docker란

- 리눅스 컨테이너에 여러 기능을 추가함으로써 애플리케이션을 컨테이너로서 좀더 쉽게 사용할 수 있게 만들어진 오픈소스 프로젝트
- 도커는 Go 언어로 작성되어 2013년 3월에 첫 번째 릴리스가 발표된 이후 지금까지 꾸준히 개발되고 있음
- 기존에 쓰이던 가상화 방법인 가상머신하고는 다르게 도커 컨테이너는 성능의 손실이 거의 없어서 차세대 클라우드 인프라 솔루션으로 많은 개발자들에게 주목받고 있음

가상머신과 도커 컨테이너

가상머신

- 기존의 가상화 기술은 하이퍼바이저를 이용해 여러 개의 운영체제를 하나의 호스트에 생성해 사용하는 방식
- 여러 개의 운영체제는 가상 머신이라는 단위로 구별되고, 각 머신에 우분투, CentOS등의 운영체제가 설치
- 하이퍼바이저에 의해 생성되고 관리되는 운영체제는 게스트 운영체제라고 하고, 다른 게스트 운영체제와는

완전히 독립된 공간과 시스템 자원을 할당 받아 사용

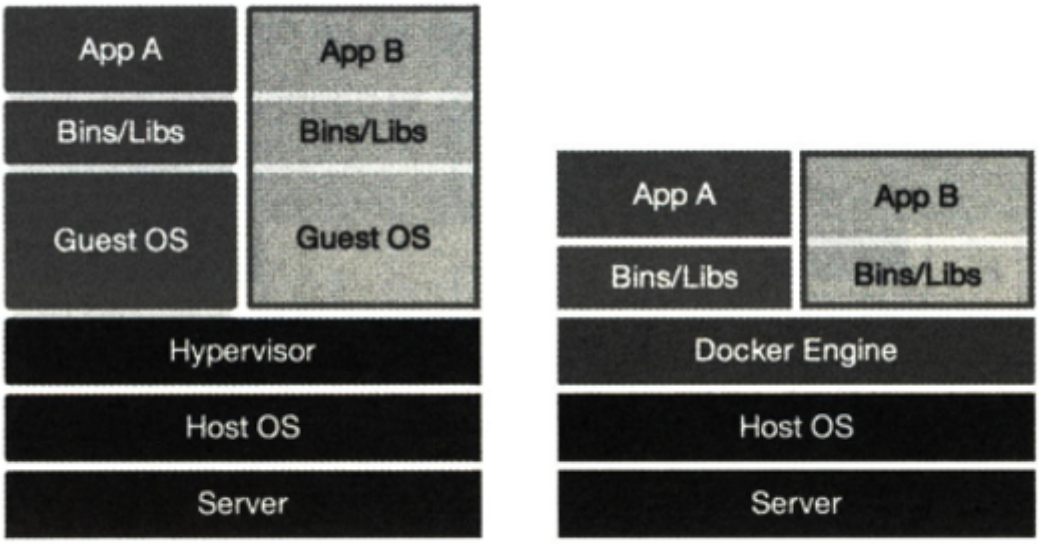


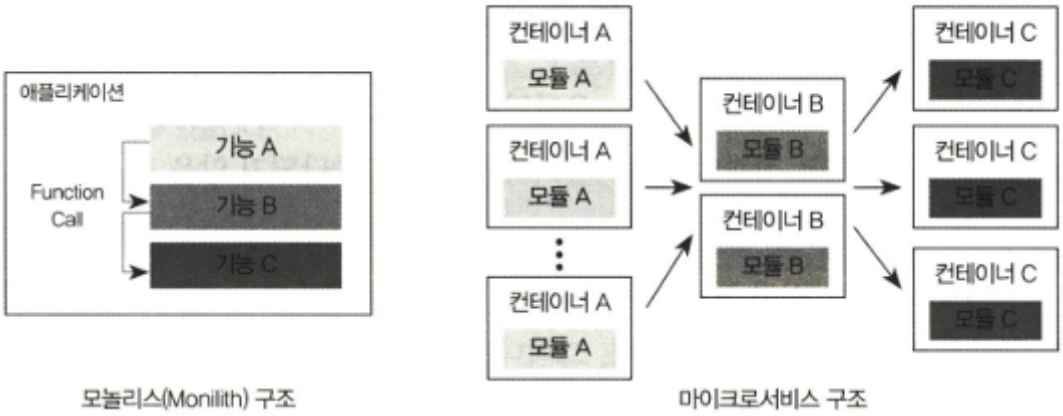
그림 1.1 가상 머신과 도커 컨테이너의 구조[1]

- 각종 시스템 자원을 가상화하고 독립된 공간을 생성하는 작업은 하이퍼바이저를 반드시 거치기 때문에 일반 호스트에 비해 성능의 손실이 발생함
- 또한 가상 머신을 배포하기 위한 이미지로 만들었을 때 이미지의 크기가 큰 단점이 있음

도커 컨테이너

- 도커 컨테이너는 가상화된 공간을 생성하기 위해 리눅스의 자체 기능인 chroot, 네임스페이스, cgroup을 사용함
- 위의 기능을 사용하기 때문에 프로세스 단위의 격리 환경을 만들기 때문에 성능 손실이 거의 없음
- 컨테이너에 필요한 커널은 호스트의 커널을 공유해 사용

- 컨테이너 안에는 애플리케이션을 구동하는 데 필요한 라이브러리 및 실행 파일만 존재함
- 그로 인해서 컨테이너를 이미지로 만들었을 때 이미지의 용량 또한 가상머신에 비해 매우 작음



모놀리스(Monilith) 구조 마이크로서비스 구조

장점

- 애플리케이션의 개발과 배포가 편해집니다.
  - 컨테이너 내부에서 여러 작업을 마친 뒤 이를 운영 환경에 배포하려고 한다면,
  - 해당 컨테이너를 도커 이미지라고 하는 일종의 패키지로 만들어 운영 서버에 전달하기만 하면 됨
- 여러 애플리케이션의 독립성과 확장성이 높아집니다.
  - 소프트웨어의 여러 모듈이 상호 작용하는 로직을 하나의 프로그램 내에서 구동시키는 방식을 모놀리스 애플리케이션이라고 함
  - 모놀리스는 소규모 서비스에는 적합하지만 확장성과 유연성은 포기해야 한다는 단점

- 모놀리스를 대체할 수 있는 방식이 마이크로서비스 구조
- 컨테이너는 마이크로서비스 구조에서 가장 많이 사용되고 있는 가상화 기술입니다.

도커 엔진

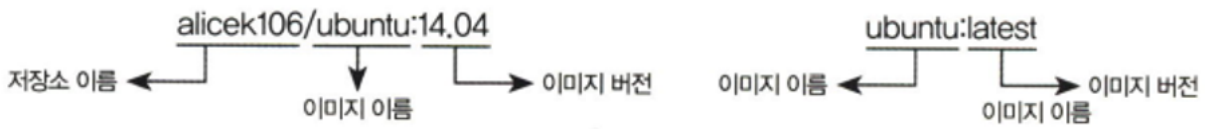
- 도커 엔진에서 사용하는 기본 단위는 이미지와 컨테이너이며, 이 두 가지가 도커 엔진의 핵심

도커 이미지

- 이미지는 컨테이너를 생성할 때 필요한 요소이며, 가상머신을 생성할 때 사용하는 iso 파일과 비슷한 개념
- 이미지는 여러 개의 계층으로 된 바이너리 파일로 존재하고, 컨테이너를 생성하고 실행할 때 읽기 전용으로 사용
- 이미지는 도커 명령어로 내려 받을 수 있으므로 별도로 설치할 필요는 없음

사용법

- [저장소 이름]/[이미지 이름]:[태그]



- 저장소(Repository) 이름은 이미지가 저장된 장소를 의미. 저장소
- 이름이 명시되지 않은 이미지는 도커에서 기본적으로 제공하는 이미지 저장소인 도커 허브의 공식 이미지를 의미
- 이미지 이름은 해당 이미지가 어떤 역할을 하는지 나타냄
- 태그는 이미지의 버전관리, 혹은 리비전(Revision) 관리에 사용합니다. 생략하면 도커 엔진은 이미지의 태그를 latest로 인식합니다.

