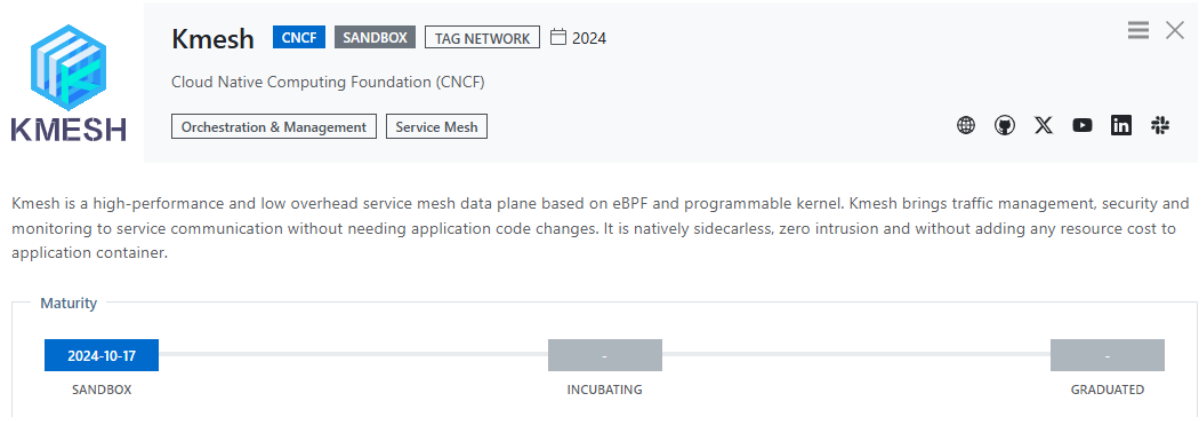
开源项目源码解析——Kmesh

1. 引言

作为一名刚刚了解开源社区的小白，我感受到社区的活跃氛围和丰富的issue讨论，这让我对参与其中充满期待。Kmesh项目隶属于华为云，得益于它的资源和支持，我们在开源社区交流时显得尤为便利。此外，我的实验室师兄也在积极参与Kmesh项目，这种环境的影响促使我选择Kmesh这个开源项目来进行介绍。

我对网络相关的开源项目有着浓厚的兴趣，而Kmesh通过优化BPF和eBPF，显著提升了内核态网络处理性能。它不仅提供了多种协议解析、伪链构建和QoS控制等功能，还采用双引擎架构，实现了用户态与内核态之间的性能与灵活性的完美平衡。同时，Kmesh支持Kubernetes CNI插件，提供高度定制化的集群网络解决方案，为构建高性能网络基础设施奠定了坚实的基础。借助强大的社区支持和创新平台，Kmesh展现了广阔的发展潜力和前景。



