

Практика docker

Подключение к удаленной машине

ssh (*Secure Shell*) - сетевой протокол **удаленного доступа** и **туннелирования** соединения.

Туннелирование означает возможность создания туннеля между локальной и удаленными машинами. Если вы когда-нибудь пользовались vpn (*virtual private network*) то можете представить что такое туннелирование, именно по такому принципу построены все vpn: между выходным узлом расположенным на удаленной машине (например за границей) и вашей локальной машиной выстраивается туннель, через который передается зашифрованный трафик, при этом фактически все ваши локальные запросы дублируются и выполняются на выходном узле. О том как пропробить туннель мы подробнее поговорим ниже.

Удаленный доступ как можно догодаться обеспечивается через командную оболочку (shell). Командная оболочка - естественный в контексте развития вычислительной техники способ взаимодействия оператора с компьютером. Представляет собой текстовый интерфейс взаимодействия: принимает на вход некоторые команды и ключевые слова синтаксиса оболочки и отвечает текстовыми сообщениями.

Наиболее распространенные оболочки в unix подобных системах: `sh` и `bash`, в системах с windows: `cmd` и `powershell`

▼ В windows

что бы открыть командную оболочку мы можем набрать сочетание клавиш `Win + R` и затем набрать в открывшемся окне `powershell`

Удобнее в работе может быть использование [Windows terminal \(github\)](#)

Если не установлен openSSH

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

Синтаксис команды `ssh` для подключения к удалённой машине:

```
ssh -p <port_number> <user_name>@<host_name>
```

В нашем случае для подключения от имени пользователя `user1` необходимо выполнить

```
ssh -p 12074 user1@4.tcp.eu.ngrok.io
```

в пароль `user1@docker`

доступные пользователи `user<id>` где `<id>` - число от 1 до 12, пароль: `user<id>@docker`

Можно использовать `tmux` для лучшего контроля сессий. `tmux ls` - список активных сессий, `tmux a` - подключение к последней сессии

Копирование файлов

```
scp -P <port_number> <source_path> <destination_path>
```

при этом если какой-то из путей на удаленной машине то синтаксис разворачивается следующим образом `<user-name>@<host-name>:/path/to/file`

Запуск докера

▼ Если докер установлен локально

Убедимся что докер работает

После установки в Linux необходимо запустить **демона**

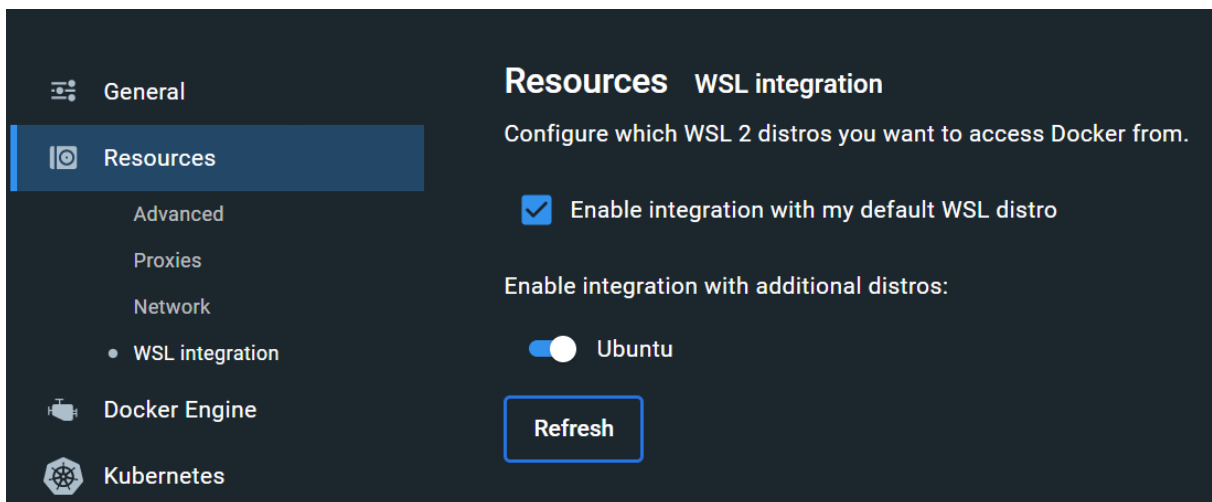
```
sudo systemctl start docker
```

▼ Что бы демон запускался автоматически при старте системы

его надо зарегистрировать в systemd

```
sudo systemctl enable docker.service  
sudo systemctl enable containerd.service
```

