深入理解并实现基本的 IPv6 协议栈

李睿远

Oct 16, 2025

从数据包结构到代码实现，亲手构建网络核心

随着互联网的快速发展，IPv4 地址的枯竭已成为不争的事实。IPv4 仅提供约 43 亿个地址，而全球设备数量早已远超这一数字，导致地址分配紧张和各种过渡技术的出现。相比之下，IPv6 采用 128 位地址长度，理论上可提供 2^{128} 个地址，这一数量级足以满足未来数十年的需求。根据最新数据，全球 IPv6 部署率持续上升，许多大型网络和服务商已全面支持 IPv6。因此，对于现代开发者和网络工程师而言，深入理解 IPv6 不再是可选技能，而是必备知识。

IPv6 的核心优势远不止于地址空间的扩展。其报头格式经过简化，固定为 40 字节，去除了 IPv4 中的可选字段，转而使用扩展报头链式处理，这大大提升了路由效率。此外，IPv6 内置了对 IPsec 的支持，增强了端到端的安全性，同时更好地支持移动性和无状态地址自动配置（SLAAC），使得设备能够快速接入网络。本文旨在通过理论与实践相结合的方式，引导读者从零开始实现一个用户态的 IPv6 协议栈。我们将聚焦于 IPv6 层、ICMPv6 和邻居发现协议（NDP），暂不涉及 TCP/UDP 等上层协议，最终目标是实现一个能响应 Ping（ping6）并进行邻居发现的微型协议栈。

1　理论基础：深入剖析 IPv6 核心机制

IPv6 数据包结构是理解其设计理念的基础。IPv6 基本报头包含多个字段：版本（Version）固定为 6，流量类别（Traffic Class）用于 QoS，流标签（Flow Label）标识特定流，载荷长度（Payload Length）指示扩展报头和数据的总长度，下一个报头（Next Header）指定后续内容类型，跳数限制（Hop Limit）类似 IPv4 的 TTL，以及源和目的地址各 128 位。与 IPv4 报头相比，IPv6 报头更加简化，去除了校验和、分片等相关字段，转而依赖上层协议处理。扩展报头以链式结构存在，例如 Hop-by-Hop Options 和 ICMPv6（下一个报头值为 58），它们允许灵活地添加功能而不改变基本结构。

ICMPv6 在 IPv6 中扮演着多重角色，不仅是错误报告和网络诊断的工具，更是邻居发现协议（NDP）的载体。NDP 替代了 IPv4 中的 ARP，用于解析 IPv6 地址到 MAC 地址的映射。关键报文类型包括 Echo Request 和 Echo Reply 用于实现 ping6，Neighbor Solicitation 和 Neighbor Advertisement 用于地址解析，以及 Router Solicitation 和 Router Advertisement 用于无状态地址自动配置。NDP 的状态机涉及多个状态：INCOMPLETE、REACHABLE、STALE、DELAY 和 PROBE，它们根据超时和事件进行转换，确保邻居关系的有效管理。

在地址体系方面，IPv6 定义了单播、组播和任播地址。单播地址包括全球单播（如 2000::/3）和链路本地地址（fe80::/10），其中链路本地地址在 NDP 通信中至关重要。无状态地址自动配置（SLAAC）允许设备通过接收 Router Advertisement 报文并结合 EUI-64 格式生成接口标识符，自动配置全球单播地址，这简化了网络部署过程。

2 实战准备：搭建开发环境与架构设计

在技术选型上，我们推荐使用 C 语言进行实现，因为它贴近系统底层，能够直接操作内存和网络接口，适合协议栈开发。开发平台选择 Linux，因其原生支持 TUN/TAP 设备，允许用户态程序模拟网络接口。关键工具包括 tcpdump 或 Wireshark 用于抓包分析，ping6 和 traceroute6 用于测试，而 TUN/TAP 设备则是核心，它通过字符设备接口提供原始数据包的读写能力。

协议栈的软件架构应模块化设计。TUN 驱动模块负责从 TUN 设备读取和写入原始以太网帧；以太网帧处理模块解析帧类型，识别 IPv6 数据包（以太网类型 0x86DD）；IPv6 解包/组包模块处理基本报头和扩展报头；ICMPv6 处理模块响应 Echo 请求和邻居发现报文；邻居缓存表存储 IPv6 地址到 MAC 地址的映射及其状态；定时器模块处理 NDP 状态超时和重传。数据流从 TUN 设备进入，经以太网帧解析后，如果是 IPv6 包，则递交给 IPv6 模块，再根据下一个报头调用 ICMPv6 模块，最终可能更新邻居缓存或返回响应。