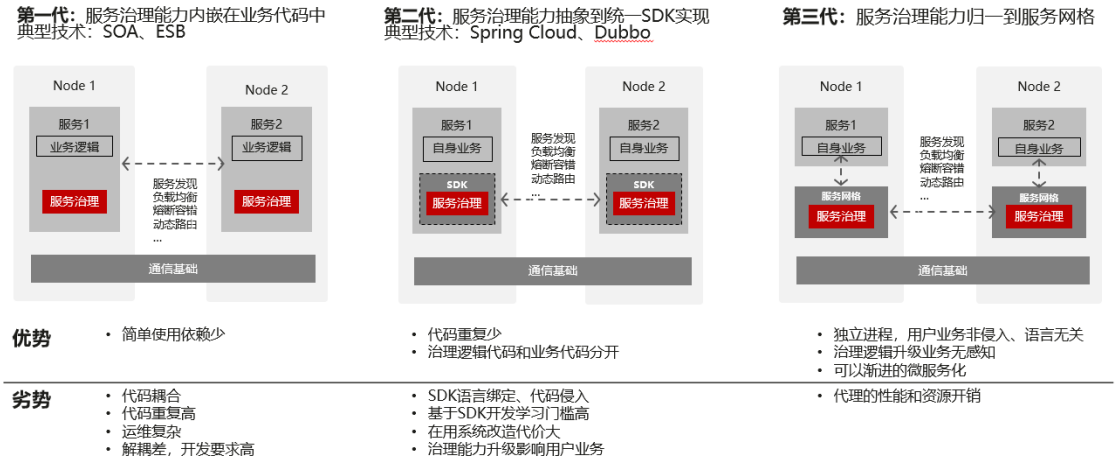
2. Kmesh介绍

2.1 Kmesh提出背景

Mesh（服务网格）是2016年由开发Linkerd软件的buoyant公司提出，Willian Morgan（Linkerd 的CEO）给出了Service Mesh的最初定义：服务网格（service mesh）是处理服务间通信的基础设施层。通过网络代理阵列的形式，为现代云原生应用提供透明、可靠的网络通信。

服务网格本质是解决微服务间如何更好通信的问题，通过负载均衡、灰度路由、熔断限流等治理规则，合理编排流量，实现最大化的集群服务能力，是服务治理演进的产物；

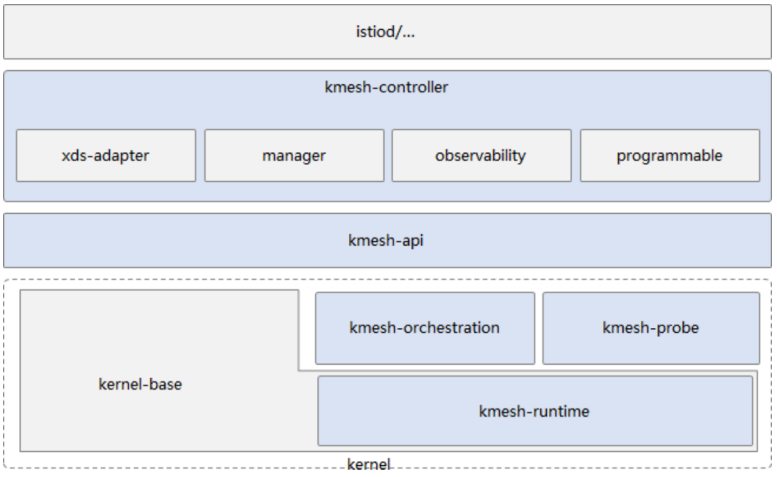
将服务治理的演进过程分为三代，并将其简单对比，从演进过程可以看出：服务治理能力逐步从业务中解耦，下沉到基础设施；



Kmesh是一款基于ebpf + 可编程内核实现的高性能服务网格数据面软件。通过将流量治理下沉到OS，实现网格内服务通信无需经过代理软件，大大缩减了流量转发路径，有效提升了服务访问的转发性能。

