Marathon 说明文档

Marathon 说明文档

中移苏研-邹能人

一、Marathon 介绍

Marathon 是一个 mesos 框架，能够支持运行长服务，比如 web 应用等。是集群的分布式 Init.d，能够原样运行任何 Linux 二进制发布版本，如 Tomcat Play 等等，可以集群的多进程管理，实现服务的发现，为部署提供提供 REST API 服务,SSL 与基础认证、配置约束,通过 HAProxy、DNS 实现服务发现和负载平衡，可定制化监控策略实现 Task(一个 App 对应多个 Task)的自动扩缩，Dcos 架构与操作系统架构对比如图 1.1 所示。

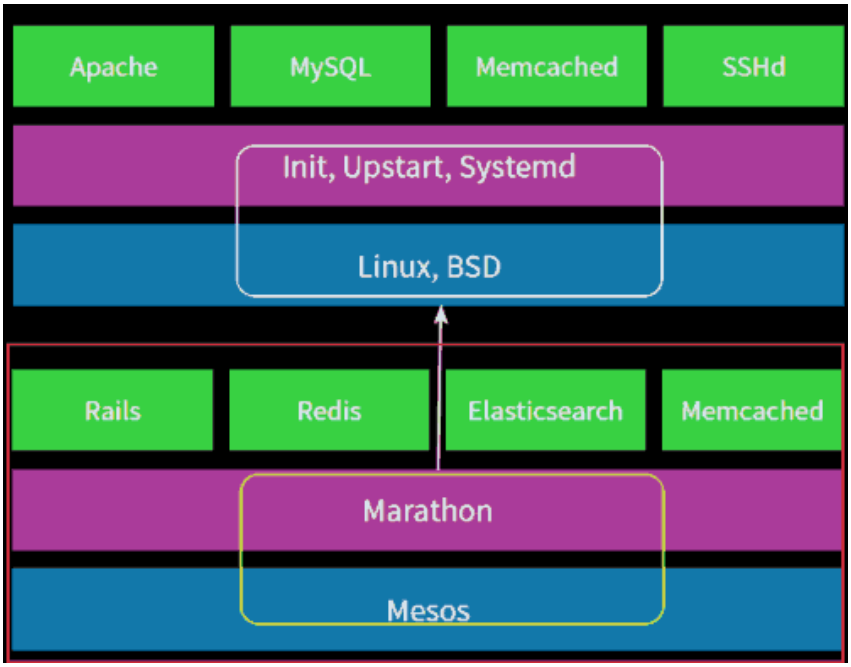

图 1.1 Dcos 架构与操作系统架构

Mesos 仅仅是适用于集群的管理，这意味着它可以隔离不同的任务负载。但是仍然需要额外的工具来帮助工程师查看不同系统上运行的工作负载。不然的话，如果某些工作负载消耗了所有资源，那么重要的工作负载可能就难以及时地获得资源。

Marathon 是一个“元架构”,它可以让 Mesos 和 Chronos 变得更好用，随着 Mesos 一起运行，并且在运行工作负载的同时提供了更高的可用性，让用户可以添加资源以及自动的故障转移,如图 1.2 所示。

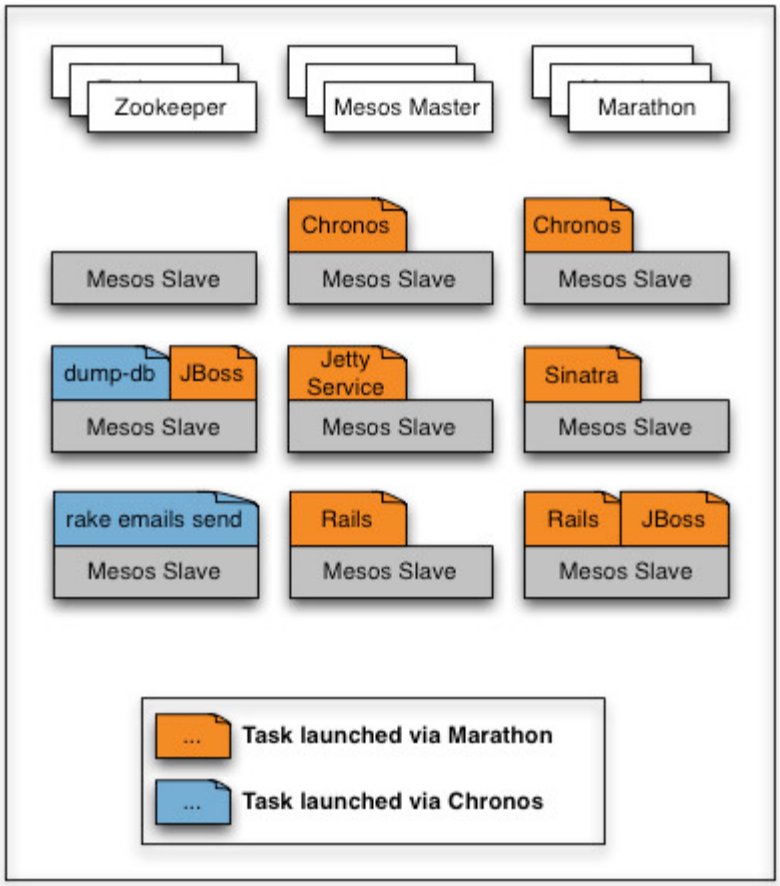

图 1.2 Chronos 与 marathon

不像 Chronos 在 Mesos 之上调度作业,Marathon 让 Chronos 在 Mesos 的内部进行运行，通过这种方式，Chronos 也变成 Marathon 所管理的一项工作。Chronos 的优势在于处理和调度 Hadoop 作业和其他短期的任务,而 Marathon 则可以直接管理 Chronos 和那些长期运行的 Web 服务。Marathon 甚至可以运行多个实例。

图 1.3 集群运行三个不同的应用

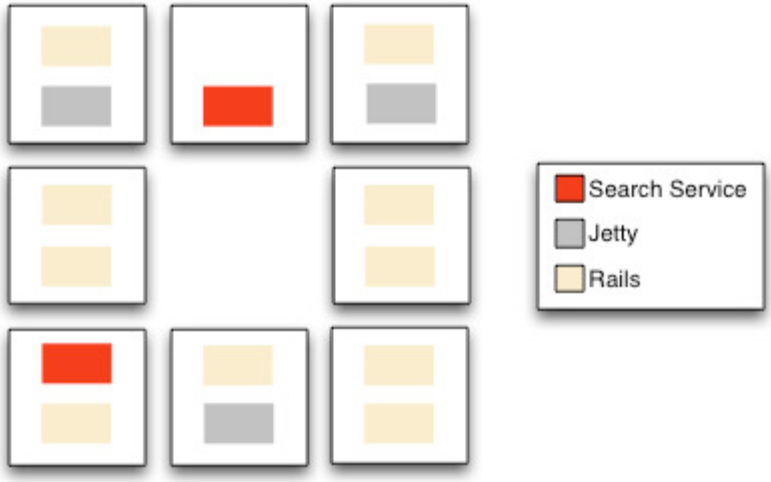

图 1.4 某一节点失联

从更广的层面而言，像 Marathon 这样的项目将来可能会在 SDN（软件定义的网络）、存储，甚至是 DCOS（数据中心操作系统）有更大的作为。

二、Marathon 之安装指引篇

安装需求:
Apache Mesos 0.24.0+
Apache ZooKeeper
JDK 1.6+

测试环境

节点	角色
10.133.19.25	docker 私有镜像库
10.134.29.141	dcos 集群部 署 workstation
10.134.29.136	mesos-master、zookeeper
10.134.29.135	mesos-master、zookeeper
10.134.29.134	mesos-master、zookeeper、UI
10.134.29.133	mesos-slave
10.134.29.132	mesos-slave
10.134.29.144	mesos-slave
10.134.29.129	mesos-slave

安装过程

① 安装 Mesos
简单的方法是通过系统的包管理来安装，当然也可以使用源码安装 mesos,具体可以参考 Mesos 官网。

② 安装 Marathon
通过 Package Manager,Marathon 可以从软件仓库中下载。
$ curl -O
http://downloads.mesosphere.com/marathon/v0.15.2/marathon-0.15.2.tgz
$ tar xzf marathon-0.15.2.tgz

③ 生产环境下运行

生产环境下启动 Marathon,需要 ZooKeeper 和 Mesos 同时运行,下面的命令将会在生产环境中启动 Marathon,将你的 web 浏览器的 localhost 设为 8080,将会看到 Marathon 的界面。

```
$ ./bin/start --master zk://10.134.29.134:2181
10.134.29.135:2181,10.134.29.136:2181/mesos --zk
zk://10.134.29.134:2181 , 10.134.29.135:2181,10.134.29.136:2181/marathon
```

Marathon 使用--master 参数去发现 Mesos 的 master 节点,使用--zk 发现 ZooKeeper,因为两者之间相对独立,所以 Mesos master 节点也使用其他方式来发现，比如 etcd 等。

对于 marathon 所有的参数设置选项,请查看 marathon 之命令行篇。对于 Marathon 更多的高可用特性,请查阅 marathon 之高可用篇。

④ Mesos Library

MESOS_NATIVE_JAVA_LIBRARY:通过 bin/start 可以找到通常的安装路径, /usr/lib 和/usr/local/lib,如果你为链接库设置了其他路径,MESOS_NATIVE_JAVA_LIBRAR 需要设置为其它环境变量,设置如下。

```
$MESOS_NATIVE_JAVA_LIBRARY=/Users/bob/libmesos.dylib ./bin/start

--maste zk://10.134.29.134:2181
10.134.29.135:2181,10.134.29.136:2181/mesos
--zk zk://10.134.29.134:2181 ,
10.134.29.135:2181,10.134.29.136:2181/marathon
```

启动应用

Marathon 应用的介绍以及如何执行,请查阅 Marathon 之应用篇。

三、Marathon 之应用篇

3.1 基础应用

将下述代码，保存为 shell.json:

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

执行下述命令行:

```
curl http://10.134.29.134/v2/apps -d @shell.json -H "Content-type: application/json"
```

其将 json 文件通过 marathon api /v2/apps 传递给 marathon 调度器，请求创建一个实例，资源需求为 0.1cpu、10M 内存，执行 shell 命令行。cmd 将被发送给 Mesos 底层执行器进行执行，通过/bin/bash -c ${cmd}。

打开 dcos 控制台，跳转到 marathon 管理界面，如图 3-1 所示，可以看到名为 shell 的 App 正在运行,App 在 marathon 框架中被定义为长服务，一对多的关系，即为一个 App 可以有多个 task。

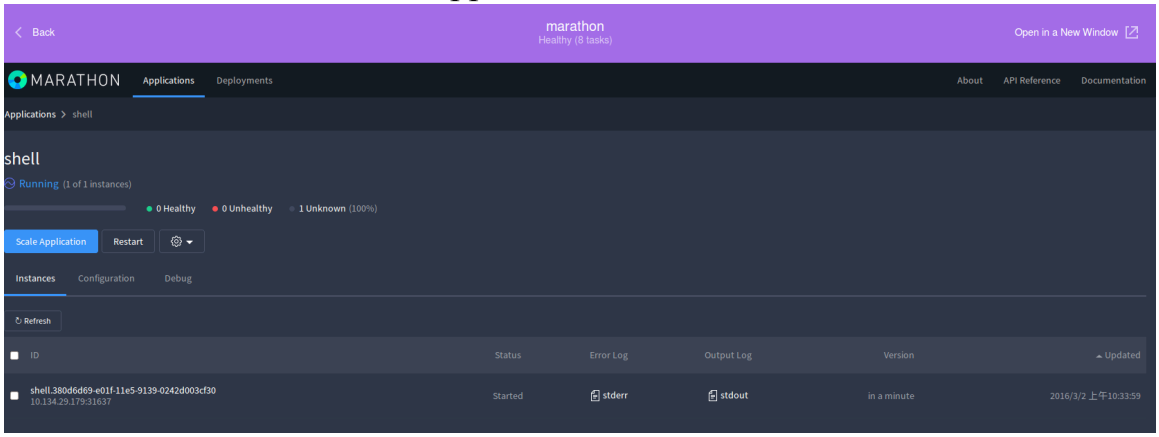

图 3-1

对图 3-1，可以发现 marathon 提供 app 的自动扩展，健康检查以及运行情况的健康，日志管理等功能。

marathon 对 app 的生命周期进行有效的管理,首先介绍其自动扩缩功能，选择 Scale Application 进行快速扩容，在如图 3-2 中填上 10，即可扩展到 10 个实例。

