Apache Mesos 官方文档 V1.0

关于《Apache Mesos 官方文档》众包翻译

《Apache Mesos 官方文档》翻译悬赏已经完结，感谢各位译者和校对者的参与，文档贡献地址如下：http://mesos.mydoc.io

参与人员如下：

翻译：

Yol：《对 Mesos 贡献》《Mesos 扩展》

土豆妮珠：《高级功能》

抢小孩糖吃：《运行 Mesos 框架》《开发 Mesos 框架》

黑传说：《Mesos 基础》《运行 Mesos》

校对：

KevinShu ：《Mesos 基础》《运行 Mesos》

秣马儿：《对 Mesos 贡献》《Mesos 扩展》

itfanr：《高级功能》《开发 Mesos 框架》

船老大：《运行 Mesos 框架》

大家对《Apache Mesos 官方文档》有任何意见或者建议，请在 http://www.oschina.net/news/68503 留下评论。

Mesos 基础

Mesos Architecture

Mesos 架构

上图是Mesos的主要部件。 Mesos包括一个 **主控** 守护进程，用来管理子节点的 **被控** 守护进程。

同时还包含Mesos **应用**（application，下文也称 框架 **framework**）负责在 子节点上运行 **任务**(tasks)。

主控进程 通过 给予各个应用框架来提供资源邀约（**resource offers**）来提供更细粒度的资源共享能力（资源类型包括CPU，内存……）。每种资源邀约 包含 一个列表<slave ID, resource1: amount1, resource2, amount2, ...>即：被控端ID，资源1：数量，资源2：数量……

主控进程 可根据给定的组织策略 来决定分配给每个应用多少资源，比如：平均分配，或者严格按优先级分配等。

为了支持多样化的策略，主控采用了模块化的架构，这样一来，通过插件机制来添加新的资源分配模块变得很容易。

一个运行在 Mesos 之上的应用框架包含两个组件：**调度器** (scheduler)进程和 **执行器**(executor)进

程。

调度器进程 用于从 主控提供的资源里选定使用哪些资源。执行器进程在被控节点启动来执行应用任务（详见App/应用开发指南(null)）。

主控 负责决定给每个应用分配多少资源，而应用框架的调度器才会真正选择使用哪些被分配到的资源。当应用框架接收了分配的资源，它会向Mesos发送一个它希望运行任务的描述信息。然后，Mesos会负责在相应的被控节点上启动任务。

示例：资源分派流程

下图给出了一个例子，展示了一个应用如何被调度来执行任务：

根据上图，看一下图片中的事件：

1. 被控端1 报告给 主控，报告内容是它有 4个CPU和4 GB内存可用。然后，主控触发分配策略模块（allocation module），该模块告诉它应该给应用框架1分派所有可用的资源。

2. 主控将 被控端1 可以的资源信息通过资源邀约（resource offer）发送至 应用框架1。

3. 应用框架1 的 调度进程 发送给主控相应的反馈信息，内容是它会在被控端上运行两个任务，第一个要用“2 CPU，1 GB内存”，第二个要用“1 CPU，2 GB内存”。

4. 最后，主控发送任务至被控端1，将其可用资源分配给应用1的执行器。之后，执行器会启动两个相应的任务（在图中显示为点划线边框）。由于被控端还剩余1个CPU和1 GB内存没有被分配，分配模块之后可能会把它们提供给应用框架2。

之后，当任务结束，被控端的资源被释放时，资源调度分派进程会根据新任务来循环执行上面的流程。

通过这样简化的交互接口设计，Mesos集群可以方便的进行横向扩展。而与此同时，每一个运行其上的应用框架也会相互独立，完成各自的演进和升级。

那么，问题来了：如何在Mesos不知道应用框架存在特殊限制条件的前提下满足其限制条件？比如，在Mesos不知道哪个节点存储着应用所需要的数据时，应用框架如何满足数据处理的本地化要求？Mesos的处理方式是让应用直接拒绝（**reject**）资源邀约。应用会拒绝不满足其限制条件的资源邀约，而只接受满足条件的资源。特别指出的是，我们发现了一种简单的策略，称之为延迟调度（delay scheduling）机制，即应用可以等待一定的时间，来获得存储需要数据的相应节点，从而获得近似优化过的数据本地处理结果。

想了解更多 Mesos架构，可见论文 Mesos的实现技术(null).

Video and Slides of Mesos Presentations

运行 Mesos

快速入门

快速入门

下载Mesos

有两种方法：

1. 从 Apache(null)官方网站下载 (***推荐***)

```
$ wget http://www.apache.org/dist/mesos/0.24.0/mesos-0.24.0.tar.gz
$ tar -zxf mesos-0.24.0.tar.gz
```

1. 克隆Mesos的 git 代码仓库 repository(null) (***适用于高级用户***)

```
$ git clone https://git-wip-us.apache.org/repos/asf/mesos.git
```

系统要求

Mesos 可运行在 Linux (64 Bit) 和 Mac OS X (64 Bit)上。

Ubuntu 14.04

下面介绍Ubuntu 14.04中的安装步骤说明：

```
# 更新包库
$ sudo apt-get update

# 安装最新版本的 OpenJDK.
$ sudo apt-get install -y openjdk-7-jdk

# 安装编译工具 (需要编译时需要)
$ sudo apt-get install -y autoconf libtool

# 安装相关依赖
$ sudo apt-get -y install build-essential python-dev python-boto libcurl4-nss-dev
libsasl2-dev maven libapr1-dev libsvn-dev
```

Mac OS X Yosemite

下面 Mac OS X Yosemite 中的步骤说明：

```
# 安装命令行工具
$ xcode-select --install

# 安装 Homebrew
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

```
# 安装关联库
$ brew install autoconf automake libtool subversion maven
```

CentOS 6.6

下面 CentOS 6.6 中的步骤说明：

```
    # 安装基础工具
    $ sudo yum install -y tar wget which

    # 'Mesos > 0.21.0' 需要 完整支持 C++11 的编译器， (比如 GCC > 4.8)的工具中 有 'devtoolset-2'可用。
    # 下载 Scientific Linux CERN devtoolset 仓库文件。
    $ sudo wget -O /etc/yum.repos.d/slc6-devtoolset.repo http://linuxsoft.cern.ch/cern/devtoolset/slc6-devtoolset.repo

    # 导入CERN GPG 密钥
    $ sudo rpm --import http://linuxsoft.cern.ch/cern/centos/7/os/x86_64/RPM-GPG-KEY-cern

    # 下载 Apache Maven 仓库文件
    $ sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo

    # 'Mesos > 0.21.0' 需要 'subversion > 1.8' 开发包
    # 不过这版本没在默认仓库里，需要手动添加。
    # 在文件'/etc/yum.repos.d/wandisco-svn.repo'中添加如下内容：

      [WANdiscoSVN]
      name=WANdisco SVN Repo 1.8
      enabled=1
      baseurl=http://opensource.wandisco.com/centos/6/svn-1.8/RPMS/$basearch/
      gpgcheck=1
      gpgkey=http://opensource.wandisco.com/RPM-GPG-KEY-WANdisco

    # 安装相应开发工具
    $ sudo yum groupinstall -y "Development Tools"

    # 安装 'devtoolset-2-toolchain'，内有 GCC 4.8.2 和相关工具
    $ sudo yum install -y devtoolset-2-toolchain

    # 安装 Mesos 依赖
```

```
    $ sudo yum install -y apache-maven python-devel java-1.7.0-openjdk-devel zlib-devel libcurl-devel openssl-devel cyrus-sasl-devel cyrus-sasl-md5 apr-devel subversion-devel apr-util-devel

    # 在终端中，开启'devtoolset-2'
    $ scl enable devtoolset-2 bash
    $ g++ --version  # 注意：一定要 GCC > 4.8！
```

CentOS 7.1

下面 CentOS 7.1 中的步骤说明：

```
    # 安装基础工具
    $ sudo yum install -y tar wget

    # 下载 Apache Maven 仓库文件
    $ sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo

    # 'Mesos > 0.21.0' 需要 'subversion > 1.8' 开发包
    # 不过这版本没在默认仓库里，需要手动添加。
    # 在文件'/etc/yum.repos.d/wandisco-svn.repo'中添加如下内容：

      [WANdiscoSVN]
      name=WANdisco SVN Repo 1.9
      enabled=1
      baseurl=http://opensource.wandisco.com/centos/7/svn-1.9/RPMS/$basearch/
      gpgcheck=1
      gpgkey=http://opensource.wandisco.com/RPM-GPG-KEY-WANdisco

    # 安装相应开发工具
    $ sudo yum groupinstall -y "Development Tools"

    # 安装 Mesos 依赖
    $ sudo yum install -y apache-maven python-devel java-1.7.0-openjdk-devel zlib-devel libcurl-devel openssl-devel cyrus-sasl-devel cyrus-sasl-md5 apr-devel subversion-devel apr-util-devel
```

编译 Mesos

```
# 切换到源码目录
$ cd mesos

# 自检 (只在从 git 编译时需要)
$ ./bootstrap

# 配置并编译
$ mkdir build
$ cd build
$ ../configure
$ make
```

可设置-j <使用核心数量> V=0 来加快编译进度，减少日志输出。

```
# 运行测试
$ make check

# 安装
$ make install
```

示例

Mesos 自带 C++, Java 和 Python 框架实例

```
# 切换到源码目录
$ cd build

# 启动 mesos 主控端(请确保 目录存在，且权限合适)
$ ./bin/mesos-master.sh --ip=127.0.0.1 --work_dir=/var/lib/mesos

# 启动 被控
$ ./bin/mesos-slave.sh --master=127.0.0.1:5050

# 访问 mesos页面
$ http://127.0.0.1:5050

# 运行 C++ 框架 (运行任务成功后会退出)
$ ./src/test-framework --master=127.0.0.1:5050

# 运行Java 框架 (运行任务成功后会退出)
$ ./src/examples/java/test-framework 127.0.0.1:5050
```

```
# 运行 Python 框架 (运行任务成功后会退出)
$ ./src/examples/python/test-framework 127.0.0.1:5050
```

NOTE: 需要使用 make check编译测试套件，才会编译 实例框架。

配置

Mesos 配置

Mesos主控端和被控端的配置可选项 可通过 命令行参数 或 设置环境变量 来配置。

输入命令 mesos-master --help 和 mesos-slave --help可查看所有 主控端和被控端 的可用参数，参数可通过两种方式设置：

• 在命令行 使用 --option_name=value 设置，可直接输入 设置的值，也可指定一个存储了设置的文件 (`--option_name=file://path/to/file`)，可以是 绝对路径也可以当前路径的相对路径。

• 设置环境变量MESOS*OPTION*NAME（名称格式是参数名字加上前缀 MESOS_ ）。

配置的使用顺序：先搜索环境变量，然后再使用命令行变量。

重要提示

若需特殊编译，可在预配置时，通过命令 ./configure --help 查看编译配置帮助。

另外，下文所列参数仅仅是Mesos最近版本的，具体可以使用哪些参数，需要根据你的Mesos版本而定，可通过 在命令后添加 --help 查询，比如 mesos-master --help。

主控和被控端通用参数

下面参数可通用于主控和被控端

<table class="table table-striped">

<thead>

```
<tr>
 <th width="30%">
  标记
 </th>
 <th>
  解释
 </th>
</tr>
```

</thead>

<tr>

```
 <td>
```

--external_log_file=VALUE
</td>
<td>
设定外部日志文件。此日志可通过 web界面和HTTP api 查看。用途：使用了Mesos不识别的stderr logging 作为日志文件。
</td>
</tr>
<tr>
<td>
--firewall_rules=VALUE
</td>
<td>
设置防火墙规则，所设置的值 必须是个 json格式字串，或者是 使用了json格式的文档路径。路径格式： <code>file:///path/to/file</code> 或者
<code>/path/to/file</code>.
<p/>
可在防火墙信息输出带flags.proto中，查看可用格式。
<p/>
示例：

<pre><code>{
"disabled_endpoints" : {
"paths" : [
"/files/browse",
"/slave(0)/stats.json",
]
}
}</code></pre>

</td>
</tr>
<tr>
<td>
--[no-]help
</td>
<td>
屏显帮助信息（默认设置：false，也就是不输出）
</td>
</tr>

```
</td>
```

</tr>

<tr>

```
<td>
 --[no-]initialize_driver_logging
</td>
<td>
 设置是否初始化 google 调度日志 和 / 或者 执行器驱动。（默认：true）
</td>
```

</tr>

<tr>

```
<td>
 --ip=VALUE
</td>
<td>
 设置监听的IP地址，当使用 `--ip_discovery_command` 时，这个设定会失效。
</td>
```

</tr>

<tr>

```
<td>
 --ip_discovery_command=VALUE
</td>
<td>
 可选项，若设置，将使用主控绑定的IP地址。这个设定和 `--ip`  互斥。
</td>
```

</tr> <tr>

```
<td>
 --log_dir=VALUE
</td>
<td>
 设置日志文件存储地址（无默认设置，当无设置时，将不会生成任何日志文件。但不会影响到  stderr 日志）。

</td>
```

</tr>

<tr>

<td>
--logbufsecs=VALUE
</td>
<td>
设置多少分钟记录一次日志信息（默认是0）

</td>

</tr>

<tr>

<td>
--logging_level=VALUE
</td>
<td>
设置记录的日志信息级别，可用的设置：
'INFO'（一般信息），'WARNING'（警告），'ERROR'（出错）；若使用 `quiet` 参数，将只会作用到 log_dir（若设置了的话）里的日志。（默认级别：INFO）。

</td>

</tr>

<tr>

<td>
--port=VALUE
</td>
<td>
设置监听的端口（主控默认：5050，被控默认：5051）

</td>

</tr>

<tr>

<td>
--[no-]quiet
</td>
<td>
设置不把日志输出到 stderr（默认：false）

</td>

</tr>

<tr>

<td>
--[no-]version
</td>
<td>
显示版本并退出（默认：false）
</td>

</tr>

</table>

主控参数

必设参数

<table class="table table-striped">

<thead>

<tr>
<th width="30%">
标记
</th>
<th>
解释
</th>
</tr>

</thead>

<tr>

<td>
--advertise_ip=VALUE
</td>
<td>
设置外网可通达主控的对公IP地址。Mesos主控自身不会绑定这个IP地址，但这IP将用来操控Mesos主控端。
</td>

</tr>

<tr>

<td>
--advertise_port=VALUE
</td>

<td>
设置可通达主控的对公端口。Mesos主控自身不会绑定这个端口，但这端口和上面提到的对公IP将用来操控Mesos主控端。
</td>

</tr>

<tr>

<td>
--quorum=VALUE
</td>
<td>
当登记为“同步日志”（'replicated_log'）时，设置 相对多数的值。
这个值必须大于主控数的一半，比如：相对多数 > (主控数)/2。
<p/>

NOTE 当主控只有一个时(non-HA)，不需要设置这个值。
</td>

</tr>

<tr>

<td>
--work_dir=VALUE
</td>
<td>
设置　注册的信息　存储位置。

</td>

</tr>

<tr>

<td>
--zk=VALUE
</td>
<td>
ZooKeeper的连接地址（用于主控们选举主导者）。格式使用下面之一即可：

<pre><code>zk://主机地址1:端口1,主机地址2:端口2,.../路径
zk://用户名:密码@主机地址1:端口1,主机地址2:端口2,.../路径
file:///path/to/file (包含上面信息文件的位置)</code></pre>

<p/>

NOTE 当主控只有一个时(non-HA)，不需要设置这个值。
</td>
</tr>
</table>
可选参数
<table class="table table-striped">
<thead>
<tr>
<th width="30%">
标记
</th>
<th>
解释
</th>
</tr>
</thead>
<tr>
<td>
--acls=VALUE
</td>
<td>
设置可执行列表，格式是JSON格式的字串或者是使用JSON的文件路径。可以绝对路径也可以相对路径，如：<code>file:///path/to/file</code> 或 <code>/path/to/file</code>。
<p/>
在mesos.proto的ACL的protobuf中，查看可用格式。
<p/>
JSON配置文件示例:
<pre><code>{
"register_frameworks": [
{
"principals": { "type": "ANY" },
"roles": { "values": ["a"] }
}
],
"run_tasks": [

```
{
  "principals": { "values": ["a", "b"] },
  "users": { "values": ["c"] }
}
```

],

"shutdown_frameworks": [

```
{
  "principals": { "values": ["a", "b"] },
  "framework_principals": { "values": ["c"] }
}
```

]

}</code></pre>

```
</td>
```

</tr>

<tr>

```
<td>
 --allocation_interval=VALUE
</td>
<td>
 设置（成批）分配资源时的等待回馈时间量。 (比如 500ms, 1sec, 等). (默认1秒： 1secs)
</td>
```

</tr>

<tr>

```
<td>
 --allocator=VALUE
</td>
<td>
 分配资源的调度器，默认<code>HierarchicalDRF</code>，或者使用 <code>--
modules</code>来调用 调度器。
 (默认： HierarchicalDRF)
</td>
```

</tr>

<tr>

```
<td>
 --[no-]authenticate
```

</td>
<td>
若认证被设定为`true`，将只有被认证的应用可被注册。默认所有非认证的应用都可注册。(默认： false)
</td>

</tr>
<tr>

<td>
--[no-]authenticate_slaves
</td>
<td>
若设为'true'，认证的被控端将可注册。
<p/>
默认所有非认证的被控端皆可注册。 (默认： false)
</td>

</tr>
<tr>

<td>
--authenticators=VALUE
</td>
<td>
设定认证方式，默认<code>crammd5</code>，或者使用 <code>--modules</code>来调用 认证模块。
(默认： crammd5)
</td>

</tr>
<tr>

<td>
--authorizers=VALUE
</td>
<td>
设置授权方式，默认使用<code>local</code>，或者使用 <code>--modules</code>来调用 授权模块。

Note：当 <code>--authorizers</code>的值不是<code>local</code>时，可执行列表<code>--acls</code>传递的参数将被忽略。

当前不支持混用多种授权方式。
(默认：<code>local</code>)
</td>
</tr>
<tr>
<td>
--cluster=VALUE
</td>
<td>
设定在web界面里显示的集群名。
</td>
</tr>
<tr>
<td>
--credentials=VALUE
</td>
<td>
设定证书列表路径。证书列表可以是一个文本文件（每行记录一个 授权对象和密钥，用空格隔开），也可以是一个json格式文档。路径格式可以是<code>file:///path/to/file</code>或者
<code>/path/to/file</code>
<p/>
JSON文件示例：
<pre><code>{
"credentials": [
{
"principal": "sherman",
"secret": "kitesurf"
}
]
}</code></pre>
<p/>
文本文件示例:
<pre><code> username secret </code></pre>
</td>

</tr>
<tr>
<td>
--framework_sorter=VALUE
</td>
<td>
设置 用户的应用 间的资源分配策略。值和user_allocator一样。 (默认按优先度分配： drf)
</td>
</tr>
<tr>
<td>
--hooks=VALUE
</td>
<td>
设置主控将安装的可调用模块，模块名间用逗号隔开。
</td>
</tr>
<tr>
<td>
--hostname=VALUE
</td>
<td>
设置主控在ZooKeeper中的主机名。
若不设置，将使用IP地址标识。
</td>
</tr>
<tr>
<td>
--[no-]log_auto_initialize
</td>
<td>
设置是否在注册时自动初始化 “同步日志” ，若设为 flase，则以后首次使用的时候，需要手动创建。 (默认： true)
</td>
</tr>
<tr>

<td>
--max_slave_ping_timeouts=VALUE
</td>
<td>
设置 呼叫 被控端无回应的超时时间。超时的被控端将被移除。
(默认： 5)
</td>

</tr>

<tr>

<td>
--modules=VALUE
</td>
<td>
设置内部子系统需要加载的模块。
<p/>
可用 <code>--modules=filepath</code>指定文件路径。文件是以JSON格式保存的需加载模块列表。
文件内路径格式可用 <code>file:///path/to/file</code> 或
<code>/path/to/file</code>。<p/>
也可用 <code>--modules="{...}"</code>在行内列出所需加载的模块。
<p/>
JSON文件示例：

<pre><code>{
"libraries": [
{
"file": "/path/to/libfoo.so",
"modules": [
{
"name": "org_apache_mesos_bar",
"parameters": [
{
"key": "X",
"value": "Y"
}
]
},
{
"name": "org_apache_mesos_baz"
}

```
  ]
 },
 {
  "name": "qux",
  "modules": [
   {
    "name": "org_apache_mesos_norf"
   }
  ]
 }
```

]

}</code></pre>

```
 </td>
```

</tr>

<tr>

```
 <td>
  --offer_timeout=VALUE
 </td>
 <td>
  设置应用申请资源的超时时间。
  <p/>
  这可用于：当应用应用一直不停申请资源时，或者应用意外丢失申请时。

 </td>
```

</tr>

<tr>

```
 <td>
  --rate_limits=VALUE
 </td>
 <td>
  设置应用的资源使用最大比例，可以是一个JSON格式的字串，也可以一个设定的JSON格式
文件。
  <p/>
  文件内路径格式可用 <code>file:///path/to/file</code> 或
  <code>/path/to/file</code>。
  <p/>
```

见mesos.proto里的 RateLimits protobuf，查看可用格式。
<p/>

示例：

<pre><code>{
"limits": [
{
"principal": "foo",
"qps": 55.5
},
{
"principal": "bar"
}
],
"aggregate*default*qps": 33.3
}</code></pre>

</td>

</tr>
<tr>
<td>
--recovery_slave_removal_limit=VALUE
</td>
<td>
设置被移出注册表以及重新注册超时后关机的被控端比例，以启动 切换备用 模式。
若超过此设定的比例，则主控将会 启用备用 模式，而不是移除被控。
<p/>
在生产环境中，这种模式可令系统更为稳健。生产环境中，主控间频繁切换就相当于 被控端永久崩溃的比例已经超过最大设定。（比例由rack-level设定）。
<p/>
此项设置可用于 当集群发生大规模被控崩溃的时候，让人工干预。
<p/>
值： [0%-100%] (默认： 100%)
</td>

</tr>
<tr>
<td>

```
 --registry=VALUE
</td>
<td>
 设置注册的策略。
 <p/>
 可用设置：'replicated_log'（ ）, 'in_memory' (测试中的功能). (默认： replicated_log)
</td>
```

</tr>

<tr>

```
<td>
 --registry_fetch_timeout=VALUE
</td>
<td>
 设置获取注册信息的超时时间。(默认1分钟： 1mins)
</td>
```

</tr>

<tr>

```
<td>
 --registry_store_timeout=VALUE
</td>
<td>
 设置存储注册信息的超时时间。(默认5秒： 5secs)
</td>
```

</tr>

<tr>

```
<td>
 --[no-]registry_strict
</td>
<td>
 设置是否验证注册信息。若设置为 false，则代表接受所有的注册，包括：被控加入申请、被控读取申请、移除被控申请。这样设置不需系统重构，从而可令运行中的集群 保持持续运转。
 <p/>
 NOTE: 此项功能目前仍处 *实验* 阶段，不可用于生产环境！(默认：false)
</td>
```

</tr>

<tr>

<td>
--roles=VALUE
</td>
<td>
设置应用所属 角色，格式是用逗号分隔 所属角色名的字串。
</td>

</tr>

<tr>

<td>
--[no-]root_submissions
</td>
<td>
设置是否可以以 root 身份登录。(默认： true)
</td>

</tr>

<tr>

<td>
--slave_ping_timeout=VALUE
</td>
<td>
设置 主控 呼叫 被控 的超时时间。超时无回应的被控将被移出注册表。
(默认15秒：15secs)
</td>

</tr>

<tr>

<td>
--slave_removal_rate_limit=VALUE
</td>
<td>
设置 移除被控端的 最大速度（比如10分钟移除一个，或者3小时移除2个：1/10mins, 2/3hrs 等），默认是遇到失联的被控则立刻移除。
<p/>
设置值的计算公式：'被控数量'/'间隔时间'
</td>

</tr>

<tr>

<td>
--slave_reregister_timeout=VALUE
</td>
<td>
设置 当新主导端被选定后，所有被控端 重新注册 的超时时间。超时的被控端将被剔除出成员表，它们对主控的连接将被关闭。
<p/>
NOTE: 此值最小10分钟。(默认10分钟： 10mins)
</td>

</tr>

<tr>

<td>
--user_sorter=VALUE
</td>
<td>
设置用户间资源分配策略，如下：
<p/>
dominant_resource_fairness (drf) (默认按优先度分配： drf)
</td>

</tr>

<tr>

<td>
--webui_dir=VALUE
</td>
<td>
web界面的路径 (默认： /usr/local/share/mesos/webui)
</td>

</tr>

<tr>

<td>
--weights=VALUE
</td>
<td>
角色和权重（优先度）列表，用逗号分隔。如 'role=weight,role=weight'。
</td>

</tr>

<tr>

<td>
--whitelist=VALUE
</td>
<td>
设定一个白名单文件名，此文件里每行记录一个被控端可供给的资源信息，此文件会被监视，一定周期内会被重读刷新　被控端白名单。默认没有白名单，所有机器都可被接受。（默认：None）
<p/>

示例：
<pre><code>file:///etc/mesos/slave_whitelist</code></pre>
<p/>
</td>

</tr>

<tr>

<td>
--zk_session_timeout=VALUE
</td>
<td>
设置ZooKeeper会话超时时间。(默认10秒：10secs)
</td>

</tr>

</table>

当使用了'--with-network-isolator'时，可用参数设置

标记	解释
--max_executors_per_slave=VALUE	设置 每个被控端 可用执行器最大数量。网络监控和隔离技术会让每个执行器占用一定的资源(# ephemeral ports)，因此每个被控端只能运行一定数量的执行器。 <p/> 当被控端的临时端口都被分配后，此标识用于避免 应用再来获取资源申请。

被控参数

必设参数

<table class="table table-striped">

<thead>

```
<tr>
 <th width="30%">
  标记
 </th>
 <th>
  解释
 </th>
</tr>
```

</thead>

<tr>

```
<td>
 --master=VALUE
</td>
<td>
 设置连接主控的方式。有三种方式：
 <ol>
  <li> 主控的主机名或者IP地址，多个用逗号分隔。如：
```

<pre><code>--master=localhost:5050

--master=10.0.0.5:5050,10.0.0.6:5050

</code></pre>

```
  </li>

  <li> zookeeper 或 quorum hostname/ip + 端口 + 主控注册表路径</li>
```

<pre><code>--master=zk://host1:port1,host2:port2,.../path

--master=zk://username:password@host1:port1,host2:port2,.../path

</code></pre>

```
  </li>

  <li> 可以把上面两种设定中的一种放进一个文件里，然后指定文件路径
<code>file:///path/to/file</code>，连接主控。
  </li>
 </ol>
</td>
```

</tr>

</table>

可选参数

<table class="table table-striped">
<thead>

```
<tr>
 <th width="30%">
  标志
 </th>
 <th>
  解释
 </th>
</tr>
```

</thead>
<tr>

```
<td>
 --attributes=VALUE
</td>
<td>
 设置机器参数，格式如下：
 <p/>
 <code>rack:2</code> 或 <code>'rack:2;u:1'</code>
</td>
```

</tr>
<tr>

```
<td>
 --authenticatee=VALUE
</td>
<td>
 设定认证方式，默认<code>crammd5</code>，或者使用 <code>--modules</code>来调用 认证模块。
</td>
```

</tr>
<tr>

```
<td>
 --[no-]cgroups_cpu_enable_pids_and_tids_count
</td>
<td>
 设置使用分组策略cgroup来计算容器中的进程和线程。(默认： false)
</td>
```

</tr>
<tr>
<td>
--[no-]cgroups_enable_cfs
</td>
<td>
设置使用分组策略cgroup的CFS带宽限额来限制CPU使用量。
(默认： false)
</td>
</tr>
<tr>
<td>
--cgroups_hierarchy=VALUE
</td>
<td>
设置cgroup是的路径
(默认： /sys/fs/cgroup)
</td>
</tr>
<tr>
<td>
--[no-]cgroups_limit_swap
</td>
<td>
设置使用分组策略cgroup限制内存和缓存使用量。
(默认： false)
</td>
</tr>
<tr>
<td>
--cgroups_root=VALUE
</td>
<td>
设置cgroup的根名字。
(默认： mesos)
</td>
</tr>
</tr>

<tr>
<td>
--container_disk_watch_interval=VALUE
</td>
<td>
设置当使用<code>posix/disk</code>分隔策略时，内部容器间对磁盘轮询的时间间隔。
(默认15秒： 15secs)
</td>
</tr>
<tr>
<td>
--containerizer_path=VALUE
</td>
<td>
设置当 外部网络隔离启用时(--isolation=external)，外部管理容器的路径。

</td>
</tr>
<tr>
<td>
--containerizers=VALUE
</td>
<td>
设置使用的容器管理工具，多个的话用逗号分隔，系统将按出现的顺序使用。
<p/>
可用的值：'mesos', 'external', 和
'docker' (on Linux)。
(默认： mesos)
</td>
</tr>
<tr>
<td>
--credential=VALUE
</td>
<td>
设定证书列表路径。证书列表可以是一个文本文件（每行记录一个 授权对象和密钥，用空格隔开），也可以是一个json格式文档。路径格式可以是<code>file:///path/to/file</code>
</td>
</tr>

或者
<code>/path/to/file</code>

<p>JSON文件示例：<p/>

<pre><code>{
"principal": "username",
"secret": "secret"
}</code></pre>

</td>

</tr>

<tr>

<td>
--default_container_image=VALUE
</td>
<td>
设置当使用外部容器工具时，默认使用的镜像。
</td>

</tr>

<tr>

<td>
--default_container_info=VALUE
</td>
<td>
设定默认容器信息ContainerInfo路径，信息保存在一个JSON格式文档中。当没有指定容器信息的时候，ExecutorInfo将使用这个默认值。
<p/>
可在mesos.proto的ContainerInfo protobuf 中，查看可用格式。
<p/>
内容示例如下：

<pre><code>{
"type": "MESOS",
"volumes": [

{
"host_path": "./.private/tmp",

```
  "container_path": "/tmp",
  "mode": "RW"
 }
]
}</code></pre>
```

```
 </td>
```

</tr>

<tr>

```
 <td>
  --default_role=VALUE
 </td>
 <td>
  设置默认角色：当 `--resources` 但却没有设置角色的时候，或者没有列在`--resources`中的，都将使用此默认角色。
  (默认： `*`)
 </td>
```

</tr>

<tr>

```
 <td>
  --disk_watch_interval=VALUE
 </td>
 <td>
  设置监测磁盘空间使用状况的周期时间。 (默认1分钟： 1mins)
 </td>
```

</tr>

<tr>

```
 <td>
  --docker=VALUE
 </td>
 <td>
  设置docker的绝对路径。
  (默认： docker)
 </td>
```

</tr>

<tr>

<td>
--docker_remove_delay=VALUE
</td>
<td>
延迟移除docker容器的时间（比如3天，2周等 3days, 2weeks）。
(默认6小时： 6hrs)
</td>

</tr>

<tr>

<td>
--[no-]docker_kill_orphans
</td>
<td>
设置让docker管理器关闭已经空载的容器。当需要在同一个系统里面运行多个被控的时候，需要设置为 false，已免docker移除被其他 被控端使用的 容器。同时，也需要给被控开启运行监测，以使得 被控的识别名能够重用，否则被控上的docker任务在重启的时候将无法被清空。
(默认： true)
</td>

</tr>

<tr>

<td>
--docker_sock=VALUE
</td>
<td>
设置 将被docker执行器加载的UNIXsocket路径，以使docker命令行界面里可操作docker守护进程。这个路径必须是被控的docker 镜像里用的。
(默认： /var/run/docker.sock)
</td>

</tr>

<tr>

<td>
--docker_mesos_image=VALUE
</td>
<td>
设置docker镜像，用于运行 mesos被控实例。若镜像被指定，被控端运行于docker容器中，当被控端重启并恢复的时候，将用docker容器加载执行器，来恢复状态。
</td>

```
</td>
```

</tr>

<tr>

```
<td>
 --docker_stop_timeout=VALUE
</td>
<td>
 设置docker停用到杀死一个实例的等待时间，默认0秒表示停止后立即杀死。 (默认0秒：
0secs)
</td>
```

</tr>

<tr>

```
<td>
 --sandbox_directory=VALUE
</td>
<td>
 设置容器中沙箱所指的绝对路径。
 (默认： /mnt/mesos/sandbox)
</td>
```

</tr>

<tr>

```
<td>
 --[no-]enforce_container_disk_quota
</td>
<td>
 设置容器是否开启磁盘使用限额。此项用于 `posix/disk` 隔离器。 (默认： false)
</td>
```

</tr>

<tr>

```
<td>
 --executor_environment_variables
</td>
<td>
 设置传递给执行器及随后任务的环境变量，格式为JSON对象。默认执行器继承被控端的环境
变量。
 示例：
```

<pre><code>{
"PATH": "/bin:/usr/bin",
"LD*LIBRARY*PATH": "/usr/local/lib"
}</code></pre>

</td>

</tr>

<tr>

<td>
--executor_registration_timeout=VALUE
</td>
<td>
设置执行器向被控注册的超时时间，超过设定时间则被认为是挂起并被关闭。(可以 60secs, 3mins, 等) (默认1分钟：1mins)
</td>

</tr>

<tr>

<td>
--executor_shutdown_grace_period=VALUE
</td>
<td>
设置执行器关闭前的等待时间(可以 60secs, 3mins, 等) (默认5秒： 5secs)
</td>

</tr>

<tr>

<td>
--frameworks_home=VALUE
</td>
<td>
设置应用的家路径，用于作为相应执行器URI的前缀。(默认无：)
</td>

</tr>

<tr>

<td>
--gc_delay=VALUE
</td>