В windows после запуска приложения docker-desktop необходимо включить интеграцию с wsl, (по умолчанию командой `wsl --install` ставится Ubuntu)



После этого можно залогиниться в Ubuntu

```
wsl -d Ubuntu
```

Убеждаемся что docker доступен

```
docker -v
```

▼ Для пользователей Linux что бы не набирать каждый раз `sudo`

```
sudo usermod -aG docker $USER
```

Первый запуск

```
docker run hello-world
```



Команда `run` создает и запускает **новый** контейнер из образа (`start` - запускает существующий но остановленный контейнер, `exec` - позволяет выполнить команду на активном контейнере).

синтаксис команды `docker run <image_name>`

`<image_name>` - имя **образа контейнера**, существование которого проверяется в локальном хранилище, и если его там нет то по умолчанию производится поиск по адресу docker.io/library - официальное хранилище образов, и затем выполняется команда `pull` после чего выполняется команда `run`. Если у нас есть собственное хранилище, то мы можем указать полный путь к нужному образу в качестве аргумента `<image_name>`, кроме того через двоеточие мы можем указать `tag`, если мы его не указываем то по-умолчанию присваивается `latest`.

Фактически выполняется следующее

```
docker pull docker.io/library/hello-world:latest
docker run hello-world
```

Образ - файл содержащий в себе исходное состояние будущего контейнера, шаблон, с помощью которого можно создавать контейнеры.

Список локальных образов

```
docker images
```

чтобы удалить образ

```
docker rmi <image_name or id>
# вместе со всеми контейнерами
docker rmi -f <image_name or id>
```

Список активных контейнеров


```
docker ps
# всех контейнеров
docker ps -a
```

Имена контейнерам присваиваются автоматически, если мы хотим задать конкретное имя можем сделать это ключом

```
--name <container_name>
```

Чтобы удалить контейнер

```
docker rm <container_name or id>
```

A terminal window with a dark background and light text. The prompt is '~> docker run hello-world'. The output reads: 'Hello from Docker! This message shows that your installation appears to be working correctly. To generate this message, Docker took the following steps: 1. The Docker client contacted the Docker daemon. 2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64) 3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading. 4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal. To try something more ambitious, you can run an Ubuntu container with: \$ docker run -it ubuntu bash Share images, automate workflows, and more with a free Docker ID: https://hub.docker.com/ For more examples and ideas, visit: https://docs.docker.com/get-started/'. The prompt at the bottom is '~> docker p'.

Заглянуть внутрь контейнера можно с помощью ключа `-it`

```
docker run -it alpine sh
```

Информация о версии операционной системы хранится в файле `/etc/os-release`

Версию ядра можно посмотреть спомощью `uname -srm`

Проброс портов

Слушаемые порты можно посмотреть командой

```
ss -ntpl
```

пробрасывание портов ключом `-p`

```
-p <host_port>:<app_in_container_port>
```

```
docker run -d -p 10000:8888 jupyter/minimal-notebook
```

Ключ `-d` означает что запускаем в режиме detached

Проверить доступность портов можно командами `nc` (netcat) или `nmap` (<https://nmap.org/download.html#windows>)

```
nmap localhost
```

```
nc localhost 8080
```

С помощью Powershell

```
Test-Netconnection -ComputerName Localhost -port 8080
```

Создание туннеля

Синтаксис для пробрасывания порта на локальную машину

```
ssh -p <ssh_port_num> <user>@<host> -L <local_port>:localhost:<remote_port>
```

Аналогично ключом `-R` можно пробросить порт на удаленную машину

Пример использования

```
ssh -p 12074 user1@4.tcp.eu.ngrok.io -L 8080:localhost:10000
```

