Marathon

Marathon（马拉松）是一个全新的框架，它将[Mesos](http://www.oschina.net/p/apache+mesos)变成一个更有活力的工具，进而可以在单一的集群上运行不同的应用程序。

它的设计宗旨就是让用户在同一组服务器之上，更智能地运行多种应用程序和服务——[Hadoop](http://www.oschina.net/p/hadoop)、[Storm](http://www.oschina.net/p/storm)，甚至一个标准的Web应用。Marathon出自于一家初创公司 [Mesosphere](http://www.mesosphere.io/)之手，这家公司主要就是想构建一个数据中心操作系统，不过这个系统是运行在[Apache Mesos](http://www.oschina.net/p/apache+mesos)集群管理软件之上，这也是 [Twitter基础设施的重要组成部分](https://blog.twitter.com/2013/mesos-graduates-from-apache-incubation)。该公司的联合创始人是前Airbnb的工程师Florian Leibert（也曾在Twitter工作过）和Tobias Knaup。

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。

特性：

* HA -- run any number of Marathon schedulers, but only one gets elected as leader; if you access a non-leader, your request gets proxied to the current leader
* [Constraints](https://github.com/mesosphere/marathon/wiki/Constraints) - e.g., only one instance of an application per rack, node, etc.
* [Service Discovery & Load Balancing](https://github.com/mesosphere/marathon/wiki/Service-Discovery-&-Load-Balancing) via HAProxy or the events API (see below).
* [Health Checks](https://github.com/mesosphere/marathon/wiki/Health-Checks): check your application's health via HTTP or TCP checks.
* [Event Subscription](https://github.com/mesosphere/marathon/wiki/Event-Bus) lets you supply an HTTP endpoint to receive notifications, for example to integrate with an external load balancer.
* Web UI
* JSON/REST API for easy integration and scriptability
* Basic Auth and SSL
* Metrics: available at/metricsin JSON format

# 1.安装和使用

**1）下载安装包**

$curl -O http://downloads.mesosphere.com/marathon/v1.1.1/marathon-1.1.1.tgz

$tar xzf marathon-1.1.1.tgz

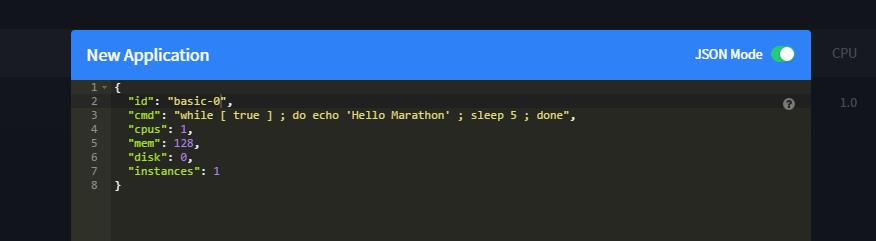
**2）启动**

$./bin/start --master zk://zk1.foo.bar:2181,zk2.foo.bar:2181/mesos --zk zk://zk1.foo.bar:2181,zk2.foo.bar:2181/marathon

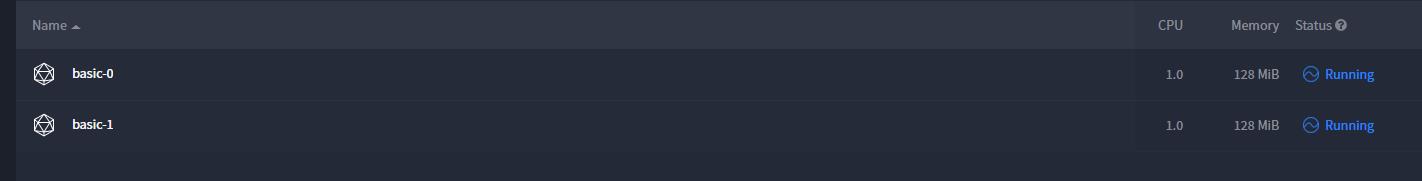
注：marathon在启动后，要依赖MESOS\_NATIVE\_JAVA\_LIBRARY，运行时会搜索/usr/lib和/usr/local/lib目录，例如libmesos-0.28.0.so

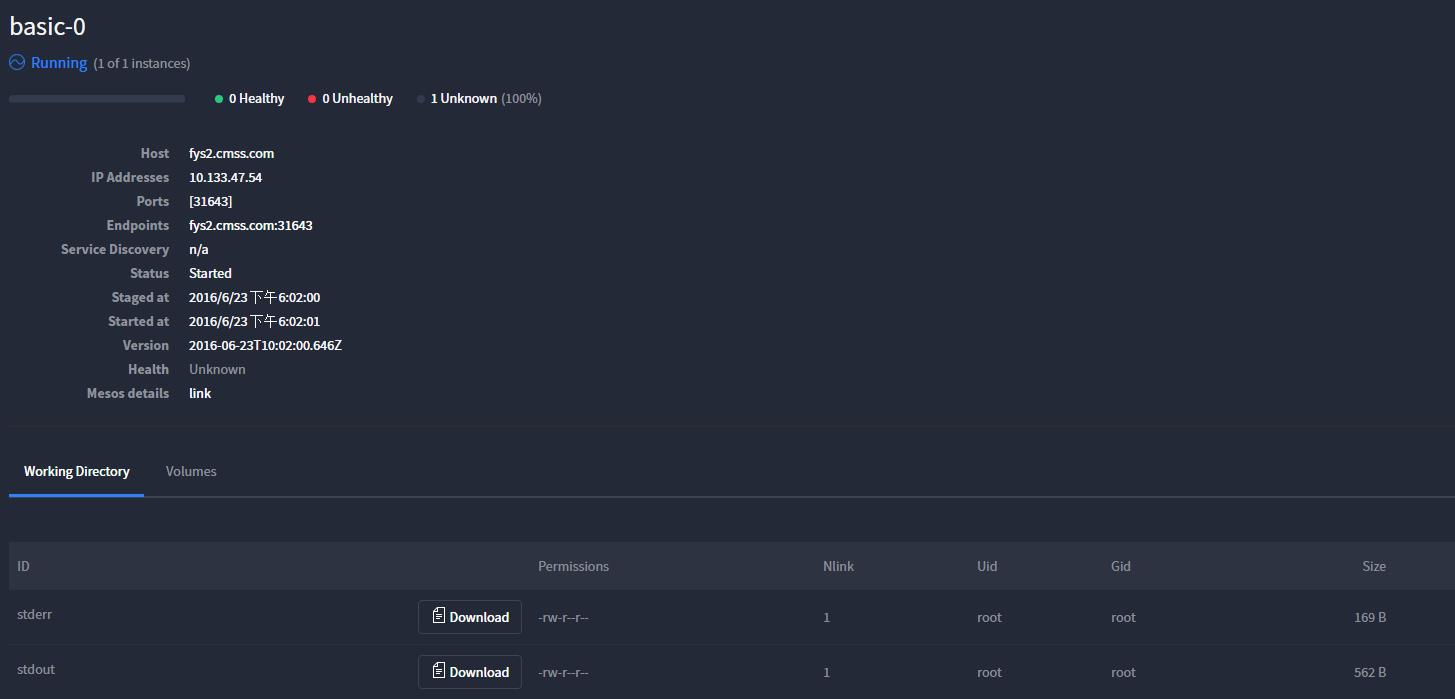
**3）提交程序（Json格式或者Web UI）**

以Json格式如下：



创建后，如下图：





可以通过stderr和stdout查看日志

参考文献：

http://mesosphere.github.io/marathon/docs/application-basics.html

**4）测试使用resource的应用**

稍复杂的应用依赖一定的资源：files或者 archives of files。为了对资源分配进行管理，Marathon有资源的URIs的概念，使用Mesos Fetcher进行资源的获取。下面是一个简单的例子：