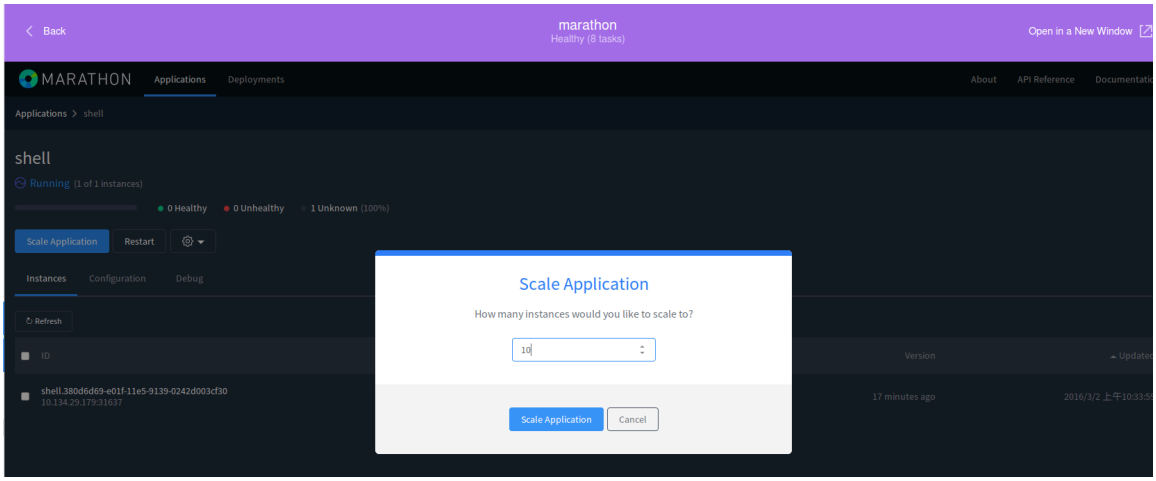

图 3-2

如下图 3-3 所示，shell app 扩展到 10 个实例。

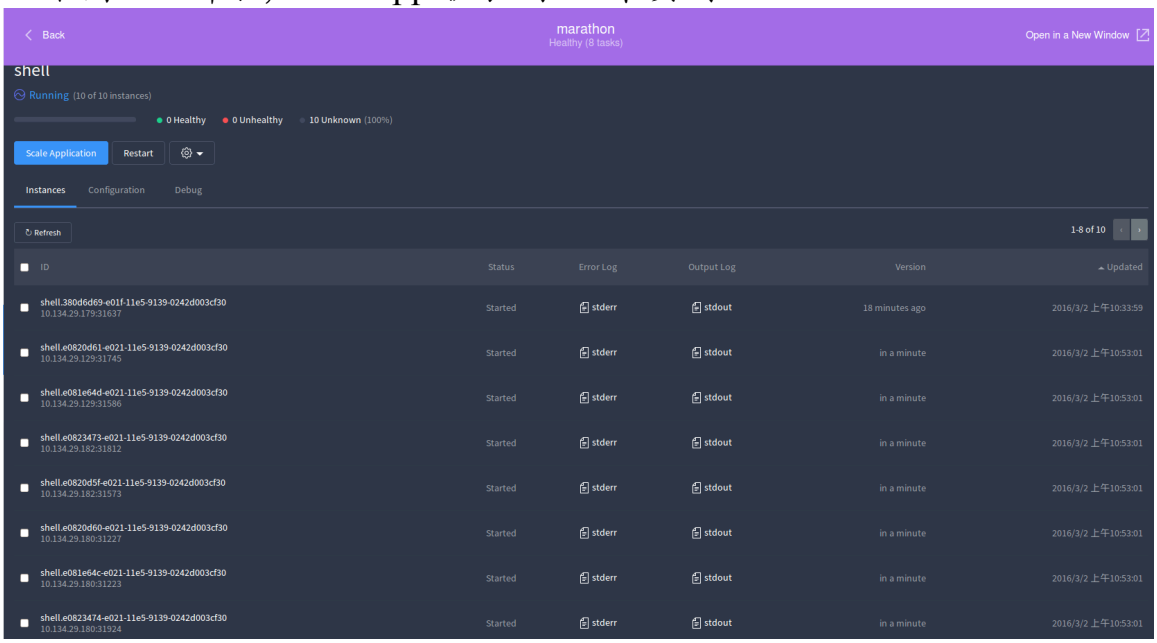

图 3-3

既然可以扩展，当然也可以销毁，选择你所需销毁的 task，点击 kill & scale 即可，如果点击 kill，marathon 将会继续创建 task 至 10 个为止。使用 ps 命令查看 shell 所在节点的进程 ID，使用 lsof 查看该进行打开的文件，结果如图 3-4 所示。

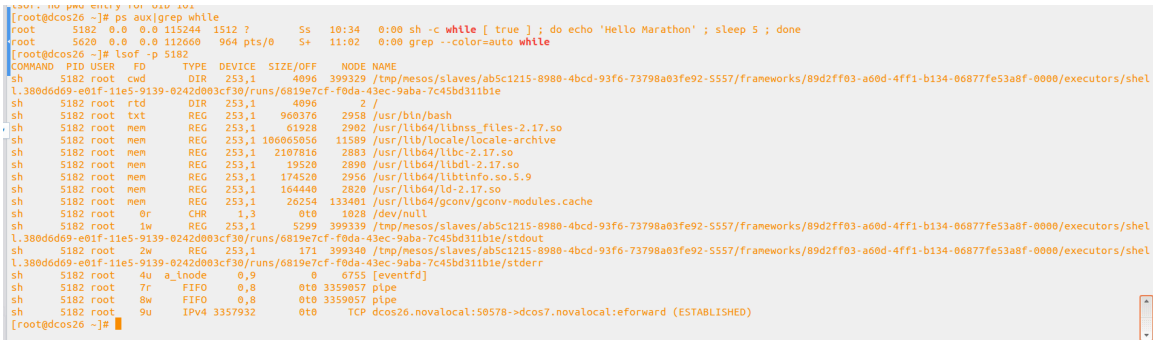

图 3-4

除了使用命令行创建 app，当然也可以使用 Web-UI 创建，点击主界面的 create，在弹出选框中填入相应参数即可，具体如图 3-5 所示。

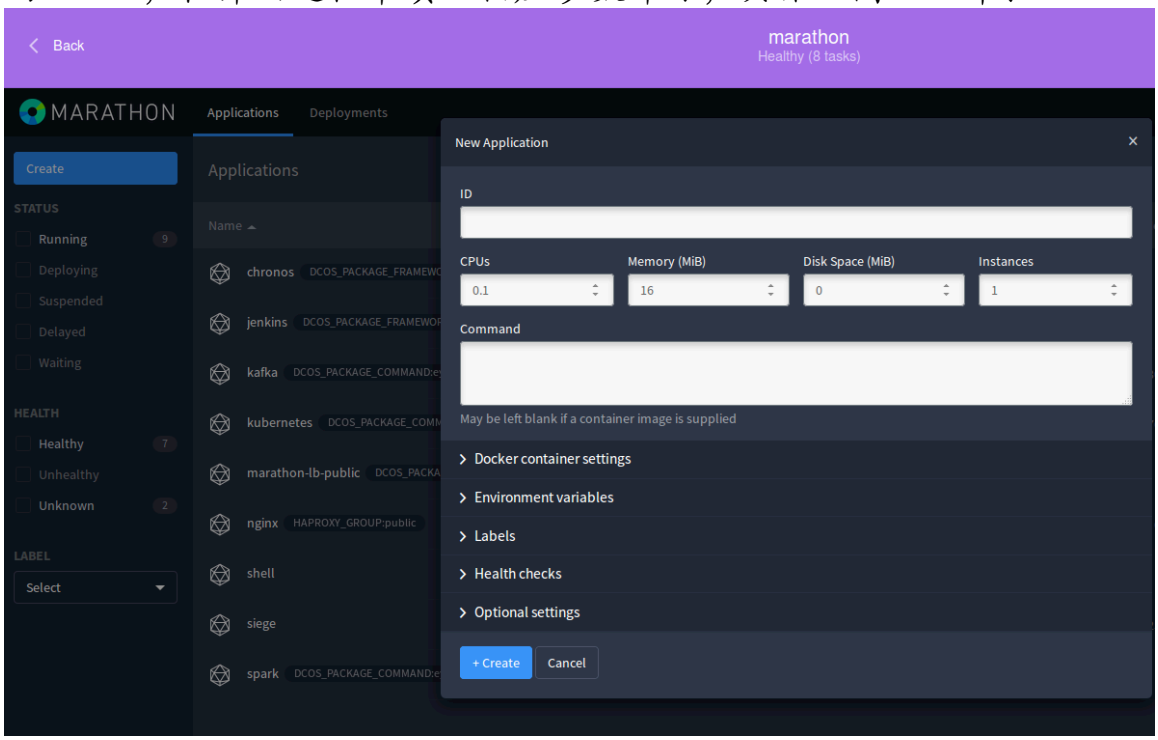

图 3-5

这一小节，关于 marathon 的基础应用部分介绍完毕，下面小节将介绍 marathon 如何应用远程资源。

3.2 运行远程资源

对于复杂应用，无法通过 cmd 命令传递所有操作，对于此类情况，Marathon 提供 uri 参数，在执行调度前，利用 Mesos fetcher 来下载、解压操作来提取资源。

在深入探讨此话题前，透过实例来了解应用场景：

```
{
    "id": "resource",
    "cmd": "./shell.sh",
    "cpus": 0.1,
    "mem": 10.0,
    "instances": 1,
    "uris": [
        "http://10.128.3.75/images/shell.sh"
    ]
}
```

上述实例在执行 cmd 前，会通过 Mesos 下载 http://10.128.3.75/images/shell.sh，之后会在所选 slave 节点的应用 sandbox 中执行该资源，可以通过 web 界面查看该任务的 sandbox，单击进入页面，可以发现 Mesos 下载的 shell.sh 脚本。

需要注意的是，Mesos v0.22 及以上版本在默认情况下执行 cmd 的方式，是先设权限后执行，因此，cmd 命令类似于 chmod u+x shell.sh && sh shell.sh。

除了上述提及的功能之外，Marathon 框架自身清楚框架内的应用资源。当然，Marathon 对于下述文件将首先尝试解压并提取资源：

- .tgz
- .tar.gz
- .tbz2
- .tar.bz2
- .txz
- .tar.xz
- .zip

对于此功能，可以假设以下场景：有一应用的压缩文件位于 http://10.128.3.75/images/shell.zip，此压缩文件包含 shell 脚本 shell.sh,cmd 需要执行此脚本，json 文件可以如下：

```
{
    "id": "shell",
```

```
    "cmd": "shell/shell.sh",
    "cpus": 0.1,
    "mem": 10.0,
    "instances": 1,
    "uris": [
        "http://10.128.3.75/shell.zip"
    ]
}
```

uris 对资源进行定位下载，Marathon 支持多种协议类型，种类如下所示：

- file:
- http:
- https:
- ftp:
- ftps:
- hdfs:
- s3:
- s3a:
- s3n:

uri 的值是数组类型，可以支持同时输入多个资源：

```
{
    ...
    "uris": [
"https://github.com/zouyee/repo.zip",
"http://10.128.3.75/images/shell.sh",
"ftp://10.128.3.75/images/my-other-file.css"
    ]
    ...
}
```

3.4 容器运行

3.4.1 简单应用

Marathon 可以使用 docker 对应用进行高效快捷的部署，在下述应用实例中，使用 docker 部署一简单 web 应用：使用 Docker 的 python:3 镜像，启动一个容器内部端口 8080 的服务，网络模式选择 bridge，因此有 portMapping 选项，在其字段中，hostport 值设为 0，意味着 Marathon 任意分配映射到外部的端口，json 内容如下所示：

```
{
  "id": "web",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 32.0,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "python:3",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 8080, "hostPort": 0 }
      ]
    }
  }
}
```

通过 HTTP API 接口启动该应用：

curl -X POST http://10.134.29.134:8080/v2/apps -d web.json -H "Content-type: application/json"

通过 dcos client 启动该应用，dcos marathon app add web.json

通过 Marathon web UI 界面可以看到名为 web 的应用已经运行。

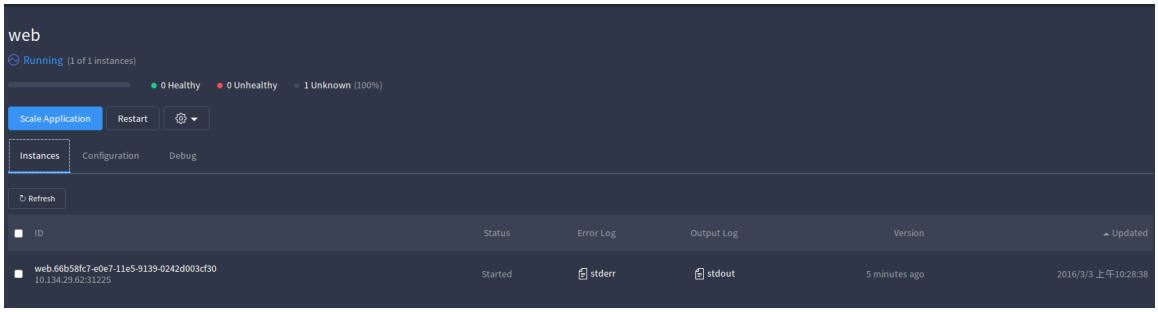

图 3-6

3.4.2 端口分配

Marathon 涉及到端口配置或者端口概念的地方有三处，第一部分是在应用配置的 container 中的 portMapping，主要有 containerport、

hostport、serviceport，如图 3-7 所示，第二处在应用配置的 Optional settings 中的 Ports，如图 3-8 所示，第三处在实际 App 中某一 Task 分配的 port(s)，如图 3-9 所示。

