容器资源管理

> 这是本专栏的第二部分：容器篇，共 6 篇，帮助大家由浅入深的认识和掌握容器。前两篇，我为你介绍了容器生命周期管理相关的内容，带你掌握容器生命周期。本篇我将为你介绍容器资源管理相关的内容。

前两篇我已经为你介绍了容器生命周期管理相关的内容，本篇的主题是容器资源管理。我们带着以下三个问题开始本篇的内容：

- 哪些分配给容器的资源可被我们管理
- 容器实际使用了多少资源
- 如何对容器使用的资源进行管理

资源类型

对于第一个问题，当我们启动一个容器的时候，它可以使用一些系统资源，这与我们在物理机上启动程序基本是一致的。比如主要的几类：

- CPU
- 内存
- 网络
- I/O
- GPU

这些系统资源是在我们启动容器时，需要考虑和可被我们管理的。比如，我们可以执行 `docker run --help` 查看 `docker run` 命令所支持的全部参数。现在 `docker run` 命令所支持的参数已超过 90 项，这里就不一一列出了。

查看容器占用资源

docker stats

Docker 提供了一个很方便的命令 `docker stats`，可供我们查看和统计容器所占用的资源情况。

我们仍然启动一个 Redis 容器作为示例。

复制

```
# 启动一个容器
(MoeLove) → ~ docker run -d redis
c98c9831ee73e9b71719b404f5ecf3b408de0b69aec0f781e42d815575d28ada
# 查看其所占用资源的情况
(MoeLove) → ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
c98c9831ee73        amazing_torvalds    0.08%               2.613MiB / 15.56GiB
```

这里传递了一个 `--no-stream` 的参数，是因为 `docker stats` 命令默认是一个持续的动态流式输出（每秒一次），给它传递 `--no-stream` 参数后，它就只输出一次便会退出了。

接下来我为你介绍下它输出内容的含义：

- Container ID：容器的 ID，也是一个容器生命周期内不会变更的信息。
- Name：容器的名称，如果没有手动使用 `--name` 参数指定，则 Docker 会随机生成一个，运行过程中也可以通过命令修改。
- **CPU %：容器正在使用的 CPU 资源的百分比**，这里面涉及了比较多细节，下面会详细说。
- **Mem Usage/Limit：当前内存的使用及容器可用的最大内存**，这里我使用了一台 16G 的电脑进行测试。
- **Mem %：容器正在使用的内存资源的百分比。**
- Net I/O：容器通过其网络接口发送和接受到的数据量。
- Block I/O：容器通过块设备读取和写入的数据量。
- Pids：容器创建的进程或线程数。

docker top

除了上面提到的 `docker stats` 命令外，Docker 也提供了另一个比较简单的命令 `docker top`，与我们平时用的 `ps` 命令基本一致，也支持 `ps` 命令的参数。

复制

```
(MoeLove) → ~ docker top $(docker ps -ql)
UID                 PID                 PPID                C                   STI
systemd+            6275                6248                0                   16:
# 可以使用 ps 命令的参数
(MoeLove) → ~ docker top $(docker ps -ql)  -o pid,stat,cmd
PID                 STAT                CMD
6275                Ssl                 redis-server *:6379
```

管理容器的 CPU 资源

在我们使用容器的时候，CPU 和内存是我们尤为关注的资源。不过，对于 CPU 资源的管理，涉及的内容会比较偏底层一些，有些涉及到了内核的 CPU 调度器，比如 CFS（Completely Fair Scheduler）等。

我们可以先来查看下 Docker 提供了哪些控制 CPU 资源相关的参数。 使用 `docker run --help |grep CPU` 即可查看。

复制

```
(MoeLove) ➜  ~ docker run --help |grep CPU
      --cpu-period int                 Limit CPU CFS (Completely Fair Scheduler)
      --cpu-quota int                  Limit CPU CFS (Completely Fair Scheduler)
      --cpu-rt-period int              Limit CPU real-time period in microseconds
      --cpu-rt-runtime int             Limit CPU real-time runtime in microsecond
  -c, --cpu-shares int                 CPU shares (relative weight)
      --cpus decimal                    Number of CPUs
      --cpuset-cpus string             CPUs in which to allow execution (0-3, 0,
```