도커 컨테이너

- 아파치 웹, nginx, mysql, 하둡, 스파크등의 도커 이미지가 존재
- 이러한 이미지로 컨테이너를 생성하면 해당 이미지의 목적에 맞는 파일이 들어 있는 파일시스템과 격리된 시스템 자원 및 네트워크를 사용할 수 있는 독립된 공간이 생성
- 이것이 바로 **도커 컨테이너**가 됨
- 컨테이너는 이미지를 읽기 전용으로 사용하여 이미지에 변경된 사항만 컨테이너 계층에 저장하므로 컨테이너에서 무엇을 하든 원래 이미지는 영향을 받지 않음
- 생성된 각 컨테이너는 각기 독립된 파일시스템을 제공받으며 호스트와 분리돼 있으므로 특정 컨테이너에서

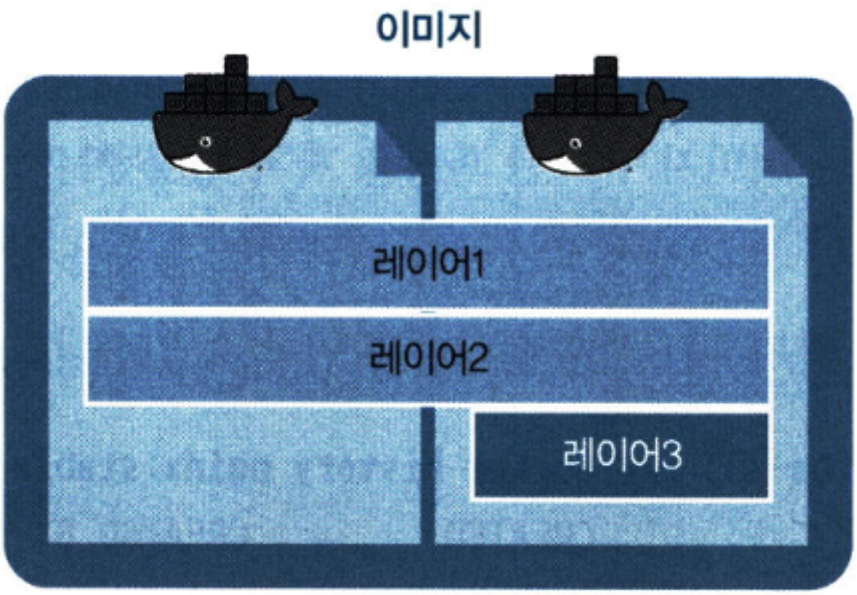
어떤 애플리케이션을 설치하거나 삭제해도 다른 컨테이너와 호스트는 변화가 없음

도커 컨테이너의 활용

- 웹 서버 도커 이미지로부터 여러 개의 컨테이너를 생성하면 생성된 컨테이너의 개수만큼 웹 서비스가 생성되고, 이 컨테이너들은 외부에 웹 서비스를 제공하는 데 사용

이미지의 레이어 구조

- 이미지는 애플리케이션과 각종 파일을 담고 있는 점에서 압축 파일에 가까움
- 이미지는 같은 내용일 경우 여러 이미지에 동일한 레이어를 공유하므로 전체 용량이 감소



- 내용이 같은 레이어1, 레이어2를 공유하기 때문에 전체 공간에서 봤을 때 상대적으로 용량을 적게 차지

도커 컨테이너 명령어

컨테이너 생성

- 도커 버전 확인

```
docker -v
```

- 도커 컨테이너 생성
    - i 옵션으로 상호 입출력을, -t 옵션으로 tty를 활성화 해서 bash shell을 사용하도록 컨테이너를 설정

```
docker run -it ubuntu:14.04
```

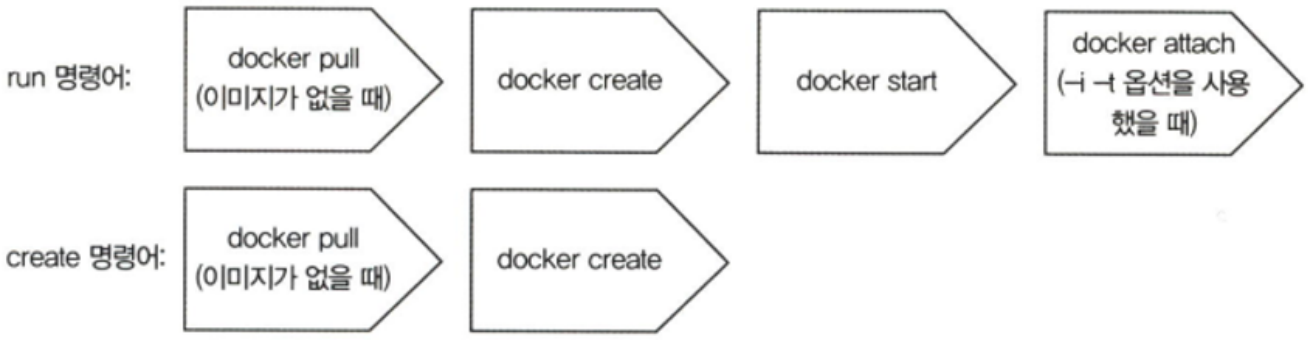
- ubuntu:14.04 이미지가 로컬 도커 엔진에 존재하지 않으면 도커 중앙 이미지 저장소인 도커 허브에서 자동으로 이미지 다운로드
- 컨테이너와 호스트의 파일시스템은 서로 독립적이므로 ls 명령어로 파일 시스템을 확인해보면 아무것도 설치되지 않은 상태
- 컨테이너 내부에서 빠져나오기

    - 완전 종료 -> exit 실행 or Ctrl + D
    - 종료시키지 않고 빠져만 나오기 -> Ctrl + P, Q
- 무작위 16진수 해시값은 컨테이너의 고유 ID
- 너무 길어 일반적으로 앞의 12자리만 사용
- pull, create, start, attach

```
docker pull centos:7
docker create -it --name mycentos centos:7
docker start mycentos
docker attach mycentos
```

- 위의 명령어를 하나로 통합하여 사용할 수 있는데 그 명령어가 run 명령어임

```
docker run -it --name mycentos2 centos:7
```



- 컨테이너를 생성함과 동시에 시작하기 때문에 run 명령어를 더 많이 사용

run 명령어 옵션

- docker run [옵션] 이미지:[태그명]
- https://docs.docker.com/engine/reference/commandline/container_run/

지정할 수 있는 주요 옵션

옵션	설명
--attach, -a	표준 입력(STDIN), 표준 출력(STDOUT), 표준 오류 출력(STDERR)에 어태치한다.
--cidfile	컨테이너 ID를 파일로 출력한다.
--detach, -d	컨테이너를 생성하고 백그라운드에서 실행한다.
--interactive, -i	컨테이너의 표준 입력을 연다.
--tty, -t	단말기 디바이스를 사용한다.