```
<td>
 设置清理执行器目录的周期。
 <p/>
 注意：可根据磁盘使用情况确定周期。(默认一周： 1weeks)
</td>
```

</tr>

<tr>

```
<td>
 --gc_disk_headroom=VALUE
</td>
<td>
 设置执行器目录的纯空置率（就是完全没被用到的磁盘空间比例，）。纯空置率用于计算目录存续期，存续期计算方式：</p>
 <code>gc_delay * max(0.0, (1.0 - gc_disk_headroom - disk usage))</code>
 every <code>--disk_watch_interval</code> duration.
 <code>gc_disk_headroom</code>数字必须在0.0到 1.0之间。
 (默认： 0.1)
</td>
```

</tr>

<tr>

```
<td>
 --hadoop_home=VALUE
</td>
<td>
 设置Hadoop的家目录，以便可在HDFS中找到应用执行器。若无设置，则会从环境变量中寻找 HADOOP_HOME 或在 PATH中找 可执行程序hadoop。(默认无）
</td>
```

</tr>

<tr>

```
<td>
 --hooks=VALUE
</td>
<td>
 设置将被安装在主控上的可注入模块，多个用逗号分隔。
</td>
```

</tr>

<tr>

<td>
--hostname=VALUE
</td>
<td>
设置被控的主机名。
<p/>
若没有设置，则会用被控的IP地址来绑定主机名。
</td>

</tr>

<tr>

<td>
--isolation=VALUE
</td>
<td>
设置隔离机制。比如'posix/cpu,posix/mem', or
'cgroups/cpu,cgroups/mem', 或 network/port_mapping
(可用 --with-network-isolator 开启并设置),
或者 'external', 或者可用载入一个 <code>--modules</code>隔离模块。
注意：只对Mesos容器管理器有效。(默认：posix/cpu,posix/mem)
</td>

</tr>

<tr>

<td>
--launcher_dir=VALUE
</td>
<td>
设置Mesos可执行程序路径(默认： /usr/local/lib/mesos)
</td>

</tr>

<tr>

<td>
--modules=VALUE
</td>
<td>
设置内部子系统可用的模块名。
<p/>
可用 <code>--modules=filepath</code>指定文件路径。文件是以JSON格式保存的需加

载模块列表。 文件内路径格式可用 <code>file:///path/to/file</code> 或
<code>/path/to/file</code>。<p/>
也可用 <code>--modules="{...}"</code>在行内列出所需加载的模块。 <p/>

JSON文件内容示例：

<pre><code>
{
"libraries": [
{
  "file": "/path/to/libfoo.so",
  "modules": [
    {
      "name": "org_apache_mesos_bar",
      "parameters": [
        {
          "key": "X",
          "value": "Y"
        }
      ]
    },
    {
      "name": "org_apache_mesos_baz"
    }
  ]
},
{
  "name": "qux",
  "modules": [
    {
      "name": "org_apache_mesos_norf"
    }
  ]
}
]
}</code></pre>
</td>
</tr>
<tr>

<td>
--oversubscribed_resources_interval=VALUE
</td>
<td>
设置间隔汇报时间：被控会把目前估算的 已用（也就是可被回收）及还没用的过剩资源总量，定期告诉主控。
(默认15秒： 15secs)
</td>

</tr>

<tr>

<td>
--perf_duration=VALUE
</td>
<td>
设置一个perf统计周期的长度，这个值必须低于perf_interval。(默认10秒： 10secs)
</td>

</tr>

<tr>

<td>
--perf_events=VALUE
</td>
<td>
设置使用 perf_event 分隔器时，用于统计每个容器的perf事件。默认为 none。
<p/>
可输入命令 'perf list' 查看所有的事件。在PerfStatistics protobuf里，事件名会被转为小写，连字符会被替换为下划线。比如cpu-cycles 变成 cpu_cycles，可在 PerfStatistics protobuf中查看所有的名字。
</td>

</tr>

<tr>

<td>
--perf_interval=VALUE
</td>
<td>
设置 perf 统计周期间的间隔，系统会根据这个间隔时间，获取perf的数据，并把最近的数据返回而不是展现。可见，这个perf_interval和资源监控间隔是不一样的。(默认 1 分钟：1mins)
</td>

```
</td>
```

</tr>

<tr>

```
<td>
 --qos_controller=VALUE
</td>
<td>
 设置QoS Controller的名字，用于过载检验。
</td>
```

</tr>

<tr>

```
<td>
 --qos_correction_interval_min=VALUE
</td>
<td>
 设置据QoS而调整的最小间隔周期：QoS Controller会根据自己监测到的任务运行性能进行调整，被控会轮询并据此调整。(默认0秒： 0secs)
</td>
```

</tr>

<tr>

```
<td>
 --recover=VALUE
</td>
<td>
 设置是否以更新的状态 还原 并重新连接 原有执行器。
 <p/>
 可用的值：
 <p/>
 reconnect: 重新连接那些 失联过但可用的执行器。
 <p/>
 cleanup ：杀死所有 失联但仍存在的执行器，并退出。——当被控和执行器需要 不向前兼容的升级时可用这项。
 <p/>
 <p/>
 NOTE: 若被控的运行监测没有退出，还原操作将不会发生，而是当作新被控在主控注册。(默认重连： reconnect)
</td>
```

</tr>
<tr>
<td>
--recovery_timeout=VALUE
</td>
<td>
设置还原被控端时的操作超时总时间。若被控端还原所需时间超过设定的时间，则此指向此被控端、待重连的执行器都将自动中断。
<p/>
NOTE: 此设置只有当`--checkpoint`启用时才有效。
(默认15分钟：15mins)
</td>
</tr>
<tr>
<td>
--registration_backoff_factor=VALUE
</td>
<td>
设置被控端到一个新主控重新注册的时间因子。被控会在 [0, b] 之间随机选择一个时间。
b = registration_backoff_factor 时间因子。
<p/>
随后的重新注册时间间隔，将以此为基础指数级增加直到满1分钟，比如第一次[0, b * 2^1]之间随机选择，第二次[0, b * 2^2]，第三次[0, b * 2^3] 等。
(默认1秒： 1secs)
</td>
</tr>
<tr>
<td>
--resource_estimator=VALUE
</td>
<td>
设置监测资源过剩的估算器名称。
</td>
</tr>
<tr>
<td>
--resource_monitoring_interval=VALUE
</td>
</tr>

```
</td>
<td>
 设置监控执行器资源使用情况的间隔时间。(比如：10secs, 1min等) (默认1秒：1secs)
</td>
```

</tr>

<tr>

```
<td>
 --resources=VALUE
</td>
<td>
 设置每个被控总共可用的资源量。形式如下：
 </p>
 <code>name(role):value;name(role):value...</code>.
</td>
```

</tr>

<tr>

```
<td>
 --[no-]revocable_cpu_low_priority
</td>
<td>
 设置是否以低优先度在可资源回收的CPU中运行，目前仅有cgroups/cpu隔离器支持此功能。 (默认： true)
</td>
```

</tr>

<tr>

```
<td>
 --slave_subsystems=VALUE
</td>
<td>
 设置 以cgroup子系统 来运行被控程序，多个以逗号分隔，比如 <code>memory,cpuacct</code>。默认none。
 目前此功能用于资源监控，默认不限制群组，将继承mesos cgroup设置。
</td>
```

</tr>

<tr>

```
<td>
```

--[no-]strict
</td>
<td>
若设置 `strict=true`，则将只恢复没有任何出错时的状态。
<p/>
若设置`strict=false`，所有之前的出错（比如类似：被控无法恢复执行器的信息，因为执行器在重注册之前被控端就挂了。），将被忽略，将尽可能恢复到最近状态。
(默认： true)
</td>

</tr>

<tr>

<td>
--[no-]switch_user
</td>
<td>
设置是否在被控端，以不同于被控端当前用户的身份运行任务。(需要 setuid 操作权限) (默认： true)
</td>

</tr>

<tr>

<td>
--fetcher_cache_size=VALUE
</td>
<td>
设置 获取器的缓存大小。单位：Bytes.
(默认：2 GB)
</td>

</tr>

<tr>

<td>
--fetcher_cache_dir=VALUE
</td>
<td>
设置 获取器的 缓存路径（这是父级路径，每个被控端一个子文件夹），默认路径放在工作目录中，方便操作。但生产环境则经常会把缓存单独存放到缓存卷，缓存和缓存卷的数据可能会互相干扰，因此，需要明确指定路径，以防不测。
(默认： /tmp/mesos/fetch)

</td>

</tr>

<tr>

<td>
--work_dir=VALUE
</td>
<td>
设置应用的工作路径。(默认： /tmp/mesos)
</td>

</tr>

</table>

当使用'--with-network-isolator'时，可用的参数

标记	解释
--ephemeral_ports_per_container=VALUE	设置分配给容器的临时端口数。数字必须是2的乘方。(默认： 1024)
--eth0_name=VALUE	设置对外连接使用的网络接口（比如eth0）。若没有指定，网络分隔器将根据主机的网关猜。
--lo_name=VALUE	设置环连接的接口（比如 lo）。若没指定，网络分隔器将自己猜。
--egress_rate_limit_per_container=VALUE	设置每个容器的对外网速限制，单位是Bytes/s。若无设置，或者设置为0，网络分隔器将不限速。
--[no-]network_enable_socket_statistics_summary	设置是否为每个容器采集 socket通讯的汇总信息。这个用于 'network/port_mapping' 设置。(默认： false)
--[no-]network_enable_socket_statistics_details	设置是否为每个容器采集 socket通讯的详细信息。这个用于 'network/port_mapping' 设置。 (默认：false)

Libprocess 进程管理　参数

内置的libprocess可用下面环境变量控制

变量	解释
LIBPROCESS_IP	设置用于和libprocess沟通的IP地址。
LIBPROCESS_PORT	设置用于和libprocess沟通的端口。
LIBPROCESS_ADVERTISE_IP	若设置，则会使用此IP用于libprocess的外部通讯。好处是有些不想让主控执行的通讯任务可以使用非对公ip。
LIBPROCESS_ADVERTISE_PORT	若设置，则会使用此端口用于libprocess的外部通讯。注意：此端口实际上不会被绑定（本地的LIBPROCESS_PORT则会），因此本地ip和端口必须要单独设置。

Mesos编译配置参数

编译配置中有下面可选参数设置：

标记	解释
--enable-shared[=PKGS]	编译共享库 [默认：=yes]
--enable-static[=PKGS]	编译静态库 [默认：=yes]
--enable-fast-install[=PKGS]	优化为快速安装 [默认：=yes]
--disable-libtool-lock	避免锁定（并行编译时锁死会打断编译）
--disable-java	不编译Java绑定
--disable-python	不编译Python绑定
--enable-debug	启用调试信息。若CFLAGS/CXXFLAGS 已经有设定，这个设定将不起作用（默认：no）。
--enable-optimize	开启优化。若CFLAGS/CXXFLAGS 已经有设定，这个设定将不起作用（默认：no）。
--disable-bundled	使用已装依赖，而不把依赖一起封包。

--disable-bundled-distribute	不编译distribute，而是使用PYTHONPATH已安装的版本。
--disable-bundled-pip	不编译pip，而是使用PYTHONPATH已安装的版本。
--disable-bundled-wheel	不编译wheel，而是使用PYTHONPATH已安装的版本。
--disable-python-dependency-install	当python依赖已经在`make install`阶段里装入时，不再需要下载或安装额外的依赖。

编译配置中有下面可选包设置：

标记	解释
--with-gnu-ld	使用 GNU ld 来复位 C编译器。 [默认：=no]
--with-sysroot=DIR	在设定的路径下（若没设定，则使用编译器的sysroot），查找关联库。
--with-zookeeper[=DIR]	不编译ZooKeeper，而是使用已安装的版本，此处指定已装版本路径。
--with-leveldb[=DIR]	不编译LevelDB，而是使用已安装的版本，此处指定已装版本路径。
--with-glog[=DIR]	不编译glog，而是使用已安装的版本，此处指定已装版本路径。
--with-protobuf[=DIR]	不编译protobuf，而是使用已安装的版本，此处指定已装版本路径。
--with-gmock[=DIR]	不编译gmock，而是使用已安装的版本，此处指定已装版本路径。
--with-curl=[=DIR]	设定curl路径。
--with-sasl=[=DIR]	设定sasl2路径。
--with-zlib=[=DIR]	设定zlib路径。
--with-apr=[=DIR]	设定apr-1路径。

--with-svn=[=DIR]	设定svn-1路径。
--with-network-isolator	编译网络分隔器。

对编译配置有影响的环境变量：

下面有些变量可能需要根据实际情况设定，以改变 configure 默认值，让编译器能够识别正确名字和路径。

变量	解释
JAVA_HOME	Java Development Kit (JDK)的路径
JAVA_CPPFLAGS	JNI　的预编译器设定
JAVA_JVM_LIBRARY	libjvm.so的完整路径
MAVEN_HOME	mvn的路径。指MAVEN_HOME/bin/mvn中的MARVEN_HOME部分
PROTOBUF_JAR	预编译时 protobuf jar的完整路径。
PYTHON	使用的Python解析器
PYTHON_VERSION	使用的Python版本，比如"2.3"。此字串将被按一定规则附加在Python解析器的名字后。

Mesos 容器化

Mesos 容器化

MesosContainerizer 使用Linux特有的功能（比如控制cgroups和namespaces）来分隔执行器使用的资源，从而提供了比较轻量级的容器化和执行器的资源隔离效果。因为其功能是可以配置组合的，所以操作人员也可以选择使用不同的分隔方式。

Mesos同时也对POSIX系统提供基本的支持，比如OSX，但没有实际可用的资源隔离功能，只能提供资源的使用报告。

共享文件系统

在linux主机上可选用共享文件系统 分隔器，从而可以使每个容器能对共享文件系统的视图进行修改。

可以通过在ExecutorInfo里的ContainerInfo的配置进行设置，即通过应用框架本身设计或者对–default*container*info 被控flag的值来对应配置。

ContainerInfo 指定了Volumes值，它把共享文件系统的 host_path 映射到容器中的路径 `container\_path`，执行权限为只读或者可读写。

如果主机路径host_path是绝对路径，其下子目录也可以在每个容器的 `container\_path` 里进行相应操作。

如果主机路径host_path是相对路径，则会以执行器工作目录为其对应的上级目录。当新建目录时，会复制共享文件系统里存在的相应目录（一定存在的）权限设置。

此隔离器一般的使用场景是把共享文件系统的某部分分配给每个容器作为私有空间。

比如，一个私有的"/tmp" 目录，主机端可通过 host_path="tmp" 访问，容器端则是通过 `container\_path="/tmp"` 访问，容器会在执行器的工作目录下新建一个 `tmp`目录(mode 1777)，同时把其挂载到容器的 `tmp`目录，这对容器内运行中的进程是可见的，但容器将无法看到主机和其他容器的 `/tmp`目录。

进程号（PID）命名空间

进程号命名空间级隔离可把每个容器放在独立的进程号命名空间里，这样做的好处有二：

1. 可见度控制： 容器命名空间之外的进程相对于容器内部运行的进程（比如执行器和子进程）是不可见的。

2. 一门清的清理方法：在命名空间里的主导线程执行中断操作，将会中断整个命名空间里的所有线程。

在销毁容器的时候，加载器将使用方法2而不是冻结cgroup，以避开在在OOM状态下cgroups冻结会引发的kernel的问题。

/proc目录会被挂载，以便类似于ps 的工具集可以正常工作。

Posix磁盘隔离

Posix磁盘隔离只提供基本的磁盘使用隔离功能，可用于监测每个沙箱的磁盘使用情况以及可选择的强制磁盘使用配额。此功能可在 Linux和 OS X上使用。

为了启用Posix磁盘隔离，可以在启动被控端时，在--isolation配置标识之前添加posix/disk。

默认情况下，Posix磁盘隔离不启用限额设置，可在启动被控端时通过指定--enforce*container* disk_quota配置项来启动此功能。

Posix磁盘隔离器通过运行du命令 ，周期性的监测磁盘使用情况。监测结果可在资源统计终端(/monitor/statistics.json)中查看。

执行 du 命令的间隔，可通过被控端的--container*disk*watch_interval配置项来进行设置，默认是15秒一次。例如`--container*disk*watch_interval=1mins`可将监测间隔设为一分钟一次。

Docker 容器化工具

Docker容器化工具（Docker Containerizer）

Mesos 0.20.0开始支持在docker镜像中运行任务，同时是Docker选项支持的子集，后续版本还会添加更多支持。

用户可以以任务或者执行器的方式加载docker镜像。

下文包含关于Docker支持的详细的API改变和设置Docker的方法。

安装docker

使用docker来运行被控，需要在启动的时候，添加参数，如下：

mesos-slave --containerizers=docker,mesos

docker容器化的被控端，还需要在被控端里安装 Docker 命令行工具客户端（版本 >= 1.0.0）。

若在被控端启用iptables，使用下面规则，以确保iptables允许docker桥连的流量能够通过：

```
iptables -A INPUT -s 172.17.0.0/16 -i docker0 -p tcp -j ACCEPT
```

使用docker

版本0.20.0以前，TaskInfo只允许使用 CommandInfo 来加载 一个命令行任务，或ExecutorInfo来加载自定义的Executor任务。

版本0.20.0之后，TaskInfo 和 ExecutorInfo 多了一项 ContainerInfo ，以允许使用容器化工具，比如Docker，来运行任务或者执行器。

以任务方式运行 Docker 镜像，在TaskInfo中必须设置 命令行和container field。因为Docker Containerizer需要使用伴随的命令来加载docker镜像。ContainerInfo必须设类型为Docker，DockerInfo设为所需要加载的docker镜像。

以执行器方式运行Docker镜像，在TaskInfo中必须设置 ExecutorInfo 包含类型为docker的ContainerInfo。CommandInfo 会被用来启动执行器。注意，当docker镜像被当作Mesos执行器加载，会在它启动时注册为被控。

容器工具的作用方式

docker容器工具会把 任务和执行器的加载和销毁命令转换为Docker CLI命令。

docker容器工具在把容器当任务加载时，会：

1. 把所有 CommandInfo指定的文件载入沙箱。

2. 从远程仓库里拉取docker镜像。

3. 用docker执行器运行docker镜像，把沙箱路径指向Docker容器，将目录指向设为环境变量MESOS_SANDBOX的值。执行器同时会把容器日志不断地发往沙箱里的stdout/stderr文件。

4. 在容器退出或者销毁后，停止或移除docker容器。

docker容器工具通过标示 前缀为 mesos- 后附着被控端slave id （比如mesos-slave1-abcdefghji）的命名规则来加载、停止和销毁容器，并认为所有mesos-前缀开头的容器都是在被控端管理的，可以按需被停止或结束。

当以执行器来加载docker容器的时候，和上面略有不同：不用加载命令行执行器，只需要知道dokcer容器执行器的进程号（PID）。

Note：目前Docker镜像的网络使用默认的主机连接方式，以方便让docker镜像以执行器方式运行

。

容器工具也支持强制拉取镜像，当关闭这功能时，若主机上没有镜像时，docker镜像将只被重新更新。

私有Docker仓库

可在 .dockercfg 中指定 URI地址和登录信息，来使用 Docker私用库。 .dockercfg 会被docker容器工具拉入沙箱中，并设为HOME环境变量，以使docker CLI自动应用该配置。

CommandInfo运行Docker镜像

运行docker镜像有两种方式：默认命令行和自设命令入口。

若要使用默认命令行来载入 Docker 镜像的方式(比如 docker run image), 那么CommandInfo配置项则不能设值。因为若设置了，则会覆盖掉默认命令。

若使用自定义命令入口载入docker镜像的方式，CommandInfo的shell项必须设为 false。因为若设为true，docker容器工具将会以 /bin/sh -c 来运行自设命令，同时也会被当作运行镜像的参数。

在slave节点重启时恢复docker容器

Docker容器化工具支持在slave节点重启的时候恢复Docker容器的功能（不论该slave节点是否运行在Docker容器中运行与否）。

当启用 docker*mesos*image配置项的时候，Docker容器化工具会认为自己容器中运行，从而更变其相应的恢复和加载容器机制。

外部容器化工具

External Containerizer 外部容器化工具

• EC = external containerizer.外部容器化管理器，Mesos slave节点的一个部分，提供给外部插件执行容器化操控的API。

• ECP = external containerizer program. 外部容器化工具，指与容器化系统（例如：Docker）接口，实际实现容器化的外部插件。

容器化

概况

外部容器化管理器 会在 命令行中 执行 外部容器化工具，以命令形式传递参数给外部容器化工具，额外的数据则通过 stdin 和 stdout 交换。

若所有命令都可执行，外部容器化工具将不会退出。如果退出，则表示可能出错了。下文将会列出把所有 外部容器化工具需要实现的功能的概况，及其机制示意图。

外部容器化工具 使用stderr来获得状态和调试信息，这些信息被记录在文件里，详见 Enviroment:

Sandbox(null).

调用和通信示意图

外部容器化工具 通过命令行，实现下面所述的功能。

很多 外部容器化工具 会通过 stdin 传输一个 protobuf 信息。

有些 外部容器化工具 还会通过 stdout 传回一个结果 protobuf 信息。

所有的 protobuf 信息都会把其原有长度放在前缀里——这叫 "Record-IO"-格式。

详见Record-IO De/Serializing示例(null).

命令	传入INPUT-PROTO	结果RESULT-PROTO	
`launch`	`containerizer::Launch`	无	
`update`	`containerizer::Update`	无	
`usage`	`containerizer::Usage`	`mesos::ResourceStatistics`	
`wait`	`containerizer::Wait`	`containerizer::Termination`	
`destroy`	`containerizer::Destroy`	无	
`containers`	无	`containerizer::Containers`	
`recover`	无	无	

命令顺序

命令的执行顺序一般不用特别在意，但有一个例外：当载入一个任务时，外部容器管理器将在特定容器上，先接收到launch，并返回后，容器才可以运行其他的命令。

使用实例

外部容器化管理器和外部容器工具 加载任务概况

• 外部容器化管理器在 外部容器化工具中调入执行launch操作。

• 外部容器化工具将通过 stdin 收到containerizer::Launch protobuf 信息

• 外部容器化管理器因此可确认执行器已经开始工作。

Note launch操作不会等待执行结果，而是直接返回 - 外部容器化工具里通过fork-exec实现。

• 外部容器化管理器在 外部容器化工具里执行等待wait操作。

• 外部容器化工具将通过 stdin 收到containerizer::Wait protobuf 信息

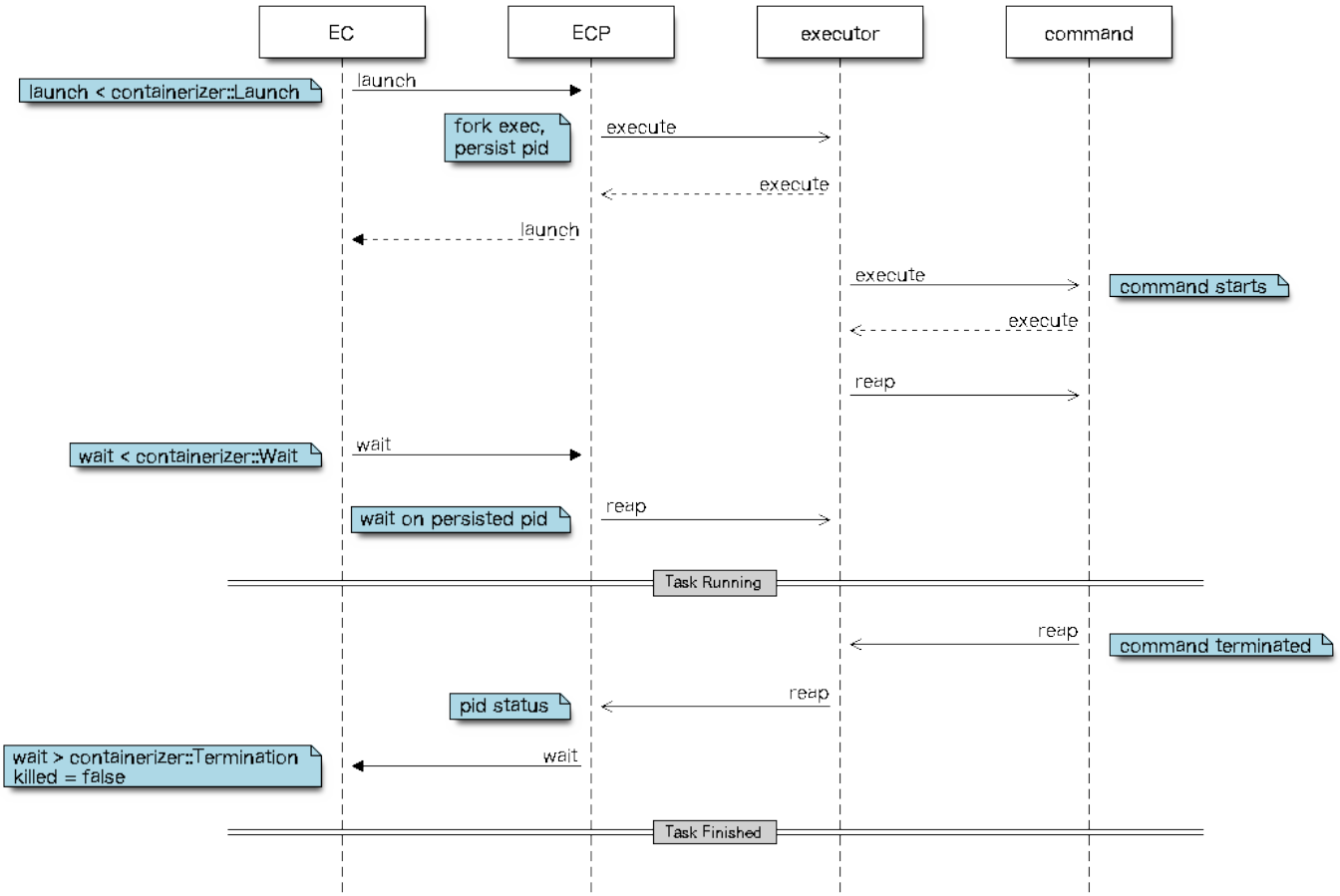
• 外部容器化工具将等待，直到载入的命令执行有结果。——可在外部容器化工具通过waitpid实现。

• 当命令执行有结果了，外部容器化工具将通过 stdout发出一个 containerizer::Termination protobuf信息，然后返回外部容器化管理器

容器生命周期示意图

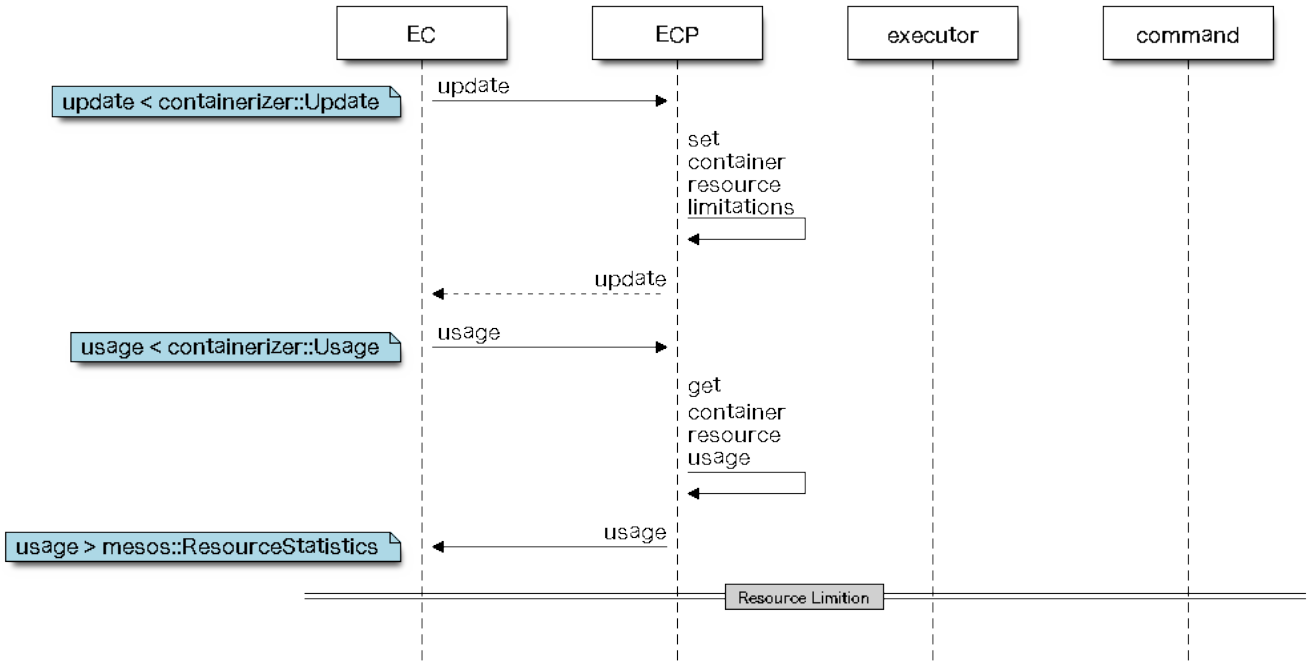
容器加载

仔细观察 从已经载入的容器启动，直到其结束整个状态。



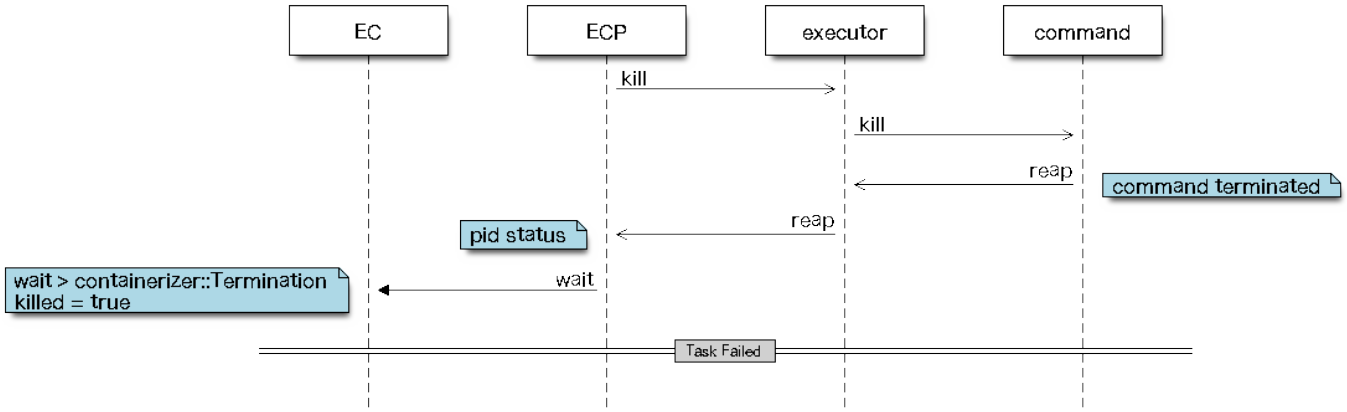
运行容器

容器载入后，slave节点将不再认为其仍然处于命令行环境中，外部容器化工具里的容器，其整个生命周期里都将被下面的命令操控。注：命令顺序不重要。



资源限额

当容器是活动状态时，可通过 外部容器化工具 的分隔机制 设置资源限额（比如 内存限额）。



Slave节点还原 概况

- Slave节点可以通过运行状态锚点进行还原。
- 外部容器化管理器通过 外部容器化工具 调用命令recover，执行还原操作。 - 操作的结果将不返回任何 protobuf 的信息。
- 必要时，外部容器化工具会利用 自身failover 机制恢复其内部状态。
- 在recover命令返回后，外部容器化管理器将调用外部容器化工具里的containers命令。
- 外部容器化工具会返回所有当前活动状态的容器列表

Note 所有的容器信息可以通过外部容器化工具获得，但部分信息slave节点可能并不知道（比如slave节点若在加载后的等待时间前或之中故障）。 - 这些不为slave节点管理的容器可视为孤机容

器。

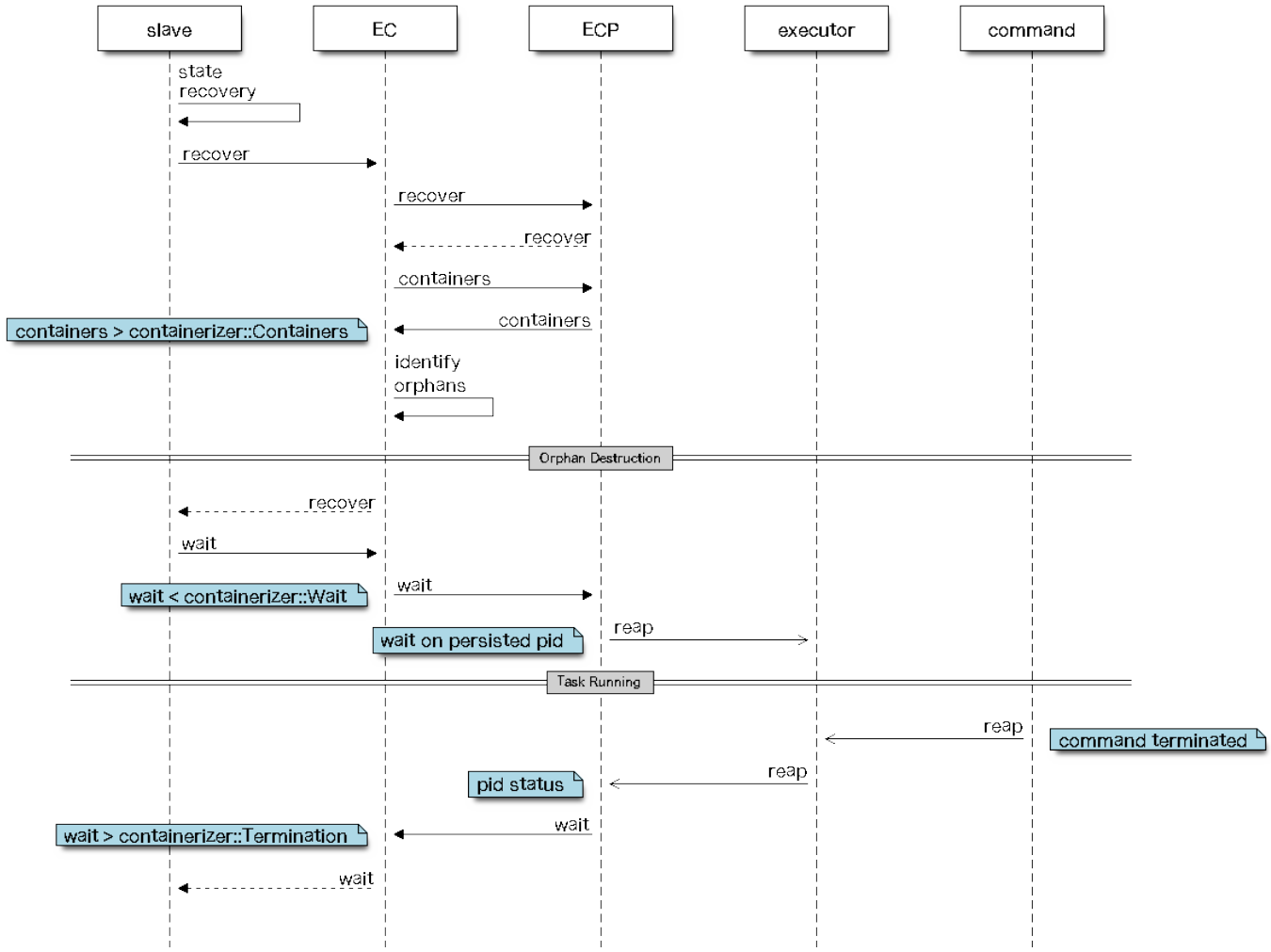
• 外部容器化管理器会对比 slave节点已知的容器列表和完整的Containers命令返回的容器列表，对于标记为孤机的容器，slave节点将在调用等待 wait 命令之后，执行destroy销毁之。

• slave节点将在 外部容器化工具里通过 外部容器化管理器 对所有已经恢复的容器调用等待 wait命令。在生命周期所有其他命令执行完毕后， wait 才退出。

Slave节点还原流程示意图

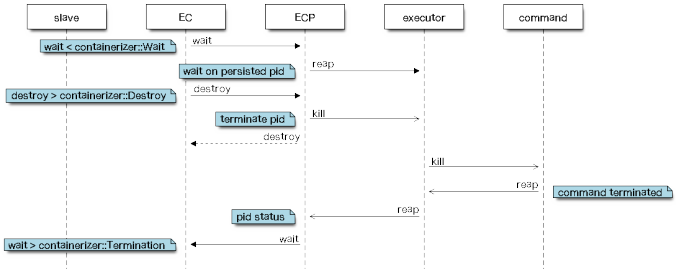
还原

当容器处于活动状态时，slave节点故障。



孤机销毁

外部容器化管理器标记的孤机，处于虽然活动但无法被slave节点恢复的状态，最终将被销毁。



命令详解

加载launch

启动容器化执行器

执行器载入任务 的所有必须的信息都通过该命令获取。 此命令不会等待执行器或命令返回信息。wait 才会等待程序执行完成返回信息后才退出。