当前，网格数据面性能已是网格技术推广的关键，数据面技术也是多种多样的。Kmesh致力于为客户提供更轻量、更高效的服务治理能力，满足客户对安全、敏捷和效率的诉求。

2.2 Kmesh架构



Kmesh的主要部件包括：

`kmesh-controller`：kmesh管理程序，负责Kmesh生命周期管理、XDS协议对接、观测运维等功能；

`kmesh-api`：kmesh对外提供的api接口层，主要包括：xds转换后的编排API、观测运维通道等；

`kmesh-runtime`：kernel中实现的支持L3~L7流量编排的运行时；

`kmesh-orchestration`：基于ebpf实现L3~L7流量编排，如路由、灰度、负载均衡等；

`kmesh-probe`：观测运维探针，提供端到端观测能力；

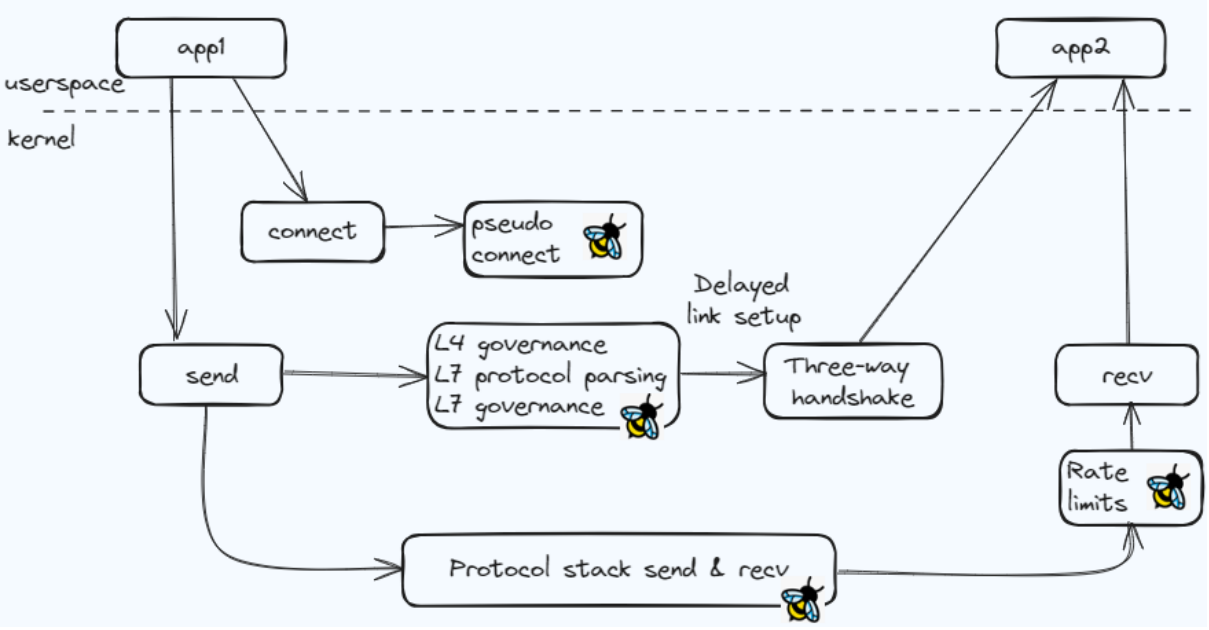
3. 源代码分析

Kmesh的主要功能就是内核级流量编排运行时，以及流量治理的编排，我将在下面主要分析这两部分的源代码。

3.1 内核级流量编排运行时

微服务通信一般先建立链接，再发送业务报文，如果想要无感的对业务报文做编排处理，通常需要对流量进行劫持，编排完成后再基于调整后的报文继续转发，这也是现在 Proxy 代理的实现；Kmesh 考虑随流完成治理工作，将链路建立时机推迟到业务报文发送阶段，以实现更高的编排处理性能。

整个治理过程大致如下图所示：



3.1.1 伪建链

pre_connect 流程挂载 bpf prog，若当前访问的目标地址在 xds 的 listener 范围，则调用 bpf_setsockopt 通过 TCP_ULP 将当前 socket 的 tcp proto hook重载到 kmesh_defer 的内核模块实现；

```
#if ENHANCED_KERNEL
    // todo build when kernel support http parse and route
    // defer conn
    ret = bpf_setsockopt(ctx, IPPROTO_TCP, TCP_ULP,
      (void *)kmesh_module_name, sizeof(kmesh_module_name));
    if (ret)
        BPF_LOG(ERR, KMESH, "bpf set sockopt failed! ret:%d\n", ret);
#else  // KMESH_ENABLE_HTTP
    ret = listener_manager(ctx, listener, NULL);
    if (ret != 0) {
        BPF_LOG(ERR, KMESH, "listener_manager failed, ret %d\n", ret);
        return ret;
    }
#endif // KMESH_ENABLE_HTTP
```

通过检查是否需要设置 ULP 选项来扩展 BPF 的 socket 选项设置功能。它允许在 TCP 连接中使用用户定义的上层协议，从而实现更灵活的网络性能优化。整体逻辑是先检测选项，然后处理和设置相关的协议参数。

