Docker Compose 简明指南

Docker Compose是docker生态中一个常用的命令行工具，可用来编排（定义和运行等）由多个容器组成的应用。换句话说，由Compose管理的一个项目（project）是由一组关联的应用容器组成的完整的业务单元，它定义在docker-compose.yml 文件中，并由docker-compose命令读取和按需编排相应的业务容器单元。

提示：docker-compose命令的-p选项即用于指定相应project的名称，省略时默认为docker-compose.yml文件所在的目录名称。

编排和部署

编排(orchestration)

编排指根据被部署的对象之间的耦合关系，以及被部署对象对环境的依赖，制定部署流程中各个动作的执行顺序，部署过程所需要的依赖文件和被部署文件的存储位置和获取方式，以及如何验证部署成功。这些信息都会在编排工具中以指定的格式(比如配置文件或特定的代码)来要求运维人员定义并保存起来，从而保证这个流程能够随时在全新的环境中可靠有序地重现出来。

部署(deployment)

部署是指按照编排所指定的内容和流程，在目标机器上执行环境初始化，存放指定的依赖文件，运行指定的部署动作，最终按照编排中的规则来确认部署成功。

Compose文件

docker-compose命令默认于当前目录中搜索docker-compose.yaml配置文件，除了version以外，该文件主要由三个顶级参数组成：services、networks和volumes。

- serivces：用于定义组成每个服务的容器的相关配置，相关配置项类似于传递给docker container create命令的相关参数；每个服务启动的容器实例可能不止一个，而每个容器依赖的镜像要么通过二级指令image指定位置直接获取，要么由二级指令build指定的Dockerfile临时构建；另外，服务中可内嵌networks指令引用顶级指令networks中定义的网络，以及内嵌volumes引用顶级指令volumes中定义的存储卷。
- networks：用于定义容器网络，相关配置项类似于传递给docker network create命令的相关参数；
- volumes：用于定义容器存储卷，相关配置项类似于传递给docker volume create命令的相关参数；

Services的常用指令

build

用于定义构建镜像的相关配置，它自身可以直接以上下文路径为参数，例如“build ./dir”即为于当前目录下的dir子目录中搜索Dockerfile配置文件。不过，也可以使用二级参数context来完成此类功能。它常用的二级参数如下：

- context：包含Dockerfile的目录的路径，或者是git仓库的url；相对路径通常意味着以docker-compose.yaml文件所在的目录为起始目录；
- dockerfile：备用的Dockerfile文件，此指令依赖于context指令；
- args：构建参数，常用于向Dockerfile中由ARG指令定义的构建参数传值；
- cache_from：解析缓存时使用的镜像列表，由3.2版本引入；
- labels：向镜像添加元数据的标签，由3.3版本引入；
- target：基于多阶段镜像构建方式构建镜像时指定目标阶段，由3.4版本引入；

command和entrypoint

基于镜像启动容器时，command指定的命令覆盖镜像上默认的由CMD定义要运行的命令，而entrypoint指定的命令则覆盖镜像上由ENTRYPOINT定义要运行的命令，二者命令格式分别同Dockerfile的CMD和ENTRYPOINT。

cap_add和cap_drop

用于调整容器操作内核的权利与能力，cap_add用于添加容器运行时可用的能力，而cap_drop则相反。群集模式（swarm）会忽略此参数。

```
version: '3'
services:
  phpfpm_test:
    cap_add:
      - ALL
    cap_drop:
      - NET_ADMIN
      - SYS_ADMIN
```

depends_on

定义该服务的依赖关系；在启动该服务时，被依赖的每个服务必须处于运行状态（已启动而非就绪）；启动和关闭时过程以对称的方式进行，即启动时先启动被依赖的服务，而关闭时要先关闭依赖方。例如，对于下面的示例定义的服务列表，启动web时要先启动db和redis服务，而停止时需要先停止web服务。

```
version: "3.7"
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

deploy

定义服务的部署机制，不过，这仅在将服务部署到swarm集群时有效。下面是一个简单的配置示例。

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```


deploy支持多个二级参数，例如replicas用于定义容器实例的个数，update_config用于定义更新策略，而restart_policiy则用于指定重启策略。

dns和dns_search

dns指令用于自定义使用的DNS服务器，dns_search则用于自定义dns的搜索域。两个指令各自都分别支持单值和列表格式。以下示例中，dns指令使用了单值格式，而dns_search使用了列表格式。