теперь можем подключиться по тоннелю к нашему контейнеру по адресу <http://localhost:8080>

Чтобы узнать токен и залогиниться заглянем в логи

```
docker logs <container_name>
```

Что бы отправлять сообщения между машинами/пользователями можно воспользоваться одной из команд

```
wall, who, write user_name pts/<num>
```

Проброс папок

Пробрасывание папки

```
-v /host/data:/data
```

Создадим еще один юпитер с проброшенной папкой

Внутри контейнера глобальный путь можем узнать с помощью команды `pwd`

```
docker run -p 10000:8888 -v ~/jupyter_data:/home/jovyan/work jupyter/minimal-notebook
```

Создание собственного контейнера

Что бы собрать свой контейнер надо для начала понять как работают другие.

Вы могли обратить внимание что образ скачивается какое-то время, а контейнер поднимается довольно быстро. Например контейнер с юпитером занимает 1.55 гб дискового пространства, однако создание нового контейнера занимает меньше секунды.

Замерим скорость создания нового контейнера

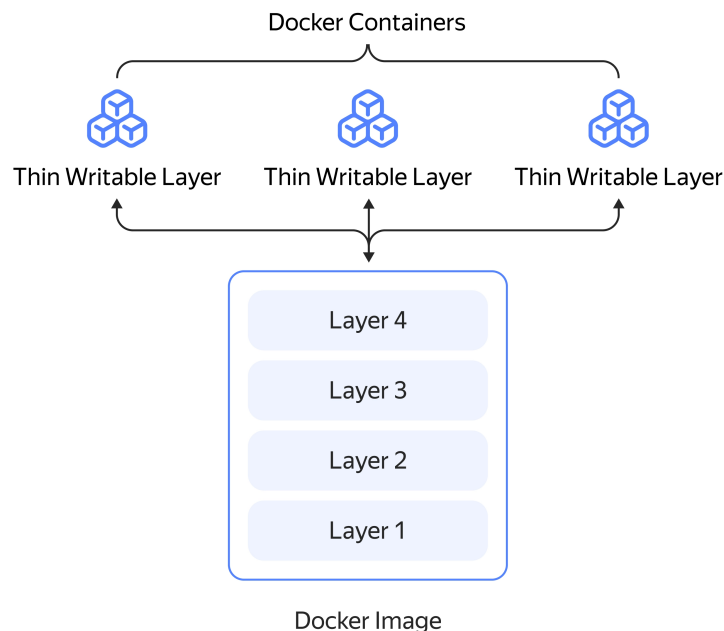
```
date +%T.%N && docker run -d jupyter/minimal-notebook && date +%T.%N
07:52:52.260789580
486e7f5dd54c329c439f771596aba2a64f303e1d8c0d6041ca4efc8e2ac4af43
07:52:52.933474506
```

время выполнения 672 мс

Если бы при этом происходило физическое копирование файлов то для этого скорость записи на диске должна быть больше 18 Гбит/с что физически могло бы быть достижимо разве что на очень дорогих серверных ssd формата PSI-express, потому как пропускная способность обычного SATA v3.0 до 6 Гбит/с.

Нетрудно догадаться что контейнер хранит только информацию о своих изменениях относительно образа (diff). Идея использовать диффы для отслеживания состояния чего либо пришла из систем версий контроля исходного кода: последовательное наложение изменений при соблюдении определенной культуры этой процедуры позволяет очень гибко отслеживать и управлять состоянием исходного кода, откатываться на предыдущие версии при необходимости и находить критические баги, если известно что он появился только после определенного изменения, что бы не искать его во всем коде. По такому же принципу работают системы контроля версий наборов данных, весов моделей и даже некоторые файловые системы (BetrFS).