在`0003-ipv4-bpf-Introduced-to-support-the-ULP-to-modify.patch`补丁中添加了对 ULP（Upper Layer Protocol）的支持，用于修改连接行为。是伪建链的逻辑所在。

```
static int _bpf_setsockopt(struct sock *sk, int level, int optname,
                           int optlen, const char __user *optval)
{
    // 检查传入的选项名称是否为 TCP_ULP
    if (optname == TCP_ULP) {
        char name[TCP_ULP_NAME_MAX] = {0}; // 定义字符数组用于存储 ULP 名称，初始化为 0

        // 从用户空间的 optval 复制数据到 name 中，确保不溢出
        strncpy(name, optval, min_t(long, optlen, TCP_ULP_NAME_MAX - 1));

        // 调用 tcp_set_ulp 函数，将 name 中的 ULP 名称应用到 socket sk 上
        return tcp_set_ulp(sk, name);
    } else {
        // 如果不是 TCP_ULP，则获取对应的 inet_connection_sock 和 tcp_sock 结构体
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct tcp_sock *tp = tcp_sk(sk);

        // 省略的代码部分处理其他 socket 选项...
    }
}
```

3.1.2 延迟建链

kmesh_defer 内核模块重写了 connect/send hook（在原生 hook 上做了增强）：

- 服务第一次走到 connect hook 时，会设置 bpf_defer_connect 标记，并不真正触发握手流程；
- send hook 中，若 sock 上设置了 bpf_defer_connect 标记，则触发connect，此时，会基于扩展的BPF_SOCK_OPS_TCP_DEFER_CONNECT_CB 调用 bpf prog，并在 bpf prog 中完成流量治理，然后基于调整后的通信五元组、报文进行建链和发送；

```
// 定义了一个全局的proto结构体指针kmesh_defer_proto，用于存储自定义的协议操作
static struct proto *kmesh_defer_proto = NULL;
// KMESH_DELAY_ERROR定义了一个错误码，用于表示延迟连接
#define KMESH_DELAY_ERROR -1000
```

在发送消息之前检查是否需要延迟连接，并在必要时调用defer_connect函数

```c
// 在发送消息之前检查是否需要延迟连接，并在必要时调用defer_connect函数
static int defer_connect_and_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
{
    struct socket *sock;
    int err = 0;

    // bpf_defer_connect 标志
    if (unlikely(inet_sk(sk)->bpf_defer_connect == 1)) {
        // 锁定套接字，防止其他操作干扰
        lock_sock(sk);
        // 将 defer_connect 标志重置为 0，表示当前正在处理连接
        inet_sk(sk)->defer_connect = 0;

        // 调用 defer_connect 函数，尝试建立连接
        err = defer_connect(sk, msg, size);
        if (err) {
            release_sock(sk);
            return -EAGAIN;
        }

        // 从套接字结构中获取实际的 socket 对象
        sock = sk->sk_socket;
        // 判断是否存在锁定的发送操作
        if (sock->ops->sendmsg_locked)
            // 如果存在，调用锁定的发送操作 sendmsg_locked，将消息发送出去，
            // 并将返回的错误代码存储在 err 中
            err = sock->ops->sendmsg_locked(sk, msg, size);
        release_sock(sk);
    }
    return err;
}
```

处理发送消息的请求，如果需要延迟连接，它会调用 defer_connect_and_sendmsg。

```c
// 处理发送消息的请求，如果需要延迟连接，它会调用defer_connect_and_sendmsg
static int defer_tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
{
    int ret;

    // 调用 defer_connect_and_sendmsg 函数，尝试在连接后发送消息
    ret = defer_connect_and_sendmsg(sk, msg, size);
    if (ret)
        return ret;

    return tcp_sendmsg(sk, msg, size);
}
```

