



# StratoVirt系统剖析

主讲人：李佳杰



# 课程目标

- 了解 StratoVirt 整体代码架构
- 了解StratoVirt CPU子系统的设计和实现
- 了解StratoVirt 内存子系统的设计和实现
- 了解StratoVirt IO 子系统的设计和实现



Stratovirt 整体代码架构

CPU子系统

内存子系统

IO子系统



# 01 Stratovirt 总体代码架构



## ► StratoVirt的核心架构

### 逻辑架构

#### •外部API:

- QMP
- 兼容OCI
- libvirt

#### •BootLoader:

- Direct boot
- Standard boot

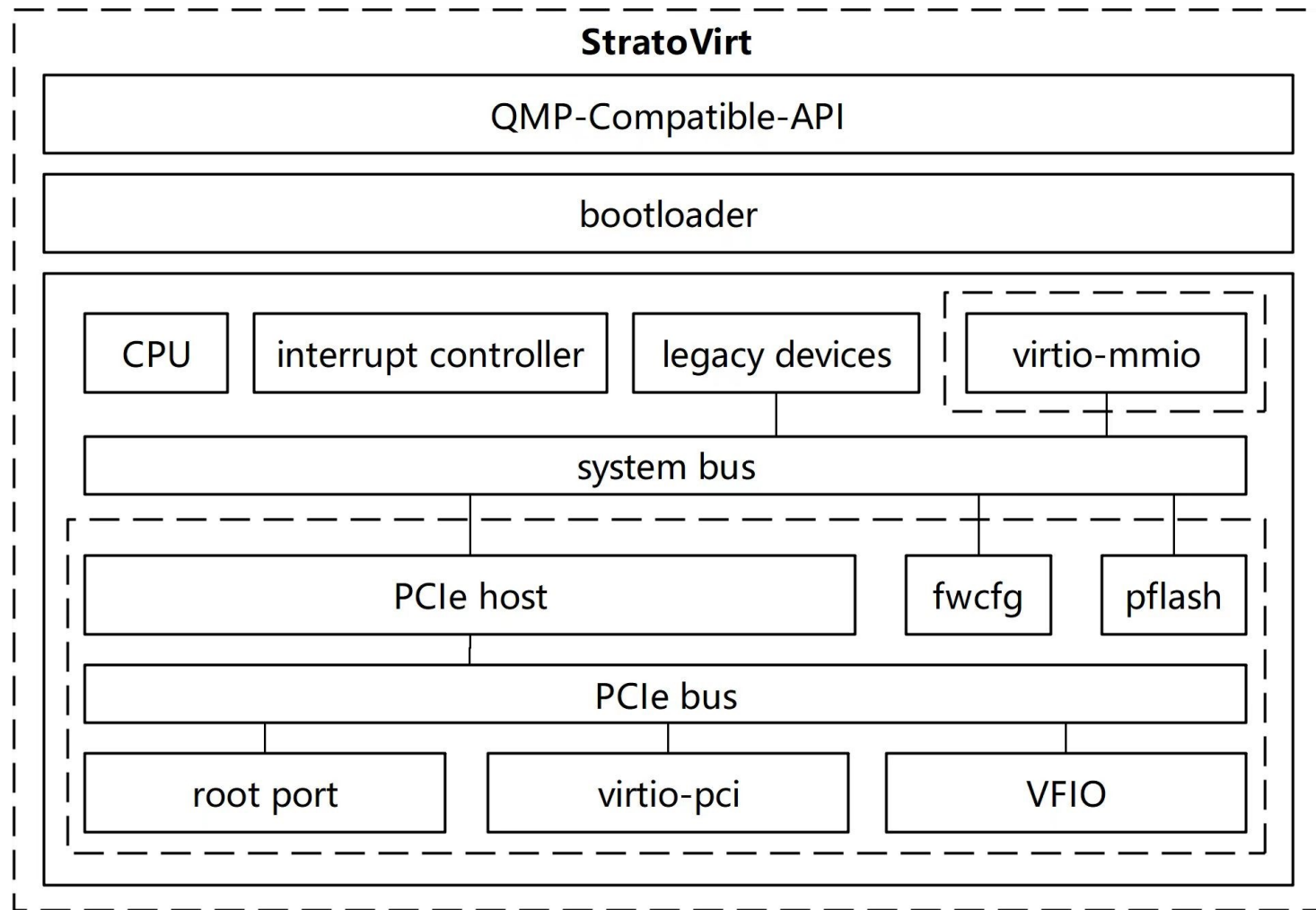
#### •模拟主板:

- 轻量机型
- 标准机型

### 运行架构

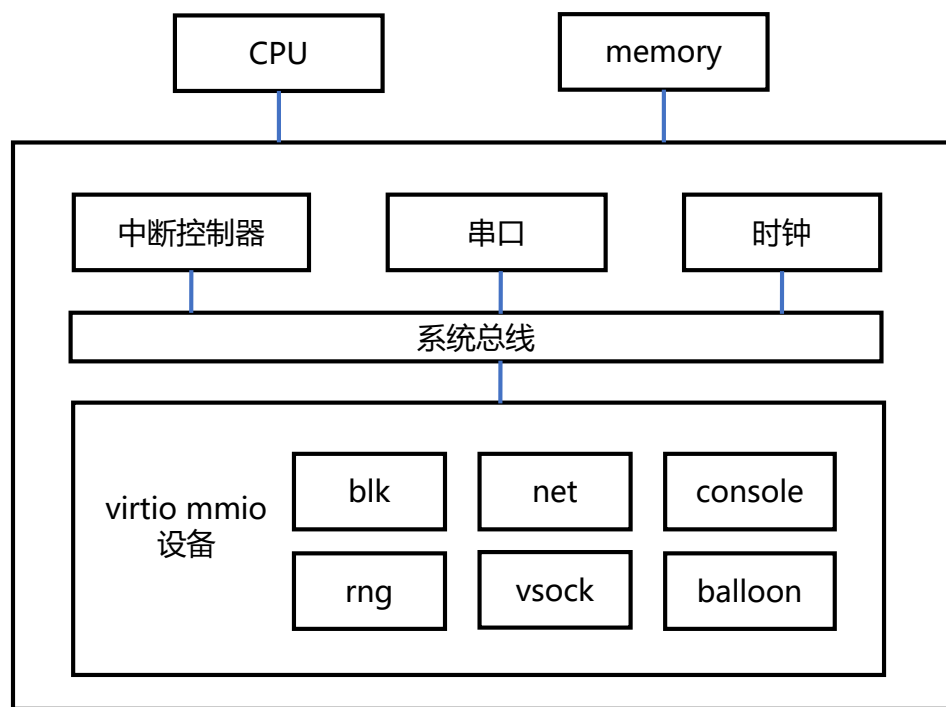
#### •StratoVirt虚拟机是Linux中一个独立的进程。

- 主线程：主线程是异步收集和处理来自外部模块事件的循环
- VCPU线程：每个VCPU都有一个线程处理本VCPU的trap事件
- I/O线程：可以为I/O设备配置iothread提升I/O性能

