

DockerCon 2023: Наше ежегодное мероприятие для разработчиков возвращается – онлайн и лично. [Узнать больше.](#) ✕

[Главная](#) / [Ссылка](#) / [Ссылка на файл Dockerfile](#)

Ссылка

Docker может создавать изображения автоматически, читая инструкции из а `Dockerfile`. А `Dockerfile` - это текстовый документ, содержащий все команды, которые пользователь может вызвать в командной строке для сборки изображения. На этой странице описаны команды, которые вы можете использовать в `Dockerfile`.


Дайте отзыв

Формат

Вот формат `Dockerfile`:

```
# Comment
INSTRUCTION arguments
```

Инструкция регистр не учитывается. Однако Конвенция по их данным ЗАГЛАВНЫМИ буквами, чтобы отличить их легко из аргументов.

Docker выполняет инструкции в `Dockerfile` порядке. А `Dockerfile` должно начинаться с `FROM` инструкции. Это может быть после [директив синтаксического анализатора](#), [комментариев](#) и [аргументов](#) с глобальной областью действия. `FROM` Инструкция указывает [родительский образ](#) , из которого вы создаете. `FROM` может предшествовать только одна или несколько `ARG` инструкций, в которых объявляются аргументы, которые используются в `FROM` строках в `Dockerfile`.

Docker обрабатывает строки, которые *начинаются* с `#`, как комментарий, если только строка не является допустимой [директивой синтаксического анализатора](#). `#` Маркер в любом другом месте строки обрабатывается как аргумент. Это позволяет использовать такие инструкции, как:

comment

```
RUN echo 'we are running some # of cool things'
```

Строки комментариев удаляются перед выполнением инструкций Dockerfile, что означает, что комментарий в следующем примере не обрабатывается командной оболочкой выполнение `echo` команды, и оба примера ниже эквивалентны:

```
RUN echo hello \  
# comment  
world
```

```
RUN echo hello \  
world
```

Дайте отзыв

Символы продолжения строки в комментариях не поддерживаются.

❗ Обратите внимание на пробелы

Для обеспечения обратной совместимости вводные пробелы перед комментариями (`#`) и инструкциями (такими как `RUN`) игнорируются, но не рекомендуется. В этих случаях начальный пробел не сохраняется, и поэтому следующие примеры эквивалентны:

```
    # this is a comment-line  
RUN echo hello  
RUN echo world
```

```
# this is a comment-line  
RUN echo hello  
RUN echo world
```

Однако обратите внимание, что пробелы в *аргументах* инструкции, таких как команды следующие `RUN`, сохраняются, поэтому следующий пример печатается `hello world` с начальным пробелом, как указано:

```
RUN echo "\
```

```
hello\  
world"
```

Ссылка

Директивы синтаксического анализатора являются необязательными и влияют на способ обработки последующих строк в `Dockerfile`. Директивы синтаксического анализатора не добавляют слои к сборке и не будут показаны как шаг сборки. Директивы синтаксического анализатора записываются в виде специального типа комментария в форме

```
# directive=value
```

. Одна директива может быть использована только один раз.

После обработки комментария, пустой строки или инструкции конструктора Docker больше не ищет директивы синтаксического анализатора. Вместо этого он обрабатывает все, что отформатировано как директива синтаксического анализатора, как комментарий и не пытается проверить, может ли это быть директивой синтаксического анализатора. Следовательно, все директивы синтаксического анализатора должны находиться в самом верху `Dockerfile`.

Дайте отзыв

Директивы синтаксического анализатора не чувствительны к регистру. Однако по соглашению они должны быть строчными. По соглашению также следует включать пустую строку после любых директив синтаксического анализатора. Символы продолжения строки не поддерживаются в директивах синтаксического анализатора.

Из-за этих правил все следующие примеры недопустимы:

Недопустимо из-за продолжения строки:

```
# direc \  
tive=value
```

Недействителен из-за двойного отображения:

```
# directive=value1  
# directive=value2
```

```
FROM ImageName
```

Рассматривается как комментарий из-за появления после инструкции разработчика:

```
FROM ImageName  
# directive=value
```

Обрабатывается как комментарий из-за появления после комментария, который не является синтаксическим анализатором директива:

```
# About my dockerfile  
# directive=value  
FROM ImageName
```

Неизвестная директива обрабатывается как комментарий из-за того, что она не распознана. В кроме того, известная директива обрабатывается как комментарий из-за того, что она появляется после комментария, который не является директивой синтаксического анализатора.

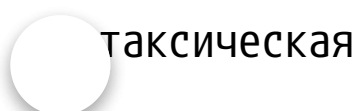
```
# unknowndirective=value  
# knowndirective=value
```

В директиве синтаксического анализатора разрешены пробелы, не разбивающие строку. Следовательно, все следующие строки обрабатываются одинаково:

```
#directive=value  
# directive =value  
#  directive= value  
# directive = value  
#      dIrEcTiVe=value
```

Поддерживаются следующие директивы синтаксического анализатора:

- `syntax`
- `escape`



Дайте отзыв

Эта функция доступна только при использовании серверной части [BuildKit](#) и игнорируется при использовании серверной части classic builder.

Для получения дополнительной информации смотрите [синтаксис пользовательского файла Dockerfile](#).

escape

```
# escape=\ (backslash)
```

Или

```
# escape=` (backtick)
```

Дайте отзыв

`escape` Директива задает символ, используемый для экранирования символов в а `Dockerfile`. Если не указано, то по умолчанию используется экранирующий символ `\`.

Управляющий символ используется как для экранирования символов в строке, так и для экранирования новой строки. Это позволяет `Dockerfile` инструкции занимать несколько строк. Обратите внимание, что независимо от того, включена ли в `escape` директива синтаксического анализатора `Dockerfile`, экранирование в `RUN` команде выполняется *только в конце строки*.

Установка ескаре-символа на ``` особенно полезна для `Windows`, где `\` это разделитель пути к каталогу. ``` совместим с [Windows PowerShell](#).

Рассмотрим следующий пример, который неочевидным образом приведет к сбою на `Windows`. Вторая строка `\` в конце второй строки будет интерпретироваться как экранирование для новой строки, а не как цель экранирования из первой `\`. Аналогично, `\` в конце третьей строки, предполагая, что она фактически обрабатывалась как инструкция, заставила бы ее рассматриваться как продолжение строки. Результатом этого файла `dockerfile` является то, что вторая и третья строки считаются одной инструкцией:

```
FROM microsoft/nanoserver
COPY testfile.txt c:\
dir c:\
```

Результаты в:

```
PS E:\myproject> docker build -t cmd .

Sending build context to Docker daemon 3.072 kB
Step 1/2 : FROM microsoft/nanoserver
----> 22738ff49c6d
Step 2/2 : COPY testfile.txt c:\RUN dir c:
GetFileAttributesEx c:RUN: The system cannot find the file specified.
PS E:\myproject>
```

Одним из решений было бы использовать `/` в качестве цели как `COPY` инструкция и `dir`. Однако этот синтаксис в лучшем случае сбивает с толку, поскольку он не является естественным для путей в `Windows`, а в худшем случае подвержен ошибкам, поскольку не все команды в `Windows` поддерживают `/` разделитель путей.

Дайте отзыв

При добавлении `escape` директивы синтаксического анализа следующее `Dockerfile` выполняется, как и ожидалось, с использованием естественной семантики платформы для путей к файлам на `Windows`:

```
# escape=`

FROM microsoft/nanoserver
COPY testfile.txt c:\
RUN dir c:\
```

Результаты в:

```
PS E:\myproject> docker build -t succeeds --no-cache=true .

Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM microsoft/nanoserver
----> 22738ff49c6d
Step 2/3 : COPY testfile.txt c:\
----> 96655de338de
Removing intermediate container 4db9acbb1682
Step 3/3 : RUN dir c:\
----> Running in a2c157f842f5
Volume in drive C has no label.
```

```
Volume Serial Number is 7E6D-E0F7
```

```
Directory of c:\
```

```
10/05/2016  05:04 PM                1,894 License.txt
10/05/2016  02:22 PM      <DIR>          Program Files
10/05/2016  02:14 PM      <DIR>          Program Files (x86)
10/28/2016  11:18 AM                62 testfile.txt
10/28/2016  11:20 AM      <DIR>          Users
10/28/2016  11:20 AM      <DIR>          Windows
                2 File(s)              1,956 bytes
                4 Dir(s)  21,259,096,064 bytes free
```

```
---> 01c7f3bef04f
```

```
Removing intermediate container a2c157f842f5
```

```
Successfully built 01c7f3bef04f
```

```
PS E:\myproject>
```

Ссылка

Переменные среды (объявленные с помощью [ENV инструкции](#)) также могут использоваться в определенных инструкциях как переменные, которые будут интерпретироваться [Dockerfile](#). Экранирование также обрабатывается для буквального включения синтаксиса, подобного переменной, в оператор.

Переменные среды обозначаются в [Dockerfile](#) либо с помощью `$variable_name`, либо `${variable_name}`. Они обрабатываются эквивалентно, и синтаксис фигурных скобок обычно используется для решения проблем с именами переменных без пробелов, например `${foo}_bar`.

`${variable_name}` Синтаксис также поддерживает несколько стандартных [bash](#) модификаторов, указанных ниже:

- `${variable:-word}` указывает, что если `variable` задано, то результатом будет это значение. Если `variable` не задано, то `word` будет результат.
- `${variable:+word}` указывает, что если `variable` установлено значение, то `word` будет результат, в противном случае результатом будет пустая строка.

Во всех случаях `word` может быть любой строкой, включая дополнительные переменные

Экранирование возможно путем добавления `\` перед переменной: `\$foo` или `\${foo}`, например, будет переводиться в `$foo` и `${foo}` литералы соответственно.

Пример (проанализированное представление отображается после `#`):

```
FROM busybox
ENV FOO=/bar
WORKDIR ${FOO} # WORKDIR /bar
ADD . $FOO # ADD . /bar
COPY \$FOO /quux # COPY $FOO /quux
```

Переменные среды поддерживаются следующим списком инструкций в `Dockerfile`:

- `ADD`
- `COPY`
- `ENV`
- `EXPOSE`
- `FROM`
- `LABEL`
- `STOPSIGNAL`
- `USER`
- `VOLUME`
- `WORKDIR`
- `ONBUILD` (в сочетании с одной из поддерживаемых инструкций выше)

При замене переменной среды будет использоваться одно и то же значение для каждой переменной на протяжении всей инструкции. Другими словами, в этом примере:

```
ENV abc=hello
ENV abc=bye def=$abc
ENV ghi=$abc
```

в результате `def` будет иметь значение `hello`, а не `bye`. Однако, `ghi` будет иметь значение `bye`, поскольку оно не является частью той же инструкции, для которой обновлено `abc` значение `bye`.

Дайте отзыв

Прежде чем docker CLI отправит контекст демону docker, он ищет файл с именем `.dockerignore` в корневом каталоге контекста. Если этот файл существует, CLI изменяет контекст, чтобы исключить файлы и каталоги, которые соответствуют шаблонам в нем. Это помогает избежать ненужной отправки больших или конфиденциальных файлов и каталогов демону и потенциального добавления их в изображения с помощью `ADD` или `COPY`.

Интерфейс командной строки интерпретирует `.dockerignore` файл как список шаблонов, разделенных новой строкой, аналогичный файловым глобусам оболочек Unix. Для целей сопоставления корневым каталогом контекста считается как рабочий, так и корневой каталог. Например, шаблоны `/foo/bar` и `foo/bar` оба исключают файл или каталог, указанный `bar` в `foo` подкаталоге `PATH` или в корне репозитория git, расположенного по адресу `URL`. Ни то, ни другое не исключает ничего другого.

Если строка в `.dockerignore` файле начинается с `#` в столбце 1, то эта строка рассматривается как комментарий и игнорируется до интерпретации CLI.

