



bykvaadm 22 января 2018 в 22:07

Реклама

Лабораторная работа: введение в Docker с нуля. Ваш первый микросервис

Настройка Linux, Системное администрирование, Серверное администрирование, Управление разработкой, DevOps

Tutorial

Привет, хабрапользователь! Сегодня я попробую представить тебе очередную статью о докере. Зачем я это делаю, если таких статей уже множество? Ответов здесь несколько. Во-первых не все они описывают то, что мне самому бы очень пригодилось в самом начале моего пути изучения докера. Во-вторых хотелось бы дать людям к теории немного практики прямо по этой теории. Одна из немаловажных причин — уложить весь накопленный за этот недолгий период изучения докера опыт (я работаю с ним чуть более полугода) в какой-то сформированный формат, до конца разложив для себя все по-полочкам. Ну и в конце-концов излить душу, описывая некоторые грабли на которые я уже наступил (дать советы о них) и вилы, решение которых в докере просто не предусмотрено из коробки и о проблемах которых стоило бы задуматься на этапе когда вас распирает от острого желания перевести весь мир вокруг себя в контейнеры до осознания что не для всех вещей эта технология годна.

Что мы будем рассматривать в данной статье?

В Части 0 (теоретической) я расскажу вам о контейнерах, что это и с чем едят

В Частях 1-5 будет теория и практическое задание, где мы напишем микросервис на python, работающий с очередью rabbitmq.

В Части 6 — послесловие

Часть 0.0: прогреваем покрывки

Что такое Docker? это такой новомодный способ изоляции ваших приложений друг от друга с помощью linux namespaces. Он крайне легок в управлении, расширении, миграции и подходит для огромного спектра задач начиная от разработки приложений и сборки пакетов и заканчивая тестами-пятиминутками (запустить-проверить 1-2 команды-закрыть и забыть, да так чтобы еще и мусор за тобой прибрали). Читайте подробнее на офф сайте. Стоит только представлять себе на первом этапе, что контейнер состоит из слоев, т.е. из слепков состояний файловой системы. Я расскажу об этом чуть позже.

Лабораторная инфраструктура данной статьи может быть воспроизведена как на одной ОС с Linux на борту, так и на серии виртуальных машин\vps\whatever. Я делал все на одном дедике (dedicated server) с ОС Debian. Здесь и далее я предполагаю что вы работаете в одной ОС Debian 9. Команды и подходы от ОС к ОС могут отличаться, описать все в одной статье нереально, поэтому выбрана наиболее знакомая мне серверная ОС. Надеюсь для вас не составит труда установить себе виртуальную машину и дать ей выход в интернет.

Нам понадобится установить docker-ce (официальный ман)

Я не ставлю своей задачей разжевывать то, что уже и так написано, поэтому здесь дам ссылку на документацию, а в листинге приведу команды для установки на **чистом** Debian9 x64. Команды, которые требуется выполнять от имени суперпользователя начинаются с символа #, а команды обычного пользователя — с \$.

```
# apt install apt-transport-https ca-certificates curl gnupg2 software-properties-common
# curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg | apt-key add -
# add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") $(lsb_release -cs) stable"
# apt update && apt install docker-ce
```

Часть 0.1 Сравнение с VM

Важно помнить, что Docker — средство изоляции процесса(задачи), а это значит относиться к докеру как к виртуалке нельзя. Он является оберткой вокруг средств контейнеризации (cgroups + namespaces) конкретно linux kernel. Но он имеет многие похожие фишки, как и у виртуализации:

1. независимость — контейнер может быть перемещен на любую ОС с docker-службой на борту и контейнер будет работать. (Официально — да, по факту, не уверен что совместимость такая радужная, как пони, радуга и бабочки. Если у вас есть иной опыт — прошу поделиться)

2. самодостаточность — контейнер будет выполнять свои функции в любом месте, где бы его не запустили.

И тем не менее он отличается от привычной виртуализации:

1. Внутри контейнера находится минимально необходимый набор софта, необходимый для работы вашего процесса. Это уже не полноценная ОС, которую надо мониторить, следить за остатком места итд итп.
2. Используется другой подход к виртуализации. почитать об этом. Я имею ввиду сам принцип — там нет привычной хостовой ОС.
3. Следует особо относиться к контейнеру и генерируемым им данным. Контейнер это инструмент обработки данных, но не инструмент их хранения. Как пример — контейнер — это гвоздезабивающая машина, подаваемые на вход данные — доска и гвоздь, результат работы — забить гвоздь в доску. При этом доска с гвоздем не остается частью той самой гвоздезабивающей машины, результат отдельно, инструмент отдельно. Выходные данные не должны сохраняться внутри контейнера (можно, но это не docker-way). Поэтому, контейнер это либо worker (отработал, отчитался в очередь), либо, если это, например, веб-сервер, то нужно использовать внешние тома. (все это очень просто, не стоит в этом моменте грустить).

Часть 0.2 Процессы в контейнерах

Теперь, когда мы выяснили что такое docker, посмотрим в бинокль что там внутри. Нужно запомнить следующие постулаты:

1. Контейнер живет, пока живет процесс, вокруг которого рождается контейнер.
2. Внутри контейнера этот процесс имеет pid=1
3. Рядом с процессом с pid=1 можно порождать сколько угодно других процессов (в пределах возможностей ОС, естественно), но убив (рестартовав) именно процесс с pid=1, контейнер выходит. (см п.1)
4. Внутри контейнера вы увидите привычное согласно стандартам FHS расположение директорий. Расположение это идентично исходному дистрибутиву (с которого взят контейнер).
5. Данные, создаваемые внутри контейнера остаются в контейнере и нигде более не сохраняются (ну, еще к этому слою есть доступ из хостовой ОС). удалив контейнер — потеряете все ваши изменения. Поэтому данные в контейнерах не хранят, а выносят наружу, на хостовую ОС.

Часть 0.3 Откуда брать эти ваши контейнеры? как хранить?

Существуют публичные и приватные хранилки официальных и неофициальных образов. Они называются docker registry. Самый популярный из них — Docker hub. Когда вы докеризуете какой-либо сервис, сходите сначала на хаб и посмотрите, а не сделал ли это кто-то уже за вас?

Также это будет сильным подспорьем для вас в момент изучения. поскольку на хабе виден готовый Dockerfile, то вам можно будет подглядеть у крутых дядек, а как они делают ту или иную штуку. Много их хранится на github и других подобных ресурсах.

Кроме публичных registry существуют еще и приватные — платные и бесплатные. Платные вам понадобятся, когда вы в замучавшись обслуживанием воскликните «да пусть вот эти ребята следят за всей этой слоёной docker-вакханалией в этих ваших registry». И правда, когда вы активно пользуетесь при DevOps докером, когда люди или автоматика билдит сотни контейнеров, то рано или поздно у вас начнет подгорать от того как это чистить и обслуживать.

Не все конечно так плохо, естественно. Для десятка человек и пары билдов в день подойдет и свой registry. Для личного пользования тем более.

Зачем может пригодиться registry? это единое место хранения и обмена контейнерами между вами, другими людьми или автоматикой. И вот это по-настоящему крутая штука. Положить и забрать готовый контейнер можно всего одной командой, а при условии того что весит он копейки (по сравнению с VM), то плюсы просто на лицо. Правда стоит помнить, что docker — инструмент, docker registry не хранит данных ваших рабочих контейнеров. Таким образом можно иметь крайне удобный воркфлоу: разработчики пишут продукт, билдят новый контейнер, пушат его в репозиторий. Тестировщики забирают его, тестируют, дают отмашку. Девопсы затем с помощью, например, ansible накатывают контейнеры на прод. Причем вам абсолютно не важно, 1 у вас сервер или 100, с помощью ansible и registry раскатывание контейнеров в прод — одно удовольствие. Здесь же и весь фарш с CI — новые версии могут быть автоматически скачаны билд-сервером. Посему очень рекомендую, даже для личного пользования, завести себе VPSIDS с registry, это очень удобно — обмениваться контейнерами.

Скачать образ из репозитория:

```
docker pull wallarm/node
```

Загрузить образ в репозиторий:

```
docker push example.com:5000/my_image
```

В приватные репозитории нужно еще и логиниться, для этого вам пригодится команда

```
docker login
```

где потребуется ввести логин и пароль.

Запустить свой registry можно всего одной командой:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Подробнее на офф сайте. SSL, авторизация — все это можно.

Часть 0.4 Контейнеры и образы (images)

Докер образ (image) — это некоторый набор слоёв. Каждый слой — результат работы команды в Dockerfile. Грубо говоря образ — это шаблон на основе которого вы будете запускать контейнеры. Все что запускается на основе этого образа есть контейнер, или инстанс. т.е. с одного образа можно запустить несколько одинаковых копий этого образа — контейнеров. Причем потом, после каких-то манипуляций, из этого контейнера можно создать шаблон — новый образ. Хранится все это безобразия в /var/lib/docker. [документация](#).