Принцип построения образов докера такой же, из отдельных слоев:



Заглянуть внутрь образа можно с помощью утилиты `dive`


```

dpkg-dev dpkg \
expat-dev \
findutils \
gcc \
gdbm-dev \
libc-dev \
libffi-dev \
libnsl-dev \
libtirpc-dev \
linux-headers \
make \
ncurses-dev \
openssl-dev \
pax-utils \
readline-dev \
sqlite-dev \
tcl-dev \
tk \
tk-dev \
util-linux-dev \
xz-dev \
zlib-dev \
; \
\
wget -O python.tar.xz "https://www.python.org/ftp/python/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz"; \
wget -O python.tar.xz.asc "https://www.python.org/ftp/python/${PYTHON_VERSION%%[a-z]*}/Python-${PYTHON_VERSION}.tar.xz.asc"; \
GNUPGHOME="$(mktemp -d)"; export GNUPGHOME; \
gpg --batch --keyserver hkps://keys.openpgp.org --recv-keys "$GPG_KEY"; \
gpg --batch --verify python.tar.xz.asc python.tar.xz; \
gpgconf --kill all; \
rm -rf "$GNUPGHOME" python.tar.xz.asc; \
mkdir -p /usr/src/python; \
tar --extract --directory /usr/src/python --strip-components=1 --file python.tar.xz; \
rm python.tar.xz; \
\
cd /usr/src/python; \
gnuArch="$(dpkg-architecture --query DEB_BUILD_GNU_TYPE)"; \
./configure \
--build="$gnuArch" \
--enable-loadable-sqlite-extensions \
--enable-optimizations \
--enable-option-checking=fatal \
--enable-shared \
--with-lto \
--with-system-expat \
--without-ensurepip \
; \
nproc="$(nproc)"; \
# set thread stack size to 1MB so we don't segfault before we hit sys.getrecursionlimit()
# https://github.com/alpinelinux/aports/commit/2026e1259422d4e0cf92391ca2d3844356c649d0
EXTRA_CFLAGS="-DTHREAD_STACK_SIZE=0x100000"; \
LDFLAGS="${LDFLAGS:-wl},--strip-all"; \
make -j "$nproc" \
"EXTRA_CFLAGS=${EXTRA_CFLAGS:-}" \
"LDFLAGS=${LDFLAGS:-}" \
"PROFILE_TASK=${PROFILE_TASK:-}" \
; \
# https://github.com/docker-library/python/issues/784
# prevent accidental usage of a system installed libpython of the same version
rm python; \
make -j "$nproc" \
"EXTRA_CFLAGS=${EXTRA_CFLAGS:-}" \
"LDFLAGS=${LDFLAGS:-wl},-rpath='\${$ORIGIN}/../lib" \
"PROFILE_TASK=${PROFILE_TASK:-}" \
python \
; \
make install; \
\
cd /; \
rm -rf /usr/src/python; \
\
find /usr/local -depth \
\{ \
\{ \
\{ -type d -a \{ -name test -o -name tests -o -name idle_test \} \} \
-o \{ -type f -a \{ -name '*.pyc' -o -name '*.pyo' -o -name 'libpython*.a' \} \} \
\} -exec rm -rf '{}' + \

```



```

; \
\
find /usr/local -type f -executable -not \( -name '*tkinter*' \) -exec scanelf --needed --nobanner --format '%n#p' '{}' ';' \
| tr ',' '\n' \
| sort -u \
| awk 'system("[ -e /usr/local/lib/" $1 " ]") == 0 { next } { print "so:" $1 }' \
| xargs -rt apk add --no-network --virtual .python-rundeps \
; \
apk del --no-network .build-deps; \
\
python3 --version

# make some useful symlinks that are expected to exist ("/usr/local/bin/python" and friends)
RUN set -eux; \
for src in idle3 pydoc3 python3 python3-config; do \
dst="$(echo "$src" | tr -d 3)"; \
[ -s "/usr/local/bin/$src" ]; \
[ ! -e "/usr/local/bin/$dst" ]; \
ln -svT "$src" "/usr/local/bin/$dst"; \
done

# if this is called "PIP_VERSION", pip explodes with "ValueError: invalid truth value '<VERSION>'"
ENV PYTHON_PIP_VERSION 23.1.2
# https://github.com/docker-library/python/issues/365
ENV PYTHON_SETUPTOOLS_VERSION 65.5.1
# https://github.com/pypa/get-pip
ENV PYTHON_GET_PIP_URL https://github.com/pypa/get-pip/raw/9af82b715db434abb94a0a6f3569f43e72157346/public/get-pip.py
ENV PYTHON_GET_PIP_SHA256 45a2bb8bf2bb5eff16fdd00faef6f29731831c7c59bd9fc2bf1f3bed511ff1fe

RUN set -eux; \
\
wget -O get-pip.py "$PYTHON_GET_PIP_URL"; \
echo "$PYTHON_GET_PIP_SHA256 *get-pip.py" | sha256sum -c -; \
\
export PYTHONDONTWRITEBYTECODE=1; \
\
python get-pip.py \
--disable-pip-version-check \
--no-cache-dir \
--no-compile \
"pip==$PYTHON_PIP_VERSION" \
"setuptools==$PYTHON_SETUPTOOLS_VERSION" \
; \
rm -f get-pip.py; \
\
pip --version

CMD ["python3"]

