浪潮信息 | OpenAnolis 龙蜥社区 | 阿里云

# eBPF技术实践

## 白皮书

（第二版）

浪潮信息龙蜥联合实验室

## 编写说明

**编写单位**：浪潮电子信息产业股份有限公司、济南浪潮数据技术有限公司、阿里云计算有限公司、东南大学

**参编组织**：龙蜥社区

**贡献专家**：

苏志远 浪潮电子信息产业股份有限公司

吴栋 济南浪潮数据技术有限公司

方浩 济南浪潮数据技术有限公司

甄鹏 浪潮电子信息产业股份有限公司

王传国 浪潮电子信息产业股份有限公司

梁媛 浪潮电子信息产业股份有限公司

黄吉旺 济南浪潮数据技术有限公司

彭彬彬 济南浪潮数据技术有限公司

毛文安 阿里云计算有限公司

程书意 阿里云计算有限公司

廖肇燕 阿里云计算有限公司

李光水 阿里云计算有限公司

冯富秋 阿里云计算有限公司

卢烈 阿里云计算有限公司

沈典 东南大学

杨彬 东南大学

# 前言

eBPF 的诞生是 BPF 技术的一个转折点，使得 BPF 不再仅限于网络栈，而是成为内核的一个顶级子系统。在内核发展的同时，eBPF 繁荣的生态也进一步促进了 eBPF 技术的蓬勃发展。随着内核的复杂性不断增加，eBPF 以安全、稳定、零侵入、方便的内核可编程性等特点成为实现内核定制与功能多样性的最佳选择，为内核提供了应对未来挑战的基础。

本白皮书重点介绍 eBPF 技术的概念、技术实践以及发展趋势。本书首先梳理了 eBPF 的架构和重要技术原理，然后分析了 eBPF 在多种典型应用场景的使用方案，并进一步对 eBPF 技术的发展趋势做了探讨。

# 目录

1 eBPF 简介 ........ 8

2 eBPF 技术介绍 ........ 9

2.1 eBPF 架构 ........ 9

2.1.1 eBPF 加载过程 ........ 10

2.1.2 JIT 编译 ........ 11

2.1.3 挂载与执行 ........ 12

2.2 eBPF 常见的开发框架 ........ 19

2.2.1 BCC ........ 19

2.2.2 bpfTrace ........ 20

2.2.3 libbpf ........ 20

2.2.4 libbpf-bootstrap ........ 21

2.2.5 cilium-ebpf ........ 21

2.2.6 Coolbpf ........ 21

3 基于 eBPF 的技术创新与应用实践 ........ 25

3.1 基于 eBPF 的系统诊断 ........ 25

3.1.1 系统诊断面临挑战 ........ 25

3.1.2 基于 eBPF 的系统诊断方案........................................................ 30

3.1.3 基于 eBPF 的 Profiling........................................................ 36

3.2 基于 eBPF 的虚拟化 IO 全链路时延监测 ........................................................ 44

3.2.1 虚拟化 IO 全路径分析主要面临的挑战........................................................ 44

3.2.2 基于 bpftrace 虚拟化 IO 路径追踪解决方案........................................................ 45

3.3 基于 eBPF 的 TCP 监控 ........................................................ 50

3.3.1 TCP 监控面临的挑战........................................................ 50

3.3.2 基于 eBPF 的 TCP 监控方案 ........................................................ 52

3.4 基于 eBPF 的网络性能优化........................................................ 55

3.4.1 Linux 网络性能优化面临的挑战........................................................ 55

3.4.2 基于 eBPF 的 Linux 内核网络性能优化解决方案........................................................ 58

3.5 基于 eBPF 的流量镜像........................................................ 67

3.5.1 传统流量镜像面临的挑战 ........................................................ 67

3.5.2 基于 eBPF 的流量镜像解决方案........................................................ 67

3.5.3 应用实践........................................................ 69

3.6 网络访问控制 ........................................................ 70

3.6.1 传统网络访问控制面临的挑战 ........................................................ 70

3.6.2 基于 eBPF 的网络访问控制解决方案........................................ 71

3.6.3 应用实践........................................................................................ 72

3.7 优化基于 eBPF 的软件网络功能........................................................... 73

3.7.1 基于 eBPF 实现网络功能的优势................................................. 73

3.7.2 eBPF 实现网络功能面临的技术挑战.......................................... 74

3.7.3 基于标准库的优化 eBPF 网络功能技术方案............................. 76

3.8 基于 eBPF 的安全实践......................................................................... 83

3.8.1 传统解决方案面临挑战................................................................ 83

3.8.2 基于 eBPF 的新一代安全解决方案............................................. 84

4 挑战与展望.............................................................................................. 98

# 1 eBPF 简介

eBPF 是一项起源于 Linux 内核的革命性技术，可以在特权上下文中(如操作系统内核)运行沙盒程序。它可以安全有效地扩展内核的功能，并且不需要更改内核源代码或加载内核模块。

内核因具有监督和控制整个系统的特权，一直是实现可观察性、安全性和网络功能的理想场所。同时，由于对稳定性和安全性的高要求，内核发展相对缓慢，与内核之外实现的功能相比，创新速度较慢。

eBPF 从根本上改变了上述情况，它允许在内核中运行沙箱程序，即通过运行 eBPF 程序向正在运行中的操作系统添加额外的功能，并通过验证引擎和即时编译器保证安全性和执行效率，由此衍生出了一系列的产品和项目，涉及下一代网络、可观察性和安全技术。

如今，eBPF 在多种应用场景中起到重要作用：在现代数据中心和云原生环境中提供高性能网络和负载均衡，以低开销提取细粒度的安全可观察性数据，帮助应用程序开发人员跟踪应用程序的运行状态，高效进行性能故障定位，保证应用程序和容器运行时安全等。

eBPF 引领的创新才刚刚开始，一切皆有可能。

# 2 eBPF 技术介绍

## 2.1 eBPF 架构

eBPF 包括用户空间程序和内核程序两部分，用户空间程序负责加载 BPF 字节码至内核，内核中的 BPF 程序负责在内核中执行特定事件，用户空间程序与内核 BPF 程序可以使用 map 结构实现双向通信，这为内核中运行的 BPF 程序提供了更加灵活的控制。eBPF 的工作逻辑如下：

1、eBPF Program 通过 LLVM/Clang 编译成 eBPF 定义的字节码；

2、通过系统调用 bpf()将字节码指令传入内核中；

3、由 Verifier 检验字节码的安全性、合规性；

4、在确认字节码程序的安全性后，JIT Compiler 会将其转换成可以在当前系统运行的机器码；

5、根据程序类型的不同，把可运行机器码挂载到内核不同的位置/HOOK 点；

6、等待触发执行，其中不同的程序类型、不同的 HOOK 点的触发时机是不相同的；并且在 eBPF 程序运行过程中，用户空间可通过 eBPF map 与内核进行双向通信。

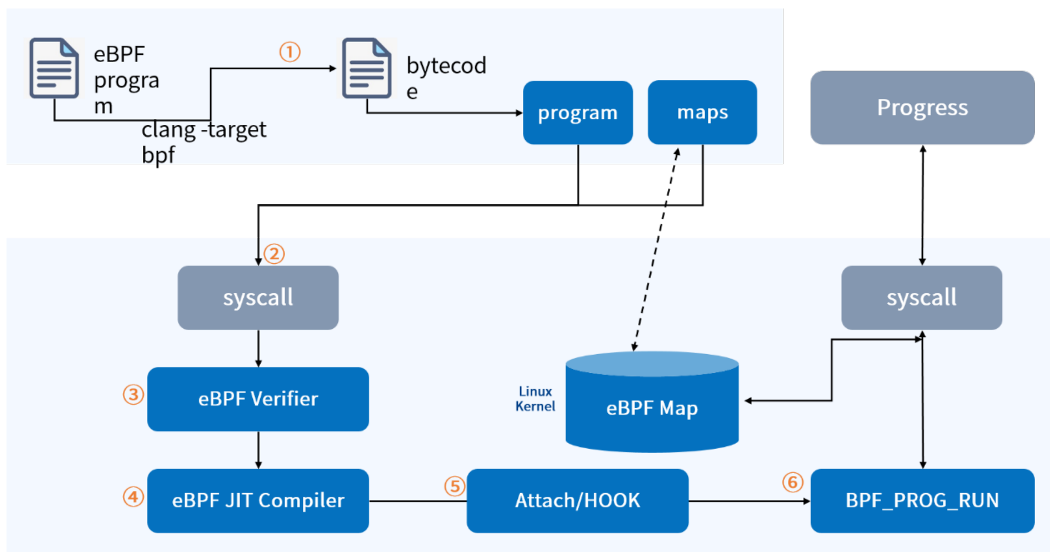


图 2-1-1 eBPF 基本架构与使用示意图

### 2.1.1 eBPF 加载过程

编译得到的 BPF 字节码文件，经过字节码读取、重定位和 Verifier 等过程，加载到内核中。

**1、字节码读取**

通常 eBPF 字节码文件都是 ELF 格式，各 section 中保存着字节码所有的信息，包括字节码指令、map 定义、BTF 信息，以及需要重定位的变量和函数等；eBPF 加载时，依照 ELF 格式读取字节码文件，把各种信息按照一定的格式保存起来。

**2、重定位**

重定位是指在编译、加载的过程中把字节码中一些临时数据以更准确的信息进行替换，比如用 map 句柄替换 map 索引，用 map 地址替换 map 句柄。经过多轮重定位，可以确保 eBPF 程序在内核中运行所需数据的正确性和完整性。需要重定位的对象有：map、函数调用、Helper 函数调用、字段、Extern 内核符号和 kconfig。

重定位操作主要分为 2 类：

1) BPF 基础设施提供了一组有限的“稳定接口”，通过 convert_ctx_access 对 CTX 进行转换，实现内核版本升级时的稳定性和兼容性。

2）CO-RE 采用(BTF)非硬编码的形式对成员在结构中的偏移位置进行描述，解决不同版本之间的差异性问题。

**3、Verifier**

Verifier 是一个静态代码安全检查器，用于检查 eBPF 程序的安全性，保证 eBPF 程序不会破坏内核，影响内核的稳定性。

安全检查主要分成两个阶段。

第一个阶段检查函数的深度和指令数，通过使用深度优先遍历，来检查是否为有向无环图（DAG）。

第二个阶段检查每条 bytecode 指令，根据指令的分类（class），检查其操作寄存器的读写属性、内存访问是否越界、BPF_CALL 是否符合接口协议等。

指针的安全性检查在第二个阶段实现，每次把指针加载到寄存器时都会进行指针类型的判定，根据指针类型定义确定读写属性和大小，实现针对指针操作的安全性检查；未识别的指针类型不允许解引用。

### 2.1.2 JIT 编译

执行字节码是一个模拟 CPU 执行机器码的过程，在运行时需要先把指令依次翻译成机器码之后才能运行，所以比机器码的执行效率低很多。JIT（Just In Time）的

中文意思是即时编译，主要为了解决虚拟机运行中间码时效率不高的问题。

JIT 编译在执行前先把中间码编译成对应的机器码并缓存起来，从而运行时能够直接执行机器码，这样就解决了每次执行都需要进行中间码解析的问题，如下图所示：

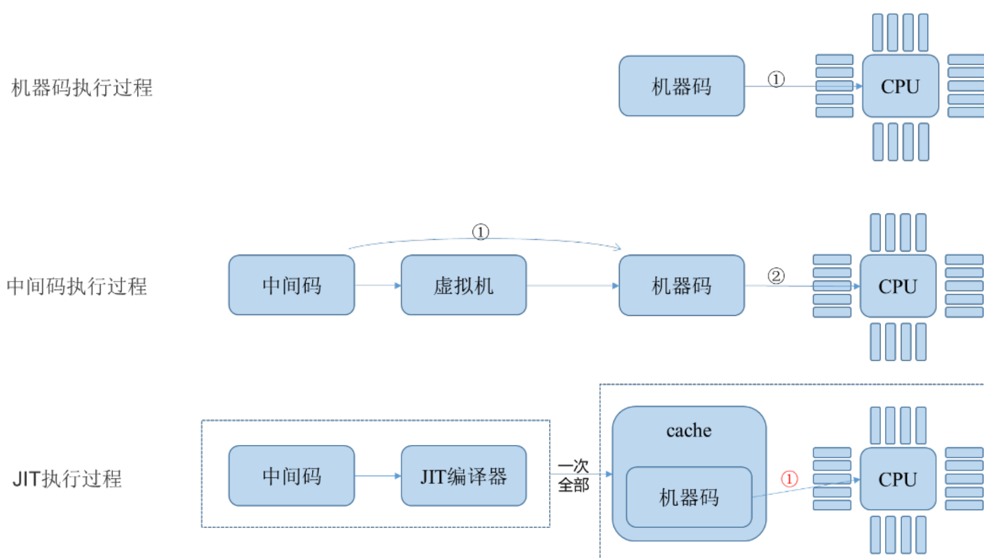


图 2-1-2 JIT 的作用示意图

### 2.1.3 挂载与执行

eBPF 在内核提供了大量的挂载点，算上 kprobe 挂载点，几乎可以在内核代码的任意函数挂载一段 eBPF 程序。不同 eBPF 程序类型的挂载点各不相同，挂载点是根据设计与需求提前在内核指定位置通过提前嵌入代码实现的，初始时函数指针是空，挂载 eBPF 程序后，对指针赋值，指向 eBPF 内核程序。

为了安全性，挂载 eBPF 程序需要 root 权限或 CAP_BPF capability，不过目前也有设计允许非 root 权限帐号载入 eBPF 程序，比如将 kernel.unprivileged_bpf_disabled sysctl 设置为 false 的情况下，非 root 帐号能

够使用 bpf()系统调用。

eBPF 内核程序基于事件触发，当内核执行到对应的挂载点时就会执行挂载在此处的 eBPF 程序。eBPF 执行过程中会用到一个很重要的数据结构 map，map 的主要功能是用来实现 BPF 程序执行过程中用户空间程序与内核 BPF 程序之间的双向通信，这也为内核中运行的 BPF 程序提供了更加灵活的控制。

**1、map 的实现**

map 的定义信息在编译后会保存到字节码文件中名为 maps 的 section 中，bpf 加载器读取到 map 信息后调用系统调用创建 eBPF map，系统调用返回由 anon_inodefs 文件系统生成的 fd。

加载器将内核返回的 map 的 fd，替换到使用 map 的 eBPF 指令的常量字段中，相当于直接修改编译后的 BPF 指令。

加载器在指令替换 map fd 后，才会调用 cmd 为 BPF_PROG_LOAD 的 bpf 系统调用，将程序加载到内核。

内核在加载 eBPF 程序系统调用过程中，会根据 eBPF 指令中常量字段存储的 map fd，找到内核态中的 map，然后将 map 内存地址替换到对应的 BPF 指令。

最终，BPF 程序在内核执行阶段能根据指令存储的内存地址访问到 map。

在 eBPF 数据面中，使用 eBPF map 只需要按照规范定义 map 的结构，然后使用 bpf_map_lookup_elem、bpf_map_update_elem、bpf_map_delete_elem 等 helper function 就可以对 map 进行查询、更新、删除等操作。

**2、map 的性能**

eBPF map 有多种不同类型，支持不同的数据结构，最常见的例如 Array、Percpu Array、Hash、Percpu Hash、Lru Hash、Percpu lru Hash、Lpm 等等。

常用 map 的查询性能比较如下：

Array > Percpu Array > Hash > Percpu Hash > Lru Hash > Lpm

需要着重说明的是，Array 的查询性能比 Percpu Array 更好，Hash 的查询性能也比 Percpu Hash 更好，这是由于 Array 和 Hash 的 lookup helper 层面在内核有更多的优化。对于读多写少的情况下，使用 Array 比 Percpu Array 更优（Hash、Percpu Hash 同理），而对于读少写多的情况（比如统计计数）使用 Percpu 更优。

### 2.1.3.1 eBPF 常见程序类型

eBPF 作为一个通用执行引擎，可用于性能分析工具、软件定义网络等诸多场景的开发工作。根据应用场景的不同，eBPF 程序类型大致可以分为三类：

表 2-1-1 eBPF 程序类型分类

| 分类 | 用途 | 包含的程序类型 |
| --- | --- | --- |
| 跟踪 | 主要用于从系统中提取跟踪信息，进而为监控、排错、性能优化等提供数据支撑 | tracepoint，kprobe，perf_event 等 |
| 网络 | 主要用于对网络数据包进行过滤和处 | xdp，sock_ops，sk_msg，sk_skb， |

| | | |
|---|---|---|
| | 理，进而实现网络的观测、过滤、流量控制以及性能优化等各种丰富的功能 | sk_reuseport,socket_filter, cgroup_sock_addr 等 |
| 其他 | 主要用于安全和其他功能 | LSM,flow_dissector, lwt_in,lwt_out,lwt_xmit 等 |

**1、kprobe**

kprobe是linux系统的一个动态调试机制，使用它可以向内核添加探针（Probe），在代码执行前或执行后触发一个回调函数。这个机制通常用于调试内核代码，跟踪应用程序执行或收集性能统计信息。通过使用 kprobe，开发人员可以在不影响系统运行逻辑的情况下，对操作系统进行深入的分析和调试。

kprobe 机制提供了两种形式的探测点，

一种最基本的 kprobe：用于在指定代码执行前、执行后进行探测，挂载点可以是内核代码的任何指令处；

一种是 kretprobe：用于完成指定函数返回值的探测功能，挂载点是内核函数的返回位置；

当内核执行到指定的探测函数时，会调用回调函数，用户便可收集所需的信息，在完成回调函数后，内核会继续回到原来的位置正常执行。如果用户已经收集足够的信息，不再需要继续探测，则同样可以动态的移除探测点。

eBPF 可以在内核态高效的分析 kprobe 采集到的数据，仅把结果反馈到用户空间，而不需要像传统系统一样必须将大量的采样数据全部传输到用户空间再进行分

析，极大地减轻了内核态与用户态的通信压力，使得持续地实时分析成为可能。

## 2、XDP

XDP 全称 eXpress Data Path，即快速数据路径，XDP 是 Linux 网络处理流程中的一个 eBPF 钩子，能够挂载 eBPF 程序，它能够在网络数据包到达网卡驱动层时对其进行处理，具有非常优秀的数据面处理性能，打通了 Linux 网络处理的高速公路。

XDP 程序最常用的模式是 native，它的挂载点处于网卡驱动之中，在网络驱动程序刚收到数据包时触发，无需通过网络协议栈，可用来实现高性能网络处理方案，常用于 DDos 防御，防火墙，4 层负载均衡等场景。

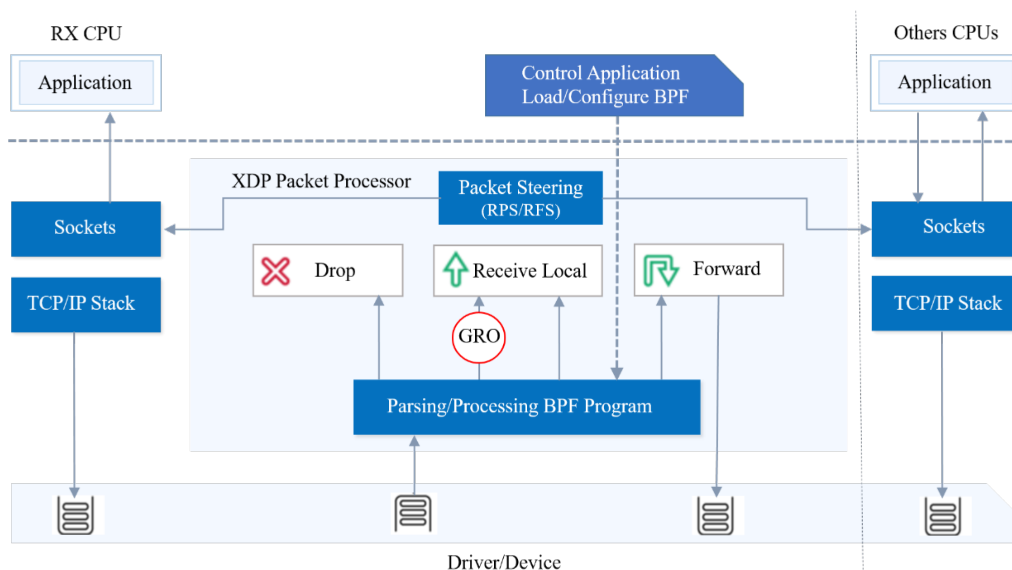


图 2-1-3 XDP 的工作过程示意图

XDP 程序在处理网络包后，有不同处理方式

表 2-1-2 XDP 的处理方式

| 命令码 | 含义 | 使用场景 |
| --- | --- | --- |
| XDP_DROP | 丢包 | 防火墙，DDOS 攻击防御 |
| XDP_PASS | 传递到内核协议栈 | 正常处理 |
| XDP_TX / XDP_REDIRECT | 转发到同一/不同网卡 | 负载均衡 |
| XDP_ABORTED | 错误 | XDP 处理错误 |

**3、TC**

Linux TC 是 Linux 操作系统中的一个调度器，它是 Traffic Control 的缩写，是一种网络流量控制、流量压缩和 QoS（Quality of Service）保证机制，在 Linux 内核中相当于一个独立的子系统，它管理着 Linux 内核中的网络设备和队列，可以实现对不同类型的流（如 HTTP、FTP、SSH 等）的流量限制、限流、分类和优化等，所有这些功能都是经过用户态 TC 命令工具来完成的，一般包括以下的步骤:

1、 为网卡配置一个队列;

2、 在该队列上创建分类;

3、 根据需要创建子队列和子分类;

4、 为每一个分类创建过滤器。

eBPF 在 TC 功能的 sch_handle_ingress、sch_handle_egress 函数中分别添加 HOOK 点实现了 TC ingress 和 TC egress 两个挂载点，用于实现对网络流量的整形

调度和带宽控制。

eBPF/TC 直接获取内核解析后的网络报文数据结构 sk_buff，可以获取到更详细的数据。而且由于 eBPF 程序的引入，可以通过 eBPF map 实现用户态与内核态的数据交互，而 map 数据结构则相比 TC 的规则更加直观，也更加好管理。

**4、Sock_ops**

sock_ops 是一种通过 eBPF 程序拦截 socket 操作，然后动态设置 TCP 参数的机制。它总共提供了 15 个挂载点，分别位于 TCP socket 实现的各个阶段，包括三次握手和四次挥手，可用于动态设置 TCP 参数，也可用于统计套接字信息。

sock_ops BPF 程序会利用 socket 的一些信息（例如 IP 和 port）来决定 TCP 的最佳配置。例如，在 TCP 建立连接时，如果判断 client 和 server 处于同一个数据中心（网络质量非常好），那么就可以通过如下设置优化这个 TCP 连接：

1、设置更合适的 buffer size：RTT 越小，所需的 buffer 越小；

2、修改 SYN RTO 和 SYN-ACK RTO，大大降低重传等待时间；

3、如果通信双方都支持 ECN，就将 TCP 拥塞控制设置为 DCTCP（DataCenter TCP）。

**5、LSM**

LSM（Linux Security Modules）是一个框架，允许用户空间程序向 Linux 内核添加自定义的安全模块。这些模块可以实施 MAC（Mandatory Access Control）、RBAC（Role-Based Access Control）等各种策略。

