

**编写说明**

**编写单位**：浪潮电子信息产业股份有限公司、阿里云计算有限公司

**参编组织**：龙蜥社区

**编写组成员**：苏志远、吴栋、方浩、王传国、梁媛、黄吉旺、毛文安、程书意、甄鹏、李宏伟、彭彬彬

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 6](#_Toc148196219)

[2 eBPF技术介绍 7](#_Toc148196220)

[2.1 eBPF架构 7](#_Toc148196221)

[2.1.1 eBPF加载过程 8](#_Toc148196222)

[2.1.2 JIT编译 9](#_Toc148196223)

[2.1.3 挂载与执行 10](#_Toc148196224)

[2.2 eBPF常见的开发框架 17](#_Toc148196225)

[2.2.1 BCC 17](#_Toc148196226)

[2.2.2 bpfTrace 18](#_Toc148196227)

[2.2.3 libbpf 18](#_Toc148196228)

[2.2.4 libbpf-bootstrap 19](#_Toc148196229)

[2.2.5 cilium-ebpf 19](#_Toc148196230)

[2.2.6 Coolbpf 19](#_Toc148196231)

[3 eBPF的应用场景与实践 23](#_Toc148196232)

[3.1 基于eBPF的系统诊断 23](#_Toc148196233)

[3.1.1 系统诊断面临挑战 23](#_Toc148196234)

[3.1.2 基于eBPF的系统诊断方案 28](#_Toc148196235)

[3.2 基于eBPF的虚拟化IO全链路时延监测 34](#_Toc148196236)

[3.2.1 虚拟化IO全路径分析主要面临的挑战 34](#_Toc148196237)

[3.2.2 基于bpftrace虚拟化IO路径追踪解决方案 36](#_Toc148196238)

[3.3 基于eBPF的网络性能优化 41](#_Toc148196239)

[3.3.1 Linux网络性能优化面临的技术挑战 41](#_Toc148196240)

[3.3.2 基于eBPF的Linux内核网络性能优化解决方案 44](#_Toc148196241)

[3.4 基于eBPF的主机安全 53](#_Toc148196242)

[3.4.1 传统解决方案面临挑战 53](#_Toc148196243)

[3.4.2 基于eBPF的新一代主机安全解决方案 54](#_Toc148196244)

[4 挑战与展望 63](#_Toc148196245)

# 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。

# eBPF技术介绍

## 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；

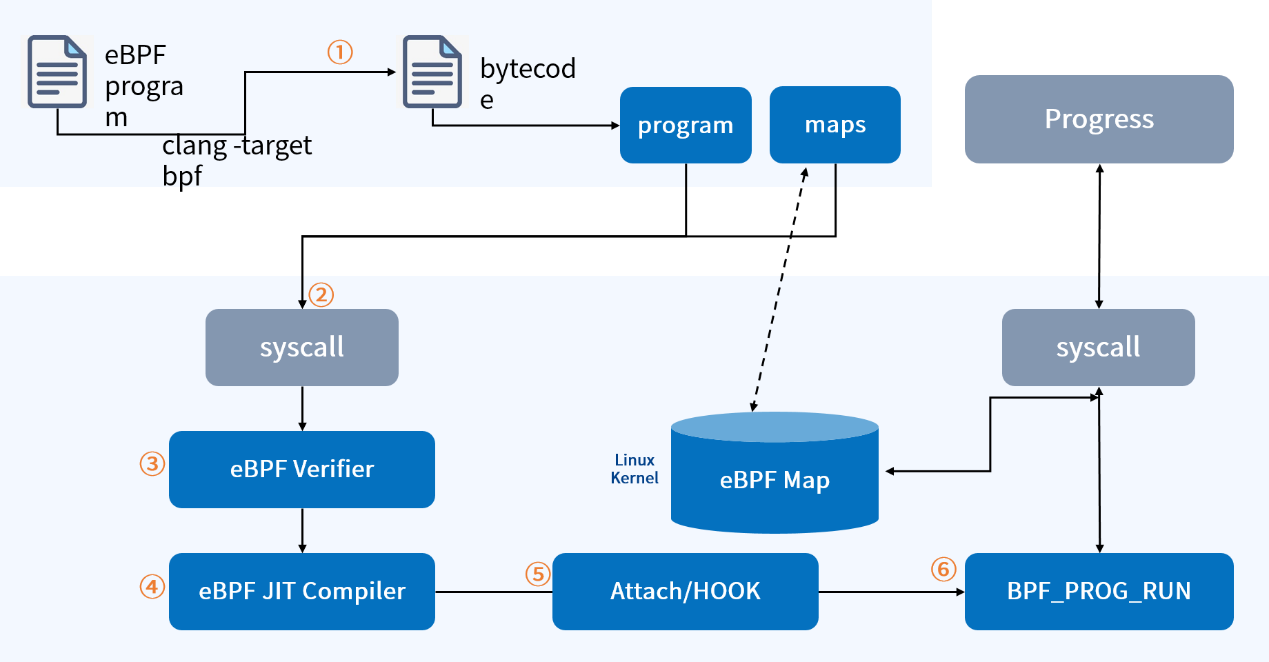


图2-1-1 eBPF基本架构与使用示意图

### 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。

### 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：

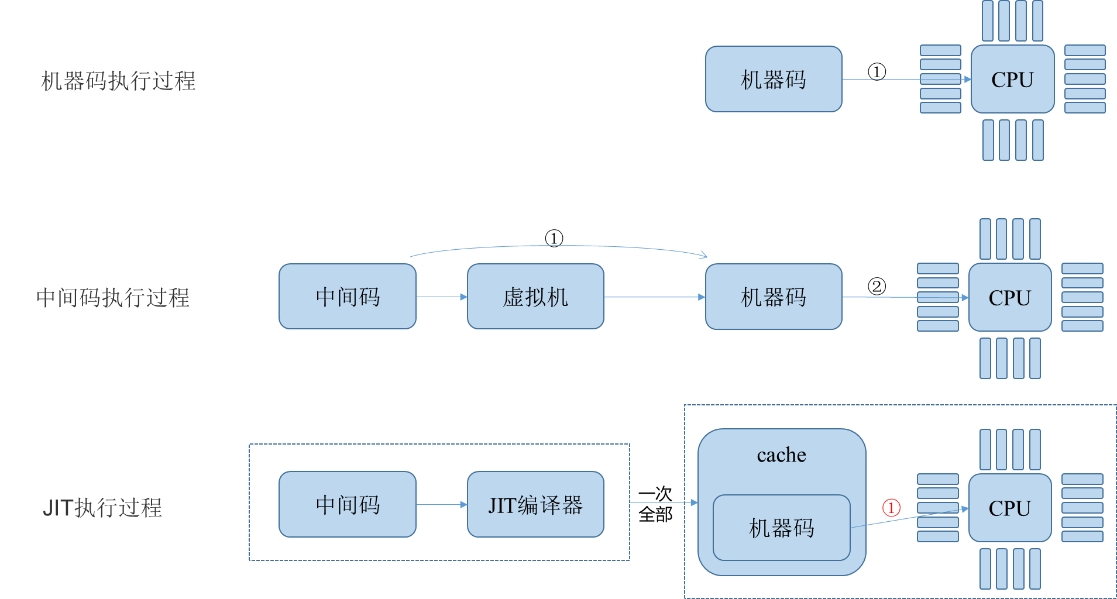


图2-1-2 JIT的作用示意图

### 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。

#### 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 | xdp, sock\_ops, sk\_msg, sk\_skb, 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。