```

Таким образом минимальный докер файл должен иметь хотябы команду `FROM`

```
FROM alpine
```

Попробуем собрать

```
docker build -t dummy .
```

Запустим теперь что-нибудь внутри контейнера

напишем простейший скрипт назовем `main.py`

```
print("hello from docker again")
```

рядом создадим `Dockerfile`

```
FROM python:alpine
COPY . .
CMD ["python3", "main.py"]
```

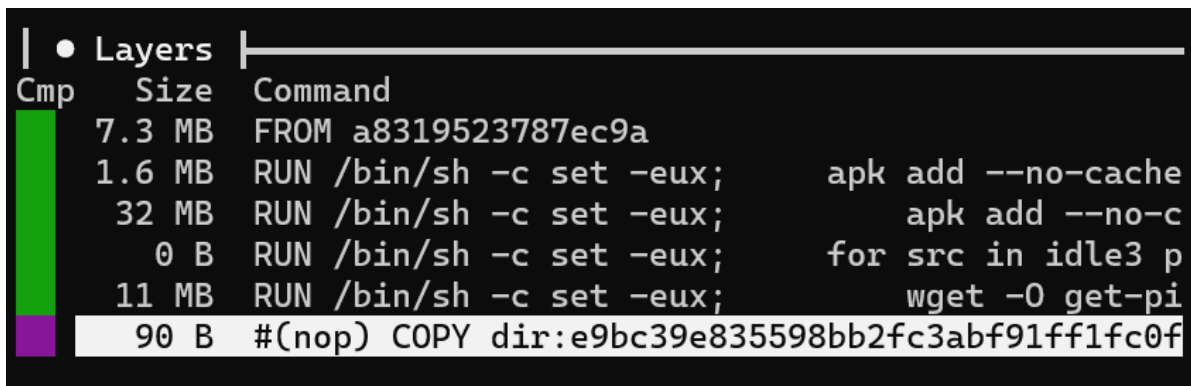
Соберём

```
docker build -t hello_again .
```

Запустим и убедимся что скрипт выполняется

```
docker run hello_again
```

Видно что добавился еще один слой командой `COPY`



Сmp	Size	Command
	7.3 MB	FROM a8319523787ec9a
	1.6 MB	RUN /bin/sh -c set -eux; apk add --no-cache
	32 MB	RUN /bin/sh -c set -eux; apk add --no-c
	0 B	RUN /bin/sh -c set -eux; for src in idle3 p
	11 MB	RUN /bin/sh -c set -eux; wget -O get-pi
	90 B	##(nop) COPY dir:e9bc39e835598bb2fc3abf91ff1fc0f

Связывание контейнеров внутри сети

Напишем простое сетевое приложение которое будет читать файл с диска имя которого получено через сокет

Сервер:

```
import socket
import os

HOST = "127.0.0.1"
PORT = 5000
BUFFER_SIZE = 2048
DIR_NAME = "./input"

# Запускаем сервер и слушаем сокет
with socket.socket() as serv:
    serv.bind((HOST, PORT))
    serv.listen()
    print("server is up")
    conn, addr = serv.accept()
    with conn:
        print(f"Connectted by {addr}")
        while True:
            data = conn.recv(BUFFER_SIZE)
            if not data:
                break

            # Интерпретируем полученные из сокета байты
            # как имя файла, если такой файл существует
            # читаем его содержимое и отправляем обратно
            # через сокет, предварительно закодировав

            file_name = str(data, 'utf-8').strip()
```

```
print(f"resived {file_name}")
path = os.path.join(DIR_NAME, file_name)
if os.path.exists(path):
    with open(path, "r") as file:
        text = file.read()
else:
    text = f"Error, file {path} not found"

conn.sendall(bytes(text+"\n", "utf-8"))
```

