第 5 章

# 容 器 编 排

## 5.1 容器编排简介

首先理解编排两个字的含义，编排在计算机领域是指对系统的自动配置和管理。像 Ansible 这样的工具就是用来对服务器进行配置和管理的编排工具，而容器编排是指容器的配置、部署和管理等任务的自动化。容器编排可以把开发者从复杂且重复的容器管理工作中拯救出来。

如果只需管理一个容器，则大可不必使用容器编排，但现实是一个容器往往无法承载日益复杂的应用，如果把应用所有的组件都放到一个容器中运行，就丧失了使用容器的诸多优势，无法完成快速扩容缩容、无法对某个组件进行单独部署和升级及无法灵活地调度到其他服务器。现在微服务的概念已经深入人心，越来越多的应用以松耦合的形式来设计架构。对这样的应用容器化后，一定是以多容器的形式来运行的。

从 4.6 节手动运行和管理容器的过程中可以体会到多容器的管理比较复杂。手工管理不仅非常低效，而且极易出错，在容器数量增长到一定量级的情况下，使用容器编排的必要性就更加显著。尤其是对于稳定性有极高要求的生产环境，最大可能减少手工操作就能降低出错的风险。

中大型的应用一般会由多个组件构成，例如一个常见 Web 应用程序可能需要同时用到 Web 服务器容器、数据库容器及用作缓存的 Redis 容器，后续可能会引入消息队列容器。把应用容器化之后，每个组件都会运行在独立的容器中。这样应用就需要多个容器紧密配合才能运行。为了让多个容器正确协作，需要保证容器按一定的顺序启动，需要维持运行中的容器数量，需要控制各个容器的状态，通过容器编排技术，可以自动化完成这些任务，使多个容器组成的整体达到期望的状态。

容器编排是使用容器的实践中比较复杂的一个课题。Docker 提供了一定的容器编排的能力，而作为容器编排领域事实标准的 Kubernetes 提供了更全面、更强大的编排能力。

## 5.2 Docker Compose

Docker 提供了 Compose 组件，用于实现简单的容器编排。Docker Compose 会从文件中读取应用所需的全部容器的定义，这个文件默认的文件名是 docker-compose.yaml，可以

把它称为 Compose 文件。有了这个文件以后，只需运行 docker-compose up 命令就可以把应用所需的所有容器启动，把所需的网络和存储准备就绪。Compose 文件相当于把项目的运行环境文档化。实现了用一个命令就完成了烦琐启动多个容器的任务。提供给项目一个完整且隔离的运行环境。

Docker Compose 只能管理单节点上的容器，所以更适合在开发和测试环境下使用。

下面使用 Docker Compose 对第 1 章中介绍的应用作容器编排。

### 5.2.1 Compose 文件

在后端项目的根目录下定义一个 docker-compose.yaml 文件，内容如下：

```
version: "3.9"

services:
  web:
    image: accounts-frontend:v1.0.0
    depends_on:
      - api
    ports:
      - "8088:80"

api:
    depends_on:
      - mysql
    image: accounts:v1.0.0
    ports:
      - "8081:80"
    volumes:
      - ./config:/config
    restart: always

  mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 123
MYSQL_DATABASE: accounts
    command: --character-set-server=utf8mb4 --default-time-zone=+08:00

db-migration:
    image: goose
    depends_on:
```

```
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
    command: 'goose up'
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:123@tcp(mysql)/accounts

volumes:
  mysql:
    external: true
```

下面分析一下这个文件的内容。

下面一行定义了 Compose 文件格式的版本，推荐使用最新的版本，但是也要注意和 Docker 的版本兼容，代码如下：

```
version: "3.9"
```

从这一行开始定义应用需要的所有服务。服务是对多个运行相同任务的容器的抽象。因为 Compose 支持扩展运行某个服务的实例数，所以服务会运行在一个或多个容器实例中，代码如下：

```
services:
```

定义前端项目 accounts-frontend 提供的服务，并把它命名为 web，web 这个名字将成为这个服务中容器的网络别名，也就是说，同一网络中的其他容器可以使用 web 作为主机名解析到这个容器。image 字段用于指定 web 容器的镜像。depends_on 字段用于指定 web 容器依赖的其他所有容器。depends_on 字段决定了容器启动的顺序，它会使依赖的容器先行启动，然后启动 web 容器。ports 字段定义了容器端口的映射，会把宿主机的 TCP 8080 端口映射到容器中的 TCP 80 端口，代码如下：

```
web:
  image: accounts-frontend:v1.0.0
  depends_on:
    - api
  ports:
    - "8088:80"
```

定义后端项目 accounts 提供的服务，并把它命名为 api。depends_on 字段用于指定这个服务依赖于 mysql 服务。ports 字段把宿主机的 TCP 8081 端口映射到容器中的 TCP 80 端口。volumes 字段用于将宿主机上与这个 docker-compose.yaml 文件同级的 config 文件

挂载到容器中的/config 文件夹下。restart 字段用于指定容器在中止运行后重启的策略，always 表示在任何情况下中止运行后都会重启，代码如下：

```
api:
    depends_on:
      - mysql
    image: accounts:v1.0.0
    ports:
      - "8081:80"
    volumes:
      - ./config:/config
    restart: always
```

定义 MySQL 服务并把它命名为 mysql。image 字段指定了容器的镜像是 mysql:5.7。volumes 字段用于把名为 mysql 的数据卷挂载到容器中的/var/lib/mysql 文件夹下。

environment 字段用于定义容器中的环境变量。这里定义了 MYSQL_ROOT_PASSWORD，用来设置 root 用户的密码。MYSQL_DATABASE 环境变量用来设置 MySQL 启动时自动创建的数据库名字，当指定的数据库已经存在时，不会影响现有的数据。

command 中的--character-set-server=utf8mb4 指定了 MySQL 启动时的默认字符集为 utf8mb4，不设置默认字符集会默认为 latin1，--default-time-zone=+08:00 将默认时区设置为东八时区。由于 mysql 容器的默认执行的命令是 mysqld，代码如下：

```
mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 123
MYSQL_DATABASE: accounts
    command: --character-set-server=utf8mb4 --default-time-zone=+08:00
```

定义运行数据迁移的服务并把它命名为 db-migration。image 字段指定了容器的镜像是 goose，代码如下：

```
db-migration:
    image: goose
    depends_on:
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
```

```
    command: 'goose up'
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:123@tcp(mysql)/accounts
```

定义应用中需要用到的数据卷。第1个数据卷mysql正是上面定义的mysql容器，此容器用来挂载数据卷，规定了它的属性external是true，表示这个数据卷是在外部的Docker中已有的，无须由Compose创建。

```
volumes:
  mysql:
    external: true
```

### 5.2.2 Compose环境变量

目前的Compose文件中没有使用环境变量，当需要改变文件中的某些值的时候需要修改Compose文件，这样不太方便。如果镜像版本发生了变化，就需要在文件中改动。

Compose支持使用环境变量替代Compose文件中的值。例如可以把文件中的这部分代码进行替换：

```
image: accounts-frontend: v1.0.0
```

替换成使用环境变量的格式：

```
image: ${ACCOUNTS_FRONTEND_IMAGE}
```

这样当accounts-frontend镜像发生变化的时候，通过改动环境变量ACCOUNTS_FRONTEND_IMAGE就可以避免修改Compose文件。例如当镜像版本升级到v1.1.0时，把ACCOUNTS_FRONTEND_IMAGE环境变量设置为accounts-frontend:v1.1.0，这样Compose就会在启动后自动把Compose文件中相应的变量替换成accounts-frontend:v1.1.0。如果Compose文件中用到的环境变量没有定义，则最终对应的位置会变成一个空字符串。

也可以把环境变量写到一个叫作.env的文件中，这会使环境变量的管理变得方便。Compose在启动时会自动读取Compose文件所在文件夹下的.env文件。在.env文件中写入以下内容：

```
ACCOUNTS_FRONTEND_IMAGE=accounts-frontend: v1.0.0
```

为了验证环境变量配置是否正确，预览一下经过替换变量处理后的Compose文件，命令如下：

```
docker-compose config
```

在输出的信息中会看到变量 ACCOUNTS_FRONTEND_IMAGE 替换后的结果如下：

```
image: accounts-frontend:v1.0.0
```

如果环境变量文件的位置和 Compose 文件不在同一个文件夹下或者文件名并不是默认的.env，就可以使用--env-file 参数来告诉 Compose 它的具体位置。利用这个功能，可以把不同环境下用到的环境变量写到不同的文件中，例如把集成环境用到的环境变量写到.env.ci 文件中，把开发环境用到的环境变量写到.env.dev 中。之后在运行 Compose 的时候把对应的文件通过--env-file 参数传递到 Compose。

如果 Compose 文件中的某些值重复出现在多个位置，则使用环境变量还可以使修改值的过程变得更简单。不再需要在文件中多处修改同一个值，只需修改一次环境变量。

下面是 Compose 文件引入更多环境变量后最终的版本：

```
version: "3.9"

services:
  web:
    image: ${ACCOUNT_FRONTEND_IMAGE}
    depends_on:
      - api
    ports:
      - "8088:80"

api:
    depends_on:
      - mysql
    image: ${ACCOUNT_IMAGE}
    ports:
      - "8081:80"
    volumes:
      - ./config:/config
    restart: always
    command: go run src/server/main.go

  mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
```

```
      MYSQL_DATABASE: ${MYSQL_DB}
    command: --character-set-server=utf8mb4 --default-time-zone=+08:00

  db-migration:
    image: goose
    depends_on:
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
    command: goose up
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:${MYSQL_PASSWORD}@tcp(mysql:3306)/${MYSQL_DB}
volumes:
  mysql:
    external: true
```

可以看到引入环境变量之后，降低了 Compose 文件中值的耦合。相应地，需要在.env 文件中增加新引入的环境变量，修改后的代码如下：

```
ACCOUNTS_FRONTEND_IMAGE=accounts-frontend:v1.0.0
ACCOUNT_IMAGE=accounts:v1.0.0
MYSQL_PASSWORD=123
MYSQL_DB=accounts
```

### 5.2.3 Compose 运行应用

现在 docker-compose.yaml 文件已经准备好了，下面用 Docker Compose 来运行应用。命令如下：

```
docker-compose up
```

运行命令后会看到输出信息中包含以下部分：

```
Creating network "accounts_default" with the default driver
Creating accounts_mysql_1 ... done
Creating accounts_api_1 ... done
Creating accounts_db-migration_1 ... done
Creating accounts_web_1        ... done
```

输出日志比较多，并且都显示在终端，如果希望在后台运行，则可以在命令中加上-d 参数。如果以-d 参数运行，则查看日志需要使用 docker-compose logs 命令。

输出信息第一行表示 Docker Compose 自动创建了一个基于默认驱动器（也就是

bridge)的网络,网络名为 accounts_default。这样就为应用提供了网络隔离。

用 docker network 命令进行验证,命令如下:

```
docker network ls
NETWORK ID     NAME                DRIVER    SCOPE
f64a9cafa09e   accounts_default    bridge    local
```

