



下载APP



加餐06 | BCC：入门eBPF的前端工具

2021-02-10 李程远

容器实战高手课

[进入课程 >](#)**讲述：李程远**

时长 14:37 大小 13.39M



你好，我是程远。

今天是我们专题加餐的最后一讲，明天就是春节了，我想给还在学习的你点个赞。这里我先给你拜个早年，祝愿你牛年工作顺利，健康如意！


上一讲，我们学习了 eBPF 的基本概念，以及 eBPF 编程的一个基本模型。在理解了这些概念之后，从理论上来说，你就能自己写出 eBPF 的程序，对 Linux 系统上的一些问题做跟踪和调试了。



不过，从上一讲的例子里估计你也发现了，eBPF 的程序从编译到运行还是有些复杂。

为了方便我们用 eBPF 的程序跟踪和调试系统，社区有很多 eBPF 的前端工具。在这些前端工具中，BCC 提供了最完整的工具集，以及用于 eBPF 工具开发的 Python/Lua/C++ 的接口。那么今天我们就一起来看看，怎么使用 BCC 这个 eBPF 的前端工具。

如何使用 BCC 工具


 **BCC** (BPF Compiler Collection) 这个社区项目开始于 2015 年，差不多在内核中支持了 eBPF 的特性之后，BCC 这个项目就开始了。

BCC 的目标就是提供一个工具链，用于编写、编译还有内核加载 eBPF 程序，同时 BCC 也提供了大量的 eBPF 的工具程序，这些程序能够帮我们做 Linux 的性能分析和跟踪调试。

这里我们可以先尝试用几个 BCC 的工具，通过实际操作来了解一下 BCC。


大部分 Linux 发行版本都有 BCC 的软件包，你可以直接安装。比如我们可以在 Ubuntu 20.04 上试试，用下面的命令安装 BCC：

```
1 # apt install bpfcc-tools
```

 复制代码

安装完 BCC 软件包之后，你在 Linux 系统上就会看到多了 100 多个 BCC 的小工具（在 Ubuntu 里，这些工具的名字后面都加了 bpfcc 的后缀）：

```
1 # ls -l /sbin/*-bpfcc | more
2 -rwxr-xr-x 1 root root 34536 Feb  7  2020 /sbin/argdist-bpfcc
3 -rwxr-xr-x 1 root root  2397 Feb  7  2020 /sbin/bashreadline-bpfcc
4 -rwxr-xr-x 1 root root  6231 Feb  7  2020 /sbin/biolatency-bpfcc
5 -rwxr-xr-x 1 root root  5524 Feb  7  2020 /sbin/biosnoop-bpfcc
6 -rwxr-xr-x 1 root root  6439 Feb  7  2020 /sbin/biotop-bpfcc
7 -rwxr-xr-x 1 root root  1152 Feb  7  2020 /sbin/bitesize-bpfcc
8 -rwxr-xr-x 1 root root  2453 Feb  7  2020 /sbin/bpflist-bpfcc
9 -rwxr-xr-x 1 root root  6339 Feb  7  2020 /sbin/btrfsdist-bpfcc
10 -rwxr-xr-x 1 root root  9973 Feb  7  2020 /sbin/btrfs slower-bpfcc
11 -rwxr-xr-x 1 root root  4717 Feb  7  2020 /sbin/cachestat-bpfcc
12 -rwxr-xr-x 1 root root  7302 Feb  7  2020 /sbin/cachetop-bpfcc
13 -rwxr-xr-x 1 root root  6859 Feb  7  2020 /sbin/capable-bpfcc
14 -rwxr-xr-x 1 root root    53 Feb  7  2020 /sbin/cobjnew-bpfcc
15 -rwxr-xr-x 1 root root  5209 Feb  7  2020 /sbin/cpudist-bpfcc
```