```
launch < containerizer::Launch
```

此操作会通过stdin 收到 containerizer::Launch protobuf 如下：

```
/**
 * Encodes the launch command sent to the external containerizer
 * program.
 */
message Launch {
  required ContainerID container_id = 1;
  optional TaskInfo task_info = 2;
  optional ExecutorInfo executor_info = 3;
  optional string directory = 4;
  optional string user = 5;
  optional SlaveID slave_id = 6;
  optional string slave_pid = 7;
  optional bool checkpoint = 8;
}
```

此操作不会通过stdout返回任何数据。

等待wait

从终止的执行器处获得信息

用于中止执行器或命令，此命令会阻塞直至执行器或命令被中止。

```
wait < containerizer::Wait > containerizer::Termination
```

此操作会通过stdin 收到 containerizer::Wait protobuf 如下：

```
/**
```

```
 * Encodes the wait command sent to the external containerizer
 * program.
 */
message Wait {
  required ContainerID container_id = 1;
}
```

此操作会通过stdout 返回 containerizer::Termination protobuf 如下：

```
/**
 * Information about a container termination, returned by the
 * containerizer to the slave.
 */
message Termination {
  // A container may be killed if it exceeds its resources; this will
  // be indicated by killed=true and described by the message string.
  required bool killed = 1;
  required string message = 2;

  // Exit status of the process.
  optional int32 status = 3;
}
```

只有当 容器化工具或者 底层隔离器通过终止任务进程来强制使用限制时(例如：任务使用内存超过内存使用限额)，终止参数 killed才会被设置。

更新update

更新容器的资源限额

用于给指定的容器发送新的资源配额。新的资源配额会改变容器内任务的处理速度。

```
update < containerizer::Update
```

此操作会通过stdin 收到 containerizer::Update protobuf 如下：

```
/**
 * Encodes the update command sent to the external containerizer
 * program.
 */
message Update {
```

```
  required ContainerID container_id = 1;
  repeated Resource resources = 2;
}
```

此操作不会通过stdout返回任何数据。

使用情况usage

采集容器任务的资源使用数据

用于获知指定容器当前资源使用情况。

```
usage < containerizer::Usage > mesos::ResourceStatistics
```

此操作会通过stdin 收到 containerizer::Usage protobuf 如下：

```
/**
 * Encodes the usage command sent to the external containerizer
 * program.
 */
message Usage {
  required ContainerID container_id = 1;
}
```

此操作会通过stdout 收到 containerizer::ResourceStatistics protobuf 如下：

```
/*
 * A snapshot of resource usage statistics.
 */
message ResourceStatistics {
  required double timestamp = 1; // Snapshot time, in seconds since the Epoch.

  // CPU Usage Information:
  // Total CPU time spent in user mode, and kernel mode.
  optional double cpus_user_time_secs = 2;
  optional double cpus_system_time_secs = 3;

  // Number of CPUs allocated.
  optional double cpus_limit = 4;

  // cpu.stat on process throttling (for contention issues).
```

```
  optional uint32 cpus_nr_periods = 7;
  optional uint32 cpus_nr_throttled = 8;
  optional double cpus_throttled_time_secs = 9;

  // Memory Usage Information:
  optional uint64 mem_rss_bytes = 5; // Resident Set Size.

  // Amount of memory resources allocated.
  optional uint64 mem_limit_bytes = 6;

  // Broken out memory usage information (files, anonymous, and mmaped files)
  optional uint64 mem_file_bytes = 10;
  optional uint64 mem_anon_bytes = 11;
  optional uint64 mem_mapped_file_bytes = 12;
}
```

销毁destroy

终止容器化执行器

比较少用，有点像被控关机，不过是在slave节点出现故障切换的时候。详见Slave Recovery。

```
destroy < containerizer::Destroy
```

此操作会通过stdin 收到 containerizer::Destroy protobuf 如下：

```
/**
 * Encodes the destroy command sent to the external containerizer
 * program.
 */
message Destroy {
  required ContainerID container_id = 1;
}
```

此操作不会通过stdout返回任何数据。

容器containers

获取所有活动的容器识别号

返回所有活动状态的容器识别号。

```
containers > containerizer::Containers
```

此操作不会通过stdin接收任何额外数据。

此操作会通过stdout 传回 containerizer::Containers protobuf 如下：

```
/**
 * Information on all active containers returned by the containerizer
 * to the slave.
 */
message Containers {
  repeated ContainerID containers = 1;
}
```

还原recover

外部容器化工具 状态恢复

可以让外部容器化工具对其本身进行状态恢复。

若 外部容器化工具通过 文件系统来使用状态检查锚点，那就可以方便的将该状态信息进行反序列化。详见slave节点还原概况(null)

```
recover
```

此操作不会通过stdin收到任何数据。

此操作不会通过stdout返回任何数据。

Protobuf 消息定义

上面所提及的protobuf和消息可通过下面方式查看详细信息：

• containerizer::XXX 是在 include/mesos/containerizer/containerizer.proto 中定义的。

• mesos::XXX 是在 include/mesos/mesos.proto中定义的。

环境变量

沙箱

沙箱环境，由两部分构成：

执行器会通过 cd 进入工作目录，同时 stderr 重定向进执行器的"stderr"日志文件。

Note 不是**所有的** 容器化调用 都有完整的沙箱环境。

附加的环境变量

在外部容器化工具里，有一些额外的环境变量可供设置：

• MESOS*LIBEXEC*DIRECTORY = mesos-executor, mesos-usage, ...等的路径。

This information is always present.

• MESOS*WORK*DIRECTORY = slave节点的工作目录。这用于区分不同的slave实例。

This information is always present.

Note 这两项设置对于将一批容器绑定于一个slave节点中十分有用，且可以在需要的时候正确的还原。

• MESOS*DEFAULT*CONTAINER*IMAGE = 默认由slave节点的`default*container_image提供的镜像。这变量只有当调用加载命令launch`时可用。

调试

增强细化的日志

在mesos启动slave节点时，在其命令前面添加 GLOG 级别，以便显示更多 外部容器管理工具的状态信息。级别可设置为大于或等于2。

GLOG_v=2 ./bin/mesos-slave --master=[...]

外部容器工具 stderr 日志

外部容器工具将从执行器的stderr日志文件获得其本身 stderr 日志，可通过增强细化日志(null) 展开具体位置。

日志输出示例：

```
I0603 02:12:34.165662 174215168 external_containerizer.cpp:1083] Invoking external
containerizer for method 'launch'
I0603 02:12:34.165675 174215168 external_containerizer.cpp:1100] calling:
[/Users/till/Development/mesos-till/build/src/test-containerizer launch]
I0603 02:12:34.165678 175824896 slave.cpp:497] Successfully attached file
'/tmp/ExternalContainerizerTest_Launch_lP22ci/slaves/20140603-021232-16777343-
51377-7591-0/frameworks/20140603-021232-16777343-51377-7591-
0000/executors/1/runs/558e0a69-70da-4d71-b4c4-c2820b1d6345'
I0603 02:12:34.165686 174215168 external_containerizer.cpp:1101] directory:
/tmp/ExternalContainerizerTest_Launch_lP22ci/slaves/20140603-021232-16777343-
51377-7591-0/frameworks/20140603-021232-16777343-51377-7591-
```

```
0000/executors/1/runs/558e0a69-70da-4d71-b4c4-c2820b1d6345
```

容器工具stderr输出的目录可在 stderr 文件里找到（文件目录参见上方最后一行日志）。

```
cat /tmp/ExternalContainerizerTest_Launch_lP22ci/slaves/20140603-021232-16777343-51377-7591-0/frameworks/20140603-021232-16777343-51377-7591-0000/executors/1/runs/558e0a69-70da-4d71-b4c4-c2820b1d6345/stderr
```

附录

Record-IO Proto 示例：加载

record-io格式化的protobuf形式如下：

name: offset

• length: 00 - 03 = record length in byte

• payload: 04 - (length + 4) = protobuf payload

Example length: 00000240h = 576 byte total protobuf size

十六进制dump示例：

```
   00000000:  4002 0000 0a26 0a24 3433 3532 3533 6162 2d64 3234 362d 3437  :@....&.$435253ab-d246-47
   00000018:  6265 2d61 3335 302d 3335 3432 3034 3635 6438 3638 1a81 020a  :be-a350-35420465d868....
   00000030:  030a 0131 2a16 0a04 6370 7573 1000 1a09 0900 0000 0000 0000  :...1*...cpus..........
   00000048:  4032 012a 2a15 0a03 6d65 6d10 001a 0909 0000 0000 0000 9040  :@2.**...mem.........@
   00000060:  3201 2a2a 160a 0464 6973 6b10 001a 0909 0000 0000 0000 9040  :2.**...disk.........@
   00000078:  3201 2a2a 180a 0570 6f72 7473 1001 220a 0a08 0898 f201 1080  :2.**...ports..".......
   00000090:  fa01 3201 2a3a 2a1a 2865 6368 6f20 274e 6f20 7375 6368 2066  :..2.*:*.(echo 'No such f
   000000a8:  696c 6520 6f72 2064 6972 6563 746f 7279 273b 2065 7869 7420  :ile or directory'; exit
   000000c0:  3142 2b0a 2932 3031 3430 3532 362d 3031 3530 3036 2d31 3637  :1B+.)20140526-015006-167
   000000d8:  3737 3334 332d 3535 3430 332d 3632 3536 372d 3030 3030 4a3d
```

```
:77343-55403-62567-0000J=
   000000f0:  436f 6d6d 616e 6420 4578 6563 7574 6f72 2028 5461 736b 3a20
:Command Executor (Task:
   00000108:  3129 2028 436f 6d6d 616e 643a 2073 6820 2d63 2027 7768 696c  :1) (Command: sh -c 'whil
   00000120:  6520 7472 7565 203b 2e2e 2e27 2952 0131 22c5 012f 746d 702f  :e true ;...')R.1"../tmp/
   00000138:  4578 7465 726e 616c 436f 6e74 6169 6e65 7269 7a65 7254 6573
:ExternalContainerizerTes
   00000150:  745f 4c61 756e 6368 5f6c 5855 6839 662f 736c 6176 6573 2f32
:t_Launch_lXUh9f/slaves/2
   00000168:  3031 3430 3532 362d 3031 3530 3036 2d31 3637 3737 3334 332d
:0140526-015006-16777343-
   00000180:  3535 3430 332d 3632 3536 372d 302f 6672 616d 6577 6f72 6b73
:55403-62567-0/frameworks
   00000198:  2f32 3031 3430 3532 362d 3031 3530 3036 2d31 3637 3737 3334
:/20140526-015006-1677734
   000001b0:  332d 3535 3430 332d 3632 3536 372d 3030 3030 2f65 7865 6375  :3-55403-62567-0000/execu
   000001c8:  746f 7273 2f31 2f72 756e 732f 3433 3532 3533 6162 2d64 3234
:tors/1/runs/435253ab-d24
   000001e0:  362d 3437 6265 2d61 3335 302d 3335 3432 3034 3635 6438 3638  :6-47be-a350-35420465d868
   000001f8:  2a04 7469 6c6c 3228 0a26 3230 3134 3035 3236 2d30 3135 3030
:*.till2(.&20140526-01500
   00000210:  362d 3136 3737 3733 3433 2d35 3534 3033 2d36 3235 3637 2d30  :6-16777343-55403-62567-0
   00000228:  3a18 736c 6176 6528 3129 4031 3237 2e30 2e30 2e31 3a35 3534
::.slave(1)@127.0.0.1:554
   00000240:  3033 4000
```

Record-IO De/Serializing 示例

用Python来发送和接收 record-io 格式化信息

• 可在 src/examples/python/test_containerizer.py 见下面示例源码*

```
    # Read a data chunk prefixed by its total size from stdin.
    def receive():
        # Read size (uint32 => 4 bytes).
        size = struct.unpack('I', sys.stdin.read(4))
        if size[0] <= 0:
            print >> sys.stderr, "Expected protobuf size over stdin. " \
```

```
                    "Received 0 bytes."
        return ""

    # Read payload.
    data = sys.stdin.read(size[0])
    if len(data) != size[0]:
        print >> sys.stderr, "Expected %d bytes protobuf over stdin. " \
                    "Received %d bytes." % (size[0], len(data))
        return ""

    return data

# Write a protobuf message prefixed by its total size (aka recordio)
# to stdout.
def send(data):
    # Write size (uint32 => 4 bytes).
    sys.stdout.write(struct.pack('I', len(data)))

    # Write payload.
    sys.stdout.write(data)
```

框架认证

认证

Mesos 在0.15.0 版本增加了 应用框架的认证支持。在0.19.0 版本增加了 slave节点认证。认证功能用于确保只有信任实体可以与 mesos集群进行交互。

在Mesos中，有三种情况需要用到认证功能：

1. 应用框架需要注册到Master节点时。

2. Slave节点可用资源注册到Master节点时。

3. 限制对卸载的endpoint进行访问时。

工作方式

Mesos认证引擎：Cyrus SASL(null) 库，非常灵活的一个认证框架，可双向认证，支持多种认证机制（ANONYMOUS, PLAIN, CRAM-MD5, GSSAPI 等）。Mesos默认使用 CRAM-MD5 认证，用户可自定义其他认证方式。

CRAM-MD5 用的是 **principal 和 secret** 值对，principal 用于表示 应用框架的**身份**（注意，身份和应用框架*用户*、应用框架角色不一样，应用框架用户指 运行应用框架执行器的帐号，应用框架角色用于确定应用资源使用权限。）。

配置

配置选项(null)，如下：

Master节点

• --[no-]authenticate 设定是否开启应用框架认证。若开启，则只有经过认证的应用框架可注册到Master节点，若关闭，则未认证的应用框架也可以注册。

• --[no-]authenticate_slaves 设定是否开启Slave节点认证。若开启，则所有的Slave节点只有经过认证才可注册到Master节点，若关闭，则未认证的Slave节点也可以注册。

• --authenticators 设定认证方式，默认 crammd5,可通过--modules指定自己想用的其他认证方式。

• --credentials 设定证书列表存放目录。证书列表可以是 纯文本也可以是json格式，内容根据认证方式不同而不同，若设定好了，则不管设定的是什么认证方式，证书在卸载的endpoint里仍然都有效。

Slave节点

• --authenticatee 设定认证方式，默认 crammd5。

• --credential 证书列表，和Master节点类似，不同的是，这里只能有一个证书。

CRAM-MD5 示例

1. 在Master节点创建 证书列表，内容格式如下：

```
principal1 secret1
principal2 secret2
```

1. 在Master节点，启动 证书验证服务：（假设证书文件存放在 ~/credentials ）：

```
./bin/mesos-master.sh --ip=127.0.0.1 --work_dir=/var/lib/mesos --authenticate --authenticate_slaves --credentials=~/credentials
```

1. 在Slave节点，新建 证书列表，内容格式如下，只有一行，假设存放在 ~/slave_credential：

```
principal1 secret1
```

1. 在Slave节点，启动 证书验证服务：若运行成功，Slave节点就获得了Master节点认证。

```
./bin/mesos-slave.sh --master=127.0.0.1:5050 --credential=~/slave_credential
```

1. 应用框架：若开启认证功能，则所有应用，也都需要到Mesos Master节点处获得认证。应用的认证方式：当一个证书对象 构建的时候，调度器 调用相关驱动 处理 认证。

可使用测试应用来测试 Mesos的认证框架，如下：

```
    MESOS_AUTHENTICATE=true DEFAULT_PRINCIPAL=principal2
DEFAULT_SECRET=secret2 ./src/test-framework --master=127.0.0.1:5050
```

框架授权

授权

Mesos 0.20.0 起，可对应用框架进行授权管理，包括：

1. 应用框架使用 授权的roles的角色进行 注册或重注册。
2. 应用框架以 授权的 users的用户加载任务和执行器。
3. 授权的principals可通过 HTTP 端传入 "/shutdown" 来关闭应用框架。

授权列表

授权是通过 授权列表（ Access Control Lists (ACLs)） 来实现的。上面三种情形在 授权列表中都有对应的设置，文件是JSON格式，详见 mesos.proto(null) 。

每项授权列表设置了 一系列的 Subjects可在Objects执行 Action。

目前支持的操作Actions：

1. register_frameworks : 应用注册
2. run_tasks : 运行任务和执行器
3. shutdown_frameworks : 关闭应用

目前支持的 Subjects：

1. principals

• 应用名称(用于 register_frameworks 和 `run_tasks`操作)

• 用户名 (用于shutdown_frameworks 操作)

目前支持的 Objects：

1. roles : 应用可以注册的资源操控角色(用于 "register_frameworks" 操作)
2. users : 可加载任务和执行器的用户(用于run_tasks 操作)
3. framework_principals : 可执行 HTTP POST 关闭操作的应用名 (用于 `shutdown_frameworks`操作).

> NOTE: Subjects和Objects都可用特殊值 ANY 或NONE。

工作方式

Mesos master节点通过匹配 授权列表来 确定一个请求授权与否。

例如，一个应用框架注册到master节点的时候，授权列表里的"register_frameworks"会检查该应用框架FrameworkInfo.principal是否有对应的角色FrameworkInfo.role授权。若无，该应用框架将不允许注册，将收到Error消息（中断资源需求计划）。

同样的，当一个应用框架加载一个任务，授权列表里的"run*tasks"会检查该应用框架FrameworkInfo.principal是否有对应的用户user授权。若无，则无法加载，并收到一个丢失任务的信息`TASK*LOST`。

同样的，当一个 用户或应用框架 想通过 "/teardown" HTTP 关闭master节点上的一个应用框架时，授权列表里的"shutdown_frameworks"被检查是否该principal有相应授权。若无，则无法关闭，并收到一个没有获得授权的消息Unauthorized。

需要注意事项：

1. 授权表是有顺序的，也就是说，第一个匹配的授权起决定作用。

2. 若无匹配的授权，该请求将根据 ACLs.permissive设定进行授权，默认是 true，也就是说没有匹配授权的请求当已授权处理。

设置示例

1. 应用foo 和 bar可以用户alice的身份运行任务。

```
{
  "run_tasks": [
              {
                "principals": { "values": ["foo", "bar"] },
                "users": { "values": ["alice"] }
              }
            ]
}
```

1. 任何应用框架都以身份guest运行任务。

```
{
  "run_tasks": [
              {
                "principals": { "type": "ANY" },
                "users": { "values": ["guest"] }
              }
            ]
}
```

1. root身份不允许运行任何应用。

```
{
```

```
  "run_tasks": [
                {
                  "principals": { "type": "NONE" },
                  "users": { "values": ["root"] }
                }
              ]
}
```

1. 应用只能用guest身份运行任务，其他身份不行。

```
{
  "run_tasks": [
                {
                  "principals": { "values": [ "foo" ] },
                  "users": { "values": ["guest"] }
                },
                {
                  "principals": { "values": [ "foo" ] },
                  "users": { "type": "NONE" }
                }
              ]
}
```

1. 应用foo可注册 analytics和 ads 角色

```
{
  "register_frameworks": [
                        {
                          "principals": { "values": ["foo"] },
                          "roles": { "values": ["analytics", "ads"] }
                        }
                      ]
}
```

1. 只有应用框架 foo可注册analytics角色，其他不行。

```
{
  "register_frameworks": [
                        {
                          "principals": { "values": ["foo"] },
                          "roles": { "values": ["analytics"] }
                        },
                        {
                          "principals": { "type": "NONE" },
                          "roles": { "values": ["analytics"] }
                        }
                       ]
}
```

1. 应用框架foo只能注册为analytics角色，不允许其他角色。

```
{
  "permissive" : false,

  "register_frameworks": [
                        {
                          "principals": { "values": ["foo"] },
                          "roles": { "values": ["analytics"] }
                        }
                       ]
}
```

1. 只有 ops可通过 HTTP /teardown 关闭其他应用。

```
{
  "permissive" : false,

  "shutdown_frameworks": [
                        {
                          "principals": { "values": ["ops"] },
                          "framework_principals": { "type": "ANY" }
                        }
                       ]
}
```

开启授权

通过在主控增加一个acls开启授权。

• acls： 格式是JSON格式的字符串或者是使用JSON的文件路径。可以绝对路径也可以相对路径，如：<code>file:///path/to/file</code> 或 <code>/path/to/file</code>。

在mesos.proto的ACL的protobuf中，查看可用格式。

详情可用命令 ./mesos-master.sh --help 查看

应用框架速率限额

应用框架速率限额

应用框架速率限额限额是Mesos 0.20.0开始引入的功能。

什么是应用框架资源速率限额

在多应用框架运行环境中，可通过对不重要的应用框架（比如开发，批处理）设置速率限额，来保障需要资源高消费high-SLA的应用框架（比如生产、服务）运转。

Mesos集群使用者可以通过设置 qps（每秒查询次数）的值来达到为了一个特定应用框架发送的的信息进行节流的目的。可以对每个应用（通过principle主键来唯一指定）进行单独设置，也可以以群组为单位来进行（下文没有注明的情况则都是以逐个为例）。详见RateLimits ProtoBuf definition(null) 和下文配置说明。 ）当框架消息超过 qps设定速率时，master节点将不会 处理 额外的消息。超出限额的消息会被保存在master节点内存中。

速率限额配置

下面示例：JSON格式，是主控设置里 --rate_limits的值。

```
{
  "limits": [
    {
      "principal": "foo",
      "qps": 55.5
      "capacity": 100000
    },
    {
      "principal": "bar",
      "qps": 300
    },
    {
```

```
      "principal": "baz",
    }
  ],
  "aggregate_default_qps": 333,
  "aggregate_default_capacity": 1000000
}
```

这示例里，应用框架 foo的限额通过 qps 和 capacity 来配置，应用框架 bar 则被设置为capacity可无限制使用，而 baz则根本没有限额设定。当有第四个qux应用或者有一个连接到master节点的没有主键的应用时，其限额将使用默认的 aggregate*default*qps 和 aggregate*default*capacity 设定。

配置详细说明

下面解释配置文件中各个部分的含义。

• **principal**: (必须) 对于需要进行速率节流或显示不进行限制的应用框架的唯一识别名。

• 必须是应用 FrameworkInfo.principal里的名字相匹配。详见 definition(null)。

• 可给多个应用框架一个同样的 唯一识别名（比如一些 Mesos应用框架会为每项任务新开一个同样的应用框架实例）。若如此，qps的设置是指所有这个识别名下应用框架的总额度。

• **qps**: (可选) 每秒查询次数。

• 若设置了此项，则主控将不会对超过这个频率的信息进行处理。一般来说，master节点 实际的查询速率会比设置值低，尤其是设置了过高速度的。

• 当在limits设置了限额，但没有加入 qps 项时，表示 qps 将无限额。

• **capacity**: (可选) 设置应用框架 可在master节点放置多少 待处理消息（也就是排队等待资源分配）。 若无设置，则为无限制。注意：如果设置过多或者无限额，可能导致排队消息过多，导致master节点内存溢出。

• NOTE: 若 qps 没有设置， capacity 将被忽略。

• **aggregatedefaultqps** 和 **aggregatedefaultcapacity** 设置默认值，那些没有列入限额设置的应用框架都将使用默认值。

• 没有加入限额设置的应用框架，其查询速率和队列数将同时使用默认值。

• 若 aggregate*default*qps 没有设置，则aggregate*default*capacity 的设置将被忽略。

• 若应用在限额列表里，该配置项没有进行设置，则未指定的应用框架将无速率和队列限制。我们建议还是进行显示的设置以免未指定的应用框架使master节点过载崩溃。

设置 应用框架的速率限额

监控应用流量

当应用框架向master节点注册时，master节点会把应用框架发来和处理的所有消息都纳入计数，具体信息放在http://<master>/metrics/snapshot里。例如，应用框架 foo有两个消息计数器 frameworks/foo/messages_received和`frameworks/foo/messages_processed`，一个是接收

到的消息，一个是处理过的消息。若无限额设定，这两个数字差异会很小甚至一样，(因为消息都是立刻得到了处理)

，而当设定了速率限额的时候，这两个数则会根据限额设定而起变化。

若持续观察计数器，可获知消息到达的速度和消息被处理的速度。通过该信息可以推测应用框架在网络流量上的特性。

配置速率限额

应用框架的速率限额是用来避免 low-SLA 的应用框架没有通过精细的流量和任务所需要的资源的估计儿消耗了大量集群资源。

对于此类框架，可先使用较大的 qps 值来设置。而当对high-SLA应用设置高优先级的时候，也会对low-SLA形成事实上的限额。因为他们的消息会立刻得到处理。

如何知道master节点的可处理消息队列处理能力capacity 呢？首先，要知道master节点主线程的内存限额，执行类似任务但不设限额时使用的内存（比如，用 ps -o rss $MASTER_PID 查询）和队列消息的平均大小（队列消息以serialized Protocol Buffers with a few additional fields(null)方式存储），之后在进行汇总配置中的所有的capacity值。

不过这种汇总并不精确，因此，一开始可用小一点的限额，这样可以让master节点和没设限额的应用保有充分的内存空间，以应对可能的突发需求。

处理 "容量超出" 错误

当应用框架 超出了容量 时，会发出"FrameworkErrorMessage"给 应用框架，应用框架将 中止调度器驱动并调用 error()回调函数 (null)，但不会因此强制结束任何任务或调度器本身。应用框架开发者可选择重启或者在其他节点恢复调度实例，来应对消息丢失的后果（除非你的应用框架并不需要知道所有发给maseter节点的消息是否都被处理）。

在0.20.0版本后，我们会迭代的增强该功能。即当此应用开始建立消息队列的时候，master节点会发出告警（MESOS-1664(null)相当于一个soft limit），调度器将自身会根据改告警进行节流行为（来避免错误信息），或是忽略改告警（设计好的的临时突发情况）。

在告警消息被实现之前，不建议使用速率限额功能来对生产级别的应用框架进行速率限额的配置（除非你很清楚错误消息带来的后果）。但可以通过对其他非重要框架进行的限额的设置来保护生产级别的应用框架。并且若不显示的开启该配置，是不会对master节点造成任何影响的。

日志和调试

日志和调试

Mesos使用 Google Logging library(null)，默认把日志写到MESOS_HOME/logs，

其中，MESOS_HOME 是指mesos的安装位置。日志目录可根据通过`log_dir`进行设置。

应用框架则在其所运行的机器的"work"目录记录日志。默认是在MESOS_HOME/work目录下，应用框架的输出会被记录在`slave-X/fw-Y/Z`这样目录的 `stdout` 和 `stderr`的文件中，其中X是slave节点的ID，Y是应用框架的的ID，而每次尝试通过执行器运行应用框架则会产生多个Z级子目录。所有这些文件也可通过slave端守护经进程里的Web界面进行操作。

高可用模式

高可用模式

如果Mesos master节点失效，那么现有的而任务仍然可以继续执行，但是新的资源就无法被分配而导致新的任务无法启动。为了减少这种情况发生的可能性，Mesos设计了高可用的模式，即使用多个master节点，并选取其中一个作为活动的master（成为leader或者leading master），其他的master节点则作为备用来避免leader的崩溃。正果过程同，msos通过使用 Apache ZooKeeper(null) 协调选举和通过master,slave和调度驱动检测leader master节点的健康程度。详见leader选举过程(null)

Note: 本文假设你知道如何 使用启动、运行和使用ZooKeeper运作。ZooKeeper的客户端库内建于标准的Mesos构建。

用法

让Mesos变成高可用模式：

1. 确认集群中 ZooKeeper 已经开启且在运行中。

2. 提供znode路径给在所有的Master、slave、应用框架调度器，设置如下：

```
* 运行mesos-master程序时，附加 `--zk`参数，比如 `--
zk=zk://host1:port1,host2:port2,.../path'