Вот пример `.dockerignore` файла:

```
# comment
*/temp*
**/temp*
temp?
```

Этот файл вызывает следующее поведение сборки:

| Правило | Поведение |
|------------------------|---|
| <code># comment</code> | Игнорируется. |
| <code>*/temp*</code> | Исключите файлы и каталоги, имена которых начинаются с <code>temp</code> , в любом ближайшем подкаталоге корневого каталога. Например, обычный файл <code>/somedir/temporary.txt</code> исключается, как и каталог <code>/somedir/temp</code> . |
| <code>**/temp*</code> | Исключите файлы и каталоги, начинающиеся с <code>temp</code> , из любого подкаталога, который находится на два уровня ниже корневого. Например, <code>/somedir/subdir/temporary.txt</code> исключается. |

Дайте отзыв



| Правило | Поведение |
|---------|-----------|
|---------|-----------|

| | |
|--------------------|---|
| <code>temp?</code> | Исключите файлы и каталоги в корневом каталоге, имена которых являются односимвольным расширением <code>temp</code> . Например, исключаются <code>/tempa</code> и <code>/tempb</code> . |
|--------------------|---|

Сопоставление выполняется с использованием [пути к файлу Go](#). Сопоставьте правила [правила](#).

На этапе предварительной обработки удаляются начальные и конечные пробелы и удаляются элементы `.` и `..`, использующие [путь к файлу Go](#). Очистите [Очистите](#). Строки, которые остаются пустыми после предварительной обработки, игнорируются.

Помимо пути к файлу Go. Правила соответствия, Docker также поддерживает специальную строку с подстановочным знаком `**`, которая соответствует любому количеству каталогов (включая ноль). Например, `**/*.go` будут исключены все файлы, заканчивающиеся на `.go`, которые находятся во всех каталогах, включая корневой контекст сборки.

Строки, начинающиеся с `!` (восклицательный знак), могут использоваться для создания исключений из исключений. Ниже приведен пример `.dockerignore` файла, который использует этот механизм:

```
*.md
!README.md
```

Все файлы markdown, находящиеся непосредственно в каталоге контекста, за `README.md` исключением исключены из контекста. Обратите внимание, что файлы markdown в подкаталогах по-прежнему включены.

Размещение `!` правил исключения влияет на поведение: последняя строка `.dockerignore`, которая соответствует определенному файлу, определяет, включен он или исключен. Рассмотрим следующий пример:

```
*.md
!README*.md
README-secret.md
```

Никакие файлы markdown не включены в контекст, за исключением файлов README, отличных от `README-secret.md`.

Теперь рассмотрим этот пример:

```
*.md
README-secret.md
!README*.md
```

Включены все файлы README. Средняя строка не влияет, потому что `!README*.md` совпадает `README-secret.md` и идет последней.

Вы даже можете использовать `.dockerignore` файл для исключения файлов `Dockerfile` и `.dockerignore`. Эти файлы по-прежнему отправляются демону, поскольку они нужны ему для выполнения своей работы. Но инструкции `ADD` и `COPY` не копируют их в изображение.

Наконец, вы можете указать, какие файлы включать в контекст, а какие исключать. Для достижения этой цели укажите `*` в качестве первого шаблона, за которым следует один или несколько `!` шаблонов исключений.

Дайте отзыв

❗ Примечание

По историческим причинам шаблон `.` игнорируется.

ПО


```
FROM [--platform=<platform>] <image> [AS <name>]
```

Или

```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
```

Или

```
FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

`FROM` Инструкция инициализирует новый этап сборки и устанавливает [Базовый образ](#)  последующих инструкций. Таким образом, действительный файл `Dockerfile` должен

начинаться с `FROM` инструкции. Изображение может быть любым допустимым изображением – особенно легко начать с извлечения изображения из [общедоступных репозиториях](#).

- `ARG` это единственная инструкция, которая может предшествовать `FROM` в `Dockerfile`. Смотрите, [как взаимодействуют ARG и FROM](#).
- `FROM` может появляться несколько раз в пределах одного `Dockerfile` для создания нескольких образов или использования одного этапа сборки в качестве зависимости для другого. Просто запишите последний идентификатор изображения, выведенный фиксацией перед каждой новой `FROM` инструкцией. Каждая `FROM` инструкция очищает любое состояние, созданное предыдущими инструкциями.
- При необходимости новому этапу сборки можно присвоить имя, добавив его `AS name` к `FROM` инструкции. Это имя может использоваться в последующих `FROM` и `COPY --from=<name>` инструкциях для ссылки на образ, созданный на этом этапе.
- Значения `tag` или `digest` являются необязательными. Если вы опустите любое из них, конструктор по умолчанию использует `latest` тег. Конструктор возвращает ошибку, если не может найти `tag` значение.

Дайте отзыв

Необязательный `--platform` флаг может использоваться для указания платформы изображения в случае, если `FROM` ссылается на многоплатформенный образ. Например, `linux/amd64`, `linux/arm64` или `windows/amd64`. По умолчанию используется целевая платформа запроса на сборку. В значении этого флага могут использоваться глобальные аргументы сборки, например, [автоматические аргументы платформы](#) позволяют вам принудительно перевести этап на собственную платформу сборки (`--platform=$BUILDPLATFORM`) и использовать ее для кросс-компиляции на целевую платформу внутри этапа.

Понять, как ARG и FROM взаимодействуют

`FROM` инструкции поддерживают переменные, которые объявляются любыми `ARG` инструкциями, которые встречаются перед первой `FROM`.

```
ARG CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD /code/run-app

FROM extras:${CODE_VERSION}
/ code/run-extras
```

`ARG` Объявленный перед а `FROM` находится за пределами стадии сборки, поэтому он не может быть использован ни в одной инструкции после а `FROM`. Чтобы использовать значение по умолчанию `ARG` объявленное перед первым, `FROM` используйте `ARG` инструкцию без значения внутри этапа сборки:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

ВЫПОЛНИТЬ

ЗАПУСК имеет 2 формы:

- `RUN <command>` (форма *оболочки*, команда выполняется в оболочке, которая по умолчанию находится `/bin/sh -c` в Linux или `cmd /S /C` в Windows)
- `RUN ["executable", "param1", "param2"]` (форма *exec*)

`RUN` Инструкция выполнит любые команды в новом слое поверх текущего изображения и зафиксирует результаты. Полученное зафиксированное изображение будет использовано для следующего шага в `Dockerfile`.

Многоуровневые `RUN` инструкции и генерация коммитов соответствуют основным концепциям Docker, где коммиты дешевы, а контейнеры могут быть созданы из любой точки истории изображения, во многом как система управления версиями.

Форма *exec* позволяет избежать переполнения строк оболочки и выполнять `RUN` команды с использованием базового образа, который не содержит указанного исполняемого файла оболочки.

Оболочка по умолчанию для формы *shell* может быть изменена с помощью `SHELL` команды.

В форме *оболочки* вы можете использовать `\` (обратную косую черту), чтобы продолжить выполнение одной инструкции на следующей строке. Например, рассмотрим эти две строки:

```
RUN /bin/bash -c 'source $HOME/.bashrc && \
    echo $HOME'
```

Дайте отзыв

Вместе они эквивалентны этой единственной строке:

```
RUN /bin/bash -c 'source $HOME/.bashrc && echo $HOME'
```

Чтобы использовать другую оболочку, отличную от `/bin/sh`, используйте форму *exec*, передаваемую в желаемую оболочку. Например:

```
RUN ["/bin/bash", "-c", "echo hello"]
```

❗ Примечание

Форма *exec* анализируется как массив JSON, что означает, что вы должны использовать двойные кавычки (") вокруг слов, а не одинарные кавычки (').

Дайте отзыв

В отличие от формы *shell*, форма *exec* не вызывает командную оболочку. Это означает, что обычная обработка оболочки не выполняется. Например, `RUN ["echo", "$HOME"]` не будет выполнять подстановку переменных в `$HOME`. Если вам нужна обработка оболочки, то либо используйте форму *оболочки*, либо запустите оболочку напрямую, например:

```
RUN [ "sh", "-c", "echo $HOME" ]
```

При использовании формы *exec* и непосредственном выполнении командной строки, как в случае с формой *shell*, расширение переменной среды выполняет оболочка, а не `docker`.

❗ Примечание

В форме *JSON* необходимо избегать обратных косых черт. Это особенно актуально в Windows, где обратная косая черта является разделителем путей. В противном случае следующая строка обрабатывалась бы как форма *оболочки* из-за недопустимого JSON и завершалась бы ошибкой неожиданным образом:

```
RUN ["c:\windows\system32\tasklist.exe"]
```

Правильный синтаксис для этого примера:

```
RUN ["c:\\windows\\system32\\tasklist.exe"]
```

Кэш для `RUN` инструкций не становится недействительным автоматически во время следующей сборки. Кэш для инструкции типа `RUN apt-get dist-upgrade -y` будет повторно использован во время следующей сборки. Кэш для `RUN` инструкций можно сделать недействительным, например, с помощью `--no-cache` флага `docker build --no-cache`.

Для получения дополнительной информации смотрите [Dockerfile](#) [Руководство по применению лучших практик](#) [↗](#).

Кэш для `RUN` инструкций может быть аннулирован с помощью `ADD` и `COPY` instructions.

Дайте отзыв

ВЫПОЛНИТЬ --смонтировать

❗ Примечание

Добавлено в `docker/dockerfile:1.2`

`RUN --mount` позволяет создавать подключения файловой системы, к которым доступна сборка. Это можно использовать для:

- Создание, привязка, подключение к файловой системе хоста или другие этапы сборки
- Доступ к секретам сборки или сокетам ssh-агента
- Используйте постоянный кэш управления пакетами для ускорения сборки

Синтаксис: `--mount=[type=<TYPE>][,option=<value>[,option=<value>]...]`

Типы монтирования

| Тип | Описание |
|----------------------------------|--|
| <code>bind</code> (по умолчанию) | Привязка-монтирование контекстных каталогов (только для чтения). |
| <code>cache</code> | Смонтируйте временный каталог для кэширования каталогов компиляторов и менеджеров пакетов. |

| Тип | Описание |
|---------------------|---|
| <code>secret</code> | Разрешить контейнеру сборки получать доступ к защищенным файлам, таким как закрытые ключи, не запекая их в образ. |
| <code>ssh</code> | Разрешить контейнеру сборки получать доступ к SSH-ключам через SSH-агенты с поддержкой парольных фраз. |

ВЫПОЛНИТЬ --смонтировать=тип= привязать

Этот тип монтирования позволяет привязывать файлы или каталоги к контейнеру сборки. По умолчанию монтирование с привязкой доступно только для чтения.

| Опция | Описание |
|---|---|
| <code>target</code> ¹ | Путь монтирования. |
| <code>source</code> | Исходный путь в <code>from</code> . По умолчанию используется корень <code>from</code> . |
| <code>from</code> | Название этапа сборки или образа для корня исходного кода. По умолчанию используется контекст сборки. |
| <code>rw</code> , <code>readwrite</code> | Разрешить запись при монтировании. Записанные данные будут удалены. |

Дайте отзыв

ВЫПОЛНИТЬ --смонтировать=тип=кэш

Этот тип монтирования позволяет контейнеру сборки кэшировать каталоги для компиляторов и менеджеров пакетов.

| Опция | Описание |
|--|--|
| <code>id</code> | Необязательный идентификатор для идентификации отдельных кэшей. По умолчанию используется значение <code>target</code> . |
| <code>target</code> ¹ | Путь монтирования. |
| <code>ro</code> , <code>readonly</code> | Доступен только для чтения, если задан. |
| <code>sharing</code> | Одно из <code>shared</code> , <code>private</code> или <code>locked</code> . Значение по умолчанию равно <code>shared</code> . <code>shared</code> Монтирование кэша может использоваться одновременно несколькими авторами. <code>private</code> создает новое монтирование, если имеется несколько устройств записи. |

| Опция | Описание |
|---------------------|---|
| <code>locked</code> | приостанавливает второе устройство записи до тех пор, пока первое не освободит монтирование. |
| <code>from</code> | Этап сборки для использования в качестве основы для монтирования кэша. По умолчанию используется пустой каталог. |
| <code>source</code> | Вложенный путь в <code>from</code> для монтирования. По умолчанию используется корневой каталог <code>from</code> . |
| <code>mode</code> | Файловый режим для нового каталога кэша в восьмеричном формате. По умолчанию <code>0755</code> . |
| <code>uid</code> | Идентификатор пользователя для нового каталога кэша. По умолчанию <code>0</code> . |
| <code>gid</code> | Идентификатор группы для нового каталога кэша. По умолчанию <code>0</code> . |

Дайте отзыв

Содержимое каталогов кэша сохраняется между вызовами `builder` без аннулирования кэша команд. Монтирование кэша следует использовать только для повышения производительности. Ваша сборка должна работать с любым содержимым каталога кэша, поскольку другая сборка может перезаписать файлы или `bc` может очистить их, если требуется больше места для хранения.

Пример:

```
# syntax=docker/dockerfile:1
FROM golang
RUN --mount=type=cache,target=/root/.cache/go-build \
    go build ...
```

Пример:

```
# syntax=docker/dockerfile:1
FROM ubuntu
RUN rm -f /etc/apt/apt.conf.d/docker-clean; echo 'Binary::apt::APT::Keep-Downloaded-
RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
    apt update && apt-get --no-install-recommends install -y gcc
```

Apt необходим эксклюзивный доступ к своим данным, поэтому в кэшах используется опция `sharing=locked`, которая гарантирует, что несколько параллельных сборок с использованием одного и того же монтирования кэша будут ждать друг друга и не будут обращаться к одним и тем же файлам кэша одновременно. файлы кэша. Вы также могли бы использовать `sharing=private`, если в этом случае вы предпочитаете, чтобы каждая сборка создавала другой каталог кэша .

ВЫПОЛНИТЬ --mount=type=tmpfs

Этот тип монтирования позволяет монтировать tmpfs в контейнер сборки.

| Опция | Описание |
|----------------------------------|--|
| <code>target</code> ¹ | Путь монтирования. |
| <code>size</code> | Укажите верхний предел размера файловой системы. |

Дайте отзыв

ВЫПОЛНИТЬ --смонтировать=тип=секретную

Этот тип монтирования позволяет контейнеру сборки получать доступ к защищенным файлам, таким как закрытые ключи, не запекая их в образ.

| Опция | Описание |
|-----------------------|---|
| <code>id</code> | ИДЕНТИФИКАТОР секрета. По умолчанию используется базовое имя целевого пути. |
| <code>target</code> | Путь монтирования. По умолчанию используется значение <code>/run/secrets/</code> + <code>id</code> . |
| <code>required</code> | Если установлено значение <code>true</code> , команда выдает ошибку, когда секрет недоступен. По умолчанию используется значение <code>false</code> . |
| <code>mode</code> | Режим файла для секретного файла в восьмеричном формате. По умолчанию <code>0400</code> . |
| <code>uid</code> | Идентификатор пользователя для секретного файла. По умолчанию <code>0</code> . |
| <code>gid</code> | Идентификатор группы для секретного файла. По умолчанию <code>0</code> . |

Пример: доступ к



`yntax=docker/dockerfile:1`

```
FROM python:3
RUN pip install awscli
RUN --mount=type=secret,id=aws,target=/root/.aws/credentials \
    aws s3 cp s3://... ...
```

```
$ docker buildx build --secret id=aws,src=$HOME/.aws/credentials .
```

ВЫПОЛНИТЬ --mount=type=ssh

Этот тип монтирования позволяет контейнеру сборки получать доступ к SSH-ключам через SSH-агенты с поддержкой парольных фраз.

| Опция | Описание |
|----------|---|
| id | ИДЕНТИФИКАТОР сокета или ключа SSH-агента. По умолчанию используется значение "по умолчанию". |
| target | Путь к сокету агента SSH. По умолчанию используется значение <code>/run/buildkit/ssh_agent.\${N}</code> . |
| required | Если установлено значение <code>true</code> , команда выдает ошибку, когда ключ недоступен. По умолчанию используется значение <code>false</code> . |
| mode | Режим файла для сокета в восьмеричном формате. По умолчанию <code>0600</code> . |
| uid | Идентификатор пользователя для сокета. По умолчанию <code>0</code> . |
| gid | Идентификатор группы для сокета. По умолчанию <code>0</code> . |

Пример: доступ к

```
# syntax=docker/dockerfile:1
FROM alpine
RUN apk add --no-cache openssh-client
RUN mkdir -p -m 0700 ~/.ssh && ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
RUN --mount=type=ssh \
    ssh -q -T git@gitlab.com 2>&1 | tee /hello
# "Welcome to GitLab, @GITLAB_USERNAME_ASSOCIATED_WITH_SSHKEY" should be printed here
# with the type of build progress is defined as `plain`.
```

```
$ eval $(ssh-agent)
$ ssh-add ~/.ssh/id_rsa
(Input your passphrase here)
$ docker buildx build --ssh default=$SSH_AUTH_SOCK .
```

Вы также можете указать путь к `*.pem` файлу на хосте напрямую вместо `$SSH_AUTH_SOCK`. Однако файлы `pem` с парольными фразами не поддерживаются.

ВЫПОЛНИТЬ --сетевая

Примечание

Добавлено в `docker/dockerfile:1.1`

Дайте отзыв

`RUN --network` позволяет контролировать, в какой сетевой среде выполняется команда.

Синтаксис: `--network=<TYPE>`

Ссылка

| Тип | Описание |
|-------------------------------------|------------------------------------|
| <code>default</code> (по умолчанию) | Запуск в сети по умолчанию. |
| <code>none</code> | Запуск без доступа к сети. |
| <code>host</code> | Выполняется в сетевой среде хоста. |

ВЫПОЛНИТЬ --сеть=

Что эквивалентно отсутствию флага вообще, команда выполняется в сети по умолчанию для сборки.

ВЫПОЛНИТЬ --network=none

Команда выполняется без доступа к сети (`lo` по-прежнему доступна, но изолирована от остального процесса)

Пример: изоляция

```
# syntax=docker/dockerfile:1
FROM python:3.6
ADD mypackage.tgz wheels/
RUN --network=none pip install --find-links wheels mypackage
```

`pip` will only be able to install the packages provided in the tarfile, which can be controlled by an earlier build stage.

RUN --network=host

The command is run in the host's network environment (similar to `docker build --network=host`, but on a per-instruction basis)

Дайте отзыв

ⓘ Warning

Использование `--network=host` защищено `network.host` правом, которое необходимо включить при запуске демона buildkitd с

`--allow-insecure-entitlement network.host` флагом или в [конфигурации buildkitd](#) [↗](#), а также для запроса на сборку с `--allow network.host` [флагом](#) [↗](#). `{:.warning}`

ВЫПОЛНИТЬ --безопасность

ⓘ Примечание

Пока недоступен в стабильном синтаксисе, используйте `docker/dockerfile:1-labs` версию.

ВЫПОЛНИТЬ --security=небезопасная

С помощью `--security=insecure` builder запускает команду без изолированной среды в небезопасном режиме, что позволяет запускать потоки, требующие повышенных привилегий (например, `containerd`). Это эквивалентно запуску `docker run --privileged`.

ⓘ Предупреждение

Чтобы получить доступ к этой функции, право `security.insecure` должно быть включено при запуске демона `buildkitd` с

`--allow-insecure-entitlement security.insecure` флагом или в [конфигурации buildkitd](#) [↗](#), а также для запроса на сборку с `--allow security.insecure` [флагом](#) [↗](#). `{:}.warning`

Пример:

```
# syntax=docker/dockerfile:1-labs
FROM ubuntu
RUN --security=insecure cat /proc/self/status | grep CapEff
```

Дайте отзыв

```
#84 0.093 CapEff: 0000003fffffffff
```

ВЫПОЛНИТЬ

Режим изолированной среды по умолчанию может быть активирован через

`--security=sandbox`, но это не-оп.

CMD Инструкция имеет три формы:

- `CMD ["executable","param1","param2"]` (форма *exec*, это предпочтительная форма)
- `CMD ["param1","param2"]` (в качестве *параметров по умолчанию для ТОЧКИ входа*)
- `CMD command param1 param2` (форма *оболочки*)

В **CMD** а может быть только одна **Dockerfile** инструкция. Если вы перечислите более одной, **CMD** то вступит в силу только последняя **CMD**.

Основная цель а **CMD** - предоставить значения по умолчанию для исполняемого **йнера**. Эти значения по умолчанию могут включать исполняемый файл или они могут

не включать исполняемый файл, и в этом случае вы также должны указать `ENTRYPOINT` инструкцию.

Если `CMD` используется для предоставления аргументов по умолчанию для `ENTRYPOINT` инструкции, то обе инструкции `CMD` и `ENTRYPOINT` должны быть указаны в формате массива JSON.

❗ Примечание

Форма `exec` анализируется как массив JSON, что означает, что вы должны использовать двойные кавычки (") вокруг слов, а не одинарные кавычки (').

В отличие от формы `shell`, форма `exec` не вызывает командную оболочку. Это означает, что обычная обработка оболочки не выполняется. Например, `CMD ["echo", "$HOME"]` не будет выполнять подстановку переменных в `$HOME`. Если вам нужна обработка оболочки, то либо используйте форму `оболочки`, либо запустите оболочку напрямую, например: `CMD ["sh", "-c", "echo $HOME"]`. При использовании формы `exec` и непосредственном выполнении командной строки, как в случае с формой `shell`, расширение переменной среды выполняет оболочка, а не `docker`.

При использовании в форматах `shell` или `exec` `CMD` инструкция задает команду, которая будет выполняться при запуске образа.

Если вы используете форму `оболочки` `CMD`, то `<command>` будет выполняться в `/bin/sh -c`:

```
FROM ubuntu
CMD echo "This is a test." | wc -
```

Если вы хотите запустить свой `<command>` без оболочки, то вы должны выразить команду в виде массива JSON и указать полный путь к исполняемому файлу. Эта форма массива является предпочтительным форматом `CMD`. Любые дополнительные параметры должны быть индивидуально выражены в виде строк в массиве:

```
FROM ubuntu
["/usr/bin/wc", "--help"]
```

Дайте отзыв

If you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See [ENTRYPOINT](#).

If the user specifies arguments to `docker run` then they will override the default specified in `CMD`.

Note

Не путайте `RUN` с `CMD`. `RUN` на самом деле выполняет команду и фиксирует результат; `CMD` ничего не выполняет во время сборки, но указывает предполагаемую команду для образа.

ЯРЛЫК

Дайте отзыв

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

`LABEL` Инструкция добавляет метаданные к изображению. А `LABEL` - это пара ключ-значение. Чтобы включить пробелы в `LABEL` значение, используйте кавычки и обратную косую черту, как при синтаксическом анализе командной строки. Несколько примеров использования:

```
LABEL "com.example.vendor"="ACME Incorporated"  
LABEL com.example.label-with-value="foo"  
LABEL version="1.0"  
LABEL description="This text illustrates \  
that label-values can span multiple lines."
```

Изображение может иметь более одной метки. Вы можете указать несколько меток в одной строке. До Docker 1.10 это уменьшало размер конечного изображения, но теперь это не так. Вы все еще можете указать несколько меток в одной инструкции одним из следующих двух способов:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```



```

LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"

```

Note

Be sure to use double quotes and not single quotes. Particularly when you are using string interpolation (e.g. `LABEL example="foo-$ENV_VAR"`), single quotes will take the string as is without unpacking the variable's value.

Labels included in base or parent images (images in the `FROM` line) are inherited by your image. If a label already exists but with a different value, the most-recently-applied value overrides any previously-set value.

Дайте отзыв

To view an image's labels, use the `docker image inspect` command. You can use the `--format` option to show just the labels;

```
{% raw %}
```

```
$ docker image inspect --format='{{json .Config.Labels}}' myimage
```

```
{% endraw %}
```

```

{
  "com.example.vendor": "ACME Incorporated",
  "com.example.label-with-value": "foo",
  "version": "1.0",
  "description": "This text illustrates that label-values can span multiple lines.",
  "multi.label1": "value1",
  "multi.label2": "value2",
  "other": "value3"
}

```

ПРОВОЖДАЮЩИЙ (устаревший)

```
MAINTAINER <name>
```

MAINTAINER Инструкция задает поле *Автора* сгенерированных изображений. **LABEL** Инструкция является гораздо более гибкой версией этой программы, и вам следует использовать ее вместо этого, поскольку она позволяет устанавливать любые требуемые метаданные и может быть просмотрена легко, например с помощью `docker inspect`. Чтобы задать метку, соответствующую полю **MAINTAINER**, вы могли бы использовать:

```
LABEL org.opencontainers.image.authors="SvenDowideit@home.org.au"
```

Затем это будет видно из `docker inspect` вместе с другими метками.

Дайте отзыв

ВЫСТАВИТЬ

```
EXPOSE <port> [<port>/<protocol>...]
```

EXPOSE Инструкция информирует Docker о том, что контейнер прослушивает указанные сетевые порты во время выполнения. Вы можете указать, прослушивает ли порт TCP или UDP, и по умолчанию используется TCP, если протокол не указан.

EXPOSE Инструкция фактически не публикует порт. Она функционирует как тип документации между пользователем, который создает образ, и пользователем, который запускает контейнер, о том, какие порты предназначены для публикации. Чтобы фактически опубликовать порт при запуске контейнера, используйте `-p` флаг на `docker run` для публикации и сопоставления одного или нескольких портов или `-P` флаг для публикации всех доступных портов и сопоставления их с портами высокого порядка.

По умолчанию **EXPOSE** предполагается TCP. Вы также можете указать UDP:

```
EXPOSE 80/udp
```

Чтобы предоставлять доступ как к TCP, так и к UDP, включите две строки:

```
EXPOSE 80/tcp
```

```
EXPOSE 80/udp
```

В этом случае, если вы используете `-P` with `docker run`, порт будет предоставлен один раз для TCP и один раз для UDP. Помните, что `-P` используется эфемерный хост высокого порядка порт на хосте, поэтому порт не будет одинаковым для TCP и UDP.

Независимо от `EXPOSE` настроек, вы можете переопределить их во время выполнения, используя `-p` флаг. Например

```
$ docker run -p 80:80/tcp -p 80:80/udp ...
```

Чтобы настроить перенаправление портов в хост-системе, см. [использование флага -P](#).
`docker network` Команда поддерживает создание сетей для обмена данными между контейнерами без необходимости предоставлять или публиковать определенные порты, поскольку контейнеры, подключенные к сети, могут обмениваться данными друг с другом через любой порт. Для получения подробной информации см. [обзор этой функции](#).

Дайте отзыв

ENV

```
ENV <key>=<value> ...
```

`ENV` Инструкция присваивает переменной среды `<key>` значение `<value>`. Это значение будет в среде для всех последующих инструкций на этапе сборки и может быть [заменено встроенным](#) во многих также. Значение будет интерпретироваться для других переменных среды, поэтому символы кавычек будут удалены, если они не экранированы. Подобно синтаксическому анализу командной строки, кавычки и обратная косая черта могут использоваться для включения пробелов в значения.

Пример:

```
ENV MY_NAME="John Doe"  
ENV MY_DOG=Rex\ The\ Dog  
ENV MY_CAT=fluffy
```

Инструкция позволяет устанавливать несколько `<key>=<value> ...` переменных одновременно, и приведенный ниже пример даст те же чистые результаты в конечном итоге

изображение:

```
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \  
    MY_CAT=fluffy
```

Переменные среды, установленные с помощью `ENV`, будут сохраняться при запуске контейнера из результирующего изображения. Вы можете просмотреть значения с помощью `docker inspect` и изменить их с помощью `docker run --env <key>=<value>`.

Этап наследует любые переменные среды, которые были установлены с помощью `ENV` его родительского этапа или любого предка. Обратитесь [сюда](#) для получения дополнительной информации о многоступенчатых сборках.

Сохранение переменной среды может вызвать неожиданные побочные эффекты. Например, настройка `ENV DEBIAN_FRONTEND=noninteractive` изменяет поведение `apt-get` и может сбить с толку пользователей вашего изображения.

Если переменная среды необходима только во время сборки, а не в конечном изображении, рассмотрите возможность установки значения для одной команды вместо этого:

```
RUN DEBIAN_FRONTEND=noninteractive apt-get update && apt-get install -y ...
```

Или использование `ARG`, которое не сохраняется в конечном изображении:

```
ARG DEBIAN_FRONTEND=noninteractive  
RUN apt-get update && apt-get install -y ...
```

Дайте отзыв

❗ Альтернативный синтаксис

`ENV` Инструкция также допускает альтернативный синтаксис `ENV <key> <value>`, опуская `=`. Например:

```
ENV MY_VAR my-value
```

Этот синтаксис не позволяет устанавливать несколько переменных среды в одной инструкции и может сбивать с толку. Например, следующее задает

единственную переменную среды (`ONE`) со значением `"TWO= THREE=world"`:

```
ENV ONE TWO= THREE=world
```

Альтернативный синтаксис поддерживается для обеспечения обратной совместимости, но не рекомендуется по причинам, изложенным выше, и может быть удален в будущей версии.

Добавить

ДОБАВИТЬ имеет две формы:

```
ADD [--chown=<user>:<group>] [--chmod=<perms>] [--checksum=<checksum>] <src>... <dest>  
ADD [--chown=<user>:<group>] [--chmod=<perms>] ["<src>", ... "<dest>"]
```

Последняя форма требуется для путей, содержащих пробелы.

Примечание

Функции `--chown` и `--chmod` поддерживаются только в файлах Dockerfile, используемых для сборки контейнеров Linux, и не будут работать в контейнерах Windows. Поскольку концепции владения пользователями и группами не переводятся между Linux и Windows, использование `/etc/passwd` и `/etc/group` для перевода имен пользователей и групп в идентификаторы ограничивает эту функцию жизнеспособностью только для контейнеров на базе ОС Linux.

Примечание

`--chmod` поддерживается начиная с [Dockerfile 1.3](#). В настоящее время поддерживается только восьмеричная система счисления. Поддержка не-восьмеричных чисел отслеживается в [moby / buildkit #1951](#).

The `ADD` instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the image at the path `<dest>`.

Multiple `<src>` resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.

Each `<src>` may contain wildcards and matching will be done using Go's [filepath.Match](#) rules. For example:

To add all files starting with "hom":

```
ADD hom* /mydir/
```

В приведенном ниже примере `?` заменяется любым отдельным символом, например, "home.txt".

```
ADD hom?.txt /mydir/
```

`<dest>` Это абсолютный путь или путь относительно `WORKDIR`, в который исходный файл будет скопирован внутри контейнера назначения.

В приведенном ниже примере используется относительный путь и добавляется "test.txt" к `<WORKDIR>/relativeDir/`:

```
ADD test.txt relativeDir/
```

Принимая во внимание, что в этом примере используется абсолютный путь и добавляется "test.txt" к `/absoluteDir/`

```
ADD test.txt /absoluteDir/
```

При добавлении файлов или каталогов, содержащих специальные символы (такие как `[` и `]`), вам необходимо экранировать эти пути, следуя правилам Go lang, чтобы предотвратить их обработку как совпадающего шаблона. Например, чтобы добавить файл с именем `arr[0].txt`, используйте следующее;

Дайте отзыв

```
ADD arr[[]0].txt /mydir/
```

Все новые файлы и каталоги создаются с UID и GID, равными 0, если только флаг необязательный `--chown` не указывает данное имя пользователя, имя группы или UID / GID комбинация для запроса конкретного владельца добавленного содержимого. Формат `--chown` флага допускает использование либо строк `username` и `groupname`, либо прямых целых UID и GID в любой комбинации. При указании имени пользователя без `groupname` или UID без GID будет использоваться тот же числовой UID, что и GID. Если указано имя пользователя или имя группы, корневая файловая система контейнера `/etc/passwd` и `/etc/group` файлы будут использоваться для выполнения перевода из `name` в `integer` UID или GID соответственно. В следующих примерах показаны допустимые определения для `--chown` флага:

```
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
ADD --chown=1 files* /somedir/
ADD --chown=10:11 files* /somedir/
ADD --chown=myuser:mygroup --chmod=655 files* /somedir/
```

Если корневая файловая система контейнера не содержит ни `/etc/passwd`, ни `/etc/group` файлов, ни имен пользователей или групп, которые используются в `--chown` флага, при `ADD` операции сборки произойдет сбой. Использование числовых идентификаторов не требует никакого поиска и не будет зависеть от содержимого корневой файловой системы контейнера.

В случае, когда `<src>` это URL-адрес удаленного файла, у адресата будут разрешения 600. Если извлекаемый удаленный файл имеет HTTP `Last-Modified` заголовок, временная метка из этого заголовка будет использоваться для установки `mtime` в целевом файле. Однако, как и любой другой файл, обработанный во время `ADD`, `mtime` не будет учитываться при определении того, изменился файл или нет, и кэш должен быть обновлен.

❗ Примечание

Если вы создаете, передавая `Dockerfile` через STDIN (`docker build - < somefile` контекст сборки отсутствует, поэтому `Dockerfile` может содержать только `ADD`

Дайте отзыв

инструкцию на основе URL. Вы также можете передать сжатый архив через STDIN: (`docker build - < archive.tar.gz`), `Dockerfile` находящийся в корне архива, а остальная часть архива будет использоваться в качестве контекста сборки.

Если ваши URL-файлы защищены с помощью аутентификации, вам необходимо использовать `RUN wget`, `RUN curl` или использовать другой инструмент из контейнера, поскольку `ADD` инструкция не поддерживает аутентификацию.

❗ Примечание

Первая встреченная `ADD` инструкция сделает недействительным кэш для всех последующих инструкций из `Dockerfile`, если содержимое `<src>` файла изменилось. Это включает в себя аннулирование кэша для `RUN` инструкций. Для получения дополнительной информации смотрите [Dockerfile Руководство по рекомендациям – Использовать кэш сборки](#).

Дайте отзыв

`ADD` подчиняется следующим правилам:

- `<src>` Путь должен находиться внутри *контекста* сборки; вы не можете `ADD ../something /something`, потому что первым шагом `docker build` является отправка контекстного каталога (и подкаталогов) демону `docker`.
- Если `<src>` это URL-адрес, который `<dest>` не заканчивается косой чертой, то файл загружается с URL-адреса и копируется в `<dest>`.
- Если `<src>` это URL-адрес, который `<dest>` заканчивается косой чертой в конце, то имя файла выводится из URL-адреса и файл загружается в `<dest>/<filename>`. Например, `ADD http://example.com/foobar /` создал бы файл `/foobar`. URL-адрес должен иметь нетривиальный путь, чтобы в этом случае можно было найти подходящее имя файла (`http://example.com` не сработает).
- Если `<src>` является каталогом, копируется все содержимое каталога, включая метаданные файловой системы.

❗ Примечание

Сам каталог не копируется, копируется только его содержимое.

- Если `<src>` это *локальный* архив tar в распознанном формате сжатия (identity, gzip, bzip2 или xz), то он распаковывается как каталог. Ресурсы с *удаленных* URL-адресов не распаковываются. Когда каталог копируется или распаковывается, он ведет себя так же, как `tar -x`, результатом является объединение:
 1. Что бы ни существовало в пути назначения и
 2. Содержимое дерева исходных текстов с разрешенными конфликтами в пользу "2". по каждому файлу.

❗ Примечание

Идентификация файла как распознанного формата сжатия или нет определяется исключительно на основе содержимого файла, а не имени файла. Например, если пустой файл заканчивается на `.tar.gz`, он не будет распознан как сжатый файл и не выдаст какое-либо сообщение об ошибке распаковки, скорее файл будет просто скопирован по назначению.

- Если `<src>` это файл любого другого типа, он копируется отдельно вместе со своими метаданными. В этом случае, если файл `<dest>` заканчивается косой чертой `/`, он будет считаться каталогом, и содержимое `<src>` будет записано в `<dest>/base(<src>)`.
- Если указано несколько `<src>` ресурсов, либо напрямую, либо благодаря использованию подстановочного знака, то `<dest>` должен быть каталог, и он должен заканчиваться косой чертой `/`.
- Если `<dest>` файл не заканчивается косой чертой, он будет считаться обычным файлом, и содержимое `<src>` будет записано в `<dest>`.
- Если `<dest>` файл не существует, он создается вместе со всеми отсутствующими каталогами в его пути.

Для проверки контрольной суммы удаленного файла **ДОБАВЬТЕ** -
`checksum=<контрольная сумма> <http src> <dest>`

Контрольную сумму удаленного файла можно проверить с помощью `--checksum` флага:

```
ADD --checksum=sha256:24454f830cdb571e2c4ad15481119c43b3cafd48dd869a9b2945d1036d1dc6
```

`--checksum` Флаг в настоящее время поддерживает только HTTP-источники.

Добавление репозитория git ДОБАВЬТЕ `<git ref> <dir>`

Эта форма позволяет добавлять репозиторий git к изображению напрямую, без использования `git` команды внутри изображения:

```
ADD [--keep-git-dir=<boolean>] <git ref> <dir>
```

```
# syntax=docker/dockerfile:1
FROM alpine
ADD --keep-git-dir=true https://github.com/moby/buildkit.git#v0.10.1 /buildkit
```

`--keep-git-dir=true` Флаг добавляет `.git` каталог. По умолчанию этот флаг равен `false`.

Добавление

Чтобы добавить частное хранилище через SSH, создайте Dockerfile со следующей формой:

```
# syntax=docker/dockerfile:1
FROM alpine
ADD git@git.example.com:foo/bar.git /bar
```

Этот файл Dockerfile может быть создан с помощью `docker build --ssh` или `buildctl build --ssh`, например,

```
$ docker build --ssh default
```

```
buildctl build --frontend=dockerfile.v0 --local context=. --local dockerfile=. --s
```

ПОБЛАГДАРИТЕСЬ

Смотрите `COPY --link`.

СКОПИРУЙТЕ

КОПИРОВАНИЕ имеет две формы:

```
COPY [--chown=<user>:<group>] [--chmod=<perms>] <src>... <dest>
COPY [--chown=<user>:<group>] [--chmod=<perms>] ["<src>", ... "<dest>"]
```

Дайте отзыв


Эта последняя форма требуется для путей, содержащих пробелы

❗ Примечание

Функции `--chown` и `--chmod` поддерживаются только в файлах Dockerfile, используемых для сборки контейнеров Linux, и не будут работать в контейнерах Windows. Поскольку концепции владения пользователями и группами не переводятся между Linux и Windows, использование `/etc/passwd` и `/etc/group` для перевода имен пользователей и групп в идентификаторы ограничивает эту функцию жизнеспособностью только для контейнеров на базе ОС Linux.

`COPY` Инструкция копирует новые файлы или каталоги из `<src>` и добавляет их в файловую систему контейнера по пути `<dest>`.

Может быть указано несколько `<src>` ресурсов, но пути к файлам и каталогам будут интерпретироваться как относящиеся к источнику контекста сборки.

Каждый из них `<src>` может содержать подстановочные знаки, и сопоставление будет производиться с использованием [пути к файлу Go. Соответствует правилам](#) . Например:

Чтобы добавить все файлы, начинающиеся с "hom":

```
/ hom* /mydir/
```

В приведенном ниже примере `?` заменяется любым отдельным символом, например, `"home.txt"`.

