

46 | 案例篇：为什么应用容器化后，启动慢了很多？

倪朋飞 2019-03-11



00:00

讲述：冯永吉

大小：12.98M

14:10

你好，我是倪朋飞。

不知不觉，我们已经学完了整个专栏的四大基础模块，即 CPU、内存、文件系统和磁盘 I/O、以及网络的性能分析和优化。相信你已经掌握了这些基础模块的基本分析、定位思路，并熟悉了相关的优化方法。

接下来，我们将进入最后一个重要模块——综合实战篇。这部分实战内容，也将是我们对前面所学知识复习和深化。

我们都知道，随着 Kubernetes、Docker 等技术的普及，越来越多的企业，都已经走上了应用程序容器化的道路。我相信，你在了解学习这些技术的同时，一定也听说过不少，基于 Docker 的微服务架构带来的各种优势，比如：

使用 Docker，把应用程序以及相关依赖打包到镜像中后，部署和升级更快捷；

把传统的单体应用拆分成多个更小的微服务应用后，每个微服务的功能都更简单，并且可以单独管理和维护；

每个微服务都可以根据需求横向扩展。即使发生故障，也只是局部服务不可用，而不像以前那样，导致整个服务不可用。

不过，任何技术都不是银弹。这些新技术，在带来诸多便捷功能之外，也带来了更高的复杂性，比如性能降低、架构复杂、排错困难等等。

今天，我就通过一个 Tomcat 案例，带你一起学习，如何分析应用程序容器化后的性能问题。

案例准备

今天的案例，我们只需要一台虚拟机。还是基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

机器配置：2 CPU，8GB 内存。

预先安装 docker、curl、jq、pidstat 等工具，如 `apt install docker.io curl jq sysstat`。

其中，jq 工具专门用来在命令行中处理 json。为了更好的展示 json 数据，我们用这个工具，来格式化 json 输出。

你需要打开两个终端，登录到同一台虚拟机中，并安装上述工具。

注意，以下所有命令都默认以 root 用户运行，如果你用普通用户身份登陆系统，请运行 `sudo su root` 命令切换到 root 用户。


如果安装过程有问题，你可以先上网搜索解决，实在解决不了的，记得在留言区向我提问。

到这里，准备工作就完成了。接下来，我们正式进入操作环节。

案例分析

我们今天要分析的案例，是一个 Tomcat 应用。Tomcat 是 Apache 基金会旗下，Jakarta 项目开发的轻量级应用服务器，它基于 Java 语言开发。Docker 社区也维护着 Tomcat 的[官方镜像](#)，你可以直接使用这个镜像，来启动一个 Tomcat 应用。

我们的案例，也基于 Tomcat 的官方镜像构建，其核心逻辑很简单，就是分配一点儿内存，并输出 “Hello, world!” 。

 复制代码

```
1 <%
2 byte data[] = new byte[256*1024*1024];
3 out.println("Hello, world!");
4 %>
5
```

为了方便你运行，我已经将它打包成了一个 [Docker 镜像](#) feisky/tomcat:8，并推送到了 Docker Hub 中。你可以直接按照下面的步骤来运行它。

在终端一中，执行下面的命令，启动 Tomcat 应用，并监听 8080 端口。如果一切正常，你应该可以看到如下的输出：

[复制代码](#)

```
1 # -m 表示设置内存为 512MB
2 $ docker run --name tomcat --cpus 0.1 -m 512M -p 8080:8080 -itd feisky/tomcat:8
3 Unable to find image 'feisky/tomcat:8' locally
4 8: Pulling from feisky/tomcat
5 741437d97401: Pull complete
6 ...
7 22cd96a25579: Pull complete
8 Digest: sha256:71871cff17b9043842c2ec99f370cc9f1de7bc121cd2c02d8e2092c6e268f7e2
9 Status: Downloaded newer image for feisky/tomcat:8
10 WARNING: Your kernel does not support swap limit capabilities or the cgroup is not mount
11 2df259b752db334d96da26f19166d662a82283057411f6332f3cbdbcab452249
12
```