发现多了一个 accounts_default 网络。

通过 Docker Compose 运行起来的容器同样也可以用 docker ps 命令看到。为了方便观察,这里对 docker ps 命令输出的信息做了适当截取,命令如下:

```
docker ps
CONTAINER ID   IMAGE                NAMES
77a16cd7ac42   accounts-frontend:v1.0.0   accounts_web_1
7a5e6fc79984   accounts:v1.0.0 accounts_api_1
608fbe95fc17   mysql:5.7              accounts_mysql_1
```

下面介绍一下 Docker Compose 对容器命名的方式。以 accounts_web_1 容器为例,accounts_web_1 中的 accounts 部分是项目名,默认为用项目根目录的文件夹名字命名,如果希望自己指定项目名,则可以在启动时加上参数-p,web 是在 Compose 文件中定义的容器名或者叫服务名,最后的数字 1 表示 web 容器的个数,也就是它一共运行了多少个实例。

也可以用 Compose 提供的命令来查看容器的运行情况,命令如下:

```
docker-compose ps
```

输出如下:

```
Name                        Command                        State          Ports
-------------------------------------------------------------------------------
accounts_api_1   /accounts                          Up     0.0.0.0:8081->80/tcp
accounts_db-migration_1   goose up                         Exit 1
accounts_mysql_1   docker-entrypoint.sh mysqld         Up       3306/tcp, 33060/tcp
accounts_web_1    /docker-entrypoint.sh ngin ...       Up     0.0.0.0:8088->80/tcp
```

和 docker ps 命令不同的是:这条命令会只显示 docker-compose.yaml 文件中有关的容器。这样就可以方便观察。

在输出的信息中可以发现 accounts_db-migration_1 的状态是 Exit 1,证明容器中产生了某种错误而退出了。

查看 Compose 输出的日志,会发现下面的信息:

```
db-migration_1   | 2021/03/05 12:47:25 goose run: dial tcp 172.18.0.2:3306: connect:
connection refused
```

表明 db-migration_1 连接 MySQL 数据库失败。这个容器用来运行 goose 进行数据迁移，如果无法与数据库时连接，就无法正常运行。

尝试单独运行 db-migration，使用的命令如下：

```
docker-compose run db-migration
```

运行命令会看到以下输出：

```
Creating accounts_db-migration_run ... done
2021/03/05 12:56:50 OK    20210304203736_create_table_user.sql
2021/03/05 12:56:50 goose: no migrations to run. current version: 20210304203736
```

这次 db-migration 运行成功了，说明之前出现连接失败是因为 MySQL 没有准备就绪。

虽然这个 Compose 文件中定义了 db-migration 与 mysql 之间的依赖关系，但是这只能使 Compose 按照依赖关系的顺序运行容器，并不保证容器就绪的顺序也严格遵从这个依赖关系。只要被依赖的容器开始运行，Compose 就会开始创建依赖它的容器。容器进入运行状态不一定表示它已经准备就绪，有的容器如 mysql 从开始运行到就绪需要一段时间来初始化，mysql 就绪的标志是在日志中输出[Note] mysqld: ready for connections。从日志中可以看出 mysql 容器从启动到就绪所用的时间远大于 db-migration，当 db-migration 尝试连接 mysql 的时候，mysql 并没有准备就绪。这就是为什么 db-migration 在启动后连接 mysql 会失败的原因。

同样地，api 服务的容器也会由于这个原因在启动过程中连接数据库失败，但是 api 会在连接数据库失败后不断重试，在 mysql 就绪以后，重试连接就成功了，所以不需要人工介入 accounts 也能在最后进入正常状态。

服务就绪顺序与依赖顺序不一致问题的一个很好的解决方案就是提高对依赖的服务处于未就绪状态的容忍度，也就是增加服务的弹性。

到这里应用就已经通过 Compose 这种方式正常运行起来了。打开网址 http://localhost:8088 可以看到前端的页面可正常显示。

当前版本的 docker-compose.yaml 文件引用了一个外部的 mysql 数据卷，增加了在其他机器上复现环境的复杂性，在其他机器上要用这个文件运行 docker-compose up，也要先手动创建数据卷。在不需要引用外部数据卷的场景下，可以把文件中的 volumes 部分做如下修改：

```
volumes:
    mysql:
```

去掉了属性 external:true。这样，mysql 数据卷就会由 Compose 来创建和管理了。

为了观察修改后的文件所带来的变化，用 Compose 重新运行应用。重启之前需要关闭整个应用。命令如下：

```
docker-compose down
```

这个命令会把与应用相关的所有容器都中止，和手动管理每个容器相比，Docker Compose 把多个容器作为整体进行管理，带来了很大的便捷。

接着重新运行 docker-compose up，会看到输出信息中相较之前多了一行这样的文本：

```
Creating volume "accounts_mysql" with default driver
```

表明 Compose 创建了一个名为 accounts_mysql 的数据库，可以用 docker volume 命令验证，命令如下：

```
$ docker volume ls
DRIVER    VOLUME NAME
local     accounts_mysql
```

本地的数据卷中出现了新创建的 accounts_mysql 数据卷。证明 Compose 会根据文件中的定义自动创建所需要的数据卷。

Compose 会保留应用使用的数据卷，下次启动应用时会把上次运行产生的数据卷复制到新的容器中，这样就不会使数据丢失了。

要关闭应用，需要用到下面的命令：

```
docker-compose stop
```

运行命令会看到以下输出信息：

```
Stopping accounts_web_1       ... done
Stopping accounts_api_1 ... done
Stopping accounts_mysql_1     ... done
```

可以看到 docker-compose stop 命令会把与应用相关的所有运行中的容器都中止。

再次启动容器时可以用的命令如下：

```
docker-compose start
```

如果需要中止应用并且把应用产生的所有容器和网络都清除，则可以运行的命令如下：

```
docker-compose down
```

运行命令后输出的信息如下：

```
Stopping accounts_web_1       ... done
Stopping accounts_api_1 ... done
```

```
Stopping accounts_mysql_1     ... done
Removing accounts_web_1           ... done
Removing accounts_api_1      ... done
Removing accounts_db-migration_1 ... done
Removing accounts_mysql_1         ... done
Removing network accounts_default
```

在 Compose 文件中声明了命名数据卷的情况下，为数据安全起见，使用 docker-compose down 命令关闭应用并不会清除它创建的数据卷。需要加上--volume 参数才会把数据卷也清除。重新执行 docker-compose up 启动应用，然后输入的命令如下：

```
docker-compose down --volume
```

运行这条命令会看到下面的输出信息：

```
Stopping accounts_web_1       ... done
Stopping accounts_api_1 ... done
Stopping accounts_mysql_1     ... done
Removing accounts_web_1           ... done
Removing accounts_db-migration_1 ... done
Removing accounts_api_1      ... done
Removing accounts_mysql_1         ... done
Removing network accounts_default
Removing volume accounts_mysql
```

可以看到加了--volume 参数后，输出信息中出现了下面这一行：

```
Removing volume accounts_mysql
```

说明把数据卷 accounts_mysql 删除了。

### 5.2.4 Compose 更新应用

当更新应用代码以后，会构建出新版本的容器镜像，接下来应如何对运行中的应用进行升级呢？不需要手动中止旧容器，然后运行新容器来完成升级，只需修改 Compose 文件中相应的镜像版本，修改完 Compose 文件后，重新运行 docker-compose up，Compose 就会自动中止旧容器，创建新容器。这样就完成了对应用的更新。

## 5.3 Docker Swarm

与 Docker Compose 只能管理本地单个节点上的容器不同的是 Docker Swarm 可以管理多个节点上的容器，也就是具有了管理 Docker 集群的能力。学习 Swarm 可以熟悉运维

和管理容器集群过程中会遇到的一些常见问题。

使用 Swarm 要求 Docker Engine 的版本不低于 1.12。

Swarm 集群中的节点分为 manager 节点和 worker 节点。节点即 Docker 主机，也就是运行 Docker Engine 的机器，Swarm 支持在多个节点上运行，也支持在单节点上运行。manager 节点也可以执行 worker 节点的任务，当然也可以设置成 manager 节点只执行 manager 的任务。Swarm 中的节点必须以 Swarm 模式运行。

在分布式环境中，节点之间需要确保安全的通信，不仅需要对通信进行加密，还需对身份进行验证。Swarm 集群节点之间通过双向 TLS 实现安全通信，当运行 docker swarm init 命令时，这个 manager 节点上会生成一个新的 root CA 和一对公私钥。同时 manager 节点还会生成 2 个 token，一个是 manager token 用于其他 manager 节点加入集群，另一个是 worker token，用于其他 worker 节点加入集群。每个 token 都包含 root CA 证书的摘要信息还有一个随机生成的密钥。当节点加入集群时，节点就可以根据 token 中的 root CA 摘要来验证 manager 的真实性，而 manager 节点用 token 中的密钥来验证待加入节点的有效性。

当节点加入集群时，还会在节点生成一对公私钥及 root CA 证书。文件保存在/var/lib/docker/swarm/certificates 下。其中公钥证书的文件名是 swarm-node.crt，私钥的文件名为 swarm-node.key。root CA 的证书名为 swarm-root-ca.crt。swarm-root-ca.crt 文件内容和集群中第 1 个 manager 节点上的 root CA 的文件内容完全一致。

manager 节点负责管理 worker 节点，具有 leader 身份的 manager 节点负责将任务分配到 worker 节点。manager 节点之间使用 Raft 算法来达成共识。所有的 manager 节点都参与维护 Swarm 集群内部状态，使其在不同的 manager 节点保持一致。

Swarm 集群中的 manager 节点数量决定了集群的容错能力，如果集群节点数是 $n$，则集群最多可以容忍$(n-1/2)$个节点失败。例如 3 节点的集群，最多可以容忍 1 个节点失败，4 节点的集群，同样也只能容忍 1 个节点失败，所以最好使用奇数个 manager 节点。偶数个节点和比它小 1 的奇数相比并不能提高集群的容错能力。为了更佳的可用性，可以把 manager 节点分布到不同的 IDC 区域。

虽然增加 manager 数量可以提高集群容错能力，但是也会影响 Swarm 的性能。Docker 官方建议使用的 manager 节点数不超过 7 个。

worker 节点只负责运行服务而不负责管理集群。所有的 worker 节点上都运行一个代理，用来通知 master 节点自己的状态。这样 master 节点就可以根据通知来对 worker 节点进行管理。在 worker 上的任务失败后，manager 可以把这个任务调度到其他节点执行。

### 5.3.1 创建 Swarm 集群

下面演示 Swarm 集群功能，准备 3 台 Linux 服务器，1 台作为 manager 节点，其余 2 台作为 worker 节点。所有服务器都安装了 Docker。3 台服务器的操作系统都是 Ubuntu 20.04。

在第 1 台服务器上创建 Swarm 集群的 manager 节点，命令如下：

```
docker swarm init --advertise-addr 54.255.243.130
```

命令中的--advertise 参数用来指定 manager 节点的 IP 地址。运行命令输出的信息如下：

```
Swarm initialized: current node (f2hkxhks4g4msvann1ehargyy) is now a manager.

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

    docker swarm join --token \
    SWMTKN-1-4loyfwaqwwoqdoutbscaw0t877h4o3pnoa0yytf6t5mstd5uwj-e97svc3o150sm7xp8qx18ad5p
54.255.243.130:2377

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

表示集群中的 manager 节点启动了。输出信息中提示 worker 节点加入这个集群的命令如下：

