

52 | 案例篇：服务吞吐量下降很厉害，怎么分析？

倪朋飞 2019-03-25



00:00

讲述：冯永吉

大小：15.30M

16:42

你好，我是倪朋飞。

上一节，我们一起学习了怎么使用动态追踪来观察应用程序和内核的行为。先简单来回顾一下。

所谓动态追踪，就是在系统或者应用程序还在正常运行时候，通过内核中提供的探针，来动态追踪它们的行为，从而辅助排查出性能问题的瓶颈。

使用动态追踪，便可以在不修改代码也不重启服务的情况下，动态了解应用程序或者内核的行为。这对排查线上的问题、特别是不容易重现的问题尤其有效。

在 Linux 系统中，常见的动态追踪方法包括 ftrace、perf、eBPF/BCC 以及 SystemTap 等。

使用 perf 配合火焰图寻找热点函数，是一个比较通用的性能定位方法，在很多场景中都可以使用。

如果这仍满足不了你的要求，那么在新版的内核中，eBPF 和 BCC 是最灵活的动态追踪方法。

而在旧版本内核，特别是在 RHEL 系统中，由于 eBPF 支持受限，SystemTap 和 ftrace 往往是更好的选择。

在 [网络请求延迟变大](#) 的案例中，我带你一起分析了一个网络请求延迟增大的问题。当时我们分析知道，那是由于服务器端开启 TCP 的 Nagle 算法，而客户端却开启了延迟确认所导致的。

其实，除了延迟问题外，网络请求的吞吐量下降，是另一个常见的性能问题。那么，针对这种吞吐量下降问题，我们又该如何进行分析呢？


接下来，我就以最常用的反向代理服务器 Nginx 为例，带你一起看看，如何分析服务吞吐量下降的问题。

案例准备

今天的案例需要用到两台虚拟机，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、wrk、perf、FlameGraph 等工具，比如

 复制代码

```
1 # 安装必备 docker、curl 和 perf
2 $ apt-get install -y docker.io curl build-essential linux-tools-common
3 # 安装火焰图工具
4 $ git clone https://github.com/brendangregg/FlameGraph
5 # 安装 wrk
6 $ git clone https://github.com/wg/wrk
7 $ cd wrk && make && sudo cp wrk /usr/local/bin/
8
```

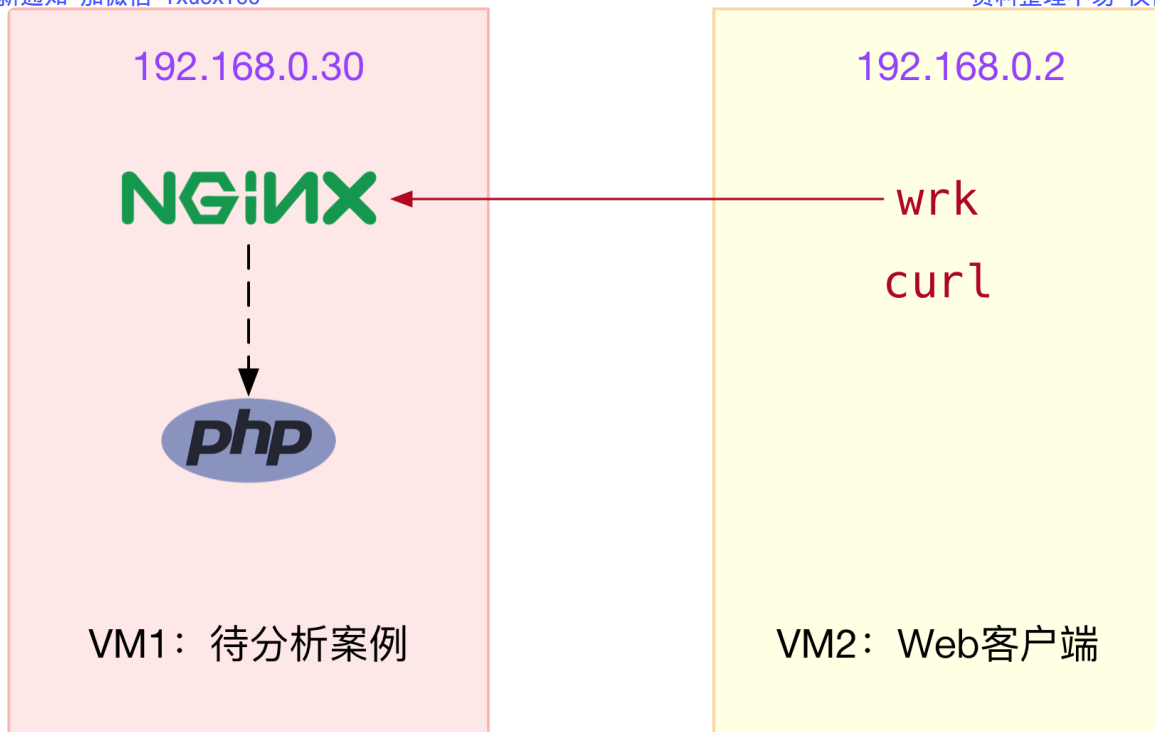
这些工具，我们在前面的案例中已经多次使用，这儿就不再重复。你可以打开两个终端，分别登录到这两台虚拟机中，并安装上述工具。