3 代码实现：构建核心模块

首先，我们搭建项目骨架并初始化 TUN 接口。以下代码展示如何在 C 语言中创建和配置 TUN 设备。

```
#include <fcntl.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>

int tun_alloc(char *dev) {
    struct ifreq ifr;
    int fd, err;
    if ((fd = open("/dev/net/tun", O_RDWR)) < 0) {
        return -1;
    }
    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
    if (dev) {
        strncpy(ifr.ifr_name, dev, IFNAMSIZ);
    }
    if ((err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0) {
        close(fd);
        return err;
    }
    strcpy(dev, ifr.ifr_name);
    return fd;
```

```
}
```

这段代码通过打开 /dev/net/tun 设备文件并使用 ioctl 调用配置 TUN 接口。IFF_TUN 标志表示创建 TUN 设备（三层网络设备），IFF_NO_PI 表示不提供包信息，从而简化数据包处理。函数返回文件描述符，用于后续读写操作。初始化后，程序可以进入数据包读写循环，不断从文件描述符读取原始数据包并进行处理。

接下来，实现以太网帧与 IPv6 报文的解析。我们定义相关结构体并处理字节序转换。

```
#include <arpa/inet.h>
#include <stdint.h>

struct ethhdr {
    unsigned char h_dest[6];
    unsigned char h_source[6];
    uint16_t h_proto;
};

struct ip6_hdr {
    uint32_t v_tc_fl;
    uint16_t plen;
    uint8_t nxt;
    uint8_t hlim;
    struct in6_addr src;
    struct in6_addr dst;
};

void parse_ethernet(unsigned char *buffer, int length) {
    struct ethhdr *eth = (struct ethhdr *)buffer;
    uint16_t proto = ntohs(eth->h_proto);
    if (proto == 0x86DD) {
        parse_ipv6(buffer + sizeof(struct ethhdr), length - sizeof(struct ethhdr));
    }
}
```

在这段代码中，我们定义了以太网帧头和 IPv6 报头的结构体。ethhdr 包含目的和源 MAC 地址以及协议类型字段；ip6_hdr 使用组合字段处理版本、流量类别和流标签，注意这些字段需要字节序转换，例如 ntohs 用于将网络字节序转换为主机字节序。在 parse_ethernet 函数中，我们检查协议类型是否为 IPv6（0x86DD），如果是，则调用 IPv6 解析函数。IPv6 报头解析需要校验版本字段是否为 6，并处理载荷长度以确保数据完整性。

现在，实现 ICMPv6 协议处理，首先是响应 Ping 请求。

```
struct icmp6_hdr {
    uint8_t type;
    uint8_t code;
```

```
    uint16_t checksum;
};

void handle_icmp6(unsigned char *buffer, int length, struct in6_addr src, struct
    ↪ in6_addr dst) {
    struct icmp6_hdr *icmp6 = (struct icmp6_hdr *)buffer;
    if (icmp6->type == 128) {
        icmp6->type = 129;
        struct in6_addr tmp = src;
        src = dst;
        dst = tmp;
        icmp6->checksum = 0;
        icmp6->checksum = compute_checksum(src, dst, buffer, length);
        send_packet(buffer, length);
    }
}
```

这里，我们定义 ICMPv6 报头结构，类型字段为 128 表示 Echo Request，129 表示 Echo Reply。处理时，我们改变类型，交换源和目的地址，并重新计算校验和。校验和计算涉及 IPv6 伪首部，包括源地址、目的地址、载荷长度和下一个报头值。计算函数 compute_checksum 需要实现，它基于标准算法，对伪首部和 ICMPv6 报文进行求和。