```
docker swarm join --token \
    SWMTKN-1-4loyfwaqwwoqdoutbscaw0t877h4o3pnoa0yytf6t5mstd5uwj-e97svc3o150sm7xp8qx18ad5p
54.255.243.130:2377
```

之后会在 worker 节点上运行此命令来加入集群。

worker 加入集群用到的 worker token 后续还可以单独查询，命令如下：

```
docker swarm join-token worker
```

运行命令后输出的信息如下：

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

    docker swarm join --token \
    SWMTKN-1-4loyfwaqwwoqdoutbscaw0t877h4o3pnoa0yytf6t5mstd5uwj-e97svc3o150sm7xp8qx18ad5p
54.255.243.130:2377
```

如果希望其他节点以 manager 角色加入集群，则需要获取 manager token，命令如下：

```
docker swarm join-token manager
```

运行命令后输出的信息如下：

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

    docker swarm join --token \
    SWMTKN-1-55zvv6p3lqogypvzwmbo76sd9c90wx2l4a74o4myqgs3cq4ht8-btap3xq3znef742sageey94yt
172.31.30.250:2377
```

创建集群后，查看一下目前集群中的所有节点，命令如下：

```
docker node ls
```

运行命令后输出类似如下信息：

```
ID          HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION
r29f..  *  ip-172-31-30-250 ReadyActive Leader  20.10.5
```

在输出信息中对 ID 列做了截取，STATUS 列是 Active，表示 manager 节点运行正常。
下面在第 2 台服务器（作为 worker 节点）运行命令如下：

```
docker swarm join --token \
SWMTKN-1-4loyfwaqwwoqdoutbscaw0t877h4o3pnoa0yytf6t5mstd5uwj-e97svc3o150sm7xp8qx18ad5p
54.255.243.130:2377
```

运行命令后输出的信息如下：

```
This node joined a swarm as a worker.
```

表示这台服务器作为 worker 节点成功加入了 Swarm 集群中。
同样地，在第 3 台服务器也执行同样的操作，命令如下：

```
docker swarm join --token \
SWMTKN-1-4loyfwaqwwoqdoutbscaw0t877h4o3pnoa0yytf6t5mstd5uwj-e97svc3o150sm7xp8qx18ad5p
54.255.243.130:2377
```

同样地，会输出如下信息：

```
This node joined a swarm as a worker.
```

现在重新查看集群中的节点，命令如下：

```
docker node ls
```

运行命令后输出类似如下信息（前 4 列）：

```
ID                            HOSTNAME              STATUS    AVAILABILITY
f2hkxhks4g4msvann1ehargyy *   ip-172-31-30-250      Ready     Active
iobh29mwpmnq0fu3w1q85qt73     ip-172-31-32-49       Ready     Active
t1e9y94rrw6rrrxz16og1djt5     ip-172-31-35-212      Ready     Active
```

输出信息中后 2 列的内容如下：

```
MANAGER STATUS    ENGINE VERSION
Leader              20.10.5
                    20.10.5
                    20.10.5
```

表示 3 台服务器节点都成功加入了一个 Swarm 集群。输出信息中的 STATUS 列全部是 Ready，表示集群中的节点都处于正常状态。AVAILABILITY 列全部是 Active，表示节点都可以用于运行服务所包含的任务，也就是说可以把运行任务的容器调度到这些节点上。

管理 Swarm 集群时需要在 manager 节点运行命令，如果在 worker 节点运行管理 Swarm 集群相关的命令，则提示如下：

```
Error response from daemon: This node is not a swarm manager. Worker nodes can't be used to view or modify cluster state. Please run this command on a manager node or promote the current node to a manager.
```

目前集群中有了 3 个节点，1 个 manage，2 个 worker，为了提升集群的容错能力，需要至少 3 个 manager 节点，可以在节点加入集群时使用 manager token 来成为 manager 节点，也可以把现在集群中的 2 个 worker 节点升级为 manager 节点，命令如下：

```
docker node promote ip-172-31-32-49
```

运行命令后输出的信息如下：

```
Node ip-172-31-32-49 promoted to a manager in the swarm.
```

同样地，将另外一个 worker 节点也升级为 manager 节点，命令如下：

```
docker node promote ip-172-31-35-212
```

重新查看集群节点信息，命令如下：

```
docker node ls
```

输出信息如下：

```
ID                            HOSTNAME              STATUS    AVAILABILITY
r29f4jyze4zlaydw5gqvljb2w *   ip-172-31-30-250      Ready     Active
uks3x7m5xth4efe6ywn5q4ost     ip-172-31-32-49       Ready     Active
a4odmh76r7983qskgix939os9     ip-172-31-35-212      Ready     Active
```

后两列信息如下：

```
MANAGER STATUS   ENGINE VERSION
Leader           20.10.5
Reachable        20.10.5
Reachable        20.10.5
```

同样支持把 manager 节点降级为 worker 节点，命令如下：

```
docker node demote ip-172-31-32-49
```

运行命令后输出的信息如下：

```
Manager ip-172-31-32-49 demoted in the swarm.
```

如果需要把节点从集群中移除，可以在节点上运行命令 docker swarm leave。

### 5.3.2 将样例服务部署到 Swarm 集群

Swarm 中运行的任务称为服务。1 个服务可以有多个容器实例，可部署到 1 个或多个节点上。Swarm 中服务的部署模式分为两种，一种是 replicated，另一种是 global。replicated 是默认的部署模式，在这种模式下，服务可以运行多个副本，但是不能保证服务在每个节点都有容器实例在运行，而在 global 模式下，集群中的所有节点都会部署 1 个服务实例。新加入集群的节点也会由 Swarm 自动启动 1 个服务实例。通常需要以 global 模式部署的服务，包括日志收集、监控系统及其他需要在每个节点都运行的服务。

下面演示如何在 Swarm 集群部署服务。登录到已经搭建好的集群中的第 1 台服务器，也就是 manager 节点，然后创建一个简单的服务，命令如下：

```
docker service create --replicas 1 --name helloworld alpine ping docker.com
```

命令会生成一个 alpine 镜像的容器，在容器中运行 ping 命令。--replicas 参数用来设置同时运行多少个容器实例，--replicas 1 表示只运行一个容器实例。运行命令后输出的信息如下：

```
ew7ri8clfc34fmtrhq00t2o9h
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
```

如果输出信息，则表示服务创建成功了。

下面查看 Docker 中的容器信息，命令如下：

```
docker ps
```

运行命令后输出的信息如下：

```
CONTAINER ID   IMAGE            COMMAND                CREATED        STATUS
fe9661fe7f93   alpine:latest    "ping docker.com"   8 seconds ago   Up 7 seconds
```

还可以用 docker service 命令查看 Swarm 集群中的服务列表，命令如下：

```
docker service ls
```

运行命令后输出的信息如下：

```
ID              NAME          MODE          REPLICAS     IMAGE          PORTS
ew7ri8clfc34    helloworld    replicated    1/1          alpine:latest
```

输出信息中的 REPLICAS 列显示的 1/1 表示服务一共有 1 个容器在运行，期望运行的实例数是 1。

查看服务的详细信息，命令如下：

```
docker service inspect --pretty helloworld
```

运行命令后输出的信息如下：

```
ID:           ew7ri8clfc34fmtrhq00t2o9h
Name:         helloworld
Service Mode:Replicated
 Replicas:  1
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:
    alpine:latest@sha256:a75afd8b57e7f34e4dad8d65e2c7ba2e1975c795ce1ee22fa34f8cf46f96a3be
 Args:              ping docker.com
 Init:              false
Resources:
Endpoint Mode:      vip
```

### 5.3.3 伸缩样例服务

Swarm 中的服务可以很方便地进行伸缩，首先登录到 manager 节点，把 helloworld 服务扩展到 2 个容器实例，命令如下：

```
docker service scale helloworld=2
```

运行命令后输出的信息如下：

```
helloworld scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

表示 helloworld 服务扩展成功，接着查看 helloworld 服务的容器实例的信息，命令如下：

```
docker service ps helloworld
```

运行命令后输出的信息如下(前 5 列)：

```
ID                  NAME                IMAGE               NODE                DESIRED STATE
84361iq8n17e        helloworld.1        alpine:latest       ip-172-31-30-250    Running
7bjpfhxppb1q        helloworld.2        alpine:latest       ip-172-31-32-49     Running
```

后 3 列信息如下：

```
CURRENT STATE             ERROR     PORTS
Running 2 minutes ago
Running 30 seconds ago
```

从输出信息中的 NODE 列可以看出 Swarm 把 helloworld 服务的一个容器实例调度到了一个 worker 节点(ip-172-31-32-49)。把另一个实例调度到了 manager 节点(ip-172-31-

30-250)。

实验结束,删除 helloworld 服务,命令如下:

```
docker service rm helloworld
```

### 5.3.4 更新样例服务

这一节演示如何对 Swarm 中的服务升级,首先部署 3 个使用 redis:3.0.6 镜像的容器实例,然后升级到 redis:3.0.7 镜像。Swarm 具有支持滚动更新的特性,也就是按照指定的并发度,依次对节点中服务的容器实例进行更新。

下面创建服务 redis,命令如下:

```
docker service create \
    -- replicas 3 \
    -- name redis \
    -- update - delay 10s \
   redis:3.0.6
```

命令中的--replicas 3 表示运行 3 个容器实例,--update-delay 10s 表示依次更新每个容器的时间间隔为 10s。时间单位可以是小时、分钟和秒,分别用 h、m、s 表示。可以搭配起来使用,例如 1h5m30s 表示 1 小时 5 分钟 30 秒。Swarm 调度器默认在同一时刻只会更新 1 个容器实例,如果想改变同一时刻更新容器的数量,则需要使用参数--update-parallelism。例如指定--update-parallelism 3 就会同时对 3 个容器实例进行更新。

运行命令后输出的信息如下:

```
5rf7xzeu2rmw2e8j65mk4spm7
overall progress: 3 out of 3 tasks
1/3: running   [ ================================================>]
2/3: running   [ ================================================>]
3/3: running   [ ================================================>]
verify: Service converged
```

输出信息表示已经成功运行了 3 个容器,查看 Swarm 中的服务列表,命令如下:

```
docker service ls
```

运行命令后输出的信息如下:

```
ID            NAME      MODE         REPLICAS  IMAGE         PORTS
5rf7xzeu2rmw  redis     replicated   3/3       redis: 3.0.6
```

进一步查看运行 redis 服务的容器信息和所在节点,命令如下:

```
docker service ps redis
```

运行命令后输出的信息如下(前 4 列)：

```
ID              NAME      IMAGE          NODE                    DESIRED STATE
z78tjmschmpa    redis.1   redis:3.0.6    ip-172-31-35-212        Running
0ftx5ta6mq7t    redis.2   redis:3.0.6    ip-172-31-32-49         Running
etui5zcr9xt9    redis.3   redis:3.0.6    ip-172-31-30-250        Running
```

后 3 列信息如下：

```
CURRENT     STATE       ERROR       PORTS
Running     49          seconds     ago
Running     48          seconds     ago
Running     48          seconds     ago
```

