Московский Государственный Универстит имени М. В. Ломоносова

Реферат

на тему:«Технология Docker: внутренняя архитектура и применение»

Выполнил: студент гр. 420

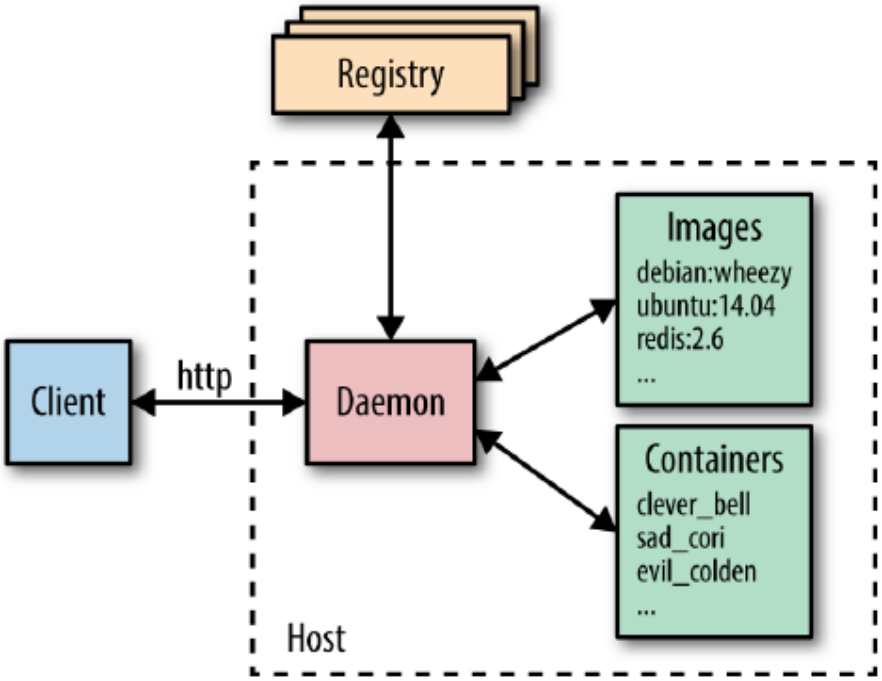
Немешаева Алиса

Москва 2021

Содержание

1 Архитектура технологии Docker

1.1 Компоненты Docker



Основные компоненты Docker на изображены на диаграмме выше:

■ В центре всей системы находится фоновый процесс Docker, который отвечает за создание, запуск и мониторинг контейнеров, а также создание и хранение образов. Обе эти цели изображены справа на диаграмме. Фоновый процесс Docker запускается при запуске команды docker daemon, о чем обычно заботится операционная система хоста.

■ Клиент Docker показан слева вне образа (пунктирная линия) и используется для связи с фоновым процессом Docker через HTTP. По умолчанию это происходит через сокет домена Unix, но он также может использовать сокет TCP для включения удаленных клиентов или файловый дескриптор для сокетов, управляемых системой. Поскольку все коммуникации должны осуществляться через HTTP, легко подключаться к удаленным фоновым процессам Docker и разрабатывать привязки к языкам программирования, но это также имеет последствия для реализации функций, таких как требование контекста сборки для файлов Docker.
API, используемый для связи с фоновым процессом, хорошо определен и задокументирован, что позволяет разработчикам писать программы, которые взаимодействуют напрямую с фоновым процессом, без использования клиента Docker. Клиент и фоновый процесс Docker

распространяются как единый двоичный файл.

■ Реестры Docker (в верхней части диаграммы) хранят и распространяют образы. Реестром по умолчанию является Docker Hub, в котором размещаются тысячи общедоступных образов, а также тщательно отобранные «официальные» образы.
Многие организации имеют собственные реестры, которые можно использовать для хранения коммерческих или конфиденциальных изображений, а также избежать накладных расходов, связанных с необходимостью загрузки изображений из Интернета. Фоновый процесс Docker будет загружать образы из реестров в ответ на запросы docker pull. Он также будет автоматически загружать образы, указанные в запросах запуска Docker и в инструкции FROM файлов Docker, если они недоступны локально.

1.2 Возможности Docker

Первое преимущество Docker — это минимальное потребление ресурсов - контейнер не виртуализирует всю операционную систему, но использует ядро хоста и изолирует программу на уровне процесса. Последний использует гораздо меньше локальных вычислительных ресурсов, чем виртуальные машины.