## eBPF常见的开发框架

### 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。

### 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。

### 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。

### 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。

### 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

### 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。

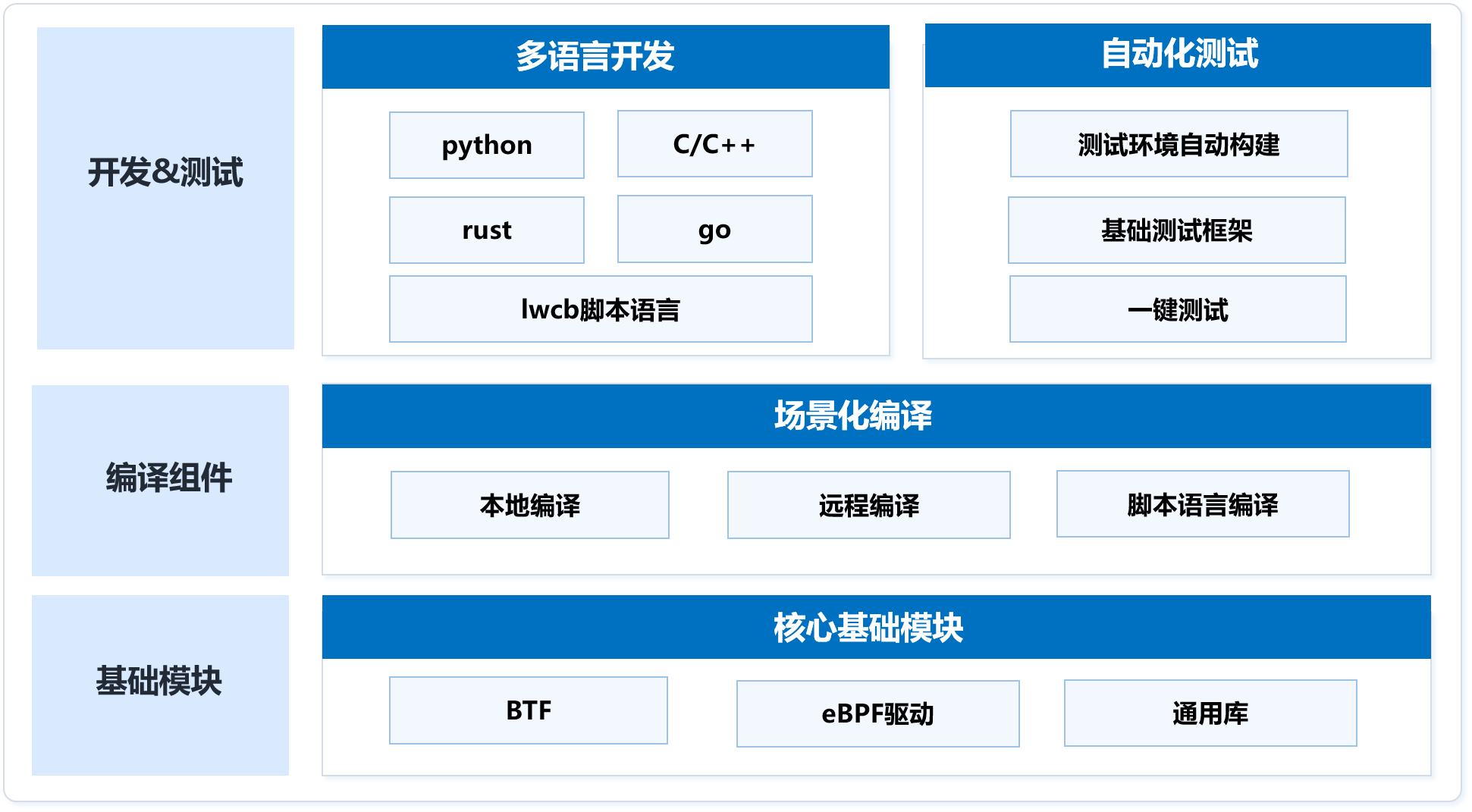


图2-2-1 Coolbpf功能架构图

