Engineer

Середовище виконання контейнерів

програма, яка насправді запускає контейнер

Що це таке

Середовище виконання контейнерів, англійською container runtime, — це програмне забезпечення між образом контейнера на диску та ізольованим процесом у ядрі. Ця назва охоплює два рівні, і чимало плутанини виникає через те, що люди мають на увазі різні: низькорівневе середовище, яке ще називають OCI-runtime, і високорівневе, яке частіше звуть рушієм контейнерів. Людина здебільшого працює із середовищами виконання контейнерів через рушій, а не напряму.

Низькорівневе середовище має одне вузьке завдання. Воно читає бандл, який специфікація середовища виконання від Open Container Initiative визначає як каталог із файлом `config.json` і кореневою файловою системою контейнера, просить у ядра простори імен, cgroups і монтування, описані в конфігурації, і запускає процес усередині них. Специфікація називає операції, які мусить підтримувати кожне таке середовище: create, start, kill, delete і запит стану, а командний рядок залишає на розсуд реалізації. Версія 1.0 вийшла в липні 2017 року. runc, який підтримує сама OCI, описує себе як низькорівневий інструмент, не призначений для кінцевого користувача. crun виконує ту саму роботу, написаний мовою C, споживає менше пам’яті й названий у документації Podman типовим. youki — ще одне, написане на Rust.

Рушій розташований вище. containerd описує себе як демон, що керує повним життєвим циклом контейнерів на своєму хості: передаванням і зберіганням образів, виконанням контейнерів і наглядом за ними, а також підключенням сховища та мережі. CRI-O виконує подібну роботу, але створений лише для Kubernetes. Рушій завантажує й розпаковує образ, записує бандл, передає його низькорівневому середовищу й продовжує стежити за процесом і після того, як те завершило роботу. Docker Engine стоїть ще на щабель вище: за його документацією, він використовує

containerd для життєвого циклу контейнерів, а containerd типово запускає runc. Podman іде іншим шляхом і обходиться без постійного демона. Кожна команда сама готує бандл, викликає crun або runc і залишає запущений контейнер під наглядом невеликого процесу conmon.

Kubernetes додає до словника ще одне поняття. Kubelet на кожному вузлі спілкується з рушієм через Container Runtime Interface, або CRI, — протокол на gRPC, який реалізують і containerd, і CRI-O. Docker Engine старший за CRI, тож Kubernetes роками тримав для нього адаптер dockershim і прибрав його у версії 1.24.

Застосуйте поняття до рішення

Застосуйте «Що таке середовище виконання контейнерів?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Запакована так, що поводиться однаково на ноутбуці, на тестовому сервері й у продакшні, і масштабується, коли настає напружений тиждень.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Вела 20 застосунків Docker у Google Cloud і AWS — 20 контейнеризованих застосунків розгорнуто й поставлено під моніторинг

Вела 20 застосунків Docker у Google Cloud і AWS Зв'язатися з нами →

Чому це важливо

Вибір середовища виконання насправді є вибором рівня, який змінюють, і ці два рівні дають різне.

Рушій визначає повсякденне життя контейнерів: хто завантажує й зберігає образи, хто перезапускає контейнер після збою, куди йдуть його журнали, яким API користуються інші інструменти і кому належить дерево процесів. Заміна Docker Engine на Podman на сервері чи на containerd на вузлі Kubernetes змінює все це, а самі контейнери лишає як є, бо формат образу й бандл спираються на ті самі стандарти OCI. Документація Kubernetes про вилучення dockershim каже прямо: образи, зібрані через `docker build`, працюють з усіма реалізаціями CRI.

Низькорівневе середовище визначає, наскільки ізольований процес. runc і crun запускають звичайний процес на ядрі хоста: це швидко й повністю сумісно, і всі контейнери на машині ділять одне ядро. Середовище gVisor, runsc, ставить між контейнером і хостом прикладне ядро, написане на Go, і

саме відповідає на системні виклики контейнера; документація проєкту називає ціною цього нижчу сумісність застосунків і більші витрати на кожен системний виклик. Kata Containers запускає кожне навантаження в легкій віртуальній машині з власним ядром, використовує апаратну віртуалізацію як другий рубіж захисту і потребує гіпервізора на кожному вузлі, де працює.

Kubernetes дозволяє робити другий вибір для окремого пода, а не для всього кластера. RuntimeClass указує на обробник, налаштований у рушії, і под із полем `runtimeClassName` працює під цим обробником. Так навантаження, якому не довіряють, може жити в gVisor чи Kata, а решта лишається на runc. Документація описує це як баланс між продуктивністю та безпекою.

Як це виглядає на практиці

У повсякденній мові назва тяжіє до рівня рушія, і робота на цьому сайті вживає її саме так.

Двадцять застосунків, упакованих у Docker і розподілених між двома хмарами й кількома середовищами виконання контейнерів, мали розгортатися однаковим способом і швидко діагностуватися в разі проблем. Docker їх пакував, усунення несправностей ішло через Podman, а налагодження оркестрації — через Kubernetes. Така робота з кількома рушіями тримається на описаному вище поділі на рівні: рушії різні, а стандарти OCI для образів і середовищ виконання під ними спільні.