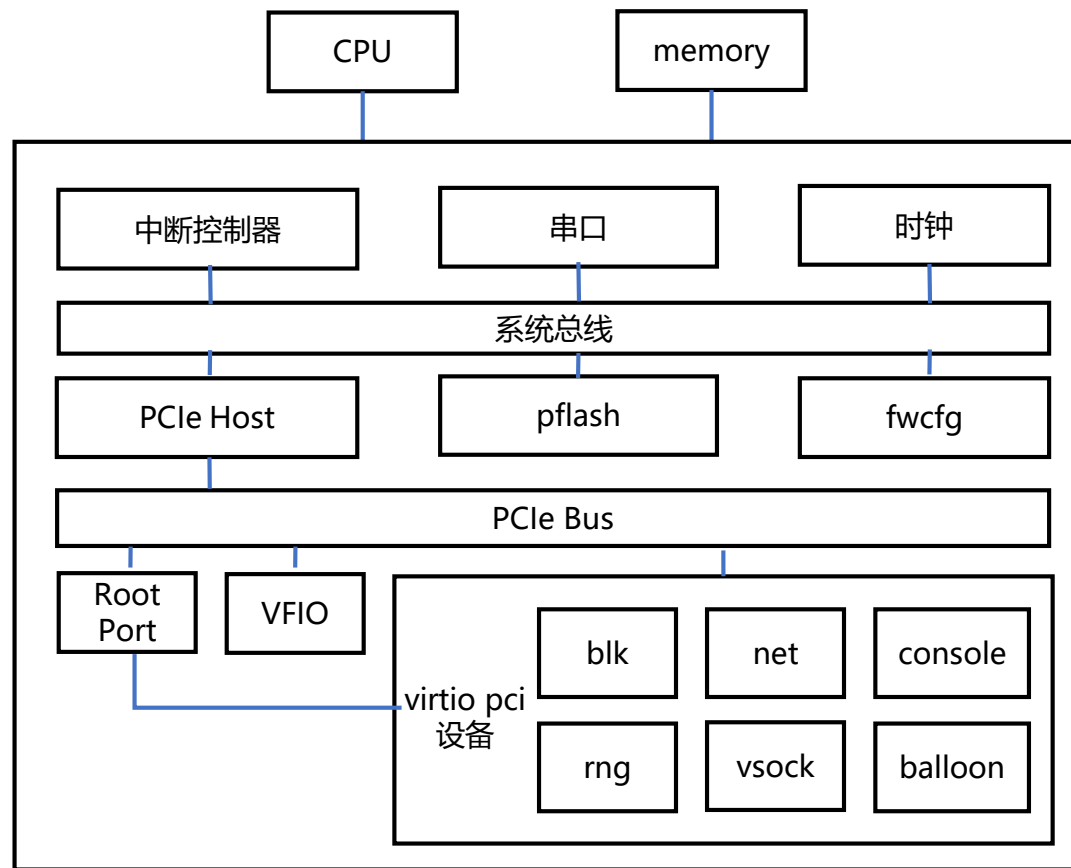


## ► StratoVirt实现的机型

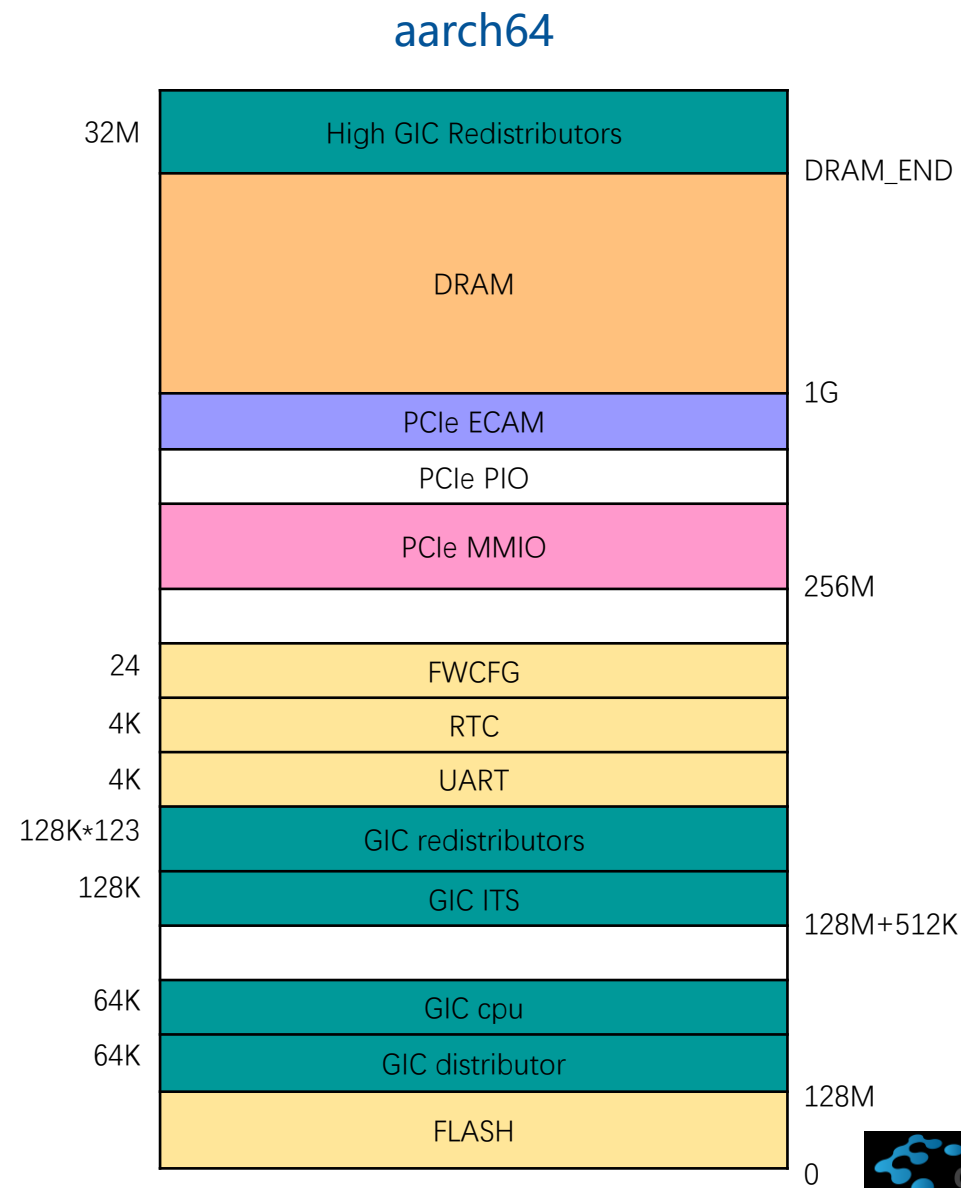
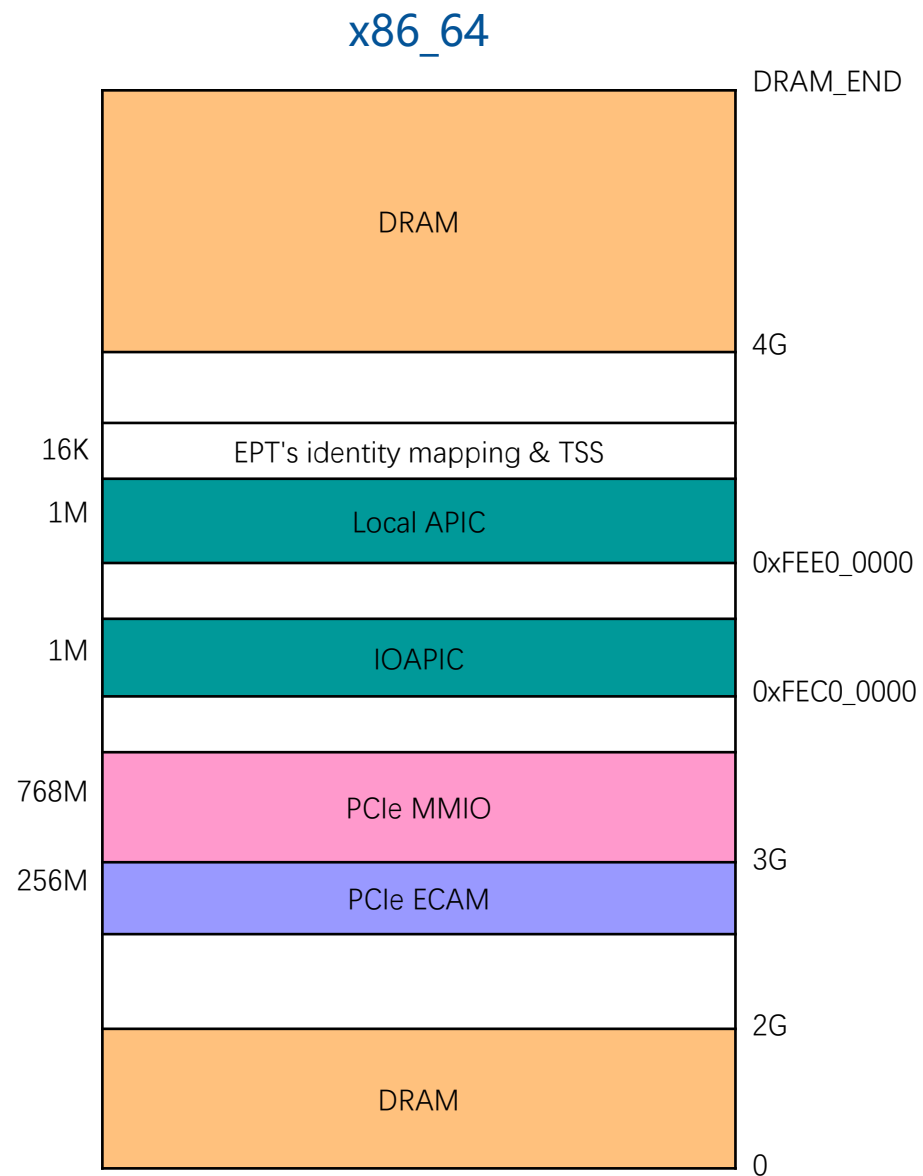
轻量机型



标准机型



# ► StratoVirt标准机型内存布局





02

# CPU子系统

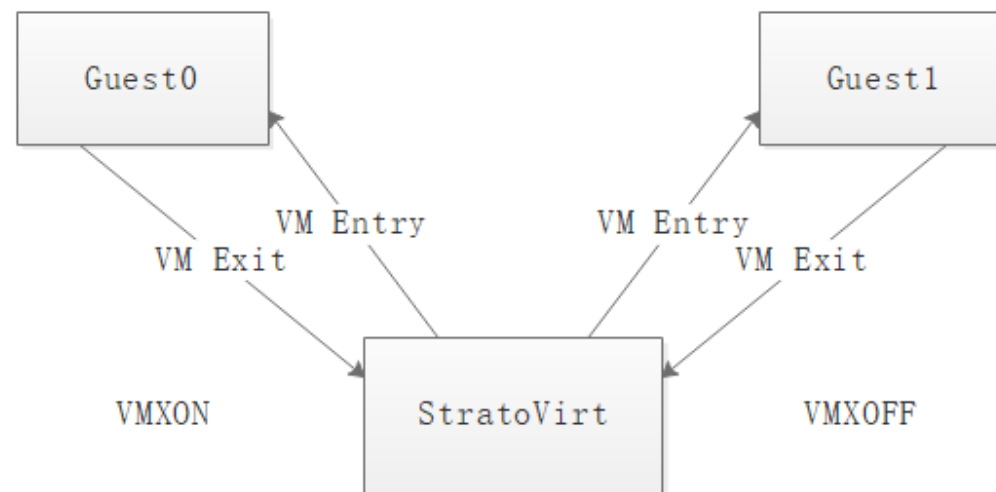
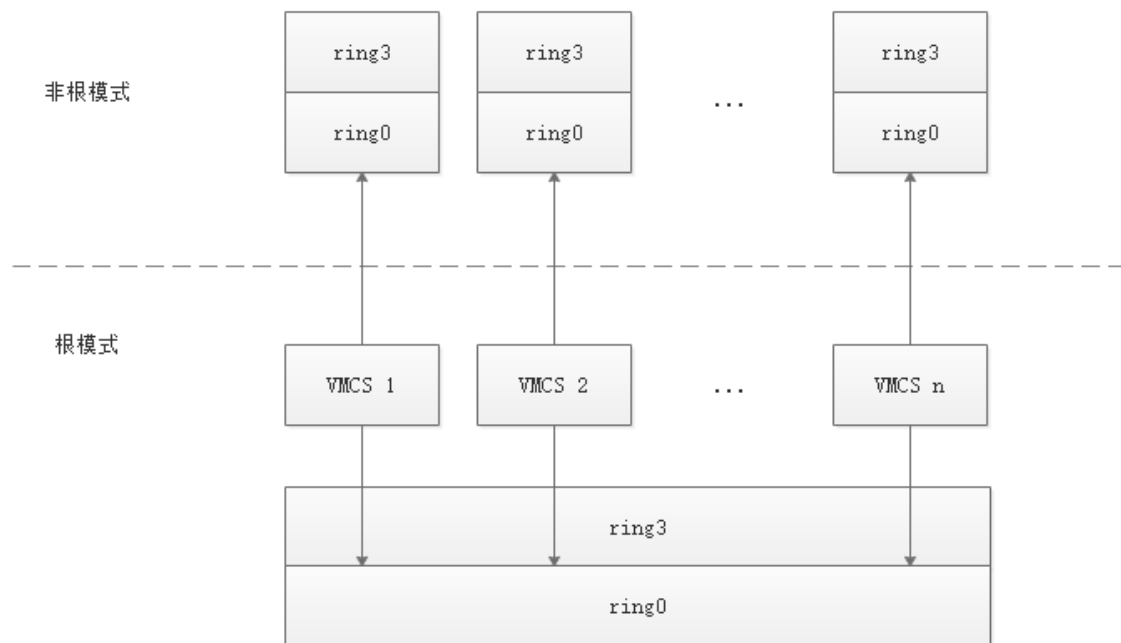
- 硬件辅助虚拟化的概念
- CPU处理退出事件代码
- StratoVirt中的CPU模型





## ► 硬件辅助虚拟化

StratoVirt虚拟机的CPU子系统依赖于硬件辅助虚拟化的能力(如vt-x):



特权指令(如IO指令)不能通过这种方式执行，还是会强制将CPU退出到根模式下交给VMM程序 (StratoVirt) 处理，处理完再重新进入到非根模式执行下一条指令。

## ► 处理退出事件

StratoVirt关于退出事件的处理主要在KVM\_vcpu\_exec函数中。

```
match self.fd.run() {
  Ok(run) => match run {
    #[cfg(target_arch = "x86_64")]
    VcpuExit::IoIn(addr, data) => {           // 该分支处理vCPU的端口读(IoIn)操作
      self.vm.pio_in(u64::from(addr), data); // 将端口读操作转发到虚拟机的pio_in函数
                                              // 上,内部通过端口地址搜索对应的端口设
                                              // 备,并调用设备的read接口
    }
    #[cfg(target_arch = "x86_64")]
    VcpuExit::IoOut(addr, data) => {          // 该分支处理vCPU端口写(IoOut)操作
      self.vm.pio_out(u64::from(addr), data); // 将端口写操作转发到虚拟机的pio_out函
                                              // 数上,内部通过端口地址搜索对应的端口设
                                              // 备,并调用设备的write端口
    }
    VcpuExit::MmioRead(addr, data) => {       // 该分支处理vCPU MMIO地址读
                                              // (MmioRead)操作。
      self.vm.mmio_read(addr, data);
    }
    VcpuExit::MmioWrite(addr, data) => {      // 该分支处理vCPU MMIO地址写
                                              // (MmioWrite)操作。
      self.vm.mmio_write(addr, data);
    }
  }
}
```


## ► StratoVirt中的CPU模型

每个vCPU都有单独的一个vCPU线程用来处理退出事件。

```
/// `CPU` is a wrapper around creating and using a kvm-based VCPU.
pub struct CPU {
    /// ID of this virtual CPU, `0` means this cpu is primary `CPU`.
    id: u8,
    /// The file descriptor of this kvm-based VCPU.
    fd: Arc<VcpuFd>,
    /// Architecture special CPU property.
    arch_cpu: Arc<Mutex<ArchCPU>>,
    /// LifeCycle state of kvm-based VCPU.
    state: Arc<(Mutex<CpuLifecycleState>, Condvar)>,
    /// Works need to handled by this VCPU.
    work_queue: Arc<(Mutex<u64>, Condvar)>,
    /// The thread handler of this virtual CPU.
    task: Arc<Mutex<Option<thread::JoinHandle<()>>>>,
    /// The thread tid of this VCPU.
    tid: Arc<Mutex<Option<u64>>>,
    /// The VM combined by this VCPU.
    vm: Arc<Box<Arc<dyn MachineInterface + Send + Sync>>>,
}
```

```
/// A wrapper around creating and using a kvm-based micro VM.
pub struct LightMachine {
    /// KVM VM file descriptor, represent VM entry in kvm module.
    vm_fd: Arc<VmFd>,
    /// `vCPU` devices.
    cpus: Arc<Mutex<Vec<Arc<CPU>>>>,
    ...
}
```

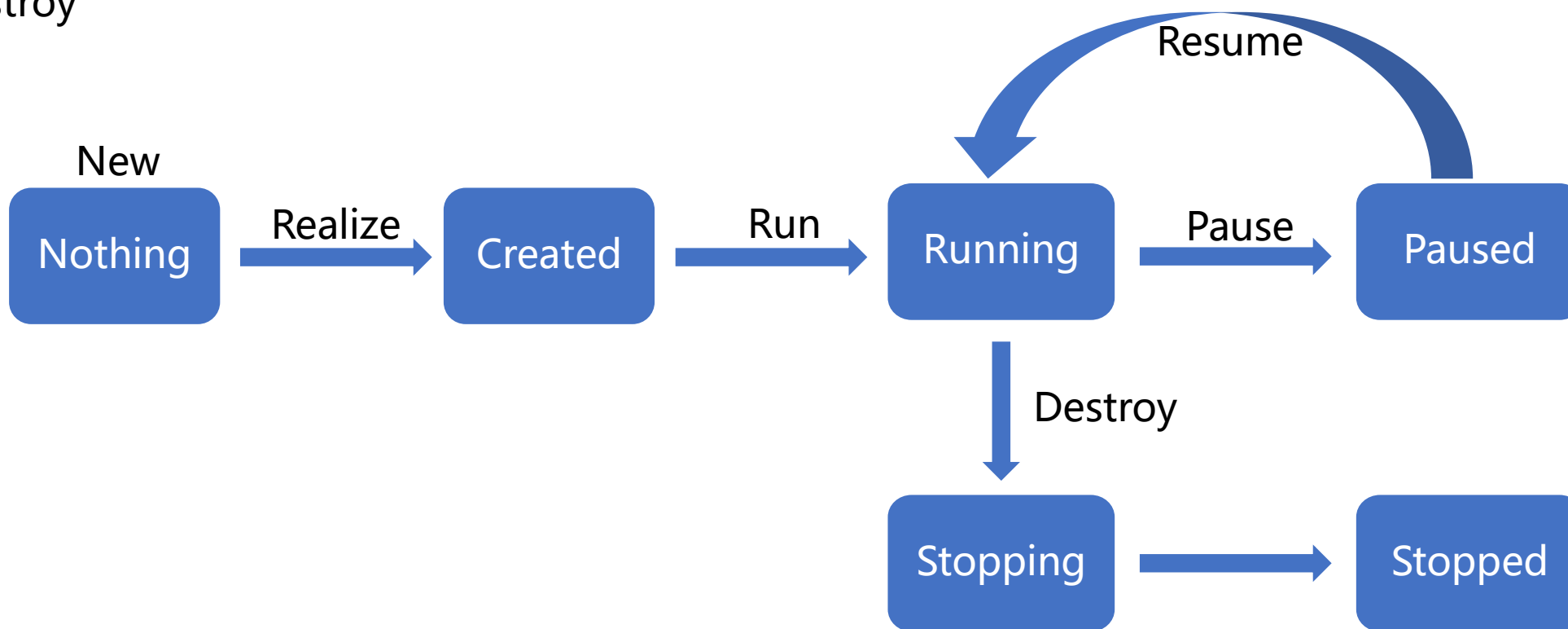
```
impl MachineInterface for LightMachine {}
```



Q: 为什么不直接把  
LightMachine放进  
CPU结构里?

