AXIOM JDK

Axiom Linux
Руководство по настройке JDK для контейнеров с ограниченными ресурсами

Axiom Linux I Декабрь 2025

Программное обеспечение АКСИОМ содержит программное обеспечение с открытым исходным кодом. Дополнительная информация о коде сторонних разработчиков доступна на сайте https://axiomjdk.ru/third_party_licenses. Для дополнительной информации о том, как получить копию исходного кода, можно обратиться по адресу info@axiomjdk.ru.

ДАННАЯ ИНФОРМАЦИЯ МОЖЕТ ИЗМЕНЯТЬСЯ БЕЗ ПРЕДВАРИТЕЛЬНОГО УВЕДОМЛЕНИЯ. АКСИОМ ПРЕДОСТАВЛЯЕТ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ "КАК ЕСТЬ" БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, АКСИОМ ПРЯМО ОТКАЗЫВАЕТСЯ ОТ ВСЕХ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ПОДРАЗУМЕВАЕМЫМИ ГАРАНТИЯМИ ТОВАРНОЙ ПРИГОДНОСТИ И ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ.

АКСИОМ НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ КОСВЕННЫЕ, СЛУЧАЙНЫЕ, СПЕЦИАЛЬНЫЕ, ШТРАФНЫЕ ИЛИ КОСВЕННЫЕ УБЫТКИ, ИЛИ УБЫТКИ ОТ ПОТЕРИ ПРИБЫЛИ, ДОХОДА, ДАННЫХ ИЛИ ИСПОЛЬЗОВАНИЯ ДАННЫХ, ПОНЕСЕННЫЕ ВАМИ ИЛИ ЛЮБОЙ ТРЕТЬЕЙ СТОРОНОЙ, БУДЬ ТО В РЕЗУЛЬТАТЕ ДЕЙСТВИЯ ДОГОВОРА ИЛИ ДЕЛИКТА, ДАЖЕ ЕСЛИ АКСИОМ БЫЛО ПРЕДУПРЕЖДЕНО О ВОЗМОЖНОСТИ ТАКИХ УБЫТКОВ.

Использование любого программного продукта АКСИОМ регулируется соответствующим лицензионным соглашением, которое никоим образом не изменяется условиями данного уведомления. Программные продукты и фирменные наименования: Axiom JDK, Axiom JDK Pro, Axiom Runtime Container Pro, Axiom Linux, Libercat, Libercat Certified и АКСИОМ принадлежат АКСИОМ и их использование допускается только с разрешения правообладателя.

Товарный знак Linux® используется в соответствии с сублицензией от Linux Foundation, эксклюзивного лицензиата Линуса Торвальдса, владельца знака на всемирной основе. Java и OpenJDK являются товарными знаками или зарегистрированными товарными знаками компании Oracle и/или ее аффилированных лиц. Другие торговые марки являются собственностью их соответствующих владельцев и используются только в целях идентификации.

Содержание

1. Введение

Мировая тенденция использовать приложения и сервисы в контейнерной среде не собирается замедляться и будет продолжать стимулировать предстоящие обновления программного обеспечения и ИТ-инфраструктуры.

Говоря о контейнерах, люди всегда думают о микросервисах, микроконтейнерах или минимизации использования ресурсов. В некоторой степени все это верно и действительно помогает более разумно управлять ресурсами, используя небольшие контейнеры, настроенные для конкретного приложения, и функциональные группы cgroups для контроля ресурсов, используемых запущенными контейнерами. Control groups — это функция ядра Linux, которая помогает организовывать процессы в иерархические группы с использованием различных ресурсов, которые можно ограничивать и контролировать. Сейчас она поддерживается во многих дистрибутивах Linux, включая новые, такие как Axiom. Поскольку Axiom Linux предоставляет механизмы управления контейнерами, пользователи могут выбрать любой из них и разрешить cgroups управлять ресурсами на хост-компьютере контейнера.