从 NODE 列可以看出 3 个容器实例运行在 3 个不同的节点上。

查看 redis 服务的详细信息，命令如下：

```
docker service inspect --pretty redis
```

--pretty 参数表示不使用 JSON 格式输出。运行命令后输出的信息如下：

```
ID:     5rf7xzeu2rmw2e8j65mk4spm7
Name:     redis
Service Mode:Replicated
 Replicas:  3
Placement:
UpdateConfig:
 Parallelism:  1
 Delay:     10s
 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:
    redis:3.0.6@sha256:6a692a76c2081888b589e26e6ec835743119fe453d67ecf03df7de5b73d69842
```

```
Init:            false
Resources:
Endpoint Mode:    vip
```

在 UpdateConfig 条目下有一项 Parallelism:1,表示更新服务的时候最多只能同时更新一个容器实例。也就是更新操作的并行度,可以想象,这个值越大,更新全部服务所用的时间就会越短,但是为了系统的稳定性,通常会选择增量更新。

下面演示把 redis 服务的镜像升级到 3.0.7,同样需要在 manager 节点上操作。命令如下:

```
docker service update --image redis:3.0.7 redis
```

运行命令后输出的信息如下:

```
redis
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
```

重新查看 redis 服务的详细信息,命令如下:

```
docker service inspect --pretty redis
```

运行命令后输出的信息如下:

```
ID:     5rf7xzeu2rmw2e8j65mk4spm7
Name:     redis
Service Mode:  Replicated
 Replicas:  3
UpdateStatus:
 State:     updating
 Started:  55 seconds ago
 Message:  update in progress
Placement:
UpdateConfig:
 Parallelism:  1
 Delay:     10s
 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:
    redis:3.0.7@sha256:730b765df9fe96af414da64a2b67f3a5f70b8fd13a31e5096fee4807ed802e20
 Init:    false
Resources:
Endpoint Mode:  vip
```

### 5.3.5 维护 Swarm 节点

当 Swarm 集群中的节点需要维护时，需要把节点上运行的任务调度到正常工作的节点，在把节点设置为维护状态时，Swarm 会自动完成任务调度。

下面演示维护节点时 Swarm 自动调度任务的过程。

首先登录到 manager 节点，创建一个服务，命令如下：

```
docker service create --replicas 3 --name redis --update-delay 10s redis:3.0.6
```

运行命令，以便创建 3 个运行 redis:3.0.6 镜像的容器作为 redis 服务。

查看运行 redis 服务的节点信息，命令如下：

```
docker service ps redis
```

运行命令后输出的信息如下(前 5 列)：

```
ID             NAME      IMAGE          NODE                  DESIRED STATE
hjrzup5y7gxj   redis.1   redis:3.0.6    ip-172-31-32-49       Running
xegd5ktg3yo5   redis.2   redis:3.0.6    ip-172-31-30-250      Running
se1tz60os1u1   redis.3   redis:3.0.6    ip-172-31-35-212      Running
```

后 3 列信息如下：

```
CURRENT STATE             ERROR     PORTS
Running 39 seconds ago
Running 39 seconds ago
Running 39 seconds ago
```

从输出信息中的 NODE 列可以看出 3 个容器实例运行在 3 个不同的节点上。

下面把节点 ip-172-31-32-49 设置成维护状态，命令如下：

```
docker node update --availability drain ip-172-31-32-49
```

命令的--availability drain 表示把节点 ip-172-31-32-49 设置为 drain，也就是维护状态。drain 在英文中是抽干、排干的意思，设置成 drain 状态的节点上的容器实例会被 Swarm 结束运行，然后调度到其他正常节点上继续运行。相当于把节点上的容器抽出去，就像把泳池中的水排走一样。

现在再次查看运行 redis 服务的节点信息，命令如下：

```
docker service ps redis
```

运行命令后输出的信息如下(前 5 列)：

```
ID          NAME   IMAGE   NODE     DESIRED STATE
eo1jkk73u2q9   redis.1       redis:3.0.6   ip-172-31-30-250   Running
hjrzup5y7gxj    \_ redis.1   redis:3.0.6   ip-172-31-32-49    Shutdown
xegd5ktg3yo5   redis.2       redis:3.0.6   ip-172-31-30-250   Running
se1tz60os1u1   redis.3      redis:3.0.6   ip-172-31-35-212   Running
```

后 3 列信息如下：

```
CURRENT STATE                ERROR     PORTS
Running 1 second ago
Shutdown 2 seconds ago
Running about a minute ago
Running about a minute ago
```

输出信息中显示节点 ip-172-31-32-49 运行 redis 服务的容器实例被关闭了，Swarm 把这个关闭的任务调度到了节点 ip-172-31-30-250 上继续运行，并且保持了原来的实例名字 redis.1。这样在节点 ip-172-31-30-250 上就有了 2 个运行 redis 服务的容器实例。

可以查看一个节点 ip-172-31-30-250 上的 Docker 容器的进程，命令如下：

```
docker ps
```

运行命令后输出的信息如下(部分截取)：

```
CONTAINER ID   IMAGE          COMMAND                  CREATED
7e3db2f9b28e   redis:3.0.6   "/entrypoint.sh redi..."About a minute
agod3e4344a4aaf   redis:3.0.6   "/entrypoint.sh redi..."2 minutes ago
```

输出信息显示有 2 个 redis:3.0.6 镜像的容器。

现在节点 ip-172-31-32-49 处于不可用于调度的状态，再次查看节点信息，命令如下：

```
docker node ls
```

运行命令后输出的信息如下(前 4 列):

```
ID                            HOSTNAME        STATUS    AVAILABILITY
r29f4jyze4zlaydw5gqvljb2w *   ip-172-31-30-250   Ready     Active
6zxgzwnpaurtlcd874qozn18l     ip-172-31-32-49    Ready     Drain
a4odmh76r7983qskgix939os9     ip-172-31-35-212   Ready     Active
```

后 2 列信息如下:

```
MANAGER STATUS   ENGINE VERSION
Leader            20.10.5
                  20.10.5
                  20.10.5
```

可以看到节点 ip-172-31-32-49 对应的 AVAILABILITY 列显示的状态为 Drain。
单独查看节点 ip-172-31-32-49 的具体信息,命令如下:

```
docker node inspect --pretty ip-172-31-32-49
```

运行命令后输出的信息如下(部分截取):

```
ID:                     6zxgzwnpaurtlcd874qozn18l
Hostname:               ip-172-31-32-49
Joined at:              2021-03-10 02:27:10.221061622 +0000 utc
Status:
 State:                      Ready
 Availability:          Drain
 Address:               172.31.32.49
```

同样可以看到节点 ip-172-31-32-49 的 Availability 属性是 Drain。
节点维护结束后,再把它恢复为正常状态。命令如下:

```
docker node update --availability active ip-172-31-32-49
```

恢复为 active,也就是活跃状态后,重新查看节点 ip-172-31-32-49 的信息,命令如下:

```
docker node inspect --pretty ip-172-31-32-49
```

运行命令后输出的信息如下(部分截取):

```
ID:                     6zxgzwnpaurtlcd874qozn18l
Hostname:               ip-172-31-32-49
Joined at:              2021-03-10 02:27:10.221061622 +0000 utc
```

```
Status:
 State:            Ready
 Availability:           Active
 Address:    172.31.32.49
```

可以看到节点 ip-172-31-32-49 的 Availability 属性恢复成了 Active。这时 Swarm 又可以调度任务到它上面运行了。

### 5.3.6 Swarm 路由网格

Swarm 集群中的服务一般会部署到多个节点上运行，当 Swarm 中的服务通过某个网络端口对外提供服务时，Swarm 的 Routing Mesh（路由网格）功能使集群中任何一个节点都能在服务暴露的端口上接受请求，包括没有运行服务实例的节点，也可以在服务暴露的端口接受请求。Swarm 路由网格会把节点收到的请求自动转发到运行服务的容器实例中。这将有利于提高服务的可用性，例如当某个节点上运行服务的容器实例意外中止后，这个节点上收到的请求会被转发到正常的节点上进行处理。

启用 Swarm 路由网格功能需要使 Swarm 集群中的所有节点都开放以下端口：

TCP 或 UDP 协议的 7946 端口用于查找网络中的容器，称为容器网络发现，UDP 协议的 4789 端口用于容器入口网络。

在保证节点间可以互相访问上述端口的情况下，创建 Swarm 集群，这样才能开启 Swarm 的路由网格功能。这需要设置节点的防火墙或节点使用的网络安全组中的入站策略。

下面演示 Swarm 路由网格的功能，首先登录到 Swarm 集群中的 manager 节点。在 manager 节点创建一个名为 nginx 的服务，命令如下：

```
docker service create \
   -- name nginx \
   -- publish published = 8080,target = 80 \
   -- replicas 2 \
  nginx
```

运行命令后输出的信息如下：

```
zljsb4htvze5vm7mgj5nb2zfc
overall progress: 2 out of 2 tasks
1/2: running   [ ================================================== >]
2/2: running   [ ================================================== >]
verify: Service converged
```

现在查看 nginx 服务的容器实例的节点分布情况，命令如下：

```
docker service ps nginx
```

运行命令后输出的信息如下(前 5 列):

```
ID              NAME       IMAGE          NODE                  DESIRED STATE
jhp6u085n5rh    nginx.1    nginx:latest   ip-172-31-30-250      Running
9hdrifxm9xf3    nginx.2    nginx:latest   ip-172-31-35-212      Running
```

后 3 列信息如下:

```
CURRENT STATE           ERROR     PORTS
Running 4 minutes ago
Running 4 minutes ago
```

输出信息表示 nginx 服务现在有两个容器实例,分别运行在节点 ip-172-31-30-250 和节点 ip-172-31-35-212 上。

在节点 ip-172-31-30-250 上验证服务是否正常,命令如下:

```
curl localhost:8080
```

运行命令后输出的信息如下:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to Nginx!</title>
...
```

查看目前集群中的所有节点,命令如下:

```
docker node ls
```

运行命令后输出的信息如下(前 4 列):

```
ID                            HOSTNAME        STATUS    AVAILABILITY
r29f4jyze4zlaydw5gqvljb2w *   ip-172-31-30-250   Ready     Active
6zxgzwnpaurtlcd874qozn18l     ip-172-31-32-49    Ready     Active
a4odmh76r7983qskgix939os9     ip-172-31-35-212   Ready     Active
```

表明 ip-172-31-32-49 节点上没有 nginx 服务的容器实例,登录到 ip-172-31-32-49 节点上查看 Docker 进程信息来验证一下,命令如下:

```
Ubuntu@ip-172-31-32-49:~$ docker ps
```

运行命令后输出的信息如下:

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS     PORTS     NAMES
```

输出信息中没有出现任何运行中的容器，而 Swarm 路由网格功能可以使集群中没有运行服务实例的节点也可以在服务暴露的端口接受请求。下面验证 ip-172-31-32-49 节点是否监听了 nginx 服务暴露的 8080 端口，命令如下：

```
sudo lsof -i:8080
```

这个命令会打印出所有监听 8080 端口的进程列表，输出的信息如下：

```
COMMAND PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
dockerd 577 root   20u  IPv6  69103       0t0  TCP *:http-alt (LISTEN)
```