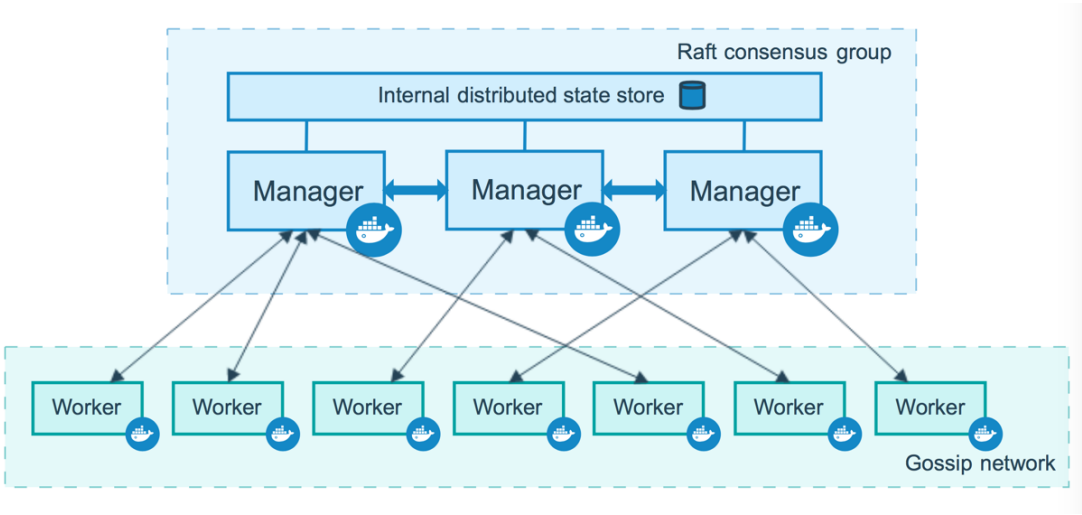
Кроме того, полезным будет быстрое развертывание - не нужно устанавливать вспомогательные компоненты, но можно использовать готовый образ Docker. Например, нет смысла постоянно устанавливать и настраивать Linux Ubuntu. Нужно установить его только один раз, создать образ, использовать его постоянно и при необходимости обновить версию.

Docker обладает возможностью скрыть фоновые процессы, используя разные методы обработки данных для каждого контейнера; а также позволяет работать с технологией изоляции опасного кода-контейнера и может запускать любой код без ущерба для ОС.

Кроме того, всегда можно расширить любой проект, добавив новые контейнеры; любой проект можно запускать внутри контейнера любого драйвера докера.

Оптимизация файловой системы помогает организовать образы в слои, что позволяет очень эффективно использовать файловую систему.

1.3 Способы управления и конфигурирования



Docker включает в себя swarm mode для встроенного управления кластером модулей Docker, который называется swarm. Docker CLI использвуется для создания swarm, развертывания сервисов приложений в swarm и управления поведением swarm. Режим Docker Swarm встроен в Docker Engine.

1.3.1 Основные характеристики

Управление кластером, интегрированное с Docker Engine: позволяет использовать интерфейс командной строки Docker Engine, чтобы создать swarm для Docker Engine, где пользователь Docker может развертывать службы приложений. Для создания swarm или управления им не требуется дополнительное программное обеспечение для оркестровки.

Децентрализованный дизайн: Docker позволяет создать целый рой из одного образа диска.

Декларативная модель обслуживания: Docker Engine использует декларативный подход, позволяющий определять желаемое состояние различных служб в стеке приложения.

Масштабирование: для каждой службы вы можете объявить количество задач, которые нужно запустить. Когда масштаб увеличивается или уменьшается, менеджер swarm автоматически адаптируется, добавляя или удаляя задачи, чтобы поддерживать желаемое

состояние.

Согласование желаемого состояния: узел менеджера swarm постоянно отслеживает состояние кластера и согласовывает любые различия между фактическим состоянием и выраженным желаемым состоянием.

Сеть с несколькими хостами: пользователь Docker может указать оверлейную сеть для своих служб. Менеджер swarm автоматически назначает адреса контейнерам в оверлейной сети при инициализации или обновлении приложения.

Обнаружение служб: узлы диспетчера Swarm присваивают каждой службе в swarm уникальное DNS-имя и распределяют нагрузку с запущенными контейнерами. Можно запросить каждый контейнер, работающий в swarm, через DNS-сервер, встроенный в swarm.

Балансировка нагрузки: Docker позволяет предоставить порты для служб внешнему балансировщику нагрузки. Внутри swarm позволяет указать, как распределять сервисные контейнеры между узлами.

Безопасность по умолчанию: каждый узел в swarm применяет взаимную аутентификацию и шифрование TLS для защиты связи между собой и всеми другими узлами. Docker позволяет использовать самозаверяющие корневые сертификаты или сертификаты от настраиваемого корневого ЦС.

