



下载APP



加餐02 | 理解perf: 怎么用perf聚焦热点函数?

2021-02-01 李程远

容器实战高手课

[进入课程 >](#)**讲述: 李程远**

时长 17:15 大小 15.81M



你好, 我是程远。今天我要和你聊一聊容器中如何使用 perf。

🔗 **上一讲**中, 我们分析了一个生产环境里的一个真实例子, 由于节点中的大量的 IPVS 规则导致了容器在往外发送网络包的时候, 时不时会有很高的延时。在调试分析这个网络延时问题的过程中, 我们会使用多种 Linux 内核的调试工具, 利用这些工具, 我们就能很清晰地找到这个问题的根本原因。

在后面的课程里, 我们会挨个来讲解这些工具, 其中 perf 工具的使用相对来说要简单些。所以这一讲我们先来看 perf 这个工具。



问题回顾

在具体介绍 perf 之前，我们先来回顾一下，上一讲中，我们是在什么情况下开始使用 perf 工具的，使用了 perf 工具之后给我们带来了哪些信息。

在调试网路延时的时候，我们使用了 ebpf 的工具之后，发现了节点上一个 CPU，也就是 CPU32 的 Softirq CPU Usage（在运行 top 时，%Cpu 那行中的 si 数值就是 Softirq CPU Usage）时不时地会增高一下。

在发现 CPU Usage 异常增高的时候，我们肯定想知道是什么程序引起了 CPU Usage 的异常增高，这时候我们就可以用到 perf 了。

具体怎么操作呢？我们可以通过**抓取数据、数据读取和异常聚焦**三个步骤来实现。

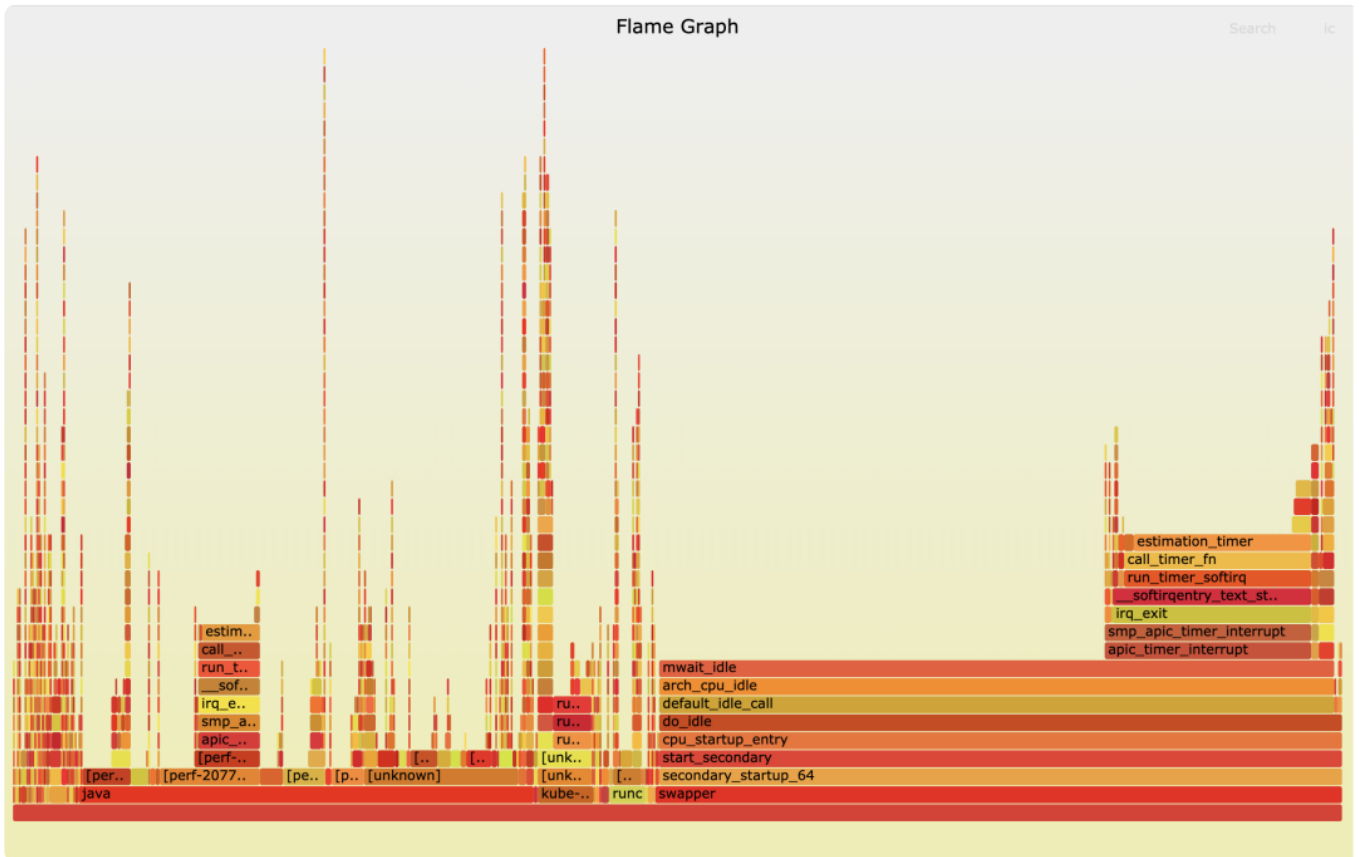
第一步，抓取数据。当时我们运行了下面这条 perf 命令，这里的参数 -C 32 是指定只抓取 CPU32 的执行指令；-g 是指 call-graph enable，也就是记录函数调用关系；sleep 10 主要是为了让 perf 抓取 10 秒钟的数据。

```
1 # perf record -C 32 -g -- sleep 10
```