На новішому хості вибір зроблено на рівні рушія. Контейнерний шар працює на Podman і Quadlet замість Docker, бо типовий рушій публікує порти контейнерів вище за правила брандмауера, які встановлює роль посилення захисту, і додав би другий менеджер процесів поруч із systemd. Вибір змінив те, хто запускає кожен контейнер і хто за ним потім стежить: контейнери стали службами, які запускаються так, як запускаються служби, під менеджером процесів, якому хост уже довіряв.

Те саме міркування поширюється на хост із 512 МБ для публічних сервісів компанії, де пам’ять вирішувала кожну суперечку. Його контейнерний шар — це юніти Quadlet під systemd, а не демон із власним супервізором; такий вибір зроблено на машині, де придатні до використання близько 464 МБ пам’яті.

Де виникають проблеми

Проблеми із середовищами виконання зазвичай з'являються на стиках між рівнями, де кожна сторона вважає, що інша налаштована так, як вона очікує.

Інструменти на вузлі сліпнуть після оновлення кластера. Починаючи з Kubernetes 1.24, kubelet спілкується з containerd або CRI-O напряму, тож контейнери, які планує кластер, оминають Docker Engine. `docker ps` на вузлі показує лише те, що запустив сам Docker, а агенти моніторингу, стартові скрипти й привілейовані поди, які викликають Docker CLI або монтують `docker.sock`, втрачають огляд. Навантаження працюють далі. Переводити на `crictl` або на власний API рушія треба інструменти навколо них.

Kubelet і середовище виконання керують cgroups по-різному. Обидва мусять використовувати той самий драйвер cgroup, `cgroupfs` або `systemd`, а kubelet типово бере `cgroupfs`. На хості, де systemd є системою ініціалізації, це дає два менеджери cgroup із двома різними поглядами на ті самі ресурси, і документація Kubernetes попереджає, що такі вузли можуть ставати нестабільними, коли ресурсів бракує. Виправлення — драйвер systemd з обох боків; саме його рекомендують і для cgroup v2. Для containerd 2.x з runc бік середовища виконання має такий вигляд:

```
# /etc/containerd/config.toml, containerd 2.x
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc.options]
  SystemdCgroup = true
```

На боці kubelet це `cgroupDriver: systemd` у його конфігурації, а containerd після цього треба перезапустити. containerd 1.x пише назву розділу інакше, і сторінка Kubernetes наводить обидві форми. Зміна драйвера на вузлі, який уже приєднано до кластера, — делікатна операція: поди, створені з одним драйвером, можуть отримати помилки під час повторного створення з іншим, тому документація радить замінити вузол або перевстановити його з новою конфігурацією.

Под, перенесений до ізольованого середовища, зупиняється на старті. RuntimeClass з обробником, якого рушій вузла не має, одразу переводить под у фазу Failed, а причину видно в події. Далі навантаження має пасувати до пісочниці. gVisor відповідає на системні виклики у власному ядрі, тож програми, що спираються на рідше вживані можливості ядра, можуть

поводитися інакше, а документація називає навантаження з великою кількістю системних викликів поганим варіантом. Kata потребує апаратної віртуалізації на вузлі, а на хмарній віртуальній машині це означає вкладену віртуалізацію. Селектор вузлів `scheduling` у RuntimeClass тримає такі поди на вузлах, які здатні їх запустити.

Образ, імпортований вручну, лишається невидимим для kubelet. containerd зберігає назви образів і метадані окремо для кожного простору імен. Його клієнт `ctr` працює в просторі `default`, якщо не вказати інший, а втулок CRI, з яким говорить kubelet, — у `k8s.io`. Тож образ, завантажений звичайним `ctr images import`, опиняється там, куди kubelet не дивиться, і под знову йде до реєстру або зупиняється з помилкою завантаження образу. Імпорт через `ctr -n k8s.io images import` і перевірка через `crictl images`, яка бачить те саме, що й kubelet, закривають цю щілину.

Що почитати далі

- OCI Runtime Specification — стандарт, який реалізує кожне низькорівневе середовище: бандл, `config.json` і операції життєвого циклу.
- runc — власне середовище виконання OCI з поясненнями, як ним керують інструменти вищого рівня.
- Container Runtimes — драйвери cgroup, версії CRI та налаштування containerd і CRI-O, яких очікує Kubernetes.
- Container Runtime Interface (CRI) — протокол gRPC між kubelet і рушієм.
- Runtime Class — вибір іншого обробника для окремого пода і як тримати такі поди на вузлах, що його підтримують.
- Alternative container runtimes — як Docker Engine передає контейнери до gVisor, Kata Containers, youki чи Wasmtime.

Словник

Обирайте за фактичними умовами

Опишіть шлях розгортання, експлуатаційні ризики та чергову команду. Ми можемо визначити зміну з перевірками, шляхом відновлення та чіткою відповідальністю.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке середовище виконання контейнерів?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/container-runtime/