Последовательные обновления: во время развертывания Docker даёт постепенно применять обновления служб к узлам. Менеджер swarm позволяет контролировать задержку между развертыванием сервиса на разных наборах узлов. Если что-то пойдет не так, можно вернуться к предыдущей версии сервиса.

Конфигурации служб Docker Swarm позволяют хранить неконфиденциальную информацию, такую как файлы конфигурации, вне образа службы или запущенных контейнеров. Это позволяет сохранять образы как можно более общими, без необходимости связывать и монтировать файлы конфигурации в контейнеры или использовать переменные среды.

Конфигурации работают аналогично секретам, за исключением того, что они не шифруются в состоянии покоя и монтируются непосредственно в файловую систему контейнера без использования RAM-дисков. Конфигурации могут быть добавлены или удалены из службы в любое время, и службы могут совместно использовать конфигурацию. Также можно использовать конфигурации в сочетании с переменными среды или метками для максимальной гибкости. Значения конфигурации могут быть стандартными строками или двоичным содержимым (размером до 500 КБ).

2 Технологии ОС, используемые в реализации Docker

Фоновый процесс Docker использует «драйвер исполнения» для создания контейнеров. По умолчанию это собственный драйвер Docker runc, но есть и устаревшая поддержка LXC. Runc очень тесно связан со следующими функциями ядра:

■ cgroups, которые отвечают за управление ресурсами, используемыми контейнером (например, использование центрального процессора и памяти). Они также отвечают за замораживание и размораживание контейнеров, используемых в функциональности docker pause.

■ namespaces - пространства имен отвечают за изоляцию контейнеров; помогая убедиться в том, что файловая система контейнера, имя хоста, пользователи, сеть и процессы отделены от остальной части системы.

2.1 cgroups

Контейнеры Linux полагаются на группы управления (control groups), которые не только отслеживают группы процессов, но также предоставляют показатели использования центрального процессора, памяти и блочного ввода-вывода. Есть возможность получить доступ к этим показателям, а также получить метрики использования сети.

Группы управления представлены через псевдофайловую систему. В последних дистрибутивах эта файловая система находится в /sys/fs/cgroup. В этом каталоге находится несколько подкаталогов с именами devices, freezer, blkio и так далее; каждый подкаталог фактически соответствует разной иерархии cgroup.

В старых системах контрольные группы могли монтироваться в /cgroup без четкой иерархии. В этом случае вместо подкаталогов отображается группа файлов в этом каталоге и, возможно, некоторые каталоги, соответствующие существующим контейнерам.

Чтобы выяснить, где смонтированы контрольные группы, нужно запустить:

grep cgroup /proc/mounts

Расположение файлов контрольных групп значительно различается в версиях v1 и v2. Если в системе присутствует /sys/fs/cgroup/cgroup.controllers, значит используется v2, в противном случае это v1.

Метрики памяти находятся в контрольной группе «memory». Группа управления памятью добавляет небольшие накладные расходы, потому что она очень детально учитывает использование памяти на хосте. Поэтому многие дистрибутивы решили не включать его по умолчанию. Как правило, чтобы включить его, всё, что нужно сделать, это добавить некоторые параметры командной строки ядра: cgroup_enable = memory swapaccount = 1.

Некоторые метрики являются значениями, которые могут увеличиваться или уменьшаться. Например, swap - это объем пространства подкачки, используемого членами контрольной группы. Некоторые другие являются значениями, которые могут только увеличиваться, потому что они представляют собой случаи определенного события. Например, pgfault указывает количество ошибок страниц с момента создания контрольной группы.

2.1.0.1 Метрики

■ cache — объем памяти, используемый процессами этой группы управления, который может быть точно связан с блоком на устройстве. Когда файлы читаются и записываются на диске, это количество увеличивается. Это тот случай, когда используется «обычный» ввод-вывод (системные вызовы открытия, чтения, записи), а также отображаемые файлы (с помощью mmap).

■ mapped_file — указывает объем памяти, отображаемый процессами в группе управления. Он не дает информации о том, сколько памяти используется, он скорее говорит, как его использовать.

