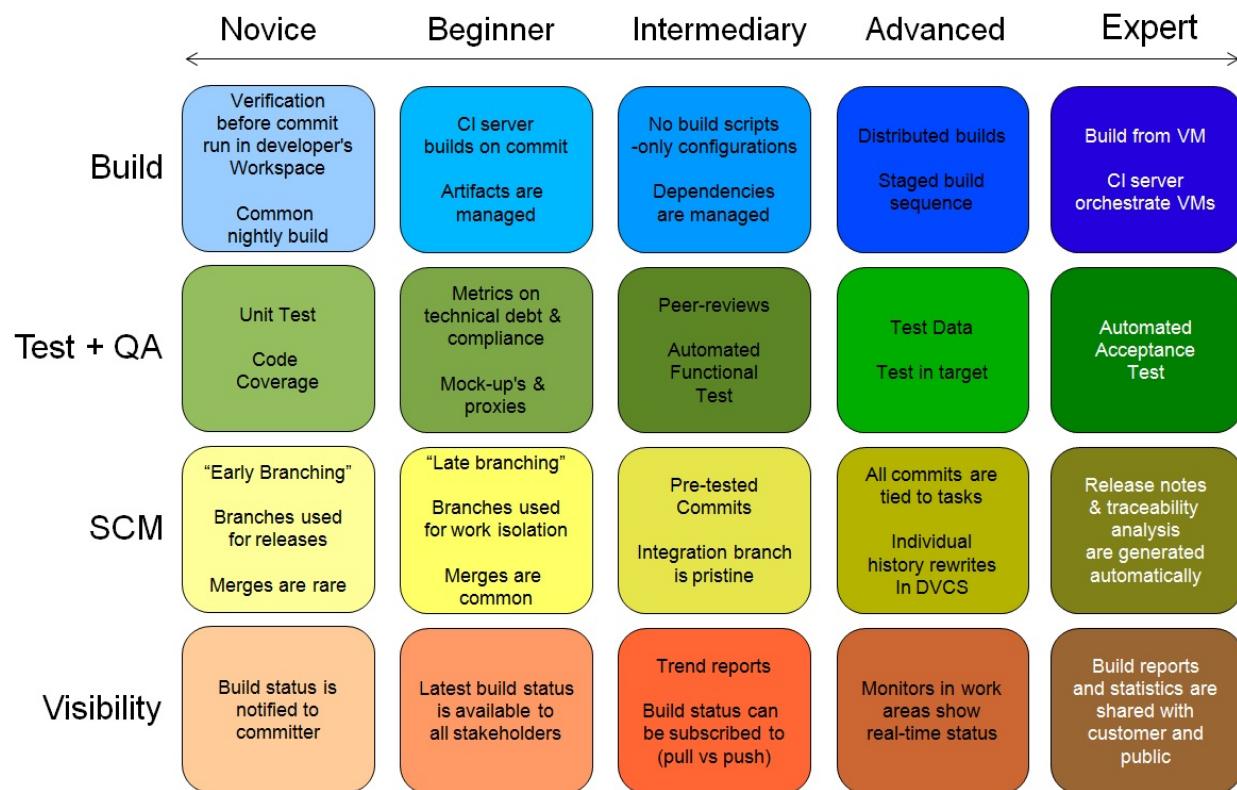


图7-2-1 持续交付成熟度矩阵

通过这张图片我们可以看出，CI流程的演进过程，从初期的 **nightly build** 到最终的 **镜像交付**；从 **merges are rare** 到自动生成 **ReleaseNote**；从简单的将构建结果通知 **committer** 到向客户分享构建报告与统计结果，等等。我们可以发现，一个成熟的继续交付系统可以极大的提高生产率。

7.3 Jenkins

在上一章提到的常用CI工具中，Jenkins 常用的CI开源工具。其官方网站对 Jenkins 是这样定义的：

Jenkins 是一个[获奖无数](#)的，能够提高你产出的，跨平台的持续集成与持续交付工具。通过使用 Jenkins 来不间断的构建，测试你的项目，可以更方便的把开发者的 changes 集成到项目中，更快的向用户交付最新的构建。同时，通过提供接口来定义构建流程，集成多种测试和部署技巧，Jenkins 也可以帮助你持续的交付软件。

Jenkins 以其

- 易于安装：只要把 jenkins.war 部署到 Servlet 容器，不需要数据库支持。
- 易于配置：所有配置都是通过其提供的 Web 界面实现。
- 丰富的插件生态
- 可扩展性：支持扩展插件，你可以开发适合自己团队使用的工具。
- 分布式构建：Jenkins 能够让多台计算机一起构建/测试。
- 集成 RSS/E-mail：通过 RSS 发布构建结果或当构建完成时通过 E-mail 通知。
- 生成 JUnit/TestNG 测试报告。
- 文件识别：Jenkins 能够跟踪哪次构建生成哪些 jar，哪次构建使用哪个版本的 jar 等。

等特性广受开发者欢迎。许多的公司，开源项目都在使用 Jenkins，譬如 [github](#), [yahoo!](#), [Dell](#), [LinkedIn](#), [eBay](#) 等，你在这里可以看到[Who is using Jenkins](#)。

7.3.1 为什么要把 Jenkins 运行到 Apache Mesos 上

使用 Jenkins 时，如果遇到构建作业很多，又需要同一时间进行处理，超过单机计算能力的硬件资源时，单机版本 Jenkins 就无法胜任工作了。在这个情况下，Jenkins 提供了添加独立的 Slave 节点的方式给 Jenkins 任务更多的计算资源来解决这个问题(再进一步 Jenkins 还提供 Docker 化的 Slave 节点)。比如 ebay 之前的模式是每个开发工程师各有一个虚拟机跑自己的 jenkins 来解决持续集成问题，最后导致资源利用率极低。

在动态资源调度，分布式计算盛行的今天，再使用纯静态资源分配的分布式，未免太过时了。还好 Jenkins 提供了很强大的插件功能，可以为分布式提供动态资源调度。

把 Jenkins 运行到 Apache Mesos 上，或者说利用 Apache Mesos 向 Jenkins 提供 slave 资源，最主要的目的就是利用 Mesos 的弹性资源分配来提高资源利用率。