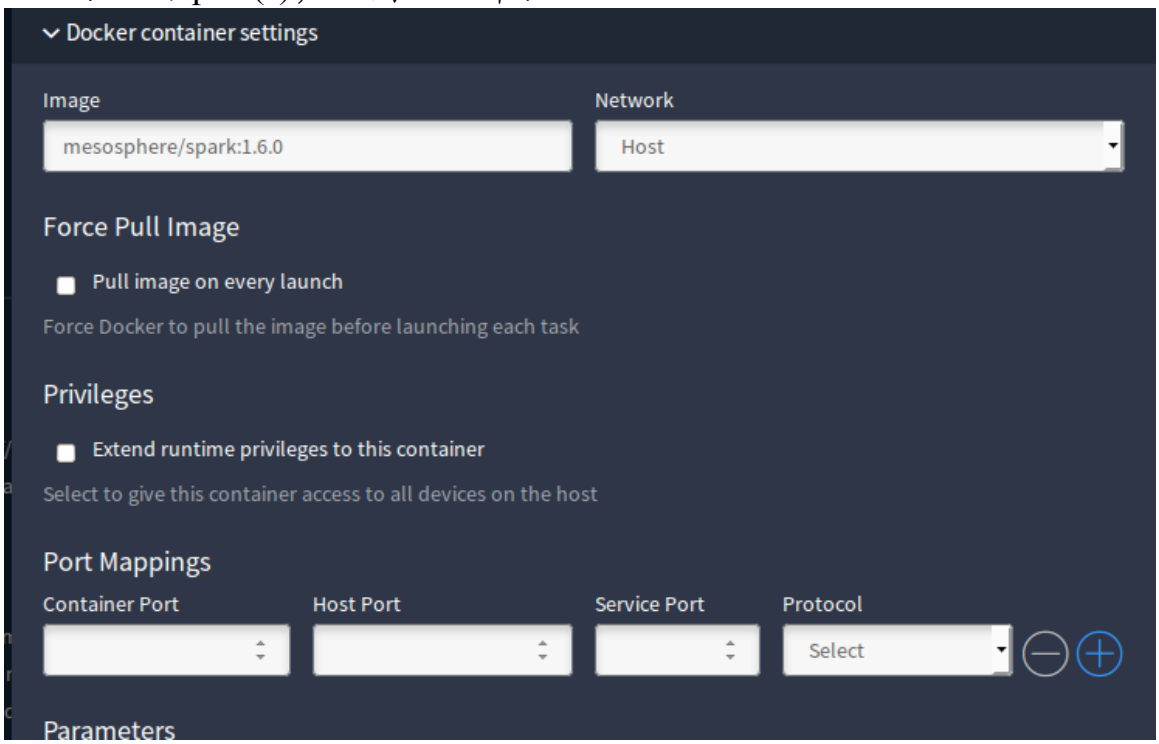

图 3-7 container 中的 portMapping

图 3-8 Optional settings 中的 Ports

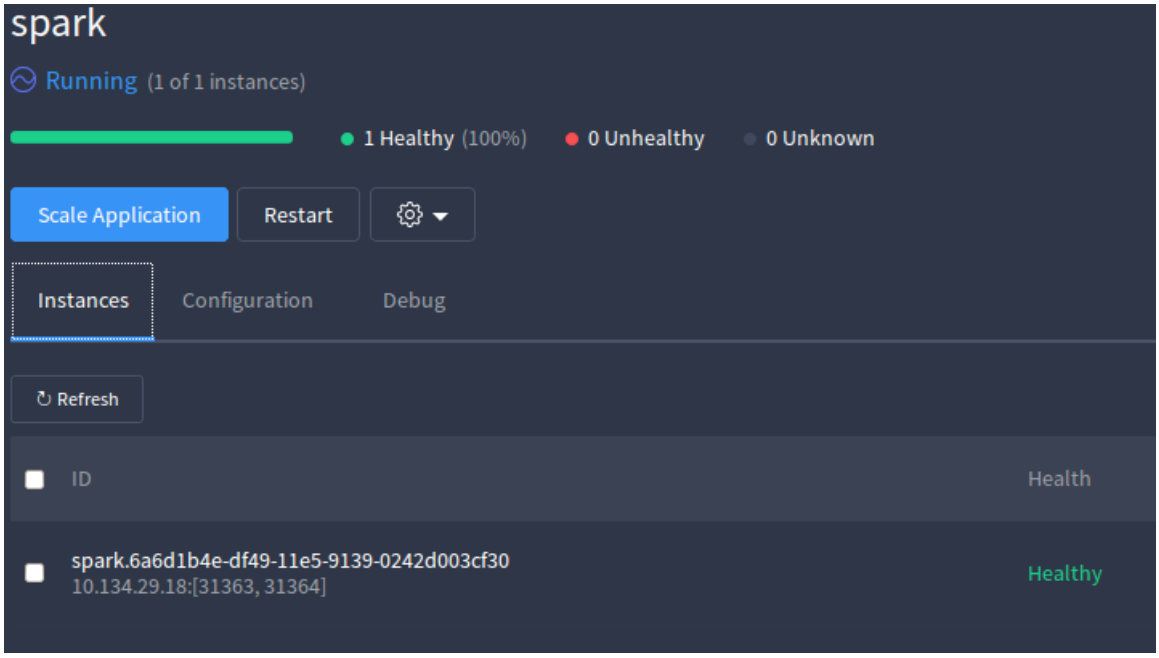

图 3-9 Task 分配到的 port

通过图 3-7 可以发现，Port Mappings 包括 Container Port、Host Port、Service Port、Protocol 等字段，图 3-8 可以发现，Optional settings 包含 Ports 字段。

containerPort：container Port 指定在容器内部的端口，它适用于 docker 的 bridge 网络做 port mapping。

hostPort：host Port 指定主机绑定端口，当使用 BRIDGE 网络，需要指定从主机端口到容器端口的 port mapping，当使用 HOST 网络，请求端口默认为主机端口。

BRIDGE 网络：docker 应用可以使用 BRIDGE 网络。在此网络环境中，container port(容器内部端口)对应 host port(主机上的端口)。

HOST 网络：HOST 网络可用于非 docker 的 Marathon 应用和 docker 应用，此模式中，应用直接绑定主机的一或者多个端口。

ports：这个 port 需要被视作一种资源，在使用 HOST 网络时，就需要设定。

protocol:协议指定使用的端口(比如 tcp、udp)

servicePort：Marathon 不绑定此端口，其被用作服务发现。

如果在 portMapping 中 containerPort 设为 0，它的值将会与 hostPort 一致，hostPort 将随机分配，默认范围在 31000-32000 之间。

3.5 健康检查

健康检查针对的是 App 下的每一个 Task,Marathon 框架通过健康检查来实现应用的故障恢复，健康检查实现了对 Task 的生命周期的管理。

Marathon 将应用的可恢复性与监控检查结合在一起，在状态发现变化时，触发 scale 操作，保证原有的可用服务的数量，如图 3-10 所示是 Marathon 健康检查的状态机。

Task 有三种活动状态：健康，非健康和扩展中，状态变化根据逻辑运算进行判定，三个参数主要是：请求实例数 i，健康实例数 h，运行实例数 r。

当 h=r !=i 时，即健康实例数等于运行实例数但不等于请求实例数，运行状态将变为 scaling，启动 i-r 个实例。

Marathon 在健康检查中设置了相关选项,健康检查主要有三种方式：HTTP、TCP、COMMAND，主要选项有以下几种：

其中 gracePeriodSeconds、intervalSeconds、MaxConsecutiveFailures、timeoutSeconds 适用于所有检查方式，portIndex、port 适用于 TCP 与 HTTP，path、ignoreHttp1xx 只适用于 HTTP。

1）intervalSeconds：健康检查周期，默认为 60s

2）timeoutSeconds：健康检查等待的超时失败时间，默认为 20s

3）path：健康检查的请求访问路径，支持 HTTP，默认为/

4）gracePeriodSeconds：允许忽略的检查失败最长时间，默认为 300s。

5）MaxConsecutiveFailures：规定在多少次健康检查失败后为 unhealthy 服务，默认为 3s

6）protocol：健康检查采用的协议，对于 COMMAND，欲使其有效，需要在 Marathon 启动时设置”--executor health_checks”选项，其表明未明确 executor 时的默认选择为 HTTP

7）portIndex：对服务进行健康检查时，访问的目的端口是 host port，在 Marathon 中是随机分配的，并且一个服务可以存在多个端口，因此使用 portIndex 定义健康检查的端口的索引值，默认为 0。

8）ignoreHttp1xx：忽略返回状态码 100-199，默认为 false，健康检查获取的返回码在此范围内，查询结果无效，状态保持不变。

9）command：Marathon 的健康检查基于最初的端口资源规则，对于 Docker 容器，服务端口即监听端口地址都与此规则不同，例如 Docker 容器要求像虚机一样有主机的 IP，并且每个服务端口都是特定的，那么这样的情况就需要使用 command 方式，使用外部命令行实现。

下述三个实例分别使用 HTTP、TCP 和 COMMAND 实现健康检查。

HTTP 如下所示：

```
}
   "path": "/api/health",
   "portIndex": 0,
   "protocol": "HTTP",
   "gracePeriodSeconds": 300,
   "intervalSeconds": 60,
   "timeoutSeconds": 20,
   "maxConsecutiveFailures": 3,
   "ignoreHttp1xx": false,
}
```

TCP 如下所示：

```
{
   "portIndex": 0,
   "protocol": "TCP",
   "gracePeriodSeconds": 300,
   "intervalSeconds": 60,
   "timeoutSeconds": 20,
   "maxConsecutiveFailures": 0
}
```

COMMAND 如下所示：

```
{
   "protocol": "COMMAND",
   "command": { "value": "curl -f -X GET http://$HOST:$PORT0/health" },
   "gracePeriodSeconds": 300,
   "intervalSeconds": 60,
   "timeoutSeconds": 20,
   "maxConsecutiveFailures": 3
}
```

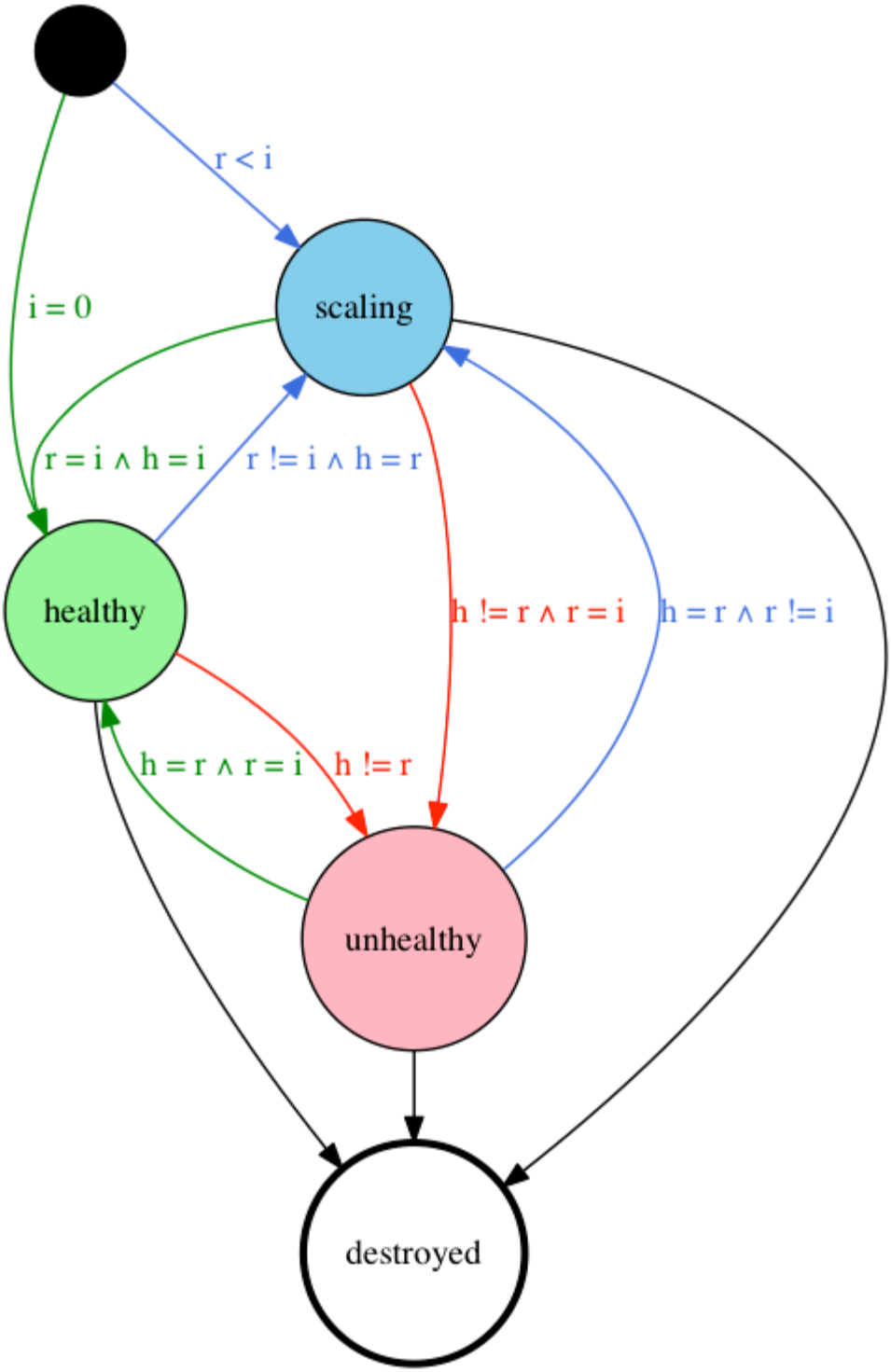

图 3-10 Marathon 健康检查的状态机

3.6 约束语法

3.6.1 概念

Marathon 可以通过 Constraints 来控制其 app 在何处运行，可以通过 Marathon 的 HTTP API 或者 dcos client 来设置 app 的 constraints 配置。