* 运行mesos-slave程序时，附加参数 `--master=zk://host1:port1,host2:port2,.../path`

* 运行应用框架调度器时，在最后两步附加 `--zk`参数。如[应用开发向导](
http://mesos.apache.org/documentation/latest/app-framework-development-
guide/)中所述，SchedulerDriver必须和该路径绑定。
```

这样，Mesos的所有Master节点和Slave节点都将通过ZooKeeper来找出哪一个master节点是当前的leading master。除此之外，就是通常的leading master和slave之间通信了。

查看Scheduler API(null) 了解如何处理leader的变更。

组件失联的处理

当ZooKeeper的监控下中有网络连接中发生了组件（master，slave或scheduler driver）失联的情况时，该组件的主检测程序将发出一个超时事件来告诉所有组件：目前没有了leading master。（注意：当有部分失联时，master节点仍会和slave节点、调度器等保持通信，反之亦然）。下面是失联后组件的状况：

• 失联的Slave节点将无法知道谁是leading Master，所以会无视所有从leader master节点发来的消息来避免执行非leader的master节点发送的命令。当slave节点重新连接到ZooKeeper上时，Zookeeper会告知其谁是现任leader。slave节点根据该信息，停止忽略leader master发来的信息。

• 失联后，master节点将进入无leader状态，无论该master节点失联前是否是leader。

• 当leader从ZooKeeper失联时，将中断所有处理进程。user/developer/administrator会新开一个master实例当备份去尝试重新连接ZooKeeper。

> **Note**: 很多生产环境部署的Mesos集群都会使用进程监控（如systemd和supervisord）来在进程崩溃时自动重启Mesos master节点
>
> * 或者，在master备份还没重新连接上ZooKeeper的时候，可能ZooKeeper已经新选了leader。

• 调度器失联则只会收到一个消息：它们和leader失联了。

当Slave节点失联时：

• Leader master对slave节点的 可用性检查会失败。

• Leader master将标记该Slave节点为失效状态，并把其未完成的工作标记为丢失状态。应用框架开发向导(null)对这些任务的状态有详细说明。

• 被标记为非活动状态的slave节点，将无法完成重新注册，并会被随后通知关闭。

实现细节

Mesos实现了对ZooKeeper选择leader抽象化的两个层次，一个在`src/zookeeper`，另一个在`src/master` (具体细节可查询 `contender	detector.hpp	cpp`)

• 底层的LeaderContender和LeaderDetector 实现了一个松散的通用ZooKeeper选举算法。详见recipe(null)（master小组节点小于3时的无群处理方法）。

• 高一层的MasterContender和MasterDetector包装了ZooKeeper的contender和detector的抽象层，来和ZooKeeper进行数据交互。

• 每个Mesos master节点都同时使用了一个contender和detector，用于选自己为leader或者发现现任leader。单独的detector是必须的，因为需要将从浏览器发来的请求访问任何一个没选成的leader的master节点上的Web界面时，需要将该请求重定向到现任leader master几点上。其他的Mesos部件（比如slave和scheduler）也都使用detector来找到现任leader master，并与其建立连接。

• leader候选组的概念通过Group对象实现。通过队列和错误重试的方式，处理ZooKeeper候选者的注册、注销和监控的流程。在过程中，它会监控Zookeeper会话的事件：

• 连接

• 重连

• 会话过期失效

• Znode的创建、删除、更新

• 超时：在和ZooKeeper失联一段时间后，系统会主动将会话设为过期超时的状态。详见MASTER_CONTENDER_ZK_SESSION_TIMEOUT 和 MASTER_DETECTOR_ZK_SESSION_TIMEOUT。这是因为

ZooKeeper的客户端的库函数只能通过重连来提示会话过期。这些超时设定可在网络设定部分查看
。

操作指南

操作指南

使用进程监视器

Mesos使用fail-fast方法来处理错误：如果发生了一个严重的错误，典型情况下Mesos会退出而不是尝试继续运行在一个可能错误的状态。比如，当Mesos被配置为高可用的情况下，当leading master节点发现它已经从Zookeeper集群中分离，它会终止自己的进程。这是一个安全的预防措施，它保证了之前的leader不会继续在一个不安全的状态下通信。

为了确保这样的错误被合理地处理，典型情况下的Mesos生产部署下是使用进程监视器（supervisor, 如systemd或者supervisord）来检测Mesos进程退出时的事件发生。supervisor可以配置为自动重启失败的进程或者通知集群运维人员来分析这种错误情形。

更改Master集群数目

译者注：集群数目为master quorum

Master决策机制是基于 paxos 共识算法的多副本日志作为存储后端实现的(--registry=replicated_log 是唯一支持的存储后端)。

每个master以日志备份者的身份参与集群组中。--quorum 用于设置master的投票过半数。

下表是master集群的投票过半数和容灾节点数的对应表：

master集群节点数	投票过半数	容灾节点数
1	1	0
3	2	1
5	3	2
...	...	...

2N - 1	N	N - 1

当需要高可用性时，建议使用3或者5个主控。

注意

设置master集群数目时，确保运行中的master节点数目符合上面列表的数字。若多于此数，选举将冲突，日志将损坏！因此建议对运行中的master线程进行准入设置：比如使用添加master节点的主机名白名单。参见MESOS-1546(null)

日志的其他设置，可参见：MESOS-683(null).

扩大master集群节点数目

Mesos集群规模扩大后，相应地需要扩大master集群规模以减少故障率。下面是扩编操作过程示例：从3个变为5个，投票过半数从2个变为3个。

1. 原先是使用 --quorum=2 启用3个master节点。

2. 使用 --quorum=3 重启原先的3个master节点。

3. 使用 --quorum=3 启用另外2个新加入的master节点

若要扩大到N个节点，则需要执行N次上面的操作。

NOTE: 从单master节点变为多master集群的时候，需要清空其副本日志状态并开新的，这将清空所有的持久化存储数据，包括 slave节点，维护信息，限额信息等。示例：把一个变为3个主控。

1. 停用该master节点。

2. 清空副本日志的数据 (在--work_dir目录下的`replicated_log`)。

3. 使用--quorum=2参数开启该master节点和其他两个新的maser节点。

减少master集群节点数目

下面是缩编操作过程示例：从5个变为3个，相应的投票过半数从3变2。

1. 最初是使用--quorum=3参数启用5个master节点集群。

2. 从集群中移除两个master节点，并确保其不被重启（看上面注意事项！），这样只有3个master节点根据--quorum=3的设定在运行。

3. 使用--quorum=2参数重启这三个master节点。

若要缩编到N个，则需要执行N次上面的操作。

替换一个master节点

注意查看上面 注意事项。当一个失效master节点保证不再重新加入master集群的时候，可用安全放心的来启用一个新的*空日志*master节点。

外部访问Mesos master节点

若默认IP（或通过命令行参数 --ip 设置的IP）指向一个内部IP，则外部的程序，比如应用框架调度器，将无法连接到master节点。若不想改ip又想让外部程序访问，可通过设置 --advertise_ip 和 `--advertise_port` 来让外部程序通过该 IP:port 访问。

Mesos 可观测的度量

Mesos可观测的度量

本文将介绍master和slave节点所提供的可观察的度量。通过这些度量可以监控集群的的运行状况。

概括

Mesos master节点与slave节点都提供一系列的统计与度量数据，可用于监控资源使用，提前发现异常。

由Mesos提供的的监控信息包含：可用资源，已用资源，注册的应用框架，活动的slave节点，任务状态。可根据这些信息，可以在监控仪表盘中创建相应的自动示警或者统计图。

度量的类型

Mesos提供两种类型的度量: 累计和计量。

计数器Counters 会跟踪零散发生的事件，并累积起来。这类结果一般是一个数字。比如：失败的任务数，注册的slave节点数。另外，这类统计数字中，某些度量记数的数值变动速度往往更有用。

计量器Gauges 则表示某些度量的瞬间值，比如集群中已用的内存数，已经连接的slave节点数。这类数字经常用于判断一定时间段内某些度量数值是否超出或者低于相对合理的正常水平。

Master节点

Master节点可用的统计度量可从下面连接查看：

```
http://<mesos-master-ip>:5050/metrics/snapshot
```

得到的是一个JSON对象，内含统计项和统计值的键值对。

直观统计

根据组别，把Master节点可用的统计项列出如下：

资源使用情况

下表包含集群中可用的资源总量，现在的使用情况。持续的高资源使用率，表明管理员需要及时给集群添加容量或是集群行为发生异常等。

统计项	解释	类型
`master/cpus_percent`	已分配的cpu比例	计量
`master/cpus_used`	已分配的cpu数	计量
`master/cpus_total`	CPU总数	计量
`master/cpus_revocable_percent`	可回收的CPU比例	计量

`master/cpus_revocable_total`	可回收的CPU数	计量
`master/cpus_revocable_used`	在用的可回收CPU数	计量
`master/disk_percent`	已分配的磁盘空间比例	计量
`master/disk_used`	已分配的磁盘空间大小：单位 MB	计量
`master/disk_total`	磁盘空间总量：单位 MB	计量
`master/disk_revocable_percent`	可回收磁盘空间比例	计量
`master/disk_revocable_total`	可回收磁盘空间大小：单位 MB	计量
`master/disk_revocable_used`	已用的可回收磁盘空间大小：单位 MB	计量
`master/mem_percent`	已分配的内存比例	计量
`master/mem_used`	已分配的内存：单位 MB	计量
`master/mem_total`	内存总量：单位 MB	计量
`master/mem_revocable_percent`	可回收内存比例	计量
`master/mem_revocable_total`	可回收内存大小：单位 MB	计量
`master/mem_revocable_used`	已用的可回收内存大小：单位 MB	计量

Master节点

下面度量包含：Master节点是否被选为leader master节点，运行时长。一段时间无leader表示该集群出现故障：可能是选举功能失效（需要查看ZooKeeper连接是否正常），也可能是Master进程不稳定。

在线运行时间值较低表示该Master节点最近被重启过。

统计项	解释	类型
`master/elected`	当前是否是leader	计量
`master/uptime_secs`	运行时间：单位 秒	计量

系统

下面包含：Master节点的可用资源，和使用情况。Master节点内若持续过高资源使用比例，将导致集群的性能和效率下降。

统计项	解释	类型
`system/cpus_total`	此Master节点的可用CPU数	计量
`system/load_15min`	过去15分钟内的平均负载	计量
`system/load_5min`	过去5分钟内的平均负载	计量
`system/load_1min`	过去1分钟内的平均负载	计量
`system/mem_free_bytes`	可用内存：单位 bytes	计量
`system/mem_total_bytes`	内存总量：单位 bytes	计量

Slave节点

下面包含：Slave节点事件，节点总数，状态。活跃的Slave节点过少，表示这些Slave节点存在故障，或者它们无法连接到Master节点。

统计项	解释	类型
`master/slave_registrations`	Master节点失联时，可以重新加入集群并连回master节点的slave节点数。	计数

`master/slave_removals`	被移除的slave节点数：包括维护需要等各种原因被移除的。	计数
`master/slave_reregistrations`	重新注册的slave节点数	计数
`master/slave_shutdowns_scheduled`	健康监控中被判定失效，计划被移除的slave节点数，但不会立刻移除，因为有移除slave节点速率限制。而当它们真正被移除时，`master/slave_shutdowns_completed`才会累计进去。	计数
`master/slave_shutdowns_cancelled`	取消关机的slave节点数量：当slave节点移除速率限制下被允许slave节点重连，并在移除前发送`PONG`给Master节点的slave节点数。	计数
`master/slave_shutdowns_completed`	已完成关机的slave节点数：已经超过移除slave速率限制的，且已经被移出Master的注册slave节点列表的数目。	计数
`master/slaves_active`	活动的slave节点数量	计量
`master/slaves_connected`	连接中的slave节点数量	计量
`master/slaves_disconnected`	未连接的slave节点数量	计量
`master/slaves_inactive`	非活动中的slave节点数量	计量

应用框架

下面包含：集群中注册的应用框架数。若出现无活动或连接状态下的的应用框架的情况，则一般说明调度器没有注册或者出现故障。

统计项	解释	类型
`master/frameworks_active`	活跃的应用框架数量	计量
`master/frameworks_connected`	已经连接的应用框架数量	计量
`master/frameworks_disconnected`	未连接的应用框架数量	计量
`master/frameworks_inactive`	非活动的应用框架数量	计量
`master/outstanding_offers`	未处理的资源邀约数量	计量

任务

下面包含：活动和中断的任务。若任务丢失频率高，则表明集群有故障。

统计项	解释	类型
`master/tasks_error`	已经失效的任务数量	累计
`master/tasks_failed`	失败的任务数量	累计
`master/tasks_finished`	已经完成的任务数量	累计
`master/tasks_killed`	被杀死的任务数量	累计
`master/tasks_lost`	丢失的任务数量	累计
`master/tasks_running`	运行中的任务数量	累计
`master/tasks_staging`	待处理任务数量	累计
`master/tasks_starting`	开始的任务数量	累计

消息

下面包含：Master节点、Slave节点、应用狂降、执行器间的消息。若消息丢弃严重，表明网络有问题。

统计项	解释	类型
`master/invalid_framework_to_executor_messages`	应用框架传给执行器无效消息的数量	累计
`master/invalid_status_update_acknowledgements`	无效的状态更新通知的数量	累计
`master/invalid_status_updates`	无效的状态更新的数量	累计
`master/dropped_messages`	丢弃消息的数量	累计
`master/messages_authenticate`	验证消息的数量	累计
`master/messages_deactivate_framework`	应用框架没激活消息的数量	累计
`master/messages_exited_executor`	被终止的执行器消息的数量	累计
`master/messages_framework_to_executor`	应用框架传给执行器消息的数量	累计
`master/messages_kill_task`	任务被终止消息的数量	累计

`master/messages_launch_tasks`	任务载入消息的数量	累计
`master/messages_reconcile_tasks`	任务核对消息的数量	累计
`master/messages_register_framework`	应用框架注册消息的数量	累计
`master/messages_register_slave`	slave节点注册消息的数量	累计
`master/messages_reregister_framework`	应用框架重注册消息的数量	累计
`master/messages_reregister_slave`	slave节点重注册消息的数量	累计
`master/messages_resource_request`	请求资源消息的数量	累计
`master/messages_revive_offers`	邀约恢复消息的数量	累计
`master/messages_status_udpate`	状态更新消息的数量	累计
`master/messages_status_update_acknowledgement`	状态更新通知消息的数量	累计
`master/messages_unregister_framework`	应用框架注销消息的数量	累计

`master/messages_unregister_slave`	slave节点注销消息的数量	累计
`master/valid_framework_to_executor_messages`	应用框架传给执行器有效消息的数量	累计
`master/valid_status_update_acknowledgements`	有效状态更新通知消息的数量	累计
`master/valid_status_updates`	有效状态更新消息的数量	累计

事件队列

下面包含：事件队列里的事件类型。

统计项	解释	类型
`master/event_queue_dispatches`	事件队列里调度的数量	计量
`master/event_queue_http_requests`	事件队列里HTTP请求的数量	计量
`master/event_queue_messages`	事件队列里消息传输的数量	计量

注册

下面包含：slave节点注册的读写延迟信息。

统计项	解释	类型
`registrar/state_fetch_ms`	读注册信息的延迟，单位毫秒：ms	计量

`registrar/state_store_ms`	写注册信息的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/max`	最大写注册信息的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/min`	最小写注册信息的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/p50`	写注册信息延迟的中位数，单位毫秒：ms	计量
`registrar/state_store_ms/p90`	写注册信息90%起的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/p95`	写注册信息95%起的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/p99`	写注册信息99%起的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/p999`	写注册信息99.9%起的延迟，单位毫秒：ms	计量
`registrar/state_store_ms/p9999`	写注册信息99.99%起的延迟，单位毫秒：ms	计量

基本告警信息

可用下面的告警来获知集群是否存在错误。

master/uptime_secs is low

master节点最近被重启过。

master/uptime_secs < 60 for sustained periods of time

集群内有个master进程不稳定。

master/tasks_lost is increasing rapidly

集群中的任务丢失数增长较快，可能原因包括硬件故障，应用漏洞，或者Mesos程序的bug。

master/slaves_active is low

slave节点与master节点有连接问题。

master/cpus_percent > 0.9 for sustained periods of time

集群的CPU将不再处理新的排队信息。

master/mem_percent > 0.9 for sustained periods of time

集群的内存将不再存储新的排队信息。

master/elected is 0 for sustained periods of time

目前没有master节点被选为leader。

slave节点

每个slave节点可获得的统计数据可从下面连接查看：

```
http://<mesos-slave>:5051/metrics/snapshot
```

返回得到的是一个JSON对象，内含统计项和统计值的键值对。

直观统计

根据组别，把slave节点可用的统计项列出如下：

资源使用情况

下面包含：slave节点的可用资源量和使用情况。

统计项	解释	类型
`slave/cpus_percent`	已分配的cpu比例	计量
`slave/cpus_used`	已分配的cpu数	计量
`slave/cpus_total`	CPU总数	计量

`slave/cpus_revocable_percent`	可回收的CPU比例	计量
`slave/cpus_revocable_total`	可回收的CPU数	计量
`slave/cpus_revocable_used`	在用的可回收CPU数	计量
`slave/disk_percent`	已分配的磁盘空间比例	计量
`slave/disk_used`	已分配的磁盘空间大小：单位 MB	计量
`slave/disk_total`	磁盘空间总量：单位 MB	计量
`slave/disk_revocable_percent`	可回收磁盘空间比例	计量
`slave/disk_revocable_total`	可回收磁盘空间大小：单位 MB	计量
`slave/disk_revocable_used`	已用的可回收磁盘空间大小：单位 MB	计量
`slave/mem_percent`	已分配的内存比例	计量
`slave/mem_used`	已分配的内存：单位 MB	计量
`slave/mem_total`	内存总量：单位 MB	计量
`slave/mem_revocable_percent`	可回收内存比例	计量
`slave/mem_revocable_total`	可回收内存大小：单位 MB	计量

`slave/mem_revocable_used`	已用的可回收内存大小：单位 MB	计量

slave节点

下面包含：slave节点是否已经注册到master节点与运行时长。

统计项	解释	类型
`slave/registered`	是否已注册到master节点	计量
`slave/uptime_secs`	运行时间：单位 秒	计量

系统

统计项	解释	类型
`system/cpus_total`	此节点的可用CPU数	计量
`system/load_15min`	过去15分钟内的平均负载	计量
`system/load_5min`	过去5分钟内的平均负载	计量
`system/load_1min`	过去1分钟内的平均负载	计量
`system/mem_free_bytes`	可用内存：单位 bytes	计量
`system/mem_total_bytes`	内存总量：单位 bytes	计量

执行器

下面包含：在slave节点上运行的执行器的具体信息

统计项	解释	类型
`slave/frameworks_active`	活动中的应用数	计量
`slave/executors_registering`	注册中的执行器数	计量
`slave/executors_running`	运行中的执行器数	计量

`slave/executors_terminated`	已终止的执行器数	计量
`slave/executors_terminating`	正在终止的执行器数	计量

任务

下面包含：活动和终止的任务信息。

统计项	解释	类型
`slave/tasks_failed`	失败的任务数量	累计
`slave/tasks_finished`	已经完成的任务数量	累计
`slave/tasks_killed`	被杀死的任务数量	累计
`slave/tasks_lost`	丢失的任务数量	累计
`slave/tasks_running`	运行中的任务数量	计量
`slave/tasks_staging`	等待运行的任务数量	计量
`slave/tasks_starting`	开始的任务数量	计量

消息

下面包含：master节点和slave节点间的消息信息。

统计项	解释	类型
`slave/invalid_framework_messages`	无效的应用框架信息的数量	累计
`slave/invalid_status_udpates`	无效的状态更新消息的数量	累计

`slave/valid_framework_messages`	有效的应用框架信息的数量	累计
`slave/valid_status_udpates`	有效的状态更新消息的数量	累计

单容器网络监控和隔离

单容器网络监控和隔离

Linux上的Mesos可支持 单容器网络监控和隔离，每个活动容器的网络统计信息会被写入slave节点的 /monitor/statistics.json 。

网络隔离 功能可避免 单个容器 占用过多可用端口，消耗过多带宽以及过分阻滞其他容器的信息传输等。

网络隔离功能对于大部分运行在slave节点的任务（端口绑定为0，且由内核分配端口）来说使透明的。

安装

容器网络监控和隔离在默认配置下是不开启的。需要在编译时，设置相关配置，并安装相关依赖，才可开启。

关联依赖

单容器网络监控和分隔 只能在 kernel版本大于3.6 的linux 里使用，另外，linux kernel必须包含以下补丁（kernel 3.15后，这些补丁已经包含进内核中）。

- 6a662719c9868b3d6c7d26b3a085f0cd3cc15e64(null)
- 0d5edc68739f1c1e0519acbea1d3f0c1882a15d7(null)
- e374c618b1465f0292047a9f4c244bd71ab5f1f0(null)
- 25f929fbff0d1bcebf2e92656d33025cd330cbf8(null)

slave节点需要的依赖：

- libnl3(null) >= 3.2.26
- iproute(null) >= 2.6.39 is advised for debugging purpose but not required.

若是从源码编译，则需要libnl3开发版：

- libnl3-devel / libnl3-dev(null) >= 3.2.26

编译

开启 单容器网络监控和隔离，需要在 编译配置中设定：

```
$ ./configure --with-network-isolator
$ make
```

配置

节点网络监控和隔离需要在slave节点，加入下面参数 设置：

```
--isolation="network/port_mapping"
```

若编译的时候没有开启 节点网络监控和隔离 支持，则会无法启动，并输出错误信息如下：

```
I0708 00:17:08.080271 44267 containerizer.cpp:111] Using isolation:
network/port_mapping
Failed to create a containerizer: Could not create MesosContainerizer: Unknown or
unsupported
    isolator: network/port_mapping
```

配置网络端口

在没有 网络隔离功能的时候，所有容器都将共用slave节点的IP地址，绑定操作系统允许的任意端口。

而当网络隔离启用的时候，slave节点上的每个容器将有单独的网络链路（基于Linux network namespaces(null)），虽然仍然共用被控的对公IP（这样的好处是不用更改 服务发现机制）。slave节点会根据设定的可用端口范围给容器分配端口，只有端口内的网络链路才可用。

应用若向核心申请分配端口，则会在 容器所设的可用端口中选一个。应用虽然也可以在这个可用端口范围外绑定一个端口，但这样的结果是主机将无视其网络请求，网络链路不通。

Mesos给容器提供两个可用端口范围：

+ Mesos从系统分配到的临时端口ephemeral(null)：用于容器。

+ Mesos分配的非临时端口：用于应用，和cpu、内存等资源类似，需要的时候申请。

另外，主机本身也会需要临时端口用于网络通讯，也就是总共有三个不重叠的端口范围需要在主机设置。

设置主机临时端口范围

用命令sysctl net.ipv4.ip*local*port_range可查看目前主机的临时端口范围。若需要更改，可在`/etc/sysctl.conf`里设置，注意不要设置其他进程使用的端口，设置完后重启，使其生效。

例子如下：把主机临时端口设置为57345到61000了。

```
# net.ipv4.ip_local_port_range defines the host ephemeral port range, by
# default 32768-61000.  We reduce this range to allow the Mesos slave to
```

```
# allocate ports 32768-57344
# net.ipv4.ip_local_port_range = 32768 61000
net.ipv4.ip_local_port_range = 57345 61000
```

设置容器端口范围

容器的临时和非临时端口，使用--resources 来设置。非临时端口是由master节点提供的，将会被分配给应用。

临时端口则会被 slave节点分配，使用 ephemeral*ports*per_container（默认 1024） 分配给容器。每个被控上的容器因为网络资源有限，其数量也是有限的：容器最大可用数量=可用临时端口量/每容器端口数。

```
number of ephemeral_ports / ephemeral_ports_per_container
```

master节点的 --max*executors*per_slave （每slave节点最大执行器数）设定用于避免给slave节点分配过多的执行器而致临时端口超额。

建议把ephemeral*ports*per_container 设为2的幂数，比如512，1024，临时端口则是`ephemeral*ports*per_container` 的倍数。这样可最小化CPU信息传输时的负载。示例：

```
--resources=ports:[31000-32000];ephemeral_ports:[32768-57344] \
--ephemeral_ports_per_container=512
```

容器网络限速

容器对外的网络连接需要限速，以免单个容器占用过多带宽。可在每个容器启动时带 --egress*rate*limit*per*container 参数（单位 bytes/秒）设置速度限制，高于此限速的传输将被延迟处理，TCP协议将会增加间隔时间，降低速度，以使不掉包。

示例如下：

```
--egress_rate_limit_per_container=100MB
```

由于只能在接收到传输后根据拥堵状况再处理限速，因此，Mesos只能限制出口带宽，无法限制入口带宽。

出口带宽流量隔离

如果所有容器都使用同一网络链路，单个容器的网络拥堵，将严重影响其他容器的网络传输。

Mesos里可以使用链路隔离来 让容器间网络传输相互间不受影响：根据容器端口范围来划定专属链路，使用FQ_Codel(null)算法实现。使用 --egress*unique*flow*per*container 设置，示例如下：

```
--egress_unique_flow_per_container
```

综合设置（集合上述配置项）

下面综合网络设置命令包括：开启网络隔离，端口57345-61000给主机当临时端口，端口32768-57344给容器当临时端口，端口31000-32000给应用非临时分配，将容器出口链路设为专属，出口带宽限制为300 Mbits/second (37.5MBytes)

```
mesos-slave \
--isolation=network/port_mapping \
--resources=ports:[31000-32000];ephemeral_ports:[32768-57344] \
--ephemeral_ports_per_container=1024 \
--egress_rate_limit_per_container=37500KB \
--egress_unique_flow_per_container
```

监控容器网络统计数字

Mesos从linux网络监测里获得的每个容器的日志存放到slave节点的/monitor/statistics.json文件中。

容器中的网络工具里，在 statistics 下，可查看如下信息：

统计项	解释	类型
`net_rx_bytes`	收到的流量：单位 bytes	累计
`net_rx_dropped`	接收端的丢包数	累计
`net_rx_errors`	接收端的错误数	累计
`net_rx_packets`	收到的包数	累计
`net_tx_bytes`	发出的流量：单位 bytes	累计
`net_tx_dropped`	发出端的丢包数	累计
`net_tx_errors`	发送端的错误数	累计
`net_tx_packets`	发出的包数	累计

另外Linux Traffic Control(null) 可在 statistics/net*traffic*control_statistics里采集如下信息：限速和过载。

统计项	解释	类型

`backlog`	排队待传输的字节数 [1]	计量
`bytes`	发出的字节数	累计
`drops`	发出端的丢包数	累计
`overlimits`	超限额次数，当超过限额时，每收到一包算一次。[2]	累计
`packets`	发出的包数	累计
`qlen`	排队待传输的包数	计量
`ratebps`	传输速度——每秒字节数：单位 bytes/second [3]	计量
`ratepps`	传输速度——每秒包数：单位 packets/second [3]	计量
`requeues`	因资源占用(比如 kernel locking)而导致的传输失败包数 [3]	累计

[1] Backlog只在 bloat_reduction接口列出。

[2] Overlimits只在 bw_limit 接口列出。

[3] 目前都报为0。

示例：可在slave节点中，输入下面命令，获得如下的结果：

```
$ curl -s http://localhost:5051/monitor/statistics | python2.6 -mjson.tool
[
   {
      "executor_id": "job.1436298853",
      "executor_name": "Command Executor (Task: job.1436298853) (Command: sh -c
'iperf ....')",
      "framework_id": "20150707-195256-1740121354-5150-29801-0000",
      "source": "job.1436298853",
      "statistics": {
         "cpus_limit": 1.1,
         "cpus_nr_periods": 16314,
         "cpus_nr_throttled": 16313,
         "cpus_system_time_secs": 2667.06,
         "cpus_throttled_time_secs": 8036.840845388,
         "cpus_user_time_secs": 123.49,
         "mem_anon_bytes": 8388608,
```

```
"mem_cache_bytes": 16384,
"mem_critical_pressure_counter": 0,
"mem_file_bytes": 16384,
"mem_limit_bytes": 167772160,
"mem_low_pressure_counter": 0,
"mem_mapped_file_bytes": 0,
"mem_medium_pressure_counter": 0,
"mem_rss_bytes": 8388608,
"mem_total_bytes": 9945088,
"net_rx_bytes": 10847,
"net_rx_dropped": 0,
"net_rx_errors": 0,
"net_rx_packets": 143,
"net_traffic_control_statistics": [
  {
    "backlog": 0,
    "bytes": 163206809152,
    "drops": 77147,
    "id": "bw_limit",
    "overlimits": 210693719,
    "packets": 107941027,
    "qlen": 10236,
    "ratebps": 0,
    "ratepps": 0,
    "requeues": 0
  },
  {
    "backlog": 15481368,
    "bytes": 163206874168,
    "drops": 27081494,
    "id": "bloat_reduction",
    "overlimits": 0,
    "packets": 107941070,
    "qlen": 10239,
    "ratebps": 0,
    "ratepps": 0,
    "requeues": 0
  }
],
"net_tx_bytes": 163200529816,
"net_tx_dropped": 0,
"net_tx_errors": 0,
```

```
            "net_tx_packets": 107936874,
            "perf": {
                "duration": 0,
                "timestamp": 1436298855.82807
            },
            "timestamp": 1436300487.41595
        }
    }
]
```

Slave 节点恢复机制

Slave节点恢复机制

恢复slave节点是Mesos的一个功能：

1. 当slave节点进程死亡的时候，让执行器和任务保持运行。

2. 让slave节点运行的执行器和任务去连接slave节点已重启的进程。

Mesos被控端有两种情况需要重启：升级或者崩溃。此功能在**0.14.0** 版本后启用。

恢复机制

slave节点恢复工作的前提是：有充足的slave节点监测点信息，包括本地磁盘中运行中的任务和执行器的位置（比如：任务信息，执行器信息，状态变更）。

当slave节点 和应用框架 开启运行锚点监控时，随后的slave进程重启机制 就可以恢复 监测点信息，重新连接 执行器。

注意：当slave节点所运行的slave进程被重启的时候，所有的执行器和任务都将被终止。

> NOTE: 若要开启应用框架恢复功能，必须获取精确的监测信息。
>
> 另外，可选择不试用监测的情形：当磁盘io读写性能要求较高，不希望监测进程带来额外磁盘读写负载时可选择不用锚点监测。

启用slave节点运行锚点监测

> NOTE: 从Mesos 0.22.0开始，slave节点 运行锚点监测将自动在所有slave节点开启。

此功能有四个参数可供slave节点设置。

• checkpoint： 设置是否把锚点监测 slave节点和应用框架的信息写入磁盘。 [默认： true].

• 开启此项将可将重启的slave节点恢复到之前的状态，使用 (--recover=reconnect) 来恢复和终止 (--recover=cleanup) 原有执行器。

> NOTE: 从Mesos 0.22.0起，此设置被移除，因为这将作用于所有slave节点。

- strict：设置是否执行严格恢复模式 [默认：true].
- 若设置strict=true，则将只恢复没有任何出错时的状态。
- 若设置strict=false，所有之前的出错，将被忽略，将尽可能恢复到最近状态。
- recover：设置是否恢复状态并重新连接原先的执行器。 [默认重连：reconnect].
- 若设置为recover=reconnect，将重连运行中的执行器。
- 若设置为recover=cleanup，则会终止所有原有的、还在活动的执行器然后退出。这可用于想要不向前兼容的slave升级或执行器升级。

> NOTE: 若无任何监测点信息存在，则无法执行恢复，slave节点在master节点上将被注册为新的slave节点。

- recovery_timeout：设置恢复slave节点时的超时总时间 [默认15分钟： 15 mins].
- 若slave节点恢复所需时间超过设定的时间，则此指向此slave节点、待重连的执行器都将自动中断。

> NOTE: 此设置只有当配置项`--checkpoint`启用时才有效。

> NOTE: 若所有应用框架都没启用锚点监测，
>
> 当slave节点死亡后，应用框架的执行器和任务都将无法恢复。

重启的slave节点将需要在限定时间内（目前设定为 75秒）重新到master节点那里注册，若超过此限定时间，master节点将关闭slave节点，slave节点里活动的执行器和任务也将被关闭。

所以，强烈建议配置一个slave节点自动重启的工具/机制。（比如使用monit(null)）。

slave节点的完整选项可./mesos-slave.sh --help 查看

启用应用框架锚点监测

若开启应用框架锚点监测，FrameworkInfo在更新的时候，将包含checkpoint字段。 应用若要开启运行监测，需要在向master节点注册之前，设置FrameworkInfo.checkpoint=True 。

> NOTE: 启用锚点监测的应用框架将只会选用启用锚点监测的slave节点。所以，在设置checkpoint=True之前，要确保集群中存在启用了锚点监测的slave节点。
>
> 若不存在这种slave节点，应用框架将申请不到任何资源，当然也更无法运行任何任务或者执行器了。

systemd 和 POSIX 隔离方式时会出现的问题

当使用 systemd 加载 mesos-slave 时，若只使用了posix隔离机制，将导致 应用的任务 无法恢复。

由于systemd进程的默认终止进程模式KillMode(null) 是采用 cgroup 以组定策略的，因此当slave节点停止运行的时候，所有子进程也将被停止。

解决办法：需要指定 KillMode为process以进程定策略，这样执行器就可以在slave节点死亡状态时不被殃及，从而可以在之后重连。

下面是具体的systemd设置方法：

```
[Service]
ExecStart=/usr/bin/mesos-slave
KillMode=process
```

> NOTE: 由于知道这个问题，docker容器技术里，使用了一个非Posix机制来隔离。

升级到0.14.0

若想让一个运行中的Mesos集群升级到0.14.0，以获得 slave恢复的功能，可参考 升级说明(null).

辅助工具

辅助工具

运维工具

下面工具可方便建立和运行Mesos集群。

- collectd-mesos(null) Mesos集群常用监控指标得收集工具。
- 部署脚本(null) 用于把Mesos部署到一群机器的集群中。
- Everpeace版的Chef手册(null) 安装和配置Mesos的master和slave。也可完全从源码构建。
- Mdsol版的Chef手册(null) 安装Mesos集群的工具，此工具使用Mesosphere的包来安装Mesos。
- Deric的Puppet 模块(null) Puppet模块，用于管理集群中的Mesos节点。
- Everpeace版Vagrant构建Mesos(null) 使用Vagrant来构建集群。
- Mesosphere版Vagrant构建Mesos(null) 使用Vagrant来快速构建Mesos沙箱环境。

开发用工具

如果你想修改Mesos或者写一个新的应用框架，可以使用以下工具：

- clang格式(null)来自动应用Mesos C++风格指南(null)定义的代码风格
- Go 绑定和示例(null) 用go语言写Mesos应用：调度器和执行器。
- Mesos 应用giter8模板(null) giter8模板，scala语言，使用SBT编译，Vagrant 部署到集群测试。
- Scala入门(null) 一个简单的Mesos示例：下载代码以及在集群的节点中部署一个网络服务器。

• Xcode 工场(null) 在Xcode中修改Mesos。

如果上面列表没有你要列的，可提交一个补丁，或者电邮user@mesos.apache.org。

SSL

在默认配置下，所有流进Mesos的消息流都为非加密形式。这样带来现在的漏洞，会让别有居心的人劫持或任意操纵已有任务。

在Mesos 0.23.0之后，引入了对SSL/TLS地支持，使得消息在Mesos的组件网络传输时可以被加密。并且在Mesos WebUI上，可以使用HTTPS的支持。

配置

目前只有libprocess socket interface(null)通过使用 libevent(null)支持SSL。另外，这个实现需要封包的openssl的libevent-openssl函数库。

当从源码编译Mesos 0.23.0时，在安装了所需依赖Dependencies(null)后，可在 编译配置里，开启SSL，如下：