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。

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 Nginx + PHP 应用，它们的关系如下图所示：



其中，wrk 和 curl 是 Nginx 的客户端，而 PHP 应用则是一个简单的 Hello World：

```
1 <?php
2 echo "Hello World!"
3 ?>
4
```

[复制代码](#)

为了方便你运行，我已经把案例应用打包成了两个 Docker 镜像，并推送到 Docker Hub 中。你可以直接按照下面的步骤来运行它。

同时，为了分析方便，这两个容器都将运行在 host network 模式中。这样，我们就不用切换到容器的网络命名空间，而可以直接观察它们的套接字状态。

我们先在终端一中，执行下面的命令，启动 Nginx 应用，并监听在 80 端口。如果一切正常，你应该可以看到如下的输出：

```
1 $ docker run --name nginx --network host --privileged -itd feisky/nginx-tp
2 6477c607c13b37943234755a14987ffb3a31c33a7f04f75bb1c190e710bce19e
3 $ docker run --name phpfpn --network host --privileged -itd feisky/php-fpm-tp
4 09e0255159f0c8a647e22cd68bd097bec7efc48b21e5d91618ff29b882fa7c1f
5
```

[复制代码](#)

然后，执行 docker ps 命令，查询容器的状态，你会发现，容器已经处于运行状态（Up）了：


```
1 $ docker ps
2 CONTAINER ID          IMAGE                COMMAND              CREATED              ST/
```

[复制代码](#)

```
3 09e0255159f0      feisky/php-fpm-tp  "php-fpm -F --pid /o..." 28 seconds ago Up
4 6477c607c13b      feisky/nginx-tp   "/init.sh"                29 seconds ago Up
5
```

不过，从 docker ps 的输出，我们只能知道容器处于运行状态。至于 Nginx 能不能正常处理外部的请求，还需要我们进一步确认。

接着，切换到终端二中，执行下面的 curl 命令，进一步验证 Nginx 能否正常访问。如果你看到“Hello World!” 的输出，说明 Nginx+PHP 的应用已经正常启动了：


 复制代码

```
1 $ curl http://192.168.0.30
2 Hello World!
3
```

提示：如果你看到不一样的结果，可以再次执行 docker ps -a 确认容器的状态，并执行 docker logs < 容器名 > 来查看容器日志，从而找出原因。

接下来，我们就来测试一下，案例中 Nginx 的吞吐量。

我们继续在终端二中，执行 wrk 命令，来测试 Nginx 的性能：

 复制代码

```
1 # 默认测试时间为 10s，请求超时 2s
2 $ wrk --latency -c 1000 http://192.168.0.30
3 Running 10s test @ http://192.168.0.30
4 2 threads and 1000 connections
5   Thread Stats   Avg      Stdev     Max    +/-  Stdev
6   Latency      14.82ms   42.47ms  874.96ms   98.43%
7   Req/Sec      550.55     1.36k    5.70k    93.10%
8   Latency Distribution
9     50%     11.03ms
10    75%     15.90ms
11    90%     23.65ms
12    99%    215.03ms
13 1910 requests in 10.10s, 573.56KB read
14 Non-2xx or 3xx responses: 1910
15 Requests/sec:   189.10
16 Transfer/sec:   56.78KB
17
```