2. Введение в control groups

Control groups (cgroups) v1 и v2 продолжают существовать параллельно, и обе поддерживаются. В этом документе представлены примеры на основе control groups v1 как наиболее зрелой технологии, которая хорошо работает с Podman и Docker. Реализация control groups v1 обеспечивает иерархию контроллеров, зависящую от ресурса. Другими словами, каждый ресурс, например ЦП, память, ввод-вывод и т. д., имеет свою собственную иерархию групп управления.

Давайте в качестве примера настроим cgroups в Axiom Linux. Axiom Linux - это небольшой и безопасный дистрибутив, настроенный для запуска рабочих нагрузок Java. Любой другой дистрибутив Linux также подходит, если он обеспечивает поддержку cgroups и пакетов Podman/Docker.

Выполните следующую команду для установки cgroups в Axiom Linux:

```
sudo rc-update add cgroups
sudo rc-service cgroups start
```

cgroups v1 можно установить, отредактировав `/etc/rc.conf`, присвоив параметру `legacy` значение `rc_cgroup_mode`. Если вы решите настроить cgroups v2, вам следует назначить параметр `unified` для `rc_cgroup_mode`. Вы можете включить контроллеры для cgroups v2 в параметре `rc_cgroup_controllers` со следующими опциями: `cpuset cpu io memory hugetlb pids`.

Пример модификации файла `rc.conf`:

```
# This sets the mode used to mount cgroups.
# "hybrid" mounts cgroups version 2 on /sys/fs/cgroup/unified and
# cgroups version 1 on /sys/fs/cgroup.
# "legacy" mounts cgroups version 1 on /sys/fs/cgroup
# "unified" mounts cgroups version 2 on /sys/fs/cgroup
rc_cgroup_mode="legacy"
```

Возможно, вам потребуется выполнить следующую команду, чтобы изменения вступили в силу, или перезагрузить хост:

```
sudo rc-service cgroups restart
```

Для правильного управления ресурсами с помощью cgroups используйте соответствующую среду выполнения OCI в настройках. Мы рекомендуем настроить среду выполнения Podman `crun` в качестве среды выполнения по умолчанию, поскольку она отлично работает с cgroups v1.

Отредактируйте файл `/etc/containers/containers.conf` и измените значение `Default OCI runtime` на постоянное использование `crun`.

Типичное управления ресурсами заключается в предоставлении соответствующих опций в команде `podman run`. В таблице ниже перечислены несколько подходящих опций, которые вы можете сразу же использовать в своих экспериментах и настройках.

Option	Description
–cgroup-conf=KEY=VALUE	При запуске в cgroups v2 укажите файл cgroup для записи и его значение. Например, `–cgroup-conf=memory.high=1073741824` устанавливает ограничение `memory.high` равным 1 ГБ.
–cpu-period=limit	Устанавливает период использования ЦП для планировщика CFS, который представляет собой длительность в микросекундах. Как только квота ЦП контейнера будет полностью использована, его запуск не будет запланирован до окончания текущего периода. Значение по умолчанию равно 100000 микросекунд.
–cpu-quota=limit	Ограничивает квоту процессора для планировщика CFS, т.е. ограничивает использование процессора контейнером. По умолчанию контейнеры запускаются с полным ресурсом процессора. Ограничение - это число в микросекундах. Если указано число, контейнеру будет разрешено использовать процессорное время, пока не закончится период работы процессора (контролируется через –cpu-period).
–cpus=number	Устанавливает количество используемых процессоров. Значение по умолчанию равно 0.0, что означает отсутствие ограничений.

Option	Description
–cpuset-cpus=number	Указывает процессоры, в которых разрешено выполнение. Может быть указан в виде списка, разделенного запятыми (например, 0,1), в виде диапазона (например, 0-3) или любой их комбинации (например, 0-3,7,11-15).
–memory, **-m**=number[unit]	Устанавливает ограничение памяти. Единицей измерения могут быть b (bytes), k (kibibytes), m (mebibytes), org (gibibytes). Позволяет ограничить объем памяти, доступный контейнеру.