**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。

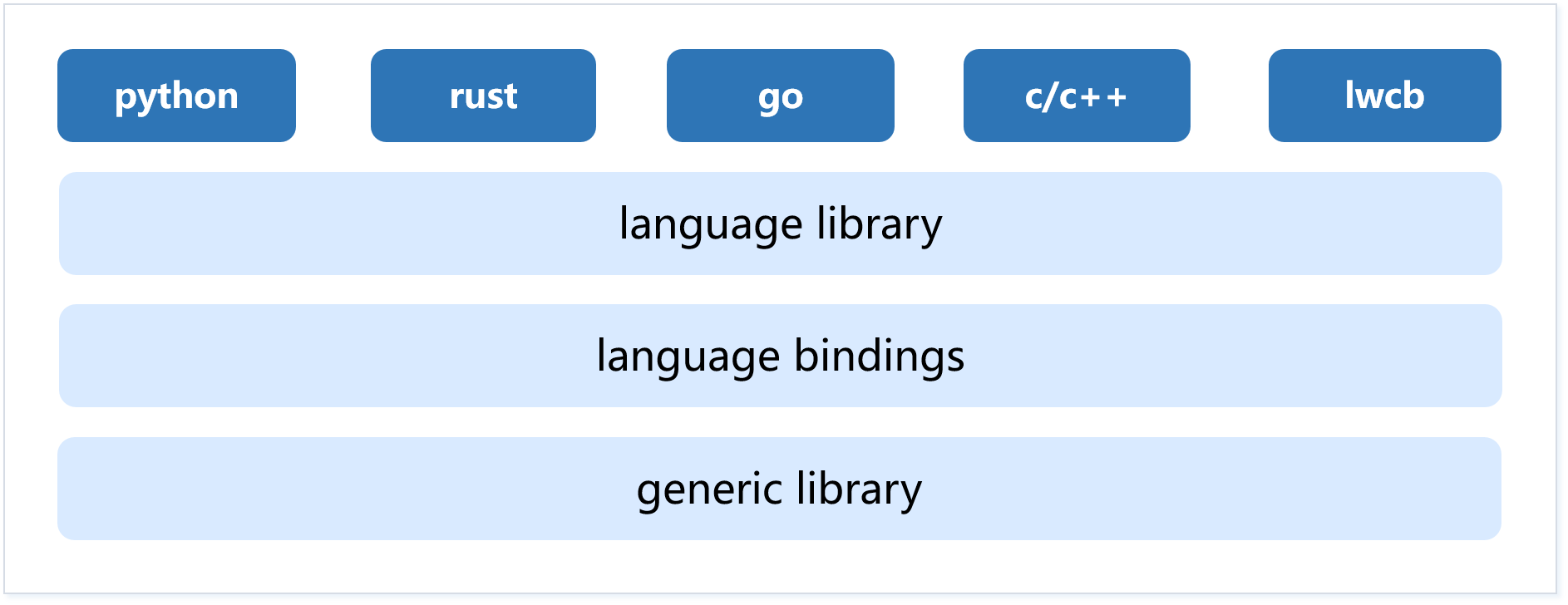


图2-2-2 Coolbpf开发模块层级结构图

**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。

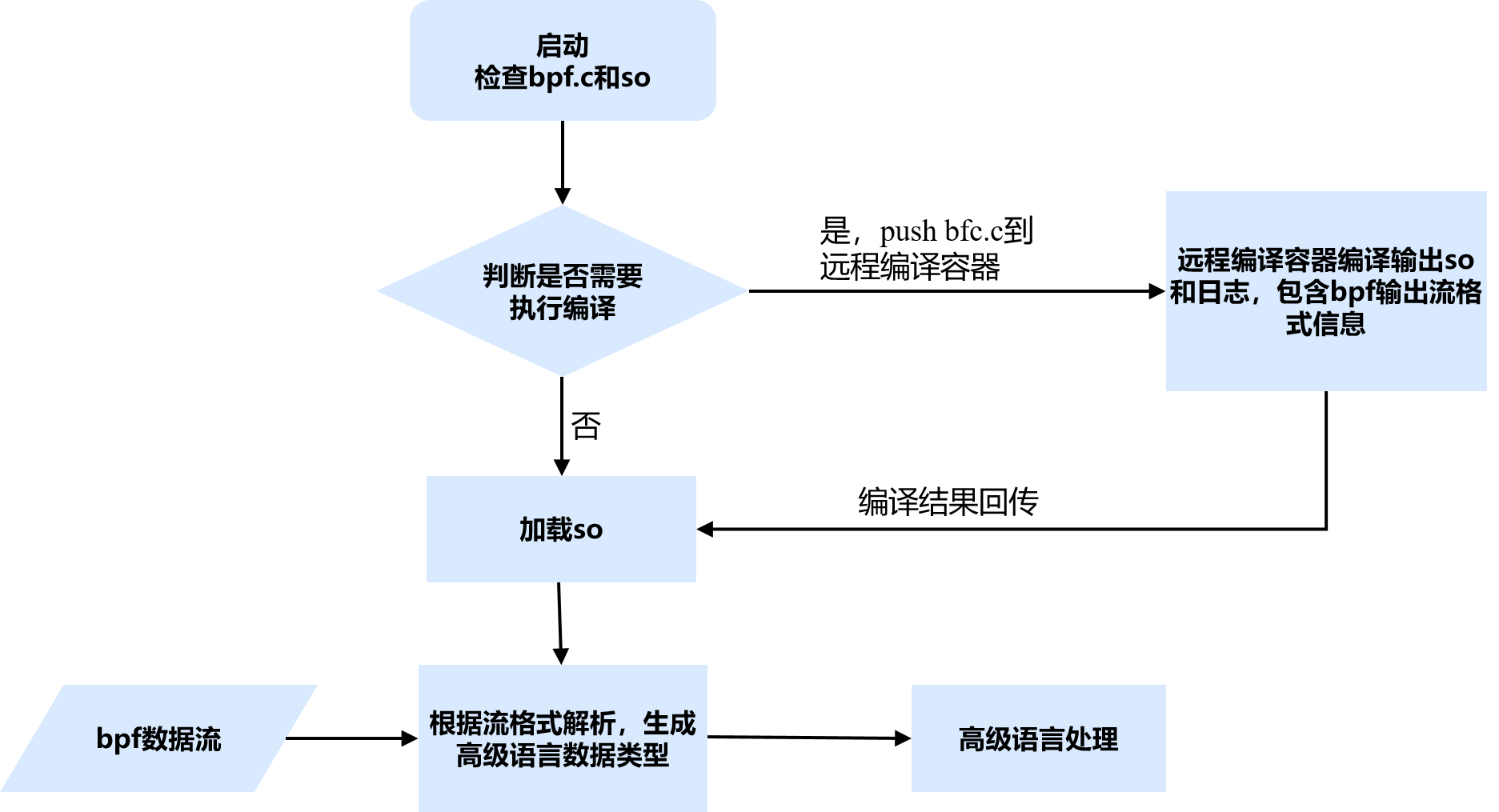


图2-2-3 Coolbpf编译模块逻辑图

**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。

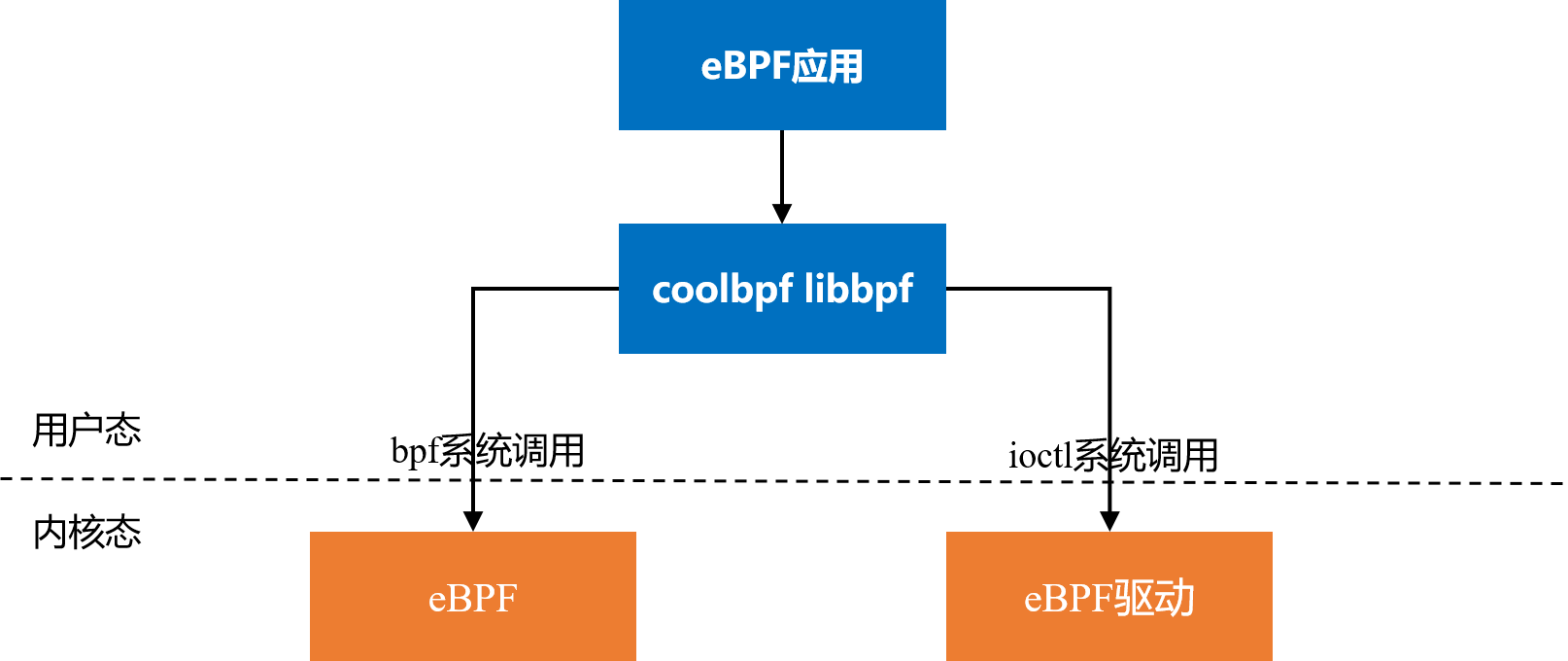


图2-2-4 Coolbpf基础模块结构图

# eBPF的应用场景与实践

eBPF具有高性能、高扩展、安全性等优势，目前已经在[网络](https://m.elecfans.com/v/tag/1722/)、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。