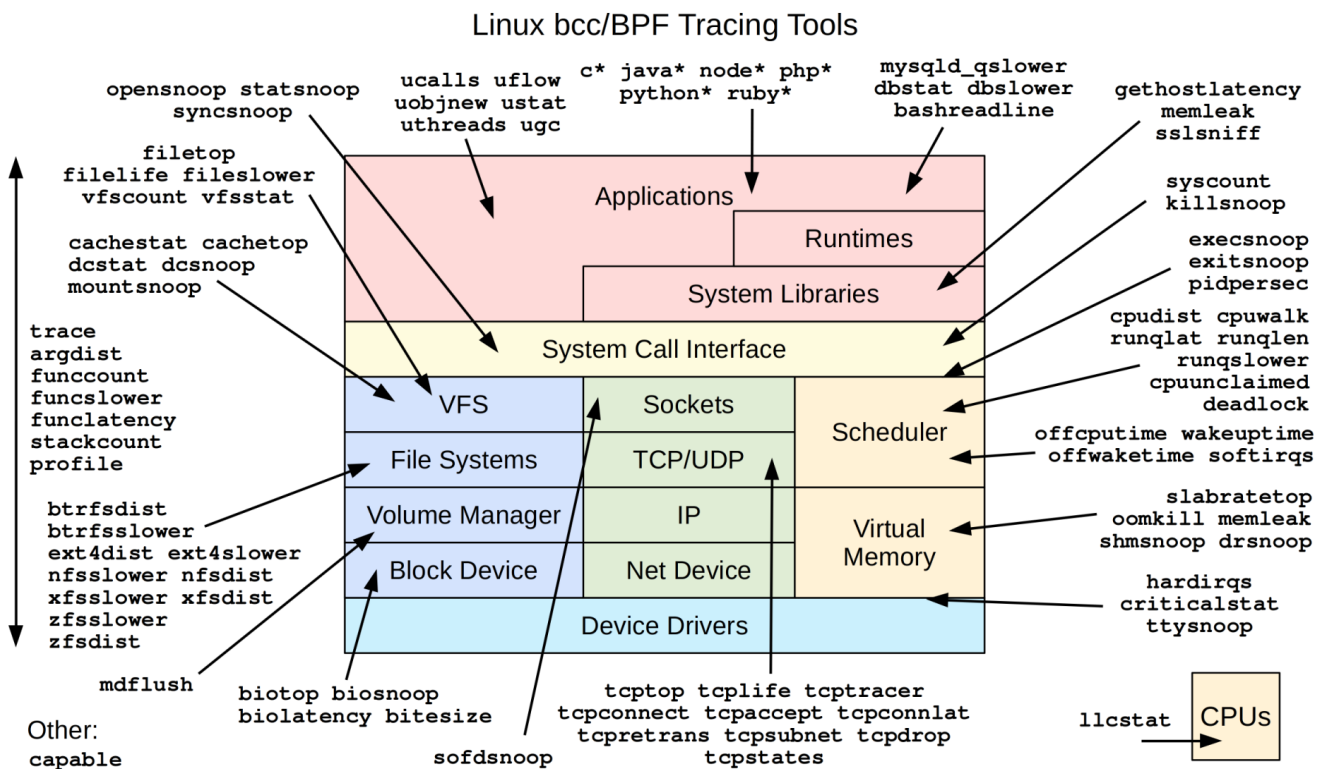
 复制代码

```

16 -rwxr-xr-x 1 root root 14597 Feb 7 2020 /sbin/cpuunclaimed-bpfcc
17 -rwxr-xr-x 1 root root 8504 Feb 7 2020 /sbin/criticalstat-bpfcc
18 -rwxr-xr-x 1 root root 7095 Feb 7 2020 /sbin/dbslower-bpfcc
19 -rwxr-xr-x 1 root root 3780 Feb 7 2020 /sbin/dbstat-bpfcc
20 -rwxr-xr-x 1 root root 3938 Feb 7 2020 /sbin/dcsnoop-bpfcc
21 -rwxr-xr-x 1 root root 3920 Feb 7 2020 /sbin/dcstat-bpfcc
22 -rwxr-xr-x 1 root root 19930 Feb 7 2020 /sbin/deadlock-bpfcc
23 -rwxr-xr-x 1 root root 7051 Dec 10 2019 /sbin/deadlock.c-bpfcc
24 -rwxr-xr-x 1 root root 6830 Feb 7 2020 /sbin/drsnoop-bpfcc
25 -rwxr-xr-x 1 root root 7658 Feb 7 2020 /sbin/execsnoop-bpfcc
26 -rwxr-xr-x 1 root root 10351 Feb 7 2020 /sbin/exitsnoop-bpfcc
27 -rwxr-xr-x 1 root root 6482 Feb 7 2020 /sbin/ext4dist-bpfcc
28