지정할 수 있는 주요 옵션

옵션	설명
--detach, -d	백그라운드에서 실행
--user, -u	사용자명을 지정
--restart=[no | on-failure | on-failure:횟수n | always | unless-stopped]	명령의 실행 결과에 따라 하는 옵션
--rm	명령 실행 완료 후에 자동으로 삭제

- --env
  - 컨테이너의 환경 변수 지정
- -v, --volume=호스트경로:컨테이너경로
  - 호스트 경로와 컨테이너 경로의 공유 볼륨 설정
  - Bind mount라고 함
- -h

    - 컨테이너의 호스트명 지정(미저정 시 컨테이너 ID가 호스트명으로 등록)
- -p 호스트포트:컨테이너포트
    - 호스트 포트와 컨테이너 포트 연결
- -P, --publish-all=[true|false]
    - 컨테이너 내부의 노출된 포트를 호스트 임의의 포트에 게시
- --link=[container:container_id]
    - 동일 호스트의 다른 컨테이너와 연결 설정으로 IP가 아닌 컨테이너의 이름을 이용해 통신

컨테이너 목록 확인

```
# 현재 실행중인 컨테이너 보기
docker ps
# stop 상태까지 모든 컨테이너 보기
docker ps -a
# ubuntu 기반으로 생성된 컨테이너만 보기
docker ps -f ancestor=ubuntu
# id이름에 ce7으로 시작하는 컨테이너 조회
docker ps -f id=ce7
```

- 출력 결과 설명

- IMAGE : 컨테이너를 생성할 대 사용된 이미지의 이름
- COMMAND : 컨테이너가 시작될 때 실행될 명령어
- CREATED : 컨테이너가 생성되고 난 뒤 흐른 시간을 나타냄
- STATUS : 컨테이너의 상태, UP : 실행, Exited : 종료된 상태, Pause: 중지된 상태
- PORTS : 컨테이너가 개방한 포트와 호스트에 연결한 포트를 나열
- NAMES : 컨테이너의 고유한 이름, --name 옵션으로 이름을 설정하지 않으면 도커 엔진이 임의로 형용사와 명사를 무작위로 조합해 이름을 설정

컨테이너 이름 변경

```
docker rename [현재이름] [변경이름]
```

컨테이너 정지

```
docker stop [컨테이너 이름 ]
```

컨테이너 삭제

```
docker rm [컨테이너 이름]
```

- 한 번 삭제한 컨테이너는 복구 할 수 없음
- 실행 중인 컨테이너는 삭제할 수 없으므로 정지한 뒤 삭제하거나 강제로 삭제하는 옵션 추가

```
docker rm -f [이름]

# 모든 컨테이너 삭제
docker container prune

# ps 명령어와 조합하여 삭제
# 아래 명령어는 컨테이너의 id 값을 모두 출력
docker ps -a -q

docker stop $(docker ps -a -q)
docker rm $(docker ps -a -q)
```

포트 바인딩

- 컨테이너를 외부에 노출
- p 옵션을 사용하면 컨테이너 내부와 외부를 연결
- [host port] : [container port]

```
docker run -it --name myweb -p 80:80 ubuntu:14.04
```

- 여러 개의 포트를 개방하려면 -p 옵션을 여러 번 사용
- 한번 만들어진 컨테이너에는 포트를 바인딩 할 수 없음
    - 곧 다시 컨테이너 생성해야 함

port 바인딩 확인

```
docker port [컨테이너 이름]
```

컨테이너 명령어

- exec를 사용하여 내부 명령어 사용

```
docker exec -it wordpressdb /bin/bash
```

- 파일 복사

```
# 컨테이너 안의 파일을 호스트에 저장
docker cp [컨테이너 식별자]:[컨테이너안의 파일 경로]
[저장할 호스트의 경로]
예) docker cp webserver:/etc/nginx/nginx.conf
/tmp/nginx.conf
```

```
# 호스트의 파일을 컨테이너 안으로 저장
docker cp [호스트 파일] [컨테이너 식별자]:[컨테이너 안의 경로]
예) docker cp ./test.txt  webserver:/tmp/test.txt
```

컨테이너 자원 할당 제한

- 컨테이너를 생성하는 run, create 명령어에서 컨테이너의 자원 할당량을 조정하도록 옵션을 입력 할 수 있음
- 아무 옵션을 쓰지 않을 경우 컨테이너는 호스트의 자원을 제한 없이 쓸 수 있게 설정되어 있음
- 호스트에서 다른 컨테이너와 함께 사용한다면 자원 할당을 제한해 호스트와 다른 컨테이너의 동작을 방해하지 않게 설정하는 것이 좋음

```
docker update --cpuset-cpus=1 ubuntu
```

Memory 리소스 제한

- 제한 단위는 b, k, m, g로 할당

옵션	의미
--memory, -m	컨테이너가 사용할 최대 메모리 양을 지정

옵션	의미
--memory-swap	컨테이너가 사용할 스왑 메모리 영역에 대한 설정 메모리+스왑 생략 시 메모리의 2배가 설정됨
--memory-reservation	--memory 값보다 적은 값으로 구성하는 소프트 제한 값 설정
--oom-kill-disable	OOM Killer가 프로세스 kill하지 못하도록 보호

```
docker run -d \
        --memory="1g" \
        --name memory_1g \
        nginx
```

- 주의해야 할 점은 컨테이너 내에서 동작하는 프로세스가 컨테이너에 할당된 메모리를 초과하면 컨테이너는 자동으로 종료됨

```
docker run -it --name swap_500m \
        --memory=200m \
        --memory-swap=500m \
        ubuntu:14.04
```

- 기본적으로 컨테이너의 Swap 메모리는 메모리의 2배로 설정됨

- 별도의 저정은 --memory-swap 옵션을 사용해서 정할 수 있음

컨테이너 cpu 제한

옵션	의미
--cpu-shares	컨테이너에 가중치를 설정해 해당 컨테이너가 cpu를 상대적으로 얼마나 사용할 수 있는지 나타냄 컨테이너에 cpu를 한 개씩 할당하는 방식이 아닌, 시스템에 존재하는 cpu를 어느 비중만큼 나눠 쓸 것인지를 명시하는 옵션 cpu 비중을 1024 값을 기반으로 설정되므로 2048이라고 할 경우 기본 값보다 두 배 많은 cpu 자원을 할당
--cpuset-cpu	호스트에 cpu가 여러 개 있을 때 해당 옵션을 지정해 컨테이너가 특정 cpu만 사용하도록 설정 cpu 집중적인 작업이 필요하다면 여러 개의 cpu를 사용하도록 설정해 작업을 적절하게 분배하는 것이 좋음
--cpus	cpu의 개수를 직접 지정. 예를 들어 --cpus 옵션에 0.5를 설정하면 컨테이너의 cpu를 제한할 수 있음

```
docker run -d --name cpuset_2 \
        --cpu-shares 2048 \
```

```
        ubuntu:14.04
```

```
docker run -d --name cpuset_2 \
        --cpuset-cpus=2 \
        ubuntu:14.04
```

docker monitoring

```
# docker stats
CONTAINER ID   NAME                          CPU %
MEM USAGE / LIMIT     MEM %     NET I/O
BLOCK I/O         PIDS
2ebe4a1b024d   airflow_work_webserver_1   111.63%
964.6MiB / 3.813GiB   24.70%    247MB / 193MB
4.49MB / 48.2MB   18
512a21f2df17   airflow_work_postgres_1    0.00%
46.55MiB / 3.813GiB   1.19%     194MB / 248MB
3.04MB / 90.1MB   12
61bc41edfaf3   myweb                      0.05%
5.949MiB / 3.813GiB   0.15%     11.9kB / 12kB
672kB / 16.4kB    57
```

컨테이너 애플리케이션 구축 예제

- mysql과 wordpress 컨테이너 실행
- mysql 5.7 버전 컨테이너 생성

```
docker run -d --name wordpressdb -e
MYSQL_ROOT_PASSWORD=□encore -e
MYSQL_DATABASE=wordpress mysql:5.7
```