7.3.2 Marathon 运行 Jenkins Master

Jenkins Master 负责提供整个 Jenkins 的设置、webui、工作流控制定制等。另外，Marathon 会对发布到它之上的应用程序进行健康检查，从而在应用程序由于某些原因意外崩溃后自动重启该应用。这样，选择利用 Marathon 管理 Jenkins Master 保证了该构建系统的全局高可用。而且，Jenkins Master 本身也通过 Marathon 部署运行在 Mesos 资源池内，进一步实现了资源共享，提高了资源利用率。

下面这张图形形象的说明了 Marathon 将 Jenkins Master 部署到 Mesos 资源池(图 7-3-1)的过程。

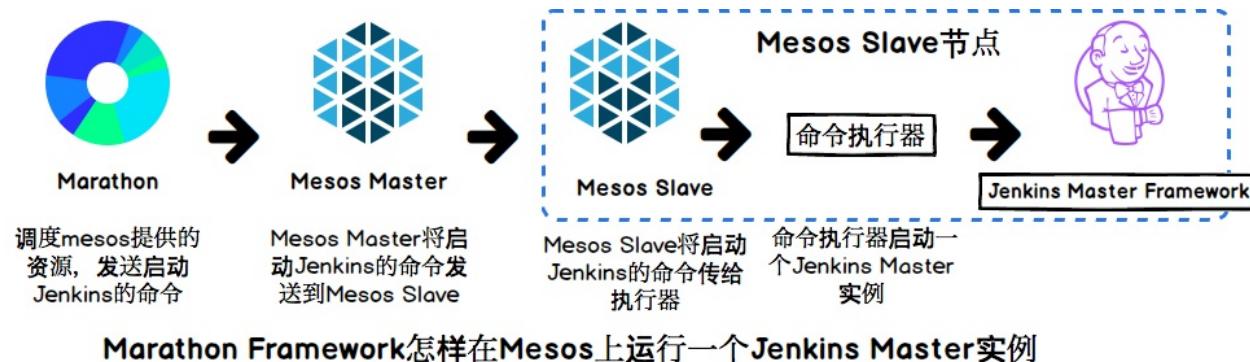


图 7-3-1 Marathon 运行 Jenkins Master

7.3.3 Jenkins 使用 Mesos 资源池

通过配置 Jenkins-mesos-plugin 插件，Jenkins Master 可以在作业构建时根据实际需要动态的向 Mesos 申请 Jenkins-slave 节点，并在构建完成后的一段时间后，将节点归还给 Mesos。下图 Jenkins Master 使用 Mesos 资源池进行作业构建的整个过程(图 7-3-2)。