从输出中，你可以看到，docker run 命令，会自动拉取镜像并启动容器。

这里顺便提一下，之前很多同学留言问，到底要怎么下载 Docker 镜像。其实，上面的 docker run，就是自动下载镜像到本地后，才开始运行的。

由于 Docker 镜像分多层管理，所以在下载时，你会看到每层的下载进度。除了像 docker run 这样自动下载镜像外，你也可以分两步走，先下载镜像，然后再运行容器。

比如，你可以先运行下面的 docker pull 命令，下载镜像：

[复制代码](#)

```
1 $ docker pull feisky/tomcat:8
2 8: Pulling from feisky/tomcat
3 Digest: sha256:71871cff17b9043842c2ec99f370cc9f1de7bc121cd2c02d8e2092c6e268f7e2
4 Status: Image is up to date for feisky/tomcat:8
5
```

显然，在我的机器中，镜像已存在，所以就不需要再次下载，直接返回成功就可以了。


接着，在终端二中使用 curl，访问 Tomcat 监听的 8080 端口，确认案例已经正常启动：

[复制代码](#)

```
1 $ curl localhost:8080
2 curl: (56) Recv failure: Connection reset by peer
3
```

不过，很不幸，curl 返回了 “Connection reset by peer” 的错误，说明 Tomcat 服务，并不能正常响应客户端请求。


是不是 Tomcat 启动出问题了？我们切换到终端一中，执行 `docker logs` 命令，查看容器的日志。这里注意，需要加上 `-f` 参数，表示跟踪容器的最新日志输出：

 复制代码

```
1 $ docker logs -f tomcat
2 Using CATALINA_BASE:   /usr/local/tomcat
3 Using CATALINA_HOME:   /usr/local/tomcat
4 Using CATALINA_TMPDIR: /usr/local/tomcat/temp
5 Using JRE_HOME:        /docker-java-home/jre
6 Using CLASSPATH:        /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat
7
```

从这儿你可以看到，Tomcat 容器只打印了环境变量，还没有应用程序初始化的日志。也就是说，Tomcat 还在启动过程中，这时候去访问它，当然没有响应。

为了观察 Tomcat 的启动过程，我们在终端一中，继续保留 `docker logs -f` 命令，并在终端二中执行下面的命令，多次尝试访问 Tomcat：


 复制代码

```
1 $ for ((i=0;i<30;i++)); do curl localhost:8080; sleep 1; done
2 curl: (56) Recv failure: Connection reset by peer
3 curl: (56) Recv failure: Connection reset by peer
4 # 这儿会阻塞一会
5 Hello, wolrd!
6 curl: (52) Empty reply from server
7 curl: (7) Failed to connect to localhost port 8080: Connection refused
8 curl: (7) Failed to connect to localhost port 8080: Connection refused
9
```

观察一会儿，可以看到，一段时间后，curl 终于给出了我们想要的结果 “Hello, wolrd!”。但是，随后又出现了 “Empty reply from server”，和一直持续的 “Connection refused” 错误。换句话说，Tomcat 响应一次请求后，就再也不响应了。

这是怎么回事呢？我们回到终端一中，观察 Tomcat 的日志，看看能不能找到什么线索。

从终端一中，你应该可以看到下面的输出：

 复制代码

```
1 18-Feb-2019 12:43:32.719 INFO [localhost-startStop-1] org.apache.catalina.startup.HostC
2 18-Feb-2019 12:43:33.725 INFO [localhost-startStop-1] org.apache.catalina.startup.HostC
3 18-Feb-2019 12:43:33.726 INFO [localhost-startStop-1] org.apache.catalina.startup.HostC
4 18-Feb-2019 12:43:34.521 INFO [localhost-startStop-1] org.apache.catalina.startup.HostC
5 18-Feb-2019 12:43:34.722 INFO [main] org.apache.coyote.AbstractProtocol.start Starting f
6 18-Feb-2019 12:43:35.319 INFO [main] org.apache.coyote.AbstractProtocol.start Starting f
7 18-Feb-2019 12:43:35.821 INFO [main] org.apache.catalina.startup.Catalina.start Server !
8 root@ubuntu:~#
9
```

