

# linux 虚拟化技术

森林的天空（微信号：efenyekyj888）

## 目录

|                                       |    |
|---------------------------------------|----|
| 1. 认识虚拟化.....                         | 4  |
| 1.1. 裸金属虚拟化、宿主虚拟化、半虚拟化.....           | 4  |
| 1.2. cpu 虚拟化（全虚拟化、半虚拟化、硬件辅助虚拟化） ..... | 6  |
| 1.3. 内存虚拟化（全虚拟化、半虚拟化、硬件辅助虚拟化） .....   | 8  |
| 1.4. I/O 虚拟化（全虚拟化、半虚拟化、硬件辅助虚拟化） ..... | 11 |
| 1.5. I/O 透传与重定向.....                  | 13 |
| 2. 虚拟化的组件与安装启用 .....                  | 16 |
| 2.1. 物理机虚拟化组件 .....                   | 16 |
| 2.2. 物理机组件安装与启用 .....                 | 17 |
| 2.3. 虚拟机中相关组件 .....                   | 18 |
| 2.4. 虚拟机中组件安装 .....                   | 18 |
| 2.5. 物理主机虚拟化环境检验 .....                | 19 |
| 3. 虚拟机安装 .....                        | 20 |
| 3.1. 图形化安装 .....                      | 20 |
| 3.2. 命令化安装 .....                      | 21 |
| 3.3. 脚本方式安装 .....                     | 52 |
| 4. 虚拟机配置 .....                        | 52 |
| 4.1. 虚拟机基础信息配置 .....                  | 52 |

|                          |     |
|--------------------------|-----|
| 4.2. 内存与 VCPU 资源配置 ..... | 58  |
| 4.3. 体系架构相关配置 .....      | 87  |
| 5. 虚拟机设备管理 .....         | 105 |
| 5.1. 模拟器路径 .....         | 105 |
| 5.2. 磁盘设备管理 .....        | 105 |
| 5.3. 控制器（总线）设备 .....     | 139 |
| 5.4. 控制器（网络接口）设备 .....   | 157 |
| 5.5. 串口与控制台 .....        | 167 |
| 5.6. 通道设备 .....          | 178 |
| 5.7. 输入设备 .....          | 182 |
| 5.8. 图形显示设备 .....        | 183 |
| 5.9. 声音设备 .....          | 187 |
| 5.10. 显卡视频设备 .....       | 188 |
| 5.11. USB 重定向 .....      | 197 |
| 5.12. 内存气球设备 .....       | 199 |
| 5.13. 随机数设备 .....        | 199 |
| 6. 虚拟机日常管理 .....         | 200 |
| 6.1. 虚拟机登录 .....         | 200 |
| 6.2. 虚拟机在线配置 .....       | 200 |
| 6.3. 虚拟机信息查询与监控 .....    | 200 |
| 6.4. 虚拟机生命周期管理 .....     | 241 |
| 6.5. 虚拟机镜像管理 .....       | 267 |

|                              |     |
|------------------------------|-----|
| 7. 网络资源池管理 .....             | 268 |
| 8. 存储资源池管理 .....             | 271 |
| 9. 虚拟机迁移 .....               | 272 |
| 9.1. 虚拟机迁移准备 .....           | 273 |
| 10. 快照与备份 .....              | 278 |
| 11. 虚拟机 xml 操作 .....         | 280 |
| 11.1. 虚拟机 XML 文件导出 .....     | 281 |
| 11.2. 虚拟机 XML 文件编辑 .....     | 283 |
| 11.3. 原始配置转 XML 文件 .....     | 284 |
| 11.4. XML 文件转原始配置 .....      | 284 |
| 11.5. 虚拟机保存文件 XML 导出 .....   | 285 |
| 11.6. 虚拟机保存文件 XML 编辑 .....   | 287 |
| 11.7. 虚拟机保存文件注册 .....        | 287 |
| 11.8. 虚拟机暂停保存文件 XML 导出 ..... | 288 |
| 11.9. 虚拟机暂停文件 XML 编辑 .....   | 288 |
| 11.10. 虚拟机暂停文件注册 .....       | 288 |
| 12. 物理机管理 .....              | 288 |
| 12.1. 物理机网络设置 .....          | 288 |
| 12.2. 物理机信息查询与监控 .....       | 288 |
| 13. virsh 其它命令 .....         | 302 |
| 13.1. help 帮助关键字 .....       | 302 |
| 13.2. virsh 短命令 .....        | 302 |

## 1.认识虚拟化

### 1.1.裸金属虚拟化、宿主虚拟化、半虚拟化

裸金属虚拟化、Host 虚拟化（宿主虚拟化）、半虚拟化的区别，核心在于从 虚拟化层位置、硬件依赖、性能损耗三个维度切入。以下是三者的准确定义、关键特性及核心差异对比。

#### 1.1.1.裸金属虚拟化 (Bare-Metal Virtualization)

定义：又称原生虚拟化，虚拟化层（Hypervisor）直接运行在物理硬件上，不依赖任何宿主操作系统（OS），直接管理硬件资源并为虚拟机（VM）分配资源。

核心特性：

虚拟化层（Hypervisor）是 硬件之上的第一层软件，本质是一个轻量级的操作系统（仅负责资源调度）；

虚拟机（VM）中运行的是完整的 客户机操作系统，（如 Windows、Linux），客户机无需修改即可直接运行；

依赖硬件辅助虚拟化技术（如 Intel VT-x、AMD-V），通过硬件指令减少性能损耗；

典型代表：VMware ESXi、Citrix Hypervisor（Xen Server）、Microsoft Hyper-V（独立部署模式）。

#### 1.1.2.Host 虚拟化（宿主虚拟化）

定义：又称宿主虚拟化，虚拟化层（虚拟机监控程序）运行在宿主操作系统（Host OS）之上，依赖宿主 OS 管理物理硬件，再为虚拟机分配资源。

核心特性：

架构分层：物理硬件 → 宿主 OS → 虚拟化层 → 虚拟机（VM）；

虚拟化层本质是“宿主 OS 的一个应用程序，（如 VMware Workstation、VirtualBox），无需直接操作硬件；

客户机 OS 无需修改，但资源调度需经过“宿主 OS→虚拟化层”两层转发，性能损耗较高（尤其 I/O 密集场景）；

优势是 部署简单，（直接在现有 OS 上装虚拟化软件即可），适合个人开发、测试场景；

典型代表：VMware Workstation、Oracle VirtualBox、Parallels Desktop（Mac）。

1.1.3.半虚拟化 (Para-Virtualization)

定义：一种 优化型虚拟化，虚拟化层（Hypervisor）可直接运行在硬件（类似裸金属）或宿主 OS 上，但客户机操作系统需要修改内核，以去除 无法虚拟化的指令，并直接与 Hypervisor 通信。

核心特性：

核心逻辑：通过修改客户机内核，让客户机 知道自己在虚拟化环境中，跳过“硬件指令模拟，直接调用 Hypervisor 的接口（如 Xen 的 hypercall）；

不依赖硬件辅助虚拟化技术（早期无 VT-x/AMD-V 时的主流方案），性能接近裸金属（比 Host 虚拟化低损耗）；

局限性：客户机 OS 必须是 支持半虚拟化的版本（如定制内核的 Linux，Windows 不支持半虚拟化）；

典型代表：Xen（早期版本的半虚拟化模式，后期也支持全虚拟化）、KVM（可通过 virtio 驱动实现类半虚拟化优化）。

1.1.4.三者核心差异对比

| 对比维度    | 裸金属虚拟化        | Host 虚拟化（宿主）      | 半虚拟化        |
|---------|---------------|-------------------|-------------|
| 虚拟化层位置  | 直接运行在物理硬件上    | 运行在宿主 OS 之上（作为应用） | 为直接运行在物理硬件上 |
| 依赖宿主 OS | 不依赖（自身即底层调度层） | 依赖（资源需经宿主 OS 转发）  | 不依赖（类似裸金属）  |

|              |                         |                               |                           |
|--------------|-------------------------|-------------------------------|---------------------------|
| 客户机 OS 是否需修改 | 无需修改(直接运行原生 OS)         | 无需修改 (原生 OS)                  | 必须修改内核 (适配 Hypervisor)    |
| 硬件辅助依赖       | 依赖 (VT-x/AMD-V)         | 可选 (部分支持硬件加速)                 | 不依赖 (靠内核修改优化)             |
| 性能损耗         | 极低 (接近物理机, 5% 以内)       | 高 (10%-30%, 依赖宿主 OS)          | 低 (5%-15%, 比裸金属略高)        |
| 典型场景         | 企业服务器、云数据中心 (如 AWS EC2) | 个人开发、测试、桌面级使用                 | 早期服务器虚拟化、Linux 专属场景       |
| 代表产品         | Mware ESXi、Hyper-V (独立) | VMware Workstation、VirtualBox | Xen (半虚拟化模式)、KVM (virtio) |

1.2.cpu 虚拟化 (全虚拟化、半虚拟化、硬件辅助虚拟化)

CPU 虚拟化技术中，全虚拟化、半虚拟化和硬件辅助虚拟化是三种核心实现方式，主要区别体现在对客户机操作系统的依赖、性能损耗及实现原理上，以下是具体对比。

1.2.1.cpu 全虚拟化 (Full Virtualization)

定义：无需修改客户机操作系统，通过软件模拟完整的物理 CPU 环境，让客户机认为自己直接运行在物理硬件上。

核心原理：

虚拟化层 (Hypervisor) 拦截并模拟特权指令，如修改 CPU 状态的指令)，客户机无需感知虚拟化环境。

早期依赖二进制翻译技术 (如 QEMU 的动态指令翻译)，将客户机的特权指令转换为非特权指令执行。

优势：客户机 OS 无需任何修改 (支持 Windows、Linux 等所有系统)，兼容性极强。

劣势：软件模拟特权指令会导致性能损耗较高 (尤其 CPU 密集型场景)。

典型场景：桌面级虚拟化 (如 VirtualBox 默认模式)、早期服务器虚拟化。

1.2.2.2. cpu半虚拟化 (Para-Virtualization)

定义：通过修改客户机操作系统内核，让客户知晓自己处于虚拟化环境，直接与 Hypervisor 通信，避免指令模拟开销。

核心原理：

客户机内核去除 无法虚拟化的特权指令，改用 Hypervisor 提供的 超调用（Hypercall）接口（如 Xen 的 hypercall）直接请求资源。

无需模拟 CPU 指令，由客户机主动配合 Hypervisor 完成资源调度。

优势：性能损耗低（接近物理机），无需复杂的指令翻译。

劣势：客户机 OS 必须修改（如定制 Linux 内核），不支持未修改的操作系统（如 Windows）。

典型场景：早期 Linux 服务器虚拟化（如 Xen 的半虚拟化模式）。

1.2.3.3. cpu硬件辅助虚拟化 (Hardware-Assisted Virtualization)

定义：依赖 CPU 硬件提供的虚拟化扩展（如 Intel VT-x、AMD-V），在硬件层面支持虚拟化，减少软件模拟开销。

核心原理：

CPU 引入根模式（Hypervisor 运行）和非根模式（客户机运行），通过硬件指令（如 VMX 指令集）实现两种模式的快速切换。

特权指令由硬件直接拦截并交由 Hypervisor 处理，无需软件模拟或二进制翻译。

优势：性能接近物理机（损耗通常 < 5%），同时支持未修改的客户机 OS（兼顾全虚拟化的兼容性和半虚拟化的性能）。

劣势：依赖特定硬件（需 CPU 支持 VT-x/AMD-V），老旧硬件无法使用。

典型场景：现代服务器虚拟化（KVM、VMware ESXi、Hyper-V）、高性能桌面虚拟化。

核心区别对比表

| 维度           | 全虚拟化        | 半虚拟化      | 硬件辅助虚拟化    |
|--------------|-------------|-----------|------------|
| 客户机 OS 是否需修改 | 无需修改（完全兼容）  | 必须修改内核    | 无需修改（完全兼容） |
| 性能损耗         | 较高（10%-30%） | 低（5%-15%） | 极低（<5%）    |

|      |              |                   |                          |
|------|--------------|-------------------|--------------------------|
| 依赖硬件 | 无（纯软件实现）     | 无                 | 依赖 CPU 虚拟化扩展（VT-x/AMD-V） |
| 实现核心 | 二进制翻译 / 指令模拟 | 客户机主动调用 Hypercall | 硬件层面支持模式切换               |
| 典型代表 | QEMU（纯软件模式）  | Xen（半虚拟化模式）       | KVM、VMware ESXi、Hyper-V  |
| 适用场景 | 兼容性优先（如旧系统）  | Linux 专属场景        | 高性能场景（服务器、云平台）           |

总结

- 全虚拟化以兼容性为核心，牺牲部分性能；
- 半虚拟化以性能为核心，牺牲兼容性；
- 硬件辅助虚拟化是现代主流方案，兼顾兼容性和高性能，依赖 CPU 硬件支持。

1.3.内存虚拟化（全虚拟化、半虚拟化、硬件辅助虚拟化）

内存虚拟化是虚拟化技术的核心组成部分，负责为虚拟机提供独立的内存地址空间，并实现物理内存的高效分配与管理。全虚拟化、半虚拟化和硬件辅助虚拟化在内存虚拟化中的实现方式和特点各有不同，具体区别如下。

1.3.1.内存全虚拟化 (Full Virtualization for Memory)

定义：通过软件模拟完整的物理内存环境，让虚拟机（VM）认为自己拥有独立的物理内存空间，无需修改客户机操作系统。

核心原理：

虚拟机使用 虚拟内存地址（VA），Hypervisor 通过两次地址转换实现地址映射：虚拟机的虚拟地址（VA）→ 虚拟机物理地址（PA）→ 实际物理地址（HPA）。

当虚拟机访问内存时，Hypervisor 拦截拦截内存访问请求，通过软件模拟完成地址转换（如 QEMU 的“影子页表”技术）。

优势：客户机无需修改客户机操作系统，兼容性极强，支持所有操作系统（如 Windows、Linux）。

劣势：

地址转换依赖软件模拟（如影子页表维护），性能损耗较高（尤其内存密集访问场景）。



每一次内存访问都可能触发 Hypervisor 干预，增加额外开销。

典型场景：早期虚拟化环境（如无硬件辅助时的 QEMU）、对兼容性要求高于性能的场景。

### 1.3.2.内存半虚拟化 (Para-Virtualization for Memory)

定义：通过修改客户机操作系统内核，让虚拟机知晓“虚拟化环境，主动配合 Hypervisor 完成内存地址转换，减少软件模拟开销。

核心原理：

客户机内核集成 半虚拟化驱动，直接向 Hypervisor 注册注册地址转换请求（而非依赖 Hypervisor 拦截）。

取消 影子页表，Hypervisor 直接维护虚拟机的页表，并向客户机暴露部分接口（如内存分配、页表更新）。

优势：

减少软件模拟环节，性能损耗显著降低（接近物理机）。

内存地址转换效率更高，适合内存密集型场景。

劣势：

客户机操作系统必须修改（如定制 Linux 内核），不支持未修改的系统（如 Windows）。

依赖客户机与 Hypervisor 的协同设计，兼容性受限。

典型场景：Xen 早期的半虚拟化模式（PV 模式）、Linux 专属虚拟化环境。

### 1.3.3.内存硬件辅助内存虚拟化 (Hardware-Assisted Memory Virtualization)

定义：依赖 CPU 硬件提供的内存虚拟化扩展（如 Intel EPT、AMD NPT），在硬件层面实现内存地址转换，减少软件干预。

核心原理：

硬件引入扩展页表（EPT/NPT），直接支持三级地址转换：虚拟机虚拟地址（VA）→ 虚拟机物理地址（PA）→ 实际物理地址（HPA）。

内存访问请求由硬件自动完成地址转换，仅在权限不足或异常时才通知 Hypervisor，大幅减少软件开销。

优势：

性能接近物理机（损耗通常 < 5%），同时支持未修改的客户机 OS（兼顾兼容性和性能）。

硬件直接处理地址转换，Hypervisor 无需维护影子页表，简化实现复杂度。

劣势：

依赖特定硬件（需 CPU 支持 EPT/NPT），老旧硬件无法使用。

典型场景：现代虚拟化技术（KVM、VMware ESXi、Hyper-V）的主流内存虚拟化方案，适用于服务器、云平台等高性能场景。

| 维度           | 内存全虚拟化      | 内存半虚拟化                  | 硬件辅助内存虚拟化               |
|--------------|-------------|-------------------------|-------------------------|
| 客户机 OS 是否需修改 | 无需修改（完全兼容）  | 必须修改内核                  | 无需修改（完全兼容）              |
| 地址转换方式       | 软件模拟（影子页表）  | 客户机主动调用 + Hypervisor 协同 | 硬件直接支持（EPT/NPT）         |
| 性能损耗         | 较高（10%-30%） | 低（5%-15%）               | 极低（<5%）                 |
| 依赖硬件         | 无（纯软件实现）    | 无                       | 依赖 CPU 扩展（EPT/NPT）      |
| 典型代表         | QEMU（纯软件模式） | Xen PV 模式               | KVM、VMware ESXi、Hyper-V |
| 适用场景         | 兼容性优先（如旧系统） | Linux 专属场景              | 高性能场景（服务器、云平台）          |

总结

内存全虚拟化以兼容性为核心，通过软件模拟实现，但性能损耗较高；

内存半虚拟化通过修改客户机内核提升性能，但兼容性受限；

硬件辅助内存虚拟化是现代主流方案，依托 CPU 硬件扩展（EPT/NPT）兼顾兼容性和高性能，是企业级虚拟化的首选。

实际应用中，KVM、VMware 等技术均以硬件辅助内存虚拟化为基础，同时结合半虚拟化思想（如内存共享、气球技术）进一步优化内存利用率。

## 1.4.I/O 虚拟化 (全虚拟化、半虚拟化、硬件辅助虚拟化)

I/O 虚拟化是实现虚拟机与物理 I/O 设备（如网卡、磁盘控制器、USB 设备等）交互的核心技术，全虚拟化、半虚拟化和硬件辅助虚拟化在 I/O 虚拟化中的实现方式和特点差异显著，具体区别如下。

### 1.4.1.1. I/O 全虚拟化 (Full Virtualization for I/O)

定义：通过软件完全模拟物理 I/O 设备（如模拟真实网卡、硬盘控制器），让虚拟机认为自己在直接操作物理设备，无需修改客户机操作系统。

核心原理：

Hypervisor（或 QEMU 等模拟器）在用户态模拟完整的 I/O 设备硬件逻辑（如寄存器、中断处理流程）。

虚拟机的 I/O 请求（如读写磁盘、网络发包）会被 Hypervisor 拦截，由模拟器将请求转换为对物理设备的操作，再将结果返回给虚拟机。

典型实现：QEMU 的纯软件模拟（如模拟 Intel e1000 网卡、IDE 磁盘控制器）。

优势：

客户机操作系统无需任何修改，兼容性极强（支持所有未适配虚拟化的系统）。

无需特殊硬件支持，可在任何物理设备上运行。

劣势：

性能损耗极高：I/O 请求需经过“虚拟机→Hypervisor→模拟器→物理设备”多层转发，且软件模拟设备逻辑开销大（尤其高吞吐量场景，性能可能仅为物理机的 30%-50%）。

模拟复杂设备时，实现难度大且易出现兼容性问题。

典型场景：早期虚拟化环境、对兼容性要求极高而性能要求低的场景（如旧系统迁移）。

### 1.4.2.I/O 半虚拟化 (Para-Virtualization for I/O)

定义：通过在客户机中安装专用的半虚拟化驱动（如 virtio 驱动），让虚拟机知晓虚拟化环境，直接与 Hypervisor 通信，跳过全虚拟化的设备模拟环节。

核心原理：

Hypervisor 抽象出虚拟 I/O 接口（而非模拟物理设备），并提供标准化的接口规范（如 virtio 协议）。

客户机安装半虚拟化驱动后，I/O 请求不再通过模拟设备，而是直接通过前端驱动（客户机内）→后端驱动（Hypervisor 内）的高效通道传递，减少中间环节。

典型实现：KVM 的 virtio 驱动（virtio-net 网卡、virtio-blk 磁盘）、Xen 的 PV 驱动。

优势：

性能接近物理机（损耗通常 < 10%），大幅优于全虚拟化。

实现逻辑简单（无需模拟复杂物理设备），稳定性高。

劣势：

客户机必须安装专用驱动（需操作系统支持，如 Linux 内核默认集成 virtio，Windows 需手动安装）。

依赖 Hypervisor 与驱动的协同设计，接口不兼容物理设备。

典型场景：现代虚拟化环境（如 KVM、Xen）的主流 I/O 方案，适用于服务器、云平台等对 I/O 性能有要求的场景。

#### 1.4.3.3. I/O 硬件辅助 I/O 虚拟化 (Hardware-Assisted I/O Virtualization)

定义：依赖硬件设备（如 I/O MMU、SR-IOV）提供的虚拟化支持，在硬件层面实现 I/O 请求的直接转发，减少 Hypervisor 干预。

核心原理：

I/O MMU（如 Intel VT-d、AMD IOMMU）：硬件层面实现虚拟机物理地址到物理设备地址的转换，允许虚拟机直接访问物理 I/O 设备，同时隔离不同虚拟机的 I/O 空间。

SR-IOV（单根 I/O 虚拟化）：物理设备（如网卡）可虚拟出多个“虚拟功能（VF）”，每个 VF 可直接分配给虚拟机，实现 I/O 请求“虚拟机→物理设备”的直通，无需 Hypervisor 中转。

优势：

性能接近物理机（损耗 < 5%），是目前 I/O 性能最优的方案。

支持未修改的客户机操作系统（部分场景需驱动支持），兼顾性能与兼容性。

劣势：

依赖特定硬件（需 CPU 支持 VT-d/IOMMU、设备支持 SR-IOV），成本较高。

配置复杂，且虚拟功能数量受物理设备限制（如一块网卡最多虚拟 32 个 VF）。

典型场景：高性能计算（HPC）、数据库、大流量网络服务等对 I/O 延迟和吞吐量要求极高的场景。

| 维度       | I/O 全虚拟化                 | I/O 半虚拟化                      | 硬件辅助 I/O 虚拟化            |
|----------|--------------------------|-------------------------------|-------------------------|
| 客户机是否需驱动 | 无需（使用物理设备驱动）             | 必须安装半虚拟化驱动（如 virtio）          | 可选（VF 直通需驱动，I/O MMU 无需） |
| 数据路径     | 虚拟机→Hypervisor→模拟器→物理设备  | 虚拟机（前端）→Hypervisor（后端）→物理设备   | 虚拟机→物理设备（硬件直接转发）        |
| 性能损耗     | 高（50%-70%）               | 低（<10%）                       | 极低（<5%）                 |
| 硬件依赖     | 无                        | 无（软件定义接口）                     | 依赖 VT-d/IOMMU、SR-IOV 设备 |
| 典型实现     | QEMU 模拟 e1000 网卡、IDE 控制器 | KVM virtio（net/blk）、Xen PV 驱动 | SR-IOV 网卡直通、VT-d 磁盘直通   |
| 适用场景     | 兼容性优先（旧系统）               | 通用高性能场景（服务器、云平台）              | 极致性能场景（HPC、大流量服务）       |

总结

- I/O 全虚拟化以兼容性为核心，牺牲性能，适合简单或旧系统场景；
- I/O 半虚拟化通过专用驱动平衡性能与兼容性，是现代虚拟化的主流方案；
- 硬件辅助 I/O 虚拟化依托硬件实现高性能直通，适用于对 I/O 要求极致的场景。
- 实际应用中，KVM、VMware 等平台通常结合半虚拟化（如 virtio）和硬件辅助虚拟化（如 SR-IOV），根据场景灵活选择 I/O 方案。

1.5.I/O 透传与重定向

在虚拟化环境中，I/O 透传（I/O Passthrough）和 I/O 重定向（I/O Redirection）是两种实现虚拟机与物理 I/O 设备交互的重要方式，核心区别在于数据路径是否经过虚拟化层转发。以下是具体解析：

#### 1.5.1.1. I/O 透传 (I/O Passthrough)

定义：直接将物理 I/O 设备（如网卡、GPU、磁盘控制器）分配给虚拟机，让虚拟机绕过 Hypervisor，直接与物理设备交互，仿佛设备专门为该虚拟机所有。

核心原理：

依赖硬件辅助虚拟化技术（如 Intel VT-d、AMD IOMMU），实现物理设备与虚拟机的直接绑定，同时隔离不同虚拟机的设备访问。

物理设备从宿主机解绑，其资源（中断、I/O 地址空间）完全由目标虚拟机控制，Hypervisor 仅负责初始分配，不参与后续 I/O 请求处理。

典型实现：

全设备透传：将整个物理设备（如 PCIe 网卡、USB 设备）直接分配给虚拟机。

SR-IOV 透传：物理设备（如支持 SR-IOV 的网卡）虚拟出多个 虚拟功能（VF），每个 VF 可透传给不同虚拟机，共享物理设备资源但保持隔离。

优势：

性能接近物理机：I/O 请求无需 Hypervisor 中转，延迟极低（适合高性能场景）。

支持对硬件敏感的应用（如需要直接访问 GPU 的图形渲染、需要特定网卡功能的网络加速）。

劣势：

依赖硬件支持（VT-d/IOMMU、SR-IOV），配置复杂。

设备一旦透传，宿主机和其他虚拟机无法使用，资源灵活性低。

适用场景：高性能计算（HPC）、数据库、GPU 虚拟化、大流量网络服务等对 I/O 性能要求极高的场景。

#### 1.5.2.2. I/O 重定向 (I/O Redirection)

定义：虚拟机的 I/O 请求不直接访问物理设备，而是通过 Hypervisor 或中间层转发到物理设备（或其他目标），虚拟化层全程参与请求处理。

核心原理：

虚拟机访问的是 Hypervisor 提供的虚拟设备如虚拟网卡、虚拟磁盘，所有 I/O 请求先发送到虚拟设备。

**Hypervisor** 将虚拟设备的请求 翻译或转发到物理设备（如将虚拟磁盘请求映射到物理磁盘文件，将虚拟网卡请求转发到物理网卡）。

典型实现：

**存储重定向：**虚拟机的虚拟磁盘文件（如 `qcow2`、`vmdk`）存储在宿主机文件系统或共享存储中，I/O 请求由 **Hypervisor** 转发到对应存储介质。

**网络重定向：**虚拟机的网络请求通过虚拟交换机（如 `Linux Bridge`、`Open vSwitch`）转发到物理网卡，支持 **NAT**、桥接等网络模式。

**USB 重定向：**将宿主机或客户端的 **USB** 设备通过网络映射到虚拟机（如远程桌面中的 **USB** 设备共享）。

优势：

**灵活性高：**物理设备可被多个虚拟机共享，无需硬件透传支持。

**易于管理：****Hypervisor** 可统一控制 I/O 流量（如限速、过滤），支持快照、迁移等高级功能。

劣势：

**性能损耗：**I/O 请求经过虚拟化层转发，延迟和吞吐量略低于透传（现代半虚拟化驱动如 `virtio` 可大幅降低损耗）。

**适用场景：**大多数通用虚拟化场景（如云计算、服务器虚拟化），尤其需要资源共享和灵活管理的场景。

核心区别对比表

| 维度    | I/O 透传（Passthrough）                                                | I/O 重定向（Redirection）                           |
|-------|--------------------------------------------------------------------|------------------------------------------------|
| 数据路径  | 虚拟机→物理设备（直接交互）                                                     | 虚拟机→虚拟设备→Hypervisor→物理设备（转发）                   |
| 硬件依赖  | 强（需 <code>VT-d</code> / <code>IOMMU</code> 、 <code>SR-IOV</code> ） | 弱（无需特殊硬件）                                      |
| 性能损耗  | 极低（接近物理机）                                                          | 较低（取决于转发效率， <code>virtio</code> 可做到 $< 10\%$ ） |
| 设备共享性 | 差（通常专属一个虚拟机）                                                       | 好（多个虚拟机共享物理设备）                                 |
| 管理灵活性 | 低（设备绑定后难调整）                                                        | 高（支持动态调度、流量控制）                                 |
| 典型应用  | <b>GPU</b> 透传、高性能网卡直通                                              | 虚拟磁盘、虚拟网络、 <b>USB</b> 共享                       |

总结

I/O 透传追求极致性能，适合对硬件直接访问有需求的场景，但牺牲了灵活性和硬件利用率。

I/O 重定向通过虚拟化层转发实现资源共享，兼顾性能与灵活性，是大多数虚拟化环境的默认选择。

实际应用中，两者常结合使用：例如普通虚拟机用重定向（virtio 驱动）保证灵活性，而高性能需求的虚拟机（如 AI 训练节点）用 GPU 透传提升性能。

## 2.虚拟化的组件与安装启用

### 2.1.物理机虚拟化组件

Linux KVM（Kernel-based Virtual Machine）虚拟化技术由多个核心组件构成，各组件分工明确，共同实现完整的虚拟化功能。以下是主要组件及其用途：

#### 1. KVM 内核模块（kvm.ko 等）

核心角色：Linux 内核内置的虚拟化模块，是 KVM 的核心引擎。

功能：

提供硬件辅助虚拟化支持（如 Intel VT-x/AMD-V），将物理 CPU 资源虚拟化为 vCPU。

负责 CPU 调度、内存虚拟化（通过 EPT/NPT 技术实现内存地址转换）。

作为虚拟机与物理硬件之间的桥梁，处理虚拟机的核心指令和中断。

#### 2. QEMU（Quick Emulator）

核心角色：用户态的模拟器，与 KVM 配合提供完整的虚拟化能力。

功能：

模拟虚拟机所需的各类硬件设备（如网卡、硬盘、显卡、USB 控制器等）。

通过 /dev/kvm 接口与 KVM 内核模块交互，将设备模拟请求转化为内核级操作。

提供虚拟机的启动、关闭、快照等生命周期管理功能。

支持多种磁盘镜像格式（qcow2、raw 等）和网络模式（桥接、NAT 等）。

#### 3. libvirt

核心角色：虚拟化管理的统一接口库，抽象不同虚拟化技术的差异。

功能：

提供统一的 API（C 库）和命令行工具（virsh），简化虚拟机管理。



支持对虚拟机、存储池、虚拟网络的生命周期管理。

实现虚拟机迁移、快照、资源限制等高级功能。

兼容 KVM、Xen、VMware 等多种虚拟化技术，降低跨平台管理成本。

#### 4. 管理工具

**virsh:** libvirt 提供的命令行工具，用于通过终端管理虚拟机（启动、配置、监控等）。

**virt-manager:** 图形化管理工具，通过 GUI 界面直观管理虚拟机、存储和网络。

**virt-install/virt-clone:** 命令行工具，分别用于快速创建和克隆虚拟机。

**qemu-img:** QEMU 提供的磁盘镜像管理工具，用于创建、转换、扩容虚拟机磁盘文件。

#### 5. 存储与网络组件

**存储池 (Storage Pool):** 由 libvirt 管理的存储抽象，可对应本地目录、LVM 卷、NFS 共享等，用于集中管理虚拟机磁盘镜像。

**虚拟网络 (Virtual Network):** 由 libvirt 或 QEMU 模拟的网络环境，支持 NAT、桥接、隔离等模式，实现虚拟机网络连接。

#### 6. 辅助工具

**virt-viewer:** 轻量级虚拟机控制台工具，用于连接虚拟机的图形界面。

**spice-server/VNC:** 远程桌面协议支持，实现对虚拟机的远程访问。

**libguestfs:** 用于直接操作虚拟机磁盘镜像（如修改配置、安装软件），无需启动虚拟机。

## 2.2.物理机组件安装与启用

### 安装

```
sudo apt install qemu-system-x86 libvirt-clients virtinst virt-viewer qemu-utils  
bridge-utils spice-vdagent libosinfo-bin spice-server
```

每个 linux 发行版的包名都不一样，最好让大模型给出比较全的包用于安装。

### 启用

```
sudo usermod -aG libvirt $USER
```

```
sudo usermod -aG kvm $USER
```

```
sudo systemctl restart libvirtd
```

```
sudo systemctl enable libvirtd
```

## 2.3.虚拟机中相关组件

virtio 驱动：linux 自带，windows 系统需要安装，安装了后，性能提升非常高。

qemu-guest-agent：linux、windows 都需要安装，安装后，可以进行信息的同步，并不依赖 IP 通信。

qemu-guest-agent 是一个运行在虚拟机内部的后台服务程序，用于在虚拟机（客户机）和宿主机之间建立通信通道，实现两者的信息交互和操作协同。它是 QEMU/KVM 虚拟化环境中非常重要的组件，能增强对虚拟机的管理能力。

核心作用

信息同步让宿主机可以获取虚拟机内部的详细信息，如：

操作系统类型、版本、主机名

网络配置（IP 地址、MAC 地址）

文件系统挂载情况、磁盘使用率

内存、CPU 等资源的实时使用状态

操作协同支持宿主机对虚拟机执行更精细的操作，如：

优雅关机 / 重启（避免强制断电对虚拟机的损害）

挂载 / 卸载宿主机共享文件

设置虚拟机内部的系统时间

执行自定义命令（需配置权限）

工作原理

宿主机通过 libvirt 服务与虚拟机的 qemu-guest-agent 通信

通信方式依赖虚拟机配置中的 virtio-serial 通道（一种高效的本地通信机制，无需网络）

数据传输通过标准化协议，确保跨操作系统兼容性

## 2.4.虚拟机中组件安装

```
sudo apt-get install qemu-guest-agent
```

```
sudo systemctl start qemu-guest-agent
```

```
sudo systemctl enable qemu-guest-agent
```

## 2.5.物理主机虚拟化环境检验

virt-xml-validate

syntax: /usr/bin/virt-xml-validate XMLFILE [TYPE]

efenye@efenye-deepin:~\$ virt-host-validate

|                                                        |      |
|--------------------------------------------------------|------|
| QEMU: Checking for hardware virtualization             | : 通过 |
| QEMU: Checking if device '/dev/kvm' exists             | : 通过 |
| QEMU: Checking if device '/dev/kvm' is accessible      | : 通过 |
| QEMU: Checking if device '/dev/vhost-net' exists       | : 通过 |
| QEMU: Checking if device '/dev/net/tun' exists         | : 通过 |
| QEMU: Checking for cgroup 'cpu' controller support     | : 通过 |
| QEMU: Checking for cgroup 'cpuacct' controller support | : 通过 |
| QEMU: Checking for cgroup 'cpuset' controller support  | : 通过 |
| QEMU: Checking for cgroup 'memory' controller support  | : 通过 |
| QEMU: Checking for cgroup 'devices' controller support | : 警告 |

(Enable 'devices' in kernel Kconfig file or mount/enable cgroup controller in your system)

|                                                      |      |
|------------------------------------------------------|------|
| QEMU: Checking for cgroup 'blkio' controller support | : 通过 |
| QEMU: Checking for device assignment IOMMU support   | : 通过 |
| QEMU: Checking if IOMMU is enabled by kernel         | : 警告 |

(IOMMU appears to be disabled in kernel. Add intel\_iommu=on to kernel cmdline arguments)

|                                         |      |
|-----------------------------------------|------|
| QEMU: Checking for secure guest support | : 警告 |
|-----------------------------------------|------|

