Docker Swarm

Docker Swarm 简介

Docker Swarm 包含两方面：一个企业级的 Docker 安全集群，以及一个微服务应用编排引擎。

集群方面，Swarm 将一个或多个 Docker 节点组织起来，使得用户能够以集群方式管理它们。Swarm 默认内置有加密的分布式集群存储（encrypted distributed cluster store）、加密网络（Encrypted Network）、公用 TLS（Mutual TLS）、安全集群接入令牌 Secure Cluster Join Token）以及一套简化数字证书管理的 PKI（Public Key Infrastructure）。用户可以自如地添加或删除节点，这非常棒！

编排方面，Swarm 提供了一套丰富的 API 使得部署和管理复杂的微服务应用变得易如反掌。通过将应用定义在声明式配置文件中，就可以使用原生的 Docker 命令完成部署。此外，甚至还可以执行滚动升级、回滚以及扩缩容操作，同样基于简单的命令即可完成。

以往，Docker Swarm 是一个基于 Docker 引擎之上的独立产品。自 Docker 1.12 版本之后，它已经完全集成在 Docker 引擎中，执行一条命令即可启用。到 2018 年，除了原生 Swarm 应用，它还可以部署和管理 Kubernetes 应用。

从集群角度来说，一个 Swarm 由一个或多个 Docker 节点组成。这些节点可以是物理服务器、虚拟机、树莓派 Raspberry Pi 或云实例。唯一的前提就是要求所有节点通过可靠的网络相连。

节点会被配置为管理节点（Manager）或工作节点（Worker）。管理节点负责集群控制面（Control Plane），进行诸如监控集群状态、分发任务至工作节点等操作。工作节点接收来自管理节点的任务并执行。

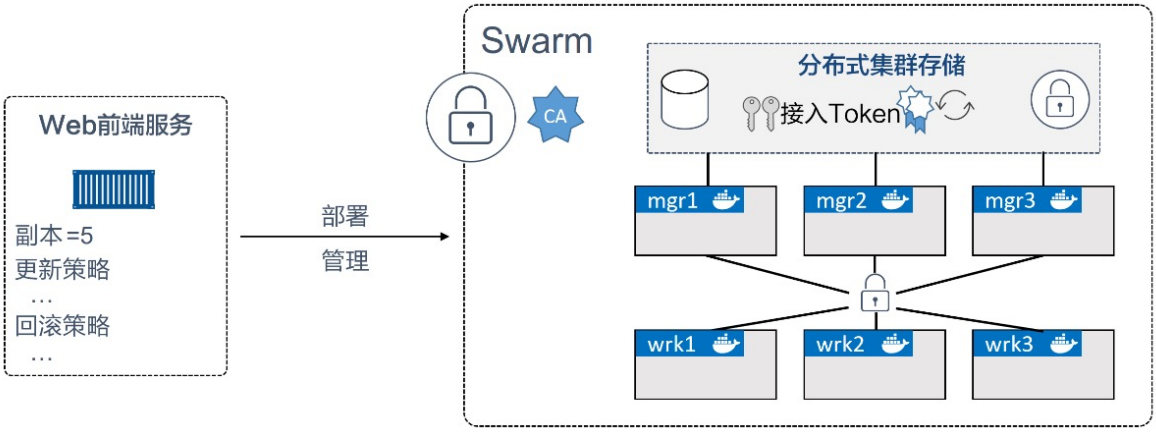
Swarm 的配置和状态信息保存在一套位于所有管理节点上的分布式 etcd 数据库中。该数据库运行于内存中，并保持数据的最新状态。关于该数据库最棒的是，它几乎不需要任何配置——作为 Swarm 的一部分被安装，无须管理。

关于集群管理，最大的挑战在于保证其安全性。搭建 Swarm 集群时将不可避免地使用 TLS，因为它被 Swarm 紧密集成。在安全意识日盛的今天，这样的工具值得大力推广。Swarm 使用 TLS 进行通信加密、节点认证和角色授权。自动密钥轮换（Automatic Key Rotation）更是锦上添花！其在后台默默进行，用户甚至感知不到这一功能的存在！

关于应用编排，Swarm 中的最小调度单元是服务。它是随 Swarm 引入的，在 API 中是一个新的对象元素，它基于容器封装了一些高级特性，是一个更高层次的概念。

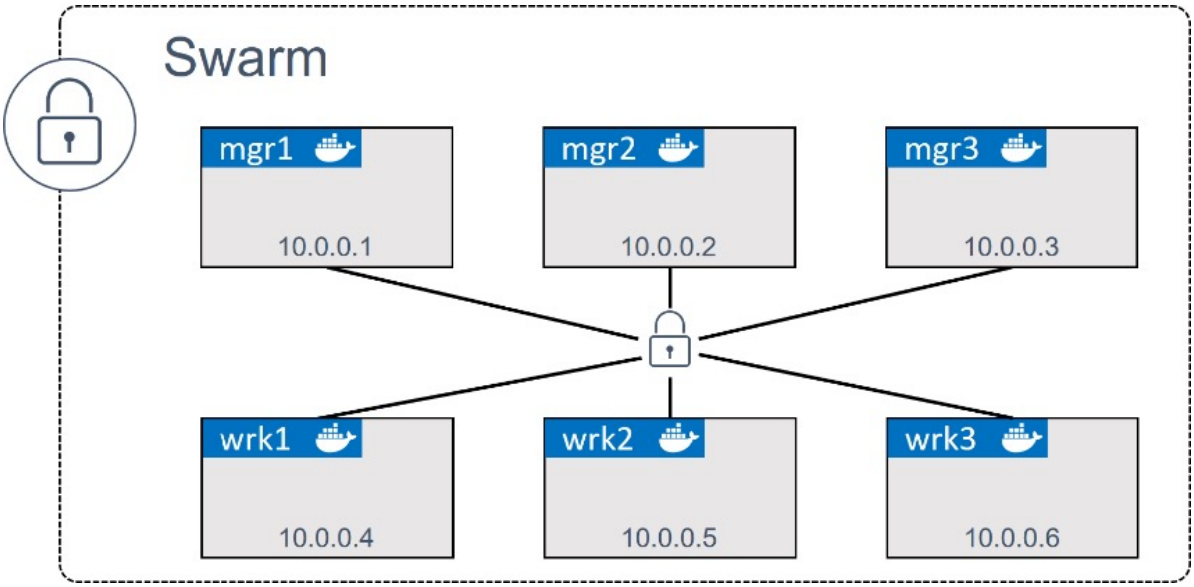
当容器被封装在一个服务中时，我们称之为一个任务或一个副本，服务中增加了诸如扩缩容、滚动升级以及简单回滚等特性。

综上，从概括性的视角来看 Swarm，如下图所示：



搭建安全集群

本节会搭建一套安全 Swarm 集群，其中包含 3 个管理节点和 3 个工作节点。搭建也可以自行调整管理节点和工作节点的数量、名称和 IP，下图的名称叫做 “Swarm 集群”：