```

这些工具几乎覆盖了 Linux 内核中各个模块，它们可以对 Linux 某个模块做最基本的 profile。你可以看看下面 [这张图](#)，图里把 BCC 的工具与 Linux 中的各个模块做了一个映射。



在 BCC 的 github repo 里，也有很完整的 [文档和例子](#)来描述每一个工具。 [Brendan D. Gregg](#)写了一本书，书名叫《BPF Performance Tools》（我们上一讲也提到过这本书），这本书从 Linux CPU/Memory/Filesystem/Disk/Networking 等角度介绍了如何使用 BCC 工具，感兴趣的你可以自行学习。

为了让你更容易理解，这里我给你举两个例子。

第一个是使用 opensnoop 工具，用它来监控节点上所有打开文件的操作。这个命令有时候也可以用来查看某个文件被哪个进程给动过。

比如说，我们先启动 opensnoop，然后在其他的 console 里运行 touch test-open 命令，这时候我们就会看到 touch 命令在启动时读取到的库文件和配置文件，以及最后建立的“test-open”这个文件。

[复制代码](#)

```
1 # opensnoop-bpffcc
2 PID      COMM          FD ERR PATH
3 2522843 touch          3  0 /etc/ld.so.cache
4 2522843 touch          3  0 /lib/x86_64-linux-gnu/libc.so.6
5 2522843 touch          3  0 /usr/lib/locale/locale-archive
6 2522843 touch          3  0 /usr/share/locale/locale.alias
7 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_IDENTIFICATION
8 2522843 touch          3  0 /usr/lib/x86_64-linux-gnu/gconv/gconv-module
9 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_MEASUREMENT
10 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_TELEPHONE
11 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_ADDRESS
12 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_NAME
13 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_PAPER
14 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_MESSAGES
15 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_MESSAGES/SYS_LC_M
16 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_MONETARY
17 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_COLLATE
18 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_TIME
19 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_NUMERIC
20 2522843 touch          3  0 /usr/lib/locale/C.UTF-8/LC_CTYPE
21 2522843 touch          3  0 test-open
```

第二个是使用 softirqs 这个命令，查看节点上各种类型的 softirqs 花费时间的分布图（直方图模式）。

比如在下面这个例子里，每一次 timer softirq 执行时间在 0~1us 时间区间里的有 16 次，在 2-3us 时间区间里的有 49 次，以此类推。

在我们分析网络延时的时候，也用这个 softirqs 工具，用它来确认 timer softirq 花费的时间。

[复制代码](#)

```
1 # softirqs-bpffcc -d
```

```

2 Tracing soft irq event time... Hit Ctrl-C to end.
3 ^C
4
5 softirq = block
6      usecs          : count      distribution
7      0 -> 1          : 2          |*****|
8      2 -> 3          : 3          |*****|
9      4 -> 7          : 2          |*****|
10     8 -> 15         : 4          |*****|
11
12 softirq = rcu
13     usecs          : count      distribution
14     0 -> 1          : 189        |*****|
15     2 -> 3          : 52         |*****|
16     4 -> 7          : 21         |****|
17     8 -> 15         : 5          |*|
18     16 -> 31        : 1          |
19
20 softirq = net_rx
21     usecs          : count      distribution
22     0 -> 1          : 1          |*****|
23     2 -> 3          : 0          |
24     4 -> 7          : 2          |*****|
25     8 -> 15         : 0          |
26     16 -> 31        : 2          |*****|
27
28 softirq = timer
29     usecs          : count      distribution
30     0 -> 1          : 16         |*****|
31     2 -> 3          : 49         |*****|
32     4 -> 7          : 43         |*****|
33     8 -> 15         : 5          |****|
34     16 -> 31        : 13         |*****|
35     32 -> 63        : 13         |*****|
36
37 softirq = sched
38     usecs          : count      distribution
39     0 -> 1          : 18         |*****|
40     2 -> 3          : 107        |*****|
41     4 -> 7          : 20         |*****|
42     8 -> 15         : 1          |
43     16 -> 31        : 1          |