## 基于eBPF的系统诊断

### 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。

### 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。

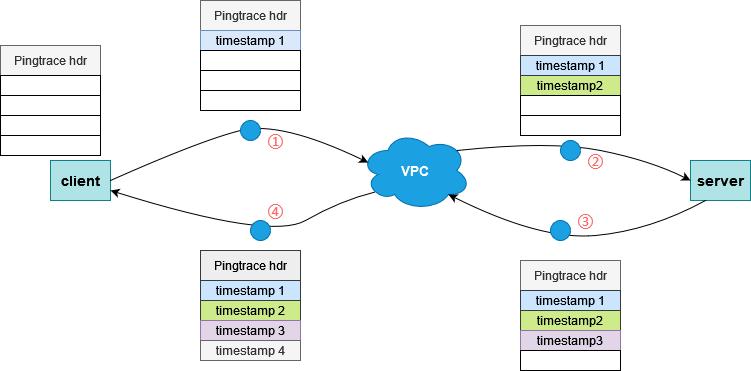


图3-1-1 PingTrace原理图

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：



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。

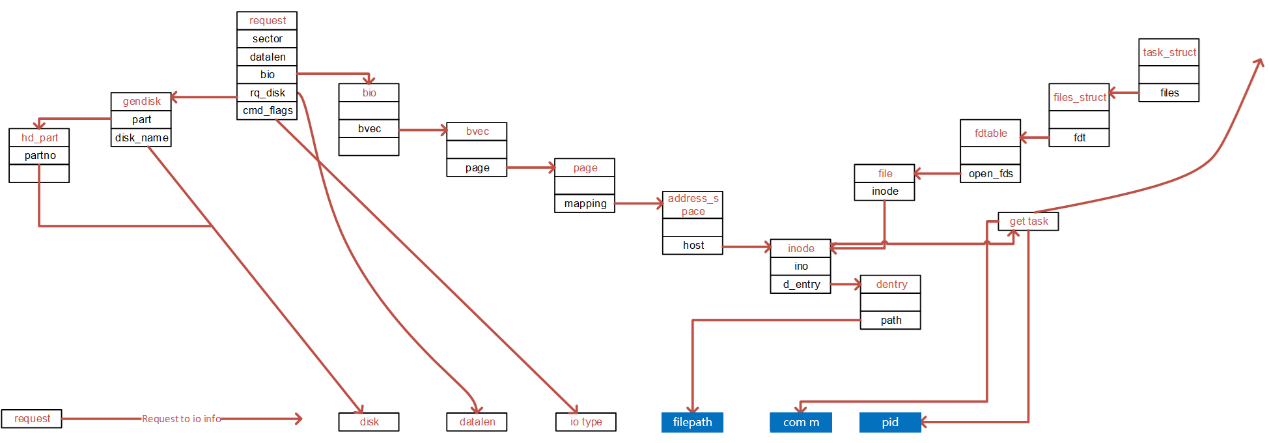


图3-1-2 iofsstat工作原理图

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：

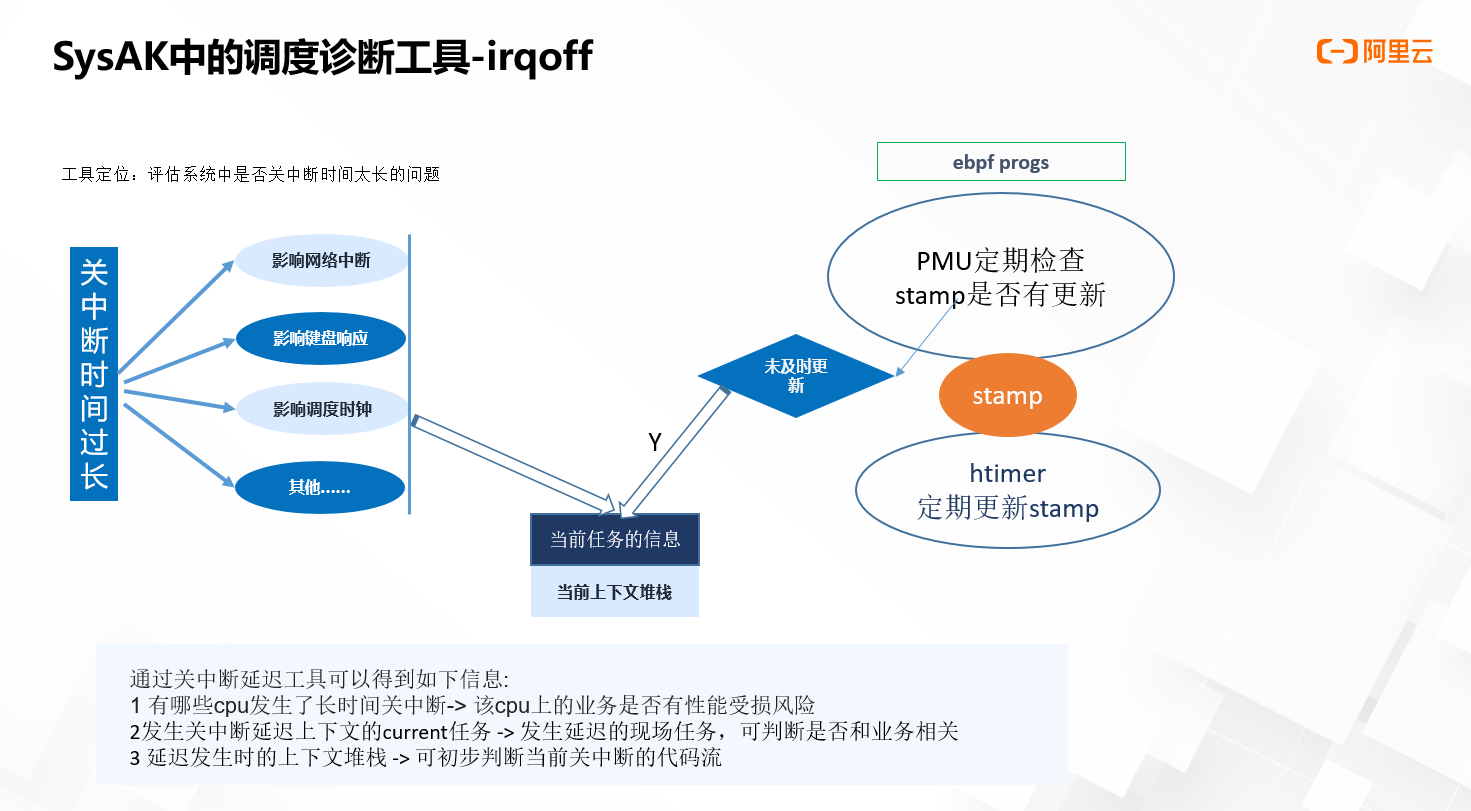


图3-1-3 irqoff工具的原理图

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。

## 基于eBPF的虚拟化IO全链路时延监测

### 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。

### 基于bpftrace虚拟化IO路径追踪解决方案

#### 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。

#### 关键技术

##### 3.2.2.2.1 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：

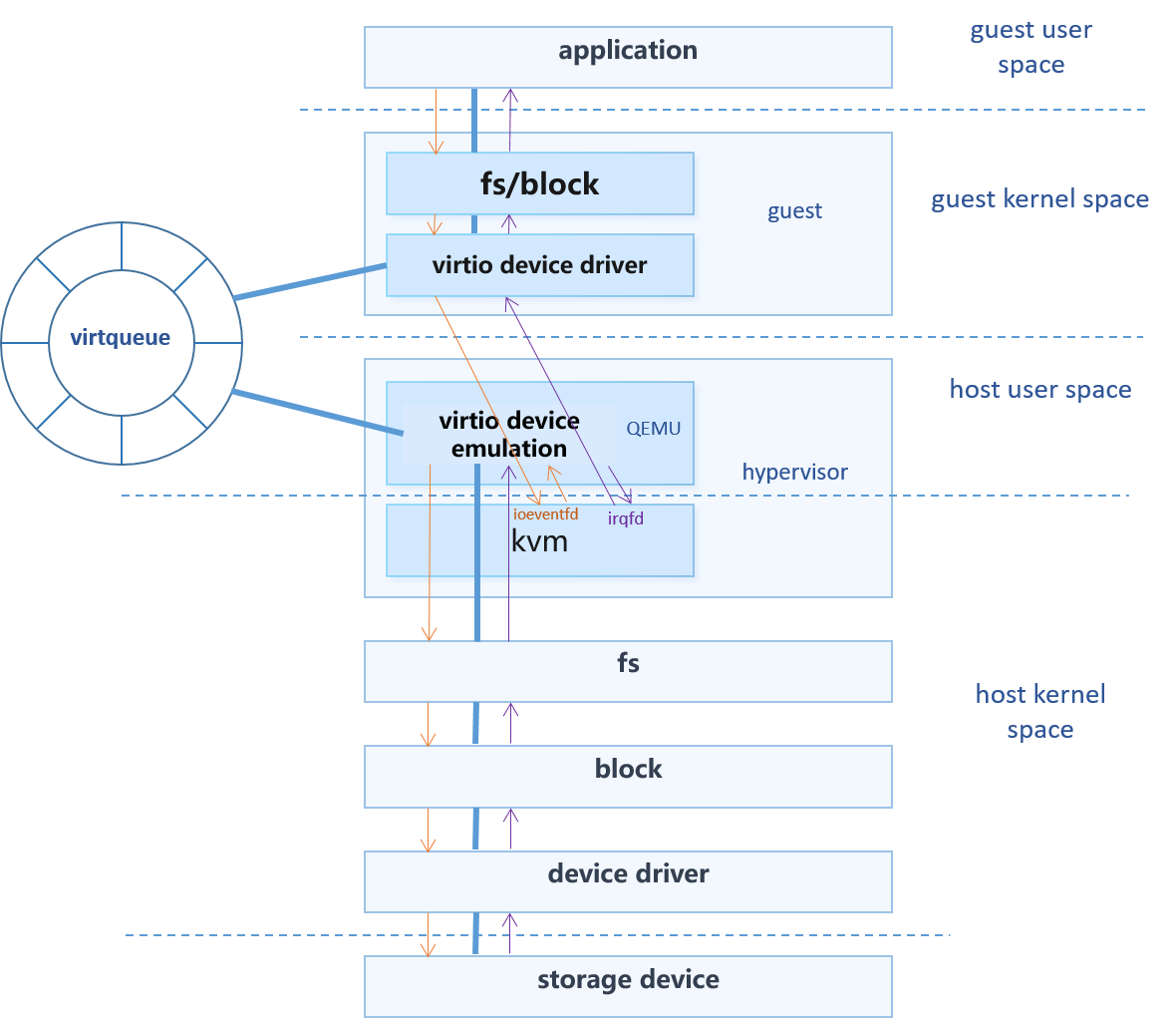


图3-2-1 基于QEMU-KVM的虚拟化IO路径全视图

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