См. документацию по команде podman run.

Ниже приведен пример вариантов назначения ресурсов для запущенного контейнера.

Типичные хосты контейнеров имеют большой объем памяти и большие мощности процессора, однако довольно часто устанавливаются ограничения для запущенных контейнеров, поэтому все контейнеры получают достаточно ресурсов, в которых они нуждаются. Вот пример выполнения контейнера Podman с ограничениями cpu=1 и mem=1G.

```
podman run --cpus=1 --memory=1G -it --rm <URL адрес репозитория>/axiom-
runtime-container-pro:jdk-all-17-glibc
catm/sys/fs/cgroup/memory/memory.limit_in_bytes
1073741824
podman run --cpus=1 --memory=1G -it --rm <URL адрес репозитория>/axiom-
runtime-container-pro:jdk-all-17-glibc cat
/sys/fs/cgroup/cpu/cpu.cfs_quota_us
100000
```

Если вы хотите увеличить лимит использования процессора, обновите значение параметра соответствующим образом.

```
podman run --cpus=2 --memory=1G -it --rm <URL адрес репозитория>/axiom-
runtime-container-pro:jdk-all-17-glibc cat
/sys/fs/cgroup/cpu/cpu.cfs_quota_us
200000
```

3. Производительность JDK

Когда дело доходит до JDK, мы должны учитывать некоторые особенности, чтобы получить максимальную отдачу от контейнеризированных приложений или сервисов. Несмотря на то, что все дело в микросервисах, микроконтейнерах и минимальном использовании ресурсов, контейнеры Java будут страдать от чрезмерно ограниченных ресурсов, предоставляемых запущенному контейнеру. Давайте рассмотрим один пример, связанный с поведением Java в условиях сокращения ресурсов процессора и памяти. Следующий пример посвящен сборке мусора, которая влияет на общие результаты производительности.

Версия Java

Современные JDK разработаны так, чтобы быть ориентированными на контейнерную среду и быть дружественными к ней. Некоторые версии JDK способны обнаруживать принудительные квоты ресурсов с поддержкой Linux control group (cgroup). На данный момент все текущие версии Axiom JDK, такие, как Axiom JDK 17.0.9, Axiom JDK 11.0.21 и Axiom JDK 8u392, а также более новые версии JDK поддерживают конфигурации cgroups v1 и cgroups v2. Эта поддержка позволяет обнаружить, что определенные квоты ресурсов устанавливаются при запуске в контейнере, так что эти квоты могут использоваться для операций Java. Ограничения ресурсов влияют на тип сборщика мусора, активируемый JVM, размеры пулов потоков, размер памяти по умолчанию и так далее.

Несмотря на то, что JDK "знает", что он запущен в контейнере, некоторые параметры по умолчанию не подходят для Java-приложений в контейнере. Вот почему имеет смысл дополнительно настроить эти параметры, а именно `-XX:InitialRAMPercentage`, `-XX:MaxRAMPercentage` и `-XX:MinRAMPercentage`.

Эти параметры указаны в процентах, что более предпочтительно, чем установка максимального и минимального размера выделенной памяти для приложения с помощью параметров `-Xmx` и `-Xms` соответственно. Параметры `-XX:RAMPercentage` могут устанавливать размер памяти относительно памяти, выделенной контейнеру с параметрами ограничения, и автоматически обновляться при повторном развертывании.