从 wrk 的结果中，你可以看到吞吐量（也就是每秒请求数）只有 189，并且所有 1910 个请求收到的都是异常响应（非 2xx 或 3xx）。这些数据显然表明，吞吐量太低了，并且请求处理都失败了。这是怎么回事呢？

根据 wrk 输出的统计结果，我们可以看到，总共传输的数据量只有 573 KB，那就肯定不会是带宽受限导致的。所以，我们应该从请求数的角度来分析。

分析请求数，特别是 HTTP 的请求数，有什么好思路吗？当然就要从 TCP 连接数入手。


连接数优化

要查看 TCP 连接数的汇总情况，首选工具自然是 ss 命令。为了观察 wrk 测试时发生的问题，我们在终端二中再次启动 wrk，并且把总的测试时间延长到 30 分钟：

 复制代码

```
1 # 测试时间 30 分钟
2 $ wrk --latency -c 1000 -d 1800 http://192.168.0.30
3
```

然后，回到终端一中，观察 TCP 连接数：

 复制代码

```
1 $ ss -s
2 Total: 177 (kernel 1565)
3 TCP:    1193 (estab 5, closed 1178, orphaned 0, synrecv 0, timewait 1178/0), ports 0
4
5 Transport Total      IP        IPv6
6 *              1565      -        -
7 RAW             1         0         1
8 UDP             2         2         0
9 TCP             15        12        3
10 INET           18        14        4
11 FRAG           0         0         0
12
```

从这里看出，wrk 并发 1000 请求时，建立连接数只有 5，而 closed 和 timewait 状态的连接则有 1100 多。其实从这儿你就可以发现两个问题：


一个是建立连接数太少了；

另一个是 timewait 状态连接太多了。

分析问题，自然要先从相对简单的下手。我们先来看第二个关于 timewait 的问题。在之前的 NAT 案例中，我已经提到过，内核中的连接跟踪模块，有可能会导致 timewait 问题。我们今天的案例还是基于 Docker 运行，而 Docker 使用的 iptables，就会使用连接跟踪模块来管理 NAT。那么，怎么确认是不是连接跟踪导致的问题呢？

其实，最简单的方法，就是通过 dmesg 查看系统日志，如果有连接跟踪出了问题，应该会看到 nf_contrack 相关的日志。

我们可以继续在终端一中，运行下面的命令，查看系统日志：


 复制代码

```
1 $ dmesg | tail
2 [88356.354329] nf_contrack: nf_contrack: table full, dropping packet
```

```
3 [88356.354374] nf_conntrack: nf_conntrack: table full, dropping packet
4
5
```

从日志中，你可以看到 `nf_conntrack: table full, dropping packet` 的错误日志。这说明，正是连接跟踪导致的问题。


这种情况下，我们应该想起前面学过的两个内核选项——连接跟踪数的最大限制 `nf_conntrack_max`，以及当前的连接跟踪数 `nf_conntrack_count`。执行下面的命令，你就可以查询这两个选项：

 复制代码

```
1 $ sysctl net.netfilter.nf_conntrack_max
2 net.netfilter.nf_conntrack_max = 200
3 $ sysctl net.netfilter.nf_conntrack_count
4 net.netfilter.nf_conntrack_count = 200
5
```


这次的输出中，你可以看到最大的连接跟踪限制只有 200，并且全部被占用了。200 的限制显然太小，不过相应的优化也很简单，调大就可以了。

我们执行下面的命令，将 `nf_conntrack_max` 增大：

 复制代码

```
1 # 将连接跟踪限制增大到 1048576
2 $ sysctl -w net.netfilter.nf_conntrack_max=1048576
3
```

连接跟踪限制增大后，对 Nginx 吞吐量的优化效果如何呢？我们不妨再来测试一下。你可以切换到终端二中，按下 `Ctrl+C`；然后执行下面的 `wrk` 命令，重新测试 Nginx 的性能：

 复制代码

```
1 # 默认测试时间为 10s，请求超时 2s
2 $ wrk --latency -c 1000 http://192.168.0.30
3 ...
4 54221 requests in 10.07s, 15.16MB read
5 Socket errors: connect 0, read 7, write 0, timeout 110
6 Non-2xx or 3xx responses: 45577
7 Requests/sec: 5382.21
8 Transfer/sec: 1.50MB
9
```