Constraints 由三个部分组成：字段名(field name)，操作(operator)，可选参数(optional parameter)，其中字段名(field name)可以为 mesos 的一个 slave 的 hostname 或者 Mesos slave 的 attribute。hostname 匹配 slave 的 hostnames。

hostname 支持所有的 operator，attribute 匹配 Mesos slave 的 attributes 字段。我们可以通过在 mesos 节点上运行 mesos-slave --help 来学习如何设置 mesos slave 的 attributes。

3.6.2 实例介绍

结合实例，来介绍下 Marathon Constrains 的用法，主要分为 UNIQUE 操作、CLUSTER 操作、GROUP_BY 操作、LIKE 操作、UNLIKE 操作。

UNIQUE 是一种唯一性约束，它强制 Marathon 对 app 中的 task 对资源进行独占，资源的类型根据字段名(field name)来确定，下面有个约束实例，其只允许名为 shell 的 app 的一个 task 只运行在一台主机之上：

```
$ curl -v -X POST http://10.134.29.134:8080/v2/apps \
      -H Content-Type:application/json -d '{
         "id":"shell",
         "container": {
         "type": "DOCKER",
            "docker": {
               "image": "10.13319.25:5000/bcec/nginx",
               "network": "BRIDGE",
               "portMappings": [
                  { "containerPort": 80, "hostPort": 0, "protocol": "tcp"}
               ]
            }
         },
```

```
        "constraints": [["hostname", "UNIQUE"]],
        "cmd": "",
        "cpus": 0.1,
        "mem": 64.0,
        "instances": 17
}'
```

其中 field name = hostname, operator = UNIQUE，在每个 slave 上只起一个 instance。Dcos 集群的 slave 节点只有 16 个 slave，因此在 Marathon 的管理界面上看到 app shell 为 ustaged 状态。

CLUSTER 与 UNIQUE 是相反的,CLUSTER 让 App 的 task 运行在共享资源中，资源的类型根据字段名(field name)来确定，下面有两个约束实例。实例一的约束要求是 task 运行到 slave_id 为 slave_public 的服务器上。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
      "id": "shell",
      "cmd": "echo ‘shell’",
      "instances": 3,
      "constraints": [["slave_id", "CLUSTER", "slave_public"]]
  }'
```

实例二要求是 task 运行到 host 为 dcos7 的主机上。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
      "id": "shell-2",
      "cmd": "echo ‘shell’",
      "instances": 2,
      "constraints": [["hostname", "CLUSTER", "dcos7"]]
  }'
```

GROUP_BY 能够让 task 按照某属性进行均匀分配到所有资源上以保证高可用。例如下述约束条件，其将 task 按照 slave_id 字典的值进行资源分组。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
      "id": "test_for_group-by",
      "cmd": "sleep 60",
      "instances": 3,
```

```
    "constraints": [["slave_id", "GROUP_BY"]]
  }'
```

如果不想按照 slave_id 进行依次分配，则可以设置参数来规定分组数。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
    "id": "test_for-group-by_value",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["slave_id", "GROUP_BY", "3"]]
  }'
```

上述实例设置 task 只在 3 个 slave_id 中进行分发。

LIKE 接受一个正则作为参数，将 task 分配到满足此正则表达式的 slave 节点上，例如，下述约束要求将 task 分配到 id 在 1-3 之间。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
    "id": "test_like-group-by",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["slave_id", "LIKE", "slave-[1-3]"]]
  }'
```

UNLIKE 与 LIKE 刚好相反，它同样接受一个正则作为参数，将 task 分配到满足此正则表达式的 slave 节点之外，例如，下述约束要求将 task 分配到 id 在 1-3 之外。

```
$ curl -X POST -H "Content-type: application/json"
10.134.29.134:8080/v2/apps -d '{
    "id": "test_unlike-group-by",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["slave_id", "UNLIKE", "slave-[1-3]"]]
  }‘
```

3.7 应用群组

对于应用来说，最小的单位就是一个独立容器即一个 Task，一个应用可以包含多个 Task，在生产环境中，一个应用由多种不同业务模块构成，Marathon 将多应用的合集称为应用组(Application Group)。

应用群组的应用场景应该是微服务改造的过程，用户可以将一个应用所包含的容器安装依赖做好定义，如图 3-11 是一个定义好的应用组。

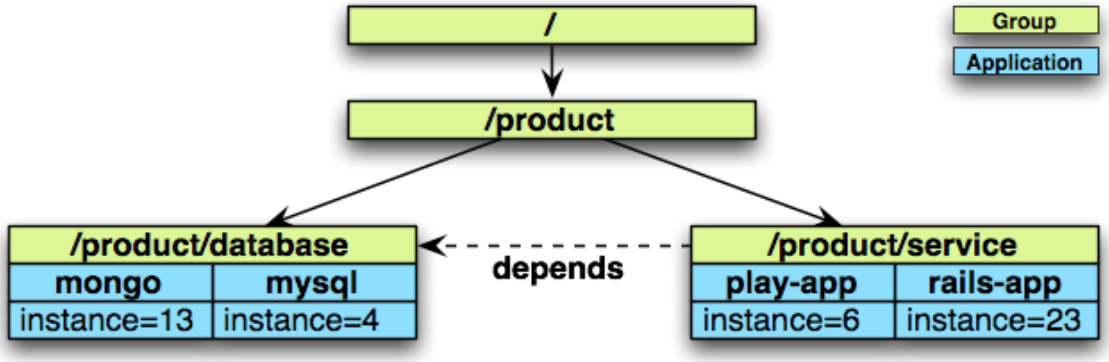

图 3-11 Marathon 应用组

上图所示是一个树状结构，跟节点为组名，如图 3-11 所示，定义了 product 应用组，其中包含 database、service 两个子应用组，database 应用组包括 mongo 应用 13 个 task，mysql 应用 4 个 task，service 应用组包括 play-app 樱桃 6 个 task，rails-app 应用 23 个 task。

上述 JSON 格式如下所示：

```
{
  "id": "/product",
  "groups": [
    {
      "id": "/product/database",
      "apps": [
          { "id": "/product/mongo", ... },
          { "id": "/product/mysql", ... }
       ]
    },{
      "id": "/product/service",
      "dependencies": ["/product/database"],
      "apps": [
          { "id": "/product/rails-app", ... },
          { "id": "/product/play-app", ... }
      ]
    }
  ]
}
```

应用之间存在依赖关系，依赖关系可以是应用级别的依赖，也可以是应用组级别的依赖，如果是应用组级别的依赖，那么组内的子分组、

应用都会等待依赖服务对象可用后才会启动。依赖定义可以表述为相对路径或者绝对路径，下面是对同一依赖的表述：

```
{
   ...
   "dependencies": ["/product/database"],
   "dependencies": ["../database"],
   "dependencies": ["specific/../../database"],
   ...
}
```

3.8 命令行设置

在 Marathon 启动时可以设置相关的命令行参数，这些参数也可以通过环境变量传入，其格式为 MARATHON_OPTION_NAME（前缀加上 MARATHON_）,例如 MARATHON_MASTER 与--master，如果同时设置了，MARATHON_MASTER 将替换掉--master。

必填选项：

--master：Mesos master URL，使用 zookeeper 可以填成 zk://host1:port,host2:port2/mesos

选填选项：

--artifact_store: 默认为 None，artifact_store URL，可以填成 hdfs://localhost:54310/path/to/store，file:///var/log/store

--access_control_allow_origin: 默认为 None，允许 HTTP 访问的 URL

例如：*,”http://localhost:8888”,”http://domain.com”

--[disable_]checkpoint: 默认为 enable，开启 Task 状态检查，需要在 slave 节点先开启状态检查(v0.13.0)

--executor: 默认为//cmd，当未指定执行器时，默认的执行器

--failover_timeout：默认 604800s，Marathon 启动而未注册超时时间长于此时间，将把该 Marathon 启动的 Task 全部 shutoff，需要开启状态检查

--framework_name: 默认为 marathon-VERSION,框架注册在 Mesos 上的名称

--[disable_]ha: 默认为 enabled：Marathon 通过选举方式实现 HA（v0.13.0）

--hostname: 默认为主机 hostname

--webui_url: 默认为 None，Marathon 的 webui URL

--leader_proxy_connection_timeout: 默认为 5000，通过当前 Marathon leader 到 Marathon 实例的超时时间(v0.9.0)

--leader_proxy_read_timeout: 默认为 10000，读取当前 Marathon leader 的时间(v0.9.0)

--local_port_max: 默认为 20000，为 app 分配全局唯一的服务端口的最大值，如果你静态指定，将可以不在此范围内

--local_port_min: 默认为 10000，为 app 分配全局唯一的服务端口的最小值，如果静态指定，将可以不在此范围内

--mesos-role: 默认为 None，此框架需要的 Mesos role，如果设定，在资源申请时，将分配 role 为'*'的资源

--default_accepted_resource_roles: 默认为 all(v0.9.0)

--mesos_user: 默认为 current user，使用该 framework 的 Mesos user

--reconciliation_initial_delay: 默认为 15000(15s)

3.9 应用部署

Marathon 内应用或者应用组的每一次改变都被视作部署，启动、停止、升级、扩展一个应用或者多应用，都属于部署的范畴，部署中需要处理应用间的依赖关系，如图 3-12 所示:

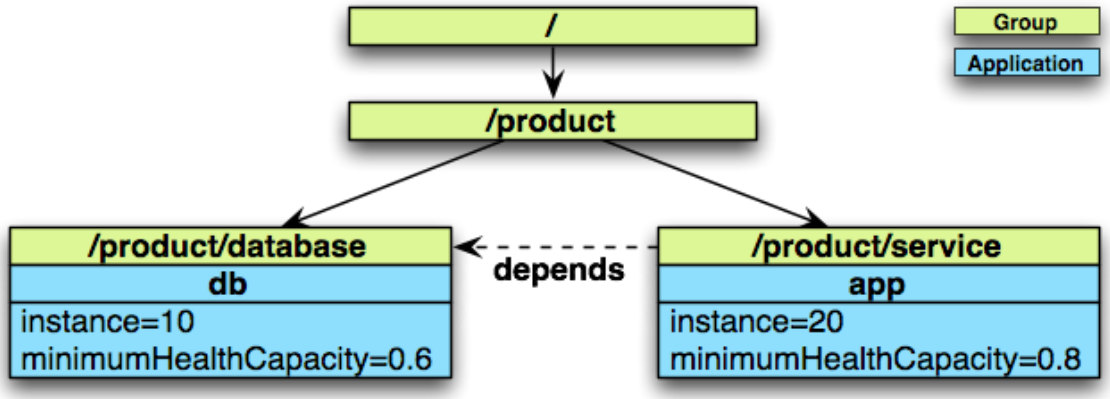

图 3-12 应用依赖

上图表明，应用组 product 包含 database 与 service 两个子应用组，service 应用组的应用 app 依赖与 database 应用组的 db 应用，在此情况下，应用部署状态如下所示:

- 启动：先启动 db 再启动 app
- 停止：先停止 app 在停止 db
- 更新：更新规则依据 minumumHealthCapacity，下面详述
- 扩展：先扩展 db，在扩展 app

关于应用更新，有以下多种策略：

1）轮询重启

应用的更新迭代过程中，如何实现应应用平滑升级，服务性能损耗最低，Marathon 通过 minumumHealthCapacity 参数，来控制应用新版本 Task 的启动与旧版本的停止。

minumumHealthCapacity==0：在新 Task 启动前，旧 Task 必须全部停止

minumumHealthCapacity==1：在新 Task 启动后，再删除旧 Task

0<minumumHealthCapacity<1：按照参数设置，实现旧 Task 的缩容，新 Task 的扩容，最后新 Task 健康检查通过，再停止旧 Task

下面通过实例进行介绍，通过图 3-12 可知，应用 db 存在实例 10 个，minumumHealthCapacity=0.6，应用 app 存在 20 个实例，minumumHealthCapacity=0.8，下面描述升级过程：

1、缩容 db 的 6 个旧版本实例
2、启动 db 的 6 个新版本实例
3、将 app 的旧版本实例缩为 16
4、将 app 的新版本扩充为 16
5、健康检查通过后停止 app 的所有旧实例
6、健康检查通过后停止 db 的所有旧实例
7、将 db 的新实例扩充到 10
8、将 app 的新实例扩充到 20

2）强制部署

部署由一系列步骤构成，步骤间存在依赖关系，如果存在以下情况，部署将无法完成：

- 一个新应用无法正确启动
- 一个新应用无法检查正常
- 依赖的新应用未申明或者无效
- 集群能力超出其本身能力
- ...

如果出现上述情况，则部署无法继续，此时为了恢复应用，需要采取强制部署。

3）/v2/deployments

Marathon 提供查询部署信息与管理部署状态的 API 接口，部署主要包括以下三种信息：

- affectedApps：哪些应用将被该部署影响
- steps：该部署包含哪些步骤
- currentStep：当前执行的部署步骤

上述步骤包含以下动作：

- ResolveArifacts：解决应用所需的资源下载解压操作
- StartApplication：开始指定 App
- StopApplication：停止指定 App
- ScaleApplication：扩展 App
- RestartApplication：重启 App
- KillAllOldTasksOf：杀掉所有旧 Task

3.10 事件总线

Marathon 内部事件总线可以捕获所有 API 请求与 scaling 事件，事件总线与负载均衡、统计数据可以有效集成，事件总线具有被动查询与主动通知的接口。

添加如下命令行选项使 Marathon 成为事件订阅者：

```
$ ./bin/start --master ... --event_subscriber http_callback --http_endpoints
http://host1/foo,http://host2/bar
```

host1 与 host2 主机将介绍事件通知。

事件类型主要有以下几种：

1）Marathon 接受关于应用变化(创建、更新、删除)的通知