##### 3.2.2.2.2 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。

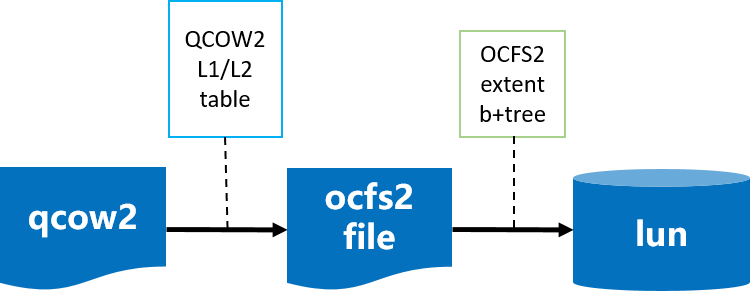


图3-2-2 层间映射图

##### 3.2.2.2.3 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；

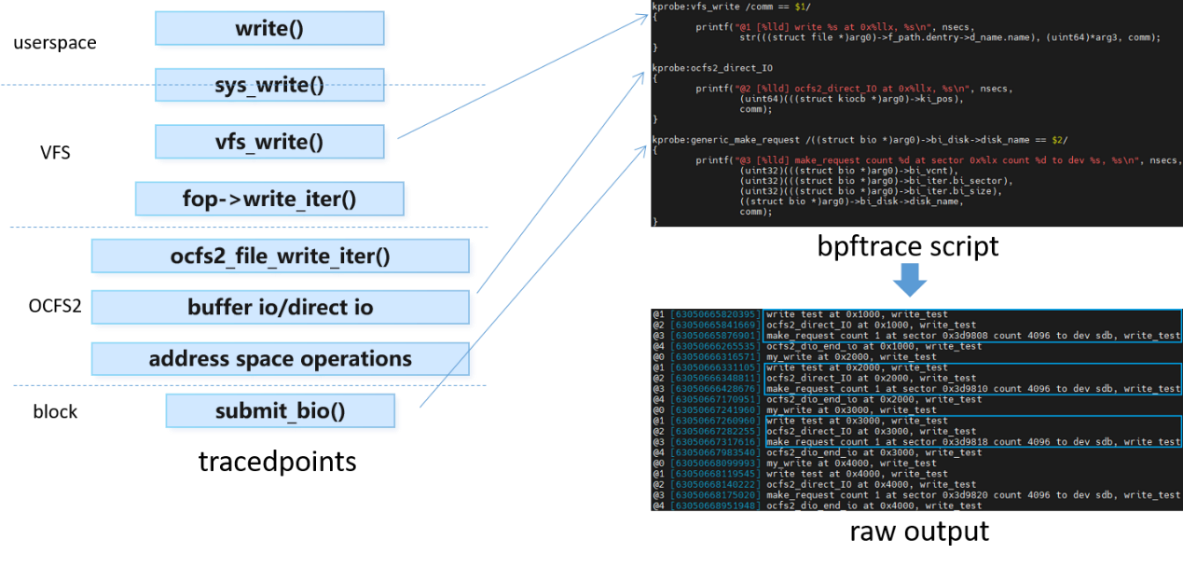


图3-2-3 IO性能追踪工具record部分工作过程示意图

**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。

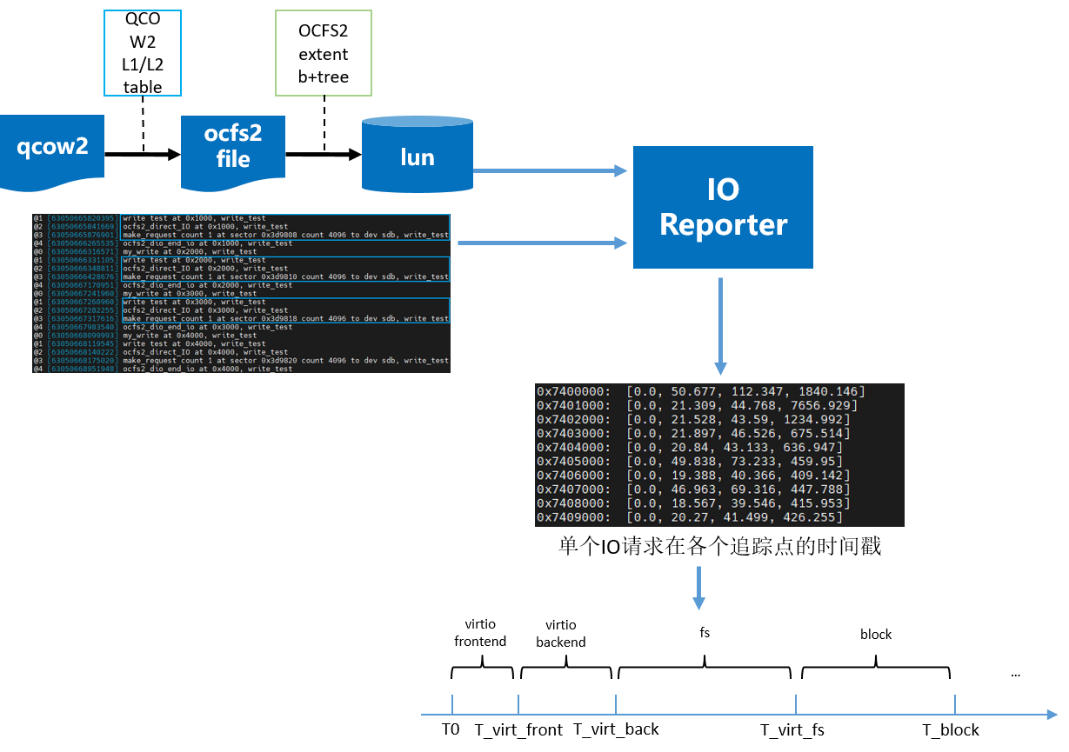


图3-2-4 IO性能追踪工具reporter部分工作过程示意图

##### 3.2.2.2.4 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

1. 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。

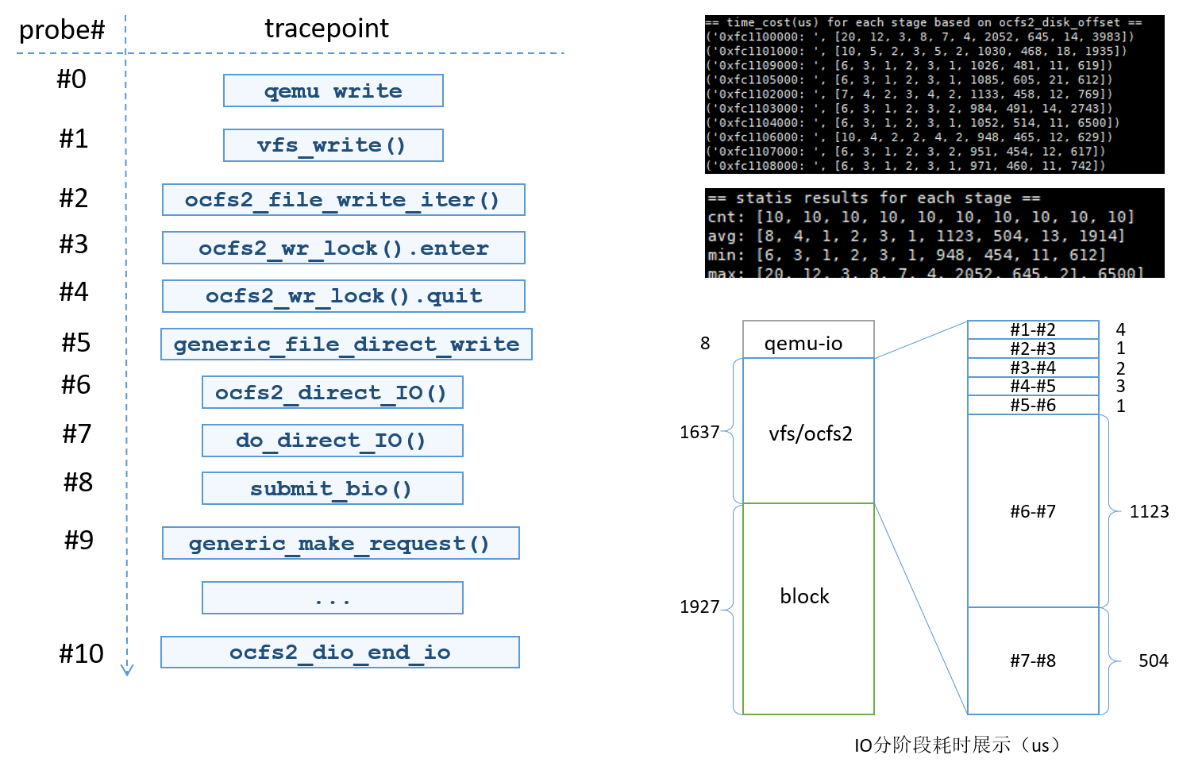


图3-2-5 IO性能追踪工具演示图

## 基于eBPF的网络性能优化

### Linux网络性能优化面临的技术挑战

#### 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。

#### 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。

##### 3.3.1.2.1 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。

##### 3.3.1.2.2 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。

### 基于eBPF的Linux内核网络性能优化解决方案

#### 整体架构

##### 3.3.2.1.1 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。