{

"id": "basic-1",

"cmd": "`chmod u+x cool-script.sh && ./cool-script.sh`",

"cpus": 0.1,

"mem": 10.0,

"instances": 1,

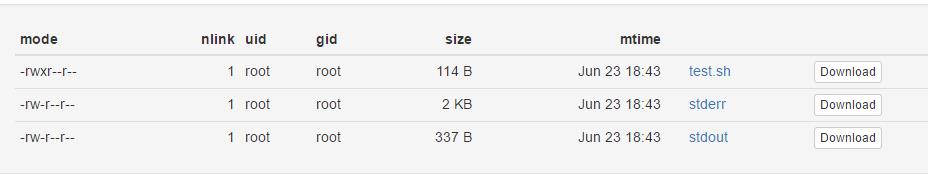
"uris": [

"https://example.com/app/cool-script.sh"

]

}

mesos-fetcher会从uris中下载cool-script.sh，在sandbox中可以看到下载的资源，如下所示：



**5) Tomcat应用**

JSON内容为：

{

"id": "tomcat",

"cmd": "mv \*.war apache-tomcat-\*/webapps && cd apache-tomcat-\* && sed \"s/8080/$PORT/g\" < ./conf/server.xml > ./conf/server-mesos.xml &&chmod 755 ./bin/catalina.sh && ./bin/catalina.sh run -config ./conf/server-mesos.xml",

"mem": 512,

"cpus": 1.0,

"instances": 1,

"uris": [

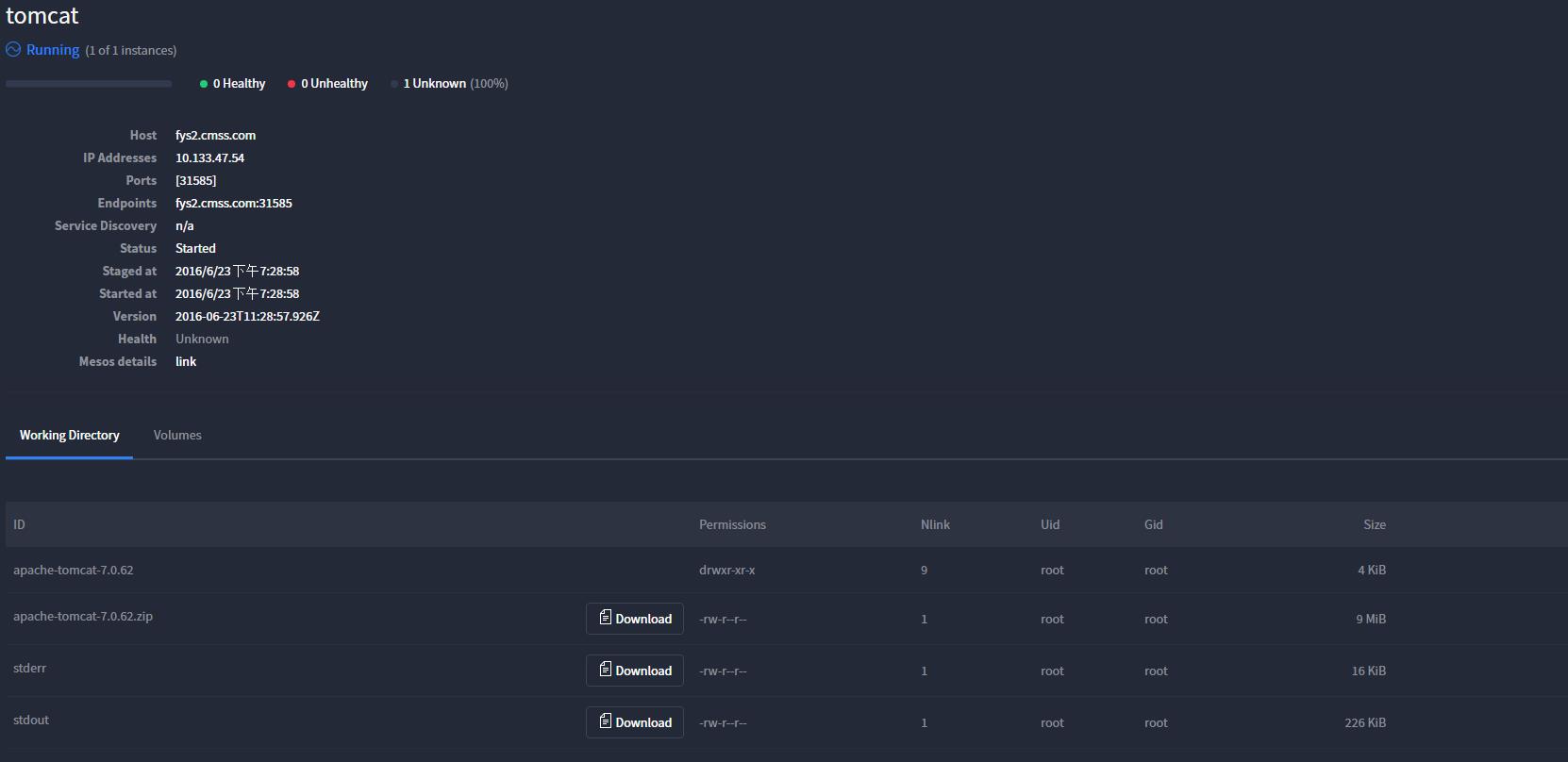
"http://10.133.17.27/software/tomcat/tarball/apache-tomcat-7.0.62.zip",

"http://10.133.17.27/tmp/test.war"

]

}

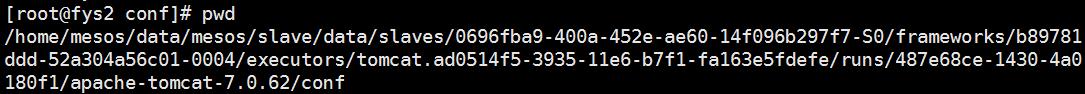
执行结果：



启动信息如下：



tomcat安装目录:



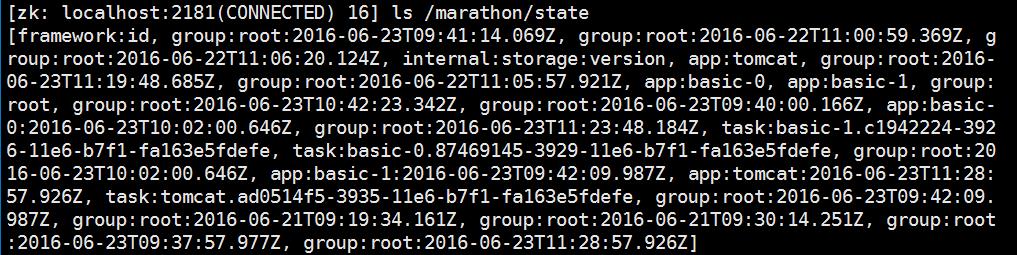
ZK目录：

|—/marathon

|— /leader

|— member\_000xx3 value{fys2.cmss.com:8080}，多marathon时，选择leader

|— /state //该目录用于记录marathon运行的framework及数据，如下所示



6）Marathon REST API

创建应用的接口：

curl -X POST http://10.141.141.10:8080/v2/apps -d @basic.json -H "Content-type: application/json"

# 2.Application

## 2.1 application Basics

应用的定义以Json的格式，定义的字段如下：

* id，
* cmd
* cpus
* mem
* instances
* uris ，资源的URIS，用于获取资源
* container ，定义的Container，Docker,Linux

type

docker

image

network

* groups ，定义应用的集群组件

## 2.2 定义

product

Groups

Application

Artifacts

Dependencies

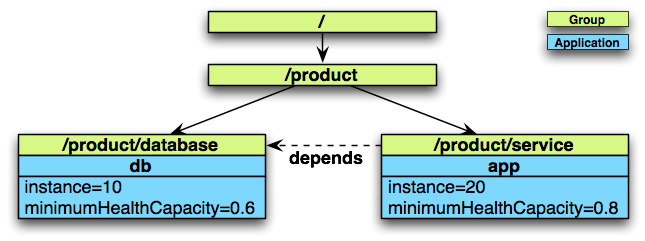
## 2.3 Application的部署

Application Deployments: 应用的配置或者groups的变化都会被转化为Deployments，Deployments是一系列Action的集合，包括一个或者多个应用的以下操作：