~~~

../configure --enable-libevent --enable-ssl

~~~

运行

编译完并安装后，下面是可通用于Master、Slave、应用框架调派器/执行器或者任何libprocess进程的环境变量：

#### SSL_ENABLED=(false	0,true	1) [默认=false	0]

默认设置为关闭。而当开启时，socket通讯将使用SSL，也就是出口和入口的通讯都将使用SSL。

下面的参数都只有在开启SSL时才有效。

#### SSL_SUPPORT_DOWNGRADE=(false	0,true	1) [默认=false	0]

设定非SSL连接是否可以建立。默认为不可建立，若设置为可用，则若连入方想使用非SSL进行连接时，系统将自动把socket连接降级为非SSL（比如HTTP），详情可见集群升级(null)。

SSL*KEY*FILE=(path to key)

设置OpenSSL使用的私钥所在路径。

~~~

// For example, to generate a key with OpenSSL:
~~~

openssl genrsa -des3 -f4 -passout pass:some_password -out key.pem 4096

~~~

SSL*CERT*FILE=(证书路径)

设置证书所在路径。

~~~

// For example, to generate a certificate with OpenSSL:

openssl req -new -x509 -passin pass:some_password -days 365 -key key.pem -out cert.pem

~~~

#### SSL_VERIFY_CERT =(false	0,true	1) [默认=false	0]

设置是否对证书进行有效性验证。默认设置为不验证，当SSL*REQUIRE*CERT被设置为true时，SSL*VERIFY*CERT会被覆盖，所有的证书都 *必须* 被验证，否则无效。

#### SSL_REQUIRE_CER T=(false	0,true	1) [默认=false	0]

设置连接客户端时，是否强制使用证书。默认不强制使用，若设置为使用，则所有连接都需要出示有效证书。

SSL*VERIFY*DEPTH=(N) [默认=4]

验证证书的深度，默认是4层。若需要修改，可查阅 OpenSSL文档，或者咨询系统管理员。

SSL*CA*DIR=(授权证书所在文件夹路径)

可根据授权方式，指定证书授权和认证的路径SSL*CA*DIR或者文件SSL*CA*FILE。

SSL*CA*FILE=(授权证书文件所在路径)

可根据授权方式，指定证书授权和认证的路径SSL*CA*DIR或者文件SSL*CA*FILE。

SSL_CIPHERS=(accepted ciphers separated by ':') [默认=AES128-SHA:AES256-SHA:RC4-SHA:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA:DHE-RSA-AES256-SHA:DHE-DSS-AES256-SHA]

指定允许的密钥类型，使用:分号作为分隔符。若需要修改，可查阅 OpenSSL文档，或者咨询系统管理员。
~~~

SSL*ENABLE*SSL_V3=(false|0,true|1) [默认=false|0]

SSL*ENABLE*TLS*V10*=(false|0,true|1) [默认=false|0]

SSL*ENABLE*TLS*V11*=(false|0,true|1) [默认=false|0]

#### SSL_ENABLE_TLS_ V1_2=(false	0,true	1) [默认=true	1]

设置协议版本。默认只有TLS V1.2开启。SSL V2则是永久不可用，此处无设置项可用。由于很多老版本的协议易受破解，因此，请小心开启。

_注：SSLv2 完全不可用，因为现在的OpenSSL通过使用multiple compile time configuration 选项关闭了该选项。

关联依赖

libevent

我们需要通过libevent获得对OpenSSL的功能支持。推荐使用版本2.0.22-stable(null)。当新版本推出的时候，我们会测试其兼容性。

~~~

// OSX安装示例：

brew install libevent

~~~

OpenSSL

SSL使用 OpenSSL(null)实现，可根据其版本信息和你的需要，确定使用具体版本。

Mesos对具体OpenSSL的版本并没有强依赖，用户可根据从自身组织的需求出发，选择可用版本。

在编译mesos时，请确认 event2 和 openssl 有出现在头文件里。

~~~

// OSX安装示例：

brew install openssl

~~~

集群升级

在集群升级的过程中，SSL对各个部件的操作并没有特定的顺序要求。

建议重启所有组件，以启用 可降级使用的SSL，当所有部件都启用SSL时，进行第二次重启，并关闭所有的SSL降级使用功能。

当然，升级的策略也允许在任意时间点，独立的将组件的重启。也可以先在集群的子集群中先行测试下SSL。

NOTE: 当集群中的不同部件使用SSL伺服，又使用非SSL信道，则web界面下的所有相对链接都将失效。详情见Web界面(null) 章节。

下面是升级集群的命令示例：

~~~

// 先重启每个部件（包括master、slave、应用框架），以支持降级使用（包括master、slave、应用框架）：

SSL*ENABLED=true SSL*SUPPORT*DOWNGRADE=true SSL*KEY*FILE=<path-to-your-private-key> SSL*CERT*FILE=<path-to-your-certificate> <Any other SSL** environment variables you may choose> <your-component (e.g. bin/master.sh)> <your-flags>

// 再重启每个部件（包括master、slave、应用框架），不带降级使用功能。

SSL*ENABLED=true SSL*SUPPORT*DOWNGRADE=false SSL*KEY*FILE=<path-to-your-private-key> SSL*CERT*FILE=<path-to-your-certificate> <Any other SSL** environment variables you may choose> <your-component (e.g. bin/master.sh)> <your-flags>

~~~

最终状态可以使得集群将只使用SSL进行消息传递。

NOTE: 此时，所有用于组件沟通的工具，都必须能支持SSL，否则将被禁止。当你需要升级工具的时候，也可以设置为SSL*SUPPORT*DOWNGRADE=true，以方便调试。

Web界面

默认的Mesos Web界面用的链接是相对链接，这些链接在master和slave节点间跳转，但web界面目前还无法通过修改 http 或 https来确定endpoint是否是使用 SSL。所以，当混用SSL和非SSL时，或系统正在两者间切换时，Web界面的相对链接就会失效。不过当使用正确的协议访问endpoint或者启用降级连接支持的时候，所使用的工具仍然是有效作用的。

注：应用框架和他们自己的web UI界面需要另外单独添加对HTTPS的支持。

证书

大部分浏览器在构建的时候，就已经禁止 使用不同证书 的页面间进行相互跳转。因此，需要在Master节点和Slave节点都使用同样的证书。若非如此，同域链接跳转（如指向Slave沙盒的链接）将被浏览器视为非安全链接而被终止。

高级功能

属性和资源

Mesos system 系统有两个基本的方法来描述集群的子集，一个是用 Mesos 主控来管理，另一个是用集群传送框架。

类型

那些由 Mesos 中属性和资源支持的数值的类型包括 scalar、Range、Set 和 Text
以下是这类类型的定义：
`

scalar : floatValue

floatValue : (intValue ("." intValue)?)	...

intValue : [0-9]+
range : "[" rangeValue ("," rangeValue)* "]"
rangeValue : scalar "-" scalar
set : "{" text ("," text)* "}"
text : [a-zA-Z0-9_/.-]
`

属性

属性是 Mesos 发送给框架的 key-value 对（其中 value 是可选的）。一个属性的 value 支持三个不同种类的类型：scalar、range、text

attributes : attribute (";" attribute)* attribute : text ":" (scalar | range | text)

资源

Mesos 系统可以管理三种不同的资源：scalar、range、set。这些被用于代表 Mesos 从属所提供的不同的资源。例如，一个 scalar 资源类型可以代表一个从属的内存。每个从属都由一个 key string 标识。
`

resources : resource (";" resource)*

resource : key ":" (scalar | range | set)

key : text ("(" resourceRole ")")?
resourceRole : text | "*"
`

预定义的用途和约定

Mesos master 有一些可以用来处理预定义的资源。目前，这个列表包括：

- cpus
- mem
- disk

• ports

当一个从属没有 cpus 和 mem 时，它就没有可以发送给框架的资源。

实例

以下是配置 Mesos 从属的几个例子：

```
--resources='cpus:24;mem:24576;disk:409600;ports:[21000-24000];bugs:{a,b,c}'
--attributes='rack:abc;zone:west;os:centos5;level:10;keys:[1000-1500]'
```

在这个例子中，我们有不同类型的资源，它们将 cpus, mem, disk, 和 ranges 称作 ports:

```
scalar called cpus, with the value 24
scalar called mem, with the value 24576
scalar called disk, with the value 409600
range called ports, with values 21000 through 24000 (inclusive)
set called bugs, with the values a, b and c
```

在 attributes 的例子中，我们最终会有三个属性：

```
rack with text value abc
zone with text value west
os with text value centos5
level with scalar value 10
keys with range value 1000 through 1500 (inclusive)
```

Mesos Fetcher

Mesos 0.23.0 针对 Mesos fetcher cache 做出了实验性的支持。

在这方便，我们简单地将“下载”视为从本地文件系统进行拷贝。

什么是 Mesos fetcher

Mesos fetcher 是一个将资源下载到用于运行任务的沙盒目录的机制。最为 TaskInfo 消息的一部分，用于整理任务执行的框架提供了，CommandInfo::URI protobuf 值的列表，并决定哪个将作为 Mesos fetcher 的输入。

默认情况下，每个要求的 URI 直接下载到沙盒目录中。另外，fetcher 还可以指导缓存 URI 在指定的目录里下载。

Mesos fetcher 机制包括以下两个部分：

• slave-internal Fetcher 进程控制和协调着所有的行动。每个从属实例只有一个内部的 fetcher 可以被各种 containerizer 使用 (除了外部 containerizer 变量)。

• 外部的 mesos-fetcher 被前者调用。 它除了文件删除和文件大小查询缓存，所有的网络和磁盘操作都执行。为了保护 I/O 相关子进程，它作为一个外部 OS 进程运行。它需要包含一个有详细 fetch 动作说明的 JSON 对象的环境变量。

提取机制

框架启动任务通过调用调度驱动程序的方法 launchTasks（ ），传递 CommandInfo 的 protobuf 的结构作为参数。

这种类型结构所指定了一个命令和需要被“fetcher”到子节点沙盒目录的 URI 列表。因此，当子节点接到一个启动任务的指令后，它会调用 fetcher，提供指定的资源投入到沙盒目录中。如果 fetch 失败了，任务不会启动并且报告的任务状态会是 TASK_FAILED。

All URIs requested for a given task are fetched sequentially in a single invocation of mesos-fetcher. Here, avoiding download concurrency reduces the risk of bandwidth issues somewhat. However, multiple fetch operations can be active concurrently due to multiple task launch requests.

Mesos 管理容器网络

网络在数据中心基础设施中起着关键的作用，它–现在–超出了Mesos的范围去强调网络的安装问题、拓扑结构和性能。然而，Mesos可以轻松地与现有的网络解决方案集成，使诸如每个容器都有IP、任务颗粒的任务隔离和服务发现等功能生效。更经常的是，提供一个放之四海而皆准的网络解决方案是具有挑战性的。只有云计算的、本地部署的和混合的部署方案之间的要求和可选方案是不尽相同的。

Mesos对网络的支持的主要目标之一是为用户提供一个可插拔的机制，当用户需要的时候，他们可以定制网络方案。因此，从0.25.0版本开始，Mesos组件添加了几个扩展以提供网络支持。并且，所有的扩展是可选的，以保证更旧的没有网络支持的框架和应用可以与新的共存。

> 本文档的其它部分描述了所有涉及的组件的总体框架、每个容器都有IP的配置步骤、所需的框架改变。

如何运作?

> 一个关键的观察是，网络支持通过使用Mesos模块启用的，因此Mesos master和代理对它是完全不知情的。它完全由Mesos网络模块提供所需的支持。下一步，IP请求提供了哦一个最佳努力的方式提供。因此，该框架应愿意处理忽略（如模块不存在的情形）或拒绝（由于各种原因不能分配IP时）的请求。
> 为了最大程度地与现有的框架向后兼容，调度程序必须对每个容器的网络隔离是可选的。调度程序使用TaskInfo信息中的新数据结构实现网络隔离。

术语

•IP Address Management (IPAM)服务器

o根据需要分配IPs

oIPs 一被释放就回收

o(可选地使用给定的字符串/id对 IPs打标签

•IPAM客户端

o与一个特定的 IPAM服务器紧耦合

o在 “Network Isolator Module” 和IPAM 服务器之间充当桥梁

o与服务器通信以请求/释放IPs

•Network Isolator Module (NIM)：

o代理实现Isolator 接口的Mesos模块

o查看TaskInfos 以发现任务的 IP需求

o与IPAM客户端通信以请求/释放IPs

o与外部网络虚拟器/隔离器通信，以启用网络隔离功能

•Cleanup模块:

o在代理丢失事件发生时负责清理工作（例如释放IPs），其它时候休眠

框架为容器请求IP地址

1.Mesos框架采用TaskInfo消息请求为每个启动的容器请求IPs。（如果Mesos集群不支持为每个容器提供IP，请求被忽略）。

2.Mesos Master处理TaskInfos并将它们转发给Agent以启动任务。

网络隔离器模块从IPAM服务器中获得IP

1.Mesos代理检查TaskInfo以发现容器需求（这种情况下Mesos被容器化），并未即将启动的容器准备各种隔离器。

oNIM检查TaskInfo，以决定是否启用网络隔离器。

2.如果启用了网络隔离器，NIM通过IPAM请求IP地址并告知代理。

代理使用命名空间启动容器

o代理在一个新的网络命名空间内启动容器。代理调用NIM执行“隔离”。

o接着NIM调用网络虚拟器隔离容器。

网络虚拟器为容器分配IP并隔离它。

1.接着NIM使用IP信息“装饰”TaskStatus。

o可以在Master的状态端点使用TaskStatus中的IP地址。

oTaskStatus被传递到框架中通知它IP地址。

o当一个任务被杀死或者丢失的时候，NIM与IPAM客户端通信以释放对应的IP地址。

Cleanup模块检测丢失的代理并执行清理操作

1.当代理丢失事件发生的时候，Cleanup模块获得通知。

2.Cleanup模块与IPAM客户端通信以释放所有与丢失代理有关的IP地址。 IPAM在回收IP地址之前，可能会有一个宽限期。

配置

网络隔离模块不是标准的Mesos分部的一部分。然而，在https://github.com/mesosphere/net-modules有一个实现样例。
一旦网络隔离模块被嵌入到一个共享的动态库中，我们可以把它加载到Mesos Agent (/documentation/latest/ 在[modules documentation,http://mesos.apache.org/documentation/latest/modules/]中查阅关于搭

建和加载一个模块的说明)。

使框架具备每个容器一个IP的能力

NetworkInfo

一个新的NetworkInfo message被引入了：

```
message NetworkInfo {
enum Protocol {
  IPv4 = 0,
  IPv6 = 1
}
optional Protocol protocol = 1;
optional string ip_address = 2;
repeated string groups = 3;
optional Labels labels = 4;
};
```

从IPAM中请求IP地址的时候，需要将protocol域设置为IPv4 or IPv6。如果 NIM支持的话，将ip_address设置成一个有效的IP地址可以让框架为每个容器指定一个静态的IP 地址。当一个任务哪怕是在被杀死然后在另外一个不同的节点重启的时候，它也必须绑定一个特定的IP地址的时候，这是很有用的。

指定网络要求的例子

想为每个容器提供IP的框架，需要在TaskInfo中提供NetworkInfo信息。以下是一些例子：

```
1.使用默认的命令执行器请求一个非指定协议版本的地址
2.TaskInfo {
3.  ...
4.  command: ...,
5.  container: ContainerInfo {
6.    network_infos: [
7.      NetworkInfo {
8.        protocol: None;
9.        ip_address: None;
10.         groups: [];
11.         labels: None;
12.     }
13.   ]
14. }
```

```
15.}
16.使用默认的命令执行器，在两个分开的组中分别请求一个IPv4和一个IPv6地址
17.TaskInfo {
18.  ...
19.  command: ...,
20.  container: ContainerInfo {
21.    network_infos: [
22.      NetworkInfo {
23.        protocol: IPv4;
24.        ip_address: None;
25.        groups: ["public"];
26.        labels: None;
27.      },
28.      NetworkInfo {
29.        protocol: IPv6;
30.        ip_address: None;
31.        groups: ["private"];
32.        labels: None;
33.      }
34.    ]
35.  }
36.}
37.使用自定义的命令执行器请求一个的指定的IP地址
38.TaskInfo {
39.  ...
40.  executor: ExecutorInfo {
41.    ...,
42.    container: ContainerInfo {
43.      network_infos: [
44.        NetworkInfo {
45.          protocol: None;
46.          ip_address: "10.1.2.3";
47.          groups: [];
48.          labels: None;
49.        }
50.      ]
```

```
51.    }
52.  }
53.}
```

> 注意：The Mesos Containerizer拒绝任何包含了ContainerInfo的CommandInfo。 因此，在使用Mesos Containerizer选择网络隔离的时候，需要设置TaskInfo.ContainerInfo.NetworkInfo。

地址发现（Address Discovery）

> NetworkInfo信息允许框架请求在任务运行时间内为Mesos代理分配IP地址。在使用这种方法为一个指定的执行器容器选择了网络隔离之后，为了进行健康检查或进行其它带外通信，框架需要知道最终分配的地址。这是通过添加新的域到TaskStatus信息中实现的。

```
message ContainerStatus {
repeated NetworkInfo network_infos;
}
message TaskStatus {
...
optional ContainerStatus container;
...
};
```

> 进一步地，容器IP地址也是通过Master的状态端点暴露的。Master的状态端点JSON输出包含了一个任务状态列表。如果一个任务容器以它自己的IP地址开始，那么被分配的IP地址将会作为TASK_RUNNING状态的部分内容暴露。
> 注意：由于框架对每个容器具有地址是严格可选的，如果没有一开始就设置好NetworkInfo，框架可能会忽略StatusUpdate提供的IP地址。

写一个自定义的网络隔离模块

> 网络隔离器模块实现Mesos提供的Isolator接口。该模块作为一个动态共享库加载到Mesos代理中，并被挂在容器启动序列中。为了满足框架的需求，网络隔离器可能要与外部IPAM和网络虚拟器进行通信。
> 就Isolator API而言，一个网络隔离器需要实现三个关键的回调：

1.Isolator::prepare() 函数为模块提供了一个机会去决定是否为给定的任务容器启用网络隔离。如果启用网络隔离，Isolator::prepare调用会通知代理为协调者创建一个私有网络命名空间。也正是这个接口会(静态地或者在外部IPAM代理的帮助下)同时为容器生成一个IP地址。

2.在容器被创建之后、其内部的执行器被运行之前，Isolator::isolate() 为模块提供了一个机会去隔离容器。这个包括为容器创建一个虚拟的以太网适配器并分配一个IP地址给它。模块也可以在外部网络虚拟器/隔离器的帮助下为容器建立网络。

3.当容器终止的时候，Isolator::cleanup()被调用。它允许模块执行任何清理操作，比如说，如有需要，可以回收资源、释放IP地址等。

超额认购

高优先级的面向用户的服务通常在大型集群上为高峰负荷和突发尖峰负荷分配（资源）。因此，大多数时候，被分配的资源没有被充分利用。超额认购利用暂时闲置的资源执行最大努力任务，如背景分析，视频/图像处理，芯片模拟，和其它低优先级的任务。

它是如何运作的?

在 Mesos 0.23.0 中引入了超额认购，并添加了两个新的slave组件：一个资源估算程序（Resource Estimator）和一个服务质量控制程序（Quality of Service (QoS) Controller），与此同斯对现有的资源分配程序、资源监控程序和mesos slave进行了扩展。这些新组件以及它们之间的相互作用将在下文予以说明。

资源估算

•(1)第一步是确定超额资源量。资源估算程序嵌入到资源监控程序中和定期通过ResourceStatistic信息获取使用量统计数据。资源估算程序将根据收集到的资源统计逻辑确定超额资源量。这可以是一系列的基于测得的资源使用量松弛（分配了但未被使用的资源）和分配松弛控制算法。

•(2)从资源估算程序中保持轮询估算并对最新的估算进行跟踪。

•(3)当最新的估算与之前的估算不同的时候，slave将超额认购资源总量发送给master。

资源跟踪与调度算法

•(4)分配器分别从常规资源中跟踪资源的超额认购，并将这些资源注释为可撤销的。由资源估算程序决定哪种类型的资源可以被超额认购。强烈建议只超额认购可压缩的资源，例如cpu共享、带宽等。

框架

•(5)框架可以使用常规的launchTasks() API在可撤销资源中运行任务。为了维护框架不设计来处理优先权，只有在框架信息中注册到了REVOCABLE_RESOURCES capability set中的框架会收到所提供的可撤销资源。并且，可撤销资源不能被动态保存，持久卷也不应该在可撤销的磁盘资源上创建。

任务启用

•通常，当runTask请求被slave收到的时候，可撤销的任务开始运行。如果某些资源需要设置得与为可撤销的常规任务不同，这些资源仍被标识为可撤销的，隔离器可以采取合适的措施。

注意：如果一个任务或者执行程序所使用的任何资源可撤销，则这整个容器被当作一个可撤销的容器，因此可以被QoS Controller杀死或者节流。

干扰检测

•(6)当可撤销的任务运行的时候，不断监控运行在这些资源上的原始任务并基于SLA保证性能是很重要的。为了对检测到的干扰作出反应，QoS控制程序需要具备杀死或节流运行可撤销任务的能力。

允许框架使用超额预购的资源

计划使用超额预购资源的框架需要在REVOCABLE_RESOURCES capability set中进行注册：

```
FrameworkInfo framework;
framework.set_name("Revocable framework");
framework.add_capabilities()->set_type(
```

```
FrameworkInfo::Capability::REVOCABLE_RESOURCES);
```

From that point on, the framework will start to receive revocable resources in offers.

注意：没人有可以保证Mesos集群的超额认购启用。如果没有（启用），将提供不可撤销的资源。下文将说明如何配置Mesos超额认购。

使用可撤销的资源运行任务

通过现有的launchTasks API使用可撤销的资源运行任务。可撤销的资源会有一个可撤销的域设置。以下例子中，提供常规和可撤销的资源。

```
{
"id": "20150618-112946-201330860-5050-2210-0000",
"framework_id": "20141119-101031-201330860-5050-3757-0000",
"slave_id": "20150618-112946-201330860-5050-2210-S1",
"hostname": "foobar",
"resources": [
  {
    "name": "cpus",
    "type": "SCALAR",
    "scalar": {
      "value": 2.0
    },
    "role": "*"
  }, {
    "name": "mem",
    "type": "SCALAR",
    "scalar": {
      "value": 512.0
    },
    "role": "*"
  },
  {
    "name": "cpus",
    "type": "SCALAR",
    "scalar": {
      "value": 0.45
    },
    "role": "*",
    "revocable": {}
  }
]
}
```

写一个自定义的资源估算程序

资源估算程序估算和预测slave所需使用的所有资源，通知master可以超额订购的资源。Mesos默认伴随着一个空的固定的资源估算程序。这个空的估算程序只给slave和stalls提供一个空的估算，并有效地禁止超额订购。固定估计不使用实际测量的松弛，但超额认购带有固定资源量的节点（通过命令行标志定义）。

接口被定义如下：

```
class ResourceEstimator
{
public:
// 初始化资源估算程序。在其它成员方法被调用之前，这个方法需要被调用。
//它在资源估算程序上注册一个回调。这个回调允许资源估算程序
//为slave上的每一个执行程序获取当前资源使用量。
virtual Try<Nothing> initialize(
    const lambda::function<process::Future<ResourceUsage>()>& usage) = 0;

//返回slave上可以被超额认购的最大资源量的当前估算。一个新的估算会使//之前返回的估算失效。Slave会定期调用这个方法并将它传递给master。
//因此，每次这个方法被调用的时候，一个估算程序需要对估算进行回复。
virtual process::Future<Resources> oversubscribable() = 0;
};
```

写一个自定义的QoS控制程序

实现自定义的QoS Controllers的接口被定义如下：

```
class QoSController
{
public:
// 初始化QoS Controller。在其它成员方法被调用之前，这个方法需要被调//用。它在QoS Controller上注册一个回调。这个回调允许
//QoS Controller为slave上的每一个执行程序获取当前资源使用量。
virtual Try<Nothing> initialize(
    const lambda::function<process::Future<ResourceUsage>()>& usage) = 0;

// QoS Controller通知slave需要采取的纠偏措施，但返回
//futures 给QoSCorrection对象。更多的信息请查阅mesos.proto。
virtual process::Future<std::list<QoSCorrection>> corrections() = 0;
};
```

注意： QoS Controller不能阻塞corrections()。使用它自己的libprocess 执行程序代替支持QoS Controller。

QoS Controller通知slave需要采取特定的纠偏措施。每一个纠正措施都包含有关执行程序或任务的信息和执行的动作类型。

```
message QoSCorrection {
enum Type {
  KILL = 1; // Terminate an executor.
}
message Kill {
  optional FrameworkID framework_id = 1;
  optional ExecutorID executor_id = 2;
}
required Type type = 1;
optional Kill kill = 2;
}
```

配置超额认购

五个新的标识被添加到了slave中：

标志说明
--oversubscribed*resources*interval=VALUE	Slave定期使用分配和可用的超额订购资源的最大量的当前估算更新master。通过这个标志控制这些纠偏的最小区间 (默认值: 15secs) 。

--qos_controller=VALUE QoS Controller的名字用于超额订购。

--qos*correction*interval_min=VALUE 基于QoS Controller所观察到的运行任务的性能，slave轮询和采取QoS纠偏。通过这个标志控制这些纠偏的最小区间 (默认值: 0ns) 。

--resource_estimator=VALUE 超额认购所使用的资源评估程序的名称。

固定的资源估计程序通过以下方式启用：

--resource*estimator="org*apache*mesos*FixedResourceEstimator"

```
--modules='{
"libraries": {
  "file": "/usr/local/lib64/libfixed_resource_estimator.so",
  "modules": {
    "name": "org_apache_mesos_FixedResourceEstimator",
    "parameters": {
      "key": "resources",
      "value": "cpus:14"
    }
  }
```

}

}'

在上述例子中，共有14个CPU可以作为可被撤销的资源提供。

为了安装一个自定义的资源估算程序和一个QoS控制程序，请查阅 [modules documentation, http://mesos.apache.org/documentation/latest/modules/]。

Persistent Volume

Mesos 提供了一个从磁盘资源中创建永久卷标的机制。当启动一个任务，你可以创建一个存在于任务沙箱外的卷标，此卷标在任务结束或者完成后依然存在。当任务退出时，它的资源 —— 包括永久卷标 —— 可以返回给框架，这样框架就可以重新启动同样任务，启动回复任务，或者启动一个新的将之前任务的输出作为输入的任务。永久卷标可以让如 HDFS 和 Cassandra 在 Mesos 内存储数据。

持久卷标只能从预定的磁盘资源创建，无论是静态预定或动态预定。一个动态预定永久卷标在没有明确破坏卷标的情况下取消预定。这些规则限制了一些意外错误，例如：在集群中，一些包括了敏感数据的永久卷标被提供给其他框架。

请查看 Reservation(null) 文档来查阅更多信息。

永久卷标可以由运营商和授权框架创建。我们需要一个从运营商或框架中获取 principal 来认证/授权操作。

授权由现有 ACL 机制指定。（即将推出）

• Offer::Operation::Create 和 Offer::Operation::Destroy 消息可以通过 acceptOffers API 发回给框架

• 运营商可以用/create 和 /destroy HTTP 端点来管理永久卷标。（即将推出）

在下面的章节中，我们会通过例子来解释：

Offer::Operation::Create

框架可以通过资源提供周期来创建卷标。假设我们需要预定一个 2048MB 的动态磁盘空间。

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "disk",
      "type" : "SCALAR",
      "scalar" : { "value" : 2048 },
      "role" : <framework_role>,
      "reservation" : {
        "principal" : <framework_principal>
      }
    }
```

```
  ]
}
```

我们可以通过 acceptOffers API 发送 Offer::Operation 消息在 2048MB 的磁盘中创建永久卷标。Offer::Operation::Create 有一个卷标域可以用于指定永久卷标信息，我们需要指定的信息如下：

- 每个 slave 的每个 role 所需要的永久卷标 ID
- 容器内到卷标的非嵌套相对路径
- 卷标的权限，目前来讲，只可能是“RW”

```
{
  "type" : Offer::Operation::CREATE,
  "create": {
    "volumes" : [
      {
        "name" : "disk",
        "type" : "SCALAR",
        "scalar" : { "value" : 2048 },
        "role" : <framework_role>,
        "reservation" : {
          "principal" : <framework_principal>
        },
        "disk": {
          "persistence": {
            "id" : <persistent_volume_id>
          },
          "volume" : {
            "container_path" : <container_path>,
            "mode" : <mode>
          }
        }
      }
    ]
  }
}
```

后续资源将包括以下永久卷标

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
```

```
  "resources" : [
    {
      "name" : "disk",
      "type" : "SCALAR",
      "scalar" : { "value" : 2048 },
      "role" : <framework_role>,
      "reservation" : {
        "principal" : <framework_principal>
      },
      "disk": {
        "persistence": {
          "id" : <persistent_volume_id>
        },
        "volume" : {
          "container_path" : <container_path>,
          "mode" : <mode>
        }
      }
    }
  ]
}
```

Offer::Operation::Destroy

框架可以通过资源供给周期取消永久卷标。在 Offer::Operation::Create(null) 内，我们在 2048 MB 磁盘资源中创建了永久卷标。

直到我们确定取消掉它后，Mesos 才会把它当垃圾收走。假设我们想取消掉创建的卷标：

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "disk",
      "type" : "SCALAR",
      "scalar" : { "value" : 2048 },
      "role" : <framework_role>,
      "reservation" : {
        "principal" : <framework_principal>
      },
      "disk": {
```

```
      "persistence": {
        "id" : <persistent_volume_id>
      },
      "volume" : {
        "container_path" : <container_path>,
        "mode" : <mode>
      }
    }
  }
 ]
}
```

我们通过 acceptOffers API 发送 Offer::Operation message 消息来摧毁永久卷标。
Offer::Operation::Destroy 有一个卷标域，我们可以用来指定需要摧毁的卷标。

```
{
  "type" : Offer::Operation::DESTROY,
  "destroy" : {
    "volumes" : [
      {
        "name" : "disk",
        "type" : "SCALAR",
        "scalar" : { "value" : 2048 },
        "role" : <framework_role>,
        "reservation" : {
          "principal" : <framework_principal>
        },
        "disk": {
          "persistence": {
            "id" : <persistent_volume_id>
          },
          "volume" : {
            "container_path" : <container_path>,
            "mode" : <mode>
          }
        }
      }
    ]
  }
}
```

永久卷标被销毁，但磁盘资源仍将保留。因此，后续的资源供给将包含下列磁盘资源：

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "disk",
      "type" : "SCALAR",
      "scalar" : { "value" : 2048 },
      "role" : <framework_role>,
      "reservation" : {
        "principal" : <framework_principal>
      }
    }
  ]
}
```

要注意的是，在 0.23 中，在你摧毁永久卷标之后，它的内容还是会存在于磁盘上。永久卷标垃圾箱即将推出：MESOS-2048(null)。

/create （即将推出）

/destroy （即将推出）

预留

Mesos 提供资源预定机制进行。这一概念和静态预定一起首次在 0.14.0 版本里引用。这使 Mesos 从属启动时可以让运营商指定所预定的资源。在 0.23.0 版本里，它和动态预定一起扩展了，这可以让运营商和授权框架可以动态地预定集群中的资源。

如果没有重大更改引入动态预定，这意味着现有的静态预留机制会继续支持。

在两种预定类型中，资源只会为一个 **role** 保留。

静态预订（自 0.14.0）

运营商可以用资源预定的方式为一个 role 配置 slave。预定的资源经由 --resources 标志指定。例如，我们假设在一个 slave 中有 12 个 CPU 和 6144MB 的 RAM 可用，我们想要预定 8 个 CPU 和 4096MB 的 RAM 资源给 ads role。我们可以这么做：

```
$ mesos-slave \
  --master=<ip>:<port> \
  --resources="cpus:4;mem:2048;cpus(ads):8;mem(ads):4096"
```

现在，在这个 slave 上就有了 8 个 CPU 和 4096MB 的 RAM。

CAVEAT:为了修改静态预定，运营商必须排空并重启新配置的病指定 --resources 标志的 slave。

NOTE:此功能支持向后兼容性。我们推荐通过 --resources 标志从 slave 中指定资源，并且用主 HTTP 端点动态管理预定。

动态预定（自 0.23.0）

这次，静态预定资源不能为另一个 role 预定，也不能被预定。动态预定可以使运营商和授权框架在 slave 启动后预定或者取消预定。

我们需要从运营商和授权框架获取指导准则来认证或者授权操作。授权经由 ACL 机制指定。（即将推出）

• Offer::Operation::Reserve 和 Offer::Operation::Unreserve 消息可以作为资源响应发回给框架。

• /reserve 和 /unreserve HTTP 端点可以为运营商管理动态预定（即将推出）。

Offer::Operation::Reserve

框架能够通过资源报价周期预定资源。假设我们收到一个资源报价有 12 个 CPU 和 6144 MB RAM 可以预定。

```
{
  "id": <offer_id>,
  "framework_id": <framework_id>,
  "slave_id": <slave_id>,
  "hostname": <hostname>,
  "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value": 12 },
      "role": "*",
    },
    {
      "name": "mem",
      "type": "SCALAR",
      "scalar": { "value": 6144 },
      "role": "*",
    }
  ]
}
```

我们可以通过发送 Offer::Operation 消息来预定 8 个 CPU 和 4096 MB RAM。

We can reserve 8 CPUs and 4096 MB of RAM by sending the following Offer::Operation message.

