

КАК СТАТЬ АВТОРОМ

Где работать в следующем году



RU VDS

Лучшее предложение на рынке
VPS хостинг с Windows от **523 ₽/месяц**

Лицензия на ОС включена в стоимость

**1923.28**

Рейтинг

RUVDS.comVDS/VPS-хостинг. Скидка 15% по коду **HABR15****ru_vds**

15 фев 2019 в 16:00

Изучаем Docker, часть 3: файлы Dockerfile

12 мин

558K

Блог компании RUVDS.com, Разработка веб-сайтов*, Виртуализация*

Перевод

Автор оригинала: Jeff Hale

В переводе третьей части серии материалов, посвящённых Docker, мы продолжим вдохновляться выпечкой, а именно – бубликами. Нашей сегодняшней основной темой будет работа с файлами Dockerfile. Мы разберём инструкции, которые используются в этих файлах.

- [Часть 1: основы](#)
- [Часть 2: термины и концепции](#)
- [Часть 3: файлы Dockerfile](#)
- [Часть 4: уменьшение размеров образов и ускорение их сборки](#)
- [Часть 5: команды](#)
- [Часть 6: работа с данными](#)



Бублики – это инструкции в файле Dockerfile

Образы Docker

Вспомните о том, что контейнер Docker – это образ Docker, вызванный к жизни. Это – самодостаточная операционная система, в которой имеется только самое необходимое и код приложения.

Образы Docker являются результатом процесса их сборки, а контейнеры Docker – это выполняющиеся образы. В самом сердце Docker находятся файлы Dockerfile. Подобные файлы сообщают Docker о том, как собирать образы, на основе которых создаются контейнеры.

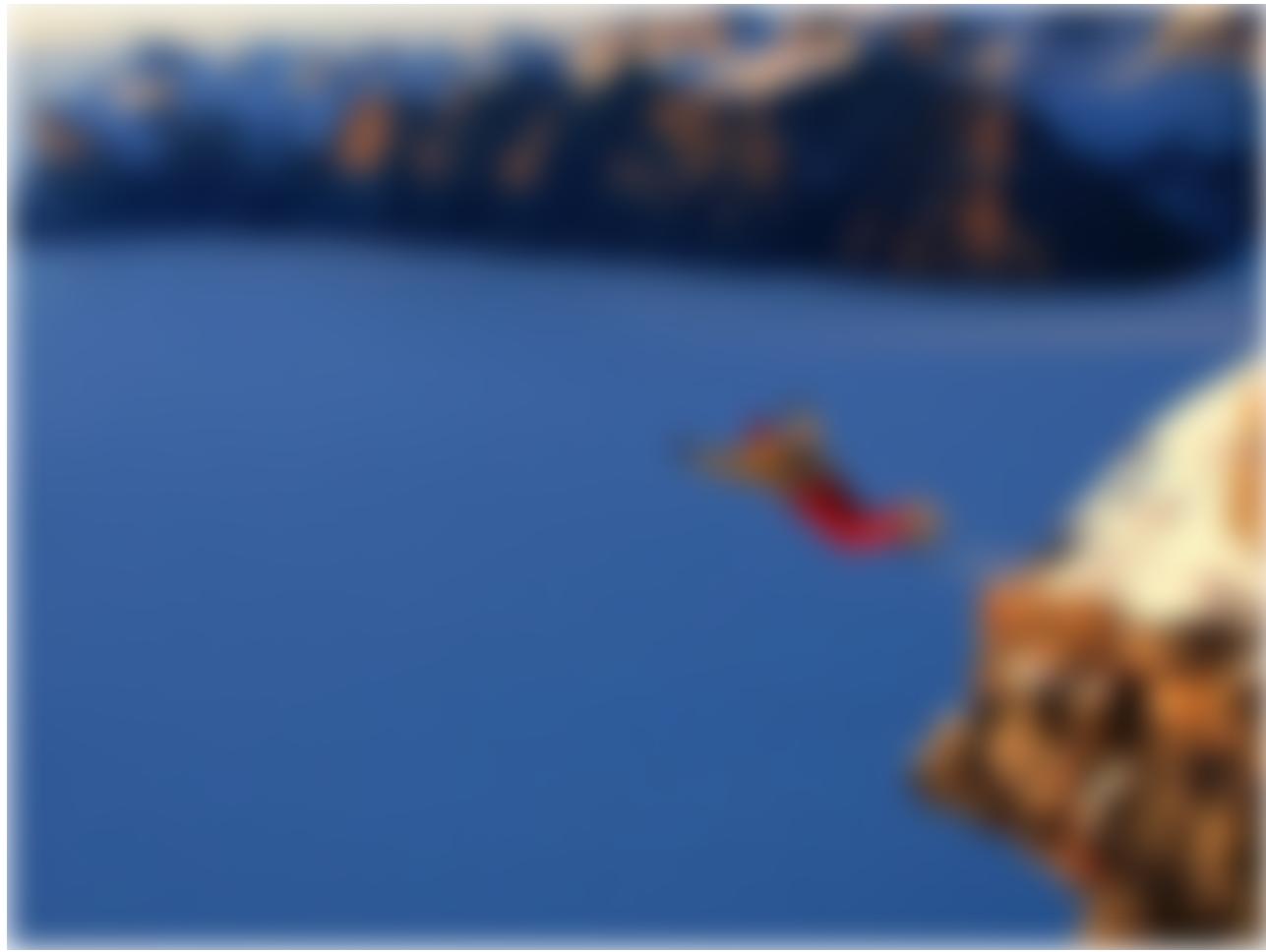
Каждому образу Docker соответствует файл, который называется Dockerfile. Его имя записывается именно так – без расширения. При запуске команды docker build для создания нового образа подразумевается, что Dockerfile находится в текущей рабочей директории. Если этот файл находится в каком-то другом месте, его расположение можно указать с использованием флага `-f`.

Контейнеры, как мы выяснили в первом материале этой серии, состоят из слоёв. Каждый

слой, кроме последнего, находящегося поверх всех остальных, предназначен только для чтения. Dockerfile сообщает системе Docker о том, какие слои и в каком порядке надо добавить в образ.

Каждый слой, на самом деле, это всего лишь файл, который описывает изменение состояния образа в сравнении с тем состоянием, в котором он пребывал после добавления предыдущего слоя. В Unix, кстати, практически всё что угодно – это файл.