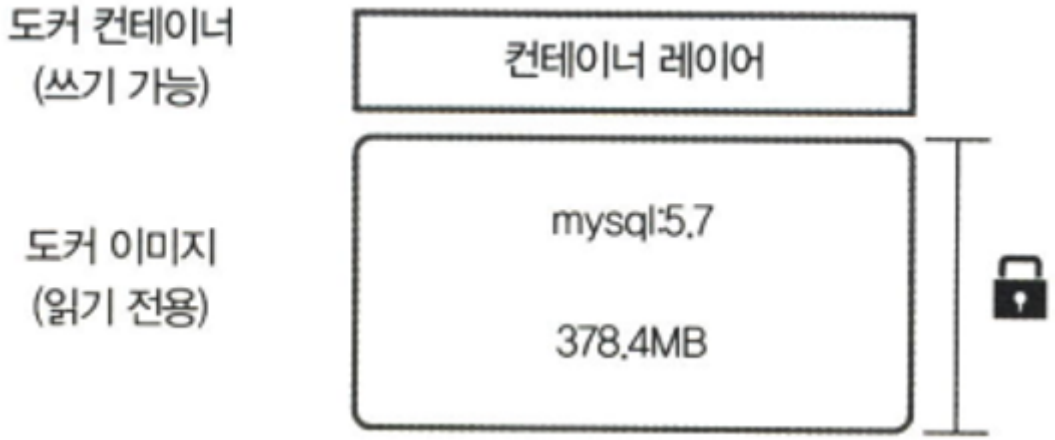
- wordpress 컨테이너 생성

```
docker run -d -e WORDPRESS_DB_USER=root -e
WORDPRESS_DB_PASSWORD=encore -e
WORDPRESS_DB_NAME=wordpress --name wordpress --link
wordpressdb:mysql -p 80:80 wordpress
```

- 사용된 옵션
  - d : detached 모드의 의미로 **백그라운드**에서 동작하는 애플리케이션으로 실행
  - e : 컨테이너 내부의 환경변수를 설정
    - 위에서 MYSQL_ROOT_PASSWORD라는 값을 □encore로 설정
  - link : 컨테이너의 alias으로 접근하도록 설정.
    - 위의 예제에서 wordpressdb를 mysql이라는 호스트명으로 접근

도커 볼륨

- 도커 이미지로 컨테이너를 생성하면 이미지는 읽기 전용이 되며 컨테이너의 변경 사항만 별도로 저장해서 각 컨테이너의 정보를 보존 함
- mysql 컨테이너는 mysql:5.7이라는 이미지로 생성됐지만 워드프레스 블로그를 위한 데이터베이스등의 정보는 컨테이너가 보유



- 생성된 이미지는 어떠한 경우로도 변경되지 않으며, 컨테이너 계층에 원래 이미지에서 변경된 파일 시스템 등을 저장
- mysql 컨테이너를 삭제하면 컨테이너 계층에 저장되어 있는 데이터베이스의 정보도 다 삭제
- 이를 방지하기 위해 컨테이너의 데이터를 영속적으로 데이터로 활용할 수 있는 방법이 필요

호스트와 공유

- 컨테이너 안의 내용을 호스트와 공유

- 컨테이너가 삭제되어도 공유되었던 호스트의 경로에는 있는 파일은 삭제되지 않음

```
docker run -d --name wordpressdb_hostvol -e
MYSQL_ROOT_PASSWORD=encore -e
MYSQL_DATABASE=wordpress -v ~/data:/var/lib/mysql
mysql:5.7
```

- 사용자 계정의 홈 디렉토리 밑에 data폴더와 컨테이너 안의 /var/lib/mysql 폴더를 공유하겠다는 의미
- 호스트의 data폴더에 들어가면 아래와 같이 보임

```
(base) gen2@DESKTOP-41H5R5E:~/data$ ls
auto.cnf    ca.pem           client-key.pem
ib_logfile0  ibdata1  mysql
performance_schema  public_key.pem   server-key.pem
ca-key.pem  client-cert.pem  ib_buffer_pool
ib_logfile1  ibtmp1   mysql.sock  private_key.pem
server-cert.pem  sys
```

호스트 폴더에 파일이 존재한다면?

- 호스트의 경로에 이미 파일이 존재한 상태에서 컨테이너에도 파일이 존재할 때

1. 컨테이너 안의 파일 확인

```
docker run -it --name volume_dummy
alicek106/volume_test
cd /home/testdir_2/



root@2883006e4d85:/home/testdir_2# cat test
testdir_2!
```

2. 컨테이너를 종료한 뒤에 ~/test/ 폴더안에 파일 생성

```
mkdir ~/test && cd ~/test
echo "encore" > a.txt
```

3. test폴더와 컨테이너의 testdir_2폴더 공유

```
docker run -it --name volume_overide -v
~/test/:/home/testdir_2 alicek106/volume_test

root@9404fee88c62:/# cd /home/testdir_2/
root@9404fee88c62:/home/testdir_2# ls
a.txt
```

- 기존에 컨테이너 안에 있던 파일은 삭제되어 있고, 호스트의 파일이 존재하는 것을 확인할 수 있음

볼륨 컨테이너

- 볼륨 컨테이너를 사용하여 다른 컨테이너와 공유
- 위의 예제에서 만든 volume_override 컨테이너의 볼륨을 사용
- --volumes-from 옵션 사용

```
docker run -it --name vol_from_cont1 --volumes-from
volume_overide ubuntu:14.04
docker run -it --name vol_from_cont2 --volumes-from
volume_overide ubuntu:14.04
```

- 위의 예제는 ~/test폴더와 두 컨테이너의 /home/testdir_2 폴더가 공유되는 것을 확인할 수 있음

도커 볼륨 명령어

- docker inspect 명령어를 사용하면 생성한 볼륨이 실제로 어디에 저장되어 있는 확인 가능

```
docker inspect myvolume
예)
(base) gen2@DESKTOP-41H5R5E:~$ docker inspect
```

```
myvolume
[
    {
        "CreatedAt": "2023-10-04T02:01:07+09:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint":
"/var/lib/docker/volumes/myvolume/_data",
        "Name": "myvolume",
        "Options": null,
        "Scope": "local"
    }
]
```

- driver : 볼륨이 쓰는 드라이버
- label : 볼륨을 구분하는 라벨
- mountpoint : 해당 볼륨이 실제로 호스트의 어디에 저장되어 있는지

도커 볼륨 생성

- docker volume 명령어 사용
- docker volume으로 시작하며, docker volume create 명령어로 볼륨을 생성

```
docker volume create --name myvolume
```

- 생성된 볼륨 확인

```
docker volume ls
```

- 컨테이너를 생성하여 /root/ 폴더를 공유

```
docker run -it --name myvolume_1 -v myvolume:/root
ubuntu:14.04

# 파일 생성해보기
echo "hello, volume" >> /root/volume.txt

# 또 다른 컨테이너 생성
docker run -it --name myvolume_2 -v myvolume:/root
ubuntu:14.04
```

도커 볼륨 삭제

- docker volume rm [volume name]

```
(base) gen2@DESKTOP-41H5R5E:~$ docker volume create
--name myvolume2
myvolume2
```

```
(base) gen2@DESKTOP-41H5R5E:~$ docker volume rm
myvolume2
myvolume2
```

- 사용되지 않은 볼륨을 일괄 삭제는 아래 명령어를 사용

```
docker volume prune
```

도커 네트워크

host의 네트워크 구조

- docker0
  - 도커 설치 시 기본적으로 제공되는 브리지 네트워크
  - 172.17.0.1 주소를 가지고 있음
  - docker0 브리지는 소프트웨어 스위치 방식으로 동작하며, 일반적인 스위치 방식과는 다르게 dhcp로 연결된 컨테이너 사전에 정의된 IP풀을 할당
- vethxxxx
  - OSI 7 계층 서비스 모델의 2계층 서비스로 컨테이너 내부에 제공되는 네트워크 인터페이스 eth0

와 한 쌍으로 제공되어 docker0와 가상의 터널링 네트워크를 제공

컨테이너안의 네트워크 구조

- 컨테이너 내부에서 ifconfig를 입력하면 아래와 같은 값을 확인할 수 있음

```
root@24bd5377829c:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>
mtu 1500
        inet 172.18.0.3  netmask 255.255.0.0
broadcast 172.18.255.255
        ether 02:42:ac:12:00:03  txqueuelen 0
(Ethernet)
        RX packets 13500  bytes 28742877 (28.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6247  bytes 417229 (417.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
```

```
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0
```