每个节点都需要安装 Docker，并且能够与 Swarm 的其它节点通信。如果配置有域名解析就更好了——这样在命令的输出中更容易识别出节点，也更有利于排除故障。

在网络方面，需要在路由器和防火墙中开放如下端口：

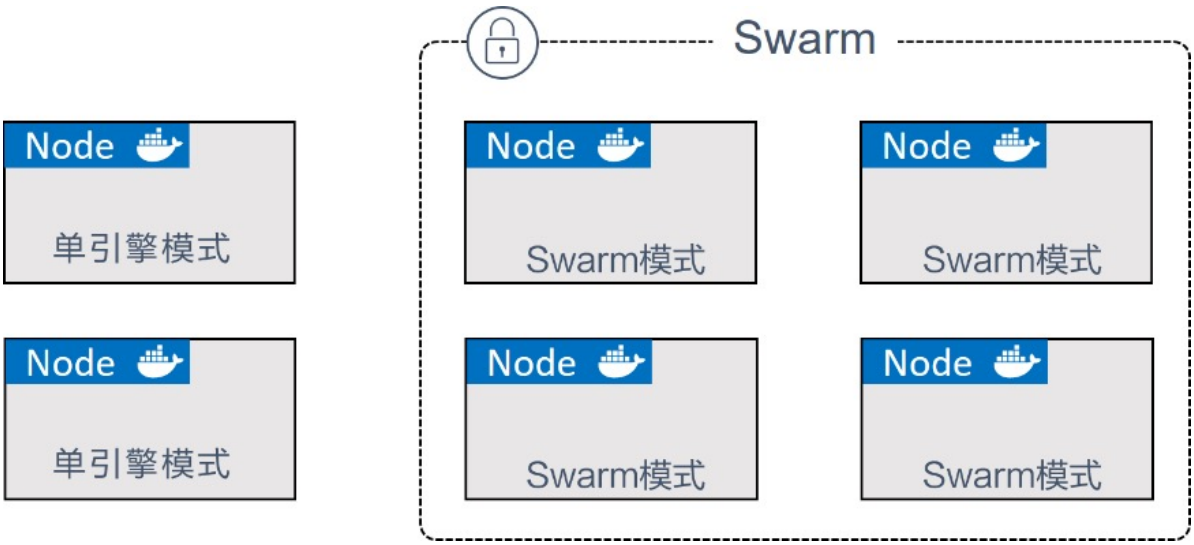
- `2377/tcp` 用于客户端与 Swarm 进行安全通信。
- `7946/tcp` 与 `7946/udp` 用于控制面 gossip 分发。
- `4789/udp` 用于基于 VXLAN 的覆盖网络。

如果满足以上前提，就可以着手开始搭建 Swarm 集群了。

搭建 Swarm 的过程有时也被称为初始化 Swarm，大体流程包括：初始化第一个管理节点 > 加入额外的管理节点 > 加入工作节点 > 完成。

初始化一个全新的 Swarm

不包含在任何 Swarm 中的 Docker 节点，称为运行于单引擎（Single-Engine）模式。一旦被加入 Swarm 集群，则切换为 Swarm 模式，如下图所示：



在单引擎模式下的 Docker 主机上运行 `docker swarm init` 会将其切换到 Swarm 模式，并创建一个新的 Swarm，将自身设置为 Swarm 的第一个管理节点。

更多的节点可以作为管理节点或工作节点加入进来。这一操作也会将新加入的节点切换为 Swarm 模式。

以下的步骤会将 **mamager1** 切换为 Swarm 模式，并初始化一个新的 Swarm。接下来将 **wrk1**、**wrk2** 和 **wrk3** 作为工作节点接入——自动将它们切换为 Swarm 模式。然后将 **mamager2** 和 **mamager3** 作为额外的管理节点接入，并同样切换为 Swarm 模式。最终有 6 个节点切换到 Swarm 模式，并运行于同一个 Swarm 中。

本示例会使用前文 “Swarm 集群” 那张图所示的各节点的 IP 地址和 DNS 名称。IP 地址有所不同，在实验环境中执行 ifconfig 命令可以查看 IP 地址：

```
shiyanlou:~/ $ ifconfig
docker0   Link encap:以太网  硬件地址 02:42:0c:93:ca:e0
          inet 地址:192.168.0.1  广播:0.0.0.0  掩码:255.255.240.0
          UP BROADCAST MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)

docker_gwbridge Link encap:以太网  硬件地址 02:42:b9:ce:48:46
          inet 地址:172.17.0.1  广播:0.0.0.0  掩码:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:0
          接收字节:0 (0.0 B)  发送字节:0 (0.0 B)

eth0      Link encap:以太网  硬件地址 00:16:3e:0a:4a:a0
          inet 地址:10.111.63.86  广播:10.111.63.255  掩码:255.255.240.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:34910 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:10548 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:45224923 (45.2 MB)  发送字节:3658102 (3.6 MB)
```

（1）登录到 **mamager1** 并初始化一个新的 Swarm ， IP 地址如上图所示：

```
$ docker swarm init --advertise-addr 10.111.63.86 --listen-addr 10.111.63.86
```

将这条命令拆开分析如下：

- `docker swarm init` 会通知 Docker 来初始化一个新的 Swarm，并将自身设置为第一个管理节点。同时也会使该节点开启 Swarm 模式。
- `--advertise-addr` 指定其它节点用来连接到当前管理节点的 IP 和端口。这一属性是可选的，当节点上有多个 IP 时，可以用于指定使用哪个 IP。此外，还可以用于指定一个节点上没有的 IP，比如一个负载均衡的 IP。其中端口可以省略不写，默认端口是 2377 。
- `--listen-addr` 指定用于承载 Swarm 流量的 IP 和端口。其设置通常与 `--advertise-addr` 相匹配，但是当节点上有多个 IP 的时候，可用于指定具体某个 IP。并且，如果 `--advertise-addr` 设置了一个远程 IP 地址（如负载均衡的 IP 地址），该属性也是需要设置的。建议执行命令时总是使用这两个属性来指定具体 IP 和端口。

Swarm 模式下的操作默认运行于 2337 端口。虽然它是可配置的，但 `2377/tcp` 是用于客户端与 Swarm 进行安全（HTTPS）通信的约定俗成的端口配置。

（2）列出 Swarm 中的节点：

```
shiyanlou:~/ $ docker node ls                                                  [21:58:40]
ID                            HOSTNAME                STATUS              AVAILABILITY
     MANAGER STATUS
y3cbu6qzn7v7dq0p0ji4a2dke *   iZbp1j3cfhjoio4462fntpZ   Ready               Active
     Leader
shiyanlou:~/ $                                                                 [21:58:51]
```

注意到 **mamager1** 是 Swarm 中唯一的节点，并且作为 Leader 列出，稍后再探讨这一点。

由于实验环境只能提供一个云主机作为 Docker 节点，本节实验接下来的操作需要大家购买并创建 6 个云主机并安装 docker 运行。

以下操作的命令和截图均为我的云主机上的执行结果。

我的 manager1 主机的内网地址是 172.16.109.89 ，执行如下命令初始化 Swarm：

```
root@manager1:~# docker swarm init --advertise-addr 172.16.109.89 --listen-addr 172.16.109.89
Swarm initialized: current node (xtm876oxndnq12ix615jnvuzl) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-1mlvnrs2rr6v9joh91jp7oy39qcpk4ac6ilqdmiczindqurdi5-2m09dmfn395wl4wynilx7r2tp 172.16.109.89:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

（3）在 mamager1 上执行 docker swarm join-token 命令来获取添加新的工作节点和管理节点到 Swarm 的命令和 Token：