Другими словами, запуск контейнера с помощью любой из следующих команд делает ненужным переопределение `-Xmx` и `-Xms` в Dockerfile, если они были предоставлены в контейнере. Параметры `-XX:MaxRAMPercentage` и `-XX:MinRAMPercentage`, настроенные в файле Docker, остаются неизменными.

```
podman run --memory=1G -it --rm <URL адрес репозитория>/axiom-runtime-
container-pro:jdk-all-17-glibc java -
XX:MaxRAMPercentage=50 -XX:MinRAMPercentage=50 -XX:+PrintFlagsFinal -version
```

```
| grep MaxHeapSize
   size_t MaxHeapSize                              = 536870912
{product} {ergonomic}
   size_t SoftMaxHeapSize                          = 536870912
{manageable} {ergonomic}
```

```
podman run --memory=2G -it --rm <URL адрес репозитория>/axiom-runtime-
container-pro:jdk-all-17-glibc java -
XX:MaxRAMPercentage=50 -XX:MinRAMPercentage=50 -XX:+PrintFlagsFinal -version
| grep MaxHeapSize
   size_t MaxHeapSize                              = 1067450368
{product} {ergonomic}
   size_t SoftMaxHeapSize                          = 1067450368
{manageable} {ergonomic}
```

Поведение сборщика мусора

Сборка мусора (GC) - важная и неизбежная часть JVM, которая влияет на общую производительность приложения. Очень полезно знать, какой сборщик мусора (GC) используется JVM, запущенной в контейнере. Чтобы проверить тип GC, используйте команду `-Xlog:gc=info`. Например, когда есть ограничение на использование только одного процессора, выбирается последовательный GC. Если активировано более одного процессора и контейнеру выделено достаточно памяти (не менее 2 ГБ), G1 GC выбирается в версии Java, дружественной к контейнеру, например версии 11 или более поздней. Обратите внимание на выбранный GC в зависимости от настроек процессора в следующих примерах:

```
podman run --cpus=1 --memory=2G -it --rm <URL адрес репозитория>/axiom-
runtime-container-pro:jdk-all-17-glibc java -Xlog:gc=info -version
[0.003s][info][gc] Using Serial
openjdk version "17.0.6" 2023-01-17 LTS
OpenJDK Runtime Environment (build 17.0.6+10-LTS)
OpenJDK 64-Bit Server VM (build 17.0.6+10-LTS, mixed mode)
```

```
podman run --cpus=2 --memory=2G -it --rm <URL адрес репозитория>/axiom-
runtime-container-pro:jdk-all-17-glibc java -Xlog:gc=info -version
[0.003s][info][gc] Using G1
openjdk version "17.0.6" 2023-01-17 LTS
OpenJDK Runtime Environment (build 17.0.6+10-LTS)
OpenJDK 64-Bit Server VM (build 17.0.6+10-LTS, mixed mode)
```

Последовательный сборщик мусора (Serial GC) - старейший механизм сбора мусора,

существующий с первых дней Java. Он может подойти для устройств с ограничениями памяти и процессора, но в работе приложения будут возникать длительные паузы, особенно если задействован значительный объем памяти. Если вы хотите, чтобы ваше приложение или сервис имели меньшую задержку отклика и дольше работали без перерывов, выделите больше ресурсов, чтобы использовать преимущества более производительных типов GC. JDK может автоматически включать G1 GC, если ресурсы достигают определенных пределов. В этом случае сборщик мусора G1 имеет более предсказуемое время паузы и достигает более высокой пропускной способности.

Влияние на производительность

Запуск сборщика мусора G1 обеспечивает преимущества в производительности по сравнению с Serial GC. Результаты тестирования могут помочь увидеть, как все это работает. Однопоточный тест подходит для запуска в обеих конфигурациях с `--cpus=1` и `--cpus=2` при условии, что ограничение памяти одинаково. Для простоты мы выбрали проект JMH и соответствующие примеры тестов. Контейнер с JDK 17 (Axiom JDK build 17.0.6+10-LTS) использовался для настройки JMH и готовых тестов.

Сначала мы клонировали репозиторий с проектом, а потом построили образцы тестов следующим образом:

```
cd jhm
mvn clean verify
```

Был создан контейнер с бенчмарками JHM, и мы можем запустить его несколько раз, чтобы собрать необходимые данные для сравнения, выбрав для этой цели один бенчмарк, поскольку полный набор выполнялся бы слишком долго.