这里暂时先不对参数的具体含义进行深入展开，我们直接以几个示例来分别进行说明，帮助大家理解。

默认无限制

> 备注：我这里以一个 4 核 CPU 的电脑进行演示。

现在我们启动一个容器，我们以体积很小的 Alpine Linux 为例好了。

复制

```
(MoeLove) ➜  ~ docker run --rm -it alpine
/ #
```

在另一个窗口，执行上面介绍的查看容器资源的命令：

复制

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
106a24399bc9        friendly_varahamihira   0.00%               1.047MiB / 15.56Gi
```

可以看到，当前容器内没有过多的 CPU 消耗，且 PIDS 为 1，表示当前只有一个进程。

现在我们回到刚才启动的容器，执行以下命令：

复制

```
sha256sum /dev/zero
```

- sha256sum 是一个用于计算和检查 SHA256 信息的命令行工具；
- /dev/zero 是 Linux 系统上一个特殊的设备，在读它时，它可以提供无限的空字符串（NULL 或者 0x00 之类的）。

所以上面的命令，会**让 sha256sum 持续地读 /dev/zero 产生的空串，并进行计算。**这将迅速地消耗 CPU 资源。

我们来看看此时容器的资源使用情况：

复制

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
106a24399bc9        friendly_varahamihira   100.59%             1.5MiB / 15.56GiB
(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
965                 99                  sha256sum /dev/zero
```

可以看到当前的 CPU 使用率已经在 100% 左右了。

我们再新打开一个窗口，进入容器内，执行相同的命令：

复制

```
(MoeLove) ➜  ~ docker exec -it $(docker ps -ql) sh
/ # sha256sum /dev/zero
```

查看容器使用资源的情况：

复制

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
f359d4ff6fc6        nice_zhukovsky      200.79%             1.793MiB / 15.56GiB
(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
965                 99                  sha256sum /dev/zero
1236                0                   sh
1297                99                  sha256sum /dev/zero
```

可以看到现在两个进程，已经让两个 CPU 满负载运行了。这里需要额外说明的是，选择 sha256sum 作为示例，是因为它是单线程程序，每次启动一个 sha256sum 并不会消耗其他 CPU 核的资源。

由此可以得出的结论是，**如果不对容器内程序进行 CPU 资源限制，其可能会消耗掉大量 CPU 资源，进而影响其他程序或者影响系统的稳定。**

分配 0.5 CPU

那接下来，我们对这个容器进行 CPU 资源的限制，比如限制它只可以使用 0.5 CPU。

我们可以重新启动一个容器，在 `docker run` 时，为它添加资源限制。

但我来给你介绍一种**动态**更改资源限制的办法，使用 `docker update` 命令。例如，在此例子中，我们使用如下命令，限制该容器只能使用 0.5 CPU。

```
(MoeLove) ➜  ~ docker update --cpus "0.5" $(docker ps -ql)
f359d4ff6fc6
```

为了方便，我们直接关闭刚才的 sha256sum 进程，按 Ctrl+c 终止进程。然后重新运行该命令：

```
# 终止进程
/ # sha256sum /dev/zero
^C
# 启动程序
/ # sha256sum /dev/zero
```

查看资源占用情况：

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
f359d4ff6fc6        nice_zhukovsky      49.87%              1.777MiB / 15.56GiB

(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
1236                0                   sh
7662                49                  sha256sum /dev/zero
```

可以看到，该进程使用了 50% 左右的 CPU。我们接下来再启动另一个 sha256sum 的进程：

```
/ # sha256sum /dev/zero
```