从内容上可以看到，Tomcat 在启动 24s 后完成初始化，并且正常启动。从日志上来看，没有什么问题。

不过，细心的你肯定注意到了最后一行，明显是回到了 Linux 的 SHELL 终端中，而没有继续等待 Docker 输出的容器日志。

输出重新回到 SHELL 终端，通常表示上一个命令已经结束。而我们的上一个命令，是 `docker logs -f` 命令。那么，它的退出就只有两种可能了，要么是容器退出了，要么就是 `dockerd` 进程退出了。

究竟是哪种情况呢？这就需要我们进一步确认了。我们可以在终端一中，执行下面的命令，查看容器的状态：

复制代码

```
1 $ docker ps -a
2 CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
3 0f2b3fcdd257        feisky/tomcat:8    "catalina.sh run"   2 minutes ago       Exited
4
```

你会看到，容器处于 `Exited` 状态，说明是第一种情况，容器已经退出。不过为什么会这样呢？显然，在前面容器的日志里，我们并没有发现线索，那就只能从 Docker 本身入手了。

我们可以调用 Docker 的 API，查询容器的状态、退出码以及错误信息，然后确定容器退出的原因。这些可以通过 `docker inspect` 命令来完成，比如，你可以继续执行下面的命令，通过 `-f` 选项设置只输出容器的状态：

复制代码


```
1 # 显示容器状态，jq 用来格式化 json 输出
2 $ docker inspect tomcat -f '{{json .State}}' | jq
3 {
4   "Status": "exited",
5   "Running": false,
6   "Paused": false,
7   "Restarting": false,
8   "OOMKilled": true,
9   "Dead": false,
10  "Pid": 0,
11  "ExitCode": 137,
12  "Error": "",
13  ...
14 }
15
```

这次你可以看到，容器已经处于 `exited` 状态，`OOMKilled` 是 `true`，`ExitCode` 是 137。这其中，`OOMKilled` 表示容器被 OOM 杀死了。

我们前面提到过，OOM 表示内存不足时，某些应用会被系统杀死。可是，为什么内存会不足呢？我们的应用分配了 256 MB 的内存，而容器启动时，明明通过 `-m` 选项，设置了 512 MB 的

内存，按说应该是足够的。

到这里，我估计你应该还记得，当 OOM 发生时，系统会把相关的 OOM 信息，记录到日志中。所以，接下来，我们可以在终端中执行 `dmesg` 命令，查看系统日志，并定位 OOM 相关的日志：