```
root@manager1:~# docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
1mlvnrs2rr6v9joh91jp7oy39qcpk4ac6ilqdmiczindqurdi5-0yo0oj059yiv5isy8w86e7n0z
172.16.109.89:2377

root@manager1:~# docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
1mlvnrs2rr6v9joh91jp7oy39qcpk4ac6ilqdmiczindqurdi5-2m09dmfn395wl4wynilx7r2tp
172.16.109.89:2377
```

请注意，工作节点和管理节点的接入命令中使用的接入 Token 是不同的。因此，一个节点是作为工作节点还是管理节点接入，完全依赖于使用了哪个 Token。接入 Token 应该被妥善保管，因为这是将一个节点加入 Swarm 的唯一所需！

（4）登录到 worker1，并使用包含工作节点接入 Token 的 `docker swarm join` 命令将其接入 Swarm：

```
root@worker1:~# docker swarm join --token SWMTKN-1-
1mlvnrs2rr6v9joh91jp7oy39qcpk4ac6ilqdmiczindqurdi5-2mO9dmfn395wl4wynilx7r2tp
172.16.109.89:2377
This node joined a swarm as a worker.
```

`--advertise-addr` 与 `--listen-addr` 属性是可选的。在网络配置方面，请尽量明确指定相关参数，这是一种好的实践。

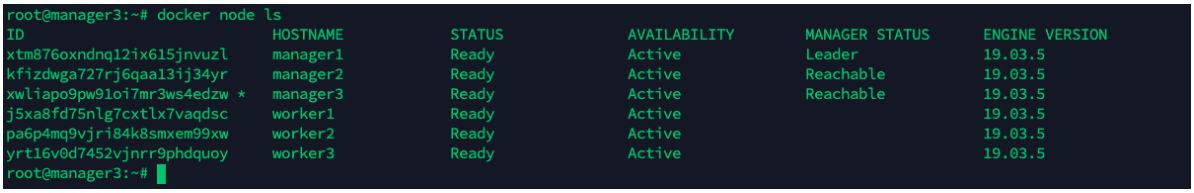
（5）在 **worker2** 和 **worker3** 上重复上一步骤来将这两个节点作为工作节点加入 Swarm。确保使用 `--advertise-addr` 与 `--listen-addr` 属性来指定各自的 IP 地址。

（6）登录到 **manager2**，然后使用含有管理节点接入 Token 的 `docker swarm join` 命令，将该节点作为工作节点接入 Swarm：