从 `wrk` 的输出中，你可以看到，连接跟踪的优化效果非常好，吞吐量已经从刚才的 189 增大到了 5382。看起来性能提升了将近 30 倍，

不过，这是不是就能说明，我们已经把 Nginx 的性能优化好了呢？


别急，我们再来看看 wrk 汇报的其他数据。果然，10s 内的总请求数虽然增大到了 5 万，但是有 4 万多响应异常，说白了，真正成功的只有 8000 多个（54221-45577=8644）。

很明显，大部分请求的响应都是异常的。那么，该怎么分析响应异常的问题呢？

工作进程优化

由于这些响应并非 Socket error，说明 Nginx 已经收到了请求，只不过，响应的状态码并不是我们期望的 2xx（表示成功）或 3xx（表示重定向）。所以，这种情况下，搞清楚 Nginx 真正的响应就很重要了。

不过这也不难，我们切换回终端，执行下面的 docker 命令，查询 Nginx 容器日志就知道了：

 复制代码


```
1 $ docker logs nginx --tail 3
2 192.168.0.2 - - [15/Mar/2019:22:43:27 +0000] "GET / HTTP/1.1" 499 0 "-" "-" "-"
3 192.168.0.2 - - [15/Mar/2019:22:43:27 +0000] "GET / HTTP/1.1" 499 0 "-" "-" "-"
4 192.168.0.2 - - [15/Mar/2019:22:43:27 +0000] "GET / HTTP/1.1" 499 0 "-" "-" "-"
5
```

从 Nginx 的日志中，我们可以看到，响应状态码为 499。

499 并非标准的 HTTP 状态码，而是由 Nginx 扩展而来，表示服务器端还没来得及响应时，客户端就已经关闭连接了。换句话说，问题在于服务器端处理太慢，客户端因为超时（wrk 超时时间为 2s），主动断开了连接。

既然问题出在了服务器端处理慢，而案例本身是 Nginx+PHP 的应用，那是不是可以猜测，是因为 PHP 处理过慢呢？

我么可以在终端中，执行下面的 docker 命令，查询 PHP 容器日志：


 复制代码

```
1 $ docker logs php-fpm --tail 5
2 [15-Mar-2019 22:28:56] WARNING: [pool www] server reached max_children setting (5), con:
3 [15-Mar-2019 22:43:17] WARNING: [pool www] server reached max_children setting (5), con:
4
```

从这个日志中，我们可以看到两条警告信息，server reached max_children setting (5)，并建议增大 max_children。


max_children 表示 php-fpm 子进程的最大数量，当然是数值越大，可以同时处理的请求数就越多。不过由于这是进程问题，数量增大，也会导致更多的内存和 CPU 占用。所以，我们还不能设置得过大。

一般来说，每个 php-fpm 子进程可能会占用 20 MB 左右的内存。所以，你可以根据内存和 CPU 个数，估算一个合理的值。这儿我把它设置成了 20，并将优化后的配置重新打包成了 Docker 镜像。你可以执行下面的命令来执行它：

 复制代码

```
1 # 停止旧的容器
2 $ docker rm -f nginx php-fpm
3
4 # 使用新镜像启动 Nginx 和 PHP
5 $ docker run --name nginx --network host --privileged -itd feisky/nginx-tp:1
6 $ docker run --name php-fpm --network host --privileged -itd feisky/php-fpm-tp:1
7
```

然后我们切换到终端二，再次执行下面的 wrk 命令，重新测试 Nginx 的性能：

 复制代码

```
1 # 默认测试时间为 10s，请求超时 2s
2 $ wrk --latency -c 1000 http://192.168.0.30
3 ...
4 47210 requests in 10.08s, 12.51MB read
5 Socket errors: connect 0, read 4, write 0, timeout 91
6 Non-2xx or 3xx responses: 31692
7 Requests/sec: 4683.82
8 Transfer/sec: 1.24MB
9
```

从 wrk 的输出中，可以看到，虽然吞吐量只有 4683，比刚才的 5382 少了一些；但是测试期间成功的请求数却多了不少，从原来的 8000，增长到了 15000 (47210-31692=15518)。