Список образов в вашей системе можно глянуть командой:

```
# docker images
```

Список контейнеров:

```
docker ps
```

ключ -a покажет в том числе остановленные, а ключ -s — его размер. т.е. сколько фактически, сейчас, в рантайме, этот контейнер занимает места на диске.

Часть 0.5 Как именовать контейнеры и образы (images)?

У образа существует 3 поля, имеющих отношения к именованию:

- 1) Репозиторий
- 2) Тег
- 3) Image ID

Репозиторий — реальный или нереальный, это то место, откуда или куда ваш образ будет скачан\загружен, или это просто выдуманное имя, если загружать куда вы его не собираетесь.

Тег — обычно это версия продукта. По идее — это любой набор символов. (из разрешенного списка [a-z0-9.-] итп.) если тега нет, то автоматически используется слово **latest**. тег ставится через символ: от имени репозитория и подставляется автоматически если его не указать при push\pull.

ImageID — локально генерируемый уникальный ID вашего образа, по которому этим образом можно оперировать.

Таким образом вы или автор контейнера влияете на репозиторий и/или тег, а система локально — на ID. Но, кстати вам никто не мешает переиспользовать любое чужое имя репозитория, другой вопрос что запушить (push, загрузить) в него вы не сможете

Например:

1/2/3-blah.blah.blah — имя вашего локального образа, вы его выдумали
projects/my_first_docker — тоже имя вашего локального образа
projects/my_first_docker:latest — то же самое имя, но с тегом. эквивалентно предыдущему.
projects/my_first_docker:1.13.33 — а вот это уже конкретная версия образа в этом репозитории.
projects/my_first_docker:1.13.34
projects/my_first_docker:1.13.35
... итд — все это один и тот же проект, но версии ваших образов будут разными.

Одна из больших фишек докера — переиспользование слоев. это значит, что если слой не менялся, то новая версия вашего образа сможет использовать слои других контейнеров. Например корневой слой с debian будет использоваться всеми контейнерами, основанными на debian. Если ваш второй слой, например, установка nginx, а третий — положить конфиг, то при изменении конфига и сборке нового образа, он будет состоять из 2-х старых слоев и одним новым. Но, не спешите радоваться. Хотя это переиспользование коллосально может экономить место, докер так еще может поднасрать этими слоями, которые он создает на каждый чих. слои эти за счет переиспользования будут иметь адовые зависимости друг от друга и в конечном итоге на крупных инсталляциях где собирается множество контейнеров это будет та самая слоеная вакханалия, о которой я уже говорил выше. Но не стоит воспринимать сказанное как то, что с вами обязательно случится, я говорю о нагруженной системе с десятками новых билдов в день, когда вся контора вкупе с автоматикой весь день что-то собирает. Такой вакханалии на обычных серверах нет — там у вас всего будет парочка контейнеров. И с другой стороны, чистится это тоже довольно просто — просто `rm -rf`. самое важное для вас — контейнеры, а они лежат в репозитории и просто автоматически скачаются заново. Ну а результат работы вы и так в контейнере не храните.

Еще примеры:

wallarm/node — а вот это уже скачанный с публичного докер хаба образ с лучшим WAF от компании Wallarm.
debian:stretch — образ debian, версия образа — stretch (не число, но слово)
centos:7 — аналогично debian.
mongo:3.2 — образ mongodb версии 3.2, скачанный с публичного репозитория
nginx — latest stable nginx — аналогично.

Причем чаще всего на хабе можно увидеть разнообразные версии нужного вам софта. Так например mongo даст вам скачать и 3.2 и 3.4 и 3.6 и даже дев версию. и разные промежуточные. и еще кучу других.

Представьте, как удобно — у вас крутится монга 3.2, вы хотите попробовать рядом другую версию. просто качаете контейнер с новой версией и запускаете. и не надо поднимать никаких виртуалок, настраивать их, чистить за собой. ведь все что нужно чтобы удалить докер образ — ввести команду `docker rmi`. А весит он несравнимо меньше. метров 200 например. А если хотите еще круче — можно воспользоваться готовым решением на базе alpine linux. bind9-alpine. один демон. 3 конфига. 15 мегабайт!!! и это полноценная, готовая к работе инсталляция.

Следует помнить: один и тот же образ может иметь сколько угодно имен репозитория и тегов, но будет иметь одинаковый `imageid`. При этом, образ не будет удален, пока все теги этого образа не будут удалены. т.е. скачав образ debian, задав ему новый тег вы получите в списке 2 образа с разными именами, но одинаковыми `imageid`. и удалив `debian:stretch`, вы просто удалите один из тегов, а сам образ останется жить.

Чтобы задать другое имя для существующего образа используется команда:

```
docker tag <existing image name> <new image name>
```

Для удаления образа:

```
docker rmi <image>
```

А вот контейнеры имеют 2 имени:

- 1) CONTAINER ID
- 2) Name

С `id` — тут то же самое — это уникальное имя конкретного уникального запущенного инстанса образа. проще говоря — уникальное имя запущенного образа. Образ может быть запущен сколько угодно раз и каждая копия получит уникальное имя.

Name — а вот это уже имя, более удобное для человечков и скриптования. Дело в том, что запуская различные образа вы никогда точно не узнаете с каким именем вы его запустили, пока не полезете не посмотрите запущенные контейнеры, или не возьмете этот выхлоп при запуске контейнера. с учетом того что вы можете логировать на stdout, то ваше имя контейнера потеряется. Так вот можно заранее задавать имя для запущенного контейнера с помощью ключа --name, тогда и оперировать им можно с сразу известным именем.

Список контейнеров:

```
docker ps
```

Для удаления контейнера:

```
docker rm <container>
```

Часть 0.6 Ладно, все это чужие контейнеры, а как мне сделать свое, с нуля?

Для создания своих контейнеров существует механизм сборки — docker build. Он использует набор инструкций в Dockerfile для сборки вашего собственного образа. При сборке контейнера внутри используется оболочка sh и ваши команды исполняются в ней.

Следует знать следующее:

1. Каждая законченная команда — создает слой файловой системы с результатами изменений, порожденными этой командой. т.е. например выполнив команду apt install htop, вы создадите слой, который будет содержать результат выполнения этой команды — бинарники, библиотеки итд. в конечном итоге каждый такой слой будет наложен друг на друга, а затем на исходный (образ операционной системы) и вы получите конечный результат. Отсюда проистекают несколько ограничений:
2. Слои независимы друг от друга. это значит что запущенная в процессе сборки какая-нибудь служба внутри контейнера существует только в пределах своего слоя. Яркий пример — попытка залить базу в mysql. Как это обычно происходит? нужно запустить mysql-сервер и следующей командой залить базу. Здесь это так не сработает. Создастся слой, который сохранит результаты запуска mysql (логи, итп) и потом mysql просто завершится. в следующем слое (при выполнении команды заливки базы) мускуль уже не будет запущен и будет ошибка. Решение этой проблемы — всего-навсего объединять команды через &&.
3. 3) А вот постоянные данные будут накладываться от первой команды до последней друг на друга и храниться постоянно от слоя к слою. Поэтому, создав какой-либо файл первой командой, вы сможете к нему обратиться и в последней команде.

Часть 0.7 Напоследок и «Это не докер-вэй»

Докер довольно таки мусорит слоями при билде. Кроме того вы можете оставлять кучу остановленных контейнеров. Удалить все это одной командой

```
docker system prune
```

Это не докер-вэй

Докер по задумке виртуализирует именно один процесс (с любым кол-вом потомков). Никто вам не мешает конечно запихнуть в один контейнер и 10 процессов, но тогда пеняйте на себя.

Сделать это можно с помощью, например, supervisor.

Но, скажете вы, как же быть со службами, которые ну всю свою жизнь будут вместе и не должны разделяться? использовать контент, который будут генерировать друг для друга?

Для того чтобы сделать это красиво и правильно есть docker-compose — это yaml файл, который описывает N ваших контейнеров и их взаимоотношения. какой контейнер с каким скрестить, какие порты друг для друга открыть, какими данными поделиться.