Базовый образ – это то, что является исходным слоем (или слоями) создаваемого образа. Базовый образ ещё называют родительским образом.



Базовый образ – это то, с чего начинается образ Docker

Когда образ загружается из удалённого репозитория на локальный компьютер, то физически скачиваются лишь слои, которых на этом компьютере нет. Docker стремится экономить пространство и время путём повторного использования существующих слоёв.

Файлы Dockerfile

В файлах Dockerfile содержатся инструкции по созданию образа. С них, набранных заглавными буквами, начинаются строки этого файла. После инструкций идут их

аргументы. Инструкции, при сборке образа, обрабатываются сверху вниз. Вот как это выглядит:

```
FROM ubuntu:18.04
COPY . /app
```

Слои в итоговом образе создают только инструкции `FROM`, `RUN`, `COPY`, и `ADD`. Другие инструкции что-то настраивают, описывают метаданные, или сообщают Docker о том, что во время выполнения контейнера нужно что-то сделать, например – открыть какой-то порт или выполнить какую-то команду.

Здесь мы исходим из предположения, в соответствии с которым используется образ Docker, основанный на Unix-подобной ОС. Конечно, тут можно воспользоваться и образом, основанным на Windows, но использование Windows – это менее распространённая практика, работать с такими образами сложнее. В результате, если у вас есть такая возможность, пользуйтесь Unix.

Для начала приведём список инструкций `Dockerfile` с краткими комментариями.

Дюжина инструкций `Dockerfile`

1. `FROM` – задаёт базовый (родительский) образ.
2. `LABEL` – описывает метаданные. Например – сведения о том, кто создал и поддерживает образ.
3. `ENV` – устанавливает постоянные переменные среды.
4. `RUN` – выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
5. `COPY` – копирует в контейнер файлы и папки.
6. `ADD` – копирует файлы и папки в контейнер, может распаковывать локальные `.tar`-файлы.
7. `CMD` – описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция `CMD`.
8. `WORKDIR` – задаёт рабочую директорию для следующей инструкции.
9. `ARG` – задаёт переменные для передачи Docker во время сборки образа.
10. `ENTRYPOINT` – предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.

11. `EXPOSE` – указывает на необходимость открыть порт.
12. `VOLUME` – создаёт точку монтирования для работы с постоянным хранилищем.

Теперь поговорим об этих инструкциях.

Инструкции и примеры их использования

Простой Dockerfile

Dockerfile может быть чрезвычайно простым и коротким. Например – таким:

```
FROM ubuntu:18.04
```

Инструкция FROM

Файл Dockerfile должен начинаться с инструкции `FROM`, или с инструкции `ARG`, за которой идёт инструкция `FROM`.

Ключевое слово `FROM` сообщает Docker о том, чтобы при сборке образа использовался базовый образ, который соответствует предоставленному имени и тегу. Базовый образ, кроме того, ещё называют родительским образом.

В этом примере базовый образ хранится в репозитории `ubuntu`. `Ubuntu` – это название официального репозитория Docker, предоставляющего базовую версию популярной ОС семейства Linux, которая называется `Ubuntu`.

Обратите внимание на то, что рассматриваемый Dockerfile включает в себя тег `18.04`, уточняющий то, какой именно базовый образ нам нужен. Именно этот образ и будет загружен при сборке нашего образа. Если тег в инструкцию не включён, тогда Docker исходит из предположения о том, что требуется самый свежий образ из репозитория. Для того чтобы яснее выразить свои намерения, автору Dockerfile рекомендуется указывать то, какой именно образ ему нужен.

Когда вышеописанный Dockerfile используется на локальной машине для сборки образа в первый раз, Docker загрузит слои, определяемые образом `ubuntu`. Их можно представить наложенными друг на друга. Каждый следующий слой представляет собой файл, описывающий отличия образа в сравнении с тем его состоянием, в котором он был после добавления в него предыдущего слоя.

При создании контейнера слой, в который можно вносить изменения, добавляется поверх всех остальных слоёв. Данные, находящиеся в остальных слоях, можно только читать.

Структура контейнера (взято из документации)

Docker, ради эффективности, использует стратегию копирования при записи. Если слой в образе существует на предыдущем уровне и какому-то слою нужно произвести чтение данных из него, Docker использует существующий файл. При этом ничего загружать не нужно.

Когда образ выполняется, если слой нужно модифицировать средствами контейнера, то соответствующий файл копируется в самый верхний, изменяемый слой. Для того чтобы узнать подробности о стратегии копирования при записи, взгляните на этот материал из документации Docker.

Продолжим рассмотрение инструкций, которые используются в `Dockerfile`, приведя пример такого файла с более сложной структурой.

Более сложный `Dockerfile`

Хотя файл `Dockerfile`, который мы только что рассмотрели, получился аккуратным и понятным, он устроен слишком просто, в нём используется всего одна инструкция. Кроме