```

BCC 中的工具数目虽然很多，但是你用过之后就会发现，它们的输出模式基本上就是上面我说的这两种。

第一种类似事件模式，就像 opensnoop 的输出一样，发生一次就输出一行；第二种是直方图模式，就是把内核中执行函数的时间做个统计，然后用直方图的方式输出，也就是

softirqs -d 的执行结果。

用过 BCC 工具之后，我们再来看一下 BCC 工具的工作原理，这样以后你有需要的时候，自己也可以编写和部署一个 BCC 工具了。


BCC 的工作原理

让我们来先看一下 BCC 工具的代码结构。

因为目前 BCC 的工具都是用 python 写的，所以你直接可以用文本编辑器打开节点上的一个工具文件。比如打开 /sbin/opensnoop-bpfcc 文件（也可在 github bcc 项目中查看 [opensnoop.py](#)），这里你可以看到大概 200 行左右的代码，代码主要分成了两部分。


第一部分其实是一块 C 代码，里面定义的就是 eBPF 内核态的代码，不过它是以 python 字符串的形式加在代码中的。

我在下面列出了这段 C 程序的主干，其实就是定义两个 eBPF Maps 和两个 eBPF Programs 的函数：

 复制代码

```
1 # define BPF program
2 bpf_text = """
3 #include <uapi/linux/ptrace.h>
4 #include <uapi/linux/limits.h>
5 #include <linux/sched.h>
6
7 ...
8
9 BPF_HASH(infotmp, u64, struct val_t); //BPF_MAP_TYPE_HASH
10 BPF_PERF_OUTPUT(events);             // BPF_MAP_TYPE_PERF_EVENT_ARRAY
11
12 int trace_entry(struct pt_regs *ctx, int dfd, const char __user *filename, int
13 {
14 ...
15 }
16
17 int trace_return(struct pt_regs *ctx)
18 {
19 ...
20 }
21 """
```


第二部分就是用 python 写的用户态代码，它的作用是加载内核态 eBPF 的代码，把内核态的函数 `trace_entry()` 以 `kprobe` 方式挂载到内核函数 `do_sys_open()`，把 `trace_return()` 以 `kproberet` 方式也挂载到 `do_sys_open()`，然后从 eBPF Maps 里读取数据并且输出。

 复制代码

```
1  ...
2  # initialize BPF
3  b = BPF(text=bpf_text)
4  b.attach_kprobe(event="do_sys_open", fn_name="trace_entry")
5  b.attach_kretprobe(event="do_sys_open", fn_name="trace_return")
6  ...
7  # loop with callback to print_event
8  b["events"].open_perf_buffer(print_event, page_cnt=64)
9  start_time = datetime.now()
10 while not args.duration or datetime.now() - start_time < args.duration:
11     try:
12         b.perf_buffer_poll()
13     except KeyboardInterrupt:
14         exit()
15 ...
```