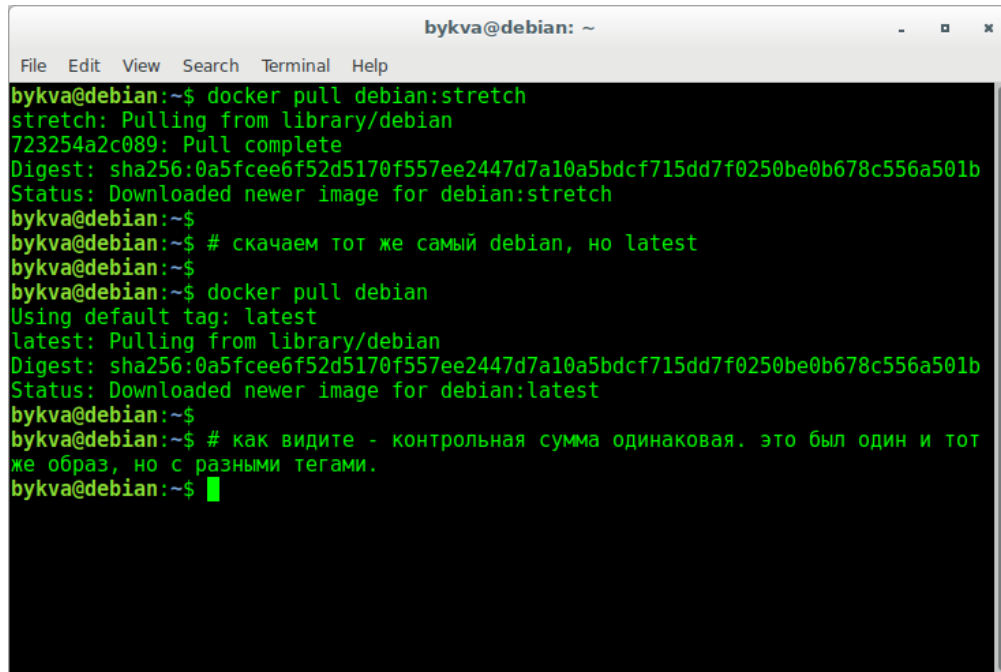
В случае с compose вам, на самом деле, и управлять своим проектом проще. например, связка nginx+uwsgi+mongo — это 3 контейнера, хотя точно известно, что никто кроме nginx не будет ходить в uwsgi, а кроме uwsgi — никто в mongo. да и жить они будут всегда вместе. (ЭТО ЧАСТНЫЙ СЛУЧАЙ). Вот тут у нас получается следующая ситуация — ваше приложение (api) будет

обновляться часто — вы его пишете и пушите каждый день. а, например, релизы nginx или mongodb выходят куда реже — может месяц, может дольше. Так зачем каждый раз билдить этого тяжеловеса, когда изменения то всего происходят в одном месте? Когда придет время обновить nginx, вы просто смените имя тега и всесто ребилда проекта, просто скачается новый контейнер с nginx.

Часть 1: когда уже практика?

На этом этапе можно попробовать потрогать контейнеры и поучить необходимые команды. Я — сторонник вбивания команд из туториалов вручную. Поэтому — только скриншоты.

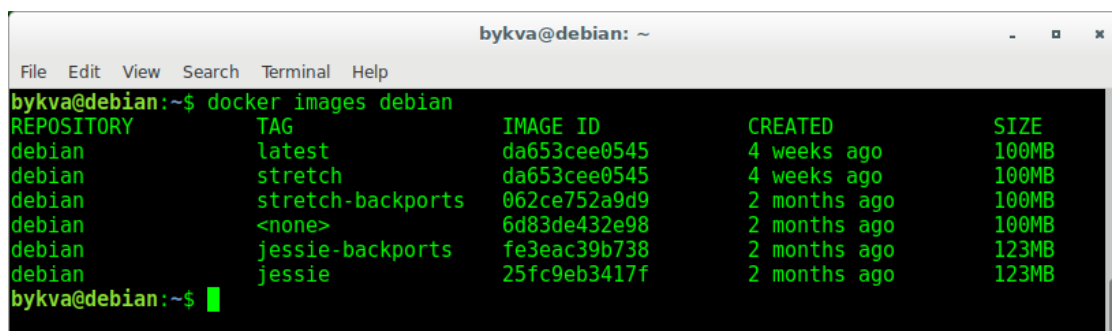
1. Попробуем скачать дебиан образ



```
bykva@debian: ~  
File Edit View Search Terminal Help  
bykva@debian:~$ docker pull debian:stretch  
stretch: Pulling from library/debian  
723254a2c089: Pull complete  
Digest: sha256:0a5fcee6f52d5170f557ee2447d7a10a5bdcf715dd7f0250be0b678c556a501b  
Status: Downloaded newer image for debian:stretch  
bykva@debian:~$  
bykva@debian:~$ # скачаем тот же самый debian, но latest  
bykva@debian:~$  
bykva@debian:~$ docker pull debian  
Using default tag: latest  
latest: Pulling from library/debian  
Digest: sha256:0a5fcee6f52d5170f557ee2447d7a10a5bdcf715dd7f0250be0b678c556a501b  
Status: Downloaded newer image for debian:latest  
bykva@debian:~$  
bykva@debian:~$ # как видите - контрольная сумма одинаковая. это был один и тот же образ, но с разными тегами.  
bykva@debian:~$
```

2. Глянем, что получилось. В моем случае, я для фильтра использовал слово debian, т.к. иначе вывалилось бы в выводе куча других образов. вам его использовать не обязательно. Ну и как результат — у вас будет 2 одинаковых imageid с разными тегами (первые две у меня).

3.Ы. привет в будущее! для тех, кто доживет до debian10 и будет читать эту статью, у вас будет не такой же результат что и у меня. Надеюсь, понимаете почему.



```
bykva@debian: ~  
File Edit View Search Terminal Help  
bykva@debian:~$ docker images debian  
REPOSITORY          TAG                 IMAGE ID           CREATED            SIZE  
debian               latest             da653cee0545      4 weeks ago       100MB  
debian               stretch           da653cee0545      4 weeks ago       100MB  
debian               stretch-backports 062ce752a9d9      2 months ago      100MB  
debian               <none>             6d83de432e98      2 months ago      100MB  
debian               jessie-backports   fe3eac39b738      2 months ago      123MB  
debian               jessie             25fc9eb3417f      2 months ago      123MB  
bykva@debian:~$
```

3. Запустим bash-процесс внутри образа debian:stretch. Посмотрите его pid — он равен единице. Т.е. это тот самый главный процесс, вокруг которого мы и пляшем. Обратите внимание — мы не смогли так просто выполнить ps aux — в контейнере нет нужного пакета. поставить его мы можем как и обычно — с помощью apt.

```
bykva@debian: ~  
File Edit View Search Terminal Help  
bykva@debian:~$ docker run -ti debian bash  
root@f42f2af21980:/# ps aux  
bash: ps: command not found  
root@f42f2af21980:/# apt update >/dev/null && apt -y install procs >/dev/null  
WARNING: apt does not have a stable CLI interface. Use with caution in scripts.  
WARNING: apt does not have a stable CLI interface. Use with caution in scripts.  
debconf: delaying package configuration, since apt-utils is not installed  
root@f42f2af21980:/#  
root@f42f2af21980:/#  
root@f42f2af21980:/# ps aux  
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND  
root         1  0.1  0.0   18128  3204 pts/0    Ss   15:16   0:00 bash  
root       256  0.0  0.0   36632  2744 pts/0    R+   15:17   0:00 ps aux  
root@f42f2af21980:/#
```

4. Выйдем из контейнера (exit | Ctrl+D) и попробуем запустить баш заново — это в верхней консоли (вы можете открыть другую, я для простоты скриншотов сделал так). В нижнем окне — посмотрим список запущенных контейнеров

```
bykva@debian: ~  
File Edit View Search Terminal Help  
bykva@debian:~$ docker run -ti debian bash  
root@82deff0e133b:/# ps aux  
bash: ps: command not found  
root@82deff0e133b:/#  
  
bykva@debian:~$ docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES  
82deff0e133b        debian             "bash"             14 seconds ago     Up 14 seconds        
ef591087eec4        6d83de432e98      "bash"             3 hours ago        Up 3 hours            
bykva@debian:~$
```