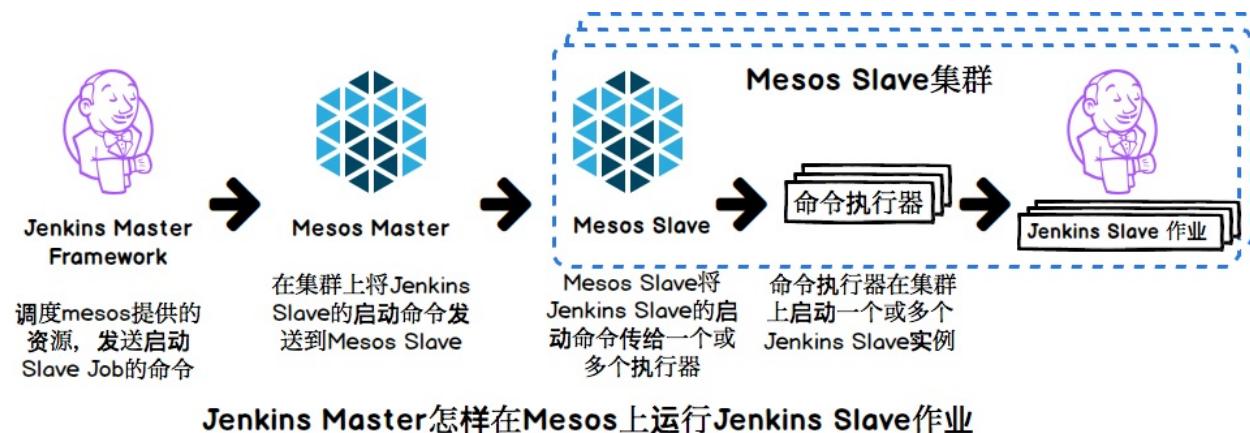


图 7-3-2 Jenkins 使用 Mesos 资源池

7.3.4 Mesos 整体调度流程

图片(图 7-3-3)来源 ebay

How Mesos advertises resource offers and provisions tasks

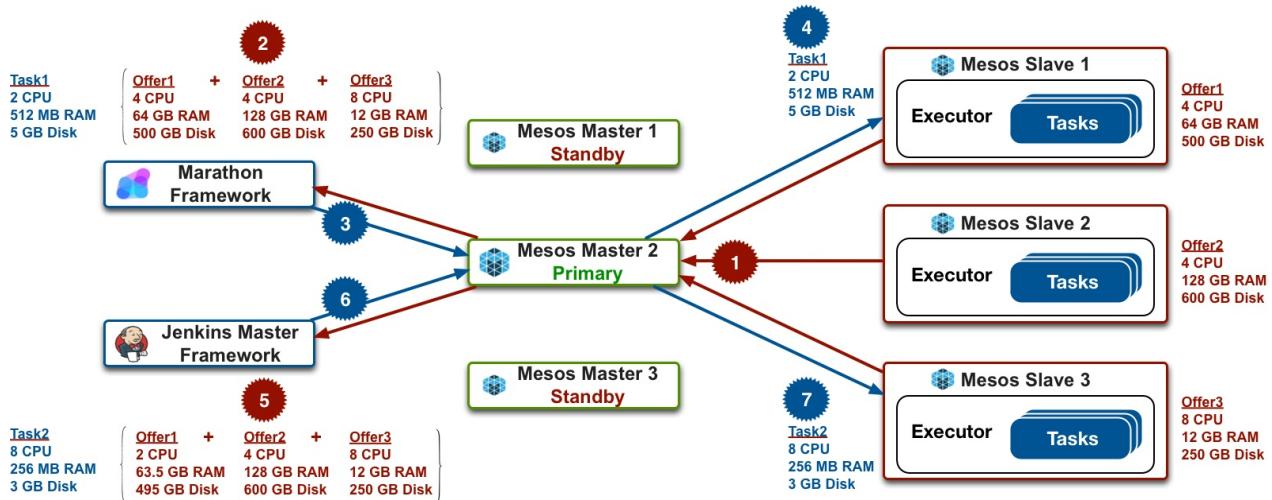


图 7-3-3 Mesos 整体调度流程

7.4 持续集成系统搭建

7.4.1 如何配置保证 Jenkins 的资源弹性，以及 Jenkins Master 的高可用

环境设置

为了便于理解，这里我简化了 Mesos/Marathon 集群的架构，不再考虑集群本身的高可用性。至于如何利用 zookeeper 配置高可用的 mesos/marathon 集群，可以参考[Mesosphere的官方文档](#)，这里不再展开。

我搭建了一个包含40个节点 192.168.3.4-192.168.3.43 的 Mesos 集群，其中一个节点用作运行 Marthon 及 Mesos-master，其它39个节点作为 mesos 的 slave，如下所示。

```
192.168.3.4 marathon/mesos-master
192.168.3.5 mesos-slave
192.168.3.6 mesos-slave
.....
192.168.3.43 mesos-slave
```

参照<http://get.dataman.io>的文档配置启动 Marathon，Mesos-Master 和 Mesos-Slave，下面的整个操作都将在这个集群上完成。

在 Marathon 上部署 Jenkins 的 master 实例

Marathon 支持 web 页面或者 RESTapi 两种方式发布应用，在 192.168.3.* 内网执行下面的 bash 命令，就会通过 Marathon 的 RESTapi 在 mesos slave 上启动一个 Jenkins master 实例。

```
git clone git@github.com:Dataman-Cloud/jenkins-on-mesos.git && cd jenkins-on-mesos &
& curl -v -X POST \
-H 'Accept: application/json' \
-H 'Accept-Encoding: gzip, deflate' \
-H 'Content-Type: application/json; charset=utf-8' \
-H 'User-Agent: HTTPie/0.8.0' \
-d@marathon.json \
http://192.168.3.4:8080/v2/apps
```

这里我在[github](#)上 fork 了 [mesosphere](#) 的 [jenkins-on-mesos](#) 的 repo 到 [DataMan-Cloud/jenkins-on-mesos](#)，并进行了一些改进。

如果Jenkins master实例被成功部署，通过浏览器访问 `http://192.168.3.4:8080` (请确定你的浏览器能够访问内网，譬如可以利用设置浏览器代理等方式来搞定)可以在running tasks列表中找到jenkins，点击进入详细信息页面，我们会看到下图(图7-4-1)：

The screenshot shows the Marathon UI for the Jenkins application. The 'Tasks' tab is selected. Key configuration details include:

- Command:** cd jenkins-on-mesos && export JENKINS_HOME=\$(pwd) && export JENKINS_URL=http://\$MESOS_HOSTNAME:\$PORT0 && java \$JAVA_OPTS -jar jenkins.war --httpPort=\$PORT0
- Constraints:** Unspecified
- Container:** Unspecified
- CPUs:** 1
- Environment:** JAVA_OPTS=-Xmx300m
- Executor:** Unspecified
- Instances:** 1
- Memory (MB):** 1500
- Disk Space (MB):** 0
- Ports:** 10119
- Backoff Factor:** 1.15
- Backoff Seconds:** 1
- Max Launch Delay Secs:** 3600
- URLs:** http://get.dataman.io/packages/jenkins-on-mesos.tgz
- Version:** 2015-05-11T06:54:17.480Z

Annotations provide additional context:

- A callout points to the command field with the text: "Jenkins Master的启动命令, Marathon会将它发送给Mesos Master".
- A brace groups the Environment, Executor, Instances, and Memory fields with the text: "Jenkins Master启动时的基本配置参数".
- A callout points to the URLs field with the text: "Jenkins Master启动之前需要获取的资源的地址".

图7-4-1 Jenkins Master 实例信息

访问 `http://192.168.3.4:5050/#/frameworks` 并在**Active Frameworks**中找到Marathon，点击进入详细信息页面，可以在该页面找到Jenkins Master具体运行到Mesos哪一台Slave上，如下图(图7-4-2)所示：

The screenshot shows the Marathon UI for the Active Tasks page. The Jenkins task is highlighted with a red box. The table shows the following tasks:

ID	Name	State	Started	Host
omega_omega-nginx.ae3dc229-f54c-11e4-99e9-56847afe9799	omega-nginx.omega	RUNNING	7 days ago	192.168.3.16 Sandbox
omega_glance.a7a649f7-f54c-11e4-99e9-56847afe9799	glance.omega	RUNNING	7 days ago	192.168.3.38 Sandbox
omega_glance-worker.109a6a1f-ecad-11e4-99e9-56847afe9799	glance-worker.omega	RUNNING	3 weeks ago	192.168.3.26 Sandbox
omega_glance-worker.09fb50ce-ecad-11e4-99e9-56847afe9799	glance-worker.omega	RUNNING	3 weeks ago	192.168.3.35 Sandbox
omega_glance-worker.095525bc-ecad-11e4-99e9-56847afe9799	glance-worker.omega	RUNNING	3 weeks ago	192.168.3.38 Sandbox
omega_glance-beat.2ae38a62-ecad-11e4-99e9-56847afe9799	glance-beat.omega	RUNNING	3 weeks ago	192.168.3.6 Sandbox
jenkins.9bd159ba-f883-11e4-b2f3-56847afe9799	jenkins	RUNNING	3 days ago	192.168.3.25 Sandbox
htxu_zepplin.0db96887-ea2d-11e4-99e9-56847afe9799	zepplin.hxtu	RUNNING	3 weeks ago	192.168.3.21 Sandbox
fchen_zepplintest.3e159332-ec8e-11e4-99e9-56847afe9799	zepplintest.fchen	RUNNING	3 weeks ago	192.168.3.34 Sandbox
fchen_zepplin.0eb072b9-ea2d-11e4-99e9-56847afe9799	zepplin.fchen	RUNNING	3 weeks ago	192.168.3.41 Sandbox

A red annotation points to the Jenkins task entry with the text: "Jenkins Master运行在Mesos Slave 192.168.3.25上".