```
{
  "eventType": "api_post_event",
  "timestamp": "2014-03-01T23:29:30.158Z",
  "clientIp": "0:0:0:0:0:0:0:1",
  "uri": "/v2/apps/my-app",
```

```
  "appDefinition": {
    "args": [],
    "backoffFactor": 1.15,
    "backoffSeconds": 1,
    "cmd": "sleep 30",
    "constraints": [],
    "container": null,
    "cpus": 0.2,
    "dependencies": [],
    "disk": 0.0,
    "env": {},
    "executor": "",
    "healthChecks": [],
    "id": "/my-app",
    "instances": 2,
    "mem": 32.0,
    "ports": [10001],
    "requirePorts": false,
    "storeUrls": [],
    "upgradeStrategy": {
        "minimumHealthCapacity": 1.0
    },
    "uris": [],
    "user": null,
    "version": "2014-09-09T05:57:50.866Z"
  }
}
```


2）Task 状态改变的通知


```
{
  "eventType": "status_update_event",
  "timestamp": "2014-03-01T23:29:30.158Z",
  "slaveId": "20140909-054127-177048842-5050-1494-0",
  "taskId": "my-app_0-1396592784349",
  "taskStatus": "TASK_RUNNING",
  "appId": "/my-app",
  "host": "slave-1234.acme.org",
  "ports": [31372],
  "version": "2014-04-04T06:26:23.051Z"
}
```

Task 终结状态主要有:

- TASK_STAGING
- TASK_STARTING
- TASK_RUNNING
- TASK_FINISHED
- TASK_FAILED
- TASK_KILLED
- TASK_LOST

3）Framework 的通知


```
{
  "eventType": "framework_message_event",
  "timestamp": "2014-03-01T23:29:30.158Z",
  "slaveId": "20140909-054127-177048842-5050-1494-0",
  "executorId": "my-app.3f80d17a-37e6-11e4-b05e-56847afe9799",
  "message": "aGVsbG8gd29ybGQh"

}
```


4）部署事件通知


```
{
  "eventType": "group_change_success",
  "timestamp": "2014-03-01T23:29:30.158Z",
  "groupId": "/product-a/backend",
  "version": "2014-04-04T06:26:23.051Z"

}
```


3.11 应用实例

下述只要介绍多种应用特性:

1）Docker 实例

```
{
    "id": "simple-docker",
    "container": {
      "docker": {
        "image": "busybox"
      }
    },
    "cmd": "echo hello from docker",
    "cpus": 0.2,
    "mem": 32.0,
    "instances": 2
}
```

2) 带 entry point 的 docker 实例

```
{
    "id": "simple-docker",
    "container": {
      "docker": {
        "image": "mesosphere/inky"
      }
    },
    "args": ["hello", "from", "docker"],
    "cpus": 0.2,
    "mem": 32.0,
    "instances": 2
}
```

Docker 镜像“mesosphere/inky”的 Dockerfile 定义如下：

```
FROM busybox
MAINTAINER support@mesosphere.io
CMD ["inky"]
ENTRYPOINT ["echo"]
```

3）启动一个 docker registry，并持久化数据到本地 volume

```
{
  "id": "/docker/registry",
  "instances": 1,
  "cpus": 0.5,
  "mem": 1024.0,
```

```
  "disk": 128,
  "container": {
    "docker": {
      "type": "DOCKER",
      "image": "registry:latest",
      "network": "BRIDGE",
      "parameters": [],
      "portMappings": [
        {
          "containerPort": 5000,
          "hostPort": 0,
          "protocol": "tcp",
          "servicePort": 5000
        }
      ]
    },
    "volumes": [
      {
        "hostPath": "/local/path/to/store/packages",
        "containerPath": "/storage",
        "mode": "RW"
      }
    ]
  },
  "env": {
    "SETTINGS_FLAVOR": "local",
    "STORAGE_PATH": "/storage"
  },
  "ports": [ 0 ]
}
```

4） Marathon 0.8.2 与 Mesos 0.22.0 支持在启动 task 前强制 docker 先拉取镜像

```
{
  "type": "DOCKER",
  "docker": {
    "image": "group/image",
```

```
    "forcePullImage": true
  }}
```

四、Marathon 之高可用篇

4.1 介绍

Marathon 默认提供高可用模式,其允许应用在 Marathon 内部个别实例失联情况下稳定允许，Marathon 的高可用特性的实现，需要启动多个 Marathon 实例并使其都指向相同的 zookeeper。

4.2 配置

当设置为 True 时, --ha 命令行参数将以 HA 模式启用 Marathon, HA 设置默认为 true。

每个 Marathon 实例启动时，都需要指定相同的 zookeeper，例如：zk=//10.134.29.134,10.134.29.135,10.134.29.136/marathon，以如下命令行启动每个 Marathon 实例:

--zk zk=//10.134.29.134,10.134.29.135,10.134.29.136/marathon

4.3 代理

跟 Mesos web 不同的是，Marathon web 没有重定向到当前的处于 leader 的 Marathon 实例。它是通过代理请求转发功能，实现 Marathon web 上显示的数据都是当前处于运行状态的应用。

五、Marathon 之 SSL 与基本认证篇

Marathon 可以通过 SSL 进行 API 端点的安全访问也恶意通过 HTTP 基本接入认证进行有限访问。如果计划使用基本认证方式,我们建议使用 SSL 方式,因为其他方式由于用户名和密码未被加密,信息容易被其他方获取。

5.1 使用 SSL

如果已经有了 Java keystore,可以将其连通密码一起通过 SSL 传递至 Marathon

```
$./bin/start --master zk://10.134.29.134:2181
10.134.29.135:2181,10.134.29.136:2181/mesos --zk
zk://10.134.29.134:2181 ,
10.134.29.135:2181,10.134.29.136:2181/marathon --ssl_keystore_path
marathon.jks
--ssl_keystore_password $MARATHON_JKS_PASSWORD
```

在默认情况下,Marathon 使用 8443 端口来对外提供 SSL 服务(可以通过--https_port 命令来更改端口),一旦 Marathon 运行,将会通过 HTTPS 端口来接入 API 以及 UI:

```
$curl https://l0134.29.134:8443/v2/apps
```

5.2 生成带有 SSL 密钥的 keystore

需要注意的是:现在的浏览器以及大部分工具,将会对于 Marathon API 以及 UI 的访问进行告警，除非 SSL 已被 CA 授信，当然也可以直接购买。

① 通过 OpenSSL 生成 RSA 私有密钥

将密码设置为 MARATHON_KEY_PASSWORD 变量

```
$openssl genrsa   -des3
                  -out marathon.key
                  -passout "env:MARATHON_KEY_PASSWORD"
```

② 通过下面的方法为密钥获得认证

-推荐从知名的 CA 购买认证。这个将确保 Marathon 实例的 API 以及 UI 信
任 SSL 认证,为用户减少额外的步奏

-(不安全方式)为密钥生成一个认证。这个命令会向与 keystore 交互的信息进行提示。“Common Name”必须是要使用认证的机器的限定主机名。

下面的命令会读取 MARATHON_KEY_PASSWORD 环境变量

```
$openssl req  -new     -x509
              -key marathon.key
              -passin   "env:MARATHON_KEY_PASSWORD"
              -out self-signed-marathon.pem
```

③ 将密钥和认证文件合并至 PKCS12 格式的文件中,PKCS12 这种格式也是 Java keystore 用的格式。如果接受的认证文件不是.pem 格式,则需要转换为.pem。

将密码设置为 MARATHON_KEY_PASSWORD 变量

将 PKCS 密码设置为 MARATHON_PKCS_PASSWORD 变量

```
$openssl pkcs12  -inkey   marathon.key
                 -passin  "env:MARATHON_KEY_PASSWORD"
                 -name    marathon
                 -in  trusted.pem
                 -password
"env:MARATHON_PKCS_PASSWORD"
                 -chain   -CAFile "trustedCA.crt"
                 -export  -out marathon.pkcs12
```

④ 导入 keystore

将 PKCS 密码设置为 MARATHON_PKCS_PASSWORD 变量

将 JKS 密码设置为 MARATHON_JKS_PASSWORD 变量

$keytool-importkeystore -srckeystore marathon.pkcs12
-srcalias marathon
-srcstorepass
$MARATHON_PKCS_PASSWORD
-srcstoretype PKCS12
-destkeystoremarathon.jks
-deststorepass
$MARATHON_JKS_PASSWORD

⑤ 通过创建的 keystore 以及其密码启动 Marathon
将 JKS 密码设置为 MARATHON_JKS_PASSWORD 变量
$./bin/start --master
zk://10.134.29.134:2181 10.134.29.135:2181,10.134.29.136:2181/mesos
--zk zk://10.134.29.134:2181 ,
10.134.29.135:2181,10.134.29.136:2181/marathon
--ssl_keystore_path marathon.jks
--ssl_keystore_password $MARATHON_JKS_PASSWORD

⑥ 通过 HTTPS 端口接入 Marathon API 与 UI(默认端口 8443)
https://10.134.29.134:8443

5.3 启用基础认证

备注:虽然可以使用基本认证,但是我们强烈建议使用 SSL。如果 SSL 不可用,
Marathon 获取的用户名和密码将会使用明文形式进行传输,可能会被第三方获取到相关信息。

将用户名和密码使用":"进行隔离传递至 --http_credentials 命令将会开启基本认证 。备注:用户名不能包含":".
$ ./bin/start --master --master
zk://10.134.29.134:2181 10.134.29.135:2181,10.134.29.136:2181/mesos
--zk zk://10.134.29.134:2181 ,
10.134.29.135:2181,10.134.29.136:2181/marathon

--http_credentials"cptPicard:topSecretPa$$word"
--ssl_keystore_path /path/to/marathon.jks
--ssl_keystore_password $MARATHON_JKS_PASSWORD

六、Marathon 之服务发现篇

Marathon 服务发现功能的实现是使用 Mesos-DNS，
~~Mesos-DNS~~ 的功能主要是提供 A 记录，SRV 服务，不提供 PTR 服务，本章关于 Mesos-DNS 主要分为下面几项内容:

- Mesos-DNS 简介
- Mesos-DNS 安装与配置
- Meoso-DNS 运行

这里主要描述的 Mesos-DNS 所提供的 DNS 服务

6.1 DNS 相关概念

①A 记录(Address): 是用来指定主机名(或者域名)对应的 IP 地址的记录，简单的说，A 记录是指定域名对应的 IP 地址。

②NS 记录(Name Server): 是域名服务器记录，用来指定该域名由哪个 DNS 服务器来进行解析，简单来说，NS 记录是指定哪个 DNS 服务器来解析你的域名。

③MX 记录(Mail Exchanger): 是邮件交换记录,它指向一个邮件服务器，用于电子邮件系统发邮件时根据收信人的地址后缀来定位邮件服务器，例如，当网上的用户要发一封信给 user@mydomain.com 时，该用户的邮件系统通过 DNS 查找 mydomain.com 这个域名的 MX 记录，如果 MX 记录存在，用户的计算机就 将邮件发送到 MX 记录所指定的邮件服务器上。

④TXT 记录:一般指某个主机名或者域名的说明，如：admin IN TXT “管理员，电话：XXX”

SOA 指定了域名服务器的 FQDN 和管理员的 email。

⑤PTR 值是 pointer 的简写，用于将一个 IP 地址映射到对应的域名，也可以视作 A 记录的反向，它提供对 IP 地址的反向解析，PTR 主要应用于邮件服务器.

⑥SRV 记录: 是服务器资源记录的缩写，SRV 记录的作用是用于描述一个服务器能够提供什么样的服务。

DNS 是一个分布式的系统，从根域名开始将次级域名的管理员独立管理，域名信息由一个称为域文件(zone file)的文件来描述，域文件又由各项资料记录(RR, Resource Records)组成，其中称为起始授权机构(SOA, start of authority)的资源记录，描述了域名的管理员、电子邮件地址，和一些时间参数。

比如通过 dig 命令来查询 oschina 的 SOA 记录

dig 工具的安装:

redhat 系列: yum install bind-utils
deb 系列: apt-get install dnsutils

查看 SOA 记录

执行查询 SOA 记录的命令，结果如下:

dig @114.114.114.114 oschina.net +nssearch
得到的其中一条结果为:
SOA ns1.dnsv2.com. level3dnsadmin.dnspod.com. 1408847720 3600 180 1209600 180 from server 221.204.186.8 in 12 ms.
依次各项的意思为:

- SOA 即为 SOA 记录
- ns1.dnsv2.com. Nameserver，该域名解析使用的服务器
- level3dnsadmin.dnspod.com. Email address，该域名管理者的电子邮件地址，第一个'.'代表电子邮件中的'@'，所以对应的邮件地址为: level3dnsadmin@dnspod.com
- 1408847720 Serial number，反映域名信息变化的序列号。每次域名信息变化该项数值需要增大。格式没有要求，但一般习惯使用 YYYYMMDDnn 的格式，表示在某年(YYYY)、月(MM)、日(DD)进行了第几次(nn)修改。
- 3600 Refresh，备用 DNS 服务器隔一定时间会查询主 DNS 服务器中的序列号是否增加，即域文件是否有变化。这项内容就代表这个间隔的时间，单位为秒。

- 180 Retry，这项内容表示如果备用服务器无法连上主服务器，过多久再重试，单位为秒。通常小于刷新时间。
- 1209600 Expiry，当备用 DNS 服务器无法联系上主 DNS 服务器时，备用 DNS 服务器可以在多长时间内认为其缓存是有效的，并供用户查询。单位为秒。1209600 秒为 2 周。
- 180 Minimum，缓存 DNS 服务器可以缓存记录多长时间，单位为秒。这个时间比较重要，太短会增加主 DNS 服务器负载。如果太长，在域名信息改变时，需要更长的时间才能各地的缓存 DNS 服务器才能得到变化信息。

这个时间比较重要，太短会增加主 DNS 服务器负载。如果太长，在域名信息改变时，需要更长的时间才能各地的缓存 DNS 服务器才能得到变化信息。

查看 SRV 服务

其中_jenkins._tcp.marathon.mesos 的命名规则: task.framwork.domain

执行查询 SRV 服务的命令，结果如下:

```
dig _jenkins._tcp.marathon.mesos SRV
; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.1 <<>>
_jenkins._tcp.marathon.mesos SRV
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id:
34049
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
ADDITIONAL: 1
;; QUESTION SECTION:
;_jenkins._tcp.marathon.mesos. IN SRV
;; ANSWER SECTION:
_jenkins._tcp.marathon.mesos. 60 IN SRV 0 0 31006
jenkins-z8kkj-s7.marathon.slave.mesos.
;; ADDITIONAL SECTION:
jenkins-z8kkj-s7.marathon.slave.mesos. 60 IN A 10.133.19.22
;; Query time: 0 msec
```

;; SERVER: 10.133.19.21#53(10.133.19.21)
;; WHEN: 三 1 月 13 11:03:12 EST 2016
;; MSG SIZE rcvd: 151

可以看一下其中的 _jenkins._tcp.marathon.mesos. 60 IN SRV 0 0 31006 jenkins-z8kkj-s7.marathon.slave.mesos.

- _jenkins 中 jenkins 是一种服务,表明这台服务器有向外提供 jenkins 服务
- _tcp 中 tcp 是本服务使用的协议，可以是 tcp 也可以是 udp
- marathon.mesos 此记录所值的域名
- 60： 此记录默认生存时间(s)
- IN: 标准 DNS Internet 类
- SRV: 将这条记录标识为 SRV 记录
- 0: 优先级，如果相同的服务有多条 SRV 记录，用户会尝试先链接优先级最低的记录
- 0: 负载平衡机制，多条 SRV 并且优先级也相同，那么用户会先尝试连接权重高的记录
- 31006: 此服务使用的服务端口
- jenkins-z8kkj-s7.marathon.slave.mesos: 提供服务的主机

下一条记录: jenkins-z8kkj-s7.marathon.slave.mesos. 60 IN A 10.133.19.22

- jenkins-z8kkj-s7.marathon.slave.mesos.： 提供服务的主机
- 60: 此记录的存活时间
- IN: 标准 DNS Internet 类
- A: address(就是上述提及的 A)
- 10.133.19.22: 对应的服务器的 ip

6.2 Mesos-DNS 介绍

Mesos-DNS 它主要提供域名解析服务，Mesos-DNS 框架支持服务发现在 Apache Mesos 集群中,允许应用程序和服务通过域名系统(DNS)来相互定位。 Mesos-DNS 充当的角色和在互联网中 DNS 的作用差不多。Mesos-DNS 的特点是轻量、无状态,易于部署和维护。由 Marathon 或者 Aurora 框架启动的应用程序或者常驻服务被赋予名字,如 earch.marathon.mesos 或者 log-aggregator.aurora.mesos,Mesos-DNS 将每

个集群中正在运行的应用程序的域名转换成 IP 地址和端口号。这使得任何链接到集群中运行的服务都可以通过 DNS 进行查找。

Mesos-DNS 被设计成简单并且无状态的,它不需要共识机制,永久储存以及日志。这是可以的,因为 Mesos-DNS 没有实现心跳,状态监测,或者管理应用方程序的生命周期。这些功能在 Mesos master, slaves,和 frameworks 中实现。

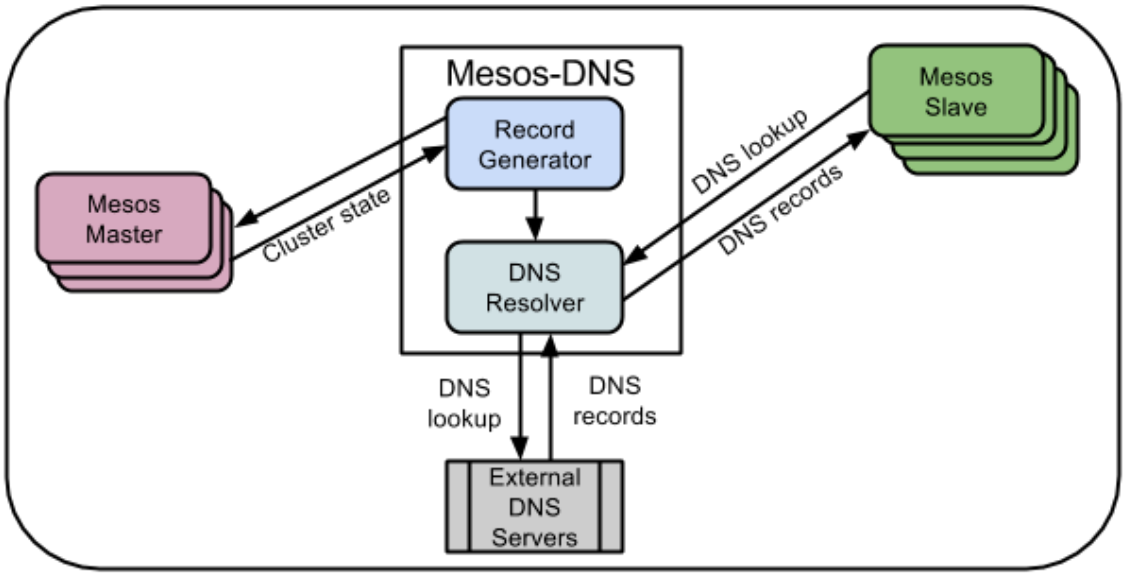

图 6.1 Mesos-DNS 架构

Frameworks 运行在 Mesos 中,不需要直接与 Mesos-DNS 通信。

Mesos-DNS 定期查询 Mesos master(s),检索所有运行的框架中正在运行的任务的状态,并为这些任务生成 DNS 记录，包括 A 记录与 SRV 记录。当 Mesos 集群中的任务开始,结束,或重启，Mesos-DNS 都需要更新 DNS 记录以保证为最新状态。

Mesos-DNS 通过 Mesos master(s) 制定的配置来运行,可以容错,并且可以通过一个框架,如 Marathon 来监视应用程序的健康状况。从故障中重启后，Mesos-DNS 检索 Mesos master(s)和 serves 的最新状态,并提供 DNS 请求。

Mesos-DNS,不是像其他的 DNS 系统,如 SkyDNS 或者 Consul。首先,我们需要一个 DNS 系统来密切配合 Mesos，而不是让每个用户或

者框架描述两次任务(一次 Mesos 执行,一次到 DNS 系统),它更容易和简洁的从Mesos传递信息到DNS。其次,我们需要一个解决方法,Mesos及其框架已经实现了容错性和生命周期的管理。我们不想强迫 Mesos 用户再为 DNS 部署另一套同步机制,永久储存或日志。

Mesos-DNS 也将延伸到 Mesos 引入的安全和网络方面,并集成到到来的 Datacenter Operating System (DCOS)中,以支持在公有云,私有数据中心以及混合部署中的服务发现。

命名规则

A(adress)即为记录,将 hostname 与 ip 进行关联(即上文所述的概念),在某一 DNS 域 A 中,当某一框架 B 启动某一任务 C,
Mesos-DNS 就会为 hostname 为 C.B.A 生成一条记录(A),对应的 ip 有下述两中情况:
1、任务执行的容器所提供的网络 ip(这个网络是 mesos 容器提供的)
2、任务在某一 slave 节点执行的 ip

需要解释的 A.B.C 即为 taks.framework.domain 这样的命名规则,例如,当 mesos 使用 marathon 框架执行 jenkins 任务,这样就可以知道其域名,即为:jenkins.marathon.mesos

SOA 记录

dig jenkins.marathon.mesos

```
; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.1 <<>> jenkins.marathon.mesos
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 40120
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 0
;; QUESTION SECTION:
;jenkins.marathon.mesos.            IN    A
;; ANSWER SECTION:
jenkins.marathon.mesos. 60    IN    A     10.133.19.22
jenkins.marathon.mesos. 60    IN    A     10.133.19.23
;; Query time: 0 msec
;; SERVER: 10.133.19.21#53(10.133.19.21)
```

;; WHEN: 三 1 月 13 13:58:02 EST 2016
;; MSG SIZE rcvd: 72

结果表明我们通过scale,扩展到2个task,位于不同的slave 节点上
当你将一个节点down掉，我们可以继续查下SOA记录:

dig jenkins.marathon.mesos

; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.1 <<>> jenkins.marathon.mesos
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 27520
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;jenkins.marathon.mesos. IN A
;; ANSWER SECTION:
jenkins.marathon.mesos. 60 IN A 10.133.19.22
;; Query time: 0 msec
;; SERVER: 10.133.19.21#53(10.133.19.21)
;; WHEN: 三 1 月 13 11:49:34 EST 2016
;; MSG SIZE rcvd: 56

除了task.framework.domain记录，Mesos-DNS通常还会产生一个A记录 task.framework.slave.domain,即为task执行的slave节点的ip。例如,jenkins.marathon.slave.mesos的A记录将运行marathon框架的jenkins应用的slave节点的ip。

在下述task 状态labels中，task的执行器必须提供容器ip

- Docker.NetworkSettings.IPAddress
- MesosContainerizer.NetworkSettings.IPAddress

SRV记录

服务器资源记录为某一服务分配一个hostname和一个ip port，对于在域A中，当框架B启动任务C,

这样 Mesos-DNS 将产生一个 SRV 记录名为_C._tcp.B.A,命名规则为_taskname._protocol.framework.domain,
其中 protocol 可以为 udp 或者 tcp，例如 mesos 使用 marathon 启动的任务 jenkins。

dig _jenkins._tcp.marathon.mesos SRV
; <<>> DiG 9.9.4-RedHat-9.9.4-29.el7_2.1 <<>>
_jenkins._tcp.marathon.mesos SRV
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34329
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 2
;; QUESTION SECTION:;_jenkins._tcp.marathon.mesos. IN SRV
;; ANSWER SECTION:
_jenkins._tcp.marathon.mesos. 60 IN SRV 0 0 31383
jenkins-hfa6x-s5.marathon.slave.mesos.
_jenkins._tcp.marathon.mesos. 60 IN SRV 0 0 31006
jenkins-z8kkj-s7.marathon.slave.mesos.
;; ADDITIONAL SECTION:
jenkins-hfa6x-s5.marathon.slave.mesos. 60 IN A 10.133.19.23
jenkins-z8kkj-s7.marathon.slave.mesos. 60 IN A 10.133.19.22
;; Query time: 0 msec;; SERVER: 10.133.19.21#53(10.133.19.21)
;; WHEN: 三 1 月 13 17:14:05 EST 2016
;; MSG SIZE rcvd: 241

可以看到上述的查询结果，对于查询内容就不一一解释了

Service	CT-IP Avai	DI Avail	Target Host	Target Port	A (Target Resolution)
{task}.{proto}.framework.domain	no	no	{task}.framework.slave.domain	host-port	slave-ip
	yes	no	{task}.framework.domain	di-port	slave-ip
	no	yes	{task}.framework.domain	di-port	slave-ip
				di-port	container-ip
{task}.{proto}.framework.slave.domain	n/a	n/a	{task}.framework.slave.domain	host-port	slave-ip

下述图标简述了 SRV 产生的规则:

其他记录

Mesos-DNS 产生一些特殊的记录:

- 对于 leading master
  产生:leader.domain 记录和 SRV(_leader._tcp.domain、_leader._udp.domain)
- 对于所有框架调度器
  产生:{framwork}.domain 和 SRV(_framwork._tcp.{framework}.domain)
- 对于 mesos master
  产生:master.domain 记录和 SRV(_master._tcp.domain、_master._udp.domain)
- 对于 mesos slave
  产生: slave.domain 记录和 SRV(_slave._tcp.domain)

如果需要配置 Mesos-DNS 去侦测 leading master(通过 Zookeeper)，只有一个 master 记录，但是对于 Mesos-DNS 来说，因为 leading master 需要经过选举过程，所以有时延。除了 A 记录和 SRV 记录以外，Mesos-DNS 还支持 SOA 以及 NS 记录，DNS 对于其他类型的请求将返回 NXDOMAIN，Mesos-DNS 不支持 PTR 记录来回溯 IP。

6.3 Mesos-DNS 安装与配置

要构建 Mesos-DNS,你必须在你的电脑上安装 go 和 godep。必须设置 GOPATH 环境变量指向 go

安装包的目录,必须添加$GOPATH/bin 到 PATH 环境变量。如果你安装 go 到自定义目录,需要

设置 GOROOT 环境变量,并且把$GOROOT/bin 添加到 PATH 环境变量中去。例如,执行以下

操作:

```
export   GOPATH=$HOME/go
export   PATH=$PATH:$GOPATH/bin
export   GOROOT=/usr/local/go
```

假设　　go 装在了/usr/local/go 目录下

export　PATH=$PATH:$GOROOT/bin

用 godep 构建 Mesos-DNS:

go　get　github.com/mesosphere/mesos-dns

cd　$GOPATH/src/github.com/mesosphere/mesos-dns

make all

这将生成一个静态的二进制 Mesos-DNS 文件,可以装在任意地方。在同一目录下你可以找到配置文件 config.json。

6.4 Mesos-DNS 运行

若要运行 Mesos-DNS,必须在所选的服务器上安装 mesos-dns 二进制文件。服务可以装在 Mesos 的任意一台机器上,或者是在同一网络的一台专用机器上。接下来,按照链接创建一个配置文件。

可以这样启动 Mesos-DNS

sudo　　mesos-dns　-config=config.json　&

为了加强容错能力,建议通过 Marathon 把 Mesos-DNS 服务发布到任意一台 Mesos 从节点上。如果 Mesos-DNS 失败,Marathon 将重新启动它,确保几乎不间断的服务。可以通过 Marathon 约束从节点的主机名或者任何从节点的属性来选择把 Mesos-DNS 发布到哪一个从节点上面。

例如下面的 json 描述了 Marathon 通过从节点的主机名发布 Mesos-DNS　　10.134.29.134:

```
{
"cmd":　"sudo /usr/local/mesos-dns/mesos-dns
-config=/usr/local/mesos-dns/config.json",
"cpus":　1.0,
"mem":　1024,
"id":　　"mesos-dns",
"instances":　1,
"constraints":　　[["hostname",　　"CLUSTER",　　"10.134.29.134"]]
}
```

注意这个主机名字段是指从节点向 Mesos 注册时使用的主机名。它可能不是一个 IP 地址,或者是任何形式有效的主机名。可以通过 Mesos 的 WEB 页面来检查从节点的主机名属性。可以通过 REST 访问状态:

curl http://10.134.29.134:5050/master/state.json|python　-mjson.tool

允许 Mesos 任务使用 Mesos-DNS 作为主 DNS 服务,你必须在每个从节点上修改文件/etc/resolv.conf 增加新的 nameserver。10.134.29.134 是 Mesos-DNS 安装节点,应该在每个从节点/etc/resolv.conf 开始加上一行 nameserver 10.134.29.134。

如果启动多个 Mesos-DNS,需要在/etc/resolv.conf 的开始为每一个服务添加 nameserver。这些条目的顺序将确定从节点连接 Mesos-DNS 实例的顺序。你可以通过设置 optionsrotate 在 nameserver 之间选择负载均衡的轮循机制。

/etc/resolv.conf 中其他的 nameserver 设置保持不变。/etc/resolv.conf 文件在主节点中,只需要修改同时做为从节点的机器。

七、Marathon 之负载均衡篇

这一章主要探讨是 Mesos 关于服务发现与应用的负载均衡的解决方案，主要侧重对服务发现与负载均衡进行讲解，需要明白的一点，Mesos 作为两层架构，Marathon 作为 Mesos 的 systemd 服务，服务发现功能只需要向 marathon 提供即可，marathon 启动的 k8s、 Cloud Foundry 都用自身的服务发现功能。下面介绍三种负载均衡方式，原理为，通过 marathon 接口，侦测服务变化，动态更新 haproxy.cfg 配置文件，对 haproxy 服务进行 reload。

7.1 Marathon-Bridge and HAProxy

服务发现的功能实现是为 Dcos 系统中的服务提供便捷的网络通信，它的侧重点在于对服务进行注册，更新，查询等功能，服务 发现的实现方法较多， 主要有 DNS、集中式服务路由、应用内部实现服务注册/服务发现等方式, Dcos 服务发现功能采用的是 Mesos-DNS 策略, 有关 Mesos-DNS 的 具体介绍详见上一篇文章。

通过 Mesos-DNS 的服务发现策略，可以通过辅助脚本利用 Marathon REST API 定时(通过 linux crond 服务)产生 HAProxy 配置文件，通过 diff 生成的 hapxoy 配置文件与已有的 haproxy，来判断是否进行 reload haproxy 操作。

Mesos-slave 默认想 master 提供的端口资源的范围是 31000-32000，当 marathon 启动一个 task 实例时，所在的 mesos-slave 会随意给其绑定一个或多个在其范围内的端口。需要注意的是应用绑定的实际端口(即 mesos-slave 分配给它的 端口)和应用在配置时所指定的形式端口(这是以后直接访问的应用端口)之间的区别，形式端口(即应用端口)是应用运行在 marathon 的一种命名空间， 不是直接绑定的，也就是说其他服务

也可以绑定这样一个端口，只是服务不同而已，它间接的被负载均衡器所使用。

服务发现功能，允许 marathon 上的应用可以通过配置的端口与其他 marathon 应用进行通信，这样做的好处就是，无需知道实际分配的端口是 多少，例如: python 的 wsgi 服务(配置时你指定的是 80)需要跟 mysql(配置时你指定的 327)进行通信，这样你可以直接与 localhost:327 进 行通信即可。

HAProxy 会把请求路由到具体的服务节点上，如果此服务路径不可达，它将继续将路由到下一个服务节点。需要注意的是，目前服务发现功能只支持 marathon 上的应用。

Marathon 附带一个简单的被叫做 haproxy-marathon-bridge 的 shell 脚本以及更高级的 Python 脚本 servicerouter.py(这个脚本在 marathon/bin 下面)。两个脚本都可以将 Marathon 的 REST API 列表中正在运行的任务推送到 HAproxy 的设置文件中，HAproxy 是一个轻量级的 TCP/HTTP 的代理。haproxy- marathon-bridge 提供了一个最小设置功能。 而 servicerouter.py 支持如 SSL 卸载，sticky 连接和虚拟主机的负载均衡的更高级的功能。

负载均衡实现原理就是上述提及的，通过辅助脚本(这里是使用 haproxy-marathon-bridge)利用 Marathon REST API 定时(通过 linux crond 服务)产生 HAProxy 配置文件，通过 diff 生成的 hapxoy 配置文件与已有的 haproxy，来判断是否进行 reload haproxy 操作。

下图描述了在一个集群分别在两个节点安装同一服务，SVC1 和 SVC2，分配配置的应用端口是 1111 和 2222，可以看到实际分配给它们的是 31100 和 31200。

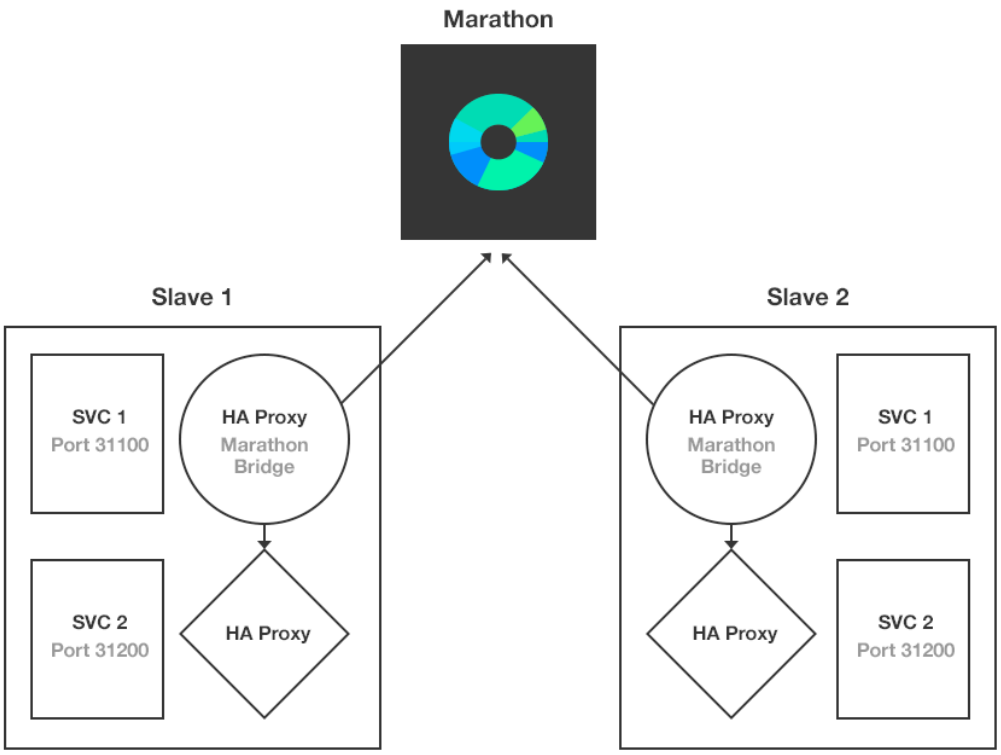

图 7.1

当 slave2 节点上的 SVC2 服务通过 localhost:2222 连接 SVC1 服务时，HAProxy 将把请求转发到第一配置项 SVC1 的 slave1 节点。

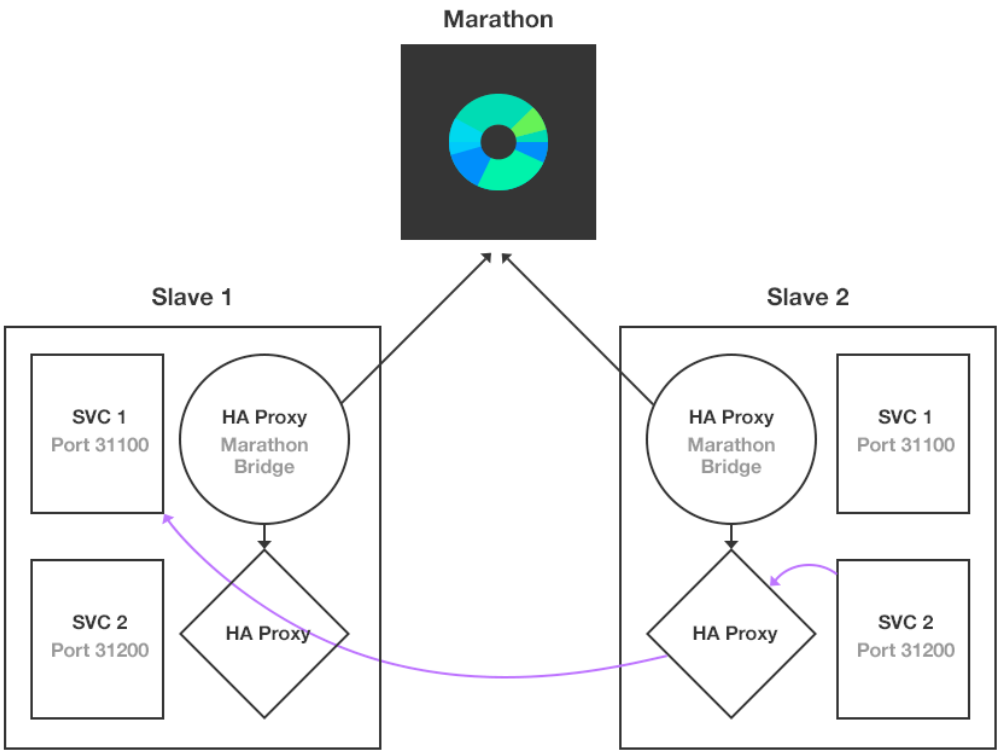

图 7.2

如果 slave1 节点挂了，下一次对 Localhost:2222 的请求，将被转发到 slave2 上。

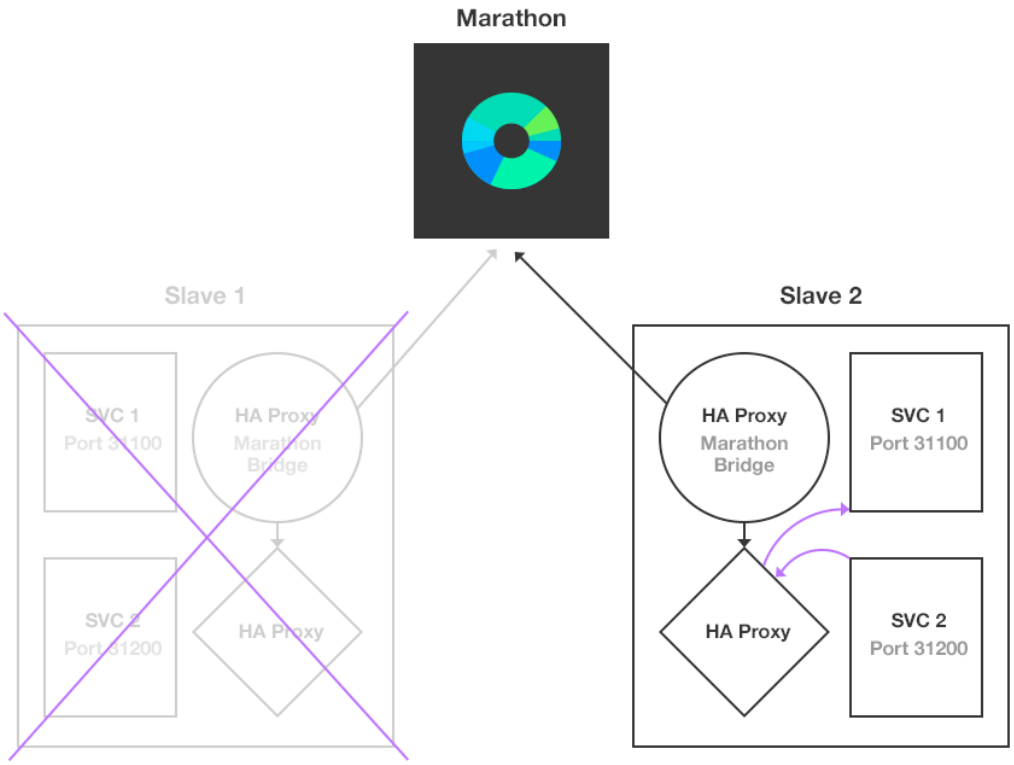

图 7.3

通过 haproxy-marathon-bridge 脚本从 Marathon 生成一个 HAProxy 配置在 leader.mesos:8080 运行:

$ ./bin/haproxy-marathon-bridge leader.mesos:8080 > /etc/haproxy/haproxy.cfg

重新加载 HAProxy 配置而不中断现有的连接:

$ haproxy -f haproxy.cfg -p haproxy.pid -sf $(cat haproxy.pid)

配置脚本并重新加载可以通过 Cron 经常触发来跟踪拓扑变化。如果一个节点在重新加载时消失， HAProxy 的健康检查将抓住它并停止向这个 node 发送 traffic 。为了方便这个设置，haproxy-marathon-bridge 脚本以另一种方式可以调用安装脚本本身，HAProxy 和定时任务每分钟 ping 一次的 Marathon 服务，如果有任何改变将立刻刷新 HAProxy。

$ ./bin/haproxy-marathon-bridge install_haproxy_system leader.mesos:8080

Marathon 需要 ping 的列表存按行存储在 /etc/haproxy-marathon-bridge/marathons

脚本安装在 /usr/local/bin/haproxy-marathon-bridge

-cronjob 安装在/etc/cron.d/haproxy-marathon-bridge 注意需要用 root 来运行。

所提供的只是一个基本的示例脚本。

servicerouter.py

通过 servicerouter.py 脚本从 Marathon 生成一个 HAProxy 配置在 leader.mesos:8080 运行:

$ ./bin/servicerouter.py --marathon http://leader.mesos:8080 --haproxy-config /etc/haproxy/haproxy.cfg

如果有任何变化,将会刷新 haproxy.cfg，这样 HAproxy 将会重新自动加载。

servicerouter.py 有许多额外的功能,像 sticky 会话,HTTP 到 HTTPS 的重定向,SSL 卸载,VHost 支持和模板功能。

7.2 Bamboo and HAProxy

场景：当你在 Mesos 集群上部署的了一系列的微服务，而这些服务能够以 HTTP 方式通过访问特定的 URL 来对外提供服务或者对内进行通信。

- Mesos 集群上通过 Marathon 框架启动应用(服务)，Marathon 通过健康检查(healthcheck)跟踪它们的状态
- Bamboo 通过监听 Marathon event 以更新 HAProxy 配置文件

- HAProxy ACL 规则通过 Bamboo 进行配置，其能够根据请求的特征，如 URL 规则、hostname、HTTP headers，来匹配应用服务。

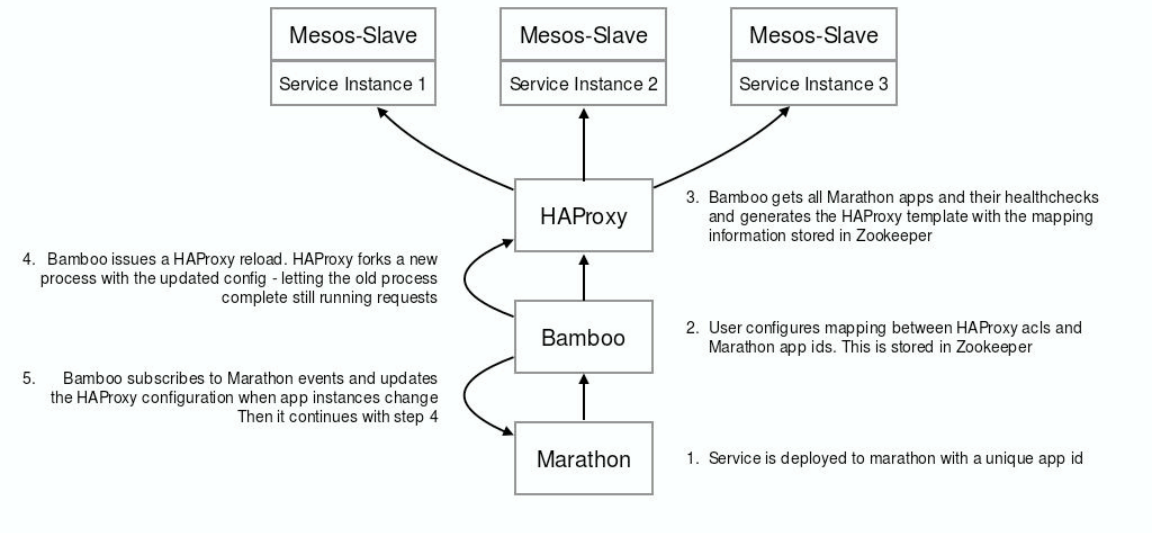

图 7.4

Bamboo 的处理流程跟上述的方案是异曲同工的。

优点:
1. 允许任意 URL 与服务进行对应
2. 允许通过 HTTP Header 与服务进行对应
3. 及时的触发 Marathon event 来促使 HAProxy 进行改变
4. HAProxy heavy lifting

不足:
1. 对于非 HTP 不适用
2. 内部需要有 HAProxy 故障切换机制除非能够实现 SmartStack 架构的服务
3. 内部非流量都邹另外的 hop(HAProxy)

① 安装 HAProxy 和 Bamboo

HAProxy 的安装可以使用如下方式:
apt-get install haproxy

Bamboo 项目地址,你可以通过构建脚本来制作 deb 或者 rpm 的软件包,当然也可以通过 build container 进行构建 deb 软件包

docker build -f Dockerfile-deb -t bamboo-build .
docker run -it -v $(pwd)/output:/output bamboo-build# package ends up as output/bamboo_1.0.0-1_all.deb

需要注意的是，需要修改/var/bamboo/production.json 来修改对应的 Marathon、HAProxy、Zookeeper 的 hostname，然后重启 bamboo，通过 retsart bamboo-server。

② 在 marathon 上部署应用

编辑 ghost.json 文件，填入下述配置：