- 도커는 컨테이너에 내부에 IP를 순차적으로 할당하며, 이 IP는 컨테이너를 재시작할 때마다 변경될 수 있음
- 이 내부 IP는 도커가 설치된 호스트, 즉 내부망에서만 쓸 수 있는 IP이므로 외부와 연결될 필요가 있음
- 이 과정은 컨테이너를 시작할 때마다 호스트에서 veth..라는 네트워크 인터페이스를 생성함으로써 이뤄짐

```
vethdfcb404:
flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu
1500
        inet6 fe80::a826:34ff:fe5d:9bed  prefixlen
64  scopeid 0x20<link>
        ether aa:26:34:5d:9b:ed  txqueuelen 0
(Ethernet)
        RX packets 7234  bytes 977836 (977.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7789  bytes 597397 (597.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier
```

```
0  collisions 0

docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>
mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0
broadcast 172.18.255.255
        inet6 fe80::42:2dff:fe89:9f63  prefixlen 64
scopeid 0x20<link>
        ether 02:42:2d:89:9f:63  txqueuelen 0
(Ethernet)
        RX packets 13572  bytes 1209863 (1.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21359  bytes 29344702 (29.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier
0  collisions 0
```

- docker0이라는 브리지도 존재하는데 docker0 브리지는 각 veth 인터페이스와 바인딩돼 호스트의 eth0 인터페이스와 이어주는 역할을 함
- 즉 컨테이너와 호스트의 네트워크는 아래 그림과 같은 구성

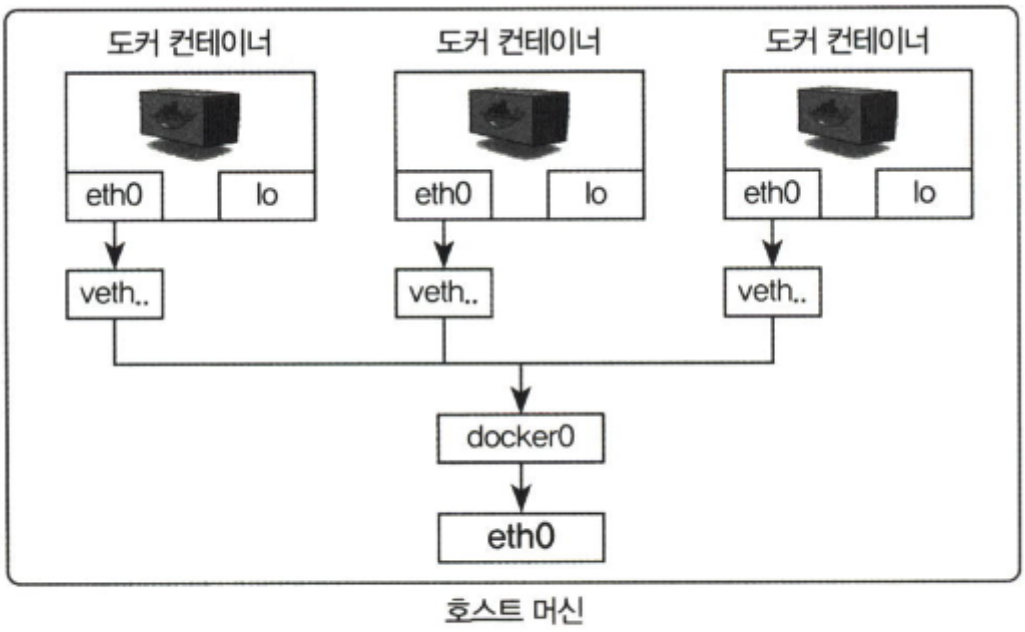


그림 2.15 도커 네트워크의 구조

- 컨테이너의 eth0 인터페이스는 호스트의 veth..라는 인터페이스와 연결되어 있으며 veth 인터페이스는 docker0 브리지와 바인딩돼 외부와 통신할 수 있음

```
gen2@DESKTOP-RB1H4O8:$ brctl show docker0
bridge name     bridge id               STP enabled
interfaces
docker0         8000.02422d899f63       no
vethdfcb404
```

도커 네트워크 기능

- 컨테이너를 생성하면 기본적으로 docker0 브리지를 통해 외부와 통신할 수 있는 환경을 사용할 수 있음

- 사용자의 선택에 따라 여러 네트워크 드라이버를 사용할 수 있음
- 도커가 자체적으로 제공하는 대표적인 네트워크 드라이버로는 브리지(bridge), 호스트(host), 논(none), 컨테이너(container), 오버레이(overlay)가 존재

```
gen2@DESKTOP-RB1H4O8:~$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
cf3b5ed4561d   bridge    bridge    local
0d281d58fad8   host      host      local
fcf006c54135   none      null      local
```

- 브리지 네트워크는 컨테이너를 생성할 때 자동으로 연결되는 docker0 브리지를 활용하도록 설정돼 있음

```
gen2@DESKTOP-RB1H4O8:~$ docker network inspect
bridge
[
    {
        "Name": "bridge",
        "Id":
"cf3b5ed4561d49ed52f66b76b90515feecdb52057f2c1a2b1d
11e5768e455a6b",
        "Created": "2023-10-
```

```
05T23:02:08.285795373+09:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {

"b64eb470732bad9b2bf54b967d4096b8bd090c3f7118b44d5f
ca6f778c23fe28": {
```

```
                "Name": "encoredb2",
                "EndpointID":
"3a0cf01014e4088824df492d90b008a4ff012012745d5be009
b32b35be7b22ca",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {

"com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc":
"true",

"com.docker.network.bridge.enable_ip_masquerade":
"true",

"com.docker.network.bridge.host_binding_ipv4":
"0.0.0.0",
            "com.docker.network.bridge.name":
"docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
```

```
    }
]
```

브리지 네트워크

- 기본적으로 존재하는 docker0를 사용하는 브리지 네트워크가 아닌 새로운 브리지 타입의 네트워크를 생성

```
docker network create --driver bridge mybridge
```

- mybridge를 사용하는 컨테이너 생성

```
docker run -it --name mynetwork_cont --net mybridge
ubuntu:14.04
```

- 컨테이너 내부에서 ifconfig를 입력하면 새로운 IP대역이 할당된 것을 알 수 있음
- 브리지 타입의 네트워크를 생성하면 도커는 IP대역을 차례대로 할당
    - mybridge라는 새롭게 생성한 네트워크에 할당한 컨테이너 주소를 보면 2번으로 할당되어 있는 것을 확인 할수 있음
    - 도커는 위에서 언급한 대로 IP대역을 차례대로 할당

- 기본 네트워크를 사용한 컨테이너와 mybridge로 만든 컨테이너와 서로 통신이 안되는 것을 확인

```
gen2@DESKTOP-RB1H4O8:$ docker run -it --name
mynetwork_cont --net mybridge ubuntu:14.04
root@b390d7b34dda:/# ifconfig
eth0      Link encap:Ethernet  HWaddr
02:42:ac:13:00:02
          inet addr:172.19.0.2
Bcast:172.19.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:11 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:962 (962.0 B)  TX bytes:0 (0.0
B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
```

```
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

gen2@DESKTOP-RB1H4O8:$ docker run -it --name
network ubuntu:14.04
root@3e0ae7e6402c:/# ifconfig
eth0      Link encap:Ethernet  HWaddr
02:42:ac:12:00:02
          inet addr:172.18.0.2
Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:9 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:782 (782.0 B)  TX bytes:0 (0.0
B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
```

```
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@3e0ae7e6402c:/# ping 172.19.0.2
PING 172.19.0.2 (172.19.0.2) 56(84) bytes of data.
```

- 사용자 정의 네트워크는 docker network disconnect, connect를 통해 컨테이너에 유동적으로 붙이고 뗄 수 있음
    - 아래 예제는 네트워크를 떼고 컨테이너에서 ping을 사용하여 네트워크 통신 여부 확인(통신이 안됨)

```
gen2@DESKTOP-RB1H4O8:$ docker network disconnect
mybridge myentwork_cont
gen2@DESKTOP-RB1H4O8:$ docker attach mynetwork_cont
root@b390d7b34dda:/#
root@b390d7b34dda:/#
root@b390d7b34dda:/# ping 8.8.8.8
connect: Network is unreachable
root@b390d7b34dda:/# ifconfig
```

