

网络系统

RocketOS 的网络系统是一个基于 *smoltcp* 协议栈的网络实现，旨在提供高效灵活的网络通信能力。它支持 AF_INET, AF_INET6, AF_UNIX 和 AF_ALG 等多种地址族的套接字，能够处理 IPv4 和 IPv6 两类 IP 地址，同时支持 TCP 和 UDP 两种传输协议并通过了 iperf, netperf, ltp 相关测试。RocketOS 通过统一的接口管理所有网络设备和套接字，并支持 riscv64 和 loongarch64 下的 glibc 和 musl 共计 4 种架构。

网络系统概述

RocketOS 的网络工作模式如下:

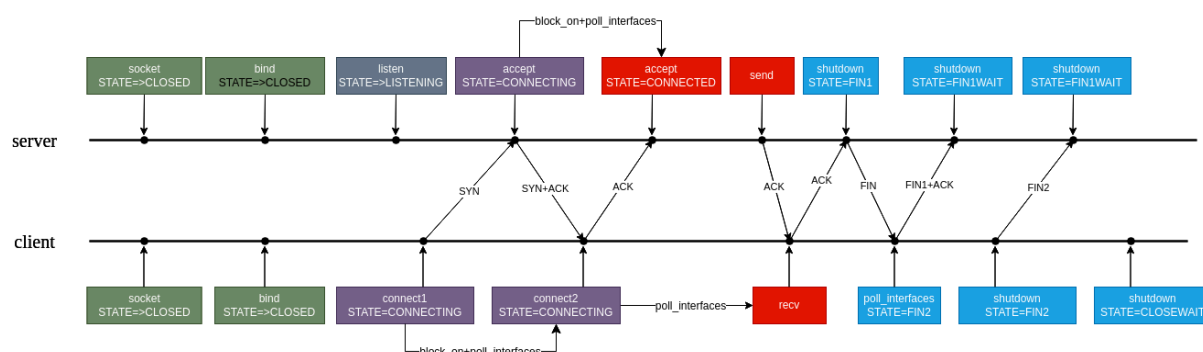


Figure 1: net 工作模式

RocketOS 的网络系统包括以下几个主要组件:

- **NetDevice:** 网络设备接口，定义了网络设备的基本操作和特性。根据抽象 NetDevice 接口可以实现不同网络设备，包括虚拟本地设备 VirtioNetDevice 和虚拟本地回环设备 LoopbackDev。
- **InterfaceWrapper:** RocketOS 的网卡抽象，系统使用 InterfaceWrapper 来封装 *smoltcp* 的 Interface 接口和 NetDeviceWrapper，提供对网卡设备的统一管理和操作。可以支持创建多个硬件网卡在 os 中的映射。在 RocketOS 中，同 linux 类似分别管理着 lo 回环设备和 eth33 虚拟网卡设备。其中 eth33 虚拟网卡设备通过 qemu 的 10.0.2.15 映射为主机的 10.0.2.2 接口。
- **ListenTable:** RocketOS 的全局监听表，用于管理所有监听的端口和连接的 socket。监听表维护一个端口到对应连接套接字的映射，通过访问 ListenTable 中连接套接字的状态判断是否允许连接。
- **Socket:** Socket 提供了对内核套接字的封装，并实现 FileOp 接口，允许通过文件描述符进行访问和操作。

网络 Device 设备-物理层

Loongarch64 与 riscv64 适配

RocketOS 的网络系统支持 loongarch64 和 riscv64 两种架构，通过分析设备树来映射到内核空间。在 riscv64 中，网络设备通过 MMIO 映射到设备地址空间，而在 loongarch64 中，网络设备通过 PCI 总线进行挂载。因此，在 RocketOS 中，选择根据不同架构条件编译分析设备树。

#riscv qemu 网络配置

```
-device virtio-net-device, netdev=net -netdev user, id=net,
```

```

hostfwd=tcp::5555-:5555, hostfwd=udp::5555-:5555\
#loongarch qemu 网络配置
-device virtio-net-pci, netdev=net -netdev user, id=net, hostfwd=tcp::5556-:5555,
hostfwd=udp::5556-:5555 \

```

- 在 riscv64 中，通过传入 rust_main 的 dtb_address 确定设备树地址，遍历设备树节点查找 compatible 属性为 virtio-net 的节点，并通过获取其 reg 属性来确定设备的 MMIO 地址并映射到内核。