[复制代码](#)

执行完 perf record 之后，我们可以用 perf report 命令进行第二步，也就是读取数据。为了更加直观地看到 CPU32 上的函数调用情况，我给你生成了一个火焰图（火焰图的生产方法，我们在后面介绍）。

通过这个火焰图，我们发现了在 Softirq 里 TIMER softirq（run_timer_softirq）的占比很高，并且 timer 主要处理的都是 estimation_timer() 这个函数，也就是看火焰图 X 轴占比比较大的函数。这就是第三步异常聚焦，也就是说我们通过 perf 在 CPU Usage 异常的 CPU32 上，找到了具体是哪一个内核函数使用占比较高。这样在后面的调试分析中，我们就可以聚焦到这个内核函数 estimation_timer() 上了。



好了，通过回顾我们在网络延时例子中是如何使用 perf 的，我们知道了这一点，**perf 可以在 CPU Usage 增高的节点上找到具体的引起 CPU 增高的函数，然后我们就可以有针对性地聚焦到那个函数做分析。**

既然 perf 工具这么有用，想要更好地使用这个工具，我们就要好好认识一下它，那我们就一起看看 perf 的基本概念和常用的使用方法。

如何理解 Perf 的概念和工作机制？

Perf 这个工具最早是 Linux 内核著名开发者 Ingo Molnar 开发的，它的源代码在 [内核源码tools](#) 目录下，在每个 Linux 发行版里都有这个工具，比如 CentOS 里我们可以运行 `yum install perf` 来安装，在 Ubuntu 里我们可以运行 `apt install linux-tools-common` 来安装。

Event

第一次上手使用 perf 的时候，我们可以先运行一下 `perf list` 这个命令，然后就会看到 perf 列出了大量的 event，比如下面这个例子就列出了常用的 event。

```
1  # perf list
2
3  ...
4  branch-instructions OR branches      [Hardware event]
5  branch-misses                        [Hardware event]
6  bus-cycles                           [Hardware event]
7  cache-misses                         [Hardware event]
8  cache-references                     [Hardware event]
9  cpu-cycles OR cycles                 [Hardware event]
10 instructions                         [Hardware event]
11 ref-cycles                           [Hardware event]
12
13 alignment-faults                     [Software event]
14 bpf-output                           [Software event]
15 context-switches OR cs                [Software event]
16 cpu-clock                            [Software event]
17 cpu-migrations OR migrations          [Software event]
18 dummy                                [Software event]
19 emulation-faults                     [Software event]
20 major-faults                         [Software event]
21 minor-faults                         [Software event]
22 page-faults OR faults                [Software event]
23 task-clock                           [Software event]
24 ...
25
26 block:block_bio_bounce                [Tracepoint event]
27 block:block_bio_complete              [Tracepoint event]
28 block:block_bio_frontmerge            [Tracepoint event]
29 block:block_bio_queue                 [Tracepoint event]
30 block:block_bio_remap                 [Tracepoint event]
```

从这里我们可以了解到 event 都有哪些类型，perf list 列出的每个 event 后面都有一个"[]", 里面写了这个 event 属于什么类型，比如"Hardware event"、"Software event"等。完整的 event 类型，我们在内核代码枚举结构 perf_type_id 里可以看到。

