## libbpf 概述

libbpf 是一个基于 C 语言的库，它包含一个 BPF 加载器，用于接收编译好的 BPF 对象文件并将其准备和加载到 Linux 内核中。libbpf 承担了加载、验证和将 BPF 程序附加到各种内核 hook 的繁重工作，使 BPF 应用程序开发人员能够只关注 BPF 程序的正确性和性能。

libbpf 支持以下高级功能：

* 为用户空间程序提供高级和低级 API，以与 BPF 程序进行交互。低级 API 封装了所有的 bpf 系统调用功能，这在用户需要对用户空间和 BPF 程序之间的交互进行更细粒度的控制时非常有用。
* 为 bpftool 生成的 BPF 对象 skeleton 提供全面的支持。skeleton 文件简化了用户空间程序访问全局变量和使用 BPF 程序的过程。
* 提供 BPF 侧 API，包括 BPF 辅助函数定义、BPF 映射支持和跟踪辅助函数，允许开发人员简化 BPF 代码的编写。
* 支持 BPF **CO-RE** 机制，使 BPF 开发人员能够编写可移植的 BPF 程序，这些程序可以编译一次并在不同的内核版本上运行。

本文档将详细阐述上述概念，从而更深入地理解 libbpf 的功能和优势，以及它如何帮助您高效地开发 BPF 应用程序。

## BPF 应用程序生命周期和 libbpf API

一个 BPF 应用程序由一个或多个 BPF 程序（协作或完全独立）、BPF 映射和全局变量组成。全局变量在所有 BPF 程序之间共享，这使它们能够在一组公共数据上进行协作。libbpf 提供了用户空间程序可用于通过触发 BPF 应用程序生命周期的不同阶段来操作 BPF 程序的 API。

以下部分简要概述了 BPF 生命周期中的每个阶段：

* **Open phase（打开阶段）**：在此阶段，libbpf 解析 BPF 对象文件并发现 BPF 映射、BPF 程序和全局变量。打开 BPF 应用程序后，用户空间应用程序可以在创建和加载所有实体之前进行额外的调整（必要时设置 BPF 程序类型；预设全局变量的初始值等）。
* **Load phase（加载阶段）**：在加载阶段，libbpf 创建 BPF 映射，解析各种重定位，并验证 BPF 程序并将其加载到内核中。此时，libbpf 验证 BPF 应用程序的所有部分并将 BPF 程序加载到内核中，但尚未执行任何 BPF 程序。加载阶段之后，可以设置初始 BPF 映射状态而不会与 BPF 程序代码执行发生竞争。
* **Attachment phase（附加阶段）**：在此阶段，libbpf 将 BPF 程序附加到各种 BPF hook 点（例如，tracepoints、kprobes、cgroup hooks、网络数据包处理管道等）。在此阶段，BPF 程序执行有用的工作，例如处理数据包，或更新可从用户空间读取的 BPF 映射和全局变量。
* **Tear down phase（拆卸阶段）**：在拆卸阶段，libbpf 分离 BPF 程序并将其从内核中卸载。BPF 映射被销毁，并且 BPF 应用程序使用的所有资源都被释放。

## BPF 对象 Skeleton 文件

BPF skeleton 是 libbpf API 的替代接口，用于处理 BPF 对象。Skeleton 代码抽象了通用的 libbpf API，以显著简化用户空间操作 BPF 程序的代码。Skeleton 代码包含 BPF 对象文件的字节码表示，简化了分发 BPF 代码的过程。通过嵌入 BPF 字节码，您的应用程序二进制文件不再需要额外的部署文件。

您可以通过将 BPF 对象传递给 bpftool 来生成特定对象文件的 skeleton 头文件（.skel.h）。生成的 BPF skeleton 提供了以下与 BPF 生命周期对应的自定义函数，每个函数都以特定对象名称为前缀：

* <name>\_\_open() – 创建并打开 BPF 应用程序（<name> 代表特定的 bpf 对象名称）
* <name>\_\_load() – 实例化、加载和验证 BPF 应用程序部件
* <name>\_\_attach() – 附加所有可自动附加的 BPF 程序（它是可选的，您可以通过直接使用 libbpf API 进行更精细的控制）
* <name>\_\_destroy() – 分离所有 BPF 程序并释放所有使用的资源

使用 skeleton 代码是使用 bpf 程序的推荐方式。请记住，BPF skeleton 提供了对底层 BPF 对象的访问，因此即使使用 BPF skeleton，使用通用 libbpf API 可以做的事情仍然是可能的。它是一个附加的便利功能，没有系统调用，也没有繁琐的代码。

### 使用 Skeleton 文件的其他优点

BPF skeleton 为用户空间程序提供了与 BPF 全局变量交互的接口。skeleton 代码将全局变量作为结构体内存映射到用户空间。结构体接口允许用户空间程序在 BPF 加载阶段之前初始化 BPF 程序，并在之后从用户空间获取和更新数据。

skel.h 文件通过列出可用的映射、程序等来反映对象文件结构。BPF skeleton 提供对所有 BPF 映射和 BPF 程序的直接访问作为结构体字段。这消除了使用 bpf\_object\_find\_map\_by\_name() 和 bpf\_object\_find\_program\_by\_name() API 进行基于字符串查找的需要，从而减少了由于 BPF 源代码和用户空间代码不同步而导致的错误。