```
dns: 8.8.8.8
dns_search:
  - dc1.example.com
  - dc2.example.com
```

environment和env_file

environment用于向容器添加环境变量，支持列表和字典格式。任何布尔值，包括true、false、yes和no，都需要使用引号，以确保YAML解析器不会将它们转换为True或False。以下示例使用了列表格式。

```
environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

而env_file则从指定的文件导入环境变量，路径为相对路径时表示起始于docker-compose.yaml文件所在目录。它同样支持单值或列表格式。下面的示例中使用了列表格式。

```
env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

需要注意的是，如果同时使用了environment和env_file，则从文件中加载的同名环境变量会被environment定义的环境变量值所覆盖。

expose和ports

expose指令仅暴露容器端口而不会将它们发布到主机，这意味着它们只能被链接的服务所访问，因此也

仅能指定内部端口。下面的指令以列表格式定义了要暴露的两个端口。

```
expose:
 - "3000"
 - "8000"
```

而ports指令则用于定义要暴露到主机外部的端口，它支持长短两种语法格式。整体语法格式如下

[host:][start_port[-end_port]:][con_start_port-]con_end_port

下面是一个短格式的完整使用示例，它给出了多种使用形式。

```
ports:
 - "3000"        # 仅容器端口，主机端口动态选定；
 - "3000-3005"   # 一组容器端口，相应的主机端口动态选定；
 - "8000:8000"   # 左侧为主机端口，右侧为容器端口；
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"   # 主机地址、端口和容器端口；
 - "127.0.0.1:5000-5010:5000-5010"
 - "6060:6060/udp"   # 主机端口、容器端口和协议；
```

而长格式的使用语法中，端口的相应各属性需要以专用参数给出。

- target：容器的端口
- published：暴露的端口
- protocol：端口协议（tcp或udp）
- mode：host用于在每个节点上发布主机端口，或者使用ingress用于负载平衡的群集模式端口；

下面的示例中实现了把容器的80/tcp端口暴露到主机的8080/tcp端口。

```
ports:
  - target: 80
    published: 8080
    protocol: tcp
    mode: host
```

注意：ports指令与“network:host“互斥。

image

指定用于启动容器的镜像文件，支持标识容器镜像的各种形式。

```
image: redis
image: ubuntu:14.04
image: tutum/influxdb
image: example-registry.com:4000/postgresql
image: a4bc65fd
```

networks

network指令用于定义要加入的网络，它需要以名称格式引用同名的顶级指令networks下定义的网络。下面的示例格式中的服务引用了some-network和other-network两个网络，这两个网络必须定义的顶级networks之下。

```
services:
  some-service:
    networks:
     - some-network
     - other-network
```

在服务中引用网络时还可以在加入的网络上给当前容器赋予一到多个别名（alias），随后，同一网络上的其它容器就可以使用服务名称或容器的别名连接到服务中的容器。需要注意的是，同一个容器在不同网络上可以使用不同的名称。下面是使用格式示例。

```
services:
  some-service:
    networks:
      some-network:
        aliases:
         - alias1
         - alias3
      other-network:
        aliases:
         - alias2
```


另外，也可以为容器在加入的网络中使用ipv4_addresses或ipv6_addresses指定使用的静态IP地址，不过这种功能要求在顶级networks配置段中必须定义了ipam驱动且指定的相关的CIDR格式的网络地址范围。下面是一个完整的使用示例。

```
version: "3.7"

services:
  app:
    image: nginx:alpine
    networks:
      app_net:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

networks:
  app_net:
    ipam:
      driver: default
      config:
        - subnet: "172.16.238.0/24"
        - subnet: "2001:3984:3989::/64"
```

network_mode

定义容器的网络模式，除了容器网络支持bridge、host、none和container:[container_name/id]四种网络模式之外，compose中还额外支持service:[service_name]模式。

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

需要注意的是，network_mode:host不能与links混用。

volumes

volumes指令用于定义容器的存储卷，它可以是同名的顶级指令volumes中定义的命名的存储卷，也可以直接使用主机的路径而无须事先定义。不过，若要跨多个服务使用同一个存储卷，则必须于顶级配置段volumes中予以定义。

同networks指令相似，volumes指令定义存储卷时也支持长短两种语法格式。短格式的语法格式为“[host_path:][container_path][:ro]”，下面的示例格式中给出了多种使用方式。

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql

  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql

  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache

  # User-relative path
  - ~/configs:/etc/configs/:ro

  # Named volume
  - datavolume:/var/lib/mysql
```