不过，虽然性能有所提升，可 4000 多的吞吐量显然还是比较差的，并且大部分请求的响应依然还是异常。接下来，该怎么去进一步提升 Nginx 的吞吐量呢？

套接字优化


回想一下网络性能的分析套路，以及 Linux 协议栈的原理，我们可以从套接字、TCP 协议等逐层分析。而分析的第一步，自然还是要观察有没有发生丢包现象。

我们切换到终端二中，重新运行测试，这次还是要用 -d 参数延长测试时间，以便模拟性能瓶颈的现场：

 复制代码

```
1 # 测试时间 30 分钟
2 $ wrk --latency -c 1000 -d 1800 http://192.168.0.30
3
```


然后回到终端一中，观察有没有发生套接字的丢包现象：

 复制代码

```
1 # 只关注套接字统计
2 $ netstat -s | grep socket
3      73 resets received for embryonic SYN_RECV sockets
4      308582 TCP sockets finished time wait in fast timer
5      8 delayed acks further delayed because of locked socket
6      290566 times the listen queue of a socket overflowed
7      290566 SYNs to LISTEN sockets dropped
8
9 # 稍等一会，再次运行
10 $ netstat -s | grep socket
11      73 resets received for embryonic SYN_RECV sockets
12      314722 TCP sockets finished time wait in fast timer
13      8 delayed acks further delayed because of locked socket
14      344440 times the listen queue of a socket overflowed
15      344440 SYNs to LISTEN sockets dropped
16
```

根据两次统计结果中 socket overflowed 和 sockets dropped 的变化，你可以看到，有大量的套接字丢包，并且丢包都是套接字队列溢出导致的。所以，接下来，我们应该分析连接队列的大小是不是有异常。


你可以执行下面的命令，查看套接字的队列大小：

 复制代码

```
1 $ ss -ltnp
2 State      Recv-Q      Send-Q       Local Address:Port      Peer Address:Port
3 LISTEN     10           10           0.0.0.0:80              0.0.0.0:*
4 LISTEN     7            10           *:9000                  *:*
```

这次可以看到，Nginx 和 php-fpm 的监听队列（Send-Q）只有 10，而 nginx 的当前监听队列长度（Recv-Q）已经达到了最大值，php-fpm 也已经接近了最大值。很明显，套接字监听队列的长度太小了，需要增大。


关于套接字监听队列长度的设置，既可以在应用程序中，通过套接字接口调整，也支持通过内核选项来配置。我们继续在终端一中，执行下面的命令，分别查询 Nginx 和内核选项对监听队列长度的配置：

 复制代码

```
1 # 查询 nginx 监听队列长度配置
2 $ docker exec nginx cat /etc/nginx/nginx.conf | grep backlog
3      listen      80  backlog=10;
4
5 # 查询 php-fpm 监听队列长度
6 $ docker exec php-fpm cat /opt/bitnami/php/etc/php-fpm.d/www.conf | grep backlog
7 ; Set listen(2) backlog.
8 ;listen.backlog = 511
9
10 # somaxconn 是系统级套接字监听队列上限
11 $ sysctl net.core.somaxconn
12 net.core.somaxconn = 10
```


从输出中可以看到，Nginx 和 somaxconn 的配置都是 10，而 php-fpm 的配置也只有 511，显然都太小了。那么，优化方法就是增大这三个配置，比如，可以把 Nginx 和 php-fpm 的队列长度增大到 8192，而把 somaxconn 增大到 65536。

同样地，我也把这些优化后的 Nginx，重新打包成了两个 Docker 镜像，你可以执行下面的命令来运行它：

 复制代码

```
1 # 停止旧的容器
2 $ docker rm -f nginx php-fpm
3
4 # 使用新镜像启动 Nginx 和 PHP
5 $ docker run --name nginx --network host --privileged -itd feisky/nginx-tp:2
6 $ docker run --name php-fpm --network host --privileged -itd feisky/php-fpm-tp:2
7
```


然后，切换到终端二中，重新测试 Nginx 的性能：

 复制代码

```
1 $ wrk --latency -c 1000 http://192.168.0.30
2 ...
3 62247 requests in 10.06s, 18.25MB read
4 Non-2xx or 3xx responses: 62247
5 Requests/sec: 6185.65
6 Transfer/sec: 1.81MB
7
```

现在的吞吐量已经增大到了 6185，并且在测试的时候，如果你在终端一中重新执行 `netstat -s | grep socket`，还会发现，现在已经没有套接字丢包问题了。