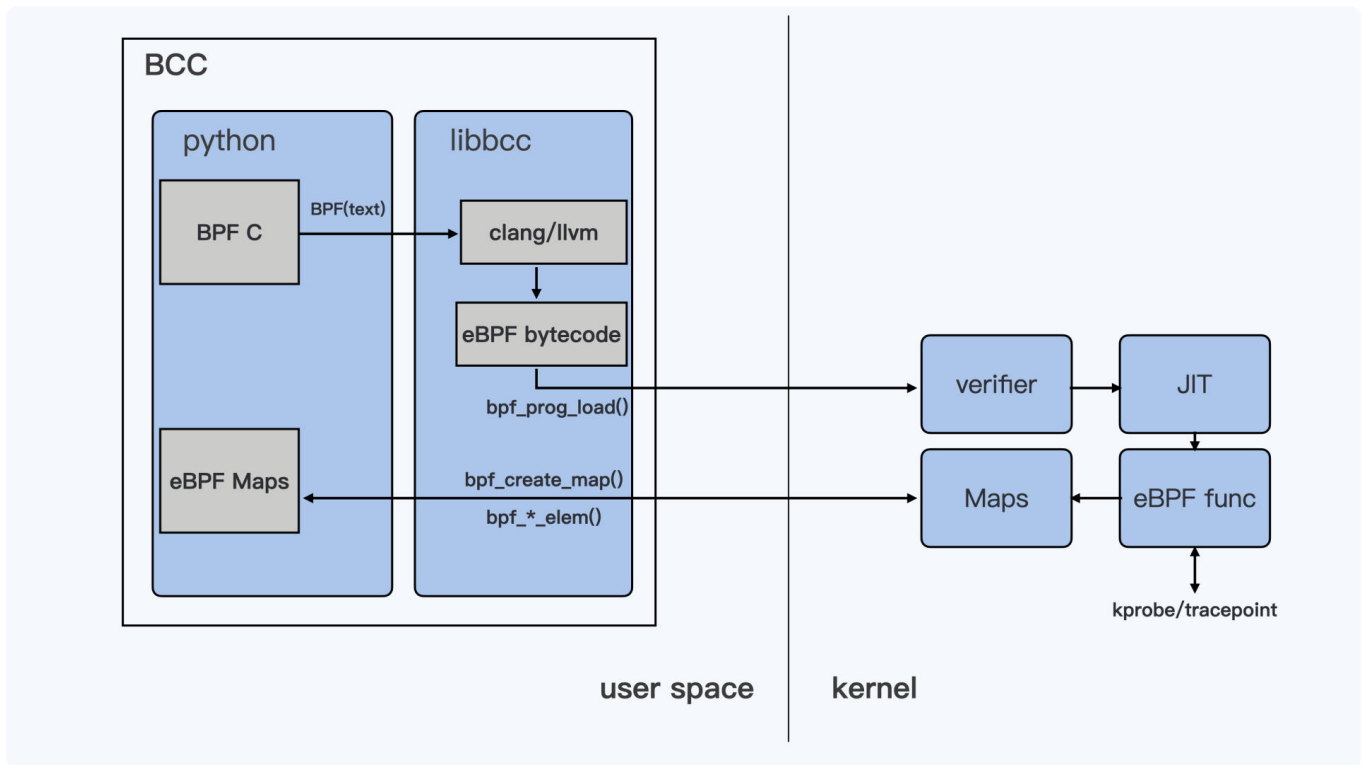
从代码的结构看，其实这和我们上一讲介绍的 eBPF 标准的编程模式是差不多的，只是用户态的程序是用 python 来写的。不过这里有一点比较特殊，用户态在加载程序的时候，输入的是 C 程序的文本而不是 eBPF bytecode。

BCC 可以这么做，是因为它通过 python `BPF()` 加载 C 代码之后，调用 `libbcc` 库中的函数 `bpf_module_create_c_from_string()` 把 C 代码编译成了 eBPF bytecode。也就是说，`libbcc` 库中集成了 `clang/llvm` 的编译器。

 复制代码

```
1  def __init__(self, src_file=b"", hdr_file=b"", text=None, debug=0,
2              cflags=[], usdt_contexts=[], allow_rlimit=True, device=None):
3      """Create a new BPF module with the given source code.
4  ...
5              self.module = lib.bpf_module_create_c_from_string(text, self.debug
6  ...
```

我们弄明白 `libbcc` 库的作用之后，再来整体看一下 BCC 工具的工作方式。为了让你理解，我给你画了一张示意图：



BCC 的这种设计思想是为了方便 eBPF 程序的开发和使用，特别是 eBPF 内核态的代码对当前运行的内核版本是有依赖的，比如在 4.15 内核的节点上编译好的 bytecode，放到 5.4 内核的节点上很有可能是运行不了的。

那么让编译和运行都在同一个节点，出现问题就可以直接修改源代码文件了。你有没有发现，这么做有点像把 C 程序的处理当成 python 的处理方式。

BCC 的这种设计思想虽然有好处，但是也带来了问题。其实问题也是很明显的，首先我们需要在运行 BCC 工具的节点上必须安装内核头文件，这个在编译内核态 eBPF C 代码的时候是必须要做的。

其次，在 libbcc 的库里面包含了 clang/llvm 的编译器，这不光占用磁盘空间，在运行程序前还需要编译，也会占用节点的 CPU 和 Memory，同时也让 BCC 工具的启动时间变长。这两个问题都会影响到 BCC 生产环境中的使用。

BCC 工具的发展

那么我们有什么办法来解决刚才说的这个问题呢？eBPF 的技术在不断进步，最新的 BPF CO-RE 技术可以解决这个问题。我们下面就来看 BPF CO-RE 是什么意思。