* Start/Stop
* Upgrade
* Scale

Deployments需要花费一定时间成功执行后才生效。在同一时间可以执行多种Deployments，但是一个应用中的Deployment需要依次执行，一个应用只能同时执行一个Deployments。

Deployments Order，如果应用无依赖，应用的部署是无序的，如果应用之间相互依赖，Deployment Actions需要按照特定的顺序执行，如下图所示：



例如应用App依赖应用db：

* Starting， db和apps在系统中启动，db在app之前启动
* Stopping，db和apps从系统中移除，app先移除
* Upgrade，Rolling Restart（先启动新Version，然后再停止old version）
* Scaling，db和apps扩容，db先scaled，然后app

Rest API:

/v2/deployments

传递参数：

1）affectedApps，该deployments影响的deployments

2）steps，执行该deployments的步骤

3）currentStep，执行的step，每个stop都包括多actions，包括

* ResolveArtifacts，解析一个应用的所有的artifacts，并保存到artifact store中
* StartApplication，启动特定的application
* StopApplication，停止特定的application
* ScalApplication，Scale特定的application
* RestartApplication，重启应用
* KillAllOldTasksOf，停止特定应用的剩余Tasks

# 3.Marathon特性

## 3.1 Constraints

Constraints Control可以设置apps运行的位置，来实现容错或者本地性（在同一个节点上运行所有该应用的tasks）。Constraints有三个部分：a field name，an operator和an optional parameter。field为agent node的hostname或者agent node的性质。

* Fileds

1）hostname: agent node 的主机名

2）attribute: 如果不是hostname,那么作为agent node的attribute。为agent node打标签

* Operators

1）UNIQUE

$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/apps -d '{

"id": "sleep-unique",

"cmd": "sleep 60",

"instances": 3,

"constraints": [["hostname", "UNIQUE"]]

}'

在一个主机上运行唯一的Task

2）CLUSTER

将apps运行在运行具有特定attribute的agent nodes。例如运行在相同的rack上

$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/apps -d '{

"id": "sleep-cluster",

"cmd": "sleep 60",

"instances": 3,

"constraints": [["rack\_id", "CLUSTER", "rack-1"]]

}'

3）GROUP\_BY

将tasks分布到不同racks中或者datacenters（high availability）

$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/apps -d '{

"id": "sleep-group-by",

"cmd": "sleep 60",

"instances": 3,

"constraints": [["rack\_id", "GROUP\_BY"]]

}'

//分布到不同的rack\_id中

4）LIKE/UNLIKE

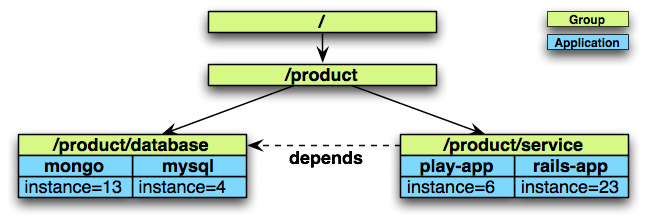
匹配四则表达式

"constraints": [["rack\_id", "LIKE", "rack-[1-3]"]]

"constraints": [["rack\_id", "UNLIKE", "rack-[7-9]"]]

## 3.2 Application Groups

Applications被组织成n叉树，Groups作为branches及applications作为leaves。Application Groups被切分成多个Applications，组成多个可管理的集合。



定义描述：

{

"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", ... }

]

}

]}

Applications之间有依赖，例如Play Application需要运行一个database。如果在Applciation描述中定义依赖关系，那么marathon会在starting、stop、和upgrading操作中依赖正确的顺序。

Rest API:PUT /v2/groups/product HTTP/1.1

Content-Length: 21

Host: localhost:8080

User-Agent: HTTPie/0.7.2

{ "scaleBy": 2 }

实例如下所示，json如下(参考examples/group.json)

{

"id": "department-a",

"apps": [

{

"id": "service-common",

"cpus": 0.1,

"mem": 32,

"ports": [0],

"cmd": "python -m SimpleHTTPServer $PORT0",

"instances": 2

}

],

"groups": [

{

"id" : "product-a",

"apps":[

{

"id": "service-a",

"cpus": 0.1,

"mem": 32,

"ports": [0],

"cmd": "python -m SimpleHTTPServer $PORT0",

"instances": 2,

"dependencies": [

"/department-b/service-common"

]

},

{

"id": "service-b",

"cpus": 0.1,

"mem": 32,

"ports": [0],

"cmd": "python -m SimpleHTTPServer $PORT0",

"instances": 2,

"dependencies": [

"/department-b/product-a/service-a"

]

}

]

}

]

}

## 3.3 Artifact Store

在分布式系统中部署应用，需要将资源放到集群主机都可以访问的位置。在Marathon中，称为artifact store。

* Artifact Store Backend

Marathon支持不同的存储系统，可以通过command line进行配置。

例如： --artifact\_store hdfs://localhost:54310/path/to/store，将HDFS作为artifact storage backend。

* Artifact REST endpoint

用于访问和操作Artifact Store中的Artifact。创建的Artifact的URL用于Application定义。

1）Upload（上传artifacts）

POST : /v2/artifacts

/v2/artifacts/special/file/name.txt

2）获取Artifact Store中的Artifact

路径为相对路径

GET : /v2/artifacts/special/file/name.txt

3）从Artifact Store中删除Artifact

DELETE: /v2/artifacts/special/file/name.txt

* Automatic Artifact Storing

AppDefinition获取URIs序列（在每个实例启动时进行获取）。处理的步骤如下:

1) 下载该URL

2) Asset Store中存储byte stream

3) 将asset Store url加到AppDefinition的URIS list

4) 将URL从storeURIs移除

使用automatic artifact resolution创建application definition，启动marathon时通过指定artifact-store，如下所示：

--artifact\_store hdfs://10.133.47.5:8020/spark/

Request :

POST /v2/apps HTTP/1.1

Accept: application/json

Accept-Encoding: gzip, deflate

Content-Length: 197

Content-Type: application/json; charset=utf-8

Host: localhost:8080

User-Agent: HTTPie/0.8.0

{

"cmd": "python toggle.py $PORT0",

"cpus": 0.2,

"id": "app",

"instances": 2,

"mem": 32,

"ports": [

0

],

"storeUrls": [

"http://downloads.mesosphere.com/misc/toggle.tgz"

]

}

Response:

HTTP/1.1 201 Created

Content-Type: application/json

Location: http://localhost:8080/v2/apps/mongo2

Server: Jetty(8.1.11.v20130520)

Transfer-Encoding: chunked

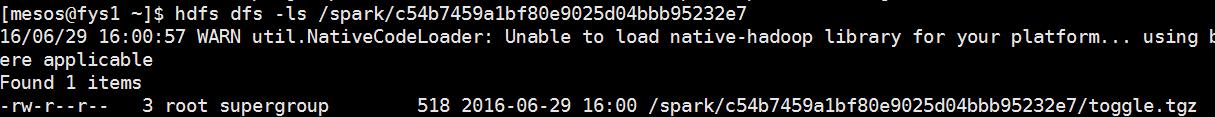
{

"deploymentId": "910ae97f-3f3d-4fdb-a9a1-9d72f0ae8e49",

"version": "2014-07-22T13:25:52.319Z"

}

当app部署后，所有的storeUrls存储到artifact store。toggle.tgz已经上传到了。通过hdfs命令查看artifact-store如下：



## 3.4 Event Bus

Marathon 内部实现了Event Bus，捕获所有的API的Requests，并分发events（订阅者）。可以基于event bus实现根据marathon状态的load balancer,compile statics等功能。

Event Bus具有两个APIs：

* event stream，例如/v2/events
* callback endpoint，以JSON格式

建议使用event stream而不是callback endpoint，easier to set up、Delivery、按顺序传递

* 通过callback endpoint，订阅event，命令行如下：

$./bin/start --master ...