长格式语法允许配置那些无法以简短形式表示的其他配置字段，

- type：支持的存储卷绑定类型，有volume、bind、tmpfs或npipe；
- source：mount的源，指主机上用于绑定挂载到容器存储卷的路径或顶级volumes键中定义的存储卷的名称 ，显然此参数不适用于tmpfs类型；
- target：容器中的存储卷路径；
- read_only：挂载标志，用于将存储卷设置为只读模式；
- bind：配置额外的bind专用选项；
    - propagation：用于绑定的传播模式；
- volume：配置额外的volume专用选项；

    - nocopy：挂载标志，用于在创建存储卷时禁用从容器复制数据；
- tmpfs：配置额外的tmpfs专用选项
    - size：tmpfs mount的大小（以字节为单位）；
- consistency：mount的一致性要求
    - consistent：完全一致，容器运行时和主机始终保持相同的绑定视图，此为默认值；
    - cached：主机的mount视图是权威的，在主机上进行的更新在容器中可见之前可能会有延迟；
    - delegated：容器运行时的mount视图是权威的，在容器中进行的更新在主机上可见之前可能会有延迟；

下面的示例中，web服务使用了长格式的存储卷配置机制，它引用的命名存储卷都定义在顶级volumes配置段中，而db服务使用了短格式的存储卷配置机制，且采用了直接绑定主机路径的方式。

```
version: "3.7"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static

  db:
    image: postgres:latest
    volumes:
      - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"
      - "dbdata:/var/lib/postgresql/data"

volumes:
  mydata:
  dbdata:
```


restart

定义容器的重启策略，常用的重启策略如下。

- no：从不重启容器；
- always：总是重启容器，即任何时候探测到非正常运行状态时即重启容器；
- on-failure：容器退出代码为错误时重启容器；
- unless-stopped：仅容器转为停止状态时将其重启；

不过，集群模式下以版本3中的提供配置中将会忽略此指令，它需要使用的是restart_policy。

stop_signal和stop_grace_period

默认情况下，停止容器的信号是SIGTERM，但也可以使用stop_signal指令在Compose文件中设置自定义的停止信号，例如使用SIGUSR1。而如果容器未能正确处理SIGTERM或用户自定义的停止信号，则容器引擎会发送SIGKILL信号强制终止容器，这其间的等待时长默认为10秒，但也可以使用stop_grace_period指令进行自定义。

sysctls

配置在容器中设置的内核参数，支持列表和字典格式，下面的示例使用了列表格式。

```
sysctls:
  - net.core.somaxconn=1024
  - net.ipv4.tcp_syncookies=0
```

tmpfs

用于在容器内安装临时文件系统，支持单值或列表格式。但于Compose 3-3.5版本中的Swarm模式下部署Stack时会忽略此选项。而3.6及以上版本，还可以使用size参数设置临时文件系统的大小（单位是字节），默认为无限制。

```
 - type: tmpfs
     target: /app
     tmpfs:
       size: 1000
```

ulimits

自定义ulimits设置以覆盖容器的默认配置，支持将单个限制指定为整数，以及将软/硬件限制指定为映射。

```
ulimits:
  nproc: 65535
  nofile:
    soft: 20000
    hard: 40000
```

Compose的网络配置

默认情况下，Compose会为一个项目自动设置单个网络，默认名称为“项目名_default”，其中项目名通常为docker-compose.yaml文件所在的目录的名称。项目中的每个服务的每个容器都自动加入此默认网络且彼此互通。同时，运行于同一主机上的同一网络中的容器可使用服务名互相发现。

网络配置概述

例如，假设存在某一应用程序位于名为myapp的目录中，其中，docker-compose.yml文件内容如下所示：

```
version: "3"
services:
  web:
    image: ikubernetes/myapp2:v0.1
    depends_on:
    - db
    ports:
    - 8080:80
  db:
    image: redis:4.0-alpine
    expose:
    - 6379
```

那么，在运行docker-compose up命令时，它会自动创建名为myapp_default的bridge模式的网络，接着创建db服务的容器并以db为名称加入网络中，创建web服务的容器并以web为名称加入到此网络中。随后，每个容器都能够以加入网络时使用的名称互相访问，例如web服务的容器能够以db为名称访问db服务的容器，路径为redis://db:6379。

修改了服务配置后再次运行docker-compose时，会创建新容器取代旧配置的容器，然而，新容器可能会以不同的IP地址加入网络，但名称保持不变，因此，使用名称进行容器间的互通是完成服务发现的必要途径。

事实上，用户也可以在docker-compose.yaml配置文件中使用顶级指令networks创建一到多个自定义网络，从而实现更复杂的拓扑并指定自定义的网络驱动程序和相关选项。而且，在networks下，用户也可以为默认的网络配置额外的网络参数。