того, там нет инструкций, вызываемых во время выполнения контейнера. Взглянем на ещё один файл, который собирает маленький образ. В нём имеются механизмы, определяющие команды, вызываемые во время выполнения контейнера.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
ENV ADMIN="jeff"
RUN apk update && apk upgrade && apk add bash
COPY . ./app
ADD https://raw.githubusercontent.com/disco-vid/master/sample_vids/vid/
/my_app_directory
RUN ["mkdir", "/a_directory"]
CMD ["python", "./my_script.py"]
```

Возможно, на первый взгляд этот файл может показаться довольно сложным. Поэтому давайте с ним разберёмся.

Базой этого образа является официальный образ Python с тегом 3.7.2-alpine3.8. Проанализировав этот код можно увидеть, что данный базовый образ включает в себя Linux, Python, и, по большому счёту, этим его состав и ограничивается. Образы OS Alpine весьма популярны в мире Docker. Дело в том, что они отличаются маленькими размерами, высокой скоростью работы и безопасностью. Однако образы Alpine не отличаются широкими возможностями, характерными для обычных операционных систем. Поэтому для того, чтобы собрать на основе такого образа что-то полезное, создателю образа нужно установить в него необходимые ему пакеты.

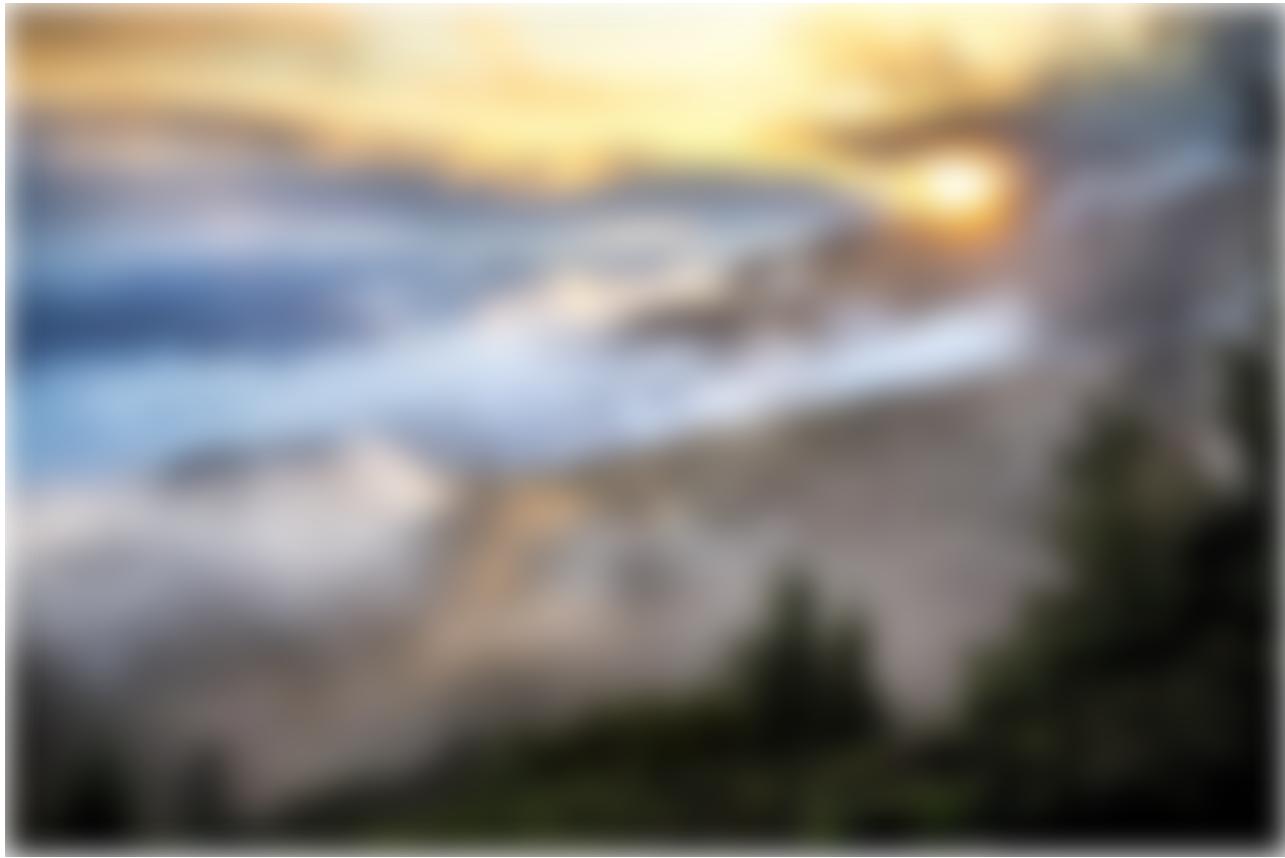
Инструкция LABEL



Метки

Инструкция LABEL (метка) позволяет добавлять в образ метаданные. В случае с рассматриваемым сейчас файлом, она включает в себя контактные сведения создателя образа. Объявление меток не замедляет процесс сборки образа и не увеличивает его размер. Они лишь содержат в себе полезную информацию об образе Docker, поэтому их рекомендуется включать в файл. Подробности о работе с метаданными в Dockerfile можно прочитать [здесь](#).

| Инструкция ENV



Окружающая среда

Инструкция ENV позволяет задавать постоянные переменные среды, которые будут доступны в контейнере во время его выполнения. В предыдущем примере после создания контейнера можно пользоваться переменной ADMIN .

Инструкция ENV хорошо подходит для задания констант. Если вы используете некое значение в Dockerfile несколько раз, скажем, при описании команд, выполняющихся в контейнере, и подозреваете, что, возможно, вам когда-нибудь придётся сменить его на другое, его имеет смысл записать в подобную константу.

Надо отметить, что в файлах Dockerfile часто существуют разные способы решения одних и тех же задач. Что именно использовать – это вопрос, на решение которого влияет стремление к соблюдению принятых в среде Docker методов работы, к обеспечению прозрачности решения и его высокой производительности. Например, инструкции RUN , CMD и ENTRYPOINT служат разным целям, но все они используются для выполнения команд.

Инструкция RUN



Инструкция *RUN*

Инструкция *RUN* позволяет создать слой во время сборки образа. После её выполнения в образ добавляется новый слой, его состояние фиксируется. Инструкция *RUN* часто используется для установки в образы дополнительных пакетов. В предыдущем примере инструкция *RUN apk update && apk upgrade* сообщает Docker о том, что системе нужно обновить пакеты из базового образа. Вслед за этими двумя командами идёт команда *&& apk add bash*, указывающая на то, что в образ нужно установить *bash*.

То, что в командах выглядит как *apk* – это сокращение от *Alpine Linux package manager* (менеджер пакетов Alpine Linux). Если вы используете базовый образ какой-то другой ОС семейства Linux, тогда вам, например, при использовании *Ubuntu*, для установки пакетов может понадобиться команда вида *RUN apt-get*. Позже мы поговорим о других способах установки пакетов.

Инструкция *RUN* и схожие с ней инструкции – такие, как *CMD* и *ENTRYPOINT*, могут быть использованы либо в *execs*-форме, либо в *shell*-форме. *Execs*-форма использует синтаксис, напоминающий описание JSON-массива. Например, это может выглядеть так:

```
RUN ["my_executable", "my_first_param1", "my_second_param2"] .
```

В предыдущем примере мы использовали *shell*-форму инструкции *RUN* в таком виде: *RUN apk update && apk upgrade && apk add bash*.

Позже в нашем Dockerfile использована ехес-форма инструкции RUN , в виде RUN ["mkdir", "/a_directory"] для создания директории. При этом, используя инструкцию в такой форме, нужно помнить о необходимости оформления строк с помощью двойных кавычек, как это принято в формате JSON.

Инструкция COPY



Инструкция COPY

Инструкция COPY представлена в нашем файле так: COPY . ./app . Она сообщает Docker о том, что нужно взять файлы и папки из локального контекста сборки и добавить их в текущую рабочую директорию образа. Если целевая директория не существует, эта инструкция её создаст.

Инструкция ADD

Инструкция ADD позволяет решать те же задачи, что и COPY , но с ней связана ещё пара вариантов использования. Так, с помощью этой инструкции можно добавлять в контейнер файлы, загруженные из удалённых источников, а также распаковывать локальные .tar-файлы.

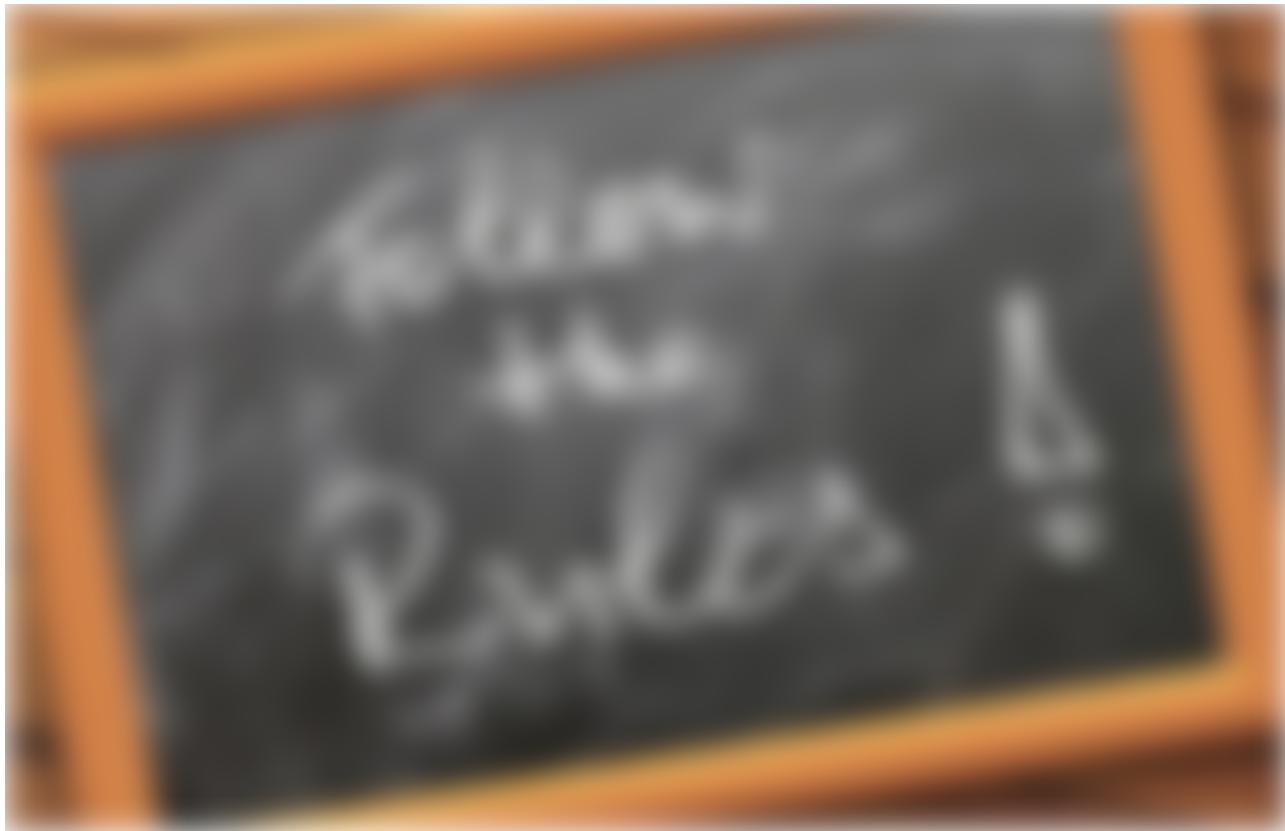
В этом примере инструкция ADD была использована для копирования файла, доступного

по URL, в директорию контейнера `my_app_directory`. Надо отметить, однако, что документация Docker не рекомендует использование подобных файлов, полученных по URL, так как удалить их нельзя, и так как они увеличивают размер образа.

Кроме того, документация предлагает везде, где это возможно, вместо инструкции ADD использовать инструкцию COPY для того, чтобы сделать файлы Dockerfile понятнее. Полагаю, команде разработчиков Docker стоило бы объединить ADD и COPY в одну инструкцию для того, чтобы тем, кто создаёт образы, не приходилось бы помнить слишком много инструкций.

Обратите внимание на то, что инструкция ADD содержит символ разрыва строки – \ . Такие символы используются для улучшения читабельности длинных команд путём разбиения их на несколько строк.

| Инструкция CMD



Инструкция CMD

Инструкция CMD предоставляет Docker команду, которую нужно выполнить при запуске контейнера. Результаты выполнения этой команды не добавляются в образ во время его сборки. В нашем примере с помощью этой команды запускается скрипт `my_script.py` во время выполнения контейнера.

Вот ещё кое-что, что нужно знать об инструкции `CMD`:

- В одном файле `Dockerfile` может присутствовать лишь одна инструкция `CMD`. Если в файле есть несколько таких инструкций, система проигнорирует все кроме последней.
- Инструкция `CMD` может иметь ехес-форму. Если в эту инструкцию не входит упоминание исполняемого файла, тогда в файле должна присутствовать инструкция `ENTRYPOINT`. В таком случае обе эти инструкции должны быть представлены в формате `JSON`.
- Аргументы командной строки, передаваемые `docker run`, переопределяют аргументы, предоставленные инструкции `CMD` в `Dockerfile`.

Ещё более сложный `Dockerfile`

Рассмотрим ещё один файл `Dockerfile`, в котором будут использованы некоторые новые команды.