--event\_subscriber http\_callback

--http\_endpoints http://host1/foo,http://host2/bar

host1和host2会接收到事件

* Event Types

API Request

Status Update //task的状态变化

Framework Message

Event Subscription

Health Checks

Deployments

参考文献：

https://mesosphere.github.io/marathon/docs/generated/api.html

## 3.5 Health Checks

Health Check支持基于Tcp和Http协议的Health监测，它的大体逻辑是Marathon周期性的扫描应用的监控检查路径，如果相应时间连续超过n次超出阈值，那么Marathon会直接重启这个应用。Health Check用于监控application task的运行状态

* 默认的health check，TASK\_RUNNING => healthy
* 可以通过REST API增加health checks，持久化到application definition中

Health Check用于返回HTTP response code[200,399]，而且在timeoutseconds时间内接收响应数据。超过maxConsecutiveFailures允许的失败次数，将该tasks kill掉

通过/bin/sh -c ""运行healthcheck命令：

{

"protocol": "COMMAND",

"command": { "value": "/bin/bash -c \\\"</dev/tcp/$HOST/$PORT0\\\"" }

}

**Health Checks Options**

1）protocol，默认HTTP，对于HTTP/TCP，Marathon通过命令测试可达性。COMMAND health checks在agent中corresponding task来实现health checks。

2）gracePerioldSeconds，默认300，执行直至healthy

3）intervalSeconds，默认60，health checks的间隔

4）maxConsecutiveFailures，默认3，health checks failure的数目。如果value设置为0，那么health check失败即执行tasks

5）timeoutSeconds

对于TCP/HTTP health checks，port或者portIndex，

1）portIndex

在apps中的用于health request中的ports或者portDefinitions，例如[0,0,0]

2）port

用于health requests的port number

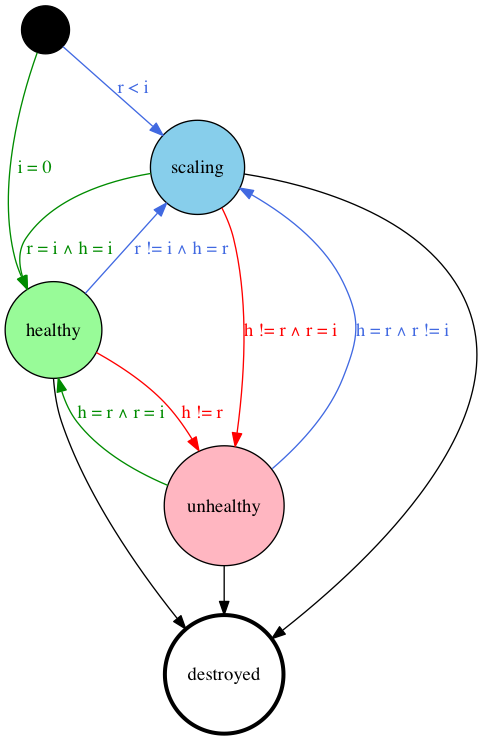
**HTTP health checks**

1）path，默认/，endpoint的路径，例如/path/to/health

2）ignoreHttp1xx，默认false，忽略HTTP status codes 100-199.

**Health Lifecycle**

Application的health lifecycle被表示成状态机，如下图所示：



i，requested instances的数目

r，running instances的数目

h，healthy instances的数目

**Examples:**

**Http health checks**

{

"id": "toggle",

"cmd": "python toggle.py $PORT0",

"cpus": 0.2,

"disk": 0.0,

"healthChecks": [

{

"protocol": "HTTP",

"path": "/health",

"portIndex": 0,

"gracePeriodSeconds": 5,

"intervalSeconds": 10,

"timeoutSeconds": 10,

"maxConsecutiveFailures": 3

}

],

"instances": 2,

"mem": 32.0,

"ports": [0],

"uris": ["http://downloads.mesosphere.com/misc/toggle.tgz"]

}

TCP health checks

{

"id": "toggle",

"cmd": "python toggle.py $PORT0",

"cpus": 0.2,

"disk": 0.0,

"healthChecks": [

{

"protocol": "TCP",

"portIndex": 0,

"gracePeriodSeconds": 5,

"intervalSeconds": 10,

"timeoutSeconds": 10,

"maxConsecutiveFailures": 3

}

],

"instances": 2,

"mem": 32.0,

"ports": [0],

"uris": ["http://downloads.mesosphere.com/misc/toggle.tgz"]}

**Command health checks**

{

"id": "toggle",

"cmd": "python toggle.py $PORT0",

"cpus": 0.2,

"disk": 0.0,

"healthChecks": [

{

"protocol": "COMMAND",

"command": { "value": "curl -f http://$HOST:$PORT0/" },

"gracePeriodSeconds": 5,

"intervalSeconds": 10,

"timeoutSeconds": 10,

"maxConsecutiveFailures": 3

}

],

"instances": 2,

"mem": 32.0,

"ports": [0],

"uris": ["http://downloads.mesosphere.com/misc/toggle.tgz"]

}

实例：创建tomcat，定义health-check

{

"id": "tomcat",

"mem": 512,

"cpus": 1.0,

"instances": 1,

"container": {

"type": "DOCKER",

"docker": {

"image": "tomcat:8.0",

"network": "BRIDGE",

"portMappings": [

{ "containerPort": 8080, "hostPort": 0, "servicePort": 10000, "protocol": "tcp" }

]

}

},

"healthChecks": [

{

"protocol": "HTTP",

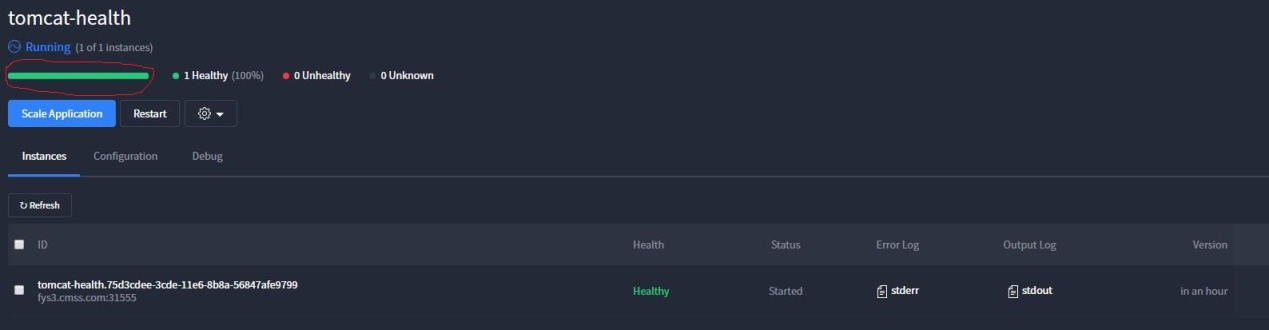
"portIndex": 0,

"path": "/"

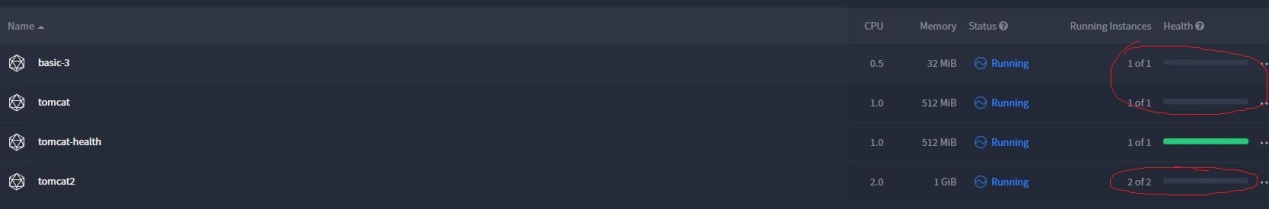
}

]}

Health-Checks检测端口是否运行正常，结果如下所示：



不添加health-checks模块，则如下图server所示，显示未知：



## 3.6 Stateful Applications

**1) Local Persistent Volumes**

当停止或者relaunched后，应用会丢失其state。例如Mysql，保存数据到本地磁盘。Local Volumes保证了tasks重启后不丢失其状态（数据），但是这些数据（资源）保存到tasks初始启动的节点上。