查看资源使用情况：

复制

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
f359d4ff6fc6        nice_zhukovsky      50.92%              1.891MiB / 15.56GiB

(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
1236                0                   sh
10106               25                  sha256sum /dev/zero
11999               25                  sha256sum /dev/zero
```

可以看到，该容器整体占用了 50% 的 CPU，而其中的两个 sha256sum 进程则各占了 25%。

我们已经成功的按预期为它分配了 0.5 CPU。

分配 1.5 CPU

接下来，重复上述步骤，但是为它分配 1.5 CPU，来看看它的实际情况如何。

复制

```
# 更新配置，使用 1.5 CPU
(MoeLove) ➜  ~ docker update --cpus "1.5" $(docker ps -ql)
f359d4ff6fc6
```

分别使用之前的两个窗口，执行 `sha256sum /dev/zero` 进行测试：

复制

```
/ # sha256sum /dev/zero
```

查看资源使用情况：

复制

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT   ME
f359d4ff6fc6        nice_zhukovsky      151.23%             2MiB / 15.56GiB     0.

(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
1236                0                   sh
25167               77                  sha256sum /dev/zero
25211               74                  sha256sum /dev/zero
```

可以看到，结果与我们的预期基本相符，150% 左右的 CPU，而两个测试进程，也差不多是平分了 CPU 资源。

指定可使用 CPU 核

可以使用 `--cpuset-cpus` 来指定分配可使用的 CPU 核，这里我指定为 0，表示使用第一个 CPU 核。

```
(MoeLove) ➜  ~ docker update --cpus "1.5" --cpuset-cpus 0  $(docker ps -ql)
f359d4ff6fc6
```

分别使用之前的两个窗口，执行 `sha256sum /dev/zero` 进行测试：

```
/ # sha256sum /dev/zero
```

查看资源情况：

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
f359d4ff6fc6        nice_zhukovsky      99.18%              1.988MiB / 15.56GiB
(MoeLove) ➜  ~ docker top $(docker ps -ql) -o pid,c,cmd
PID                 C                   CMD
825                 0                   /bin/sh
1236                0                   sh
25119               50                  sha256sum /dev/zero
25164               48                  sha256sum /dev/zero
```

可以看到，虽然我们仍然使用 `--cpus` 指定了 1.5 CPU，但由于使用 `--cpuset-cpus` 限制只允许它跑在第一个 CPU 上，所以这两个测试进程也就只能评分该 CPU 了。

小结

通过上述的示例，我为你介绍了如何通过 `--cpus` 参数限制容器可使用的 CPU 资源；通过 `--cpuset-cpus` 参数可指定容器内进程运行所用的 CPU 核心；通过 `docker update` 可直接更新一个正在运行的容器的相关配置。

现在我们回到前面使用 `docker run --help | grep CPU`，查看 Docker 支持的对容器 CPU 相关参数的选项：

复制