(Unknown if this platform has Secure Guest support)

|                                                   |      |
|---------------------------------------------------|------|
| LXC: Checking for Linux >= 2.6.26                 | : 通过 |
| LXC: Checking for namespace 'ipc'                 | : 通过 |
| LXC: Checking for namespace 'mnt'                 | : 通过 |
| LXC: Checking for namespace 'pid'                 | : 通过 |
| LXC: Checking for namespace 'uts'                 | : 通过 |
| LXC: Checking for namespace 'net'                 | : 通过 |
| LXC: Checking for namespace 'user'                | : 通过 |
| LXC: Checking for cgroup 'cpu' controller support | : 通过 |

|                                                       |       |
|-------------------------------------------------------|-------|
| LXC: Checking for cgroup 'cpuacct' controller support | : 通过  |
| LXC: Checking for cgroup 'cpuset' controller support  | : 通过  |
| LXC: Checking for cgroup 'memory' controller support  | : 通过  |
| LXC: Checking for cgroup 'devices' controller support | : 未通过 |

(Enable 'devices' in kernel Kconfig file or mount/enable cgroup controller in your system)

|                                                       |       |
|-------------------------------------------------------|-------|
| LXC: Checking for cgroup 'freezer' controller support | : 未通过 |
|-------------------------------------------------------|-------|

(Enable 'freezer' in kernel Kconfig file or mount/enable cgroup controller in your system)

|                                                           |      |
|-----------------------------------------------------------|------|
| LXC: Checking for cgroup 'blkio' controller support       | : 通过 |
| LXC: Checking if device '/sys/fs/fuse/connections' exists | : 通过 |

### 3.虚拟机安装

#### 3.1.图形化安装

安装好虚拟化组件后，会有 **virt-manager** 图形化管理系统，图形化的虚拟机管理工具，可用于创建、配置和管理虚拟机。

本地：

图形化打开，具体安装比较简单，实在遇到问题问大模型。

终端输入 **virt-manager** ；

或终端输入：**virt-managr -c qemu:///system** ；

远程：

图形化打开，里面加远程链接地址，见命令的地址：

终端输入：**virt-managr -c qemu+ssh://root@192.168.3.60/system** ,用户为 **root**,ip 为远程 **kvm** 物理机的 ip。

| 连接场景   | 连接字符串格式示例                        | 说明                                   |
|--------|----------------------------------|--------------------------------------|
| 本地     | <b>--connect qemu:///system</b>  | 连接本地系统级 <b>libvirt</b> 服务（最常用，默认可省略） |
| KVM 服务 |                                  |                                      |
| 本地用户   | <b>--connect qemu:///session</b> | 连接当前用户的私有 <b>libvirt</b> 服务（权限有限）    |
| 级服务    |                                  |                                      |
| 远程 SSH | <b>--connect</b>                 | 通过 <b>SSH</b> 协议连接远程主                |

| 连接场景        | 连接字符串格式示例                                                  | 说明                                      |
|-------------|------------------------------------------------------------|-----------------------------------------|
| 连接          | <code>qemu+ssh://user@remote-host/system</code>            | 机的系统级服务（需 SSH 认证）                       |
| 远程 TCP 连接   | <code>--connect qemu+tcp://remote-host:16509/system</code> | 通过 TCP 协议连接（需远程主机配置 libvirtd 监听 TCP 端口） |
| 远程 TLS 加密连接 | <code>--connect qemu+tls://remote-host/system</code>       | 通过 TLS 加密连接（安全性高，适合公网环境，需证书配置）          |

## 3.2.命令化安装

`virt-install` 是一款命令行工具，借助 libvirt 虚拟机监控程序管理库，可创建新的 KVM（基于内核的虚拟机）、Xen（开源虚拟化技术）或 Linux 容器客户机。

`virt-install` 工具支持通过 VNC（虚拟网络计算）或 SPICE（简单协议独立计算环境）进行图形化安装，也支持通过串行控制台进行文本模式安装。用户可将客户机配置为使用一个或多个虚拟磁盘、网络接口、音频设备、物理 USB 或 PCI 设备等。

安装介质既可以是本地的 ISO 文件或 CD-ROM 光盘，也可以是通过 HTTP、FTP 协议远程托管的发行版安装树，还可以是本地目录中的安装树。当使用安装树时，`virt-install` 会先获取启动安装过程所需的最小文件集，后续客户机可根据需要获取操作系统发行版的其余文件。此外，该工具还支持 PXE（预启动执行环境）引导以及导入现有磁盘镜像（从而跳过安装阶段）。

只要提供合适的命令行参数，`virt-install` 就能完全无人值守运行，客户机也可自行“自动安装”。这种特性便于实现客户机安装的自动化，既可以手动配置参数实现，也可以更简便地使用 `--unattended` 选项来完成。

`virt-install` 子命令非常多，参数也特别灵活，有必要去安装阶段全部写么，我觉得没有必要，因为其他参数的项目，都可以自动生成，通过脚本安装或者通过修改 xml 会更加快，当然前提是学习后面的内容。

许多参数都带有子选项，格式为 `opt1=foo,opt2=bar` 等。若要查看某个参数对应的所有子选项，可尝试使用 `--option=?` 的形式，例如：`virt-install --disk=?`

### 3.2.1.最小安装必选项

```
sudo virt-install \
    --name=efenye \
    --memory=2048 \
    --vcpus=2 \
    --disk path=/var/lib/libvirt/images/efenye.qcow2,size=20 \
    --cdrom=/home/efenye/Documents/os/deepin-desktop-community-25.0.1-amd64.iso \
    --os-variant=generic \
    --network bridge=virbr0 \
    --graphics spice \
    --boot cdrom,hd
sudo virt-install \
    --name=efenye \      #虚拟机名字
    --memory=2048 \    #最大内存，单位是 MB
    --vcpus=2 \ #vpcu 数量
    --disk path=/var/lib/libvirt/images/efenye.qcow2,size=20 \ #系统磁盘安装路径，默认是
G
    --cdrom=/home/efenye/Documents/os/deepin-desktop-community-25.0.1-amd64.iso \
    #安装文件路径
    --os-variant=generic \ #系统类型，非常多种，我一般选默认
    --network bridge=virbr0 \ #默认的，没有改过的话
    --graphics spice \ # spice 图形化界面
    --boot cdrom,hd # 系统启动顺序，先光驱，后硬盘
```

### 3.2.2.--connect

连接到非默认的虚拟机监控程序。若未指定该选项，libvirt 会尝试选择最合适的默认值。  
常见有效选项如下：

`qemu:///system`: 用于创建由系统 `libvirtd` 实例运行的 KVM 和 QEMU 客户机。这是 `virt-manager` 的默认模式，也是大多数 KVM 用户的首选。

`qemu:///session`: 用于创建由普通用户身份运行的 `libvirtd` 所管理的 KVM 和 QEMU 客户机。

举例: `qemu+ssh://root@192.168.3.60/system`

`xen:///`: 用于连接到 Xen 虚拟机监控程序。

`lxc:///`: 用于创建 Linux 容器。

### 3.2.3. --name

新客户机虚拟机实例的名称。该名称在连接到的虚拟机监控程序已知的所有客户机中必须唯一，包括当前未激活的客户机。若需重新定义现有客户机，需先使用 `virsh (1)` 工具将其关闭（`virsh shutdown`）并删除（`virsh undefine`），再运行 `virt-install`。

### 3.2.4. --metadata

`name`: （最常用）

作用: 指定虚拟机的名称。这个名称在宿主机上必须是唯一的。

示例: `--metadata name=web-server-01`

`description`:

作用: 为虚拟机添加一段描述性文字，说明其用途、负责人、环境等信息，便于管理。

示例: `--metadata description="Ubuntu 22.04 LTS Web Server"`

`title`:

作用: 设置一个简短的标题，通常用于管理工具的界面显示。

示例: `--metadata title="Production Web Server"`

`uuid`:

作用: 为虚拟机指定一个唯一的 UUID（通用唯一识别码）。如果不指定，`libvirt` 会自动生成一个。

示例: `--metadata uuid=123e4567-e89b-12d3-a456-426614174000`

注意: 除非有特殊需求（如迁移、克隆），一般不建议手动指定 UUID。

**genid:**

作用: 设置虚拟机的 generation ID, 这是一种在虚拟化环境中用于标识虚拟机实例代际的机制, 主要用于与 Windows Guest 的集成服务通信。

示例: `--metadata genid=123e4567-e89b-12d3-a456-426614174001`

**genid\_enable:**

作用: 控制是否启用 genid。可以设置为 yes 或 no。

示例: `--metadata genid_enable=yes`

**os\_name:**

作用: 指定客户机操作系统的名称 (如 linux, windows, unix)。这有助于管理工具进行分类。

示例: `--metadata os_name=linux`

**os\_full\_id:**

作用: 指定客户机操作系统的完整标识符, 比 os\_name 更详细 (如 centos7, ubuntu20.04, windows10)。

示例: `--metadata os_full_id=ubuntu22.04`

**clearxml:**

作用: 这是一个内部调试选项, 用于清除 XML 配置中的元数据。普通用户通常不需要使用。

**3.2.5.--vcpus**

| 选项       | 说明                       | 示例                                       |
|----------|--------------------------|------------------------------------------|
| vcpus    | 总 vCPU 数。最常用的选项。         | <code>--vcpus 4</code>                   |
| maxvcpus | 虚拟机支持的最大 vCPU 数 (用于热添加)。 | <code>--vcpus 2,maxvcpus=8</code>        |
| sockets  | 虚拟 CPU 插槽数。              | <code>--vcpus 4,sockets=2,cores=2</code> |
| cores    | 每个插槽的内核数。                |                                          |
| threads  | 每个内核的线程数。                |                                          |



| 选项             | 说明                                                  | 示例                                     |
|----------------|-----------------------------------------------------|----------------------------------------|
| dies           | 每个插槽的芯片数。                                           | --vcpus<br>8,sockets=1,dies=2,cores=4  |
| cpuset         | 将 vCPU 绑定到宿主机的特定物理 CPU 核心上，提升性能。                    | --vcpus 2,cpuset=2,3                   |
| vcpu.cpuset    | 同 cpuset，语法更明确。                                     | --vcpus 2,vcpu.cpuset=2,3              |
| vcpu.placement | vCPU 调度模式：<br>• static: 静态分配<br>• dynamic: 动态调度（默认） | --vcpus<br>4,vcpu.placement=static     |
| vcpu.current   | 启动时启用的 vCPU 数（小于等于 maxvcpus）。                       | --vcpus<br>2,maxvcpus=4,vcpu.current=2 |

### 3.2.6. --cpu

| 一、基础 CPU 配置 |                                                                                                                                                                             |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 选项          | 说明                                                                                                                                                                          |
| model       | 核心选项：指定 CPU 模型<br>(如 host-passthrough, qemu64, IvyBridge, EPYC)。<br>• host-passthrough: 完全透传宿主机 CPU，性能最好。<br>• host-model: 模拟宿主机 CPU，但可跨宿主机迁移。<br>• 具体型号：如 IvyBridge，用于兼容性。 |
| mode        | CPU 模式：<br>• host-passthrough<br>• host-model<br>• custom（配合 model 使用）                                                                                                      |
| match       | 匹配策略：exact, maximum, strict。定义如何处理 CPU 功能差异。                                                                                                                                |

| 选项              | 说明                                           |
|-----------------|----------------------------------------------|
| vendor          | 指定 CPU 厂商（如 GenuineIntel, AuthenticAMD）。     |
| model.fallback  | 当指定的 model 不可用时的备用策略（allow, forbid, repeat）。 |
| model.vendor_id | 设置 CPU 的 vendor_id 字符串。                      |
| secure          | 启用安全特性（如 Intel SGX, AMD SEV）。                |
| migratable      | 是否允许跨不同 CPU 的宿主机迁移（影响 host-passthrough）。     |

## 二、CPU 拓扑（Topology）

这些选项用于定义虚拟 CPU 的物理结构，影响操作系统识别。

| 选项               | 说明            |
|------------------|---------------|
| topology.sockets | 虚拟 CPU 插槽数    |
| topology.dies    | 每个插槽的芯片数（Die） |
| topology.cores   | 每个芯片/插槽的内核数   |
| topology.threads | 每个内核的线程数      |

✓ 示例：--cpu topology.sockets=1,topology.cores=4,topology.threads=2 表示 1 路 CPU，

4 核，8 线程。

## 三、NUMA（非统一内存访问）配置

用于模拟多路 NUMA 架构，优化大型虚拟机性能。

### 1. NUMA 节点（Cell）配置

| 选项                        | 说明                      |
|---------------------------|-------------------------|
| numa.cell[0-9]*.id        | NUMA 节点 ID              |
| numa.cell[0-9]*.cpus      | 分配给该节点的 vCPU 范围（如 0-3）  |
| numa.cell[0-9]*.memory    | 分配给该节点的内存（MiB）          |
| numa.cell[0-9]*.unit      | 内存单位（默认 MiB）            |
| numa.cell[0-9]*.memAccess | 内存访问模式（shared, private） |

## 2. NUMA 节点间距离

| 选项                                            | 说明            |
|-----------------------------------------------|---------------|
| numa.cell[0-9]*.distances.sibling[0-9]*.id    | 目标 NUMA 节点 ID |
| numa.cell[0-9]*.distances.sibling[0-9]*.value | 距离值（越小越近）     |

## 3. NUMA 节点缓存（Cache）

| 选项                                        | 说明               |
|-------------------------------------------|------------------|
| numa.cell[0-9]*.cache[0-9]*.level         | 缓存级别（L1, L2, L3） |
| numa.cell[0-9]*.cache[0-9]*.size.value    | 缓存大小             |
| numa.cell[0-9]*.cache[0-9]*.size.unit     | 单位（KB, MB）       |
| numa.cell[0-9]*.cache[0-9]*.line.value    | 缓存行大小            |
| numa.cell[0-9]*.cache[0-9]*.associativity | 关联性              |
| numa.cell[0-9]*.cache[0-9]*.policy        | 策略（back, index）  |

## 4. NUMA 互联带宽与延迟

| 选项                                           | 说明             |
|----------------------------------------------|----------------|
| numa.interconnects.bandwidth[0-9]*.initiator | 起始节点           |
| numa.interconnects.bandwidth[0-9]*.target    | 目标节点           |
| numa.interconnects.bandwidth[0-9]*.value     | 带宽值            |
| numa.interconnects.bandwidth[0-9]*.unit      | 单位（MBps, GBps） |
| numa.interconnects.latency[0-9]*.value       | 延迟（纳秒）         |
| numa.interconnects.latency[0-9]*.unit        | 单位（ns）         |

## 四、高级与调试选项

| 选项          | 说明                                      |
|-------------|-----------------------------------------|
| cache.mode  | CPU 缓存模式<br>(emulate, directsync, none) |
| cache.level | 缓存级别                                    |

| 选项                                               | 说明                                 |
|--------------------------------------------------|------------------------------------|
| maxphysaddr.bits                                 | 最大物理地址位数                           |
| maxphysaddr.mode                                 | 地址模式                               |
| xpath[0-9]*.set / .create / .delete              | 使用 XPath 直接修改 libvirt XML 配置（高级用法） |
| force, check, disable, forbid, require, optional | 控制功能的强制启用/禁用                       |

### 3.2.7.--cdrom

### 3.2.8.--import

从现有的磁盘导入虚拟机

### 3.2.9.--location

### 3.2.10.--unattended

#### 一、基本语法与核心作用

##### 1. 语法格式

`virt-install [安装源选项 (--install/--cdrom/--location)] --unattended [子参数=值,...]`

必需前提：需配合“安装源选项”使用（如 `--install` 指定发行版、`--cdrom` 指定 ISO、`--location` 指定安装树），否则无法确定自动安装脚本的适配对象；

核心目标：替代人工干预，自动完成“分区、用户创建、密码设置、软件包选择”等安装步骤，适合批量创建虚拟机或自动化部署场景。

##### 2. 核心特性（文档定义）

依赖 libosinfo 数据库：libosinfo 包含主流操作系统的自动安装配置模板（如 Fedora 的 kickstart 模板、Windows 的 autounattend.xml 模板），`--unattended` 会自动匹配并调用对应模板；

支持多系统类型：文档明确提到可用于 Red Hat 系（kickstart）、Debian 系（debconf）、Windows（无人值守安装），以及其他支持自动安装的发行版；

简化配置：无需手动编写完整的自动安装脚本（如 kickstart 文件），只需通过子参数指定关键信息（如密码、用户名）即可。

二、关键子参数详解（结合文档）

--unattended 支持多个子参数，用于定制自动安装的关键配置，未指定时会使用 libosinfo 的默认值。常见子参数如下：

| 子参数                    | 作用说明                                                                                          | 注意事项                                                      |
|------------------------|-----------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| profile=桌面版/jeos       | 选择 libosinfo 中的无人值守配置<br>模板类型：- desktop: 桌面版（默认，含图形界面）；- jeos: 精简版（Just Enough OS，无图形，适合服务器）。 | 部分发行版可能仅支持其中一种（如部分服务器发行版无 desktop 模板）                     |
| admin-password-file=路径 | 从文件中读取 “管理员 /root 密码”，仅读取文件第一行内容（含空格，不含换行符）。                                                  | 1. 避免密码明文写在命令行（防止日志泄露）；<br>2. 确保文件权限为 600（仅所有者可读），避免权限泄露。 |
| user-login=用户名         | 指定虚拟机中创建的普通用户名，默认使用当前主机用户名（如 user 运行行 virt-install，必须显式指定则默认 user）。                           | 若以 root 身份运行该参数（否则报错）。                                    |
| user-password-file=路径  | 从文件中读取 “普通用户密码”，规格会默认禁用该用户密码则同 admin-password-file。                                           | 若不指定，部分发行版会默认禁用该用户密码登录（需用密钥或管理员授权）。                       |
| product-key=密钥         | 仅用于 Windows 安装，指定 Windows 产品密钥（激活系统）。                                                         | 非必填，未指定时 Windows 会进入 “试用模式”，需后续手动激活。                      |

三、实际使用示例（覆盖主流场景）

1. 场景 1：无人值守安装 Fedora 39（用 --install 指定发行版）

通过 --install 自动获取 Fedora 39 的安装树，结合 --unattended 自动完成安装：

bash

# 1. 先创建管理员密码文件（权限设为 600，避免泄露）echo "MyFedoraPass123" >

/tmp/admin-pass.txtchmod 600 /tmp/admin-pass.txt

# 2. 无人值守安装 Fedora 39

virt-install \

--name fedora39-unattended \

--connect qemu:///system \

--memory 2048 \

--vcpus 2 \

--disk size=30 \ # 自动创建 30GB 虚拟磁盘

--osinfo fedora39 \ # 匹配 Fedora 39 的优化配置

--install fedora39 \ # 从 libosinfo 获取 Fedora 39 安装源

--unattended \

profile=desktop \ # 安装桌面版

admin-password-file=/tmp/admin-pass.txt \ # 管理员密码从文件读

user-login=fedora-user \ # 创建普通用户 fedora-user

user-password-file=/tmp/admin-pass.txt # 普通用户密码同管理员（简化示例）

2. 场景 2：无人值守安装 Windows 10（用 --cdrom 指定 ISO）

需提前准备 Windows 10 安装 ISO（libosinfo 需支持该 ISO 的自动识别）：

bash

# 1. 创建管理员密码文件 echo "Win10Pass@2024" > /tmp/win-pass.txtchmod 600

/tmp/win-pass.txt

# 2. 无人值守安装 Windows 10

virt-install \

--name win10-unattended \

--memory 4096 \

--vcpus 2 \

--disk size=50 \

--osinfo win10 \ # 适配 Windows 10 配置（启用 virtio 等）

--cdrom /home/user/Win10\_22H2.iso \ # Windows 10 ISO 路径

--unattended \

admin-password-file=/tmp/win-pass.txt \ # Windows 管理员 (Administrator) 密码

user-login=win-user \ # 创建普通用户 win-user

product-key=XXXXX-XXXXX-XXXXX-XXXXX-XXXXX # 可选: Windows 产品密钥

### 3. 场景 3: 无人值守初始化云镜像 (结合 --import)

云镜像 (如 Ubuntu Cloud) 默认无密码, 可通过 --unattended 自动设置密码 (无需安装, 直接初始化):

bash

# 1. 创建密码文件 echo "UbuntuCloudPass" > /tmp/cloud-pass.txtchmod 600

/tmp/cloud-pass.txt

# 2. 导入 Ubuntu 22.04 云镜像并无人值守初始化

virt-install \

--import \ # 跳过安装, 直接用现有云镜像

--name ubuntu2204-cloud \

--memory 1024 \

--vcpus 1 \

--disk /var/lib/libvirt/images/ubuntu-22.04-server-cloudimg-amd64.img \ # 云镜像路

径

--osinfo ubuntu22.04 \

--unattended \

admin-password-file=/tmp/cloud-pass.txt \ # 设置 root 密码

user-login=ubuntu \ # 云镜像默认普通用户为 ubuntu

user-password-file=/tmp/cloud-pass.txt

## 四、注意事项 (避坑关键点)

libosinfo 版本与系统兼容性

旧版本 libosinfo 可能不支持新发行版 (如 Fedora 40、Ubuntu 24.04), 导

致 --unattended 无法匹配自动安装模板, 需先升级 libosinfo (如 sudo dnf upgrade libosinfo);

可通过 osinfo-query os 命令查看 libosinfo 支持的系统列表, 确认目标系统是否在列。

密码文件的安全性

绝对不要将密码明文写在命令行（如 `--unattended admin-password=123456`），会被记录到系统日志（如 `/var/log/messages`），存在泄露风险；

密码文件使用后建议立即删除（如 `rm /tmp/admin-pass.txt`），避免残留。

#### Windows 安装的特殊要求

Windows ISO 需是“完整安装镜像”（含 `sources/install.wim`），精简版 ISO 可能缺少自动安装所需文件；

若 `libosinfo` 无法识别 Windows ISO，需手动指定 `--osinfo win10`（或对应版本），否则自动安装模板会匹配失败。

#### 与 `--cloud-init` 的区别

`--unattended` 用于操作系统安装阶段的自动化（从无到有装系统）；

`--cloud-init` 用于云镜像初始化（已有系统，仅配置密码、密钥等）；

二者可配合使用（如 `--import` 云镜像 + `--unattended` 设密码 + `--cloud-init` 注入 SSH 密钥），但需注意避免配置冲突（如同时用 `--unattended admin-password-file` 和 `--cloud-init root-password-file`，以 `--cloud-init` 为准）。

#### 五、常见问题解答

Q: 执行 `--unattended` 后报错 “no unattended profile found”? A: 原因是 `libosinfo` 未找到目标系统的自动安装模板，解决方案：① 升级 `libosinfo`；② 手动指定 `--osinfo`（如 `--osinfo fedora39`）；③ 若系统确实不支持，需改用手动编写 `kickstart` / 无人值守文件（配合 `--extra-args ks=...`）。

Q: Windows 无人值守安装卡在“准备安装”界面? A: 大概率是 ISO 路径权限不足或 `libosinfo` 模板不匹配，检查：① ISO 文件对 `root` 有读权限（`chmod +r /path/Win10.iso`）；② `--osinfo` 正确指定为 `win10`（而非 `win8` 等其他版本）。

### 3.2.11. `--reinstall DOMAIN`

根据 `virt-install` 文档，`--reinstall DOMAIN` 是一个针对“现有虚拟机”的重新安装选项——它不会创建新虚拟机，而是基于已有虚拟机（`DOMAIN` 指定）的配置，仅修改“安装源相关参数”并重新启动安装流程，安装完成后自动恢复虚拟机原有非安装配置（如网络、存储、内存等），适合“保留虚拟机配置、仅重装操作系统”的场景。以下是该选项的详细说明、核心特性、使用示例及注意事项：



一、基本语法与核心作用

1. 语法格式

bash

`virt-install --reinstall DOMAIN` [安装源选项 (`--cdrom/--location/--install`)] [其他可选安装参数]

**DOMAIN**: 必需参数, 指定要重新安装的现有虚拟机标识, 支持 3 种形式:

虚拟机名称 (如 `win10-vm`);

虚拟机 UUID (如 `5b33f1a0-78e2-4902-8a3b-9f7d6e8c4512`);

虚拟机运行 ID (仅当虚拟机处于运行 / 暂停状态时, 可通过 `virsh list` 查看)。

核心逻辑 (文档定义):

从 `libvirt` 读取 **DOMAIN** 的现有 XML 配置 (保留网络、存储、内存、vCPU 等非安装配置);

应用本次指定的 “安装源选项” (如 `--cdrom` 新 ISO、`--location` 新安装树);

引导虚拟机执行重新安装流程;

安装完成后, 自动恢复到原有 XML 配置 (仅保留安装阶段的系统盘修改, 其他配置不变)。

2. 与 “删除重建” 的核心区别

|    |                                                                         |                        |
|----|-------------------------------------------------------------------------|------------------------|
| 操作 | 手动删除 ( <code>virsh undefine</code> ) + 新建 ( <code>virt-install</code> ) |                        |
| 方式 | <code>--reinstall DOMAIN</code>                                         |                        |
| 核心 | 保留原有非安装配置 (网络、存储、vCPU 等), 无需重新配置                                        | 需重新指定所有参数 (易遗漏或配置错误)   |
| 优势 |                                                                         |                        |
| 适用 | 仅重装操作系统, 保留虚拟机硬件 / 网络配置                                                 | 彻底重建虚拟机 (需修改硬件 / 网络配置) |
| 场景 |                                                                         |                        |
| 数据 | 非系统盘数据保留 (如额外 <code>--disk</code> ), 系统盘被覆盖                             | 需手动备份所有磁盘, 否则数据丢失      |
| 保留 |                                                                         |                        |

二、关键特性 (结合文档细节)

仅处理 “安装相关选项”, 其他配置全忽略文档明确强调: “Only install related options are processed, all other VM configuration options like `--name`, `--disk`, etc. are completely ignored”—— 即本次命令中, 只有 `--cdrom`、`--location`、`--install`、`--unattended` 等 “安装源 /

安装流程相关选项”有效，--name（改名称）、--memory（改内存）、--disk（改磁盘）等非安装选项会被直接忽略，虚拟机原有配置不变。

CDROM 设备的特殊处理若配合 --cdrom 重新安装，文档规定：

优先使用虚拟机已挂载的 CDROM 设备（若存在），无需额外添加；

若虚拟机无现有 CDROM 设备，则自动添加一个永久 CDROM 设备（安装后不会删除，需手动用 `virsh attach-disk --detach` 移除）。

安装后配置“大致恢复”“revert to roughly the same starting XML”——安装完成后，除“系统盘内容被覆盖”外，虚拟机的 XML 配置（如网络接口、vCPU 数量、内存大小、其他磁盘挂载）会恢复到重新安装前的状态，无需担心配置丢失。

### 三、实际使用示例

#### 1. 基础场景：用新 ISO 重新安装 Windows 10 虚拟机

假设现有虚拟机 win10-vm 系统损坏，需用新的 Windows 10 ISO 重装，保留原有 4GB 内存、2 个 vCPU、50GB 系统盘配置：

```
bash
```

```
virt-install \
```

```
--reinstall win10-vm \ # 指定要重装的虚拟机名称
```

```
--cdrom /home/user/Win10_24H2.iso \ # 新的 Windows 10 ISO（仅安装源变化）
```

```
--unattended admin-password-file=/tmp/win-pass.txt # 无人值守安装（可选）# 注：
```

--memory、--vcpus、--disk 等非安装选项无需写，会沿用原有配置

#### 2. 进阶场景：用 HTTP 安装树重新安装 CentOS 7

现有 CentOS 7 虚拟机 centos7-vm，需从新的镜像源重装，同时启用串行控制台输出：

```
bash
```

```
virt-install \
```

```
--reinstall centos7-vm \ # 也可写 UUID: --reinstall
```

```
5b33f1a0-78e2-4902-8a3b-9f7d6e8c4512
```

```
--location https://mirror.tuna.tsinghua.edu.cn/centos/7/os/x86_64/ \ # 新的 HTTP 安  
装树
```

```
--extra-args "console=ttyS0 ks=https://myserver/new-centos7.ks" # 新的 kickstart 文  
件 + 串行控制台# 原有网络、磁盘、内存配置会自动保留
```

### 3. 特殊场景：重新安装时启用无人值守（保留用户配置）

用 `--unattended` 自动完成重装，同时保留原有虚拟机的用户密码策略（通过密码文件复用）：

```
bash
```

```
# 1. 复用原有密码文件（避免重新设置） echo "OldWinPass@2024" >
```

```
/tmp/win-pass.txtchmod 600 /tmp/win-pass.txt
```

```
# 2. 重新安装并自动配置
```

```
virt-install \
```

```
--reinstall win10-vm \
```

```
--cdrom /home/user/Win10_24H2.iso \
```

```
--unattended \
```

```
admin-password-file=/tmp/win-pass.txt \ # 沿用原有管理员密码
```

```
user-login=win-user # 保留原有普通用户名（沿用原有配置）
```

### 四、注意事项（避坑关键点）

DOMAIN 必须存在且状态建议为“关闭”

若 DOMAIN 不存在（名称 / UUID/ID 错误），会直接报错“domain not found”，需先用 `virsh list --all` 确认虚拟机标识；

若 DOMAIN 处于“运行中”，`virt-install` 会尝试强制重启并进入安装流程，但可能导致数据丢失（如未保存的文件），建议先手动关闭：`virsh shutdown win10-vm`。

系统盘数据会被覆盖，需提前备份--reinstall 仅保留“配置”，不保留“系统盘内容”——安装过程中会格式化系统盘并重新写入系统文件，若系统盘有重要数据，需提前用 `virsh snapshot-create-as` 创建快照，或用 `qemu-img convert` 备份磁盘镜像

非安装选项写了也无效，避免冗余不要在 `--reinstall` 命令中写 `--name`、`--memory`、`--vcpus`、`--network` 等非安装选项，即使写了也会被忽略，反而增加命令复杂度（如 `virt-install --reinstall win10-vm --memory 8192`，内存仍为原有 4096）。

安装后残留的 CDROM 设备需手动清理若重新安装时自动添加了 CDROM 设备（原有虚拟机无 CDROM），安装完成后该设备会保留在虚拟机配置中，后续启动可能会优先从 CDROM 引导（导致重复进入安装），需手动移除：

```
# 查看虚拟机现有设备，找到 CDROM 的目标设备（如 hda）virsh domblklist win10-vm#  
移除 CDROM 设备 virsh attach-disk --domain win10-vm --target hda --type cdrom --mode  
readonly --detach
```

## 五、常见问题解答

Q: 用 --reinstall 后，虚拟机名称变了？ A: 不可能 ——--name 是“非安装选项”，会被 --reinstall 忽略，虚拟机名称始终沿用原有配置，若名称变化，一定是误写了 --name 且用了错误的 DOMAIN（如重装了其他虚拟机）。

Q: 重新安装后，原有数据盘（额外 --disk）的数据不见了？ A: 检查是否误将“数据盘”设为了“系统盘”——--reinstall 仅覆盖“第一个 --disk（系统盘）”，若数据盘是后续添加的（非第一个 --disk），数据应保留；若数据盘被设为系统盘，需确认原有 XML 中 <disk> 的 boot.order 配置（系统盘通常 boot.order=1）。

Q: --reinstall 能否修改操作系统版本（如从 CentOS 7 升到 CentOS 8）？ A: 可以 ——只需指定 CentOS 8 的安装源（如 --location [https://mirror.centos.org/centos/8/BaseOS/x86\\_64/os/](https://mirror.centos.org/centos/8/BaseOS/x86_64/os/)），但需注意：硬件配置（如 --osinfo）会自动沿用原有 centos7.0，可能导致新系统适配问题，建议安装后手动更新 --osinfo:

```
bash
```

```
# 安装完成后，编辑 XML 并更新 osinfovirsh edit win10-vm# 将  
<os><type>...</type><osinfo>centos7.0</osinfo></os> 改为 <osinfo>centos8.0</osinfo>
```

### 3.2.12.--cloud-init

根据 virt-install 文档，--cloud-init 是一个针对“云镜像”的初始化配置选项——云镜像（如 Ubuntu Cloud、CentOS Stream Cloud）默认锁定登录账户（如 root 无密码、禁止密码登录），--cloud-init 通过生成 NoCloud 格式的 ISO 镜像（注入元数据、用户数据等配置），自动完成账户激活、密码设置、SSH 密钥注入等初始化操作，解决云镜像“无法直接登录”的问题。以下是该选项的详细说明、子参数、使用示例及注意事项：

#### 一、基本语法与核心作用

##### 1. 语法格式

```
bash
```

```
virt-install --import [--disk 云镜像路径] --cloud-init [子参数=值,...] [其他选项]
```

必需前提：需配合 `--import` 使用（云镜像已预装系统，无需安装，仅需初始化），且 `--disk` 需指定云镜像文件（如 `ubuntu-22.04-server-cloudimg-amd64.img`）；

核心逻辑（文档定义）：

自动生成 NoCloud 格式的临时 ISO 文件（包含 `cloud-init` 所需的配置数据）；

将该 ISO 作为 CDROM 设备挂载到虚拟机，仅在首次引导时生效；

虚拟机启动后，`cloud-init` 服务读取 ISO 中的配置，完成初始化（如设置 `root` 密码、注入 SSH 密钥）；

初始化完成后，该 ISO 设备会自动从虚拟机移除（避免后续引导重复初始化）。

2. 与 `--unattended` 的核心区别

| 对比维度 | <code>--cloud-init</code>                  | <code>--unattended</code>         |
|------|--------------------------------------------|-----------------------------------|
| 适用场景 | 云镜像初始化（已有系统，仅配置账户 / 网络）                    | 操作系统安装阶段自动化（从无到有装系统）              |
| 依赖前提 | 虚拟机已预装 <code>cloud-init</code> 服务（云镜像默认包含） | 依赖 <code>libosinfo</code> 的自动安装脚本 |
| 生效时机 | 仅虚拟机首次引导时                                  | 仅操作系统安装过程中                        |
| 核心目标 | 解决云镜像“无法登录”问题                              | 替代人工干预完成系统安装                      |

