**性能工具**

**工具名词**

cuda

DPDK

CDN

xdp

Mindump

kprobe

perfetto

Heaptrack

ptrace

1. **性能工具**

1.1 **perfetto**

https://www.jianshu.com/p/f4cf101cc64f

https://blog.csdn.net/qrx941017/article/details/128984365

Perfetto 是一个用于性能分析的开源项目，它支持从 Android、Linux 和 Chrome OS（以及其他基于 Linux 的平台）采集系统级和应用级的性能数据。Perfetto 设计用于记录和分析系统性能问题，如 CPU、内存、磁盘 I/O 和网络使用情况。它特别适合理解复杂的系统级问题，并支持长时间的跟踪能力，以便洞察低概率发生的事件。

Perfetto 包括几个关键部分：

1. 收集器（Tracer）：这是在目标系统上运行的一部分，负责实际收集跟踪数据。
2. 后端服务（Trace Service）：在 Android 等平台上，这是一个后台服务，由收集器将跟踪数据发送给它。
3. 前端界面（UI）：提供交互式界面供用户查看和分析跟踪数据。
4. 跟踪配置和控制（Trace Config & Control）：用于配置跟踪会话的参数，如要记录的性能计数器、跟踪的持续时间等。

Perfetto 提供了强大的 Web UI，可以通过 https://ui.perfetto.dev/ 访问，该界面允许用户上传和分析跟踪文件。用户可以在 UI 中做很多事情，包括查看线程的 CPU 使用情况、系统调用的情况、内存分配以及其他各种系统和应用级别的性能指标。

Perfetto 特别受到 Android 开发者的青睐，因为它被集成在了 Android 平台中，但它的设计使其足够通用，可以应用于任何 Linux 基础的系统。

1.2 **Benchmark**

https://github.com/topics/google-benchmark

https://blog.csdn.net/study\_code\_ing/article/details/124679687?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522170893160216800185887976%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=170893160216800185887976&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-124679687-null-null.142^v99^pc\_search\_result\_base3&utm\_term=benchmark&spm=1018.2226.3001.4187

1.3 **ebpf**

https://gitcode.com/eunomia-bpf/bpf-developer-tutorial/tree/main/

[eBPF内核可观测性技术介绍](https://li.feishu.cn/wiki/wikcnyX2pkIbhdZvHi6ePjZxUdh)

原理：

eBPF（Extended Berkeley Packet Filter）是一个强大的技术，允许用户态代码在内核空间安全地运行，提供了对系统的深度观察和高级网络功能而不需要改变内核源码或加载内核模块。

最初作为一种更高效的数据包过滤解决方案而设计，eBPF已经发展成为Linux内核中一个通用的运行时，允许程序在内核中运行，执行包括网络相关操作、性能监控、系统调用监控、安全性增强等多种任务。

一些特点和使用场景包括：

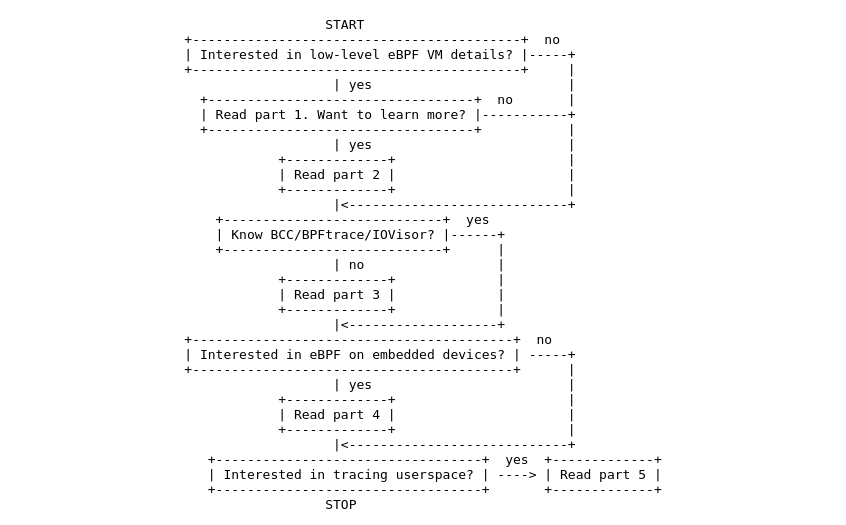
1. 安全性：eBPF程序在加载到内核前要通过严格的验证器（Verifier）检查，确保其不会崩溃系统或执行不安全的操作。
2. 性能监控：利用eBPF可以创建强大的系统分析工具来跟踪和监视系统调用、内核函数和网络事件等，工具如bpftrace和BCC（BPF Compiler Collection）使得编写eBPF跟踪工具更加容易。
3. 网络功能：eBPF支持开发高级网络功能，如软件定义网络（SDN）和网络功能虚拟化（NFV）。此外，它还可以用于构建复杂的网络监控和过滤系统。
4. 容器监控：在容器化的环境中，eBPF能够监控和过滤容器网络流量，从而提供更好的隔离和安全性。
5. 可编程性：eBPF的一个关键特性是允许开发者编写在运行时定义系统的行为的代码，而不需要修改内核源代码。