## ► 控制vCPU的生命周期

New  
Realize  
Run  
Pause and Resume  
Destroy



## ► 创建vCPU(new)

vCPU的创建过程包含两个部分，一是vcpu\_fd的创建，二是CPU结构体的创建。

```
...
                                // 从vm_config中读取vcpu的个数
let nrcpus = vm_config.machine_config.nr_cpus;
let mut vcpu_fds = vec![];      // 创建存放vcpu句柄的向量
for cpu_id in 0..nrcpus {
                                // 根据cpu_id创建vcpu句柄放入向量中
    vcpu_fds.push(Arc::new(vm_fd.create_vcpu(cpu_id)?));
}
...
                                // 循环创建nrcpus个vcpu
                                // 依据架构不同创建不同的arch_cpu
#[cfg(target_arch = "aarch64")]
let arch_cpu = ArchCPU::new(&vm_fd, u32::from(vcpu_id));

#[cfg(target_arch = "x86_64")]
let arch_cpu = ArchCPU::new(&vm_fd, u32::from(vcpu_id), u32::from(nrcpus));

let cpu = CPU::new(              // 通过CPU的new函数创建一个CPU
    vcpu_fds[vcpu_id as usize].clone(),
    vcpu_id,
    Arc::new(Mutex::new(arch_cpu)),
    cpu_vm.clone(),
)?;

let mut vcpus = vm.cpus.lock().unwrap();
let newcpu = Arc::new(cpu);      // 将创建好的cpu放置在Arc中，在主线程和vCPU线程间共享
vcpus.push(newcpu.clone());      // 将cpu添加至vm结构的向量集中
}
```

## ► 初始化vCPU寄存器信息(realize)

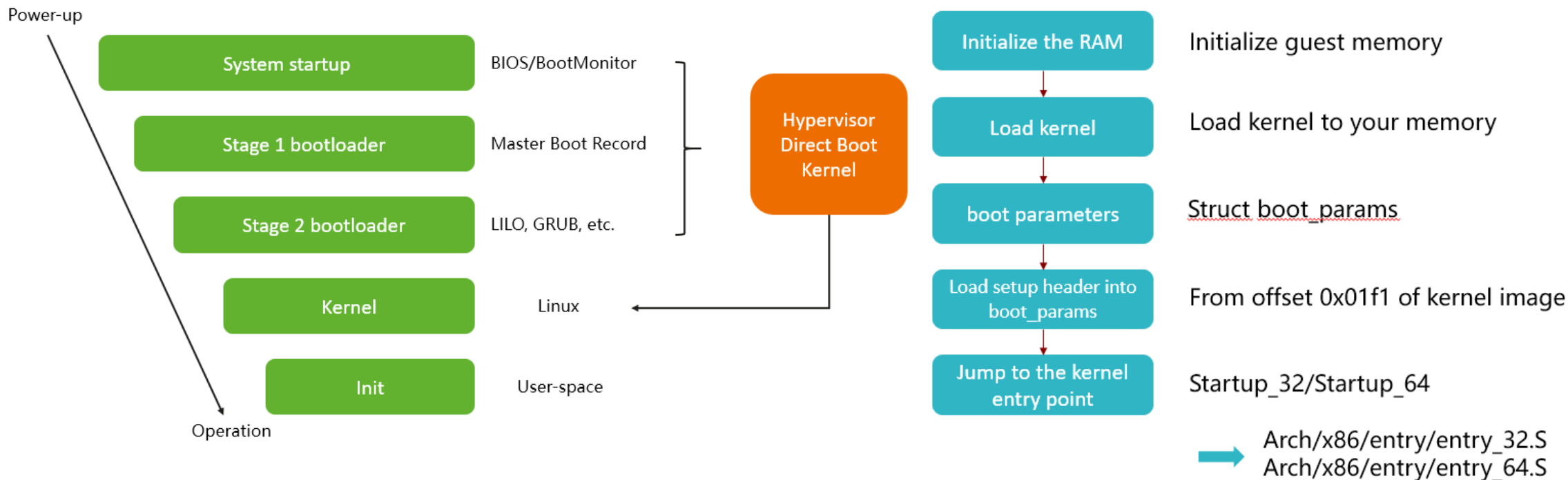
vCPU的寄存器信息与架构强相关，在不同的架构(x86\_64和aarch64)下，vCPU中的寄存器是不一样的，这部分与架构强相关的寄存器信息被存放在CPU结构中的ArchCPU下。

当boot\_loader将vmlinux内核读入内存，会根据架构不同生成不同的BootLoader结构记录启动所必须的信息，CPU模块根据该信息会生成相应的CPUBootConfig:

```
/// AArch64 CPU booting configure information
pub struct X86CPUBootConfig {
    /// Register %rip value
    pub boot_ip: u64,
    /// Register %rsp value
    pub boot_sp: u64,
    /// zero page address, as the second parameter of __startup_64
    /// arch/x86/kernel/head_64.S:86
    pub zero_page: u64,
    pub code_segment: kvm_segment,
    pub data_segment: kvm_segment,
    pub gdt_base: u64,
    pub gdt_size: u16,
    pub idt_base: u64,
    pub idt_size: u16,
    pub pml4_start: u64,
}
```

```
/// AArch64 CPU booting configure information
///
/// Before jumping into the kernel, primary CPU general-purpose
/// register `x0` need to setting to physical address of device
/// tree blob (dtb) in system RAM.
#[derive(Default, Copy, Clone)]
pub struct AArch64CPUBootConfig {
    pub fdt_addr: u64,
    pub kernel_addr: u64,
}
```

## ► 初始化vCPU寄存器信息(realize)



boot\_loader跳过bios和grub直接启动内核

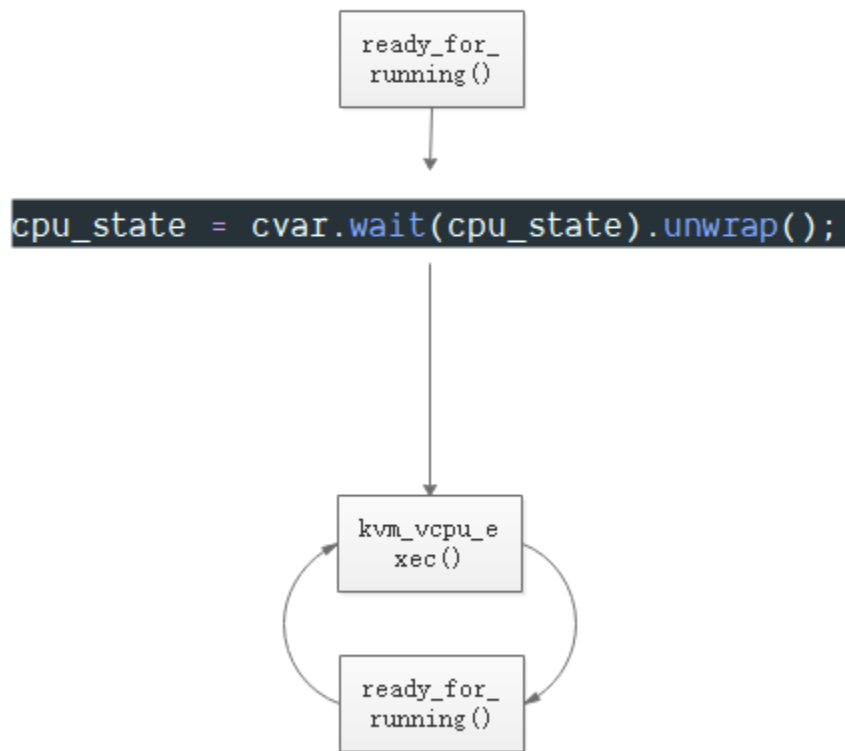
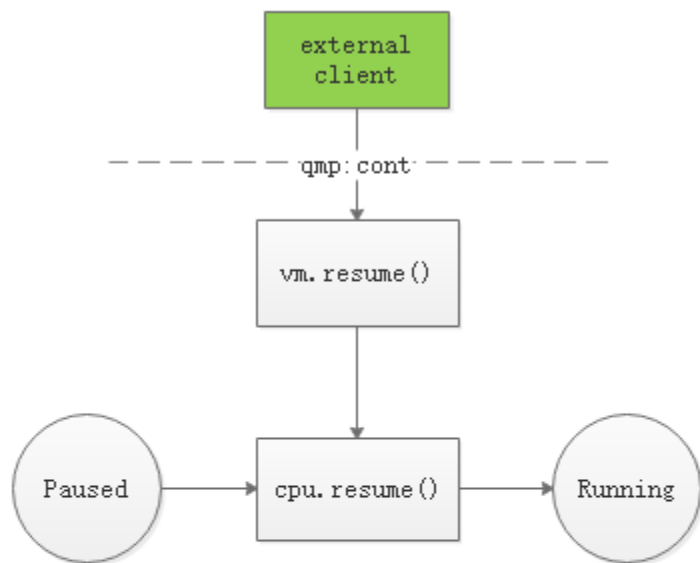
X86 linux 启动协议



```
/// The start address for `kernel image`, `initrd image` and `dtb` in guest memory.
pub struct AArch64BootLoader {
    /// Start address for `kernel image` in guest memory.
    pub kernel_start: u64,
    /// Start address for `initrd image` in guest memory.
    pub initrd_start: u64,
    /// Start address for `dtb` in guest memory.
    pub dtb_start: u64,
}
```

## ► 暂停和恢复vCPU(pause and resume)

StratoVirt的vCPU主要通过线程信号量来实现虚拟机的暂停和恢复。





## ► 停止vCPU运行(destroy)

StratoVirt的vCPU正常生命周期的destroy有两种情况：

- Guest 内部正常关机

Guest内部正常关机后，vCPU会在事件处理中得到一个Shutdown事件：

```
VcpuExit::Shutdown => {  
    info!("Vcpu{} Received an KVM_EXIT_SHUTDOWN signal", self.id());  
    let (cpu_state, _) = &*self.state;  
    *cpu_state.lock().unwrap() = CpuLifecycleState::Stopped;  
    self.vm.destroy();  
    ...  
    return Ok(false);  
}
```