 复制代码

```
1 $ dmesg
2 [193038.106393] java invoked oom-killer: gfp_mask=0x14000c0(GFP_KERNEL), nodemask=(null)
3 [193038.106396] java cpuset=0f2b3fcdd2578165ea77266cdc7b1ad43e75877b0ac1889ecda30a78cb7f
4 [193038.106402] CPU: 0 PID: 27424 Comm: java Tainted: G OE 4.15.0-1037 #39-Ubuntu
5 [193038.106404] Hardware name: Microsoft Corporation Virtual Machine/Virtual Machine, B
6 [193038.106405] Call Trace:
7 [193038.106414] dump_stack+0x63/0x89
8 [193038.106419] dump_header+0x71/0x285
9 [193038.106422] oom_kill_process+0x220/0x440
10 [193038.106424] out_of_memory+0x2d1/0x4f0
11 [193038.106429] mem_cgroup_out_of_memory+0x4b/0x80
12 [193038.106432] mem_cgroup_oom_synchronize+0x2e8/0x320
13 [193038.106435] ? mem_cgroup_css_online+0x40/0x40
14 [193038.106437] pagefault_out_of_memory+0x36/0x7b
15 [193038.106443] mm_fault_error+0x90/0x180
16 [193038.106445] __do_page_fault+0x4a5/0x4d0
17 [193038.106448] do_page_fault+0x2e/0xe0
18 [193038.106454] ? page_fault+0x2f/0x50
19 [193038.106456] page_fault+0x45/0x50
20 [193038.106459] RIP: 0033:0x7fa053e5a20d
21 [193038.106460] RSP: 002b:00007fa0060159e8 EFLAGS: 00010206
22 [193038.106462] RAX: 0000000000000000 RBX: 00007fa04c4b3000 RCX: 0000000009187440
23 [193038.106463] RDX: 00000000943aa440 RSI: 0000000000000000 RDI: 000000009b223000
24 [193038.106464] RBP: 00007fa006015a60 R08: 0000000002000002 R09: 00007fa053d0a8a1
25 [193038.106465] R10: 00007fa04c018b80 R11: 0000000000000206 R12: 0000000100000768
26 [193038.106466] R13: 00007fa04c4b3000 R14: 0000000100000768 R15: 0000000100000000
27 [193038.106468] Task in /docker/0f2b3fcdd2578165ea77266cdc7b1ad43e75877b0ac1889ecda30a7f
28 [193038.106478] memory: usage 524288kB, limit 524288kB, failcnt 77
29 [193038.106480] memory+swap: usage 0kB, limit 9007199254740988kB, failcnt 0
30 [193038.106481] kmem: usage 3708kB, limit 9007199254740988kB, failcnt 0
31 [193038.106481] Memory cgroup stats for /docker/0f2b3fcdd2578165ea77266cdc7b1ad43e75877f
32 [193038.106494] [ pid ] uid tgid total_vm rss pgtables_bytes swapents oom_score_
33 [193038.106571] [27281] 0 27281 1153302 134371 1466368 0 0 j
34 [193038.106574] Memory cgroup out of memory: Kill process 27281 (java) score 1027 or sa
35 [193038.148334] Killed process 27281 (java) total-vm:4613208kB, anon-rss:517316kB, file
36 [193039.607503] oom_reaper: reaped process 27281 (java), now anon-rss:0kB, file-rss:0kB.
37
```

从 `dmesg` 的输出，你就可以看到很详细的 OOM 记录了。你应该可以看到下面几个关键点。

第一，被杀死的是一个 java 进程。从内核调用栈上的 `mem_cgroup_out_of_memory` 可以看出，它是因为超过 cgroup 的内存限制，而被 OOM 杀死的。

第二，java 进程是在容器内运行的，而容器内存的使用量和限制都是 512M (524288kB)。目前使用量已经达到了限制，所以会导致 OOM。

第三，被杀死的进程，PID 为 27281，虚拟内存为 4.3G (total-vm:4613208kB)，匿名内存为 505M (anon-rss:517316kB)，页内存为 19M (20168kB)。换句话说，匿名内存是主要的内存占用。而且，匿名内存加上页内存，总共是 524M，已经超过了 512M 的限制。

综合这几点，可以看出，Tomcat 容器的内存主要用在了匿名内存中，而匿名内存，其实就是主动申请分配的堆内存。

不过，为什么 Tomcat 会申请这么多的堆内存呢？要知道，Tomcat 是基于 Java 开发的，所以应该不难想到，这很可能是 JVM 堆内存配置的问题。

我们知道，JVM 根据系统的内存总量，来自动管理堆内存，不明确配置的话，堆内存的默认限制是物理内存的四分之一。不过，前面我们已经限制了容器内存为 512 M，java 的堆内存到底是多少呢？

我们继续在终端中，执行下面的命令，重新启动 tomcat 容器，并调用 java 命令行来查看堆内存大小：

[复制代码](#)

```
1 # 重新启动容器
2 $ docker rm -f tomcat
3 $ docker run --name tomcat --cpus 0.1 -m 512M -p 8080:8080 -itd feisky/tomcat:8
4
5 # 查看堆内存，注意单位是字节
6 $ docker exec tomcat java -XX:+PrintFlagsFinal -version | grep HeapSize
7     uintx ErgoHeapSizeLimit                = 0
8     uintx HeapSizePerGCThread               = 87241520
9     uintx InitialHeapSize                   := 132120576
10    uintx LargePageHeapSizeThreshold        = 134217728
11    uintx MaxHeapSize                        := 2092957696
12
```