为了编写和部署eBPF程序，一般的流程是：

* 使用C语言编写eBPF代码。
* 利用LLVM将eBPF C代码编译成BPF字节码。
* 使用特殊的BPF系统调用，将编译后的eBPF字节码加载到内核。
* eBPF程序被绑定到某个内核事件，例如网络数据包到达、系统调用进入或退出、进程调度等。

值得注意的是，eBPF是Linux特有的，需要较新版本的Linux内核才能使用其全部特性。开发人员可以通过像iproute2、cilium等工具来管理eBPF程序的加载和卸载，以及监测和调试eBPF程序。

使用：



|  |
| --- |
| C++ ## 工具安装步骤  ```console $ wget https://aka.pw/bpf-ecli -O ecli && chmod +x ./ecli $ ./ecli -h Usage: ecli [--help] [--version] [--json] [--no-cache] url-and-args ```  ```console $ wget https://github.com/eunomia-bpf/eunomia-bpf/releases/latest/download/ecc && chmod +x ./ecc $ ./ecc -h eunomia-bpf compiler Usage: ecc [OPTIONS] <SOURCE\_PATH> [EXPORT\_EVENT\_HEADER] ....  ```shell sudo apt install clang llvm sudo apt install fuse3 ```  使用 ecc 编译程序：  ```console $ ./ecc minimal.bpf.c Compiling bpf object... Packing ebpf object and config into package.json... ```  然后使用 ecli 运行编译后的程序：  ```console $ sudo ./ecli run package.json Runing eBPF program... ```  ## fentry-link.bpf.c 使用demo说明  代码 ```c #include "vmlinux.h" #include <bpf/bpf\_helpers.h> #include <bpf/bpf\_tracing.h>  char LICENSE[] SEC("license") = "Dual BSD/GPL";  SEC("fentry/do\_unlinkat") int BPF\_PROG(do\_unlinkat, int dfd, struct filename \*name) {  pid\_t pid;   pid = bpf\_get\_current\_pid\_tgid() >> 32;  bpf\_printk("fentry: pid = %d, filename = %s\n", pid, name->name);  return 0; }  SEC("fexit/do\_unlinkat") int BPF\_PROG(do\_unlinkat\_exit, int dfd, struct filename \*name, long ret) {  pid\_t pid;   pid = bpf\_get\_current\_pid\_tgid() >> 32;  bpf\_printk("fexit: pid = %d, filename = %s, ret = %ld\n", pid, name->name, ret);  return 0; } ```   1、do\_unlinkat 是 Linux 内核中执行文件删除的底层函数。   2、SEC("fexit/do\_unlinkat") 说明程序应挂载在内核的 do\_unlinkat 函数返回时 eBPF 程序的部分和挂载点。   3、BPF\_PROG 是一个宏，用于声明一个 eBPF 程序，do\_unlinkat\_exit 是此 eBPF 程序的名字。此程序将在 do\_unlinkat 返回时被调用，  并且提供了通过 do\_unlinkat 的参数：目录文件描述符 dfd，文件名结构体指针 name，以及系统调用的返回值 ret。    4、在另外一个窗口中：  ```shell touch test\_file rm test\_file touch test\_file2 rm test\_file2 ```  运行这段程序后，可以通过查看 `/sys/kernel/debug/tracing/trace\_pipe` 文件来查看 eBPF 程序的输出：  ```console $ sudo cat /sys/kernel/debug/tracing/trace\_pipe  rm-9290 [004] d..2 4637.798698: bpf\_trace\_printk: fentry: pid = 9290, filename = test\_file  rm-9290 [004] d..2 4637.798843: bpf\_trace\_printk: fexit: pid = 9290, filename = test\_file, ret = 0  rm-9290 [004] d..2 4637.798698: bpf\_trace\_printk: fentry: pid = 9290, filename = test\_file2  rm-9290 [004] d..2 4637.798843: bpf\_trace\_printk: fexit: pid = 9290, filename = test\_file2, ret = 0 ``` |