```
(MoeLove) ➜  ~ docker run --help |grep CPU
      --cpu-period int                   Limit CPU CFS (Completely Fair Scheduler)
      --cpu-quota int                    Limit CPU CFS (Completely Fair Scheduler)
      --cpu-rt-period int                Limit CPU real-time period in microseconds
      --cpu-rt-runtime int               Limit CPU real-time runtime in microsecond
  -c, --cpu-shares int                   CPU shares (relative weight)
      --cpus decimal                     Number of CPUs
      --cpuset-cpus string               CPUs in which to allow execution (0-3, 0,
```

`--cpus` 是在 Docker 1.13 时新增的，可用于替代原先的 `--cpu-period` 和 `--cpu-quota`。这三个参数通过 cgroups 最终会实际影响 Linux 内核的 CPU 调度器 CFS（Completely Fair Scheduler, 完全公平调度算法）对进程的调度结果。

一般情况下，**推荐直接使用 `--cpus`，而无需单独设置 `--cpu-period` 和 `--cpu-quota`**，除非你已经对 CPU 调度器 CFS 有了足够多的了解，提供 `--cpus` 参数也是 Docker 团队为了可以简化用户的使用成本增加的，它足够满足我们大多数的需求。

而 `--cpu-shares` 选项，它虽然有一些实际意义，但却不如 `--cpus` 来的直观，并且它会受到当前系统上运行状态的影响，为了不因为它给大家带来困扰，此处就不再进行介绍了。

`--cpu-rt-period` 和 `--cpu-rt-runtime` 两个参数，会影响 CPU 的实时调度器。但实时调度器需要内核的参数的支持，并且配置实时调度器也是个高级或者说是危险的操作，有可能会导致各种奇怪的问题，此处也不再进行展开。

管理容器的内存资源

前面已经介绍了如何管理容器的 CPU 资源，接下来我们看看如何管理容器的内存资源。相比 CPU 资源来说，内存资源的管理就简单很多了。

同样的，我们先看看有哪些参数可供我们配置，对于其含义我会稍后进行介绍：

复制

```
(MoeLove) ➜  ~ docker run --help |egrep 'memory|oom'
      --kernel-memory bytes              Kernel memory limit
  -m, --memory bytes                     Memory limit
      --memory-reservation bytes         Memory soft limit
      --memory-swap bytes                Swap limit equal to memory plus swap: '-1'
      --memory-swappiness int            Tune container memory swappiness (0 to 100)
      --oom-kill-disable                 Disable OOM Killer
      --oom-score-adj int                Tune host's OOM preferences (-1000 to 1000)
```

OOM

在开始进行容器内存管理的内容前，我们不妨先聊一个很常见，又不得不面对的问题：**OOM**（Out Of Memory）。

当内核检测到没有足够的内存来运行系统的某些功能时候，就会触发 OOM 异常，并且会使用 OOM Killer 来杀掉一些进程，腾出空间以保障系统的正常运行。

这里简单介绍下 OOM killer 的大致执行过程，以便于大家理解后续内容。

内核中 OOM Killer 的代码，在 torvalds/linux/mm/oom_kill.c 可直接看到，这里以 Linux Kernel 5.2 为例。

引用其中的一段注释：

> If we run out of memory, we have the choice between either killing a random task (bad), letting the system crash (worse).
>
> OR try to be smart about which process to kill. Note that we don't have to be perfect here, we just have to be good.

翻译过来就是，当我们处于 OOM 时，我们可以有几种选择，随机地杀死任意的任务（不好），让系统崩溃（更糟糕）或者尝试去了解可以杀死哪个进程。注意，这里我们不需要追求完美，我们只需要变好（be good）就行了。

事实上确实如此，无论随机地杀掉任意进程或是让系统崩溃，那都不是我们想要的。

回到内核代码中，当系统内存不足时，out_of_memory() 被触发，之后会调用 select_bad_process() 函数，选择一个 bad 进程来杀掉。

那什么样的进程是 bad 进程呢？总是有些条件的。select_bad_process() 是一个简单的循环，其调用了 oom_evaluate_task() 来对进程进行条件计算，最核心的判断逻辑是其中的 oom_badness()。

复制

```
unsigned long oom_badness(struct task_struct *p, struct mem_cgroup *memcg,
              const nodemask_t *nodemask, unsigned long totalpages)
{
    long points;
    long adj;

    if (oom_unkillable_task(p, memcg, nodemask))
        return 0;

    p = find_lock_task_mm(p);
    if (!p)
        return 0;

    /*
     * Do not even consider tasks which are explicitly marked oom
     * unkillable or have been already oom reaped or the are in
     * the middle of vfork
     */
    adj = (long)p->signal->oom_score_adj;
    if (adj == OOM_SCORE_ADJ_MIN ||
            test_bit(MMF_OOM_SKIP, &p->mm->flags) ||
            in_vfork(p)) {
        task_unlock(p);
        return 0;
    }