Вот это да! а куда делся procs? да никуда. его тут нет и не было. когда мы запустили образ заново — то взяли тот самый слепок без установленной программы. а куда делся результат нашей работы? А вон он валяется — в состоянии exited. Как и куча других запущенных мною ранее контейнеров и остановленных впоследствии.

```
bykva@debian: ~  
File Edit View Search Terminal Help  
bykva@debian:~$ docker run -ti debian bash  
root@82deff0e133b:/# ps aux  
bash: ps: command not found  
root@82deff0e133b:/#  
  
bykva@debian:~$ docker ps -a  
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES  
82deff0e133b        debian             "bash"             3 minutes ago      Up 3 minutes        
f52d7753fc25        debian             "bash"             8 minutes ago      Exited (127) 6 minutes ago  
f42f2af21980        debian             "bash"             15 minutes ago     Exited (0) 9 minutes ago  
cf328a302d4b        debian             "bash"             16 minutes ago     Exited (0) 15 minutes ago  
ef591087eec4        6d83de432e98      "bash"             3 hours ago        Up 3 hours            
24ea4a9c946f        alpine             "sh"               7 hours ago        Exited (1) 6 hours ago  
cf1f592c4ca6        alpine             "bash"             7 hours ago        Created  
06cee1dca259        sameersbn/bind     "/sbin/entrypoint.sh..." 3 days ago         Exited (0) 3 days ago  
0e189ed47738        php:7.0-apache     "docker-php-entrypoi..." 2 weeks ago        Exited (0) 2 weeks ago  
f2d25a0054de        mysql:latest       "docker-entrypoint.s..." 2 weeks ago        Exited (0) 2 weeks ago  
bykva@debian:~$
```

Все это — мусор. Но его еще можно оживить:

1. выходим из контейнера в 1-м терминале
2. во втором терминале копируем ID контейнера, подходящего по timestamp как и тот который мы тогда остановили
3. в 1 терминале запускаем этот контейнер командой start. он уходит в background
4. во втором терминале смотрим список запущенных контейнеров — по таймштампу тот что запущен 7 секунд — явно наш.
5. Финт ушами. аттачимся к уже запущенному контейнеру с помощью команды exes и запускаем второй инстанс bash'a.
6. выполняем ps aux — обратите внимание, тот, первый bash, с pid 1 живет. а мы в контейнере теперь управляем им через bash с pid=7. и теперь, если мы выйдем, контейнер будет жить.

```
bykva@debian: ~
File Edit View Search Terminal Help
bykva@debian:~$ docker run -ti debian bash
root@82deff0e133b:/# ps aux
bash: ps: command not found
root@82deff0e133b:/# exit
bykva@debian:~$ docker start f42f2af21980
f42f2af21980
bykva@debian:~$

bykva@debian:~$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
82deff0e133b   debian    "bash"                  3 minutes ago Up 3 minutes           brave_pike
f52d7753fc25   debian    "bash"                  8 minutes ago Exited (127) 6 minutes ago elated_lewin
f42f2af21980   debian    "bash"                  15 minutes ago Exited (0) 9 minutes ago tender_hamilton
cf328a302d4b   debian    "bash"                  16 minutes ago Exited (0) 15 minutes ago agitated_mccarthy
ef591087eec4   6d83de432e98 "bash"                  3 hours ago   Up 3 hours           admiring_villani
24ea4a9c946f   alpine    "sh"                    7 hours ago   Exited (1) 6 hours ago silly_almeida
cf1f592c4ca6   alpine    "bash"                  7 hours ago   Created           vigorous_cori
06cee1dca259   sameersbn/bind "/sbin/entrypoint.sh..." 3 days ago   Exited (0) 3 days ago agitated_ramanujan
0e189ed47738   php:7.0-apache "docker-php-entrypoi..." 2 weeks ago   Exited (0) 2 weeks ago mommy_web_1
f2d25a0054de   mysql:latest "docker-entrypoint.s..." 2 weeks ago   Exited (0) 2 weeks ago mommy_db_1

bykva@debian:~$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS          NAMES
f42f2af21980   debian    "bash"                  21 minutes ago Up 7 seconds           tender_hamilton
ef591087eec4   6d83de432e98 "bash"                  3 hours ago   Up 3 hours           admiring_villani

bykva@debian:~$ docker exec -ti f42f2af21980 bash
root@f42f2af21980:/# ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0  18128  3160 pts/0    Ss+   15:38   0:00 bash
root         7  0.2  0.0  18128  3160 pts/1    Ss    15:38   0:00 bash
root        13  0.0  0.0  36632  2020 pts/1    R+    15:38   0:00 ps aux
root@f42f2af21980:/#
```

Итак:

run — взять образ X и создать контейнер Z с процессом Y

exec — взять контейнер Z и запустить в нем процесс N, при этом процесс Y будет работать как и прежде.

Выводы:

- 1) Контейнер может жить как в background, так и в foreground, лишь бы был жив процесс, который мы виртуализируем.
- 2) контейнер — это развернутый из образа инстанс
- 3) контейнеры можно останавливать и запускать без потери данных (но не стоит, это не докер-вэй)

Часть 2: делаем собственный докер-образ

Учимся использовать dockerfile.

Здесь — два подхода к разработке:

- 1) взять готовый и подточить под себя
- 2) сделать самому с нуля

Вариант первый — когда вы уверены что за вас прекрасно сделали уже всю работу. Например, зачем устанавливать nginx, когда можно взять готовый официальный контейнер с nginx. Особенно это касается тех систем, которые не ставятся в одну команду или которые например в debian имеют устаревшие версии, а на докер хабе их билдят более свежие стабильные. Тем более там уже сделана автосборка — свежие версии там прилетают довольно быстро.

Вариант второй — вы параноик или вам не нравится подход автора образа (например, официального не нашлось, а есть только на хабе васи пупкина). Никто не мешает посмотреть как сделал это василий, а потом пойти и сделать самому. Ну или в случае если вы просто пишете свою логику, которой точно нигде нет, как например докер образ для билда ваших deb-пакетов в jenkins. А ведь это очень удобно!

Dockerfile это файл с набором инструкций, которые будут совершены в чистом контейнере указанного вами образа, а на выходе вы получите свой образ.

Самое главное что вам нужно указать — это директивы FROM и CMD (в принципе не обязательно). Приведу пример такого файла:


```
bykva@debian: /tmp/prj
File Edit View Search Terminal Help
GNU nano 2.7.4 File: Dockerfile Modified
FROM debian:stretch
MAINTAINER bykva

RUN apt-get update && \
    DEBIAN_FRONTEND=noninteractive apt-get -q -y upgrade >/dev/null && \
    DEBIAN_FRONTEND=noninteractive apt-get -q -y install >/dev/null \
    nginx \
    procps \
    iproute2

EXPOSE 80/tcp, 443/tcp

CMD ["nginx", "-g", "daemon off;"]

Get Help Write Out Where Is Cut Text Justify Cur Pos
Exit Read File Replace Uncut Text To Spell Go To Line
```

FROM — какой образ взять за основу.

MAINTAINER — автор данной разработки (кого пинать за работу этого, нового образа)

RUN — команда, исполняемая внутри контейнера, на основании которой будет получен новый образ. Обратите внимание — обычные bash-переносы и && — делает код читаемым и красивым.

Вы также можете писать хоть 10 RUN — команд (но каждая с новой строки), в которых описывать свою логику.

EXPOSE — какие порты, по какому протоколу будут доступны снаружи контейнера при использовании docker-compose. Тут важно помнить, что это не значит что они будут доступны извне сразу. т.е. имеется ввиду, что пока вы при запуске этого образа не укажете мапинг портов хост: контейнер, никак внутрь попасть не получится. поэтому нужно знать какие порты у вас на каком этапе прописаны:

1) внутри контейнера приложение слушает 0.0.0.0:80 (и обязательно 0.0.0.0!!!, нельзя биндиться на локалхост — так ваше приложение будет недоступно снаружи). даже если речь идет о mysql. Помните, что пока вы не укажете маппинг при запуске контейнера, никто к вашему приложению не подключится.

2) docker run -p 80:80 image

И только так — получается, обращаясь на хост по 80 порту вы мапитесь в контейнер на 80 порт. при запуске контейнера этот порт был открыт, а внутри контейнера приложение слушает 80 порт. и так, вы с 80 порта хоста попадете на 80 порт приложения. Естественно порты можно делать любыми. Так, можно запустить множество контейнеров, внутри которых nginx'ы слушают 80-й порт, а мапинг снаружи сделать такой: 8000:80, 8001:80, 8002:80... ну итд, идея понятна. таким образом можно сделать 1 образ с одним конфигом, а запустить его параллельно и независимо множество раз.

CMD — и вот это самое ключевое место. Это та команда, которая будет выполняться при запуске контейнера и работать все это время. Помните да? пока работает процесс, живет контейнер. Здесь подход такой же как в systemd — процесс не должен уходить в background — это незначит, процесс то всего один внутри этого контейнера. Отсюда кстати и разные подходы логирования. например можно оставлять вывод в STDOUT, а можно в логи. Главное, чтобы процесс (главный процесс) после запуска оставался жить. а форков-тредов может быть сколько душе угодно.

Кроме CMD есть еще один вид шаманства — ENTRYPOINT. это обычно shell скрипт, которому в качестве аргумента передается значение CMD.

Нужно запомнить следующие вещи:

- 1) итоговая команда запуска процесса в контейнере = сумма ENTRYPOINT + CMD.
- 2) вы можете писать в докер файле любой вариант сочетания entrypoint и cmd — оставлять или то или другое, или оба сразу.