##### 3.3.2.1.2 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。

###### 3.3.2.1.2.1 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3-3-1 网络包发送流程示意图

###### 3.3.2.1.2.2 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。

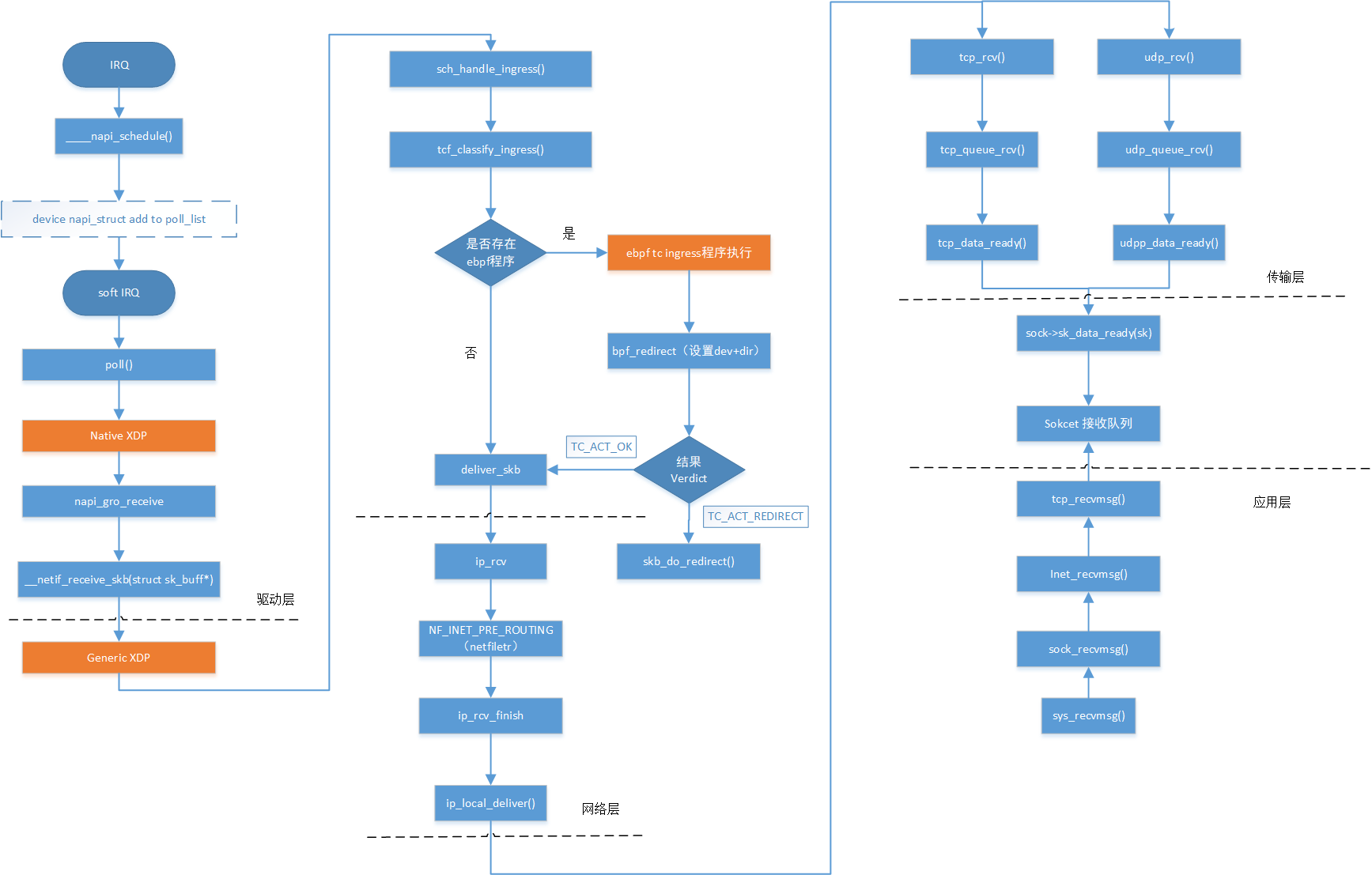


图3-3-2 网络包收包流程示意图

#### 场景实践

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

##### 3.3.2.2.1 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：

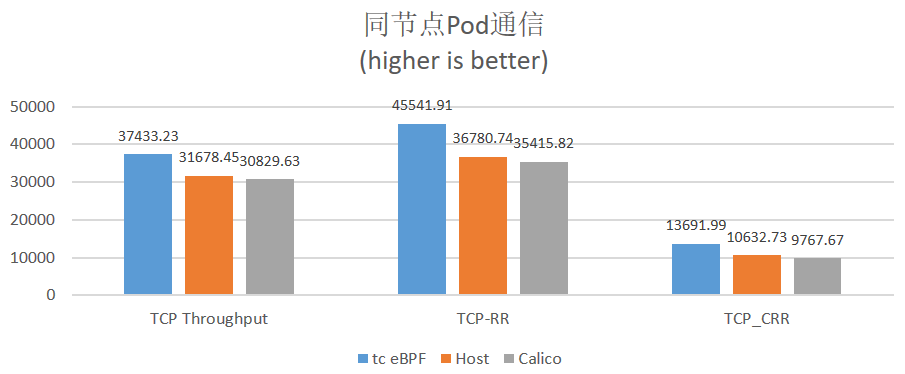


图3-3-3 基于TC eBPF加速同节点Pod通信的效果展示图

##### 3.3.2.2.2 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：

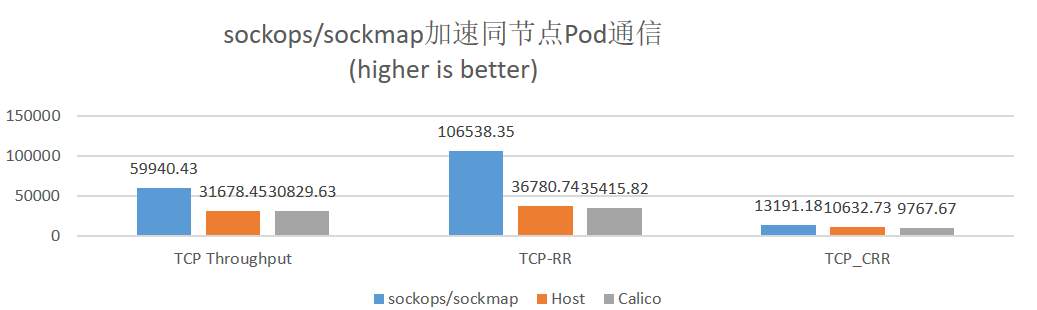


图3-3-4 基于sockops/sockmap加速同节点Pod通信的效果展示图

##### 3.3.2.2.3 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：

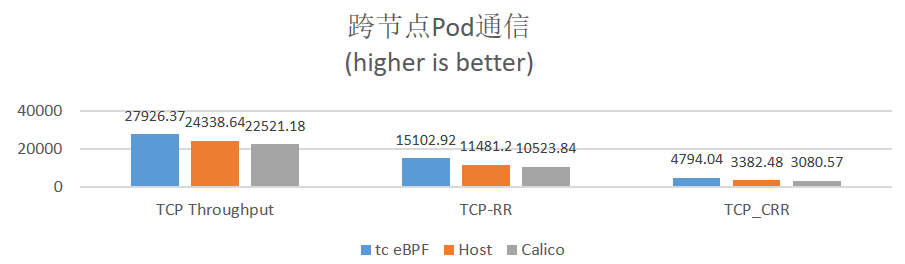


图3-3-5 基于TC eBPF加速跨节点Pod通信的效果展示图

##### 3.3.2.2.4 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%。

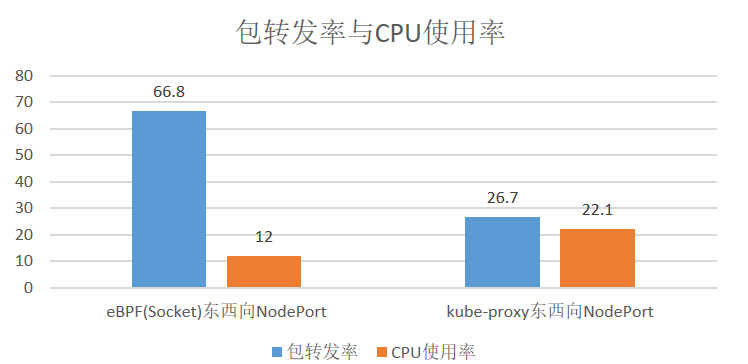


图3-3-6 基于Socket拦截加速东西向clusterIP访问的效果展示图