■ pgfault, pgmajfault — указывают, сколько раз процесс контрольной группы вызывал «page fault» и «major fault» соответственно. Ошибка страницы происходит, когда процесс обращается к несуществующей или защищенной части своего пространства виртуальной памяти. Первое может произойти, если процесс содержит ошибки и пытается получить доступ к недопустимому адресу (ему отправляется сигнал SIGSEGV, обычно уничтожающий его известным сообщением об ошибке сегментации). Последнее может произойти, когда процесс

читает из зоны памяти, которая была выгружена или которая соответствует отображенному файлу: в этом случае ядро загружает страницу с диска и позволяет процессору завершить доступ к памяти. Это также может произойти, когда процесс выполняет запись в зону памяти, предназначенную для копирования при записи: аналогично, ядро вытесняет процесс, дублирует страницу памяти и возобновляет операцию записи на собственной копии страницы процесса. «Основные» сбои случаются, когда ядру действительно нужно прочитать данные с диска. Когда он просто дублирует существующую страницу или выделяет пустую страницу, это обычная (или «незначительная») ошибка.

■ swap - объем подкачки, используемой в настоящее время процессами в этой контрольной группе.

■ rss - определение счетчика rss = active_anon + inactive_anon - tmpfs (где tmpfs - это объем памяти, используемый файловыми системами tmpfs, смонтированными этой группой управления).

■ active_anon, inactive_anon — объем анонимной памяти, которая была идентифицирована ядром как активная и неактивная соответственно. «Анонимная» память - это память, не связанная со страницами диска. Страницы изначально «активные»; и через определенные промежутки времени ядро просматривает память и помечает некоторые страницы как «неактивные». Каждый раз, когда к ним обращаются снова, они немедленно помечаются как «активные». Когда ядру почти не хватает памяти и приходит время выгрузить его на диск, ядро меняет местами «неактивные» страницы.

■ active_file, inactive_file — кэш-память, с активной и неактивной, аналогичной анонимной памяти выше. Точная формула: cache = active_file + inactive_file + tmpfs. Точные правила, используемые ядром для перемещения страниц памяти между активными и неактивными наборами, отличаются от правил, используемых для анонимной памяти, но общий принцип тот же. Когда ядру необходимо освободить память, дешевле вернуть немодифицированную страницу из этого пула, поскольку она может быть восстановлена немедленно (в то время как анонимные страницы и измененные страницы должны быть сначала записаны на диск).

■ unevictable — объем памяти, который не может быть освобождён; как правило, он учитывает память, «заблокированную» с помощью mlock. Он часто используется криптографи-

ческими структурами, чтобы гарантировать, что секретные ключи и другой конфиденциальный материал никогда не будут выгружены на диск.

■ memory_limit, memsw_limit — на самом деле это не показатели, а напоминание об ограничениях, применяемых к этой контрольной группе. Первый указывает максимальный объем физической памяти, который может использоваться процессами этой группы управления; второй указывает максимальный объем RAM + swap.

2.2 namespaces

Docker использует пространства имен ядра для предоставления изолированного рабочего пространства, называемого контейнером. Когда запускается контейнер, Docker создает для этого контейнера набор пространств имён. Эти пространства имен обеспечивают уровень изоляции. Каждый аспект контейнера работает в отдельном пространстве имён, и его доступ ограничен этим пространством имён.

Docker Engine использует следующие пространства имен в Linux:

■ PID namespace для изоляции процесса.

■ NET namespaceдля управления сетевыми интерфейсами.

■ IPC namespace для управления доступом к ресурсам IPC.

■ MNT namespace для управления точками монтирования файловой системы.

■ UTS namespace для изоляции идентификаторов ядра и версии.

Лучший способ предотвратить атаки с повышением привилегий из контейнера - это настроить приложения контейнера для работы от имени непривилегированных пользователей. Для контейнеров, процессы которых должны запускаться от имени пользователя root в контейнере, Docker позволяет переназначить этого пользователя менее привилегированному пользователю на хосте Docker. Сопоставленному пользователю назначается диапазон UID,

которые функционируют в пространстве имен как обычные UID от 0 до 65536, но не имеют привилегий на самом хост-компьютере.

2.3 Union File Systems

2.3.1 UnionFS

UnionFS - это служба файловой системы для Linux, FreeBSD и NetBSD, которая реализует объединенное монтирование для других файловых систем. Она позволяет накладывать файлы и каталоги отдельных файловых систем, называемых ветвями, образуя единую согласованную файловую систему. Содержимое каталогов, имеющих одинаковый путь в объединенных ветвях, будет отображаться вместе в одном объединенном каталоге в новой виртуальной файловой системе.

При монтировании ветвей указывается приоритет одной ветви над другой. Поэтому, когда обе ветки содержат файл с одинаковым именем, одна получает приоритет над другой.

Различные ветви могут быть файловыми системами только для чтения или чтения / записи, так что запись в виртуальную объединенную копию направляется в конкретную реальную файловую систему. Это позволяет файловой системе выглядеть доступной для записи, но без фактического разрешения операций записи для изменения файловой системы, также известной как копирование при записи. Это может быть желательно, если носитель физически доступен только для чтения, например, в случае Live CD.