- 通过外部qmp接口直接让vmm执行destroy函数

两种方式最终都是调用每个CPU的destroy函数，让CPU的状态如下转换：



正常关机后，所有的CPU都会处于Stopped状态。



03

# 内存子系统



## ► 内存虚拟化基本原理

### 需要硬件辅助虚拟化支持

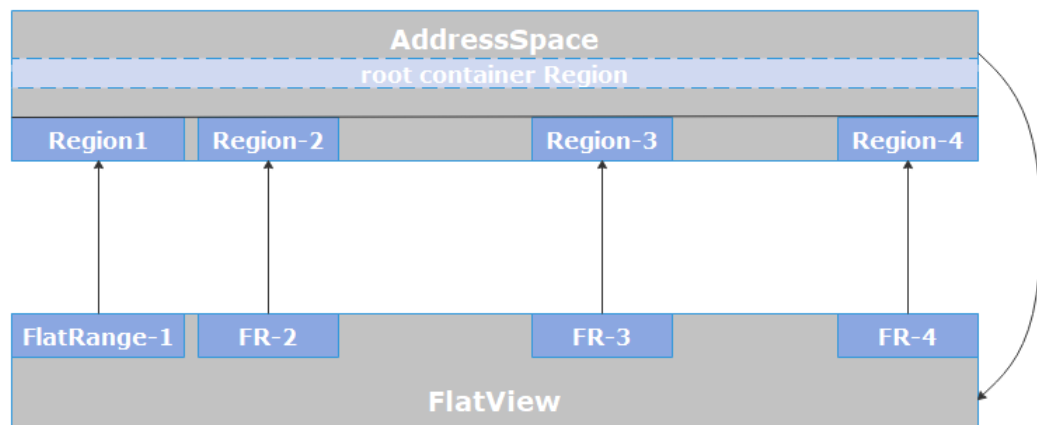
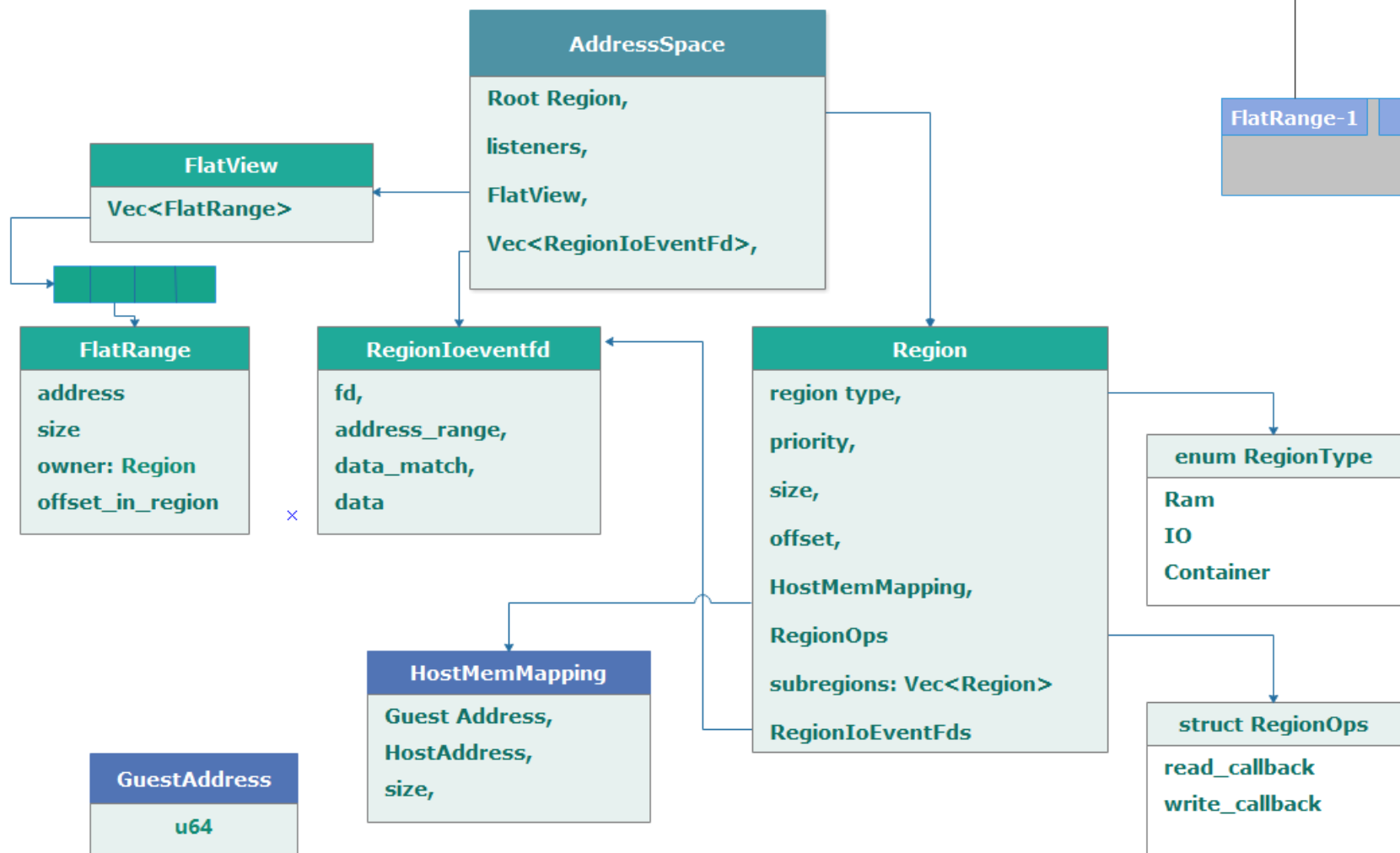
- X86: EPT页表
- Arm: Stage1 & 2 地址翻译
  - Stage-1: GVA -> IPA
  - Stage-2: IPA -> HPA



# ► 地址管理

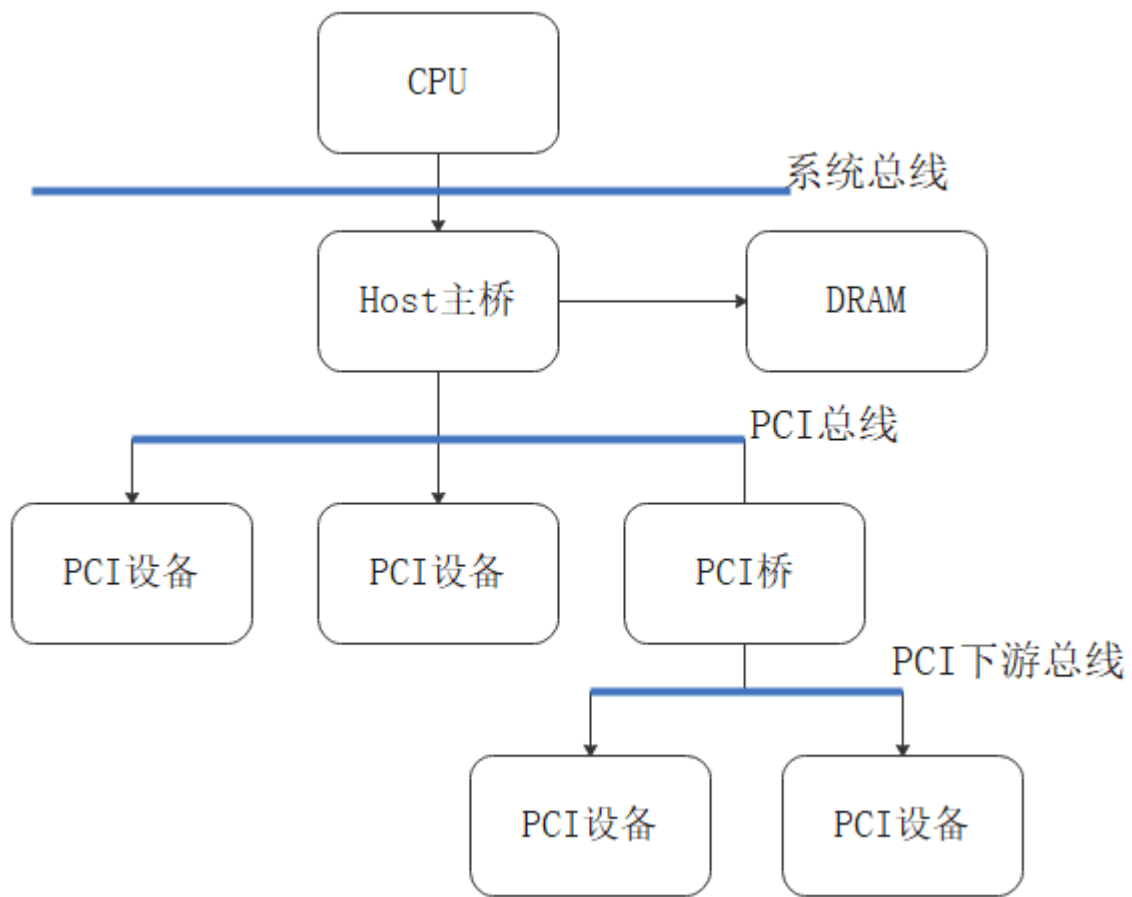
## 1. AddressSpace

整个内存空间



## ► 地址空间拓扑结构

- 模拟 标准虚拟化下，不同总线有不同的 地址空间  
需要更灵活的内存地址管理！



# ► 地址管理

## 1. AddressSpace

整个内存空间

## 2. Region

占有内存空间中一段地址资源，供设备/Ram使用

Region类型:

- IO
- Ram
- Container

## 3. RegionIoEventFd

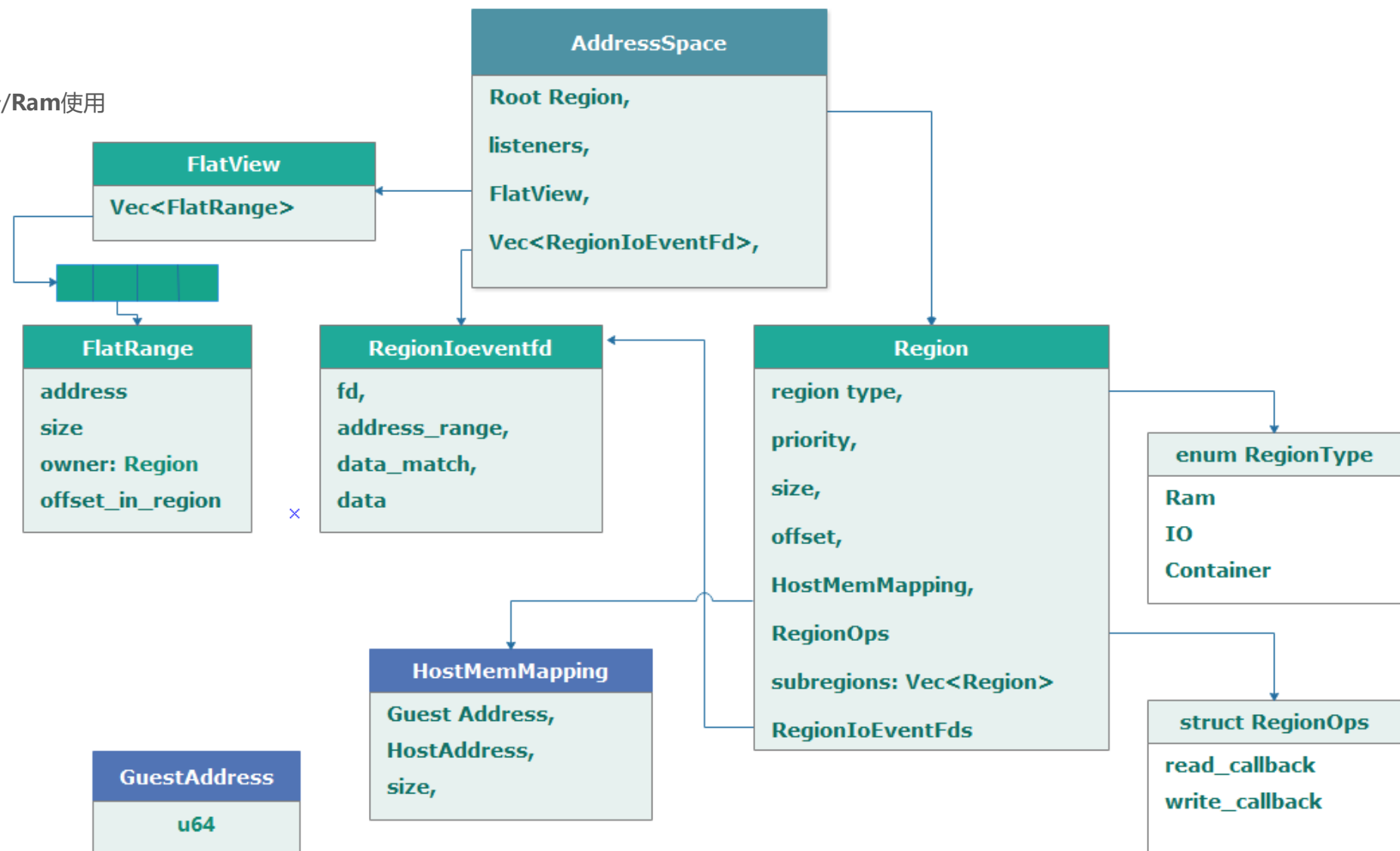
Region包含的EventFd信息

## 4. RegionOps

Region对应的读写方法

## 5. HostMemMapping

提供给物理机使用的物理内存

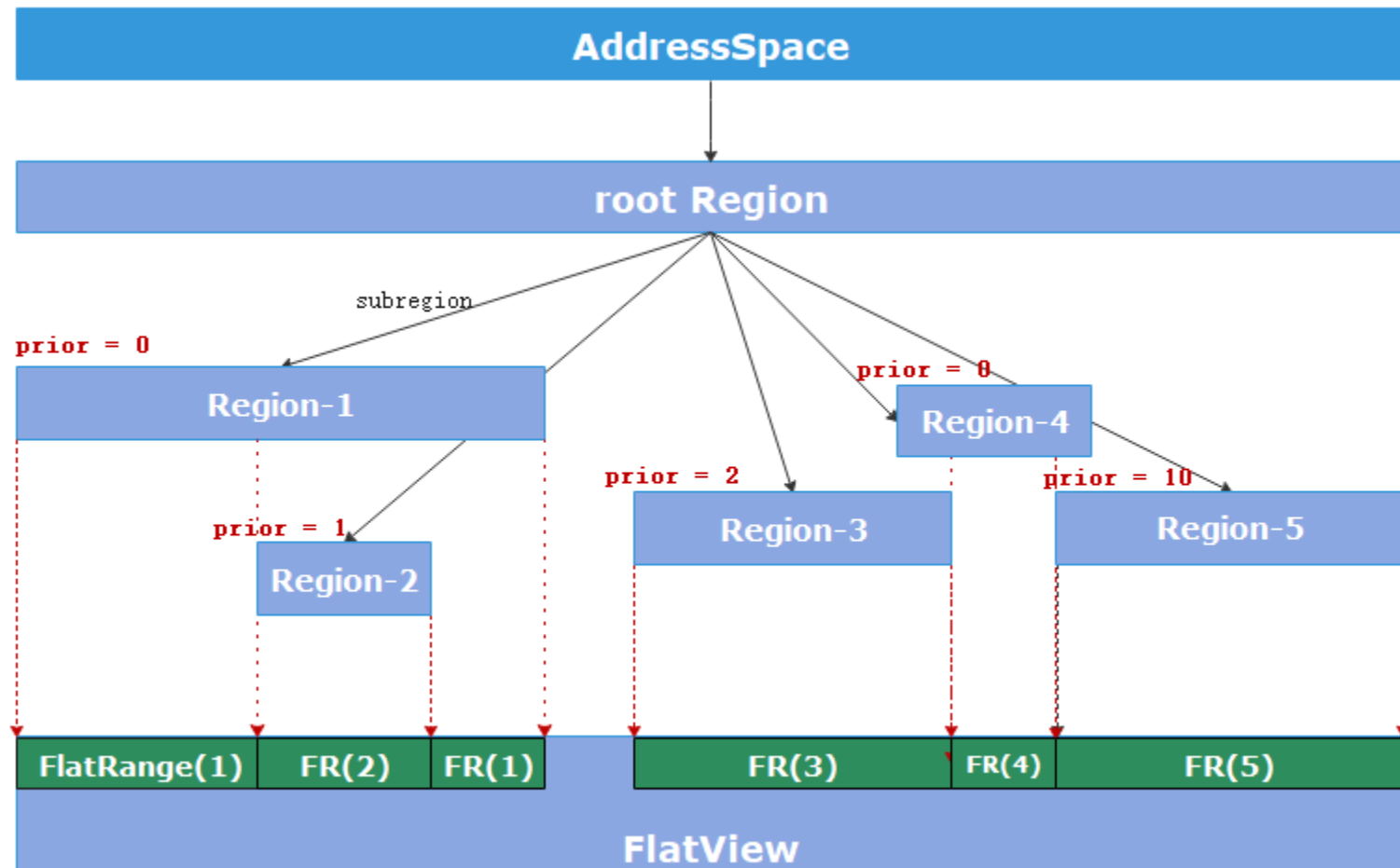


## ► 地址空间拓扑结构

- AddressSpace 树状拓扑结构
- FlatView 表示 平坦视图

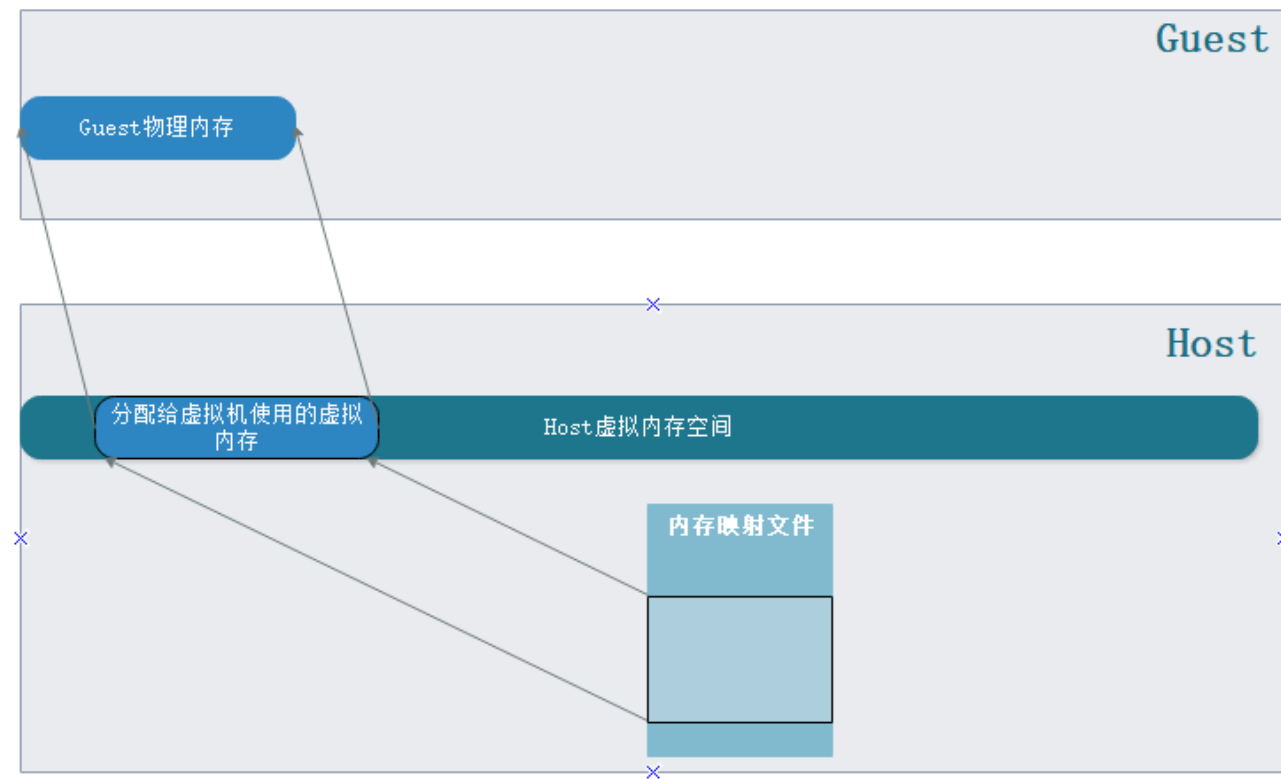
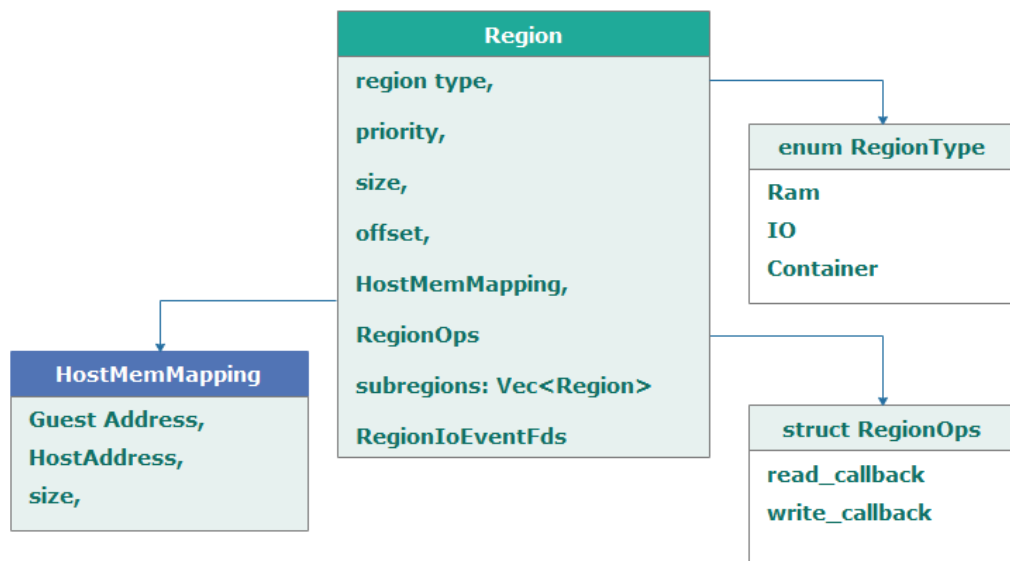
平坦视图的作用：

- 作用：地址空间读写
- 生成：Region 优先级 会影响  
最终 AddressSpace 平坦视图生成



## ► Ram Region: 内存映射

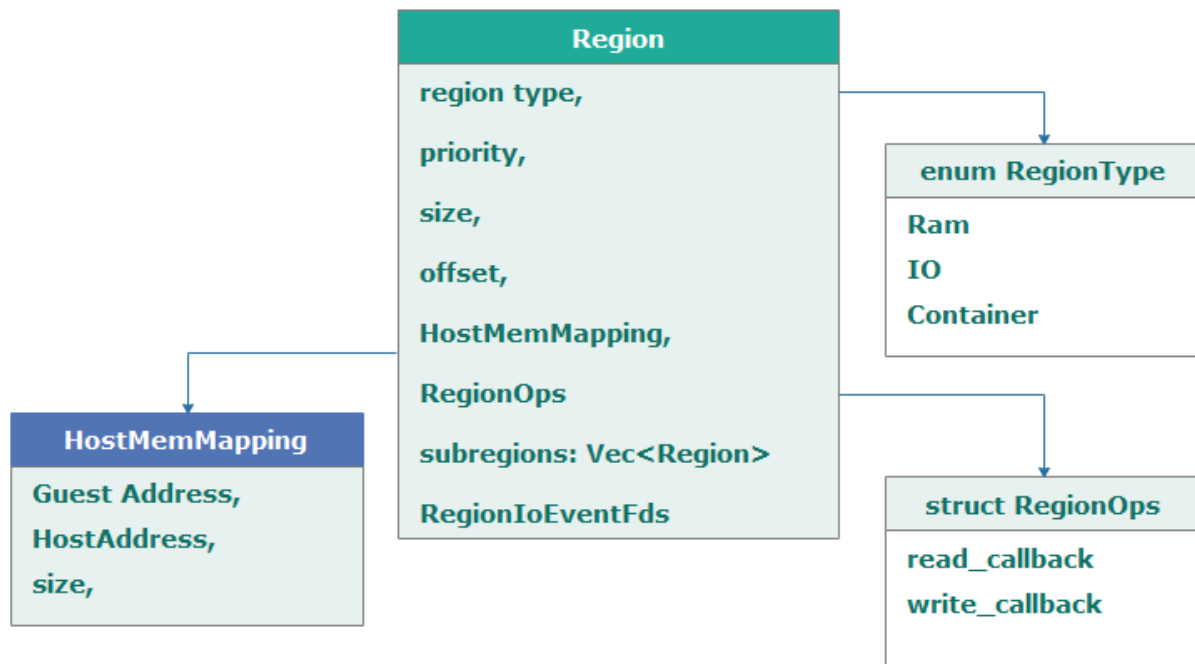
- 内存映射
  - 匿名映射
  - 文件映射





## ► 设备IO Region

- 设备 IO Region
  - 提供 线程安全的读写方法 -> RegionOps
  - 初始化Region
  - 将Region添加到 IO地址空间或者 MMIO地址空间



## ► 设备IoEventFd

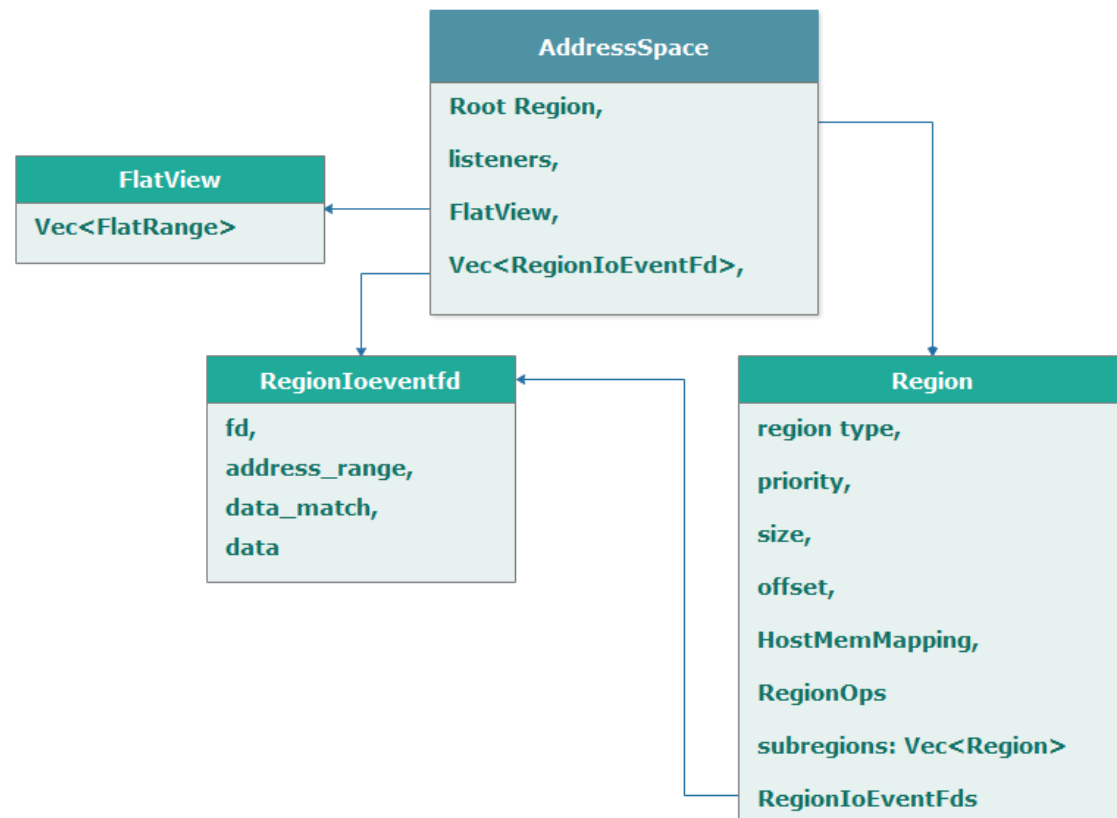
- 设备 IoEventFd

- 初始化 Region后， 设置Region中的 IoEventFd
- Region添加到地址空间 AddressSpace 中

```
impl MmioDevice {  
    pub fn realize(  
        &self,  
        vm_fd: &VmFd,  
        ...  
    ) -> Result<()> {  
        ...  
        let region = Region::init_io_region(  
            self.resource.size,  
            self.region_ops.clone()  
        );  
        region.set_ioeventfds(  
            &self.device.lock().unwrap().ioeventfds()  
        );  
        ...  
    }  
}
```