##### 3.3.2.2.5 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%。

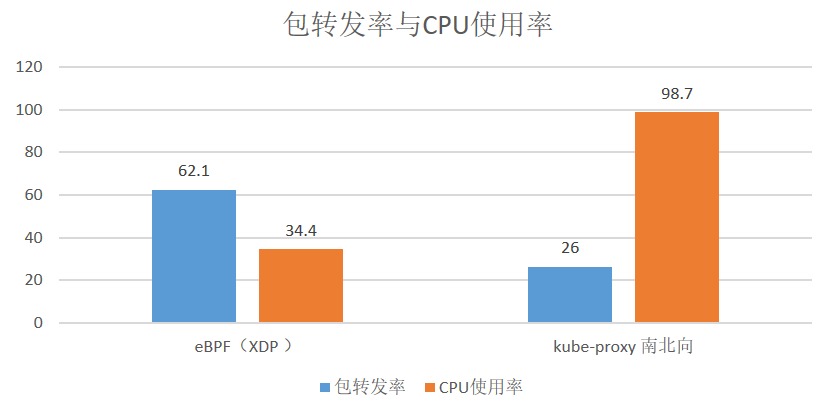


图3-3-7基于XDP加速南北向本地NodePort访问的效果展示图

##### 3.3.2.2.6 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%。

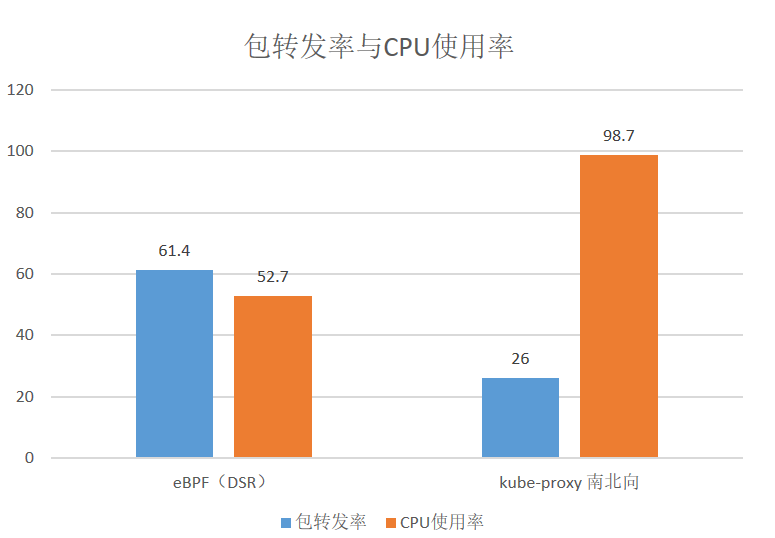


图3-3-8基于DSR加速南北向远端NodePort访问的效果展示图

##### 3.3.2.2.7 基于TC eBPF针对指定端口定向到指定网卡的流量加速转发

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：

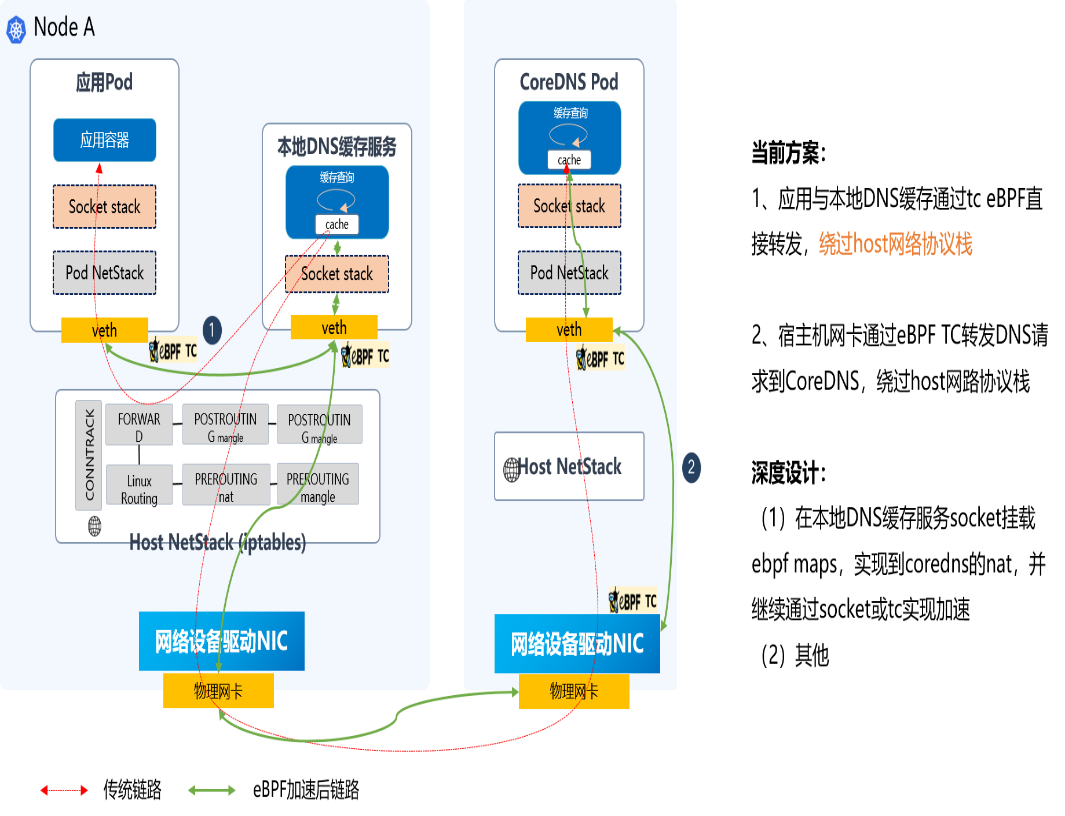


图3-3-9 基于TC eBPF的流量转发路径示意图

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。

## 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。

### 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3-4-1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 |

### 基于eBPF的新一代主机安全解决方案

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。

#### 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、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；

**4、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3-4-1 KSecure安全组件功能架构图

#### 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。

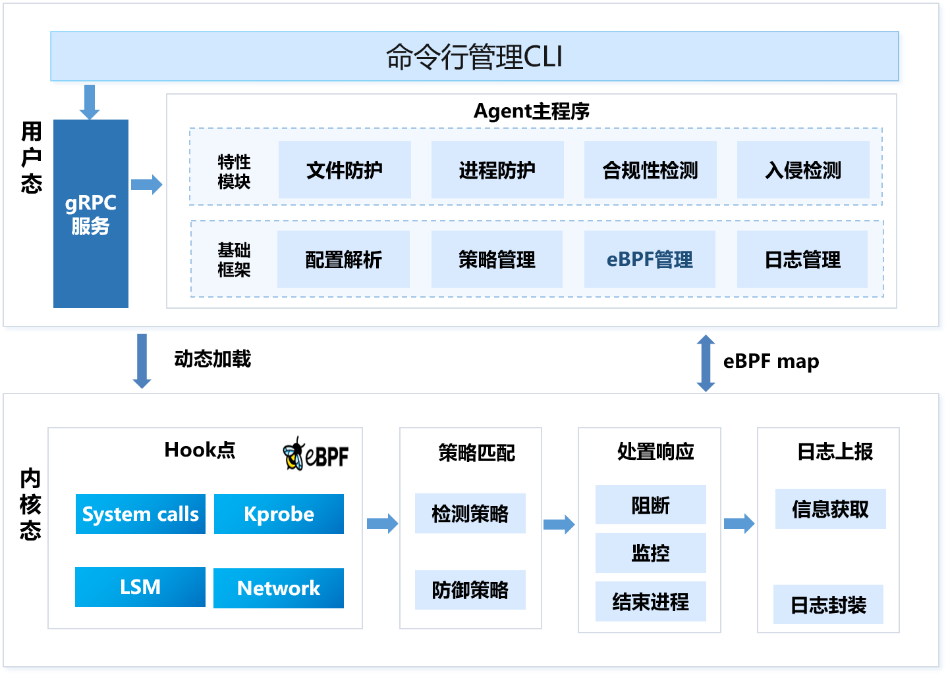