例如，下面的示例自定义了两个网络frontend和servicemesh，front-pront服务将用户请求代理至servicemesh网络中的app之上，而app则通过servicemesh网络与db服务互通。

```
version: "3"
services:
  front-proxy:
    build: ./envoy-front-proxy
    networks:
      - frontend
      - servicemesh
  app:
    image: ikubernetes/myapp2:v0.1
    networks:
      - servicemesh
  db:
    image: redis:4.0-alpine
    networks:
      - servicemesh

networks:
  frontend:
    # Use a custom driver
    driver: custom-driver-1
```

```
  servicemesh:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

网络配置指令

name

为网络指定自定义名称，不过，仅支持3.5及之后的配置版本。

```
version: "3.7"
networks:
  backend:
    name: servicemesh
```

另外，name也可与external一同使用，在引用已存在的网络时为其指定一个自定义名称。

driver

为网络指定要使用的驱动程序，默认的驱动程序取决于Docker引擎的配置，不过，通常对于单主机模式来说为bridge，而对于Swarm来说为overlay。Compose在配置时支持使用如下驱动：

- bridge：桥接式网络；
- overlay：叠加式网络，例如VXLAN或MACVLAN等；
- host或none：使用主机网络（hostnet）或不使用网络（none）；

driver_opts

为驱动程序传递的参数列表，每个列表项是一个键值对。各驱动程序所支持的参数也可能会有有所不同。

attachable

是否允许独立容器附加到此网络，仅支持overlay驱动程序以及3.2及其之后的配置版本。

enable_ipv6

是否启用ipv6功能，仅支持2版本的配置文件，目前3版本的配置文件尚不支持此功能。

ipam

指定自定义的IPAM配置，其配置是一个多属性对象，且每个属性均可选。

- driver：自定义IPAM的驱动程序，而不是默认值；
- config：包含零个或多个配置块的列表，每个配置块包含subnet键以指定使用的子网。

下面是一个完整的配置示例，它使用默认的驱动程序，并使用172.28.0.0/16子网网络。

```
ipam:
  driver: default
  config:
    - subnet: 172.28.0.0/16
```

external

使用某个事先存在的外部网络，但3.3之前的版本不支持与其它配置键同时使用，而3.4及之后的版本不再存在此限制。

Volumes的常用指令

存储卷（后面简称卷）通常是比容器的可写层进行数据持久的更的选择，毕竟卷不会增加其附属的容器的大小，并且其内容存在于给定容器的生命周期之外。另外，使用卷也更容易进行数据备份、迁移、共享、加密和预置等。显然，若容器生成的是非持久状态数据，应考虑使用 tmpfs挂载以避免将数据永久存储在任何位置，并通过避免写入容器的可写层来提高容器的性能。

name

由3.4版本添加的新我，用于为卷指定自定义名称，支持使用特殊字符，并支持与external一同使用。

```
version: "3.7"
volumes:
  data:
    name: my-app-data
```

labels

支持以列表或字典格式向容器添加元数据，建议使用反向DNS标签格式以防止与其它标签冲突。下面的示例以字典格式为容器添加标签。

```
labels:
  com.example.description: "Database volume"
  com.example.department: "IT/Ops"
  com.example.label-with-empty-value: ""
```

driver和driver_opts

driver指令用于为卷设定使用的驱动程序，默认的配置由Docker引擎定义，不过，大多数情况下用的都是local。而driver_opts用于为驱动程序以键值列表的格式传递配置参数，但不同驱动支持的配置参数也会有所不同。

```
volumes:
  example:
    driver_opts:
      type: "nfs"
      o: "addr=10.40.0.199,nolock,soft,rw"
      device: ":/docker/example"
```

external

用于指定使用事先已存的卷。但3.3及其之前的版本中，external不支持同driver、driver_opts和labels指令同时使用，但之后的版本无此限制。以下面的示例为例，Compose启动时会查找事先存储的data存储卷并将其挂载至db容器中。

```
version: "3.7"

services:
  db:
    image: postgres
    volumes:
      - data:/var/lib/postgresql/data

volumes:
  data:
    external: true
```

docker-compose命令行 马哥教育

docker-compose命令用于构建和管理Compose文件定义的服务等对象，命令的语法格式：

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
```