LSM 在 Linux 内核安全相关的关键路径上预置了一批 hook 点，从而实现了内核和安全模块的解耦，使不同的安全模块可以自由地在内核中加载/卸载，无需修改原有的内核代码就可以加入安全检查功能。

在过去，使用 LSM 主要通过配置已有的安全模块（如 SELinux 和 AppArmor）或编写自己的内核模块；而在 Linux 引入 BPF LSM 机制后，一切都变得不同了：现在，开发人员可以通过 eBPF 编写自定义的安全策略，并将其动态加载到内核中的 LSM 挂载点，而无需配置或编写内核模块。

当 eBPF 与 LSM 结合使用时，可以通过在 eBPF 程序中访问 LSM 接口来增强内核安全性，并对系统进行更细粒度的访问控制。例如，在 eBPF 程序中使用 LSM 接口来限制进程对某些敏感文件或目录的访问权限。此外，还可以在 eBPF 程序中启用 LSM Hook 以监视系统调用和网络连接，并执行适当的操作以保护系统免受恶意软件攻击。结合使用 eBPF 和 LSM 还可以使系统更加安全和高效，并提供更多的可定制化选项。

## 2.2 eBPF 常见的开发框架

### 2.2.1 BCC

BCC 是如今最热门也是对新手最友好的开源平台，它用 python 封装了编译、加载和读取数据的过程，提供了很多非常好用的 API；和 libbpf 需要提前把 bpf 程序编译成 bpf.o 不同，BCC 是在运行时才调用 LLVM 进行编译的，所以要求用户环境上有 LLVM 和 kernel-devel。

### 2.2.2 bpfTrace

bpfTrace 是基于 BPF 和 BCC 构建的开源跟踪程序。与 BCC 一样，bpfTrace 附带了许多性能工具和支持文档。但是，它还提供了高级编程语言，使您可以创建功能强大的单行代码和简短的工具。它自定义了自己的 DSL 作为前端，底层也是调用 LLVM 的。事实上，它依赖于 BCC 提供的 libbcc.so 文件。

### 2.2.3 libbpf

libbpf 是指 linux 内核代码库中的 tools/lib/bpf，这是内核提供给外部开发者的 C 库，用于创建 BPF 用户态的程序。目标是为了使得 bpf 程序像其它程序一样，编译好后，可以放在任何一台机器，任何一个 kernel 版本上运行（当然要对内核版本有一些要求）。

libbpf 解决了如下问题：

**1、头文件的问题**：依赖内核态特性支持 BTF，将内核的数据结构类型构建在内核中；用户态的程序可以导出 BTF 成一个单独的.h 头文件，bpf 程序只要依赖这个头文件就行，不需要再安装内核头文件（vmlinux.h）的包。

**2、兼容性问题**：使用 clang-11 的针对 eBPF 后端专门的特性：preserve_access_index，支持记录数据结构 field 相对位置信息，可实现 relocation，从而解决了由于数据结构在不同内核版本间的变化导致的兼容性问题。

**3、性能提升**：内核中 bpf 模块做了一些增强，bpf Verifier 支持直接字段访问，不需要 call bpf 函数的方式来访问结构体字段，有效提升了 eBPF 程序的性能。

### 2.2.4 libbpf-bootstrap

libbpf-bootstrap 是一个项目，基于 libbpf 开发的框架，提供了一些样例程序和模板，帮助开发者理解如何使用 libbpf 创建、加载、管理 eBPF 程序，并与这些程序进行交互；提供了集成到构建系统的模板，可以方便地编译和链接 eBPF 程序。

### 2.2.5 cilium-ebpf

cilium-ebpf 是一个基于 Go 语言的 BPF 库，其将 eBPF 系统调用抽象为 Go 接口。相比于其他开发框架，Cilium/eBPF 具有如下特点：

1、用户态程序可使用纯 go 语言开发，无需引入 cgo，减少环境依赖

2、编译相对简单，只需执行简单命令即可完成编译，无需编写复杂 Makefile

3、支持 CO-RE，编译出的二进制可执行文件可在同架构任何环境运行

4、依托于 Cilium 在 eBPF 研究上的持续投入，Cilium/eBPF 社区较活跃，受关注度较高

### 2.2.6 Coolbpf

Coolbpf 项目，以 CORE（Compile Once--Run Everywhere）为基础实现，保留了资源占用低、可移植性强等优点，还融合了 BCC 动态编译的特性，适合在生产环境批量部署应用。Coolbpf 分为三大模块：开发&测试、编译组件和基础模块。开发&测试组件提供了多语言开发和自动化测试功能；编译组件提供了多种编译方式，灵活性高；基础模块提供了 BTF、低版本内核 eBPF 驱动、通用库等多个功能。

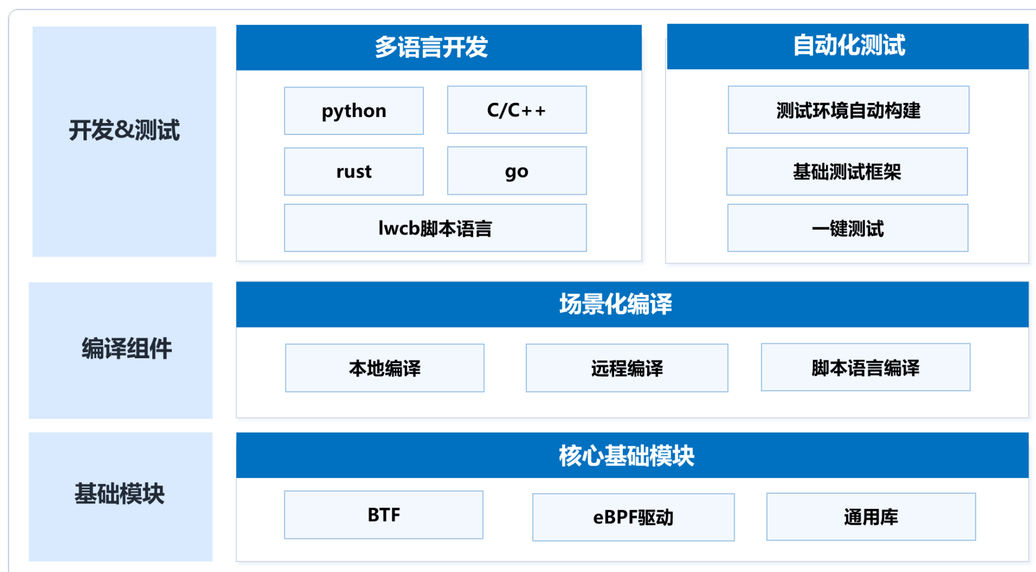


图 2-2-1 Coolbpf 功能架构图

**1、开发测试模块**

目前 Coolbpf 项目支持 python、rust、go 及 c 语言，覆盖了当前主流的高级开发语言。此外 Coolbpf 还支持 lwcb 脚本语言，便于开发者快速开发 eBPF 功能脚本。

Generic library：基础通用库，提供了 eBPF 相关的 API，如 eBPF map、eBPF program 等；

Language bindings：基础通用库的 bindings，使得多种编程语言能够直接使用基础通用库；

Language library：由于 bindings 缺少高级语言的面向对象编程思想，为此在 language bindings 的基础上做进一步的封装。

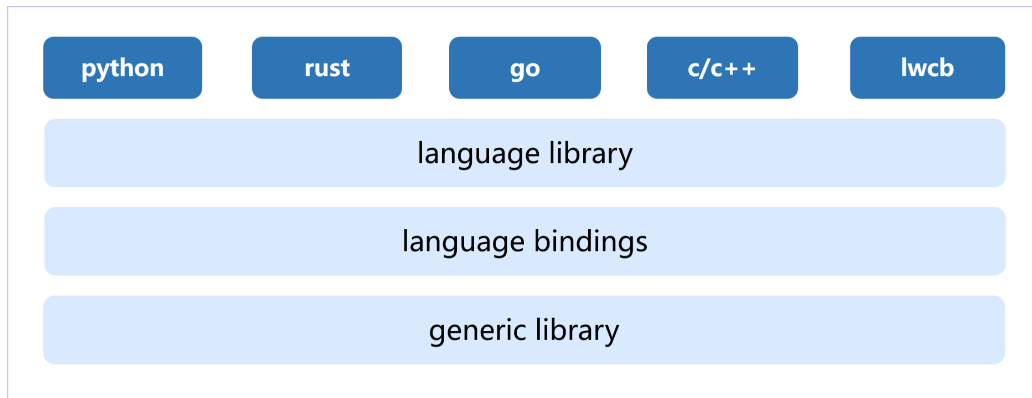


图 2-2-2 Coolbpf 开发模块层级结构图

**2、编译模块**

本地编译服务，基础库封装，客户使用本地容器镜像编译程序，调用封装的通用函数库简化程序编写和数据处理；本地编译服务，不需要推送 bpf.c 到远程，一些常用的库和工具，已经包含在提供的镜像里面，省去了构建环境的繁杂。

远程编译服务，接收 bpf.c，生成 bpf.so 或 bpf.o，提供给高级语言进行加载，用户只专注自己的功能开发，不用关心底层库安装、环境搭建；远程编译服务，目前用户开发代码时只需要 pip install Coolbpf，程序就会自动到编译服务器进行编译。

脚本语言编译主要功能是编译 lwcb 脚本，生成 eBPF 字节码。

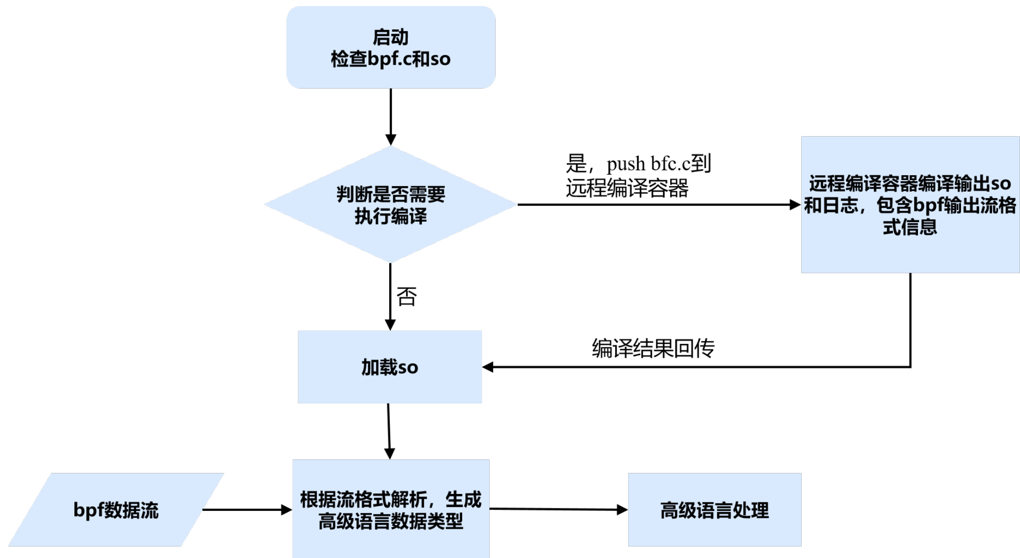


图 2-2-3 Coolbpf 编译模块逻辑图

### 3、基础模块

基础模块提供了关键组件。Coolbpf 为了保证 eBPF 程序能够运行在低版本内核，提供了 Coolbpf 库和 eBPF 驱动。其中 Coolbpf 库用于发起 eBPF 程序运行时的系统调用或向 eBPF 驱动发起的 ioctl 请求。eBPF 驱动则根据 ioctl 请求的具体信息执行相应的动作，如创建 map，prog 的安全检查、JIT 等。

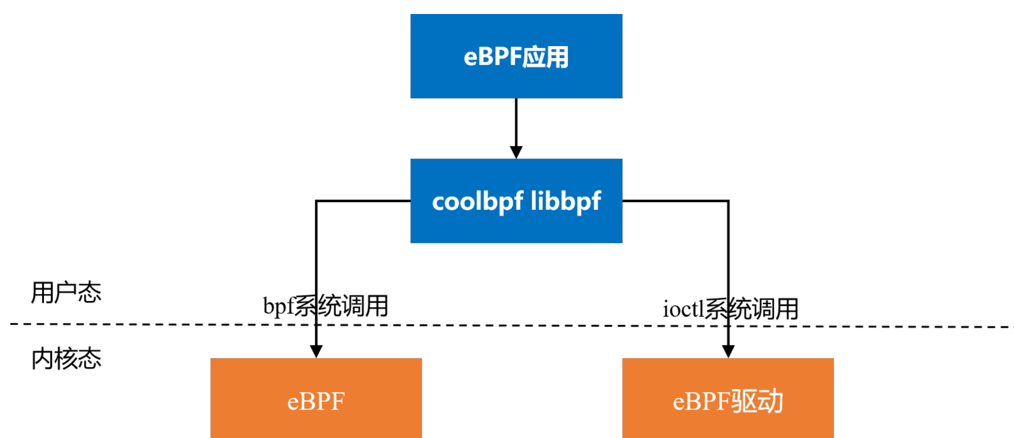


图 2-2-4 Coolbpf 基础模块结构图

# 3 基于 eBPF 的技术创新与应用实践

eBPF 具有高性能、高扩展、安全性等优势，目前已经在网络、安全、可观测性等领域广泛应用，并且随着云计算和容器化技术的不断发展，eBPF 将会在更多的应用场景中得到广泛应用。

## 3.1 基于 eBPF 的系统诊断

### 3.1.1 系统诊断面临挑战

传统系统诊断在网络、IO、内存和调度等方面存在局限性

**网络**

网络诊断中网络抖动是最疑难的问题。网络抖动是指在网络传输过程中，因为各种原因导致数据包的传输延迟增加的现象。网络抖动可能会导致网络连接不稳定，影响网络应用程序的性能和用户体验。一般情况下，网络抖动问题会对客户的业务产生很大的影响。下面是一些比较典型的案例：

1、网络连接不稳定：网络抖动会导致网络连接的不稳定性，使得客户在使用网络应用程序时经常出现连接断开、页面加载缓慢等问题。

2、影响在线业务：对于在线业务来说，如在线游戏、视频会议、在线购物等，网络抖动会导致数据传输的延迟增加，使得用户无法流畅地进行交互或观看，影响用户体验，甚至导致交易失败。

3、影响远程办公：在远程办公的情况下，网络抖动会导致视频会议、文件传输

等协作工具的不稳定性，影响远程团队的工作效率和沟通效果。

4、影响云服务：对于使用云服务的客户来说，网络抖动会导致云服务的不稳定性，使得客户无法正常访问或使用云服务提供的功能。

5、业务中断：在某些情况下，网络抖动可能会导致网络连接断开，使得客户无法访问网络或者无法与服务器建立稳定连接，导致业务中断，造成损失。

传统的定位网络抖动问题主要通过一下几类工具。

1、网络数据包分析工具：如 Wireshark、tcpdump 等，这些工具可以捕获和分析网络数据包，通过分析数据包的延迟、丢包情况，定位网络抖动问题所在。

2、网络故障排查工具：如 Ping、Traceroute 等，这些工具可以用于排查网络故障，如检测网络连接的连通性、确定网络路径和跳点等，帮助确定网络抖动问题的发生位置。

3、网络设备监控工具：如 SNMP（Simple Network Management Protocol）管理工具，可以实时监控网络设备的状态和性能，包括路由器、交换机等，有助于发现设备故障引起的网络抖动问题。

网络链路是非常复杂且漫长的，一个报文从发送端到达接收端，要穿越各种网络组件，比如容器、ECS、OS 内核、物理机、VPC 网络、物理网络等。上面那些工具很难达到细粒度的网络抖动问题。

**IO**

IO 问题也是常见的问题之一，比如一个进程可能会导致磁盘 I/O 负载过高，直

接影响整个系统的 I/O 性能。在定位这类问题时，有两个关键点需要考虑：

1、确定哪个进程贡献了大量的 I/O：需要找出是哪个进程导致了 I/O 负载过高的情况。

2、确定 I/O 最终是由哪个磁盘消耗的，或者进程在访问哪个文件：如果系统统计显示某个进程产生了大量的 I/O，那么我们希望能够了解这些 I/O 最终是由哪个磁盘处理的，或者进程在访问哪个文件。如果这个进程是来自某个容器，我们也希望能够获取到文件访问信息以及该进程所在的容器。

这些信息有助于我们更准确地定位问题，并找出导致 I/O 问题的根本原因。

传统的定位手段主要包括以下几种：

1、基于内核 diskstats 衍生的工具，如 iostat、tsar --io、vmstat -d，这些工具主要从整个磁盘的角度统计 I/O 信息，例如统计整个磁盘的 IOPS（每秒 I/O 操作数）和 BPS（每秒字节数），但无法准确统计单个进程贡献的 IOPS 和 BPS。

2、基于内核 proc/$pid/io 衍生的工具，如 pidstat -d，这些工具可以统计进程总体的 I/O 情况，但无法追溯具体的 I/O 是归属于哪个磁盘或哪个文件。

3、基于 taskstat 衍生的工具，如 iotop，这些工具可以统计进程总体的 I/O 情况以及贡献的 IOWait 时间，但也无法追溯具体的 I/O 是归属于哪个磁盘或哪个文件。

以上工具虽然在一定程度上能够提供 I/O 性能的信息，但无法提供进一步的细粒度定位，无法准确追踪 I/O 归属的磁盘或文件。

**内存**

内存子系统常见的问题是内存泄漏。当出现内存泄漏时，程序会占用大量系统内存，甚至导致系统内存耗尽，对业务的正常运行造成影响。内存泄漏可能有多种潜在原因，例如在程序执行过程中忘记释放不再需要的内存。此外，如果应用程序的内存使用量过大，系统性能可能因为页面交换（swapping）而显著下降，甚至可能导致系统强制终止应用程序（如 Linux 的 OOM killer）。

调试内存泄漏问题是一项复杂而具有挑战性的任务。这涉及详细检查应用程序的配置、内存分配和释放情况，通常需要专门的工具来辅助诊断。

1、有一些工具可以在应用程序启动时将 malloc()函数调用与特定的检测工具关联起来，如 Valgrind memcheck。这类工具可以模拟 CPU 来检查所有内存访问，但可能会导致应用程序运行速度大幅降低。

2、另一个选择是使用堆分析器，如 libtcmalloc。相比之下，它的性能较高，但仍可能导致应用程序运行速度下降五倍以上。

3、此外，还有一些工具，如 gdb，可以获取应用程序的核心转储，并进行后处理以分析内存使用情况。然而，这些工具通常需要在暂停应用程序时获取核心转储，或在应用程序终止后才能调用 free()函数。

**调度**

在日常业务运行中，经常会遇到各种干扰，导致抖动和影响客户体验。其中一个常见的干扰源是中断抢占。当中断抢占时间过长时，可能会导致业务进程调度不及

时，数据收发延迟等问题。这种干扰已经存在于 Linux 内核中很长时间了，因此 Linux 内核和业界已经提供了多种相关的中断抢占检测手段。在 Linux 系统中，可以使用以下工具来定位抖动问题：

1、sysstat：sysstat 是一个系统性能监测工具集，其中包含了 sar、mpstat、pidstat 等工具。可以使用 sar 命令来查看系统的 CPU 使用率、内存使用率、I/O 情况等，以判断是否存在系统抖动问题。然而，该工具输出的指标较多，对于非专业用户来说可能需要一定的专业知识进行分析，并且难以准确定位根本原因。

2、top：top 命令用于实时监测系统的 CPU 使用率、内存使用率、进程状态等，并按照 CPU 或内存使用率进行排序。通过查看 top 命令的输出，可以找出占用 CPU 或内存较高的进程，以判断是否存在抖动问题。然而，top 命令只提供了进程级别的信息，对于复杂的抖动问题可能无法提供全面的分析。

3、Perf：Perf 是 Linux 内核提供的性能分析工具，可以用于监测系统调度情况，包括调度延迟和抖动。通过 perf sched 命令可以收集和分析调度相关的事件和数据。然而，该工具输出的调度信息较多，需要进行大量的、长时间的分析，仍然可能难以定位到根本原因。

4、Latencytop：Latencytop 是一个基于 Linux 内核的工具，用于检测系统中的延迟问题。它可以提供实时的进程级别的延迟信息，包括调度延迟，有助于定位调度抖动问题。然而，对于非专业用户来说，解读和理解 Latencytop 输出的详细延迟信息和统计数据可能会有一定的难度。

可以看出，传统的工具在定位抖动问题时仍然存在短板，例如输出过多的数据难

以分析，甚至即使进行了分析，仍难以准确定位根本原因。

### 3.1.2 基于 eBPF 的系统诊断方案

针对传统系统诊断工具在网络、IO、内存和调度领域的限制，基于 eBPF 的系统诊断方案应运而生，为系统在网络、IO、内存和调度等方面的诊断提供了更强大的能力。

**网络**

网络抖动问题定位周期长，定位难度大（跨很多组件），业务影响严重的特点，在这个背景下，基于 eBPF 开发的抖动诊断工具：PingTrace，可以快速定位网络中哪个位置出现了问题，提高解决问题的速度和效率。PingTrace 是一个基于 eBPF 实现的网络延迟探测定界工具，该工具实现了一个基于 ICMP 回显(ICMP_ECHO 和 ICMP_ECHOREPLY)协议的网络延迟探测协议，通过发送和解析探测报文来定位报文在不同传输阶段的延迟信息。

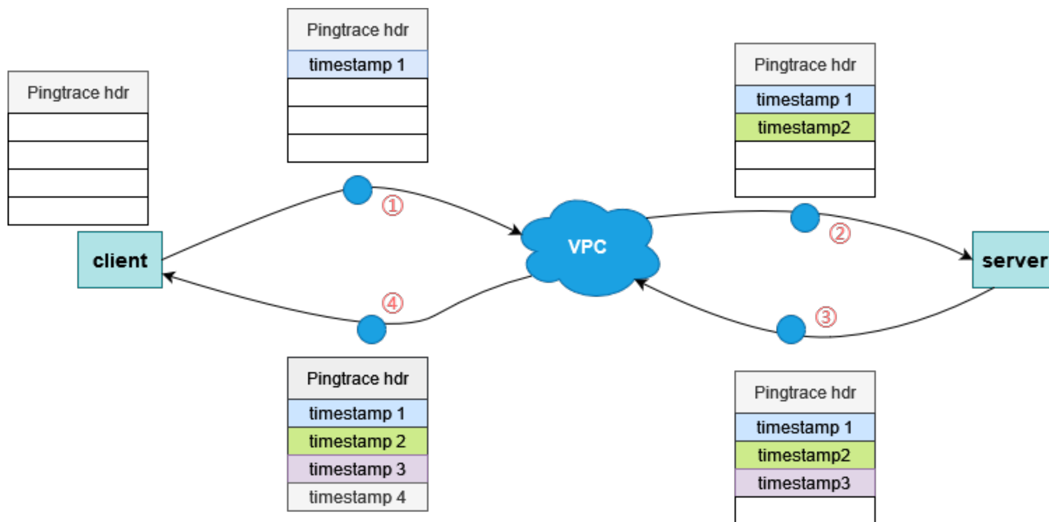


图 3-1-1 PingTrace 原理图

上图展示了 PingTrace 的基本原理。其大致流程如下：1）基于 ICMP 回显协议定义了 PingTrace 报文的格式。2）在报文的传输路径上，使用 eBPF 部署检测点。每经过一个检测点，报文中会添加该检测点的时间戳信息。3）当发送端收到接收端的回包后，可以获取所有时间戳数据进行分析，以确定存在延迟的位置。