```
{
  "type": Offer::Operation::RESERVE,
  "reserve": {
    "resources": [
      {
        "name": "cpus",
        "type": "SCALAR",
        "scalar": { "value": 8 },
        "role": <framework_role>,
        "reservation": {
          "principal": <framework_principal>
        }
      },
      {
        "name": "mem",
        "type": "SCALAR",
        "scalar": { "value": 4096 },
        "role": <framework_role>,
        "reservation": {
          "principal": <framework_principal>
        }
      }
    ]
  }
}
```

然后，资源报价将包含下列预定资源：

```
{
  "id": <offer_id>,
  "framework_id": <framework_id>,
  "slave_id": <slave_id>,
  "hostname": <hostname>,
  "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value": 8 },
      "role": <framework_role>,
      "reservation": {
        "principal": <framework_principal>
      }
```

```
    },
    {
      "name": "mem",
      "type": "SCALAR",
      "scalar": { "value": 4096 },
      "role": <framework_role>,
      "reservation": {
        "principal": <framework_principal>
      }
    },
  ]
}
```

Offer::Operation::Unreserve

框架能够通过资源报价周期取消预定的资源。

在 Offer::Operation::Reserve(null)中，我们为 role 预定了 8 个 CPU 和 4096 MB 的 RAM。首先，我们收到一个资源报价（复制/粘贴上述内容）：

```
{
  "id": <offer_id>,
  "framework_id": <framework_id>,
  "slave_id": <slave_id>,
  "hostname": <hostname>,
  "resources": [
    {
      "name": "cpus",
      "type": "SCALAR",
      "scalar": { "value": 8 },
      "role": <framework_role>,
      "reservation": {
        "principal": <framework_principal>
      }
    },
    {
      "name": "mem",
      "type": "SCALAR",
      "scalar": { "value": 4096 },
      "role": <framework_role>,
      "reservation": {
        "principal": <framework_principal>
      }
    },
```

```
  ]
}
```

我们通过发送 Offer::Operation 消息来取消 8 个 CPU 和 4096 MB RAM 的预定。
Offer::Operation::Unreserve 有一个资源域可以让我们来取消资源的预定。

```
{
  "type": Offer::Operation::UNRESERVE,
  "unreserve": {
    "resources": [
      {
        "name": "cpus",
        "type": "SCALAR",
        "scalar": { "value": 8 },
        "role": <framework_role>,
        "reservation": {
          "principal": <framework_principal>
        }
      },
      {
        "name": "mem",
        "type": "SCALAR",
        "scalar": { "value": 4096 },
        "role": <framework_role>,
        "reservation": {
          "principal": <framework_principal>
        }
      }
    ]
  }
}
```

未预定的资源现在可以提供给其他框架。

/reserve (自 0.25.0)

假设我们希望保留 8 个 CPU 和 4096 MB RAM 用于 slave 的 ads role，我们要像这样发送 HTTP POST 给 /reserve HTTP 端点:

```
$ curl -i \
  -u <operator_principal>:<password> \
  -d slaveId=<slave_id> \
  -d resources='[ \
    { \
```

```
      "name": "cpus", \
      "type": "SCALAR", \
      "scalar": { "value": 8 }, \
      "role": "ads", \
      "reservation": { \
        "principal": <operator_principal> \
      } \
    }, \
    { \
      "name": "mem", \
      "type": "SCALAR", \
      "scalar": { "value": 4096 }, \
      "role": "ads", \
      "reservation": { \
        "principal": <operator_principal> \
      } \
    } \
  ]' \
  -X POST http://<ip>:<port>/master/reserve
```

用户收到下面的 HTTP 响应之一：

- 200 OK：成功
- 400 BadRequest：无效的参数（如缺少参数）。
- 401 Unauthorized：未经授权的请求。
- 409 Conflict：资源不足，无法满足预约操作。

/unreserve (自 0.25.0)

假设我们要取消预定上述动态预定的资源，我们可以像这样发送一个 HTTP POST 请求到 /unreserve：

```
$ curl -i \
  -u <operator_principal>:<password> \
  -d slaveId=<slave_id> \
  -d resources='[ \
    { \
      "name": "cpus", \
      "type": "SCALAR", \
      "scalar": { "value": 8 }, \
      "role": "ads", \
      "reservation": { \
        "principal": <operator_principal> \
```

```
      } \
    }, \
    { \
      "name": "mem", \
      "type": "SCALAR", \
      "scalar": { "value": 4096 }, \
      "role": "ads", \
      "reservation": { \
        "principal": <operator_principal> \
      } \
    } \
  ]' \
  -X POST http://<ip>:<port>/master/unreserve
```

如果用户接收到以下信息：

• 200 OK：成功

• 400 BadRequest：无效的参数（如缺少参数）。

• 401 Unauthorized：未经授权的请求。

• 409 Conflict：资源不足，无法满足取消预定操作。

运行 Mesos 框架

运行 Mesos

Mesos框架列出了一系列在Mesos基础上构建的app，并说明如何运行它们。

如果你是Mesos新手

查看开始页面(null)获取更多关于Mesos下载、搭建和部署的信息。

如果你想参与或寻求赞助，查看我们的社交网页(null)获取详细信息。

在Mesos基础上搭建的软件项目

长期提供的服务

•Aurora (null)是在Mesos上运行的服务调度，使得你可以利用Mesos在扩展性、容错能力和资源隔离等方面的优势运行长期运行的服务。

•Marathon (null)是在Mesos上搭建的私有Paas。它自动处理软硬件故障从而使得一个app总是可用。

•Singularity (null)是一个运行Mesos任务的调度 (HTTP API及we b接口)：长期运行的进程、一次性任务和计划作业。

•SSSP (null)是一个简单的web应用，它为在S3中存储和共享文件提供一个“Megaupload”的白

标签。

大数据处理

•Cray Chapel (null)是一个高产出的并行编程语言。Chapel Mesos调度程序使得Chapel程序于Mesos上。

•Dpark (null)是Spark的Python克隆，是运行在Mesos上的用Python写的类MapReduce框架。

•Exelixi(null)在一个可扩展的运行通用算法的分布式框架。

•Hadoop(null)有效地在整个集群中的Mesos分布式MapReduce作业中运行Hadoop。

•Hama(null)是一个基于Bulk Synchronous Parallel技术的海量科学计算（例如：矩阵、图和网络算法）分布式计算框架。

•MPI(null)是一个可以在各种并行计算机上运行的信息传递系统。

•Spark(null)是一个快速、通用的集群计算系统，使得并行计算作业的编写简单。

•Storm(null)是一个分布式的实时计算系统。Storm可以为实时处理简单可靠地处理没绑定的数据流，就像Hadoop为批处理和批调度做的工作一样。

批调度

•Chronos(null)是一个支持负载作业拓扑的分布式作业调度程序。可以作为Cron的一个容错能力更强的替代品。

•Jenkins(null)是一个持续整合服务器。mesos-jenkins plugin允许它根据负载情况在Mesos集群中动态地启动服务器(worker)。

•JobServer(null)是一个分布式作业调度程序和处理器，开发者可以通过点击Web UI创建定制的批处理任务队列。

数据存储

•[Cassandra] (https://github.com/mesosphere/cassandra-mesos)是一个高性能和高可用的分布式数据库。它在商用硬件或者云基础设施上的线性可扩展性和良好的容错能力，使得它成为存储关键任务数据的完美平台。

•ElasticSearch(null)是一个分部式的搜索引擎。Mesos使其易于运行和扩展。

•Hypertable(null)是一个高性能、可扩展、分布式的存储和处理结构化、半结构化数据的系统。

开发 Mesos 框架

计算框架开发指南

计算框架开发指南

在本文档中，我们把Mesos程序集称为计算框架。

你可以在源代码目录MESOS_HOME/src/examples/看到计算框架调度器的例子。通过例子来理解Mesos计算框架调度器并选择你喜欢语言进行执行。RENDLER提供了以
C++，Go，Haskell，Java，Python和Scala语言所实现的框架例子供你选择。

创建框架调度器（Framework Scheduler）

你可以用C、C++、Java/Scala或者Python语言编写一个框架调度器，它需要继承Scheduler类（见下面Scheduler API）。调度器应当创建一个SchedulerDriver，然后调用SchedulerDriver.run()函数。

调度器API(Scheduler API)

声明如下代码位于：MESOS_HOME/include/mesos/scheduler.hpp

```
/*
 * 空的虚拟的析构函数 (需要把析构函数实例化成子类).
 */
virtual ~Scheduler() {}

/*
 * 函数在当调度器成功在Mesos管理器中注册时被调用。
 * FrameworkID是在框架中由管理器生成一个唯一的ID，用于区别其他调度器。
 * MasterInfo以参数的形式提供当前的管理器IP地址和端口。
 */
virtual void registered(SchedulerDriver* driver,
                        const FrameworkID& frameworkId,
                        const MasterInfo& masterInfo) = 0;



/*
 * 函数在调度器再次在新当选的管理器注册时被调用。
 * MasterInfo以参数的形式囊括新当选的管理器信息。
 */
virtual void reregistered(SchedulerDriver* driver,
                          const MasterInfo& masterInfo) = 0;

/*
 * 函数在调度器与管理器变成"无链接"时被调用。
 * (举例来说, 当前管理器关闭并由其他的管理接管)。
 */
virtual void disconnected(SchedulerDriver* driver) = 0;

/*
 * 函数在资源已经被提供给计算框架时被调用。最简单的offer仅包含一个简单slave的资源。
 * 这些资源以一个offer的形式打包提供给当前计算框架对象，除非发生异常情况，则不在提交
。
 * 第一种当前或者某个计算框架拒绝了这些资源，才能够再次提交offer。
 * (请查看 SchedulerDriver::launchTasks) 或者第二种情况取消了这些资源
 * (请查看 Scheduler::offerRescinded)。
```

```
 * 注意：资源可能在同一时间提交给一个或者多个计算框架（根据分配器的分配情况）。
 * 如果上面的事情发生, 首先拿到offer的计算框架将会使用这些资源来启动任务，导致后获取
 * offer的计算框架取消这些资源的使用(或者某个计算框架已经使用这些资源启动了任务，
 * 这些任务将会伴随着TASK_LOST状态而失败，并发送过多的消息通知)。
 */
virtual void resourceOffers(SchedulerDriver* driver,
                            const std::vector<Offer>& offers) = 0;

/*
 * 函数在某个offer不在有效时被调用。(举例来说, 节点不可用或者资源被其他计算框架的
offer占用)。
 * 如下发生以下情况offer均不会撤销 (举例来说, 丢弃信息，计算框架运行失败，等等。)，
 * 当计算框架尝试启动那些没有有效offer的任务时，计算框架会收到那些任务发送
TASK_LOST的状态更新
 * (请查看Scheduler::resourceOffers).
 */
virtual void offerRescinded(SchedulerDriver* driver,
                            const OfferID& offerId) = 0;

/*
 * 函数在一个任务的状态发生变化时被调用。(举例来说, 一个节点（slave）丢失并且任务丢失,
 * 一个任务完成并且执行器发送了一个状态更新回话，等等）。 如果使用隐式定义implicit
 * acknowledgements, 以 _acknowledges_ 的收据作为这个状态的更新作为回调函数返回!
 * 如果发生调度器无论何种原因在回调函数的时候终止（或者进程退出）另一个状态更新将会
被提交
 * （注意，无论如何，如果slave发送状态更新是丢失或者失败。在那段时间是不正确的）。
 * 如果使用的是显示explicit acknowledgments，调度器必须在驱动中知道这个状态。
 */
virtual void statusUpdate(SchedulerDriver* driver,
                          const TaskStatus& status) = 0;

/*
 * 函数在当执行器发送消息时被调用。
 * 这些消息是尽力服务：在任何可靠的方式下，绝不期望计算框架消息会被重新提交。
 */
virtual void frameworkMessage(SchedulerDriver* driver,
                              const ExecutorID& executorId,
                              const SlaveID& slaveId,
                              const std::string& data) = 0;

/*
 * 函数在当某个slave确定不能找到时被调用。(举例来说,设备故障，网络隔离)。
```

```
 * 绝大部分计算框架会以在新的slave上重新启动所有任务的方式进行调度。
 */
virtual void slaveLost(SchedulerDriver* driver,
                       const SlaveID& slaveId) = 0;

/*
 * 函数在执行器退出或者中断时被调用。注意：任何任务的运行将会自动生成TASK_LOST的状
态更新。
 */
virtual void executorLost(SchedulerDriver* driver,
                          const ExecutorID& executorId,
                          const SlaveID& slaveId,
                          int status) = 0;

/*
 * 函数在一个未被调度器或者调度器驱动不能捕获的错误发生时被调用。
 * 调度器驱动将会在这个回调函数执行之前执行。
 */
virtual void error(SchedulerDriver* driver, const std::string& message) = 0;
```

创建计算框架执行器Framework Executor

你的计算框架执行器(Framework Executor)必须从Executor类继承。它必须覆盖launchTask()函数。你需要在Mesos决定在那些节点(slave)运行你的执行器,并在执行器内部使用 $MESOS_HOME 环境变量。

执行器 API(Executor API)

声明如下代码位于：MESOS_HOME/include/mesos/executor.hpp

```
/*
 * 函数在执行器驱动第一次成功链接到Mesos时被调用。特别的是，
 * 调度器可以把一些数据内容传递给执行器的FrameworkInfo.ExecutorInfo数据区。
 */
virtual void registered(ExecutorDriver* driver,
                        const ExecutorInfo& executorInfo,
                        const FrameworkInfo& frameworkInfo,
                        const SlaveInfo& slaveInfo) = 0;



/*
 * 函数在某节点重启后再次注册执行器时被调用。
```

```
 */
virtual void reregistered(ExecutorDriver* driver,
                          const SlaveInfo& slaveInfo) = 0;


/*
 * 函数在节点要发送的执行器"无法链接"时调用。(举例来说, 节点由于升级导致的重启)。
 */
virtual void disconnected(ExecutorDriver* driver) = 0;

/*
 * 函数在执行器要启动任务时调用。(通过Scheduler::launchTasks进行初始化)。
 * 注意：任务必须属于线程、进程、或者简单的计算，否则直到执行器返回回调时，
 * 没有任何回调函数会被调用。
 */
virtual void launchTask(ExecutorDriver* driver,
                        const TaskInfo& task) = 0;

/*
 * 函数在调度器内正在运行的任务要终止时调用。(通过 SchedulerDriver::killTask)。
 * 注意：函数将代表执行器发送没有状态更新。执行器需要对创建新的任务状态负责
 * (换种说明, TASK_KILLED)并执行ExecutorDriver::sendStatusUpdate。
 */
virtual void killTask(ExecutorDriver* driver, const TaskID& taskId) = 0;

/*
 * 函数在计算框架要传递给执行器信息时调用。这些消息是唯一正确的途径。
 * 不要指望计算框架的信息以任何其他的可靠的方式重新提交。
 */
virtual void frameworkMessage(ExecutorDriver* driver,
                              const std::string& data) = 0;

/*
 * 函数在执行器需要终止所有现在运行任务时被调用。
 * 注意：函数在Mesos确定执行器将要终止所有的任务时，执行器不会发送终止状态的更新
 * (举例来说, TASK_KILLED, TASK_FINISHED,TASK_FAILED, 等)而会创建TASK_LOST状态更
新。
 */
virtual void shutdown(ExecutorDriver* driver) = 0;

/*
 * 函数在执行器或者执行器驱动发生了一个致命性的错误是时被调用。驱动会在函数的回调之
```

```
前终止。
 */
virtual void error(ExecutorDriver* driver, const std::string& message) = 0;
```

安装计算框架Install your Framework

你需要把计算框架放在集群所能够得到的所有节点（slaves）。如果你运行需要HDFS，你可以把你的执行器放到HDFS。你可以通过ExecutorInfo参数把执行器放到HDFS这件事情告诉MesosSchedulerDriver的构造器。(举例来说：请示例代码查看src/examples/java/TestFramework.java)。ExecutorInfo 是协议缓存信息类(在include/mesos/mesos.proto中进行定义), 并且你可以设置URI字段，例如“HDFS://path/to/executor/”. 或者, 你可以通过 frameworks*home 的配置项 (在这里进行定义: MESOS*HOME/frameworks) 告诉mesos节点守护器你所制定的执行器所存储的位置 (举例来说 所有节点（slave）均使用的NFS挂载方式), 然后设置ExecutorInfo为相对路径, 节点（slave）将预先提供frameworks_home的相对路径的值。

你一旦确定执行器在mesos的那些节点可以运行，你需要运行在Mesos管理器中注册的调度器，然后开始接收资源offer！

标签Labels

可以在FrameworkInfo, TaskInfo, DiscoveryInfo和TaskStatus信息中找到标签（Labels）。计算框架和模块的编写者可以在Mesos内使用标签来标记和传递非机构化的信息。标签是以key-value的自由格式提供给计算框架调度器或标签装饰钩子。下面是protobuf的标签定义：

```
optional Labels labels = 11;
```

```
/**
 * 标签集合。
 */
message Labels {
    repeated Label labels = 1;
}

/**
 * 以Key，value的形式来自由存储用户数据。
 */
message Label {
  required string key = 1;
  optional string value = 2;
}
```

标签并不是Mesos给它自己进行解释，但作为管理器和节点之间的终点状态。此外，执行器和调度器在TaskInfo和TaskStatus以编程的方式进行标签解析。下面是列举了两个对标签的例子("environment": "prod" 和 "bananas": "apples")可以从管理器状态终点进行获取。

```
$ curl http://master/state.json
...
{
  "executor_id": "default",
  "framework_id": "20150312-120017-16777343-5050-39028-0000",
  "id": "3",
  "labels": [
    {
      "key": "environment",
      "value": "prod"
    },
    {
      "key": "bananas",
      "value": "apples"
    }
  ],
  "name": "Task 3",
  "slave_id": "20150312-115625-16777343-5050-38751-S0",
  "state": "TASK_FINISHED",
  ...
},
```

发现服务Service discovery

当你的计算框架注册了执行器，并启动了任务，可以给发现服务提供额外的信息。这些信息存储在Mesos管理器伴随着其他重要信息 例如节点（slave）正在运行的任务。在Mesos集群运行多计算框架多任务时，发现服务系统可以以编程的方式准备号纠正并恢复创建DNS索引、配置代理、或者更新不一致的存储信息。

DiscoveryInfo选项信息将会传递给TaskInfo和ExecutorInfo，声明如下代码位于：MESOS_HOME/include/mesos/mesos.proto

```
message DiscoveryInfo {
  enum Visibility {
    FRAMEWORK = 0;
    CLUSTER = 1;
    EXTERNAL = 2;
  }

  required Visibility visibility = 1;
  optional string name = 2;
  optional string environment = 3;
  optional string location = 4;
```

```
  optional string version = 5;
  optional Ports ports = 6;
  optional Labels labels = 7;
}
```

显而易见的，通知发现服务器系统应当发现已经通知过的Key参数。我们现在列举三个不同的案例：

任务不除了所属的计算框架以外的不应当被发现。

任务应当在所运行的Mesos集群中的所有计算框架发现，而不是在外部被发现。

任务应当无误的发现。

大量的发现服务系统提供附加的管理可见的字段。（举例来说，系统中基于代理的ACL，DNS的安全扩展，VLAN或者子网选择）不打算使用可见的资源来管理这些功能。服务发现系统重新从管理器获取任务或者执行器信息，这种方式解决没有DiscoveryInfo如何获取任务。举例，任务不会对其他计算框架可见。（等价于 visibility=FRAMEWORK）或者对所有计算框架可见（等价于 visibiility=CLUSTER）。

名称字段的数据格式是字符串，字符串可以保证服务发现系统通过名字的形式发现任务。典型的名称字段是提供有效的域名。如果名称无法提供，则需要服务发现系统在taskInfo或者其他信息中的名称字段来为任务创建名称。

环境变量、地址信息、和版本等字段在大型部署中提供了第一类的支持，对象的属性相同但是在不同地方应用的相似服务。环境可以接收如吃产品/质量检查/开发，位置字段可以接收如下值：EAST-US/WEST-US/EUROPE/AMEA, 以及版本字段可以接收如下值v2.0/v0.9。发现系统需要正确的把这些字段提供给服务发现系统。

端口字段允许计算框架定义任务的监听端口并明确名称的功能，端口字段是可以再次提交并使用网络的第四层通讯协议（TCP，UDP或者其他）。举例，Cassandra的任务定义如下端口 "7000,Cluster,TCP", "7001,SSL,TCP", "9160,Thrift,TCP", "9042,Native,TCP", 和 "7199,JMX,TCP"。由服务发现系统以适当形式的协议把潜在的把DiscoveryInfo的名称字段进行绑定。

标签字段允许计算框架以key/value对的格式传递主观标签给服务发现系统。注意：通过这些字段传递的任何事情，是无法保障执行移动到下一步。虽然如此，字段提供扩展性。这些共同使用的字段允许我们确认需要请求第一类支持的例子。

@2015 译者：抢小孩糖吃

一致性

一致性

无可否认，**在Mesos的计算框架是分布式系统。**

分布式系统必须对失败和分隔（两个系统内部无法进行区分）进行处理。

实际上，计算框架是什么意思呢？Mesos使用执行起来像**信息传递**的编程模式，无论那个信息只被传递**最多一次。**

(包含任务状态更新是例外情况，大部分信息传递最少用来至少一次确认)。**消息在管理器和计算框架之间通信，当存在故障时，容易发生丢弃。**

当这些不可靠的消息丢弃时，会在计算框架和Mesos发生不一致的状态。

列举一个简单的例子，需要给计算框架发送决定启动任务。有许多失败方式可以导致这些任务的丢失。举例：

• 计算框架在意图持续启动任务后发生失败，但启动的任务已经发送了信息。

• 管理器在接收到信息之前发生失败。

• 管理器在接收到信息之后发生失败，但在这之前已经发送给节点信息。

在上面这些情况下，计算框架认为任务在执行中，但是任务并不知道如何给Mesos。为了应对这种局面，**任务状态必须随时发现错误协调计算框架和Mesos。**

发现错误

发现错误是Mesos的责任（调度器驱动/管理器）来确保可以通知无链接的计算框架，随后（再次）登记错误所发生的事情。在这点上，调度器需要前去对任务状态进行调解。

任务一致性

Mesos提供了两种保持一致的方式：

• “显式” 一致性: 如果可能，调度器发送那些未结束的任务并且管理器对每个任务进行响应。

• “隐式” 一致性: 调度器发送一个空的任务列表，管理器响应所有现在已知非终结的任务的最新状态。

在计算框架发生错误后，以显式方式使任务保持一致。

上面这种方式是由于调度器驱动不能持续掌握任何任务信息。在未来的版本，调度器驱动(或者mesos开发库)可以在代表计算框架下无缝覆盖运行任务一致性。

因此，现在在计算框架调度器中，让我们来看看如何实现一件必须要一致的任务状态。

API

计算框架会给管理器发送TaskStatus信息的列表：

```
// 允许计算框架去查询非终结任务的状态。如果可能的话，在'statuses'管理器
// 返回为每个任务最新状态返回的原因。当TASK_LOST更新，任务不再知道结果。
// 如果statuses为空，管理器将会给当前每个已知的任务发送最新的状态。
message Reconcile {
  repeated TaskStatus statuses = 1; // 应当仅仅非终结。
}
```

当前，管理器仅仅对TaskStatus检查两项数据列：

• TaskID: 这个是必备的。

• SlaveID: 可选项，在存在状态间转换的节点时，会导致更快的一致。

算法

这个显式一致性技术保持所有未结束任务的一致，直到每个任务接收到更新，使用指数退避算法再

次尝试剩下未取得一致的任务。重试是必要的，管理器可能不能对特殊任务进行应答。例如，在管理器失效时候，管理器为了重建它已知的任务必须重新注册所有的节点(重建过程会花费大型集群几分钟的时间，并在管理器通过--slave*reregister*timeout标记进行上限限制)。

步骤:

1. 让 start = now()
2. 让 remaining = { T in tasks | T is non-terminal }
3. 执行一致性: reconcile(remaining)
4. 等待状态更新信息到达(使用短指数退避算法)。对于每个更新，注意到达的时间
5. 让 remaining = { T ϵ remaining | T.last*update*arrival() < start }
6. 如果 remaining 不为空,则跳转到步骤3。

一致性算法**必须**在每次(再次)注册之后。

隐性一致性(通过一个为空的列表)应当周期性执行，在计算框架中作为防御数据丢失事件的发生。除非在管理以严格注册方式使用，这种方法可能使那些进入LOST状态的任务复活(没有严格注册，管理器无无法给任务施加故障移除)。当遇到一个未知任务时，调度器应当把任务关闭或者恢复。

注意:

• 当等待更新到来时，**使用短指数退避算法**。这个算法在备份驱动或者管理器时避免滚雪球效应。

• 隐性一致性有益于确定仅在同一时间只进行一次一致操作，避免面对大量重新注册的情况发生滚雪球效应。如果要启动另一个一致操作，那之前的一致性计算程序应当停止运行。

要约一致性

在失败发生后，要约会自动保持一致:

• 要约不会超过管理器的声明周期。

• 如果发生链接中断，要约不在有效。

• 要约在每次计算框架(再次)注册发生撤销和新建。

Scheduler HTTP API

译者注：Content-Type不翻译，直接关联到HTTP的使用

Scheduler HTTP API

在Mesos 0.24.0版本添加了v1 Scheduler HTTP API的实验性支持。

概述

调度器通过“/api/v1/scheduler”主机地址经过Mesos管理器与Mesos进行通信。描述完整的唯一地址URL看起来像如下形式：

```
http://masterhost:5050/api/v1/scheduler
```

注意：将会在这篇文档的剩余部分在唯一地址提供“/scheduler”的后缀。这个地址接收以HTTP POST请求的JSON格式编码数据(Content-Type: application/json)或者二进制协议缓冲(Content-

Type: application/x-protobuf)。第一次请求时，调度器给地址“/scheduler”发送**订阅**，返回应答流的结果(“200 OK” 状态码的传输编码方式：分块传输)。**调度器会尽可能的保持订阅连接(除了网络、软件、硬件等错误。)并且增量处理响应** (注意: HTTP客户端的开发库只能对连接后的响应发生连接不可用的情况进行分析)。关于编码的使用，请参考下面的**事件**部分。

所有向“/scheduler”地址订阅之后的（非订阅）请求（详细见下面的调用段落）必须使用不同的连接方式发送给订阅的一个不同的连接 必须以使用订阅的不同连接方式进行发送。Mesos会对这些HTTP POST请求进行响应，“202 Accepted”状态码（或者，如果请求不成功，返回4xx或者5xx的状态码；在后面的章节中介绍）。“202 Accepted”的响应意味着接受处理请求，而不是请求已经被处理完成。这些请求不能确定经Mesos执行。（举例来说，管理器在处理请求时失败）。从这些请求派生的任何异步的响应，响应被传输到长生命周期的订阅连接。

调用Calls

调用之后的属于正常请求。信息的标准来源是scheduler.proto(null)（注意：在beta API完成后协议缓冲定义会经受改变）。注意，当发送以JSON编码的调用时，调度器应当按照把UTF-8字符串按照Base64原始比特进行编码。

订阅

这是调度器和管理器通信过程的第一步。我们考虑后把订阅划分到“/scheduler”事件流。

向管理器进行订阅时，调度器发送以FrameworkInfo请求的编码SUBSCRIBE信息的HTTP POST请求。注意：没有设置“subscribe.framework_info.id”，管理器认为是一个新的调度器，给他分配FrameworkID进行订阅。在第一个SUBSCRIBE事件，HTTP响应是一个RecordIO编码的流。（请看下面事件章节）。

订阅请求 (JSON):

```
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json
Accept: application/json
Connection: close

{
    "type": "SUBSCRIBE",
    "subscribe": {
        "framework_info": {
            "user": "foo",
            "name": "Example HTTP Framework"
        },
        "force": true
    }
}
```

```
订阅响应事件 (JSON):
HTTP/1.1 200 OK

Content-Type: application/json
Transfer-Encoding: chunked

<event length>
{
  “type”          : “SUBSCRIBED”,
  “subscribed”    : {
     “framework_id”              : { “value” : “12220-3440-12532-2345” },
    "heartbeat_interval_seconds" : 15
  }
}
<more events>
```

另外，如果设置“subscribe.framework_info.id”，管理器认为这是一个从已经订阅过的调度器断开，再次连接的请求(举例来说, 由于故障或者网络断开)并且SUBSCRIBED事件的应答包含同一FrameworkID。“subscribe.force”字段描述如下，当分布的调度器实例（具有相同framework id）如何试图在同一时间对管理器发出订阅，管理器如何执行（举例来说，由于分区网络）。请看下面的断开章节的定义部分。

注意：旧版本的API，（重新）注册回调函数同样也包换MasterInfo，关于管理器驱动所包含现有连接的信息。新的API，自从调度器明确订阅领导管理器（请看下面的章节，在管理器检测部分）今后不在相关。

无论什么原因发生订阅失败（例如，无效的请求），作为信息关闭主体的一部分返回一个的HTTP 4xx响应的错误信息。

仅仅在调度器开启持久链接通过发送SUBSCRIBE请求和接收SUBSCRIBED响应之后,调度器必须向“/scheduler”地址执行额外的HTTP请求。调用在没有订阅的情况下将会导致以“403 Forbidden“应答代替“202 Accepted“应答的结果。如果HTTP请求错误（例如，扭曲了HTTP报文头），调度器还可能接收“400 Bad Request”应答。

卸载

当调度器希望把自己终结时，由调度器发送卸载调用。当Mesos接收这一请求，将会关闭所有执行器（并且因此杀死任务）并且移除持久的卷（如果请求过）。之后删除计算框架并关闭所有调度器向管理器开启的连接。

```
TEARDOWN Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json
```

```
{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “TEARDOWN”,
}

TEARDOWN Response:
HTTP/1.1 202 Accepted
```

接受ACCEPT

当调度器接受管理器发送的票据时，由调度器发送接受调用。ACCEPT请求包含几种操作（例如，启动任务，后备资源，创建卷）这几种操作调度器希望在票据上执行。注意：直到调度器给票据回复（接受或者拒绝），那些资源给计算框架深思熟虑的分配号。同样的，在ACCEPT调用的票据的资源（例如，任务启动）不能使用，那些经过考虑拒绝的可能被重新定给其他计算框架。换句话说，具有相同的OfferID不能用于多于一个ACCEPT调用。当我们给Mesos添加新的特征，这些定义可能会发生变化（例如，持久化，可保留的，乐观的票据，调整任务，等等）。

```
ACCEPT Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “ACCEPT”,
   “accept”           : {
     “offer_ids”      : [
                  { “value” : “12220-3440-12532-O12” },
                  { “value” : “12220-3440-12532-O12” }
                 ],
     “operations”     : [ { “type” : “LAUNCH”, “launch” : {...}} ],
     “filters”        : {...}
  }
}

ACCEPT Response:
HTTP/1.1 202 Accepted
```

拒绝DECLINE

为了明确拒绝接收票据，由调度器发送拒绝调用。注意：拒绝调用同样会发送没有操作的

ACCEPT调用。

```
DECLINE Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “DECLINE”,
   “decline”          : {
    “offer_ids” : [
                { “value” : “12220-3440-12532-O12” },
                { “value” : “12220-3440-12532-O13” }
               ],
    “filters”    : {...}
  }
}

DECLINE Response:
HTTP/1.1 202 Accepted
```

恢复REVIVE

为了删除任何/所有的事先通过设定ACCEPT或者DECLINE调用的过滤器，通过调度器发送恢复调用。

```
REVIVE Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “REVIVE”,
}

REVIVE Response:
HTTP/1.1 202 Accepted
```

杀死KILL

为了杀死特定的任务，由调度器发送调用。如果调度器有一个特定的执行器，杀死调用发送给执行器并且直到执行器来杀死任务后，发送TASK*KILLED (或者TASK*FAILED)更新。Mesos一旦接收任务的终止状态更新，释放任务资源。如果是对于管理器不知道任务状态，将会产生TASK_LOST。