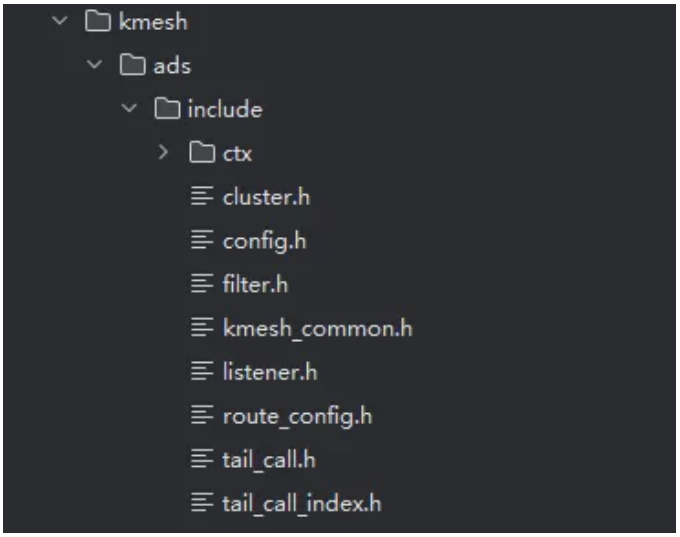
处理TCP连接请求，如果defer_connect标志被设置，它会返回一个错误，否则它会设置必要的标志并模拟TCP连接的开始

```c
// 处理TCP连接请求，如果defer_connect标志被设置，它会返回一个错误
// 否则它会设置必要的标志并模拟TCP连接的开始
static int defer_tcp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
{
     // 如果 defer_connect 标志为 1，表示不支持此连接
    if (inet_sk(sk)->defer_connect == 1)
        return -ENOTSUPP;
     // 如果 bpf_defer_connect 为真，表示已经存在延迟连接
     // 调用 tcp_v4_connect 函数以正常建立连接
    if (inet_sk(sk)->bpf_defer_connect)
        return tcp_v4_connect(sk, uaddr, addr_len);
    // 将 bpf_defer_connect 和 defer_connect 均设置为 1，表示现在正在处理延迟连接
    inet_sk(sk)->bpf_defer_connect = 1;
    inet_sk(sk)->defer_connect = 1;
    // 从用户提供的地址结构中获取目标端口，并将其赋值给套接字的 sk_dport
    sk->sk_dport = ((struct sockaddr_in *)uaddr)->sin_port;
    // 从用户提供的地址结构中获取目标地址，并调用 sk_daddr_set 函数设置该地址
    sk_daddr_set(sk, ((struct sockaddr_in *)uaddr)->sin_addr.s_addr);
    // 将套接字状态更新为 SS_CONNECTING，表示正在连接中
    sk->sk_socket->state = SS_CONNECTING;
    // 将 TCP 状态设置为 TCP_SYN_SENT，表示发送了 SYN 包，正在等待连接确认
    tcp_set_state(sk, TCP_SYN_SENT);
    // 返回 KMESH_DELAY_ERROR，表示连接被延迟
    return KMESH_DELAY_ERROR;
}
```

3.2 流量治理编排实现

xds 的治理规则复杂，层层匹配，超过了单个 eBPF 程序的复杂度限制；Kmesh 基于 eBPF Tail Calls 特性，将治理过程拆分成多个独立的 eBPF progs，具备较好的扩展性；

项目结构：



3.2.1 监视器组件-listener

这段代码是一个用于BPF（Berkeley Packet Filter，伯克利包过滤器）程序中的监听器（Listener）组件的实现，它主要用于处理网络流量并根据配置的过滤器链（FilterChain）来决定如何处理这些流量。代码使用了eBPF（extended BPF）的特性，特别是在内核中处理网络数据包的能力。

监听器查找：

这个函数根据网络地址查找对应的监听器配置。

```
static inline Listener__Listener *map_lookup_listener(const address_t *addr)
{
    return kmesh_map_lookup_elem(&map_of_listener, addr);
}
```

过滤器链匹配检查：

这个函数检查给定的过滤器链是否与特定的网络地址和上下文匹配。它检查目标端口和传输协议是否匹配。

```
static inline bool listener_filter_chain_match_check(
    const Listener__FilterChain *filter_chain, const address_t *addr, const
ctx_buff_t *ctx)
{}
```

过滤器链匹配：

这个函数遍历监听器的过滤器链，使用 `listener_filter_chain_match_check` 函数来找到第一个匹配的过滤器链。

```
static inline int listener_filter_chain_match(
    const Listener__Listener *listener,
    const address_t *addr,
    const ctx_buff_t *ctx,
    Listener__FilterChain **filter_chain_ptr,
    __u64 *filter_chain_idx)
{}
```