PingTrace 已经开源到 sysak，可以通过 sysak pingtrace 来使用 PingTrace 功能。PingTrace 工具使用方法如下：

```
Usage: pingtrace [OPTIONS]
Options:
  -v,--version              显示版本号
  -h,--help                 帮助信息
  -s,--server               以 server 模式运行
  -c,--client ip            以 client 模式运行
  -C,--count UINT           探测报文数量，默认无限
  -i interval_us            以微秒为单位，报文发送间隔时间
  -t UINT                   以秒为单位，程序运行时间
  -m,--maxdelay us          以微秒为单位，判定为毛刺的阈值。只有超过该值的
                            报文数据才会被记录下来，默认为 0
  -b INT=556                发送探测报文的大小，至少 144 字节
  --log TEXT=./trace.log 日志文件名称
  --logsize INT             日志文件最大占用磁盘空间
  --logbackup INT=3         日志文件最多备份数量
  --mode auto/pingpong/compact          PingTrace 运行模式
  -o,--output image/json/log/imagelog   PingTrace 数据输出格式
  -n,--namespace            探测与 net namespace 相关的信息
  --nslocal                 在探测 net namespace 相关信息时，告知 PingTrace
                            client 和 server 运行在同一 host 上，以避免获取到冗余数据
  --userid UINT             在探测 net namespace 相关信息时，为不同 Host 指定
                            不同 userid，以帮助 PingTrace 识别和修正不同 Host
                            上时间不同步问题
  --debug                   打印相关 debug 信息，主要为 libbpf 信息
```

通过 Pingtrace 能够达到网络抖动问题分钟级定界。

**IO**

基于 eBPF 的 iofsstat 工具可以从进程的角度统计 IO 信息和文件读写信息，这有助于确定问题的根源，并采取下一步的措施，例如停止进程或容器，检查逻辑是否合理，特别是对于大量写入某个文件的情况。下面是 iofsstat 的大致工作原理图。

图 3-1-2 iofsstat 工作原理图

从 iofsstat 的工作原理图可以看出，该工具主要用于统计某个进程对某个文件的 IO 贡献。它以进程角度和磁盘角度为基础来呈现用户的指标。磁盘角度的统计是为了提供对整个磁盘的综合了解，而进程角度的统计则更具有细粒度，能够显示在某个时刻各个进程的贡献情况。值得注意的是，进程角度的统计和磁盘角度的统计并不一定完全一致。因为统计的原理不同，两者之间可能存在差异。在分析时，我们更关注两个角度之间的关联性。例如，在某个时刻磁盘统计显示 IO 负载较高，此时可以查看对应时刻各个进程的贡献程度。通过 iofsstat 可以更准确地了解进程与文件的 IO 关系，从而有针对性地解决问题。下面是 iofsstat 工具的指标解释：

进程角度：

comm：进程名

pid：进程 id

cnt_rd：读文件次数

bw_rd：读文件"带宽"

cnt_wr：写文件次数

bw_wr：写文件"带宽"

inode：文件 inode 编号

filepath：文件路径，当在一次采集周期内由于进程访问文件很快结束情况下，获取不到文件名则为"-"

如进程来自某个容器，在文件名后缀会显示[containterId:xxxxxx]

磁盘角度：

xxx-stat:r_iops：  磁盘总的读 iops

xxx-stat:w_iops：磁盘总的写 iops

xxx-stat:r_bps：磁盘总的读 bps

xxx-stat:w_bps：磁盘总的写 bps

xxx-stat:wait：磁盘平均 io 延迟

xxx-stat:r_wait：磁盘平均读 io 延迟

xxx-stat:w_wait：磁盘平均写 io 延迟

xxx-stat:util%：磁盘 io utils

**内存**

Coolbpf 提供的基于 eBPF memleak 工具用于检测内存泄漏问题。memleak 工具的原理如下：

1、memleak 使用 eBPF 来跟踪内核分配的内存块。它通过在内核中插入 eBPF 程序来监测内存分配和释放的事件。

2、当内核分配内存时，memleak 会在 eBPF 程序中记录下内存块的地址和大小，并将其保存在一个哈希表中。

3、当内核释放内存时，memleak 会检查该内存块是否存在于哈希表中。如果不存在，则将其标记为泄漏。

4、 memleak 还会记录泄漏内存块的调用栈信息，以帮助定位泄漏的源代码位置。

5、memleak 会周期性地打印出已泄漏的内存块的信息，包括内存地址、大小和泄漏的调用栈。

通过使用 memleak 工具，开发人员可以监测和定位内存泄漏问题，从而改善系统的内存管理和性能。它提供了一种非侵入式的内存泄漏检测方法，不需要修改应用程序的源代码，而且对性能影响较小。

**调度**

针对调度问题，经过分析将其分为三类具体场景：

1、关中断：包括硬中断和软中断执行时间过长的情况；

2、进程唤醒后长时间无法调度；

3、CPU 任务队列过长。

针对每个问题类别，都有相应的 eBPF 工具提供了场景化的解决方案，避免用户面对难以理解和分析的问题时感到困惑。这些工具可以精确地找到问题所在。以关中断场景为例，可以使用 irqoff 工具来定位关中断场景下的抖动问题。其原理图如下：

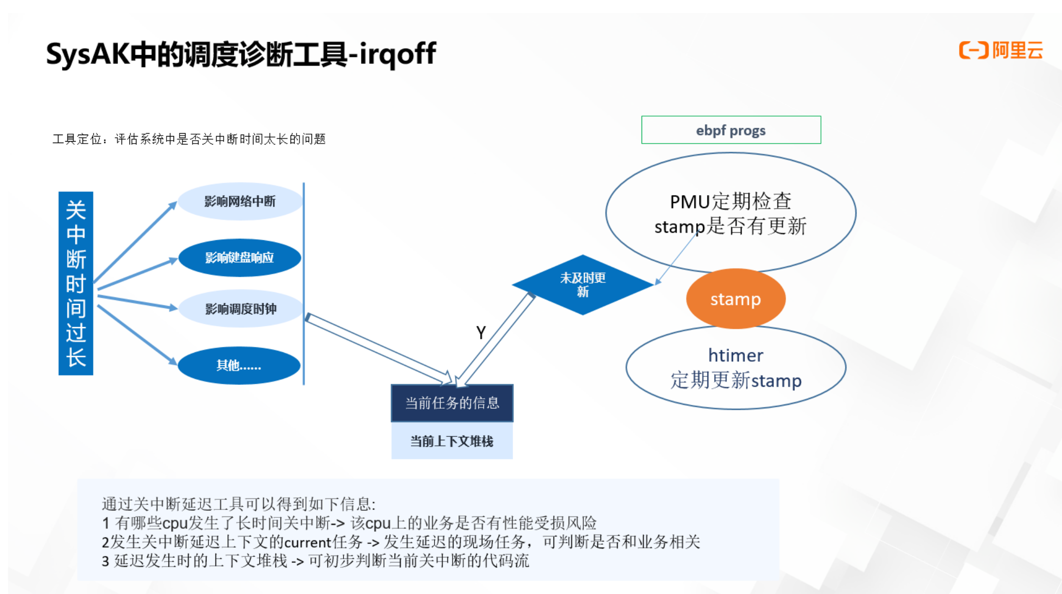


图 3-1-3 irqoff 工具的原理图

irqoff 的使用方法如下：

sysak irqoff [--help] [-t THRESH(ms)] [-f LOGFILE] [duration(s)]

-t：关中断的门限值，单位是 ms。 -f：指定 irqoff 结果记录的文件。 duration：工具的运行时长，如果不指定默认会一直运行。

通过内核模块创建 worker 来构造了一个长时关中断的场景，下面是通过 irqoff 抓取的结果展示。

| TIME(irqoff) | CPU | COMM | TID | LAT(us) |
| --- | --- | --- | --- | --- |
| 2022-05-05_11:45:19 | 3 | kworker/3:0 | 379531 | 1000539 |

<0xffffffffc04e2072> owner_func

<0xffffffff890b1c5b> process_one_work

<0xffffffff890b1eb9> worker_thread

<0xffffffff890b7818> kthread

<0xffffffff89a001ff> ret_from_fork

结果可以分为几个部分：

1、首先是 log header，总共有 5 列。从左到右依次是时间戳（模块信息）、关中断长的 CPU、关中断长的当前线程 ID、总的关中断延迟。

2、接下来是与 log header 相对应的实际信息。

3、然后是抓取到的关中断现场的堆栈信息，这些信息有助于进行下一步的源码分析。

通过堆栈信息，可以清楚地看到是由 worker 线程关中断导致的系统抖动。

### 3.1.3 基于 eBPF 的 Profiling

#### 3.1.3.1 为什么要 Continues Profiling?

众所周知，可观测的三大支柱包括 Log、Trace、Metric，过去，我们一般分析问题的步骤是：

- 通过日志和告警发现异常（Metrics/Log 日志）；
- 通过监控大盘（曲线、统计值）找到异常的模块（Metrics）；
- 通过调用链数据定位到问题的微服务或应用及代码块（Tracing）；

这看起来非常顺利，但是通过 tracing 也只能看到哪个应用延时高等问题，还有很多做不到的地方：

- 函数级别的消耗无法知晓；
- 整体资源消耗情况，后台运行任务做了什么不清楚；
- 内存占用和释放情况不清晰；
- 等待锁和循环操作无法分析；

因此为了定位问题，我们需要去做持续剖析（Continues Profiling），不断获取并保存函数调用栈，保证问题发生后能够回溯到当时的问题现场，而不是临时去采集数据再排查，因为此时问题现场可能已经不存在了。

不同的编程语言有不同的 profiling 工具，像 Go 的 pprof、Java 的 jstack 等，如果我们希望观测不同应用但又想抛开语言的差异化，可以借助 eBPF 来实现栈信息的获取，包括应用在用户态执行和内核态执行的全部信息。使用 eBPF 的好处在于我们可以做到 profiling 过程的可控 ：频率快慢、运行时安全、系统资源占用小等。

eBPF 技术的引入，使得应用程序和系统本身的运行时行为具有前所未有的可见性。通过赋予应用程序和系统两方面的检测能力，将两种视图结合起来，从而获得强大而独特的洞察力来排查各类性能问题。随着 Continuous Profiling 持续剖析在性能分析、故障诊断、函数级调试中发挥越来越大的作用，它极有望成为可观测的第四大支柱。

#### 3.1.3.2 如何做 Continues Profiling?

一个程序的运行时最简单的可以概括为执行和不执行两种状态 ，即 on cpu 和 off cpu。on cpu 我们希望看到程序占用 CPU 时的执行逻辑，哪个任务甚至任务的哪一段代码在 CPU 上消耗资源。而 off cpu 我们希望看到应用是否是自愿放弃的 CPU，出于何种原因不占用 CPU，如等锁、等 IO 等，以此希望发现一些应用损耗时造成的收发包延迟等问题。另外，我们也可以根据 IO 请求的调度行为、内存分配和释放的行为、系统负责 load 的高低来生成对应火焰图，让开发和运维人员快速的通过图的形式就能分析系统存在的问题。

On cpu 一般通过采样的方式进行检测；off cpu 需要通过对内核函数进行 hook（sched_switch）跟踪任务的状态转换；IO、Memory 内存、load 等也可以通过跟踪具体的函数调用情况来形成火焰图。

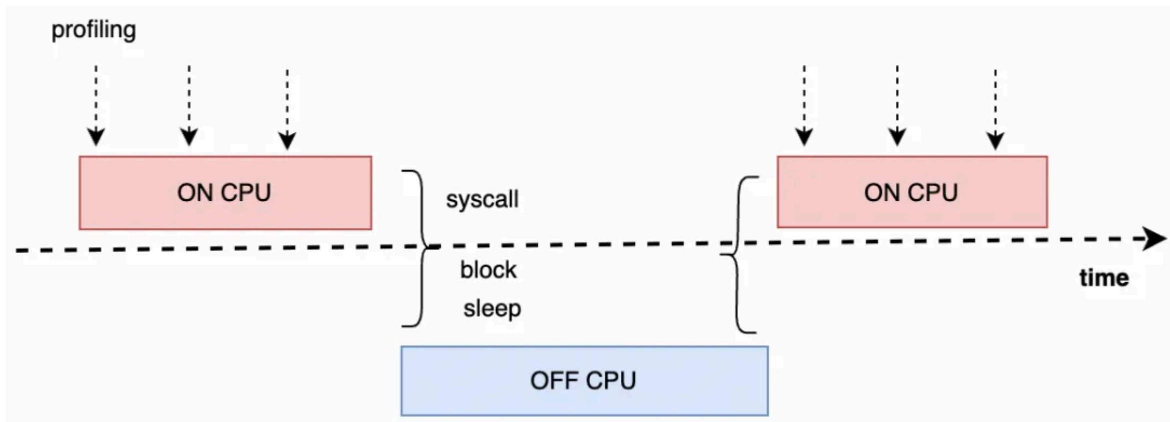


图 3-1-4 On cpu 和 Off cpu 两种状态

对于持续剖析应用的行为，主要体现在 On cpu 上的热点追踪，目前大部分实现基于 eBPF 开发，利用 Linux 中的 PMU(Performance Monitoring Unit) 事件进行性能分析。它定期对每个处理器进行采样，以便捕获内核函数和用户空间函数的执行栈信息，如：

- 地址：函数调用的内存地址
- 符号：函数名称
- 文件名：源代码文件名称
- 行号：源代码中的行号

如下图所示，通过 eBPF 加 PMU 我们就可以定期获取调用栈信息，同时利用 eBPF 的 map 对每个栈信息做统计。

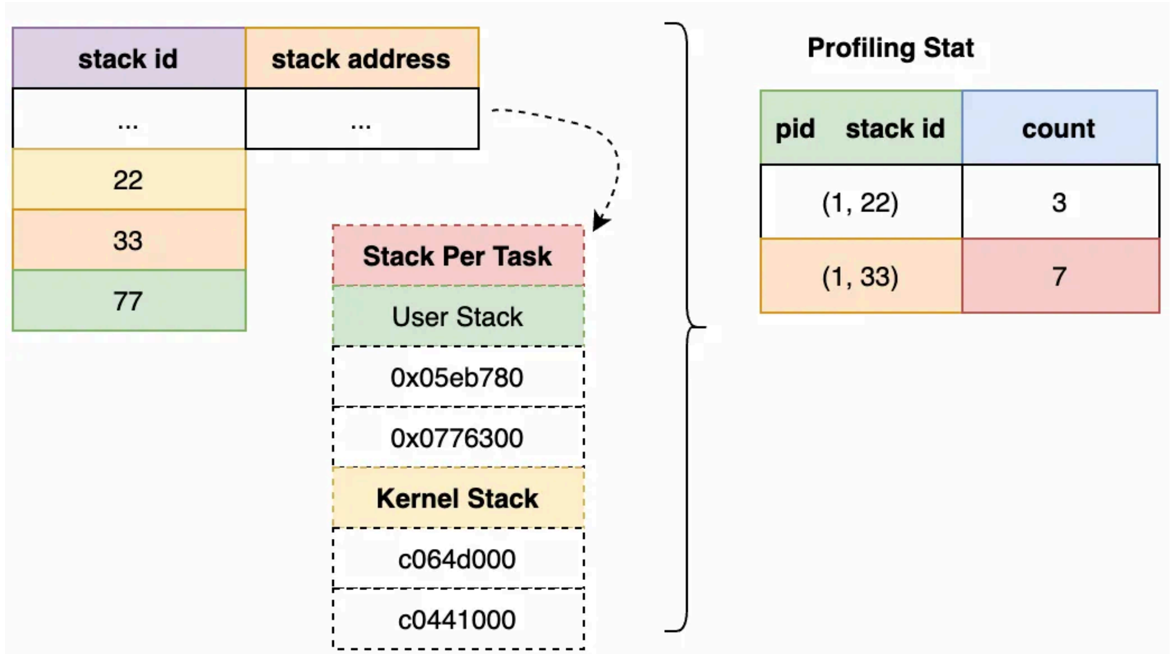


图 3-1-5 获取调用栈信息

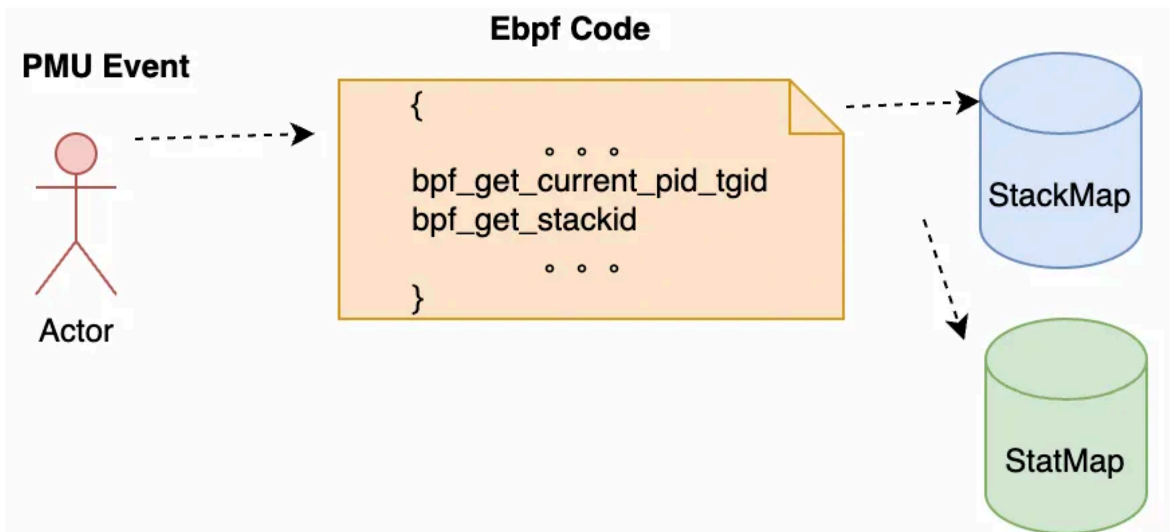


图 3-1-6 信息统计

profiling 功能由两个部分组成：内核态中的 eBPF 获取栈信息程序和用户态中的函数符号解析程序。

内核态 eBPF 程序:

内核态 eBPF 程序的实现逻辑主要是借助 perf event，对程序的堆栈进行定时采样，从而捕获程序的执行流程。

bpf_get_stackid(ctx, pstack, KERN_STACKID_FLAGS);

bpf_get_stackid(ctx, pstack, USER_STACKID_FLAGS);

bpf_map_update_elem(maps, pstack, &value, BPF_NOEXIST);

上面两个函数分别获取应用的内核态栈和用户态栈，然后将栈信息更新到 maps 里保存。

用户态符号解析程序：

通过前面的方法获取到了内核态和用户态栈信息后，需要在用户态将栈信息转换为调用符号信息。转换一般通过以下三个步骤：

1、基于/proc/kallsyms 获取内核全量符号表；

2、调用 elf 函数获取应用全量符号表：

sym_name=elf_strptr(elf,shdr.sh_link,sym.st_name);

3、有了上述内核态和用户态符号表信息后，就可以通过栈查找函数 usym_search 获取栈对应的符号信息：

Int usym_search(structfast_usym_ctrl*ctrl,structfast_usym_query*

query,addr_taddr)；

其中 ctrl 为全量符号（含 elf/kallsyms）控制结构体，query 用于保存查询结

果，addr 为 eBPF 保存的栈地址信息。

#### 3.1.3.1 Continues Profiling 整体架构

我们事先需要部署 agent 去负责 profiling，在 server 端去查看数据。在 agent 侧，将 profiling 数据做切片处理，定时从 map 中拿数据并清空 map 上一周期的采样数据，确保我们在做数据回放的时候看到的是对应时间段的 profiling 结果。考虑用户对云上环境数据安全的要求，我们也可以借助 SLS 通道完成数据上传到中心端，通过中心端对数据进行各种加工，可以分析出当前系统存在的问题，一般通过火焰图的形式，直观的展现当前的调用栈及热点函数在哪个模块。