```
KILL Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “KILL”,
   “kill”             : {
     “task_id”    : { “value” : “12220-3440-12532-my-task” },
     “agent_id”   : { “value” : “12220-3440-12532-S1233” }
  }
}

KILL Response:
HTTP/1.1 202 Accepted
```

关闭SHUTDOWN

为了关闭特定的定制的执行器，由调度器发送关闭调用（注意：这个是新调用，在旧API中不会出现）。当执行器获取关闭事件，将会杀死它名下所有的任务（并且发送TASK*KILLED更新）并且终止。如果执行器在一定时间内（通过“–executor*shutdown*grace*period”代理标志配置）没有终止，代理会强行销毁容器（执行器和相关任务）并且把对应的运行状态的任务转换为TASK_LOST。

```
SHUTDOWN Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “SHUTDOWN”,
   “shutdown”         : {
     “executor_id”    : { “value” : “123450-2340-1232-my-executor” },
     “agent_id”       : { “value” : “12220-3440-12532-S1233” }
  }
```

```
}

SHUTDOWN Response:
HTTP/1.1 202 Accepted
```

确认ACKNOWLEDGE

由调度器发送确认状态更新。注意：随着新的API，调度器负责通过“status.uuid()”设置明确地确认状态更新收到。这些更新状态会确实地重试，知道他们通过调度器确认。当“status.uuid()”设置为不能重试时，调度器必须不能够对状态更新进行确认。“uuid”是用Base64原始二进制编码。

```
ACKNOWLEDGE Request (JSON):
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “ACKNOWLEDGE”,
   “acknowledge”      : {
     “agent_id”   : { “value” : “12220-3440-12532-S1233” },
     “task_id”    : { “value” : “12220-3440-12532-my-task” },
     “uuid”       :  “jhadf73jhakdlfha723adf”
  }
}

ACKNOWLEDGE Response:
HTTP/1.1 202 Accepted
```

调解RECONCILE

由调度器发送调解，为了查询未终结的任务状态。导致管理器返回所有任务列表的UPDATE事件。那些不在被Mesos了解状态的任务导致TASK_LOST更新。如果任务列表为空，管理器为了所有当前已知任务将给计算框架发送UPDATE事件。

```
RECONCILE Request (JSON):
POST /api/v1/scheduler   HTTP/1.1

Host: masterhost:5050
Content-Type: application/json
```

```
{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “RECONCILE”,
   “reconcile”        : {
     “tasks”      : [
                { “task_id” : { “value” : “312325” },
                  “agent_id” : { “value” : “123535” }
                }
               ]
  }
}

RECONCILE Response:
HTTP/1.1 202 Accepted
```

消息MESSAGE

调度给执行器发送指挥二进制数据。注意：对于传递给执行器的这些信息，Mesos既不会解析这些数据也不会对这些数据做出任何保证。“data”是以Base64的原始二进制编码。

```
MESSAGE Request (JSON):
POST /api/v1/scheduler   HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “MESSAGE”,
   “message”          : {
     “agent_id”       : { “value” : “12220-3440-12532-S1233” },
     “executor_id”     : { “value” : “my-framework-executor” },
     “data”           : “adaf838jahd748jnaldf”
  }
}

MESSAGE Response:
HTTP/1.1 202 Accepted
```

请求REQUEST

从master/allocator由调度器发送资源请求。内嵌分层分配器简单的忽略掉这些请求，但是其他的

分配器（模块）可以以定制的方式去解释请求。

```
Request (JSON):
POST /api/v1/scheduler   HTTP/1.1

Host: masterhost:5050
Content-Type: application/json

{
   “framework_id”     : { “value” : “12220-3440-12532-2345” },
   “type”             : “REQUEST”,
   “requests”         : [
     {
        “agent_id”       : { “value” : “12220-3440-12532-S1233” },
        “resources”      : {}
     },
 ]
}

REQUEST Response:
HTTP/1.1 202 Accepted
```

事件Events

调度器希望对“/scheduler”地址保持持久连接的开启状态，甚至在获取SUBSCRIBED HTTP应答事件之后。通过没有设置“Content-Length”的“Connection: keep-alive”和“Transfer-Encoding: chunked”头来表明。在链接上，所有后续事件实质是计算框架通过Mesos产生的数据流。管理器在RecordIO格式下编码所有事件，举例来说，用字符串长度表现事件，以字节为单位通过JSON或者二进制Protobuf（可能有压缩）编码事件。注意：长度值永远不会是‘0’并且长度的大小将是无符号整形（64位）。同时，注意：RecordIO编码应当通过调度器解码，然而底层HTTP块的编码通常在应用（调度器）层是看不到的。 用于事件的内容类型的编码由POST请求（例如，Accept: application/json）的报文头确定。

以下事件通常由管理器发送。标准资料来源在scheduler.proto。注意：当发送JSON编码的事件，管理器以UTF-8用字符串和Base64的形式编码原始字节。

订阅SUBSCRIBED

管理器发送的第一个事件是订阅事件，当调度器在持久连接中发送SUBSCRIBE请求。请看当前版面SUBSCRIBE的调用部分。

票据OFFERS

无论何时有新资源可以提供给计算框架时由管理器发送票据事件。在代理上所有票据对应一组资源。除非发生取消票据,例如因为丢失代理或者票据超时(--offer_timeout)，否则直到调度器‘接受

’或者'拒绝’，这些资源会经过仔细考虑后分配给调度器。

```
OFFERS Event (JSON)
<event-length>
{
   “type”     : “OFFERS”,
   “offers”   : [
   {
      “offer_id” :{ “value” : “12214-23523-O235235” },
      “framework_id” :{ “value” : “12124-235325-32425” },
      “agent_id” :{ “value” : “12325-23523-S23523” },
      “hostname” : “agent.host”,
      “resources” :[...],
      “attributes” :[...],
      “executor_ids” :[]
   }
 ]
}
```

解除RESCIND

由管理器发送解除事件，当不在有效的特定票据（举例，应当由代理保持一致的票据，被删除后）时因此需要解除。在未来的调用（ACCEPT / DECLINE）通过调度器关联的票据是无效的。

```
RESCIND Event (JSON)
<event-length>
{
   “type”     : “RESCIND”,
   “rescind”  : {
     “offer_id”   : { “value” : “12214-23523-O235235” }
  }
}
```

更新UPDATE

无论何时由执行器、代理或者管理器产生状态更新通过管理器发送。状态更新应为了管理这些任务当通过执行器用于可靠传输状态。至关重要的是，一旦任务终止由执行器发送的终止更新（举例，TASK*FINISHED, TASK*KILLED, TASK_FAILED），为了Mesos去释放已经分配给任务的资源。 同样是调度器负责，为了明确地告知以收到状态更新的收据，会反复进行可靠性发送。请看ACKNOWLEDGE的Call在上面的部分的定义。注意：uuid和数据的原始字节编码是Base64。

```
UPDATE Event (JSON)
```

```
<event-length>
{
   “type”    : “UPDATE”,
   “update”  : {
     “status”    : {
        “task_id”   : { “value” : “12344-my-task” },
        “state”     : “TASK_RUNNING”,
        “source”    : “SOURCE_EXECUTOR”,
        “uuid”      : “adfadfadbhgvjayd23r2uahj”,
       "bytes"    : "uhdjfhuagdj63d7hadkf"

     }
  }
}
```

消息MESSAGE

通过管理器，执行器产生定义信息由调度器进行转发。注意：信息是通过Mesos不会进行解析，仅仅进行向调度器转发（不保证可靠性）。如果信息以任何原因丢失执行器将会重试。注意：数据的原始字节编码为Base64。

```
MESSAGE Event (JSON)

<event-length>
{
   “type”    : “MESSAGE”,
   “message” : {
     “agent_id”      : { “value” : “12214-23523-S235235” },
     “executor_id”   : { “value” : “12214-23523-my-executor” },
     “data”          : “adfadf3t2wa3353dfadf”
  }
}
```

故障FAILURE

管理器当代理移除时（举例，错误的健康检查）或者当执行器终止时给集群发送故障事件。注意：故障事件伴随着如下两个事件同时发生，属于代理或者执行器任何活跃任务中断UPDATE事件，所有属于代理的未完成要约收到RESCIND事件。注意：不能保证FAILURE、UPDATE和RESCIND事件的顺序。

```
FAILURE Event (JSON)

<event-length>
```

```
{
  “type”    : “FAILURE”,
  “failure” : {
    “agent_id”      : { “value” : “12214-23523-S235235” },
    “executor_id”   : { “value” : “12214-23523-my-executor” },
    “status”        : 1
  }
}
```

错误ERROR

当异步错误事件发生时，管理器发送ERROR事件（举例，计算框架是不会向给定的角色授权订阅）订阅权限是采用选举的方式产生，计算框架当接收错误并尝试订阅时必须终止。

```
ERROR Event (JSON)

<event-length>
{
  “type”    : “ERROR”,
  “message” : “Framework is not authorized”
}
```

心跳HEARTBEAT

这个事件是由管理器定期给调度器发送连接是激活状态的信息。同时也能帮助确保网络中间设备不会因为缺乏数据流而关闭持久订阅连接。请看下一部分，调度器使用这个事件来解决网络分区。

```
HEARTBEAT Event (JSON)

<event-length>
{
  “type”    : “HEARTBEAT”,
}
```

断开

管理器如果注意到调度器持续的向“/scheduler”订阅链接（通过SUBSCRIBE请求开启）请求，则断开调度器链接。链接可能会有若干种断开原因，举例来说，调度器重启、调度器失效、网络错误。注意：管理器不会持续跟踪“/scheduler”的非订阅链接，因为不需要成为持续链接。

如果管理器了解订阅链接断开，将会给调度器以“disconnected”标记并启动故障转移超时（故障转移超时是FrameworkInfo的一部分）。另外会把队列中任何挂起的事件丢弃。此外，会以“403 Forbidden”拒绝随后向“/scheduler”发起的没有订阅的HTTP请求，直到调度器再次发送向“/scheduler”订阅。如果故障转移超时后调度器没有再次订阅，管理器认为调度器永久丢失并且

关闭该调度器所有的执行器，从而杀死所有的任务。因此，我们推荐所有生产的调度器使用最高等级（举例来说，4周）的故障转移超时。

注意：在计算框架超时过后，会强制关闭计算框架（例如，在计算框架开发和测试时候），或者计算框架可以发送TEARDOWN调用（属于Scheduler API的部分），或者可以用地址“/master/teardown”操作（属于Operator API部分）。

如果调度器知道向“/scheduler”发送的订阅链接断开，将会试图向“/scheduler”开启新的持久链接（在管理器检查的结果的基础上，尽可能新的管理器）和订阅。直到获取SUBSCRIBED事件才会向“/scheduler”发送新的非订阅HTTP请求；这样的请求将导致“403 Forbidden”。

如果管理器不知道订阅链接断开，但是调度器知道，调度器可能向“/scheduler”通过SUBSCRIBE开启新持久链接。在这种情况下，依据协议中定义的subscribe.force的值。如果设置为true，管理器关闭已经存在的订阅链接并允许新链接的订阅。如果设置未false，新链接尝试驳回现存的链接。这里不变的是，在管理器中只允许给出FrameworkID的持久订阅链接。对于高可用性集群的调度器，我们推荐把调度器实例的subscribe.force值设置为true，为了随后所有尝试重新连接的调度器（举例，由于断开或者管理器故障转移），仅仅把选举出来的调度器的值设置为false。

网络分裂Network partitions

在网络分裂的情况下，在调度器和管理器之间的订阅链接可能不发生断开。假设能够检测到这种情况，管理器周期性的（举例，15秒）发送HEARTBEAT心跳事件（按照Twitter的Streaming API相似的脉络）。如果调度器没有在一定时间范围内（举例，5）收到一组心跳，将会立刻断开链接并尝试重新订阅。上面的策略高度推荐给调度器使用，这是一个指数级的退避策略（举例，最大上限15秒）可以避免在极端情况下管理器重新连接时发生。调度器为了那些HTTP请求可以用类似超时（举例，75秒）接受应答。

管理器检测

Mesos具有分布式Mesos管理器的高可用模式；一个主管理器（最新的文档称之为领导者或者主管理器）并且在他失败的情况下有几个备份。管理者在ZooKeeper协调下选举出领导者。更多情况请了解相关文档(null)。

调度器预计给主管理器发送HTTP请求。如果作为一个无主管理器的“HTTP 307 Temporary Redirect”请求 调度器将会接收指向主管理器”Location“报文头。

以订阅流程举例，当调度器发现没有主管理器时进行重定向。

```
Scheduler → Master
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost1:5050
Content-Type: application/json
Accept: application/json
Connection: keep-alive

{
   “framework_info”  : {
     “user”  :  “foo”,
     “name”  :  “Example HTTP Framework”
```

```
  },
    “type”              :  “SUBSCRIBE”
}

Master → Scheduler
HTTP/1.1 307 Temporary Redirect
Location: masterhost2:5050


Scheduler → Master
POST /api/v1/scheduler  HTTP/1.1

Host: masterhost2:5050
Content-Type: application/json
Accept: application/json
Connection: keep-alive

{
    “framework_info”    : {
      “user”  :  “foo”,
      “name”  :  “Example HTTP Framework”
  },
    “type”              :  “SUBSCRIBE”
}
```

如果调度器知道某集群的管理器域名清单，就可以使用上面这种机制查找到负责订阅的主管理器。另外，调度器可以通过设定ZooKeeper（或者etcd）URL使用Pure语言库来检测主管理器。对于C++库，ZooKeeper基于管理器检测，请查看src/scheduler/scheduler.cpp

Javadoc

包org.apache.mesos

接口概述

接口名称	描述
[Executor](http://mesos.apache.org/api/latest/java/org/apache/mesos/Executor.html)	框架的执行器以这个回调接口的方式来实现。

[ExecutorDriver](http://mesos.apache.org/api/latest/java/org/apache/mesos/ExecutorDriver.html)	执行器链接到Mesos的抽象接口。
[Scheduler](http://mesos.apache.org/api/latest/java/org/apache/mesos/Scheduler.html)	框架的调度器以这个回调接口的方式来实现。
[SchedulerDriver](http://mesos.apache.org/api/latest/java/org/apache/mesos/SchedulerDriver.html)	调度器链接到Mesos的抽象接口。

类的概述

类	描述
[Log](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.html)	提供了一个分布追加日志的入口。
[Log.Entry](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.Entry.html)	代表Log.Position日志的隐式数据入口。
[Log.Position](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.Position.html)	在日志记录位置的隐式标识符。
[Log.Reader](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.Reader.html)	提供日志的读取访问接口。
[Log.Writer](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.Writer.html)	提供日志的写访问接口。
[MesosExecutorDriver](http://mesos.apache.org/api/latest/java/org/apache/mesos/MesosExecutorDriver.html)	ExecutorDriver的具体实现，把执行器和Mesos节点链接。
[MesosSchedulerDriver](http://mesos.apache.org/api/latest/java/org/apache/mesos/MesosSchedulerDriver.html)	SchedulerDriver的具体实现，把调度器和Mesos节点链接。

异常概述

异常	描述
[Log.OperationFailedException](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.OperationFailedException.html)	在执行读写操作的过程中抛出的异常
[Log.WriterFailedException](http://mesos.apache.org/api/latest/java/org/apache/mesos/Log.WriterFailedException.html)	写对象不再具备执行操作时抛出异常（举例来说，由其他写对象取代）

Doxygen

Mesos命名空间参考

授权给Apache基金会(ASF)下的一个或者多个贡献者协议。更多...(null)

Classes

	名称	描述
class	[Executor](http://mesos.apache.org/api/latest/c++/classmesos_1_1Executor.html)	框架的执行器以这个回调接口的方式来实现。[更多...](http://mesos.apache.org/api/latest/c++/classmesos_1_1Executor.html#details)
class	[ExecutorDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1ExecutorDriver.html)	执行器链接到Mesos的抽象接口。[更多...](http://mesos.apache.org/api/latest/c++/classmesos_1_1MesosExecutorDriver.html#details)

class	[MesosExecutorDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1MesosExecutorDriver.html)	[ExecutorDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1ExecutorDriver.html)的具体实现，把 [Executor](http://mesos.apache.org/api/latest/c++/classmesos_1_1Executor.html)和Mesos节点链接。[更多 ...](http://mesos.apache.org/api/latest/c++/classmesos_1_1ExecutorDriver.html#details)
class	[MesosSchedulerDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1MesosSchedulerDriver.html)	[SchedulerDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1SchedulerDriver.html)的具体实现，把 [Scheduler](http://mesos.apache.org/api/latest/c++/classmesos_1_1Scheduler.html)和Mesos节点链接。[更多 ...](http://mesos.apache.org/api/latest/c++/classmesos_1_1MesosSchedulerDriver.html#details)
class	[Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html)	
class	[Scheduler](http://mesos.apache.org/api/latest/c++/classmesos_1_1Scheduler.html)	框架的调度器以这个回调接口的方式来实现。[更多 ...](http://mesos.apache.org/api/latest/c++/classmesos_1_1Scheduler.html#details)

class	[SchedulerDriver](http://mesos.apache.org/api/latest/c++/classmesos_1_1SchedulerDriver.html)	调度器链接到Mesos的抽象接口。[更多...](http://mesos.apache.org/api/latest/c++/classmesos_1_1SchedulerDriver.html#details)

Functions

返回值	函数体
bool	**operator==** (const Resource &left, const Resource &right)
bool	**operator!=** (const Resource &left, const Resource &right)
bool	**operator<=** (const Resource &left, const Resource &right)
Resource	**operator+** (const Resource &left, const Resource &right)
Resource	**operator-** (const Resource &left, const Resource &right)
Resource &	**operator+=** (Resource &left, const Resource &right)
Resource &	**operator-=** (Resource &left, const Resource &right)
bool	**matches** (const Resource &left, const Resource &right)
std::ostream &	**operator<<** (std::ostream &stream, const Resource &resource)
std::ostream &	**operator<<** (std::ostream &stream, const [Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html) &resources)

std::ostream &	**operator<<** (std::ostream &stream, const google::protobuf::RepeatedPtrField< Resource > &resources)
[Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html)	**operator+** (const google::protobuf::RepeatedPtrField< Resource > &left, const [Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html) &right)
[Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html)	**operator-** (const google::protobuf::RepeatedPtrField< Resource > &left, const [Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html) &right)
bool	**operator==** (const google::protobuf::RepeatedPtrField< Resource > &left, const [Resources](http://mesos.apache.org/api/latest/c++/classmesos_1_1Resources.html) &right)
std::ostream &	**operator<<** (std::ostream &stream, const Value::Scalar &scalar)
bool	**operator==** (const Value::Scalar &left, const Value::Scalar &right)
bool	**operator<=** (const Value::Scalar &left, const Value::Scalar &right)
Value::Scalar	**operator+** (const Value::Scalar &left, const Value::Scalar &right)
Value::Scalar	**operator-** (const Value::Scalar &left, const Value::Scalar &right)
Value::Scalar &	**operator+=** (Value::Scalar &left, const Value::Scalar &right)

Value::Scalar &	**operator-=** (Value::Scalar &left, const Value::Scalar &right)
std::ostream &	**operator<<** (std::ostream &stream, const Value::Ranges &ranges)
bool	**operator==** (const Value::Ranges &left, const Value::Ranges &right)
bool	**operator<=** (const Value::Ranges &left, const Value::Ranges &right)
Value::Ranges	**operator+** (const Value::Ranges &left, const Value::Ranges &right)
Value::Ranges	**operator-** (const Value::Ranges &left, const Value::Ranges &right)
Value::Ranges &	**operator+=** (Value::Ranges &left, const Value::Ranges &right)
Value::Ranges &	**operator-=** (Value::Ranges &left, const Value::Ranges &right)
std::ostream &	**operator<<** (std::ostream &stream, const Value::Set &set)
bool	**operator==** (const Value::Set &left, const Value::Set &right)
bool	**operator<=** (const Value::Set &left, const Value::Set &right)
Value::Set	**operator+** (const Value::Set &left, const Value::Set &right)
Value::Set	**operator-** (const Value::Set &left, const Value::Set &right)
Value::Set &	**operator+=** (Value::Set &left, const Value::Set &right)
Value::Set &	**operator-=** (Value::Set &left, const Value::Set &right)

std::ostream &	**operator<<** (std::ostream &stream, const Value::Text &value)
bool	**operator==** (const Value::Text &left, const Value::Text &right)
std::ostream &	**operator<<** (std::ostream &stream, const Attribute &attribute)
ostream &	**operator<<** (ostream &stream, const Resource &resource)
bool	**operator==** (const Environment &left, const Environment &right)
bool	**operator==** (const CommandInfo &left, const CommandInfo &right)
bool	**operator==** (const ExecutorInfo &left, const ExecutorInfo &right)
bool	**operator==** (const SlaveInfo &left, const SlaveInfo &right)
bool	**operator==** (const MasterInfo &left, const MasterInfo &right)
std::ostream &	**operator<<** (std::ostream &stream, const FrameworkID &frameworkId)
std::ostream &	**operator<<** (std::ostream &stream, const OfferID &offerId)
std::ostream &	**operator<<** (std::ostream &stream, const SlaveID &slaveId)
std::ostream &	**operator<<** (std::ostream &stream, const TaskID &taskId)
std::ostream &	**operator<<** (std::ostream &stream, const ExecutorID &executorId)
std::ostream &	**operator<<** (std::ostream &stream, const ContainerID &containerId)

std::ostream &	**operator<<** (std::ostream &stream, const TaskState &state)
std::ostream &	**operator<<** (std::ostream &stream, const TaskInfo &task)
std::ostream &	**operator<<** (std::ostream &stream, const SlaveInfo &slave)
std::ostream &	**operator<<** (std::ostream &stream, const ExecutorInfo &executor)
std::ostream &	**operator<<** (std::ostream &stream, const MasterInfo &master)
std::ostream &	**operator<<** (std::ostream &stream, const ACLs &acls)
std::ostream &	**operator<<** (std::ostream &stream, const RateLimits &limits)
bool	**operator==** (const FrameworkID &left, const FrameworkID &right)
bool	**operator==** (const FrameworkInfo &left, const FrameworkInfo &right)
bool	**operator==** (const Credential &left, const Credential &right)
bool	**operator==** (const OfferID &left, const OfferID &right)
bool	**operator==** (const SlaveID &left, const SlaveID &right)
bool	**operator==** (const TaskID &left, const TaskID &right)
bool	**operator==** (const ExecutorID &left, const ExecutorID &right)
bool	**operator==** (const ContainerID &left, const ContainerID &right)

bool	**operator!=** (const ContainerID &left, const ContainerID &right)
bool	**operator==** (const FrameworkID &left, const std::string &right)
bool	**operator==** (const OfferID &left, const std::string &right)
bool	**operator==** (const SlaveID &left, const std::string &right)
bool	**operator==** (const TaskID &left, const std::string &right)
bool	**operator==** (const ExecutorID &left, const std::string &right)
bool	**operator==** (const ContainerID &left, const std::string &right)
bool	**operator<** (const FrameworkID &left, const FrameworkID &right)
bool	**operator<** (const OfferID &left, const OfferID &right)
bool	**operator<** (const SlaveID &left, const SlaveID &right)
bool	**operator<** (const TaskID &left, const TaskID &right)
bool	**operator<** (const ExecutorID &left, const ExecutorID &right)
bool	**operator<** (const ContainerID &left, const ContainerID &right)
bool	**operator==** (const CommandInfo::URI &left, const CommandInfo::URI &right)
std::size_t	**hash_value** (const FrameworkID &frameworkId)

std::size_t	**hash_value** (const OfferID &offerId)
std::size_t	**hash_value** (const SlaveID &slaveId)
std::size_t	**hash_value** (const TaskID &taskId)
std::size_t	**hash_value** (const ExecutorID &executorId)
std::size_t	**hash_value** (const ContainerID &containerId)
ostream &	**operator<<** (ostream &stream, const Value::Scalar &scalar)
ostream &	**operator<<** (ostream &stream, const Value::Ranges &ranges)
ostream &	**operator<<** (ostream &stream, const Value::Set &set)
ostream &	**operator<<** (ostream &stream, const Value::Text &value)
double	**getScalarResource** (const Offer &offer, const std::string &name)

Variables

[tuple](mesos.apache.org/api/latest/c++/classtuple.html)	**__path__** = extend_path(__path__, __name__)

详细描述

一个或者多个捐助者同意授权给Apache软件基金会(ASF)。

可以在*注意*文件看到， 分配著作所有权相关的额外信息。在Apache授权下的ASF授权文件，版本2.0("许可证");除非符合授权，否则不能使用该文件。你可以在如下地址获取许可证副本

```
http://www.apache.org/licenses/LICENSE-2.0
```

除非以适用法律或者书面同意，在许可证下分发软件以*"现状"*为基础，*没有任何种类的担保或者条件*，无论明示或者暗示。查看许可证的具体语言管理权限和限制。 详细的语言控制权限和限制 查看许可证详细的语言控制权限和限制。Mesos执行器接口和执行器驱动。一个执行器是需要在计算框架内以明确的方式启动任务(举例来说，创建新线程、新进程，等等)。同一计算框架的一个或者

多个执行器需要同时运行在同样的计算设备。注意：我们使用“执行器”这个术语相当随意的描述实现Executor(null)接口的代码（见下文）除此之外，程序还负责实例化一个新的MesosExecutorDriver(null)（也在下文）。事实上，当一个Mesos节点（分支）负责执行“执行器”，没有任何理由，应当执行它自己的节点去实际上执行另外一个实质上实例化和运行MesosSchedulerDriver(null)程序。直到“执行器”运行完结为止，担任运行的执行器的节点程序才能够调用到。 因此，那些节点的可能仅仅执行一个实际上（交叉或者等待）真正的执行器运行的脚本。

如你你发现这里有修改意见，请确认对其他语言做相同的变更 (举例来说, Java: src/java/src/org/apache/mesos, Python: src/python/src, 等等).

可以在*注意*文件看到， 分配著作所有权相关的额外信息。在Apache授权下的ASF授权文件，版本2.0("许可证");除非符合授权，否则不能使用该文件。你可以在如下地址获取许可证副本

```
http://www.apache.org/licenses/LICENSE-2.0
```

除非以适用法律或者书面同意，在许可证下分发软件以*"现状"*为基础，*没有任何种类的担保或者条件*，无论明示或者暗示。查看许可证的具体语言管理权限和限制。Resources(null) 有三种类型：标量、范围和集合。使用协议缓冲区来表示。为了使Mesos核心和调度器的编写者更容易操作这些资源，我们提供了泛型重载运算器（见下文）以及把协议缓冲区资源对象组成的集，合封装成为常规Resources(null)类。Resources(null)类还提供了一些允许分析资源的静态程序(举例来说，从命令行)，以及确认资源对象的有效和可分配。注意：许多的操作没有达到最佳，反而仅仅编写正确的语义。

注意！资源通过元组（名字、类别、作用）来描述。做"算术"操作（下面定义的）两个资源系相同名称但是不同列别、或者相同名称类别但不同作用、没有定义、因此虽然在定义上进行二个操作数吗，实际上只是一个空的资源（好像你没有做操作一样）。此外，相同类型但不同名称的两种资源的操作是没有实际运算。

可以在*注意*文件看到， 分配著作所有权相关的额外信息。在Apache授权下的ASF授权文件，版本2.0("许可证");除非符合授权，否则不能使用该文件。你可以在如下地址获取许可证副本

```
http://www.apache.org/licenses/LICENSE-2.0
```

除非以适用法律或者书面同意，在许可证下分发软件以*"现状"*为基础，*没有任何种类的担保或者条件*，无论明示或者暗示。查看许可证的具体语言管理权限和限制。 Mesos调度器接口和调度器驱动。调度器与Mesos交互，用来规划运行分布式计算。

如你你发现这里有修改意见，请确认对其他语言做相同的变更 (举例来说, Java: src/java/src/org/apache/mesos, Python: src/python/src, 等等)。

可以在*注意*文件看到， 分配著作所有权相关的额外信息。在Apache授权下的ASF授权文件，版本2.0("许可证");除非符合授权，否则不能使用该文件。你可以在如下地址获取许可证副本