二、关键子参数详解（结合文档）

`--cloud-init` 支持多个子参数，覆盖“账户配置”“配置持久化”“自定义数据”等场景，未指定时默认执行 `root-password-generate=on,disable=on`（生成随机 `root` 密码 + 禁用后续 `cloud-init`）。常见子参数如下：

| 子参数                                    | 作用说明                                                                            | 适用场景               |
|----------------------------------------|---------------------------------------------------------------------------------|--------------------|
| <code>root-password-generate=on</code> | 自动生成随机 <code>root</code> 密码，打印到控制台并暂停 10 秒（方便用户复制），初始化后密码生效。                    | 临时测试场景，无需提前准备密码文件。 |
| <code>disable=on</code>                | 初始化完成后，禁用虚拟机内的 <code>cloud-init</code> 服务，避免后续引导时用，防止 <code>cloud-init</code> 重 | 所有场景推荐启            |

| 子参数                   | 作用说明                                                                  | 适用场景                                                   |
|-----------------------|-----------------------------------------------------------------------|--------------------------------------------------------|
|                       | 重置认证配置（如密码、密钥）。 复修改配置。                                                |                                                        |
| root-password-file=路径 | 从文件读取 root 密码（仅读取第一行内容，含空格、不含换行），避免密码泄露需将文件替代随机生成。                    | 需固定 root 密码的场景（如生产环境），避免密码泄露需将文件权限设为 600。              |
| meta-data=路径          | 直接指定自定义的 meta-data 文件（cloud-init 标准 名、实例 ID 等元数据格式），覆盖命令行中的其他元数据配置。   | 需要自定义主机的场景（如 instance-id: my-vm-01）。                   |
| user-data=路径          | 直接指定自定义的 user-data 文件（cloud-init 标准格式），覆盖命令行中的其他用户数据配置。               | 需要复杂初始化（如安装软件、执行脚本），示例：runcmd: [yum install -y nginx]。 |
| root-ssh-key=路径       | 注入指定的 SSH 公钥到 root 账户的 ~/.ssh/authorized_keys，实现 root 免密 SSH 登录。      | 远程管理场景，无需密码，安全性更高（推荐）。                                 |
| clouduser-ssh-key=路径  | 注入 SSH 公钥到“云镜像默认普通用户”（如 Ubuntu Cloud 的 ubuntu、CentOS Cloud 的 centos）。 | 云镜像默认禁用 root SSH 登录，需通过普通用户跳转的场景。                      |
| network-config=路径     | 直接指定自定义的 network-config 文件（cloud-init 标准格式），配置虚拟机网络（如静态 IP）。          | 需要固定 IP 而非 DHCP 的场景（默认云镜像用 DHCP 获取 IP）。                |

### 三、默认行为（文档重点强调）

若仅写 --cloud-init（不带任何子参数），文档明确其等效于：

```
bash
```

```
--cloud-init root-password-generate=on,disable=on
```

即：

自动生成随机 root 密码，打印到控制台（如 Root password: XXXXX-XXXXX-XXXXX），并暂停 10 秒；

初始化完成后，禁用 cloud-init 服务，避免后续引导重置密码；

不注入 SSH 密钥，不自定义网络，默认用 DHCP 获取 IP。

#### 四、实际使用示例

##### 1. 基础场景：云镜像简单初始化（生成随机 root 密码）

针对 Ubuntu 22.04 云镜像，仅需生成随机密码登录，无需复杂配置：

```
bash
```

```
virt-install \
```

```
--import \ # 导入云镜像，跳过安装
```

```
--name ubuntu2204-cloud \
```

```
--memory 2048 \
```

```
--vcpus 2 \
```

```
--disk /var/lib/libvirt/images/ubuntu-22.04-server-cloudimg-amd64.img \ # 云镜像路径
```

```
--osinfo ubuntu22.04 \ # 适配 Ubuntu 22.04 配置
```

```
--network network=default \ # 连接默认虚拟网络（DHCP 获取 IP）
```

```
--cloud-init # 默认：生成随机 root 密码 + 禁用后续 cloud-init
```

执行后，控制台会打印类似信息：Root password for this VM: abc123-def456，复制密码即可通过 virsh console 或 SSH 登录。

##### 2. 进阶场景：注入 SSH 密钥（免密登录 root）

生产环境推荐用 SSH 密钥免密登录，避免密码泄露风险：

```
bash
```

```
virt-install \
```

```
--import \
```

```
--name centos9-cloud \
```

```
--memory 1024 \
```

```
--vcpus 1 \
```

```
--disk /var/lib/libvirt/images/CentOS-Stream-9-x86_64-20240501.qcow2 \ # CentOS 云镜像
```

```
--osinfo centos-stream9 \

--network bridge=br0 \ # 连接物理网桥（获取局域网 IP）

--cloud-init \

    root-ssh-key=/home/user/.ssh/id_rsa.pub \ # 注入 root 的 SSH 公钥

    disable=on # 禁用后续 cloud-init
```

启动后，可直接通过 `ssh root@虚拟机 IP` 免密登录（无需密码）。

### 3. 复杂场景：自定义 `user-data`（初始化时安装软件）

通过 `user-data` 文件指定初始化脚本（如安装 `Nginx`），实现“启动即可用”：

```
bash
```

```
# 1. 创建自定义 user-data 文件（cloud-init 标准格式）cat > /tmp/user-data << EOF
```

```
#cloud-config
```

```
package_update: true
```

```
package_upgrade: true
```

```
packages:
```

```
- nginx
```

```
runcmd:
```

```
- systemctl enable --now nginx
```

```
EOF
```

```
# 2. 导入云镜像并注入 user-data
```

```
virt-install \
```

```
--import \
```

```
--name debian12-cloud \
```

```
--memory 4096 \
```

```
--vcpus 4 \
```

```
--disk /var/lib/libvirt/images/debian-12-genericcloud-amd64.img \ # Debian 云镜像
```

```
--osinfo debian12 \
```

```
--cloud-init \
```

```
    user-data=/tmp/user-data \ # 注入自定义 user-data
```

```
    root-password-file=/tmp/root-pass.txt \ # 从文件指定 root 密码（文件内容：
```

```
MyDeb12Pass!）
```



```
network-config=/tmp/network-config.yaml # 自定义静态 IP（可选）
```

虚拟机启动后，会自动更新系统、安装 Nginx 并启动服务，无需手动操作。

## 五、注意事项（避坑关键点）

仅云镜像支持，普通 ISO 无效--cloud-init 依赖虚拟机内预装的 cloud-init 服务，普通操作系统 ISO（如 Ubuntu 桌面版 ISO）默认不含该服务，使用后会无任何效果，需确保 --disk 指定的是“云镜像”（文件名通常含 cloudimg 或 cloud）。

SSH 密钥文件的权限要求注入 SSH 密钥时（root-ssh-key/clouduser-ssh-key），公钥文件（如 id\_rsa.pub）的权限需为 644 或更严格（避免 others 有写权限），否则 cloud-init 会拒绝读取，导致密钥注入失败。

NoCloud ISO 的自动移除机制文档提到“该设备仅在首次引导时挂载”，初始化完成后会自动从虚拟机配置中移除，无需手动删除；若首次引导未完成初始化（如 cloud-init 服务异常），ISO 会保留，需检查 /var/log/cloud-init.log 排查问题。

与 --unattended 的冲突处理若同时使用 --unattended（安装系统）和 --cloud-init（初始化云镜像），仅 --cloud-init 生效（因 --import 与 --unattended 互斥，--import 会跳过安装阶段），建议根据场景二选一：

装系统自动化：用 --unattended；

云镜像初始化：用 --import + --cloud-init。

## 六、常见问题解答

Q：执行 --cloud-init 后，虚拟机仍无法用 root 密码登录？A：原因可能有 3 点：① 云镜像默认禁用 root SSH 登录（需通过 user-data 开启：ssh\_pwauth: true + allow\_root: true）；② 密码文件路径错误或权限不足（需 chmod 600 /tmp/root-pass.txt）；③ cloud-init 未执行（检查 /var/log/cloud-init.log 看是否有报错）。

Q: 注入 SSH 密钥后, 仍需输入密码? A: 检查: ① 公钥是否正确注入到 `root/.ssh/authorized_keys` (登录后查看该文件); ② 虚拟机 SSH 配置是否禁用密钥登录 (`/etc/ssh/sshd_config` 中 `PasswordAuthentication` 设为 `no` 不影响密钥登录, `PubkeyAuthentication` 需设为 `yes`)。

Q: 如何验证 `cloud-init` 是否执行成功? A: 登录虚拟机后, 检查:  
① `/var/log/cloud-init.log` 无报错 (最后一行含 `cloud-init[xxxx]: finished at ...`); ② 配置是否生效 (如密码登录、SSH 密钥、安装的软件); ③ `cloud-init` 服务是否已禁用 (`systemctl status cloud-init` 显示 `inactive`)。

### 3.2.13.--boot

`--boot hd,cdrom,menu=on`

`--boot init=/sbin/init` (针对容器)

`--idmap IDMAP` 为 LXC 容器启用用户名称空间。示例:

`--idmap uid.start=0,uid.target=1000,uid.count=10`

### 3.2.14.--os-variant OS\_VARIANT, --osinfo OS\_VARIAN

要安装到客户机中的操作系统。

这用于确定最佳默认值, 例如 `VirtIO`。

示例值: `fedora29, rhel7.0, win10, ...`

使用 `'--osinfo list'` 查看完整列表。

### 3.2.15.--disk

一、核心: 存储定位与创建 (最常用)

这类子参数用于指定 “存储在哪里、是否新建、多大尺寸”, 是配置磁盘的基础, 对应文档中 `--disk` 章节的 `path/pool/vol/size` 等核心选项。

| 子参数               | 功能说明                                 | 示例                                               |
|-------------------|--------------------------------------|--------------------------------------------------|
| <code>path</code> | 指定存储介质路径 (现有文件 / 块设备, 或新建存储的目标路径)。新建 | <code>path=/var/lib/libvirt/images/vm.img</code> |

| 子参数                 | 功能说明                                                                       | 示例                                                    |
|---------------------|----------------------------------------------------------------------------|-------------------------------------------------------|
|                     | 时需配合 <code>size</code> 。                                                   |                                                       |
|                     | 指定现有 <code>libvirt</code> 存储池                                              |                                                       |
| <code>pool</code>   | （如 <code>default</code> 池），在池中新建存储，<br>需配合 <code>size</code> 。             | <code>pool=default,size=20</code> （在默认池建<br>20GiB 磁盘） |
| <code>vol</code>    | 指定现有 <code>libvirt</code> 存储卷（格式：存储<br>池名/卷名），直接复用已有存储。                    | <code>vol=default/vm-disk</code>                      |
| <code>size</code>   | 新建存储的大小（单位：GiB），仅<br>当 <code>path</code> 不存在或指定 <code>pool</code> 时有效。     | <code>size=30</code> （创建 30GiB 磁盘）                    |
| <code>sparse</code> | 是否“稀疏分配”（不立即占用全部主<br>机磁盘空间），可选 <code>yes</code> （默认）/ <code>no</code> 。更稳） | <code>sparse=no</code> （立即分配全部空间，性能                   |
| <code>format</code> | 磁盘镜像格式（新建时指定，或覆盖<br>现有镜像的自动检测），支<br>持 <code>qcow2/raw/vmdk</code> 等。       | <code>format=qcow2</code> （推荐，支持快照）                   |

## 二、设备类型与总线配置（影响兼容性 / 性能）

这类子参数控制“虚拟机识别的磁盘类型”和“数据传输总线”，直接影响存储 I/O 性能（如 `virtio` 总线比 `ide` 快）。

| 子参数                           | 功能说明                                                                                                                     | 示例                                                               |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
|                               | 磁盘设备类型，可选 <code>disk</code> （默认，<br>硬盘）/ <code>cdrom</code> （光盘）/ <code>floppy</code> （软盘）<br>/ <code>lun</code> （逻辑单元）。 | <code>device=cdrom</code> （将 ISO 作<br>为光盘挂载）                     |
| <code>bus/target.bus</code>   | 磁盘连接的总线类型，可选 <code>virtio</code><br>（推荐，高性能）/ <code>ide</code> （兼容性好）<br>/ <code>sata/scsi</code> 。                      | <code>bus=virtio</code> （KVM/QEMU 首<br>选）                        |
| <code>target.dev</code>       | 虚拟机内显示的设备名<br>（如 <code>vda/hda</code> ），仅为标识，不影响功<br>能。                                                                  | <code>target.dev=vda</code> （客户机内<br>显示为 <code>/dev/vda</code> ） |
| <code>target.removable</code> | 是否为可移除设备（如光盘），可<br>选 <code>yes/no</code> ，仅 <code>device=cdrom/floppy</code> 有                                           | <code>target.removable=yes</code>                                |

| 子参数 | 功能说明 | 示例 |
|-----|------|----|
|-----|------|----|

用。

三、性能与缓存优化（控制 I/O 效率）

这类子参数用于调整存储 I/O 的缓存策略、资源限制、错误处理，适合对性能或稳定性有要求的场景（如数据库虚拟机）。

| 子参数                 | 功能说明                                                                                                                                                                             | 示例                                         |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| cache               | 主机缓存模式，影响读写性能： <ul style="list-style-type: none"> <li>- writeback（默认，读写缓存，性能好）</li> <li>- writethrough（读缓存，数据安全）</li> <li>- none（无缓存，低延迟）</li> <li>- unsafe（激进缓存，风险高）</li> </ul> | cache=writethrough（数据库场景，防丢数据）             |
| driver.discard      | 是否支持“丢弃”（TRIM/UNMAP，释放客户机空闲空间到主机），可选 unmap（支持）/ignore（忽略），仅 QEMU/KVM 支持。                                                                                                         | driver.discard=unmap（配合 qcow2 释放空间）        |
| driver.io           | I/O 后端模式，可选 threads（线程）/native（原生）/io_uring（Linux 5.1+，高性能）。                                                                                                                     | driver.io=io_uring（高 I/O 场景首选）             |
| iotune.*            | I/O 资源限制（避免单个虚拟机占用过多主机 I/O），如： <ul style="list-style-type: none"> <li>- iotune.read_bytes_sec: 每秒最大读字节</li> <li>- iotune.write_iops_sec: 每秒最大写 I/O 次数</li> </ul>                 | iotune.read_bytes_sec=10485760（限制读 10MB/s） |
| driver.error_policy | 遇到写错误时的处理策略： <ul style="list-style-type: none"> <li>- stop（默认，停止虚拟机）</li> </ul>                                                                                                  | driver.error_policy=stop（重要数据场景）           |

| 子参数 | 功能说明                                  | 示例 |
|-----|---------------------------------------|----|
|     | - ignore（忽略错误）- enospace<br>（空间不足时处理） |    |

#### 四、高级特性（快照、共享、后端镜像）

这类子参数用于特殊场景（如差分镜像、多机共享磁盘、网络存储），对应文档中 `backing_store/shareable/snapshot` 等选项。

| 子参数                         | 功能说明                                                                                                     | 示例                                                       |
|-----------------------------|----------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| <code>backing_store</code>  | 指定“后端镜像”（基于现有镜像创建差分镜像，仅写入增量数据，节省空间）。<br>后端镜像的格式                                                          | <code>backing_store=/img/base.qcow2</code> （基于基础镜像创建差分盘） |
| <code>backing_format</code> | （若 <code>backing_store</code> 未自动识别）。                                                                    | <code>backing_format=qcow2</code>                        |
| <code>snapshot</code>       | 磁盘在快照时的默认行为<br>（如 <code>external</code> 外部快照、 <code>internal</code> 内部快照），依赖 <code>hypervisor</code> 支持。 | <code>snapshot=external</code>                           |
| <code>shareable</code>      | 是否允许多个虚拟机共享该磁盘（如集群场景），可选 <code>yes/no</code> （默认）。                                                       | <code>shareable=yes</code> （多虚拟机共享数据盘）                   |
| <code>readonly</code>       | 设为只读磁盘（防止客户机修改数据），可选 <code>yes/no</code> 。                                                               | <code>readonly=yes</code> （挂载只读 ISO）                     |
| <code>serial</code>         | 磁盘序列号（Linux 客户机中对应 <code>/dev/disk/by-id</code> 符号链接），用于固定设备标识。                                          | <code>serial=WD-WMAP9A966149</code>                      |

#### 五、存储源配置（网络存储 / 特殊设备）

这类子参数用于配置“非本地文件”的存储源（如 iSCSI、NFS、物理块设备），对应文档中 `source.*` 相关说明。

| 子参数 | 功能说明 | 示例 |
|-----|------|----|
|-----|------|----|

| 子参数                   | 功能说明                                                                         | 示例                                                                |
|-----------------------|------------------------------------------------------------------------------|-------------------------------------------------------------------|
|                       | 指定主机物理块设备                                                                    |                                                                   |
| source.dev            | （如 <code>/dev/sdb1</code> ），直接作为虚拟机磁盘。                                       | <code>source.dev=/dev/sdb1,format=raw</code>                      |
|                       | 网络存储协议，                                                                      |                                                                   |
| source.protocol       | 如 <code>iscsi/nfs/gluster</code> ，配合 <code>source.host</code> 指定服务器。         | <code>source.protocol=iscsi,source.host0.name=iscsi-server</code> |
|                       | 网络存储服务器地址（支持多个，如 <code>iSCSI</code> 多路径）。                                    |                                                                   |
| source.host[0-9].name |                                                                              | <code>source.host0.name=192.168.1.100</code>                      |
|                       | 网络存储卷名（如 <code>iSCSI</code> 的 <code>LUN</code> 名、 <code>Gluster</code> 的卷名）。 |                                                                   |
| source.volume         |                                                                              | <code>source.volume=vm-lun1</code>                                |

## 六、XML 自定义与地址配置（精细控制）

这类子参数用于直接修改虚拟机 XML 中 `<disk>` 元素的配置（适合 `virt-install` 默认不支持的高级场景），对应文档中 `address.*/xpath.*` 选项。

| 子参数                | 功能说明                                                                   | 示例                                            |
|--------------------|------------------------------------------------------------------------|-----------------------------------------------|
|                    | 磁盘设备在主机总线上的地址（如 <code>bus</code> 总线号、 <code>slot</code> 插槽号），用于固定设备位置。 | <code>address.bus=0,address.slot=4</code>     |
| address.*          |                                                                        |                                               |
| xpath[0-9]*.create | 用 XPath 语法创建 XML 节点（如布尔型 <code>&lt;readonly/&gt;</code> ）。读节点          | <code>xpath.create=./readonly</code> （添加只读节点） |

| 子参数             | 功能说明                                       | 示例                              |
|-----------------|--------------------------------------------|---------------------------------|
| xpath[0-9]*.set | 用 XPath 语法修改 XML 属性（如设置总线类型）。              | xpath.set=../target/@bus=virtio |
| alias.name      | 磁盘设备的别名（XML 中 <alias name="xxx"/>），用于标识设备。 | alias.name=disk0                |

## 七、常见组合示例（实际应用）

创建 10GiB qcow2 磁盘（默认池，virtio 总线，支持 TRIM）

```
bash
```

```
virt-install --disk pool=default,size=10,format=qcow2,bus=virtio,driver.discard=unmap
```

复用现有磁盘 + 设为只读（IDE 总线，光盘设备）

```
bash
```

```
virt-install --disk path=/home/user/ubuntu.iso,device=cdrom,bus=ide,readonly=yes
```

基于基础镜像创建差分盘（节省空间）

```
bash
```

```
virt-install --disk
```

```
path=/img/vm1.img,backing_store=/img/base.qcow2,backing_format=qcow2
```

限制 I/O 带宽（读 10MB/s，写 5MB/s）

```
bash
```

```
virt-install --disk
```

```
path=/img/vm.img,iotune.read_bytes_sec=10485760,iotune.write_bytes_sec=5242880
```

关键注意事项

单位问题: size 单位是 GiB (1GiB=1024MiB), iotune.\* 单位是 字节 (如 10MB/s=10×1024×1024=10485760 字节)。

格式兼容性: format=vmdk 适合与 VMware 互操作, raw 适合高性能场景 (无压缩 / 快照), qcow2 是 KVM/QEMU 首选 (支持快照、差分镜像)。

权限问题: path 指定的路径需让 virt-install 运行用户 (如 root) 有读写权限, 否则会报 “无法创建 / 访问磁盘” 错误

### 3.2.16. --network

NETWORK, --network NETWORK

配置客户机网络接口。例如:

--network bridge=mybr0

--network network=my\_libvirt\_virtual\_net

--network network=mynet,model=virtio,mac=00:11...

--network none

--network help

### 3.2.17. --graphics

配置客户机显示设定。示例:

--graphics spice

--graphics vnc,port=5901,listen=0.0.0.0

--graphics none

### 3.2.18. 其它

--controller CONTROLLER

配置客户机控制器设备。示例:

--controller type=usb,model=qemu-xhci

--controller type=scsi,model=virtio-scsi

--input INPUT

配置客户机输入设备。例如:



--input tablet

--input keyboard,bus=usb

--serial SERIAL      配置客户机串口设备

--parallel PARALLEL    配置客户机并口设备

--channel CHANNEL      配置客户机通信通道

--console CONSOLE      配置文本控制台连接主机与客户机

--hostdev HOSTDEV      配置物理 USB/PCI 等主机设备与客户机共享

--filesystem FILESYSTEM

传递主机目录到客户机。例如：

--filesystem /my/source/dir,/dir/in/guest

--filesystem template\_name/,type=template

--sound [SOUND]      配置客户机声音设备仿真

--audio AUDIO      配置声音设备的主机音频后端

--watchdog WATCHDOG    配置客户机看门狗设备

--video VIDEO      配置客户机视频硬件。

--smartcard SMARTCARD

配置客户机智能卡设备。例如：

--smartcard mode=passthrough

--redirdev REDIRDEV    配置客户机重定向设备。例如：

--redirdev usb,type=tcp,server=192.168.1.1:4000

--memballoon MEMBALLOON

配置客户机 memballoon 设备。例如：

--memballoon model=virtio

--tpm TPM      配置客户机 TPM 设备。例如：

--tpm /dev/tpm

--rng RNG      配置客户机 RNG 设备。示例：

--rng /dev/urandom

--panic PANIC      配置客户机 panic 设备。例如：

--panic default

--shm SHMEM      配置客户机共享内存设备。示例：

--shmем name=shmем0

--memdev MEMDEV      配置客户机内存设备。示例：  
                           --memdev dimm,target.size=1024

--vsock VSOCK      配置客户机 vsock 套接字。示例：  
                           --vsock cid.auto=yes  
                           --vsock cid.address=7

--iommu IOMMU      配置 IOMMU 设备。示例：  
                           --iommu model=intel,driver.aw\_bits=48

客户机配置选项:

--iothreads IOTHEADS      设定域 <iothreads> 和 <iothreadids> 配置。

--seclabel SECLABEL, --security SECLABEL      设定域 seclabel 配置。

--keywrap KEYWRAP      设定客户机以执行 S390 加密密钥管理操作。

--cputune CPUTUNE      调整域进程的 CPU 参数。

--numatune NUMATUNE      为域进程调整 NUMA 策略。

--memtune MEMTUNE      为域进程调整内存策略。

--blkio tune BLKIOTUNE      为域进程调整 blkio 策略。

--memorybacking MEMORYBACKING      为域进程设置内存后备策略。例如：  
                           --memorybacking hugepages=on

--features FEATURES      设定域 <features> XML。示例：  
                           --features acpi=off  
                           --features apic=on,apic.eoi=on

--clock CLOCK      设置域 <clock> XML。例如：  
                           --clock offset=localtime,rtc\_tickpolicy=catchup

--pm PM      配置 虚拟系统 电源管理功能

--events EVENTS      配置 虚拟系统 生命周期管理策略

--resource RESOURCE      配置 虚拟系统 资源分区(cgroups)

`--sysinfo SYSINFO`      配置 SMBIOS 系统信息。示例：  
  
    `--sysinfo host`  
  
    `--sysinfo bios.vendor=MyVendor,bios.version=1.2.3,...`

`--qemu-commandline QEMU_COMMANDLINE`  
  
    将参数直接传递给 QEMU 模拟器。示例：  
  
    `--qemu-commandline='-display gtk,gl=on'`  
  
    `--qemu-commandline env=DISPLAY=:0.1`

`--launchSecurity LAUNCHSECURITY, --launchsecurity LAUNCHSECURITY`  
  
    配置虚拟系统启动安全性设置（例如：SEV 内存加密）。示

例：

`--launchSecurity sev`

虚拟化平台选项：

`-v, --hvm`              这个客户机应该是一个全虚拟化客户机  
  
`-p, --paravirt`        这个客户机应该是一个半虚拟化客户机  
  
`--container`            这个客户机应该是一个容器客户机  
  
`--virt-type VIRT_TYPE`  
  
                        要使用的虚拟机管理程序名称 (kvm, qemu, xen, ...)  
  
`--arch ARCH`            模拟 CPU 架构  
  
`--machine MACHINE`      机器类型为仿真类型

其它选项：

`--autostart`            主机启动时自动启动域。  
  
`--transient`            创建一个临时域。  
  
`--destroy-on-exit`      关闭控制台查看器后，强制关闭域电源。  
  
`--wait [WAIT]`          请等待数分钟以便完成安装。

`--autoconsole AUTOCONSOLE`  
  
    配置客户机控制台自动连接。示例：  
  
    `--autoconsole text`  
  
    `--autoconsole graphical`  
  
    `--autoconsole none`  
  
`--noautoconsole`        不要自动尝试连接到客户端控制台

- `--noreboot` 安装完成后不启动客户机。
- `--print-xml [XMLONLY]` 打印生成的 XML 域，而不是创建客户机。
- `--dry-run` 运行安装程序，但不创建设备或定义客户机。
- `--check CHECK` 启用或禁用验证检查。例如：
  - `--check path_in_use=off`
  - `--check all=off`
- `-q, --quiet` 抑制非错误输出
- `-d, --debug` 输入故障排除信息

使用 `'--option=?'` 或 `'--option help'` 来查看可用的子选项

3.3.脚本方式安装

`virt-install` 的命令可以写成脚本文件，生成速度比手敲命令更加快，更加内容容易错理解，不容易敲错，效率更加高。

4.虚拟机配置

4.1.虚拟机基础信息配置

| 内容          | 命令                                    | 说明                                                                                                   |
|-------------|---------------------------------------|------------------------------------------------------------------------------------------------------|
| 虚拟机名称变更     | <code>virsh domrename</code>          | 修改虚拟机在 libvirt 中注册的名字, 不是虚拟机中操作系统的计算机名称                                                              |
| 虚拟机 UUID 变更 | <code>visrsh edit</code><br>编辑 xml 文件 | 没有必要变更，可以通过 <code>virsh edit</code> 进去，找到 UUID 字段，直接删除，系统会自动生成                                       |
| 虚拟机虚拟方式     | <code>visrsh edit</code><br>编辑 xml 文件 | 常见的 domain type 包括 <code>kvm</code> (硬件加速)、 <code>qemu</code> (纯软件模拟)、 <code>xen</code> 等，需根据你的虚拟化环境 |

|          |                |                      |
|----------|----------------|----------------------|
|          |                | 选择支持的类型，默认为 kmv，不用设置 |
| 虚拟机描述或标题 | virsh desc     | 设置虚拟机描述或标题           |
| 虚拟机元数据   | virsh metadata | 显示或设置虚拟机的自定义 Xml 元数据 |

4.1.1.虚拟机名称变更

```
<name>ubuntuv</name>
```

名称变更的时候/etc/libvirt/qemu 下的 xml 文件名称一起会变更，包括 xml 文件中的字段。

注：如果直接用 vim 编辑 xml 文件编辑，并 xml 的文件名词改变，都不会影响虚拟机在 libvirt 的

如果直接用 virsh edit 虚拟机名，只更改名称，是无法保存的，因为会被认为是新注册一台虚拟机，需要把 UUID 字段直接删除才能保存，它会自动生成一个唯一的 UUID，不要手工生成，这样会在管理命令中发现新注册了一个虚拟机，老的还在。

4.1.2.虚拟机 UUID 变更

```
<uuid>e20d65bb-2f88-4be2-80f0-b107f9d64c44</uuid>
```

虚拟机的 UUID 在全局范围内是唯一的，是由 KVM 物理机自动生成的，正常情况下不用修改，通过 virsh edit 进去编辑，删除 UUID 内容，都会自动生成。

UUID 修改需要先导出虚拟机 XML 配置，修改后再重新定义：

```
bash
```

```
# 导出 XML 配置到文件 virsh dumpxml <虚拟机名称> > vm_config.xml
```

```
# 编辑 XML 文件，找到并修改 <uuid> 字段# 可以使用 uuidgen 命令生成新的
```

```
UUIdnano vm_config.xml
```

```
# 先删除原虚拟机定义（不会删除磁盘文件）virsh undefine <虚拟机名称>
```

```
# 使用修改后的 XML 重新定义虚拟机 virsh define vm_config.xml
```

注意事项：

修改 UUID 时，需要确保新 UUID 格式正确（36 个字符，包含连字符）

可以用 `uuidgen` 命令生成符合标准的新 UUID

操作前建议备份虚拟机 XML 配置

修改后可以用 `virsh dominfo <新名称>` 验证是否生效

例如，完整的修改流程：

```
bash
```

```
# 生成新 UUIDnew_uuid=$(uuidgen)
```

```
# 导出配置 virsh dumpxml old_vm > old_vm.xml
```

```
# 替换 UUID（使用 sed 命令）sed -i "s/<uuid>.*</uuid>/<uuid>$new_uuid</uuid>/"
```

```
old_vm.xml
```

```
# 替换名称（如果同时需要修改名称）sed -i
```

```
"s/<name>.*</name>/<name>new_vm</name>/" old_vm.xml
```

```
# 重新定义虚拟机 virsh undefine old_vmvirsh define old_vm.xml
```

#### 4.1.3.虚拟机虚拟方式

```
<domain type='kvm'>
```

常见的 domain type 包括 `kvm`（硬件加速）、`qemu`（纯软件模拟）、`xen` 等，需根据你的虚拟化环境选择支持的类型，默认为 `kvm`，不用设置。

从 `kvm` 改为 `qemu` 可能会导致性能下降（因为失去硬件加速），反之（`qemu` 改 `kvm`）需要确保主机支持 KVM 虚拟化。

操作前建议备份 XML 配置文件，以防配置错误。

通过上述步骤，即可完成虚拟机 domain type 的变更。

要修改虚拟机的 domain type（如从 `kvm` 变更其他类型，如 `qemu`），需要通过编辑虚拟机的 XML 配置文件来实现，具体步骤如下：

操作步骤

确保虚拟机处于关闭状态：

```
virsh shutdown <虚拟机名称>
```

导出当前 XML 配置：

```
virsh dumpxml <虚拟机名称> > vm_config.xml
```

编辑 XML 文件修改 domain type：

```
nano vm_config.xml
```

找到开头的 <domain> 标签，修改 type 属性值：

```
xml
```

```
<!-- 原配置 -->
```

```
<domain type='kvm'>
```

```
<!-- 修改为其他类型，例如 qemu -->
```

```
<domain type='qemu'>
```

删除原虚拟机定义（保留磁盘文件）：

```
virsh undefine <虚拟机名称>
```

使用修改后的 XML 重新定义虚拟机：

```
virsh define vm_config.xml
```

验证修改结果：

```
virsh dumpxml <虚拟机名称> | grep "<domain type"
```

#### 4.1.4.虚拟机描述或者标题

```
<name>fnos</name>
```

```
<uuid>a5d703c4-203d-4f93-a2e5-38a753f8f755</uuid>
```

```
<title>fnos</title>
```

```
<description>www</description>
```

name 是给系统看的，title 是给用户看的，description 是简单描述干啥的。

virsh desc 命令用于查看或修改虚拟机（域）的描述信息（包括描述文本和标题）。你列出的是该命令的可选参数，具体说明如下：

--config: 针对虚拟机的配置文件进行操作（修改后需重启生效）

--current: 对当前活动的虚拟机状态进行操作

--domain <名称/UUID>: 指定要操作的虚拟机（必选参数，需提供虚拟机名称或 UUID）

--edit: 编辑虚拟机的描述信息（会打开编辑器）

--live: 对运行中的虚拟机进行实时操作（修改立即生效）

--new-desc <描述文本>: 设置新的描述文本（替换原有描述）

#### 4.1.5.虚拟机元数据

```
<metadata>
  <libosinfo:libosinfo
xmlns:libosinfo="http://libosinfo.org/xmlns/libvirt/domain/1.0">
    <libosinfo:os id="http://libosinfo.org/linux/2022"/>
  </libosinfo:libosinfo>
</metadata>
```

`virsh metadata` 命令用于管理虚拟机的自定义元数据，允许用户为虚拟机添加、查询、修改或删除自定义的键值对信息（如所有者、部门、用途标签等），这些元数据不会影响虚拟机的运行，仅作为附加信息用于管理和识别。

基本语法

bash

# 查询元数据 `virsh metadata <虚拟机名/ID/UUID> [--key <键名>] [--uri <命名空间>]`

# 设置元数据 `virsh metadata <虚拟机名/ID/UUID> --key <键名> --value <值> [--uri <命名空间>] [--edit]`

# 删除元数据 `virsh metadata <虚拟机名/ID/UUID> --key <键名> [--uri <命名空间>] --delete`

核心参数说明

`<虚拟机名/ID/UUID>`: 指定目标虚拟机。

`--key <键名>`: 自定义元数据的键（如 `owner`、`department`）。

`--value <值>`: 键对应的取值（如 `IT 部门`、`张三`）。

`--uri <命名空间>`: 可选，用于区分不同来源的元数据（避免键名冲突），默认使用 `http://libvirt.org/schemas/libvirt-metadata-1.0`。

`--edit`: 以编辑模式修改元数据（适合复杂值）。

`--delete`: 删除指定键的元数据。

使用示例

1. 添加 / 设置元数据

bash

# 为虚拟机 `fnos` 添加 `"owner"` 元数据(所有者)  
`virsh metadata fnos --key owner --value "张三"`



```
# 添加 "department" 元数据（部门），并指定命名空间 virsh metadata fnos --key
department --value "IT 部" --uri "http://company.com/metadata"
```

## 2. 查询元数据

```
bash
```

```
# 查询 fnos 的所有元数据 virsh metadata fnos
```

```
# 仅查询 "owner" 键的元数据 virsh metadata fnos --key owner
```

```
# 查询指定命名空间下的 "department" 元数据 virsh metadata fnos --key department
--uri "http://company.com/metadata"
```

查询输出示例（XML 格式）：

```
xml
```

```
<metadata xmlns="http://libvirt.org/schemas/libvirt-metadata-1.0">
```

```
  <owner>张三</owner></metadata>
```

## 3. 修改元数据

```
bash
```

```
# 修改 "owner" 的值为 "李四" virsh metadata fnos --key owner --value "李四"
```

```
# 用编辑模式修改（适合多行文本或复杂内容） virsh metadata fnos --key description --edit
```

## 4. 删除元数据

```
bash
```

```
# 删除 "owner" 元数据 virsh metadata fnos --key owner --delete
```

```
# 删除指定命名空间的 "department" 元数据 virsh metadata fnos --key department --uri
"http://company.com/metadata" --delete
```