Например:

```
ENTRYPOINT ["/entrypoint.sh"]
```

```
CMD [«haproxy», "-c", "/etc/haproxy/haproxy.conf"]
```

итого при старте контейнера будет запущена следующая команда:

```
sh -c "/entrypoint.sh haproxy -c /etc/haproxy/haproxy.conf".
```

что делать с этими аргументами внутри скрипта — вам самим решать.

Зачем нужна такая сложность? Правильные пацаны пишут entrypoint для раскрытия всего многообразия возможностей. Если вы пишете образ чисто под себя, можно оставить чисто только один CMD и не писать ENTRYPOINT вообще.

Сценарий 1:

запуск контейнера без аргументов:

```
docker run haproxy
```

выполнится ENTRYPOINT + CMD:

```
sh -c "/entrypoint.sh haproxy -c /etc/haproxy/haproxy.conf"
```

оба они вшиты внутрь контейнера. В скрипте entrypoint.sh автор зашил такую логику: если первый аргумент = haproxy, то haproxy = haproxy -db -W \$@. И запустить полученное. в результате чего будет запущена такая команда:

```
haproxy -db -W -c /etc/haproxy/haproxy.conf"
```

Сценарий 2:

запустим контейнер с аргументами:

```
docker run haproxy bash
```

Где bash — переопределяет команду CMD из dockerfile. В итоге ENTRYPOINT + CMD дадут вам:

```
sh -c "/entrypoint.sh bash"
```

По логике автора, если первый аргумент != haproxy, просто запустить этот процесс. В итоге вы войдете в контейнер просто в оболочку.

Сценарий 3:

хочу указать запуск haproxy с конфигом в нужном для меня месте.

```
docker run haproxy haproxy -c /opt/haproxy.conf
```

здесь первый haproxy — имя образа, а все что за ним — аргументы, передаваемые внутрь. Они переопределяют наш CMD, в итоге получится:

```
sh -c "/entrypoint.sh haproxy -c /opt/haproxy.conf"
```

И будет запущено приложение:

```
haproxy -db -W -c /opt/haproxy.conf"
```

Это примитивный пример что можно сделать, манипулируя вводами. Рассмотреть можно десятки сценариев различного использования комбинаций ENTRYPOINT+CMD. Просто помните, что можно предусмотреть несколько различных видов запуска вашего контейнера у конечного пользователя, который не имеет доступа к вашему dockerfile.

Помимо указанных мною команд, существуют и многие другие, например ADD, COPY — позволяет положить набор данных внутрь контейнера, причем если ADD в качестве источника указать архив, она распакует и положит содержимое в указанное место. удобно для редко изменяемого набора файлов.

Порядок в Dockerfile важен! (с точки зрения оптимизации)

При внесении изменений в Dockerfile и последующий ребилд образа затрагивает тот набор команд, начиная с которого происходит изменение.

Пример:

```
COPY config config
RUN apt install 100500-programs-pack
```

Внося изменения хоть в один символ конфига вы будете вызывать снова и снова каждый раз установку пака из 100500 программ. а вот если эти строчки расположить в обратном порядке, то билд будет происходить почти мгновенно — т.к. слой с этими программами уже существует.

Часть 3: связь ФС контейнера с хостом

Теперь давайте рассмотрим способ доступности данных с хостовой ос из контейнера. Причин этого может быть несколько: передача файлов, конфигурации, или просто сохранение обработанных данных на диск. Как я уже упоминал ранее данные не пропадают просто так если выключить контейнер, но они и остаются в этом контейнере. вытащить потом их — это ненужная и глупая задача. Докер-вэй — монтирование хостового каталога\файла внутрь контейнера. А если проще — мапинг папки\файла снаружи на папку\файл внутри. поскольку это монтирование, то изменение с одной стороны всегда будет видно и с другой. Таким образом, внося изменение в конфиг с хостовой ос — вы можете заставить сервис внутри контейнера работать по-другому. И наоборот — для сохранения базы данных на хосте в одном, строго определенном, удобном для вас месте. Или логов.

делается эта штука при помощи ключа -v:

```
-v /source/folder:/destination/folder -v /path/to/file:/path/to/config
```

В этом месте позволю себе дать вам еще один линк на полезный ман, а именно на статью-сборник команд по докеру с кратким пояснением. [Тыц](#)

В отдельных случаях проще передавать переменные сред окружения, чем конфиги. и докер позволяет это делать, как зашитыми для билда в Dockerfile (ENV key=value), так и через ключ при запуске контейнера, так и через отдельный файл, если таких переменных много.

Часть 4: Удобный запускатор ваших сервисов

Когда вы уже набилдили себе парочку контейнеров, возникает вопрос, как же их автоматически запускать, не прикладывая к этому усилий? ответ прост — через systemd!

Например, создайте файл:

```
# >/etc/systemd/system/my-project.service
# systemctl daemon-reload
# systemctl edit --full my-project.service
```

и занесите туда строки:

```
[Unit]
Description=my first docker service
Requires=docker.service
After=docker.service

[Service]
Restart=always
RestartSec=3
ExecStartPre=/bin/sh -c "/usr/bin/docker rm -f my-project 2> /dev/null || /bin/true"
ExecStart=/usr/bin/docker run --rm -a STDIN -a STDOUT -a STDERR -p 80:80 -v /etc/my-project:/etc/my-project --name my-project:2.2
ExecStop=/usr/bin/docker stop my-project

[Install]
WantedBy=multi-user.target
```

на что здесь следует обратить внимание:

ExecStartPre — команда которая выполняется при запуске\перезапуске и убивает если вдруг существует какой-то завалившийся контейнер с именем my-project. Ну мало ли откуда он взялся (глюк, кривые руки) — с этой строчкой мы в любом случае прибьем то, что мешает стартовать и в любом случае завершим команду положительно.

--rm — ключ позволяет удалить контейнер после его остановки. мусора не останется после остановки службы

--name my-project — имя вашего контейнера

-a STDIN -a STDOUT -a STDERR — attach to std* — присоединиться к потокам ввода\вывода\ошибок вашего процесса. В бекграунде это значит что все что не пишется в лог попадет в журнал systemd

-p — мапинг 0.0.0.0:80 хоста -> 80 порт внутри контейнера

-v — мапинг папки с конфигами

Ключей можно указывать несколько, например несколько разных портов. В том числе можно еще указать и протокол — tcp или udp. По-умолчанию — tcp.

Сохраняем закрываем-запускаем:

```
systemctl restart my-project.service && journalctl -u my-project.service --no-pager -f
```

Вот и пожалуйста — управляем вашей службой как обычным systemd демоном, а journalctl — смотрим stdout вашего демона.

Часть 5: приблизимся к реальности. Построим нашу инфраструктуру

У нас будут следующие синтетические примеры:

send.py -> rabbitmq -> read.py

Т.е. будет 3 контейнера, первый будет ставить задачи в очередь, последний считывать. Все это будет управляться через systemd.

sender.py будет ставить каждые 5 секунд в очередь случайное число от 1 до 7, а воркер (receiver.py) будет считывать это число и имитировать работу — методом простоя кол-ва секунд равное полученному числу.

Мы будем пытаться сделать наши микросервисы так, как это делают правильные пацаны — внутри контейнера код сервиса, а конфиги и логи — снаружи. Это позволит нам поменять адреса очереди, логин, пароль итд без нового билда контейнера. Просто поправить файл и перезапустить контейнер, вот и все дела. Таким образом нам понадобится директория в /etc и в /var/log, куда мы будем мапить логи и конфиги, соответственно. Вообще такой подход удобен. зайдя на любой сервер с докером, где могут сосуществовать рядом один или несколько микросервисов, вы всегда знаете где искать конфиги или логи. А главное, у вас на сервере будет полная чистота. ssh да docker, чего еще надо? (ну, много чего, понятно, например службу мониторинга или puppet, но мы уже избавляемся от кучи всего сервисо-зависимого. А главное — сервис уже не зависит от хостовой ОС. Имейте весь парк серверов на дебиане, а внутри контейнера — все что угодно. Да, эти же аргументы приводят и к виртуализации, но тут история куда легчевеснее. ведь мы тащим по сервакам инструмент, а не целую виртуализированную ОС)