Запустим локально и убедимся что сервер работает

```
python3 main.py
```

Если мы используем `tmux` разделим окно терминала: перейдем в командный режим сочетанием клавиш `ctrl+b` а затем `~` для горизонтального разделения окна терминала или `%` для вертикального. Что бы переходить между окнами так же перейдем в командный режим `Ctrl+b` а затем стрелками переключимся на нужное окно, если сочетание `Ctrl+b` не отпускать, то стрелками можно изменять положение разделителя.

В соседнем терминале запустим клиент - приложение `netcat`

```
nc localhost 5000
```

и отправим что-нибудь на вход

```
user7@neurolab:~/docker_workshop$ python3 main.py
server is up
Connectted by ('127.0.0.1', 54184)
resived file1

user7@neurolab:~$ nc localhost 5000
file1
Error, file ./input/file1 not found
```

создадим недостающую папку и файл

```
mkdir input
echo "hello from socket" > input/file1
```

Повторим

```
user7@neurolab:~/docker_workshop$ python3 main.py
server is up
Connectted by ('127.0.0.1', 60924)
resived file1

user7@neurolab:~$ nc localhost 5000
file1
hello from socket
```

Соберем в контейнер как мы это уже умеем

```
FROM python:alpine
COPY . .
CMD ["python3", "main.py"]
```

```
docker build -t server .
```

Убедимся что всё работает

```
docker run -e PYTHONUNBUFFERED=1 server
```

с помощью ключа `-e` мы задаем переменные окружения. `PYTHONUNBUFFERED=1` позволяет нам увидеть в логах докера результаты команды `print`. Обычно для логов используется специальный модуль `logging` и каждое сообщение отправленное с его помощью помечается определенным уровнем `INFO`, `WARN`, `ERROR` и т.д., а сообщения вызванные командой `print` обычно не имеют уровня и не логируются.

Контейнер запустился но порты мы не пробрасывали и достучаться до него командой `netcat` из хост системы не получается.

Исправить эту ситуацию мы можем если примонтируем контейнер внутри нашей локальной сети:

```
docker run -e PYTHONUNBUFFERED=1 server --network=host
```

Напишем простейший скрипт клиент

Клиент будет посылать одно сообщение серверу и возвращать его ответ в виде ascii-art

```
import socket
import pyfiglet

HOST = "127.0.0.1"
PORT = 5000
MESSAGE = "1"

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(bytes(MESSAGE, "utf-8"))
    data = s.recv(1024)

art = pyfiglet.figlet_format(str(data, "utf-8").strip())
print(art)
```

Что бы собрать зависимости в одном файле

```
python -m venv env
source .env/bin/activate
(env) $: pip install package_name
...
(env) pip freeze > requirements.txt
```

Распишем по контейнерам

```
docker build -t myapp:1.0 .
```

Что бы взаимодействовать с контейнером мы можем пробросить из него порты в домашнюю систему -p, примонтировать папку -v, задать переменные окружения -e, перейти в интерактивный режим и получить доступ к терминалу -it

Чтобы не запускать контейнеры по очереди и не вводить каждый раз все необходимые ключи, удобнее воспользоваться плагином docker compose описав все необходимое в docker-compose.yml а затем запускать командой `docker compose up`

Пример

```
version: "3.7"

networks:
  kafka-net:
    name: my-net
    driver: bridge

services:
  server:
    build:
      context: ./server
    image: server
    container_name: 'server'
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - my-net
  app:
    build:
      context: ./app
    image: myapp
    container_name: 'app'
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - my-net
```

docker-compose изначально был отдельным приложением но со временем стал стандартом и поставляется как плагин вместе с докером и доступен из коробки.