```
COPY hom?.txt /mydir/
```

`<dest>` Это абсолютный путь или путь относительно `WORKDIR`, в который исходный файл будет скопирован внутри контейнера назначения.

В приведенном ниже примере используется относительный путь и добавляется `"test.txt"` к `<WORKDIR>/relativeDir/`:

```
COPY test.txt relativeDir/
```

Принимая во внимание, что в этом примере используется абсолютный путь и добавляется `"test.txt"` к `/absoluteDir/`

```
COPY test.txt /absoluteDir/
```

При копировании файлов или каталогов, содержащих специальные символы (такие как `[` и `]`), вам необходимо избегать этих путей, следуя правилам Go lang, чтобы предотвратить их обработку как совпадающего шаблона. Например, чтобы скопировать файл с именем `arr[0].txt`, используйте следующее;

```
COPY arr[[]0].txt /mydir/
```

Все новые файлы и каталоги создаются с UID и GID, равными 0, если только флаг необязательный `--chown` не указывает данное имя пользователя, имя группы или UID / GID комбинация для запроса конкретного владельца скопированного содержимого. Формат `--chown` флага допускает использование либо строк `username` и `groupname`, либо прямых целых UID и GID в любой комбинации. При указании имени пользователя без `groupname` или UID без GID будет использоваться тот же числовой UID, что и GID. Если указано имя пользователя или имя группы, корневая файловая система контейнера `/etc/passwd` и `/etc/group` файлы будут использоваться для выполнения перевода из `name` в `integer` UID и GID соответственно. В следующих примерах показаны допустимые определения для `--chown` флага:

```
COPY --chown=55:mygroup files* /somedir/  
COPY --chown=bin files* /somedir/  
COPY --chown=1 files* /somedir/  
COPY --chown=10:11 files* /somedir/  
COPY --chown=myuser:mygroup --chmod=644 files* /somedir/
```

Если корневая файловая система контейнера не содержит ни `/etc/passwd`, ни `/etc/group` файлов, ни имен пользователей или групп, которые используются в `--chown` флаге, при `COPY` операции сборки произойдет сбой. Использование числовых идентификаторов не требует никакого поиска и не зависит от содержимого корневой файловой системы контейнера.

Дайте отзыв

❗ Примечание

При сборке с использованием STDIN (`docker build - < somefile`) отсутствует контекст сборки, поэтому `COPY` его нельзя использовать.

Необязательно `COPY` принимает флаг `--from=<name>`, который можно использовать для установки исходного местоположения на предыдущий этап сборки (созданный с помощью `FROM .. AS <name>`), который будет использоваться вместо контекста сборки, отправленного пользователем. В случае, если этап сборки с указанным именем не может быть найден, вместо него пытаются использовать изображение с таким же именем.

`COPY` подчиняется следующим правилам:

- `<src>` Путь должен находиться внутри *контекста* сборки; вы не можете `COPY ../something /something`, потому что первым шагом `docker build` является отправка контекстного каталога (и подкаталогов) демону `docker`.
- Если `<src>` является каталогом, копируется все содержимое каталога, включая метаданные файловой системы.

❗ Примечание

Если каталог не копируется, копируется только его содержимое.

- Если `<src>` это файл любого другого типа, он копируется отдельно вместе со своими метаданными. В этом случае, если файл `<dest>` заканчивается косой чертой `/`, он будет считаться каталогом, и содержимое `<src>` будет записано в `<dest>/base(<src>)`.
- Если указано несколько `<src>` ресурсов, либо напрямую, либо благодаря использованию подстановочного знака, то `<dest>` должен быть каталог, и он должен заканчиваться косой чертой `/`.
- Если `<dest>` файл не заканчивается косой чертой, он будет считаться обычным файлом, и содержимое `<src>` будет записано в `<dest>`.
- Если `<dest>` файл не существует, он создается вместе со всеми отсутствующими каталогами в его пути.

Дайте отзыв

❗ Примечание

Первая встреченная `COPY` инструкция сделает недействительным кэш для всех последующих инструкций из Dockerfile, если содержимое `<src>` файла изменилось. Это включает в себя аннулирование кэша для `RUN` инструкций. Для получения дополнительной информации смотрите [Dockerfile Руководство по рекомендациям – Использовать кэш сборки](#) [↗](#).

КОПИРОВАТЬ `--link`

❗ Примечание

Добавлено в `docker/dockerfile:1.4`

Включение этого флага в командах `COPY` или `ADD` позволяет копировать файлы с улучшенной семантикой, при которой ваши файлы остаются независимыми на своем собственном слое и не становятся недействительными при изменении команд на предыдущих слоях.

При использовании `--link` ваши исходные файлы копируются в пустой каталог назначения. Этот каталог превращается в слой, который связан поверх вашего предыдущего состояния.

```
# syntax=docker/dockerfile:1
FROM alpine
COPY --link /foo /bar
```

Эквивалентно выполнению двух сборок:

```
FROM alpine
```

и

```
FROM scratch
COPY /foo /bar
```

Дайте отзыв

и объединение всех слоев обоих изображений вместе.

Преимущества использования `--link`

Используется `--link` для повторного использования уже созданных слоев в последующих сборках с `--cache-from`, даже если предыдущие слои изменились. Это особенно важно для многоступенчатых сборок, где `COPY --from` инструкция ранее становилась недействительной, если какие-либо предыдущие команды на том же этапе изменялись, что вызывало необходимость повторного построения промежуточных этапов. С `--link` слоем, сгенерированным предыдущей сборкой, повторно используется и объединяется поверх новых слоев. Это также означает, что вы можете легко перебазировать свои изображения, когда базовые изображения получают обновления, без необходимости выполнять всю сборку заново. В поддерживаемых бэкэндах BuildKit может выполнять это действие перебазирования без необходимости нажимать или извлекать какие-либо уровни между клиентом и реестром. BuildKit обнаружит этот случай и создаст только новый манифест изображения, который содержит новые слои и старые слои в правильном порядке.

То же поведение, при котором BuildKit может избежать удаления базового образа, также может происходить при использовании `--link` и никаких других команд, которые могли бы доступа к файлам базового образа. В этом случае BuildKit будет создавать

слои только для `COPY` команд и помещать их в реестр непосредственно поверх слоев базового изображения.

Несовместимости с `--link=false`

При использовании `--link` этих `COPY/ADD` команд не разрешается считывать какие-либо файлы из предыдущего состояния. Это означает, что если в предыдущем состоянии каталог назначения был путем, который содержал символическую ссылку, `COPY/ADD` по нему нельзя следовать. В конечном изображении путь назначения, созданный с помощью `--link`, всегда будет путем, содержащим только каталоги.

Если вы не полагаетесь на поведение следующих символических ссылок в пути назначения, всегда рекомендуется использовать `--link`. Производительность `--link` эквивалентна или лучше, чем поведение по умолчанию, и это создает гораздо лучшие условия для повторного использования кэша.

Дайте отзыв

Ссылка

ТОЧКА ВХОДА имеет две формы:

Форма *ехес*, которая является предпочтительной формой:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

Форма *оболочки*:

```
ENTRYPOINT command param1 param2
```

`ENTRYPOINT` позволяет настроить контейнер, который будет запускаться как исполняемый файл.

Например, следующий запускает `nginx` с его содержимым по умолчанию, прослушивая порт 80:

```
$ docker run -i -t --rm -p 80:80 nginx
```


Аргументы командной строки для `docker run <image>` будут добавлены после всех элементов в *форме exes* `ENTRYPOINT` и переопределяют все элементы, указанные с помощью `CMD`. Это позволяет передавать аргументы в точку входа, т.е. `docker run <image> -d` передаст `-d` аргумент в точку входа. Вы можете переопределить `ENTRYPOINT` инструкцию, используя `docker run --entrypoint` флаг.

Форма *оболочки* предотвращает использование каких-либо аргументов `CMD` или `run` командной строки, но имеет тот недостаток, что ваша `ENTRYPOINT` команда будет запущена как подкоманда `/bin/sh -c`, которая не передает сигналы. Это означает, что исполняемый файл не будет принадлежать контейнеру `PID 1` и не будет получать сигналы Unix, поэтому ваш исполняемый файл не получит `SIGTERM` from `docker stop <container>`.

Только последняя `ENTRYPOINT` инструкция в `Dockerfile` будет иметь эффект.

Дайте отзыв

Пример точки ВХОДА в форму Exes

Вы можете использовать форму *exes* `ENTRYPOINT` для установки довольно стабильных команд и аргументов по умолчанию, а затем использовать любую форму `CMD` для установки дополнительных значений по умолчанию, которые с большей вероятностью будут изменены.

```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

Когда вы запускаете контейнер, вы можете видеть, что `top` это единственный процесс:

```
$ docker run -it --rm --name test top -H

top - 08:25:00 up 7:27, 0 users, load average: 0.00, 0.01, 0.05
Threads: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.1 us, 0.1 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 2056668 total, 1616832 used, 439836 free, 99352 buffers
KiB Swap: 1441840 total, 0 used, 1441840 free. 1324440 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR S %CPU %MEM    TIME+  COMMAND
    1 root        20   0   19744   2336   2080 R  0.0  0.1    0:00.04 top
```

Для дальнейшего изучения результата вы можете использовать `docker exec`:

```
$ docker exec -it test ps aux
```

| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | START | TIME | COMMAND |
|------|-----|------|------|-------|------|-----|------|-------|------|-----------|
| root | 1 | 2.6 | 0.1 | 19752 | 2352 | ? | Ss+ | 08:24 | 0:00 | top -b -H |
| root | 7 | 0.0 | 0.1 | 15572 | 2164 | ? | R+ | 08:25 | 0:00 | ps aux |

И вы можете изящно запросить `top` завершение работы с помощью `docker stop test`.