```
{
  "id": "ghost-0",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "ghost",
      "network": "BRIDGE",
      "portMappings": [{ "containerPort": 2368 }]
    }
  },
  "env": {},
  "instances": 1,
  "cpus": 0.5,
  "mem": 256,
  "healthChecks": [{ "path": "/" }]
}
```

然后使用 curl -X POST -H "Content-Type: application/json" http://marathon.mesos:8080/v2/apps -d@ghost.json 即可部署该应用，可以看到 marathon UI

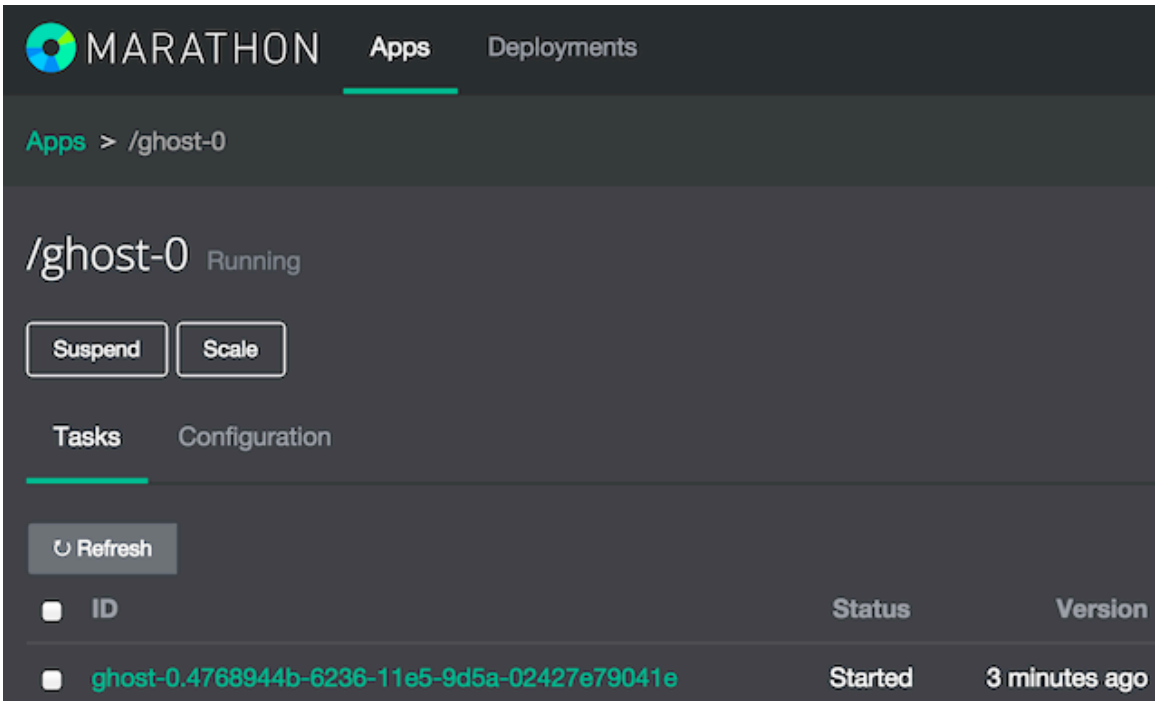

图 7.5

④ Bamboo 配置 rules

可以定义 rules 来告诉 HAProxy 如何去 proxy：

图 7.6

首先，在/etc/hosts 添加一行，这样就可以匹配 Host Header：

ip of HAProxy192.168.99.100 ghost.local

访问 Bamboo UI，通常是 http://haproxy:8000，然后添加对应的 name：ghost-0，如下图：

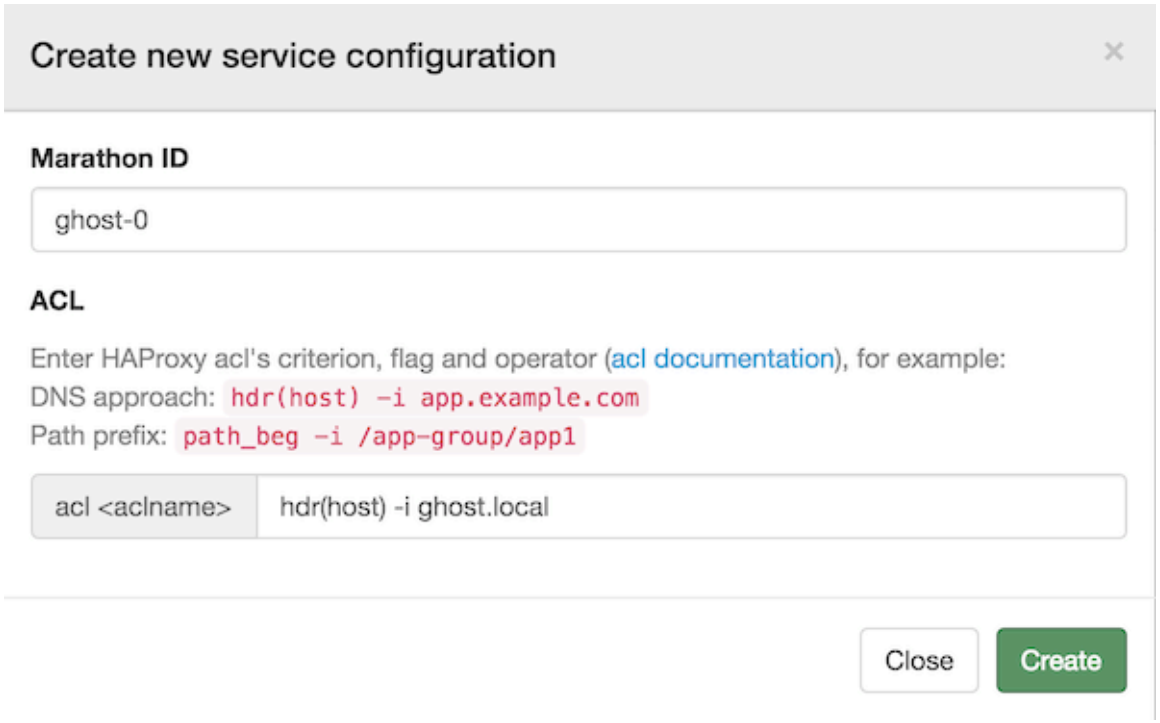

图 7.6

查看一下是否添加成功：

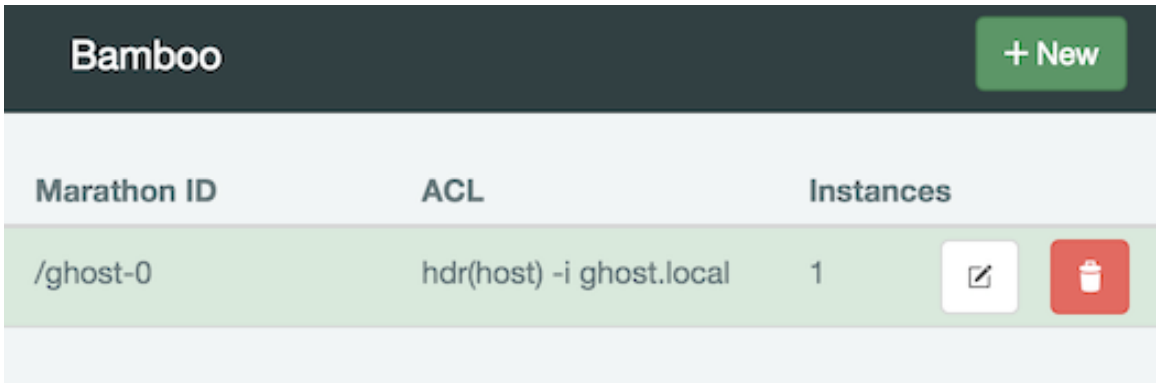

图 7.7

八、Marathon 之微服务篇

关于 CRM 系统、OA 系统、能力开发平台的迁移，先对其进行微服务化改造，目前正在调研当中。