Первоначально UnionFS был разработан профессором Эрезом Задоком и его командой из Университета Стоуни-Брук.

2.3.2 aufs

Aufs (Advanced Multilayer Integrated File System) - это альтернативная версия UnionFS, вспомогательной файловой системы, которая выполняет каскадное монтирование слиянием в файловых системах Linux. Изначально аббревиатура является сокращением от Another Union FS.

Aufs - это полностью переписанный код UnionFS, направленный на повышение стабильности и производительности. Кроме того, были введены некоторые новые концепции, такие как балансировка ветвей записи и другие улучшения. Некоторые из этих идей реализованы в UnionFS версии 2.x.

Aufs заменил UnionFS на дистрибутив Linux Knoppix Live CD с конца 2006 года, «улучшив стабильность и производительность». Эта вспомогательная файловая система заменила UnionFS в SLAX (и, в принципе, скрипты Linux-Live), начиная с версии 6. Aufs заменил UnionFS, дистрибутив Linux Xandros, используемый в ASUS Eee PC901. Aufs используется для загрузки live-образов Debian Linux.

2.3.3 OverlayFS

OverlayFS представляет собой реализацию файловой системы с монтируемым объединением для Linux. Она объединяет несколько различных базовых точек монтирования в одну, что приводит к единой структуре каталогов, содержащей базовые файлы и подкаталоги из всех источников. Общие приложения накладывают раздел для чтения / записи на раздел, доступный только для чтения, например, с LiveCD и устройствами IoT с ограниченными циклами записи во флэш-память.

2.3.4 ZFS

ZFS (ранее: файловая система Zettabyte) объединяет файловую систему с диспетчером томов. Изначально создавалась как часть операционной системы Sun Microsystems Solaris в 2001 году. Большие части Solaris, включая ZFS, публиковались под лицензией с открытым исходным кодом как OpenSolaris в течение примерно 5 лет с 2005 года, прежде чем они были помещены под лицензию с закрытым исходным кодом, когда корпорация Oracle приобрела Sun в 2009/2010 гг.

В течение 2005–2010 годов версия ZFS с открытым исходным кодом была перенесена на Linux, Mac OS X (позднее MacZFS) и FreeBSD. В 2010 году проект illumos разветвил последнюю версию OpenSolaris, чтобы продолжить своё развитие как проект с открытым исходным кодом, включая ZFS. В 2013 году была основана OpenZFS для координации разработки ZFS

с открытым исходным кодом. OpenZFS поддерживает и управляет основным кодом ZFS, в то время как организации, использующие ZFS, поддерживают определенный код и процессы проверки, необходимые для интеграции ZFS в свои системы. OpenZFS широко используется в Unix-подобных системах.

2.3.5 Btrfs

Btrfs - это компьютерный формат хранения который объединяет файловую систему, основанную на принципе копирования при записи, с менеджером логических томов (не путать с LVM Linux), разработанным совместно. Первоначально он был разработан в Oracle Corporation в 2007 году для использования в Linux, а с ноября 2013 года формат файловой системы на диске был объявлен в ядре Linux стабильным. Согласно Oracle, Btrfs «не является настоящей аббревиатурой».

Btrfs предназначен для решения проблемы отсутствия пулов, снимков состояния, контрольных сумм и интегрированного охвата нескольких устройств в файловых системах Linux. Крис Мейсон, главный автор Btrfs, заявил, что его цель состояла в том, чтобы «позволить [Linux] масштабироваться для доступного хранилища. Масштабирование - это не только обращение к хранилищу, но и возможность администрировать и управлять им с помощью чистого интерфейса, который позволяет людям видеть, что используется, и делает его более надёжным».

2.4 SELinux

Security-Enhanced Linux (SELinux) - это модуль безопасности ядра Linux, который предоставляет механизм для поддержки политик безопасности контроля доступа, включая обязательный контроль доступа (MAC).

SELinux - это набор модификаций ядра и инструментов пользовательского пространства, которые были добавлены в различные дистрибутивы Linux. Его архитектура стремится отделить исполнение решений по безопасности от политики безопасности и оптимизирует объём программного обеспечения, задействованного в обеспечении соблюдения политики безопасности. Ключевые концепции, лежащие в основе SELinux, восходят к нескольким более ранним проектам Агентства национальной безопасности США (АНБ).

Команда Linux с усиленной безопасностью NSA описывает SELinux NSA как набор исправлений для ядра Linux и утилит для обеспечения сильной, гибкой архитектуры обязательного контроля доступа к основным подсистемам ядра.