```
root@worker2:~# docker swarm join --token SWMTKN-1-
1mlvnrs2rr6v9joh91jp7oy39qcpk4ac6ilqdmiczindqurdi5-2mO9dmfn395wl4wynilx7r2tp
172.16.109.89:2377
This node joined a swarm as a worker.
```

（7）在 **manager3** 上重复以上步骤，记得在 `--advertise-addr` 与 `--listen-addr` 属性中指定 **manager3** 的 IP 地址。

（8）在任意一个管理节点上执行 `docker node ls` 命令来列出 Swarm 节点，如下所示为 manager3 中的截图：



很好，我已经创建了 6 个节点的 Swarm，其中包含 3 个管理节点和 3 个工作节点。在这个过程中，每个节点的 Docker 引擎都被切换到 **Swarm 模式** 下。贴心的是，**Swarm** 已经自动启用了 TLS 以策安全。

观察 `MANAGER STATUS` 一列会发现，3 个节点分别显示为 `Reachable` 或 `Leader`。关于主节点稍后很快会介绍到。`MANAGER STATUS` 一列无任何显示的节点是工作节点。注意，**mamager3** 的 ID 列还显示了一个星号（*），这个星号会告知用户执行 `docker node ls` 命令所在的节点。本例中，命令是在 **mamager3** 节点执行的。

注：

> 每次将节点加入 Swarm 都指定 `--advertise-addr` 与 `--listen-addr` 属性是痛苦的。然而，一旦 Swarm 中的网络配置出现问题将会更加痛苦。况且，手动将节点加入 Swarm 也不是一种日常操作，所以在执行该命令时额外指定这两个属性是值得的。不过选择权在读者手中。对于实验环境，或节点中只有一个 IP 的情况来说，也许并不需要指定它们。

现在已经有一个运行中的 Swarm 了，下面看一下如何进行高可用（HA）管理。

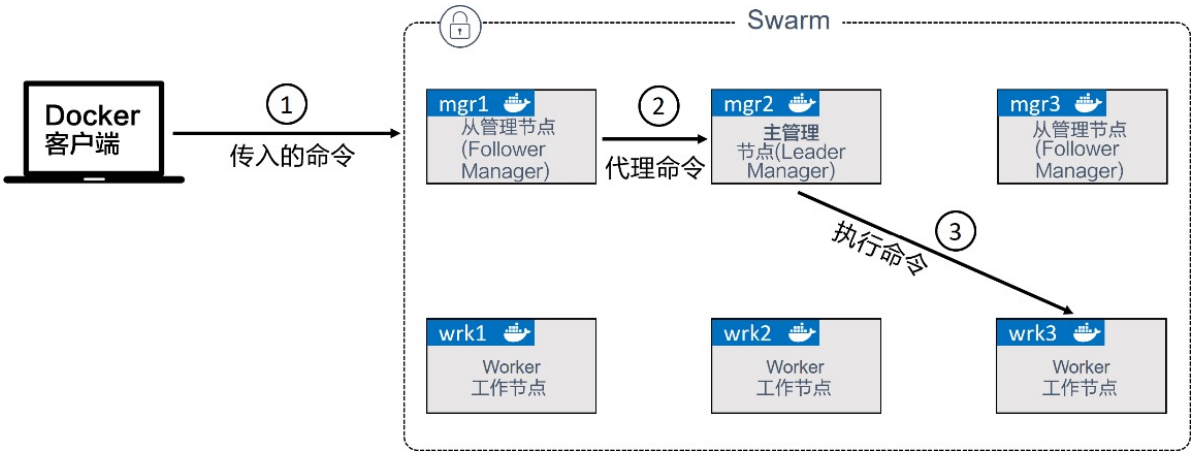
Swarm 管理器高可用性（HA）

至此在 Swarm 中已经加入了 3 个管理节点。为什么添加 3 个，以及它们如何协同工作？ 本节将就此以及更多问题展开介绍。

Swarm 的管理节点内置有对 HA 的支持。这意味着，即使一个或多个节点发生故障，剩余管理节点也会继续保证 Swarm 的运转。

从技术上来说，Swarm 实现了一种主从方式的多管理节点的 HA。这意味着，即使你可能并且应该有多个管理节点，也总是仅有一个节点处于活动状态。通常处于活动状态的管理节点被称为“主节点”（leader），而主节点也是唯一一个会对 Swarm 发送控制命令的节点。也就是说，只有主节点才会变更配置，或发送任务到工作节点。如果一个备用（非活动）管理节点接收到了 Swarm 命令，则它会将其转发给主节点。

这一过程如下图所示。步骤 ① 指命令从一个远程的 Docker 客户端发送给一个管理节点；步骤 ② 指非主节点将命令转发给主节点；步骤 ③ 指主节点对 Swarm 执行命令：



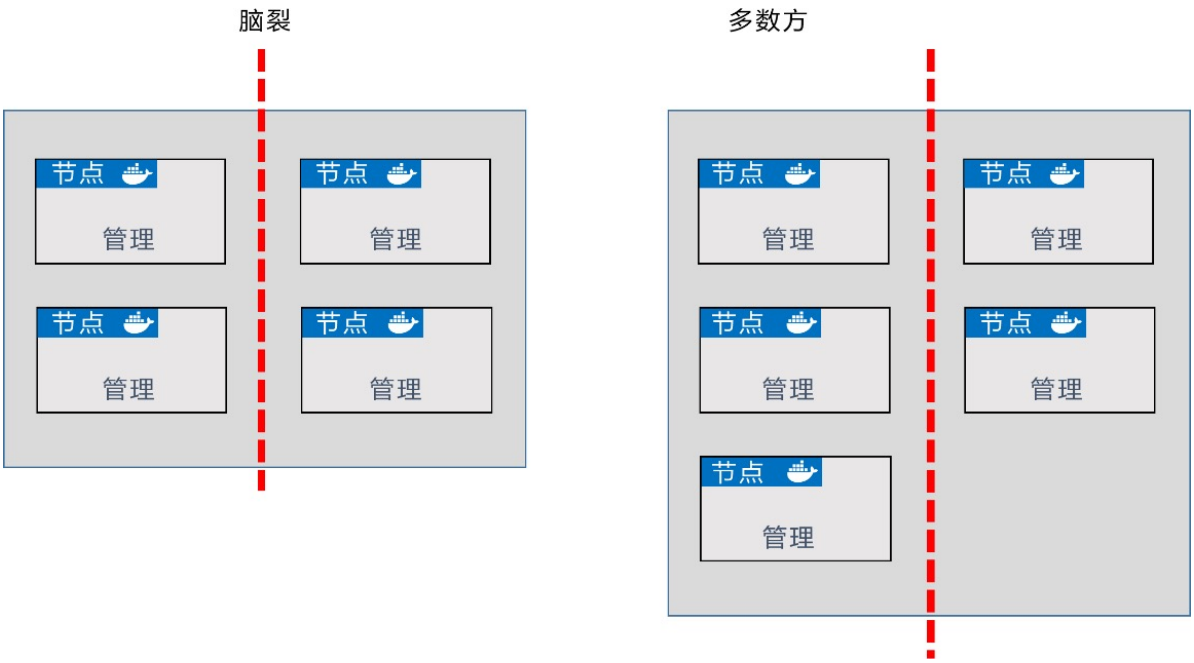
仔细观察上图会发现，管理节点或者是 Leader 或者是 Follower。这是 Raft 的术语，因为 Swarm 使用了 Raft 共识算法的一种具体实现来支持管理节点的 HA。关于 HA，以下是两条最佳实践原则。

- 部署奇数个管理节点。
- 不要部署太多管理节点（建议 3 个或 5 个）。

部署奇数个管理节点有利于减少脑裂情况的出现机会。假如有 4 个管理节点，当网络发生分区时，可能会在每个分区有两个管理节点。这种情况被称为脑裂——每个分区都知道曾经有 4 个节点，但是当前网络中仅有两个节点。糟糕的是，每个分区都无法知道其余两个节点是否运行，也无从得知本分区是否掌握大多数（Quorum）。虽然在脑裂情况下集群依然在运行，但是已经无法变更配置，或增加和管理应用负载了。

不过，如果部署有 3 个或 5 个管理节点，并且也发生了网络分区，就不会出现每个分区拥有同样数量的管理节点的情况。这意味着掌握多数管理节点的分区能够继续对集群进行管理。图 10.5 中右侧的例子，阐释了这种情况，左侧的分区知道自己掌握了多数的管理节点。

对于所有的共识算法来说，更多的参与节点就意味着需要花费更多的时间来达成共识。这就像决定去哪吃饭——只有 3 个人的时候总是比有 33 个人的时候能更快确定。考虑到这一点，最佳的实践原则是部署 3 个或 5 个节点用于 HA。7 个节点可以工作，但是通常认为 3 个或 5 个是更优的选择。当然绝对不要多于 7 个，因为需要花费更长的时间来达成共识。



关于管理节点的 HA 再补充一点。显然将管理节点分布到不同的可用域（Availability Zone）中是一种不错的实践方式，但是一定要确保它们之间的网络连接是可靠的，否则由于底层网络分区导致的问题将是令人痛苦的！这意味着，目前将生产环境的应用和基础设置部署在多个不同的公有云（例如 AWS 和 Azure）上的想法仍然是天方夜谭。请一定要确保管理节点之间是有高速可靠的网络连接的！

内置的 Swarm 安全机制

Swarm 集群内置有繁多的安全机制，并提供了开箱即用的合理的默认配置——如 CA 设置、接入 Token、公用 TLS、加密集群存储、加密网络、加密节点 ID 等。更多细节我们会在后面的实验中阐述。

锁定 Swarm

尽管内置有如此多的原生安全机制，重启一个旧的管理节点或进行备份恢复仍有可能对集群造成影响。一个旧的管理节点重新接入 Swarm 会自动解密并获得 Raft 数据库中长时间序列的访问权，这会带来安全隐患。进行备份恢复可能会抹掉最新的 Swarm 配置。

为了规避以上问题，Docker 提供了自动锁机制来锁定 Swarm，这会强制要求重启的管理节点在提供一个集群解锁码之后才有权从新接入集群。

通过在执行 `docker swarm init` 命令来创建一个新的 Swarm 集群时传入 `--autolock` 参数可以直接启用锁。然而，前面已经搭建了一个 Swarm 集群，这时也可以使用 `docker swarm update` 命令来启用锁。

在某个 Swarm 管理节点上运行 `docker swarm update --autolock=true` 命令：

```
root@manager3:~# docker swarm update --autolock=true
Swarm updated.
To unlock a swarm manager after it restarts, run the `docker swarm unlock`
command and provide the following key:

    SWMKEY-1-SgWJAI+xH5RR54ld7860+O8A1SXJjO99ECyuTEg3xhY

Please remember to store this key in a password manager, since without it you
will not be able to restart the manager.
root@manager3:~#
```

请确保将解锁码妥善保管在安全的地方！

重启某一个管理节点，以便观察其是否能够自动重新接入集群。读者可以在以下命令前添加 sudo 执行：

```
$ service docker restart
```

我以 manager2 为例尝试列出 Swarm 中的节点：

```
root@manager2:~# service docker restart
root@manager2:~# docker node ls
Error response from daemon: Swarm is encrypted and needs to be unlocked before it
can be used. Please use "docker swarm unlock" to unlock it.
root@manager2:~#
```

尽管 Docker 服务已经重启，该管理节点仍然未被允许重新接入集群。为了进一步验证，读者可以到其他管理节点执行 `docker node ls` 命令，会发现重启的管理节点的 STATUS 会显示 `Down`，MANAGER STATUS 会显示 `Unreachable`。

在 manager1 中查看截图如下：