对象文件的嵌入式字节码表示确保了 skeleton 和 BPF 对象文件始终同步。

## BPF 辅助函数

libbpf 提供 BPF 侧 API，BPF 程序可以使用它们与系统交互。BPF 辅助函数定义允许开发人员像任何其他普通 C 函数一样在 BPF 代码中使用它们。例如，有辅助函数用于打印调试消息、获取系统启动以来的时间、与 BPF 映射交互、操作网络数据包等。

有关辅助函数功能的完整说明、它们接受的参数和返回值，请参阅 bpf-helpers 手册页。

## BPF CO-RE (Compile Once – Run Everywhere) BPF CO-RE（一次编译 – 随处运行）

BPF 程序在内核空间工作，并可以访问内核内存和数据结构。BPF 应用程序遇到的一个限制是缺乏跨不同内核版本和配置的可移植性。BCC 是 BPF 可移植性的解决方案之一。但是，它带来了运行时开销和由于将编译器嵌入到应用程序中而导致的大二进制文件大小。

libbpf 通过支持 BPF CO-RE 概念来提升 BPF 程序的便携性。BPF CO-RE 将 BTF 类型信息、libbpf 和编译器结合起来，生成一个可执行二进制文件，您可以在多个内核版本和配置上运行它。

为了使 BPF 程序可移植，libbpf 依赖于运行中内核的 BTF 类型信息。内核还通过 sysfs 在 /sys/kernel/btf/vmlinux 暴露了这种自描述的权威 BTF 信息。

您可以使用以下命令为运行中的内核生成 BTF 信息：

Bash  重击

$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

该命令生成一个 vmlinux.h 头文件，其中包含运行中内核使用的所有内核类型 (BTF 类型)。在您的 BPF 程序中包含 vmlinux.h 消除了对系统范围内核头的依赖。

libbpf 通过查看 BPF 程序的记录 BTF 类型和重定位信息，并将其与运行中内核提供的 BTF 信息 (vmlinux) 匹配，从而实现 BPF 程序的可移植性。libbpf 然后解析并匹配所有类型和字段，并更新必要的偏移量和其他可重定位数据，以确保 BPF 程序的逻辑在主机上的特定内核上正确运行。BPF CO-RE 概念从而消除了与 BPF 开发相关的开销，并允许开发人员编写可移植的 BPF 应用程序，而无需在目标机器上进行修改和运行时源代码编译。

以下代码片段展示了如何使用 BPF CO-RE 和 libbpf 读取内核 task\_struct 的 parent 字段。以 CO-RE 可重定位方式读取字段的基本辅助函数是 bpf\_core\_read(dst, sz, src)，它将从 src 引用的字段中读取 sz 字节到 dst 指向的内存中。

C

//...

struct task\_struct \*task = (void \*)bpf\_get\_current\_task();

struct task\_struct \*parent\_task;

int err;

err = bpf\_core\_read(&parent\_task, sizeof(void \*), &task->parent);

if (err) {

/\* handle error \*/

}

/\* parent\_task contains the value of task->parent pointer \*/

在代码片段中，我们首先使用 bpf\_get\_current\_task() 获取指向当前 task\_struct 的指针。然后我们使用 bpf\_core\_read() 将 task 结构体的 parent 字段读取到 parent\_task 变量中。bpf\_core\_read() 就像 bpf\_probe\_read\_kernel() BPF 辅助函数一样，不同之处在于它记录了应该在目标内核上重定位的字段信息。也就是说，如果 parent 字段由于在其前面添加了一些新字段而导致 struct task\_struct 中的偏移量发生变化，libbpf 将自动将实际偏移量调整为正确的值。

## libbpf 入门

请查看 [libbpf-bootstrap](https://github.com/libbpf/libbpf-bootstrap" \t "_blank) 仓库，其中包含使用 libbpf 构建各种 BPF 应用程序的简单示例。

另请参阅 [libbpf API 文档](https://libbpf.readthedocs.io/en/latest/api.html" \t "_blank)。

## libbpf 和 Rust

如果您正在使用 Rust 构建 BPF 应用程序，建议使用 [Libbpf-rs](https://github.com/libbpf/libbpf-rs" \t "_blank) 库而不是直接使用 bindgen 绑定到 libbpf。Libbpf-rs 以 Rust 惯用接口封装了 libbpf 功能，并提供了 libbpf-cargo 插件来处理 BPF 代码编译和 skeleton 生成。使用 Libbpf-rs 将使构建 BPF 应用程序的用户空间部分更容易。请注意，BPF 程序本身仍必须用纯 C 语言编写。

## libbpf 日志记录

默认情况下，libbpf 将信息和警告消息记录到 stderr。这些消息的详细程度可以通过将环境变量 LIBBPF\_LOG\_LEVEL 设置为 warn、info 或 debug 来控制。可以使用 libbpf\_set\_print() 设置自定义日志回调。