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

- connect를 사용하여 네트워크를 붙임
  - 붙인 이후에는 통신이 가능

```
gen2@DESKTOP-RB1H4O8:$ docker network connect
mybridge mynetwork_cont
gen2@DESKTOP-RB1H4O8:$ docker attach myentwork_cont
root@b390d7b34dda:/#
root@b390d7b34dda:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=57 time=26.9
ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=57 time=115
ms
```

- 주의 : **논 네트워크, 호스트 네트워크 등과 같은 특별한 네트워크 모드에서는 network connect, disconnect 명령어 사용 불가**
- 네트워크 생성
    - 서브넷, 게이트웨이, IP 할당 범위 등을 임의로 설정하려면 네트워크 생성할 때 아래와 같이 입력

```
gen2@DESKTOP-RB1H4O8:~$ docker network create --
driver=bridge --subnet=172.72.0.0/16 --ip-
range=172.72.0.0/24 --gateway=172.72.0.1
my_custom_network
a984f987c5808e5ae1d77155dbdc4e4e582004340f50a09a050
16714eeb13b0b
```

호스트 네트워크

- 네트워크를 호스트로 설정하면 호스트의 네트워크 환경을 그대로 사용할 수 있음
- 위의 브리지 드라이버 네트워크와 달리 호스트 드라이버의 네트워크는 별도로 생성할 필요 없이 기존의 host라는 이름의 네트워크를 사용

```
gen2@DESKTOP-RB1H4O8:~$ docker run -it --name
network_host --net host ubuntu:14.04
```

- 컨테이너의 네트워크를 호스트 모드로 설정하면 컨테이너 내부의 애플리케이션을 별도의 포트 포워딩 없이 바로 서비스할 수 있음
- 실제 호스트에서 애플리케이션을 외부에 노출하는 것과 같음
- 이 컨테이너안에서 아파치 웹 서버를 구동한다면 호스트의 IP와 컨테이너의 아파치 웹 서버 포트인 80으로 바로 접근할 수 있음

논 네트워크

- none은 말 그대로 아무런 네트워크를 쓰지 않는 다는 것을 의미

```
docker run -it --name network_none --net none
ubuntu:14.04
```

- --net 옵션으로 none을 설정한 컨테이너 내부에서 네트워크 인터페이스를 확인하면 로컬호스트를 나타내는 lo외에는 존재하지 않은 것을 확인

컨테이너 네트워크

- --net 옵션으로 container를 입력하면 다른 컨테이너의 네트워크 네임스페이스 환경을 공유할 수 있음

- 공유되는 속성은 내부 IP, 네트워크 인터페이스의 MAC 주소

```
docker run -itd --name network_cont_1 ubuntu:14.04
```

- itd 옵셥을 같이 쓰는 이유는 ubuntu 이미지를 실행하면 bash가 실행되기 때문에 -d로만 실행하면 바로 stop 상태가 되기 때문에 it 옵셥을 같이 사용

```
docker run -itd --name network_cont_2 --net
container:network_cont_1 ubuntu:14.04
```

- 결과 확인

```
root@4a267e989e26:/# ifconfig
eth0      Link encap:Ethernet  HWaddr
02:42:ac:12:00:02
          inet addr:172.18.0.2
Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:11 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
```

```
overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:906 (906.0 B)  TX bytes:0 (0.0
B)



lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)



root@4a267e989e26:/# ifconfig
eth0      Link encap:Ethernet  HWaddr
02:42:ac:12:00:02
          inet addr:172.18.0.2
Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
          RX packets:11 errors:0 dropped:0
```

```
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:906 (906.0 B)  TX bytes:0 (0.0
B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0
overruns:0 frame:0
          TX packets:0 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

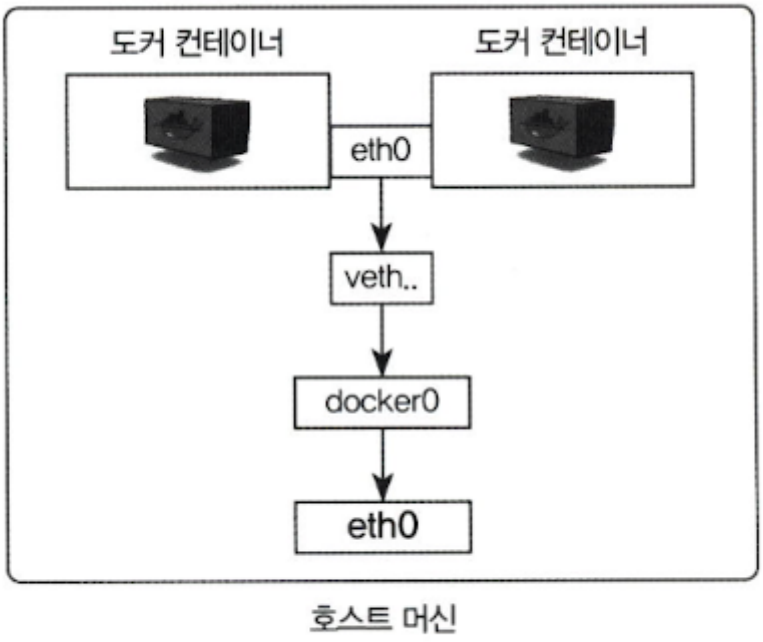


그림 2.16 컨테이너 네트워크의 구조

브리지 네트워크와 --net-alias

- 브리지 타입의 네트워크와 run 명령어의 --net-alias 옵션을 함께 쓰면 특정 호스트 이름으로 컨테이너 여러 개에 접근할 수 있음
- --net-alias 또는 --link 옵셥으로 묶인 모든 컨테이너에는 기본적으로 서비스를 검색할 수 있는 내장 DNS 서버가 제공
- 자동화 DNS 확인(automatic DNS resolution)

```
docker network create \
        --driver bridge \
        --subnet 172.200.1.0/24 \
        --ip-range 172.200.1.0/24 \
```

```
        --gateway 172.200.1.1 \
        netlb
```

- 컨테이너 실행

```
docker run -itd --name=nettest1 \
        --net=netlb \
        --net-alias inner-dns-test \
        ubuntu:14.04

docker run -itd --name=nettest2 \
        --net=netlb \
        --net-alias inner-dns-test \
        ubuntu:14.04

docker run -itd --name=nettest3 \
        --net=netlb \
        --net-alias inner-dns-test \
        ubuntu:14.04
```

- 컨테이너 ip 확인

```
gen2@DESKTOP-RB1H4O8:~$ docker inspect nettest1 |
grep IPAddress
```

```
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.200.1.2",
gen2@DESKTOP-RB1H4O8:~$ docker inspect nettest2 |
grep IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.200.1.3",
gen2@DESKTOP-RB1H4O8:~$ docker inspect nettest3 |
grep IPAddress
            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.200.1.4",
```

- 동일한 네트워크에 컨테이너 생성

```
docker run -it --name=front --net=netlb
ubuntu:14.04

root@2604abcf69f9:/# ping -c 2 inner-dns-test
PING inner-dns-test (172.200.1.4) 56(84) bytes of
data.
64 bytes from nettest3.netlb (172.200.1.4):
icmp_seq=1 ttl=64 time=0.048 ms
64 bytes from nettest3.netlb (172.200.1.4):
```

```
icmp_seq=2 ttl=64 time=0.096 ms

--- inner-dns-test ping statistics ---
2 packets transmitted, 2 received, 0% packet loss,
time 1001ms
rtt min/avg/max/mdev = 0.048/0.072/0.096/0.024 ms
root@2604abcf69f9:/# ping -c 2 inner-dns-test
PING inner-dns-test (172.200.1.2) 56(84) bytes of
data.
64 bytes from nettest1.netlb (172.200.1.2):
icmp_seq=1 ttl=64 time=0.040 ms
64 bytes from nettest1.netlb (172.200.1.2):
icmp_seq=2 ttl=64 time=0.107 ms