图7-4-2 Jenkins Master 运行在 Mesos slave 上

点击sandbox(图7-4-3)

The screenshot shows a Jenkins master interface on a Mesos slave. At the top, there are tabs for 'Mesos', 'Frameworks', 'Slaves', and 'Offers'. The 'Slaves' tab is selected. Below it, a navigation bar shows 'Master / Slave / Browse'. A breadcrumb trail indicates the path: '/ tmp / mesos / slaves / 20150415-022556-33794240-5050-8159-S37 / frameworks / 20150319-231746-33794240-5050-968-0000 / executors / jenkins.9bd159ba-f883-11e4-b2f3-56847afe9799 / runs / 2d2ea469-32c9-4b85-9594-aeb7c5b1106d'. The main content area displays a file listing:

mode	nlink	uid	gid	size	mtime	
drwxrwxr-x	9	core	core	4 KB	May 14 18:48	
-rw-r--r-X	1	core	core	72 MB	May 12 16:49	
-rw-r--r-X	1	core	core	7 KB	May 14 17:43	
-rw-r--r-X	1	core	core	877 B	May 12 16:49	

There are three 'Download' buttons next to the files. A tooltip for the 'jenkins-on-mesos.tgz' file says: '在启动Jenkins Master之前，Mesos Slave首先从网络上获取了jenkins-on-mesos.tgz'.

图 7-4-3 Jenkins Master 运行在 Mesos slave 上

至此，我们的 Jenkins Master 已经在 marathon 上被成功启动了。

配置 Jenkins Master 实现弹性伸缩

接下来是配置 Jenkins 注册成为 Mesos 的 Framework，需要通过浏览器访问 <http://192.168.3.25:31052/> 来到 Jenkins Master 的 UI 页面。下面的截图是我逐步配置的全过程。

The screenshot shows the Jenkins master configuration page. On the left, there's a sidebar with links: '新建', '用户', '任务历史', '系统管理' (with a note '1. 点击“系统管理”'), 'Credentials', and 'Jenkins 100K'. The main area is titled '管理 Jenkins' and contains several configuration items:

- 不安全的Jenkins**: 允许网络上的任何人以你的身份访问程序。考虑至少启用身份验证来阻止滥用。
 - 系统设置**: 全局设置 (2. 点击系统设置)
 - Configure Global Security**: Secure Jenkins; define who is allowed to access/use the system.
- 读取设置**: 放弃当前内存中所有的设置信息并从配置文件中重新读取 仅用于当您手动修改配置文件时重新读取设置。
- 管理插件**: 添加、删除、禁用或启用Jenkins功能扩展插件。 (可用更新)
- 系统信息**: 显示系统环境信息以帮助解决问题。
- System Log**: 系统日志从java.util.logging捕获Jenkins相关的日志信息。
- 负载统计**: 检查您的资源利用情况，看看是否需要更多的计算机来帮助您构建。
- Jenkins CLI**: 从您命令行或脚本访问或管理您的Jenkins。