可以看到 ip-172-31-32-49 节点虽然没有运行 nginx 的服务容器实例，但是依然监听了 nginx 服务暴露的 8080 端口。

尝试访问 ip-172-31-32-49 节点上的 8080 端口，命令如下：

```
curl localhost:8080
```

运行命令后输出的信息如下：

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to Nginx!</title>
...
```

证明 Swarm 的路由网络功能工作正常。

还会对请求做负载均衡，为了验证这一点，在 ip-172-31-32-49 节点上多次访问 nginx 服务，构造一个特殊的地址访问，命令如下：

```
curl localhost:8080?node=ip-172-31-32-49
```

接着登录到 ip-172-31-30-250，查看 nginx 服务对应的容器的日志，查看日志命令如下：

```
docker logs -f 1f00fef00d2e
```

其中 1f00fef00d2e 是运行 nginx 服务的容器 ID。容器 ID 可通过 docker ps 命令查询。在输出的日志底部会看到类似的请求日志：

```
10.0.0.12 - - [10/Mar/2021:06:37:41 +0000] "GET /?node=ip-172-31-32-49 HTTP/1.1"
200 612 "-""curl/7.68.0""-"
```

在 ip-172-31-35-212 节点上进行同样的操作，同样会看到类似的请求日志。说明 Swarm 路由网络具有负载均衡的功能。

在节点 ip-172-31-30-250 上查看 nginx 服务日志还可以使用命令 docker service logs nginx。这是因为该节点是 manager 节点。

## 5.3.7 开发环境 Swarm 部署

### 1. 准备 Swarm 环境

与 Docker Compose 类似，Swarm 也支持在描述应用的清单文件中读取容器的所有配置，包括镜像、端口、存储和网络等，并且文件格式和 Compose 是兼容的。在 Swarm 中称这个文件为 stack 文件。

这一节演示把用户认证项目部署到本地开发环境中的 Swarm。本地运行单节点的 Swarm 集群。Docker Desktop 默认只支持单节点的 Swarm 集群。如果需要在本地运行多节点的 Swarm，则需要搭配使用 Docker Machine。

在使用 Swarm 之前，需要让 Docker Engine 以 Swarm 模式运行。需运行的命令如下：

```
docker swarm init
```

### 2. 定义 stack 文件

下面是 accounts 项目根目录下 docker-compose.yaml 文件中的内容：

```
version: "3.9"

services:
  web:
    image: ${ACCOUNT_FRONTEND_IMAGE}
    depends_on:
      - api
    ports:
      - "8088:80"

api:
    depends_on:
      - mysql
    image: ${ACCOUNT_IMAGE}
    ports:
      - "8081:80"
    volumes:
      - ./config:/config
command: go run src/server/main.go
    restart: always

  MySQL:
```

```
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DB}
    command: --character-set-server=utf8mb4 --default-time-zone=+08:00

  db-migration:
    image: goose
    depends_on:
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
    command: goose up
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:${MYSQL_PASSWORD}@tcp(mysql:3306)/${MYSQL_DB}
volumes:
  mysql:
```

把 docker-compose.yaml 复制为 stack.yaml 文件，下面会用 stack.yaml 文件作为 Swarm stack 文件。

Swarm 不支持 restart，所以需要修改 stack.yaml 文件，把 restart:always 删除掉。

Swarm 中的服务在退出后会被自动重启，而 db-migration 服务在完成数据库迁移后会自动退出，不需要重启。修改 db-migration 服务的重启策略，代码如下：

```
db-migration:
    deploy:
      restart_policy:
        condition: none
```

其他保持不变，添加了 deploy 字段。用于指定与服务部署相关的配置。deploy 字段下的 restart_policy 用于指定服务的重启策略。restart_policy 下的 condition 用于指定触发重启的条件。默认值是 any，也就是任何情况下，只要应用中止运行都会被重启。这里将 condition 指定为 none，意思是任何情况下都不会自动重启。

**3. 部署应用**

现在就可以用 Swarm 来部署应用 accounts 了，在后端项目的根目录下运行的命令如下：

```
docker stack deploy --compose-file stack.yaml accounts
```

可以看到以下输出信息：

```
Ignoring unsupported options: restart

Creating network accounts_default
Creating service accounts_mysql
Creating service accounts_web
Creating service accounts_api
```

这个 deploy 子命令拥有别名 up，也就是说可以把上面命令中的 docker stack deploy 换成 docker stack up。命令中--compose-file 用于指定 Compose 文件的位置。最后一个参数 accounts 是为应用所包括的所有容器的集合起的名字。

如何理解 docker stack 命令名字的含义呢？英文 stack 的意思是一堆，引申含义是大量、许多。应用包含了多个服务，所以把多服务组成的整体称为 stack。

**4. 查看服务状态**

确认应用包含的容器已经在正常运行，命令如下：

```
docker stack services accounts
```

运行命令后输出的信息如下(前 4 列)：

```
ID                  NAME                      MODE              REPLICAS
2kxwbk7xo6lk   accounts_api                   replicated        1/1
fdyzh2ojm1h6   accounts_db-migration          replicated        0/1
s9lg0nf74b9r   accounts_mysql                 replicated        1/1
237i5hbprlz7   accounts_web                   replicated        1/1
```

后 2 列信息如下：

```
IMAGE                          PORTS
accounts:dev                   *:8081->80/tcp
goose:latest
mysql:5.7
accounts-frontend:latest       *:8088->80/tcp
```

输出信息的 REPLICAS 字段显示 api、mysql 和 web 服务都已经正常运行。

**5. 关闭应用**

关闭整个应用可以使用的命令如下：

```
docker stack rm accounts-stack
```

运行命令后输出的信息如下：

```
Removing service accounts - stack_accounts
Removing service accounts - stack_mysql
Removing service accounts - stack_web
Removing network accounts - stack_default
```

### 5.3.8 生产环境 Swarm 部署

集群环境下，如果服务可以在任意节点运行将有利于提高服务的可用性。当运行服务的某个节点失败后，可以快速地在其他节点重新启动服务。另外因为这样的服务对节点不挑剔，所以调度成功的概率会更高。

5.3.7 节中的 docker-compose.yaml 文件中的 api 服务声明需要挂载本地的配置文件./config，代码如下：

```
api:
    volumes:
      - ./config:/config
```

依赖本地文件就需要在服务启动前在所有节点准备好依赖的文件，这样显然比较烦琐。Docker Swarm 支持把配置信息存储在 Docker configs 中，这样就不需要把配置文件挂载到容器中了。

Docker configs 一般用来存储非机密的配置信息，如果需要存储密码、API 密钥及证书等机密信息，最好使用 Docker secrets。

需要注意的是 Docker configs 只对 Swarm 中的服务生效，对于使用 docker run 命令创建的独立容器和 Docker Compose 生成的容器则无法使用 configs。

当添加新的 configs 时，configs 中的信息会被安全地发送到 Swarm 中的 manager 节点，然后存储在 Raft 日志中。Raft 日志会被复制到集群中的所有 manager 节点中。这样保证了 configs 数据的高可用性。

服务使用 configs 后，相应的 configs 数据会在运行服务的容器中以文件的形式出现。对于 Linux 环境下的 Docker 容器，这个文件的默认位置是根目录/<配置名字>下。

**1. 定义 stack 文件**

保留用于开发环境的 docker-compose.yaml 文件，新建 stack.yaml 文件。把 docker-compose.yaml 文件的内容复制到 stack.yaml 文件中，下面开始修改新创建的文件。

在新创建的文件 stack.yaml 中加入 configs 定义，代码如下：

```
configs:
  plain_config:
    file: ./config/plain.yaml
```

接着对 api 服务相应的部分进行修改，代码如下：

```
api:
    configs:
      - source: plain_config
        target: /config/plain.yaml
        mode: 0440
```

configs 的定义有两种语法，一种是长语法，另一种是短语法。这里使用的是长语法，长语法提供了更细粒度的参数。source:plain_config 表示配置来源是 plain_config，也就是引用全局定义的 plain_config。target:/config/plain.yaml 表示将配置文件挂载到容器中的/config/plain.yaml 文件。mode:0440 表示将容器中配置文件的读写权限设置为 0440。

这样配置文件./config/plain.yaml 中的内容就会通过 configs 的方式出现在 api 服务的容器中的/config/plain.yaml 文件。

配置文件 config/plain.yaml 中包含了数据库密码，可以把密码等机密信息放到单独的文件，以便分开管理。这样当将应用部署到 Swarm 中时，可以利用 Docker secret 来存取数据库密码。

下面把配置文件 config/plain.yaml 中的数据库密码放到单独的文件 config/secret.yaml 中，新建文件 secret.yaml，在文件中写入的内容如下：

```
database:
  password: 123
```

这时 plain.yaml 文件的内容如下：

```
server:
  port: 80
database:
  host: mysql
  port: 3306
  username: root
  schema: accounts
```

在 stack.yaml 中加入 configs 定义，代码如下：

```
secrets:
  secret_config:
    file: ./config/secret.yaml
```

接着在 api 服务中添加 secrets 定义，修改后代码如下：

```
api:
    secrets:
```

```
      - source: secret_config
        target: /config/secret.yaml
        mode: 0440
```

修改完成后的 stack.yaml 内容如下：

```
version: "3.9"

services:
  web:
    image: ${ACCOUNT_FRONTEND_IMAGE}
    depends_on:
      - api
    ports:
      - "8088:80"

  api:
    depends_on:
      - mysql
    image: ${ACCOUNT_IMAGE}
    ports:
      - "8081:80"
    configs:
      - source: plain_config
        target: /config/plain.yaml
        mode: 0440
    secrets:
      - source: secret_config
        target: /config/secret.yaml
        mode: 0440

  mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DB}
    command: --character-set-server=utf8mb4 --default-time-zone=+08:00

  db-migration:
    image: goose
    depends_on:
      - mysql
    working_dir: /migrations
```

```
    volumes:
      - ./src/database/migrations:/migrations
    command: goose up
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:${MYSQL_PASSWORD}@tcp(mysql)/${MYSQL_DB}
    deploy:
      restart_policy:
        condition: none

volumes:
  mysql:

configs:
  plain_config:
    file: ./config/plain.yaml

secrets:
  secret_config:
    file: ./config/secret.yaml
```

接下来登录到 manager 节点进行操作。

**2. 设置环境变量**

登录到 manager 节点以后，把 accounts 项目复制到当前工作目录下，因为 stack.yaml 中使用了环境变量，所以需要把.env 文件复制到 accounts 项目根目录下。

.env 文件内容如下：

```
ACCOUNT_FRONTEND_IMAGE=bitmyth/accounts-frontend:v1.0.0
ACCOUNT_IMAGE=bitmyth/accounts:v1.0.0
MYSQL_PASSWORD=123
MYSQL_DB=accounts
```

.env 文件中出现的镜像都必须从镜像仓库下载。

当使用 stack.yaml 文件把应用部署到 Swarm 时，Swarm 并不会像 Compose 一样自动读取.env 文件中配置的环境变量，需要手动设置用到的环境变量。命令如下：

```
export $(cat .env)
```

这样就可以使.env 文件中定义的全部环境变量在当前的 shell 进程中生效。

**3. 部署应用**

完成环境变量设置后，将应用部署到生产环境的 Swarm 集群中，命令如下：