```
root@manager1:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
xtm876oxndnq12ix615jnvuzl *   manager1            Ready               Active              Leader              19.03.5
kfizdwga727rj6qaa13ij34yr     manager2            Down                Active              Unreachable         19.03.5
xwliapo9pw91oi7mr3ws4edzw     manager3            Ready               Active              Reachable           19.03.5
j5xa8fd75nlg7cxtlx7vaqdsc     worker1             Ready               Active                                  19.03.5
pa6p4mq9vjri84k8smxem99xw     worker2             Ready               Active                                  19.03.5
yrt16v0d7452vjnrr9phdquoy     worker3             Ready               Active                                  19.03.5
root@manager1:~#
```

执行 `docker swarm unlock` 命令来为重启的管理节点解锁 Swarm。该命令需要在重启的节点上执行，同时需要提供解锁码：

```
$ docker swarm unlock
Please enter unlock key: <enter your key>
```

该节点将被允许重新接入 Swarm，并且再次执行 `docker node ls` 命令会显示 `ready` 和 `reachable`：

```
root@manager2:~# docker swarm unlock
Please enter unlock key:
root@manager2:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
xtm876oxndnq12ix615jnvuzl     manager1            Ready               Active              Leader              19.03.5
kfizdwga727rj6qaa13ij34yr *   manager2            Ready               Active              Reachable           19.03.5
xwliapo9pw91oi7mr3ws4edzw     manager3            Ready               Active              Reachable           19.03.5
j5xa8fd75nlg7cxtlx7vaqdsc     worker1             Ready               Active                                  19.03.5
pa6p4mq9vjri84k8smxem99xw     worker2             Ready               Active                                  19.03.5
yrt16v0d7452vjnrr9phdquoy     worker3             Ready               Active                                  19.03.5
root@manager2:~#
```

至此，Swarm 集群已经搭建起来，并且对主节点和管理节点 HA 有了一定了解，下面开始介绍服务。

Swarm 服务

本节介绍的内容可以使用 Docker Stack 进一步改进。然而，本章的概念对于准备 Docker Stack 的学习是非常重要的。

就像在 Swarm 初步介绍中提到的，服务是自 Docker 1.12 后新引入的概念，并且仅适用于 Swarm 模式。

使用服务仍能够配置大多数熟悉的容器属性，比如容器名、端口映射、接入网络和镜像。此外还增加了额外的特性，比如可以声明应用服务的期望状态，将其告知 Docker 后，Docker 会负责进行服务的部署和管理。举例说明，假如某应用有一个 Web 前端服务，该服务有相应的镜像。测试表明对于正常的流量来说 5 个实例可以应对。那么就可以将这一需求转换为一个服务，该服务声明了容器使用的镜像，并且服务应该总是有 5 个运行中的副本。

我们稍后再介绍声明服务过程中的其他参数，在此之前，首先看如何创建刚刚描述的内容。

使用 `docker service create` 命令创建一个新的服务。

注：

> 在 Windows 上创建新服务的命令也是一样的。然而本例中使用的是 Linux 镜像，它在 Windows 上并不能运行。请使用 Windows 的读者将镜像替换为一个 Windows Web Server 的镜像，以便能正常运行。再次强调，在 PowerShell 终端中输入命令的时候，使用反引号（`）进行换行。

```
$ docker service create --name web-fe \
   -p 8080:8080 \
   --replicas 5 \
   nigelpoulton/pluralsight-docker-ci
```

```
root@manager1:~# docker service create --name web-fe -p 8080:8080 --replicas 5 nigelpoulton/pluralsight-docker-ci
7wa2zz179ef05o1u7h7ows3at
overall progress: 5 out of 5 tasks
1/5: running   [==================================================>]
2/5: running   [==================================================>]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: running   [==================================================>]
verify: Service converged
root@manager1:~#
```

请注意，该命令与熟悉的 `docker container run` 命令的许多参数是相同的。这个例子中，使用 `--name` 和 `-p` 定义服务的方式，与单机启动容器的定义方式是一样的。

回顾一下命令和输出。使用 `docker service creale` 命令告知 Docker 正在声明一个新服务，并传递 `--name` 参数将其命名为 `web-fe` 。将每个节点上的 8080 端口映射到服务副本内部的 8080 端口。接下来，使用 `--replicas` 参数告知 Docker 应该总是有 5 个此服务的副本。最后，告知 Docker 哪个镜像用于副本——重要的是，要了解所有的服务副本使用相同的镜像和配置。

敲击回车键之后，主管理节点会在 Swarm 中实例化 5 个副本——请注意管理节点也会作为工作节点运行。相关各工作节点或管理节点会拉取镜像，然后启动一个运行在 8080 端口上的容器。

这还没有结束。所有的服务都会被 Swarm 持续监控——Swarm 会在后台进行轮训检查（Reconciliation Loop），来持续比较服务的实际状态和期望状态是否一致。如果一致，则皆大欢喜，无须任何额外操作；如果不一致，Swarm 会使其一致。换句话说，Swarm 会一直确保实际状态能够满足期望状态的要求。

举例说明，假如运行有 **web-fe** 副本的某个工作节点宕机了，则 **web-fe** 的实际状态从 5 个副本降为 4 个，从而不能满足期望状态的要求。Docker 变回启动一个新的 **web-fe** 副本来使实际状态与期望状态保持一致。这一特性功能强大，使得服务在面对节点宕机等问题时具有自愈能力。

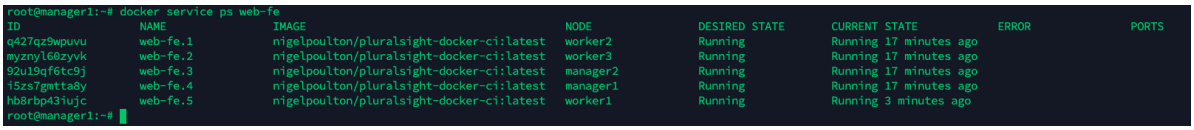
1．查看服务

使用 `docker service ls` 命令可以查看 Swarm 中所有运行中的服务。

```
root@manager1:~# docker service ls
ID                  NAME                MODE                REPLICAS
IMAGE                                          PORTS
7wa2zz179ef0        web-fe              replicated          5/5
nigelpoulton/pluralsight-docker-ci:latest    *:8080->8080/tcp
```

输出显示有一个运行中的服务及其相关状态信息。比如，可以了解服务的名称，以及 5 个期望的副本（容器）中有 5 个是运行状态。如果在部署服务后立即执行该命令，则可能并非所有的副本都处于运行状态。这通常取决于各个节点拉取镜像的时间。

执行 `docker service ps` 命令可以查看服务副本列表及各副本的状态：



此命令格式为 `docker service ps` 。每一个副本会作为一行输出，其中显示了各副本分别运行在 Swarm 的哪个节点上，以及期望的状态和实际状态。

关于服务更为详细的信息可以使用 `docker service inspect` 命令查看：