算法 0-0: riscv 网络设备初始化流程

```

input: dtb_addr
output: initialized net device
1 let dev_tree ← Fdt::from_ptr(dtb_addr + KERNEL_BASE)
2 address_cells ← dev_tree.root().prop("address-cells").value[3]
3 size_cells ← dev_tree.root().prop("size-cells").value[3]
4 for node in dev_tree.all_nodes() do
5   for prop in node.properties() do
6     log(prop.name)
7 for node in dev_tree.all_nodes() do
8   if node.name == "soc" then
9     for child in node.children() do
10      if child.name == "virtio_mmio@10008000" then
11        reg ← parse_reg(child, address_cells, size_cells)
12        mmio_base ← reg[0].start
13        mmio_size ← reg[0].length
14        map_area ← MapArea::new( VPNRange(KERNEL_BASE+mmio_base,
15        KERNEL_BASE+mmio_base+mmio_size), Linear, R|W )
16        KERNEL_SPACE.lock().push(map_area)
17        sfence.vma()
18        NET_DEVICE_ADDR.lock().replace(KERNEL_BASE+mmio_base)
19        header ← NonNull((KERNEL_BASE+mmio_base) as mut VirtIOHeader)
20        transport ← MmioTransport::new(header)
21        log("vendor=", transport.vendor_id(), "version=", transport.version(), "type=",
22        transport.device_type())
23        dev ← VirtioNetDevice::new(transport)
24        net::init(Some(dev))
25      return
26 log("not find a net device")

```

- 而在 loongarch64 中，通过遍历 PCI 总线设备，查找 device_type 为 network 的节点，并获取其 BAR 寄存器来确定设备的地址并映射到内核。

算法 0-0: 基于 PCI 的 VirtIO 设备初始化流程

```

input: pci_root, allocator
output: 初始化并启动 VirtIO 设备
1 for (device_fn, info) in pci_root.enumerate_bus(0) do

```

```

2   status, command ← pci_root.get_status_command(device_fn)
3   log("Found", info, "at", device_fn, "status", status, "command", command)
4   if virtio_device_type(&info) then virtio_type
5       log(" VirtIO", virtio_type)
6       allocateBars(&mut pci_root, device_fn, &mut allocator)
7       dump_bar_contents(&mut pci_root, device_fn, 4)
8       transport ← PciTransport::new::(&mut pci_root, device_fn).unwrap()
9       log("Detected virtio PCI device with type", transport.device_type(), "features",
10         transport.read_device_features() )
11       virtio_device(transport)
12   fn virtio_device(transport) do
13       match transport.device_type() with
14         DeviceType::Block => virtio_blk(transport)
15         DeviceType::Network =>
16             log("[initialize net]")
17             virtio_net(transport)
18         t => log("Unsupported VirtIO device type", t)

```

NetDevice 封装

RocketOS 的网络设备封装了 smoltcp 的 Device 接口，并通过 NetDeviceWrapper 实现了对底层设备的抽象。这使得 RocketOS 能够支持多种类型的网络设备，包括虚拟网卡和回环设备。

```

1 // 网络设备管理，实现 sync 和 send 特性，以便在多线程环境中安全使用
2 pub trait NetDevice: Sync + Send {
3     // 获取设备容量
4     fn capabilities(&self) -> smoltcp::phy::DeviceCapabilities;
5     // 获取设备 mac 地址
6     fn mac_address(&self) -> EthernetAddress;
7     // 是否可以发送数据
8     fn isok_send(&self) -> bool;
9     // 是否可以接收数据
10    fn isok_recv(&self) -> bool;
11    // 一次最多可以发送报文数量
12    fn max_send_buf_num(&self) -> usize;
13    // 一次最多可以接收报文数量
14    fn max_recv_buf_num(&self) -> usize;
15    // 回收接收 buffer
16    fn recycle_recv_buffer(&mut self, recv_buf: NetBufPtr);
17    // 回收发送 buffer
18    fn recycle_send_buffer(&mut self) -> Result<(), ()>;
19    // 发送数据
20    fn send(&mut self, ptr: NetBufPtr);
21    // 接收数据
22    fn recv(&mut self) -> Option<NetBufPtr>;
23    // 分配一个发送的网络缓冲区
24    fn alloc_send_buffer(&mut self, size: usize) -> NetBufPtr;
25 }

```

代码 1: NetDevice trait

同时为了使 NetDevice 定义符合 smoltcp 的 Device 接口需求，定义了 NetDeviceWrapper 结构体，通过 RefCell 包装 Box<dyn NetDevice>，允许内部的可变访问，以便在实现 smoltcp 中的 Device trait 时提供对底层设备的操作。

```
1 pub struct NetDeviceWrapper {
2     inner: RefCell<Box<dyn NetDevice>>,
3 }
```