Marathon在application创建时预留disk资源。

提供本地存储资源的好处包括：

1）有状态的服务保存所有资源，保证在相同的节点上重启任务时使用资源

2）不需要指定task，启动在原节点上

3）使用constraints来指定distributed logic

4）Marathon提供定位或者destroy无用的本地数据

配置项: //需要磁盘控制

{

"containerPath": "data", //数据读取目录，

"mode": "RW",

"persistent": {

"size": 10

}

}

**2) External Persistent Volumes**

使用第三方数据存储，例如EBS作为数据持久化，可以应用有更好的容错性，而且在另外一台主机上启动，保证数据的不丢失。

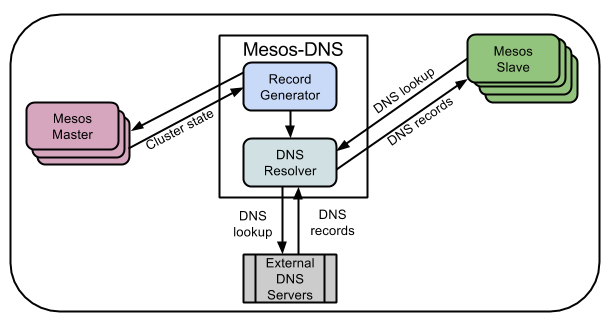
// DC/OS

## 3.7 Service Discovery & Load Balancing

服务创建成功后，需要为相同集群的应用或者集群外客户端提供访问途径。当前，有Mesos-DNS、Marathon-lb及haproxy-marathon-bridge三种。

### 3.7.1 Mesos-DNS

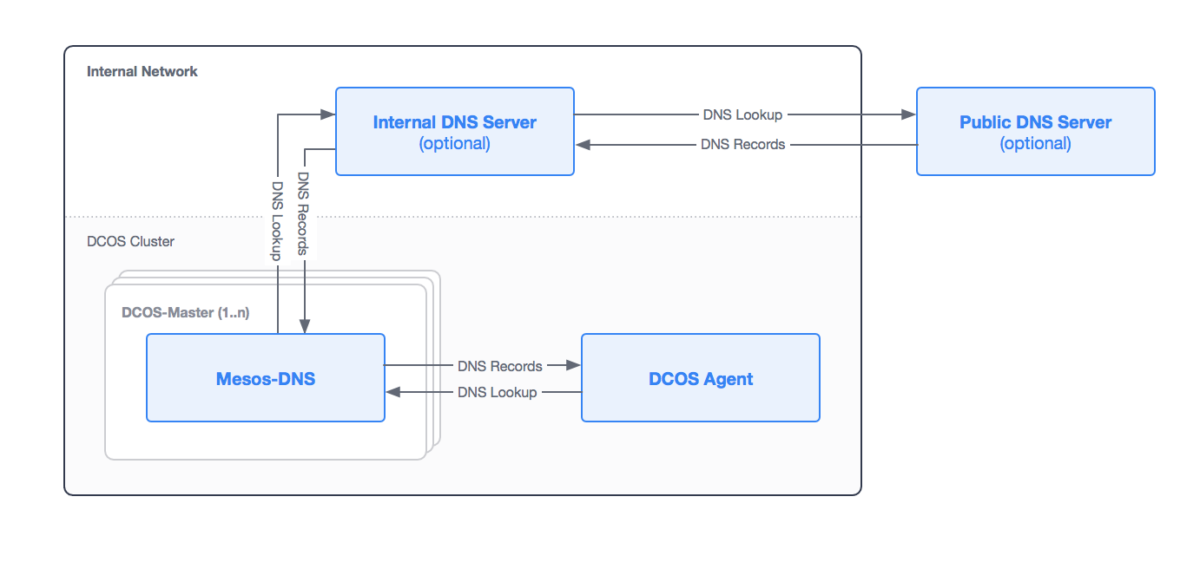
Mesos-DNS是无状态的，其支持Restful API，通过下图可以大致看出Mesos-DNS通过监听Mesos-master来发现服务地址生成DNS解析（并同步到外部的DNS服务器），内部的服务发现可以直接通过内部的DNS解析来保证：



通过DNS实现服务发现，在服务创建后，Mesos-DNS为每个mesos task（包括marathon 实例）生成一条SRV记录和A记录。访问时，将这些记录转换为IP地址和端口。Mesos-DNS适用以下几种情况：

1. 在Mesos中多种framework（不仅仅Marathon）执行程序
2. 使用IP Per Container的方案（Project Calico）启动程序
3. 使用Marathon中创建随机主机和端口的分配方式来创建应用

架构如下所示：



使用Marathon创建一个tomcat 应用，如下所示：

{

"id": "tomcat",

"cmd": "mv \*.war apache-tomcat-\*/webapps && cd apache-tomcat-\* && sed \"s/8080/$PORT/g\" < ./conf/server.xml > ./conf/server-mesos.xml &&chmod 755 ./bin/catalina.sh && ./bin/catalina.sh run -config ./conf/server-mesos.xml",

"mem": 512,

"cpus": 1.0,

"instances": 1,

"uris": [

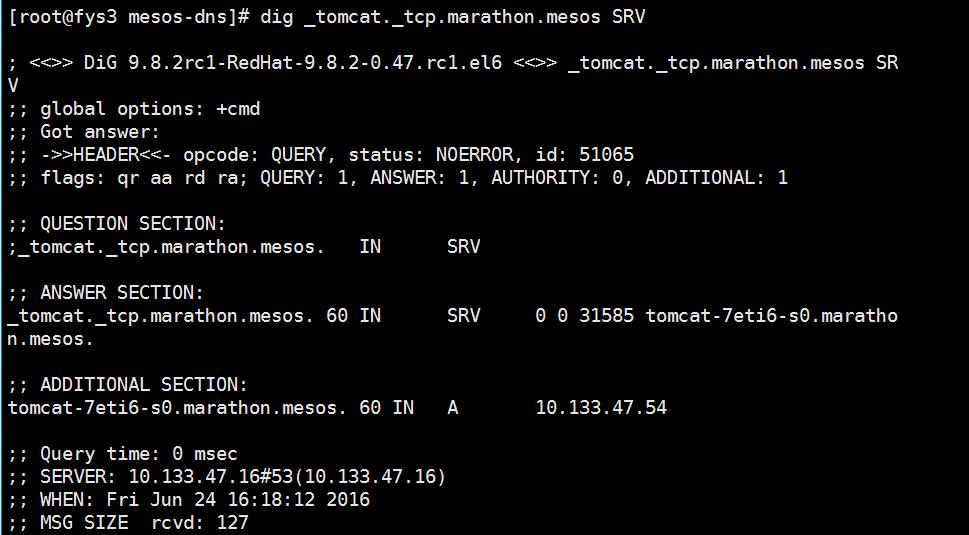
"http://10.133.17.27/software/tomcat/tarball/apache-tomcat-7.0.62.zip",

"http://10.133.17.27/tmp/test.war"

]

}

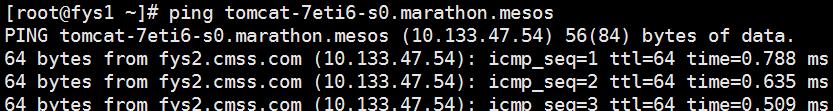
在DNS服务器中，生成的SRV记录如下：



测试：

ping tomcat-7eti6-s0.marathon.mesos

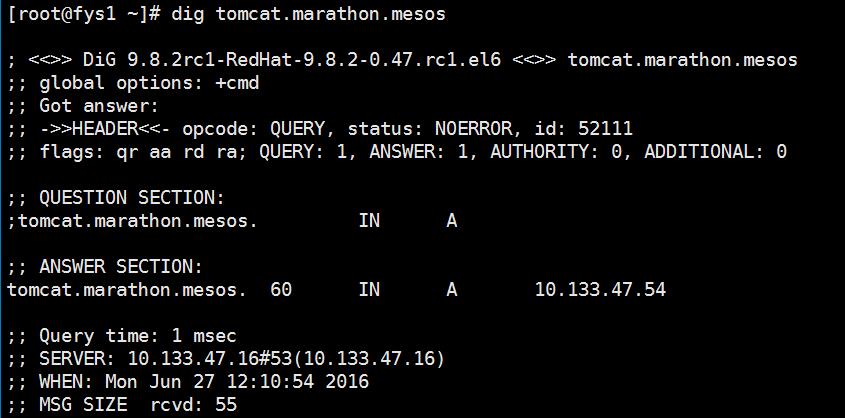
结果如下:



HTTP 查看结果如下：



A记录如下：



即tomcat启动的主机，如上所示，tomcat.marathon.mesos启动在节点10.133.47.54上。

### 3.7.2 Marathon-lb

另外一种解决方案是在集群的每个主机上运行TCP/HTTP代理，然后将localhost上的static service port静态转换成Marathon app instances上随机分配的host/port。这样客户端仅需要访问well-known defined sevice port，而不需要知道服务发现的细节。这种方式可以满足通过Marathon来创建应用的服务发现。

Marathon-lb是基于Dockerized的应用，包括使用Marathon REST API生成HAProxy 配置；基于load balancing的VHost的两种方案。

注:HAProxy-Marathon-bridge，使用Marathon REST API生成HAProxy的脚本。Marathon-lb的简化

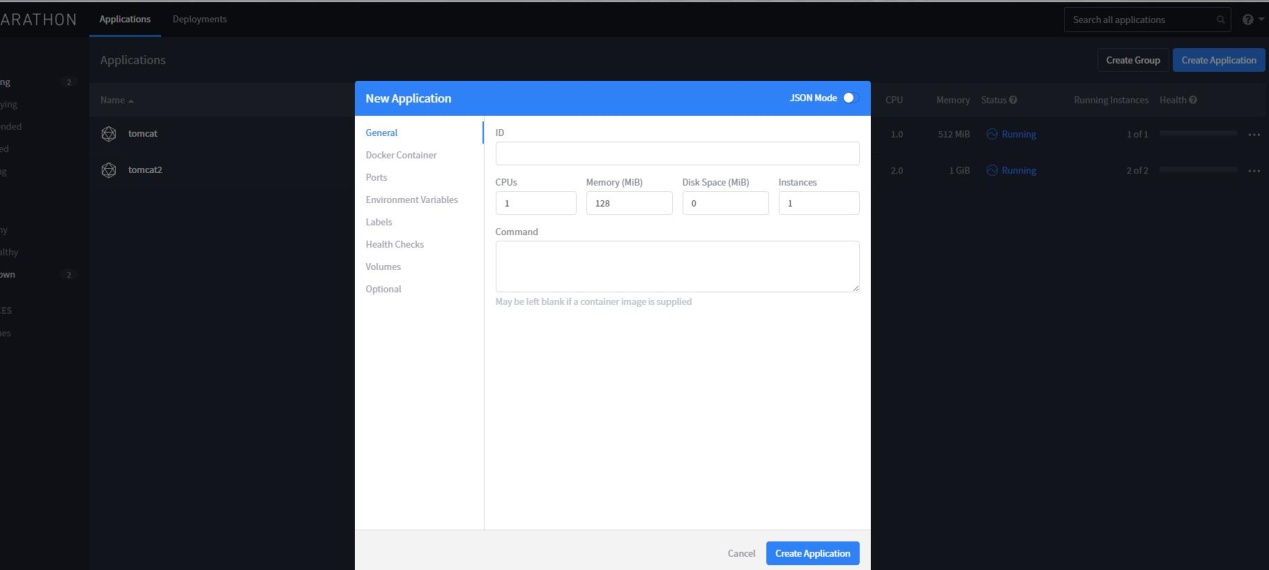
## 3.8 Auto-Scaling

marathon-autoscale

## 3.9 Marathon App Deploy

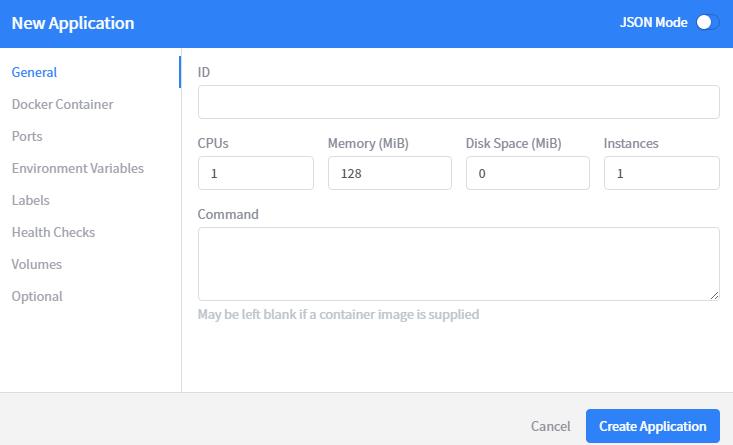
**1) Web UI**

应用创建页面如下所示



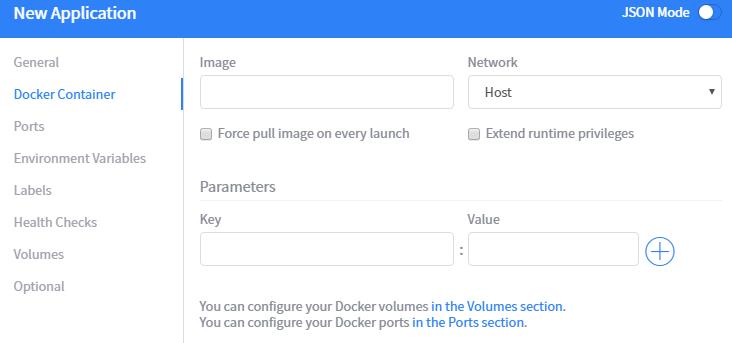
步骤如下：

1）设置Genernal配置

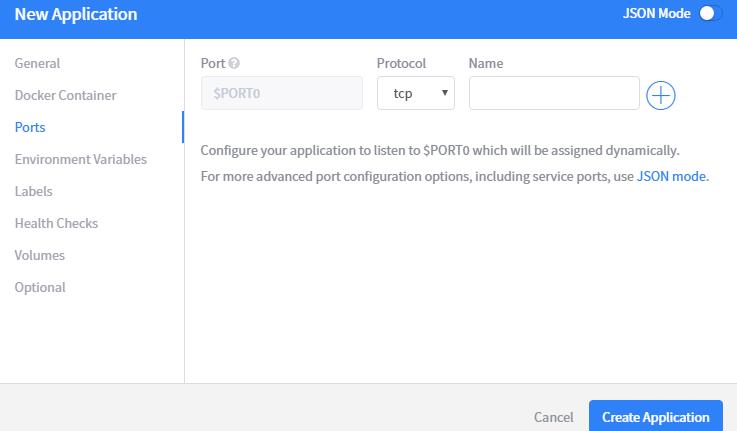


设置唯一标识符，指定消耗资源及container中执行的命令

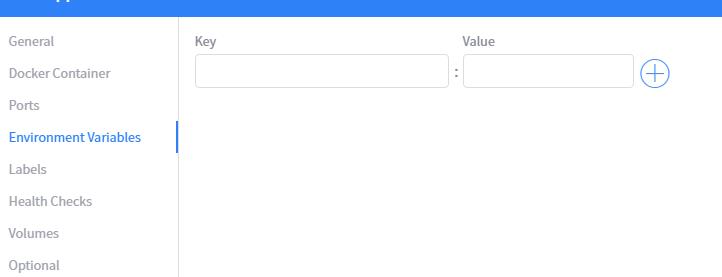
2）是否使用docker container



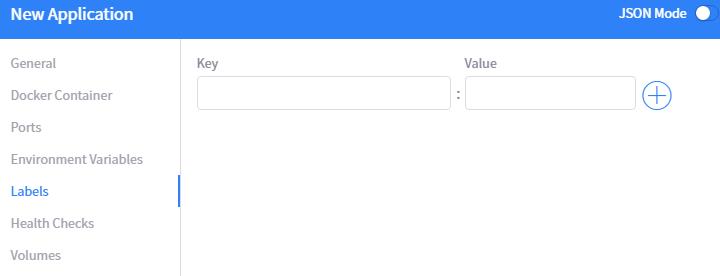
3）设置端口



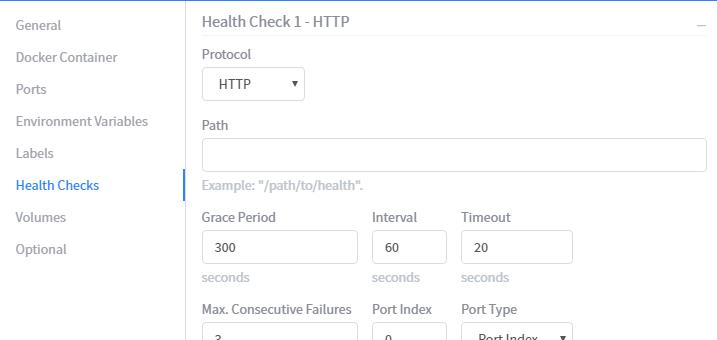
4）增加参数



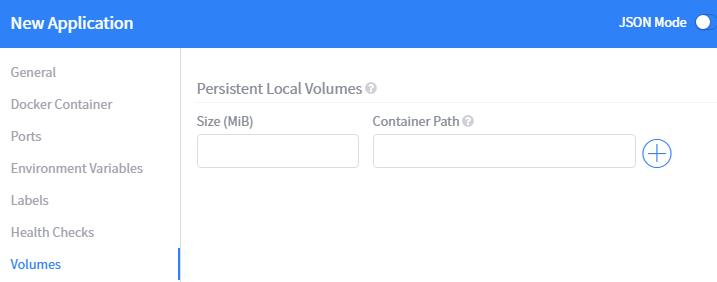
5）设置label



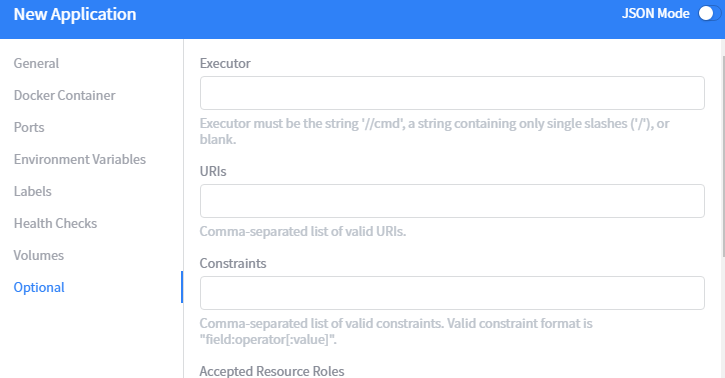
6）设置health Checks