图3-4-2 KSecure安全组件技术架构图

#### 关键技术

在KSecure安全组件中，涉及如下两个关键技术。

##### 3.4.2.3.1 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。

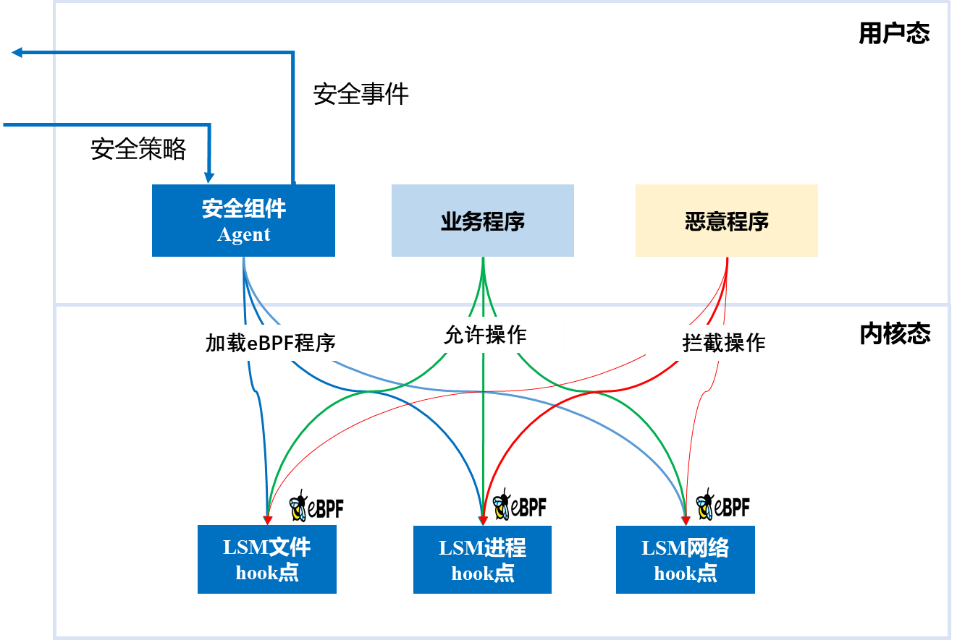


图3-4-3 关键业务防护架构图

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：

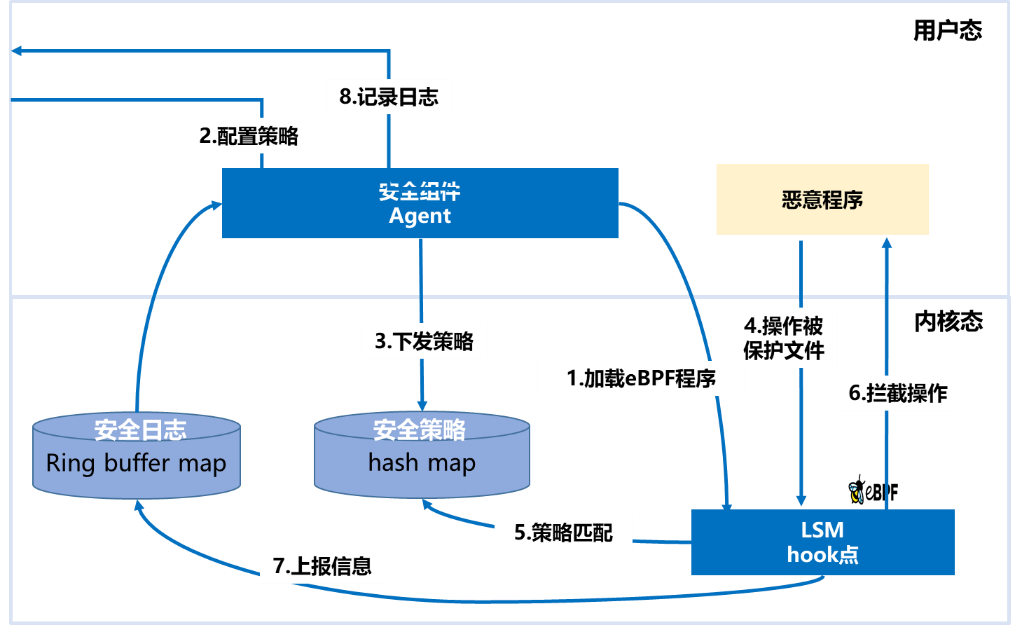


图3-4-4文件防护示例流程图

**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3-4-2 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。

##### 3.4.2.3.2 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。

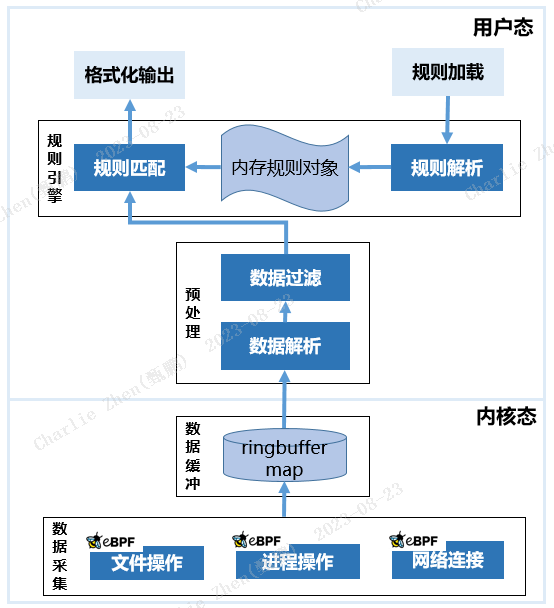


图3-4-5 入侵检测功能模块架构图

**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3-4-6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3-4-3 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.。

#### 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。

# 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。

能够继续在云原生、可观测、性能调优、安全、硬件加速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。