```
docker stack deploy --compose-file stack.yaml accounts
```

运行命令后输出的内容如下：

```
Ignoring unsupported options: restart

Creating network accounts_default
Creating secret accounts_secret_config
Creating config accounts_plain_config
Creating service accounts_api
Creating service accounts_mysql
Creating service accounts_db-migration
Creating service accounts_web
```

输出信息表示所有的服务都已经开始创建了，并且创建了一个 config，名字叫 accounts_plain_config。另外创建了一个 secret，名字叫 accounts_secret_config。

用 docker config 命令可查看 config 列表，命令如下：

```
docker config ls
```

运行命令后输出的信息如下：

```
ID                          NAME                   CREATED          UPDATED
pte1p0uyrb7021wioxhrqnt64   accounts_plain_config  21 seconds ago   21 seconds ago
```

继续查看 accounts_plain_config 的详细内容，命令如下：

```
docker config inspect accounts_plain_config
```

运行命令后输出的信息如下：

```
[
    {
"ID": "pte1p0uyrb7021wioxhrqnt64",
"Version": {
"Index": 39458
        },
"CreatedAt": "2021-03-11T01:31:52.595487186Z",
"UpdatedAt": "2021-03-11T01:31:52.595487186Z",
"Spec": {
"Name": "accounts_plain_config",
"Labels": {
"com.docker.stack.namespace": "accounts"
            },
"Data": "c2VydmVyOgogIHBvcnQ6IDgwCmRhdGFiYXNlOgogIGhvc3Q6IG15c3FsCiAgcG9ydDogMzMwNgogIHVzZXJuYW1lOiByb290CiAgcGFzc3dvcmQ6IDEyMwogIHNjaGVtYTogYWNjb3VudHMK"
        }
```

```
    }
]
```

输出信息中显示的 Data 字段就是配置的具体内容，这是 base64 编码后的结果。可以用命令 base64 来解码查看，命令如下：

```
base64 -d <(echo c2VydmVyOgogIHBvcnQ6IDgwCmRhdGFiYXNlOgogIGhvc3Q6IG15c3FsCiAgcG9ydDogMzMw
NgogIHVzZXJuYW1lOiByb290CiAgcGFzc3dvcmQ6IDEyMwogIHNjaGVtYTogYWNjb3VudHMK)
```

运行命令后输出的解码的内容如下：

```
server:
  port: 80
database:
  host: mysql
  port: 3306
  username: root
  password: 123
  schema: accounts
```

对比这个输出内容和./config/plain.yaml 文件，发现它们是一致的。

下面查看 secret 列表以确认 secret 是否创建成功，命令如下：

```
docker secret ls
```

运行命令后输出的信息如下：

```
ID                          NAME                DRIVER    CREATED     UPDATED
sicei6vjehoh8hj4b64ia2esi   accounts_secret_config   2 minutes ago   2 minutes ago
```

继续查看 accounts_secret_config 的详细内容，命令如下：

```
docker inspect accounts_secret_config
```

运行命令后输出的信息如下：

```
[
    {
"ID": "sicei6vjehoh8hj4b64ia2esi",
"Version": {
"Index": 74589
        },
"CreatedAt": "2021-03-11T07:02:22.774283061Z",
"UpdatedAt": "2021-03-11T07:02:22.774283061Z",
```

```
    "Spec": {
    "Name": "accounts_secret_config",
    "Labels": {
    "com.docker.stack.namespace": "accounts"
                }
            }
        }
    ]
```

查看 Swarm 中的服务列表，命令如下：

```
docker service ls
```

运行命令后输出的内容如下（前 4 列）：

```
ID                  NAME                       MODE          REPLICAS
9iqolp4cb3jj   accounts_api               replicated         1/1
rwptwtkorrhl   accounts_db-migration      replicated         0/1
0v6re1vdchf4   accounts_mysql             replicated         1/1
sx7b95d9au5q   accounts_web               replicated         1/1
```

后两列内容如下：

```
IMAGE                              PORTS
bitmyth/accounts:v1.0.0            *:8081->80/tcp
goose:latest
mysql:5.7
bitmyth/accounts-frontend:v1.0.0   *:8088->80/tcp
```

**4. 查看服务状态**

accounts_api 服务启动后，可以请求它所提供的注册接口，以此来验证是否工作正常，命令如下：

```
curl -v   localhost:8081/v1/register -d'{"name":"test"}'
```

如果服务正常运行，则会收到类似响应，响应内容如下：

```
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOjQsInVzZXIiOnRydWUsImV4cCI6MTYxNT
U2MjA5NywiaWF0IjoxNjE1Mzg5Mjk3LCJpc3MiOiJCaWthc2gifQ.USttmL2zRAzlLzNb30dBu7txcWn5NLHzMh88
d4-ybBc","user":{"ID":4,"Name":"test","Password":"","Email":"","Phone":"","Avatar":"",
"CreatedAt":"2021-03-10T15:14:57.870701764Z","UpdatedAt":"2021-03-10T15:14:57.870701764Z",
"DeletedAt":null}}
```

accounts_api 服务是无状态的，所以可以很方便地把 accounts_api 服务实例扩展到

3 个，命令如下：

```
docker service scale accounts_api = 3
```

运行命令后输出的内容如下：

```
accounts_api scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [ ================================================>]
2/3: running   [ ================================================>]
3/3: running   [ ================================================>]
verify: Service converged
```

以上内容表明全部启动成功，说明 Docker configs 可以让不同的节点共享。
查看 accounts_api 服务的容器实例分布在哪些节点，命令如下：

```
docker service ps accounts_api
```

运行命令后输出的内容如下(前 4 列)：

```
ID              NAME              IMAGE                        NODE
jd0p13e9s39r    accounts_api.1    bitmyth/accounts:v1.0.0    ip-172-31-30-250
7zy6mq1vfkrz    accounts_api.2    bitmyth/accounts:v1.0.0    ip-172-31-35-212
tdf2qiithyz4    accounts_api.3    bitmyth/accounts:v1.0.0    ip-172-31-32-49
```

后 4 列内容如下：

```
DESIRED STATE   CURRENT STATE              ERROR     PORTS
Running          Running 9 minutes ago
Running          Running 4 minutes ago
Running          Running 4 minutes ago
```

**5. 关闭应用**

关闭整个应用，命令如下：

```
docker stack rm accounts
```

运行命令后输出的信息如下：

```
Removing service accounts_api
Removing service accounts_db-migration
Removing service accounts_mysql
Removing service accounts_web
Removing secret accounts_secret_config
Removing config accounts_plain_config
Removing network accounts_default
```

### 5.3.9 约束服务调度

在生产环境中 Swarm 集群通常有多个节点。在调度下，服务会在多节点间转移。这样依赖本地数据卷的服务在调度到其他节点继续运行时，就会面临数据丢失的情况。例如 MySQL 服务，在前面定义的 stack.yaml 文件中，MySQL 使用了一个本地数据卷。MySQL 的数据会保存到这个数据卷中。如果 MySQL 被调度到了其他节点，则在其他节点上就无法找到之前保存的数据。

这个问题可以通过指定 MySQL 服务运行在固定的节点来解决。也就是在调度过程中限制 MySQL 服务，使它只能调度到满足要求的节点。这个功能在 Swarm 中称为放置约束(Placement Constraints)。顾名思义就是对把服务放到哪个节点运行进行约束。

放置约束条件通过对节点属性进行筛选实现。支持通过匹配节点的标签(node.labels)实现，也可以通过匹配节点的角色(node.role)、节点 ID(node.id)及节点的主机名(node.hostname)等实现。

如果使用标签实现放置约束，则需要定义标签并为节点设置标签，然后在服务定义中加上需要匹配的节点标签作为限制条件，这样服务就会调度到和标签匹配的节点上。

只有节点拥有的标签和服务放置约束中的指定的标签匹配。服务才会被调度成功。指定了放置限制的服务，如果在调度时没有节点拥有匹配的标签，则服务将无法调度，会一直处于待处理状态。如果节点无法满足服务的其他放置约束条件，则服务同样无法调度。

Swarm 集群中节点的标签格式为 key=value。可以为节点指定一个或多个标签。

下面使用标签作为 MySQL 服务的放置约束。假设 MySQL 服务可以运行的节点需要有标签 role=db，修改 stack.yaml 文件中 MySQL 服务的定义，代码如下：

```
mysql:
    image: mysql:5.7
    deploy:
      placement:
        constraints:
          - "node.labels.role == db"
```

image 字段下添加的 deploy 字段用于指定服务部署相关的配置。deploy 字段下的 placement 用于指定服务放置配置。placement 下的 constraints 用于指定放置的限制条件，这里指定了标签匹配条件。约束条件为 node.labels.role==db，表示部署到拥有 role=db 标签的节点。注意条件中的操作符是 2 个等号“==”。约束条件中用“==”表示匹配，用“!=”表示排除。

同样地，db-migration 服务因为挂载了本地的数据迁移文件，也需要在固定的节点执行，所以要为它指定放置约束。修改后的 db-migration 服务的定义如下：

```
db-migration:
  image: goose
  depends_on:
    - mysql
  working_dir: /migrations
  volumes:
    - ./src/database/migrations:/migrations
  command: goose up
  environment:
    GOOSE_DRIVER: mysql
    GOOSE_DBSTRING: root:${MYSQL_PASSWORD}@tcp(mysql)/${MYSQL_DB}
  deploy:
    restart_policy:
      condition: none
    placement:
      constraints:
        - "node.labels.role==db"
```

假设希望 MySQL 运行的节点是 ip-172-31-30-250，首先为节点 ip-172-31-30-250 设置标签 role=db，命令如下：

```
docker node update --label-add role=db ip-172-31-30-250
```

验证标签设置是否生效，命令如下：

```
docker node inspect ip-172-31-30-250
```

在运行命令后输出的信息中会看到 Spec 字段，字段如下：

```
"Spec": {
"Labels": {
"role": "db"
            },
```

表示节点 ip-172-31-30-250 已经拥有了标签 role=db。

完成修改后，重新部署应用，命令如下：

```
docker stack deploy --compose-file stack.yaml accounts
```

运行命令后等待部署结束，然后查看 MySQL 服务是否被调度到节点 ip-172-31-30-250 运行，命令如下：

```
docker service ps accounts_mysql
```

运行命令后输出信息中的 NODE 列的内容如下：

```
NODE
ip-172-31-30-250
```

说明放置约束生效了。MySQL 服务运行在包含 role=db 标签的节点 ip-172-31-30-250 上。

## 5.3.10 日志收集

### 1. 简介

日志对于观测应用运行情况，定位和修复故障及系统运行状态监控等起着重要作用。Swarm 中运行的服务如果有多个容器实例，则查看服务所产生的日志可以使用 docker service logs 命令。它会把服务中所有容器的日志集中起来显示，但是使用 docker service logs 命令查看日志的方式在搜索时不方便，并且需要登录到服务器上操作。

常用的解决方案是用专门的收集工具把容器日志转发到搜索引擎。下面介绍如何使用 EFK(Elasticsearch+Fluentd+Kibana)工具栈实现日志收集和处理。

### 2. Fluentd

Fluentd 是开源的日志数据收集工具，使用简单并且性能优异。支持对日志进行自定义格式处理和转发到自定义的日志处理服务。Fluentd 对 Docker 的支持很好，是 Docker 内置的日志驱动之一。Fluentd 是 CNCF(云原生计算基金会)已完成项目之一。

Fluentd 同时具有良好的扩展性和灵活性，支持使用自定义插件来满足个性化需求。

EFK 技术栈中 Fluentd 需要把收集的日志转发到 Elasticsearch，所以要为 Fluentd 安装 Elasticsearch 插件，而 Fluentd 官方的镜像没有安装 Elasticsearch 插件，需要基于官方镜像构建自定义的镜像。下面开始构建 Fluentd 镜像，所需的 Dockerfile 如下：