图 7-4-4 Jenkins Master 配置页面

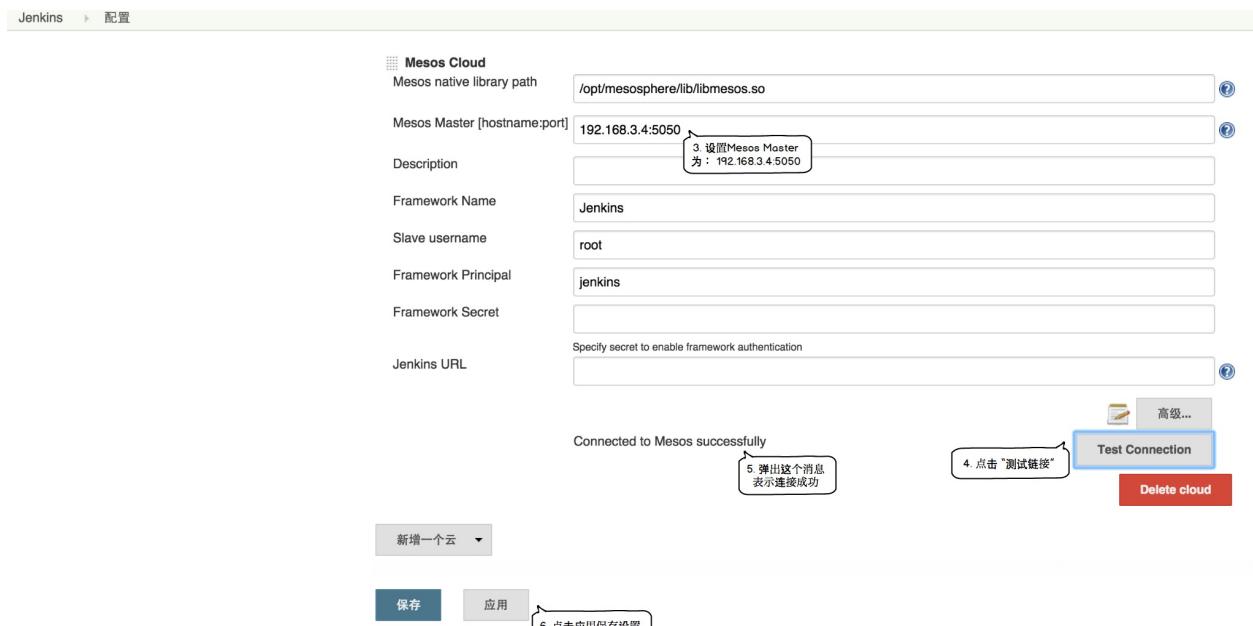


图7-4-5 Jenkins Master 连接 Mesos 集群

1. 来到 Jenkins Master 配置页面
2. 点击系统设置
3. Mesos native library path 设置: Mesos lib 库路径, 一般在 /usr/lib/libmesos.so, 拷贝无效, 必须安装 Mesos
4. Mesos Master [hostname:port] 设置: Mesos-Master 地址加端口, 如果单 Mesos-Master 模式, 使用 mesos-master-ip:5050 格式, 如果是多 Mesos-Master 使用 zk://zk1:2181,zk2:2181,zk3:2181/mesos 格式设置 Mesos Master 为 192.168.3.4:5050
5. Framework Name 设置: Mesos Master 查看到的应用框架名称
6. Slave username 设置: Slave 的名字
7. On-demand framework registration 设置: 是否在无任务的情况下, 从 Mesos-Master 注销应用框架
8. 点击测试链接, 如果链接成功, 页面会弹出连接到 Mesos 成功
9. 点击应用, 将设置保存

如果 Jenkins 在 Mesos 上注册成功, 访问 <http://192.168.3.4:5050/#/frameworks>, 我们可以找到 jenkins Framework, 如下图(图7-4-6)所示:

ID	Host	User	Name	Active Tasks	CPU	Mem	Max Share	Re-Registered	Re-Registered
...5050-26233-0068	i-e0apbw7l	core	Jenkins	0	-2.44249065417534e-15	0 B	0%	a minute ago	-
...5050-26233-0067	i-rtc60w8l	bjoa	Zeppelin	0	-2.08277839419679e-13	0 B	0%	13 hours ago	-
...5050-26233-0066	i-5mrlyqn3	root	chronos-2.3.3	0	4.9293902293357e-14	0 B	6.161737786669625e-14%	19 hours ago	18 hours ago
...5050-26233-0064	i-6w1bc0s0	aezq	Zeppelin	0	-2.14836481937652e-12	0 B	0%	yesterday	-

图 7-4-6 Jenkins framework on mesos

现在我们可以同时启动多个构建作业来看一下 Jenkins 在 Mesos 上的弹性伸缩，在 `http://192.168.3.25:31052/` 上新建一个名为 `test` 的工程，配置其构建过程为运行一个 shell 命令 `top`，如下图(图7-4-7)所示：

The screenshot shows the Jenkins configuration page for the 'test' job. Under the 'Build' section, there is a single 'Execute shell' step. The 'Command' field contains the value 'top'. At the bottom of the page, there are two buttons: '保存' (Save) and '应用' (Apply).

图 7-4-7 配置构建作业

把该工程复制3份 `test2`、`test3` 和 `test4`，并同时启动这4个工程的构建作业，Jenkins Master 会向 Mesos 申请资源，如果资源分配成功，Jenkins Master 就在获得的 slave 节点上进行作业构建，如下图(图7-4-8)所示：



图 7-4-8 构建作业列表

因为在前面的系统配置里我们设置了执行者数量为2（即最多有两个作业同时进行构建），所以在上图中我们看到两个正在进行构建的作业，而另外两个作业在排队等待。

下图(图7-4-9)展示了当前的Jenkins作业构建共使用了0.6CPU和1.4G内存，

该截图展示了Mesos框架下的资源使用情况。上方是Mesos控制台的头部，显示了Mesos、Frameworks、Slaves、Offers以及Omega-Cluster-Prod。下方是“Active Frameworks”表格，列出了正在运行的框架及其资源消耗。

ID	Host	User	Name	Active Tasks	CPU	Mem	Max Share	Registered	Re-Registered
...5050-26233-0082	i-ehq28fe9	root	Jenkins	2	0.600	1.4 GB	1.006%	60 minutes ago	-
...5050-26233-0067	i-rtc60w8l	bjoa	Zeppelin	13	0 B	0 %	19 hours ago	-	
...5050-26233-0066	i-5mrllyqn3	root	chronos-2.3.3	0	4.9293902293357e-14	0 B	6.161737786669625e-14%	yesterday	yesterday
...5050-26233-0064	i-6w1bc0s0	aезq	Zeppelin	0	-2.04969374806296e-12	0 B	0 %	2 days ago	-
...5050-966-0592	i-vcojyphg	mesos	Jenkins Scheduler	0	-2.06443195871486e-12	0 B	0 %	3 days ago	3 days ago
...5050-966-0333	i-rtc60w8l	avxa	Zeppelin	4	4.000	3.5 GB	5.000%	3 days ago	3 days ago
...5050-966-0002	i-6vzgzt62	bnjk	Zeppelin	0	-6.72528699396935e-12	0 B	0 %	3 days ago	3 days ago
...5050-962-0130	i-d76ji268	root	chronos-2.3.2_mesos-0.20.1-SNAPSHOT	1	0.100	128 MB	0.125%	3 days ago	3 days ago
...5050-968-0000	i-d76ji268	root	marathon	117	42.6	58.9 GB	53.25%	3 days ago	3 days ago

图 7-4-9 Jenkins 资源使用

正在使用的slave节点的详细信息

ID	Name	State	Started	Host
mesos-jenkins-97f58c04-c65b-40df-82a1-e2c93b7cf8c8	task mesos-jenkins-97f58c04-c65b-40df-82a1-e2c93b7cf8c8	RUNNING	52 minutes ago	192.168.3.11 Sandbox
mesos-jenkins-40ebb5f2-e157-4f59-a044-fb9bf243a06a	task mesos-jenkins-40ebb5f2-e157-4f59-a044-fb9bf243a06a	RUNNING	6 minutes ago	192.168.3.11 Sandbox

ID	Name	State	Started	Stopped	Host
mesos-jenkins-0235ed48-e07e-45e7-86df-3cb02185d843	task mesos-jenkins-0235ed48-e07e-45e7-86df-3cb02185d843	KILLED	48 minutes ago	43 minutes ago	192.168.3.38 Sandbox