1. RabbitMQ

RabbitMQ — диспетчер обмена сообщениями между приложениями. в нашем случае будет использована примитивная очередь. в эту очередь будет помещаться задание и приниматься воркером. Конфиги настройки очереди мы захардкодим внутрь контейнера. Всего в папке rabbitmq у нас будет 3 файла — Dockerfile и 2 файла с конфигами. Их содержимое — в спойлерах под листингом. Настройка очереди, пользователя, прав доступа — в json файле definitions.

```
mkdir -p ~/Documents/my_project/{sender,receiver,rabbitmq}
cd ~/Documents/my_project/rabbitmq
## создаем файлы, которые пойдут внутрь контейнера (спойлеры под листингом)
docker build -t rabbitmq:1.0 .
>/etc/systemd/system/my_project-rabbitmq.service
## помещаем конфиг сервиса systemd
systemctl edit --full my_project-rabbitmq.service
## запускаем службу
systemctl start my_project-rabbitmq
## проверяем
docker ps
systemctl status my_project-rabbitmq
```

[/etc/systemd/system/my_project-rabbitmq.service](#)

[~/Documents/my_project/rabbitmq/rabbitmq.conf](#)

[~/Documents/my_project/rabbitmq/definitions.json](#)

[~/Documents/my_project/rabbitmq/Dockerfile](#)

2. Отправляльщик сообщений

```
cd ~/Documents/my_project/sender
## кладем сюда sender.py и Dockerfile, билдим образ
docker build -t sender:1.0 .
>/etc/systemd/system/my_project-sender.service
## помещаем конфиг сервиса systemd
systemctl edit --full my_project-sender.service
## запускаем службу
systemctl start my_project-sender
## проверяем
docker ps
systemctl status my_project-sender
```

[/etc/systemd/system/my_project-sender.service](#)

[sender.py](#)

[~/Documents/my_project/sender/Dockerfile](#)

[/etc/my_project/sender/sender.yaml](#)

3. Receiver — или наш работяга-worker.

```
cd ~/Documents/my_project/receiver
## кладем сюда receiver.py и Dockerfile, билдим образ
docker build -t sender:1.0 .
>/etc/systemd/system/my_project-sender.service
## помещаем конфиг сервиса systemd
systemctl edit --full my_project-sender.service
## запускаем службу
systemctl start my_project-sender
## проверяем
docker ps
systemctl status my_project-sender
```

[/etc/systemd/system/my_project-receiver.service](#)

[/etc/my_project/receiver/receiver.yaml](#)

[~/Documents/my_project/receiver/receiver.py](#)

[~/Documents/my_project/receiver/Dockerfile](#)

А теперь смотрите, финт ушами. Выставленные параметры превышают возможность воркера разгребать очередь. Поднимите рядом еще один докер с воркером и любуйтесь, как они будут разгребать очередь. Удобство масштабируемости просто на лицо. Перенесите воркер в другое место — и он будет разгребать очередь оттуда. главное — сетевая связность.

Часть 6: Реальность

Чаще всего подбирать параметры приходится вручную. Особенно когда вы переносите готовые проекты в докер. Вам придется досконально разбираться, какие порты нужно пробросить наружу, куда пишутся ваши логи, где лежит конфигурационный файл, а если его нет — как сделать так чтобы он был, чтобы его можно было вынести из контейнера. А самое главное, черт побери, из каких компонент состоит ваш сервис! Одна из крутых фишек-последствий использования Dockerfile — то, что у вас перед глазами набор инструкций, который доводит ваш образ до состояния готового продукта. И если у вас установка какой-либо службы не прописана в системах автоматического конфигурирования типа ansible\puppet\chef, то с большой долей вероятности у вас нет и полной инструкции как это сделать. А уж тем более, если у вас вдруг оказалось несколько служб рядом. разделять их — такое удовольствие! А уж зависимости служб от версий пакетов — просто сказка. Да, люди придумали такие штуки как virtualenv, но, согласитесь, когда у вас все жестко отделено друг от друга — гораздо приятнее.

К сожалению, не все так гладко в этом мире, как просто взять и использовать готовый образ с docker hub. Приходится кроме вышеописанного, подбирать еще и способ запуска вашей службы. такие, как, например, /etc/init.d/uwsgi start — уже не подойдут. почему? Вот моя небольшая история попытки запихнуть uwsgi в docker.

Проблемы запуска uwsgi в docker ровно две — и обе они кроются в init скриптах автора.

1) /etc/init.d/uwsgi start вываливается с ошибкой. вот просто так, из коробки, сразу. Погружаемся в исходники и начинаем искать проблему. Она кроется в выполнении этой команды:

```
start-stop-daemon --start --quiet \
  --pidfile "$PIDFILE" \
  --exec "$DAEMON" \
  --test > /dev/null \
  && return 2
```

До этого этапа все выполняется прекрасно, процессы стартуют и главное начинают работать даже после выпадения с ошибкой init-скрипта. Проблема кроется в том, что внутри докера нет прав на прочтение /proc/{id}/exe. После чего считается что процесса нет и согласно заложенному поведению выдается код возврата 0, условие положительно, значит будет исполнена правая часть && и в результате в исходную функцию выдается код возврата 2 (return 2), который и дает в результате ошибку запуска (это заложено

автором). Обсуждение этого косяка при запуске разного ПО идет аж с 14 года: github.com/moby/moby/issues/6800.

Workaround:

Я подобрал несколько решений для данной задачи, но все они не столь хороши, как последнее.

а) исправляем в файле `/usr/share/uwsgi/init/specific_daemon` в приведенном мною выше коде ключ `--exec` на `--startas`. Согласно ману (да нет, шутка, согласно ману они одинаково работают, так что согласно форумам) при использовании ключа `startas` будет пропущена проверка `/proc/{id}/exe` и просто проверяется запущен ли процесс с таким `pid`. (<https://chris-lamb.co.uk/posts/start-stop-daemon-exec-vs-startas>)

б) запуск контейнера с ключем `--cap-add=SYS_PTRACE`. В этом случае править ничего не нужно.

с) не использовать авторские `init`-скрипты.

2) После запуска `/etc/init.d/uwsgi start` скрипт прекрасно отработывает и выходит. Докер естественно после этого завершается (помните почему?). Причем демонизация захардкожена автором в скрипте запуска, без возможности указать в конфиге хочешь ты этого или нет. На этом моменте я все же психанул и сначала поправил скрипт запуска, но потом понял что все же это не тру путь и нужно от этого отказываться, делая просто одну строчку запуска и один конфиг файл.

В итоге что было сделано:

1) в конфиг добавлены следующие параметры (и остальные из дефолтного конфига):

```
...

stats = 0.0.0.0:9090
socket = 0.0.0.0:3031
pidfile = /run/uwsgi/pid
socket = /run/uwsgi/socket
...
```

2) в `Dockerfile`:

```
...

COPY app.ini /etc/uwsgi/apps-enabled/app.ini
RUN mkdir /run/uwsgi && chown www-data /run/uwsgi

CMD ["/usr/bin/uwsgi", "--ini", "/etc/uwsgi/apps-enabled/app.ini"]
```

таким образом мы избавились и от той левой проверки из п.1 и от ключа демонизации из п.2, и сами выставили права и место `pid/socket`.

Кроме того в обычном community-edition докере о безопасности особо не думают. Обновляться в контейнере ни в коем случае нельзя! любой перезапуск и вы все потеряете. Поэтому вопрос решается или мутными самописками или самым подходом к разработке — когда выкатка новых версий софта происходит довольно часто, вы билдите контейнер с последними версиями, обновляя весь софт в целом.

Проблем у докера довольно-таки много. Какие-то из них можно решить дополнительными уровнями абстракции — `kubernetes/docker swarm`. А над `kubernetes` можно поставить еще и `helm`. Штуки без сомнения крутые, но это уже совсем другая история ;)

Фух. Это было долго. Я постарался рассказать всего о малой толике того, как можно использовать докер. скоуп применения контейнеризации гораздо шире.

Спасибо за прочтение! Жду ваших комментариев и пожеланий. А еще я публикую повседневную информацию по администрированию в свой канал в телеграме.

Теги: `docker`, `linux`, `containers`, `microservices`, `agile`

↑ +106 ↓ 1354 👁 240k 💬 36



55,0

Карма

0,9

Рейтинг

81

Подписчики

0

Подписки

Александр @bykvaadm
DevOps

Telegram

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

1 марта 2018 в 09:30

Краш-курс на Docker: научитесь плавать с большой рыбой

↑ +9

👁 40k

🔖 237

💬 16

13 июля 2017 в 23:50

Yet another tutorial: запускаем dotnet core приложение в docker на Linux

↑ +17

👁 12,1k

🔖 134

💬 10

1 октября 2012 в 09:03

CRUI 0.2 и Linux Containers — новые возможности

↑ +9

👁 4,5k

🔖 15

💬 0

ВАКАНСИИ

Habr Career



Системный администратор (Windows, Linux, ЛВС, немного телефонии)
ФГУП МНИИ «Интеграл» • Москва

от 70 000 до 100 000 ₽



DevOps / Linux system administrator
Vsemayki.ru • Новосибирск

от 80 000 до 150 000 ₽



Linux-администратор / DevOps-инженер
ЦИФ МГУ им. М.В. Ломоносова • Москва

от 100 000 до 130 000 ₽



DevOps
Factory5 • Казань • Можно удаленно

до 200 000 ₽



DevOps инженер
Платформа НТИ • Москва

от 160 000 до 190 000 ₽

Все вакансии

Реклама

Комментарии 36



TimsTims 23 января 2018 в 00:50






+3






Спасибо за статью. Docker как-раз лежал в ящике «изучить обязательно», но руки всё не доходили)



 **hiwent** 23 января 2018 в 01:30  

 +4 

Отличный материал, спасибо за статью!