Ниже `Dockerfile` показано использование `ENTRYPOINT` для запуска Apache на переднем плане (т. Е. как `PID 1`):

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Дайте отзыв

Если вам нужно написать начальный скрипт для одного исполняемого файла, вы можете убедиться, что конечный исполняемый файл получает сигналы Unix с помощью `exec` и `gosu` команд:

```
#!/usr/bin/env bash
set -e

if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
fi

exec "$@"
```

еще, если вам нужно выполнить дополнительную очистку (или связаться с другими контейнерами) при завершении работы или при координации нескольких исполняемых

файлов, вам может потребоваться убедиться, что что `ENTRYPOINT` скрипт получает сигналы Unix, передает их дальше, а затем выполняет еще некоторую работу:

```
#!/bin/sh
# Note: I've written this using sh so it works in the busybox container too

# USE the trap if you need to also do manual cleanup after the service is stopped,
# or need to start multiple services in the one container
trap "echo TRAPed signal" HUP INT QUIT TERM

# start service in background here
/usr/sbin/apachectl start

echo "[hit enter key to exit] or run 'docker stop <container>'"
read

# stop service and clean up here
echo "stopping apache"
/usr/sbin/apachectl stop

echo "exited $0"
```

Дайте отзыв

Если вы запустите этот образ с помощью

`docker run -it --rm -p 80:80 --name test apache`, затем вы можете изучить процессы контейнера с помощью `docker exec` или `docker top`, а затем попросить скрипт остановить Apache:

```
$ docker exec -it test ps aux
```

| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | START | TIME | COMMAND |
|----------|-----|------|------|--------|------|-----|------|-------|------|---------------------|
| root | 1 | 0.1 | 0.0 | 4448 | 692 | ? | Ss+ | 00:42 | 0:00 | /bin/sh /run.sh 123 |
| root | 19 | 0.0 | 0.2 | 71304 | 4440 | ? | Ss | 00:42 | 0:00 | /usr/sbin/apache2 - |
| www-data | 20 | 0.2 | 0.2 | 360468 | 6004 | ? | Sl | 00:42 | 0:00 | /usr/sbin/apache2 - |
| www-data | 21 | 0.2 | 0.2 | 360468 | 6000 | ? | Sl | 00:42 | 0:00 | /usr/sbin/apache2 - |
| root | 81 | 0.0 | 0.1 | 15572 | 2140 | ? | R+ | 00:44 | 0:00 | ps aux |

```
$ docker top test
```

| PID | USER | COMMAND |
|-----|------|---------------------------------------|
| 35 | root | {run.sh} /bin/sh /run.sh 123 cmd cmd2 |

```
10054          root          /usr/sbin/apache2 -k start
10055          33             /usr/sbin/apache2 -k start
10056          33             /usr/sbin/apache2 -k start
```

```
$ /usr/bin/time docker stop test
```

```
test
real    0m 0.27s
user    0m 0.03s
sys 0m 0.03s
```

Note

You can override the `ENTRYPOINT` setting using `--entrypoint`, but this can only set the binary to `exec` (no `sh -c` will be used).

Дайте отзыв

Note

The `exec` form is parsed as a JSON array, which means that you must use double-quotes (") around words not single-quotes (').

Unlike the `shell` form, the `exec` form does not invoke a command shell. This means that normal shell processing does not happen. For example, `ENTRYPOINT ["echo", "$HOME"]` will not do variable substitution on `$HOME`. If you want shell processing then either use the `shell` form or execute a shell directly, for example:

`ENTRYPOINT ["sh", "-c", "echo $HOME"]`. When using the `exec` form and executing a shell directly, as in the case for the `shell` form, it is the shell that is doing the environment variable expansion, not docker.

Shell form ENTRYPOINT example

Вы можете указать простую строку для `ENTRYPOINT`, и она будет выполняться в `/bin/sh -c`. Эта форма будет использовать обработку командной строки для замены переменных среды командной строки и будет игнорировать любые `CMD` или `docker run` аргументы командной строки. Чтобы гарантировать, что `docker stop` будет правильно

сигнализировать о любом долго выполняющемся `ENTRYPOINT` исполняемом файле, вам нужно не забыть запустить его с `exec`:

```
FROM ubuntu
ENTRYPOINT exec top -b
```

При запуске этого изображения вы увидите единый `PID 1` процесс:

```
$ docker run -it --rm --name test top

Mem: 1704520K used, 352148K free, 0K shrd, 0K buff, 140368121167873K cached
CPU:  5% usr  0% sys  0% nic 94% idle  0% io  0% irq  0% irq
Load average: 0.08 0.03 0.05 2/98 6
  PID  PPID  USER      STAT  VSZ %VSZ %CPU COMMAND
   1     0  root       R    3164  0%  0% top -b
```

Дайте отзыв

Который завершается чисто на `docker stop`:

```
$ /usr/bin/time docker stop test

test
real    0m 0.20s
user    0m 0.02s
sys 0m 0.04s
```

Если вы забыли добавить `exec` в начало вашего `ENTRYPOINT`:

```
FROM ubuntu
ENTRYPOINT top -b
CMD -- --ignored-param1
```

Затем вы можете запустить его (присвоив ему имя для следующего шага):

```
$ docker run -it --name test top --ignored-param2

- 13:58:24 up 17 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:  2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
```

```
%Cpu(s): 16.7 us, 33.3 sy,  0.0 ni, 50.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 1990.8 total, 1354.6 free, 231.4 used, 404.7 buff/cache
MiB Swap: 1024.0 total, 1024.0 free,  0.0 used. 1639.8 avail Mem
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-----|------|----|----|------|------|------|---|------|------|---------|---------|
| 1 | root | 20 | 0 | 2612 | 604 | 536 | S | 0.0 | 0.0 | 0:00.02 | sh |
| 6 | root | 20 | 0 | 5956 | 3188 | 2768 | R | 0.0 | 0.2 | 0:00.00 | top |

Из выходных данных вы можете видеть, `top` что указанное `ENTRYPOINT` не `PID 1`.

Если вы затем запустите `docker stop test`, контейнер не завершится чисто - `stop` команда будет вынуждена отправить `SIGKILL` после истечения времени ожидания:

```
$ docker exec -it test ps waux
```

| USER | PID | %CPU | %MEM | VSZ | RSS | TTY | STAT | START | TIME | COMMAND |
|------|-----|------|------|------|------|-------|------|-------|------|---------------------|
| root | 1 | 0.4 | 0.0 | 2612 | 604 | pts/0 | Ss+ | 13:58 | 0:00 | /bin/sh -c top -b - |
| root | 6 | 0.0 | 0.1 | 5956 | 3188 | pts/0 | S+ | 13:58 | 0:00 | top -b |
| root | 7 | 0.0 | 0.1 | 5884 | 2816 | pts/1 | Rs+ | 13:58 | 0:00 | ps waux |

```
$ /usr/bin/time docker stop test
```

```
test
real    0m 10.19s
user    0m 0.04s
sys 0m 0.03s
```

Дайте отзыв

Понять, как взаимодействуют CMD и ENTRYPOINT

Обе `CMD` и `ENTRYPOINT` инструкции определяют, какая команда выполняется при запуске контейнера. Существует несколько правил, описывающих их взаимодействие.

1. В Dockerfile должна быть указана хотя бы одна из команд `CMD` или `ENTRYPOINT`.
2. `ENTRYPOINT` должно быть определено при использовании контейнера в качестве исполняемого файла.
3. `CMD` следует использовать как способ определения аргументов по умолчанию для `ENTRYPOINT` команды или для выполнения специальной команды в контейнере.

4. `CMD` будет переопределен при запуске контейнера с альтернативными аргументами.

В таблице ниже показано, какая команда выполняется для разных `ENTRYPOINT` / `CMD` комбинаций:

| | Нет точки входа | ТОЧКА ВХОДА exec_entry p1_entry | ТОЧКА ВХОДА ["exec_entry", "p1_entry"] |
|-------------------------------|-------------------------------|------------------------------------|---|
| Нет CMD | <i>ошибка, не разрешена</i> | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry |
| CMD ["exec_cmd", "p1_cmd"] | exec_cmd p1_cmd | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry exec_cmd p1_cmd |
| CMD exec_cmd p1_cmd | /bin/sh -c exec_cmd p1_cmd | /bin/sh -c exec_entry p1_entry | exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd |

Дайте отзыв

Примечание

Если `CMD` он определен на основе базового образа, значение `ENTRYPOINT` будет сброшено `CMD` до пустого значения. В этом сценарии `CMD` должен быть определен в текущем изображении, чтобы иметь значение.

ССЫЛКА

`VOLUME ["/data"]`

`VOLUME` Инструкция создает точку монтирования с указанным именем и помечает ее как содержащую подключенные извне тома с собственного хоста или других контейнеров. Значением может быть массив JSON, `VOLUME ["/var/log/"]` или обычная строка с несколькими аргументами, такими как `VOLUME /var/log` или `VOLUME /var/log /var/db`. Дополнительную информацию / примеры и инструкции по монтажу можно получить на Клиент Docker, см. [Общий доступ к каталогам через тома](#) документация.

`docker run` Команда инициализирует вновь созданный том любыми данными, которые твуют в указанном расположении базового образа. Например, рассмотрим следующий

фрагмент файла Dockerfile:

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME /myvol
```

Результатом работы с этим файлом Dockerfile является образ, который приводит `docker run` к созданию новой точки монтирования в `/myvol` и копированию `greeting` файла во вновь созданный том.

Примечания по указанию томов

Имейте в виду следующие моменты, касающиеся томов в `Dockerfile`.

Дайте отзыв

- Тома в контейнерах на базе Windows: При использовании контейнеров на базе Windows местом назначения тома внутри контейнера должен быть один из:
 - несуществующий или пустой каталог
 - диск, отличный от `C:`
- Изменение тома внутри Dockerfile: Если какие-либо шаги сборки изменяют данные в томе после его объявления, эти изменения будут отброшены.
- Форматирование в формате JSON: список анализируется как массив JSON. Вы должны заключать слова в двойные кавычки (`"`), а не в одинарные кавычки (`'`).
- Каталог узла объявляется во время выполнения контейнера: каталог узла (точка монтирования) по своей природе зависит от узла. Это делается для сохранения переносимости образа, поскольку нельзя гарантировать доступность данного каталога хоста на всех хостах. По этой причине вы не можете смонтировать каталог хоста из файла Dockerfile. `VOLUME` Инструкция не поддерживает указание `host-dir` параметра. Вы должны указать точку монтирования при создании или запуске контейнера.

ПОЛЬЗОВАТЕЛЬСКАЯ

```
USER <user>[:<group>]
```


или

```
USER <UID>[:<GID>]
```

`USER` Инструкция задает имя пользователя (или UID) и, необязательно, пользователя group (или GID) для использования в качестве пользователя и группы по умолчанию на оставшуюся часть текущего этапа. Указанный пользователь используется для `RUN` инструкций и во время выполнения выполняет соответствующие `ENTRYPOINT` и `CMD` команды.

- ❗ Обратите внимание, что при *только* указанное членство в группе. Любое указании группы для пользователя *другое* настроенное членство в группе у пользователя будет *будет* проигнорировано.

❗ Предупреждение

Если у пользователя нет основной группы, то изображение (или следующие инструкции) будет запущено с `root` группой.

В Windows сначала необходимо создать пользователя, если это не встроенная учетная запись. Это можно сделать с помощью `net user` команды, вызываемой как часть Dockerfile.

```
FROM microsoft/windowsservercore
# Create Windows user in the container
RUN net user /add patrick
# Set it for subsequent commands
USER patrick
```

Ссылка

```
CDIR /path/to/workdir
```

Дайте отзыв

`WORKDIR` Инструкция устанавливает рабочий каталог для любого `RUN`, `CMD`, `ENTRYPOINT`, `COPY` и `ADD` инструкции, которые следуют за ним в `Dockerfile`. Если файл `WORKDIR` не существует, он будет создан, даже если он не используется ни в одной из последующих `Dockerfile` инструкций.

`WORKDIR` Инструкция может использоваться несколько раз в `Dockerfile`. Если указан относительный путь, он будет относиться к пути предыдущей `WORKDIR` инструкции.

Например:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

Вывод последней `pwd` команды в этом `Dockerfile` будет `/a/b/c`.

`WORKDIR` Инструкция может разрешать переменные среды, ранее установленные с помощью `ENV`. Вы можете использовать только переменные среды, явно заданные в `Dockerfile`.

Например:

```
ENV DIRPATH=/path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

Вывод последней `pwd` команды в этом `Dockerfile` будет `/path/$DIRNAME`

Если не указано, рабочим каталогом по умолчанию является `/`. На практике, если вы не создаете файл `Dockerfile` с нуля (`FROM scratch`), `WORKDIR` скорее всего, он может быть задан используемым вами базовым образом.

Поэтому, чтобы избежать непреднамеренных операций в неизвестных каталогах, лучше всего указывать ваш `WORKDIR` явно.

ARG

```
<name>[=<default value>]
```

ARG Инструкция определяет переменную, которую пользователи могут передавать во время сборки разработчику с помощью `docker build` команды, использующей `--build-arg <varname>=<value>` флаг. Если пользователь указывает аргумент сборки, который не был определен в Dockerfile, сборка выводит предупреждение.