图 7-4-10 slave 节点详细信息

mode	nlink	uid	gid	size	mtime	
drwxr-xr-x	3	root	root	4 KB	May 15 19:26	jenkins
-rw-r--r-x	1	root	root	429 KB	May 15 19:26	slave.jar
-rw-r--r-x	1	root	root	2 KB	May 15 19:26	stderr
-rw-r--r-x	1	root	root	389 B	May 15 19:26	stdout

图 7-4-11 slave 节点详细信息

配置 Jenkins Slave 参数(可选)

在使用 Jenkins 进行项目构建时，我们经常会面临这样一种情形，不同的作业会有不同的资源需求，有些作业需要在配置很高的 slave 机器上运行，但是有些则不需要。为了提高资源利用率，显然，我们需要一种手段来向不同的作业分配不同的资源。通过设置 Jenkins Mesos Cloud 插件的 slave info，我们可以很容易的满足上述要求。具体的配置如下图(图 7-4-12)所示：

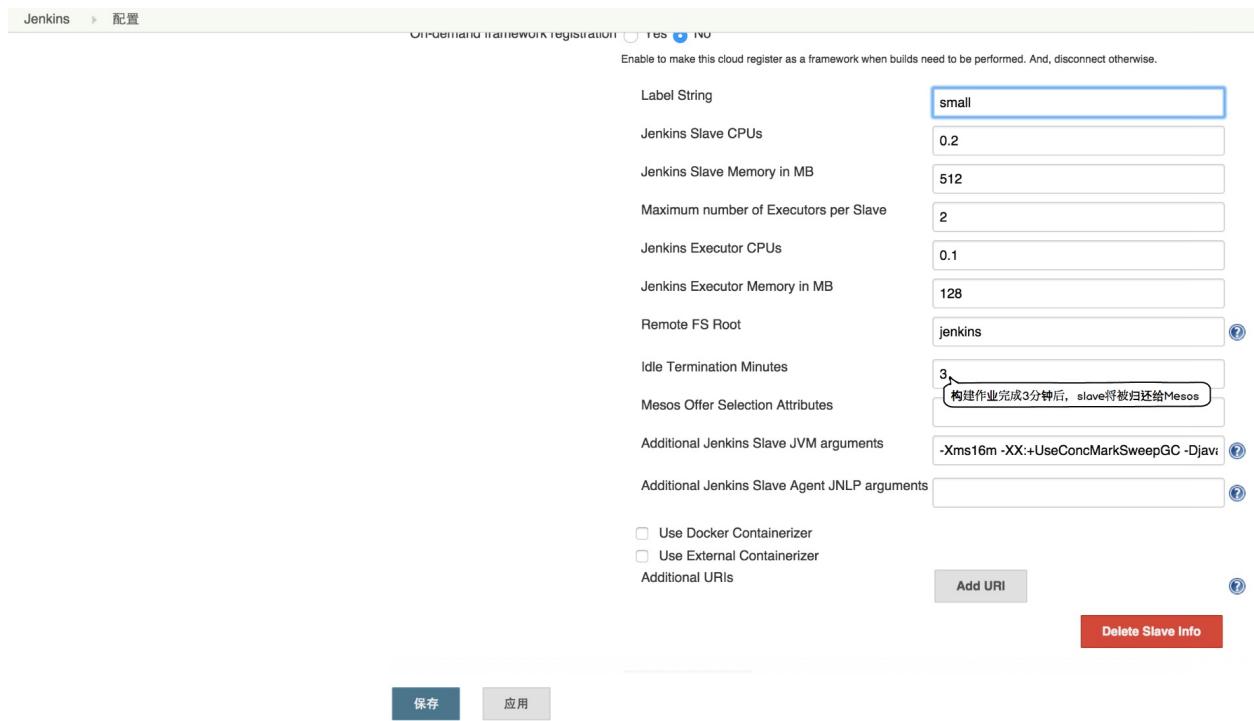


图 7-4-12 Jenkins 配置 slave

1. Label String 设置: Slave 标签
2. Maximum number of Executors per Slave : 每个 Slave 可以同时执行几个任务
3. Mesos Offer Selection Attributes : 选择在哪些 Mesos Slave 标签资源上运行，格式

```
{"clusterType": "标签"}
```

利用mesos为jenkins弹性的提供资源，同时配置Jenkins Slave的参数来满足不同作业的资源需求，这些都大大提高了集群的资源利用率。另外，由于Marathon会自动检查运行在它之上的app的健康状态，并重新发布崩溃掉的应用程序。

7.4.2 如何达到 Jenkins 数据持久化

由于我们通过 Marathon 来启动 Jenkins Master，在 Jenkins Master 异常导致重新部署时，我们需要考虑 Jenkins Master 的数据持久化问题。一种显而易见的方式是限制 Marathon 将 Jenkins Master 部署到同一个数据节点，但这会导致分布式的单点问题。这里我们介绍另一种方法，即：使用 jenkins 插件 [SCM Sync configuration plugin](#) 来将数据同步到 git repo 上。把锅扔出去，扔到 github 上:-)。

在内部的代码库或者 github 上创建一个 git repo

我们需要在内部的代码库或者公共代码库创建一个名为 **jenkins-on-mesos** 的 gitrepo，譬如：<git@gitlab.dataman.io:wtzhou/jenkins-on-mesos.git>。这个 repo 是 jenkins 插件 [SCM Sync configuration plugin](#) 用来同步 jenkins 数据的。

另外，对于 SCM-Sync-Configuration 来说，非常关键的一步是保证其有权限 pull/push 上面我们所创建的 gitrepo。以我们公司的内部环境为例，在 mesos 集群搭建时，我们首先使用 ansible 为所有的 mesos slave 节点添加了用户 **core** 并生成了相同的 **ssh keypair**，同时在内部的gitlab上注册了用户 **core** 并上传其在slave节点上的公钥，然后添加该用户 **core** 为 repo `git@gitlab.dataman.io:wtzhou/jenkins-on-mesos.git` 的 **developer** 或者 **owner**，这样每个 mesos slave 节点都可以以用户 **core** 来 pull/push 这个gitrepo了。