--- inner-dns-test ping statistics ---
2 packets transmitted, 2 received, 0% packet loss,
time 1008ms
rtt min/avg/max/mdev = 0.040/0.073/0.107/0.034 ms
root@2604abcf69f9:/# ping -c 2 inner-dns-test
PING inner-dns-test (172.200.1.3) 56(84) bytes of
data.
64 bytes from nettest2.netlb (172.200.1.3):
icmp_seq=1 ttl=64 time=0.131 ms
```

```
64 bytes from nettest2.netlb (172.200.1.3):
icmp_seq=2 ttl=64 time=0.097 ms
```

- dns 확인 (dig 명령어는 dnsutils를 설치하면 사용 가능)

```
root@2604abcf69f9:/# dig inner-dns-test

; <<>> DiG 9.9.5-3-Ubuntu <<>> inner-dns-test
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id:
24707
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY:
0, ADDITIONAL: 0

;; QUESTION SECTION:
;inner-dns-test.                        IN      A

;; ANSWER SECTION:
inner-dns-test.         600     IN      A
172.200.1.2
inner-dns-test.         600     IN      A
172.200.1.4
```

```
inner-dns-test.         600     IN      A
172.200.1.3
```

- 사용자 정의 브리지 네트워크의 DNS 서비스 기능은 네트워크 별칭(--net-alias)을 통해 부하를 분산시킬 수 있는 로드 밸런스(load balancing)으로 활용할 수 있음

nginx를 이용한 컨테이너 로드 밸런스 구축

- HaProxy, Nginx/Apache Load Balancer 등 외부 서비스와 컨테이너를 결합한 로드 밸런스 구현이 가능
- 부하 분산을 위한 필수 네트워크 기술
- 로드 밸런서는 클라이언트 접속량이 많은 경우 이를 여러 대의 동일 웹 서버 등에 분산시켜 요청 처리를 함
- 이러한 구성은 리소스 활용도를 최적화하고 처리량을 최대화하며 지연 시간을 줄이고 내결함성(fault tolerance) 구성을 보장하기 때문에 안정적인 시스템 운영에 도움이 됨
- nginx를 호스트 운영체제에 설치하고, nginx를 proxy 역할로 구성을 변경하여 nginx로 들어오는 패킷을 연결된 컨테이너에 upstream 한다
- 이때 업스트림 대상을 컨테이너로 설정하여 애플리케이션 요청에 대한 처리를 하게 된다.

```
# 호스트 운영체제에 nginx 설치하고 데몬 구동

sudo apt update
sudo apt install nginx -y
sudo service nginx start

# host의 80번 포트 확인
sudo netstat -nlp | grep 80
```

- docker image bulid

```
# 해당 파일이 있는 곳에서...
Dockerfile  index.phpQQQQQ  index.php2  index.php3

docker build -t phpserver:1.0 .
```

- 컨테이너 생성

```
docker run -itd -p 5001:80 \
        -h nginx-lb01 \
        -v ./lb01:/var/log/apache2 \
        -e SERVER_PORT=5001 \
        --name=nginx-lb01 \
        phpserver:1.0
```

```
docker run -itd -p 5002:80 \
        -h nginx-lb02 \
        -v ./lb02:/var/log/apache2 \
        -e SERVER_PORT=5002 \
        --name=nginx-lb02 \
        phpserver:1.0

docker run -itd -p 5003:80 \
        -h nginx-lb03 \
        -v ./lb03:/var/log/apache2 \
        -e SERVER_PORT=5003 \
        --name=nginx-lb03 \
        phpserver:1.0

# 파일 복사
docker cp index.php3 nginx-
lb01:/var/www/html/index.php
docker cp index.php3 nginx-
lb02:/var/www/html/index.php
docker cp index.php3 nginx-
lb03:/var/www/html/index.php



# nginx.conf 파일 수정
```

```
sudo vim /etc/nginx/nginx.conf
# 기존 내용을 지우고 아래 내용으로
# 기본 listen 되는 80번 포트를 통해 패킷이 들어오면
proxy_pass를 이용하여 http://nginx-lb 도메인명의
upstream으로 이동
# 각 포트별로 라운드 로빈 방식으로 컨테이너에 패킷을
전달
---------------------------------------
events {
        worker_connections 1024;
}

http {
        upstream backend-lb {
                server 127.0.0.1:5001;
                server 127.0.0.1:5002;
                server 127.0.0.1:5003;
        }

        server {
                listen 80 default_server;
                listen [::]:80 default_server;

                # 연결 프락시 정보
                location / {
```

```
                        proxy_pass
http://backend-lb;
                }
        }
}
```

- upstream이란
    - Upstream 서버는 다른 말로 Origin 서버라고도 한다.
    - 즉 여러대의 컴퓨터가 순차적으로 어떤 일을 처리 할 때 어떤 서비스를 받는 서버를 의미한다.
- 연결했던 볼륨의 access.log 확인

```
tail -f lb01/access.log
tail -f lb02/access.log
tail -f lb03/access.log
```

nginx 로드 밸런스 연결 알고리즘

- Round-Robin, RR(기본값)
    - 클라이언트 요청을 서버 가중치를 고려하여 구성된 서버에 균등 배분하는 방식
    - 경로 보장 안됨
- least connections

    - 현재 연결된 클라이언트 수가 가장 적은 서버로 요청 전달
    - 경로 보장 안됨
- IP hash
    - 해시 키를 이용하여 IP별 Index를 생성하여 동일 IP 주소는 동일 서버로의 경로 보장
    - 해당 서버 장애 시 주소 변경됨
    - 균등 배분 보장 안됨
- general hash
    - 사용자가 정의하는 키(IP, Port, URI 문자열등)을 이용한 서버 지정 방식
- least time
    - 요청에 대한 낮은 평균 지연시간을 다음 지시자를 기준으로 계산하여 서버 지정
- random
    - 요청에 대한 무작위 서버를 선택하여 특정 기준을 두고 부합되는 서버 선택도 가능

MacVLAN 네트워크

- 호스트의 네트워크 인터페이스 카드를 가상화해 물리 네트워크 환경을 컨테이너에게 동일하게 제공
- MacVLAN을 사용하면 컨테이너는 물리 네트워크상에서 가상의 MAC 주소를 가지며, 해당 네트워크에 연결

된 다른 장치와의 통신이 가능해 짐

도커 이미지 삭제

이미지 삭제

- 이미지 삭제는 rmi를 사용하면 삭제할 수 있다
  - -a : 사용하지 않은 이미지를 모두 삭제
  - -f : 이미지를 강제로 삭제
  - -no-prune : 중간 이미지를 삭제하지 않음

```
docker rmi [옵션] [이미지 이름:태그]
```

- 강제 삭제는 -f 옵션 사용

```
docker rmi -f [이미지 이름:태그]
```

- 컨테이너가 사용 중인 이미지를 강제로 삭제 할 경우 이미지의 이름이 < none > 으로 변경되어, 이러한 이미지들을 댕글링(dangling) 이미지라고 함
- 댕글링 이미지는 아래 명령어로 확인

```
docker images -f dangling=true
```

- 아래 명령어로 삭제

```
docker image prune
```

특정 이름이 들어간 이미지 삭제

- 예를 들어 ostock으로 시작한 이름이 들어갈 경우 예시