 **xlin** 23 января 2018 в 02:21  

 +1 

Взял на заметочку. Интересовался им когда только набирал популярность. Интересно прочитать что нового появилось.

 **dsixteen** 23 января 2018 в 10:44  

 0 

Спасибо большое

 **beduin01** 23 января 2018 в 10:46  

 -1 

Чем больше разбираюсь с Docker тем сильнее начинает казаться, что это просто костыль такой являющийся следствием:

1. Крайне плохой совместимости диструтивов между собой
2. Стремлением бездумно писать софт таская с собой десятки либ разных версий. И очень часто из-за того, чтобы из огромной либы вызывать лишь один метод.

В итоге приложение-уродец без докера ну никак не заработает.

Нормально спроектированному софту никакие докеры не нужны.

 **bykvaadm** 23 января 2018 в 11:54    

 0 

докер — не панацея, а подход к проектированию. Никто не говорит, что его нужно использовать везде. Докер подойдет, например, для микросервисной архитектуры.

С другой стороны, уверен, что в гугле тоже не дураки сидят. а они используют докер. и множество-множество других крупных и не очень корпораций. Так может быть это вы не разобрались с чем его готовить?

 **dmnBrest** 23 января 2018 в 12:24    

 0 

В сферическом вакууме возможно и не нужны. В реальном мире когда работаешь с кучей проектов со своим уникальным окружением без виртуализации не обойтись. Сначала был Vagrant, теперь Docker. Удобно. Да, на счет продакшена тут уже 50/50. Можно использовать контейнеры, а можно а разворачивать все старым дедовским способом. Но без Docker на стадии разработки уже никак. Типичный недавний пример проект клиента — сервис на Python, сервис на NodeJS, Postgres, Redis (+для отладки почты mail сервер я добавил schickling/mailcatcher). Как с таким проектом вы будете работать? А с Docker достаточно иметь правильно составленные Dockerfile + docker-compose.yml и одна команда docker-compose up (или ручной запуск в любом сочетании). Можно конечно использовать Ansible + VirtualBox, но это уже дело вкуса.

 **carebellum** 23 января 2018 в 16:56    

 +1 

Зачастую тут дело не во вкусе, а в системных ресурсах. Зачем разводить N виртуальных систем, когда можно полностью утилизировать одну-две?

 **bykvaadm** 23 января 2018 в 16:57    

 0 

верно. одна система виртуализации и одна система контейнерезации. этого хватит за глаза.

 **InoMono** 23 января 2018 в 22:17    

 0 

Чем больше разбираюсь с Docker тем сильнее начинает казаться, что это просто костыль такой являющийся следствием:

1. Крайне плохой совместимости диструтивов между собой
2. Стремлением бездумно писать софт таская с собой десятки либ разных версий. И очень часто из-за того, чтобы из огромной либы вызывать лишь один метод.

В итоге приложение-уродец без докера ну никак не заработает.

Нормально спроектированному софту никакие докеры не нужны.

Докер просто предлагает кардинальный способ решения проблемы и предоставляет для этого универсальный инструмент.

Ну во первых проблема не столько в различных дистрибутивах, а в их различные поколения. Типичная проблема: в одном приложении библиотека aaa.so требуется версии 1.31 и не выше, а другом приложении так же aaa.so, но не ниже 1.33. И эта проблема возникает **на одном и том же дистрибутиве** запросто.

Вот вторых вы не можете написать все ПО всего мира. Типичный серверный проект (а Докер прежде всего для серверов решение) — скорее всего будет использовать кучу не написанного Вами лично ПО. **Помимо вашего собственного.**

С Докером мы просто получаем то, что могли бы получить если бы программисты писали код строго по правилам — то есть получаем софт, который смотрит в мир жестко определенным портом и пишет на диск в жестко определенное место. И никому не мешает. Все остальное — сочетать этот софт друг с другом — это уже задача админа.

Собственно в современных средствах написани программ также предпринято попытки создавать программы без Докера и без проблемы с зависимостями. Например Go/golang представляет из себя единственный бинарный файл, который может запускаться на любой системе (разумеется бинарный файл должен соответствовать классу системы 32/64 бита, Linux, FreeBSD, Windows, MacOSX). На конкретный дистрибутив операционной системе Go плевать.

Докер же пытается решить эту проблему более глобально — для любого ПО. Независимо от того, сколько дополнительных файлов нужно этому ПО для работы. Строго говоря, подход Докера соответствует подходу заказчиков — наплевать что там внутри, пусть программисты извращаются как хотят, если им так удобнее, если так лучше для дела. Но это должно работать.



beduin01 24 января 2018 в 12:14



↑ +1 ↓

Проблема на 80% является средством использования разных кривых инструментов. NodeJS яркий пример. Там Hello World без десятка плагинов и тройки оберток сделать практически невозможно. Как следствие любой большой проект превращается к грудку трудноподдерживаемого говна.

В итоге поддержка всего этого превращается просто в ад.

Просто скоро мы придем к тому, что появятся обертки над Докером, у которого со временем свои приколы вылезут и с версиями и с еще чем-то там. А первопричина проблем так и останется.



justboris 24 января 2018 в 14:02



↑ +2 ↓

| NodeJS яркий пример.

Не соглашусь. NodeJS как раз хороший пример портательных приложений, которые все зависимости хранят в своей папке, а не разбрасывают по разным глобальным системным путям.



InoMono 30 января 2018 в 16:51



↑ +1 ↓

| Проблема на 80% является средством использования разных кривых инструментов.

Так уж сложилось, что программистами в этом мире являются и еще другие люди, кроме вас, идеального программиста.

Вместо того, чтобы хаять других, авторы Докера просто решили проблему. Программа не выпускается за пределы контейнера.

А хаять других не помогает — ведь постоянно новые программисты появляются.

| Просто скоро мы придем к тому, что появятся обертки над Докером, у которого со временем свои приколы вылезут и с версиями и с еще чем-то там. А первопричина проблем так и останется.

Ну а вы что предлагаете?

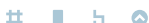
Расстреливать за «кривую программу»?

Кто кривизну будет определять?

Не бойтесь, что в каких то условиях и ваша программа будет менее прямой, чем у большинства?



justboris 23 января 2018 в 23:05



↑ 0 ↓

Едиобразия в версиях и пакетах можно добиться только если вы являетесь единоличным владельцем кода. У разных команд разработчиков может быть свое мнение какие библиотеки им нужны.

Поэтому либо разносить модули по разным серверам, либо завернуть их в докер на одном хосте и сэкономить на расходах на сервера.



InoMono 24 января 2018 в 00:10

↑ 0 ↓

У разных команд разработчиков может быть свое мнение какие библиотеки им нужны.

Более того: это могут быть разные языки программирования



m0nstermind 23 января 2018 в 10:59

↑ +1 ↓

В «Часть 0.1 Сравнение с VM» строго говоря все сильно не правда. Докер нельзя назвать средством виртуализации ни в коей мере. Он является оберткой вокруг средств контейнеризации (cgroups + namespaces) конкретно linux kernel.

В последнее время MS тоже начал что то подобное вливать в Windows. Но! Контейнеры для linux и windows не совместимы между собой ни в коей мере (есть решения, запускающие контейнеры linux в других ОС, но они используют отдельные настоящие виртуальные машины с guest os linux в них).