代码 2: NetDeviceWrapper

RocketOS 的网络设备还支持在有限的空间中动态对 Device 发送接收的报文空间进行分配和回收，系统通过定义 NetBufPool 统一管理 Device 的报文空间并实现 alloc 和 dealloc 方法。Device 在 recycle_recv_buffer 和 recycle_send_buffer 便可通过调用 alloc 和 dealloc 方法来分配和回收报文空间，从而提高网络通信的效率并降低内存碎片化。

```
1 /// A pool of [NetBuf]s to speed up buffer allocation.
2 ///
3 /// It divides a large memory into several equal parts for each buffer.
4 pub struct NetBufPool {
5     //可以存储的 netbuf 个数
6     capacity: usize,
7     //每个 netbuf 的长度
8     buf_len: usize,
9     pool: Vec<u8>,
10    //用于存储每个待分配的 netbuf 的 offset
11    free_list: Mutex<Vec<usize>>,
12 }
```

代码 3: NetBufPool

算法 0-0: NetBuf 缓冲区分配与回收

```
fn alloc(self: Arc) → NetBuf
    output: 新分配的 NetBuf
1  offset ← self.free_list.lock().pop().unwrap()
2  buf_ptr ← NonNull(self.pool.as_ptr().add(offset) as mut u8)
3  return NetBuf { header_len: 0, packet_len: 0, capacity: self.buf_len, buf_ptr: buf_ptr, pool_offset:
4      offset, pool: Arc::clone(self), }
5
fn dealloc(self, offset: usize) → ()
    precondition: offset % self.buf_len == 0
5  assert(offset % self.buf_len == 0)
6  self.free_list.lock().push(offset)
```

系统还为具体的网络设备实现了 smoltcp 的 Device trait，以便在使用 smoltcp 的 poll 轮询机制来嗅探网络事件时，通过使用 Device trait 方法调用 NetDeviceWrapper 的相关方法来处理网络数据包的发送和接收。

如下代码和图所示，*smoltcp* 通过环形令牌网络实现对网络设备的轮询，这里实现的 Device trait 便是在轮询中对令牌进行分配和管理，并在 NetDeviceWrapper 获得令牌时，通过 NetDevice 接口来处理网络数据包的发送和接收。

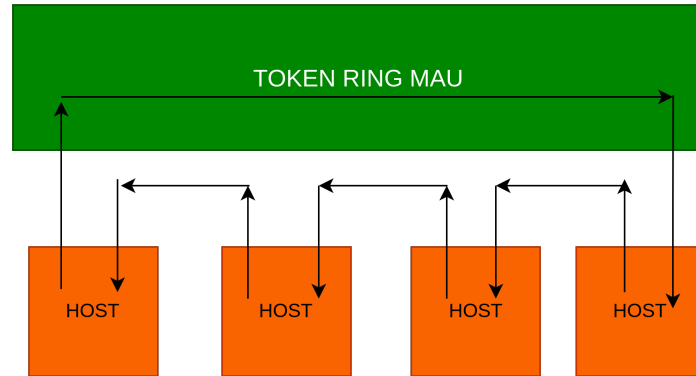


Figure 2: 环形令牌网络

算法 0-0: NetDeviceWrapper 驱动接口实现伪代码

```

impl Device for NetDeviceWrapper
  output: Rx/Tx 令牌或设备能力
  1 fn receive(self, timestamp: Instant) → Option<(RxToken, TxToken)>
  2   dev ← self.inner.borrow_mut()
  3   if let Err(e) = dev.recycle_tx_buffers() then
  4     warn("recycle_tx_buffers failed:", e)
  5     return None
  6   if ¬dev.can_transmit() then return None
  7   match dev.receive() with
  8     Ok(buf) => rx_buf ← buf
  9     Err(DevError::Again) => return None
 10    Err(err) =>
 11      warn("receive failed:", err)
 12      return None
 13    return Some((NetRxToken(&self.inner, rx_buf), NetTxToken(&self.inner)))
 14
 15 fn transmit(self, timestamp: Instant) → Option
 16   dev ← self.inner.borrow_mut()
 17   if let Err(e) = dev.recycle_tx_buffers() then
 18     warn("recycle_tx_buffers failed:", e)
 19     return None
 20   if dev.can_transmit() then
 21     return Some(AxNetTxToken(&self.inner))
 22   else return None
 23
 24 fn capabilities(self) → DeviceCapabilities
 25   caps ← DeviceCapabilities::default()
 26   caps.max_transmission_unit ← 1514

```

```
27 caps.max_burst_size ← None
28 caps.medium ← Medium::Ethernet
29 return caps
```

Interface 设备-数据链路层

RocketOS 的网络接口设备通过 InterfaceWrapper 封装了 *smoltcp* 的 Interface 和 NetDeviceWrapper, 提供对网卡设备的统一管理和操作。