```
docker rmi $(docker images | grep ^ostock | awk '{ print $3 }')
```

도커 명령어 정리

이미지 관련 명령어

- 이미지 검색
    - docker search 이미지명
- 이미지 다운로드
    - docker pull 이미지명:태그
- 이미지 목록 확인
    - docker images
- 이미지 삭제
    - docker rmi 이미지명:태그

컨테이너 관련 명령어

- 컨테이너 생성 및 실행
  - docker run 이미지명:태그
- 컨테이너 목록 확인
  - docker ps (실행 중인 컨테이너), docker ps -a (전체 컨테이너)
- 컨테이너 중지
  - docker stop 컨테이너ID 또는 이름
- 컨테이너 시작: docker start 컨테이너ID 또는 이름
- 컨테이너 재시작: docker restart 컨테이너ID 또는 이름
- 컨테이너 삭제: docker rm 컨테이너ID 또는 이름
- 컨테이너 로그 확인: docker logs 컨테이너ID 또는 이름
- 컨테이너에 접속: docker exec -it 컨테이너ID 또는 이름 /bin/bash

네트워크 관련 명령어

- 네트워크 목록 확인: docker network ls
- 네트워크 생성: docker network create 네트워크명
- 컨테이너에 네트워크 연결: docker network connect 네트워크명 컨테이너ID
- 컨테이너와 네트워크 연결 해제: docker network disconnect 네트워크명 컨테이너ID

볼륨 관련 명령어:

- 볼륨 목록 확인: docker volume ls
- 볼륨 생성: docker volume create 볼륨명
- 컨테이너에 볼륨 연결: docker run -v 볼륨명:컨테이너 내_경로 이미지명
- 볼륨 삭제: docker volume rm 볼륨명

그 외 명령어

- 도커 정보 확인: docker info
- 도커 버전 확인: docker version
- docker cp <컨테이너ID 또는 이름>:<컨테이너 내 파일 경로> <로컬 경로>
- docker cp encore_mysql:/root/stock.sql ~/
- 컨테이너 스펙 정보 보기 : docker stats 컨테이너이름

Dockerfile

- 도커 파일에 사용되는 문법
- FROM : 컨테이너의 BASE IMAGE
- MAINTAINER : 이미지 생성한 사람의 이름 및 정보
- RUN : 컨테이너 빌드를 위해 base images에서 실행할 commands

- COPY : 컨테이너 빌드시 호스트의 파일을 컨테이너로 복사
- ADD : 컨테이너 빌드시 호스트의 파일을 컨테이너로 복사
    - ADD와 COPY의 차이점은 없는 것처럼 보이지만 (복사의 기능은 동일) COPY는 로컬의 파일만 이미지에 추가할 수 있고, ADD는 외부 URL 및 tar 파일에서도 파일을 추가할 수 있다는 점이 다름
    - COPY의 기능이 ADD에 포함되어 있다고 보면 됨
- WORKDIR : 컨테이너 빌드시 명령이 실행될 작업 디렉터리 설정
- ENV : 환경변수 지정
- USER : 명령 및 컨테이너 실행시 적용할 유저 설정
- VOLUME : 파일 또는 디렉터리를 컨테이너의 디렉터리로 마운트
- EXPOSE : 컨테이너 동작 시 외부에서 사용할 포트 지정
    - EXPOSE에서 PORT를 지정해도 컨테이너 실행할 때 반드시 -p 옵션을 사용해야 port가 바인딩됨
    - EXPOSE는 다른 사람에게 해당 해당 도커파일이 어떤 PORT를 사용하는지를 알려주는 성격이 강함

- CMD : 컨테이너 동작 시 자동으로 실행할 서비스나 스크립트 지정
- ENTRYPOINT : CMD와 함께 사용하면서 command 지정 시 사용

docker-compose

docker-compose 설치

```
sudo curl -L
"https://github.com/docker/compose/releases/downloa
d/1.27.0/docker-compose-$(uname -s)-$(uname -m)" -o
/usr/local/bin/docker-compose

 sudo chmod +x /usr/local/bin/docker-compose
 sudo ln -s /usr/local/bin/docker-compose
/usr/bin/docker-compose
```

형식

- YAML 형식에는 공백에 따라 의미가 달라지므로 탭은 의미가 없으며 '공백 두 개로'로 맨 처음 들여쓰기기를 했다면 그 뒤로도 '공백 두 개'가 한 단이 되도록 해야 함
- 이름 뒤에는 반드시 콜론(:)을 붙여야 함

- 해당 줄에 이어서 설정을 기재하려면 콜론 뒤로 공백이 하나 있어야 하고, 이 자리에 공백을 넣는 것을 잊어버리기 쉽기 때문에 오류가 발생한다면 여기부터 확인하는 것이 좋음

컴포즈 파일의 작성 예(이름을 추가)

```
version: "3"
services:
  컨테이너_이름1:
  컨테이너_이름2:
networks:
  네트워크_이름1:
volumes:
  볼륨_이름:1
  볼륨_이름:2
```

컴포즈 파일의 작성 예(설정을 기재)

```
version: "3"
services:                          ← 컨테이너 설정 내용
  컨테이너_이름1:                   ← 컨테이너1의 설정 내용
    image: 이미지_이름
    networks:
      - 네트워크_이름
    ports:
      - 포트_설정
    ...
  컨테이너_이름2:                   ← 컨테이너2의 설정 내용
    image: 이미지_이름
    ...
networks:                          ← 네트워크 설정 내용
  네트워크_이름1:
  ...
volumes:                           ← 볼륨 설정 내용
  볼륨_이름1:
  볼륨_이름2:
  ...
```

항목

- services : 컨테이너를 정의
- networks : 네트워크를 정의
- volumes : 볼륨을 정의

컴포즈 문법

- image : 사용할 이미지를 지정
- service
  - 배포할 서비스를 지정
  - 서비스 이름은 도커 인스턴스에 대한 DNS 엔트리이며, 다른 서비스에서 엑새스하는 데 사용
- network
  - --net
  - 접속할 네트워크 지정
  - 복잡한 토폴로지를 만들 수 있도록 커스텀 네트워크를 지정
  - 타입(host, overlay, macvlan, none)을 지정하지 않았따면 디폴트 타입은 bridge
  - 브리지 네트워크를 사용하면 동일한 넽워크 내 컨테이너 연결을 관리할 수 있고, 이 네트워크는 동일한 도커 데몬 호스트에서 실해되는 컨테이너에만 적용됨
- alias
  - 네트워크 내 서비스에 대한 호스트 별명을 지정
- volums
  - -v, --mount
  - 스토리지 마운트를 지정

- ports
    - -p
    - 포트 설정
    - 시작한 도커 컨테이너가 외부에 노출할 포트 번호를 지정
    - 내부 및 외부 포트를 매핑
- environment
    - -e
    - 환경변수 설정
    - 시박하는 도커 이미지에 환경 변수를 전달
- depends_on
    - 다른 서비스에 대한 의존관계를 정의
- restart
    - 컨테이너 종료 시 재시작 여부를 설정
    - 설정값
        - no : 재시작하지 않는다.
        - always : 항상 재시작한다.
        - on-failure : 프로세스가 0 외의 상태로 종료됐다면 재시작
        - unless-stopeed : 종료 시 재시작하지 않음. 그 외에는 재시작

명령어

- docker-compose up -d
  - 애플리케이션 이미지를 빌드하고 정의된 서비스를 시작
  - 이 명령은 필요한 모든 이미지를 내려받아 배포하고 컨테이너를 시작
  - -d 매개변수는 백그라운드 모드에서 도커를 실행하도록 지정
- docker-compose logs
  - 최신 배포에 대한 모든 정보를 볼 수 있음
- docker-compose logs <service_id>
  - 특정 서비스에 대한 로그를 볼 수 있음
- docker-compose ps
  - 시스템에 배포한 모든 컨테이너 목록을 출력
- docker-compse stop
  - 서비스를 마치고 나서 서비스를 중지
  - 이렇게 하면 컨테이너도 중지됨
- docker-compse down
  - 모든 것을 종료하고 **컨테이너도 모두 제거**

tip

특정 이미지가 사용된 컨테이너 삭제

```
docker rm $(docker ps -a | grep ostock | awk
'{print $1}')
```

특정 이미지 삭제

```
docker rmi $(docker images | grep ostock | awk
'{print $3}')
```