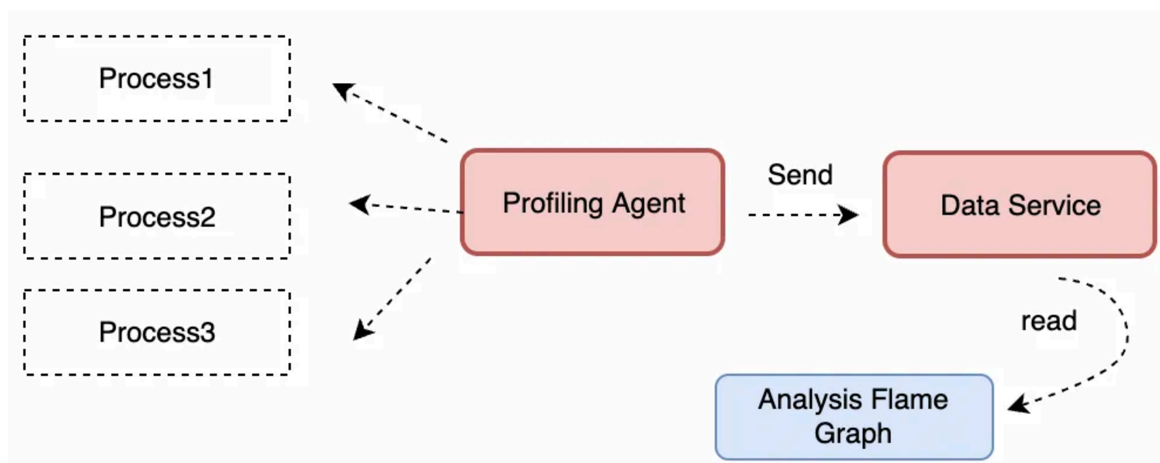


图 3-1-7 profiling 原理示意图

在界面展示时，可以通过火焰图的方式，看到函数占用 cpu 的时长、百分比和调用顺序：

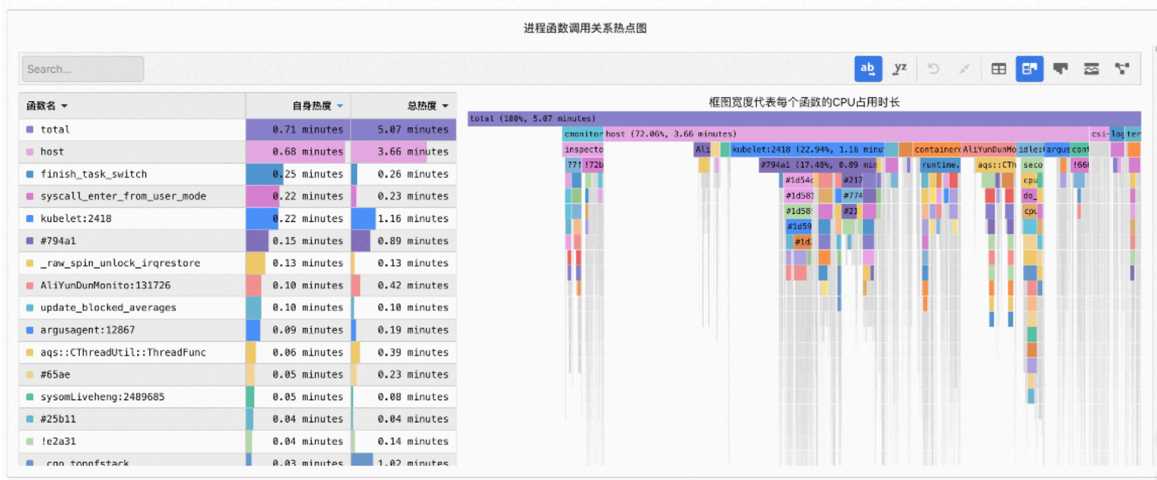


图 3-1-8 火焰图

或者通过调用图的方式，红色代表这一分支占比高，通过箭头粗细能很清楚的知道性能瓶颈在哪个函数：

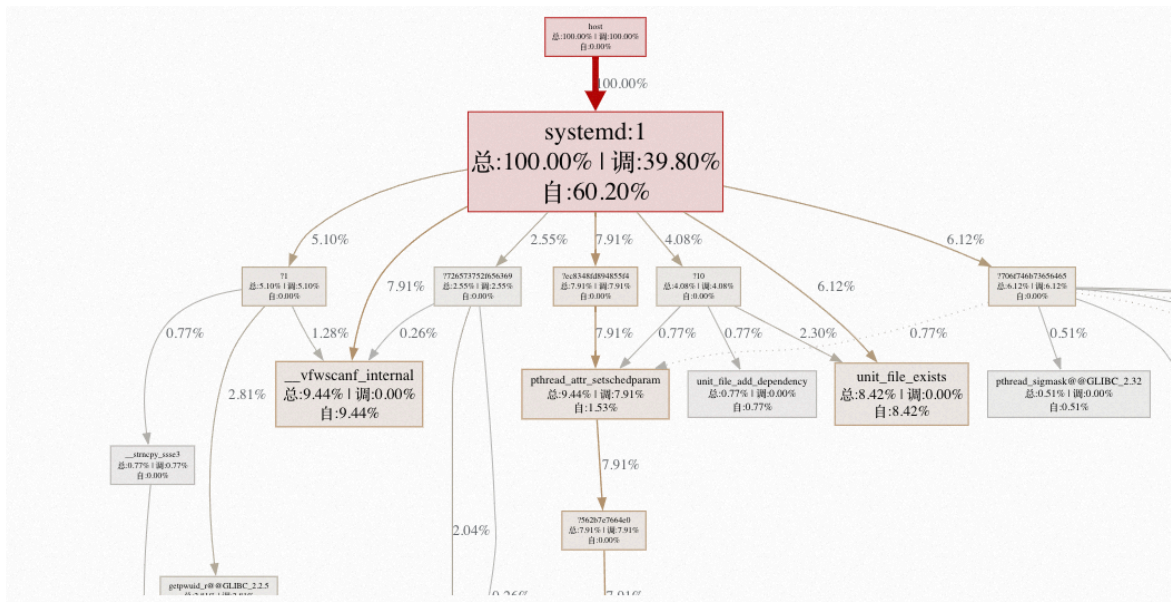


图 3-1-9 调用图谱

在龙蜥社区开源的项目上，可以看到具体的代码，请大家参考：

中心端 SysOM: [https://gitee.com/anolis/sysom](https://gitee.com/anolis/sysom)

节点端 SysAK: https://gitee.com/anolis/sysak

eBPF 采集端 Coolbpf: https://gitee.com/anolis/coolbpf

## 3.2 基于 eBPF 的虚拟化 IO 全链路时延监测

### 3.2.1 虚拟化 IO 全路径分析主要面临的挑战

虚拟化是构建云计算基础架构的关键技术之一，虚拟化存储 I/O 则是构成的主要基础，其性能与稳定性直接影响到云平台服务。虚拟化存储 IO 性能分析主要面临以下挑战：

**IO 路径长**：虚拟化 IO 路径贯穿客户操作系统的用户态与内核态、hypervisor、主机操作系统用户态与内核态、后端存储设备等。

**地址映射复杂**：客户操作系统的 IO 请求生命周期内会经历多层地址映射与转换：

1、客户操作系统文件内逻辑偏移(guest file offset)到客户操作系统块设备物理地址(guest block LBA)转换。如客户操作系统为 ext4，需要处理文件内偏移到 ext4 后端块设备 LBA 地址转换；

2、客户操作系统块设备物理地址(guest block LBA)到虚拟磁盘文件内偏移(guest vdisk offset)。如对于 QCOW2 格式虚拟磁盘文件，需要处理虚拟磁盘地址到 QCOW2 data cluster 的映射转换；

3、虚拟磁盘文件内偏移(guest vdisk offset)到主机操作系统块设备设备物理地址（host block LBA）转换。由于虚拟磁盘只是主机操作系统内的一个普通文件，因

此这一过程和 guest file offset 到 guest block LBA 转换方法类似。

此外，对于超融合场景，还需要分析从 host block LBA 到软件定义存储副本之间映射转换等。

**“观测者效应”**：存储 IO 路径作为虚拟化的关键数据平面之一，额外的性能追踪逻辑会直接影响到 IO 的性能，从而影响到了观测的准确性，甚至会影响到虚拟机业务本身。

Linux 下现有 IO 性能追踪分析工具主要有：blktrace、perf、iostat 等。这些工具的主要问题是只能从特定 IO 层(如块层)追踪或统计 IO 请求状态，无法从 IO 请求角度完成 IO 全生命周期的性能追踪，不能够从虚拟化系统角度完成全链路 IO 性能的监控、追踪，无法形成全局 IO 性能视图。

### 3.2.2 基于 bpftrace 虚拟化 IO 路径追踪解决方案

#### 3.2.2.1 主要功能

基于 bpftrace 虚拟化 IO 路径追踪方案主要功能包括 2 点：

1、支持虚拟化 IO 全链路追踪分析，即应用业务的单个 IO 请求从虚拟机到主机后端存储的整个 IO 栈；

2、支持 IO 请求跨用户态/内核态的完整生命周期分析，有助于从系统整体视角完成性能追踪及调优。

#### 3.2.2.2 关键技术

- **虚拟化 IO 路径全视图分析**

基于 QEMU-KVM 的虚拟化 IO 路径全视图如下图所示：

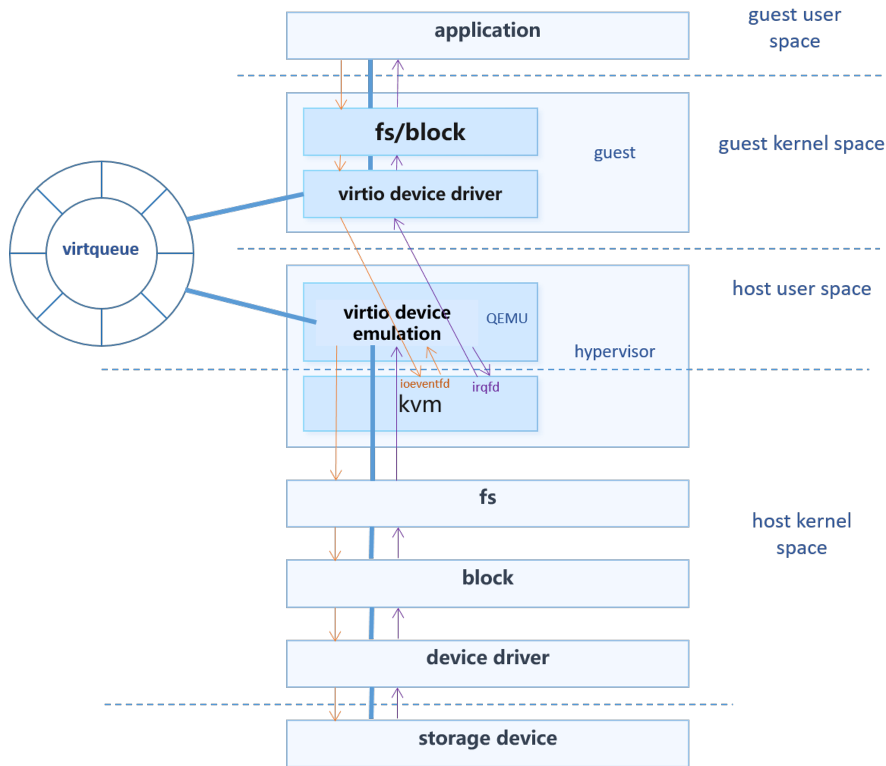


图 3-2-1 基于 QEMU-KVM 的虚拟化 IO 路径全视图

虚拟化 IO 路径分析工作主要从以下角度出发：

**1、Linux IO 栈分析**：对 VFS 层、块层、device mapper 层和设备驱动层原理及实现进行分析

**2、virtio 前后端分析**：完成 virtio 前端（virtio-blk/virtio-scsi、virtio-pci）、后端(QEMU vhost）实现、及前后端通知机制分析

**3、IO 路径层间接口分析**：梳理获得 IO 路径层间接口及栈内关键函数，是 IO 请求必经点，也是进行 IO 性能追踪的观测点。虚拟化 IO 路径层间接口主要有：

1）virtio 前后端之间接口

2）virtio 后端与 VFS 层接口

3）VFS 层与块层接口

- **基础追踪工具选型：bpftrace**

1、基于 eBPF 实现的类 Dtrace、systemtap 的追踪工具，相较于 systemtap 等具有更高的安全性和易用性；

2、支持 kprobe、uprobe、tracepoint、usdt 等追踪点，可基于 bpftrace 构建跨用户态与内核态、跨 IO 栈的追踪工具。

**IO 请求层间映射表**

1、IO 请求在 IO 栈层间会经过多次转换，通过构建 IO 层间映射表可对 IO 请求建立起层间映射关系，实现了 IO 请求全生命周期追踪的可能。

2、IO 请求层间映射表建立需通过解析文件布局来建立，以 qcow2 及文件系统 ocfs2 为例：

1）virtio-blk IO 请求到 qcow2 后端映射：需要解析 qcow2 L1、L2 表获得 IO 请求地址与后端文件内逻辑偏移地址的映射关系；

2）qcow2 虚拟磁盘文件到后端 lun 映射：解析 ocfs2 文件 extent map 得到文件内偏移地址与 lun 上逻辑地址映射关系。

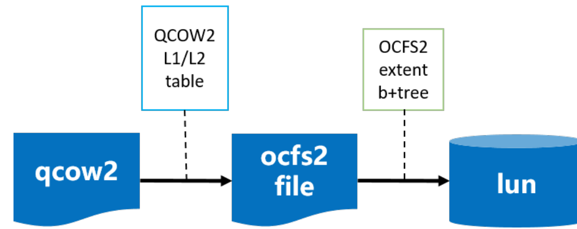


图 3-2-2 层间映射图

- **基于 eBPF 的 IO 性能追踪工具设计及原型验证**

基于 eBPF 的 IO 性能追踪工具由两部分组成：recorder（获取 IO 追踪原始数据）、reporter（解析原始数据，构建 IO 栈层间 IO 关联）。

**recorder**：

1、以 IO 路径关键追踪点作为 probe 构建 bpftrace 程序，实时追踪进程 IO 请求信息；

2、追踪结果按预定义格式保存到 raw 输出，供 reporter 解析；

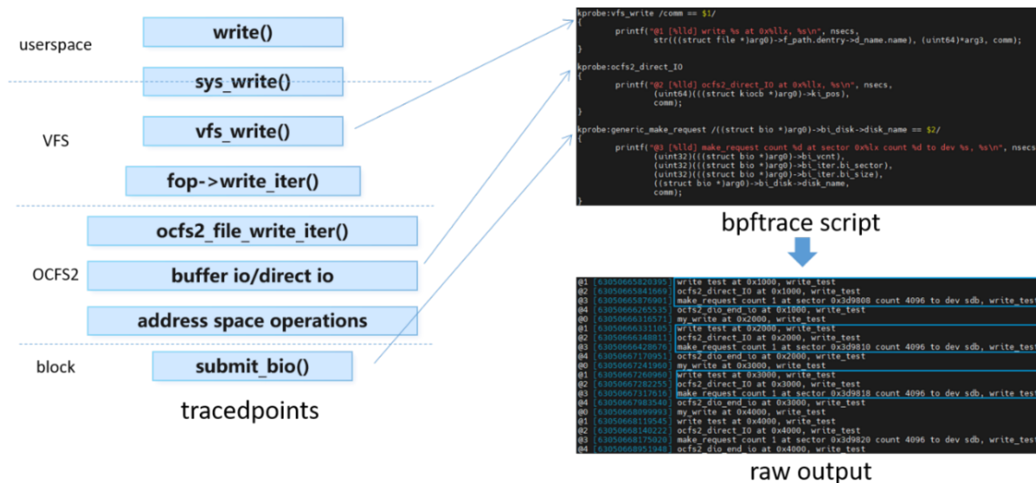


图 3-2-3 IO 性能追踪工具 record 部分工作过程示意图

**reporter**：

1、reporter 解析磁盘文件元数据、主机文件元数据等构建层间 IO 映射关系表；

2、解析 recorder 原始输出，基于 IO 映射关系建立起层间 IO 映射，输出所有 IO 请求在各个追踪点的时间戳信息；

3、基于 report 结果可进一步提取单个 IO 请求各层用时统计信息、最大延迟 IO 请求等。

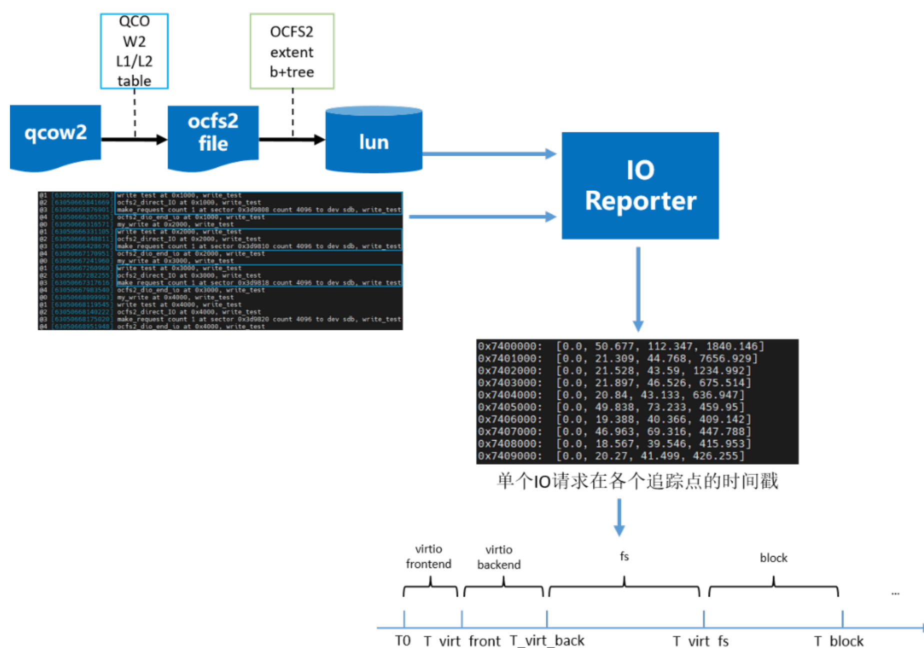


图 3-2-4 IO 性能追踪工具 reporter 部分工作过程示意图

- **基于 eBPF 的 IO 性能追踪工具应用场景演示**

**演示目标**：模拟 qemu io 线程下发 10 个 IO 请求，分析请求在 IO 栈上各层耗时信息。

**IO 性能追踪工具应用展示**：

1、 设置追踪点

1）重点关注文件系统接口，包括集群锁耗时、direct IO 各阶段耗时

2）块层只关注 IO 调度耗时

2、 IO 追踪结果：

1)可以观察 IO 请求在 IO 栈各层的统计信息：qemu（用户态）、vfs、块层

2）如果需要分析 IO 栈内部耗时，可增加栈内部的关测点，得到 IO 栈内部展开的耗时统计：实验中我们在 ocfs2 direct IO 关键函数中增加追踪点，能够进一步细化栈内耗时。

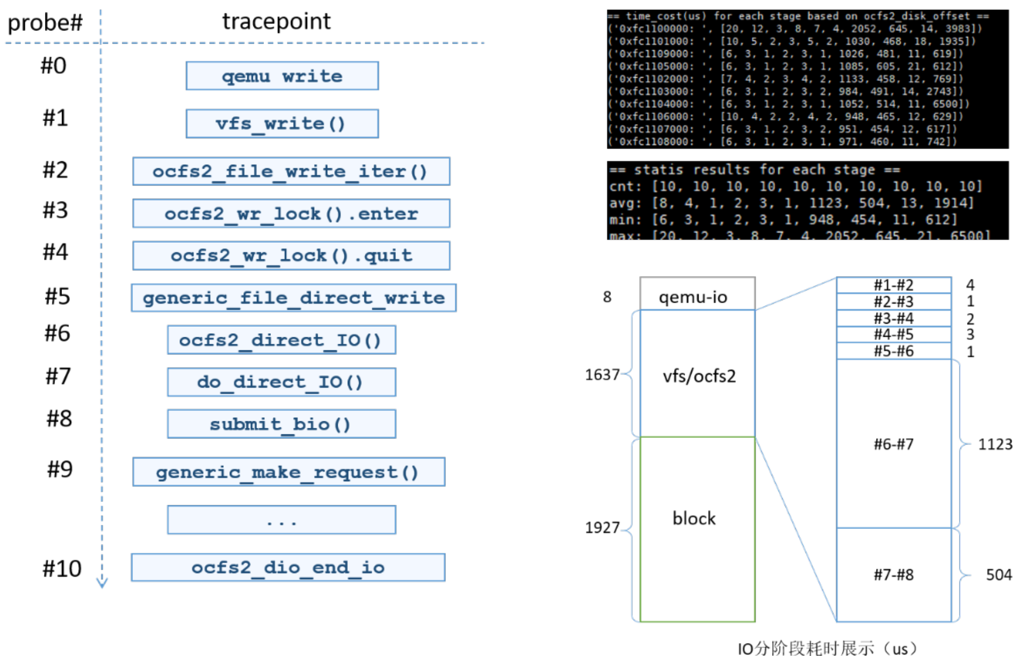


图 3-2-5 IO 性能追踪工具演示图

# 3.3 基于 eBPF 的 TCP 监控

## 3.3.1 TCP 监控面临的挑战

大量实际业务属于请求-响应的通信模式，比如数据库、Web 应用等。对这些应

用而言，端到端的连接数据是重要的监控指标。尤其对云上用户而言，其感知到的云应用的延迟是终端的延迟，也就是整个网络路径上的延迟。理想情况下，要求统计到路径上每个节点处理延迟以及网络上所有延迟的总和，以准确识别网络异常，定位网络问题。

然而，传统的 TCP 监控方法存在一定局限性，不能满足上述监控场景的要求：（1）采集到的信息有限：传统 TCP 监控工具往往仅提供连接粒度的信息，故只能感知到连接整体状态，比如收发数据量、RTT、拥塞窗口等，难以获取请求粒度的监控信息；（2）引入较高的额外性能负载：抓包是另一种常用的网络监控方式，虽然该方式能够获取到相当多的信息，但是运行时引入较大开销，应用中更多作为调试手段，而不能满足常态化监控的要求；（3）对用户业务逻辑有侵入：有一类跟踪方式，入侵式地在业务代码中添加埋点，记录各个节点状态信息，通过汇总分析，获取各阶段耗时等信息。但是这种方式依赖业务逻辑适配，实用性不足。总而言之，现有 TCP 监控方法难以应对云场景大规模网络下，端到端 TCP 常态化监控的需要。

现有的 TCP 监控工具主要有：

（1）netstat：可显示各种网络信息，包括网络连接、路由表、接口统计信息、连接信息等。对 TCP 层，能够显示连接状态、收发队列数据，统计丢包数量与原因等。仅采集到连接粒度的信息。

（2）ss：是一个用于查看网络套接字状态的工具，可以获取到多种 TCP 连接的信息。包括四元组、连接状态、窗口大小、重传信息等。与 netstat 类似，也仅针对连接粒度。

（3）tcpdump：是一种数据包分析工具，能够抓取网络数据包，采集数据包的收发时间戳信息与数据包内容。该方式会引入较大的性能开销。

（4）socket timestamp：能够在监控 TCP socket 的收发包时间戳，配合应用侧采集时间，能够获取到整个通信路径上的延迟。该方式需要用户修改业务逻辑适配。

以上监控手段不能满足上述场景下的监控需要，对实际业务场景而言，具有较大的局限性。

### 3.3.2 基于 eBPF 的 TCP 监控方案

为了克服了传统 TCP 监控方式的缺陷，龙蜥社区基于 eBPF 开发实现了一种低开销、可常态化的请求-响应模式的 TCP 时延监控工具 tcprt，一定程度上 tcprt 能够在 TCP 层，面向链路、TCP 连接、应用进行实时监控，为故障诊断、性能优化提供分析依据。在 Web 服务、MySQL 数据库服务等场景中具有应用价值。

tcprt 基于 eBPF 的 tracepoint 和 sockops 机制实现，通过对 TCP 连接收发包状态的跟踪，监控应用的“请求”与“响应”处理状态，并采集相关的网络参数，包括请求在协议栈中接收的时间，及服务进程处理过程中的耗时等数据信息。最终以日志形式传递到用户空间。

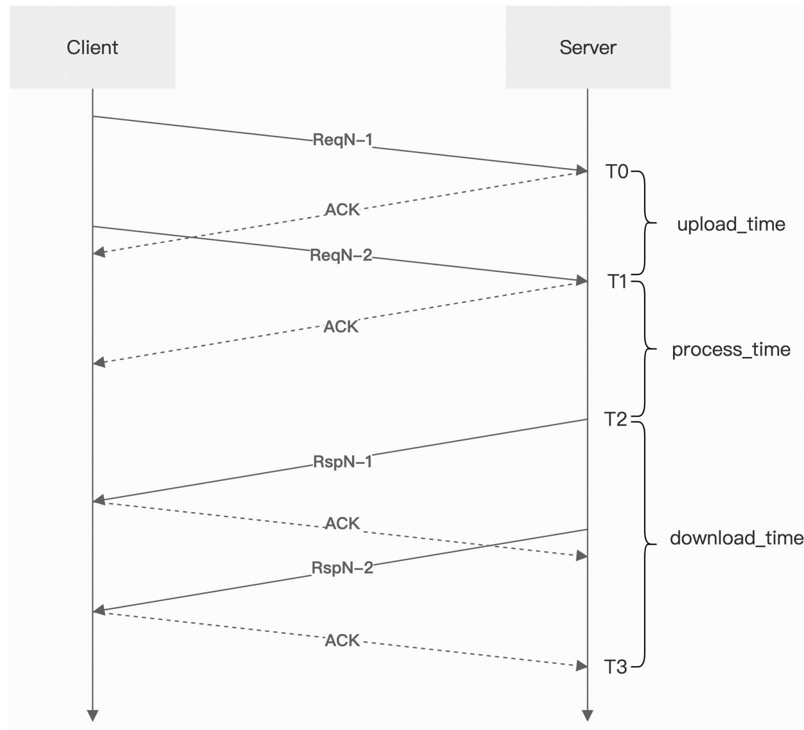


图 3-3-1 tcprt 信息采集示意图

上图展示了tcprt划分业务流量的各个阶段。客户端(Client)向服务端(Server)发送的第 N 个请求（request）表示为 ReqN。该请求由 ReqN-1 和 ReqN-2 两个数据包组成。服务端收到第一个包的时间被记录为 T0，第二个包的时间被记录为 T1。服务端收到请求后进行处理，完成处理后会向客户端发送两个响应的数据包 RspN-1 和 RspN-2，服务端发送第一个包的时间被记录为 T2。客户端收到响应包后会发送确认信息 ACK 给服务器，服务端收到最后一个 ACK 的时间被记录为 T3。

tcprt 基于以上的时间点的记录，可以采集到以下信息：（1）upload_time：用户请求上传所使用的时间；（2）process_time：服务端的处理时间。服务端从收到最后一个请求的数据包开始，到服务端向客户端开始发送响应的数据包为止，这段时间为服务端的处理时间；（3）download_time：数据下载时间。从服务端开始向客户端发送响应的数据包开始，到服务端收到客户端最后一个 ACK 为止，这段时间为