你可以看到，初始堆内存的大小（InitialHeapSize）是 126MB，而最大堆内存则是 1.95GB，这可比容器限制的 512 MB 大多了。

之所以会这么大，其实是因为，容器内部看不到 Docker 为它设置的内存限制。虽然在启动容器时，我们通过 -m 512M 选项，给容器设置了 512M 的内存限制。但实际上，从容器内部看到的限制，却并不是 512M。

我们在终端中，继续执行下面的命令：


[复制代码](#)

```
1 $ docker exec tomcat free -m
2
3      total        used        free      shared  buff/cache   available
4 Mem:    7977        521       1941          0        5514        7148
5 Swap:      0          0           0
```

果然，容器内部看到的内存，仍是主机内存。


知道了问题根源，解决方法就很简单了，给 JVM 正确配置内存限制为 512M 就可以了。

比如，你可以执行下面的命令，通过环境变量 `JAVA_OPTS='-Xmx512m -Xms512m'`，把 JVM 的初始内存和最大内存都设为 512MB：

 复制代码

```
1 # 删除问题容器
2 $ docker rm -f tomcat
3 # 运行新的容器
4 $ docker run --name tomcat --cpus 0.1 -m 512M -e JAVA_OPTS='-Xmx512m -Xms512m' -p 8080:8080
5
```

接着，再切换到终端二中，重新在循环中执行 `curl` 命令，查看 Tomcat 的响应：

 复制代码

```
1 $ for ((i=0;i<30;i++)); do curl localhost:8080; sleep 1; done
2 curl: (56) Recv failure: Connection reset by peer
3 curl: (56) Recv failure: Connection reset by peer
4 Hello, wolrd!
5
6 Hello, wolrd!
7
8 Hello, wolrd!
9
```

可以看到，刚开始时，显示的还是 “Connection reset by peer” 错误。不过，稍等一会儿后，就是连续的 “Hello, wolrd!” 输出了。这说明，Tomcat 已经正常启动。

这时，我们切换回终端一，执行 `docker logs` 命令，查看 Tomcat 容器的日志：

 复制代码

```
1 $ docker logs -f tomcat
2 ...
3 18-Feb-2019 12:52:00.823 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig
4 18-Feb-2019 12:52:01.422 INFO [localhost-startStop-1] org.apache.catalina.startup.HostConfig
5 18-Feb-2019 12:52:01.920 INFO [main] org.apache.coyote.AbstractProtocol.start Starting
6 18-Feb-2019 12:52:02.323 INFO [main] org.apache.coyote.AbstractProtocol.start Starting
7 18-Feb-2019 12:52:02.523 INFO [main] org.apache.catalina.startup.Catalina.start Server
8
9
```

这次，Tomcat 也正常启动了。不过，最后一行的启动时间，似乎比较刺眼。启动过程，居然需要 22 秒，这也太慢了吧。

由于这个时间是花在容器启动上的，要排查这个问题，我们就要重启容器，并借助性能分析工具来分析容器进程。至于工具的选用，回顾一下我们前面的案例，我觉得可以先用 `top` 看看。

我们切换到终端二中，运行 `top` 命令；然后再切换到终端一，执行下面的命令，重启容器：


```
1 # 删除旧容器
2 $ docker rm -f tomcat
3 # 运行新容器
4 $ docker run --name tomcat --cpus 0.1 -m 512M -e JAVA_OPTS='-Xmx512m -Xms512m' -p 8080:8080
5
```

[复制代码](#)