接下来我们就说三个主要的 event，它们分别是 Hardware event、Software event 还有 Tracepoints event。

Hardware event

Hardware event 来自处理器中的一个 PMU (Performance Monitoring Unit)，这些 event 数目不多，都是底层处理器相关的行为，perf 中会命名几个通用的事件，比如 cpu-cycles，执行完成的 instructions，Cache 相关的 cache-misses。

不同的处理器有自己不同的 PMU 事件, 对于 Intel x86 处理器, PMU 的使用和编程都可以在 “[Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B](#)” (Intel 架构的开发者手册) 里查到。

我们运行一下 `perf stat`, 就可以看到在这段时间里这些 Hardware event 发生的数目。

[复制代码](#)

```
1 # perf stat
2 ^C
3 Performance counter stats for 'system wide':
4
5          58667.77 msec cpu-clock           #    63.203 CPUs utilized
6          258666      context-switches      #    0.004 M/sec
7           2554       cpu-migrations         #    0.044 K/sec
8          30763       page-faults           #    0.524 K/sec
9      21275365299     cycles                  #    0.363 GHz
10     24827718023     instructions            #    1.17   insn per cycle
11     5402114113      branches                #   92.080 M/sec
12     59862316        branch-misses          #    1.11% of all branches
13
14          0.928237838 seconds time elapsed
```

Software event

Software event 是定义在 Linux 内核代码中的几个特定的事件, 比较典型的有进程上下文切换 (内核态到用户态的转换) 事件 `context-switches`、发生缺页中断的事件 `page-faults` 等。

为了让你更容易理解, 这里我举个例子。就拿 `page-faults` 这个 perf 事件来说, 我们可以看到, 在内核代码处理缺页中断的函数里, 就是调用了 `perf_sw_event()` 来注册了这个 `page-faults`。

[复制代码](#)

```
1 /*
2  * Explicitly marked noline such that the function tracer sees this as the
3  * page_fault entry point. __do_page_fault 是Linux内核处理缺页中断的主要函数
4  */
5 static noline void
6 __do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
```

```
7         unsigned long address)
8     {
9         prefetchw(&current->mm->mmap_sem);
10
11         if (unlikely(kmmio_fault(regs, address)))
12             return;
13
14         /* Was the fault on kernel-controlled part of the address space? */
15         if (unlikely(fault_in_kernel_space(address)))
16             do_kern_addr_fault(regs, hw_error_code, address);
17         else
18             do_user_addr_fault(regs, hw_error_code, address);
19         /* 在do_user_addr_fault()里面调用了perf_sw_event() */
20
21     }
22
23     /* Handle faults in the user portion of the address space */
24     static inline
25     void do_user_addr_fault(struct pt_regs *regs,
26                            unsigned long hw_error_code,
27                            unsigned long address)
28     {
29         ...
30         perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS, 1, regs, address);
31         ...
32     }
```

Tracepoints event

你可以在 `perf list` 中看到大量的 Tracepoints event, 这是因为内核中很多关键函数里都有 Tracepoints。它的实现方式和 Software event 类似, 都是在内核函数中注册了 event。

不过, 这些 tracepoints 不仅是用在 perf 中, 它已经是 Linux 内核 tracing 的标准接口了, ftrace, ebpf 等工具都会用到它, 后面我们还会再详细介绍 tracepoint。

好了, 讲到这里, 你要重点掌握的内容是, **event 是 perf 工作的基础, 主要有两种: 有使用硬件的 PMU 里的 event, 也有在内核代码中注册的 event。**

那么在这些 event 都准备好了之后, perf 又是怎么去使用这些 event 呢? 前面我也提到过, 有计数和采样两种方式, 下面我们分别来看看。

计数 (count)

计数的这种工作方式比较好理解, 就是统计某个 event 在一段时间里发生了多少次。

那具体我们怎么进行计数的呢? `perf stat` 这个命令就是来查看 event 的数目的, 前面我们已经运行过 `perf stat` 来查看所有的 Hardware events。

这里我们可以加上 "-e" 参数, 指定某一个 event 来看它的计数, 比如 page-faults, 这里我们看到在当前 CPU 上, 这个 event 在 1 秒钟内发生了 49 次:

[复制代码](#)

```
1 # perf stat -e page-faults -- sleep 1
2
3 Performance counter stats for 'sleep 1':
4
5          49          page-faults
6
7      1.001583032 seconds time elapsed
8
9      0.001556000 seconds user
10     0.000000000 seconds sys
```

采样 (sample)