CO-RE 是 “Compile Once – Run Everywhere” 的缩写，BPF CO-RE 通过对 Linux 内核、用户态 BPF loader (libbpf 库) 以及 Clang 编译器的修改，来实现编译出来的 eBPF 程序可以在不同版本的内核上运行。

不同版本的内核上，用 CO-RE 编译出来的 eBPF 程序都可以运行。在 Linux 内核和 BPF 程序之间，会通过 [BTF](#) (BPF Type Format) 来协调不同版本内核中数据结构的变量偏移或者变量长度变化等问题。

在 BCC 的 github repo 里，有一个目录 [libbpf-tools](#)，在这个目录下已经有一些重写过的 BCC 工具的源代码，它们并不是用 python+libbcc 的方式实现的，而是用到了 libbpf+BPF CO-RE 的方式。

如果你的系统上有高于版本 10 的 CLANG/LLVM 编译器，就可以尝试编译一下 libbpf-tools 下的工具。这里可以加一个 “V=1” 参数，这样我们就能清楚编译的步骤了。

[复制代码](#)

```
1 # git remote -v
2 origin https://github.com/iovisor/bcc.git (fetch)
3 origin https://github.com/iovisor/bcc.git (push)
4 # cd libbpf-tools/
5 # make V=1
6 mkdir -p .output
7 mkdir -p .output/libbpf
8 make -C /root/bcc/src/cc/libbpf/src BUILD_STATIC_ONLY=1
9         OBJDIR=/root/bcc/libbpf-tools/.output//libbpf DESTDIR=/root/bc
10         INCLUDEDIR= LIBDIR= UAPIDIR=
11         Install
12 ...
13
14 ar rcs /root/bcc/libbpf-tools/.output//libbpf/libbpf.a ...
15
16 ...
17
18 clang -g -O2 -target bpf -D__TARGET_ARCH_x86 \
19         -I.output -c opensnoop.bpf.c -o .output/opensnoop.bpf.o &&
20 llvm-strip -g .output/opensnoop.bpf.o
21 bin/bpftool gen skeleton .output/opensnoop.bpf.o > .output/opensnoop.skel.h
22 cc -g -O2 -Wall -I.output -c opensnoop.c -o .output/opensnoop.o
23 cc -g -O2 -Wall .output/opensnoop.o /root/bcc/libbpf-tools/.output/libbpf.a .o
24
25 ...
```

我们梳理一下编译的过程。首先这段代码生成了 libbpf.a 这个静态库，然后逐个的编译每一个工具。对于每一个工具的代码结构是差不多的，编译的方法也是差不多的。

我们拿 opensnoop 做例子来看一下，它的源代码分为两个文件。opensnoop.bpf.c 是内核态的 eBPF 代码，opensnoop.c 是用户态的代码，这个和我们之前学习的 eBPF 代码的标准结构是一样的。主要不同点有下面这些。

内核态的代码不再逐个 include 内核代码的头文件，而是只要 include 一个 “vmlinux.h” 就可以。在 “vmlinux.h” 中包含了所有内核的数据结构，它是由内核文件 vmlinux 中的 BTF 信息转化而来的。

[复制代码](#)

```
1 # cat opensnoop.bpf.c | head
2 // SPDX-License-Identifier: GPL-2.0
3 // Copyright (c) 2019 Facebook
4 // Copyright (c) 2020 Netflix
5 #include "vmlinux.h"
6 #include <bpf/bpf_helpers.h>
7 #include "opensnoop.h"
8
9 #define TASK_RUNNING 0
10
11 const volatile __u64 min_us = 0;
```

我们使用 [bpftool](#) 这个工具，可以把编译出来的 opensnoop.bpf.o 重新生成成为一个 C 语言的头文件 opensnoop.skel.h。这个头文件中定义了加载 eBPF 程序的函数，eBPF bytecode 的二进制流也直接写在了这个头文件中。

[复制代码](#)

```
1 bin/bpftool gen skeleton .output/opensnoop.bpf.o > .output/opensnoop.skel.h
```