```
FROM python:3.7.2-alpine3.8
LABEL maintainer="jeffmshale@gmail.com"
# Устанавливаем зависимости
RUN apk add --update git
# Задаём текущую рабочую директорию
WORKDIR /usr/src/my_app_directory
# Копируем код из локального контекста в рабочую директорию образа
COPY . .
# Задаём значение по умолчанию для переменной
ARG my_var=my_default_value
# Настраиваем команду, которая должна быть запущена в контейнере во время его выполнения
ENTRYPOINT ["python", "./app/my_script.py", "my_var"]
# Открываем порты
EXPOSE 8000
# Создаём том для хранения данных
VOLUME /my_volume
```

В этом примере, кроме прочего, вы можете видеть комментарии, которые начинаются с символа `#`.

Одно из основных действий, выполняемых средствами `Dockerfile` – это установка пакетов. Как уже было сказано, существуют различные способы установки пакетов с помощью инструкции `RUN`.

Пакеты в образ Alpine Docker можно устанавливать с помощью apk . Для этого, как мы уже говорили, применяется команда вида RUN apk update && apk upgrade && apk add bash .

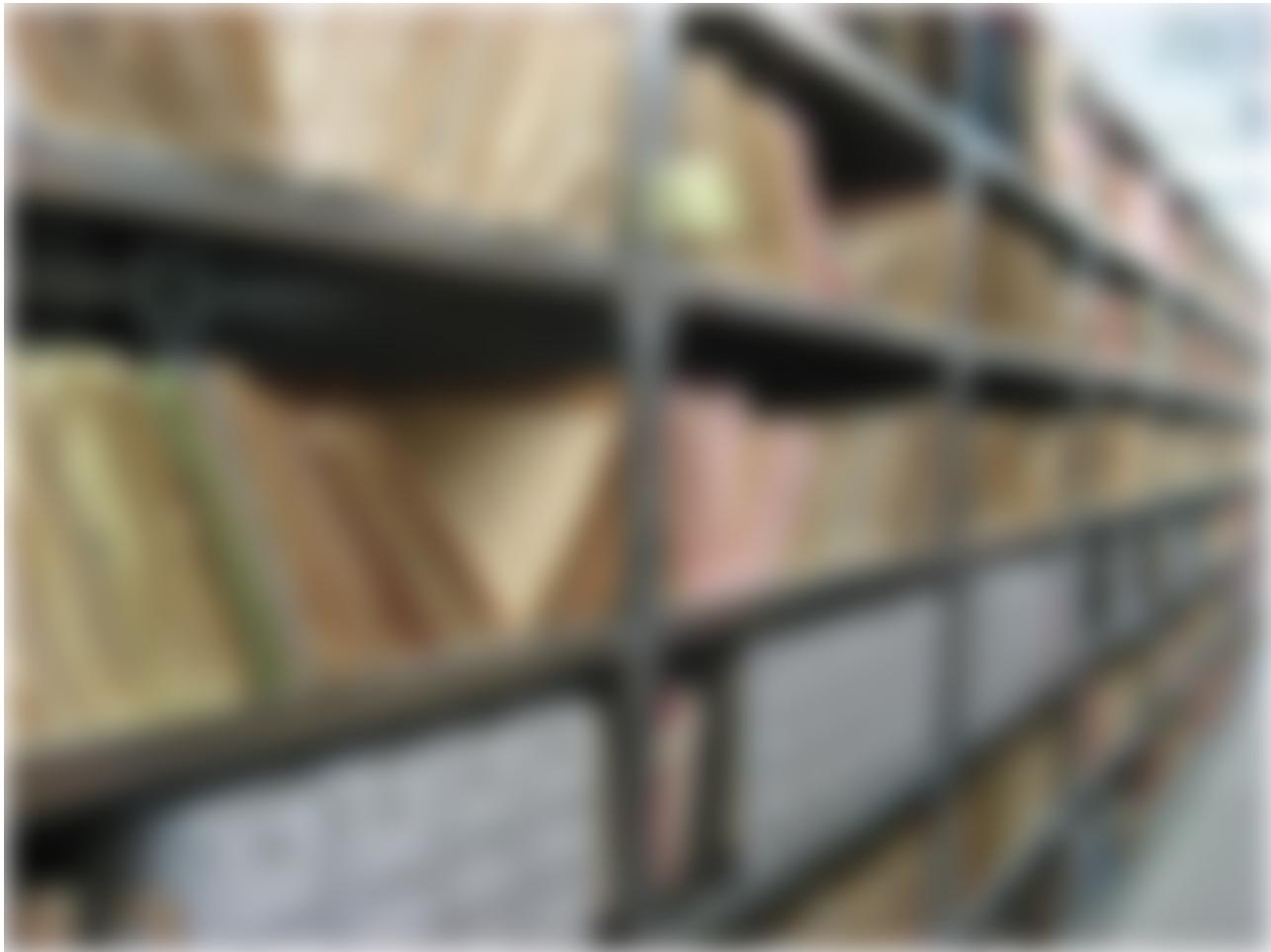
Кроме того, пакеты Python в образ можно устанавливать с помощью pip , wheel и conda . Если речь идёт не о Python, а о других языках программирования, то при подготовке соответствующих образов могут использоваться и другие менеджеры пакетов.

При этом для того, чтобы установка была бы возможной, нижележащий слой должен предоставить слою, в который выполняется установка пакетов, подходящий менеджер пакетов. Поэтому если вы столкнулись с проблемами при установке пакетов, убедитесь в том, что менеджер пакетов установлен до того, как вы попытаетесь им воспользоваться.

Например, инструкцию RUN в Dockerfile можно использовать для установки списка пакетов с помощью pip . Если вы так поступаете – объедините все команды в одну инструкцию и разделите её символами разрыва строки с помощью символа \ . Благодаря такому подходу файлы будут выглядеть аккуратно и это приведёт к добавлению в образ меньшего количества слоёв, чем было бы добавлено при использовании нескольких инструкций RUN .

Кроме того, для установки нескольких пакетов можно поступить и по-другому. Их можно перечислить в файле и передать менеджеру пакетов этот файл с помощью RUN . Обычно таким файлам дают имя requirements.txt .

Инструкция WORKDIR



Рабочие директории

Инструкция `WORKDIR` позволяет изменить рабочую директорию контейнера. С этой директорией работают инструкции `COPY`, `ADD`, `RUN`, `CMD` и `ENTRYPOINT`, идущие за `WORKDIR`. Вот некоторые особенности, касающиеся этой инструкции:

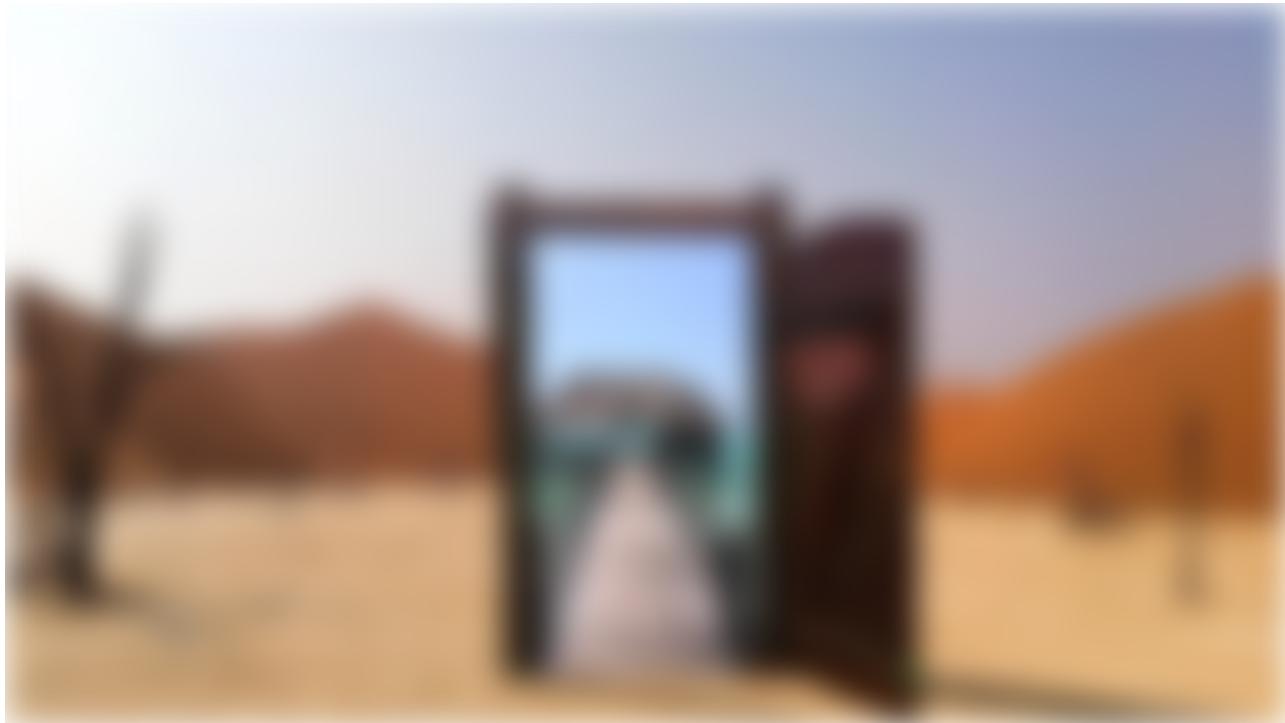