## 特点与用途

自定义灵活性：元数据的键和值完全由用户定义，可根据管理需求添加任意信息（如资产编号、维护周期、联系方式等）。

不影响虚拟机运行：元数据仅存储在虚拟机的 XML 配置中，不会对虚拟机的性能、配置或运行状态产生任何影响。

管理与自动化：在批量管理或脚本中，可通过元数据对虚拟机进行分类、筛选（如 `grep` 筛选出属于“IT 部”的虚拟机）。

命名空间隔离：通过 `--uri` 可区分不同场景的元数据（如公司内部标签和第三方工具标签），避免键名冲突。

注意事项

元数据存储在售机的 XML 配置中，virsh dumpxml 命令可查看完整的元数据信息。

若虚拟机被克隆或迁移，元数据会随 XML 配置一起复制，需手动调整克隆机的元数据以避免混淆。

virsh metadata 是虚拟机精细化管理的实用工具，尤其适合多虚拟机环境中对设备进行分类、标记和备注，提升管理效率。

4.2.内存与 VCPU 资源配置

| 内容        | 命令                             | 说明                                                                      |
|-----------|--------------------------------|-------------------------------------------------------------------------|
| 虚拟机最大内存配置 | virsh setmaxmem                | 用于管理 KVM 虚拟机最大内存限制的命令行工具，它允许你修改虚拟机可以使用的最大内存量。                           |
| 虚拟机当前内存配置 | virsh setmem                   | 用于调整虚拟机的内存大小，你列出的是该命令的核心参数                                              |
| 虚拟机内存气球配置 | visrsh edit<br>编辑 xml 文件       | 用于动态调整虚拟机的内存做前提设置                                                       |
| 虚拟机内存限制配置 | virsh memtune                  | 用于调整内存的 hard-limit 、 soft-limit 、<br>--swap-hard-limit<br>min-guarantee |
| 虚拟机大页缓存配置 | visrsh edit<br>编辑 xml 文件       |                                                                         |
| 虚拟机内存共享配置 | 物理机开启                          | 多虚拟机场景部署                                                                |
| 虚拟机内存深度调优 | virsh<br>update-memory-devicve | 是用于更新虚拟机内存设备配置的命令，主要用于<br>精细化调整虚拟机的内存相关设备参数，比如内存<br>插槽等。                |

|                 |                          |                                           |
|-----------------|--------------------------|-------------------------------------------|
| 虚拟机 NUMA 内存绑定配置 | virsh edit<br>编辑 xml 文件  | 设置内存与 vcpu 的绑定                            |
| 虚拟机 VCPU 配置     | virsh setcpus            | 用于精确控制虚拟机 vCPU 的调整方式                      |
| 虚拟机 VCPU 热插拔    | virsh setcpu             | 用于对 vcpu 进行增加或下线                          |
| 虚拟机 VCPU 亲和性    | vish cpupin              | 用于对 vcpu 与物理 cpu 进行绑定                     |
| 虚拟机 VCPU 缓存优化   | visrsh edit<br>编辑 xml 文件 | 要求 CPU 模式为直通                              |
| 虚拟机 VCPU 调度策略   | virsh schedinfo          | 主要通过调整调度权重、周期和配额等参数，实现对不同虚拟机的 CPU 资源优先级管理 |
| 虚拟机 VCPU 物理拓扑   | visrsh edit<br>编辑 xml 文件 | 模拟物理 CPU 的插槽、核心、线程结构                      |

#### 4.2.1.虚拟机最大内存配置

```
<memory unit='KiB'>10485760</memory>
```

virsh setmaxmem 是一个用于管理 KVM 虚拟机最大内存限制的命令行工具，它允许你修改虚拟机可以使用的最大内存量。

基本语法

bash

```
virsh setmaxmem <虚拟机名称/ID> <内存大小> [--config] [--live] [--current]
```

主要参数说明

<虚拟机名称/ID>: 指定目标虚拟机的名称或 ID

<内存大小>: 要设置的最大内存量，可使用单位（如 K、M、G），例如 2G 表示 2GB

--config: 仅修改配置文件，下次启动时生效

--live: 对运行中的虚拟机生效（需要虚拟机支持）

--current: 对当前运行状态（无论是运行中还是已定义）生效

示例

为名为 vm1 的虚拟机设置最大内存为 4GB，仅修改配置（下次启动生效）：

```
virsh setmaxmem vm1 4G --config
```

为运行中的 vm2 虚拟机动态调整最大内存为 8GB：

```
virsh setmaxmem vm2 8G --live
```

同时修改当前运行状态和配置文件：

```
virsh setmaxmem vm3 2G --live --config
```

注意事项

最大内存 (maxmem) 限制了虚拟机可以使用的内存上限，而实际使用的内存由 virsh setmem 控制

有些虚拟机可能不支持动态修改最大内存，需要先关闭虚拟机

最大内存不能小于当前分配给虚拟机的内存

修改后可以用 virsh dominfo <虚拟机名称> 验证设置是否生效

使用这个命令可以灵活调整虚拟机的内存资源分配，适应不同的应用场景需求。

#### 4.2.2.虚拟机当前内存配置

```
<currentMemory unit='KiB'>2097152</currentMemory>
```

virsh setmem 命令用于调整虚拟机的内存大小，你列出的是该命令的核心参数，具体说明如下：

--domain <名称/UUID>：指定要调整内存的虚拟机（必选参数，需提供虚拟机名称或 UUID）

--size <大小>：设置新的内存大小（需指定单位，如 1024M 表示 1024MB，2G 表示 2GB）

--live：对运行中的虚拟机实时调整内存（立即生效，需虚拟机支持动态内存调整）

--config：仅修改虚拟机的配置文件（下次启动时生效）

--current：根据虚拟机当前状态生效（运行中则实时调整，未运行则修改配置）

#### 4.2.3.虚拟机内存气球配置

```
<memballoon model='virtio'> <!-- virtio 模式性能最佳 -->
```

```
<address type='pci' bus='0x00' slot='0x05' />
```

```
</memballoon>
```

在 KVM 虚拟化中，内存气球（`memballoon`）的设置方式主要根据驱动模型和功能需求划分，常见的有以下几种方式，分别适用于不同场景：

### 一、按驱动模型划分（核心设置方式）

内存气球的核心是通过不同的驱动模型实现，在 XML 配置中通过 `<memballoon model='... '>` 指定，主要有 3 种模型：

#### 1. virtio 模型（推荐）

特点：基于 `virtio` 半虚拟化驱动，性能最佳，支持动态内存调整（`ballooning`）和内存统计。

适用场景：所有支持 `virtio` 驱动的现代操作系统（如 Linux、Windows 10/Server 2012+）。

配置示例：

```
xml
<devices>
  <memballoon model='virtio'>
    <address type='pci' bus='0x00' slot='0x05' /> <!-- 分配 PCI 地址 -->
  </memballoon></devices>
```

要求：虚拟机需安装 `virtio` 驱动（Linux 通常内置，Windows 需手动安装 `virtio-win` 驱动）。

#### 2. qemu 模型（兼容性优先）

特点：基于 `QEMU` 全虚拟化驱动，兼容性强，但性能略低于 `virtio`（有额外模拟开销）。

适用场景：不支持 `virtio` 驱动的老旧操作系统（如 Windows XP、Windows Server 2003）。

配置示例：

```
xml
<devices>
  <memballoon model='qemu'>
    <address type='pci' bus='0x00' slot='0x05' />
  </memballoon></devices>
```

#### 3. none 模型（禁用气球功能）

特点：完全禁用内存气球功能，虚拟机内存一旦分配则固定不变，无法动态调整。

适用场景：对内存稳定性要求极高的场景（如实时系统），或避免气球驱动带来的微小性能开销。

配置示例：

xml

```
<devices>
```

```
  <memballoon model='none'/>
```

```
  <!-- 无需 PCI 地址 -->
```

```
</devices>
```

## 二、按功能扩展划分（附加配置）

在基础驱动模型之上，还可添加额外配置实现更精细的功能控制：

### 1. 启用内存统计（stats 子标签）

作用：允许宿主机通过 `virsh dommemstat` 精确获取虚拟机内存使用情况（如实际使用量、共享内存等）。

配置示例（配合 `virtio` 模型）：

xml

```
<devices>
```

```
  <memballoon model='virtio'>
```

```
    <stats period='10'/>  <!-- 每 10 秒更新一次统计数据 -->
```

```
    <address type='pci' bus='0x00' slot='0x05'/>
```

```
  </memballoon></devices>
```

### 2. 限制气球最大调整范围（clip 子标签）

作用：限制动态调整时的内存下限（避免气球过度收缩导致虚拟机内存不足）。

配置示例：

xml

```
<devices>
```

```
  <memballoon model='virtio'>
```

```
    <clip minimum='2048' unit='MiB'/>  <!-- 内存下限不低于 2GiB -->
```

```
    <address type='pci' bus='0x00' slot='0x05'/>
```

```
  </memballoon></devices>
```

## 三、总结：设置方式对比与选择

| 模型 / 方式 | 性能 | 兼容性 | 动态调整 | 适用场景 |
|---------|----|-----|------|------|
|---------|----|-----|------|------|

| 模型 / 方式  | 性能  | 兼容性 | 动态调整 | 适用场景       |
|----------|-----|-----|------|------------|
| virtio   | 最优  | 较好  | 支持   | 现代操作系统（推荐） |
| qemu     | 一般  | 最优  | 支持   | 老旧操作系统     |
| none     | 无开销 | 无依赖 | 不支持  | 需固定内存的场景   |
| 附加 stats | -   | -   | -    | 需要精确监控内存使用 |
| 附加 clip  | -   | -   | -    | 需限制内存调整下限  |

实际配置中，优先选择 **virtio** 模型（性能最佳），并根据需要添加 **stats** 或 **clip** 子标签。

对于老旧系统，退化为 **qemu** 模型；无需动态调整时，使用 **none** 模型禁用气球功能。

配置后可通过 `virsh dumpxml 虚拟机名 | grep -A 5 "memballoon"` 验证设置是否生效。

内存气球（**Memory Ballooning**）是虚拟化技术中一种动态调整虚拟机内存分配的机制，允许宿主机在不同虚拟机之间灵活调度物理内存资源，实现内存的弹性管理。

### 一、核心原理：通过“气球”调整内存占用

内存气球技术通过在虚拟机内部加载一个特殊的驱动程序（“气球驱动”），动态改变虚拟机实际使用的物理内存量：

内存回收（宿主机需要更多内存时）：宿主机向虚拟机的气球驱动发送指令，让气球“充气”—— 气球驱动在虚拟机内申请并锁定一部分内存（这些内存对虚拟机内的应用不可见），然后将这部分内存的物理地址归还给宿主机，供其他虚拟机使用。

内存分配（虚拟机需要更多内存时）：当虚拟机内存不足时，宿主机让气球“放气”—— 气球驱动释放之前锁定的内存，这些内存重新变为虚拟机内应用可用的物理内存。

整个过程中，虚拟机的“虚拟内存总量”保持不变（如配置的 **4GB**），但实际使用的“物理内存”会动态增减。

### 二、主要作用与优势

提高物理内存利用率避免虚拟机“独占”分配的内存（很多虚拟机实际使用的内存远低于配置值）。例如，**10** 台各配置 **4GB** 内存的虚拟机，实际总需求可能仅 **20GB**，通过气球技术可在 **20GB** 物理内存上运行，无需 **40GB**。

实现内存超分（**Overcommit**）宿主机可分配给虚拟机的总内存超过实际物理内存（如物理内存 **32GB**，却为虚拟机分配 **40GB**），通过气球技术动态回收闲置内存，满足高峰期需求。

减少内存交换（Swap）相比直接将虚拟机内存交换到磁盘（Swap），气球技术回收的内存是“活跃但暂时不用”的，性能损耗更低（磁盘 Swap 延迟通常是毫秒级，气球调整是微秒级）。

### 三、缺点与限制

依赖虚拟机配合必须在虚拟机内安装气球驱动（如 QEMU 的 virtio-balloon），否则无法使用。部分老旧操作系统或特殊定制系统可能不支持。

#### 性能波动风险

气球“充气”回收内存时，若虚拟机内应用正在使用被回收的内存，可能导致短暂卡顿。

频繁的充气 / 放气操作会增加 CPU 开销（驱动运行和内存页迁移）。

无法回收“锁定内存”虚拟机内被应用锁定（mlock）的内存（如数据库缓存）无法被气球驱动回收，限制了内存调度的灵活性。

### 四、适用场景

虚拟机密度较高的场景（如云服务器、虚拟桌面 VDI），通过动态调度提高资源利用率。

内存需求波动大的负载（如电商促销、临时计算任务），高峰期自动扩容，低谷期释放资源。

资源紧张的边缘节点，在有限物理内存上运行更多虚拟机。

#### 总结

内存气球是一种轻量级的内存动态调度技术，通过虚拟机内的驱动程序实现物理内存的弹性分配，核心价值是提高内存利用率和支持超分。但需注意其对虚拟机的依赖性和潜在的性能波动，适合在内存资源紧张且负载有波动的场景中使用。

## 4.2.4.虚拟机内存限制配置

```
<memtune>
  <hard_limit unit="KiB">6291456</hard_limit>
  <soft_limit unit="KiB">2097152</soft_limit>
  <swap_hard_limit unit="KiB">12582912</swap_hard_limit>
</memtune>
virsh memtune
```

| 项 | 作用与使用场景 | 语法示例 |
|---|---------|------|
|---|---------|------|



| 项                    | 作用与使用场景                                                                                             | 语法示例                                                                                    |
|----------------------|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| --config             | 仅修改持久化配置（虚拟机 XML 文件），修改后需重启虚拟机才生效，不影响当前运行状态。适用于需要长期生效的内存限制配置。                                       | <code>virsh memtune 虚拟机名 --config --hard-limit 4GiB</code> （设置下次启动后内存硬限制为 4GiB）         |
| --live               | 仅修改运行中虚拟机的内存配置，实时生效，但重启后会恢复为持久化配置（若未同时用 --config）。适用于临时调整运行中虚拟机的内存限制。                               | <code>virsh memtune 虚拟机名 --live --soft-limit 2GiB</code> （实时设置运行中虚拟机的内存软限制为 2GiB）       |
| --current            | 根据虚拟机当前状态自动选择生效方式：- 若虚拟机正在运行，等效于 --live；- 若虚拟机未运行，等效于 --config。避免手动判断虚拟机状态，简化操作。                    | <code>virsh memtune 虚拟机名 --current --swap-hard-limit 8GiB</code> （根据虚拟机状态设置交换分区硬限制）     |
| --hard-limit 大小      | 设置虚拟机可使用的最大物理内存（硬限制），超出后会被强制限制（如触发 OOM 或无法分配更多内存）。单位支持 KiB/MiB/GiB 等（默认 KiB，详见 virsh 文档的“带比例整数”规则）。 | <code>virsh memtune 虚拟机名 --live --hard-limit 4GiB</code> （运行中虚拟机最多使用 4GiB 物理内存）         |
| --soft-limit 大小      | 设置内存软限制（仅在主机内存紧张时生效）：主机内存充足时，虚拟机可超软限制使用内存；主机内存不足时，会优先回收超软限制的部分，避免影响其他虚拟机。                           | <code>virsh memtune 虚拟机名 --config --soft-limit 2GiB</code> （下次启动后，虚拟机软限制为 2GiB）         |
| --swap-hard-limit 大小 | 设置虚拟机可使用的最大内存 + 交换分区总和（硬限制），防止虚拟机过度使用交换分区导致主机性能下降。（运行中虚拟机的内存 + 交换需大于 --hard-limit 的值。               | <code>virsh memtune 虚拟机名 --live --swap-hard-limit 8GiB</code> （运行中虚拟机的内存 + 交换分区最多 8GiB） |
| --min-guarantee      | 设置虚拟机的最小内存保障（仅                                                                                      | <code>virsh memtune 虚拟机名</code>                                                         |

| 项  | 作用与使用场景                                                                                                           | 语法示例                                                           |
|----|-------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| 大小 | ESX 虚拟机监控程序支持），确保虚拟机至少能获得该额度的内存，避免被他虚拟机抢占导致资源不足。                                                                  | <code>--config --min-guarantee 1GiB</code><br>虚拟机至少获得 1GiB 内存） |
| 名  | 显式指定要操作的虚拟机（可省略， <code>--domain 虚拟机名</code> 直接将虚拟机名作为命令参数，如 <code>virsh memtune 虚拟机名 --live ...</code> ），提高命令可读性。 |                                                                |

#### 4.2.5.虚拟机大页内存配置

减少内存寻址开销，适合数据库、虚拟化等高性能场景：

虚拟机配置大页（编辑 XML）`virsh edit 虚拟机名` 添加内存后端配置

```
<memoryBacking>
  <hugepages>
    <page size='2048' unit='KiB' nodeset='0' />
    <!-- 绑定到 NUMA 节点 0 -->
  </hugepages>
</memoryBacking>
```

验证大页使用：

```
bash
```

```
grep Huge /proc/meminfo# 输出中 AnonHugePages 和 HugePages_Used 应不为 0
```

内存管理是操作系统的核心功能之一，涉及多个紧密关联的概念和机制。以下从核心操作到关键概念进行系统解析：

一、内存管理的核心操作

内存申请（分配）

用户态：通过 `malloc()`（C 语言）、`new`（C++）等函数向操作系统请求内存，底层依赖 `brk()`（堆扩展）或 `mmap()`（文件映射）系统调用。

内核态：操作系统先在进程的虚拟地址空间中分配连续的“虚拟页”，但不立即分配物理内存（延迟分配）；当进程首次访问这些虚拟页时，触发“缺页异常”，才实际分配物理页并建立映射。

内存回收（释放）

主动释放：进程通过 `free()` 或 `delete` 释放内存，操作系统标记虚拟页为“未使用”，物理页可能被缓存供后续复用。

被动回收：物理内存不足时，通过页置换算法（如 LRU）将不常用的物理页写入磁盘（Swap 分区）释放空间，或直接回收不再使用的物理页（如进程退出时）。

内存映射将进程的虚拟地址空间与物理内存或磁盘文件建立关联，分为两种类型：

文件映射：通过 `mmap()` 将磁盘文件直接映射到虚拟地址，访问虚拟地址即读写文件（避免数据拷贝，提升 I/O 效率）。

匿名映射：虚拟页映射到物理内存（不关联文件），如进程的堆、栈内存。

地址转换将进程使用的“虚拟地址”转换为实际的“物理地址”，步骤：

CPU 拆分虚拟地址为页表索引（如多级页表的多级索引）。

先查询 TLB（CPU 高速缓存），命中则直接获取物理地址。

未命中则遍历内存中的页表，找到物理页地址并更新 TLB。

## 二、关键概念解析

为什么需要虚拟内存？

进程隔离：每个进程拥有独立的虚拟地址空间，避免相互干扰（如进程 A 的 0x1000 与进程 B 的 0x1000 可映射不同物理页）。

地址扩展：虚拟地址空间可远大于物理内存（如 64 位系统支持 16EB），超出部分通过磁盘 Swap 临时存储。

安全性：通过页表权限（读 / 写 / 执行）限制访问，防止越界操作。

简化编程：虚拟地址连续，无需关心物理内存的碎片化。

物理内存实际的硬件内存（DRAM），是 CPU 可直接访问的字节数组，按物理地址（如 0x12345678）寻址，容量有限（如 32GB），被所有进程共享。

为什么内存需要分页？分页是将虚拟地址空间和物理内存划分为固定大小的“页”（如 4KB），目的是：

灵活分配：避免为进程分配连续物理内存（物理内存易碎片化），通过页表映射实现虚拟地址连续。

高效置换：内存不足时，可按需换出部分页（而非整个进程），减少 I/O 开销。

粒度均衡：页是内存管理的基本单位，太小则管理成本高，太大则浪费空间（内碎片）。

内存页为什么默认是 4KB？

历史原因：Intel 80386 处理器首次采用 4KB 页大小，成为行业标准，后续系统为兼容性延续这一设计。

性能平衡：

过小（如 1KB）：管理 1GB 内存需要 26 万个页表项，页表占用大量内存，TLB 命中率低。

过大（如 1MB）：内碎片严重（如 1.2MB 数据需 2MB 页，浪费 0.8MB）。

I/O 适配：与磁盘扇区大小（传统 512B，现代 4KB）匹配，减少读写时的拆分开销。

内存页表存储“虚拟页→物理页”映射关系的数据结构，相当于“地址转换字典”：

多级结构：如 x86-64 的 4 级页表（PML4→PDPT→PD→PT），避免完整页表占用过多内存（64 位系统完整页表可达数 GB）。

页表项（PTE）：记录物理页地址、权限（读 / 写 / 执行）、是否在内存（缺页标记）等信息。

TLB（Translation Lookaside Buffer）CPU 内部的高速缓存，存储最近使用的页表项，用于加速地址转换：

作用：避免每次内存访问都遍历内存中的页表（耗时 100-200ns），TLB 命中仅需 1-3ns。

管理：进程切换或页表更新时，需刷新 TLB 以保证映射一致性。

大页内存（HugePages）页大小为 2MB、1GB 等的内存页（远大于 4KB），用于优化内存密集型应用：

优势：

减少页表项数量（1GB 大页仅需 1 个项，4KB 小页需 26 万个）。

提高 TLB 命中率（1 个 2MB 大页可覆盖 512 个 4KB 小页）。

缺点：内碎片增加，需连续物理内存（分配难度高，易受碎片化影响）。

总结

内存管理通过“虚拟内存 - 物理内存”的分层抽象，结合分页、页表、TLB 等技术，实现了进程隔离、地址扩展和高效利用。4KB 页是平衡性能与效率的历史选择，而大页则是针对特定场景的优化。这些机制共同支撑着现代计算机在有限物理资源下的高效运行。

一、内存管理的整体框架

操作系统通过“虚拟内存”机制将程序的虚拟地址映射到物理内存，核心组件包括：

虚拟地址空间：每个进程独立的地址空间（如 32 位系统最大 4GB，64 位系统理论上接近无限）。

物理内存：实际的硬件内存，被划分为固定大小的“页（Page）”（默认 4KB）。

页表：记录虚拟页到物理页的映射关系。

TLB：CPU 内部的高速缓存，加速页表查询。

大页内存：通过更大的页（如 2MB、1GB）优化地址转换效率。

## 二、核心组件的管理机制

### 1. 内存管理：虚拟内存与物理内存的映射

操作系统通过分页机制管理内存：

页划分：将虚拟地址空间和物理内存均划分为固定大小的页（默认 4KB），虚拟页（Virtual Page）与物理页（Physical Page）一一映射。

内存分配：

进程申请内存时，操作系统先分配虚拟页，仅在实际访问时（缺页异常）才分配物理页并建立映射。

物理内存不足时，通过页面置换算法（如 LRU，最近最少使用）将不常用的物理页换出到磁盘（Swap 分区），释放空间。

内存保护：页表项记录权限（读 / 写 / 执行），CPU 访问时检查权限，防止越界操作。

### 2. 页表管理：多级页表与动态创建

页表是存储映射关系的数据结构，操作系统通过多级页表优化其存储效率：

多级结构（如 x86-64 的 4 级页表）：

虚拟地址被拆分为多个段（如 PML4、PDPT、PD、PT），每段对应一级页表的索引。

只有被使用的虚拟页才会创建对应的页表项，避免完整页表占用大量内存（如 64 位系统完整页表可达数 GB）。

页表存储：页表本身也存储在物理内存中，操作系统通过 CR3 寄存器（x86 架构）定位进程的顶级页表。

动态更新：当进程访问新的虚拟页（缺页）或释放内存时，操作系统动态创建 / 删除页表项。

### 3. TLB 管理：加速地址转换

TLB 是 CPU 内部的高速缓存，存储最近使用的页表项，操作系统与硬件协同管理：

**TLB 加载：**当 CPU 访问虚拟地址时，先查 TLB：

**命中（TLB Hit）：**直接获取物理地址，无需访问页表（耗时 1-3ns）。

**未命中（TLB Miss）：**CPU 遍历内存中的页表查找映射，找到后更新 TLB（耗时约 100ns）。

**TLB 刷新：**

进程切换时，操作系统通过指令（如 `invlpg`）刷新 TLB，避免访问其他进程的页表项。

更新页表项（如权限变更）时，需同步刷新 TLB 中对应的条目，防止使用旧映射。

**软件优化：**操作系统通过内存局部性调度（如将相关数据放在连续页）提高 TLB 命中率。

#### 4. 大页内存管理：提升大块内存访问效率

**大页（HugePages）**通过增大页大小（如 2MB、1GB）减少页表项数量，操作系统需特殊管理：

**大页分配：**

**静态大页：**系统启动前通过内核参数（如 `hugepages=1024`）预留连续物理内存，供应用直接使用。

**动态大页（Transparent HugePages, THP）：**操作系统自动将连续的 4KB 小页合并为 2MB 大页，无需应用显式配置。

**大页映射：**

应用可通过系统调用（如 `mmap` 指定 `MAP_HUGETLB`）申请大页，页表项直接指向大页物理地址。

大页的 TLB 条目覆盖范围更大（如 1 个 2MB 大页 TLB 条目可覆盖 512 个 4KB 小页），显著提升 TLB 命中率。

**限制与回收：**

大页需连续物理内存，碎片化会导致分配失败（静态大页尤甚）。

动态大页在内存紧张时会拆分回小页，平衡性能与灵活性。

### 三、各组件的协同工作流程

以“进程访问虚拟地址”为例，完整流程如下：

进程生成虚拟地址（如 `0x7f8a3c2d1000`）。

**CPU 拆分虚拟地址为页表索引**（如 4 级页表的 4 个索引段）。

**检查 TLB：**

若命中，直接获取物理地址，访问物理内存。

若未命中，CPU 通过页表索引遍历内存中的多级页表，找到物理页地址。

将新的映射关系写入 TLB（更新缓存）。

若物理页未分配（缺页异常）：

操作系统分配物理页（若内存不足则换出旧页）。

更新页表项，建立虚拟页与物理页的映射。

若使用大页，页表索引段减少（如 1GB 大页仅需 1 级页表），TLB 未命中概率降低。

#### 四、关键优化策略

局部性原理：操作系统通过调度算法将进程的内存访问集中在少数页，提高 TLB 和缓存命中率。

页表缓存：部分架构（如 ARM）支持“页表走查缓存（PTW Cache）”，缓存页表中间级条目，加速页表遍历。

大页自适应：动态大页（THP）自动在“性能（大页）”和“内存利用率（小页）”间平衡，适合大多数应用。

NUMA 感知：在多 NUMA 节点服务器上，操作系统优先将内存分配到进程所在 CPU 的本地 NUMA 节点，减少跨节点访问延迟。

#### 总结

操作系统通过虚拟内存机制将虚拟地址映射到物理内存，其中：

页表负责记录映射关系，多级结构优化存储效率；

TLB 作为 CPU 缓存加速地址转换，依赖操作系统维护一致性；

大页内存通过减少页表项和提升 TLB 命中率优化性能；

各组件协同工作，在“资源利用率”和“访问性能”之间取得平衡，支撑现代计算机的高效运行。

#### 4.2.6.虚拟机内存共享配置

合并相同内存页（适合多台同配置虚拟机）：

在物理机上开启，虚拟机不需要开启。

bash

# 查看 KSM 状态 cat /sys/kernel/mm/ksm/run # 1=开启，0=关闭

# 临时开启 KSMecho 1 > /sys/kernel/mm/ksm/run

# 永久开启（通过服务）

```
systemctl enable --now ksmtuned
```

监控 KSM 效果：

```
bash
```

```
cat /sys/kernel/mm/ksm/pages_sharing # 共享的内存页数
```

```
<memoryBacking>
```

```
  <nosharepages/>
```

```
<!-- 禁用内存页共享（可选，适合高性能场景） -->
```

```
</memoryBacking>
```

虚拟机内存共享（Memory Sharing）是虚拟化场景中一种节省物理内存资源的技术，通过让多个虚拟机（或同一虚拟机内的多个进程）共享物理内存中相同的数据页，避免重复存储，从而提高内存利用率。

#### 一、核心原理：识别并合并相同内存页

虚拟机运行时，不同虚拟机（尤其是同类型操作系统或应用）往往会加载大量相同的数据（如操作系统内核代码、公共库文件、重复的应用程序代码等）。例如：

10 台运行相同 Linux 发行版的虚拟机，都会加载相同的内核代码（约几十 MB）；

多台虚拟机同时运行 Java 应用，可能会共享相同的 JVM 字节码。

内存共享技术通过以下步骤实现优化：

扫描内存页：宿主机定期扫描所有虚拟机使用的物理内存页，识别内容完全相同的页（通过哈希值比对）。

合并共享页：将相同内容的内存页合并为一个“共享页”，并让所有相关虚拟机映射到这一个物理页。

写时复制（Copy-on-Write）：当某台虚拟机需要修改共享页时，会为其创建该页的私有副本，避免影响其他虚拟机的共享数据。

#### 二、常见实现方式

KSM（Kernel SamePage Merging）这是 Linux 内核提供的内存共享机制，也是 KVM、QEMU 等虚拟化平台最常用的方案。通过内核进程（ksmd）后台扫描并合并相同内存页，可通过 `--memorybacking shared=on` 配置启用（如 `virt-install` 命令）。



用户态内存共享部分虚拟化工具（如 Docker 的 overlay 文件系统、虚拟机的共享磁盘镜像）通过用户态逻辑实现特定数据的共享（如只读的系统文件），本质是内存共享的简化形式。

### 三、主要用途与优势

提高内存利用率对运行相同操作系统或应用的虚拟机集群，可节省 30%-50% 的物理内存。例如，10 台各分配 2GB 内存的同类型虚拟机，实际物理内存消耗可能从 20GB 降至 10GB 以下。

支持更高密度部署在物理内存有限的服务器上，通过内存共享可部署更多虚拟机（如原本能跑 10 台，优化后可跑 15 台），降低硬件成本。

不影响虚拟机隔离性借助“写时复制”机制，虚拟机对共享内存的修改只会影响自身副本，不会破坏其他虚拟机的数据，保持了隔离性。

### 四、缺点与限制

CPU 开销增加扫描和比对内存页需要消耗 CPU 资源（尤其是虚拟机数量多时），ksmd 进程可能占用 10%-20% 的 CPU 算力，可能影响高负载场景的性能。

延迟风险当虚拟机修改共享页时，“写时复制”会触发内存页复制操作，可能产生短暂延迟（通常微秒级，但对实时应用可能有影响）。

适用场景有限仅对存在大量重复数据的场景有效（如同构虚拟机集群）。若虚拟机运行的应用差异极大（如一台跑数据库、一台跑 Web 服务），内存共享的收益会很低。

### 五、与其他内存技术的区别

与 NUMA 内存共享（memAccess=shared）：前者是跨虚拟机的物理内存页合并，后者是单虚拟机内 NUMA 节点的内存访问模式（允许节点内进程共享内存）。

与内存气球（Ballooning）：气球技术是动态调整单台虚拟机的内存分配，而内存共享是跨虚拟机的静态页合并，两者可配合使用。

### 总结

虚拟机内存共享是通过合并相同物理内存页来提高资源利用率的技术，特别适合同构虚拟机集群的高密度部署，但会带来一定的 CPU 开销和延迟风险。实际使用中需根据虚拟机负载特性权衡（如在资源紧张的边缘节点启用，在高性能计算场景谨慎使用）。

#### 4.2.7..优化虚拟机内存深度调优

`virsh update-device` 是用于更新虚拟机内存设备配置的命令，主要用于精细化调整虚拟机的内存相关设备参数，比如内存插槽、内存气球（`balloon`）设备等。

它比 `setmaxmem` 或 `memtune` 更侧重于内存设备的底层配置。

基本语法

bash

```
virsh update-memory-device <域名/ID> <XML 配置> [--config] [--live] [--current]
```

主要作用

调整虚拟机的内存设备属性（如内存大小、地址参数等）

支持热添加 / 修改内存设备（部分参数需虚拟机支持动态调整）

通常需要配合 XML 配置片段来指定具体的修改内容

示例用法

假设要更新虚拟机的内存气球设备，限制其最大内存：

先创建一个 XML 配置片段（如 `mem-balloon.xml`）：

xml

```
<memory balloon model='virtio'>
```

```
  <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0' />
```

```
  <max_memory unit='GiB'>4</max_memory></memory>
```

使用 `update-memory-device` 应用配置：

bash

```
# 对运行中的虚拟机生效（需支持）virsh update-memory-device vm1 mem-balloon.xml
```

--live

```
# 同时修改配置文件（永久生效）virsh update-memory-device vm1 mem-balloon.xml --live
```

--config

常见应用场景

配置内存气球设备的参数（如最大 / 最小内存）

调整内存插槽的地址或容量

启用 / 禁用内存热插拔功能

修改内存相关的 PCI 设备属性

## 注意事项

操作前需确认虚拟机支持内存设备的动态更新（部分参数需关闭虚拟机）

XML 配置片段需符合 libvirt 的规范，可参考 `virsh domxml-to-native` 生成模板

错误配置可能导致虚拟机无法启动，建议先备份原配置（`virsh dumpxml 虚拟机名 > backup.xml`）

如果需要具体调整某类内存设备，可以通过 `virsh dumpxml 虚拟机名 | grep -i memory` 查看当前内存设备配置，再针对性编写 XML 片段进行修改。

### 4.2.8.虚拟机 NUMA 内存绑定配置

```
<domain type='kvm'>
  <name>ubuntuv</name>
  <memory unit='KiB'>8388608</memory>
  <vcpu placement='static'>8</vcpu>
  <!-- NUMA 绑定配置 -->
  <numa>
    <!-- 模拟 2 个内存节点（插槽），每个 4GiB -->
    <cell id='0' cpus='0-1' memory='4194304' unit='KiB' node='0'/>
    <!-- 绑定到宿主机 NUMA 节点 0 -->
    <cell id='1' cpus='2-3' memory='4194304' unit='KiB' node='1'/>
    <!-- 绑定到宿主机 NUMA 节点 1 -->
  </numa>
```

多 NUMA 节点宿主机中，将内存绑定到 vCPU 所在节点：

```
bash
```

```
# 1. 查看宿主机 NUMA 布局
```

```
numactl --hardware
```

```
# 2. 编辑虚拟机 XML 绑定 NUMA
virsh edit 虚拟机名# 添加 NUMA 配置<numa>
```

```
  <node cellid='0' cpus='0-3' memory='4194304' unit='KiB'/>
  <!-- 节点 0 分配 4G -->
```

</numa>

## 一、NUMA 内存绑定的核心目的

在多 CPU 服务器中，物理内存通常按 CPU 核心划分为多个 NUMA 节点（如双路 CPU 服务器有 node0 和 node1 两个节点），每个节点的内存与本地 CPU 的访问延迟远低于跨节点访问（例如 node0 的 CPU 访问 node0 的内存延迟为 50ns，访问 node1 的内存可能达 100ns）。

内存绑定的目的是：强制进程或虚拟机的内存从指定 NUMA 节点分配，避免跨节点内存访问，从而降低延迟、提升性能。

## 二、内存绑定的实现方式（以 KVM 虚拟化为例）

### 1. 虚拟机层面的 NUMA 内存绑定

通过 virt-install 或 virsh 配置虚拟机的 NUMA 拓扑，指定内存从特定物理 NUMA 节点分配：

```
bash
```

```
# 创建虚拟机时绑定内存到宿主机 node0
```

```
virt-install \
```

```
--name numa-bound-vm \
```

```
--memory memory=4096 \
```

```
--vcpus 2 \
```

```
# 虚拟机 NUMA 节点 0 绑定宿主机 node0 的 CPU 和内存
```

```
--numa node=0,cpus=0-1,memory=4096,memnode=0 \
```

```
--disk path=/var/lib/libvirt/images/bound-vm.qcow2,size=20 \
```

```
--os-variant centos8 \
```

```
--network bridge=virbr0 \
```

```
--graphics none
```

memnode=0：指定虚拟机 NUMA 节点 0 的内存从宿主机 NUMA node0 分配。

### 2. 进程层面的内存绑定（宿主机或虚拟机内部）

通过 numactl 命令限制进程只能使用指定 NUMA 节点的内存：

```
bash
```

```
# 强制进程在宿主机 node0 上分配内存
```

```
numactl --membind=0 --runpart ./my_application
```

# 查看进程的 NUMA 内存分配

numastat -p <进程 PID>

### 三、优势与适用场景

降低内存访问延迟避免跨 NUMA 节点访问，对 latency 敏感的应用（如数据库、高频交易系统）性能提升明显（通常 5%-20%）。

提高 CPU 缓存利用率本地 NUMA 节点的内存数据更易被本地 CPU 缓存命中，减少缓存失效。

资源隔离在多租户场景中，可将不同虚拟机的内存绑定到不同 NUMA 节点，避免资源争抢。

### 四、注意事项

内存充足性绑定的 NUMA 节点必须有足够的空闲内存，否则虚拟机或进程可能启动失败或触发 OOM（内存溢出）。

CPU 与内存绑定配合内存绑定需与 CPU 绑定（--cpuset）配合使用才有意义，否则 CPU 跨节点调度时，仍会导致跨节点内存访问。

灵活性降低绑定后无法利用其他 NUMA 节点的空闲内存，可能导致整体内存利用率下降。

### 总结

NUMA 内存绑定是优化多 CPU 服务器性能的关键技术，通过限制内存访问范围减少跨节点延迟，尤其适合高性能计算、数据库等场景。但需平衡性能与资源利用率，避免因过度绑定导致资源浪费。

## 4.2.9.虚拟机 VCPU 配置

```
<vcpu placement='auto' current='2'>4</vcpu>
```

```
<vcpu placement="static">3</vcpu>
```

static 的意思是 vcpu 不会动态根据负载情况调整，auto 的意思是启动的是很是 2 个，负载大了会动态调整到 3、4。

```
virsh setvcpus
```

2 为当前 vcpu 核心数，4 为最大 vcpu 核心数

你列出的是 virsh setvcpus 命令的核心参数，这些参数用于精确控制虚拟机 vCPU 的调整方式，以下是具体用法和场景说明：

#### 一、必选参数

--domain <虚拟机名称/ID>指定目标虚拟机（必须填写），例如 --domain fnos

--count <数量>指定 vCPU 数量（必须填写），例如 --count 4（设置为 4 核）

#### 二、生效范围参数（控制配置何时生效）

--live 仅对运行中的虚拟机生效（动态调整，无需关机），但不修改配置文件（重启后失效）。示例：virsh setvcpus --domain fnos --count 4 --live（运行中动态扩至 4 核）

--config 仅修改配置文件，对当前运行的虚拟机无影响，下次启动时生效。示例：virsh setvcpus --domain fnos --count 4 --config（下次启动后为 4 核）

--current 根据虚拟机当前状态自动生效：

若虚拟机正在运行，等同于 --live；

若虚拟机未运行，等同于 --config。

#### 三、功能参数（控制 vCPU 调整类型）

--maximum 调整虚拟机支持的最大 vCPU 上限（硬件规格），通常需要关闭虚拟机才能修改。示例：virsh setvcpus --domain fnos --count 8 --maximum --config（最大支持 8 核，下次启动生效）

--hotpluggable 设置可热插拔的 vCPU 数量（预留动态添加的能力），需配合最大上限使用。示例：virsh setvcpus --domain fnos --count 6 --hotpluggable --config（允许动态添加至 6 核）

--guest 通知虚拟机内部操作系统 vCPU 数量变化（通常自动触发，无需手动指定）。

#### 四、典型组合示例

动态调整并永久生效（运行中生效 + 保存到配置）：

```
bash
```

```
virsh setvcpus --domain fnos --count 4 --live --config
```

关闭虚拟机后调整最大上限：

bash

```
virsh shutdown fnosvirsh setvcpus --domain fnos --count 8 --maximum --config # 最大 8
```

核 virsh setvcpus --domain fnos --count 4 --config # 当前 4 核 virsh start fnos

查看配置结果：

bash

```
virsh vcpucount fnos # 查看当前核数和最大核数
```

注意

动态调整 (--live) 需要虚拟机操作系统支持 CPU 热插拔 (如 Linux 内核开启相关模块, Windows 需专业版 / 企业版)。

最大 vCPU 数量 (--maximum) 不能小于当前正在使用的 vCPU 数量。

根据实际需求选择参数组合, 可灵活实现 vCPU 的临时调整或永久配置。

分享

如何查看当前虚拟机的 vCPU 数量?