```
root@manager1:~# docker service inspect --pretty web-fe

ID:             7wa2zz179ef05o1u7h7ows3at
Name:           web-fe
Service Mode:   Replicated
 Replicas:      5
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:         nigelpoulton/pluralsight-docker-
ci:latest@sha256:7a6b0125fe7893e70dc63b2c42ad779e5866c6d2779ceb9b12a28e2c38bd8d3
d
 Init:          false
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 8080
  Protocol = tcp
  TargetPort = 8080
  PublishMode = ingress
```

以上例子使用了 `--pretty` 参数，限制输出中仅包含最感兴趣的内容，并以易于阅读的格式打印出来。不加 `--pretty` 的话会给出更加详尽的输出。强烈建议读者能够通读 `docker inspect` 命令的输出内容，其中不仅包含大量信息，也是了解底层运行机制的途径。

稍后还会再次探讨输出中的部分内容。

2．副本服务 vs 全局服务

服务的默认复制模式（Replication Mode）是副本模式（`replicated`）。这种模式会部署期望数量的服务副本，并尽可能均匀地将各个副本分布在整个集群中。

另一种模式是全局模式（`global`），在这种模式下，每个节点上仅运行一个副本。

可以通过给 `docker service create` 命令传递 `--mode global` 参数来部署一个全局服务。

3．服务的扩缩容

服务的另一个强大特性是能够方便地进行扩缩容。

假设业务呈爆发式增长，则 Web 前端服务接收到双倍的流量压力。所幸通过一个简单的 `docker service scale` 命令即可对 **web-fe** 服务进行扩容：

```
root@manager1:~# docker service scale web-fe=10
web-fe scaled to 10
overall progress: 10 out of 10 tasks
1/10: running   [==================================================>]
2/10: running   [==================================================>]
3/10: running   [==================================================>]
4/10: running   [==================================================>]
5/10: running   [==================================================>]
6/10: running   [==================================================>]
7/10: running   [==================================================>]
8/10: running   [==================================================>]
9/10: running   [==================================================>]
10/10: running   [==================================================>]
verify: Service converged
```

该命令会将服务副本数由 5 个增加到 10 个。后台会将服务的期望状态从 5 个增加到 10 个。运行 `docker service ls` 命令来检查操作是否成功：

```
root@manager1:~# docker service ls
ID                  NAME                MODE                REPLICAS
IMAGE                                          PORTS
7wa2zz179ef0        web-fe              replicated          10/10
nigelpoulton/pluralsight-docker-ci:latest   *:8080->8080/tcp
```

执行 `docker service ps` 命令会显示服务副本在各个节点上是均衡分布的：

```
root@manager1:~# docker service ps web-fe
ID                  NAME                IMAGE
  NODE                DESIRED STATE       CURRENT STATE                ERROR
          PORTS
q427qz9wpuvu        web-fe.1            nigelpoulton/pluralsight-docker-ci:latest
  worker2             Running             Running 21 minutes ago
myznyl6Ozyvk        web-fe.2            nigelpoulton/pluralsight-docker-ci:latest
  worker3             Running             Running 21 minutes ago
92u19qf6tc9j        web-fe.3            nigelpoulton/pluralsight-docker-ci:latest
  manager2            Running             Running 21 minutes ago
i5zs7gmtta8y        web-fe.4            nigelpoulton/pluralsight-docker-ci:latest
  manager1            Running             Running 21 minutes ago
hb8rbp43iujc        web-fe.5            nigelpoulton/pluralsight-docker-ci:latest
  worker1             Running             Running 7 minutes ago
hokmwr2pu8gi        web-fe.6            nigelpoulton/pluralsight-docker-ci:latest
  worker1             Running             Running about a minute ago
tfyt5ua1Odov        web-fe.7            nigelpoulton/pluralsight-docker-ci:latest
  worker2             Running             Running about a minute ago
p8rl7rkowh95        web-fe.8            nigelpoulton/pluralsight-docker-ci:latest
  worker3             Running             Running about a minute ago
4ltsf5zp8jtx        web-fe.9            nigelpoulton/pluralsight-docker-ci:latest
  manager3            Running             Running about a minute ago
hdutsa7peclb        web-fe.10           nigelpoulton/pluralsight-docker-ci:latest
  manager3            Running             Running about a minute ago
```

在底层实现上，Swarm 执行了一个调度算法，默认将副本尽量均衡分配给 Swarm 中的所有节点。截至目前，各节点分配的副本数是平均分配的，并未将 CPU 负载等指标考虑在内。

再次执行 `docker service scale` 命令将副本数从 10 个降为 5 个：

```
root@manager1:~# docker service scale web-fe=5
web-fe scaled to 5
overall progress: 5 out of 5 tasks
1/5: running   [==================================================>]
2/5: running   [==================================================>]
3/5: running   [==================================================>]
4/5: running   [==================================================>]
5/5: running   [==================================================>]
verify: Service converged
```

关于服务的扩缩容就介绍这些，下面看一下如何删除服务。

4. 删除服务

删除一个服务的操作相对比较简单——也许太简单了。

如下 `docker service rm` 命令可用于删除之前部署的服务：

```
$ docker service rm web-fe
web-fe
```

执行 `docker service ls` 命令以验证服务确实已被删除：

```
$ docker service ls
ID      NAME    MODE    REPLICAS    IMAGE    PORTS
```

请谨慎使用 `docker service rm` 命令，因为它在删除所有服务副本时并不会进行确认。

了解了如何删除一个服务，下面介绍一下如何对一个服务进行滚动升级。

5．滚动升级

对部署的应用进行滚动升级是常见的操作。长期以来，这一过程是令人痛苦的。我曾经牺牲了许多的周末时光来进行应用程序主版本的升级，而且再也不想这样做了。

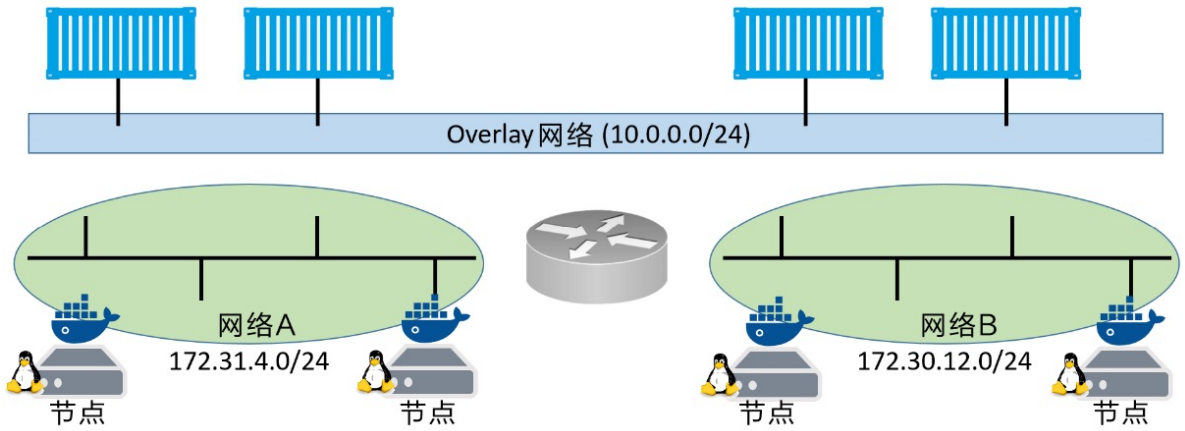
然而，多亏了 Docker 服务，对一个设计良好的应用来说，实施滚动升级已经变得简单多了！

为了演示如何操作，下面将部署一个新的服务。但是在此之前，先创建一个新的覆盖网络（Overlay Network）给服务使用。这并非必须的操作，但希望读者能够了解如何创建网络并将服务接入网络。