使用 **marathon** 部署可持久化的 Jenkins Master

我们首先需要 wget 两个文件：

```
wget -O start-jenkins.app.sh https://raw.githubusercontent.com/Dataman-Cloud/jenkins-on-mesos/master/start-jenkins.app.sh.template
wget https://raw.githubusercontent.com/Dataman-Cloud/jenkins-on-mesos/master/marathon.json
```

其中 `start-jenkins.app.sh` 是需要配置的，

```
#!/bin/bash

# Sync the config with SCM_SYNC_GIT
# SCM_SYNC_GIT format: git@gitlab.dataman.io:wtzhou/jenkins-on-mesos.git
SCM_SYNC_GIT=

# deploy jenkins on marathon as user APP_USER, who has been granted to pull/push repo
SCM_SYNC_GIT
APP_USER=

# Marathon PORTAL, for example: http://192.168.3.4:8080/v2/apps
MARATHON_PORTAL=
.....
.....
.....
```

编辑如下3个变量：

- SCM_SYNC_GIT**: 上面所配置的 gitrepo 地址, 格式例子：
`git@gitlab.dataman.io:wtzhou/jenkins-on-mesos.git`
- APP_USER**: marathon 会以用户 **APP_USER** 来部署 jenkins ，从而插件**SCM-Sync-Configuration**会以用户**APP_USER**来跟gitrepo进行同步。所以在我们的这个例子里，我们让 `APP_USER=core` 。
- MARATHON_PORTAL**: marathon 的 RESTapi 入口，例如：
<http://marathon.dataman.io:8080/v2/apps>

接下来就可以执行命令：

```
bash start-jenkins.app.sh
```

来让 marathon 部署我们的 Jenkins Master 了。这样，我们在 Jenkins Master 上所保存的任何配置，创建的任何 job 都会被 **SCM-Sync-Configuration** 同步到 repo 里，并在 Jenkins Master 被重新发布后 download 到本地。

关于**SCM-Sync-Configuration**的更多信息

SCM-Sync-Configuration 初始化完成后（在我们环境里初始化过程会被自动触发），每次配置更新或者添加，编辑构建作业时，我们会得到一个提示页面来为新的 commit message 添加 comment，如下图(图7-4-13)所示，

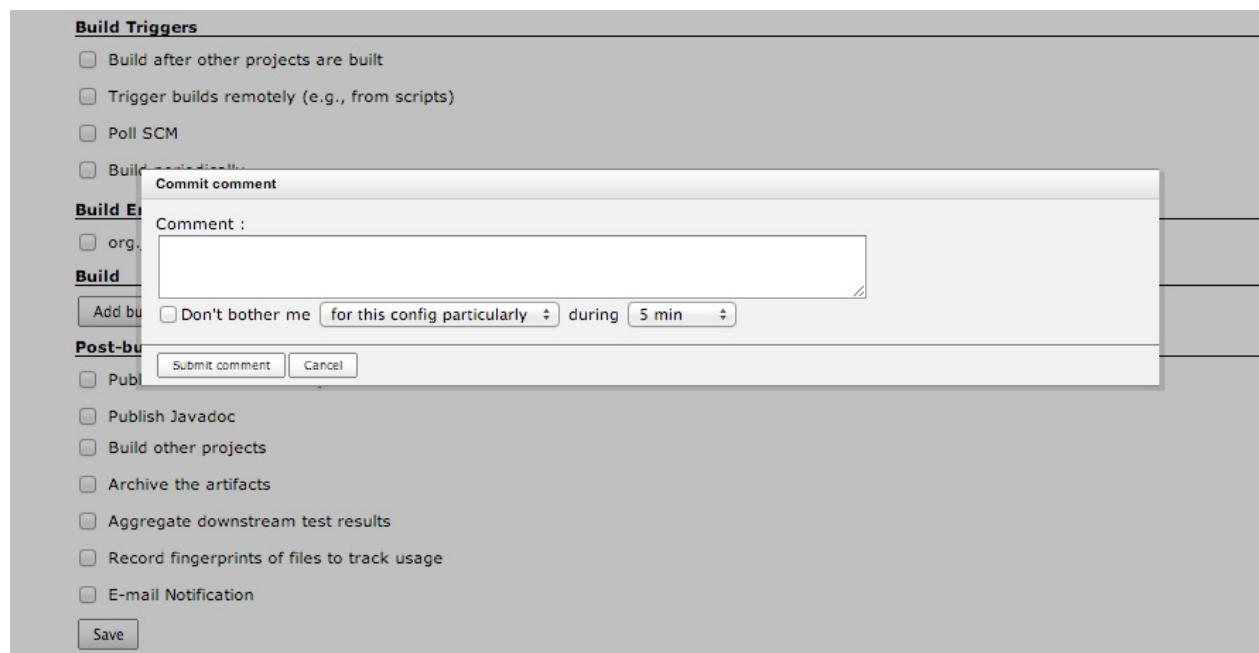


图7-4-13 commit comment

当前，所支持的配置文件如下：

1. 构建作业的配置文件 (/jobs/*/config.xml)
2. 全局的 Jenkins/Hudson 系统配置文件 (/config.xml)
3. 基本的插件的配置文件 (/hudson*.xml, /scm-sync-configuration.xml)
4. 用户手动指定的配置文件

另外，我们可以在每一页的下面看到 scm sync config 的状态，下图(图7-4-14)是同步出错时的截图，你可以去**System Log**查看具体的出错信息。