```
http://www.apache.org/licenses/LICENSE-2.0
```

除非以适用法律或者书面同意，在许可证下分发软件以*"现状"*为基础，*没有任何种类的担保或者条件*，无论明示或者暗示。查看许可证的具体语言管理权限和限制。

开发者工具

辅助工具

辅助操控工具

下面工具可方便建立和运行Mesos集群。

• collectd-mesos(null) Mesos集群常用监控功能。

• 部署脚本(null) 用于把Mesos部署到一群机器里。。

• Everpeace版整体解决方案(null) 安装和配置Mesos的主控和被控。也可完全从源码构建。

• Mdsol版整体解决方案(null) 安装Mesos集群的工具，此工具使用Mesosphere的包来安装Mesos。

• Deric的Puppet 模块(null) Puppet模块，用于管理集群中的Mesos节点。

• Everpeace版Vagrant构建Mesos(null) 使用Vagrant来构建集群。

• Mesosphere版Vagrant构建Mesos(null) 使用Vagrant来快速构建Mesos沙箱环境。

开发用工具

• Go 绑定和示例(null) 用go语言写Mesos应用：调派器和执行器。

• Mesos 应用giter8模板(null) giter8模板，scala语言，使用SBT编译，Vagrant 部署到集群测试。

• Scala入门(null) 一个简单的Mesos示例：下载代码以及在集群的节点中部署一个网络服务器。

• Xcode 工场(null) 在Xcode中修改Mesos。

若上面列表没有你要列的，可提交一个补丁，或者电邮user@mesos.apache.org。

对 Mesos 贡献

报告问题，改进，或功能

使用JIRA反馈问题，改进建议和功能请求。

当您有问题要反馈时 (例如 bug、 改进建议、 功能请求)，请尽可能多地提供上下文。

• 我们通过 Apache 的 JIRA(null) 跟踪所有问题，所以如果你还没有帐户，那就注册一个。这很简单，也很快。

• 应该为每个task，feature，bug-fix都创建一个JIRA，这样便于追踪进度。

• 如果你打算加入，在开始之前请把JIRA问题分配给自己。这有助于避免重复工作，也让大家知道谁在跟这个问题。我们非常鼓励大家在社区中互相讨论如何解决问题。本贡献指南会更详细的论述如何对Mesos的发展做贡献。

提交补丁

提交补丁

• 你已经修改了一个bug，或增加了一个功能，然后想要提交它。真棒！

• 我们使用 Apache Review Board代码审查工具(null)进行代码审查。如果你还没有帐户，则需要创建一个 (它和您的 Apache JIRA 帐户是不同的)。

• 每个包含了要修改代码库的JIRA都要创建一个代码审查请求。

编码之前

1. 通过Git，把代码从Apache的资源库上check out下来。教程请戳这里(null)。

2. 加入开发，问题跟踪，审查，搭建的邮件列表，请发送邮件至相应邮箱：dev-subscribe@mesos.apache.org, issues-subscribe@mesos.apache.org, reviews-subscribe@mesos.apache.org 和 builds-subscribe@mesos.apache.org。

3. 在JIRA(null)上找一条你想解决的，并且还未分派的JIRA记录，或者你自己创建的记录（你需要JIRA账号，见下文）

> 1. 这条JIRA记录也许是一个bug反馈（可能是你遇到并上报的一个bug. 比如在试图编译的时候碰到的bug.）或是一个新增的功能。
> 2. 相对于仅仅"Open"来说，把问题标记为"Accepted"更好。如果一个问题被接受了，这表示至少有个Mesos开发者认为，这个问题提出的想法后期值得去做。
> 3. 标记为"新手"的标签的问题适用于"启动器"项目。

1. 给自己分派JIRA，你需要如下步骤：

> 1. 一个Apache JIRA用户账户（点击[这里注册](https://issues.apache.org/jira/secure/Signup!default.jspa)）
> 2. 需要让一位Mesos提交者（发邮件到dev@mesos.apache.org）把你添加到Mesos"贡献者"列表里，这样才能被分配（或分配给自己）JIRA问题。

1. 制定计划来解决问题，建议在 JIRA 的评论区写计划

2. 找个**导师**一起完成补丁。导师就是Mesos提交者，他和你一起干活，对你提议的设计给予反馈，最终把你的修改提交到Mesos代码树。

> 1. 可以通过发邮件给dev邮件列表（带上JIRA问题链接）来找个导师。也可以尝试给JIRA问题添加个评论来寻找。
> 2. 还可以在IRC里问开发者们来找个导师（/documentation/latest/in the [mesos channel](irc://irc.freenode.net/mesos) on [Freenode](https://freenode.net/)）。在[这里](http://mesos.apache.org/documentation/latest/submitting-a-patch/committers/)可以找到当前的提交者列表：已经做过你要修改的组件的开发者可能会适合做导师。

创建补丁

1. 创建一个或多个测试用例来验证bug或功能（Mesos团队使用测试驱动开发(null)）。在开始编码之前，确保这些测试用例都失败。

1. [测试 patterns](http://mesos.apache.org/documentation/latest/testing-patterns/)页面提供了一些写测试用例的建议。

1. 确保代码的修改（不管你用哪个IDE/编辑器）真正地修复了bug或实现了功能。

1. 在开始前，请阅读 [Mesos C ++风格 指南](http://mesos.apache.org/documentation/latest/mesos-c++-style-guide/)。建议使用git pre-commit hook（```support/hooks/pre-commit```）自动检查风格错误。请参阅hook脚本的使用说明。
2. 大部分你做的更改应该是```BASE_MESOS_DIR```里的文件
3. 在Mesos根目录里面执行:```./bootstrap```（只有从git 资源库上编译才会用到）
4.如果做编译的话，我们不建议你从源目录编译，建议这样：
    1. 在Mesos根目录里面执行： ```mkdir build && cd build```
    2. ```../configure```
    3. ```make```
    4. 现在编译程序生成的所有文件都在创建的build目录里，而不是零乱地散落在src目录。也更干净，易于清理，比如想删除configure和make生成的文件。也就是说，可以重置编译过程，不需要冒着在src做变更的风险，只需要删除生成目录，并创建一个新的就可以了。

1. 确保你所有的测试用例通过，包括新加的用例：make check

1. 使用：```make tests```来构造所有用例但并不执行它们。
2. 使用：```make check GTEST_FILTER="HTTPTest.Delete"来执行单个单元测试（当debug一个用例失败时有用）。

1. 确保被提交到master主分支上的所有的修改都pull了。在Git中这样操作：

1. ```git checkout master```
2. ```git pull```
3. ```git checkout my_branch```
4. 检查```git diff master```的输出，确保在列表中的只有你的修改。如果有其它你没做的修改也在列表中。使用```git rebase master```命令使你的分支内容是下载master上最新的。

提交补丁

1. 准备提交你的patch去审核！

1. 在Apache Review Board(null)上登录或注册账户。

2. 最快速（也最推荐）的提交方法是通过post-reviews.py 。它是一个对post-review的封装。

3. 首先，安装 RBTools。点这里(null)查看介绍。

4. 配置post-review。最简单的方法是建立符号链接: ln -s support/reviewboardrc .reviewboardrc。

5. 从命令行登录Apache Review Board：执行 rbt status。

6. 在你的本地分支上执行 support/post-reviews.py。

7. 注意：和master不同分支的每次提交，post-reviews.py 都会创建新的review。

8. 确保在“Bugs”字段上加上你的JIRA id（例“MESOS-01”）（这将会自动链接到bug记录）。

9. 在“Reviewers”节里“People”字段下面加上你的导师。参与讨论的其他Mesos社区成员也应该被包含进来。

10. 在“Description”中加上更改的详细信息，包括需要添加任何文件页面的描述，或是受你影响的更改（例如，你是否改变或添加了任何配置选项/标志？你有添加新的编译文件吗？）

11. 在“Testing Done”下面，描述你创建了哪些新用例，修改了哪些用例，你都做了什么来测试更改。

1. 等待其它的Mesos开发者在Review Board进行代码审查，处理他们的反馈意见，并上传最新的补丁，直到您收到其它提交者发过来的"Ship It"。

> 1. 如果没有收到任何反馈，联系你的导师提醒一下他们。虽然提交者们想尽量提供快递反馈，但他们太忙了，有时候会漏掉。
> 2. 当处理反馈意见的时候，修改你当前的提交，而不是创建新的提交，否则```post-reviews.py```又会创建一个新的审查请求（```git rebase -i``` 会帮你不少）。
> 3. Review Board comments should be used for code-specific discussions, and JIRA comments for bigger-picture design discussions.代码相关的讨论应该使用Review Board评论，而宏观的设计讨论应该使用JIRA评论。
> 4. 对每个 RB 注释回应处理，你可以直接回应为 (即每个注释可以直接回应) "完成"或说明你是如何处理它的。
> 5. 审查中如果有问题被发起，请以“Fixed”或者“Dropped”的方式处理它。如果处理为"Dropped"请添加注释说明原因。同时，如果你的修复是授权评论式的 (例如，你的修复方式与评论者建议的方式不同)，也请添加注释。

1. 当你的你 JIRA/补丁修改被通过后，你将收到提交者发来的"Ship It!"，然后提交者将会把你的补丁提交到 git 仓库。祝贺和感谢您参与我们的社区!

2. 最后一步是确保创建或更新了必要的文件，以让全世界都知道你的新功能或 bug 修复。

风格向导

• 关于核心的修补程序，我们建议参考 Mesos C++ Style Guide(null)。

Mesos 测试模式

Mesos测试模式

在 Mesos 测试中，常使用的测试模式是集合。如果你有好的觉得有用的测试方法，请写下来并加上相关的动机和背景。

使用Clock确保事件的执行

调度异步环境中的事件顺序并非易事: 函数调用通常启动操作然后立即返回，而其中的操作在后台运行。一种简单、 明显和并不好的解决方案是使用 os::sleep() 来等待操作完成。在不同的机器上完成的时间不同，而增加了测试执行时间也就增加了睡眠时间以及最终减慢了检查时间。一种好的方法是，做到之一是等待操作一完成就执行。这可能要用到 libprocess 的Clock 检测。

每个消息队列在一个libprocess 进程的 (或执行者，目的是为了避免歧义的操作系统进程) mailbox 是由 ProcessManager (每个 OS 进程ProcessManager 所拥有的当前一个单例 ，但这可能会在以后改变) 处理的。 ProcessManager从可执行者列表中取来执行者列表，然后服务于执行者mailbox中的所有事件。通过调用Clock::settle()，我们可以阻止调用线程，直到 ProcessManager 执行完所有执行者mailbox中的事件。例子如下:

```
// As Master::killTask isn't doing anything, we shouldn't get a status update.
EXPECT_CALL(sched, statusUpdate(&driver, _))
  .Times(0);

// Set expectation that Master receives killTask message.
Future<KillTaskMessage> killTaskMessage =
  FUTURE_PROTOBUF(KillTaskMessage(), _, master.get());

// Attempt to kill unknown task while slave is transitioning.
TaskID unknownTaskId;
unknownTaskId.set_value("2");

// Stop the clock.
Clock::pause();

// Initiate an action.
driver.killTask(unknownTaskId);

// Make sure the event associated with the action has been queued.
AWAIT_READY(killTaskMessage);

// Wait for all messages to be dispatched and processed completely to satisfy
// the expectation that we didn't receive a status update.
Clock::settle();

Clock::resume();
```

拦截一条发送到不同的操作系统进程的消息

截获发送于libprocess 进程(我们称之为actors(执行者)，为了避免与操作系统进程产生歧义)间的消息，这些执行者在同一个操作系统的进程，这个方法是容易的，例子如下:

```
Future<SlaveReregisteredMessage> slaveReregisteredMessage =
```

```
  FUTURE_PROTOBUF(SlaveReregisteredMessage(), _, _);
...
AWAIT_READY(slaveReregisteredMessage);
```

但是，如果我们想要拦截发送到另一个操作系统进程中actor (技术上 是 UPID) 的消息，这个方法就不管用了。例如，在我们的测试中(见 mesos/src/tests/cluster.hpp)，虽然主设备和从设备实例在同一操作系统的进程上，但是slave上的 CommandExecutor 却是在一个独立的操作系统进程中，这样的话wait在代码中将会失效:

```
Future<ExecutorRegisteredMessage> executorRegisteredMessage =
  FUTURE_PROTOBUF(ExecutorRegisteredMessage(), _, _);
...
AWAIT_READY(executorRegisteredMessage);
```

为什么OS进程外部发送的信息不会被拦截？

Libprocess事件也许会被过滤（见libprocess/include/process/filter.hpp ）。FUTURE_PROTOBUF`` 使用了这个功能，它设置了一个过滤期望值，是通过```TestsFilter```类的```filter``` 方法与```MessageMatcher``` 实现的，```MessageMatcher``` 会匹配我们想要拦截的消息。过滤真正起作用的是```ProcessManager::resume()```，它从所接收的事件的队列中提取消息。

发送，编码，或传送信息时不发生过滤（见ProcessManager::deliver() 或 SocketManager::send() ） 。因此，在上述例子中， 通过过滤器，ExecutorRegisteredMessage 会留下未被探测的slave，直达另一个操作系统进程中的执行者，被入队到theCommandExecutorProcess的mailbox,然后在这里过滤，但不要忘了，我们的过滤期望配置是在前一个操作系统的进程中！

考虑在相应的传入消息设置过滤期望值，确保它们被处理，然后再返回确认信息（译者注：ACK消息）。

对于上述的例子，代替截取ExecutorRegisteredMessage，我们可以拦截RegisterExecutorMessage并等到其被处理了，包括发送ExecutorRegisteredMessage （见 Slave::registerExecutor() ） ：

```
Future<RegisterExecutorMessage> registerExecutorMessage =
  FUTURE_PROTOBUF(RegisterExecutorMessage(), _, _);
...
AWAIT_READY(registerExecutorMessage);
Clock::pause();
Clock::settle();
Clock::resume();
```

有效的代码审查

有效的代码审查

代码审查准备

我们发现清晰,干净,小的,独立的,增量更改更容易得到彻底代码审查 ,更容易得到认可。在代码提交审核前，这里有一些参考建议:

1.使用前面讨论的:如果总体设计需要改变，会使审核者惊讶，这最有可能导致努力白费。所以在发送审核请求前，尽量在同一页面中联系审核者。

2.考虑如何终止你的分支:在独立需要审核的分支中，你有做任何改动吗？我们通过使用`

support/post-reviews.py

,这使得易于创建基于提交的“链”。这和互动rebasing(

git rebase -i

`)以分开提交相似。

3.提供上下文的变化情况:明确的动机变化的审查请求,会使审核者不用猜测。强烈建议把JIRA问题与您的审核加到上下文中。

4.遵守代码的风格。

5.在发表前，对自我更改进行回顾:方法是从审稿人的角度来看,如果没有上下文，找出改变的动机容易吗?它符合上下文的代码风格吗?有不必要的变化吗?你做了任何测试吗?

审核开始

1.做高阶回顾之前先做低阶回顾:你进入任何底层细节的代码前考虑全局。如,你明白改变动机?为什么作者选择特定的解决方案?会感觉有更多不必要的复杂地方吗?如果是有,花时间考虑如果有更简单的方法问题,或者是否可以减少工作的范围。这些都应该先得到解决，然后再做一个逐行深入代码评审,后者是受很多评论反复面对改变了整体的方法。

2.有耐心、有思考和尊重感:当提供(a)反馈评论和(b)评论的反馈(要有耐心、有思考和尊重感)。实践ego-less programming ,这并不是一场拳击比赛!审核的先决条件，是承认编程很难!应该给审稿人质疑审稿人的好处,即使他们不能表达得很好。审核应该准备预测所有的评论者关切和思考为什么他们决定做一些特定的方式(尤其是如果它偏离标准代码库)。另一方面,审稿人不应该使用“编程是困难的”这一事实作为匆忙的反馈。评论者应该投入时间和精力试图解释他们的担忧的清晰和具体的情况。这个过程是双向的!

3.解决问题之前:我们倾向于给出一个“‘ship it”,即使我们认为有些问题需要纠正。有时一个特定的问题需要更多的讨论,有时我们会通过IM或IRC或邮件讨论，去加快解决时间。这是很重要的,然而,我们公开的通过这些方式去找讨论的问题。但是会涉及其他想参与但没有邀请到的人。

a. 如果评论者和提交者有问题解决一个特定的“对抗性”问题，然后双方应该考虑邀请另一个评论者参与沟通。我们致力于建立高质量代码,我们应该影响另外的人也这样做。

b. 当一个问题被提交者定位是“已抛弃”状态时,我们期望是必须有一个声音，说明为什么它是被抛弃了。无理由的抛弃是不被鼓励的。

c.如果一个问题被标记为“解决”,我们期望是新旧的不同比照已经更新，(或多或少)参照审核者的评论。如果有重大变化,在issue上的回复评论是我们欣赏的作法。

4.直率要求更多的反馈:随时自由的更新评论，令你欣喜又发愁是,在很多情况下，模棱两可的审核者评审时回“就绪”状态，所以，当审核者准备好时。 提交者应该随时与他们通过电子邮件沟通

5.提交者等待“ship it”:我们经常使用原始的开发人员，或当前的“管理者”，作为一些特定的代码审查者，再添加一些别人的反馈来作参考。值得一提的是,我们经常直接接触另一个关于一个特定的变化/特性之前就开始编码。小的环境通过很长的线化实现审核流程。

6.周密思考，给出"ship it "的审核者:理解审查需要投入大量的时间和注意力:

a.你将会为理解和支持的代码给出“ship it ”

b.你将有责任为高质量的代码给“ship it ” 。

工程原则和实践

工程原则和实践

本文档是为了说明项目共享的价值观。许多公司依靠Mesos 作为基本层的软件基础设施,当务之急是我们建立健壮的、高质量的代码。我们的目标是培养一种文化,我们可以信任和依赖的工作社区。

The following are some of the aspirational principles and practices that guide us:

以下是一些有抱负的原则和实践:

1. 我们重视工艺:代码应该易于阅读和理解,应该用对细节的高度关注,其风格应该是一致的。编写代码是为了使我们阅读和维护!

2. 我们重视弹性 :系统必须可用性、稳定和向后兼容。我们必须考虑系统的失败。

3. 我们重视责任:我们拥有和支持我们的软件,负责提高,解决问题,从错误中学习。

4. 我们重视设计和代码复查:回顾可以帮助我们维持一个高质量的系统架构和高质量的代码,它也帮助我们指导新的贡献者,学会更有效地协作,减少错误。

5. 我们重视自动化测试:自动化测试可以让我们迭代和重构大型代码库,而验证的正确性,防止回归。

6. 我们重视基准:基准测试可以让我们确定正确的位置目标的性能改进。它还允许我们迭代和重构大型代码库,同时观察性能的影响

提交

提交

只有提交者有提交你的新变化代码的能力,所以要确保你在代码审核中代码起作用，以使你的代码可以被提交。

如果你是一个提交者,这里有一些指南:

1.这里是列表文本检查JIRA:确保没有进一步讨论关于补丁中使用的方法。

2.遵循提交格式 :我们目前不利用任何工具来执行消息的格式,所以:

a. 提交消息是清晰和明确的。

b. 包括审查的链接(使用`

support/post-reviews.py

完成自己的提交，

support/apply-review.sh

` 完成合并他人的提交，这会使工作自动完成)。当pull了别人变化的代码 ，一定要再提交一次。

c. 使用72个字符列。注意,我们并不总是有50个字符或少总结,因为限制往往导致人们写的不好。

3.保持“作者”完整: support/apply-review.sh 将为你处理这个问题,但执行rebasing（合并） 或

ammending 时要仔细。

4.不要提交合并:使用rebase合并替代，不要使用强制提交。（会覆盖他人的代码）

5.不要打破构建:我们支持Linux和Mac OS X,但是并不是所有的配置都被Jenkins 测试过,所以要意识到这一点。另外,要注意Jenkins 审查，如果重新build，它将会标记。请注意,如果你打破构建,修复通常很少,且无关紧要，所以不用担心能否通过评审周期(但不要把这个当成一个许可，即“等到构建失败时才去解决问题”)。

提交者

提交者

一个Apache Mesos 提交者是一个有写访问Apache Mesos Apache代码存储库和相关的基础设施的贡献者 。Mesos 项目中,每个提交者也是一个PMC的投票成员。

成为一个提交者

目前提出的每一个新提交者必须被当前提交者提名,然后Mesos PMC的成员投票。这个过程的细节和候选人要求，请参考 Apache guidelines for assessing new candidates for committership(null). 。候选人根据他们的项目贡献及其社区准备他们的提名权,根据Apache Way(null), 和from contributor to committer(null) 。在Mesos项目中,您可以使用Apache Mesos Committer Candidate Checklist(null) 建议什么样的贡献,并演示可以帮助的行为,并记录你的进展。

当前的提交者

我们想感谢以下提交者，是他们的帮助，让Apache Mesos 项目走到了今天。这个列表可能有过期,最实时的请访问Apache’ s website(null).

Timezone GMTFull NameCompany/OrganizationIRC HandleEmail Address

-8Ross Allenssorallen@apache.org

-5Kapil AryaMesosphere / Northeastern Universitykapil@apache.org

-8Adam BMesosphereme@apache.org

-8Tim ChenMesospheretnachentnachen@apache.org

-8Ian DownesTwitteridownesidownes@apache.org

-8Ali GhodsiUC Berkeleyalig@apache.org

-8Dominic Hamondma@apache.org

-8Benjamin HindmanMesosphere*benh*benh@apache.org

-8Ian Holsmanianh@apache.org

-8Vinod KoneTwittervinodkonevinodkone@apache.org

-8Andy KonwinskiUC Berkeleyandrew@apache.org

-8Dave LesterTwitterdlesterdlester@apache.org

-8Benjamin MahlerTwitterbmahlerbmahler@apache.org

-8Thomas MarshallCarnegie Mellon Universitytmarshall@apache.org

-8Brenden MatthewsMesospherebrenden_brenden@apache.org

-8Chris MattmannNASA JPLmattmann@apache.org

-8Brian McCallisterGrouponbrianm@apache.org

-8Niklas Quarfot NielsenMesosphereniq_nnielsen@apache.org

-8Charles ReissUC Berkeleywoggle@apache.org

-5Timothy St ClairRedhattstclairtstclair@apache.org

+2Till ToenshoffMesospheretilltttillt@apache.org

-8Thomas WhiteClouderatomwhite@apache.org

-8Yan XuTwitterxujyanyan@apache.org

-8Jie YuTwitterjieyujieyu@apache.org

-8Matei Alexandru ZahariaDatabricksmatei@apache.org

如果你有兴趣成为一个提交者,最好的方法是通过参与到项目开发人员讨论和贡献补丁。

组件维护人员

我们目前提交者和PMC成员之间没有区别。因此,每一个提交者负责整个代码库的质量。一些组件被广泛维护(如构建和支持工具,测试等),而一些组件比较更加关键，复杂，为了长远角度考虑，需要有提交者为其维护。

我们的目标是为每个组件有一个以上的维护者,为了确保参与者可以获得及时的反馈。为了避免信息孤岛,我们鼓励提交者学习他们不熟悉的代码。

当发送反馈时，这可能对组件维护者有价值,就像如下的指定的。组件维护人员没有任何特殊的“所有权”的代码,而仅仅是作为一种资源及时获取有价值的反馈。我们相信每一个提交者使用良好的判断力来决定何时从组件维护人员获得反馈。

以下三个表格分别对应

通用组件

容器

C++ 库

Component	Maintainers (alphabetical)
Master / Slave	Benjamin Hindman, Vinod Kone, Benjamin Mahler, Jie Yu
Framework API	Benjamin Hindman, Vinod Kone, Benjamin Mahler, Jie Yu

State Libraries	Benjamin Hindman, Benjamin Mahler
Replicated Log	Benjamin Hindman, Jie Yu
ZooKeeper Bindings	Benjamin Hindman, Vinod Kone, Benjamin Mahler, Yan Xu
Authentication / Authorization	Adam B, Vinod Kone, Till Toenshoff
Modules / Hooks	Kapil Arya, Benjamin Hindman, Niklas Nielsen
Oversubscription	Vinod Kone, Niklas Nielsen, Jie Yu
CLI	maintainers needed
WebUI	maintainers needed
Project	Website Dave Lester

Component	Maintainers (alphabetical)
Mesos Containerizer	Ian Downes, Jie Yu
Docker Containerizer	Tim Chen, Benjamin Hindman
External Containerizer	Till Toenshoff, Benjamin Hindman

Component	Maintainers (alphabetical)
Libprocess	Benjamin Hindman, Benjamin Mahler, Jie Yu
Stout	Benjamin Hindman, Vinod Kone, Benjamin Mahler, Jie Yu

Mesos 路线图

Mesos 路线图

Mesos Mesos JIRA Road Map (null) 提供了一个即将发布 的路线视图

‘Epic’ issues in JIRA(null) 提供观察到的更大的项目，已经开始计划工作中。

还有一个特别的发布计划文档（ Release Planning(null) ）,试图捕捉时间和功能，也即将推出。

如果你有意见或建议,随时联系dev列表。(dev@mesos.apache.org)

Doxygen

Mesos 扩展

Mesos 模块

Mesos 模块

Mesos 模块介绍基于Mesos 0.21.0。

免责声明：

- Mesos 的使用和开发中出现的风险，由使用方承担!
- 所有相关问题，请您直接发邮件给相关模块的作者，或邮件给modules@mesos.apache.org。

Mesos模块是什么？

Mesos 模块提供了一种方便扩展其内部运作的方式，即通过创建和使用共享库，使之可以在需要时加载。模块可以用来定制化，无需为每个特定的用例重新编译。模块可以分隔外部依赖使之成为独立的库, 这最终成为了Mesos的一个核心功能。这样模块也可以很容易地测试新的功能。例如,假设可加载的分配器包含VM(Lua,Python,...),可以尝试新的用脚本语言写的分配器算法,而不用被迫使那些依赖关系增加到项目中。总结就是,模块提供了一种简单的方法，可以使Mesos轻松的被第三方模块扩展，而第三方模块却不必知道所有Mesos的内部细节。

调用Mesos的模块

命令行标志--modules 用于Mesos master ,slave 和测试指定的模块列表的加载情况和在内部子系统的可用情况。

使用 --modules=filepath 来指定列表模块 ，所指的文件中应包含被JSON格式化的字符串， 。 “filepath” 的格式应该是 ‘file:///path/to/file’ 或 ‘/path/to/file’ .的形式。

使用--modules="{...}" 在命令行指定模块列表。

JSON字符串示例：

*. 加载库libfoo.so ，其中包含两个模块org*apache*mesos_bar`` 和```org*apache*mesos_baz```。

```
{
 "libraries": [
  {
   "file": "/path/to/libfoo.so",
   "modules": [
    {
     "name": "org_apache_mesos_bar",
```

```
      },
      {
        "name": "org_apache_mesos_baz"
      }
    ]
   }
  ]
}
```

*. 从foo加载模块org*apache*mesos_bar``和传递命令行参数X和Y值(模块加载```org*apache*mesos_baz```没有任何命令行参数):

```
{
  "libraries": [
   {
     "name": "foo",
     "modules": [
      {
        "name": "org_apache_mesos_bar"
        "parameters": [
         {
           "key": "X",
           "value": "Y",
         }
        ]
      },
      {
        "name": "org_apache_mesos_baz"
      }
    ]
   }
  ]
}
```

*. 在命令行中指定

```
 --modules='{"libraries":[{"file":"/path/to/libfoo.so",
"modules":[{"name":"org_apache_mesos_bar"}]}]}'
```

库名

对于每一个library,至少有一个的“file”或“name”参数必须被指定。“file”参数可能指一个文件名(例如“libfoo.so”),相对路径(如“myLibs / libfoo.so”)或绝对路径(例如

“/home/mesos/lib/libfoo.so”）。“name”参数是指库名称(如“foo”）。如果“name”被指定了,它会在当前平台上自动改为一个合适的库名称(如在Linux上：“foo”自动改为“libfoo.so”。在OS X 改为“libfoo.dylib”）。

如果库路径没有在“file”中指定参数,库就会搜索标准库的路径或目录：LD*LIBRARY*PATH(在OS X上是：DYLD*LIBRARY*PATH)。

如果“file”和“name”两个都指定了，“name”将被忽略。

Mesos支持哪些模块?

当前可用的各种模块类型

分配器

Mesos master’s 分配器定期确定哪些框架(s)应该提供集群的可用资源。分配器模块使试验专用的资源分配算法。这些可能是一个分配程序的一个例子,它提供了一个主导资源公平分配程序。

在Mesos master加载自定义分配器,你需要:

- 介绍其给Mesos master，通过使用 --modules配置,
- 选择它作为分配器通过 --allocator。

例如,下面的命令将运行Mesos master，其内包含ExternalAllocatorModule(见本节为JSON格式):

```
./bin/mesos-master.sh --work_dir=m/work --modules="file://<modules-including-allocator>.json" --allocator=ExternalAllocatorModule
```

匿名模块

匿名模块不会收到任何回调, 它与他们的父进程共存。

与其他命名模块不同的是,一个匿名模块并不直接提供基本便功能(如一个隔离器模块)。和装饰模块也不同，它也不直接提供添加或注入数据。

匿名模块不需要任何特定的选择器(标志),Mesos master 或 slave 通过--modules， 他们会立即被实例化

验证模块

验证模块允许第三方快速开发和插件的新身份验证方法。这样的模块可以支持PAM(LDAP、MySQL NIS,UNIX)对身份的验证。

类似于Apache web服务器模块,hooks 允许模块作者绑定模块到内部组件，模块可能不完全适合内部组件,但是可以被定义操作，这就是所谓的hooks。

可用的hook API中在Mesos中定义为hook.hpp 。每个hook 定义了插入点和可用的上下文。一个示例就是任务信息传递给master 的LaunchTaskHook。

一些hooks 在一个对象(例如 任务信息)并返回全部或部分的对象(如 任务标签),以可以修改或替换动态内容。这些hooks 被称为。

为了支持装饰模块删除元数据(环境变量或标签),其返回值的定义Mesos做了一些改变，请参见Mesos 0.23.0（如下）。

状态(0.22.x)版本之前 (0.23.0+)版本后

ErrorError is propagated to the call-siteNo change

NoneThe result of the decorator is not appliedNo change

SomeThe result of the decorator is appendedThe result of the decorator overwrites the final labels/environment object

在Mesos中加载一个hook，你需要：

• 把hook通过 --modules 的方式，介绍给Mesos.

• 通过--hooks标识选择它

例如,下面的命令将运行有TestTaskHook hook的Mesos slave :

```
./bin/mesos-slave.sh --master=<IP>:<PORT> --modules="file://<path-to-modules-config>.json" --hooks=TestTaskHook
```

隔离器

隔离器模块支持试验专门隔离和监视功能。这些例子可以是第三方的资源隔离机制的GPGPU硬件,网络,等等。

编写Mesos 模块

一个HelloWorld模块

以下代码片段描述了一个模块的实现名为“org*apache*mesos_bar”的“TestModule”类:

```
#include <iostream>
#include "test_module.hpp"

class TestModuleImpl : public TestModule
{
public:
  TestModuleImpl()
  {
    std::cout << "HelloWorld!" << std::endl;
  }

  virtual int foo(char a, long b)
  {
    return a + b;
```

```
  }

  virtual int bar(float a, double b)
  {
    return a * b;
  }
};

static TestModule* create()
{
    return new TestModule();
}

static bool compatible()
{
  return true;
}


// Declares a module named 'org_apache_mesos_TestModule' of
// 'TestModule' kind.
// Mesos core binds the module instance pointer as needed.
// The compatible() hook is provided by the module for compatibility checks.
// The create() hook returns an object of type 'TestModule'.
mesos::modules::Module<TestModule> org_apache_mesos_TestModule(
    MESOS_MODULE_API_VERSION,
    MESOS_VERSION,
    "Apache Mesos",
    "modules@mesos.apache.org",
    "This is a test module.",
    compatible,
    create);
```

build 模块

以下假设Mesos 安装在标准的位置,即Mesos 动态库和头文件是可用的。

g++ -lmesos -fpic -o test*module.o test*module.cpp

$ gcc -shared -o libtest*module.so test*module.o

测试模块

除了显式使用–modules标识测试模块 ,通常运行整个mesos 测试套件 加上给定的模块。例如,下面

的命令将运行mesos 测试套件 ，其中给定org*apache*mesos_TestCpuIsolator 隔离 模块:

./bin/mesos-tests.sh --modules="/home/kapil/mesos/isolator-module/modules.json" --isolation="org*apache*mesos_TestCpuIsolator"

模块的命名约定

每个模块的名字应该是唯一的。在Json字符串有重复的模块名称会导致进程异常中止。

因此,我们鼓励模块作者定义名字模块根据Java包命名方(http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html)。

例如:

```
Module NameModule Domain nameModule Symbol Namefoobarmesos.apache.orgorg_apache_mesos_foobarbarBazexample.comcom_example_barBaz
```

简而言之:保持模块名称。小写和反向域。——单独的强调。不直接使用类名作为模块的名字。——不同的模块从相同的组织仍然需要不同的名称。

模块版本控制和向后兼容性

在装载以上模块,动态库包含模块需要加载到Mesos 。这在Mesos 启动时加载。当引入新模块类型时，Mesos 开发人员不需要碰代码。但是,开发人员负责登记新模块，包括版本信息，以保证任何给定的模块将保持兼容。这些信息保存在一个表src/module/manager.cpp. 。一个条目为一个模块,每个都有一个相应的Mesos 发布版本号。这个数字需要开发人员维护,以和当前的Mesos 版本以及模块之间的兼容性,。在旧Mesos 版本中的模块不能直接写到后来的版本。

为了成功加载模块,不同版本之间必须存在以下关系:

```
kind version <= Library version <= Mesos version
```

Mesoskind versionLibraryIs module loadableReason

0.18.00.18.00.18.0yes

0.29.00.18.00.18.0yes

0.29.00.18.00.21.0yes

0.18.00.18.00.29.0NOLibrary compiled against a newer Mesos release.

0.29.00.21.00.18.0NOModule/Library older than the kind version supported by Mesos.

0.29.00.29.00.18.0NOModule/Library older than the kind version supported by Mesos.

Mesos 模块API的变化

不兼容的更改记录模块API。

版本2

• Added support for module-specific command-line parameters.

添加特定于模块支持命令行参数

• Changed function signature for create().

改变了创建()函数签名。

版本1

模块的初始版本的API。

附录：

```
JSON Schema:
{
  "type":"object",
  "required":false,
  "properties":{
    "libraries":{
      "type":"array",
      "required":false,
      "items":{
        "type":"object",
        "required":false,
        "properties":{
          "file":{
            "type":"string",
            "required":false
          },
          "name":{
            "type":"string",
            "required":false
          },
          "modules":{
            "type":"array",
            "required":false,
            "items":{
              "type":"object",
              "required":false,
              "properties":{
                "name":{
                  "type":"string",
                  "required":true
                },
```

```
            "parameters":{
              "type":"array",
              "required":false,
              "items":{
                "type":"object",
                "required":false,
                "properties":{
                  "key":{
                    "type":"string",
                    "required":true
                  },
                  "value":{
                    "type":"string",
                    "required":true
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Mesos 配置模块

Mesos配置模块

Mesos master 的用于确定哪些框架资源的逻辑，是封装 在分配器模块里的。分配器是一个可插入组件,Mesos 可以使用它来实现自己的共享策略,例如fair-sharing、优先级等,或调整默认等级优势资源公平算法((见DRF 文献(null))。

使用定制的分配器,便一个必须:

- 实现分配器接口 mesos/master/allocator.hpp
- 包装分配器实现到模块然后在Mesos master中加载

编写一个定制的分配器

分配器模块是在c++中实现的,Mesos 也是c++写的。他们的子类中定义的分配器接口在

mesos/master/allocator.hpp 。但是,你的实现可以是一个c++代理,这表明调用实际的分配器的语言是可以选择的。

默认的分配器是

```$MESOS_HOME/src/master/allocator/mesos/allocator.hpp```中定义子类。然后在```MesosAllocator```封装基于actor 的```MesosAllocator```分配器。这将调用底层的执行者和控制它的生命周期。您可以参考```HierarchicalDRFAllocatorProcess``` .

此外,内置的分层分配器可以扩展,而不需要重新实现整个分配的逻辑。这可以通过使用```theSorter```抽象类。```Sorters```定义层次结构层的顺序(如角色或框架)，通过调用“客户端”对象和一些客户信息，应该提供资源，返回客户的有序列表。

分类器是在c++和继承中实现```Sorter``` 类的，这定义在```$MESOS_HOME/src/master/allocator/sorter/sorter.hpp``` 。默认分类器是```DRFSorter``` ,实现了公平分享，相关位置在```$MESOS_HOME/src/master/allocator/sorter/drf/sorter.hpp.``` 。这个分类器能够根据权重，表决出优先顺序,需要指定```-add()```。每个客户端的资源是通过它的权重而来的。例如,一个权重为2的客户端将比权重为1的客户端得到两倍的资源。

###连接定制的分配器

编写一个定制的分配器,下一步是要重写内置的实现。这个过程包括几个步骤:

* 封装你的分配器便分配器模块,

* 在Mesos maste 加载这个模块

一个分配器模块是一个工厂函数和模块描述、在定义```mesos/module/allocator.hpp```. 定义。假设分配逻辑由```theExternalAllocator```实现类中声明```external_allocator```。以下代码片段描述了一个实现名为```ExternalAllocatorModule```的分配器模块：

```
include <mesos/master/allocator.hpp>

include <mesos/module/allocator.hpp>
```
```

```
# include <stout/try.hpp>
# include "external_allocator.hpp"
using namespace mesos;
using mesos::master::allocator::Allocator;
using mesos::internal::master::allocator::HierarchicalDRFAllocator;
static Allocator* createExternalAllocator(const Parameters& parameters)
{
Try<Allocator*> allocator = ExternalAllocator::create();
if (allocator.isError()) {
  return NULL;
}
return allocator.get();
}
// Declares an ExternalAllocator module named 'ExternalAllocatorModule'.
mesos::modules::Module<Allocator> ExternalAllocatorModule(
  MESOS_MODULE_API_VERSION,
  MESOS_VERSION,
  "Mesos Contributor",
  "engineer@example.com",
  "External Allocator module.",
  NULL,
  createExternalAllocator);
```

`

请参阅便模块文档(null)说明如何编译和加载模块便主人。

更多 Mesos 信息

Powered by Mesos

原文链接