```
root@manager1:~# docker network create -d overlay uber-net
93mx4u9pj7uwe16jnaomywqit
```

该命令会创建一个名为 `uber-net` 的覆盖网络，接下来会将其与要创建的服务结合使用。覆盖网络是一个二层网络，容器可以接入该网络，并且所有接入的容器均可互相通信。即使这些容器所在的 Docker 主机位于不同的底层网络上，该覆盖网络依然是相通的。本质上说，覆盖网络是创建于底层异构网络之上的一个新的二层容器网络。

如下图所示，两个底层网络通过一个三层交换机连接，而基于这两个网络之上是一个覆盖网络。Docker 主机通过两个底层网络相连，而容器则通过覆盖网络相连。对于同一覆盖网络中的容器来说，即使其各自所在的 Docker 主机接入的是不同的底层网络，也是互通的。



执行 `docker network ls` 来查看网络是否创建成功，且在 Docker 主机可见：

```
root@manager1:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
4b7d09289c04        bridge              bridge              local
51f7e617547b        docker_gwbridge     bridge              local
5e3446fd4305        host                host                local
yq0q1msc9561        ingress             overlay             swarm
3758c1f9cf9a        none                null                local
93mx4u9pj7uw        uber-net            overlay             swarm
root@manager1:~#
```

可见，`uber-net` 网络已被成功创建，其 SCOPE 为 `swarm` ，并且目前仅在 Swarm 的管理节点可见。

下面创建一个新的服务，并将其接入 uber-net 网络：

```
root@manager1:~# docker service create --name uber-svc \
>    --network uber-net \
>    -p 8088:8088 --replicas 12 \
>    nigelpoulton/tu-demo:v1
vrj1ny7i8mh91l3951czwy64g
overall progress: 12 out of 12 tasks
1/12: running   [==================================================>]
2/12: running   [==================================================>]
3/12: running   [==================================================>]
4/12: running   [==================================================>]
5/12: running   [==================================================>]
6/12: running   [==================================================>]
7/12: running   [==================================================>]
8/12: running   [==================================================>]
9/12: running   [==================================================>]
10/12: running   [==================================================>]
11/12: running   [==================================================>]
12/12: running   [==================================================>]
verify: Service converged
root@manager1:~#
```

看一下上面的 `docker service create` 命令中做了哪些声明。

首先，将服务命名为 `uber-svc` ，并用 `--network` 参数声明所有的副本都连接到 `uber-net` 网络。然后，在整个 `swarm` 中将 80 端口暴露出来，并将其映射到 12 个容器副本的 80 端口。最后，声明所有的副本都基于 `nigelpoulton/tu-demo:v1` 镜像。

执行 `docker service ls` 和 `docker service ps` 命令以检查新创建服务的状态：

```
root@manager1:~# docker service ls
ID                  NAME                MODE                REPLICAS
IMAGE                          PORTS
vrj1ny7i8mh9        uber-svc            replicated          12/12
nigelpoulton/tu-demo:v1   *:80->80/tcp
root@manager1:~# docker service ps uber-svc
ID                  NAME                IMAGE                          NODE
     DESIRED STATE       CURRENT STATE            ERROR                PORTS
j48vt7ubiqfm        uber-svc.1          nigelpoulton/tu-demo:v1   manager3
     Running             Running 25 minutes ago
u863x5vx8j9h        uber-svc.2          nigelpoulton/tu-demo:v1   manager2
     Running             Running 18 minutes ago
v6yhzrsionxt        uber-svc.3          nigelpoulton/tu-demo:v1   manager1
     Running             Running 16 minutes ago
t8t9r3mskcgz        uber-svc.4          nigelpoulton/tu-demo:v1   worker2
    Running             Running 20 minutes ago
tdmmn3sfl8s1        uber-svc.5          nigelpoulton/tu-demo:v1   manager3
     Running             Running 25 minutes ago
ke1hnppzocmo        uber-svc.6          nigelpoulton/tu-demo:v1   worker1
    Running             Running 15 minutes ago
ufd9b984l9nn        uber-svc.7          nigelpoulton/tu-demo:v1   manager1
     Running             Running 16 minutes ago
m3b19qgkcc7i        uber-svc.8          nigelpoulton/tu-demo:v1   worker1
    Running             Running 15 minutes ago
vph6ftOcdrq6        uber-svc.9          nigelpoulton/tu-demo:v1   manager2
     Running             Running 18 minutes ago
```

```
pbjacuk9886j        uber-svc.10         nigelpoulton/tu-demo:v1   worker3
    Running             Running 16 minutes ago
ehnhviltr6jn        uber-svc.11         nigelpoulton/tu-demo:v1   worker3
    Running             Running 16 minutes ago
4hygbckqkpy7        uber-svc.12         nigelpoulton/tu-demo:v1   worker2
    Running             Running 20 minutes ago
root@manager1:~#
```

通过对服务声明 `-p 80:80` 参数，会建立 **Swarm 集群范围**的网络流量映射，到达 Swarm 任何节点 80 端口的流量，都会映射到任何服务副本的内部 80 端口。

默认的模式，是在 Swarm 中的所有节点开放端口——即使节点上没有服务的副本——称为入站模式（Ingress Mode）。此外还有主机模式（Host Mode），即仅在运行有容器副本的节点上开放端口。以主机模式开放服务端口，需要较长格式的声明语法，代码如下：

```
docker service create --name uber-svc \
   --network uber-net \
   --publish published=80,target=80,mode=host \
   --replicas 12 \
   nigelpoulton/tu-demo:v1
```

打开浏览器，使用 Swarm 中任何一个节点的 IP，进入 80 端口的界面，查看服务运行情况，如下图所示：

如读者所见，这是一个简单的投票程序，它能够注册对“footbal”或“soccer”的投票。读者可随意在浏览器中使用其他节点的 IP，均能够打开该页面，因为 `-p 80:80` 参数会在所有 Swarm 节点创建一个入站模式的端口映射。即使某个节点上并未运行服务的副本，依然可以进入该页面——

所有节点都配置有映射，因此会将请求转发给运行有服务副本的节点。

假设本次投票已经结束，而公司希望开启一轮新的投票。现在已经为下一轮投票构建了一个新镜像，并推送到了 Docker Hub 仓库，新镜像的 tag 由 `v1` 变更为 `v2` 。

此外还假设，本次升级任务在将新镜像更新到 Swarm 中时采用一种阶段性的方式——每次更新两个副本，并且中间间隔 20s。那么就可以采用如下的 `docker service update` 命令来完成。

```
$ docker service update \
   --image nigelpoulton/tu-demo:v2 \
   --update-parallelism 2 \
   --update-delay 20s uber-svc
```

仔细观察该命令，`docker service update` 通过变更该服务期望状态的方式来更新运行中的服务。这一次我们指定了 tag 为 `v2` 的新镜像。接下来用 `--update-parallelism` 和 `--update-delay` 参数声明每次使用新镜像更新两个副本，其间有 20s 的延迟。最终，告知 Docker 以上变更是对 `uber-svc` 服务展开的。

如果对该服务执行`docker service ps`命令会发现，有些副本的版本号是 `v2` 而有些依然是 `v1` 。如果给予该操作足够的时间（4min），则所有的副本最终都会达到新的期望状态，即基于 `v2` 版本的镜像。