Поскольку мы хотим увидеть разницу в однопоточном тестировании с различными типами сборщиков мусора, 2 ГБ памяти были указаны как наименьшее возможное значение для активации G1. Выбор одного или двух процессоров активирует Serial GC или G1 соответственно. Укажем два процессора для активации сборщика мусора G1.

```
podman run --cpus=2 --memory=2G -it --rm 5d412591e12c java -jar
/root/jmh/test/target/benchmarks.jar
org.openjdk.jmh.samples.JMHSample_25_API_GA
```

Укажем один процессор с тем же объемом памяти для активации сборщика мусора Serial GC.

```
podman run --cpus=1 --memory=2G -it --rm 5d412591e12c java -jar
/root/jmh/test/target/benchmarks.jar
org.openjdk.jmh.samples.JMHSample_25_API_GA
```

Кол-во CPU / Тип GC	Результаты
–cpus=1 –memory=2G, SerialGC	646749.890 операций/сек
–cpus=2 –memory=2G, G1GC	694589.369 операций/сек

Результаты тестирования показывают, что G1 работает лучше, чем Serial GC. Назначение двух процессоров даже для однопоточных приложений или служб может привести к повышению производительности. Общая разница в производительности составляет около 4-7%, измеренная несколько раз в разных условиях. Абсолютные цифры могут незначительно отличаться в зависимости от состояния системы и настройки регулятора процессора в данный момент.

Нативный образ

Инструментарий нативных образов Axiom NIK PRO основан на проекте GraalVM с открытым исходным кодом и предоставляет утилиту, способную преобразовать ваше Java-приложение в предварительно (AOT) скомпилированный исполняемый файл, который запускается автономно и почти моментально. GraalVM - это высокопроизводительный JDK, разработанный для ускорения производительности Java-приложений при одновременном потреблении меньшего количества ресурсов. GraalVM предлагает два способа запуска Java-приложений: на HotSpot JVM с помощью компилятора Graal just-in-time (JIT) или в виде собственного предварительно (AOT) скомпилированного исполняемого файла. Axiom NIK PRO поставляется в виде пакета, который может быть установлен на вашем хостинге сборки, поскольку он не нужен во время выполнения.

GraalVM (текущая последняя версия 22.3.2) поддерживает Serial GC и Epsilon GC, которые работают медленнее, даже если контейнеру с приложением или службой назначено больше ресурсов. Предстоящие выпуски GraalVM 23.x будут включать "параллельную" реализацию GC, которую можно включить, используя параметр `--gc=parallel at build time` во время сборки образа, а количество рабочих потоков можно установить во время выполнения, используя параметр `-XX:ParallelGCWorkers`. Рабочие потоки запускаются на ранней стадии при запуске приложения.

Влияние на производительность

Меньшее время пауз влияет на производительность приложения. Если для конкретной службы требуется как можно меньшее время отклика, SerialGC - не ваш выбор.

Новая "параллельная" реализация GC расширяет разнообразие типов GC, доступных в GraalVM, и позволяет уменьшить время отклика.

Соответствующее время пауз было измерено путем компиляции и запуска теста Hyper Alloc. Цифры на приведенной ниже диаграмме представляют время паузы GC в миллисекундах. Тест был выполнен на Ubuntu, 8-ядерном процессоре i7 с 8 рабочими потоками и отключенным инкрементным сбором.