如何在不重启虚拟机的情况下, 动态调整 vCPU 的数量?

除了 virsh setvcpus 命令, 还有哪些命令可以用于调整虚拟机的 vCPU 资源?

#### 4.2.10.虚拟机 VCPU 热插拔

```
<cpu mode="host-model" check="partial">
```

```
<hotplug>yes</hotplug>
```

```
<topology sockets="1" dies="1" cores="4" threads="1"/>
```

```
</cpu>
```

```
virsh setvcpu
```

你列出的是 virsh setvcpu 命令的核心参数, 这些参数用于控制 vCPU 配置的生效时机、操作对象等细节, 具体含义如下:

--config: 仅修改虚拟机的配置文件, 设置将在虚拟机下次启动时生效, 不影响当前运行状态。

--current: 同时作用于虚拟机的当前运行状态 (如果正在运行) 和配置文件, 即 "立即生效 + 永久保存"。

--disable: 禁用指定的 vCPU (需配合 --vcpulist 使用), 通常用于临时关闭部分 vCPU。

**--domain <名称/ID>**: 指定要操作的虚拟机（必须参数），例如**--domain ubuntuvm**。

**--enable**: 启用指定的 vCPU（需配合**--vcpulist** 使用），与**--disable** 相反。

**--live**: 仅对正在运行的虚拟机实时生效，不修改配置文件，重启后设置会失效。

**--vcpulist <编号列表>**: 指定具体操作的 vCPU 编号（如 0,1 表示第 0 和 1 号 vCPU），常用于单独启用 / 禁用某几个 vCPU。

这些参数通常需要组合使用，例如：

实时调整运行中虚拟机的 vCPU 数量（不保存到配置）：**virsh setvcpu --domain ubuntuvm --live --count 4**

禁用虚拟机的 2 号 vCPU（下次启动生效）：**virsh setvcpu --domain ubuntuvm --config --vcpulist 2 --disable**

#### 4.2.11.虚拟机 VCPU 亲和性

```
<cputune>
  <vcpupin vcpu='0' cpuset='0' />
  <vcpupin vcpu='1' cpuset='1' />
  <vcpupin vcpu='2' cpuset='2' />
  <vcpupin vcpu='3' cpuset='3' />
</cputune>
```

你列出的是 **virsh vcpupin** 命令的核心参数，这些参数用于控制 vCPU 与物理 CPU 的绑定（亲和性配置），具体含义如下：

**--config**: 仅修改配置文件，绑定关系在虚拟机下次启动时生效，不影响当前运行状态。

**--cpulist <物理 CPU 列表>**: 指定要绑定的物理 CPU 核心编号（如 0,2-3 表示物理核心 0、2、3）。

**--current**: 同时对虚拟机的当前运行状态（若运行中）和配置文件生效，即 "立即绑定 + 永久保存"。

**--domain <名称/ID>**: 指定要操作的虚拟机（必选参数），例如**--domain ubuntuvm**。

**--live**: 仅对运行中的虚拟机实时生效，不修改配置文件，重启后绑定关系失效。

**--vcpu <vCPU 编号>**: 指定要绑定的虚拟机 vCPU 编号（单个编号，如 0）。

常用示例：

查看当前 vCPU 与物理 CPU 的绑定关系：



```
bash
```

```
virsh vcpupin --domain ubuntuvm
```

实时绑定 vCPU 0 到物理 CPU 1（重启后失效）：

```
bash
```

```
virsh vcpupin --domain ubuntuvm --live --vcpu 0 --cpulist
```

永久绑定 vCPU 1 到物理 CPU 2 和 3（下次启动生效）：

```
bash
```

```
virsh vcpupin --domain ubuntuvm --config --vcpu 1 --cpulist 2-3
```

批量绑定多个 vCPU 到物理 CPU（同时生效）：

```
bash
```

```
# 将 vCPU 0-2 分别绑定到物理 CPU 0-2
virsh vcpupin --domain ubuntuvm --current
--vcpulist 0-2 --cpulist 0-2
```

这些参数通过控制 vCPU 在物理核心上的调度，可减少 CPU 切换开销，提升对延迟敏感的应用性能（如数据库、实时服务等）。

#### 4.2.12.虚拟机 VCPU 缓存优化

```
<cpu mode='host-passthrough' check='none'>
```

```
<cache mode='passthrough' /> <!-- 完全适配，性能最优 -->
```

```
</cpu>
```

对 CPU 模式有一定要求，主要与虚拟机 CPU 模式是否支持直接使用宿主机 CPU 缓存特性相关，具体如下：

##### 1. 最适配的 CPU 模式：host-passthrough

原因：host-passthrough 模式会完全直通宿主机 CPU 的硬件特性（包括缓存结构），此时 <cache mode='passthrough' /> 可以正常生效，实现虚拟机直接使用物理 CPU 的缓存，减少虚拟化层的缓存模拟开销，性能最佳。

配置示例：

```
<cpu mode='host-passthrough' check='none'>
```

```
<cache mode='passthrough' /> <!-- 完全适配，性能最优 -->
```

```
</cpu>
```

4.2.13.虚拟机 VCPU 调度策略

```
<cputune>
  <shares>2048</shares>
  <!-- 对应 cpu_shares -->
  <period>100000</period>
  <!-- 对应 vcpu_period -->
  <quota>50000</quota>
  <!-- 对应 vcpu_quota -->
  <emulator_period>100000</emulator_period>
<emulator_quota>30000</emulator_quota>
  <iothread_period>100000</iothread_period>
  <iothread_quota>20000</iothread_quota>
</cputune>

virsh schedinfo
```

“虚拟机 VCPU 调度策略”是 KVM 虚拟化中优化 CPU 资源分配的重要手段，主要通过调整调度权重、周期和配额等参数，实现对不同虚拟机的 CPU 资源优先级管理。以下是具体配置方法和参数含义：

一、核心调度策略参数（通过 virsh schedinfo 配置）

KVM 通过 CFS（Completely Fair Scheduler）调度器管理 VCPU，核心参数包括：

| 参数                 | 含义                           | 取值范围                    | 调优建议                       |
|--------------------|------------------------------|-------------------------|----------------------------|
| <b>vcpu_weight</b> | CPU 权重（相对优先级），值越高获得 CPU 时间越多 | 1-1048575（默认 1024）      | 核心业务虚拟机设为 2048，普通虚拟机设为 512 |
| <b>vcpu_period</b> | 调度周期（微秒），表示多久重新分配一次 CPU 时间   | 1000-1000000（默认 100000） | 一般保持默认，高负载场景可缩短至 50000     |
| <b>vcpu_quota</b>  | 一个周期内可使用的最大 CPU 时间（微秒），≥1000 | -1（无限制）或                | 限制非核心虚拟机：如周期 100000，配额     |

| 参数 | 含义           | 取值范围 | 调优建议           |
|----|--------------|------|----------------|
|    | 控制 CPU 使用率上限 |      | 50000 表示上限 50% |

参数含义详解

| 参数              | 当前值       | 含义说明                                             |
|-----------------|-----------|--------------------------------------------------|
| 调度程序: posix     | -         | 使用 Linux 系统默认的 POSIX 调度器（KVM 默认，适用于大多数场景）。       |
| cpu_shares      | 1024      | CPU 权重（相对优先级），值越高，竞争 CPU 资源时获得的时间片越多（默认 1024）。   |
| vcpu_period     | 100000    | VCPU 调度周期（单位：微秒，即 0.1 秒），表示每 0.1 秒重新分配一次 CPU 时间。 |
| vcpu_quota      | -1        | VCPU 在一个周期内的最大可用时间（微秒），-1 表示无限制（可全额使用 CPU）。      |
| emulator_period | 100000    | 虚拟机模拟器（如 QEMU 进程）的调度周期（同 vcpu_period 逻辑）。        |
| emulator_quota  | -1        | 模拟器在一个周期内的最大可用时间（-1 表示无限制）。                      |
| global_*        | 100000/-1 | 全局调度参数（通常与 vcpu_*一致，暂未单独配置）。                     |
| iothread_*      | 100000/-1 | I/O 线程的调度周期和配额（控制磁盘 / 网络 I/O 的 CPU 占用，-1 表示无限制）。 |

二、配置命令示例

1. 查看当前调度策略

```
bash
virsh schedinfo 虚拟机名 # 如 virsh schedinfo ubuntuvm
```

输出示例：

```
plaintext
Scheduler      : posix
vcpu_weight    : 1024
vcpu_period    : 100000
```

```
vcpu_quota      : -1
```

```
...
```

## 2. 调整 VCPU 权重（优先级）

```
bash
```

```
# 提高核心虚拟机权重（优先级）virsh schedinfo --domain 虚拟机名 --set
```

```
vcpu_weight=2048
```

```
# 降低非核心虚拟机权重 virsh schedinfo --domain 虚拟机名 --set vcpu_weight=512
```

## 3. 限制 VCPU 最大使用率（配额）

```
bash
```

```
# 周期 100000 微秒（0.1 秒）内，最多使用 50000 微秒（即 CPU 使用率上限 50%）virsh  
schedinfo --domain 虚拟机名 --set vcpu_period=100000 --set vcpu_quota=50000
```

```
# 取消限制（恢复无上限）virsh schedinfo --domain 虚拟机名 --set vcpu_quota=-1
```

## 三、调度策略调优原则

按业务优先级分配权重

核心服务（如数据库、关键应用）：vcpu\_weight=2048-4096

普通服务（如 Web 服务器）：vcpu\_weight=1024

低优先级服务（如测试环境）：vcpu\_weight=512-256

合理设置配额防止资源独占

对 CPU 消耗较高但非核心的服务（如日志分析），设置 vcpu\_quota 限制其最大使用率（如 50%），避免影响其他虚拟机。

平衡周期与响应速度

周期过短（如 <10000 微秒）会增加调度开销；过长（如 >200000 微秒）会导致高优先级虚拟机响应延迟，建议保持默认 100000 或根据负载调整。

## 四、验证调优效果

监控宿主机 CPU 分配情况：

```
bash
```

```
top # 查看虚拟机进程（qemu-kvm）的 CPU 使用率是否符合预期
```

检查虚拟机内部 CPU 负载：在虚拟机内执行 mpstat 或 vmstat，观察 CPU 使用率是否被有效限制或优先分配。

通过调整这些参数，可以实现对不同虚拟机的 CPU 资源精细化管理，确保核心业务获得足够资源，同时避免单个虚拟机过度占用 CPU。

#### 4.2.14.虚拟机 VCPU 物理拓扑

```
<cpu mode="host-model" check="partial">
<hotplug>yes</hotplug>
<topology sockets="1" dies="1" cores="4" threads="1"/>
</cpu>
```

在 KVM 中设置 vCPU 物理拓扑（模拟物理 CPU 的插槽、核心、线程结构），需要通过编辑虚拟机 XML 配置文件中的 `<topology>` 标签实现，以匹配宿主机物理 CPU 结构或优化虚拟机性能。以下是具体方法和最佳实践：

##### 一、核心配置参数

vCPU 拓扑通过 `<cpu>` 标签内的 `<topology>` 子标签定义，包含 3 个关键参数：

**sockets:** 虚拟 CPU 插槽数量（模拟物理 CPU 的插槽）

**cores:** 每个虚拟插槽的核心数（模拟物理核心）

**threads:** 每个虚拟核心的线程数（模拟超线程）

总 vCPU 数量 = sockets × cores × threads，需与 `<vcpu>` 标签中定义的数量一致。

##### 二、配置步骤

###### 1. 查看宿主机物理 CPU 拓扑（用于参考）

先通过 `lscpu` 了解宿主机结构，例如：

```
bash
```

```
lscpu | grep -E 'Socket|Core|Thread'# 示例输出：# Socket(s):                2# Core(s) per
```

```
socket: 8# Thread(s) per core: 2
```

###### 2. 编辑虚拟机 XML 配置

```
bash
```

```
virsh edit 虚拟机名 # 如 virsh edit ubuntuvm
```

###### 3. 添加 / 修改 `<topology>` 配置

在 `<cpu>` 标签内添加拓扑配置，确保总 vCPU 数匹配：

示例 1：模拟单插槽 4 核心（无超线程）

```
xml
<vcpu placement='static'>4</vcpu> <!-- 总 vCPU=4 --><cpu mode='host-model'
check='partial'>
    <topology sockets='1' cores='4' threads='1'/> <!-- 1×4×1=4 --></cpu>
```

示例 2：模拟 2 插槽 4 核心 2 线程（超线程，总 16vCPU）

```
xml
<vcpu placement='static'>16</vcpu> <!-- 总 vCPU=16 --><cpu mode='host-model'
check='partial'>
    <topology sockets='2' cores='4' threads='2'/> <!-- 2×4×2=16 --></cpu>
```

#### 4. 保存配置并重启虚拟机

```
bash
```

```
virsh shutdown 虚拟机名 virsh start 虚拟机名
```

#### 三、验证配置是否生效

查看虚拟机 XML 中的拓扑配置：

```
bash
```

```
virsh dumpxml 虚拟机名 | grep -A 5 "<cpu"
```

在虚拟机内部验证（Linux 系统）：

```
bash
```

```
# 进入虚拟机后执行
```

```
lscpu | grep -E 'Socket|Core|Thread'
```

输出应与配置的拓扑一致（如 sockets=2, cores=4, threads=2）。

#### 四、拓扑配置最佳实践

匹配宿主机结构：虚拟机拓扑尽量与宿主机保持一致（如宿主机 2 插槽 8 核心，虚拟机可设 2 插槽 4 核心），减少虚拟化层的调度开销。

避免过度模拟超线程：若宿主机支持超线程（threads>1），虚拟机可开启；若宿主机无超线程，虚拟机设 threads=1 更高效。

平衡插槽与核心数：

对需要多 NUMA 节点优化的应用（如数据库），增加 sockets 数（模拟多 NUMA 节点）。

普通应用优先增加 cores 数（单插槽多核心性能更优）。

总 vCPU 数不超过物理核心数的 1.5 倍：避免超分过多导致性能下降（如 8 核物理 CPU，虚拟机总 vCPU 建议≤12）。

## 五、常见问题

配置后虚拟机无法启动：检查总 vCPU 数是否与 sockets×cores×threads 相等，或拓扑参数是否超过宿主机支持范围。

虚拟机内看不到拓扑变化：部分老操作系统（如 Windows Server 2008）对虚拟拓扑支持有限，需升级系统或调整模式为 host-passthrough。

通过合理配置 vCPU 拓扑，可使虚拟机更好地利用物理 CPU 资源，尤其对依赖 CPU 缓存和 NUMA 架构的应用（如大数据、数据库）性能提升明显。

## 4.3.体系架构相关配置

### 4.3.1.操作系统启动

```
<os>

  <type arch='x86_64' machine='pc-q35-4.1'>hvm</type>

  <boot dev='hd' />

</os>
```

这段代码是虚拟机配置文件（如 QEMU/KVM 的 XML 配置）中用于定义操作系统（OS）启动参数的核心片段，主要作用是告诉虚拟化平台如何模拟硬件、加载操作系统。下面将逐字段拆解其含义、参数及背后的技术逻辑，确保覆盖技术细节和实际应用场景。

#### 一、整体结构：<os> 根节点

<os> 是虚拟机“操作系统启动配置”的顶层容器，所有与 OS 启动相关的硬件模拟、启动顺序、架构匹配等参数都嵌套在该节点下。它的核心作用是建立“虚拟化硬件”与“Guest OS（客户机操作系统）”之间的适配关系，比如告诉 QEMU：“用 x86\_64 架构的硬件模拟，以硬盘为第一启动项，运行一个支持硬件辅助虚拟化的系统”。

#### 二、核心子字段详解

##### 1. <type> 节点：定义虚拟化架构与硬件模拟类型

<type arch='x86\_64' machine='pc-q35-4.1'>hvm</type> 是整个 <os> 中最关键的配置，决定了虚拟机的硬件架构、主板模拟型号、虚拟化模式，三个属性（arch、machine、值 hvm）缺一不可。

（1）arch='x86\_64': CPU 架构

含义：指定虚拟机模拟的 CPU 指令集架构，必须与 Guest OS 支持的架构一致（否则 OS 无法启动）。

本质：决定虚拟机能运行 32 位还是 64 位系统，以及 CPU 支持的指令集（如 x86、ARM、PowerPC 等）。

常见参数及场景：

| 参数值       | 架构类型             | 支持的 OS 示例                  | 适用场景                        |
|-----------|------------------|----------------------------|-----------------------------|
| x86_64    | 64 位 x86 架构      | Windows 10/11、Linux x86_64 | 主流 PC、服务器虚拟化                |
| i686/i386 | 32 位 x86 架构      | Windows XP、Linux i386      | 老旧系统兼容（如工业软件）               |
| aarch64   | 64 位 ARM 架构      | Android、Linux aarch64      | 移动端、ARM 服务器（如 AWS Graviton） |
| ppc64le   | 64 位 PowerPC（小端） | AIX、Linux ppc64le          | 大型机、IBM 服务器虚拟化              |

注意：若 Guest OS 是 64 位（如 Windows 11），arch 必须设为 x86\_64；若设为 i686，会因架构不兼容导致启动失败。

（2）machine='pc-q35-4.1': 主板硬件模拟型号

含义：指定虚拟机模拟的物理主板型号，决定了 Guest OS 能识别的硬件接口（如 PCIe、SATA、USB）、BIOS/UEFI 版本等底层硬件特性。

本质：QEMU 通过预定义的“主板模板”模拟真实硬件，pc-q35-4.1 是 x86\_64 架构下最常用的“现代主板模板”（基于 Intel Q35 芯片组）。

参数解析：

pc: 表示“PC 类主板”（通用 x86 架构主板，区别于服务器主板如 q35-server）；

q35: 基于 Intel Q35 芯片组（2007 年推出的主流桌面 / 工作站芯片组，支持 PCIe 2.0、SATA 2.0，比老旧的 i440fx 芯片组更现代）；



4.1: QEMU 的主板模板版本号（不同版本可能修复硬件兼容性问题，如支持新的 PCIe 设备）。

常见替代参数：

pc-i440fx-4.1: 基于 Intel i440FX 芯片组（老旧主板模板，兼容 Windows XP 等早期系统，不支持 PCIe 3.0+）；

pc-q35-7.0: 更新版本的 Q35 模板（支持 PCIe 4.0、UEFI Secure Boot，适配 Windows 11、最新 Linux 内核）；

virt-7.0: 轻量级主板模板（无冗余硬件模拟，启动速度快，适合云环境如 OpenStack）。

（3）节点值 hvm：虚拟化模式

含义：指定虚拟机使用硬件辅助虚拟化（Hardware-Assisted Virtualization）模式，是现代虚拟化的核心技术。

本质：借助 CPU 硬件指令（如 Intel VT-x、AMD-V）直接让 Guest OS 访问物理 CPU 资源，避免软件模拟指令的性能损耗，实现“接近物理机的性能”。

与其他虚拟化模式的对比：

| 模式   | 技术原理                    | 性能        | 适用场景                              |
|------|-------------------------|-----------|-----------------------------------|
| hvm  | 硬件辅助虚拟化<br>(VT-x/AMD-V) | 极高（接近物理机） | 运行完整 OS（如 Windows、Linux 桌面 / 服务器） |
| qemu | 纯软件指令模拟（无硬件辅助）          | 极低        | 架构不匹配场景（如 x86 主机运行 ARM 系统）        |
| xen  | 半虚拟化（Xen Hypervisor）    | 中高        | 早期 Xen 架构，需 Guest OS 安装驱动         |

注意：使用 hvm 模式的前提是物理 CPU 支持 VT-x/AMD-V，且需在主机 BIOS/UEFI 中开启该功能（否则虚拟机启动会报错）。

2. <boot dev='hd' />：启动设备顺序

含义：指定虚拟机的第一启动设备，即虚拟机通电后优先从哪个设备加载操作系统。

本质：模拟物理机的“BIOS 启动顺序”（如“硬盘优先”“光驱优先”），确保 Guest OS 能从预安装的设备启动。

常见 dev 参数及场景：

| 参数值     | 启动设备类型          | 适用场景                               |
|---------|-----------------|------------------------------------|
| hd      | 硬盘（Hard Disk）   | 已安装 OS 的场景（如日常使用的 Windows 虚拟机）     |
| cdrom   | 光驱（CD-ROM）      | 从 ISO 镜像安装 OS（如首次安装 Linux、Windows） |
| network | 网络（PXE）         | 无盘启动 / 网络安装（如数据中心批量部署虚拟机）          |
| floppy  | 软盘（Floppy Disk） | 老旧系统启动（如 DOS 系统，现代已极少使用）           |

扩展配置：若需指定多启动顺序（如“先光驱，再硬盘”），可通过多个 `<boot>` 节点加 `order` 属性实现，示例：

```
xml
<os>
  <type arch='x86_64' machine='pc-q35-4.1'>hvm</type>
  <boot dev='cdrom' order='1'/> <!-- 第一启动项：光驱 -->
  <boot dev='hd' order='2'/>      <!-- 第二启动项：硬盘 -->
  <boot dev='network' order='3'/> <!-- 第三启动项：网络 --></os>
```

### 三、关联技术背景与实际配置建议

BIOS/UEFI 的适配：上述配置默认使用 BIOS 启动，若需支持 UEFI（更现代的启动方式，支持 Secure Boot、大硬盘），需在 `<os>` 中添加 `<firmware>` 节点，示例：

```
xml
<os>
  <type arch='x86_64' machine='pc-q35-4.1'>hvm</type>
  <firmware>efi</firmware> <!-- 启用 UEFI -->
  <boot dev='hd'/></os>
```

（注：UEFI 需配合 pc-q35 等现代主板模板，且 Guest OS 需支持 UEFI，如 Windows 10/11、CentOS 8+）。

兼容性问题排查：若虚拟机启动失败，优先检查 `<type>` 节点：

若报错“架构不兼容”：确认 `arch` 与 Guest OS 镜像匹配（如 64 位 OS 用 `x86_64`）；

若报错“硬件不支持”：检查 machine 模板是否过旧（如 Windows 11 需 pc-q35-7.0+UEFI）；

若报错“VT-x 未启用”：进入主机 BIOS 开启 CPU 虚拟化功能，确保 <type> 是 hvm。

四、总结

| 字段 / 属性    | 核心作用                  | 关键参数示例                   | 技术依赖                     |
|------------|-----------------------|--------------------------|--------------------------|
| <os>       | 容器：管理 OS 启动相关配置       | -                        | QEMU/KVM 虚拟化平台           |
|            | 定义 CPU 架构，匹配 Guest OS | x86_64、aarch64、i686      | 物理 CPU 架构支持（如 x86_64 主机） |
| machine    | 模拟主板型号，定义硬件接口特性       | pc-q35-4.1、pc-i440fx-4.1 | QEMU 主板模板库               |
|            | 启用硬件辅助虚拟化，保障性能        | hvm、qemu、xen             | 物理 CPU 支持 VT-x/AMD-V     |
| <boot dev> | 指定启动设备顺序，控制 OS 加载路径   | hd、cdrom、network         | 虚拟机存储设备配置（如硬盘、ISO）       |

通过理解这些字段，可灵活调整虚拟机的硬件兼容性、性能和启动逻辑，适配从“老旧工业系统”到“最新 Windows 11”的各类场景。

4.3.2.虚拟机硬件特性配置

```
<features>
  <acpi/>
  <apic/>
  <vmport state='off' />
</features>
```

虚拟机配置文件（如 QEMU/KVM XML）中用于定义“硬件特性开关”的核心片段，通过启用 / 禁用特定硬件功能，决定 Guest OS（客户机操作系统）能使用的底层硬件能力，直接影响系统稳定性、电源管理、中断处理等关键功能。下面逐字段拆解其含义、技术原理、参数及实际应用场景。

一、整体结构：<features> 根节点

`<features>` 是虚拟机“硬件特性配置”的顶层容器，所有与 CPU、主板相关的可选硬件功能（如电源管理、中断控制器、老式设备接口）都通过该节点下的子标签启用或禁用。它的核心作用是在“虚拟化硬件模拟”中筛选出 Guest OS 需要的功能——既避免加载无用功能占用资源，也确保关键功能（如休眠、多设备中断）正常工作，同时规避部分老旧功能的安全风险。

## 二、核心子字段详解

### 1. `<acpi/>`: 启用 ACPI 高级配置与电源管理接口

#### （1）基本定义

全称：Advanced Configuration and Power Interface（高级配置与电源管理接口）。

含义：向 Guest OS 提供一套标准化的“硬件配置 + 电源管理”接口，是现代操作系统实现电源控制（如关机、休眠）和硬件状态识别的核心依赖。

本质：模拟物理机主板上的 ACPI 控制器，让 Guest OS 能通过标准化指令读取硬件信息（如 CPU 型号、硬盘数量）、控制电源状态（如软关机、进入休眠），而非依赖底层硬件的私有接口。

#### （2）为什么必须启用？

没有 ACPI 的虚拟机将面临两大问题：

电源管理失效：Guest OS 无法执行“软关机”（只能强制断电，可能导致数据丢失）、“休眠 / 待机”（系统只能一直运行）；

硬件识别不全：Guest OS 无法正确识别虚拟化硬件的型号、资源分配（如无法识别多块硬盘、USB 设备，甚至可能蓝屏 / 启动失败）。

#### （3）适用场景与例外

必启用场景：运行 Windows（XP 及以上）、Linux（内核 2.6 及以上）、macOS 等现代操作系统时，`<acpi/>` 是默认且必须开启的配置，无替代方案。

例外场景：仅运行 DOS、老旧嵌入式系统（无 ACPI 驱动）时，可省略该标签（但现代虚拟化平台默认仍会启用，避免兼容性问题）。

### 2. `<apic/>`: 启用 APIC 高级可编程中断控制器

#### （1）基本定义

全称：Advanced Programmable Interrupt Controller（高级可编程中断控制器）。

含义：替代传统的 8259 PIC（可编程中断控制器），为多 CPU / 多核虚拟机提供更高效、更灵活的“中断管理”能力，是虚拟机支持多核心、多设备并发的关键。

本质：模拟物理机的 APIC 芯片，负责接收硬件设备（如网卡、硬盘、键盘）的“中断请求”（IRQ），并将请求分配给不同的 CPU 核心处理 —— 解决了传统 PIC 仅支持 15 个中断、无法分配给多 CPU 的瓶颈。

（2）核心作用（对比传统 PIC）

| 特性           | 传统 PIC（无 APIC）     | APIC（启用<apic/>）           |
|--------------|--------------------|---------------------------|
| 中断数量上限       | 仅 15 个（易冲突）        | 支持 256 个以上（无冲突）           |
| 多 CPU / 多核支持 | 不支持（仅能分配给 1 个 CPU） | 支持将中断分配给任意 CPU 核心         |
| 中断优先级控制      | 无（固定优先级）           | 可自定义中断优先级（高效调度）           |
| 适用系统         | DOS、单核心老旧系统        | Windows、Linux、macOS 等现代系统 |

（3）关键注意点

与多核 CPU 的强关联：若虚拟机配置了多个 CPU 核心（如 <vcpu>4</vcpu>），必须启用 <apic/> —— 否则 Guest OS 无法识别多核心，仅能使用 1 个核心，且可能因中断冲突导致设备（如网卡、硬盘）无法工作。

与 IO-APIC 的关系：<apic/> 通常与 <ioapic/> 配合使用（现代虚拟化平台默认自动启用 IO-APIC），IO-APIC 负责管理“外部设备中断”，APIC 负责“CPU 核心间的中断分配”，二者共同构成完整的中断管理体系。

3. <vmport state='off'/>：禁用 VMPort 老式虚拟机调试接口

（1）基本定义

全称：Virtual Machine Port（虚拟机端口）。

含义：早期 VMware、QEMU 等虚拟化平台提供的一套“调试 / 通信接口”，Guest OS 可通过访问特定的 IO 端口（如 0x5658，对应“VMware”的 ASCII 码）与宿主机（Host）交互，实现调试、状态查询等功能。

本质：一套老旧的、非标准化的虚拟化通信接口，主要用于早期虚拟机工具（如 VMware Tools 旧版本）的调试，现代已被更安全的接口（如 virtio、VSOCK）替代。

（2）为什么默认设为 state='off'（禁用）？

禁用 VMPort 的核心原因是安全风险：

接口无认证：任何程序（包括恶意软件）都可通过 VMPort 访问宿主机的部分资源，可能导致宿主机信息泄露（如宿主机 IP、硬件配置）；

漏洞历史：早期 VMPort 实现存在多个安全漏洞（如缓冲区溢出），恶意软件可利用漏洞突破虚拟机隔离（“逃逸”到宿主机）；

功能冗余：现代虚拟化工具（如 QEMU Guest Agent、VMware Tools 新版本）已完全不依赖 VMPort，改用 virtio、VSOCK 等标准化、安全的接口实现宿主机 - 客户机交互。

（3）参数与例外场景

state='off'：默认且推荐配置，完全禁用 VMPort 接口，规避安全风险。

state='on'：仅在特殊场景下使用（如运行依赖旧版 VMware Tools 的 Windows XP 虚拟机、调试老旧嵌入式系统），启用后需确保 Guest OS 无恶意程序，且宿主机已修复 VMPort 相关漏洞。

三、关联配置与常见问题排查

1. 与其他特性的配合（现代虚拟机完整配置示例）

除上述三个字段外，现代虚拟机（如支持 Windows 11、Linux 9）通常还需配合以下特性配置，形成完整的硬件功能集：

```
xml
<features>
    <acpi/>                <!-- 必启：电源管理与硬件识别 -->
    <apic/>                 <!-- 必启：多核心中断管理 -->
    <vmport state='off'/>   <!-- 必禁：规避安全风险 -->
    <smm state='on'/>       <!-- 启用 SMM：支持 UEFI Secure Boot(Windows 11 必需) -->
    <hyperv>                <!-- 启用 Hyper-V 兼容特性：提升 Windows 客户机性能 -->
        <relaxed state='on'/>
        <vapic state='on'/>
        <spinlocks state='on' retries='8191'/>
    </hyperv>
    <vmx state='on'/>       <!-- 启用 VMX：支持 Intel VT-x 嵌套虚拟化（如在虚拟机中再开虚拟机） --></features>
```