Он обеспечивает расширенный механизм для принудительного разделения информации на основе требований конфиденциальности и целостности, который позволяет устранять угрозы взлома и обхода механизмов безопасности приложений и позволяет ограничить ущерб, который может быть причинён вредоносными или некорректными приложениями. Он включает набор образцов файлов конфигурации политики безопасности, предназначенных для достижения общих целей безопасности.

3 Другие свойства Docker

3.1 Сетевое взаимодействие

Одна из причин, по которой контейнеры и службы Docker настолько мощны, заключается в возможности соединять их вместе или подключать к рабочим нагрузкам, не относящимся к Docker. Контейнеры и службы Docker даже не должны знать, развернуты ли они в Docker и являются ли их одноранговые узлы также рабочими нагрузками Docker или нет. Независимо от того, работают ли хосты Docker под управлением Linux, Windows или их сочетанием, можно использовать Docker для управления ими независимо от платформы.

Сетевая подсистема Docker подключается с помощью драйверов. По умолчанию существует несколько драйверов, обеспечивающих основные сетевые функции:

■ bridge — сетевой драйвер по умолчанию. bridge сети обычно используются, когда приложения работают в автономных контейнерах, которым необходимо обмениваться данными.

■ host — для автономных контейнеров: позволяет удалить изоляцию сети между контейнером и хостом Docker и напрямую использовать сеть хоста.

■ overlay — overlay сети соединяют несколько фоновых процессов Docker вместе и позволяют службам Swarm взаимодействовать друг с другом. Можно использовать overlay сети для облегчения связи между службой Swarm и автономным контейнером или между двумя автономными контейнерами на разных фоновых процессах Docker. Эта стратегия устраняет необходимость выполнять маршрутизацию на уровне ОС между этими контейнерами.

■ macvlan — сети macvlan позволяют назначать MAC-адрес контейнеру, чтобы он отображался как физическое устройство в сети. Фоновый процесс Docker направляет трафик в контейнеры по их MAC-адресам. Использование драйвера macvlan иногда является лучшим выбором при работе с устаревшими приложениями, которые ожидают прямого подключения к физической сети, а не маршрутизации через сетевой стек хоста Docker.

■ none: отключает все сети для этого контейнера. Обычно используется вместе с настраиваемым сетевым драйвером. none недоступен для служб Swarm.

■ Сетевые плагины — Docker позволяет устанавливать и использовать сторонние сетевые плагины для контейнера. Эти плагины доступны в Docker Hub или же у сторонних разработчиков.

3.2 Журналирование

Docker включает в себя несколько механизмов ведения журнала, которые помогут получать информацию о запущенных контейнерах и службах. Эти механизмы называются драйверами журналирования. Каждый фоновый процесс Docker имеет драйвер ведения журнала по умолчанию, который использует каждый контейнер, если пользователь не настроил его для использования другого драйвера ведения журнала («драйвера журнала»).

По умолчанию Docker использует драйвер ведения журнала JSON-файла, который внутренне кэширует журналы контейнера как JSON. Помимо использования драйверов ведения журнала, включенных в Docker, пользователь также получает возможность реализовать и использовать плагины драйверов ведения журнала.

3.3 Мониторинг

Prometheus - это набор инструментов для мониторинга и оповещения систем с открытым исходным кодом. Docker может быть настроен как цель Prometheus. Prometheus позволяет отслеживать только сам Docker.

3.4 Ограничение ресурсов контейнеров

По умолчанию контейнер не обладает ограничениями по ресурсам и может использовать столько заданного ресурса, сколько позволяет планировщик ядра хоста. Docker предоставляет способы контролировать, сколько памяти или мощностей центрального процессора может использовать контейнер, устанавливая флаги конфигурации времени выполнения команды docker run. Многие из этих функций требуют, чтобы ядро поддерживало возможности Linux.

3.5 Концепции технологий, позволяющих организовывать кластеры контейнеров

Kubernetes - это система оркестровки контейнеров с открытым исходным кодом для автоматизации развёртывания компьютерных приложений, масштабирования и управления. Первоначально он был разработан Google и сейчас поддерживается Cloud Native Computing Foundation. Его цель — предоставить «платформу для автоматизации развёртывания, масштабирования и операций контейнеров приложений в кластерах хостов». Он работает с рядом инструментов для работы с контейнерами и запускает контейнеры в кластере, часто с образами, созданными с помощью Docker.