监听器管理器：

这个函数是监听器管理器的入口点。它首先尝试匹配过滤器链，如果找到匹配的链，则执行尾调用以继续处理请求。

```
static inline int listener_manager(ctx_buff_t *ctx, Listener__Listener *list
ener, struct bpf_mem_ptr *msg)
```

3.2.2 过滤器组件-filter

该部分展示了如何在Kmesh框架中使用eBPF来实现网络流量的过滤和处理。通过定义过滤器链和过滤器，Kmesh能够灵活地处理不同类型的网络流量，并根据需要将其转发到相应的处理逻辑。

- `filter_match_check`：此函数检查传入的过滤器配置类型是否为支持的类型（目前为 HTTP 连接管理器或 TCP 代理）。如果是，则返回匹配标志（`1`），否则返回不匹配（`0`）。

- `filter_chain_filter_match`：这个函数遍历一个过滤器链，尝试找到与给定地址和上下文匹配的过滤器。如果找到匹配的过滤器，它会将过滤器的指针和索引存储在提供的变量中，并返回成功标志（`0`）。如果没有找到匹配的过滤器，则返回错误代码（`1`）。
- `handle_http_connection_manager`：当匹配到 HTTP 连接管理器类型的过滤器时，此函数会被调用来处理 HTTP 连接。它会获取路由配置名称，并执行尾调用以继续处理请求。
- `filter_manager`：作为过滤器管理器的入口点，此函数根据过滤器的类型（HTTP 连接管理器或 TCP 代理）来决定调用哪个处理函数。它处理不同类型的网络流量，并根据过滤器的配置执行相应的操作。
- `filter_chain_manager`：作为过滤器链管理器的入口点，此函数负责查找和匹配过滤器链中的过滤器，并调用 `filter_manager` 来处理匹配的过滤器。它负责在过滤器链中导航，以找到正确的过滤器来处理网络流量。

过滤器匹配检查：

这个函数检查给定的过滤器是否匹配特定的地址和上下文。目前，它只支持 HTTP 连接管理器和 TCP 代理两种类型的过滤器。

```
static inline int filter_match_check(const Listener__Filter *filter, const a
ddress_t *addr, const ctx_buff_t *ctx)
{
    int match = 0;
    switch (filter->config_type_case) {
    case LISTENER__FILTER__CONFIG_TYPE_HTTP_CONNECTION_MANAGER:
    case LISTENER__FILTER__CONFIG_TYPE_TCP_PROXY:
        match = 1;
        break;
    default:
        break;
    }
    return match;
}
```

过滤器链匹配：

这个函数遍历过滤器链，尝试找到匹配给定地址和上下文的过滤器。

```
static inline int filter_chain_filter_match(
    const Listener__FilterChain *filter_chain,
    const address_t *addr,
    const ctx_buff_t *ctx,
    Listener__Filter **filter_ptr,
    __u64 *filter_ptr_idx)
{
// ...省略部分代码...for (unsigned int i = 0; i < KMESH_PER_FILTER_NUM; i++) {
        if (filter_match_check(filter, addr, ctx)) {
            *filter_ptr = filter;
            *filter_ptr_idx = (__u64) * ((__u64 *)ptrs + i);
            return 0;
        }
    }
    return -1;
}
```

HTTP 连接管理器处理：

函数处理 HTTP 连接管理器的逻辑，包括获取路由名称并进行尾调用以进一步处理。

```
static inline int handle_http_connection_manager(
    const Filter__HttpConnectionManager *http_conn, const address_t *addr, c
tx_buff_t *ctx, struct bpf_mem_ptr *msg)
{
// ...省略部分代码...KMESH_TAIL_CALL_CTX_KEY(ctx_key, KMESH_TAIL_CALL_ROUTER_C
ONFIG, *addr);
    KMESH_TAIL_CALL_CTX_VALSTR(ctx_val, msg, route_name);
    KMESH_TAIL_CALL_WITH_CTX(KMESH_TAIL_CALL_ROUTER_CONFIG, ctx_key, ctx_va
l);
    return KMESH_TAIL_CALL_RET(ret);
}
```