```
[Warning] One or more build-args [foo] were not consumed.
```

Файл Dockerfile может содержать одну или несколько **ARG** инструкций. Например, следующий допустимый файл Dockerfile:

```
FROM busybox
ARG user1
ARG buildno
# ...
```

Дайте отзыв

ⓘ Предупреждение:

Не рекомендуется использовать переменные времени сборки для передачи секретных данных, таких как ключи GitHub, учетные данные пользователя и т.д. Значения переменных времени сборки видны любому пользователю изображения с помощью `docker history` команды.

Обратитесь к `RUN --mount=type=secret` разделу, чтобы узнать о безопасных способах использования секретов при создании образов. `{:.warning}`

Ссылка

ARG Инструкция может необязательно включать значение по умолчанию:

```
FROM busybox
ARG user1=someuser
ARG buildno=1
# ...
```

ARG инструкция имеет значение по умолчанию и если значение не передано во время сборки конструктор использует значение по умолчанию.

Ссылка

ARG Определение переменной вступает в силу из строки, в которой она определена в **Dockerfile**, а не из использования аргумента в командной строке или где-либо еще. Например, рассмотрим этот **Dockerfile**:

```
FROM busybox
USER ${username:-some_user}
ARG username
USER $username
# ...
```

Пользователь создает этот файл, вызывая:

```
$ docker build --build-arg username=what_user .
```

USER В строке 2 вычисляется значение, равное **some_user** поскольку **username** переменная определена в последующей строке 3. **USER** В строке 4 вычисляется значение **what_user**, поскольку **username** аргумент определен, и **what_user** значение было передано в командной строке. До ее определения с помощью инструкции **ARG** любое использование переменной приводит к пустой строке.

ARG Инструкция выходит за рамки области видимости в конце этапа сборки, на котором она была определена. Чтобы использовать аргумент на нескольких этапах, каждый этап должен включать **ARG** инструкцию.

```
FROM busybox
ARG SETTINGS
RUN ./run/setup $SETTINGS

FROM busybox
ARG SETTINGS
RUN ./run/other $SETTINGS
```

Дайте отзыв

Использование

Вы можете использовать `ARG` или `ENV` инструкцию для указания переменных, которые доступны для `RUN` инструкции. Переменные среды, определенные с помощью `ENV` инструкции, всегда переопределяют `ARG` инструкцию с тем же именем. Рассмотрим этот файл Dockerfile с помощью `ENV` и `ARG` инструкции.

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=v1.0.0
RUN echo $CONT_IMG_VER
```

Затем предположим, что этот образ создан с помощью этой команды:

```
$ docker build --build-arg CONT_IMG_VER=v2.0.1 .
```

Дайте отзыв

В этом случае `RUN` инструкция использует `v1.0.0` вместо `ARG` параметра переданный пользователем: `v2.0.1`. Это поведение похоже на командную оболочку сценарий, в котором локальная переменная переопределяет переменные, переданные как аргументы или унаследованы из среды, с точки зрения ее определения.

Используя приведенный выше пример, но другую `ENV` спецификацию, вы можете создавать более полезные взаимодействия между `ARG` и `ENV` инструкциями:

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=${CONT_IMG_VER:-v1.0.0}
RUN echo $CONT_IMG_VER
```

В отличие от `ARG` инструкции, `ENV` значения всегда сохраняются во встроенном изображении. Рассмотрим сборку docker без `--build-arg` флага:

```
$ docker build .
```

Используя этот пример Dockerfile, `CONT_IMG_VER` файл по-прежнему сохраняется в изображении, но его значение будет `v1.0.0` таким, как оно задано по умолчанию в строке 3 `ENV` инструкцией.

Метод расширения переменной в этом примере позволяет передавать аргументы из командной строки и сохранять их в конечном изображении, используя `ENV` инструкцию. Расширение переменной поддерживается только для [ограниченного набора инструкций Dockerfile](#).

Ссылка

Docker имеет набор predefined `ARG` переменных, которые вы можете использовать без соответствующей `ARG` инструкции в Dockerfile.

- `HTTP_PROXY`
- `http_proxy`
- `HTTPS_PROXY`
- `https_proxy`
- `FTP_PROXY`
- `ftp_proxy`
- `NO_PROXY`
- `no_proxy`
- `ALL_PROXY`
- `all_proxy`

Дайте отзыв

Чтобы использовать их, передайте их в командной строке, используя `--build-arg` флаг, например:

```
$ docker build --build-arg HTTPS_PROXY=https://my-proxy.example.com .
```

По умолчанию эти predefined переменные исключены из выходных данных `docker history`. Их исключение снижает риск случайной утечки конфиденциальной информации аутентификации в `HTTP_PROXY` переменной.

Например, рассмотрите возможность создания следующего файла Dockerfile с использованием `--build-arg HTTP_PROXY=http://user:pass@proxy.lon.example.com`

```
FROM ubuntu
echo "Hello World"
```

В этом случае значение `HTTP_PROXY` переменной недоступно в `docker history` и не кэшируется. Если вы должны были изменить местоположение, и ваш прокси-сервер изменился на `http://user:pass@proxy.sfo.example.com`, последующая сборка не приводит к отсутствию кэша.

Если вам нужно переопределить это поведение, вы можете сделать это, добавив `ARG` инструкцию в Dockerfile следующим образом:

```
FROM ubuntu
ARG HTTP_PROXY
RUN echo "Hello World"
```

При сборке этого файла Dockerfile `HTTP_PROXY` сохраняется в `docker history`, и изменение его значения делает недействительным кэш сборки.

Дайте отзыв

Автоматические аргументы платформы в глобальной области

Эта функция доступна только при использовании бэкэнда [BuildKit](#).

Docker определяет набор `ARG` переменных с информацией о платформе узла, выполняющего сборку (build platform), и о платформе результирующего изображения (target platform). Целевая платформа может быть указана с включенным `--platform` флагом `docker build`.

Следующие `ARG` переменные устанавливаются автоматически:

- `TARGETPLATFORM` - платформа результата сборки. Например, `linux/amd64`, `linux/arm/v7` `windows/amd64`.
- `TARGETOS` - Компонент операционной системы `TARGETPLATFORM`
- `TARGETARCH` - архитектурный компонент `TARGETPLATFORM`
- `TARGETVARIANT` - вариативный компонент `TARGETPLATFORM`
- `BUILDPLATFORM` - платформа узла, выполняющего сборку.
- `BUILDOS` - Компонент операционной системы `BUILDPLATFORM`
- `BUILDARCH` - архитектурный компонент `BUILDPLATFORM`
- `BUILDVARIANT` - вариативный компонент `BUILDPLATFORM`

Эти аргументы определены в глобальной области видимости, поэтому не доступны автоматически на этапах сборки или для ваших `RUN` команд. Чтобы отобразить один из этих аргументов на этапе сборки, переопределите его без значения.

Например:

```
FROM alpine
ARG TARGETPLATFORM
RUN echo "I'm building for $TARGETPLATFORM"
```

| Аргумент | Тип | Описание |
|---|--------|---|
| <code>BUILDKIT_CACHE_MOUNT_NS</code> | Строка | Задайте необязательное пространство имен cache ID. |
| <code>BUILDKIT_CONTEXT_KEEP_GIT_DIR</code> | Bool | Запустите контекст git, чтобы сохранить <code>.git</code> каталог. |
| <code>BUILDKIT_INLINE_CACHE</code> ² | Bool | Встроить метаданные кэша в конфигурацию изображения или нет. |
| <code>BUILDKIT_MULTI_PLATFORM</code> | Bool | Выберите детерминированный вывод, независимо от того, мультиплатформенный вывод или нет. |
| <code>BUILDKIT_SANDBOX_HOSTNAME</code> | Строка | Задайте имя хоста (по умолчанию <code>buildkitsandbox</code>) |
| <code>BUILDKIT_SYNTAX</code> | Строка | Установить внешний образ |
| <code>SOURCE_DATE_EPOCH</code> | Int | Установите временную метку UNIX для создаваемого изображения и слоев. Дополнительная информация из воспроизводимых сборок [↗] . Поддерживается начиная с Dockerfile 1.5, BuildKit 0.11 |

Пример: сохранить

При использовании контекста Git `.git` каталог не сохраняется при извлечении git. Может быть полезно сохранить его, если вы хотите получить информацию git во время сборки:

```
# syntax=docker/dockerfile:1
1 alpine
```



```
WORKDIR /src
RUN --mount=target=. \
    make REVISION=$(git rev-parse HEAD) build
```

```
$ docker build --build-arg BUILDKIT_CONTEXT_KEEP_GIT_DIR=1 https://github.com/user/r
```

Влияние на кеширование сборки

`ARG` переменные не сохраняются в построенном образе, как `ENV` переменные. Однако `ARG` переменные влияют на кэш сборки аналогичным образом. Если Dockerfile определяет `ARG` переменную, значение которой отличается от предыдущей сборки, то при ее первом использовании происходит "промах кэша", а не ее определение. В частности, все `RUN` инструкции, следующие за `ARG` инструкцией, используют `ARG` переменную неявно (как переменную окружения), что может привести к отсутствию кэша. Все предопределенные `ARG` переменные освобождаются от кеширования, если в `ARG` нет соответствующего Dockerfile оператора.

Дайте отзыв

Для примера рассмотрим эти два файла Dockerfile:

```
FROM ubuntu
ARG CONT_IMG_VER
RUN echo $CONT_IMG_VER
```

```
FROM ubuntu
ARG CONT_IMG_VER
RUN echo hello
```

Если вы укажете `--build-arg CONT_IMG_VER=<value>` в командной строке, в обоих случаях спецификация в строке 2 не приводит к отсутствию кэша; строка 3 делает приводит к отсутствию кэша. `ARG CONT_IMG_VER` вызывает идентификацию строки RUN как такой же, как running `CONT_IMG_VER=<value> echo hello`, поэтому, если `<value>` изменится, мы получим ошибку в кэше.

Рассмотрим другой пример в той же командной строке:

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=$CONT_IMG_VER
RUN echo $CONT_IMG_VER
```

В этом примере ошибка в кэше возникает в строке 3. Ошибка происходит потому, что значение переменной в `ENV` ссылается на `ARG` переменную и эта переменная изменяется через командную строку. В этом примере `ENV` команда приводит к тому, что изображение включает значение.

Если `ENV` инструкция переопределяет `ARG` инструкцию с тем же именем, например, этот Dockerfile:

```
FROM ubuntu
ARG CONT_IMG_VER
ENV CONT_IMG_VER=hello
RUN echo $CONT_IMG_VER
```

Строка 3 не приводит к пропуску кэша, поскольку значение `CONT_IMG_VER` является константой (`hello`). В результате переменные среды и значения, используемые в `RUN` (строка 4), не меняются между сборками.

ONBUILD <INSTRUCTION>

ONBUILD Инструкция добавляет к изображению *триггерную* инструкцию, которая будет выполнена позже, когда изображение будет использовано в качестве основы для другой сборки. Триггер будет выполнен в контексте последующей сборки, как если бы он был вставлен сразу после `FROM` инструкции в последующей сборке Dockerfile.

Любая инструкция по сборке может быть зарегистрирована в качестве триггера.

Это полезно, если вы создаете образ, который будет использоваться в качестве основы создания других образов, например, среды сборки приложения или демона, который

Дайте отзыв

может быть настроен в соответствии с пользовательской конфигурацией.

Например, если ваш образ представляет собой многоразовый конструктор приложений на Python, для этого потребуется добавить исходный код приложения в определенный каталог, и *после* этого может потребоваться вызвать скрипт сборки. Вы не можете просто вызвать `ADD` и `RUN` `now`, потому что у вас еще нет доступа к исходному коду приложения, и он будет разным для каждой сборки приложения. Вы могли бы просто предоставить разработчикам приложений шаблон `Dockerfile` для копирования и вставки в их приложение, но это неэффективно, подвержено ошибкам и трудно обновляется, поскольку смешивается с кодом, специфичным для конкретного приложения.

Решение заключается в использовании `ONBUILD` для регистрации предварительных инструкций для запуска позже, на следующем этапе сборки.