说完了计数, 我们再来看看采样。在开头回顾网路延时问题的时候, 我提到通过 `perf record -C 32 -g -- sleep 10` 这个命令, 来找到 CPU32 上 CPU 开销最大的 Softirq 相关函数。这里使用的 `perf record` 命令就是通过采样来得到热点函数的, 我们分析一下它是怎么做的。

`perf record` 在不加 -e 指定 event 的时候, 它缺省的 event 就是 Hardware event cycles。我们先用 `perf stat` 来查看 1 秒钟 cycles 事件的数量, 在下面的例子里这个数量是 1878165 次。

我们可以想一下, 如果每次 cycles event 发生的时候, 我们都记录当时的 IP (就是处理器当时要执行的指令地址)、IP 所属的进程等信息的话, 这样系统的开销就太大了。所以 `perf` 就使用了对 event 采样的方式来记录 IP、进程等信息。

[复制代码](#)


```
1 # perf stat -e cycles -- sleep 1
2
```

```
3 Performance counter stats for 'sleep 1':
4
5              1878165      cycles
```

Perf 对 event 的采样有两种模式:

第一种是按照 event 的数目 (period) , 比如每发生 10000 次 cycles event 就记录一次 IP、进程等信息, perf record 中的 -c 参数可以指定每发生多少次, 就做一次记录。

比如在下面的例子里, 我们指定了每 10000 cycles event 做一次采样之后, 在 1 秒里总共就做了 191 次采样, 比我们之前看到 1 秒钟 1878165 次 cycles 的次数要少多了。

 复制代码

```
1 # perf record -e cycles -c 10000 -- sleep 1
2 [ perf record: Woken up 1 times to write data ]
3 [ perf record: Captured and wrote 0.024 MB perf.data (191 samples) ]
```

第二种是定义一个频率 (frequency) , perf record 中的 -F 参数就是指定频率的, 比如 perf record -e cycles -F 99 -- sleep 1 , 就是指采样每秒钟做 99 次。

在 perf record 运行结束后, 会在磁盘的当前目录留下 perf.data 这个文件, 里面记录了所有采样得到的信息。然后我们再运行 perf report 命令, 查看函数或者指令在这些采样里的分布比例, 后面我们会用一个例子说明。

好, 说到这里, 我们已经把 perf 的基本概念和使用机制都讲完了。接下来, 我们看看在容器中怎么使用 perf?

容器中怎样使用 perf?

如果你的 container image 是基于 Ubuntu 或者 CentOS 等 Linux 发行版的, 你可以尝试用它们的 package repo 安装 perf 的包。不过, 这么做可能会有个问题, 我们在前面介绍 perf 的时候提过, perf 是和 Linux kernel 一起发布的, 也就是说 perf 版本最好是和 Linux kernel 使用相同的版本。

如果容器中 perf 包是独立安装的, 那么容器中安装的 perf 版本可能会和宿主机上的内核版本不一致, 这样有可能导致 perf 无法正常工作。

所以, 我们在容器中需要跑 perf 的时候, 最好从相应的 Linux kernel 版本的源代码里去编译, 并且采用静态库 (-static) 的链接方式。然后, 我们把编译出来的 perf 直接 copy 到容器中就可以使用了。

如何在 Linux kernel 源代码里编译静态链接的 perf, 你可以参考后面的代码:

[复制代码](#)

```
1 # cd $(KERNEL_SRC_ROOT)/tools/perf
2 # vi Makefile.perf
3 ##### ADD "LDFLAGS=-static" in Makefile.perf
4 # make clean; make
5 # file perf
6 perf: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically lin
7 # ls -lh perf
8 -rwxr-xr-x 1 root root 19M Aug 14 07:08 perf
```

我这里给了一个带静态链接 perf (kernel 5.4) 的 container image [例子](#), 你可以运行 make image 来生成这个 image。

在容器中运行 perf, 还要注意一个权限的问题, 有两点注意事项需要你留意。

第一点, Perf 通过系统调用 perf_event_open() 来完成对 perf event 的计数或者采样。不过 Docker 使用 seccomp (seccomp 是一种技术, 它通过控制系统调用的方式来保障 Linux 安全) 会默认禁止 perf_event_open()。

所以想要让 Docker 启动的容器可以运行 perf, 我们要怎么处理呢?