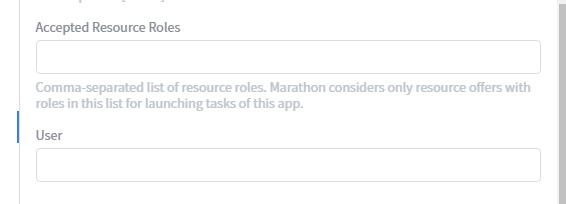


7）设置存储空间

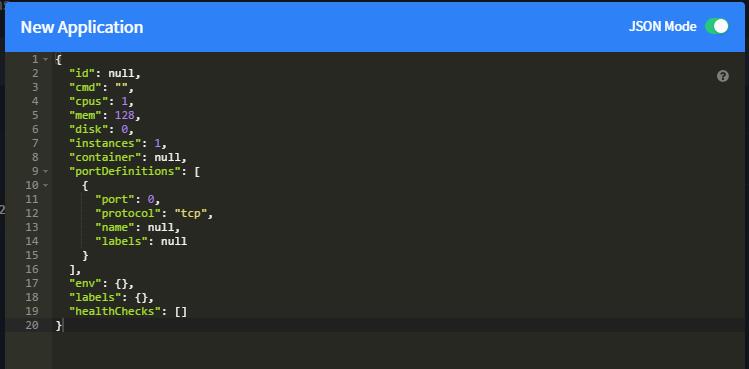


8）设置可选项





另外一种，是通过json字符串，如下所示：



和上面通过页面指定是相同的。

在Web UI上可以看到展示的启动app信息及对app的简单操作

**2) REST API**  //要与slider进行对比，提供必要的rest api

参考链接：

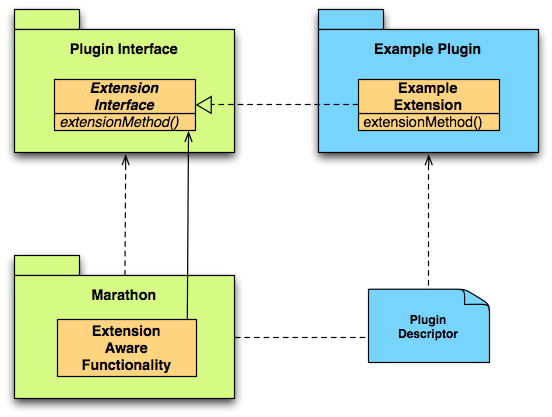
https://mesosphere.github.io/marathon/docs/generated/api.html

不再一一介绍

**3) Command**

## 3.10 Plugins

Maraton 0.16后的版本才有marathon plugin，如下所示：



通过plugins来扩展Marathon的功能

## 3.11 Authorization and Access Control

如果在DC/OS中使用Marathon，可以在native Marathon instances中创建ACL groups来实现细粒度访问。Advanced ACL groups提供多租户，控制应用的定制访问，例如只读。

ACLs可用于组和用户上，分成三种（DC/OS）策略:

|  |  |  |
| --- | --- | --- |
| Category | Base ACL | Discussion |
| super user | dcos:superuser |  |
| Admin Router | docos:admirouter:\* |  |
| marathon | Docos:service:marathon:\* |  |

# 4.Mesos Docker

安装docker不再介绍，在启动mesos slave时要通过---containerizer=docker,mesos指定mesos可以使用docker executor

通过marathon web ui创建应用，json如下：

{

"id": "basic-3",

"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 }

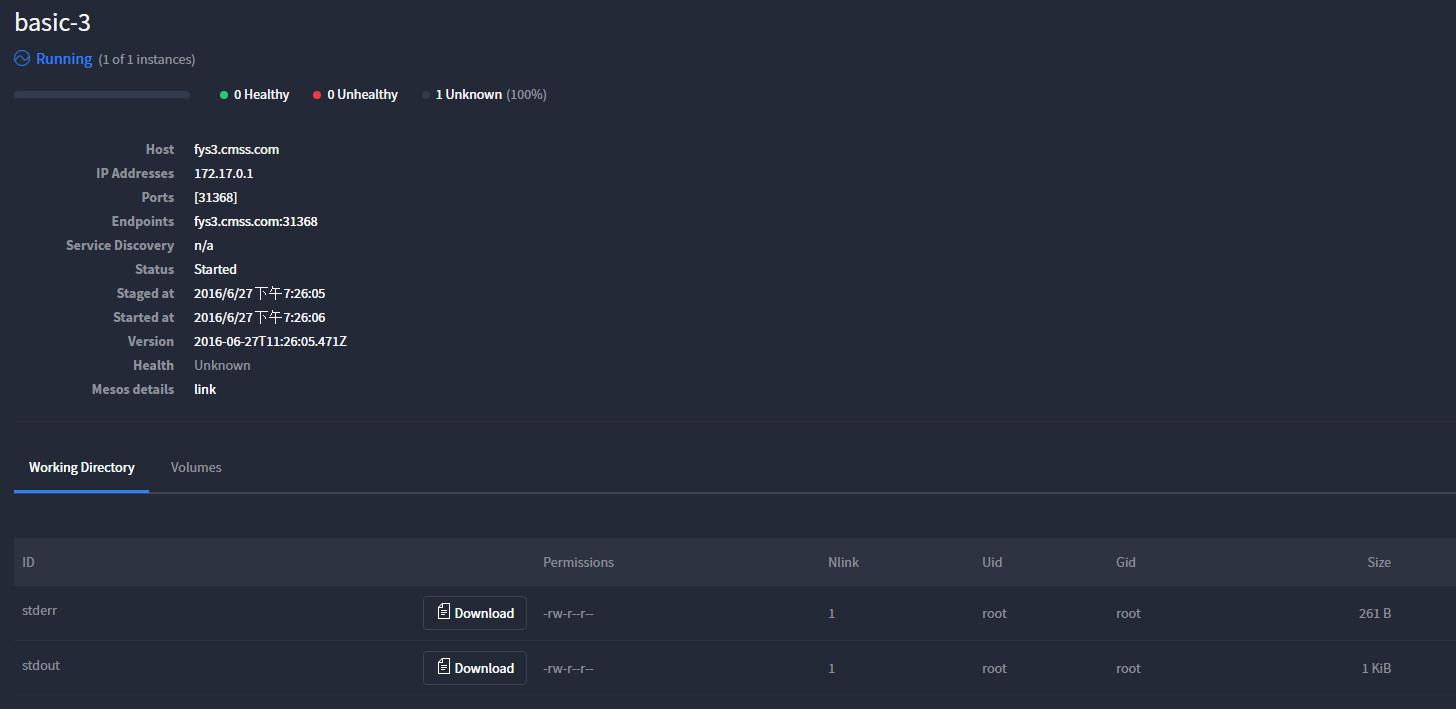
]