Вот как это работает:

1. При обнаружении `ONBUILD` инструкции конструктор добавляет триггер к метаданным создаваемого изображения. В остальном инструкция не влияет на текущую сборку.
2. В конце сборки список всех триггеров сохраняется в манифесте изображения под ключом `OnBuild`. Их можно проверить с помощью `docker inspect` команды.
3. Позже изображение может быть использовано в качестве основы для новой сборки, используя `FROM` инструкцию. В рамках обработки `FROM` инструкции нисходящий конструктор ищет `ONBUILD` триггеры и выполняет их в том же порядке, в котором они были зарегистрированы. Если какой-либо из триггеров завершается ошибкой, `FROM` инструкция прерывается, что, в свою очередь, приводит к сбою сборки. Если все триггеры завершаются успешно, `FROM` инструкция завершается, и сборка продолжается в обычном режиме.
4. Триггеры удаляются из конечного образа после выполнения. Другими словами, они не наследуются сборками "внуков".

Например, вы могли бы добавить что-то вроде этого:

```
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
```

Дайте отзыв

Связывание `ONBUILD` инструкций с использованием `ONBUILD ONBUILD` не допускается.

ⓘ Предупреждение

`ONBUILD` Инструкция может не запускать `FROM` или `MAINTAINER` инструкции.

Ссылка

`STOPSIGNAL` *signal*

`STOPSIGNAL` Инструкция задает сигнал системного вызова, который будет отправлен в контейнер для завершения. Этот сигнал может быть именем сигнала в формате `SIG<NAME>`, например `SIGKILL`, или числом без знака, которое соответствует позиции в таблице системных вызовов ядра `9`. По умолчанию используется значение `SIGTERM`, если оно не определено.

Дайте отзыв

Сигнал остановки изображения по умолчанию может быть переопределен для каждого контейнера, используя `--stop-signal` флаг на `docker run` и `docker create`.

Ссылка

`HEALTHCHECK` Инструкция имеет две формы:

- `HEALTHCHECK [OPTIONS] CMD command` (проверьте работоспособность контейнера, выполнив команду внутри контейнера)
- `HEALTHCHECK NONE` (отключите любую проверку работоспособности, унаследованную от базового образа)

`HEALTHCHECK` Инструкция сообщает Docker, как протестировать контейнер, чтобы убедиться, что он все еще работает. Это может обнаружить такие случаи, как веб-сервер, который застрял в бесконечном цикле и не может обрабатывать новые подключения, даже если серверный процесс все еще запущен.

Когда в контейнере указана проверка работоспособности, у него есть *состояние работоспособности* в дополнение к его обычному статусу. Этот статус является

первоначальным `starting`. Всякий раз, когда проверка работоспособности проходит, он переходит в `healthy` (в каком бы состоянии он ни находился ранее). После определенного количества последовательных сбоев он становится `unhealthy`.

Параметры, которые могут появиться перед `CMD`, следующие:

- `--interval=DURATION` (по умолчанию: `30s`)
- `--timeout=DURATION` (по умолчанию: `30s`)
- `--start-period=DURATION` (по умолчанию: `0s`)
- `--start-interval=DURATION` (по умолчанию: `5s`)
- `--retries=N` (по умолчанию: `3`)

Проверка работоспособности сначала выполняется с интервалом в несколько секунд после запуска контейнера, а затем снова с интервалом в несколько секунд после завершения каждой предыдущей проверки.

Дайте отзыв

Если один запуск проверки занимает больше, чем тайм-аут секунд, то проверка считается неудачной.

Для рассмотрения контейнера требуются повторные попытки с последовательными сбоями проверки работоспособности `unhealthy`.

период запуска определяет время инициализации для контейнеров, которым требуется время для начальной загрузки. Сбой проверки в течение этого периода не будет засчитан в максимальное количество попыток. Однако, если проверка работоспособности завершается успешно в течение периода запуска, контейнер считается запущенным, и все последовательные сбои будут засчитаны в максимальное количество повторных попыток.

интервал запуска - это время между проверками работоспособности в течение начального периода.

В Dockerfile может быть только одна `HEALTHCHECK` инструкция. Если вы перечислите более одной, то вступит в силу только последняя `HEALTHCHECK`.

Команда после `CMD` ключевого слова может быть либо командой оболочки (например, `HEALTHCHECK CMD /bin/check-running`), либо массивом `exec` (как и в случае с другими командами Dockerfile; подробности см., например, `ENTRYPOINT`).

Статус завершения команды указывает на состояние работоспособности контейнера.

Возможные значения:

- 0: success - the container is healthy and ready for use
- 1: unhealthy - the container is not working correctly
- 2: reserved - do not use this exit code

For example, to check every five minutes or so that a web-server is able to serve the site's main page within three seconds:

```
HEALTHCHECK --interval=5m --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Дайте отзыв

Чтобы помочь отлаживать неисправные зонды, любой выходной текст (в кодировке UTF-8), который записывает команда в stdout или stderr, будет сохранен в состоянии работоспособности и может быть запрошен с помощью `docker inspect`. Такой вывод должен быть коротким (в настоящее время хранятся только первые 4096 байт).

Когда состояние работоспособности контейнера изменяется, `health_status` генерируется событие с новым статусом.

ССЫЛКА на

```
SHELL ["executable", "parameters"]
```

`SHELL` Инструкция позволяет переопределять оболочку по умолчанию, используемую для формы *оболочки* команд. Оболочкой по умолчанию в Linux является `["/bin/sh", "-c"]`, и на Windows - это `["cmd", "/S", "/C"]`. `SHELL` Инструкция *должна* быть написана в формате JSON форма в Dockerfile.

`SHELL` Инструкция особенно полезна в Windows, где есть две широко используемые и совершенно разные собственные оболочки: `cmd` и `powershell`, а также доступные альтернативные оболочки, включая `sh`.

`SHELL` Инструкция может отображаться несколько раз. Каждая `SHELL` инструкция определяет все предыдущие `SHELL` инструкции и влияет на все последующие

инструкции. Например:

```
FROM microsoft/windowsservercore

# Executed as cmd /S /C echo default
RUN echo default

# Executed as cmd /S /C powershell -command Write-Host default
RUN powershell -command Write-Host default

# Executed as powershell -command Write-Host hello
SHELL ["powershell", "-command"]
RUN Write-Host hello

# Executed as cmd /S /C echo hello
SHELL ["cmd", "/S", "/C"]
RUN echo hello
```

Дайте отзыв

На следующие инструкции может повлиять `SHELL` инструкция, когда в файле Dockerfile используется их форма в виде, обозначенном как *оболочка*: `RUN`, `CMD` и `ENTRYPOINT`.

Следующий пример представляет собой распространенный шаблон, найденный в Windows, который можно упростить с помощью `SHELL` инструкции:

```
RUN powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

Команда, вызываемая docker, будет:

```
cmd /S /C powershell -command Execute-MyCmdlet -param1 "c:\foo.txt"
```

Это неэффективно по двум причинам. Во-первых, вызывается ненужная cmd.exe команда процессор (он же оболочка). Во-вторых, каждая `RUN` инструкция в *оболочке* форма требует дополнительного `powershell -command` префикса команды.

Чтобы сделать это более эффективным, можно использовать один из двух механизмов. Один из них заключается в использовании формы JSON команды RUN, такой как:

```
RUN ["powershell", "-command", "Execute-MyCmdlet", "-param1 \"c:\\foo.txt\""]
```

Хотя форма JSON однозначна и не использует необязательные `cmd.exe`, она требует большей детализации за счет двойных кавычек и экранирования. Альтернативный механизм заключается в использовании `SHELL` инструкции и формы *оболочки*, создавая более естественный синтаксис для пользователей Windows, особенно в сочетании с `escape` директивой синтаксического анализатора:

```
# escape= `
```

```
FROM microsoft/nanoserver
SHELL ["powershell", "-command"]
RUN New-Item -ItemType Directory C:\Example
ADD Execute-MyCmdlet.ps1 c:\example\
RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

Дайте отзыв

В результате:

```
PS E:\myproject> docker build -t shell .
```



```
Sending build context to Docker daemon 4.096 kB
```

```
Step 1/5 : FROM microsoft/nanoserver
```

```
----> 22738ff49c6d
```

```
Step 2/5 : SHELL powershell -command
```

```
----> Running in 6fcdb6855ae2
```

```
----> 6331462d4300
```

```
Removing intermediate container 6fcdb6855ae2
```

```
Step 3/5 : RUN New-Item -ItemType Directory C:\Example
```

```
----> Running in d0eef8386e97
```

```
Directory: C:\
```

| Mode | LastWriteTime | Length | Name |
|--------|---------------------|--------|---------|
| ---- | ----- | ----- | ---- |
| d----- | 10/28/2016 11:26 AM | | Example |

```
-> 3f2fbf1395d9
```

```
Removing intermediate container d0eef8386e97
```



```
Step 4/5 : ADD Execute-MyCmdlet.ps1 c:\example\  
---> a955b2621c31  
Removing intermediate container b825593d39fc  
Step 5/5 : RUN c:\example\Execute-MyCmdlet 'hello world'  
---> Running in be6d8e63fe75  
hello world  
---> 8e559e9bf424  
Removing intermediate container be6d8e63fe75  
Successfully built 8e559e9bf424  
PS E:\myproject>
```

The `SHELL` instruction could also be used to modify the way in which a shell operates. For example, using `SHELL cmd /S /C /V:ON|OFF` on Windows, delayed environment variable expansion semantics could be modified.

The `SHELL` instruction can also be used on Linux should an alternate shell be required such as `zsh`, `csh`, `tcsh` and others.

Дайте отзыв

Here-Documents

Примечание

Добавлено в `docker/dockerfile:1.4`

Здесь-документы позволяют перенаправлять последующие строки Dockerfile на ввод команд `RUN` или `COPY`. Если такая команда содержит [here-document](#), Dockerfile рассматривает следующие строки до тех пор, пока строка, содержащая только разделитель here-doc, не станет частью той же команды.

Пример: Запуск многострочной

```
# syntax=docker/dockerfile:1  
FROM debian  
RUN <<EOT bash  
  set -ex  
  apt-get update  
  apt-get install -y vim  
EOT
```

Если команда содержит только here-document, ее содержимое оценивается с помощью командной строки по умолчанию.

```
# syntax=docker/dockerfile:1
FROM debian
RUN <<EOT
    mkdir -p foo/bar
EOT
```

В качестве альтернативы, заголовок shebang может использоваться для определения интерпретатора.

```
# syntax=docker/dockerfile:1
FROM python:3.6
RUN <<EOT
#!/usr/bin/env python
print("hello world")
EOT
```

Дайте отзыв

В более сложных примерах может использоваться несколько here-документов.

```
# syntax=docker/dockerfile:1
FROM alpine
RUN <<FILE1 cat > file1 && <<FILE2 cat > file2
I am
first
FILE1
I am
second
FILE2
```

Пример: Создание

В `COPY` командах исходные параметры могут быть заменены индикаторами here-doc.





Применяются обычные правила [расширения переменной here-doc](#) и [удаления табуляции](#) .

```
# syntax=docker/dockerfile:1
FROM alpine
ARG FOO=bar
COPY <<-EOT /app/foo
    hello ${FOO}
EOT
```

```
# syntax=docker/dockerfile:1
FROM alpine
COPY <<-"EOT" /app/script.sh
    echo hello ${FOO}
EOT
RUN FOO=abc ash /app/script.sh
```

Дайте отзыв

Примеры файлов Dockerfile см.:

- [Раздел "создание образов"](#) 
- [Руководство "Начало работы"](#) 
- [Руководства по началу работы для](#) , которые зависят от 

1. Стоимость требуемых   

2. Для встроенного в Docker [BuildKit](#)  и `docker buildx build` 

0 нас

Карьера

Связаться с нами

Клиенты

Рассылка



Служба новостей

Хранилище Swag

Виртуальные
события

Что такое
контейнер?

Почему Docker?

Разработчики

Заблокировать

Сообщество

Начало работы

Открытый исходный
код

Программа
предварительного
просмотра

Примеры
использования

Характеристики

Среда выполнения
контейнера

Средства
разработки

Рабочий стол
Docker

Docker Hub

Дорожная карта
продукта Docker

Безопасная цепочка
поставок
программного
обеспечения

Доверенный контент

Предложения по продуктам

Бизнес Docker

Дайте отзыв



- Личный кабинет
- Docker
- Docker Pro
- Команда Docker
- Проверенный издатель Docker
- Партнеры
- Часто задаваемые вопросы о ценах

- Условия обслуживания
- Статус
- Юридическая информация

Дайте отзыв

Авторское право © 2013-2023 Docker Inc. Все права защищены.

Настройки файлов cookie

- Twitter
- YouTube
- GitHub
- LinkedIn
- Facebook
- Reddit