不过，这次 Nginx 的响应，再一次全部失败了，都是 Non-2xx or 3xx。这是怎么回事呢？我们再去终端一中，查看 Nginx 日志：

 复制代码

```
1 $ docker logs nginx --tail 10
2 2019/03/15 16:52:39 [crit] 15#15: *999779 connect() to 127.0.0.1:9000 failed (99: Cannot
3
```

你可以看到，Nginx 报出了无法连接 fastcgi 的错误，错误消息是 Connect 时，Cannot assign requested address。这个错误消息对应的错误代码为 EADDRNOTAVAIL，表示 IP 地址或者端口号不可用。

在这里，显然只能是端口号的问题。接下来，我们就来分析端口号的情况。

端口号优化

根据网络套接字的原理，当客户端连接服务器端时，需要分配一个临时端口号，而 Nginx 正是 PHP-FPM 的客户端。端口号的范围并不是无限的，最多也只有 6 万多。

我们执行下面的命令，就可以查询系统配置的临时端口号范围：

```
1 $ sysctl net.ipv4.ip_local_port_range
2 net.ipv4.ip_local_port_range=20000 20050
3
```

[复制代码](#)

你可以看到，临时端口的范围只有 50 个，显然太小了。优化方法很容易想到，增大这个范围就可以了。比如，你可以执行下面的命令，把端口号范围扩展为 “10000 65535”：

```
1 $ sysctl -w net.ipv4.ip_local_port_range="10000 65535"
2 net.ipv4.ip_local_port_range = 10000 65535
3
```

[复制代码](#)

优化完成后，我们再次切换到终端二中，测试性能：

```
1 $ wrk --latency -c 1000 http://192.168.0.30/
2 ...
3 32308 requests in 10.07s, 6.71MB read
4 Socket errors: connect 0, read 2027, write 0, timeout 433
5 Non-2xx or 3xx responses: 30
6 Requests/sec: 3208.58
7 Transfer/sec: 682.15KB
8
```

[复制代码](#)

这次，异常的响应少多了，不过，吞吐量也下降到了 3208。并且，这次还出现了很多 Socket read errors。显然，还得进一步优化。

火焰图


前面我们已经优化了很多配置。这些配置在优化网络的同时，却也会带来其他资源使用的上升。这样来看，是不是说明其他资源遇到瓶颈了呢？

我们不妨在终端二中，执行下面的命令，重新启动长时间测试：

```
1 # 测试时间 30 分钟
2 $ wrk --latency -c 1000 -d 1800 http://192.168.0.30
3
```

[复制代码](#)

然后，切换回终端一中，执行 `top`，观察 CPU 和内存的使用：


 复制代码

```
1 $ top
2 ...
3 %Cpu0 : 30.7 us, 48.7 sy, 0.0 ni, 2.3 id, 0.0 wa, 0.0 hi, 18.3 si, 0.0 st
4 %Cpu1 : 28.2 us, 46.5 sy, 0.0 ni, 2.0 id, 0.0 wa, 0.0 hi, 23.3 si, 0.0 st
5 KiB Mem : 8167020 total, 5867788 free, 490400 used, 1808832 buff/cache
6 KiB Swap: 0 total, 0 free, 0 used. 7361172 avail Mem
7
8 PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
9 20379 systemd+ 20   0  38068   8692   2392 R   36.1   0.1   0:28.86 nginx
10 20381 systemd+ 20   0  38024   8700   2392 S   33.8   0.1   0:29.29 nginx
11 1558 root      20   0 1118172 85868  39044 S   32.8   1.1  22:55.79 dockerd
12 20313 root      20   0  11024   5968   3956 S   27.2   0.1   0:22.78 docker-containe
13 13730 root      20   0      0      0      0 I    4.0   0.0   0:10.07 kworker/u4:0-ev
14
```

从 `top` 的结果中可以看到，可用内存还是很充足的，但系统 CPU 使用率（sy）比较高，两个 CPU 的系统 CPU 使用率都接近 50%，且空闲 CPU 使用率只有 2%。再看进程部分，CPU 主要被两个 Nginx 进程和两个 docker 相关的进程占用，使用率都是 30% 左右。