数据下载时间。对于下载比较大的数据响应，该信息的获取较为重要。此外，tcprt 还收集请求处理过程中的 TCP 连接 RTT、乱序、数据量等信息。

tcprt 已经开源到 Anolis，可以在 Anolis 系统（内核版本 5.10-17 及以上）上安装并使用 tcprt 功能，以采集请求-响应粒度的 TCP 连接状态。

tcprt 通过命令行，提供开关功能，并提供配置文件以便用户配置过滤条件。配置文件位于/etc/tcprt-bpf/tcprt.yaml，允许配置过滤端口或端口范围。tcprt 启动后，会将监控结果以 debugfs 的方式输出到/sys/kernel/debug/tcp-rt 路径下的 rt-network-log*为名称的文件中。文件名称的后缀为 CPU 内核的序号。

tcprt 将一对请求-响应视作一个 TASK，其提供的监控信息主要包括：

TASK 开始时间，开始时间的秒部分；

TASK 开始时间，开始时间的微秒部分；

TCP 连接的对端 IP 地址；

TCP 连接的对端端口；

TCP 连接的本地 IP 地址；

TCP 连接的本地端口；

TASK 发送的数据量，单位：Byte；

TASK 总耗时。即第一个请求 segment 到达与最后一个响应的 segment 被 ack 之间的时间间隔，单位：us；

TASK 最小的 TCP RTT（Round Trip Time），单位：us；

TASK 重传发送的 TCP 报文段（TCP segment）数量；

TASK 序号，TCP 建立后的第一个 TASK 序号为 1；

TASK 服务延时，即最后一个请求的 segment 到达与第一个响应的 segment 发送之间的时间间隔，单位：us；

TASK 上送延时。即第一个请求 segment 到达与最后一个请求 segment 到达之间的时间间隔，单位：us；

TASK 接收的数据量，单位：Byte；

TASK 过程是否发生接收乱序，参数值说明：1 表示发生；0 表示没有发生；

TASK 过程中 TCP 使用的最大报文长度（MSS），单位：Byte。

## 3.4 基于 eBPF 的网络性能优化

### 3.4.1 Linux 网络性能优化面临的挑战

#### 3.4.1.1 Linux 内核网络处理能力存在性能瓶颈

随着当前芯片等硬件制造工艺的持续进步，网卡支持的带宽及处理能力越来越强，100G/200G 大带宽网卡已经越来越广泛的应用于生产环境。与此形成鲜明对比的是，Linux 内核网络处理能力却无法跟上网卡的性能提升进度。有下面一组数据作为说明：

根据经验，在 C1（8 核）上，每 1W 包处理需要消耗 1%软中断 CPU，这意味

着单机的上限是 100 万 PPS（Packet Per Second）。假设要跑满 10G 网卡，每个包 64 字节，这就需要 2000 万 PPS (注：以太网万兆网卡速度上限是 1488 万 PPS，因为最小帧大小为 84B)，100G 网卡需要 2 亿 PPS，此时每个包的处理耗时不能超过 50 纳秒。而一次 CPU Cache Miss，不管是 TLB、数据 Cache、指令 Cache 发生 Miss，回内存读取大约 65 纳秒。从网卡到业务进程，经过的路径也较长，例如 netfilter 框架，其串行处理、线性规则匹配的特性，都带来一定的处理时延消耗，并容易引起 Cache Miss，更进一步降低了内核网络处理能力。

内核协议栈也提供了一系列的性能优化措施，比如网卡 RSS/多队列、中断线程化、分割锁粒度等，但其仍然基于整体的内核网络协议栈，对单条流处理并没有明显的性能提升，治标不治本。

#### 3.4.1.2 传统网络性能优化方案存在诸多问题

从根本上提升 Linux 网络处理能力需要完全或部分绕过内核网络协议栈，实现一种新的数据平面。在 eBPF 技术被广泛使用前，比较典型的网络性能优化方案包括基于网卡的硬件 Offload、以 DPDK 为代表的用户态数据平面等。

- **基于网卡的硬件 Offload**

硬件 Offload 的主旨思想是将网络处理功能下放到网卡硬件中进行，减少 CPU 的消耗及 Cache Miss，从而提高整体网络性能。该功能依赖于网卡硬件的支持，可以实现 checksum/gro 等特性卸载，部分智能网卡还能支持 ovs offload、rdma 等更多网络协议和特性卸载。但这种方案还存在以下问题：

**兼容性问题**：不同的网卡厂商和型号支持的 offload 功能可能有所不同，因此在

使用特定的 offload 功能时，需要确保网卡硬件和驱动程序的兼容性，否则可能会遇到不可预测的问题或性能下降。

**调试和故障排查困难**：当网络问题发生时，使用网卡 offload 技术时可能会增加调试和故障排查的难度。由于网络处理功能已经下放到硬件中，无法在软件层面直接观察和修改数据包，需要依赖专门的工具和技术进行调试。

**安全性考虑**：由于网卡 offload 将网络处理功能下放到硬件中，可能会对安全性产生影响。如果网络协议栈解析等敏感操作在网卡中进行，可能会增加系统面临的攻击风险。因此，在使用网卡 offload 技术时，需要评估和采取相应的安全措施。

**性价比问题**：能够支持高级特性卸载的网卡价格一般会比较昂贵，其所实现的卸载能力不一定具有通用性，带来的性能提升可能无法与其价格相匹配。

- **DPDK 用户态数据平面**

DPDK（Data Plane Development Kit）是一个用于构建高性能数据平面应用程序的开发工具集。它提供了一系列的库和驱动程序，使得应用程序可以直接访问和操作网络设备、存储设备等硬件资源，，从用户态直接收包，完全绕过操作系统内核，从而实现零拷贝和低延迟的数据包处理。DPDK 仍然存在以下问题：

**跨平台兼容性问题**：由于 DPDK 直接访问硬件资源和绕过操作系统内核，其在不同平台上的兼容性可能存在差异。某些特定的硬件设备或操作系统版本可能无法完全支持 DPDK，因此在选择使用 DPDK 时需要进行充分的兼容性测试和验证。

**配置和调优问题**：DPDK 提供了丰富的配置选项和优化参数，以适应不同的应

用场景和硬件环境。但是，正确配置和调优 DPDK 应用程序可能需要一定的经验和专业知识，尤其是在处理复杂的网络流量和高并发场景时。

**内存管理和安全性问题**：DPDK 使用了自己的内存管理机制，如 Hugepages 和预分配的内存池，以实现零拷贝和高性能。然而，这也增加了对内存管理的复杂性，需要仔细管理内存资源以避免内存泄漏和错误。此外，由于 DPDK 绕过了操作系统内核的访问控制，应用程序需要自行承担网络安全风险，需要进行适当的防护措施和安全审计。

**资源消耗问题**：DPDK 在数据包处理过程中需要维护大量的数据结构和缓冲区，以及相关的元数据，这些都需要占用一定的内存空间。DPDK 通过轮询模式来主动检查和处理网络接口上到达的数据包，这种模式也需要占用大量的 CPU 时间。在实际使用时，通常要为 DPDK 设置大页内存、绑定 CPU，其占用的资源相对较高。

## 3.4.2 基于 eBPF 的 Linux 内核网络性能优化解决方案

### 3.4.2.1 整体架构

- **基于 eBPF 的网络性能优化特点**

相比硬件 Offload、DPDK 等方案，基于 eBPF 实现网络性能优化具有灵活性强、资源占用少、应用场景更丰富、兼容性更高等特点。

**1、灵活性和可编程性**：eBPF 具有更高的灵活性和可编程性，可以在内核空间中编写自定义的网络功能。

**2、资源占用少**：eBPF 运行在内核中，不需要额外的运行时环境和驱动程序。

**3、应用场景丰富**：eBPF 适用于需要动态、灵活网络功能的场景，如网络加速、流量分析、安全策略和网络监控等。

**4、更好的兼容性**：采用 eBPF 的网络性能优化方案具有应用无侵入性，上层应用无需感知及改造；其可与传统网络路径兼容，不满足条件时可仍然按照内核网络协议栈默认流程处理；eBPF 也可以与硬件 Offload 结合使用，将可变成的网络处理过程完全卸载到硬件，实现更高的性能提升。

- **Linux 内核网络协议栈 eBPF 挂载点分析**

本章节主要分析在 Linux 内核网络协议栈中，eBPF 挂载点位置。在这些位置上加载 eBPF 程序，即可实现可编程的数据包处理，从而提升内核网络处理性能。

**发包流程**

在网络包发送流程中，eBPF 程序关键可挂载点如下图所示，主要包括两个位置：

1、socket 发送：当一个数据包被发送时，如果此时挂载了 socket eBPF 程序，此时会调用 tcp_bpf_sendmsg_redirect 函数，将数据包直接发送至接收端的 socket 队列，绕过了内核协议栈，效率极高。

2、TC egress：当数据包经网络协议栈后到达 sch_handle_egress 时，如果 TC egress 方向挂载了 eBPF 程序，此时会被调用。

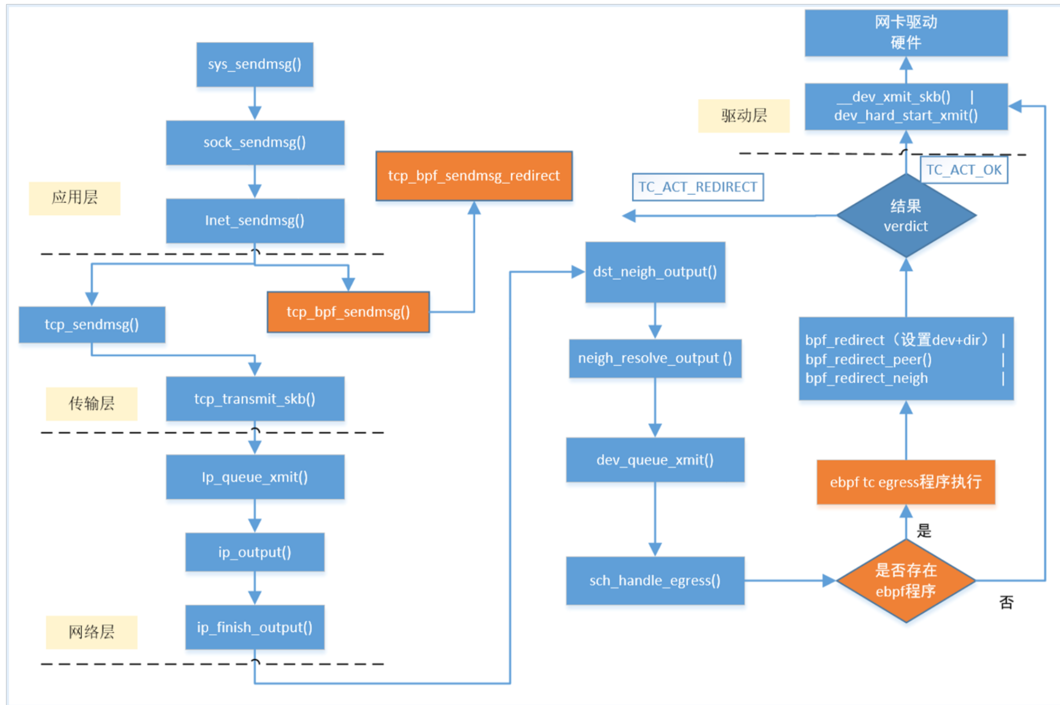


图 3-4-1 网络包发送流程示意图

**收包流程**

在网络包收包流程中，eBPF 关键可挂载点如下图所示，主要包括两个位置：

1、三种 XDP 模式：offload XDP、native XDP、generic XDP。其中 offload XDP 可直接挂载至可编程的智能网卡上，执行效率最高，但对硬件有一定要求。native XDP 挂载到网络设备的驱动上，它是 XDP 最原始的模式，因为还是先于操作系统进行数据处理，它的执行性能相对也很高。对于还没有实现 native 或 offloaded XDP 的驱动，内核提供了一个 generic XDP 选项，这是操作系统内核提供的通用 XDP 兼容模式，它可以在没有硬件或驱动程序支持的主机上执行 XDP 程序。在这种模式下，XDP 的执行是由操作系统本身来完成的，以模拟 native 模式执行，但性能相对较低。native XDP 挂载点在 poll 函数之后，在内核收包函数 reveive_skb 函数之前，而 generic XDP 挂载点在内核收包函数 reveive_skb 之后。

2、TC ingress：挂载点在 sch_handle_ingress 函数之后。

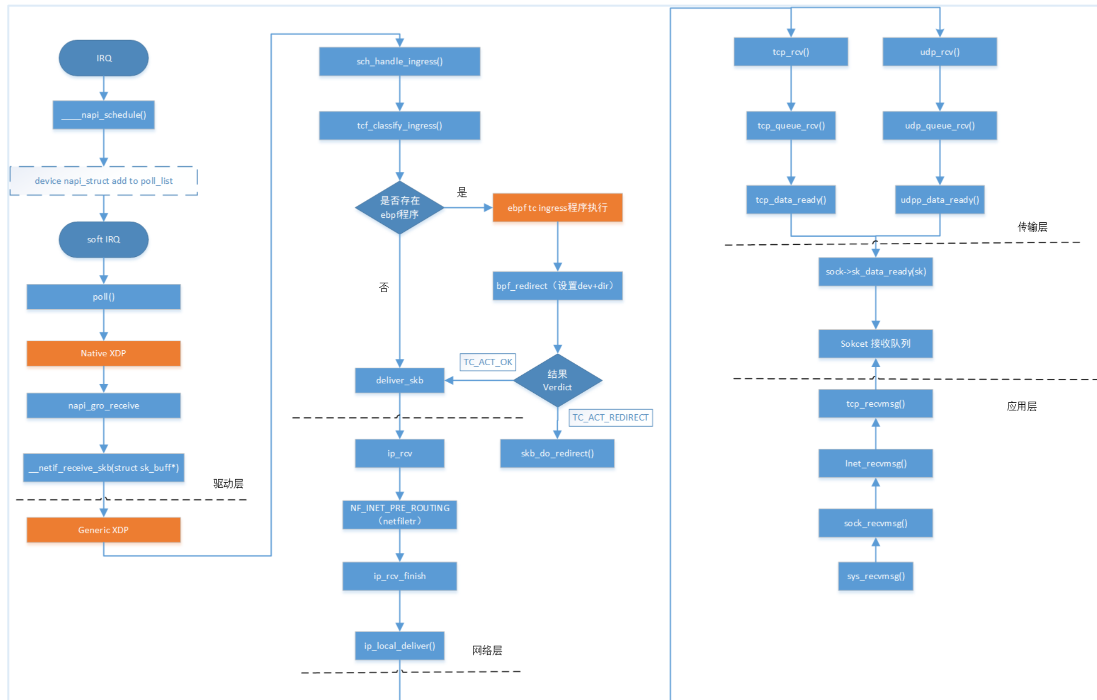


图 3-4-2 网络包收包流程示意图

### 3.4.2.2 应用实践

浪潮云海 InCloudOS 基于 Cilium CNI 及相关框架，使用 eBPF 技术进行了大量的网络加速开发和实践。

1、 基于 TC eBPF 加速同节点 Pod 通信

2、 基于 sockops/sockmap 加速同节点 Pod 通信

3、 基于 TC eBPF 加速跨节点 Pod 通信

4、 基于 Socket 拦截加速东西向 clusterIP 访问

5、 基于 XDP 加速南北向本地 NodePort 访问

6、 基于 DSR 加速南北向远端 NodePort 访问

7、 基于 TC eBPF 针对指定端口定向到指定网卡的流量加速转发，应用于 Pod DNS 请求/虚拟机间请求等场景

- **基于 TC eBPF 加速同节点 Pod 通信**

基于 eBPF 技术，可以在数据包从请求发起方 Pod 进入 host 网络协议栈前，在 TC ingress 处拦截处理并直接转发到服务方 Pod 的网卡入口，无需流经 host 网络协议栈，从而提升网络性能。相对于 Calico 等传统 CNI，TCP Throughput（1 stream）提升 21.41%，TCP-RR（1 process）提升 28.59%，TCP_CRR（1 process）提升 40.17%，如下图所示：

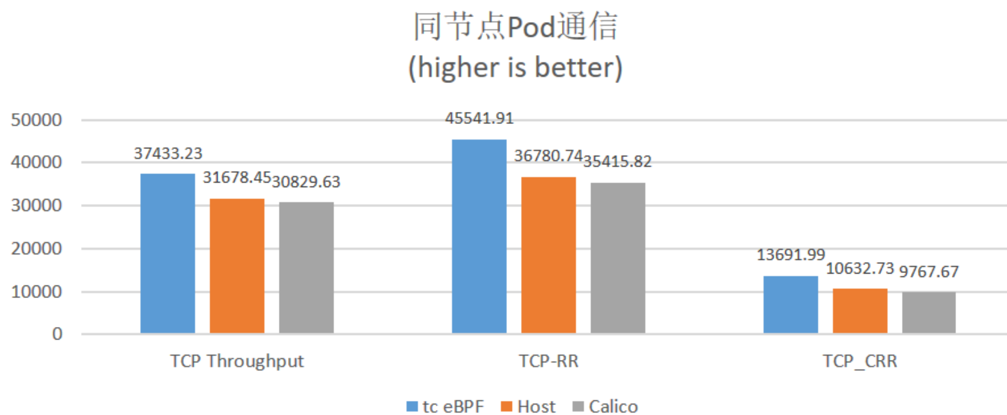


图 3-4-3 基于 TC eBPF 加速同节点 Pod 通信的效果展示图

- **基于 sockops/sockmap 加速同节点 Pod 通信**

基于 eBPF 技术可以监听所有的 tcp socket 事件并存储到 map 中。在 Pod 请求过程中，在 eBPF 挂载点针对所有 sendmsg 系统调用，从 map 里查找 socket 对端，并调用相关 eBPF 函数直接将数据发送到对端的 socket queue，从而绕过 Pod

所在网络命名空间的 netfilter 模块及全部 host 网络模块，实现了更高性能的同节点间 Pod 通信。相对于 Calico，TCP Throughput（1 stream）提升 94.42%，TCP-RR（1 process）提升 200.82%，TCP_CRR（1 process）提升 35.04%，如下图所示：

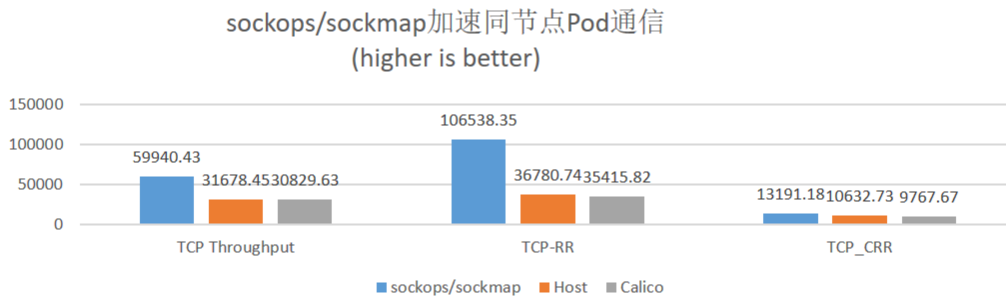


图 3-4-4 基于 sockops/sockmap 加速同节点 Pod 通信的效果展示图

- **基于 TC eBPF 加速跨节点 Pod 通信**

基于 eBPF，针对发包流程，当数据包流出 Pod 进入 host 网络协议栈时，在 TC ingress 处拦截处理并直接转发到物理网卡发出；针对收包流程，当数据包由物理网卡接收进入主机协议栈时，在 TC ingress 处拦截处理并直接转发到相应 pod。与传统 CNI (比如 Calico) 相比，此方案绕过了发起方、接收方 host 网络协议栈 netfilter 等模块，从而提升整体通信性能。相对于 Calico，TCP Throughput（1 stream）提升 24%，TCP-RR (1 process) 提升 43.51%，TCP_CRR (1 process) 提升 55.62%，如下图所示：

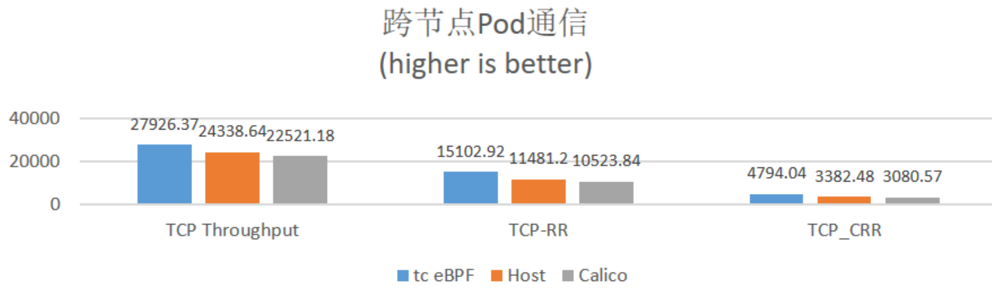


图 3-4-5 基于 TC eBPF 加速跨节点 Pod 通信的效果展示图

- **基于 Socket 拦截加速东西向 clusterIP 访问**

基于 Socket 拦截的东西向 clusterIP 访问在 Socket 挂载点完成 DNAT 操作，绕过了网络协议栈冗长的处理流程并且只需在建立连接时进行一次 DNAT 操作，能够带来大幅的性能提升。相比 kube-proxy 等传统实现方案，包转发率能够提高 140%，CPU 消耗降低 46%。