过滤器管理器：

这个函数是过滤器管理器的入口点，它根据过滤器的类型调用相应的处理函数。

```
int filter_manager(ctx_buff_t *ctx)
{
// ...省略部分代码...switch (filter->config_type_case) {
    case LISTENER__FILTER__CONFIG_TYPE_HTTP_CONNECTION_MANAGER:
// ...省略部分代码...
        ret = handle_http_connection_manager(http_conn, &addr, ctx, ctx_val->msg);
        break;
    case LISTENER__FILTER__CONFIG_TYPE_TCP_PROXY:
// ...省略部分代码...
        ret = tcp_proxy_manager(tcp_proxy, ctx);
        break;
    }
    return KMESH_TAIL_CALL_RET(ret);
}
```

过滤器链管理器：

这个函数是过滤器链管理器的入口点，它尝试找到匹配的过滤器并调用过滤器管理器。

```
int filter_chain_manager(ctx_buff_t *ctx)
{
// ...省略部分代码...
    ret = filter_chain_filter_match(filter_chain, &addr, ctx, &filter, &filter_idx);
    if (ret != 0) {
        return KMESH_TAIL_CALL_RET(-1);
    }
    KMESH_TAIL_CALL_CTX_KEY(ctx_key, KMESH_TAIL_CALL_FILTER, addr);
    KMESH_TAIL_CALL_CTX_VAL(ctx_val, ctx_val_ptr->msg, filter_idx);
    KMESH_TAIL_CALL_WITH_CTX(KMESH_TAIL_CALL_FILTER, ctx_key, ctx_val);
    return KMESH_TAIL_CALL_RET(ret);
}
```

3.2.3 路由器组件-router

这段代码是一个使用 BPF (Berkeley Packet Filter) 编写的路由配置管理器，主要用于在网络层面处理和转发数据包。BPF 程序通常在内核中运行，可以用于网络数据包过滤、监控和分析等。这个特定的代码片段是一个路由配置管理器，它使用 BPF 映射和辅助函数来处理网络流量的路由决策

- `map_lookup_route_config`：根据路由名称查找路由配置。
- `virtual_host_match_check`：检查虚拟主机是否与给定的地址和上下文匹配。

- `VirtualHost_check_allow_any`：检查虚拟主机名称是否为特殊名称 "allow_any"。
- `virtual_host_match`：根据路由配置、地址和上下文匹配虚拟主机。
- `check_header_value_match`：检查 HTTP 头部值是否与目标值匹配。
- `check_headers_match`：检查所有 HTTP 头部是否与路由匹配。
- `virtual_host_route_match_check`：检查路由是否与给定的地址、上下文和消息匹配。
- `virtual_host_route_match`：根据虚拟主机、地址、上下文和消息匹配路由。
- `select_weight_cluster`：从加权集群中选择一个集群。
- `route_get_cluster`：获取路由的集群名称。

路由配置查找：

```
static inline Route__RouteConfiguration *map_lookup_route_config(
const char *route_name)
{
    if (!route_name)
        return NULL;

    return kmesh_map_lookup_elem(&map_of_router_config, route_name);
}
```

集群选择：

```
static inline char *select_weight_cluster(Route__RouteAction *route_act)
{
    // ...省略部分代码...
    for (int i = 0; i < KMESH_PER_WEIGHT_CLUSTER_NUM; i++) {
        // ...省略部分代码...
        if (select_value <= 0) {
            cluster_name = kmesh_get_ptr_val(route_cluster_weight->name);
            BPF_LOG(DEBUG, ROUTER_CONFIG, "select cluster, name:weight %s:%d\n",
cluster_name, route_cluster_weight->weight);
            return cluster_name;
        }
    }
    return NULL;
}
```