2. 常见问题排查（基于<features>配置）

| 问题现象     | 可能原因        | 解决方案                           |
|----------|-------------|--------------------------------|
| 虚拟机无法软关机 | 未启用<acpi/>或 | 1. 添加<acpi/>标签; 2. 安装 Guest OS |

| 问题现象                        | 可能原因                                                   | 解决方案                                          |
|-----------------------------|--------------------------------------------------------|-----------------------------------------------|
| （只能强制断电）                    | Guest OS 无 ACPI 驱动的 ACPI 驱动（如 Linux 内核更新、Windows 驱动修复） |                                               |
| 虚拟机配置 4 核 CPU，但仅识别 1 核      | 未启用<apic/>或 IO-APIC                                    | 1. 确保<apic/>标签存在；2. 检查是否启用<ioapic/>（现代平台默认启用） |
| 虚拟机启动报“VMPort 漏洞风险”警告       | <vmport state='on' />未禁用                               | 将<vmport>的 state 改为 off，若需调试旧系统则临时启用并加固宿主机    |
| Windows 11 无法启用 Secure Boot | 未启用<smm state='on' />                                  | 在<features>中添加<smm state='on' />（需配合 UEFI 固件） |

四、总结

| 字段        | 核心作用                   | 关键参数 / 状态       | 技术依赖                      | 安全与兼容性建议            |
|-----------|------------------------|-----------------|---------------------------|---------------------|
| <acpi/>   | 电源管理（软关机 / 休眠）+ 硬件识别   | 无参数（启用即生效）      | Guest OS 需有 ACPI 驱动       | 必启用（现代系统不可替代）       |
| <apic/>   | 多 CPU / 多核中断管理，支持多设备并发 | 无参数（启用即生效）      | 虚拟机需配置多核心（<vcpu>>1）       | 必启用（多核虚拟机不可替代）      |
| <vmport/> | 老旧虚拟机调试 / 通信接口         | state='off'（推荐） | 仅兼容旧版 VMware Tools / 调试场景 | 必禁用（规避安全风险，现代系统无依赖） |

<features> 配置的核心原则是：“启用必需功能，禁用冗余 / 危险功能”——<acpi/>和<apic/>是现代系统的“刚需”，<vmport/>是“安全隐患”，需根据 Guest OS 的类型和版本灵活调整，确保性能、兼容性与安全性平衡。

4.3.3.CPU 特性匹配配置

```
<cpu mode='host-model' check='partial' />
```

CPU 虚拟化模式的核心设置，决定了虚拟机如何使用物理 CPU 资源，直接影响性能、兼容性和功能支持。下面详细解析每个参数的含义、技术原理及实际应用场景。

一、整体结构：<cpu> 节点

<cpu> 是虚拟机 CPU 配置的顶层容器，用于定义 CPU 虚拟化策略——包括虚拟机使用物理 CPU 的方式（如完全模拟、部分透传、全透传）、兼容性检查规则、迁移能力等。它的核心作用是在“性能”与“兼容性”之间找到平衡：既要让虚拟机高效使用物理 CPU 资源，又要确保不同硬件环境下的虚拟机（如迁移到其他宿主机时）能正常运行。

二、核心参数详解

1. mode="host-passthrough": CPU 虚拟化模式（核心参数）

（1）基本定义

含义：启用“CPU 完全透传”模式，将物理 CPU 的全部特性、指令集、型号信息直接暴露给虚拟机（Guest OS），不做任何修改或模拟。

本质：虚拟机“直接看到”物理 CPU 的真实型号（如 Intel i9-13900K）、支持的所有指令集（如 AVX512、AMX）和硬件特性（如虚拟化嵌套、TSX），仿佛直接运行在物理机上。

（2）与其他模式的对比（性能与兼容性）

| 模式               | 技术原理                   | 性能表现      |                  | 兼容性                                       | 适用场景 |
|------------------|------------------------|-----------|------------------|-------------------------------------------|------|
|                  |                        | 接近物理机（最优） | 接近虚拟机（最差）        |                                           |      |
| host-passthrough | 完全透传物理 CPU 特性，不做模拟     | 优秀        | 较差（依赖宿主机 CPU 型号） | 高性能场景（如数据库、AI 计算）、需要特定指令集的应用（如 AVX512 加速） |      |
| host-model       | 模拟与宿主机 CPU 特性接近的“标准模型” | 良好（略低于透传） | 较好（同架构通用）        | 兼顾性能与迁移的场景（如私有云）                          |      |
| custom           | 手动指定 CPU 特性（如指令集、缓存大小） | 中等（按需配置）  | 灵活（可适配旧系统）       | 需要兼容老旧 OS（如 Windows Server 2008）          |      |
| emulated         | 完全模拟物理 CPU             | 较差        | 最优（跨架构）          | 兼容性优先场景（如老旧系统迁移）                          |      |



| 模式 | 技术原理                     | 性能表现         | 兼容性               | 适用场景 |
|----|--------------------------|--------------|-------------------|------|
|    | 定型号 CPU（如 Intel Xeon E5） | （模拟开销架构兼容）大） | x86 宿主机运行 ARM 系统） |      |

（3）核心优势与限制

优势：

性能最大化：无指令集模拟开销，虚拟机可直接使用物理 CPU 的全部能力（如多线程、硬件加速指令）；

支持特殊指令集：可运行依赖高级指令集的应用（如 AI 训练依赖 AVX512，加密运算依赖 AES-NI）；

真实硬件识别：Guest OS 能看到真实 CPU 型号（部分软件需验证硬件型号时必备）。

限制：

迁移兼容性差：虚拟机只能迁移到 CPU 型号完全相同的宿主机（否则因指令集不匹配启动失败）；

依赖宿主机 CPU：若物理 CPU 不支持某特性（如 AMD CPU 无 Intel 的 TSX 指令），虚拟机也无法使用；

不支持 CPU 超分：无法通过“虚拟 CPU 核心数超过物理核心数”实现超分（需 host-model 或 custom 模式）。

2. check="none"：CPU 特性兼容性检查策略

（1）基本定义

含义：禁用虚拟机启动时的“CPU 特性兼容性检查”，即不验证物理 CPU 是否支持虚拟机所需的特性。

本质：默认情况下，虚拟化平台会检查物理 CPU 是否满足虚拟机配置的特性（如指令集、缓存），若不满足则拒绝启动；check="none" 会跳过该检查，强制启动虚拟机。

（2）常见参数及场景

| 参数值  | 行为描述           | 适用场景                    | 风险                          |
|------|----------------|-------------------------|-----------------------------|
| none | 完全跳过兼容性检查，强制启动 | 明确物理 CPU 支持所需特性，且需要快速启动 | 若物理 CPU 不支持某特性，虚拟机可能崩溃或运行异常 |

| 参数值     | 行为描述                   | 适用场景           | 风险                    |
|---------|------------------------|----------------|-----------------------|
| partial | 仅检查关键特性（如虚拟化扩展），忽略次要特性 | 平衡兼容性与启动速度     | 次要特性不支持时可能导致部分功能失效    |
| full    | 检查所有 CPU 特性，不匹配则拒绝启动   | 严格保证兼容性（如生产环境） | 启动前检查耗时增加，迁移时易因微小差异失败 |

（3）与 host-passthrough 的配合

在 mode="host-passthrough"模式下，check="none"是推荐配置：

因为 host-passthrough 已确保虚拟机使用物理 CPU 的全部特性，无需额外检查（检查反而会增加启动耗时）；

若设为 full，可能因某些次要特性（如缓存大小的微小差异）导致启动失败，而这些差异实际不影响运行。

### 3. migratable="on": CPU 迁移能力开关

（1）基本定义

含义：允许虚拟机在 host-passthrough 模式下进行跨宿主机迁移（Live Migration）。

本质：默认情况下，host-passthrough 模式因强依赖物理 CPU 型号而禁止迁移；migratable="on" 会通过以下方式 “有限度地支持迁移”：

迁移时检查目标宿主机 CPU 是否与源宿主机 “核心特性兼容”（如指令集、虚拟化扩展一致）；

忽略非核心差异（如缓存大小、主频），允许迁移到同系列 CPU（如 Intel i9-13900K 迁移到 i9-13900F）。

#### （2）参数取值与影响

| 参数值 | 迁移能力             | 适用场景                      | 限制条件                                          |
|-----|------------------|---------------------------|-----------------------------------------------|
| on  | 允许迁移到核心特性兼容的 CPU | 需要迁移的高性能场景（如负载均衡）         | 目标 CPU 必须与源 CPU 同架构、同系列（如 AMD EPYC 7003→7004） |
| off | 禁止迁移（默认值）        | 固定宿主机运行的虚拟机（如物理机替代场的兼容性问题 | 无迁移能力，但避免因迁移导致                                |

| 参数 | 迁移能力 | 适用场景 | 限制条件 |
|----|------|------|------|
| 值  |      | 景)   |      |

### (3) 实际迁移限制

即使 `migratable="on"`，`host-passthrough` 模式的迁移仍有严格限制：

跨厂商迁移不支持（如 Intel CPU 宿主机→AMD CPU 宿主机，因指令集差异大）；

跨代差异大的 CPU 可能失败（如 Intel 10 代酷睿→13 代酷睿，部分新指令集不兼容）；

迁移后若虚拟机使用了目标 CPU 不支持的特性，会立即崩溃（需提前验证 CPU 兼容性）。

## 三、实际配置建议与典型场景

### 1. 何时使用 `host-passthrough` 模式？

高性能计算（HPC）：如运行科学计算、AI 训练等依赖 CPU 密集型任务的场景，需最大化利用物理 CPU 性能；

特殊指令集依赖：应用程序明确需要 AVX512、AMX、SHA 等高级指令集（如加密软件、深度学习框架）；

软件授权绑定：部分商业软件通过 CPU 型号验证授权（如某些工业软件），需虚拟机识别真实 CPU；

物理机替代：虚拟机需完全模拟物理机环境（如替代老旧物理服务器，避免软件兼容性问题）。

### 2. 不适合使用的场景

跨宿主机迁移频繁：如公有云、动态负载均衡场景（建议用 `host-model` 模式）；

CPU 型号不一致的集群：宿主机 CPU 型号混杂（如既有 Intel i7 也有 i9），迁移易失败；

需要 CPU 超分：需分配虚拟 CPU 核心数超过物理核心数（如 4 核物理 CPU 运行 8 核虚拟机）。

### 3. 完整配置示例（配合其他 CPU 参数）

xml

```
<cpu mode="host-passthrough" check="none" migratable="on">
```

```
  <topology sockets="1" cores="8" threads="2"/> <!-- 虚拟 CPU 拓扑: 1 socket, 8 核, 超线程 (共 16 线程) -->
```

```
<cache mode="passthrough"/> <!-- 缓存透传：暴露真实 CPU 缓存大小 -->

<feature policy="require" name="vmx"/> <!-- 强制启用 Intel VT-x 嵌套虚拟化 --></cpu>
```

该配置适用于：需要 8 核 16 线程高性能、支持嵌套虚拟化（虚拟机内再开虚拟机）、允许有限度迁移的场景。

四、总结

| 参数         | 核心作用                  | 关键取值             | 核心权衡点                 |
|------------|-----------------------|------------------|-----------------------|
| mode       | 定义 CPU 虚拟化策略，决定性能与兼容性 | host-passthrough | 性能最优，但依赖物理 CPU 型号     |
|            | 控制 CPU 特性兼容性检查强度      | none（推荐）         | 跳过检查加速启动，但可能隐藏兼容性问题   |
| check      |                       |                  |                       |
| migratable | 允许 / 禁止跨宿主机迁移         | on/off           | 开启迁移支持灵活性，但受 CPU 型号限制 |
|            |                       |                  |                       |

host-passthrough 模式是“性能优先”的选择，适合对 CPU 性能要求极高且迁移需求低的场景。配置时需注意：物理 CPU 型号需稳定（避免频繁更换），迁移仅能在同系列 CPU 间进行，且需提前验证目标宿主机的兼容性。

4.3.4.时钟同步配置

```
<clock offset='utc'>

  <timer name='rtc' tickpolicy='catchup'/>

  <timer name='pit' tickpolicy='delay'/>

  <timer name='hpet' present='no'/>

</clock>
```

<clock offset='utc'>：虚拟机时钟采用 UTC 时间（世界协调时间），而非主机本地时间，避免跨时区部署时的时间偏差（Ubuntu 等 Linux 系统默认推荐 UTC 时间）。

```
<timer name='rtc' tickpolicy='catchup'/>:
```

rtc：实时时钟（Real-Time Clock），用于虚拟机断电后保存时间；

tickpolicy='catchup': 时钟偏差时的同步策略 —— 若虚拟机时钟比主机慢，快速“追赶”到正确时间（避免时间滞后）。

```
<timer name='pit' tickpolicy='delay'/>:
```

**pit:** 可编程间隔定时器（Programmable Interval Timer），用于生成固定频率的中断（如系统时钟滴答）；

**tickpolicy='delay':** 若主机负载高导致中断延迟，推迟下一次中断而非跳变，避免时钟紊乱。

**<timer name='hpet' present='no'/>:** 禁用 HPET（高精度事件定时器），部分老系统对 HPET 支持不佳，禁用可提升兼容性（Ubuntu 24.10 虽支持 HPET，但默认关闭不影响使用）。

#### 4.3.5.电源与崩溃处理策略

```
<on_poweroff>destroy</on_poweroff>
```

```
<on_reboot>restart</on_reboot>
```

```
<on_crash>destroy</on_crash>
```

```
<pm>
```

```
<suspend-to-mem enabled='no'/>
```

```
<suspend-to-disk enabled='no'/>
```

```
</pm>
```

电源管理与异常处理策略的核心设置，决定了虚拟机在遇到关机、重启、崩溃等事件时的行为，以及是否支持休眠 / 休眠到磁盘功能。下面详细解析每个字段的含义、参数及实际应用场景。

##### 一、整体结构与作用

这些配置属于虚拟机的生命周期管理策略，通过预设规则自动化处理各类系统事件，避免人工干预，同时控制虚拟机的电源状态转换（如休眠）。核心作用是：

确保虚拟机在异常情况下（如崩溃）的资源释放或恢复；

标准化开关机、重启的行为逻辑（如是否保留内存状态）；

控制高级电源管理功能（休眠到内存 / 磁盘）的可用性。

##### 二、事件处理策略（<on\_poweroff> <on\_reboot> <on\_crash>）

这三个标签定义了虚拟机在特定事件发生时的系统行为，是保障虚拟机稳定运行和资源管理的基础配置。

##### 1. <on\_poweroff>destroy</on\_poweroff>: 虚拟机主动关机时的行为

含义：当虚拟机内部执行“关机”操作（如 Windows 的“关机”、Linux 的 `shutdown -h now`）时，虚拟化平台应采取的动作。

参数解析：**destroy** 表示“彻底销毁”虚拟机进程，释放其占用的 CPU、内存、磁盘 IO 等资源（相当于物理机的“断电”）。

其他可选参数：

**preserve**：保留虚拟机状态（内存数据不释放），但停止运行（类似“暂停”，需手动恢复）；

**restart**：关机后自动重启（适用于需要持续运行的服务，如 Web 服务器）。

适用场景：**destroy** 是默认且最常用的配置，确保关机后完全释放资源，避免闲置占用。

## 2. `<on_reboot>restart</on_reboot>`：虚拟机内部重启时的行为

含义：当虚拟机内部执行“重启”操作（如 Windows 的“重启”、Linux 的 `reboot`）时，虚拟化平台的响应动作。

参数解析：**restart** 表示“自动重启”虚拟机，模拟物理机的“重启”流程（先关闭再启动，内存数据清空）。

其他可选参数：

**destroy**：重启请求被忽略，直接销毁虚拟机（适用于需要严格控制启动的场景，如安全隔离环境）；

**preserve**：重启请求被忽略，保留虚拟机当前状态（几乎不实用，仅用于特殊调试）。

适用场景：**restart** 是唯一符合用户预期的配置，确保“重启”操作与物理机体验一致。

## 3. `<on_crash>destroy</on_crash>`：虚拟机崩溃时的行为

含义：当虚拟机因内核崩溃、蓝 / 黑屏、无响应等异常情况“崩溃”时，虚拟化平台的处理策略。

参数解析：**destroy** 表示“强制销毁”崩溃的虚拟机进程，释放资源（避免僵尸进程占用资源）。

其他可选参数：

**restart**：自动重启虚拟机（适用于高可用场景，如数据库服务，崩溃后快速恢复）；

**preserve**：保留崩溃时的内存状态（用于调试，可提取内存快照分析崩溃原因）；

**dump**：先保存内存快照（用于分析），再销毁虚拟机（兼顾调试与资源释放）。

适用场景：

普通场景用 **destroy**：避免崩溃的虚拟机消耗资源；

高可用场景用 **restart**：如 Web 服务、应用服务器，优先保证服务连续性；

调试场景用 **preserve** 或 **dump**：如开发环境，需分析崩溃原因。

### 三、电源管理配置（<pm>节点）

<pm>（Power Management）节点用于控制虚拟机的高级电源状态转换，主要是“休眠到内存”和“休眠到磁盘”功能的开关。

#### 1. <suspend-to-mem enabled='no'/>：禁用“休眠到内存”

全称：Suspend to RAM（挂起到内存），类似物理机的“睡眠”模式。

含义：将虚拟机当前内存数据保存到物理内存中，关闭 CPU、硬盘等设备供电（仅保留内存供电），需少量电量维持内存数据。

参数解析：enabled='no'表示禁用该功能，虚拟机无法执行“睡眠”操作。

启用（enabled='yes'）的优缺点：

优点：恢复速度极快（内存数据未丢失，1-2 秒即可恢复运行）；

缺点：休眠期间仍占用物理内存，且宿主机断电会导致数据丢失。

适用场景：

禁用（no）：服务器虚拟机（需持续运行，无需休眠）、资源紧张的宿主机；

启用（yes）：桌面虚拟机（如开发用 Windows 虚拟机，临时不用时快速休眠节省功耗）。

#### 2. <suspend-to-disk enabled='no'/>：禁用“休眠到磁盘”

全称：Suspend to Disk（挂起到磁盘），类似物理机的“休眠”模式。

含义：将虚拟机内存数据完整写入磁盘（生成快照文件），然后完全关闭虚拟机（不占用 CPU / 内存资源）。

参数解析：enabled='no'表示禁用该功能，虚拟机无法执行“休眠”操作。

启用（enabled='yes'）的优缺点：

优点：休眠后不占用内存资源，宿主机断电也不会丢失数据；

缺点：恢复速度慢（需从磁盘读取大量数据，耗时取决于内存大小），占用磁盘空间（与虚拟机内存大小相当）。

适用场景：

禁用（no）：高频使用的虚拟机（如数据库，频繁休眠 / 恢复影响效率）；

启用（yes）：偶尔使用的虚拟机（如测试环境，长期不用时释放内存资源）。

### 四、配置组合与典型场景

1. 服务器虚拟机（如 Web 服务器、数据库）

```
<on_poweroff>destroy</on_poweroff>    <!-- 关机后释放资源
--><on_reboot>restart</on_reboot>        <!-- 正常重启服务
--><on_crash>restart</on_crash>          <!-- 崩溃后自动恢复服务 --><pm>
    <suspend-to-mem enabled='no'>        <!-- 禁用睡眠（需持续运行） -->
    <suspend-to-disk enabled='no'>        <!-- 禁用休眠（避免恢复延迟） --></pm>
```

核心目标：优先保证服务连续性，禁用不必要的电源管理功能。

2. 开发 / 测试用桌面虚拟机

```
xml
<on_poweroff>destroy</on_poweroff>    <!-- 正常关机释放资源
--><on_reboot>restart</on_reboot>        <!-- 正常重启 --><on_crash>dump</on_crash>
<!-- 崩溃后保存快照用于调试 --><pm>
    <suspend-to-mem enabled='yes'>        <!-- 允许快速睡眠（临时不用时） -->
    <suspend-to-disk enabled='yes'>        <!-- 允许休眠（长期不用时） --></pm>
```

核心目标：兼顾使用便捷性与调试需求，保留电源管理灵活性。

五、总结

| 字段 / 子节点          | 核心作用            | 关键参数 / 取值       | 核心决策因素             |
|-------------------|-----------------|-----------------|--------------------|
| <on_poweroff>     | 定义虚拟机主动关机后的行为   | destroy（默认）     | 是否需要保留资源或自动重启      |
| <on_reboot>       | 定义虚拟机内部重启后的行为   | restart（默认）     | 是否允许正常重启流程         |
| <on_crash>        | 定义虚拟机崩溃后的处理策略   | destroy/restart | 优先资源释放、服务恢复还是调试分析  |
| <suspend-to-mem>  | 控制“休眠到内存”（睡眠）功能 | enabled='no'    | 是否需要快速恢复且可接受内存占用   |
| <suspend-to-disk> | 控制“休眠到磁盘”（休眠）功能 | enabled='no'    | 是否需要长期释放资源且可接受恢复延迟 |

这些配置的核心是根据虚拟机的用途（服务器 / 桌面 / 测试）平衡“可用性”“资源效率”和“调试需求”，服务器优先保证连续性，桌面 / 测试机可保留更多灵活性。



## 5.虚拟机设备管理

### 5.1.模拟器路径

这个不需要做变更。

```
<emulator>/usr/libexec/qemu-kvm</emulator>
```

指定 KVM 虚拟机的底层模拟器为 `qemu-kvm`（路径为欧拉系统默认路径），负责模拟 CPU、内存、硬件设备，是虚拟机运行的“引擎”。

### 5.2.磁盘设备管理

| 内容        | 命令                                 | 说明                    |
|-----------|------------------------------------|-----------------------|
| 添加磁盘      | <code>virsh attach-disk</code>     | 为虚拟机临时添加磁盘            |
| 分离磁盘      | <code>virsh detach-disk</code>     | 从虚拟机移除磁盘              |
| 磁盘容量调整    | <code>virsh blockresize</code>     | 调整磁盘大小                |
| 磁盘镜像合并    | <code>virsh blockcommit</code>     | 合并磁盘镜像（如快照合并到基础镜像）    |
| 磁盘数据复制    | <code>virsh blockcopy</code>       | 复制磁盘数据（支持在线操作）        |
| 磁盘操作终止    | <code>virsh blockjob</code>        | 终止正在进行的磁盘操作（如复制）      |
| 磁盘镜像拉取    | <code>virsh blockpull</code>       | 将外部磁盘数据拉取到本地镜像        |
| 磁盘空间阈值告警  | <code>virsh domblkthreshold</code> | 设置磁盘空间阈值告警            |
| 磁盘 I/O 调优 | <code>virsh blkdeviotune</code>    | 查看 / 调整磁盘的 I/O 调度参数   |
| 磁盘 I/O 限制 | <code>virsh blkiotune</code>       | 查看 / 调整虚拟机的块设备 I/O 限制 |
| 更改 CD 介质  | <code>virsh Change-media</code>    | 更改 Cd 介质或者软盘驱动器       |

```
<disk type='file' device='disk'>
  <driver name='qemu' type='qcow2'/>
  <source file='/home/vm/.ubuntu.qcow2'/>
  <target dev='vda' bus='virtio'/>
  <address type='pci' domain='0x0000' bus='0x04' slot='0x00' function='0x0'/>
</disk>
```

| 素    | 属性     | 含义             | 属性值及其含义                                                                                |
|------|--------|----------------|----------------------------------------------------------------------------------------|
| disk | type   | 指定后端存储介质类型。    | <b>block</b> : 块设备<br><b>file</b> : 文件设备<br><b>dir</b> : 目录路径<br><b>network</b> : 网络磁盘 |
|      | device | 指定呈现给虚拟机的存储介质。 | <b>disk</b> : 磁盘（默认）<br><b>floppy</b> : 软盘<br><b>cdrom</b> : 光盘                        |

| 子元素    | 子元素含义                               | 属性说明                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| source | 指定后端存储介质，与 disk 元素的属性“type”指定类型相对应。 | <ul style="list-style-type: none"> <li>• <b>file</b>: 对应 file 类型，值为对应文件的完全限定路径。</li> <li>• <b>dev</b>: 对应 block 类型，值为对应主机设备的完全限定路径。</li> <li>• <b>dir</b>: 对应 dir 类型，值为用作磁盘目录的完全限定路径。</li> <li>• <b>protocol</b>: 使用的协议。</li> <li>• <b>name</b>: rbd 磁盘名称，格式为: \$pool/\$volume。</li> <li>• <b>host name</b>: mon 地址。</li> <li>• <b>port</b>: mon 地址的端口。</li> </ul>                                          |
| driver | 指定后端驱动器的详细信息。                       | <ul style="list-style-type: none"> <li>• <b>type</b>: 磁盘格式的类型，常用的有“raw”和“qcow2”，需要与 source 的格式一致。</li> <li>• <b>io</b>: 磁盘 IO 模式，支持“native”和“threads”选项。</li> <li>• <b>cache</b>: 磁盘的 cache 模式，可选项有“none”、“writethrough”、“writeback”、“directsync”等。</li> <li>• <b>iothread</b>: 指定为磁盘分配的 IO 线程。</li> <li>• <b>error_policy</b>: IO 写错误发生时的处理策略，可选项有“stop”、“report”、“ignore”、“enospace”、“retry”等。</li> </ul> |

| 子元素             | 子元素含义                                 | 属性说明                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-----------------|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                 |                                       | <ul style="list-style-type: none"> <li>• <b>error_policy</b>: IO 读错误发生时的处理策略, 可选项有 “stop”、“report”、“ignore”、“enospc”、“retry” 等。</li> <li>• <b>retry_interval</b>: IO 错误重试间隔, 范围为 0-MAX_INT, 单位为毫秒, 仅 <b>error_policy</b>=“retry”或 <b>error_policy</b>=“retry” 时可配置。</li> <li>• <b>retry_timeout</b>: IO 错误重试超时时间, 范围为 0-MAX_INT, 单位为毫秒, 仅 <b>error_policy</b>= “retry” 或 <b>error_policy</b>= “retry” 时可配置。</li> </ul> |
| <b>target</b>   | 指磁盘呈现给虚拟机的总线和设备。                      | <ul style="list-style-type: none"> <li>• <b>dev</b>: 指定磁盘的逻辑设备名称, 如 SCSI、SATA、USB 类型总线常用命令习惯为 <b>sd[a-p]</b>, IDE 类型设备磁盘常用命名习惯为 <b>hd[a-d]</b>。</li> <li>• <b>bus</b>: 指定磁盘设备的类型, 常见的有 “scsi”、“usb”、“sata”、“virtio” 等类型。</li> </ul>                                                                                                                                                                                    |
| <b>boot</b>     | 表示此磁盘可以作为启动盘使用。                       | <ul style="list-style-type: none"> <li>• <b>order</b>: 指定磁盘的启动顺序。</li> </ul>                                                                                                                                                                                                                                                                                                                                           |
| <b>readonly</b> | 表示磁盘具有只读属性, 磁盘内容不可以被虚拟机修改, 通常与光驱结合使用。 | -                                                                                                                                                                                                                                                                                                                                                                                                                      |

**type='file'**: 磁盘镜像为本地文件（而非块设备 **/dev/sdb** 或网络存储），适合单机部署。

**device='disk'**: 设备类型为硬盘（区别于 **cdrom** 光驱）。

**<driver name='qemu' type='qcow2'/>**:

**name='qemu'**: 使用 QEMU 内置的磁盘驱动。

`type='qcow2'`: 磁盘镜像格式为 `qcow2` (QEMU 镜像格式 2), 支持动态扩容 (初始小文件, 随数据增长变大)、快照、压缩, 是 KVM 最常用的镜像格式。

`<source file='/home/vm/.ubuntu.qcow2'/>`: 磁盘镜像文件的绝对路径 (存储在 `/home/vm/` 目录, 文件名以 `.` 开头表示隐藏文件)。

`<target dev='vda' bus='virtio'/>`:

`dev='vda'`: 虚拟机内识别的设备名 (类似物理机的 `/dev/vda`, 即第一块虚拟硬盘)。

`bus='virtio'`: 使用 `virtio` 总线 (KVM 高性能总线, 需客户机安装 `virtio` 驱动, Ubuntu 24.10 已默认集成, 无需手动安装), 相比 `IDE` 总线性能提升 3-5 倍。

`<address type='pci'.../>`: 虚拟磁盘在 `PCI` 总线上的地址 (`bus=0x04, slot=0x00`), 用于硬件寻址, 确保设备不冲突。

`<!-- 使用物理磁盘 -->`

`<disk type='block' device='disk'>`

`<driver name='qemu' type='raw'/>`

`<source dev='/dev/sdb'/>`

`<!-- 物理机整块磁盘 -->`

`<target dev='vda' bus='virtio'/>`

`</disk>`

`<!-- 使用物理分区 -->`

`<disk type='block' device='disk'>`

`<driver name='qemu' type='raw'/>`

`<source dev='/dev/sdb1'/>`

`<!-- 物理机分区 -->`

`<target dev='vdb' bus='virtio'/>`

`</disk>`

`<!-- 使用 raw 文件 -->`

`<disk type='file' device='disk'>`

`<driver name='qemu' type='raw'/>`

`<source file='/home/vm/.ubuntu.qcow2'/>`

`<target dev='vda' bus='virtio'/>`

`</disk>`

### 5.2.1.添加磁盘

以下是 `virsh attach-disk` 命令所有选项的详细说明，包括其作用和参数含义：

#### `--address`

作用：指定磁盘在虚拟机中的 PCI 地址（针对 PCI 总线上的设备）。

参数：需提供具体的 PCI 地址格式，如 `0000:04:00.0`。

适用场景：当需要为磁盘指定固定的 PCI 总线位置时使用。

#### `--alias`

作用：为磁盘设备设置一个别名，便于在 XML 配置中标识和管理。

参数：自定义字符串（如 `disk-extra`）。

示例：`--alias extra-disk`

#### `--cache`

作用：设置磁盘的缓存模式，影响 I/O 性能和数据安全性。

参数：常见值包括：

**default**：使用默认缓存策略

**none**：禁用缓存（数据直接写入磁盘）

**writeback**：写缓存启用（数据先写入缓存，再异步刷盘）

**writethrough**：写透模式（数据同时写入缓存和磁盘）

**directsync**：类似 **writethrough**，但针对直接 I/O

示例：`--cache writeback`

#### `--config`

作用：仅修改虚拟机的配置文件，不影响当前运行状态，下次启动时生效。

参数：无额外参数，作为开关使用。

注意：若需同时生效，需与 `--live` 配合（`--live --config`）。

#### `--current`

作用：根据虚拟机当前状态（运行中 / 关闭）应用配置。

参数：无额外参数，作为开关使用。

场景：当不确定虚拟机状态时，自动适配生效方式（运行中则热生效，关闭则修改配置）。

#### `--domain`

作用：指定要附加磁盘的虚拟机名称、ID 或 UUID（必填选项）。

参数：虚拟机的唯一标识（如 `ubuntuv`、`1` 或 `UUID` 字符串）。

示例：`--domain ubuntuv`

`--driver`

作用：指定磁盘的存储驱动类型。

参数：常见值包括 `qemu`、`kvm` 等，具体取决于 `hypervisor`。

示例：`--driver qemu`

`--io`

作用：设置 `I/O` 调度策略（针对块设备）。

参数：如 `native`（原生 `I/O`）、`threads`（线程化 `I/O`）等，取决于系统支持。

示例：`--io native`

`--iothread`

作用：将磁盘 `I/O` 绑定到指定的 `I/O` 线程，优化性能。

参数：线程编号（如 `1`、`2`，需提前在虚拟机配置中定义 `I/O` 线程）。

示例：`--iothread 1`

`--live`

作用：在运行中的虚拟机上立即生效（热插拔磁盘），不修改配置文件。

参数：无额外参数，作为开关使用。

注意：若需持久化，需与 `--persistent` 或 `--config` 配合。

`--mode`

作用：指定磁盘的访问模式。

参数：

`readonly`：只读模式（虚拟机无法写入磁盘）

`readwrite`：读写模式（默认，虚拟机可读写磁盘）

示例：`--mode readonly`（如附加 `ISO` 镜像时常用）

`--multifunction`

作用：将磁盘设备标记为多功能 `PCI` 设备的一部分（用于共享 `PCI` 功能）。

参数：无额外参数，作为开关使用。

适用场景：复杂 `PCI` 设备配置，一般用户较少用到。

`--persistent`

作用：使磁盘附加配置在虚拟机重启后仍然有效（持久化到配置文件）。

参数：无额外参数，作为开关使用。

注意：与 `--live` 配合可实现“热插拔且持久化”。

#### `--print-xml`

作用：不实际执行附加操作，仅输出对应的 XML 配置片段。

参数：无额外参数，作为开关使用。

用途：用于预览或手动编辑 XML 配置时参考。

#### `--rawio`

作用：启用 raw I/O 模式（绕过文件系统缓存，直接访问块设备）。

参数：无额外参数，作为开关使用。

适用场景：需要高性能 I/O 的场景，通常与块设备（`--sourcetype block`）配合。

#### `--serial`

作用：为磁盘设备设置序列号（模拟物理磁盘的序列号）。

参数：自定义字符串（如 `disk-sn-12345`）。

用途：部分操作系统或应用可能依赖磁盘序列号识别设备。

#### `--shareable`

作用：允许磁盘被多个虚拟机同时附加（需配合共享存储）。

参数：无额外参数，作为开关使用。

注意：需存储本身支持共享（如集群文件系统、iSCSI 等），否则可能导致数据损坏。

#### `--source`

作用：指定磁盘的源路径或标识（必填选项），与 `--sourcetype` 配合使用。

参数：

若为文件：本地路径（如 `/var/lib/libvirt/images/disk1.qcow2`）

若为块设备：设备路径（如 `/dev/sdb`）

若为网络存储：网络标识（如 iSCSI 的 IQN: `iqn.2023-01.com.example:storage`）

示例：`--source /var/lib/libvirt/images/extra.img`

#### `--sourcetype`

作用：明确指定 `--source` 的类型（避免自动识别错误）。

参数：

**file**：源为本地文件（默认值）

**block**：源为块设备（如物理硬盘、LVM 逻辑卷）

**network:** 源为网络存储（如 NFS、iSCSI、GlusterFS）

示例: `--source /dev/sdb --sourcetype block`

**--subdriver**

作用: 指定存储子驱动类型（针对特定格式或协议）。

参数: 如 `qcow2`、`raw`（针对文件类型），或 `iscsi`（针对网络存储）。

示例: `--subdriver qcow2`（明确指定 QCOW2 格式）

**--target**

作用: 指定磁盘在虚拟机内部的设备名称（必填选项）。

参数: 根据总线类型命名，如:

**virtio 总线:** `vda`、`vdb`（推荐，性能最优）

**IDE 总线:** `hda`、`hdb`

**SCSI 总线:** `sda`、`sdb`

示例: `--target vdb`（虚拟机内显示为 `/dev/vdb`）

**--targetbus**

作用: 指定磁盘连接的目标总线类型。

参数: `virtio`（推荐，高性能）、`ide`、`scsi`、`usb` 等。

示例: `--targetbus virtio`（使用 `virtio` 总线，需虚拟机安装对应驱动）

**--type**

作用: 指定磁盘的设备类型。

参数:

**disk:** 普通磁盘（默认）

**cdrom:** 光驱（通常用于附加 ISO 镜像）

**floppy:** 软盘（较少使用）

示例: `--type cdrom --source /tmp/ubuntu.iso`（附加 ISO 作为光驱）

**--wwn**

作用: 为磁盘设置 WWN（World Wide Name），用于唯一标识存储设备。

参数: WWN 字符串（如 `naa.6001405abcdef123`）。

用途: 模拟物理存储的 WWN，便于虚拟机内识别和管理。

总结



使用 `virsh attach-disk` 时，核心必填选项为 `--domain`、`--source`、`--target`，其他选项根据需求补充（如 `--live` 热插拔、`--persistent` 持久化、`--cache` 调整缓存策略等）。根据存储类型（文件 / 块设备 / 网络）和虚拟机需求组合选项即可

| 对比维度       | qcow2 镜像<br>(文件类型)                                                              | raw 镜像<br>(文件类型)                                                              | 物理磁盘<br>(块设备)                                                                      | 物理分区<br>(块设备)                                                                       |
|------------|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| 核心定义       | QEMU 专属镜像格式，支持动态扩容量、内部快照等高级功能                                                   | 原始二进制无额外元数据开销                                                                 | 直接使用物理机整块磁盘<br>(如 <code>/dev/sdb</code> )                                          | 直接使用物理机分区<br>(如 <code>/dev/sdb1</code> )                                            |
| XML 配置关键参数 | <code>&lt;disk type='file' device='disk'&gt;&lt;driver type='qcow2'/&gt;</code> | <code>&lt;disk type='file' device='disk'&gt;&lt;driver type='raw'/&gt;</code> | <code>&lt;disk type='block' device='disk'&gt;&lt;source dev='/dev/sdb'/&gt;</code> | <code>&lt;disk type='block' device='disk'&gt;&lt;source dev='/dev/sdb1'/&gt;</code> |
| 存储占用特性     | 动态扩容：创建时仅占少量元数据空间，随数据写入增长                                                       | 预分配：创建时即占用全部指定空间（如 20G 镜像占 20G 磁盘）                                            | 独占物理磁盘全部容量，无法动态调整                                                                  | 独占物理分区容量，无法动态调整                                                                     |
| IO 性能      | qcow2 元数据（如快照链、动态块管理）性能略低于 raw / 物理设备                                           | 高：无元数据开销，接近物理设备性能                                                             | 极高：无任何中间层，直接映射物理设备 IO                                                              | 极高：同物理磁盘，仅容量限制为分区大小                                                                 |
| 快照支持       | ✓ 原生支持内部快照（ <code>virsh snapshot-create</code> ）、快照链、增量备份                       | ✗ 不支持原生内部快照（需依赖 LVM / 外部工具实现）                                                 | ✗ 不支持快照（物理设备无法创建虚拟快照）                                                              | ✗ 不支持快照（同物理磁盘）                                                                      |
| 动态扩容       | ✓ 支持<br>( <code>qemu-img resize</code> )                                        | ✓ 支持<br>( <code>qemu-img</code> )                                             | ✗ 无法扩容<br>(物理磁盘容量固定)                                                               | ✗ 无法扩容<br>(分区容量固定，)                                                                 |

|       | qcow2 镜像                                                      | raw 镜像                                        | 物理磁盘                                          | 物理分区                          |
|-------|---------------------------------------------------------------|-----------------------------------------------|-----------------------------------------------|-------------------------------|
| 对比维度  | (文件类型)                                                        | (文件类型)                                        | (块设备)                                         | (块设备)                         |
|       | +10G)，扩容后需调整虚拟机内文件系统                                          | resize)，但需预分配空间，扩容后需调整文件系统                    | 定，需更换硬件)                                      | 需调整物理分区表，风险高)                 |
| 压缩与加密 | ✓ 支持镜像压缩 (创建时 -o compression_type=zlib)、加密 (-o encryption=on) | ✗ 不支持原生压缩 / 加密 (需依赖文件系统或外部工具)                 | ✗ 不支持 (需物理设备本身支持硬件加密)                         | ✗ 不支持 (同物理磁盘)                 |
| 迁移与备份 | 易: 镜像为单个文件，支持在线迁移可复制，但体积大 (需共享存储)、复制备份                        | 中: 单个文件 (如 20G 镜像需备份 20G)                     | 难: 需依赖共享存储 (如 iSCSI)，无法直接复制，备份需 dd 命令 (耗时)    | 难: 同物理磁盘，备份需 dd 且仅备份单个分区      |
| 资源隔离性 | ✓ 完全隔离: 镜像文件独立，不影响其他虚拟机 / 物理机                                 | ✓ 完全隔离: 同 qcow2，文件独立                          | ✗ 独占: 物理磁盘被单个虚拟机占用，无法共享                       | ✗ 独占: 物理分区被单个虚拟机占用，无法共享       |
| 权限与安全 | 安全: 仅需 qemu 进程对镜像文件有读写权限，无直接访问物理设备风险                          | 安全: 同 qcow2，仅文件权限控制                           | 风险高: 需赋予 qemu 进程物理设备权限 (如 root)，恶意虚拟机可能攻击主机设备 | 风险高: 同物理磁盘，且误操作可能破坏分区表        |
| 兼容性   | 中: 仅支持 KVM/QEMU 平台，不支持 VMware/VirtualBox                      | 高: 所有虚拟化平台 (KVM、VMware、VirtualBox) 均支持，可跨平台迁移 | 低: 依赖物理设备接口 (如 NVMe/SATA)，更换主机需重新映射，兼容性差      | 低: 同物理磁盘，且依赖分区表格式 (如 GPT/MBR) |

| 对比维度 | qcow2 镜像<br>(文件类型)                                            | raw 镜像<br>(文件类型)                                  | 物理磁盘<br>(块设备)                                                        | 物理分区<br>(块设备)                            |
|------|---------------------------------------------------------------|---------------------------------------------------|----------------------------------------------------------------------|------------------------------------------|
| 适用场景 | 开发测试环境、<br>需快照 / 备份的生<br>产环境 (如 Web 服<br>务器)、对存储利用<br>率要求高的场景 | 对性能有要<br>求但无需快照的<br>场景 (如中小规模<br>数据库)、跨平台<br>迁移场景 | 高性能生产环<br>境 (如大型数据库、<br>高性能计算)、需<br>直接使用物理设备<br>特性 (如硬件<br>RAID) 的场景 | 需复用物理机<br>分区资源、对性能<br>要求高但无需整块<br>磁盘的场景  |
| 典型缺点 | IO 性能略低,<br>元数据损坏可能导<br>致镜像不可用                                | 占用空间大,<br>无快照功能, 备份移<br>耗时                        | 资源独占, 迁<br>移 / 备份复杂, 误<br>操作风险高                                      | 资源独占, 扩<br>容风险高 (需调整<br>物理分区表), 无<br>法共享 |

### 5.2.2.分离磁盘

`virsh detach-disk` 命令用于从虚拟机中移除已附加的磁盘，其选项主要控制移除操作的生效范围和行为。以下是各选项的详细说明：

#### `--domain`

作用：指定要移除磁盘的虚拟机名称、ID 或 UUID（必填选项）。

参数：虚拟机的唯一标识（如 `ubuntuv`、`1` 或 UUID 字符串）。

示例：`--domain ubuntuv`

#### `--target`

作用：指定要移除的磁盘在虚拟机内的设备名称（必填选项，与 `attach-disk` 时的 `--target` 对应）。

参数：如 `vda`、`hdb`、`sdc` 等（需与虚拟机内实际设备名一致）。

示例：`--target vdb`（移除虚拟机内的 `/dev/vdb` 设备）

#### `--live`

作用：仅对运行中的虚拟机生效（热移除磁盘），不修改配置文件，虚拟机重启后磁盘会重新出现。

参数：无额外参数，作为开关使用。

适用场景：临时移除磁盘，无需停机，后续可能重新附加。

#### **--config**

作用：仅修改虚拟机的配置文件，不影响当前运行状态，下次启动时磁盘才会被移除。

参数：无额外参数，作为开关使用。

适用场景：计划在虚拟机下次重启时永久移除磁盘，当前运行中仍可使用。

#### **--persistent**

作用：将移除操作持久化到配置文件，等效于 **--config**（两者功能相同，仅名称不同）。

参数：无额外参数，作为开关使用。

注意：与 **--live** 配合时（**--live --persistent**），可实现“热移除且永久生效”。

#### **--current**

作用：根据虚拟机当前状态（运行中 / 关闭）自动应用生效方式：

若虚拟机运行中：等效于 **--live**（热移除）。

若虚拟机已关闭：等效于 **--config**（修改配置）。

参数：无额外参数，作为开关使用。

适用场景：不确定虚拟机状态时，简化操作（无需手动判断用 **--live** 还是 **--config**）。

#### **--print-xml**

作用：不实际执行移除操作，仅输出移除磁盘后的 **XML** 配置片段（用于预览效果）。

参数：无额外参数，作为开关使用。

用途：验证移除操作是否会影响其他配置，或手动编辑 **XML** 时参考。

常用组合示例

热移除且永久生效（运行中虚拟机，重启后不恢复）：

**bash**

```
virsh detach-disk --domain ubuntuvm --target vdb --live --config
```

仅临时移除（运行中生效，重启后磁盘重新出现）：

**bash**

```
virsh detach-disk --domain ubuntuvm --target vdb --live
```

计划下次重启移除（当前运行中仍可用，重启后消失）：

```
bash
```

```
virsh detach-disk --domain ubuntuvm --target vdb --config
```

自动适配状态移除（无需关心虚拟机是否运行）：

```
bash
```

```
virsh detach-disk --domain ubuntuvm --target vdb --current
```

注意事项

移除磁盘前，建议先在虚拟机内卸载磁盘（如 `umount /dev/vdb`），避免文件系统损坏。

若磁盘是系统盘（如 `/dev/vda`），移除后可能导致虚拟机崩溃，需谨慎操作。

`--live` 热移除功能依赖虚拟机硬件支持（如 `virtio` 总线通常支持，`IDE` 可能不支持）。

### 5.2.3.调整磁盘容量

`virsh blockresize` 命令用于动态调整虚拟机磁盘的大小（扩展或收缩），支持在线（运行中）和离线（关闭状态）操作，适用于文件镜像（`qcow2/raw`）和块设备（`LVM` / 物理分区）。

以下是该命令的详细说明：

基本语法

```
bash
```

```
virsh blockresize <域名> <目标设备> <新大小> [选项]
```

<域名>：虚拟机名称、ID 或 UUID

<目标设备>：虚拟机内的磁盘设备名（如 `vda`，需与 `attach-disk` 时的 `--target` 一致）

<新大小>：目标大小（支持单位：K/M/G/T，如 `20G`、`1024M`）

主要选项

| 选项                     | 作用                                |
|------------------------|-----------------------------------|
| <code>--live</code>    | 对运行中的虚拟机生效（在线调整，无需停机）             |
| <code>--config</code>  | 仅修改配置文件，下次启动时生效（离线调整）             |
| <code>--current</code> | 根据虚拟机当前状态自动选择生效方式（运行中则在线，关闭则修改配置） |

适用场景与限制

支持的磁盘类型：

文件镜像：`qcow2`（推荐，支持动态扩容）、`raw`（支持扩容，需提前预分配空间）。

块设备：LVM 逻辑卷、物理分区（需底层存储支持扩容，如 LVM 可先扩展卷再调整虚拟机磁盘）。

限制：

收缩磁盘：仅支持 raw 格式和块设备，且需先在虚拟机内收缩文件系统（如 `resize2fs`），否则可能导致数据损坏。

qcow2 格式：仅支持扩容，不支持收缩（需通过 `qemu-img convert` 间接实现）。

系统盘（如 `/dev/vda`）扩容需谨慎，建议先备份数据。

操作示例

#### 1. 在线扩容虚拟机磁盘（运行中）

假设虚拟机 `ubuntuvvm` 的数据盘为 `vdb`（qcow2 格式），从 10G 扩容到 20G：

bash

# 1. 在线调整磁盘大小（立即生效）`virsh blockresize --live ubuntuvm vdb 20G`

# 2. 登录虚拟机，确认磁盘大小已更新（Linux 示例）

`lsblk` # 查看 `/dev/vdb` 的大小是否变为 20G

# 3. 扩展文件系统（以 `ext4` 为例，需先卸载或在线扩容）

`resize2fs /dev/vdb` # 在线扩容已挂载的 `ext4` 分区# 若为 XFS 文件系统，使用：

`xfs_growfs` /挂载点

#### 2. 离线调整（虚拟机关闭时）

调整虚拟机 `centosvm` 的系统盘 `vda`（raw 格式）从 30G 扩容到 50G：

bash

# 1. 确保虚拟机已关闭 `virsh shutdown centosvm`

# 2. 调整磁盘大小（修改配置，下次启动生效）`virsh blockresize --config centosvm vda 50G`

# 3. 启动虚拟机并扩展文件系统 `virsh start centosvm`# 登录后执行文件系统扩容命令

（同上）

#### 3. 收缩块设备（需谨慎）

收缩虚拟机 `dbvm` 的 `vdc` 盘（LVM 块设备）从 20G 到 15G：

bash

# 1. 在虚拟机内收缩文件系统（关键！否则数据丢失）`umount /dev/vdc` # 先卸载

`e2fsck -f /dev/vdc` # 检查文件系统

`resize2fs /dev/vdc 15G` # 收缩文件系统到 15G

# 2. 关闭虚拟机（块设备收缩通常需离线）`virsh shutdown dbvm`

# 3. 收缩底层 LVM 逻辑卷（宿主机操作）

```
lvreduce -L 15G /dev/vg1/db-disk
```

# 4. 调整虚拟机磁盘大小 `virsh blockresize --config dbvm vdc 15G`

# 5. 启动虚拟机并验证 `virsh start dbvm`

注意事项

扩容前备份：尤其是系统盘，建议通过 `virsh snapshot-create-as` 创建快照或备份磁盘文件。

文件系统适配：磁盘扩容后，需在虚拟机内手动扩展文件系统（如 `resize2fs`、`xfs_growfs`），否则操作系统无法识别新增空间。

qcow2 收缩技巧：若需收缩 qcow2 磁盘，可先转换为 raw 格式（`qemu-img convert -O raw old.qcow2 new.raw`），收缩后再转回 qcow2（`qemu-img convert -O qcow2 new.raw shrunked.qcow2`）。

通过 `virsh blockresize` 可灵活调整磁盘大小，但需严格遵循“先扩文件系统再扩磁盘（收缩则相反）”的原则，避免数据损坏。

#### 5.2.4.磁盘镜像合并

`virsh blockcommit` 是 KVM 虚拟化中用于合并磁盘快照链的命令，主要解决多层快照导致的性能下降问题，同时可以释放快照占用的存储空间。以下是详细说明：

核心作用

当虚拟机使用 qcow2 格式磁盘并创建多层快照时，会形成“基础镜像 + 多层快照”的链式结构（如 `base.qcow2` → `snap1.qcow2` → `snap2.qcow2`）。`blockcommit` 的作用是：将上层快照中的数据合并到下层镜像（通常是基础镜像），最终可以删除合并后的快照，简化存储结构并提升 I/O 性能。

基本语法

bash

```
virsh blockcommit <域名> <目标设备> [选项]
```

<域名>：虚拟机名称、ID 或 UUID

<目标设备>：虚拟机内的磁盘设备名（如 `vda`，需对应快照链中的磁盘）

## 主要选项

| 选项                     | 作用                         |
|------------------------|----------------------------|
| <code>--base</code>    | 指定基础镜像路径（快照链的最底层），合并时以此为目标 |
| <code>--top</code>     | 指定要合并的最上层快照路径，默认合并所有上层快照   |
| <code>--wait</code>    | 等待合并操作完成后再返回（默认后台执行）       |
| <code>--verbose</code> | 显示合并进度详情                   |
| <code>--live</code>    | 对运行中的虚拟机执行在线合并（不中断业务）      |
| <code>--shallow</code> | 仅合并最上层快照到其直接下层，不递归合并整个链    |

## 典型场景与操作步骤

假设虚拟机 `vm1` 的磁盘 `vda` 有如下快照链：`/data/base.qcow2`（基础镜像）

→ `/data/snap1.qcow2`（一级快照）→ `/data/snap2.qcow2`（二级快照，当前使用）

1. 在线合并所有快照到基础镜像（推荐）

将 `snap1` 和 `snap2` 的数据合并到 `base.qcow2`，完成后可删除快照：

bash

```
# 1. 查看当前磁盘快照链(确认结构)virsh domblklist --details vm1# 输出示例: vda    disk
/data/snap2.qcow2  qcow2  -
```

```
# 2. 在线合并所有快照到基础镜像 virsh blockcommit --live --wait --verbose vm1 vda
--base /data/base.qcow2
```

```
# 3. 合并完成后，删除已合并的快照（可选）rm -f /data/snap1.qcow2 /data/snap2.qcow2
```

2. 仅合并最上层快照（`shallow` 模式）

只将 `snap2` 合并到 `snap1`，保留中间快照层：

bash

```
virsh blockcommit --live --wait vm1 vda --base /data/snap1.qcow2 --top /data/snap2.qcow2
--shallow
```

## 关键注意事项

适用格式：仅支持 `qcow2` 格式（`raw` 不支持快照链，无需合并）。

性能影响：合并过程会消耗 I/O 资源，建议在业务低峰期执行；`--live` 选项可避免停机，但可能短暂影响虚拟机性能。

存储空间：合并前需确保基础镜像所在分区有足够空间（至少等于所有待合并快照的总大小）。



合并后验证：

检查虚拟机能否正常启动： `virsh start vm1`

确认数据完整性（如挂载磁盘检查文件）

查看新的磁盘链： `qemu-img info /data/base.qcow2`（应显示无上层快照）

总结

`virsh blockcommit` 是优化 `qcow2` 快照链的核心工具，通过合并多层快照可：

减少 I/O 路径长度，提升虚拟机磁盘性能

释放快照占用的存储空间

简化存储管理（避免快照链过长导致的维护复杂度）

使用时需注意空间预留和业务时间选择，优先采用 `--live` 在线合并以减少停机影响。

分享

如何查看虚拟机磁盘的快照信息？

合并磁盘快照可能会导致数据丢失吗？

除了 `virsh blockcommit` 命令，还有其它方法可以合并磁盘快照吗？

### 5.2.5.磁盘复制数据

`virsh blockcopy` 命令用于复制虚拟机磁盘，支持在线操作和增量复制，其选项涵盖了复制模式、性能控制、完成策略等细节。以下是所有选项的详细说明：

#### 1. `--async`

作用：异步执行复制操作（后台运行），命令立即返回，不等待复制完成。

适用场景：无需实时监控进度的脚本或批量操作，可通过 `virsh blockjob` 后续查看进度。

#### 2. `--bandwidth <速率>`

作用：限制复制时的带宽（单位：K/M/G，如 100M），避免占用过多存储或网络资源。

示例： `--bandwidth 500M`（限制最大速率为 500MB/s）

适用场景：业务高峰期执行复制，避免影响虚拟机性能。

#### 3. `--blockdev`

作用：使用块设备节点（而非文件路径）作为目标，适用于直接复制到物理块设备（如 `/dev/sdb`）。

注意：需确保目标块设备未被占用，且与源磁盘格式兼容。

#### 4. --buf-size <大小>

作用：设置复制时的缓冲区大小（单位：K/M，如 1M），影响复制效率。

说明：默认值通常足够，仅在特殊存储环境（如低延迟 SSD）中需要调整。

#### 5. --bytes <数量>

作用：指定仅复制前 N 字节的数据（用于部分复制测试，不常用）。

示例：--bytes 1G（仅复制前 1GB 数据）

#### 6. --dest <路径>

作用：指定复制的目标路径（文件或块设备），与命令中的“目标路径”参数功能相同（两者选其一即可）。

示例：--dest /backup/vm-disk.qcow2

#### 7. --domain <名称/ID/UUID>

作用：指定要复制磁盘的虚拟机（必填选项）。

示例：--domain web-vm

#### 8. --finish

作用：完成正在进行的后台复制任务，并解除原磁盘与虚拟机的关联（仅用于持久化虚拟机）。

适用场景：配合 --async 使用，在复制完成后手动确认并结束任务。

示例：virsh blockcopy --domain fnos vdc /home/vdd --finish

#### 9. --format <格式>

作用：指定目标磁盘的格式（如 qcow2、raw），默认与源磁盘格式一致。

示例：--format qcow2（强制目标为 qcow2 格式）

#### 10. --granularity <大小>

作用：设置增量复制时的块大小（单位：K/M，如 64K），影响增量检测精度。

说明：值越小，增量复制越精确（只复制变化的小块），但检测开销越高。

#### 11. --path <路径>

作用：与 --dest 相同，指定目标路径（兼容旧版本语法）。

#### 12. --pivot

作用：复制完成后，自动将虚拟机的磁盘切换到目标路径（替代原磁盘），无需手动 detach/attach。

适用场景：磁盘迁移（如从本地存储迁移到共享存储），简化操作流程。

### 13. --raw

作用：强制以 **raw** 格式复制（忽略源格式），适用于需要转换为 **raw** 格式的场景。

### 14. --reuse-external

作用：允许复用已存在的外部快照作为复制源（高级功能，用于复杂快照链复制）。

### 15. --shallow

作用：仅复制当前磁盘层，不包含底层快照数据（用于复制快照链中的某一层）。

### 16. --timeout <秒数>

作用：设置复制超时时间（单位：秒），超时后终止任务。

示例：--timeout 3600（1 小时后未完成则终止）

### 17. --transient-job

作用：标记复制任务为临时任务，虚拟机重启后自动取消（用于临时测试）。

### 18. --verbose

作用：显示复制进度、速率、剩余时间等详细信息。

适用场景：需要实时监控复制状态的操作。

### 19. --wait

作用：等待复制完成后再返回命令行（阻塞模式），与 --async 相反。

适用场景：交互式操作或需要确保复制完成后再执行后续步骤的脚本。

### 20. --xml <文件>

作用：从 **XML** 文件中读取复制配置（用于复杂场景，如自定义复制参数）。

常用组合示例

在线迁移磁盘并自动切换目标：

bash

```
virsh blockcopy --domain fnos vdc /home/vdd --live --pivot --wait --verbose --bandwidth
```

200M

（在线复制 **vdc** 到 **/home/vdd**，完成后自动切换虚拟机使用新磁盘，限速 200MB/s）

后台增量复制并限制缓冲区：

bash

```
virsh blockcopy --domain db-vm vdb /backup/incremental.qcow2 --async --incremental
```

--buf-size 2M

这些选项覆盖了从基础复制到高级迁移的各种场景，核心需根据是否在线操作（--live）、是否自动切换磁盘（--pivot）、性能控制（--bandwidth）来组合使用。

### 5.2.6.磁盘操作终止

virsh blockjob 命令用于管理虚拟机的后台磁盘操作任务（如 blockcopy、blockcommit 等后台运行的任务），可以查看进度、调整带宽、终止任务或完成切换。以下是各选项的详细说明：

#### 1. --abort

作用：强制终止正在进行的磁盘操作任务（如复制、合并）。

适用场景：任务执行过慢、占用资源过多，或需紧急停止操作时使用。

示例：

bash

```
# 终止虚拟机 fnos 的 vdc 磁盘正在进行的 blockcopy 任务 virsh blockjob --domain  
fnos --path vdc --abort
```

#### 2. --bandwidth <速率>

作用：动态调整磁盘操作任务的带宽限制（单位：K/M/G，如 50M）。

适用场景：任务执行中发现带宽占用过高影响业务，可临时调低；或业务低峰时调高以加速完成。

示例：

bash

```
# 将虚拟机 web-vm 的 vda 磁盘任务带宽调整为 100MB/s virsh blockjob --domain  
web-vm --path vda --bandwidth 100M
```

#### 3. --domain <名称/ID/UUID>

作用：指定要操作的虚拟机（必填选项），需提供虚拟机的名称、ID 或 UUID。

示例：--domain db-vm

#### 4. --path <设备名>

作用：指定虚拟机内的磁盘设备名（如 vda、vdc），需与正在执行任务的磁盘对应（必填选项）。

示例：--path vdb

#### 5. --raw

作用：以原始格式输出任务信息（无格式化，适合脚本解析）。

示例：

bash

```
# 以原始格式查看任务信息 virsh blockjob --domain fnos --path vdc --info --raw
```

#### 6. --async

作用：异步执行操作（如调整带宽），命令立即返回，不等待确认。

适用场景：脚本中批量调整任务参数，无需阻塞等待。

#### 7. --bytes <数量>

作用：限制任务处理的总数据量（单位：K/M/G，如 10G），达到后自动停止任务（较少使用）。

示例：

bash

```
# 限制任务仅处理前 5GB 数据 virsh blockjob --domain test-vm --path vda --bytes 5G
```

#### 8. --info

作用：显示指定磁盘任务的详细进度信息，包括总大小、已完成量、速率、剩余时间等。

适用场景：监控任务执行状态，判断是否需要调整带宽或终止。

示例：

bash

```
# 查看虚拟机 fnos 的 vdc 磁盘任务进度 virsh blockjob --domain fnos --path vdc --info
```

输出示例：

plaintext

Job type: Block Copy

Status: running

Progress: 5.0 GiB / 20.0 GiB (25%)

Bandwidth: 50.0 MiB/s

Estimated remaining time: 300 seconds

#### 9. --pivot

作用：完成磁盘操作任务后，将虚拟机的磁盘访问切换到目标路径（仅用于 blockcopy 任务）。

适用场景：磁盘迁移完成后，自动切换虚拟机使用新磁盘，替代原磁盘（无需手动 detach/attach）。

示例：

bash

```
# 完成复制后，将虚拟机 fnos 的磁盘切换到新路径 virsh blockjob --domain fnos --path vdc --pivot
```

常用场景组合

监控复制进度：

bash

```
virsh blockjob --domain fnos --path vdc --info
```

动态调整带宽：

bash

```
virsh blockjob --domain web-vm --path vda --bandwidth 200M
```

终止卡住的任务：

bash

```
virsh blockjob --domain db-vm --path vdb --abort
```

迁移完成后切换磁盘：

bash

```
virsh blockjob --domain fnos --path vdc --pivot
```

virsh blockjob 是管理后台磁盘任务的核心工具，通过这些选项可实时监控、调整或终止任务，确保磁盘操作（复制、合并等）按需执行。

### 5.2.7. 磁盘镜像拉取

virsh blockpull 命令用于将虚拟机磁盘的外部数据（如快照、后端镜像）合并到当前磁盘，主要解决多层快照或镜像链导致的性能问题，同时可以将依赖外部镜像的数据“拉取”到当前磁盘，使其成为独立完整的镜像。

## 核心作用

当虚拟机使用 **qcow2** 格式磁盘且依赖外部镜像（如基于基础镜像创建的快照、或使用 **backing\_file** 配置的链式镜像）时，**blockpull** 会将所有外部依赖的数据合并到当前磁盘中，最终使当前磁盘成为一个独立完整的镜像，不再依赖任何外部文件。

例如：**base.qcow2**（基础镜像）← **snap1.qcow2**（快照，当前使用）执行 **blockpull** 后，**snap1.qcow2** 会包含 **base.qcow2** 的所有数据，成为独立镜像，可删除 **base.qcow2**。

## 基本语法

**bash**

**virsh blockpull <域名> <目标设备> [选项]**

**<域名>**：虚拟机名称、ID 或 UUID

**<目标设备>**：虚拟机内的磁盘设备名（如 **vda**，需对应链式镜像中的当前磁盘）

## 主要选项

| 选项                            | 作用                                          |
|-------------------------------|---------------------------------------------|
| <b>--bandwidth &lt;速率&gt;</b> | 限制合并时的带宽（如 <b>100M</b> ），避免占用过多资源           |
| <b>--wait</b>                 | 等待合并操作完成后返回（默认后台执行）                         |
| <b>--verbose</b>              | 显示合并进度详情                                    |
| <b>--live</b>                 | 对运行中的虚拟机执行在线合并（不中断业务）                       |
| <b>--depth &lt;层数&gt;</b>     | 仅合并指定层数的外部镜像（如 <b>--depth 1</b> 只合并直接依赖的一层） |
| <b>--copy-all</b>             | 强制复制所有数据（包括未分配的空间，确保完全独立）                   |

## 典型使用场景

### 1. 在线合并快照链，使当前磁盘独立

假设虚拟机 **vm1** 的 **vda** 磁盘基于 **base.qcow2** 创建，且有多层快照，执行以下命令将所有依赖合并到当前磁盘：

**bash**

**# 在线合并所有外部依赖，使 vda 成为独立镜像** **virsh blockpull --live --wait --verbose vm1**

**vda**

**# 合并完成后，验证当前磁盘是否独立**

**qemu-img info /path/to/current.vda#** 输出中“**backing file**”字段应为空，说明已无外部依

赖

## 2. 限制合并带宽，避免影响业务

在业务高峰期合并时，通过 `--bandwidth` 限制速率：

bash

```
virsh blockpull --domain web-vm --path vdb --live --bandwidth 50M --wait
```

## 3. 仅合并指定层数的依赖

若只想合并直接依赖的基础镜像，保留上层快照：

bash

```
virsh blockpull --domain db-vm --path vdc --depth 1 --wait
```

### 关键注意事项

适用格式：仅支持 `qcow2` 格式（`raw` 不支持链式镜像，无需合并）。

空间要求：合并前需确保当前磁盘所在分区有足够空间（至少等于所有依赖镜像的总大小）。

性能影响：`--live` 在线合并会消耗 I/O 资源，建议在低峰期执行，或通过 `--bandwidth` 限制速率。

合并后效果：当前磁盘会包含所有依赖数据，可安全删除原基础镜像或快照（需先确认合并成功）。

与 `blockcommit` 的区别：

**blockpull**：将外部数据合并到当前磁盘（上层 → 当前），最终当前磁盘独立。

**blockcommit**：将上层数据合并到底层基础镜像（上层 → 底层），最终基础镜像包含所有数据。

### 总结

`virsh blockpull` 是处理 `qcow2` 链式镜像的重要工具，核心价值在于将依赖外部数据的磁盘转换为独立镜像，简化存储管理并提升 I/O 性能（减少多层镜像的访问开销）。使用时需注意空间预留和业务时间选择，优先采用在线合并（`--live`）减少停机影响。



### 5.2.8.磁盘空间阈值告警

`virsh domblkthreshold` 命令用于为虚拟机的磁盘设备设置空间使用阈值警报，当磁盘使用量达到或超过设定阈值时，会触发 `libvirt` 的事件通知，帮助管理员及时及时监控磁盘空间使用情况，避免磁盘满导致的业务中断。

#### 核心作用

为虚拟机的指定磁盘设置空间使用阈值（如使用率达到 80% 时报警）。

当磁盘使用量超过阈值时，`libvirt` 会生成事件日志（可通过 `virsh event` 监控），便于及时处理。

支持动态调整阈值，无需重启虚拟机。

#### 基本语法

bash

`virsh domblkthreshold <域名> <目标设备> <阈值> [选项]`

<域名>: 虚拟机名称、ID 或 UUID（必填）。

<目标设备>: 虚拟机内的磁盘设备名（如 `vda`、`vdb`，需与 `attach-disk` 时的 `--target` 一致，必填）。

<阈值>: 触发警报的空间使用阈值，支持两种格式：

百分比：如 80（表示使用率达到 80% 时报警）。

绝对大小：如 10G（表示已使用空间达到 10GB 时报警）。

#### 主要选项

| 选项                     | 作用                                           |
|------------------------|----------------------------------------------|
| <code>--config</code>  | 仅修改配置文件，下次启动时生效（默认对当前运行状态生效）。                |
| <code>--current</code> | 根据虚拟机当前状态（运行中 / 关闭）应用配置（运行中则立即生效，关闭则下次启动生效）。 |
| <code>--delete</code>  | 删除已设置的阈值（取消警报配置）。                            |

#### 典型使用场景

1. 为运行中的虚拟机磁盘设置百分比阈值

为虚拟机 `web-vm` 的 `vda` 磁盘设置 85% 使用率的警报：

bash

# 当 `vda` 磁盘使用率达到 85%时触发警报 `virsh domblkthreshold web-vm vda 85`

## 2. 为磁盘设置绝对大小阈值

为虚拟机 db-vm 的 vdb 磁盘设置 “使用达到 50GB 时报警”：

```
bash
```

```
virsh domblkthreshold db-vm vdb 50G
```

## 3. 动态调整阈值

将已设置的阈值从 85% 调整为 90%：

```
bash
```

```
virsh domblkthreshold web-vm vda 90
```

## 4. 删除阈值配置

取消 db-vm 的 vdb 磁盘阈值警报：

```
bash
```

```
virsh domblkthreshold db-vm vdb --delete
```

## 5. 监控阈值事件

设置阈值后，可通过 `virsh event` 实时监控警报事件：

```
bash
```

```
# 监控所有磁盘阈值事件 virsh event --domain web-vm --block-threshold --loop
```

当达到阈值时，会输出类似以下的事件日志：

```
plaintext
```

```
event 'block-threshold' for domain 'web-vm'
```

```
device: vda
```

```
threshold: 85%
```

```
current usage: 86%
```

### 关键注意事项

适用磁盘类型：支持文件镜像（`qcow2/raw`）和块设备（`LVM / 物理分区`），但需磁盘本身支持空间大小查询（大多数现代存储均支持）。

阈值精度：百分比阈值基于磁盘总容量计算，绝对大小阈值基于已使用空间计算，实际触发可能有轻微延迟（取决于 `libvirt` 检查频率）。

事件处理：`domblkthreshold` 仅负责触发事件，需结合外部工具（如脚本、监控系统）处理警报（如自动扩容、发送邮件通知）。

持久化配置：默认设置仅对当前运行状态生效，虚拟机重启后失效；若需永久生效，需添加 `--config` 选项。

总结

`virsh domblkthreshold` 是虚拟机磁盘空间监控的辅助工具，通过设置阈值警报，可在磁盘空间不足前及时发现并处理，降低因磁盘满导致的业务中断风险。实际使用中需结合事件监控脚本，实现自动化告警和处理。

5.2.9.磁盘 I/O 调优

`virsh blkdeviotune` 命令用于动态调整虚拟机磁盘设备的 I/O 性能参数，通过限制或调整磁盘的读写速率、I/O 队列长度等，实现对虚拟机磁盘 I/O 的精细化控制，避免单个虚拟机过度占用存储资源，影响其他虚拟机性能。