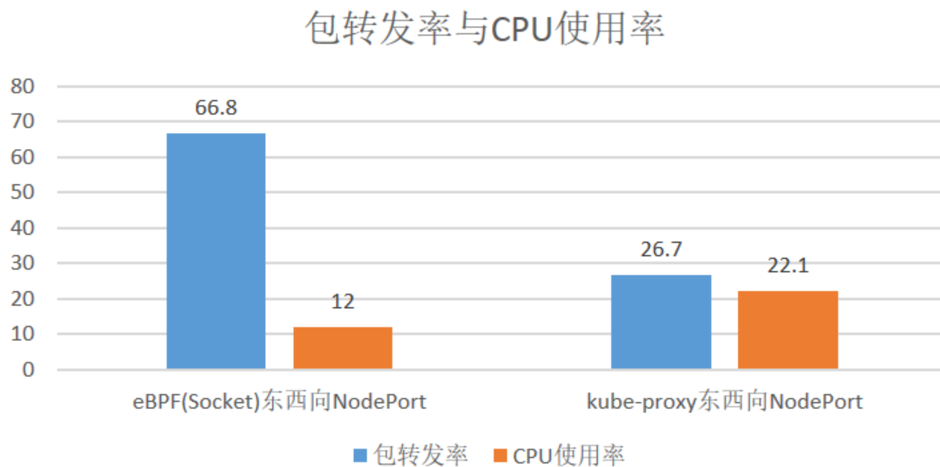


图 3-4-6 基于 Socket 拦截加速东西向 clusterIP 访问的效果展示图

- **基于 XDP 加速南北向本地 NodePort 访问**

基于 XDP 的南北向本地 NodePort 访问在网卡驱动层实现包转发转发，绕过了网络协议栈 netfilter 等处理流程，能够带来大幅的性能提升。相比 kube-proxy ipvs

模式，包转发率提高 138%，CPU 消耗降低 65%。

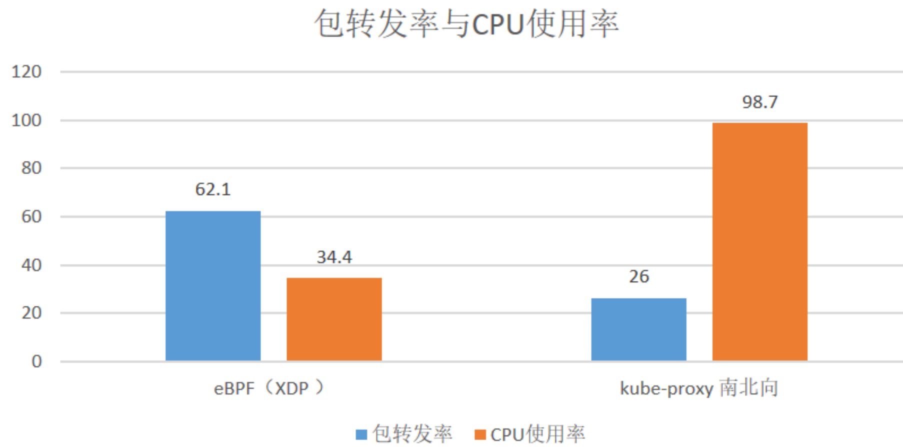


图 3-4-7 基于 XDP 加速南北向本地 NodePort 访问的效果展示图

- **基于 DSR 加速南北向远端 NodePort 访问**

基于 eBPF DSR 模式，在 NodePort 访问过程中，Pod 所在节点直接向外部客户端进行回复。相比 kube-proxy，基于 DSR 实现的南北向远端 NodePort 绕过了中转节点，能够带来大幅的性能提升，包转发率提高 136%，CPU 消耗降低 47%。

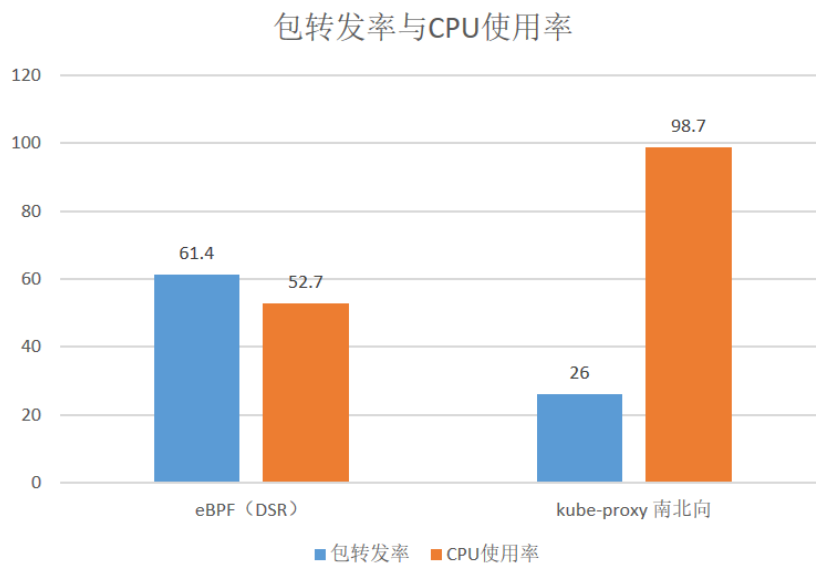


图 3-4-8 基于 DSR 加速南北向远端 NodePort 访问的效果展示图

- **基于 TC eBPF 针对指定端口定向到指定网卡的流量加速转发**

InCloud OS 基于 Cilium-eBPF 框架开发实现了针对指定端口定向到指定网卡的流量加速转发，可广泛适用于容器、虚拟化等云平台 DNS 解析、数据库访问等具有特定端口和特定网卡的应用访问加速。

以 K8S 容器云平台上 DNS 解析为例，其常用的 CoreDNS+Nodelocaldns（本地 DNS 缓存）方案均可以通过 TC eBPF 进行请求加速，如下：

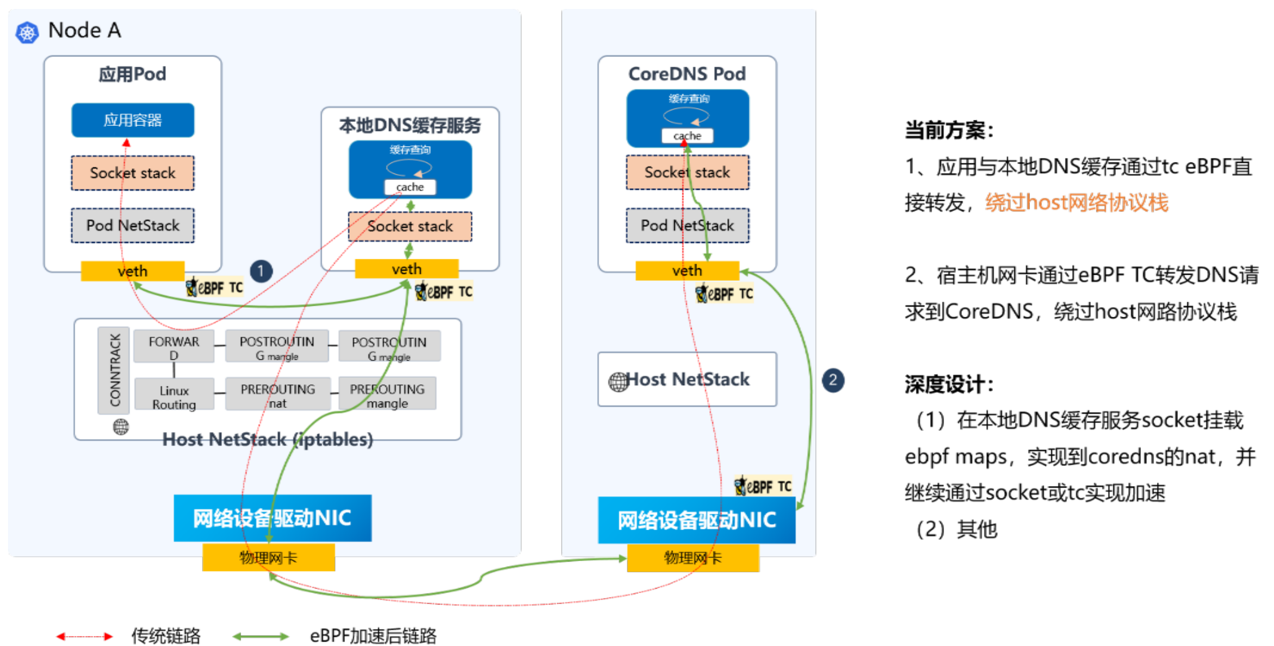


图 3-4-9 基于 TC eBPF 的流量转发路径示意图

1、应用与本地 DNS 缓存通过 TC eBPF 直接转发，绕过 host 网络协议栈。

2、宿主机网卡通过 eBPF TC 转发 DNS 请求到 CoreDNS，绕过 host 网路协议栈。

通过测试，DNS 解析整体延迟减少 20%以上，实现更加极致的性能体验。

## 3.5 基于 eBPF 的流量镜像

在数据库审计等场景中，为实现对应用操作性能、安全和统计的全面分析，往往从流量入手，通过对流入、流出应用的所有数据包进行分析，解析报文中的关键信息（比如五元组、时间等）并根据预定义的规则分析、统计并输出审计结果。为避免对应用流量直接操作带来功能异常或性能问题，常用方案是流量镜像，通过实时抓包或数据包复制方案获取所有的流量。

### 3.5.1 传统流量镜像面临的挑战

在数据中心中存在多种流量镜像方案，包括基于交换机端口镜像的硬件流量镜像、基于 iptables 或 ovs 的软件流量镜像等。这些方案在云平台中使用存在如下问题：

**流量精准度问题**：通过软件或者硬件指定某个接口，会将该接口上的流量全部复制，难以基于不同的云平台网络模型筛选出针对特定应用的流量（比如只镜像到数据库的流量），这增加了数据传输量以及后期流量筛选和分析的复杂度。

**兼容性问题：**iptables/ovs 仅适用于流量流经 netfilter 或 ovs 内核的场景中，无法用于通过 DPDK 或 eBPF 优化后绕过内核网络协议栈的网络场景。

**可扩展性问题**：基于交换机或 iptables/ovs 软件框架，较难扩展实现更多的筛选、解析等能力，无法快速满足各类差异化需求场景。

### 3.5.2 基于 eBPF 的流量镜像解决方案

基于 eBPF 实现流量镜像的总体方案如下：

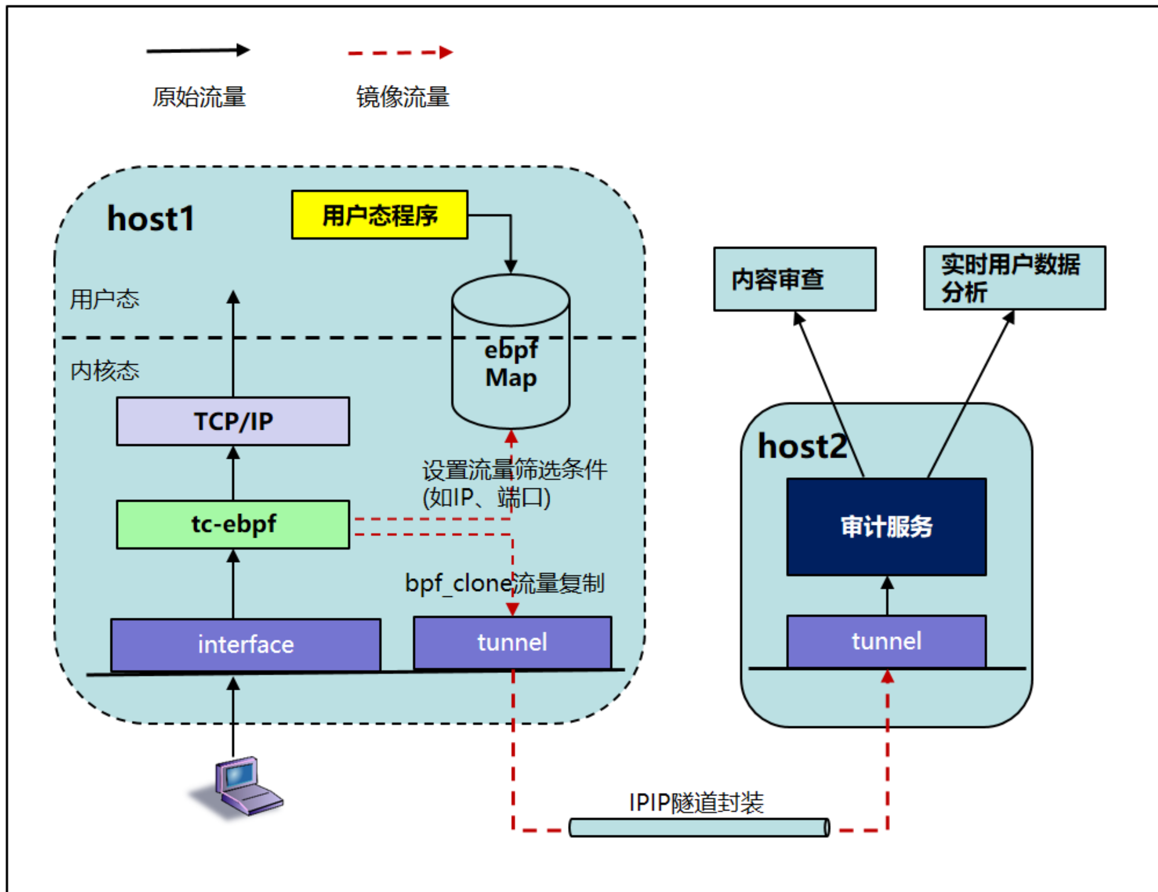


图 3-5-1 基于 eBPF 实现流量镜像的总体方案图

**1、加载 eBPF 程序**：在节点网卡 TC 位置的 ingress 方向挂载 eBPF 程序，拦截入口流量，支持通过 ebpfMap 形式动态开发和设置流量筛选、处理和转发规则。

**2、流量筛选**：当外部流量到达节点网卡后触发 eBPF 程序执行，从 ebpfMap 中获取筛选规则，复制符合条件的流量，原始流量继续放行。

**3、流量预处理**：对于复制的流量，根据预置规则进行预处理，比如取消云平台对流量进行的 overlay 封装，保证审计系统获取到的是原始流量。

**4、流量转发**：将处理后的复制流量转发至指定接口(bpf_clone_redirect)，通过 IPIP 封装经隧道转发至远端审计服务所在节点，完成流量审计。

相比传统方案，eBPF 流量镜像具有如下特点：

- 在内核态执行，轻量化、性能高、安全性高
- 基于软件实现，无特殊硬件要求，减少了整体方案复杂度
- 在网卡入口实现流量镜像，无需依赖内核网络协议栈，可兼容更多的云平台网络方案
- 可扩展性高，能快速编程实现各类差异化需求

### 3.5.3 应用实践

在某大型企业客服系统容器化迁移项目中，线上客服业务相关数据库从物理机部署变更为云原生 PaaS 数据库集群方案，其使用的基于硬件交换机端口镜像的数据库流量审计方案无法完全适用于容器场景：

- 基于云原生高可用机制，数据库服务不会固定到某个容器集群节点上，因此交换机端口镜像也无法指定端口配置
- 容器集群节点上服务密度高，指定交换机端口后采集的流量很大一部分并非数据库流量，存在无效流量多问题
- 容器网络通常采用 Overlay 方案，流量经过交换机端口时会带有封装，这样的流量采集后无法直接分析，还需要针对性的解封装，增加了整体方案复杂度

基于 eBPF 的流量镜像能够避免上述问题，实现更加精准有效的流量采集。其主要实施步骤包括：

1. eBPF 流量镜像管理服务以容器化形式运行在容器集群每个节点上，通过与容器集群控制平面交互，精准获取数据库服务的部署位置、容器 IP 等相关信息。

2. 数据库服务启动时，可将流量镜像相关 eBPF 程序挂载到容器网卡，实现仅对数据库服务流量进行采集，生成镜像流量。此处流量一般已经完成容器网络解封装，可以直接用于分析。

3. 针对镜像流量，还可以通过 eBPF 程序，按照指定要求进行报文处理，实现功能扩展。

4. 采集的流量通过 IPIP 方案，直接转发至审计服务，完成流量审计。

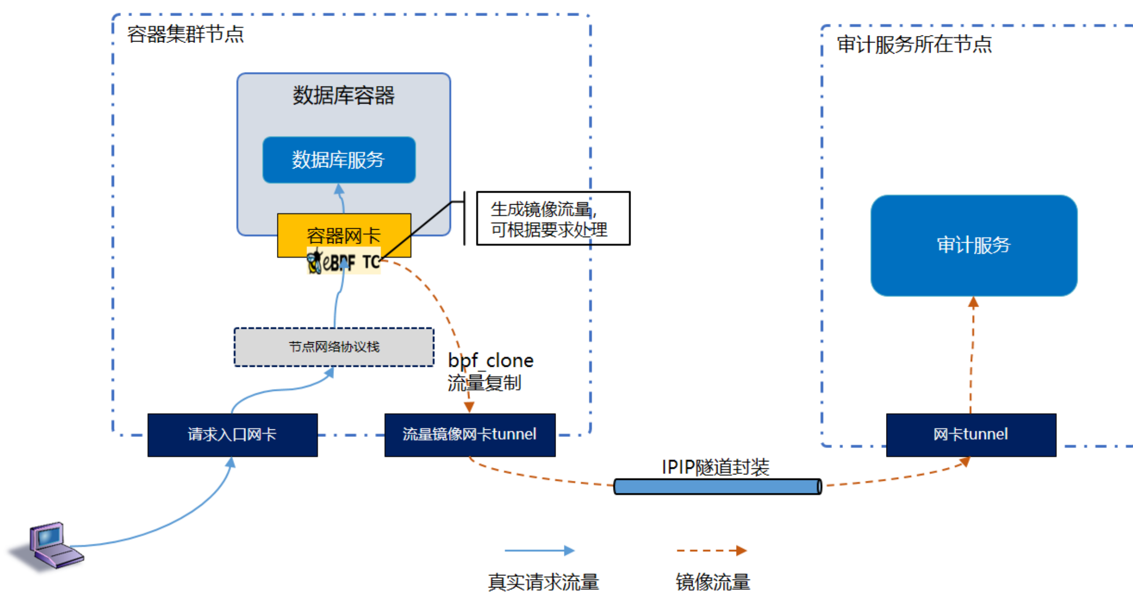


图 3-5-2 流量镜像实施步骤图

# 3.6 网络访问控制

## 3.6.1 传统网络访问控制面临的挑战

在云平台中，传统的软件网络访问控制（比如防火墙）通常基于 linux iptables

机制来实现，实际环境中通常会碰到如下问题：

**可维护性问题：**iptables 语句规则对运维专业性要求太高，且存在相互依赖关系，当出现问题时难以快速定位到具体的规则，不易于维护。

**兼容性问题：**可能与同样基于 iptables 实现数据面通信的云平台网络功能产生规则冲突；规则编写错误或操作不当，极易造成系统管理组件通信异常进而影响整体正常运行，往往需要进行详细的兼容优化和测试，提升了整体方案的复杂度。

**可扩展性问题**：受限于 linux 内核 iptables 框架，仅支持在五元组层面处理网络流量，无法根据报文其他信息实现更加精准的网络访问控制。

### 3.6.2 基于 eBPF 的网络访问控制解决方案

基于 eBPF 编写安全加固处理程序，并将其挂载在 XDP、TC 等 linux 操作系统网络流量入口，可以实现更加高效和灵活的网络访问控制。

相比 iptables 方案，其流量路径对比如下：

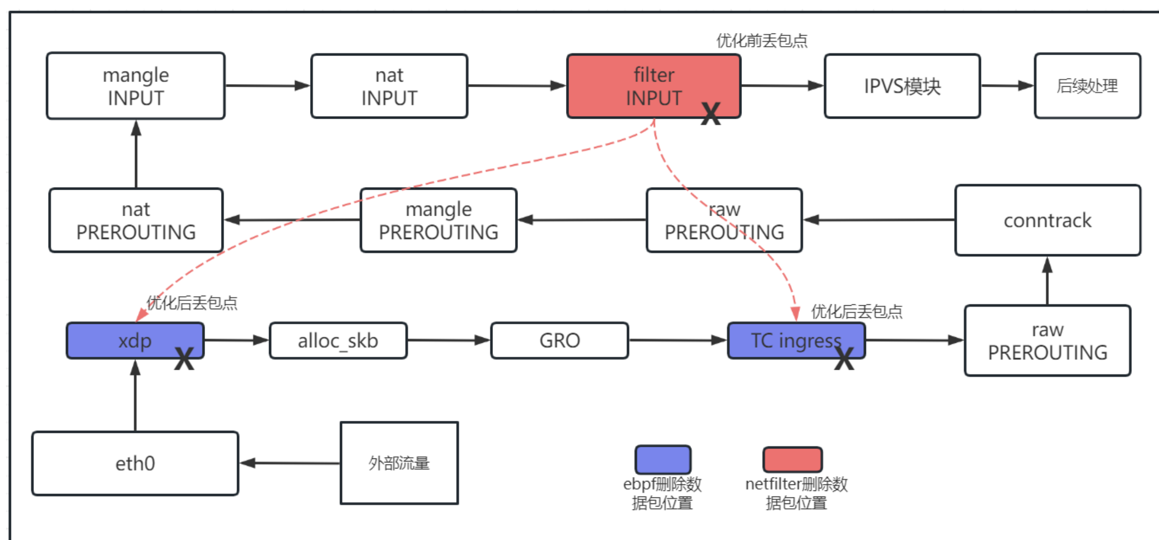


图 3-6-1 ebpf 与 iptables 的流量路径对比图

XDP 和 TC 在整个链路中位于比较靠前的位置，使需要被禁止的数据包在较早位置被丢弃，相比原有方案更优。在 XDP 和 TC 位置都可以解析到数据包中的 4 层信息，根据设置的条件即可执行特定动作，如返回 XDP_DROP 或 TC_ACT_SHOT 即可丢弃数据包。

eBPF 网络访问控制方案具有如下特点：

- **兼容性强：** eBPF 程序作用于操作系统网络入口，能够避免与运行在操作系统内核网络协议栈的 iptables、ipvs 等传统容器间通信、负载均衡等网络处理功能形成依赖或干扰，从而能够兼容不同网络插件。
- **性能更优：** 安全加固 eBPF 程序在操作系统网络入口就可以生效，缩短了网络包处理路径，整体性能可提升 20%以上。
- **可扩展性高：** 基于 eBPF 可编程的特点，可以快速开发满足更多的需求场景，比如提供更加灵活易用的安全加固语义规则、支持标记/镜像等更多的网络包处理方式。
- **易于观测：** 通过 eBPF 实现内核态与用户态交互，将被 eBPF 规则拦截的流量记录上报到用户态处理程序，从而可以直观的查询相关信息，便于问题的处理和统计分析。

### 3.6.3 应用实践

某银行清算系统部分服务运行于私有云平台互联网区，其通过传统的软件防火墙（iptables）方案满足其严格的互联网区网络访问控制要求。其云内流量访问控制

也通过 iptables 方案实现，因此两种规则混杂在一起，在系统运行过程尤其是频繁上下线业务期间常会出现配置不当、互联网与云内流量规则冲突引起访问控制异常的问题，增加了平台整体的运维复杂度。