接着，再切换到终端二，观察 top 的输出：

```
1 $ top
2 top - 12:57:18 up 2 days, 5:50, 2 users, load average: 0.00, 0.02, 0.00
3 Tasks: 131 total, 1 running, 74 sleeping, 0 stopped, 0 zombie
4 %Cpu0 : 3.0 us, 0.3 sy, 0.0 ni, 96.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
5 %Cpu1 : 5.7 us, 0.3 sy, 0.0 ni, 94.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
6 KiB Mem : 8169304 total, 2465984 free, 500812 used, 5202508 buff/cache
7 KiB Swap: 0 total, 0 free, 0 used. 7353652 avail Mem
8
9  PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
10 29457 root      20   0 2791736 73704 19164 S   10.0   0.9   0:01.61 java
11 27376 root      20   0 1031760 43768 21680 S    0.3   0.5   2:44.47 docker-containe
12 1 root      20   0  78132   9332   6744 S    0.0   0.1   0:16.12 systemd
13
```

[复制代码](#)

从 top 的输出，我们可以发现，

从系统整体来看，两个 CPU 的使用率分别是 3% 和 5.7%，都不算高，大部分还是空闲的；可用内存还有 7GB（7353652 avail Mem），也非常充足。

具体到进程上，java 进程的 CPU 使用率为 10%，内存使用 0.9%，其他进程就都很低了。

这些指标都不算高，看起来都没啥问题。不过，事实究竟如何呢？我们还得继续找下去。由于 java 进程的 CPU 使用率最高，所以要把它当成重点，继续分析其性能情况。


说到进程的性能分析工具，你一定也想起了 pidstat。接下来，我们就用 pidstat 再分析一下。我们回到终端一中，执行 pidstat 命令：

```
1 # -t 表示显示线程，-p 指定进程号
2 $ pidstat -t -p 29457 1
3 12:59:59      UID      TGID      TID    %usr %system  %guest   %wait   %CPU   CPU   Cor
4 13:00:00        0      29457        -    0.00   0.00   0.00   0.00   0.00    0   jav
5 13:00:00        0        -      29457    0.00   0.00   0.00   0.00   0.00    0   |__
6 13:00:00        0        -      29458    0.00   0.00   0.00   0.00   0.00    1   |__
7 ...
8 13:00:00        0        -      29491    0.00   0.00   0.00   0.00   0.00    0   |__
9
```

[复制代码](#)

结果中，各种 CPU 使用率全是 0，看起来不对呀。再想想，我们有没有漏掉什么线索呢？对了，这时候容器启动已经结束了，在没有客户端请求的情况下，Tomcat 本身啥也不用做，CPU 使用率当然是 0。

为了分析启动过程中的问题，我们需要再次重启容器。继续在终端，按下 Ctrl+C 停止 pidstat 命令；然后执行下面的命令，重启容器。成功重启后，拿到新的 PID，再重新运行 pidstat 命令：

 复制代码

```
1 # 删除旧容器
2 $ docker rm -f tomcat
3 # 运行新容器
4 $ docker run --name tomcat --cpus 0.1 -m 512M -e JAVA_OPTS='-Xmx512m -Xms512m' -p 8080:8080
5 # 查询新容器中进程的 Pid
6 $ PID=$(docker inspect tomcat -f '{{.State.Pid}}')
7 # 执行 pidstat
8 $ pidstat -t -p $PID 1
9 12:59:28      UID      TGID      TID      %usr %system %guest  %wait   %CPU   CPU   Cor
10 12:59:29      0      29850      -    10.00   0.00   0.00   0.00   10.00   0   ja
11 12:59:29      0      -      29850   0.00   0.00   0.00   0.00   0.00   0   |
12 12:59:29      0      -      29897   5.00   1.00   0.00  86.00   6.00   1   |
13 ...
14 12:59:29      0      -      29905   3.00   0.00   0.00  97.00   3.00   0   |
15 12:59:29      0      -      29906   2.00   0.00   0.00  49.00   2.00   1   |
16 12:59:29      0      -      29908   0.00   0.00   0.00  45.00   0.00   0   |
17
```

仔细观察这次的输出，你会发现，虽然 CPU 使用率（%CPU）很低，但等待运行的使用率（%wait）却非常高，最高甚至已经达到了 97%。这说明，这些线程大部分时间都在等待调度，而不是真正的运行。