其实这个也不难, 在用 Docker 启动容器的时候, 我们需要在 seccomp 的 profile 里, 允许 perf_event_open() 这个系统调用在容器中使用。在我们的例子中, 启动 container 的命令里, 已经加了这个参数允许了, 参数是 "--security-opt seccomp=unconfined"。

第二点, 需要允许容器在没有 SYS_ADMIN 这个 capability (Linux capability 我们在 [第 19 讲](#)说过) 的情况下, 也可以让 perf 访问这些 event。那么现在我们需要做的就是,

在宿主机上设置出 `echo -1 > /proc/sys/kernel/perf_event_paranoid`, 这样普通的容器里也能执行 perf 了。


完成了权限设置之后, 在容器中运行 perf, 就和在 VM/BM 上运行没有什么区别了。

最后, 我们再来说一下我们在定位 CPU Uage 异常时最常用的方法, 常规的步骤一般是这样的:

首先, 调用 `perf record` 采样几秒钟, 一般需要加 `-g` 参数, 也就是 `call-graph`, 还需要抓取函数的调用关系。在多核的机器上, 还要记得加上 `-a` 参数, 保证获取所有 CPU Core 上的函数运行情况。至于采样数据的多少, 在讲解 perf 概念的时候说过, 我们可以用 `-c` 或者 `-F` 参数来控制。

接着, 我们需要运行 `perf report` 读取数据。不过很多时候, 为了更加直观地看到各个函数的占比, 我们会用 `perf script` 命令把 `perf record` 生成的 `perf.data` 转化成分析脚本, 然后用 `FlameGraph` 工具来读取这个脚本, 生成火焰图。

下面这组命令, 就是刚才说过的使用 perf 的常规步骤:

 复制代码

```
1 # perf record -a -g -- sleep 60
2 # perf script > out.perf
3 # git clone --depth 1 https://github.com/brendangregg/FlameGraph.git
4 # FlameGraph/stackcollapse-perf.pl out.perf > out.folded
5 # FlameGraph/flamegraph.pl out.folded > out.svg
```

重点总结

我们这一讲学习了如何使用 perf, 这里我来给你总结一下重点。

首先, 我们在线上网络延时异常的那个实际例子中使用了 perf。我们发现可以用 perf 工具, 通过**抓取数据、数据读取和异常聚焦**这三个步骤的操作, 在 CPU Usage 增高的节点上找到具体引起 CPU 增高的函数。

之后我带你更深入地学习了 perf 是什么，它的工作方式是怎样的？这里我把 perf 的重点再给你强调一遍：

Perf 的实现基础是 event，有两大类，一类是基于硬件 PMU 的，一类是内核中的软件注册。而 Perf 在使用时的工作方式也是两大类，计数和采样。

先看一下计数，它执行的命令是 `perf stat`，用来查看每种 event 发生的次数；

采样执行的命令是 `perf record`，它可以使用 `period` 方式，就是每 N 个 event 发生后记录一次 event 发生时的 IP/ 进程信息，或者用 `frequency` 方式，每秒钟以固定次数来记录信息。记录的信息会存在当前目录的 `perf.data` 文件中。

如果我们要在容器中使用 perf，要注意这两点：

1. 容器中的 perf 版本要和宿主机内核版本匹配，可以直接从源代码编译出静态链接的 perf。
2. 我们需要解决两个权限的问题，一个是 `seccomp` 对系统调用的限制，还有一个是内核对容器中没有 `SYN_ADMIN` capability 的限制。

在我们日常分析系统性能异常的时候，使用 perf 最常用的方式是 `perf record` 获取采样数据，然后用 FlameGraph 工具来生成火焰图。

思考题

你可以在自己的一台 Linux 机器上运行一些带负载的程序，然后使用 perf 并且生成火焰图，看看开销最大的函数是哪一个。

欢迎在留言区分享你的疑惑和见解。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

12.12 大促

每日一课 VIP 年卡

10分钟，解决你的技术难题

¥159/年 ~~¥365/年~~每日一课
VIP 年卡**仅3天，【点击】图片，立即抢购 >>>**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐01 | 案例分析：怎么解决海量IPVS规则带来的网络延时抖动问题？

下一篇 加餐03 | 理解ftrace (1)：怎么应用ftrace查看长延时时核函数？

精选留言 (2)

[写留言](#)**徐少文**

2021-02-01

老师好，如果想在主机上做容器内进程的监控，直接在host上利用perf工具去获取容器的系统调用序列，这样的方法是可行的吗？

展开 ∨

作者回复: @徐少文

这样是可以的，在host pid namespace下可以看到容器中进程的pid, 你可以用perf trace对应的pid看到这个进程的系统调用。



closer
2021-02-01

这几章都比较底层，作为运维人员，需要前置学习那些知识点，很多知识点都是盲点