校验和计算是 ICMPv6 的关键部分。IPv6 的校验和使用伪首部，其结构包括源地址（16 字节）、目的地址（16 字节）、上层包长度（32 位）、下一个报头（8 位，后跟 24 位零）。公式可表示为：校验和 = ~(sum(伪首部) + sum(ICMPv6 报文))，其中 sum 是 16 位字的补码和。在代码中，我们需要遍历这些数据并计算。

```
uint16_t compute_checksum(struct in6_addr src, struct in6_addr dst, unsigned char *
    ↪ data, int len) {
    uint32_t sum = 0;
    for (int i = 0; i < 16; i += 2) {
        sum += (src.s6_addr[i] << 8) | src.s6_addr[i+1];
    }
    for (int i = 0; i < 16; i += 2) {
        sum += (dst.s6_addr[i] << 8) | dst.s6_addr[i+1];
    }
    sum += len;
    sum += 58;
    for (int i = 0; i < len; i += 2) {
        if (i+1 < len) {
            sum += (data[i] << 8) | data[i+1];
        } else {
            sum += data[i] << 8;
```

```
        }
    }
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    return ~sum;
}
```

这段代码实现了校验和计算。我们首先处理伪首部：源和目的地址各作为 8 个 16 位字处理，上层包长度作为 32 位值但以 16 位字形式添加，下一个报头值 58 也作为 16 位字添加。然后处理 ICMPv6 报文数据，以 16 位为单位求和，并处理进位。最终返回补码。注意，在实现中，我们假设数据是网络字节序，且长度参数 `len` 是 ICMPv6 报文的字节长度。

4 实现邻居发现协议

邻居发现协议（NDP）是 IPv6 的核心组件，用于解析链路层地址。我们首先设计邻居缓存表，存储 IPv6 地址到 MAC 地址的映射及其状态。

```
#include <time.h>

#define ND6_INCOMPLETE 0
#define ND6_REACHABLE 1

struct neighbor_cache {
    struct in6_addr ip6_addr;
    unsigned char mac_addr[6];
    int state;
    time_t last_updated;
};

struct neighbor_cache ncache[100];
int ncache_size = 0;

void handle_neighbor_solicitation(unsigned char *buffer, int length, struct in6_addr
    ↪ src, struct in6_addr dst) {
    struct icmp6_ns *ns = (struct icmp6_ns *)buffer;
    if (is_my_address(ns->target)) {
        send_neighbor_advertisement(ns->target, my_mac_addr);
    }
}

```

```
void send_neighbor_solicitation(struct in6_addr target) {
}
```

在这段代码中，我们定义了邻居缓存表的结构，包括 IPv6 地址、MAC 地址、状态和时间戳。`handle_neighbor_solicitation` 函数处理收到的 NS 报文，如果目标地址匹配本机地址，则发送 NA 响应。发送 NA 时，需要设置相关标志，如覆盖位（O）和请求位（S），以指示响应的性质。同时，当协议栈需要发送数据包但缓存中无对应条目时，应主动发送 NS 报文，并将状态设置为 INCOMPLETE，等待 NA 响应后更新为 REACHABLE。

5 测试与验证

搭建测试环境时，需要将协议栈程序与主机网络连接。通过配置 TUN 设备并设置路由，使主机的 IPv6 流量指向该设备。例如，在 Linux 中，可以使用 `ip link set dev tun0 up` 和 `ip route add local 2001:db8::/64 dev tun0` 等命令。

功能测试包括 Ping 测试和邻居发现测试。从主机执行 `ping6 fe80::1%tun0`（假设协议栈配置了链路本地地址），同时使用 Wireshark 抓包验证请求和回复报文。此外，通过 `ip -6 neighbor show` 命令观察邻居表，确认协议栈的条目出现并达到 REACHABLE 状态。抓包结果应显示完整的 NS/NA 和 Echo 交互过程：例如，当发送 Ping 请求时，Wireshark 应显示 Echo Request 和 Echo Reply 报文，且邻居发现报文应显示 NS 和 NA 的交换，确保地址解析成功。

通过本文的实现，我们构建了一个具备 IPv6 基本报头处理、ICMPv6 Echo 响应和邻居发现功能的用户态协议栈。这个过程不仅加深了对 IPv6 协议的理解，还展示了如何将理论转化为实践。未来，可以扩展支持更多扩展报头如分片、实现 DHCPv6 客户端、添加 TCP/UDP 上层协议支持，以及优化性能和多线程处理。网络协议的设计精巧而实用，亲手实现是掌握它的最佳途径。鼓励读者尝试实现，并参考相关开源项目进一步探索。