基于 eBPF 的网络访问控制能够避免上述问题，同时还带来了其他的有益效果。

- eBPF 流量访问规则作用于互联网通信网卡 tc 入口，不影响云内流量路径
- 流量在互联网通信网卡 tc 处被限制，可以做到与云内流量更加安全的隔离

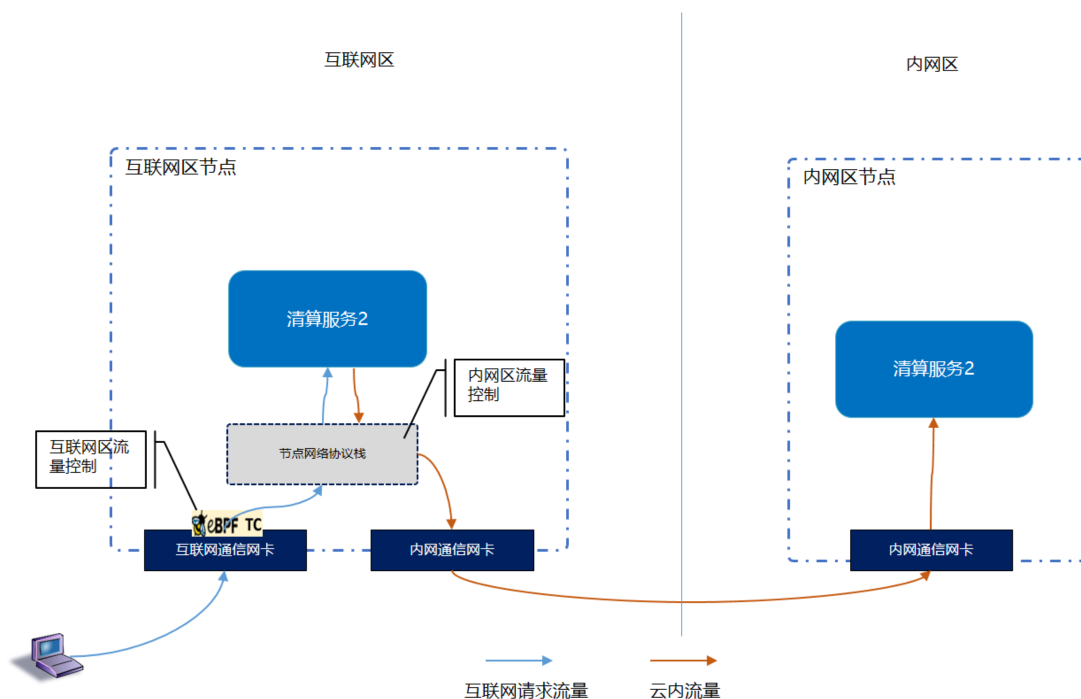


图 3-6-2 基于 eBPF 的网络访问控制流量示意图

## 3.7 优化基于 eBPF 的软件网络功能

### 3.7.1 基于 eBPF 实现网络功能的优势

eBPF 结合内核内的网络挂载点（例如 XDP 和 TC）后是一种有潜力的实现网络功能的方案。目前，基于 eBPF 实现的网络功能已经被很多公司应用于生产环境中，

成为云环境下基础设施的重要组成部分。例如 Meta 的负载均衡器 Katran, Google Cloud 目前使用基于 eBPF 的网络数据平面等。在学术研究和开源社区中，eBPF 也被广泛地用来实现网络功能。典型的例子有，学术研究: BMC (NSDI 2021), SPRIGHT (SIGCOMM 2023), Morpheus (ASPLOS 2022), Electrode (NSDI 2023), DINT (NSDI 2024) 等；开源项目: CIlium, PolyCube, Katran 等。因此基于 eBPF 实现网络功能逐渐成为一种趋势。这是因为 eBPF 拥有以下优势：

1. eBPF 作为一种起源于内核的技术，能够很好地集成到依赖于内核的云生态中。例如，根据 OvS 团队的论文，在主机内部容器通信中，内核 XDP 数据路径的性能优于 OvS 中的 DPDK 数据路径。

2. 相比于 DPDK 等方案，eBPF 实现了更好的性能和 CPU 利用率、安全、隔离性、运维成本之间的平衡。例如，eBPF 支持高性能的数据包处理而不会使 CPU 饱和，使得网络功能和非网络功能应用能够在同一设备上运行。

3. eBPF 允许动态加载用户代码然后安全地在内核中执行，无需修改内核源代码，从而提高了可维护性和灵活性，并加快了网络功能的开发和部署。

## 3.7.2 eBPF 实现网络功能面临的技术挑战

### 3.7.2.1 用 eBPF 无法实现特定的网络功能

因为 eBPF 对非连续内存的使用施加了严格限制，阻碍了部分网络功能核心组件的实现。例如基于跳表的 key-value store 和基于红黑树的优先级队列等。使用非连续内存意味着 eBPF 需要支持将可变数量的动态内存持久化。尽管最近的 Linux 内核（版本 6.1 及以上）支持分配动态内存并将其持久化到 BPF MAP 中，但验证器

强制规定了 BPF MAP 只能持久化固定数量的动态内存。因此，由于缺乏对可变动态内存的支持，现有的 eBPF 无法使用非连续内存。

例如，以下代码展示了 eBPF 目前支持动态内存，但无法支持可变数量的动态内存。

```
struct map_value {
    struct data_type __kptr *data;   //数量是固定的
}
```

#### 3.7.2.2 用 eBPF 实现网络功能性能次优

首先，eBPF 的 RISC 指令集缺乏对特定指令的支持，包括 SIMD 指令和 bitscan 指令（FFS 等），导致性能下降。例如，不支持 SIMD 导致了 eBPF 在实现网络功能时无法采用并行计算、并行查找等在网络功能中被使用的加速方式。在 sketch 等网络功能中，这会导致 49.2% 的性能下降。其次，eBPF 帮助函数 bpf_get_prandom_u32 对于网络功能来说性能开销太大。如果每一个包都调用一次 bpf_get_prandom_u32 导致 NitroSketch 46.6%的性能下降。

#### 3.7.2.3 现有的解决方案存在的缺陷

为了解决这两个技术挑战，可以考虑两种解决方案。第一种解决方案是增强 eBPF 的整体架构，例如扩展 eBPF 的指令集、增强验证器、引入新的运行时和语言级别的安全机制，以及将验证过程从内核解耦到用户空间。然而，由于对内核的修改过于激进，实用价值较低，不易于部署和推广。例如，扩展 eBPF 指令集需要对内核代码库中架构特定的 JIT 编译器进行修改，目前涵盖多达 14 种硬件架构。此外，扩展指令集要求对验证器的代码进行修改，因为验证器针对 eBPF 指令进行验证。然而

修改验证器可能会引入新的 bug 和安全问题。尽管重新设计 eBPF 的安全和编程架构在理论上是可行的，这种方案目前难以被直接部署，并且可能对以后的 eBPF 网络功能产生负面影响。

第二种解决方案是将所有功能无法实现的和性能下降的网络功能实现为内核模块（通过 kptr 和 kfunc 技术）或者集成到内核中 （实现为新的帮助函数和 BPF MAP)。然而，将所有网络功能集成到内核中将对内核造成巨大的改动，难以被内核社区接受。而根据需求集成单个网络功能，可能会由于需求变化而导致频繁的内核模块更换，进而导致内核不稳定。鉴于网络社区的快速发展，这种 "一个内核模块实现一种网络功能" 的方法可能会使内核变得相当不稳定。

### 3.7.3 基于标准库的优化 eBPF 网络功能技术方案

#### 3.7.3.1 整体架构

- **基于网络功能中的通用设计模式**

网络网络功能中存在一些通用的设计模式，总结如下：

1. 使用 bit scan 指令，例如 FFS (find the first bit), popcnt 指令等，实现快速检索。这种设计会被用在高性能的优先级队列的实现上，例如通过 FFS 快速定位第一个存放元素的 bucket 。
2. 同时计算多个 hash 函数。很多实现网络测量的网络功能，会使用一些基于概率和统计的数据结构，例如 sketch 和 bloom filter。同时计算多个 hash 函数来降低冲突概率。

3. 使用基础的数据结构。例如，top-k heap，桶链表等。

4. 使用随机数。为了提升性能，部分网络功能会根据概率执行特定的操作，例如一些 Heavy Hitter。

5. 使用非连续内存。例如使用跳表和红黑树等。

6. 将数据保存在连续内存中。例如，网络功能中的一些高性能的 hash 表，例如 DPDK 中的 cuckoo hash，将多个 key 保存在一块连续 bucket 中来降低 hash 冲突。

- **eBPF 网络标准库的设计和实现**

为了在不修改内核的前提下，解决上述的技术挑战，我们设计并实现一个可供 eBPF 调用的网络功能标准库 eNetSTL。eNetSTL 将上述的通用的模式抽象并实现为一系列高性能低开销的 API。在解决问题的同时，避免代码过度膨胀。eNetSTL 基于 eBPF 的 kernel function (kfunc) 和 kernel pointer (kptr) 技术实现，并将 API 实现在内核模块中，从而避免了内核的修改。目前 eNetSTL 的设计除了使用 kfunc 和 kptr 接口外，其他部分是 self-contain 的。因此能保持较好的内核版本的兼容性。eNetSTL 包含的内容如下图所示：

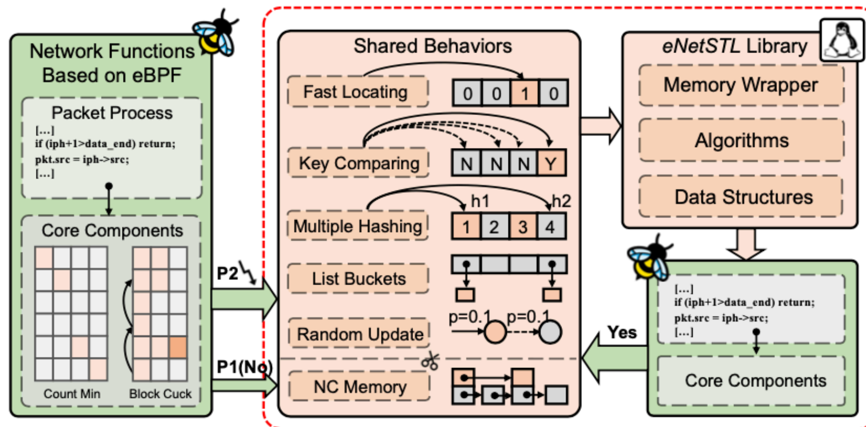


图 3-7-1 eNetSTL 包含的内容示意图

具体来说，eNetSTL 包含以下内容：

1. Memory wrapper： 支持在 eBPF 中使用非连续内存的同时，不破坏 eBPF 提供的安全保证。

2. 算法： 包括位运算、基于 SIMD 的并行 hash 计算和并行比较算法。

3. 数据结构：list bucket 数据结构，支持 GEO (几何随机数) 分布的随机数池。

其中 Memory wrapper 的实现充分利用了 kfunc 和 kptr 技术。其主要设计包括：

1. 通过用一个 proxy kptr 来管理所有新分配的 node kptr，避免 BPF MAP 中只能保存静态数量的 kptr。

2. 由 eNetSTL 管理所有的底层指针，通过 kfunc 实现节点到节点的指针路由，通过给 kfunc 增加 KF_ACQUIRE tag 来安全获取下一个节点的指针，并在 eBPF 中直接访问该指针，例如 a->b。

下面是 Memory wrapper 的部分 API：

```
BTF_SET8_START(bpf_ptr_structure_kfunc_ids)
BTF_ID_FLAGS(func, test_func)
BTF_ID_FLAGS(func, node_2_16_new, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, node_2_16_release, KF_RELEASE)
BTF_ID_FLAGS(func, node_2_16_get_child, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, node_2_16_set_onwer)
BTF_ID_FLAGS(func, node_2_16_release_child, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, node_2_16_write)
BTF_ID_FLAGS(func, ptr_create_node_container, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, ptr_destory_node_container, KF_RELEASE)
BTF_ID_FLAGS(func, ptr_alloc_node, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, ptr_release_node, KF_RELEASE)
BTF_ID_FLAGS(func, ptr_set_owner)
BTF_ID_FLAGS(func, ptr_unset_owner)
BTF_ID_FLAGS(func, ptr_container_set_tmp)
BTF_ID_FLAGS(func, ptr_container_get_tmp, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, ptr_connect)
BTF_ID_FLAGS(func, ptr_unconnect)
BTF_ID_FLAGS(func, ptr_get_out, KF_ACQUIRE | KF_RET_NULL)
BTF_ID_FLAGS(func, ptr_write_data1)
BTF_ID_FLAGS(func, ptr_write_data2)
BTF_SET8_END(bpf_ptr_structure_kfunc_ids)
```

#### 3.7.3.2 eNetSTL 使用技术实践

- **基于 eNetSTL 实现跳表**

通过 Memory wrapper API，直接在 eBPF 里使用非连续内存。我们用简化版本的单链表来展示使用非连续内存 （跳表的实现类似）：

```
typeof struct ptr_list_node {
    void* outs[1];
    void* ins[1];
    OTEHR_META_DATA;
    char data[DATA_SIZE];
} list_node;

void list_add(struct node_list *nl, list_node *head, void *data) {
    /*alloc new node with one out ptr and one in ptr*/
    list_node *new_entry = node_alloc(1, 1);
    if (new_entry == NULL)
      return; /*necessary to pass the verifier*/
    set_owner(nl, new_entry);  /*proxy-based memory management*/
    /*head->out[0]*/
    list_node *next = get_next(head, 0);
    if (next == NULL) {
      /*head->out[0]=new_entry;*/
      /*new_entry->in[0]=head;*/
      node_connect(head, 0, new_entry, 0);
    } else {
      node_connect(new_entry, 0, next, 0);
      node_connect(head, 0, new_entry, 0);
      node_release(next);
    }
    /*required by verifier*/
    node_write(new_entry, OFF, data, DATA_SIZE);
    /*the node will not be free because of set_owner*/
    node_release(new_entry);
}
```

性能测试结果（40G 网卡 单核性能）如下图所示（红色折线代表用内核模块实现，黄色折线代表用 eNetSTL 实现）：

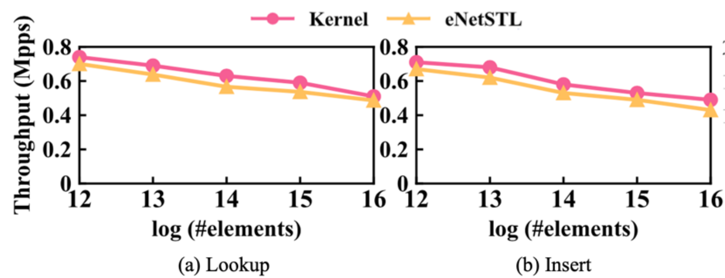


图 3-7-2 跳表的性能对比图

我们验证了跳表的查找性能和插入性能，可以看到使用 eNetSTL 在使能了原本无法直接实现的跳表的同时，其性能损耗在 10%以下。

- **基于 eNetSTL 实现 sketch**

sketch 是一种在网络测量领域常用的网络功能,其核心设计是使用多个 hash 函数将同一条流的数据包映射到多个 counter 上。我们使用 eNetSTL 的 API 来加速多个函数的计算，典型的 Count-min sketch 用 eNetSTL 实现代码如下：

```
struct cm_sketch {
    u32 cm[ROW_SZ][COL_SZ];
};
BPF_PERCPU_ARRAY(map, struct cm_sketc, 1);
static void countmin_add(struct cm_sketch *cmk, void *pkt) {
    //eNetSTL的API, 用SIMD计算HASH_NUM次hash, 对于第i个hash值, *((u32*)(cm) + (i % COL_SIZE) * ROW_SIZE + (hash % ROW_SIZE))++
    //这个API也可以用来优化实现Bloom过滤器, 只需要修改make_flags中的COL_SIZE为1即可
    hash_smid_cnt_u32(&cmk->cm, sizeof(struct cm_sketch),
                  pkt, PKT_SIZE,
                  make_flags(ROW_SIZE, COL_SIZE, HASH_NUM));
}
```

性能测试结果 (40G 网卡 单核性能) 如下图所示 （红色代表用内核模块实现，黄色代表用 eNetSTL 实现，蓝色表示用纯 eBPF 实现)：

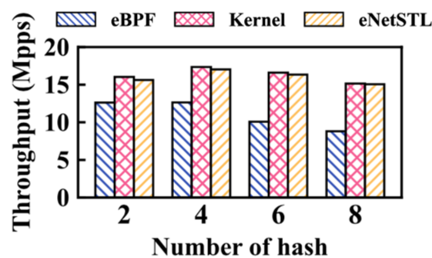


图 3-7-3 Count-min sketch 的性能对比图

实验结果显示，与 eBPF 相比，基于 eNetSTL 的实现平均性能提升了 47.9%。特别是，随着哈希函数数量的增加，这种提升变得更加显著，使用 8 个哈希函数时达到了 70.9%的峰值。这是由于随着哈希函数数量的增加，SIMD 指令能带来更多的优化效果。并且调用 eNetSTL 几乎不会带来性能损失。

- **基于 eNetSTL 优化 Cuckoo Switch 中的 hash 性能**

Cuckoo Switch 中使用了 Blocked Cuckoo hash 这一核心数据结构。相比于

原始的 Cuckoo hash， Blocked Cuckoo hash 为了降低 hash 的冲突率，在一个 bucket 中同时保存 16 个 hash 指纹。我们参考 DPDK 的实现，使用 eNetSTL 提供的 `hw_hash_crc`（用硬件指令生成 crc 来代替 hash 计算）和 基于 SIMD 的并行比较算`bpf__find_mask_u16`分别优化 hash 的计算、hash 指纹的比较、和 full-key 的比较。下面是一个简化后的例子：

```c
struct __cuckoo_hash_key {
    cuckoo_hash_value_t value;
    cuckoo_hash_key_t key;
};

struct __cuckoo_hash_bucket {
    uint16_t sig_current[CUCKOO_HASH_BUCKET_ENTRIES];
    __cuckoo_hash_key_idx_t key_idx[CUCKOO_HASH_BUCKET_ENTRIES];
};

struct cuckoo_hash {
    struct simple_rbuf__cuckoo_hash_free_slots free_slot_list;
    struct __cuckoo_hash_key key_store[CUCKOO_HASH_KEY_SLOTS];
    struct __cuckoo_hash_bucket buckets[CUCKOO_HASH_NUM_BUCKETS];   //一个bucket存放多个hash指纹

    uint32_t initialized : 1;
};

BPF_PERCPU_ARRAY(map, struct cuckoo_hash, 1);

/*查找函数*/
static int32_t __cuckoo_hash_search_one_bucket(struct cuckoo_hash *h,
                    const cuckoo_hash_key_t *key,
                    uint16_t sig,
                    cuckoo_hash_value_t **data,
                    struct __cuckoo_hash_bucket *bkt)
{
    uint32_t i;
    __cuckoo_hash_key_idx_t key_idx;
    struct __cuckoo_hash_key *k, *keys = h->key_store;

    uint32_t mask, delta;
    //利用eNetSTL的并行比较
    mask = bpf__find_mask_u16(bkt->sig_current, CUCKOO_HASH_BUCKET_ENTRIES, sig) &
           ~bpf__find_mask_u16(bkt->key_idx, CUCKOO_HASH_BUCKET_ENTRIES,
CUCKOO_HASH_EMPTY_SLOT);
    if (mask == 0) {
        goto not_found;
    }

    //对于匹配的hash finger执行完整的key比较
    __for_each_u16_avx(i, mask, delta)
    {
        asm_bound_check(i, CUCKOO_HASH_BUCKET_ENTRIES);
        key_idx = bkt->key_idx[i];
        k = keys + key_idx;
        //使用SIMD优化的full key比较
        if (__cuckoo_hash_cmp_eq(key, &k->key, h) == 0) {
            *data = &k->value;
            return bkt->key_idx[i] - 1;
        }
    }

not_found:
    return -1;
}
```

性能测试结果 （40G 网卡 单核性能） 如下图所示 （红色的折线代表用内核模块实现，黄色的折线代表用 eNetSTL 实现，蓝色的折线表示用纯 eBPF 实现）：

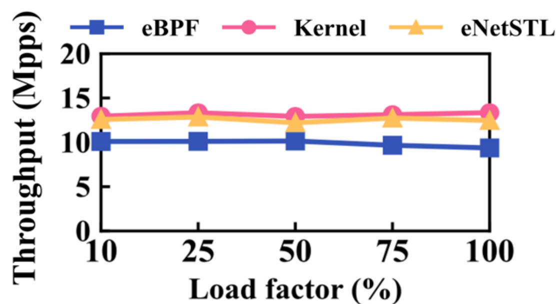


图 3-7-4 hash 的性能对比图

使用了 eNetSTL 的方案与纯 eBPF 相比，平均性能提升 27.4%，并且随着负载的增加，性能提升更加明显，在满负载时达到 33.08%。这是因为，随着负载增加，单个条目中的平均比较次数也增加。基于 SIMD 的并行比较优化效果变得更好。在

低负载场景下，优化主要体现在使用 hw_hash_crc 替代基于软件的哈希计算和 SIMD 优化的 full key 比较。与内核相比，采用 eNetSTL 的方案平均性能损失约为 4.30%。

## 3.8 基于 eBPF 的安全实践

### 3.8.1 传统解决方案面临挑战

传统安全检测和防御方案采用内核模块技术，内核模块技术是通过编写内核模块来扩展操作系统的功能，内核模块可以直接访问和修改操作系统内核，可以实现高级别控制和丰富的功能，但编写不当的内核模块可能导致内核崩溃或引入安全漏洞。

eBPF 提供了一种安全、可编程的方式来扩展内核功能，eBPF 程序在内核中运行时会受到严格的安全限制，因此不会对系统的稳定性和安全性产生直接影响，可以实现深度的系统观测能力和自定义扩展能力。

内核模块技术和 eBPF 技术对比分析如下：

表 3-8-1 内核模块技术和 eBPF 技术对比

| 类别 | 内核模块技术 | eBPF 技术 |
| --- | --- | --- |
| 内核函数 | 可以调用内核函数 | 只能通过 BPF 辅助函数调用 |
| 安全性 | 容易引入内存泄露、访问越界等错误、导致内核崩溃 | 通过验证器进行检查，有效保障内核安全 |

| 可移植性 | 需要编译内核，基于相同内核运行 | 不需要编译内核（JIT），**CO-RE** |
|---|---|---|
| 数据交互 | 通过系统 API、日志或文件，可能存在较大内存拷贝损耗 | 通过 eBPF map 数据结构，数据类型丰富，高效易用 |
| 升级支持 | 需要卸载和加载，可能导致处理流程中断 | 支持平滑升级，不会造成处理流程中断 |
| 入门门槛 | 需要深入了解操作系统内核的内部结构和机制，入门门槛高 | 需要学习新的编程语言和概念，相对内核模块技术来说，入门门槛相对较低 |

## 3.8.2 基于 eBPF 的新一代安全解决方案

浪潮信息云峦服务器操作系统 KeyarchOS 提供轻量化的安全防御组件 KSecure，采用 eBPF 技术路线，提供主机、容器安全检测和防御能力，在增强操作系统安全性和合规性的同时，解决传统内核模块方式带来的系统稳定性和性能问题。