```
FROM fluent/fluentd:v1.12.0-debian-1.0

USER root

RUN ["gem", "install", "fluent-plugin-elasticsearch", "--no-document", "--version", "4.3.3"]

USER fluent
```

在这个 Dockerfile 同级目录下运行命令，以便构建镜像，命令如下：

```
docker build -t bitmyth/fluentd-es:v1.0.0 .
```

这样就拥有了安装了 Elasticsearch 插件的 Fluentd 镜像。接着把这个镜像上传到 Docker Hub。这样集群中的节点就可以下载这个镜像了，命令如下：

```
docker push bitmyth/fluentd-es:v1.0.0 .
```

使用 Fluentd 收集 Swarm 集群中的所有容器的日志，需要将 Fluentd 设置为 Swarm 全局服务。修改 stack.yaml 文件，增加 Fluentd 服务的定义，代码如下：

```
fluentd:
    image: bitmyth/fluentd-es:v1.0.0
    environment:
      FLUENTD_CONF: 'fluent.conf'
    configs:
      - source: fluentd_config
        target: /fluentd/etc/fluent.conf
    ports:
      - "24224:24224"
      - "24224:24224/udp"
    deploy:
      mode: global
```

下面介绍指令的含义。

environment 字段中定义了 FLUENTD_CONF 环境变量，把它的值设置为 fluent.conf。这个环境变量用于指定 Fluentd 读取配置的文件名，代码如下：

```
environment:
  FLUENTD_CONF: 'fluent.conf'
```

configs 字段中定义了 fluentd_conf，target：/fluentd/etc/fluent.conf 表示把配置 fluentd_config 挂载到容器中的/fluentd/etc/fluent.conf 文件。Fluentd 默认的配置文件夹是/fluentd/etc/，这样 Fluentd 就可以读取到自定义的 fluent.conf 文件中，代码如下：

```
configs:
  - source: fluentd_config
    target: /fluentd/etc/fluent.conf
```

公开 Fluentd 监听的 TCP 端口 24224 和 UDP 端口 24224，代码如下：

```
ports:
  - "24224:24224"
  - "24224:24224/udp"
```

服务将以 global 模式部署。也就是在所有节点都会启动一个 Fluentd 容器实例。这样 Fluentd 就可以收集到全部节点上运行的容器所产生的日志。

```
deploy:
  mode: global
```

上述 Fluentd 定义中用到的 fluentd_config 定义如下：

```
configs:
  fluentd_config:
    file: ./config/fluentd/fluent.conf
```

表明 fluentd_config 配置项的内容将从./config/fluentd/fluent.conf 文件中读取。Fluentd 默认将文件的编码配置为 UTF-8 或 ASCII。

./config/fluentd/fluent.conf 文件内容如下：

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<match *.**>
  @type copy

<store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix fluentd
    logstash_dateformat %Y%m%d
    include_tag_key true
    type_name access_log
    tag_key @log_name
    flush_interval 1s
</store>

<store>
    @type stdout
</store>
</match>
```

下面介绍配置文件中的指令含义。

定义日志的输入源，也就是数据的来源。Fluentd 配置文件中支持定义多个 source。每个 source 都必须指定@type 参数来表明使用什么输入插件。@type forward 表示处理输入的插件类型是转发。除了 forward，Fluentd 还支持 http 插件。http 插件会提供 HTTP 服务，接收以 HTTP 发送的数据，而 forward 提供 TCP 服务，接收以 TCP 数据包的形式发送的数据。当然这两个输入插件可以同时启用。如果内置的输入插件无法满足需求，Fluentd 支持开发自定义输入插件。

指令 port 24224 表示 Fluentd 通过 TCP 端口 24224 接收日志数据，bind 0.0.0.0 表示监听任意的网卡设备。

Fluentd 中接收的数据会被包装成一个事件。事件包含了 tag 属性、time 属性和 record 属性。tag 属性用于识别日志数据的来源。它是用英文句点“.”分隔的字符串。Fluentd 推荐在 tag 中使用的字符是小写的字母、数字及下画线，用正则表达式表示为^[a-z0-9_]+ $。事件的 time 属性是 UNIX 时间格式，由输入插件指定它的值。事件的 record 属性是一个 JSON 对象，包含了实际输入的数据，代码如下：

```
<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>
```

指令 match 用于控制 Fluentd 如何匹配和输出数据。match *.** 表示匹配拥有 *.** 格式的 tag(标签)的事件。每个 match 指令都必须指定一个匹配模式和一个@type 参数。匹配模式用于匹配事件的标签。match 中的@type 用于指定使用什么输出插件。

这里的@type copy 指定了 copy 插件。它是 Fluentd 内置的一个标准输出插件。copy 会把事件复制到多个输出。

指令 match 下包含的 store 指定了存储输出的目的地。match 指令中至少需要指定一个 store，代码如下：

```
<match *.**>
  @type copy
```

store 指令下的@type elasticsearch 用于指定使用 elasticsearch 插件。host 用于指定 elasticsearch 的 IP 地址，因为 Swarm 中的服务可以通过 DNS 来解析它的 IP 地址，所以这里使用 elasticsearch 服务名作为 host 字段的值。port 9200 指定 elasticsearch 服务的 TCP 端口号为 9200。如果需要发送到多个 elasticsearch 服务，则可以通过 hosts 指定，例如：hosts:host1:port1,host2:port2。通过示例中这种格式指定了多个 elasticsearch 服务的情况下，指定单个 elasticsearch 服务的 host 和 port 字段会被忽略，代码如下：

```
<store>
    @type elasticsearch
    host elasticsearch
    port 9200
    logstash_format true
    logstash_prefix fluentd
    logstash_dateformat %Y%m%d
    include_tag_key true
    type_name access_log
```

```
    tag_key @log_name
    flush_interval 1s
</store>
```

采用与 Logstash 兼容的格式把数据写入索引，代码如下：

```
logstash_format true
```

将索引的前缀设置为 fluentd，代码如下：

```
logstash_prefix fluentd
```

将索引名字中的日期格式设置为%Y%m%d，例如 20210325，代码如下：

```
logstash_dateformat %Y%m%d
```

将 Fluentd 的 tag 加入 Elasticsearch 数据中，代码如下：

```
include_tag_key true
```

指定写入 Elasticsearch 中的文档类型的名字。type_name 参数在 Elasticsearch 7 中会设置为固定值_doc，而 Elasticsearch 8 会忽略这个参数，代码如下：

```
type_name access_log
```

指定写入 Elasticsearch 中的 Fluentd tag 的键名为@log_name，代码如下：

```
tag_key @log_name
```

指定将缓存数据写入目的地的时间间隔为 1s，代码如下：

```
flush_interval 1s
```

### 3. Elasticsearch

Elasticsearch 是非常流行的开源搜索引擎，在 EFK 技术栈中的角色是保存和索引 Fluentd 收集的日志数据。下面定义 Elasticsearch 服务，修改 stack.yaml 文件，代码如下：

```
elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.10.2
    environment:
      - "discovery.type=single-node"
    ports:
      - "9200:9200"
    deploy:
```

```
      resources:
        limits:
          cpus: '2'
          memory: 2G
        reservations:
          cpus: '1'
          memory: 1.5G
```

由于 Elasticsearch 服务需要使用比较多的内存，所以在上述定义中增加了对 CPU 和内存资源使用限制的声明。

限制 Elasticsearch 最多使用 2 个 CPU 和 2GB 的内存。在不限制容器使用内存的情况下，如果容器使用了过多的系统内存，则会被内核强制中止运行，代码如下：

```
limits:
  cpus: '2'
  memory: 2G
```

为 Elasticsearch 服务预留 1 个 CPU 和 1.5GB 内存。如果没有节点可以满足指定的计算资源需求，则服务将无法调度成功，代码如下：

```
reservations:
  cpus: '1'
  memory: 1.5G
```

#### 4. Kibana

Kibana 为使用 Elasticsearch 提供了友好的 UI 界面，降低了使用 Elasticsearch 的难度。方便用户查询 Elasticsearch 中的数据。

下面定义 Kibana 服务，修改 stack.yaml 文件，代码如下：

```
kibana:
  image: kibana:7.10.1
  depends_on:
    - "elasticsearch"
  ports:
    - "5601:5601"
```

depends_on 声明的 Kibana 服务依赖于 Elasticsearch，当 Elasticsearch 启动后才会启动 Kibana。

#### 5. 使用 Fluentd 日志驱动

现在修改第 1 章介绍的用户认证应用中包含的服务所用的日志驱动。首先修改后端项目，也就是 api 服务，设置它的日志驱动，代码如下：

```
api:
    logging:
driver: "fluentd"
      options:
        fluentd-address: localhost:24224
        tag: api.{{.Name}}
```

其他部分保持不变，增加了 logging 设置。将日志驱动设置为 fluentd，日志驱动选项通过 options 定义。options 下的 fluentd-address 定义了 fluentd 监听的网络地址。由于 Fluentd 是以 global 模式部署的，所以每个节点都可以通过 localhost 这个地址访问它。options 下的 tag 定义了日志的名字。{{.Name}}会被替换成容器的名字。最后 api.{{.Name}}会变成类似 api.accounts_api.1.gvq4cxpd99fdt7grxkr38zle4 这样的字符串。

同样地，可为前端项目（Web 服务）设置日志驱动，修改后的代码如下：

```
web:
    logging:
      driver: "fluentd"
      options:
        fluentd-address: localhost:24224
        tag: web.{{.Name}}
```

Web 服务定义的其他部分保持不变，增加了 logging 设置。

**6. 部署**

增加与日志相关的设置后，重新将应用部署到 Swarm 集群。首先设置环境变量，命令如下：

```
export $(cat .env.prod)
```

这样就可以使在.env 文件中定义的全部环境变量在当前的 shell 进程中生效。完成环境变量设置后，再将应用部署到生产环境的 Swarm 集群中，命令如下：

```
docker stack deploy --compose-file stack.yaml accounts
```

运行命令后输出的内容如下：

```
Creating network accounts_default
Creating secret accounts_secret_config
Creating config accounts_fluentd_config
Creating config accounts_plain_config
Creating service accounts_fluentd
Creating service accounts_elasticsearch
Creating service accounts_kibana
Creating service accounts_web
```

```
Creating service accounts_api
Creating service accounts_mysql
Creating service accounts_db-migration
```

查看 Fluentd 服务的状态，命令如下：

```
docker service ps accounts_fluentd
```

运行命令后输出信息的后 5 列如下：

```
NODE                DESIRED STATE   CURRENT STATE            ERROR     PORTS
ip-172-31-35-212    Running         Running 3 minutes ago
ip-172-31-32-49     Running         Running 3 minutes ago
ip-172-31-30-250    Running         Running 3 minutes ago
```