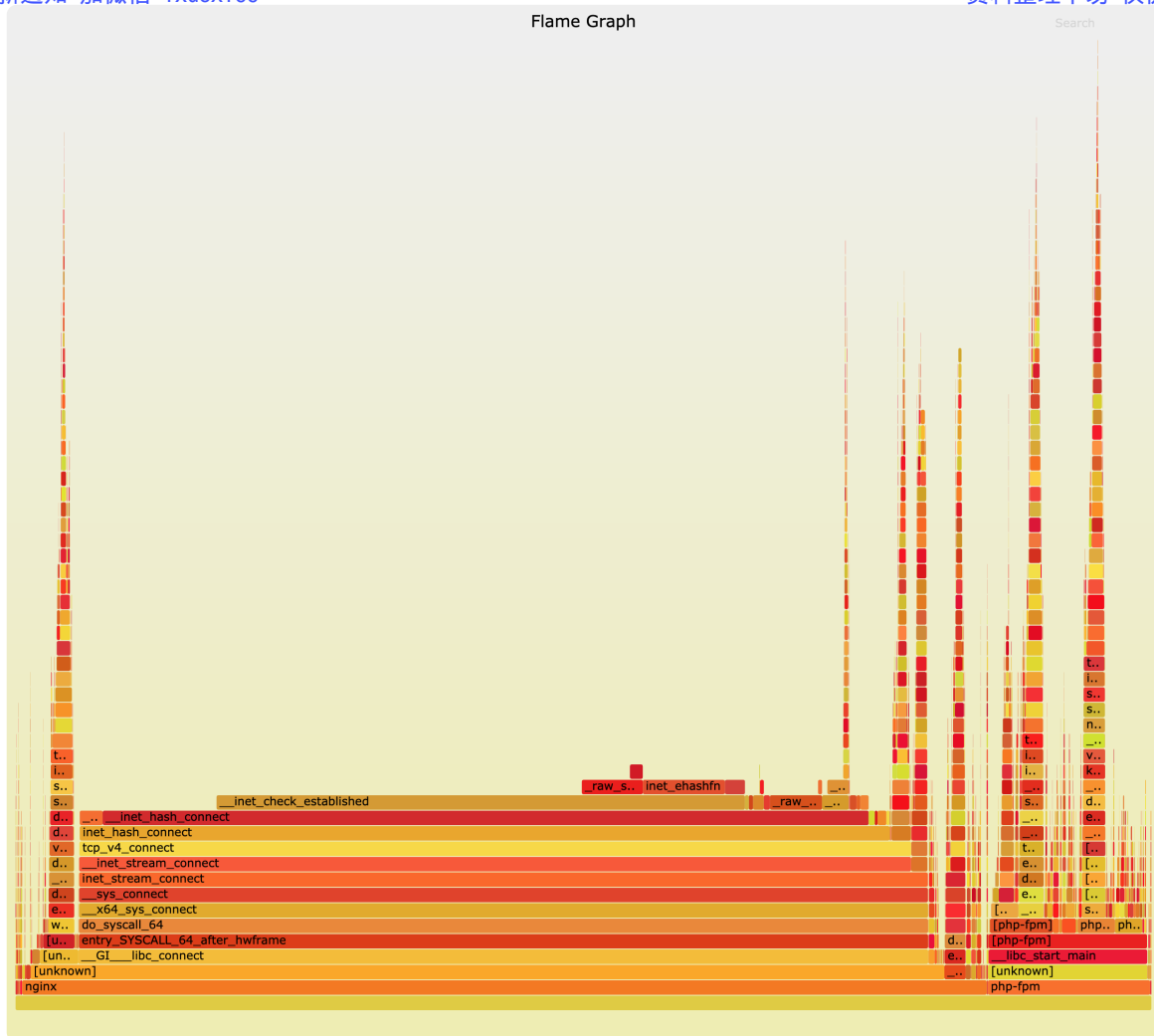
CPU 使用率上升了，该怎么进行分析呢？我想，你已经还记得我们多次用到的 `perf`，再配合前两节讲过的火焰图，很容易就能找到系统中的热点函数。

我们保持终端二中的 `wrk` 继续运行；在终端一中，执行 `perf` 和 `flamegraph` 脚本，生成火焰图：

 复制代码

```
1 # 执行 perf 记录事件
2 $ perf record -g
3
4 # 切换到 FlameGraph 安装路径执行下面的命令生成火焰图
5 $ perf script -i ~/perf.data | ./stackcollapse-perf.pl --all | ./flamegraph.pl > nginx.svg
6
```