7.4 持续集成系统搭建

The screenshot shows the Jenkins SCM Sync status page. At the top, there are three checkboxes: 'Aggregate downstream test results', 'Record fingerprints of files to track usage', and 'E-mail Notification'. Below these is a 'Save' button. In the top right corner, there are three small circular icons with question marks. The main content area displays a log of errors from July 18, 2013, at 21:31:57 CEST. The log entries are as follows:

```
Thu Jul 18 21:31:57 CEST 2013 : Checkout /Users/fcamblor/Documents/projects/Jenkins-Plugins/scm-sync-configuration/.work/scm-sync-configuration/checkoutConfiguration
Thu Jul 18 21:31:57 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:31:57 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:31:57 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:32:27 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:33:08 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:33:34 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:35:19 CEST 2013 : Error while checking in file to scm repository
Thu Jul 18 21:38:48 CEST 2013 : Checkout /Users/fcamblor/Documents/projects/Jenkins-Plugins/scm-sync-configuration/.work/scm-sync-configuration/checkoutConfiguration
Thu Jul 18 21:42:06 CEST 2013 : Checkout /Users/fcamblor/Documents/projects/Jenkins-Plugins/scm-sync-configuration/.work/scm-sync-configuration/checkoutConfiguration
```

Below the log, a message says 'To remove this message, please [click here](#)'. The page footer indicates it was generated on July 19, 2013, at 9:34:28 AM, and the Jenkins version is 1.409.

图 7-4-14 scm sysnc status

7.5 持续集成系统维护心得

7.5.1 持续集成的交付方式

这里的交付方式指的是开发者将项目以何种方式交付给运维人员，或者项目以何种方式部署到服务器。常见的交付方式有

- 源代码交付

源代码交付需要将源代码以 tar 包等方式 download 到服务器，然后在服务器上借助程序的构建脚本去构建可执行程序，显然这种方式会经常因服务器环境差异，构建环境初始化失败等问题导致无法构建可执行程序。严重依赖于构建脚本的完备程度。

- 二进制包交付

二进制包交付常见的例子是 Linux 标准包的交付，将项目的依赖通过 Linux deb 或者 rpm 来管理，由于这种方式更符合 Linux 规范，间接的提高了项目在服务器上部署的成功率。

- 虚拟镜像交付

虚拟镜像交付指的是我们将在虚拟机里测试成功后直接将该虚拟镜像部署到服务器上。显然，这种方式部署成功率接近 100%。但是随之而来的问题就是虚拟镜像本身对服务器资源的消耗。

- docker image 交付

docker image 交付是虚拟镜像交付的进一步演进，在保证系统隔离的同时，docker image 对服务器的资源消耗更低。当然，由于 docker 技术仍在发展之中，docker 本身稳定性还有待提高。我们团队目前正在使用这种方式进行交付。

7.5.2 不同环境下的配置管理

对于一个常规的项目来说，开发环境，测试环境与生产环境的配置是不一样的，譬如，这三种环境需要使用不同的域名等。在以 docker 镜像交付时，我们可以通过环境变量来保证配置的差异性。

7.5.3 持续集成可靠性的保证

这里我想阐明的是持续集成本身并不能保证项目的可靠性，持续集成是项目可靠性的一个必要非充分条件。只有为你的项目添加单元测试，集成测试等才能达到更可靠的交付。即，可靠交付的更多人力在提高测试覆盖率。

7.5.4 js 的配置管理问题

js 代码不同于在服务器端运行的代码，JS 在不同环境下的配置管理需要额外的操作，目前已知有两种：

- 使用初始化脚本动态的替换 JS 配置文件中的关键词，即初始化脚本获取环境变量后动态替换配置文件中相应的值。
- 使用 URL 匹配，为不同的环境设置不同的URL，譬如测试环境与开发环境的URL分别为：
 - 测试环境： <http://www.mesos-in-action.com?env=test>
 - 开发环境： <http://www.mesos-in-action.com?env=devel>

从而使用不同的 JS 配置文件。这是 StackOverflow 推荐的一个比较优雅的解决方案。

7.5.5 自动生成 ReleaseNote

在多人协作的项目开发中，由于多人频繁的 merge 代码，ReleaseNote 的管理也会成为团队的负担。我这里推荐的做法是团队达成一个 agreement：

对重大 feature 或 bug-fix 的提交都需要在目录 pending-release 里面创建相应的 markdown 文件，并将改动添加到里面。

这样，我们可以控制 CI 服务器在构建代码时扫描 pending-release 目录并将其中的文本 merge 到一起生成这次构建的 ReleaseNote。

同时，为了避免团队成员忘记这个agreement，我们还可以在本地的 git-precommit-hook 中添加相应的提醒。

7.5.6 频繁发布的版本确认

前期，我们团队在频繁发布项目时遇到的一个尴尬问题是无法确认线上版本是不是最新的版本，或者说无法直观的确认新的change是否部署成功。由于我们的项目是通过html页面与用户交互的，我们通过控制 CI 服务器将每次 release 的 commit id 缀到 html 页面的右下角解决了这个问题。对于非 Web 项目来说，建议的做法是控制 CI 服务器将每次 Relase 的 commit id 推送到一个内部的 Web 服务器，这样，团队就可以非常直观的确认当前版本了。

7.5.7 持续集成的消息通知

显然，团队希望 CI 服务器在执行了持续集成后能够及时的将集成结果通知团队成员，Jenkins 本身是有 `irc notification` 插件的，但是国内开发者可能使用 IRC 的并不多。微信是小团队使用比较多的沟通工具，但是微信不支持机器人调用，退而求其次，我们可以使用 `email` 通知的方式。或者使用国内的 `LessChart` 等交流工具，它们本身支持 `webhook` 调用。

7.6 参考链接

- <http://www.ebaytechblog.com/2014/04/04/delivering-ebays-ci-solution-with-apache-mesos-part-i/#.U2cRpfdVyw>
- <http://www.ebaytechblog.com/2014/05/12/delivering-ebays-ci-solution-with-apache-mesos-part-ii/>
- <https://codeship.com/continuous-integration-essentials>
- <https://blog.risingstack.com/continuous-deployment-of-node-js-applications/>
- <http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>
- <http://blog.csdn.net/lejiantian/article/details/7916483>
- http://baike.baidu.com/link?url=YUuGXCCgfPK9Z_Yf728mCkL7ccEAxEVGVOOrQETCTP_w7RN3qtqS7Hd4ZV6XibkC5YxuFw1HfLW90dL8B0KKa