Первоначально Kubernetes взаимодействовал со средой выполнения Docker через Dockershim; однако с тех пор такой способ организации процесса не рекомендуется в пользу прямого взаимодействия с контейнером через containerd или замены Docker средой выполнения, совместимой с интерфейсом времени выполнения контейнера (CRI), введенным Kubernetes в 2016 г.

4 Другие сценарии использования Docker

4.1 Сценарии использования Docker для распространения программного обеспечения

При развёртывании современных приложений и сервисов разработчики часто сталкиваются с проблемами создания приложений, которые работают на широком спектре различных аппаратных и программных платформ.

Устройства и серверы работают под управлением множества различных операционных систем, включая Windows и широкий спектр дистрибутивов Linux. Даже в одной и той же ОС приложения запускаются на самых разных стеках программного обеспечения, таких как Java, .NET, Go, Ruby, Python и так далее. Следовательно, для развёртывания любого приложения необходимы серверы, отвечающие следующим требованиям:

- Конкретная операционная система
- Конкретные библиотеки
- Конкретные программные стеки
- Конкретные версии всего вышеперечисленного

Выполнение вышеуказанных требований создает значительные трудности в процессе развёртывания программного обеспечения, затрудняя переход между стеками и версиями программного обеспечения без необходимости перенастраивать или отключать существующие серверы. В эпоху облачных вычислений часто требуется перемещать приложения из локального устройства в облако и между различными облачными средами.

Все эти вопросы возникают очень часто, поэтому стало важно всегда упаковывать приложения с их собственной средой и зависимостями таким образом, чтобы они оставались полностью независимыми от среды хоста и друг от друга, что позволяет развёртывать и запускать их в любой целевой системе.

Контейнеры были созданы для обеспечения такого типа изоляции, необходимой для создания масштабируемых современных приложений. Контейнеры упаковывают приложение, конкретную операционную систему, которую оно требует, библиотеки времени выполнения и любые зависимости приложения в виде единого развертёваемого модуля, называемого образом. Такой подход позволяет развернуть образ приложения на любом сервере, даже не спрашивая, какая операционная система, программный стек или среда выполнения существуют в целевой системе. Контейнеры позволяют приложению работать в полной изоляции от остальной части операционной системы. Эта идея позволяет убедиться в том, что образ итогового приложения будет работать на выбранном сервере Linux точно так же, как оно работает и ведет себя на персональном компьютере с Windows, используемой для разработки. Это полностью освобождает разработчиклв от каких-либо проблем с развёртыванием и позволяет изменять стеки, версии программного обеспечения и даже варианты операционной системы по своему желанию без какой-либо реконфигурации сервера.

4.2 Сценарии использования Docker для разработки программного обеспечения в рамках процессов непрерывной интеграции

Docker имеет интеграцию со многими инструментами непрерывной интеграции, в том числе популярным инструментом CI (Continuous Integration), известным как Jenkins. В Jenkins доступны плагины, которые можно использовать для работы с контейнерами.

Прежде чем подумать об оптимизации CI/CD (Continuous Integration/Continuous Delivery), всегда важно подумать о внутреннем цикле разработки для данного проекта и о том, как он соотносится с внешним циклом (CI). Для того, чтобы сделать их наиболее похожими может быть хорошей идеей запускать модульные тесты как часть команды сборки Docker, добавив для них цель в свой Dockerfile.

Заключение

Docker даёт огромное количество преимуществ при разработке. Docker он не зависит от языка на уровне локализации, что означает, что разработчики могут унифицировать среду и сэкономить много времени на настройку перед тем, как начать разработку. А для разработки программного обеспечения экономия времени - это не только повышение собственной эффективности; это также помогает клиентам сэкономить на расходах.

За время разработки Docker претерпел множество изменений. Когда Docker только появился, он имел множество недостатков, которые приходилось нейтрализовать прежде чем начать полноценное использование. Например:

■ Поддержка MacOS могла доставить трудности на начальных этапах разработки, и многие используют эту ОС для разработки.

■ Когда Docker только появился, ещё не было docker-compose, а это значит, что не было стандартного способа создания контейнера Docker. Приходилось тратить дополнительное время на настройку параметров вручную.

■ Docker был нестабильным при обработке экземпляров с отслеживанием состояния, поэтому мы довольно долгое время использовали Docker только для служб без отслеживания состояния.

■ Ещё не было известно, станет ли Docker устойчивой технологией.

Тем не менее, Docker помогает получить значительные преимущества:

а) Стандартизация окружающей среды.

Поскольку для Docker задокументированы инструкции по созданию среды с помощью Dockerfile, разработчики могут минимизировать несогласованность между различными средами. Большим преимуществом является то, что это делает среду разработки повторяемой. Это также помогает убедиться, что все члены команды работают в одной среде.