- AddressSpace 内存对 IoEventFd 的管理

- AddressSpace 中保存当前的 IoEventFds
- AddressSpace 拓扑发生变化时，更新当前的 IoEventFds
- 注册到 KVM / 从 KVM中 解注册



## ► 地址空间注册接口

StratoVirt 轻量虚拟化下，主板

LightMachine 结构体中包含：

- 内存总线 对应的地址空间
- IO总线 对应的地址空间
- MMIO Bus

```
pub struct LightMachine {  
    ...  
    /// Memory address space.  
    sys_mem: Arc<AddressSpace>,  
    /// IO address space.  
    #[cfg(target_arch = "x86_64")]  
    sys_io: Arc<AddressSpace>,  
    /// Mmio bus.  
    bus: Bus,  
    ...  
}
```

LightMachine 初始化：

1. 创建KVM对象，并创建虚拟机
2. AddressSpace 初始化

```
impl LightMachine {  
    fn new() {  
        ...  
  
        let sys_mem = AddressSpace::new(Region::init_container_region(u64::max_value()))?;  
        let nr_slots = kvm.get_nr_memslots();  
        sys_mem.register_listener(Box::new(KvmMemoryListener::new(  
            nr_slots as u32,  
            vm_fd.clone(),  
        )))?;  
  
        #[cfg(target_arch = "x86_64")]  
        let sys_io = AddressSpace::new(Region::init_container_region(1 << 16))?;  
        #[cfg(target_arch = "x86_64")]  
        sys_io.register_listener(Box::new(KvmIoListener::new(vm_fd.clone())))?;  
        ...  
    }  
}
```

## ► 地址空间注册接口

### LightMachine 初始化:

#### 3. 内存初始化

如果 未配置 内存大小, 默认为 128 M

```
// 内存初始化
impl LightMachine {
    fn new() {
        ...
        // Init guest-memory
        // Define ram-region ranges according to architectures
        let ram_ranges = Self::arch_ram_ranges(mem_size);
        let mem_mappings = create_host_mmmaps(&ram_ranges, ...)?;

        for mmap in mem_mappings.iter() {
            sys_mem.root().add_subregion(
                Region::init_ram_region(mmap.clone()),
                mmap.start_address().raw_value(),
            );
        }
        ...
    }
}
```

### LightMachine 初始化:

#### 4. CPU初始化 (e820表 创建)

#### 5. 设备初始化

```
// device 已创建好, 含有 read/write 成员方法
let dev = Arc::new(Mutex::new(device));

let dev_clone = dev.clone();
let read_ops = move |data: &mut [u8], _: GuestAddress, off: u64| -> bool {
    let mut dev_locked = dev_clone.lock().unwrap();
    dev_locked.read(data, off)
};

let dev_clone = dev.clone();
let write_ops = move |data: &[u8], _: GuestAddress, off: u64| -> bool {
    let mut dev_locked = dev_clone.lock().unwrap();
    dev_locked.write(data, off)
};

let dev_ops = RegionOps {
    read: Arc::new(read_ops),
    write: Arc::new(write_ops),
};

let io_region = Region::init_io_region(0x1000, dev_ops);
space.root().add_subregion(io_region, 0x2000);
```

## ► 地址空间读写接口

Vcpu退出，如果退出原因是 内存读写。 则交由 AddressSpace 处理

```
impl CPUInterface for CPU {
    fn kvm_vcpu_exec(&self) -> Result<bool> {
        match self.fd.run() {
            Ok(run) => match run {
                #[cfg(target_arch = "x86_64")]
                VcpuExit::IoIn(addr, data) => {
                    self.vm.pio_in(u64::from(addr), data);
                }
                #[cfg(target_arch = "x86_64")]
                VcpuExit::IoOut(addr, data) => {
                    self.vm.pio_out(u64::from(addr), data);
                }
                VcpuExit::MmioRead(addr, data) => {
                    self.vm.mmio_read(addr, data);
                }
                VcpuExit::MmioWrite(addr, data) => {
                    self.vm.mmio_write(addr, data);
                }
                ...
            },
            ...
        }
    }
}
```

```
impl MachineAddressInterface for LightMachine {
    #[cfg(target_arch = "x86_64")]
    fn pio_in(&self, addr: u64, mut data: &mut [u8]) -> bool {
        ...
    }

    #[cfg(target_arch = "x86_64")]
    fn pio_out(&self, addr: u64, mut data: &[u8]) -> bool {
        let count = data.len() as u64;
        self.sys_io
            .write(&mut data, GuestAddress(addr), count)
            .is_ok()
    }

    fn mmio_read(&self, addr: u64, mut data: &mut [u8]) -> bool {
        ...
    }

    fn mmio_write(&self, addr: u64, mut data: &[u8]) -> bool {
        let count = data.len() as u64;
        self.sys_mem
            .write(&mut data, GuestAddress(addr), count)
            .is_ok()
    }
}
```

## ► 地址空间读写接口

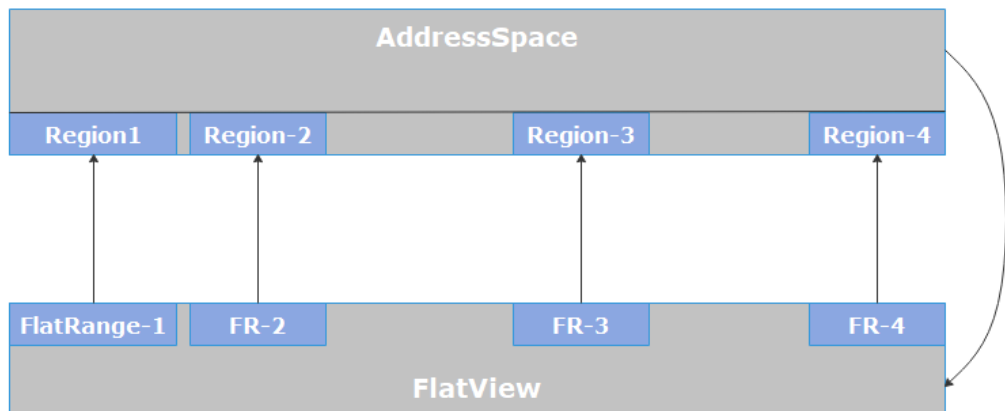
### 读写流程:

1. AddressSpace 中 FlatView 中查找 FlatRange
2. 从找到的 FlatRange 得到对应的 Region
3. 计算Region中的offset:

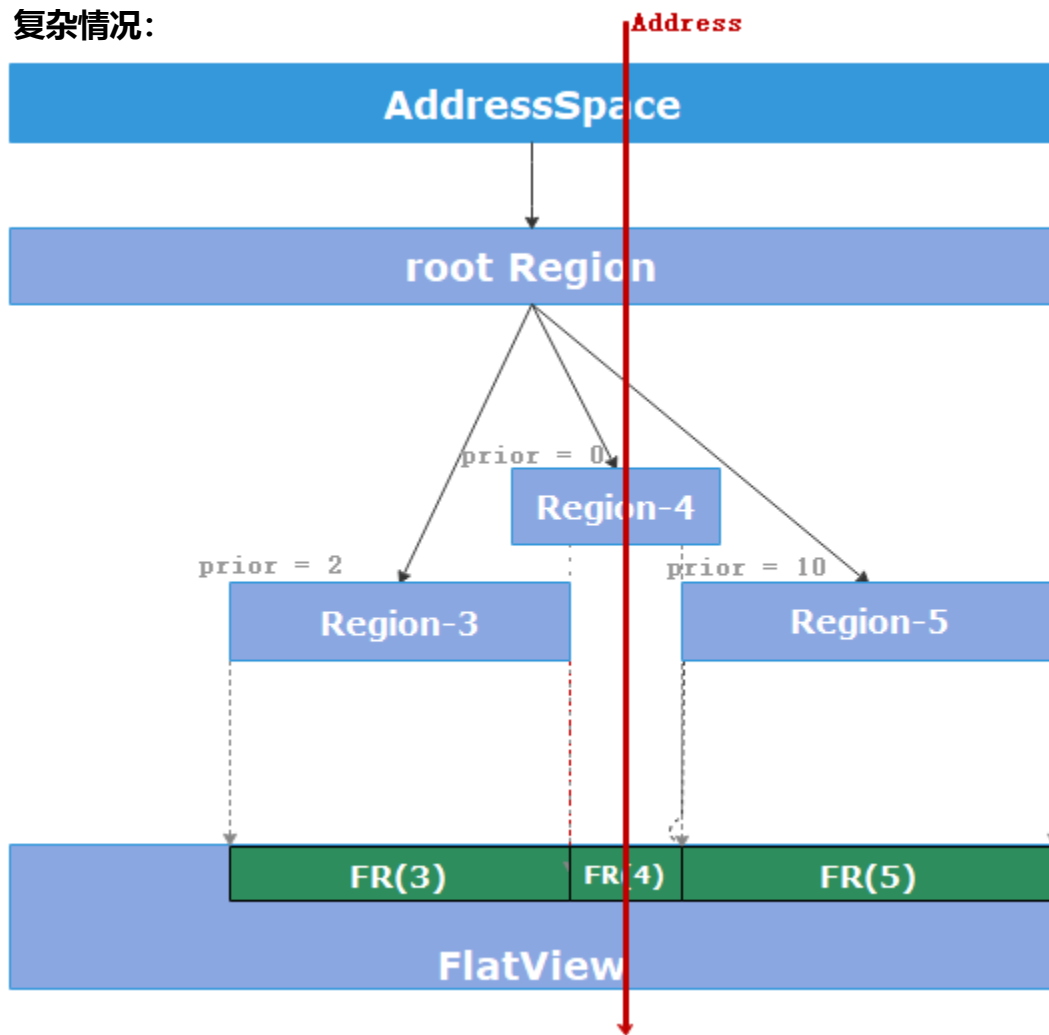
$$\text{address} - \text{FlatRange.base\_addr} \\ + \text{FlatRange.offset\_in\_region}$$

4. 调用Region的读写接口

### 简单情况:



### 复杂情况:

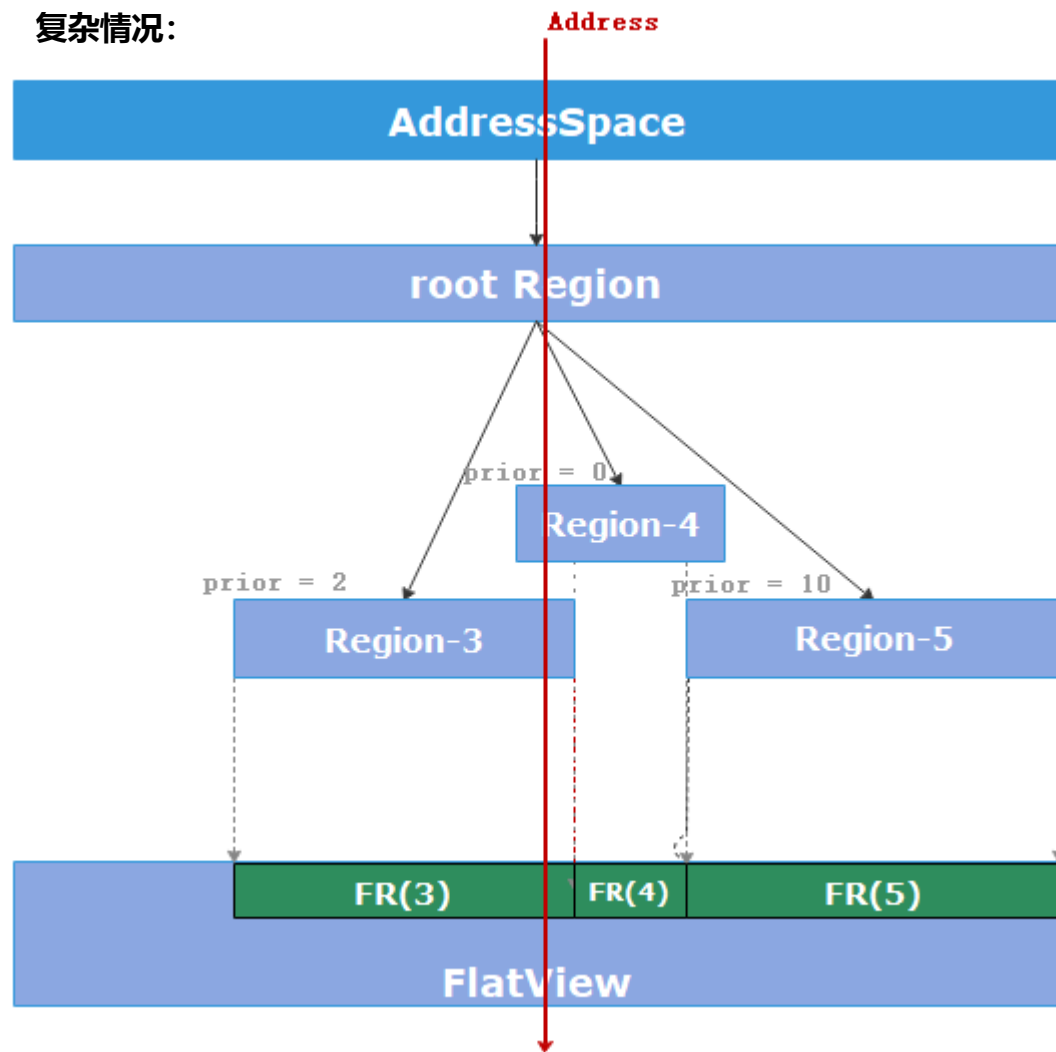


## ► 地址空间读写接口

### 读写流程:

1. AddressSpace 中 FlatView 中查找 FlatRange
2. 从找到的 FlatRange 得到对应的 Region
3. 计算Region中的offset:  
$$\text{address} - \text{FlatRange.base\_addr} + \text{FlatRange.offset\_in\_region}$$
4. 调用Region的读写接口