用户态的代码 opensnoop.c 直接 include 这个 opensnoop.skel.h，并且调用里面的 eBPF 加载的函数。这样在编译出来的可执行程序 opensnoop，就可以直接运行了，不用再找 eBPF bytecode 文件或者 eBPF 内核态的 C 文件。并且这个 opensnoop 程序可以运行在不同版本内核的节点上（当然，这个内核需要打开 CONFIG_DEBUG_INFO_BTF 这个编译选项）。

比如，我们可以把在 kernel5.4 节点上编译好的 opensnoop 程序 copy 到一台 kernel5.10.4 的节点来运行：

[复制代码](#)

```
1 # uname -r
2 5.10.4
3 # ls -lh opensnoop
4 -rwxr-x--- 1 root root 235K Jan 30 23:08 opensnoop
5 # ./opensnoop
6 PID      COMM          FD ERR PATH
7 2637411 opensnoop      24  0 /etc/localtime
8 1        systemd       28  0 /proc/746/cgroup
```

从上面的代码我们会发现，这时候的 opensnoop 不依赖任何的库函数，只有一个文件，strip 后的文件大小只有 235KB，启动运行的时候，既不需要读取外部的文件，也不会做额外的编译。

重点小结

好了，今天我们主要讲了 eBPF 的一个前端工具 BCC，我来给你总结一下。

在我看来，对于把 eBPF 运用于 Linux 内核的性能分析和跟踪调试这个领域，BCC 是社区中最有影响力的一个项目。BCC 项目提供了 eBPF 工具开发的 Python/Lua/C++ 的接口，以及上百个基于 eBPF 的工具。

对不熟悉 eBPF 的同学来说，可以直接拿这些工具来调试 Linux 系统中的问题。而对于了解 eBPF 的同学，也可以利用 BCC 提供的接口，开发自己需要的 eBPF 工具。

BCC 工具目前主要通过 python+libbcc 的模式在目标节点上运行，但是这个模式需要节点有内核头文件以及内嵌在 libbcc 中的 Clang/LLVM 编译器，每次程序启动的时候还需要再做一次编译。

为了弥补这个缺点，BCC 工具开始向 libbpf+BPF CO-RE 的模式转变。用这种新模式编译出来的 BCC 工具程序，只需要很少的系统资源就可以在目标节点上运行，并且不受内核版本的限制。

除了 BCC 之外，你还可以看一下 [bpfftrace](#)、[ebpf-exporter](#) 等 eBPF 的前端工具。

bpftrace 提供了类似 awk 和 C 语言混合的一种语言，在使用时也很类似 awk，可以用一两行的命令来完成一次 eBPF 的调用，它能做一些简单的内核事件的跟踪。当然它也可以编写比较复杂的 eBPF 程序。

ebpf-exporter 可以把 eBPF 程序收集到的 metrics 以 [Prometheus](#) 的格式对外输出，然后通过 [Grafana](#) 的 dashboard，可以对内核事件做长期的以及更加直观的监控。

总之，前面提到的这些工具，你都可以好好研究一下，它们可以帮助你容器云平台上的节点做内核级别的监控与诊断。

思考题

这一讲的最后，我给你留一道思考题吧。

你可以动手操作一下，尝试编译和运行 BCC 项目中 [libbpf-tools](#) 目录下的工具。

欢迎你在留言区记录你的心得或者疑问。如果这一讲对你有帮助，也欢迎分享给你的同事、朋友，和他一起学习进步。

提建议

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05 | eBPF：怎么更加深入地查看内核中的函数？

精选留言 (4)

写留言



莫名

2021-02-18

老师提到过一个比较好用的网络包追踪工具：<https://github.com/yadutaf/tracepkt>。

不过这个是基于 BCC 编写的，阅读起来有些晦涩。假期抽空编写了一个简单的 bpfftrace 版本（修复了 network ns 为 0、interface 为空的问题，不过没有根据 pid 过滤），供大家参考下：<https://github.com/xingfeng2510/mybpf/blob/main/tracepkt.bt...>

展开 ∨



我来也

2021-02-12

祝老师新年快乐！

展开 ∨



Joe Black

2021-02-11

春节快乐！学完课程后，感觉终究还是有必要认认真真看看内核的数据，还是需要系统化的学习下。



无名氏

2021-02-10

春节快乐

展开 ∨