    /*
     * The baseline for the badness score is the proportion of RAM that each
     * task's rss, pagetable and swap space use.
     */
    points = get_mm_rss(p->mm) + get_mm_counter(p->mm, MM_SWAPENTS) +
        mm_pgtables_bytes(p->mm) / PAGE_SIZE;
    task_unlock(p);

    /* Normalize to oom_score_adj units */
    adj *= totalpages / 1000;
    points += adj;

    /*
     * Never return 0 for an eligible task regardless of the root bonus and
     * oom_score_adj (oom_score_adj can't be OOM_SCORE_ADJ_MIN here).
     */
    return points > 0 ? points : 1;
}
```

而为了能够最快地进行选择，这里的逻辑也是尽可能的简单，**除了明确标记不可杀掉的进程外，直接选择内存占用最多的进程。**（当然，还有一个额外的 oom_score_adj 可用于控制权重）

这种选择的最主要的两个好处是：

1. 可以回收很多内存；
2. 可以避免缓解 OOM 后，该进程后续对内存的抢占引发后续再次的 OOM。

我们将注意力再回到 Docker 自身，在生产环境中，我们通常会用 Docker 启动多个容器运行服务。当遇到 OOM 时，如果 Docker 进程被杀掉，那对我们的服务也会带来很大的影响。

所以 Docker 在启动的时候默认设置了一个 -500 的 oom_score_adj 以尽可能地避免 Docker 进程本身被 OOM Killer 给杀掉。

如果我们想让某个容器，尽可能地不要被 OOM Killer 杀掉，那我们可以给它传递 `--oom-score-adj` 配置一个比较低的数值。

但是注意：**不要通过** `--oom-kill-disable` 禁用掉 OOM Killer，或者给容器设置低于 dockerd 进程的 oom_score_adj 值，这可能会导致某些情况下系统的不稳定。除非你明确知道自己的操作将会带来的影响。

管理容器的内存资源

介绍完了 OOM，相比你已经知道了内存耗尽所带来的危害，我们来继续介绍如何管理容器的内存资源。

复制

```
(MoeLove) ➜  ~ docker run --help |grep 'memory'
      --kernel-memory bytes            Kernel memory limit
  -m, --memory bytes                   Memory limit
      --memory-reservation bytes       Memory soft limit
      --memory-swap bytes              Swap limit equal to memory plus swap: '-1'
      --memory-swappiness int          Tune container memory swappiness (0 to 100)
```

可用的配置参数有上述几个，我们通常直接使用 `--memory` 参数来限制容器可用的内存大小。我们同样使用几个示例进行介绍：

启动一个容器，并传递参数 `--memory 10m` 限制其可使用的内存为 10 m。

复制

```
(MoeLove) ➜  ~ docker run --rm -it --memory 10m alpine
/ #
```

那我们如何验证它的可用内存大小是多少呢？在物理机上，我们通常使用 free 工具进行查看。但在容器环境内，它还是否生效呢？

```
/ # free -m
             total       used       free     shared    buffers     cached
Mem:         15932      14491       1441       1814        564       3632
-/+ buffers/cache:      10294       5637
Swap:         8471        693       7778
```

复制

很明显，使用 free 得到的结果是宿主机上的信息。当然，我们前面已经介绍了 `docker stats` 命令，我们使用它来查看当前的资源使用情况：

```
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT   ME
e260e91874d8        busy_napier         0.00%               1.172MiB / 10MiB    11
       0B / 0B             1
```

复制

可以看到 `MEM USAGE / LIMIT` 那一列中的信息已经生效，是我们预期的样子。

那我们是否还有其他方式查看此信息呢？当然有：

```
# 在容器内执行
/ # cat /sys/fs/cgroup/memory/memory.limit_in_bytes
10485760
```

复制

或者可以在宿主机上执行以下命令：