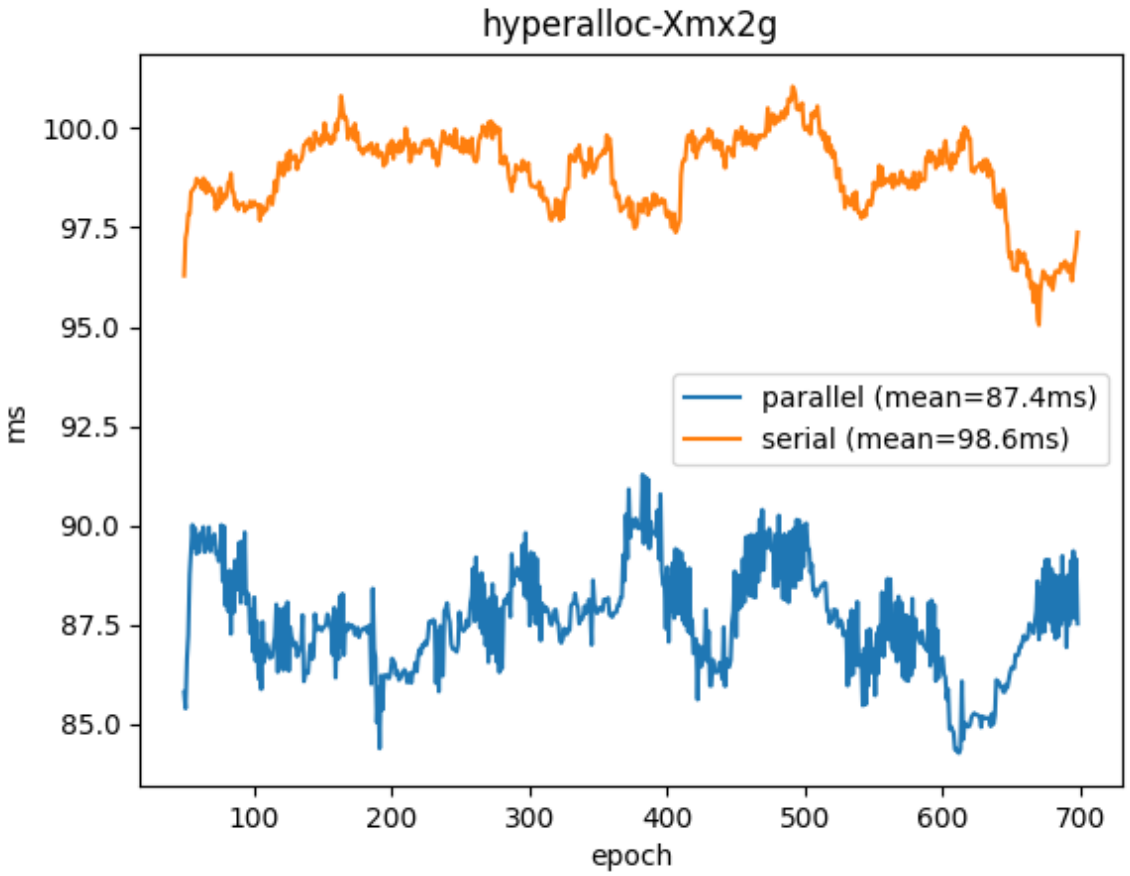

График и рассчитанные средние значения показывают, что "параллельная" реализация GC обеспечивает меньшее время пауз, тем самым повышая производительность и уменьшая время ожидания для приложений и сервисов, запущенных в аналогичных конфигурациях с двумя или более доступными потоками или процессорами.

4. Вывод

Контейнеризованные приложения и службы не обязательно должны всегда выполняться с минимальными ресурсами, выделенными контейнеру, за исключением случаев, когда это строго требуется. Во многих случаях приложения или службы Java работают лучше и имеют лучшее время отклика, если для запущенного контейнера выделяется немного больше ресурсов, так что в конечном итоге используется более подходящий тип GC. Мы рекомендуем установить ограничения как минимум на 2 процессора и 2 ГБ памяти для запущенного контейнера с рабочими нагрузками Java. В некоторых случаях вы можете захотеть выбрать Нативный образ, чтобы ускорить запуск и снизить потребление памяти во время выполнения.

AXIOM JDK

Axiom Linux
Руководство по настройке JDK для контейнеров с ограниченными ресурсами