可以看到 Swarm 集群中所有的节点运行着 Fluentd 服务。

接下来打开浏览器访问 Kibana，设置要查看的索引的模式。默认首页如图 5-1 所示。

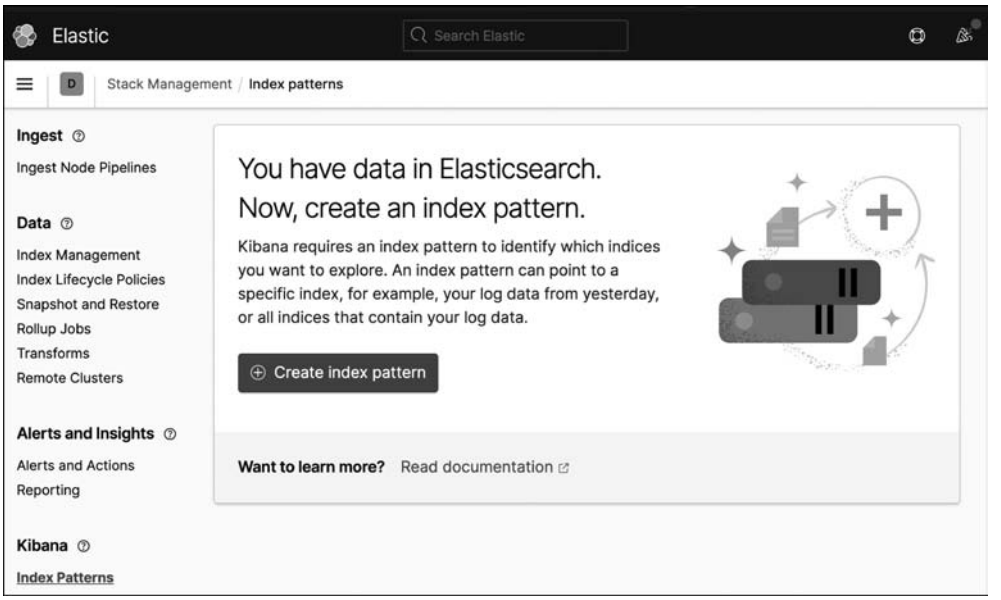


图 5-1 Kibana Index patterns 首页

单击页面中的 Create index pattern 按钮，会跳转到如图 5-2 所示页面。

在页面中 Index pattern name 输入框中输入 fluentd-*，如图 5-3 所示。

然后单击 Next step 按钮。在新页面中选择 Time field 下拉列表框中的@timestamp，之后单击 Create index pattern 按钮，如图 5-4 所示。

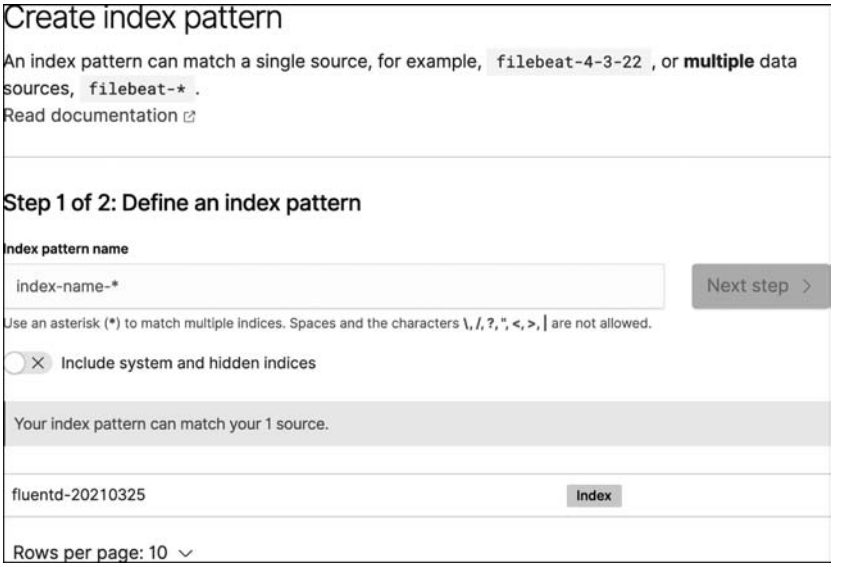


图 5-2 创建 Kibana index patterns 页

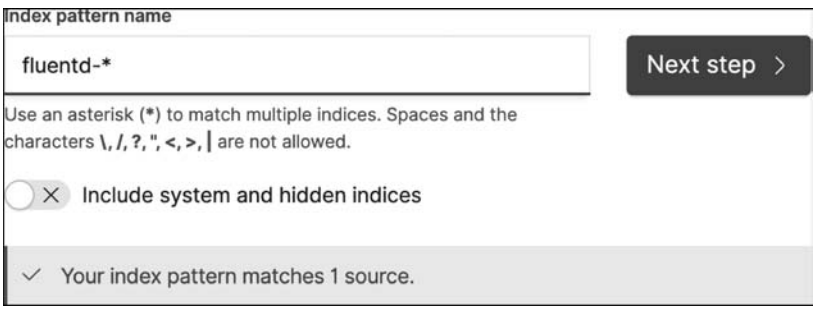


图 5-3 输入 Index pattern name

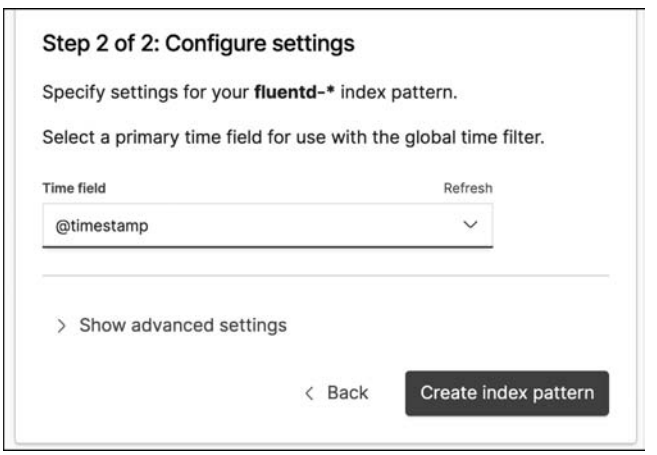


图 5-4 选择 Time field

接下来单击页面左上角菜单，在展开的菜单中选中 Discover，如图 5-5 所示。

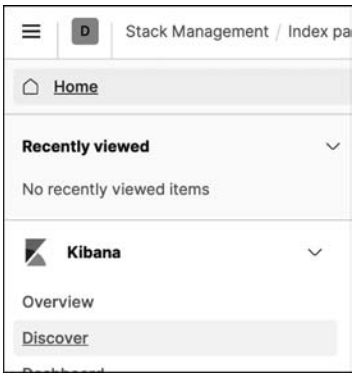


图 5-5　单击 Discover 链接

现在就可以在页面中观察日志了。访问前端项目，完成注册和登录，这样可以产生一些日志数据，然后回到 Kibana 中查看这些日志数据。Kibana 会展示最新的日志，如图 5-6 所示。

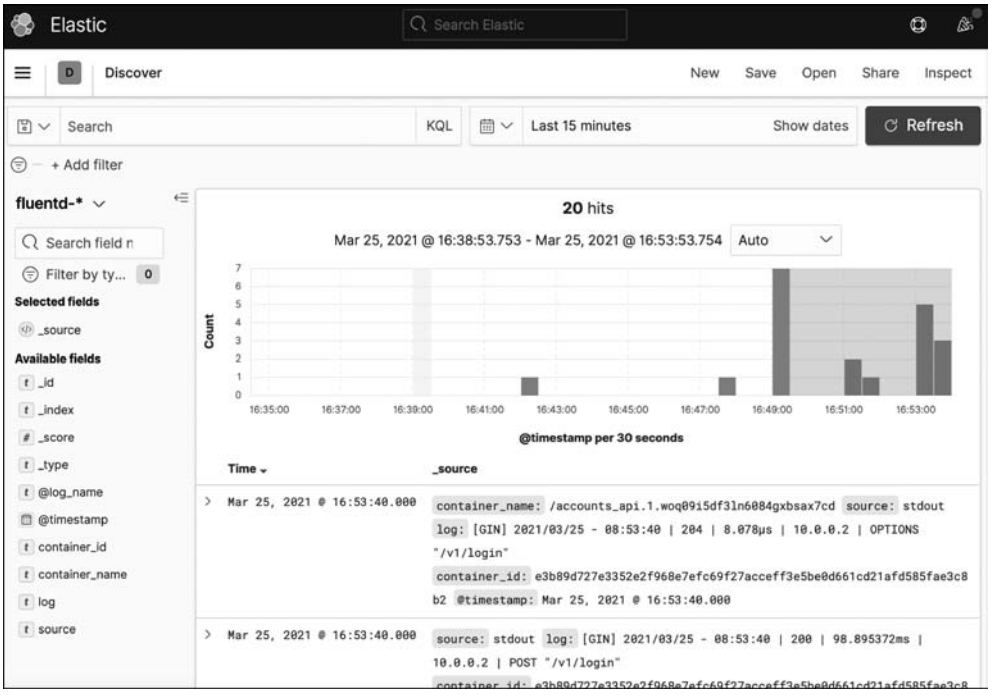


图 5-6　查看日志

在日志中可以看到前端项目和后端项目的日志。以前端项目产生的日志为例，如图 5-7 所示。

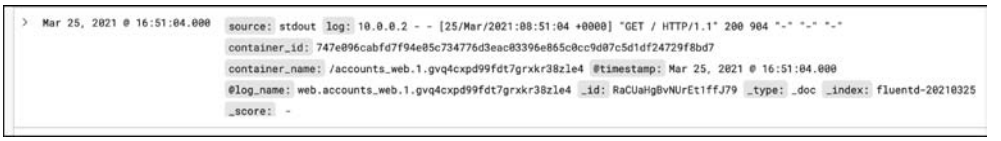


图 5-7　前端日志示例

日志记录的字段如表 5-1 所示。

**表 5-1　前端日志字段表**

| 字　段　名 | 字　段　值 |
|---|---|
| @log_name | web. accounts_web. 1. gvq4cxpd99fdt7grxkr38zle4 |
| @timestamp | Mar 25，2021 @ 16：51：04. 000 |
| _id | RaCUaHgBvNUrEt1ffJ79 |
| _index | fluentd-20210325 |
| _score | - |
| _type | _doc |
| container_id | 747e096cabfd7f94e05c734776d3eac03396e865c0cc9d07c5d1df24729f8bd7 |
| container_name | /accounts_web. 1. gvq4cxpd99fdt7grxkr38zle4 |
| log | 10. 0. 0. 2 - - [25/Mar/2021:08:51:04 +0000] "GET / HTTP/1. 1" 200 904 "-""-""-" |
| source | stdout |

可以从 container_name 字段判断出日志产生自哪个服务，通过 container_id 可以准确定位产生日志的容器。

**7. 总结**

基于 EFK 技术栈实现的日志收集系统可以实时地收集日志，并且可以方便地检索日志，以及定位故障，从而实现集群分布式环境下容器日志的统一处理。