### 复杂情况:





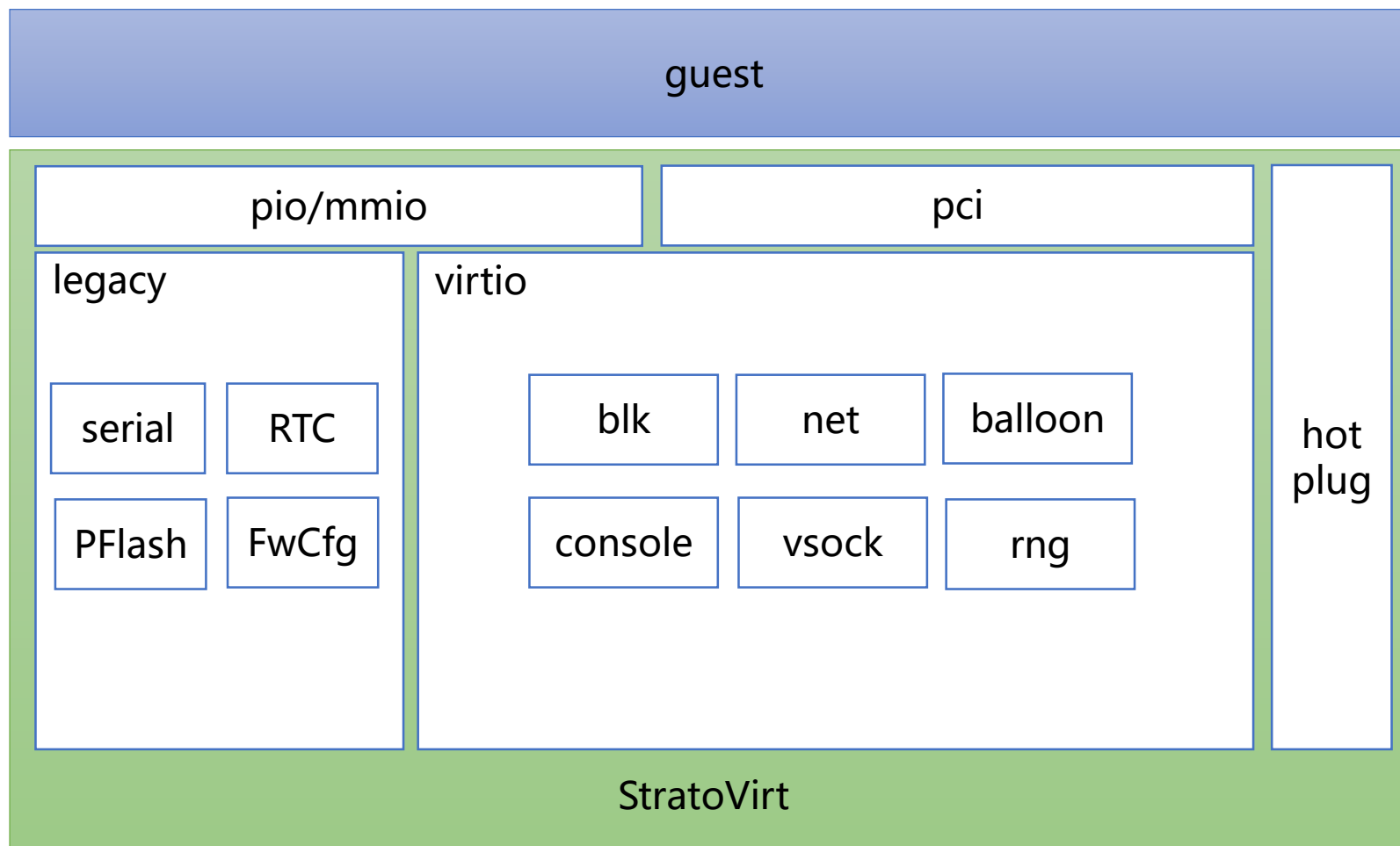
04

# IO 子系统





## ► IO子系统架构



## ► Virtio-背景介绍

### IO虚拟化

1. 全虚拟化：VMM为虚拟机模拟一个与真实设备类似的虚拟I/O设备；

优点：VM的OS不需要为I/O虚拟化做修改；缺点：VMM实时截获I/O请求，性能损耗大；

2. 半虚拟化：建立特权虚拟机，收集转发所有I/O请求，例如XEN；

修改VM的OS，VMM处理I/O造成性能损耗；

3. 硬件辅助虚拟化(主流技术)：I/O设备驱动直接安装在VM的OS，虚拟机直接访问硬件。

Virtio：Linux上设备驱动标准框架。前端(virtio-blk、virtio-net等)是虚拟机驱动模块，后端是在Stratovirt中实现。前后端连接包括Virtio接口实现虚拟队列操作，Vring实现环形缓冲区。

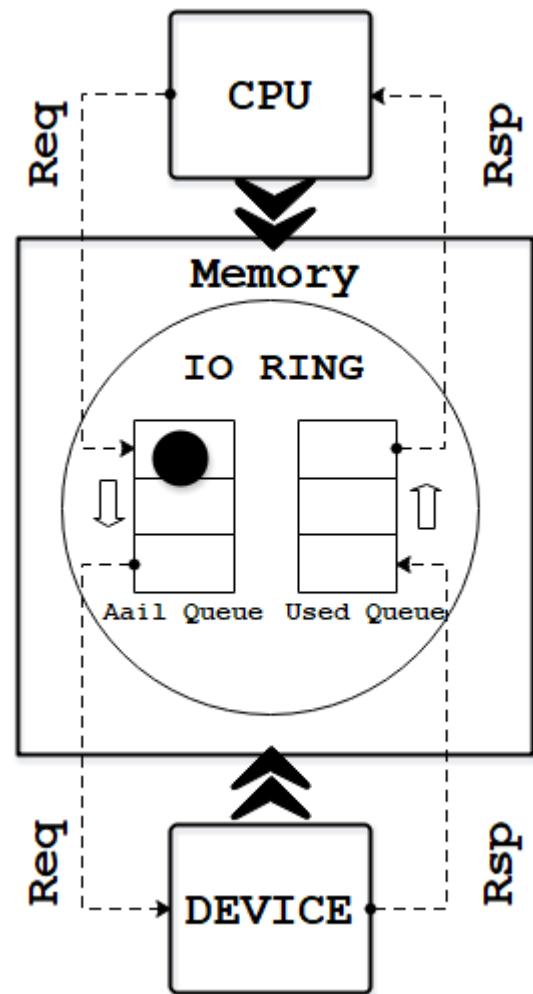
## ► Virtio-IO流程

### 原理抽象

CPU与外设可以共同访问内存；内存中存在一个称为环形队列(IO RING)的数据结构。根据存放对象不同，该队列可分成由IO请求组成的请求队列(Avail Queue)和由IO响应组成的响应队列(Used Queue)

### IO处理过程

- 第一步，应用程序下发IO时，CPU将IO请求放入环形结构(IO RING)的请求队列(Avail Queue)中并通知设备；
- 第二步，设备收到通知后从请求队列中取出IO请求并在内部进行实际处理；
- 第三步，设备将IO处理完成后，将结果作为IO响应放入响应队列(Used Queue)并以中断通知CPU；
- 第四步，CPU从响应队列中取出IO处理结果并返回给应用程序。



## ► Virtio-IO总线协议

virtio协议实现过程中，CPU与外设之间的通知机制以及外设访问内存方式由实际连接CPU与外设的总线协议决定

- Mmio总线
- PCI总线

```

/// Operations for sysbus devices.
pub trait SysBusDevOps: Send + AmlBuilder {
    /// Read function of device.
    fn read(&mut self, data: &mut [u8], base: GuestAddress, offset: u64) -> bool;

    /// Write function of device.
    fn write(&mut self, data: &[u8], base: GuestAddress, offset: u64) -> bool;

    fn ioeventfds(&self, regionIoEventFd) { ...
        self: &Self
    }

    fn interrupt_evt(&self) -> Option<&EventFd> { ...

    fn set_irq(&mut self, sysbus: &mut SysBus) -> Result<i32> { ...

    fn get_sys_resource(&mut self) -> Option<&mut SysRes> { ...

    fn set_sys_resource(
        ...
    ) -> Result<()> { ...

    fn get_type(&self) -> SysBusDevType { ...

    fn reset(&mut self) -> bool { ...
}

```

```
pub trait PciDevOps: Send {
    /// Init writable bit mask.
    fn init_write_mask(&mut self) -> Result<()>;

    /// Init write-and-clear bit mask.
    fn init_write_clear_mask(&mut self) -> Result<()>;

    /// Realize PCI/PCIe device.
    fn realize(self) -> Result<()>;

    /// Configuration space read. ...
    fn read_config(&self, offset: usize, data: &mut [u8]);

    /// Configuration space write. ...
    fn write_config(&mut self, offset: usize, data: &[u8]);

    /// Set device id to send MSI/MSI-X. ...
    fn set_dev_id(&self, bus_num: u8, devfn: u8) -> u16 { ...

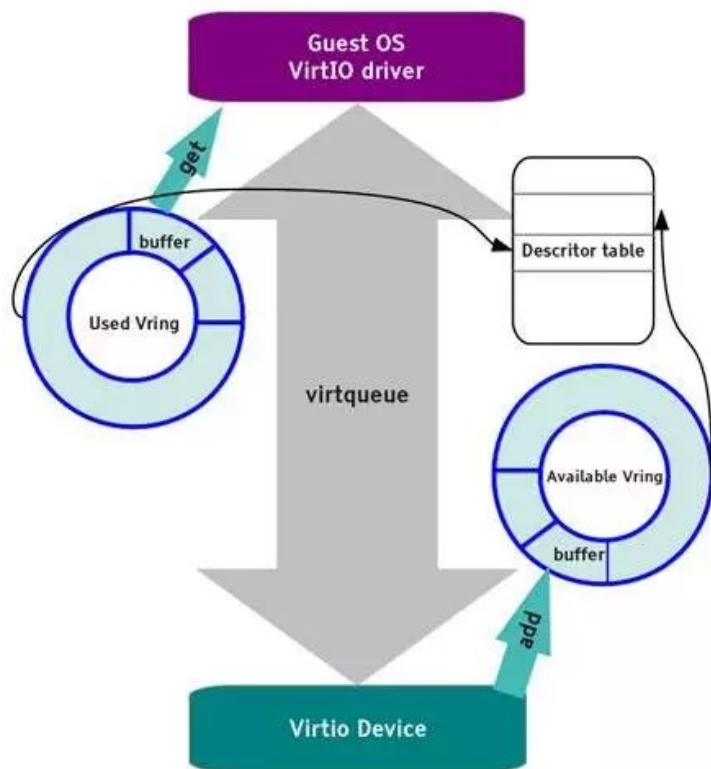
    /// Get device name.
    fn name(&self) -> String;

    /// Get struct beneath the trait.
    fn pci_reset(&mut self) -> bool;
}
```

## ► Virtio-IO队列结构

IO RING由三段连续内存组成:

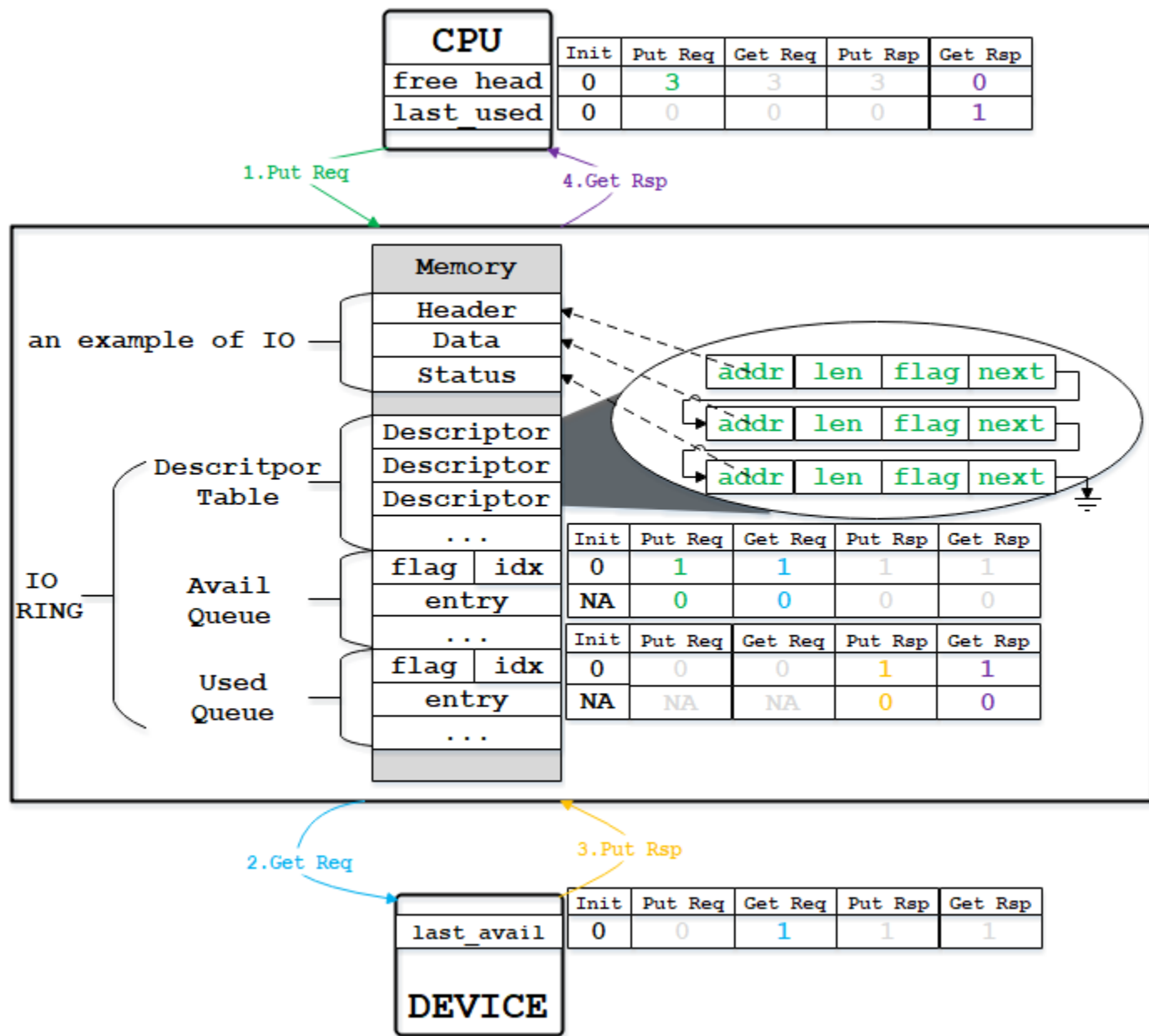
1. Descriptor Table  
描述buffer信息, device只读, driver只写。
2. Avail Queue  
提供Descriptor链index, 查找Descriptor位置。Driver写入, device读取。
3. Used Queue  
一旦device完成I/O操作, Used Ring记录被操作的buffer。  
Device写入, driver读取。



```
pub trait VirtioDevice: Send {  
    /// Realize low level device.  
    fn realize(&mut self) -> Result<()>;  
  
    /// Get the virtio device type, refer to Virtio Spec.  
    fn device_type(&self) -> u32;  
  
    /// Get the count of virtio device queues.  
    fn queue_num(&self) -> usize;  
  
    /// Get the queue size of virtio device.  
    fn queue_size(&self) -> u16;  
  
    /// Get device features from host.  
    fn get_device_features(&self, features_select: u32) -> u32;  
  
    /// Set driver features by guest.  
    fn set_driver_features(&mut self, page: u32, value: u32);  
  
    /// Read data of config from guest.  
    fn read_config(&self, offset: u64, data: &mut [u8]) -> Result<()>;  
  
    /// Write data to config from guest.  
    fn write_config(&mut self, offset: u64, data: &[u8]) -> Result<()>;  
  
    /// Activate the virtio device, this function is called by vcpu thread when frontend...  
    fn activate(...)  
    ) -> Result<()>;  
  
    /// Reset virtio device.  
    fn reset(&mut self) -> Result<()> { ...  
  
    fn restart(&mut self) -> Result<()> { ...  
  
    /// Update the low level config of MMIO device, ...  
    fn update_config(&mut self, _dev_config: Option<Arc<dyn ConfigCheck>>) -> Result<()> { ...  
}
```

## ► Virtio-IO队列操作

- Descriptor Table由固定长度(16字节)的Descriptor组成，其个数等于环形队列(IO RING)长度。
- Avail Queue由头部的flags和idx域及entry数组(entry代表数组元素)组成：entry数组元素用来存放IO请求占用的首个Descriptor在Descriptor Table中的索引，数组长度等于环形队列长度(不开启event\_idx特性)。
- Used Queue由头部的flags和idx域及entry数组(entry代表数组元素)组成：entry数组元素主要用来存放IO响应占用的首个Descriptor在Descriptor Table中的索引，数组长度等于环形队列长度(不开启event\_idx特性)。
- 环形队列结构(IO RING)被CPU和设备同见。仅CPU可见变量为free\_head(空闲Descriptor链表头，初始时所有Descriptor通过next指针依次相连形成空闲链表)和last\_used(当前已取的used元素位置)。仅设备可见变量为last\_avail(当前已取的avail元素位置)。



## ► Virtio-blk设备IO请求处理

### • 第一步，CPU放请求。

假设申请到了前三个Descriptor(free\_head更新为3，表示下一个空闲项从索引3开始，因为0、1、2已被占用)，然后将第一个Descriptor的索引值0填入Ail Queue的第一个entry中，并将idx更新为1，代表放入1个请求

### • 第二步，设备取请求。

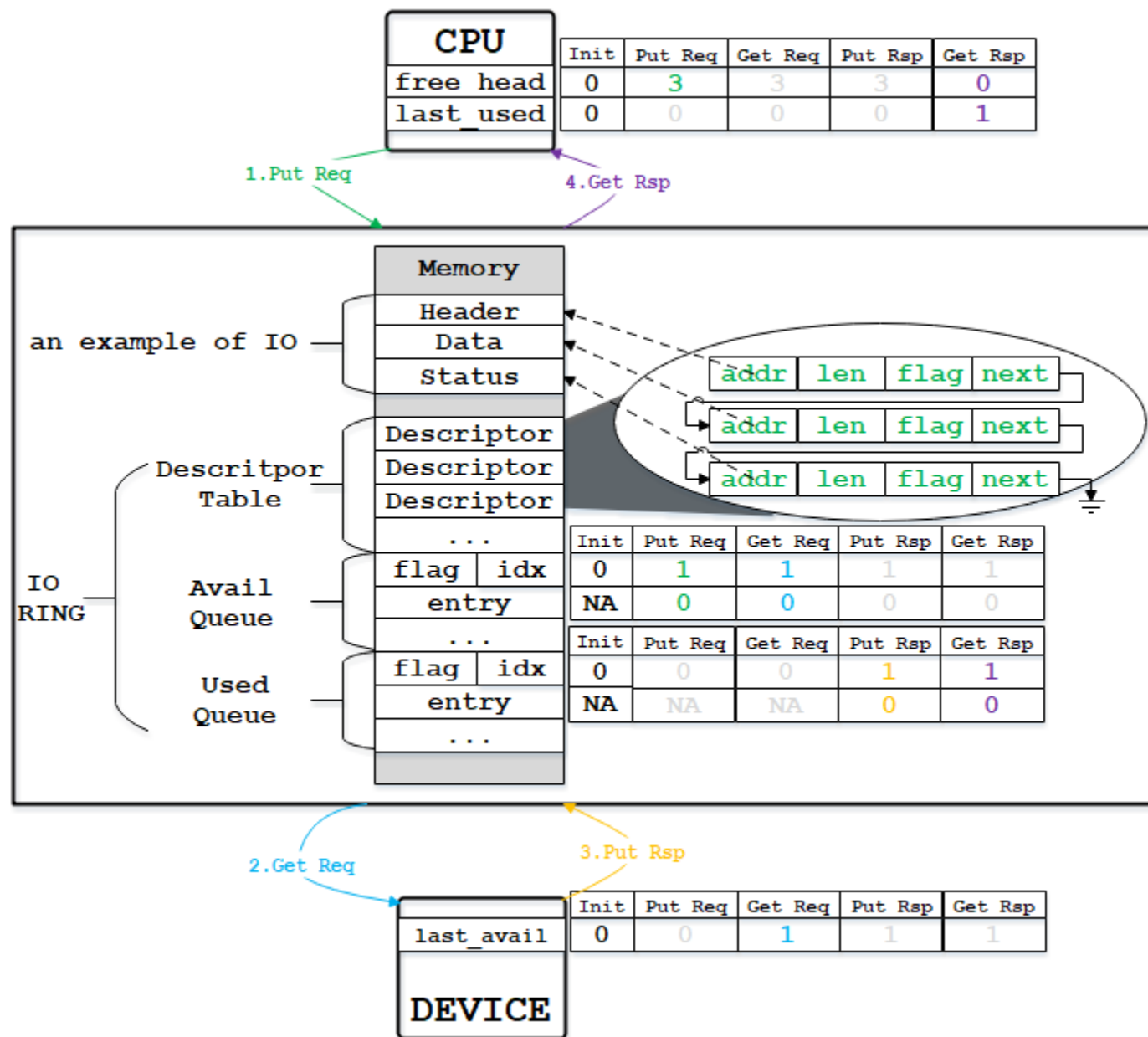
设备收到通知后，通过比较设备内部的last\_avail(初始为0)和Avail Queue中的idx(当前为1)判断是否有新的请求待处理。如果有，则取出请求(更新last\_avail为1)，并以entry的值为索引从Descriptor Table中找到请求对应的所有Descriptor来获取完整的请求信息

### • 第三步，设备放响应。

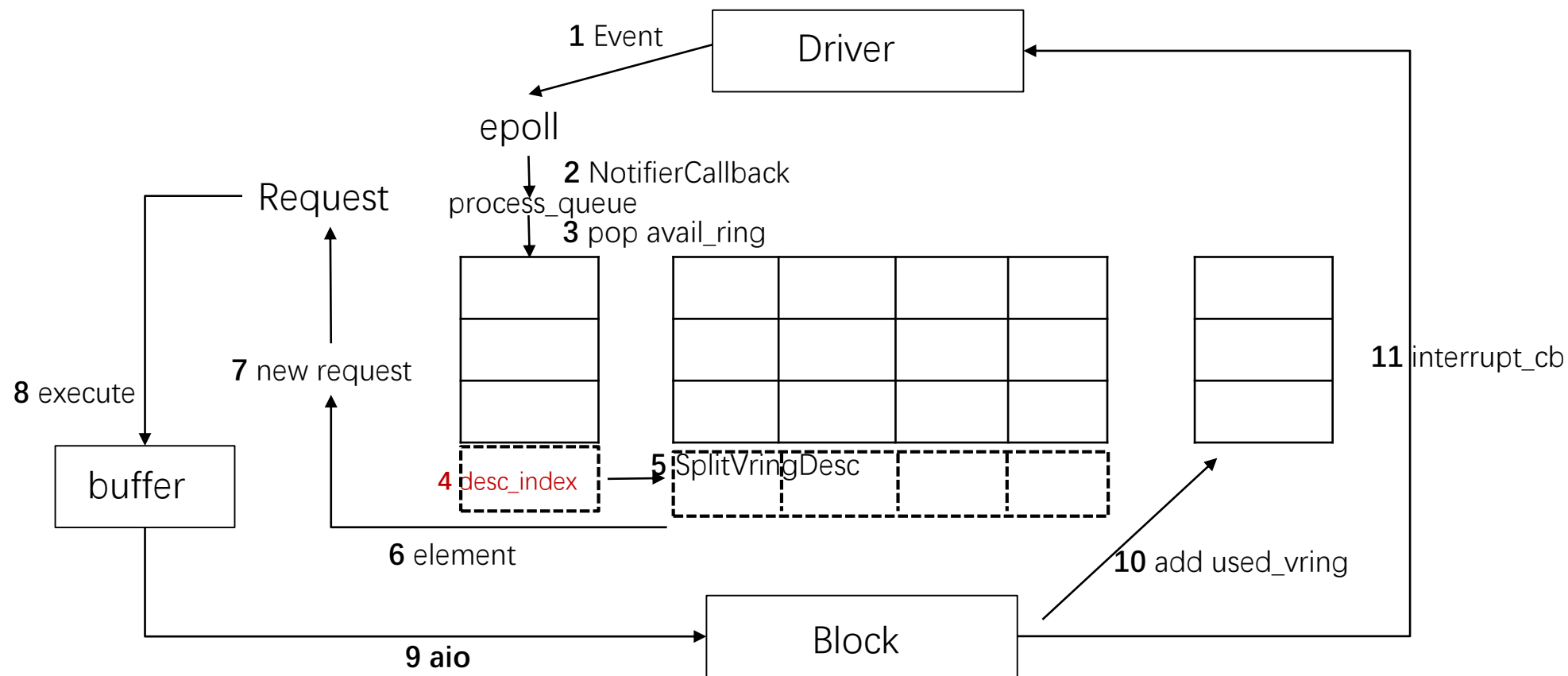
设备完成IO处理后，将已完成IO的Descriptor Table索引放入Used Queue对应的entry中，并将idx更新为1,代表放入1个响应

### • 第四步，CPU取响应。

CPU收到中断后，通过比较内部的last\_used(初始化0)和Used Queue中的idx(当前为1)判断是否有新的响应。如果有，则取出响应(更新last\_used为1)，并将Status中断的结果返回应用，最后将完成响应对应的三项Descriptor以链表方式插入到free\_head头部



## ► Virtio-blk整体流程回顾







# THANKS