然后，使用浏览器打开生成的 `nginx.svg`，你就可以看到下面的火焰图：




根据我们讲过的火焰图原理，这个图应该从下往上、沿着调用栈中最宽的函数，来分析执行次数最多的函数。

这儿中间的 `do_syscall_64`、`tcp_v4_connect`、`inet_hash_connect` 这个堆栈，很明显就是最需要关注的地方。`inet_hash_connect()` 是 Linux 内核中负责分配临时端口号的函数。所以，这个瓶颈应该还在临时端口的分配上。

在上一步的“端口号”优化中，临时端口号的范围，已经优化成了“10000 65535”。这显然是一个非常大的范围，那么，端口号的分配为什么又成了瓶颈呢？

一时想不到也没关系，我们可以暂且放下，先看看其他因素的影响。再顺着 `inet_hash_connect` 往堆栈上面查看，下一个热点是 `_init_check_established` 函数。而这个函数的目的，是检查端口号是否可用。结合这一点，你应该可以想到，如果有大量连接占用着端口，那么检查端口号可用的函数，不就会消耗更多的 CPU 吗？


实际是否如此呢？我们可以继续在终端一中运行 `ss` 命令，查看连接状态统计：

 复制代码

```
1 $ ss -s
2 TCP:    32775 (estab 1, closed 32768, orphaned 0, synrecv 0, timewait 32768/0), ports 0
3 ...
4
```

这回可以看到，有大量连接（这儿是 32768）处于 timewait 状态，而 timewait 状态的连接，本身会继续占用端口号。如果这些端口号可以重用，那么自然就可以缩短 `_init_check_established` 的过程。而 Linux 内核中，恰好有一个 `tcp_tw_reuse` 选项，用来控制端口号的重用。


我们在终端一中，运行下面的命令，查询它的配置：

 复制代码

```
1 $ sysctl net.ipv4.tcp_tw_reuse
2 net.ipv4.tcp_tw_reuse = 0
3
```

你可以看到，`tcp_tw_reuse` 是 0，也就是禁止状态。其实看到这里，我们就能理解，为什么临时端口号的分配会是系统运行的热点了。当然，优化方法也很容易，把它设置成 1 就可以开启了。

我把优化后的应用，也打包成了两个 Docker 镜像，你可以执行下面的命令来运行：

 复制代码

```
1 # 停止旧的容器
2 $ docker rm -f nginx phpfpn
3
4 # 使用新镜像启动 Nginx 和 PHP
5 $ docker run --name nginx --network host --privileged -itd feisky/nginx-tp:3
6 $ docker run --name phpfpn --network host --privileged -itd feisky/php-fpm-tp:3
7
```


容器启动后，切换到终端二中，再次测试优化后的效果：

 复制代码

```
1 $ wrk --latency -c 1000 http://192.168.0.30/
2 ...
3 52119 requests in 10.06s, 10.81MB read
4 Socket errors: connect 0, read 850, write 0, timeout 0
5 Requests/sec: 5180.48
6 Transfer/sec: 1.07MB
7
```

现在的吞吐量已经达到了 5000 多，并且只有少量的 Socket errors，也不再有 Non-2xx or 3xx 的响应了。说明一切终于正常了。

案例的最后，不要忘记执行下面的命令，删除案例应用：

 复制代码

```
1 # 停止 nginx 和 phpfpn 容器
2 $ docker rm -f nginx phpfpn
3
```

小结

今天，我带你一起学习了服务吞吐量下降后的分析方法。其实，从这个案例你也可以看出，性能问题的分析，总是离不开系统和应用程序的原理。

实际上，分析性能瓶颈，最核心的也正是掌握运用这些原理。

首先，利用各种性能工具，收集想要的性能指标，从而清楚系统和应用程序的运行状态；

其次，拿目前状态跟系统原理进行比较，不一致的地方，就是我们要重点分析的对象。

从这个角度出发，再进一步借助 perf、火焰图、bcc 等动态追踪工具，找出热点函数，就可以定位瓶颈的来源，确定相应的优化方法。

思考

最后，我想邀请你一起来聊聊，你碰到过的吞吐量下降问题。你是怎么分析它们的根源？又是怎么解决的？你可以结合我的讲述，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。

© 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(1)



ninuxer

打卡day55

缺乏由现象联想到可能原因的系统性思维～



2019-03-25