### 3.8.2.1 主要功能

KSecure 安全防御组件的主要功能如下：

**1、关键文件/进程防护**：支持文件和目录的防护，防止核心业务文件被篡改、删除等行为。支持关键进程防护，保护核心业务进程不被恶意终止、删除、信息注入；

**2、主机入侵检测**：基于规则引擎可以对黑客的入侵行为进行检测和自动处置。基于“诱饵”行为监测的勒索病毒防御，及时发现和阻止勒索病毒加密行为；

**3、容器逃逸防护**：审计容器进程的命令行参数，拦截存在容器逃逸行为的操作；监控容器进程的操作，结合上下文数据计算操作对象是否为宿主机文件，拦截非容器文件系统的文件操作；

**4、容器入侵检测**：监控容器内的文件、进程、网络行为，基于规则引擎识别黑客入侵行为，结合用户态采集的容器信息，定位被入侵容器信息并以告警形式通知用户。

**5、安全基线检测**：基于等保和 CIS 标准形成知识库，提供基于模板的基线检测、修复和回退功能。帮助发现身份鉴别、访问控制、安全审计、入侵防范、剩余信息保护等方面潜在的安全风险，支持基线值自定义和灵活扩展；

**6、安全管理：**支持安全特性动态加载、对安全组件 CPU 资源占用限制、安全策略热加载、服务启停等管理等功能。

图 3-8-1 KSecure 安全组件功能架构图

### 3.8.2.2 整体架构

基于 eBPF 的系统内多层次 hook 技术，将 eBPF 程序 hook 到操作系统内核的多个层级（LSM、syscall、network、kprobe 内核函数），其中 LSM、syscall、network 的 hook 点具有监控和拦截能力，kprobe 内核函数 hook 点只具有监控能力。通过在各个 hook 点加载安全策略实现对系统和应用程序行为的监控和拦截。

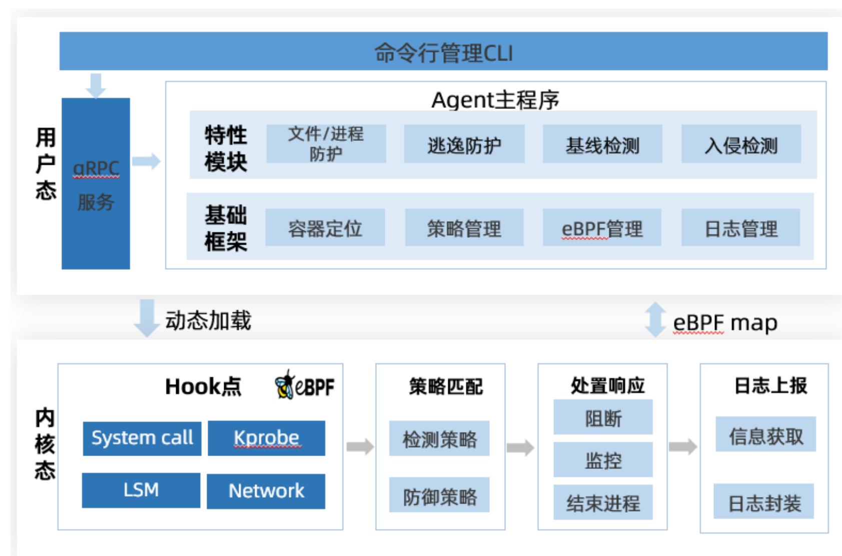

图 3-8-2 KSecure 安全组件技术架构图

#### 3.8.2.3 基于 eBPF 的主机安全

在当今数字化时代，操作系统安全的重要性愈发突显。Linux 操作系统漏洞曝出频率呈现逐年上涨的趋势，新型攻击层出不穷，攻击者利用漏洞实现权限提升、业务关键信息的获取等，严重破坏了操作系统的机密性、可用性和完整性。

**1）关键技术**

在 KSecure 主机安全中，涉及如下两个关键技术。

- **eBPF-LSM hook 技术**

LSM（Linux Security Modules）是 Linux 内核中用于支持各种计算机安全模型的框架，用于在 Linux 内核中实现安全策略和强制访问控制。Linux Kernel 5.7 引入了 LSM 扩展 eBPF(简称 BPF-LSM)，而无需配置 LSM 模块(SELinux、AppArmor 等）或加载自定义内核模块，通过在 LSM 层面的文件、进程、网络等 hook 加载 eBPF 程序，获取应用的异常行为，通过与内置和自定义的安全策略对比后，进行细粒度的（函数级）实时拦截，实现入侵检测和关键文件和进程防御功能。

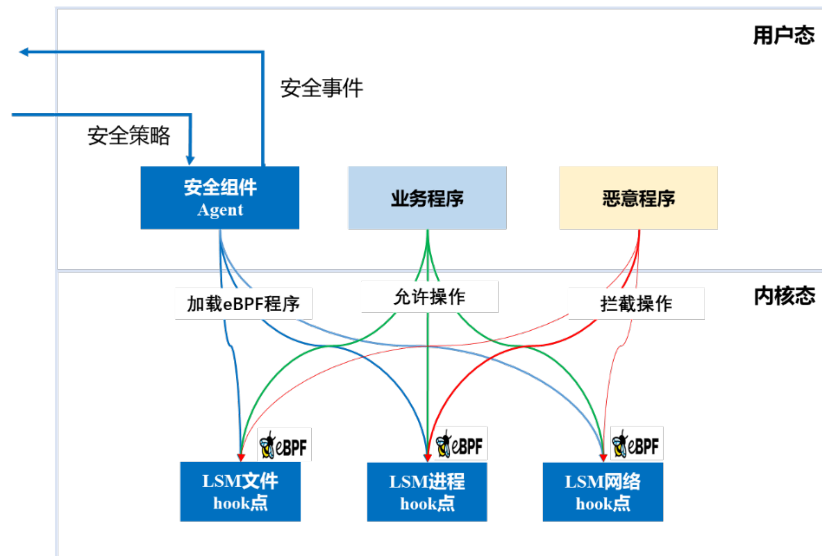


图 3-8-3 关键业务防护架构图

以文件防护为例，介绍 KSecure 安全组件如何进行文件防护，如下图所示：

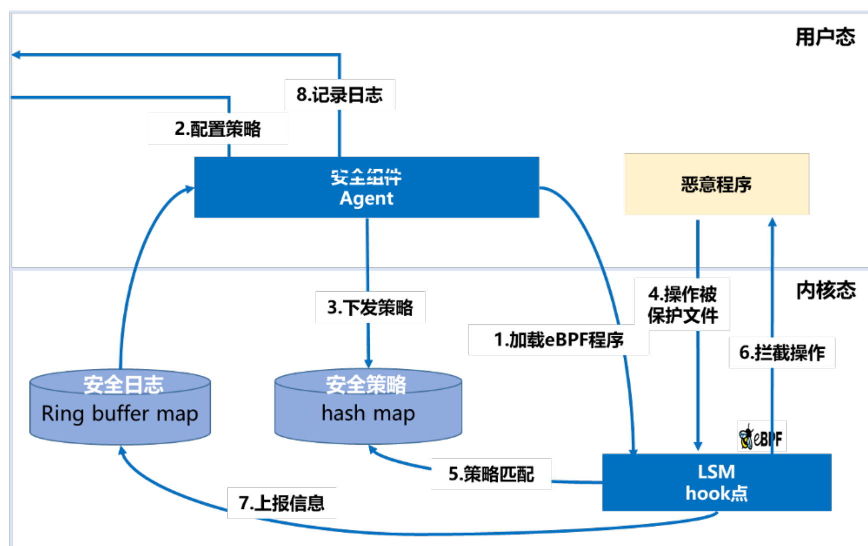


图 3-8-4 文件防护示例流程图

**1、加载 eBPF 程序**：KSecure 安全组件用户态 Agent 程序将 eBPF 程序加载到 LSM 的 Hook 点上；

**2、配置策略**：通过 KSecure 安全组件的策略配置文件（YAML 格式）进行策略设置；

表 3-8-2 文件防护规则配置说明

| 被保护对象 | 信任对象 | 权限 | 防护目标 |
|---|---|---|---|
| /test/ | /test/bin/ | 所有权限 | 完全信任业务软件自身程序，对业务软件的程序不做任何拦截，保障业务软件运行不受到干扰 |
| /test/bin/ | -- | 无权限 | 其他任何程序都无法终止以及对业务软件注入恶意代码 |

**3、下发策略**：安全组件的 Agent 将 YAML 策略解析至内核态创建的 eBPF-map；

**4、黑客入侵**：攻击者对被保护的文件进行编辑或删除等操作，进入内核 LSM hook 点，触发对应的 eBPF 程序；

**5、策略匹配**：内核中 eBPF 程序获取主体进程和客体路径等信息，与存储在 eBPF-map 安全策略和匹配。在获取主体进程时，采用进程链跟踪技术，跟踪进程的调用过程，信任进程调用的进程/脚本等均继承权限；

**6、操作拦截**：hook 点对应的 eBPF 程序阻止编辑和删除被保护文件的操作；

**7、上报信息**：eBPF 程序通过 eBPF-map（Ring buffer 类型）上报给 Agent；

**8、记录日志**：Agent 封装匹配策略信息并记录到安全日志文件。

- **基于 eBPF 的内核监控技术**

将 eBPF 程序通过 kprobe、tracepoint 技术挂载至内核，监控系统中的文件操作、进程创建、网络连接等行为。基于 MITRE ATT&CK（Adversarial Tactics, Techniques and Common Knowledge，即对抗战术和技术知识库）框架构建入侵检测内置规则，结合自定义的检测规则为入侵检测引擎提供判断依据，实现入侵事件识别和攻击阻断。KSecure 安全组件入侵检测功能主要由数据采集、数据缓冲、预处理、规则引擎等模块构成。

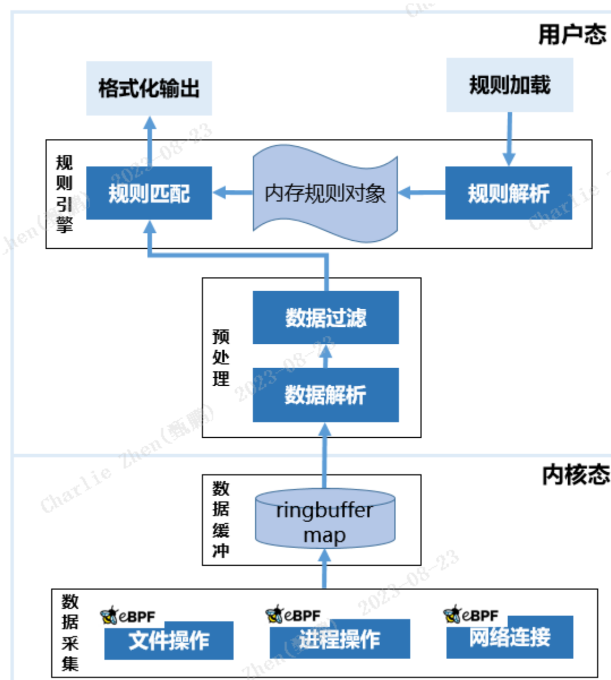


图 3-8-5 入侵检测功能模块架构图

**1、数据采集：**借助 eBPF 程序采集文件、进程、网络等系统调用入参以及返回值，并上传至数据缓冲区；

**2、数据缓冲：**设置 Ring buffer 类型的缓冲区用于已采集数据临时存储，eBPF 程序可将采集数据写入缓冲区，预处理模块循环读取缓冲区数据；

**3、预处理：**将缓冲区数据根据不同的系统调用解析成特定的数据结构，并通

过事件类型等条件，丢弃不符合条件的数据，便于规则引擎执行规则匹配；

**4、规则引擎：**将已加载的规则解析成内存对象，拉取预处理后的数据与内存中的规则对象进行比对，与规则匹配的数据诊断为入侵行为，格式化后输出。

以反弹 shell 入侵检测为例，介绍 KSecure 安全组件如何检测入侵行为，如下图所示：

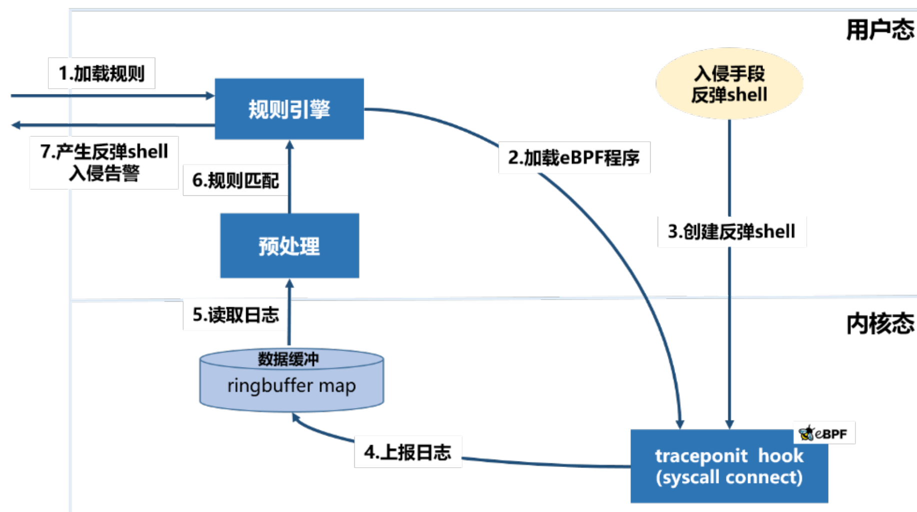


图 3-8-6 反弹 shell 入侵检测示例流程图

**1、规则**：读取 yaml 格式的检测规则至内存，逐条解析规则，将规则关键字解析为数据对象，并将规则内容保存至对象内，便于规则匹配，以下为反弹 shell 入侵行为匹配条件；

表 3-8-3 反弹 shell 入侵检测规则配置说明

| 匹配对象 | 匹配值 | 检测说明 |
| --- | --- | --- |
| 系统调用 | connect | 检测系统的网络连接 |

| 程序类型 | 自定义 | 检测创建网络连接的程序类型 |
|---|---|---|
| 文件描述符 num | 0,1,2,3 | 反弹 shell 关键特征，创建的网络连接文件描述符 num 为 0、1、2、3 |
| 文件描述符 type | IPv4\IPv6 | 连接类型为 IPv4\IPv6 |

**2、BPF 程序：**规则引擎将 eBPF 程序通过 tracepoint 和 kprobe 技术挂载至系统调用函数。具体而言，对于反弹 shell 的检测，则挂载至 connect 系统调用；

**3、反弹 shell：**反弹 shell 是一种网络攻击技术，用于通过远程主机上的 Shell(命令行终端)与攻击者建立连接，从而控制受攻击主机；

**4、日志：**将系统产生的网络连接上传至数据缓冲区；

**5、日志：**监听 Ring buffer 中网络连接信息，通过 BPF 辅助函数循环读取数据，并将数据转交预处理模块；

**6、匹配：**将解析并经过滤后的网络连接日志对象与规则对象比对，匹配规则的网络连接诊断为反弹 shell 入侵；

**7、告警：**将日志信息按照规则配置的格式输出为告警信息.。

**2）应用场景**

基于 eBPF 技术安全增强组件具以下几个方面的应用场景：

**1、安全加固：**提升操作系统的合规性和安全性。支持用户结合实际的安全需求选择实施加固和扩展，以便更好地满足对不同用户不同场景的配置安全基准要求。

**2、黑客入侵：**降低因系统漏洞利用导致的提权攻击、Rootkit 攻击、进程注入等风险。及时发现并阻止勒索病毒加密行为，减少勒索病毒对业务文件的进一步破坏，成为勒索病毒防护最后一道屏障。

**3、业务防护**：为关键业务服务器提供保护，仅允许合法应用程序对客户的关键业务文件进行操作，限制系统超级管理员权限，防止误操作或账号泄露导致的重要文件/配置的破坏。

#### 3.8.2.4 基于 eBPF 的容器安全

随着云原生技术的普及，容器安全成焦点。容器面临镜像漏洞、权限配置不当、敏感信息泄露及编排工具安全等挑战。其快速迭代和部署密度要求全周期安全防护与高效监控。构建安全稳定的容器环境，是确保云原生应用顺畅运行的关键。

容器安全从生命周期维度可分为容器运行前风险和容器运行时威胁，容器运行前风险包括镜像风险、生态风险、运行环境风险。容器运行时威胁包括容器逃逸威胁、容器入侵威胁。

**1）关键技术**

基于 eBPF 的容器安全技术聚焦于容器运行时安全检测与防御。容器与宿主机共享内核，将 eBPF 程序挂载至宿主机内核，监控所有容器进程、文件、网络行为，结合容器入侵、逃逸等恶意行为特征，识别并拦截异常操作。关键技术如下：

- **容器逃逸防护技术**

分析各类容器逃逸行为，从容器内进程操作和文件操作维度识别并阻断逃逸行为。具体而言，对进程命令行参数审计结合可疑命令库识别逃逸行为；对容器进程操作的文件路径分析，判断是否为宿主机文件，并进一步识别逃逸行为。

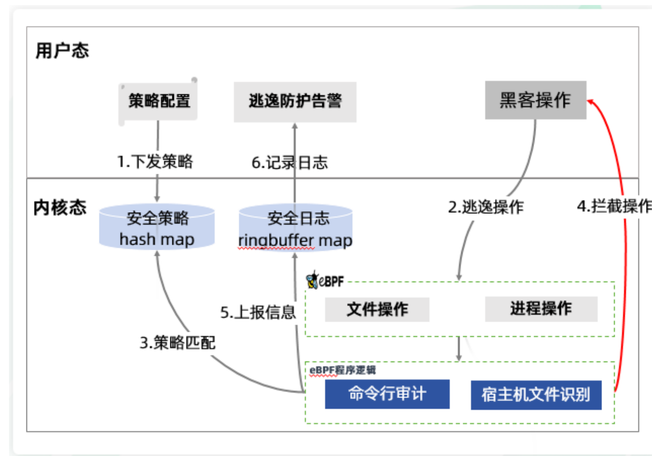


图 3-8-7 容器逃逸防护技术示例流程图

**容器内进程命令行审计：**通过 Kprobe 获取进程的命令行参数，在 LSM hook 逻辑中将命令行参数与可疑逃逸指令库比对，对匹配的进程判定为逃逸行为上报告警，并通过 eBPF LSM 机制进行细粒度拦截。

**容器内操作文件所属识别：**在文件操作的系统调用处挂载 eBPF LSM 程序，获取文件的 path 参数，结合内核数据结构目录项、虚拟文件系统挂载点等计算该文件在宿主机的实际路径，路径中包含容器信息则为容器内文件，否则识别为逃逸行为。

- **容器定位技术**

eBPF 程序均挂载在容器宿主机的内核态，部署于宿主机的容器发生入侵或逃逸行为时，内核态 eBPF 程序无法直接定位容器信息，需要内核态获取更多进程信息，配合用户态获取的容器信息定位具体容器。

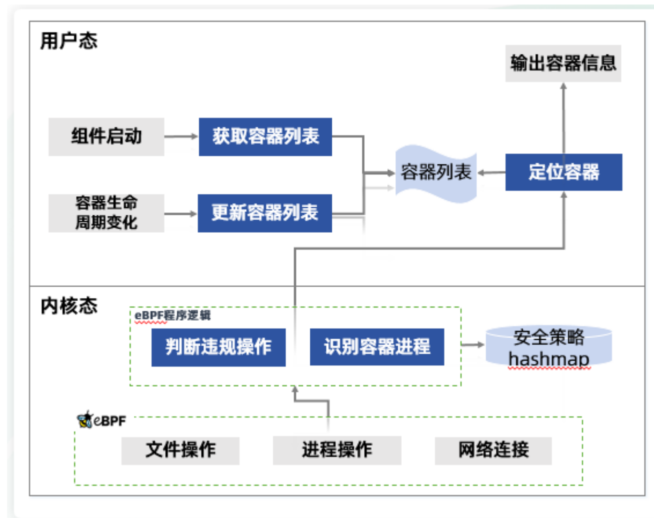


图 3-8-8 容器定位技术示例流程图

**内核态命名空间信息采集：**eBPF 程序监控内核函数，获取进程信息，通过进程 PID 命名空间和挂载命名空间，识别容器进程和宿主机进程。将该容器进程信息、PID 命名空间、挂载命名空间上传至用户态。

**用户态容器信息采集及容器定位：**获取宿主机所有运行容器列表，监控容器创建和销毁，更新容器列表。通过容器客户端获取容器信息（容器进程信息、PID 命名空间、挂载命名空间、容器名称、容器镜像、Pod 信息等）。内核态容器进程信息上传至用户态，通过 PID 命名空间、挂载命名空间定位容器。

**2）应用场景**

基于 eBPF 的容器安全技术具有以下几个方面的应用场景：

**1、容器逃逸防护：**提升容器安全能力，增强容器的隔离性，降低容器逃逸行为造成的宿主机数据的损坏。

**2、黑客入侵检测：**监控容器内进程、文件的行为，提升系统的可视性，结合入

侵检测规则，实时发现容器内的异常行为，降低容器入侵导致的资源占用、拒绝服务等攻击面。

# 4 挑战与展望

随着 eBPF 技术被广泛应用在云原生、可观测、性能调优、安全、硬件加速等领域，并且其应用场景还在快速扩展，eBPF 也不可避免地面临了一些挑战，比如：

1、恶意使用 bpfTrace 可能引发安全攻击、恶意软件、检测工程等各方面的问题；

2、eBPF 拥有 root 权限，尽管经过了验证器的安全性检测，依然无法避免漏洞存在的风险；

3、在不同 Linux 发行版、不同内核版本上构建 eBPF 程序存在兼容问题，可移植性差；

4、eBPF 程序的开发、发布、安装等相关的基础技术呈现碎片化，管理和整合存在问题，导致技术成果无法快速平移至行业客户生产环境等。

正如《The future of eBPF in the Linux Kernel》里描绘的那样，我们对 eBPF 技术的展望包括：

1、更完备的编程能力：当前 eBPF 的编程能力存在一些局限性，比如不支持变量边界的循环，指令数量受限等，未来可以提供图灵完备的编程能力。

2、更强的安全性：支持更广泛的安全特性，增强运行时 Verifier，提供媲美 Rust 的安全编程能力。

3、更广泛的移植能力：增强 CO-RE，提升 Helper 接口可移植能力，实现跨体

系、跨平台移植和兼容。

4、更强的可编程能力：广泛的支持访问/修改内核参数和返回值，实现更强的内核编程能力。

5、定制化调度：允许为特定应用或工作负载定制调度策略，实现更高效的资源分配。

6、智能化：与其他技术进行融合，如人工智能、机器学习等，以实现更智能的系统监控和优化。

能够继续在云原生、可观测、性能调优、安全、硬件加速、AI 等领域实现技术创新与应用场景扩展，提供更多的工具和方案，以应对不断增长的各种挑战和需求。

后续，我们会继续开展 eBPF 应用的研究与实践，在安全方向，实现并强化容器入侵检测和防御、网络防护、安全漏洞应急防御等；在容器网络可观测性方向，支持应用零侵入、动态、高效的网络流量信息采集、汇总及分析，实现容器间网络拓扑绘制、时延及错误响应等网络指标监控告警。

## 加入我们

关注浪潮信息操作系统公众号：扫描下方二维码。

关注龙蜥社区公众号：扫描下方二维码。

加入钉钉群：扫描下方钉钉群二维码。

欢迎开发者/用户加入“eBPF 技术探索 SIG ”交流，一起打造一个活跃的、健康的开源操作系统生态！

**微信公众号**

扫码关注微信公众号
“浪潮信息操作系统”

**微信公众号**

扫码关注微信公众号
“OpenAnolis龙蜥”

**eBPF技术探索SIG交流群**

钉钉扫码入群
交流eBPF热门技术

分享/交流 eBPF 技术：关注 ” 酷玩BPF “ 微信公众号：

**白皮书贡献路径**

https://github.com/aliyun/coolbpf/tree/white-book/docs/white-book