通过封装 *smoltcp* 的 Interface 设备, 系统可以通过 *smoltcp* 的 poll 轮询机制来嗅探网络事件; 通过封装 NetDeviceWrapper 设备, 系统可以在出现网络事件时, 通过 NetDevice 接口来处理网络数据包的发送和接收。

```
1 pub struct InterfaceWrapper {
2     //smoltcp 网卡抽象
3     iface: Mutex<Interface>,
4     //网卡 ethenet 地址
5     address: EthernetAddress,
6     //名字 eth0
7     name: &'static str,
8     dev: Mutex<NetDeviceWrapper>,
9 }
```

代码 4: InterfaceWrapper 结构体

系统通过 poll_interfaces 方法实现多个网卡设备的轮询, 并在轮询过程中借由实现的 Device trait 对存在网络事件的设备进行数据收发。

```
1 pub fn poll_interfaces(&self) {
2     //对本地回环设备轮询
3     LOOPBACK.lock().poll(
4         Instant::from_micros_const((current_time_nanos() / NANOS_PER_MICROS) as i64),
5         LOOPBACK_DEV.lock().deref_mut(),
6         &mut self.0.lock(),
7     );
8     //对 ens0 设备轮询
9     ETH0.poll(&self.0);
10 }
```

代码 5: poll_interfaces

ListenTable 监听表-网络层

RocketOS 实现通过一个全局的 LISTENTABLE 管理所有正在监听的端口和连接的 socket。监听表维护一个端口到对应连接套接字的映射, 通过访问 ListenTable 中连接套接字的状态判断是否允许连接。

```

1  static LISTEN_TABLE: LazyInit<ListenTable> = LazyInit::new();
2  pub struct ListenTable{
3      //是由 listenentry 构建的监听表
4      //监听表的表项个数与端口个数有关, 每个端口只允许一个地址使用
5      table:Box<[Mutex<Option<Box<ListenTableEntry>>>]>,
6  }
7  #[derive(Clone)]
8  struct ListenTableEntry{
9      //表示监听的 server 地址 addr
10     listen_endpoint:IpListenEndpoint,
11     task_id:usize,
12     //这里由于 sockethandle 与 socket 存在 RAII 特性, 因而可以保存 sockethandle
13     syn_queue:VecDeque<SocketHandle>
14 }

```

代码 6: ListenTable 结构体

Socket 封装-传输层

RocketOS 对于 socket 实现了 3 层封装，实现了对 AF_UNIX，AF_INET，AF_ALG，AF_INET6 套接字的管理，支持 tcp，udp 协议。实现 FileOp 接口，允许通过文件描述符进行访问和操作。内核 socket 定义如下，Socket 结构体封装了协议类 socketinner，套接字类型，以及具体的套接字实现。它还包含了一些状态信息，发送和接收缓冲区大小等。其中所有内容均通过原子操作或者 Mutex 进行保护，以确保在多线程环境下的安全性和一致性。而 socketinner 进一步封装了套接字的具体实现，包括 tcp，udp，unix 和 alg 等类型的套接字。

```

1  pub struct Socket {
2      pub domain: Domain,
3      pub socket_type: SocketType,
4      inner: SocketInner,
5      close_exec: AtomicBool,
6      send_buf_size: AtomicU64,
7      recv_buf_size: AtomicU64,
8      congestion: Mutex<String>,
9      recvtimeout: Mutex<Option<TimeSpec>>,
10     dont_route: bool,
11     ...
12 }
13 pub enum SocketInner {
14     Tcp(TcpSocket),
15     Udp(UdpSocket),
16     Unix(UnixSocket),
17     Alg(AlgSocket),
18 }

```

代码 7: InterfaceWrapper

套接字通常作为文件描述符使用，因此 RocketOS 的套接字还实现了 FileOp 接口，允许通过文件描述符进行访问和操作。这使得套接字可以像文件一样进行读写操作，并支持文件描述符的相关系统调用。

RocketOS 对于 Socket 的管理遵循 RAI 思想，为 Socket 实现 drop trait，当 socket shutdown 时会通过 drop 释放对应的资源并从全局的 SocketSetWrapper 中移除对应句柄。

```
1 pub fn remove(&self, handle: SocketHandle) {  
2     let socket=self.0.lock().remove(handle);  
3     drop(socket);  
4 }
```

代码 8: Socket remove

通过上述设计，RocketOS 可以做到统一封装多种物理与虚拟网卡（如 Virtio、回环），支持 IPv4/IPv6、TCP/UDP、AF_UNIX、AF_ALG 等多种地址族与协议，在 RISC-V64 与 LoongArch64 上高效灵活地管理网卡轮询、端口监听与 Socket 文件操作。