- Лучше устанавливать с помощью `WORKDIR` абсолютные пути к папкам, а не перемещаться по файловой системе с помощью команд `cd` в `Dockerfile`.
- Инструкция `WORKDIR` автоматически создаёт директорию в том случае, если она не существует.
- Можно использовать несколько инструкций `WORKDIR`. Если таким инструкциям предоставляются относительные пути, то каждая из них меняет текущую рабочую директорию.

Инструкция ARG

Инструкция `ARG` позволяет задать переменную, значение которой можно передать из командной строки в образ во время его сборки. Значение для переменной по умолчанию можно представить в `Dockerfile`. Например: `ARG my_var=my_default_value`.

В отличие от ENV -переменных, ARG -переменные недоступны во время выполнения контейнера. Однако ARG -переменные можно использовать для задания значений по умолчанию для ENV -переменных из командной строки в процессе сборки образа. А ENV -переменные уже будут доступны в контейнере во время его выполнения. Подробности о такой методике работы с переменными можно почитать [здесь](#).

Инструкция ENTRYPPOINT



Лункт перехода в какое-то место

Инструкция ENTRYPPOINT позволяет задавать команду с аргументами, которая должна выполняться при запуске контейнера. Она похожа на команду CMD , но параметры, задаваемые в ENTRYPPOINT , не перезаписываются в том случае, если контейнер запускают с параметрами командной строки.

Вместо этого аргументы командной строки, передаваемые в конструкции вида docker run my_image_name , добавляются к аргументам, задаваемым инструкцией ENTRYPPOINT . Например, после выполнения команды вида docker run my_image bash аргумент bash добавится в конец списка аргументов, заданных с помощью ENTRYPPOINT . Готовя Dockerfile, не забудьте об инструкции CMD или ENTRYPPOINT .

В документации к Docker есть несколько рекомендаций, касающихся того, какую инструкцию, CMD или ENTRYPPOINT , стоит выбрать в качестве инструмента для

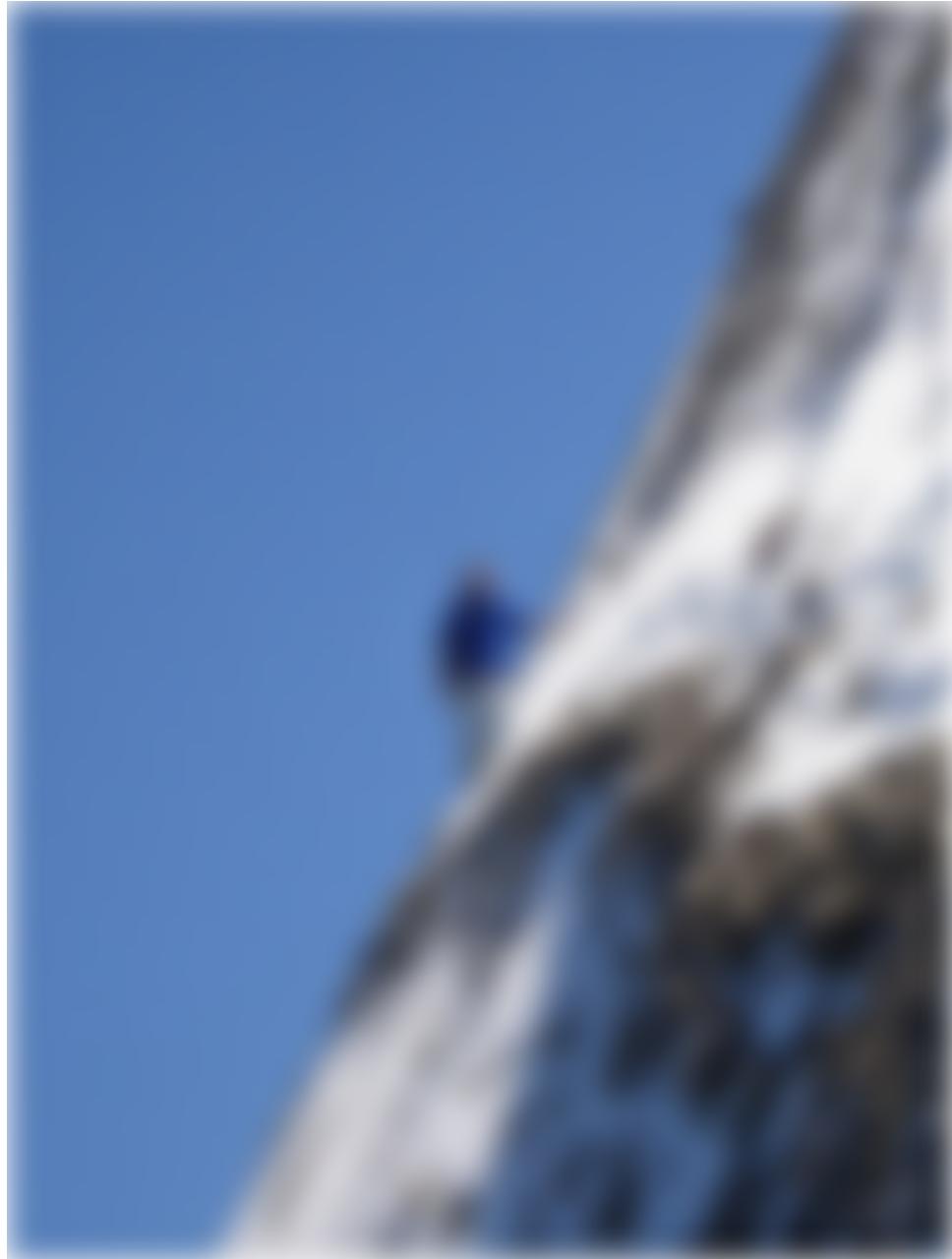
выполнения команд при запуске контейнера:

- Если при каждом запуске контейнера нужно выполнять одну и ту же команду – используйте `ENTRYPOINT` .
- Если контейнер будет использоваться в роли приложения – используйте `ENTRYPOINT` .
- Если вы знаете, что при запуске контейнера вам понадобится передавать ему аргументы, которые могут перезаписывать аргументы, указанные в `Dockerfile`, используйте `CMD` .