注：如果你看不到 %wait 指标，请先升级 sysstat 后再试试。

为什么 CPU 使用率这么低，线程的大部分时间还要等待 CPU 呢？由于这个现象因 Docker 而起，自然的，你应该想到，这可能是因为 Docker 为容器设置了限制。

再回顾一下，案例开始时容器的启动命令。我们用 --cpus 0.1，为容器设置了 0.1 个 CPU 的限制，也就是 10% 的 CPU。这里也就可以解释，为什么 java 进程只有 10% 的 CPU 使用率，也会大部分时间都在等待了。

找出原因，最后的优化也就简单了，把 CPU 限制增大就可以了。比如，你可以执行下面的命令，将 CPU 限制增大到 1；然后再重启，并观察启动日志：

 复制代码

```
1 # 删除旧容器
2 $ docker rm -f tomcat
3 # 运行新容器
4 $ docker run --name tomcat --cpus 1 -m 512M -e JAVA_OPTS='-Xmx512m -Xms512m' -p 8080:8080
5 # 查看容器日志
6 $ docker logs -f tomcat
7 ...
8 18-Feb-2019 12:54:02.139 INFO [main] org.apache.catalina.startup.Catalina.start Server :
9
```

现在可以看到，Tomcat 的启动过程，只需要 2 秒就完成了，果然比前面的 22 秒快多了。

虽然我们通过增大 CPU 的限制，解决了这个问题。不过再碰到类似问题，你可能会觉得这种方法太麻烦了。因为要设置容器的资源限制，还需要我们预先评估应用程序的性能。显然还有更简单的方法，比如说直接去掉限制，让容器跑就是了。

不过，这种简单方法，却很可能带来更严重的问题。没有资源限制，就意味着容器可以占用整个系统的资源。这样，一旦任何应用程序发生异常，都有可能拖垮整台机器。

实际上，这也是在各大容器平台上最常见的一个问题。一开始图省事不设限，但当容器数量增长上来的时候，就会经常出现各种异常问题。最终查下来，可能就是因为某个应用资源使用过高，导致整台机器短期内无法响应。只有设置了资源限制，才能确保杜绝类似问题。

小结

今天，我带你学习了，如何分析容器化后应用程序性能下降的问题。

如果你在 Docker 容器中运行 Java 应用，一定要确保，在设置容器资源限制的同时，配置好 JVM 的资源选项（比如堆内存等）。当然，如果你可以升级 Java 版本，那么升级到 Java 10，就可以自动解决类似问题了。

当碰到容器化的应用程序性能时，你依然可以使用，我们前面讲过的各种方法来分析和定位。只不过要记得，容器化后的性能分析，跟前面内容稍微有些区别，比如下面这几点。

容器本身通过 cgroups 进行资源隔离，所以，在分析时要考虑 cgroups 对应用程序的影响。

容器的文件系统、网络协议栈等跟主机隔离。虽然在容器外面，我们也可以分析容器的行为，不过有时候，进入容器的命名空间内部，可能更为方便。

容器的运行可能还会依赖于其他组件，比如各种网络插件（比如 CNI）、存储插件（比如 CSI）、设备插件（比如 GPU）等，让容器的性能分析更加复杂。如果你需要分析容器性能，别忘了考虑它们对性能的影响。

思考

最后，我想邀请你一起来聊聊，你碰到过的容器性能问题。你是怎么分析它们的？又是怎么解决根源问题的？你可以结合我的讲解，总结自己的思路。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

Linux 性能优化实战

10 分钟帮你找到系统瓶颈

倪朋飞

微软资深工程师
Kubernetes 项目维护者



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(2)



Adam

这个问题应该是/proc 文件系统并不知道用户通过 Cgroups 给这个容器做了限制导致的。



2019-03-11



ninuxer

打卡day49

前两天在我们线下环境一台docker宿主机上，一直无法create容器，后来看日志，发现有两个可疑之处：
第一:docker日志显示socket文件损坏，但是当时运行其他docker管理命令能正常返回结果
第二:宿主机上有个kworker/u80进程cpu利用率一直100%，最终是通过重启宿主机解决的~



2019-03-11