```
$ docker service ps uber-svc
ID        NAME              IMAGE       NODE  DESIRED   CURRENT STATE
7z...nys  uber-svc.1    nigel...v2  mamager2  Running   Running 13 secs
0v...7e5  \_uber-svc.1  nigel...v1  wrk3  Shutdown  Shutdown 13 secs
bh...wa0  uber-svc.2    nigel...v1  wrk2  Running   Running 1 min
e3...gr2  uber-svc.3    nigel...v2  wrk2  Running   Running 13 secs
23. ..u97  \_uber-svc.3  nigel...v1  wrk2  Shutdown  Shutdown 13 secs
82. ..5y1  uber-svc.4    nigel...v1  mamager2  Running   Running 1 min
c3...gny  uber-svc.5    nigel...v1  wrk3  Running   Running 1 min
e6...3u0  uber-svc.6    nigel...v1  wrk1  Running   Running 1 min
78. ..r7z  uber-svc.7    nigel...v1  wrk1  Running   Running 1 min
2m...kdz  uber-svc.8    nigel...v1  mamager3  Running   Running 1 min
b9...k7w  uber-svc.9    nigel...v1  mamager3  Running   Running 1 min
ag...v16  uber-svc.10   nigel...v1  mamager2  Running   Running 1 min
e6...dfk  uber-svc.11   nigel...v1  mamager1  Running   Running 1 min
e2...k1j  uber-svc.12   nigel...v1  mamager1  Running   Running 1 min
```

如果读者在更新操作完成前打开浏览器，使用 Swarm 中任一节点的 IP 进入页面，并多次单击刷新按钮，就会看到滚动更新的效果。有些请求会被旧版本的副本处理，而有些请求会被新版本的副本处理。一段时间之后，所有的请求都会被新版本的服务副本处理。

恭喜。想必读者也完成了对运行中的容器化应用程序的滚动更新。请注意，在第 14 章会介绍 Docker Stack 如何将这一操作进一步优化提升。

此时如果对服务执行 `docker inspect --pretty` 命令，会发现更新时对并行和延迟的设置已经成为服务定义的一部分了。这意味着，之后的更新操作将会自动使用这些设置，直到再次使用 `docker service update` 命令覆盖它们。

```
$ docker service inspect --pretty uber-svc
ID:             mub0dgtc8szm80ez5bs8wlt19
Name:Service uber-svc
Mode:           Replicated
 Replicas:      12
UpdateStatus:
 State:         updating
 Started:       About a minute
```

```
 Message:       update in progress
Placement:
UpdateConfig:
 Parallelism:   2
 Delay:         20s
 On failure:    pause
 Monitoring Period:  5s
 Max failure ratio:  0
 Update order:       stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:    nigelpoulton/tu-demo:v2@sha256:d3c0d8c9...cf0ef2ba5eb74c
Resources: Networks:
uber-net Endpoint
Mode: vip Ports:

 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
```

如上还应注意到关于服务的网络配置的内容。Swarm 中的所有运行副本的节点都会使用前面创建的 `uber-net` 覆盖网络。读者可以通过在运行副本的任一节点执行 `docker network ls` 命令来验证这一点。

此外，请注意 `docker inspect` 输出的 `Networks` 部分，不仅显示了 `uber-net` 网络，还显示了 Swarm 范围的 80:80 端口映射。

故障排除

Swarm 服务的日志可以通过执行 `docker service logs` 命令来查看，然而并非所有的日志驱动（Logging Driver）都支持该命令。

Docker 节点默认的配置是，服务使用 `json-file` 日志驱动，其他的驱动还有 `journald`（仅用于运行有 systemd 的 Linux 主机）、`syslog`、`splunk` 和 `gelf`。

`json-file` 和 `journald` 是较容易配置的，二者都可用于 `docker service logs` 命令。命令格式为 `docker service logs`。

若使用第三方日志驱动，那么就需要用相应日志平台的原生工具来查看日志。

如下是在 `daemon.json` 配置文件中定义使用 `syslog` 作为日志驱动的示例。

```
{
  "log-driver": "syslog"
}
```

通过在执行 `docker service create` 命令时传入 `--logdriver` 和 `--log-opts` 参数可以强制某服务使用一个不同的日志驱动，这会覆盖 `daemon.json` 中的配置。

服务日志能够正常工作的前提是，容器内的应用程序运行于 PID 为 1 的进程，并且将日志发送给 `STDOUT`，错误信息发送给 `STDERR`。日志驱动会将这些日志转发到其配置指定的位置。

如下的 `docker service logs` 命令示例显示了服务 `svc1` 的所有副本的日志，可见该服务在启动副本时出现了一些错误。

```
$ docker service logs seastack_reverse_proxy
svc1.1.zhc3cjeti9d4@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | nginx: [emerg] host not found..
svc1.1.zhc3cjeti9d4@wrk-2 | nginx: [emerg] host not found..
svc1.1.1tmya243m5um@mamager-1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

以上输出内容有删减，不过仍然可以看到来自服务的 3 个副本的日志（两个运行失败，一个运行成功）。每一行开头为副本名称，其中包括服务名称、副本编号、副本 ID 以及所在的主机。之后是日志消息。

由于输出内容有所删减，因此失败原因较难定位，不过看起来似乎是前两个副本尝试连接另一个启动中的服务而导致失败（一种所依赖的服务未完全启动导致的竞态条件问题）。

对于查看日志命令，可以使用 `--follow` 进行跟踪、使用 `--tail` 显示最近的日志，并使用 `--details` 获取额外细节。

Docker Swarm 命令

- `docker swarm init` 命令用户创建一个新的 Swarm。执行该命令的节点会成为第一个管理节点，并且会切换到 Swarm 模式。
- `docker swarm join-token` 命令用于查询加入管理节点和工作节点到现有 Swarm 时所使用的命令和 Token。要获取新增管理节点的命令，请执行 `docker swarm join-token manager` 命令；要获取新增工作节点的命令，请执行 `docker swarm join-token worker` 命令。
- `docker node ls` 命令用于列出 Swarm 中的所有节点及相关信息，包括哪些是管理节点、哪个是主管理节点。
- `docker service create` 命令用于创建一个新服务。
- `docker service ls` 命令用于列出 Swarm 中运行的服务，以及诸如服务状态、服务副本等基本信息。
- `docker service ps` 命令会给出更多关于某个服务副本的信息。
- `docker service inspect` 命令用于获取关于服务的详尽信息。附加 `--pretty` 参数可限制仅显示重要信息。
- `docker service scale` 命令用于对服务副本个数进行增减。
- `docker service update` 命令用于对运行中的服务的属性进行变更。
- `docker service logs` 命令用于查看服务的日志。
- `docker service rm` 命令用于从 Swarm 中删除某服务。该命令会在不做确认的情况下删除服务的所有副本，所以使用时应保持警惕。

总结

Docker Swarm 是使 Docker 规模化的关键方案。

Docker Swarm 的核心包含一个安全集群组件和一个编排组件。

安全集群管理组件是一个企业级的安全套件，提供了丰富的安全机制以及 HA 特性，这些都是自动配置好的，并且非常易于调整。

编排组件允许用户以一种简单的声明式的方式来部署和管理微服务应用。它不仅支持原生的 Docker Swarm 应用，还支持 Kubernetes 应用。