}

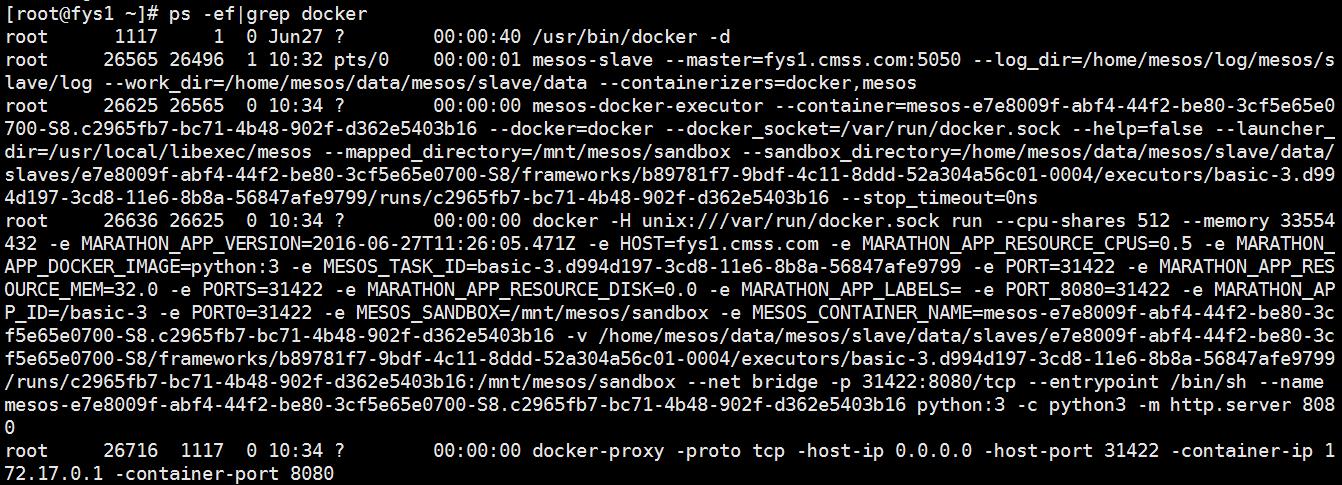
}

}

如下所示：



启动命令：



可以看到最终调用的是docker命令启动服务，如下所示：

C:\Users\yshe\Documents\Fetion\1345511229\temp\99a073b615b74b4033bdf286064414c4.jpg

查看docker container的ip地址

[root@fys3 ~]# docker inspect --format='{{.NetworkSettings.IPAddress}}' 53793bfdd9ae  
172.17.0.1

docker container中启动的端口是8080，但是主机使用31422作为代理，如下所示：



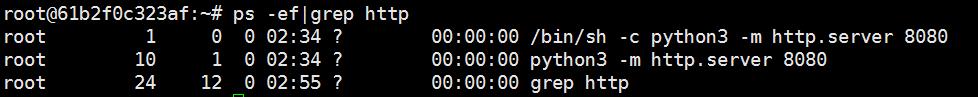
进入docker

./docker-enter.sh 61b2f0c323af

*注：docker-enter.sh是根据docker命令开发的脚本（无sshd，以nsenter为基础），链接：*

*https://github.com/jpetazzo/nsenter*

在docker container中查看如下：



# 5.Marathon框架

## 5.1 Marathon模块

|—Marathon

|— marathon-core

|— REST-API //JSON/REST API

|— Constaints

|— Health Check

|— Docker

|— Metrics

|— Basic Auth and SSL

|— HA

|— Service Discovery & Load Balancing

|— Mesos-DNS

|— Marathon-lb

|— Marathon-autoScale => marathon-lb-authoscale

|— marathon-ui

|— marathon-python // python library for interacting with Marathon Servers via REST API

|— marathon-client // Java API for interacting with Marathon Servers via REST API

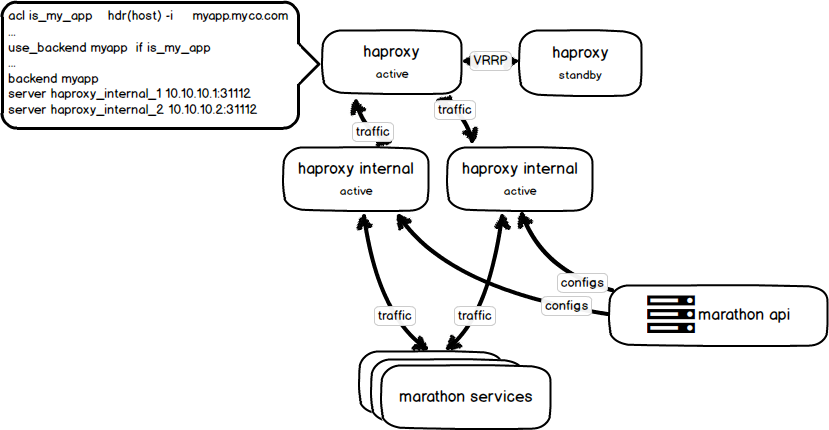
1) marathon-autoscale

各模块如下：

https://github.com/mesosphere/marathon-autoscale

2) HAproxy代理

为了使得系统对下层节点透明，需要运行一台负载均衡服务器。Marathon内置了一个简单的脚本，会从marathon api中拉去服务状况，更新HAproxy配置，但是因为marathon的工作模式，每个可访问的服务都被分配一个对外的端口号（service port），这些端口号都是HAproxy配置对外发布的。所有服务都想通过80端口发布服务，需要将一个proxy放置在marathon配置的haproxy之前。





# 6.源码分析

1）应用定义

2）主类，启动点

Marathon使用scala语言进行开发

Marathon#Main.Scala

#modules

def modules(): Seq[Module] = {

Seq(

new HttpModule(conf),

new MetricsModule,

new MetricsReporterModule(conf),

new MarathonModule(conf, conf),

new MarathonRestModule, //REST Server api

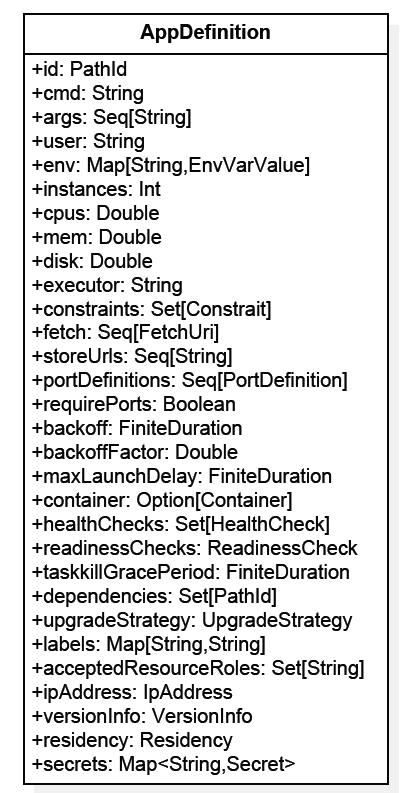
new EventModule(conf),

new DebugModule(conf),

new CoreGuiceModule

) ++ getEventsModule

}



具体含义：

id

cmd

# 7.Metics & Auto-Scaling