В нашем примере использование инструкции `ENTRYPOINT ["python", "my_script.py", "my_var"]` приводит к тому, что контейнер, при запуске, запускает Python-скрипт `my_script.py` с аргументом `my_var` . Значение, представленное `my_var` , потом можно использовать в скрипте с помощью `argparse`. Обратите внимание на то, что в `Dockerfile` переменной `my_var` , до её использования, назначено значение по умолчанию с помощью `ARG` . В результате, если при запуске контейнера ему не передали соответствующее значение, будет применено значение по умолчанию.

Документация Docker рекомендует использовать exec-форму `ENTRYPOINT : ENTRYPOINT ["executable", "param1", "param2"]` .

Инструкция EXPOSE

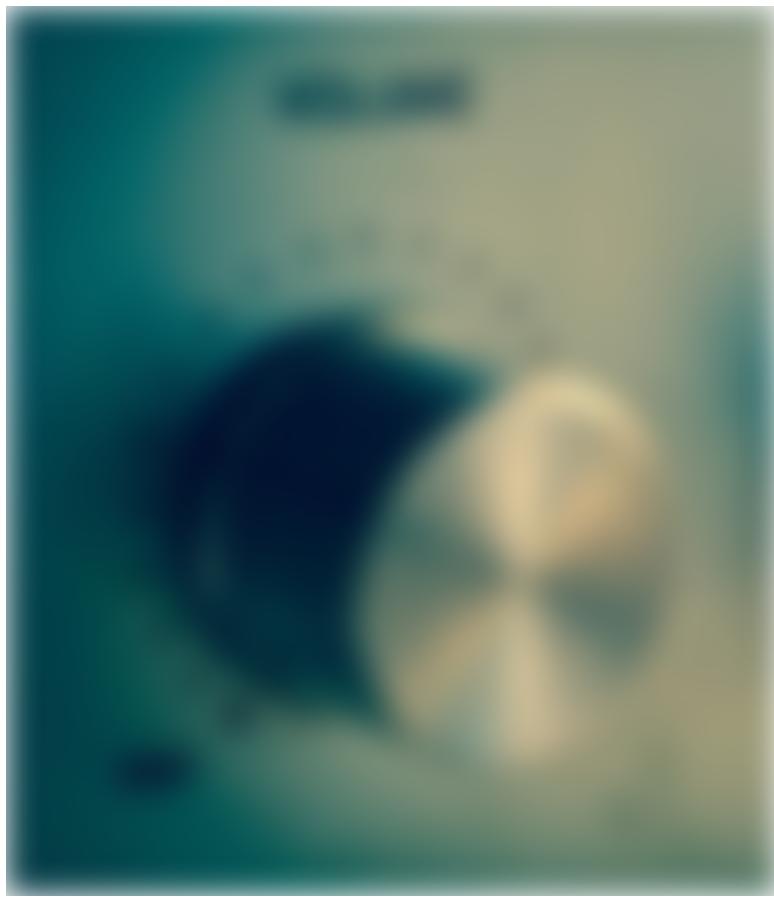


Инструкция EXPOSE

Инструкция `EXPOSE` указывает на то, какие порты планируется открыть для того, чтобы через них можно было бы связаться с работающим контейнером. Эта инструкция не открывает порты. Она, скорее, играет роль документации к образу, средством общения того, кто собирает образ, и того, кто запускает контейнер.

Для того чтобы открыть порт (или порты) и настроить перенаправление портов, нужно выполнить команду `docker run` с ключом `-p`. Если использовать ключ в виде `-P` (с заглавной буквой `P`), то открыты будут все порты, указанные в инструкции `EXPOSE`.

Инструкция VOLUME



Инструкция `VOLUME`

Инструкция `VOLUME` позволяет указать место, которое контейнер будет использовать для постоянного хранения файлов и для работы с такими файлами. Об этом мы ещё поговорим.

Итоги

Теперь вы знаете дюжину инструкций, применяемых при создании образов с помощью `Dockerfile`. Этим список таких инструкций не исчерпывается. В частности, мы не рассмотрели здесь такие инструкции, как `USER`, `ONBUILD`, `STOPSIGNS`, `SHELL` и `HEALTHCHECK`. Вот краткий справочник по инструкциям `Dockerfile`.

Вероятно, файлы `Dockerfile` – это ключевой компонент экосистемы `Docker`, работать с которым нужно научиться всем, кто хочет уверенно чувствовать себя в этой среде. Мы ещё вернёмся к разговору о них в следующий раз, когда будем обсуждать способы уменьшения размеров образов.

Уважаемые читатели! Если вы пользуетесь `Docker` на практике, просим рассказать о том, как вы пишете `Docker`-файлы.

Habrahabr10

Промо-код для скидки в 10% на наши виртуальные сервера

Теги: Docker, разработка

Хабы: Блог компании RUVDS.com, Разработка веб-сайтов, Виртуализация

Редакторский дайджест

Присылаем лучшие статьи раз в месяц



Электропочта



RUVDS.com

VDS/VPS-хостинг. Скидка 15% по коду HABR15

[Telegram](#) [ВКонтакте](#) [Twitter](#)



409

349.3

Карма Рейтинг

@ru_vds

Пользователь

Комментарии 11

Публикации

[ЛУЧШИЕ ЗА СУТКИ](#) [ПОХОЖИЕ](#)

22 часа назад

Герои напильника и паяльника: итоги сезона DIY

8 мин 8.4K

Сезон DIY

Спецпроект

+44

42

6



spiritus_sancti

23 часа назад