其中的-f选项用于指定使用Compose配置文件，默认为当前目录下的docker-compose.yaml，但也可以使用多个-f选项同时指定多个配置文件，运行时，Compose会将它们合并为一个配置，并按照人家娃人文件的顺序构建配置，后续的文件将覆盖并添加到前面的文件中。另外，每个配置都应该有一个项目名称，也可以使用-p选项指定项目名称。未指定时，Compose将使用当前目录的名称。下面是命令的简明使用帮助。

```
Define and run multi-container applications with Docker.

Usage:
  docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
  docker-compose -h|--help

Options:
  -f, --file FILE             Specify an alternate compose file
                              (default: docker-compose.yml)
  -p, --project-name NAME     Specify an alternate project name
                              (default: directory name)
  --verbose                   Show more output
  --log-level LEVEL           Set log level (DEBUG, INFO, WARNING, ERROR,
CRITICAL)
  --no-ansi                   Do not print ANSI control characters
  -v, --version               Print version and exit
  -H, --host HOST             Daemon socket to connect to
```

```
  --tls                      Use TLS; implied by --tlsverify
  --tlscacert CA_PATH        Trust certs signed only by this CA
  --tlscert CLIENT_CERT_PATH Path to TLS certificate file
  --tlskey TLS_KEY_PATH      Path to TLS key file
  --tlsverify                Use TLS and verify the remote
  --skip-hostname-check      Don't check the daemon's hostname against the
                             name specified in the client certificate
  --project-directory PATH   Specify an alternate working directory
                             (default: the path of the Compose file)
  --compatibility            If set, Compose will attempt to convert deploy
                             keys in v3 files to their non-Swarm equivalent

Commands:
  build              Build or rebuild services
  bundle             Generate a Docker bundle from the Compose file
  config             Validate and view the Compose file
  create             Create services
  down               Stop and remove containers, networks, images, and volumes
  events             Receive real time events from containers
  exec               Execute a command in a running container
  help               Get help on a command
  images             List images
  kill               Kill containers
  logs               View output from containers
  pause              Pause services
  port               Print the public port for a port binding
  ps                 List containers
  pull               Pull service images
  push               Push service images
  restart            Restart services
  rm                 Remove stopped containers
  run                Run a one-off command
  scale              Set number of containers for a service
  start              Start services
  stop               Stop services
  top                Display the running processes
  unpause            Unpause services
  up                 Create and start containers
  version            Show the Docker-Compose version information
```

案例

下面的案例中定义了五个Service：front-envoy、service_blue、service_green、service_red和jaeger，以及一个名为envoymesh的Network。其中front-envoy加入到自定义的网络envoymesh，并基于卷格式加载项目中的front-envoy.yaml配置文件给容器通过/etc/front-envoy.yaml使用，且暴露了端口80和8001。

```
version: '2'
services:

  front-envoy:
    build:
      context: ../apps
      dockerfile: Dockerfile-frontenvoy
    volumes:
      - ./front-envoy.yaml:/etc/front-envoy.yaml
    networks:
      - envoymesh
    expose:
      # Expose ports 80 (for general traffic) and 8001 (for the admin server)
      - "80"
      - "8001"
    ports:
      - "8000:80"
      - "8001:8001"

  service_blue:
    build:
      context: ../apps
      dockerfile: Dockerfile-service
    volumes:
      - ./service-blue-envoy-jaeger.yaml:/etc/service-envoy.yaml
    networks:
      envoymesh:
        aliases:
          - service_blue
    environment:
      - SERVICE_NAME=blue
    expose:
      - "80"
  service_green:
    build:
      context: ../apps
      dockerfile: Dockerfile-service
    volumes:
      - ./service-green-envoy-jaeger.yaml:/etc/service-envoy.yaml
    networks:
      envoymesh:
        aliases:
          - service_green
    environment:
      - SERVICE_NAME=green
    expose:
      - "80"
  service_red:
    build:
```

```
      context: ../apps
      dockerfile: Dockerfile-service
    volumes:
      - ./service-red-envoy-jaeger.yaml:/etc/service-envoy.yaml
    networks:
      envoymesh:
        aliases:
          - service_red
    environment:
      - SERVICE_NAME=red
    expose:
      - "80"
  jaeger:
    image: jaegertracing/all-in-one
    environment:
      - COLLECTOR_ZIPKIN_HTTP_PORT=9411
    networks:
      envoymesh:
        aliases:
          - jaeger
    expose:
      - "9411"
      - "16686"
    ports:
      - "9411:9411"
      - "16686:16686"
networks:
  envoymesh: {}
```

参考文献

1. https://docs.docker.com/compose/networking/
2. https://docs.docker.com/compose/compose-file/
3. https://github.com/docker/labs/tree/master/networking/concepts