б) Более быстрая настройка и единообразие

С Docker настройки становятся проще. Вы можете просто поместить свои конфигурации в код и развернуть его. Вы также можете многократно использовать свои конфигурации, поскольку Docker поддерживает широкий спектр сред.

Одним из самых больших преимуществ использования Docker является ускорение настройки проекта, особенно для новых разработчиков. Среда разработки одинакова для каждого разработчика. Как только разработчики добиваются согласованности, даже новые члены команды смогут пропустить трудоемкие настройки среды и сразу же начать разработку с использованием любого языка программирования. За счет экономии времени на настройку разработчики могут уделять больше времени важным вещам, таким как разработка или тестирование QA.

Еще одно преимущество состоит в том, что после стандартизации конфигурации Docker экономится много времени на подготовке документации по настройке и развертыванию, касающейся процедур и процессов. Даже при меньшем количестве документации можно быть уверенными, что операционная среда согласована.

в) Лучшее аварийное восстановление.

Катастрофа непредсказуема. Однако Docker позволяет создать резервную копию образа (также называемого «моментальным снимком») для состояния контейнера в этот момент резервного копирования и получить его позже, когда возникнут серьезные проблемы. Например, произошел аппаратный сбой, и нужно переключить работу на новое оборудование. С помощью Docker всегда можно легко реплицировать файл на новое оборудование.

Эта технология позволяет вернуться к зафикисированному рабочему состоянию для разных процессов внутри образа, даже если они связаны с разными языками программирования.

г) Более широкое распространение DevOps.

Когда люди говорят об отношениях между Docker и DevOps, первое, что приходит на ум, - это CI/CD. С Docker среда тестирования соответствует производственной среде. Если эти изменения могут пройти CI/CD, они также должны работать в производственной среде. Docker делает процесс от тестирования до производства более плавным.

Кроме того, стандартизация играет ключевую роль в автоматизации. Docker упрощает DevOps за счет стандартизации интерфейса конфигурации и упрощает настройку машины. Поскольку Docker имеет стандартный интерфейс настройки, можно гарантировать, что интерфейс стандартизирован для всех участников, и никто не будет разочарован постоянно меняющимися пользовательскими интерфейсами.

Список использованных источников

1. *Mouat, Adrian.* Using Dockers / Adrian Mouat.

2. *https://docs.docker.com/*. Официальная документация Docker / https://docs.docker.com/.

3. *https://man7.org/linux/man-pages/man7/cgroups.7.html*. Официальная документация по cgroup / https://man7.org/linux/man-pages/man7/cgroups.7.html.

4. *https://kubernetes.io/*. Официальная документация по Kubernetes / https://kubernetes.io/.

5. *Merkel, Dirk.* Docker: lightweight Linux containers for consistent development and deployment / Dirk Merkel.

6. *https://access.redhat.com/documentation*. Официальная документация по SELinux / https://access.redhat.com/documentation.

7. *https://docs.docker.com/engine/release-notes/*. Docker Engine release notes / https://docs.docker.com/engine/release-notes/.

8. *Kowalkowski, Jim.* Study of a Docker use-case for HEP / Jim Kowalkowski.

9. *Turnbull, James.* The Docker Book: Containerization is the new virtualization / James Turnbull.

10. *Nicoloff, Jeff.* Docker in Action (2nd Edition) / Jeff Nicoloff.

11. *Pethuru Raj, Vinod Singh.* Learning Docker / Vinod Singh Pethuru Raj, Jeeva S. Chelladhurai.

12. *Karl Matthias, Sean P. Kane.* Docker: Up and Running / Sean P. Kane Karl Matthias.

13. *Poulton, Nigel.* Docker Deep Dive / Nigel Poulton.

14. *Davis, Ashley.* Bootstrapping Microservices with Docker, Kubernetes, and Terraform / Ashley Davis.

15. *Ifrah, Shimon.* Getting Started with Containers in Google Cloud Platform / Shimon Ifrah.

16. *Singh, Pramod.* Deploy Machine Learning Models to Production / Pramod Singh.

17. *Scott Surovich, Marc Boorshtein.* Kubernetes and Docker / Marc Boorshtein Scott Surovich.

18. *Ghosh, Saibal.* Docker Demystified / Saibal Ghosh.

19. *Richard Bullington-McGuire Andrew K. Dennis, Michael Schwartz.* Docker for Developers / Michael Schwartz Richard Bullington-McGuire, Andrew K. Dennis.

20. *Sarmiento, Edwin M.* The SQL Server DBA's Guide to Docker Containers / Edwin M Sarmiento.