RGB-усилители. Особенности, проблемы, выбор

Простой 6 мин 5.6K

[Туториал](#)

+40

24

11



ru_vds

19 часов назад

Профилирование Python — почему и где тормозит ваш код

Средний 10 мин 3.5K

[Туториал](#)[Перевод](#)

+32

93

4



BiktorSergeev

16 часов назад

WebOne: даём жизнь старым браузерам

6 мин 3.3K

+30

29

11



zatim

1 час назад

Видеокарта VGA для микроконтроллера

Средний 17 мин 883

[Туториал](#)

+21

16

6



it_union

2 часа назад

ЗАО Гейм Инсайт Труп

Простой 4 мин 3.4K

+21

7

7



АннаVlMogozova

19 часов назад

Как повысить эффективность коммуникаций в команде: учимся решать конфликты

7 мин 2.2K

+19

43

2



Yu-Leo

вчера в 15:22

Обзор электронной книги Meebook P10 Pro

Простой 9 мин 6.3K

Обзор

+18

18

8



Sagidullin

6 часов назад

Всего два месяца — и новый релиз ядра Linux. Что появилось в ядре 6.5, что изменилось и что удалили. Новые возможности

5 мин 5.3K

+15

6

5



mr-pickles

22 часа назад

Архетипы программных архитекторов. Часть 2

Простой 9 мин 2.5K

Перевод

+15

32

0

Показать еще

ИНФОРМАЦИЯ

Ваш аккаунт

[Войти](#)[Регистрация](#)

Разделы

[Статьи](#)[Новости](#)[Хабы](#)[Компании](#)

Информация

[Устройство сайта](#)[Для авторов](#)[Для компаний](#)[Документы](#)

Услуги

[Корпоративный блог](#)[Медийная реклама](#)[Нативные проекты](#)

[Авторы](#)[Соглашение](#)[Образовательные](#)[Песочница](#)[Конфиденциальность](#)[программы](#)[Стартапам](#)[Спецпроекты](#)[Настройка языка](#)[Техническая поддержка](#)[Вернуться на старую версию](#)

© 2006–2023, Habr

[ruvds.com](#)

VDS в Цюрихе. Дата-центр TIER III – швейцарское качество по низкой цене.
[ruvds.com](#)

Антивирусная защита виртуального сервера. Легкий агент для VPS.
[ruvds.com](#)

VPS в Лондоне. Дата-центр TIER III – английская точность за рубли.
[ruvds.com](#)

VPS с видеокартой на мощных серверах 3,4ГГц
[ruvds.com](#)

ПРИЛОЖЕНИЯ

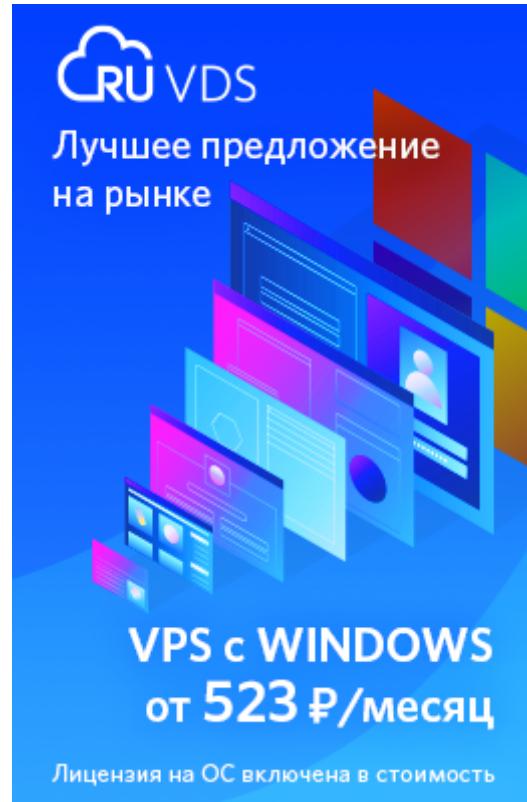


RUVDS Client

Приложение для мониторинга и управления виртуальными серверами RUVDS с мобильных устройств.

[Android](#) [iOS](#)

ВИДЖЕТ



ВИДЖЕТ



БЛОГ НА ХАБРЕ

Профилирование Python — почему и где тормозит ваш код

 3.5K  4

23 часа назад

RGB-усилители. Особенности, проблемы, выбор

 5.6K  11

27 авг в 17:00

Интернет 90-х: когда после 20 часов в онлайне тебе пишет президент ISP

 15K  40

26 авг в 17:00

История компьютерных стратегий. Часть 8. «Age of Empires»: шедевр геймдева, от которого бомбит у любителей истории

 13K  12

25 авг в 16:00

Xbox is a new Dreamcast. Зачем покупать консоль от Microsoft в 2023 году и во что играть

 6.5K  8