Подробнее о различиях виртуализации и контейнеризации можно посмотреть здесь:

www.youtube.com/watch?v=thcE53dogZk&t=1s&list=PLrCZzMib1e9rZohs_FJg8MK52Ey494z40&index=11



bykvaadm 23 января 2018 в 11:55

↑ 0 ↓

Согласен, это вводит в заблуждение. Я пытался более простым языком провести аналогию и видимо допроводился. Попробую поправить. Скорее здесь надо было заменить слово «виртуализации» на слово «изоляция».



yse 23 января 2018 в 11:22

↑ 0 ↓

Спасибо за статью! Хабр торт



AntonAK83 23 января 2018 в 12:16

↑ 0 ↓

Такой вопрос, можно ли докер контейнеру присваивать несколько IP адресов?



bykvaadm 23 января 2018 в 12:19

↑ 0 ↓

наверное это не совсем то, что вы хотите. вы можете настроить свою сеть для контейнера:
docs.docker.com/engine/userguide/networking

Но в пределах одного хоста, вероятно вы хотели следующее:

```
docker run -p 172.16.10.20:80:80 -p 192.168.10.20:443:443 . . . . .
```

Что будет направлять пакеты, приходящие на конкретный адрес и порт хоста внутрь контейнера. А эти адреса уже нужно будет настроить на интерфейсах хоста. В этом случае вам сеть докера трогать не нужно.



nskforward 23 января 2018 в 14:30

↑ 0 ↓

Контейнер необходимо добавить в разные подсети



maolo 23 января 2018 в 16:34

↑ 0 ↓

Выходные данные не должны сохраняться внутри контейнера (можно, но это не docker-way).

О, а это утверждение я впервые встречаю! Только присматриваюсь к докеру, как раз и думал, что удобно и код проекта внутри контейнера хранить...



bykvaadm 23 января 2018 в 16:55

↑ 0 ↓

код проекта — библиотеки, бинарники — т.е. само рабочее мясо хранится в контейнере. а вот те данные, которые получаете на выходе — вот их как раз не нужно хранить внутри.

ну, например, база данных mysql: сам демон живет в контейнере, а база данных — снаружи.

Вы, возможно меня так поняли.

 **nanshakov** 23 января 2018 в 17:35 # 0

Автор, большое спасибо, пишите еще.

А как сделать такое, например у меня проект на C++;


Компиляция

Развертывание в контейнере

преднастройка (если надо)

Запуск(тесты)

максимально автоматически

 **RidgeA** 23 января 2018 в 18:37 # 2

возможно я помогу — docs.docker.com/engine/userguide/eng-image/multistage-build/#use-multi-stage-builds

Не так давно добавили multistage build — когда в докерфайл пишется несколько контейнеров, каждый из которых может брать артефакты из предыдущего.

Например для компиляции нужна полноценная операционка, с кучей либ и компилятором, а для запуска — только бинарник, внешние ресурсы (файлы конфигов и т. п.) и пара либ. Весь компилятор тащить нет смысла в рабочий образ, т.к. размер сильно может раздуть.


Ну и плюс для подобных вещей используют CI инструменты — Jenkins, GitLabCI, и т. д.

 **bykvaadm** 23 января 2018 в 18:37 # 1

jenkins. пишете различные стадии (stage) в jenkinsfile (groovy lang), и с помощью докера и скриптов этот вопрос решается.

и будет это выглядеть так:

git push -> hook -> jenkins master server -> jenkins builder (здесь выполняются ваши операции, билды, тесты, выгрузка в репозиторий..)

 **yvm** 23 января 2018 в 18:03 # 0

>Часть 4: Удобный запускатор ваших сервисов

А вы точно понимаете смысл опции --restart=always?

 **viiy** 23 января 2018 в 18:55 # 5

До сих пор, спустя много времени, не могу согласиться что докер прямо таки незаменимая вещь в разработке. А еще не разу не видел железобетонного примера use-case, когда можно сказать «вот это да! без этого совершенно никак». Такое ощущение, что люди не понимают что делают, а просто гонятся за технологией без причины. Вы знаете, у нас много разработчиков, которые пишут микросервисы. Микросервисов около 30 видов, экземпляров еще больше, и они крутятся, не поверите, просто не железе, даже без виртуализации. Мы научились с этим жить и переход на докер до сих пор не очень понятен. Написаны сборки и деплой, которые собирают сервисы, выкатывают это на stage, prod. Есть мониторинг который следит за жизнью сервисов. Оговорка — микросервисы на golang. Возможно, поэтому не страшно что какая то либа будет отличаться на машине разработчика и продакшене. Не могу сказать что у нас хайлоад, но 20 тысяч пользователей онлайн набирается. Очевидно, что бы мог нам принести докер тоже понятно: более удобный способ хранения сервисов, их деплой и роллбэк (было бы просто меньше ansible), управление ресурсами (с swarm, nomad, etc), чуть удобней собирать логи. И кажется — это все. При этом добавляется головная боль, когда проявляются особенности докера. У нас часть сборок (composer, prtm) делается в докере, поэтому о глюках знаем не по наслышке. А так же особенности работы с файловой системой, прокидывание портов и в целом работа с сетью, авторегистрация. Бывают контейнеры просто зависают и причину выявить проблематично. Поэтому каждый раз, когда читаю восторженную статью про докер для одного-двух микросервисов, хочется задать вопрос — вам шашечки или ехать? Иногда докер дает не так много полезного, а вот принести может много проблем. И все же мы рассматриваем варианты перехода на докер + оркестрацию, во имя удобства деплоя и гибкости управления ресурсами.

За статью спасибо, автор молодец.

 **KIVagant** 25 января 2018 в 03:35 # 0

Особенность Докера пожалуй в том, что его польза неочевидна, если его не использовать или использовать только для решения каких-то узких задач. Вся красота раскрывается если вам приходится работать с неопределённого размера распределённой командой, которая использует разнообразный зоопарк решений. Попытки автоматизировать это всё с Ansible/Salt/... выльются в кровавые слёзы. Нужно просто разделять назначения инструментов. Ansible — прекрасная вещь, но я точно не захотел бы управлять чем-либо вроде проектов на nodejs с её помощью.

 **emacsway** 30 января 2018 в 10:25    

 0 

верно, — «Of course, you can do a better job if you have more tools in your toolbox than if you have fewer, but it is much more important to have a handful of tools that you know when not to use, than to know everything about everything and risk using too much solution.» (Kent Beck)

P.S.: автор действительно молодец.

 **cbone**  24 января 2018 в 15:28  

 0 

Спасибо за статью, очень интересно.

Я правильно понимаю, что если у меня к примеру есть NodeJS приложение, то я могу его задокеризировать, запустить несколько таких контейнеров и получу возможность распаралеливания обработки запросов по ядрам уже не на уровне приложения а на уровне инфраструктуры? Никаких заморочек с кластеризацией приложения? Подразумевается, что приложения в контейнерах stateless. И гипотетический кейс: есть к примеру 32 физических сервера. 30 для запуска контейнеров (stateless сервисы), 2 для БД основная и реплика для отказоустойчивости (stateful). Контейнеры что-то делают, взаимодействуют между собой, а потом пишут результат в базу. Это же Docker way? Как в такой инфраструктуре минимизировать администрирование хостовой ОС? Или просто берём Ansible, CoreOS и вперёд?

 **KlVagant** 25 января 2018 в 03:30    

 0 



Посмотрите на Kubernetes + Helm

 **porn**  27 января 2018 в 04:08    

 0 

32 физических сервера. 30 для запуска контейнеров

Вообще, да, почему нет. Но, чтобы не держать все яйца в одной корзине, можно раскидать разных контейнеров по 2-3 на физический сервер, я думаю.

 **niya3** 17 февраля 2018 в 17:01  

 0 

Посмотрите его pid — он равен единице.

Можно проще: `echo $$`

Но тогда не получится демонстрации с `command not found` :)

Финт ушами. аттачимся к уже запущенному контейнеру с помощью команды `exec`

...

`run` — взять образ X и создать контейнер Z с процессом Y

`exec` — взять контейнер Z и запустить в нем процесс N, при этом процесс Y будет работать как и прежде.

`docker attach` — подключиться к процессу Y контейнера Z.

 **bykvaadm** 19 февраля 2018 в 16:43    

 0 

здесь преследуются разные цели. в случае с `attach` подключается к существующему процессу, в случае `exec` — процесс создается рядом. Т.е. например в 99% это шелл для отладки

 **greenkey** 5 июня 2019 в 00:25  

 0 

давно собирался пройти по такому кейсу и понять чем так хорош докер...

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

Как американцы живого хорька в коллайдер засунули

↑ +63

👁 33,5k

🔖 36

💬 37

Что айтишнику не стоит делать в 2020?

↑ +33

👁 29,5k

🔖 94

💬 59

В Windows 10 версии 2004 можно отслеживать температуру видеокарты, а новые драйверы будут помечать как обновления

↑ +15

👁 21,9k

🔖 2

💬 38

Еще один способ высокотехнологичного мошенничества

↑ +73

👁 35,9k

🔖 79

💬 91

Посвящается: технологии, которые умерли в 2019 году

↑ +23

👁 19,4k

🔖 35

💬 37

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Правила	Реклама
Регистрация	Новости	Помощь	Тарифы
	Хабы	Документация	Контент
	Компании	Соглашение	Семинары
	Пользователи	Конфиденциальность	Мегaproекты
	Песочница		

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.