```
(MoeLove) ➜  ~ cat  /sys/fs/cgroup/memory/system.slice/docker-$(docker inspect --
10485760
```

复制

注意：以上命令在 Linux 5.2 内核下测试通过，不同版本之间目录结构略有差异。

更新容器内存资源限制

当容器运行一段时间，其中的进程使用的内存变多了，我们想允许容器使用更多内存资源，那要如何操作呢？

我们仍然可以用前面介绍的 `docker update` 命令完成。

比如使用如下命令，将可用内存扩大至 20m：

复制

```
(MoeLove) ➜  ~ docker update --memory 20m $(docker ps -ql)
e260e91874d8
# 验证是否生效
(MoeLove) ➜  ~ docker stats --no-stream $(docker ps -ql)
CONTAINER ID        NAME                CPU %               MEM USAGE / LIMIT
e260e91874d8        busy_napier         0.00%               1.434MiB / 20MiB    7.
```

如果还不够，需要扩大至 100m 呢?

复制

```
(MoeLove) ➜  ~ docker update --memory 100m $(docker ps -ql)
Error response from daemon: Cannot update container e260e91874d8181b6d0078c8534
```

会发现这里有个报错信息。大意是 memory limit 应该比已经配置的 memoryswap limit 小，需要同时更新 memoryswap。

你可能会困惑，之前我们只是限制了内存为 10m，并且扩大至 20m 的时候是成功了的。为什么到 100m 的时候就会出错?

这就涉及到了这些参数的特定行为了，我来为你一一介绍。

内存限制参数的特定行为

这里的特定参数行为，主要是指我们前面使用的 `--memory` 和未介绍过的 `--memory-swap` 这两个参数。

1. `--memory` 用于限制内存使用量，而 `--memory-swap` 则表示内存和 Swap 的总和。

这解释了上面“Memory limit should be smaller than already set memoryswap limit”，因为 `--memory-swap` 始终应该大于等于 `--memory` （毕竟 Swap 最小也只能是 0 ）。

2. 如果只指定了 `--memory` 则最终 `--memory-swap` 将会设置为 `--memory` 的两倍。也就是说，在只传递 `--memory` 的情况下，容器只能使用与 `--memory` 相同大小的 Swap。

这也解释了上面“直接扩大至 20m 的时候能成功，而扩大到 100m 的时候会出错”，在上述场景中只指定了 `--memory` 为 10m，所以 `--memory-swap` 就默认被设置成了 20m。

3. 如果 `--memory-swap` 和 `--memory` 设置了相同值，则表示不使用 Swap。

4. 如果 `--memory-swap` 设置为 -1 则表示不对容器使用的 Swap 进行限制。

5. 如果设置了 `--memory-swap` 参数，则必须设置 `--memory` 参数。

总结

至此，我便为你介绍了容器资源管理的核心内容，包括管理容器的 CPU 资源和内存资源。为容器进行合理的资源控制，有利于提高整体环境的稳定性，避免资源抢占或大量内存占用导致 OOM，进程被杀掉等情况。

对 CPU 进行管理时，建议使用 `--cpus`，语义方面会比较清晰。如果是对 Linux 的 CPU 调度器 CFS 很熟悉，并且有强烈的定制化需求，这种情况下再使用 `--cpu-period` 和 `--cpu-quota` 比较合适。

对内存进行管理时，有个 `--memory-swappiness` 参数也需要注意下，它可设置为 0~100 的百分比，与我们平时见到的 swappiness 行为基本一致，设置为 0 表示不使用匿名页面交换，设置为 100 则表示匿名页面都可被交换。如果不指定的话，它默认会从主机上继承。

在本篇中，关于在宿主机上查看容器的内存限制，我给出了一个命令：

复制

```
(MoeLove) ➜  ~ cat  /sys/fs/cgroup/memory/system.slice/docker-$(docker inspect --
10485760
```

它具体是什么含义呢？下篇《深入剖析容器》中我将详细说明。