路由配置管理器入口：

```
SEC_TAIL(KMESH_PORG_CALLS, KMESH_TAIL_CALL_ROUTER_CONFIG)
int route_config_manager(ctx_buff_t *ctx)
{
    // ...省略部分代码...
    route = virtual_host_route_match(virt_host, &addr, ctx,
     (struct bpf_mem_ptr *)ctx_val->msg);
    if (!route) {
        BPF_LOG(ERR, ROUTER_CONFIG, "failed to match route action, addr=%s\n",
         ip2str(&addr.ipv4, 1));
        return KMESH_TAIL_CALL_RET(-1);
    }

    cluster = route_get_cluster(route);
    if (!cluster) {
        BPF_LOG(ERR, ROUTER_CONFIG, "failed to get cluster\n");
        return KMESH_TAIL_CALL_RET(-1);
    }

    KMESH_TAIL_CALL_CTX_KEY(ctx_key, KMESH_TAIL_CALL_CLUSTER, addr);
    KMESH_TAIL_CALL_CTX_VALSTR(ctx_val_1, NULL, cluster);

    KMESH_TAIL_CALL_WITH_CTX(KMESH_TAIL_CALL_CLUSTER, ctx_key, ctx_val_1);
    return KMESH_TAIL_CALL_RET(ret);
}
```

3.2.4 cluster组件

用于管理网络集群（Cluster）和执行负载均衡（Load Balancing）。

集群和端点查找：

```
static inline Cluster__Cluster *map_lookup_cluster(const char *cluster_name)
{
    return kmesh_map_lookup_elem(&map_of_cluster, cluster_name);
}

static inline struct cluster_endpoints *map_lookup_cluster_eps(const char *c
luster_name)
{
    return kmesh_map_lookup_elem(&map_of_cluster_eps, cluster_name);
}

static inline struct cluster_endpoints *map_lookup_cluster_eps_data()
{
    int location = 0;
    return kmesh_map_lookup_elem(&map_of_cluster_eps_data, &location);
}
```

负载均衡：

这些函数实现了负载均衡策略，包括轮询（Round Robin）和获取端点的 socket 地址。

```
static inline void *loadbalance_round_robin(struct cluster_endpoints *eps)
{
// ...省略部分代码...}

static inline void *cluster_get_ep_identity_by_lb_policy(struct cluster_endp
oints *eps, __u32 lb_policy)
{
// ...省略部分代码...}

static inline Core__SocketAddress *cluster_get_ep_sock_addr(const void *ep_i
dentity)
{
// ...省略部分代码...}

static inline int cluster_handle_loadbalance(Cluster__Cluster *cluster, addr
ess_t *addr, ctx_buff_t *ctx)
{
// ...省略部分代码...}
```

集群管理器：

这是集群管理器的入口点，它负责查找集群配置，执行负载均衡，并更新上下文以进行下一步处理。

```
SEC_TAIL(KMESH_PORG_CALLS, KMESH_TAIL_CALL_CLUSTER)
int cluster_manager(ctx_buff_t *ctx)
{
// ...省略部分代码...}
```

3.2.5 eBPF Tail Calls

这段代码是一个 BPF (Berkeley Packet Filter) 程序的尾部调用（Tail Call）机制的实现，用于在不同的 BPF 程序之间进行跳转，而无需返回到用户空间。这种机制可以提高程序的执行效率。

尾部调用函数：

这个函数执行一个尾部调用，传递上下文 `ctx` 和调用索引 `index` 到 `map_of_tail_call_prog` 映射中定义的 BPF 程序。

```
static inline void kmesh_tail_call(ctx_buff_t *ctx, const __u32 index)
{
    bpf_tail_call(ctx, &map_of_tail_call_prog, index);
}
```

4. 总结

Kmesh作为一种创新的内核级服务网格数据平面，通过利用eBPF技术将流量治理能力下沉至操作系统内核，实现了高性能、低开销的Sidecarless服务网格架构，显著降低了服务间通信的延迟，同时减少了资源消耗，为云原生应用提供了一种高效、安全且易于管理的网络流量治理解决方案。

这段时间的学习经历让我意识到，开源项目不仅是技术实现的集合，更是知识共享和协作创新的平台。在这个过程中，我从社区的交流中获得了宝贵的经验，并领悟到开源文化的核心价值。未来，我将努力将这些知识应用于实践中，通过参与更多开源项目和贡献代码，来回馈这个充满活力的社区。同时，我深知，这一切的学习和成长离不开老师的悉心指导与支持。最后，我衷心感谢倪超老师为课程付出的辛勤努力和无私奉献！