核心作用

限制虚拟机磁盘的最大读写速率（如限制某虚拟机的磁盘写入不超过 100MB/s）。

调整磁盘的 I/O 队列长度（控制并发请求数，避免队列过长导致延迟）。

动态生效（无需重启虚拟机），可根据业务负载实时调整。

基本语法

bash

`virsh blkdeviotune <域名> <目标设备> [参数选项]`

<域名>：虚拟机名称、ID 或 UUID（必填）。

<目标设备>：虚拟机内的磁盘设备名（如 `vda`、`vdb`，需与 `attach-disk` 时的 `--target` 一致，必填）。

主要参数选项

| 选项                                        | 作用                 | 取值范围                   |
|-------------------------------------------|--------------------|------------------------|
| <code>--total-bytes-sec &lt;速率&gt;</code> | 限制磁盘的总读写速率（每秒总字节数） | 0（无限制）或正整数（单位：B/K/M/G） |
| <code>--read-bytes-sec &lt;速率&gt;</code>  | 限制磁盘的读取速率（每秒读取字节数） | 同上                     |
| <code>--write-bytes-sec &lt;速率&gt;</code> | 限制磁盘的写入速率（每秒写入字节数） | 同上                     |

| 选项                                       | 作用                          | 取值范围                      |
|------------------------------------------|-----------------------------|---------------------------|
| <code>--total-iops-sec &lt;次数&gt;</code> | 限制磁盘的总 I/O 次数（每秒总 I/O 操作数）  | 0（无限制）或正整数                |
| <code>--read-iops-sec &lt;次数&gt;</code>  | 限制磁盘的读取 I/O 次数（每秒读取操作数）     | 同上                        |
| <code>--write-iops-sec &lt;次数&gt;</code> | 限制磁盘的写入 I/O 次数（每秒写入操作数）     | 同上                        |
| <code>--iops-size &lt;大小&gt;</code>      | 定义单次 I/O 操作的大小（用于计算 I/O 次数） | 正整数（单位：B/K/M，默认 512B）     |
| <code>--queue-depth &lt;深度&gt;</code>    | 限制磁盘的 I/O 队列最大长度（并发请求数）     | 0（默认，由系统决定）或正整数（通常 1-128） |
| <code>--reset</code>                     | 重置所有 I/O 参数为默认值（取消所有限制）     | 无额外参数                     |

#### 典型使用场景

##### 1. 限制虚拟机磁盘的最大写入速率

避免某虚拟机写入过快占用过多存储带宽：

```
bash
```

```
# 限制虚拟机 db-vm 的 vda 磁盘写入速率不超过 200MB/svirsh blkdeviotune db-vm
```

```
vda --write-bytes-sec 200M
```

##### 2. 同时限制读写速率和 I/O 次数

对低优先级虚拟机进行严格限制：

```
bash
```

```
# 限制 test-vm 的 vdb 磁盘：读速≤100M，写速≤50M，总 I/O 次数≤1000 次/秒 virsh
```

```
blkdeviotune test-vm vdb --read-bytes-sec 100M --write-bytes-sec 50M --total-iops-sec 1000
```

##### 3. 调整 I/O 队列深度（优化延迟）

对于数据库等对延迟敏感的业务，减少队列深度避免请求堆积：

```
bash
```

```
# 将 web-vm 的 vda 磁盘队列深度限制为 32（默认可能更大）virsh blkdeviotune
```

```
web-vm vda --queue-depth 32
```

##### 4. 查看当前 I/O 配置

不带参数执行，可查看指定磁盘的当前 I/O 参数：

```
bash
```

```
virsh blkdeviotune db-vm vda
```

输出示例：

```
plaintext
```

```
total_bytes_sec: 0
```

```
read_bytes_sec: 0
```

```
write_bytes_sec: 209715200 # 200MB/s (200*1024*1024)
```

```
total_iops_sec: 0
```

```
read_iops_sec: 0
```

```
write_iops_sec: 0
```

```
iops_size: 512
```

```
queue_depth: 0
```

## 5. 重置所有限制

取消对磁盘的所有 I/O 限制：

```
bash
```

```
virsh blkdeviotune test-vm vdb --reset
```

### 关键注意事项

动态生效：所有参数调整立即对运行中的虚拟机生效，无需重启。

优先级：若同时设置 `--total-bytes-sec` 和 `--read/write-bytes-sec`，以更严格的限制为准（如总限制 100M，读写分别设 80M 和 30M，则实际总限制为 100M）。

适用场景：多用于共享存储环境（如多虚拟机共用一块磁盘或存储阵列），防止资源抢占；对单机独占存储的虚拟机，通常无需限制。

性能影响：过度限制可能导致虚拟机 I/O 性能下降（如数据库写入延迟增加），需根据业务需求合理设置阈值。

### 总结

`virsh blkdeviotune` 是虚拟化环境中实现磁盘 I/O 资源隔离的核心工具，通过精细控制读写速率、I/O 次数和队列深度，可确保多虚拟机环境下的存储资源公平分配，避免单虚拟机过度占用资源影响整体稳定性。实际使用中需结合业务负载测试，找到性能与资源分配的平衡点。

5.2.10.磁盘 I/O 限制

virsh blkiotune 命令用于配置虚拟机的整体磁盘 I/O 资源配额，通过设置权重、限制带宽和 I/O 次数等参数，实现多个虚拟机之间的磁盘 I/O 资源分配与隔离，避免单个虚拟机过度占用存储资源，保障整体系统的 I/O 稳定性。

核心作用

与 blkdeviotune（针对单个磁盘设备）不同，blkiotune 主要从虚拟机全局角度控制磁盘 I/O 资源：

为虚拟机分配 I/O 调度权重（权重），决定多个虚拟机竞争存储资源时的优先级。

限制虚拟机的最大磁盘读写带宽和 I/O 操作次数（全局限制，适用于所有磁盘）。

常用于共享存储环境（如多个虚拟机共用同一存储阵列），实现资源公平分配。

基本语法

bash

virsh blkiotune <域名> [参数选项]

<域名>：虚拟机名称、ID 或 UUID（必填）。

主要参数选项

| 选项                       | 作用                                              | 取值范围                   |
|--------------------------|-------------------------------------------------|------------------------|
|                          | 设置虚拟机的 I/O 权重（相对优先级），值越高，竞争资源时获得的 I/O 越多        | 1-1000（默认 500）         |
| --weight <值>             |                                                 |                        |
|                          | 为特定磁盘设备设置单独权重（覆盖全局--weight），格式如 vda:800,vdb:300 | 设备名：1-1000             |
| --device-weights <设备:权重> |                                                 |                        |
| --total-bytes-sec <速率>   | 限制虚拟机所有磁盘的总读写速率（每秒总字节数）                         | 0（无限制）或正整数（单位：B/K/M/G） |
| --read-bytes-sec <速率>    | 限制所有磁盘的总读取速率                                    | 同上                     |
| --write-bytes-sec <速率>   | 限制所有磁盘的总写入速率                                    | 同上                     |
| --total-iops-sec <次数>    | 限制所有磁盘的总 I/O 操作次数 /                             | 0（无限制）或正整              |

| 选项                  | 作用                           | 取值范围  |
|---------------------|------------------------------|-------|
| 数>                  | 秒                            | 数     |
| --read-iops-sec <次  | 限制所有磁盘的总读取 I/O 次数 /          | 同上    |
| 数>                  | 秒                            |       |
| --write-iops-sec <次 | 限制所有磁盘的总写入 I/O 次数 /          | 同上    |
| 数>                  | 秒                            |       |
| --reset             | 重置所有 I/O 参数为默认值（取消所有限制和权重设置） | 无额外参数 |

#### 典型使用场景

##### 1. 为核心虚拟机提高 I/O 优先级

确保核心业务虚拟机在存储资源竞争时获得更多 I/O:

```
bash
```

```
# 为数据库虚拟机 db-vm 设置高权重（800），高于默认的 500virsh blkiotune db-vm
```

```
--weight 800
```

##### 2. 为特定磁盘设置单独权重

同一虚拟机的不同磁盘需要不同优先级（如系统盘优先于数据盘）:

```
bash
```

```
# 为 web-vm 的 vda（系统盘）设高权重，vdb（日志盘）设低权重 virsh blkiotune web-vm
```

```
--device-weights vda:700,vdb:300
```

##### 3. 限制低优先级虚拟机的总 I/O 带宽

防止测试或非核心虚拟机占用过多存储资源:

```
bash
```

```
# 限制 test-vm 的总读写速率≤100MB/s，总 I/O 次数≤500 次/秒 virsh blkiotune test-vm
```

```
--total-bytes-sec 100M --total-iops-sec 500
```

##### 4. 查看当前 I/O 配置

不带参数执行，可查看虚拟机的当前 I/O 参数:

```
bash
```

```
virsh blkiotune db-vm
```

输出示例:

```
plaintext
```

```
weight: 800

device_weights:

total_bytes_sec: 0

read_bytes_sec: 0

write_bytes_sec: 0

total_iops_sec: 0

read_iops_sec: 0

write_iops_sec: 0
```

5. 重置所有 I/O 配置

取消对虚拟机的所有 I/O 限制和权重设置：

```
bash

virsh blkiotune test-vm --reset
```

与 blkdeviotune 的区别

| 维度   | blkiotune       | blkdeviotune           |
|------|-----------------|------------------------|
| 作用范围 | 虚拟机全局（所有磁盘）     | 单个磁盘设备（如 vda、vdb）      |
| 核心功能 | 侧重资源分配（权重）和全局限制 | 侧重单个磁盘的精细化速率 / IOPS 限制 |
| 适用场景 | 多虚拟机资源竞争时的优先级控制 | 单虚拟机内不同磁盘的差异化限制        |

关键注意事项

动态生效：所有参数调整立即对运行中的虚拟机生效，无需重启。

权重的意义：权重仅在存储资源紧张（有竞争）时生效，资源充足时即使权重低也能达到满速。

与存储类型匹配：对本地磁盘，限制效果可能不明显；对共享存储（如 NFS、SAN），资源隔离效果更显著。

组合使用：可同时使用 blkiotune（全局控制）和 blkdeviotune（单盘控制），实现多层次 I/O 管理。

总结



`virsh blkiotune` 是共享存储环境中实现虚拟机 I/O 资源分配的关键工具，通过权重设置和全局限制，可确保核心业务获得足够的存储资源，同时防止低优先级虚拟机过度抢占资源。实际使用中需根据业务重要性合理分配权重，并结合单盘限制（`blkdeviotune`）实现更精细的控制。

### 5.2.11.更改 CD 介质

`virsh change-media` 命令用于动态更改虚拟机的 CD/DVD 驱动器 或软盘驱动器的介质（如加载 / 卸载 ISO 镜像文件），无需重启虚拟机，是管理虚拟机存储介质的常用工具。

基本语法

bash

`virsh change-media <虚拟机名称/UUID> <设备目标> <介质路径|--eject> [选项]`

核心参数说明

`<虚拟机名称/UUID>`必需参数，指定目标虚拟机（通过名称或 UUID 标识）。

`<设备目标>`指定要操作的存储设备，通常是：

CD/DVD 驱动器：`hdc`、`sr0`（常见默认值，可通过 `virsh domblklist` 查看）。

软盘驱动器：`fda`、`fdb`（较少使用）。

查看虚拟机的存储设备列表：

bash

`virsh domblklist <虚拟机名称> #` 列出所有块设备，包括 CD/DVD 驱动器

`<介质路径|--eject>`

加载新介质：指定 ISO 镜像文件的绝对路径（如 `/data/ubuntu.iso`）。

卸载介质：使用 `--eject` 参数弹出当前介质。

常用选项

`--insert`：明确指定加载介质（默认行为，可省略）。

`--force`：强制操作（如介质正在使用时仍卸载）。

`--live`：对运行中的虚拟机立即生效（默认）。

`--config`：仅修改配置，下次启动生效。

使用示例

### 1. 查看虚拟机的 CD/DVD 设备

```
bash
```

```
virsh domblklist ubuntuvm
```

输出示例（sr0 为 CD/DVD 驱动器）：

```
plaintext
```

```
Target    Source
```

```
-----
```

```
vda        /var/lib/libvirt/images/ubuntuvm.qcow2
```

```
sr0        - # 未加载介质
```

### 2. 加载 ISO 镜像到 CD/DVD 驱动器

```
bash
```

```
# 向 ubuntuvm 的 sr0 设备加载 ubuntu-22.04.isovirsh change-media ubuntuvm sr0
```

```
/data/iso/ubuntu-22.04.iso
```

### 3. 卸载（弹出）当前 CD/DVD 介质

```
bash
```

```
# 弹出 ubuntuvm 的 sr0 设备中的介质 virsh change-media ubuntuvm sr0 --eject
```

### 4. 强制卸载正在使用的介质

```
bash
```

```
# 即使介质被虚拟机占用，仍强制弹出 virsh change-media ubuntuvm sr0 --eject --force
```

注意事项

虚拟机需处于运行状态（**running**）才能动态更改介质；若虚拟机已关闭，需修改 XML 配置或启动后操作。

确保 ISO 镜像文件路径对 **libvirt** 服务可见（权限正确，通常需放在 **/var/lib/libvirt/images/** 或其他共享目录）。

若操作失败，可能是设备被虚拟机锁定（如操作系统正在访问光盘），可尝试 **--force** 参数或在虚拟机内手动推出后再操作。

通过 **change-media** 命令，可灵活管理虚拟机的光学介质，适合安装系统、加载驱动等场景。

5.3.控制器（总线）设备

| 内容     | 命令                               | 说明         |
|--------|----------------------------------|------------|
| 添加总线设备 | <code>virsh attach-device</code> | 为虚拟机添加总线设备 |
| 分离总线设备 | <code>virsh detach-device</code> | 为虚拟机分离总线设备 |

总线是计算机各个部件之间进行信息通信的通道。外部设备需要挂载到对应的总线上，每个设备都会被分配一个唯一地址（由子元素 `address` 指定），通过总线网络完成与其他设备或中央处理器的信息交换。

常见的设备总线有 PCE/PCI 总线、USB 总线、SCSI 总线、ISA 总线。

PCIe 总线是一种典型的树结构，具有比较好的扩展性，总线之间通过控制器关联，这里以 PCIe 总线为例介绍如何为虚拟机配置总线拓扑。

在 KVM 虚拟化中，总线（Bus）是虚拟机内部连接设备（如磁盘、网卡、USB 设备等）的通信通道，负责设备与虚拟机 CPU、内存之间的数据传输。

KVM 对总线的管理主要通过 QEMU 设备模拟和 libvirt 配置抽象实现，核心机制包括：总线模拟 QEMU 作为 KVM 的用户态组件，负责模拟各类总线控制器（如 PCI 控制器、IDE 控制器、USB 控制器等）和总线协议，使虚拟机内的操作系统能像访问物理总线一样与虚拟设备通信。

设备附着与分离通过 `virsh attach-device/detach-device` 命令或修改 XML 配置，可动态为虚拟机添加 / 移除总线设备（如新增一块 virtio 总线的磁盘），无需重启虚拟机（部分总线支持热插拔）。

资源分配总线控制器会为连接的设备分配虚拟地址（如 PCI 设备的域 / 总线 / 槽位），避免地址冲突，libvirt 会自动管理这些地址（也可手动在 XML 中指定 `<address>` 标签）。

在 libvirt 的 XML 配置中，每个控制器元素（使用 `controller` 元素表示）可以表示一个总线，根据虚拟机架构的不同，一个控制器上通常可以挂载一个或多个控制器或设备。这里介绍常用属性和子元素。

`controller`：控制器元素，表示一个总线。

属性 `type`：控制器必选属性，表示总线类型。常用取值有“`pci`”、“`usb`”、“`scsi`”、“`virtio-serial`”、“`fdc`”、“`ccid`”。

属性 **index**: 控制器必选属性, 表示控制器的总线 “bus” 编号 (编号从 0 开始), 可以在地址元素 “address” 元素中使用。

属性 **model**: 控制器必选属性, 表示控制器的具体型号, 其可选择的值与控制器类型 “type” 的值相关, 对应关系及含义请参见下表。

子元素 **address**: 为设备或控制器指定其在总线网络中的挂载位置。

属性 **type**: 设备地址类型。常用取值有 “pci”、“usb”、“drive”。address 的 type 类型不同, 对应的属性也不同, 常用 type 属性值及其该取值下 address 的属性请参见表 5。

子元素 **model**: 控制器具体型号的名称。

属性 **name**: 指定控制器具体型号的名称, 和父元素 controller 中的属性 model 对应。

Controller 属性 type 常用取值和 model 取值对应关系

| type 属性值      | model 属性值          | 简介                                    |
|---------------|--------------------|---------------------------------------|
| pci           | pcie-root          | PCIe 根节点, 可挂载 PCIe 设备或控制器             |
|               | pcie-root-port     | 只有一个 slot, 可以挂载 PCIe 设备或控制器           |
|               | pcie-to-pci-bridge | PCIe 转 PCI 桥控制器, 可挂载 PCI 设备           |
| usb           | ehci               | USB 2.0 控制器, 可挂载 USB 2.0 设备           |
|               | nec-xhci           | USB 3.0 控制器, 可挂载 USB 3.0 设备           |
| scsi          | virtio-scsi        | virtio 类型 SCSI 控制器, 可以挂载块设备, 如磁盘, 光盘等 |
| virtio-serial | virtio-serial      | virtio 类型串口控制器, 可挂载串口设备, 如 pty 串口     |

address 元素不同设备类型下的属性说明

| 类型 type 属性值 | 含义                                   | 对应地址属性                                                                                                                                                     |
|-------------|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| pci         | 地址类型为 PCI 地址, 表示该设备在 PCI 总线网络中的挂载位置。 | domain: PCI 设备的域号<br>bus: PCI 设备的 bus 号<br>slot: PCI 设备的 device 号<br>function: PCI 设备的 function 号<br>multifunction: controller 元素可选, 是否开启 multifunction 功能 |
| usb         | 地址类型为                                | bus: USB 设备的 bus 号                                                                                                                                         |

| 类型 type 属性值 | 含义                                 | 对应地址属性                                                                                    |
|-------------|------------------------------------|-------------------------------------------------------------------------------------------|
|             | USB 地址, 表示该设备在 USB 总线中的位置。         | port: USB 设备的 port 号                                                                      |
| drive       | 地址类型存储设备地址, 表示所属的磁盘控制器, 及其在总线中的位置。 | controller: 指定所属控制器号<br>bus: 设备输出的 channel 号<br>target: 存储设备 target 号<br>unit: 存储设备 lun 号 |

### 5.3.1.PCI/PCIe 总线

全称: Peripheral Component Interconnect (外设组件互连) / PCI Express

作用: 虚拟机内部的核心总线, 所有高级设备 (如 virtio 磁盘、虚拟网卡、显卡) 最终都通过 PCI/PCIe 总线连接到 CPU, 是其他总线 (如 virtio、SCSI) 的“父总线”。

特点:

通用性强: 几乎所有虚拟设备 (除 IDE 等老旧设备) 都需通过 PCI/PCIe 地址空间通信。

支持热插拔 (PCIe 设备) 和设备直通 (PCI passthrough, 将物理 PCI 设备直接分配给虚拟机)。

PCIe 总线在 KVM 中采用“树形结构”模拟物理机的 PCIe 架构, pcie-root、pcie-root-port 和 pcie-root-bridge 是模拟 PCIe 总线架构的核心组件, 用于构建虚拟机内部的 PCIe 设备连接拓扑。

#### 1. pcie-root (PCIe 根联合体)

含义: 模拟物理机的 PCIe 根联合体 (Root Complex), 是整个 PCIe 总线的“根节点”, 直接连接虚拟机的虚拟 CPU 和内存, 是所有 PCIe 设备的最终父节点。

作用: 作为 PCIe 总线的起点, 负责管理所有下游 PCIe 设备 (如根端口、桥接器、网卡、磁盘控制器等) 的地址分配和数据路由。

配置特点: 一个虚拟机通常只需要一个 pcie-root (默认自动创建, 也可在 XML 中显式定义)。

XML 配置示例：

```
<controller type='pci' index='0' model='pci-root'>
```

## 2. pcie-root-port（PCIe 根端口）

含义：模拟物理机的 PCIe 根端口（Root Port），是 pcie-root 的直接下游设备，相当于“物理插槽的接口”，用于连接具体的 PCIe 设备（如 virtio 网卡、磁盘控制器）或下级桥接器。

作用：每个 pcie-root-port 可视为一个“虚拟插槽”，为连接的设备分配独立的 PCIe 地址空间，支持热插拔（符合 PCIe 热插拔规范）。

配置特点：一个 pcie-root 可以挂接多个 pcie-root-port（数量取决于模拟的根联合体支持的端口数），需显式指定与 pcie-root 的连接关系。

XML 配置示例（挂接在 pcie-root 下）：

```
<!-- 根端口 1，索引为 1，连接到 index=0 的 pcie-root -->
```

```
<controller type="pci" index="1" model="pcie-root-port">
```

```
<model name="pcie-root-port"/>
```

```
<target chassis="1" port="0x10"/> <!-- chassis 为插槽编号，port 为端口号 -->
```

```
<address type="pci" domain="0x0000" bus="0x00" slot="0x01" function="0x0"/> <!--
```

挂接在 pcie-root（bus=0x00）上 -->

```
</controller>
```

## 3. pcie-root-bridge（PCIe 根桥）

含义：模拟传统 PCI 根桥（Root Bridge），是兼容传统 PCI 设备的过渡组件（PCIe 向下兼容 PCI），通常用于连接不支持 PCIe 的老旧设备。

作用：提供一个独立的 PCI 总线域（与 PCIe 总线分离），支持传统 PCI 设备的地址分配和中断路由，功能类似 pcie-root 但仅针对 PCI 设备。

配置特点：现代虚拟化环境中较少使用，除非需要运行依赖传统 PCI 设备的老旧系统。

XML 配置示例：

```
<controller type="pci" index="2" model="pcie-root-bridge"/>
```

address 地址配置规则

PCIe 设备的 address 用于在虚拟机内唯一标识设备位置，格式遵循 PCIe 地址规范，由

4 个字段组成：

```
<address type="pci" domain="0x0000" bus="0x00" slot="0x01" function="0x0"/>
```

domain: PCI 域（仅在多域系统中使用，KVM 中通常固定为 0x0000）。

bus: 总线号（0x00 通常为 pcie-root 所在总线，其他总线号由控制器动态分配）。

slot: 设备在总线上的槽位号（0x00-0x1f，对应物理机的插槽编号）。

function: 功能号（用于多功能设备，默认 0x0）。

不同组件的地址配置示例：

pcie-root 的地址：作为根节点，通常占用 bus=0x00, slot=0x00（默认无需显式配置，由 KVM 自动分配）。

pcie-root-port 的地址：挂接在 pcie-root（bus=0x00）上，槽位号从 0x01 开始递增：

```
<address type="pci" domain="0x0000" bus="0x00" slot="0x01" function="0x0"/>
```

```
<!-- 第一个根端口 -->
```

```
<address type="pci" domain="0x0000" bus="0x00" slot="0x02" function="0x0"/>
```

```
<!-- 第二个根端口 -->
```

PCIe 设备（如 virtio 网卡）的地址：挂接在 pcie-root-port 上，总线号为根端口分配的总线（如 bus=0x01）：

```
<interface type="bridge">
```

```
  <model type="virtio"/>
```

```
  <address type="pci" domain="0x0000" bus="0x01" slot="0x00" function="0x0"/> <!--
```

```
挂接在 index=1 的根端口 --></interface>
```

PCI 控制器与“插槽”的关系

一个 PCI 控制器（如 pcie-root-port）可理解为一个“虚拟插槽”，但需注意：

pcie-root 不是插槽：它是根节点，本身不直接连接设备，而是通过下游的 pcie-root-port 连接设备。

pcie-root-port 对应虚拟插槽：每个 pcie-root-port 相当于一个独立的 PCIe 插槽，可插入一个 PCIe 设备（如网卡、磁盘控制器），支持热插拔（类似物理机更换插槽中的设备）。

数量限制

物理机的 PCIe 插槽数量有限，KVM 中 pcie-root-port 的数量也受模拟的根联合体性能限制（通常可配置 32 个以上，满足绝大多数场景）

Root Port 的 chassis 和 port 属性必须依次递增，由于中间插入一个 PCIe-PCI-Bridge, chassis 编号跳过 2，但是 port 编号仍然连续。

Root Port 的 address function 的取值范围为 0x0~0x7

每个 slot 最多下挂 8 个 function，挂满之后需要递增 slot 编号。

总结

pcie-root 是 PCIe 总线的根节点，管理所有下游设备；

pcie-root-port 是根节点下的“虚拟插槽”，用于连接具体 PCIe 设备，支持热插拔；

pcie-root-bridge 是兼容传统 PCI 设备的根桥，现代环境较少使用；

address 由 domain:bus:slot:function 组成，用于唯一标识设备位置；

一个 pcie-root-port 对应一个虚拟插槽，是设备连接的基本单元。

通过合理配置这些组件，可构建灵活的 PCIe 拓扑，满足虚拟机对高性能设备（如 virtio 设备、PCI 直通设备）的需求。

配置示例（XML）：

```
<!-- 定义 PCIe 控制器 -->
```

```
<controller type="pci" index="0" model="pcie-root"/>
```

```
<controller type="pci" index="1" model="pcie-root-port"/>
```

```
<!-- 虚拟网卡通过 PCIe 总线连接 -->
```

```
<interface type="bridge">
```

```
  <source bridge="br0"/>
```

```
  <model type="virtio"/>
```

```
  <address type="pci" domain="0x0000" bus="0x01" slot="0x00" function="0x0"/>
```

```
<!-- 绑定到 PCIe 总线 -->
```

```
</interface>
```

适用场景：所有需要连接高性能设备的虚拟机，是现代虚拟化中设备互连的基础。

### 5.3.2.IDE 总线

全称：Integrated Drive Electronics（集成驱动电子设备）

作用：早期主流的磁盘 / 光驱连接总线，KVM 模拟 IDE 控制器支持连接虚拟硬盘和光驱。

特点：

兼容性极强，支持几乎所有操作系统（包括老旧系统如 Windows XP、Linux 2.4 内核）。



性能较差：单控制器最多连接 2 个设备（主从设备），传输速率低（理论最大 133MB/s），且 I/O 延迟高。

不支持热插拔（多数情况下需重启虚拟机才能识别新设备）。

配置示例（XML）：

```
<disk type="file" device="disk">
  <driver name="qemu" type="qcow2"/>
  <source file="/data/disk1.qcow2"/>
  <target dev="hda" bus="ide"/>
  <!-- IDE 总线，设备名为 hda（主设备） -->
  <address type="drive" controller="0" bus="0" target="0" unit="0"/>
</disk>
```

适用场景：运行老旧操作系统的虚拟机，或需要兼容传统 IDE 设备的场景。

### 5.3.3.SCSI 总线

全称：Small Computer System Interface（小型计算机系统接口）

作用：最初为服务器设计的高速总线，支持连接磁盘、磁带机等存储设备，KVM 模拟 SCSI 控制器提供更灵活的设备管理。

特点：

兼容性较好，支持大多数现代操作系统（Windows 7+、Linux 2.6+）。

性能优于 IDE：单控制器可连接最多 15 个设备，支持更高的并发 I/O，传输速率更高（理论最大 640MB/s）。

部分控制器（如 virtio-scsi）支持热插拔和高级功能（如 SCSI 命令透传）。

配置示例（XML）：

```
<disk type="block" device="disk">
  <driver name="qemu" type="raw"/>
  <source dev="/dev/sdb"/>
  <target dev="sda" bus="scsi"/>  <!-- SCSI 总线，设备名为 sda -->
  <address type="drive" controller="0" bus="0" target="0" unit="0"/>
</disk>
```

```
<!-- SCSI 控制器配置 -->
```

```
<controller type="scsi" index="0" model="virtio-scsi"/>
```

```
<!-- 使用 virtio-scsi 控制器提升性能 -->
```

适用场景：需要连接多个存储设备的服务器虚拟机，或需要 SCSI 命令兼容的场景（如虚拟机内运行存储管理软件）。

#### 5.3.4.virtio 总线

全称：Virtual I/O（虚拟 I/O）

作用：KVM 中性能最优的半虚拟化总线，通过前端（虚拟机内驱动）和后端（宿主机 QEMU）直接通信，跳过传统模拟的冗余步骤。

特点：

性能极强：相比 IDE/SCSI 纯模拟总线，I/O 延迟降低 50% 以上，吞吐量提升数倍（接近物理设备性能）。

需安装专用驱动：虚拟机内必须安装 virtio 驱动（Linux 内核 2.6.35+ 内置，Windows 需手动安装 virtio 驱动包）。

支持热插拔、多队列（multi-queue）等高级功能，适合高负载场景（如数据库、高并发服务）。

配置示例（XML）：

```
<disk type="file" device="disk">
```

```
<driver name="qemu" type="qcow2"/>
```

```
<source file="/data/virtio-disk.qcow2"/>
```

```
<target dev="vda" bus="virtio"/> <!-- virtio 总线，设备名为 vda -->
```

```
<address type="pci" domain="0x0000" bus="0x05" slot="0x00" function="0x0"/>
```

```
</disk>
```

适用场景：所有运行现代操作系统的虚拟机（尤其是生产环境），是 KVM 中推荐的默认总线类型。

5.3.5.USB 总线

在 KVM（基于 QEMU）虚拟化中，USB 总线是虚拟机内部模拟的用于连接 USB 控制器和 USB 设备的通信路径，其设计模拟了物理机的 USB 总线架构，负责在控制器与设备之间传输数据。以下是关于 KVM 中 USB 总线的核心概念、特点及配置方式：

一、USB 总线的基本概念

总线与控制器的关系

每个 USB 控制器（如 usb-xhci、usb-ehci 等）会关联一条或多条 USB 总线，作为设备连接的“通道”。

例如：usb-xhci 控制器默认创建一条高速总线，可直接连接 USB 3.0/2.0/1.1 设备；而 usb-ehci（USB 2.0）通常需要配合 usb-uhci/usb-ohci（USB 1.1）的总线来兼容低速设备。

总线标识

每条总线通过 bus 编号区分（通常从 0 开始），与控制器的 index 对应（如 index='0' 的控制器对应 bus='0' 的总线）。

设备通过 <address type='usb' bus='0' port='1'/> 绑定到具体总线的某个端口（port 编号从 1 开始）。

二、USB 总线的类型与特点

KVM 中 USB 总线的类型由其关联的控制器决定，不同类型总线支持的设备速率和协议不同：

| 控制器类型         | 对应总线类型     | 支持速率         | 总线特点                                       |
|---------------|------------|--------------|--------------------------------------------|
| usb-xhci      | USB 3.0 总线 | 5 Gbps（超高速）  | 单总线支持所有速率设备（USB 3.0/2.0/1.1），无需额外总线，现代系统推荐 |
| usb-ehci      | USB 2.0 总线 | 480 Mbps（高速） | 仅支持高速设备，低速设备需依赖 USB 1.1 总线（uhci/ohci）      |
| usb-uhci/ohci | USB 1.1 总线 | 12 Mbps（全速）  | 支持全速 / 低速设备，常作为 ehci 的辅助总线使用               |

三、USB 总线的设备容量

单条总线支持的最大设备数：受 USB 协议限制，每条总线最多支持 127 个设备（包括集线器级联的设备）。

实际限制：实际使用中很少达到上限，受限于：

虚拟机配置的 USB 端口数量（port 编号）；

宿主机的 USB 控制器硬件资源；

虚拟化性能（过多设备可能导致延迟增加）。

#### 四、USB 总线的配置方式

USB 总线无需单独配置，而是通过控制器配置自动创建，设备则通过绑定到具体总线的端口来关联。

##### 注意事项 ↔

主机服务器上需存在支持 USB 1.1、USB 2.0 和 USB 3.0 标准的 USB 控制器硬件和模块。

为虚拟机配置 USB 控制器时，请按照 USB 1.1、USB 2.0 到 USB 3.0 的顺序来配置。

一个 xHCI 控制器有 8 个端口，最多可以挂载 4 个 USB 3.0 设备和 4 个 USB 2.0 设备。一个 EHCI 控制器有 6 个端口，最多可以挂载 6 个 USB 2.0 设备。一个 UHCI 控制器有 2 个端口，最多可以挂载 2 个 USB 1.1 设备。

每台虚拟机最多支持配置一个相同类型的 USB 控制器。

不支持热插拔 USB 控制器。

若虚拟机没有安装 USB 3.0 的驱动，可能无法识别 xHCI 控制器，USB 3.0 驱动下载和安装方法请参见对应 OS 发行商官方说明。

为了不影响操作系统的兼容性，为虚拟机配置 USB 接口的 tablet 设备时，请指定 USB 控制器 bus 号为 0（默认挂载到 USB 1.1 控制器上）。

以下是完整配置流程：

##### 1. 配置控制器（自动创建对应总线）

在虚拟机 XML 的 <devices> 中定义控制器，系统会自动为其创建对应的 USB 总线：

xml

```
<devices>
```

```
<!-- 示例：USB 3.0 控制器（自动创建 bus=0 的 USB 3.0 总线） -->
```

```
<controller type='usb' index='0' model='usb-xhci'>
```

```
</controller>
```

```
<!-- 示例：USB 2.0 + USB 1.1 控制器组合（创建两条总线） -->
```

```
<controller type='usb' index='1' model='usb-ehci'>
```

```

<!-- 对应 bus=1 (USB 2.0 总线) -->

</controller>

<controller type='usb' index='2' model='usb-uhci'>

<!-- 对应 bus=2 (USB 1.1 总线) -->

</controller></devices>

```

## 2. 绑定设备到总线

通过 `<address>` 标签指定设备所属的总线 (bus) 和端口 (port) :

### (1) 绑定物理 USB 设备到总线

xml

```

<devices>

<!-- 物理 USB 设备 (鼠标) 绑定到 bus=0 (xhci 控制器的总线) 的 port=1 -->

<hostdev mode='subsystem' type='usb' managed='yes'>

  <source>

    <vendor id='0x046d' /> <!-- 设备厂商 ID -->

    <product id='0xc077' /> <!-- 设备产品 ID -->

  </source>

  <address type='usb' bus='0' port='1' /> <!-- 关联到 USB 3.0 总线的 1 号端口 -->

</hostdev></devices>

```

### (2) 绑定虚拟 USB 设备到总线

xml

```

<devices>

<!-- 虚拟 USB 存储设备 (U 盘) 绑定到 bus=1 (ehci 控制器的总线) 的 port=2 -->

<disk type='file' device='disk'>

  <driver name='qemu' type='raw' />

  <source file='/data/usb-disk.img' /> <!-- 宿主机镜像文件 -->

  <target dev='sdc' bus='usb' />

  <address type='usb' bus='1' port='2' /> <!-- 关联到 USB 2.0 总线的 2 号端口 -->

</disk></devices>

```

## 3. 动态调整总线设备 (无需重启)

使用 `virsh attach-device` 命令可动态添加设备到总线:

bash

# 准备设备配置文件（usb-device.xml），指定 bus 和 portvirsh attach-device ubuntuvm

usb-device.xml --live # --live 表示立即生效

## 五、常见问题与注意事项

### 设备无法识别

检查设备是否绑定到正确的总线（如高速设备需绑定到 usb-xhci 或 usb-ehci 总线）。

确认控制器配置正确（如 usb-ehci 需要 usb-uhci/ohci 辅助总线才能支持低速设备）。

### 总线冲突

避免多个设备绑定到同一总线的同一端口（port 编号需唯一）。

若需添加大量设备，可通过增加控制器（index 递增）扩展总线数量。

### 性能优化

高速设备（如 USB 硬盘）优先绑定到 usb-xhci 总线（USB 3.0），避免使用 USB 1.1 总线导致性能瓶颈。

## 总结

KVM 中的 USB 总线是控制器与设备之间的通信桥梁，其类型和功能由关联的 USB 控制器决定。通过合理配置控制器和设备的总线绑定，可满足虚拟机对不同速率 USB 设备的需求，兼顾兼容性和性能。实际使用中，推荐优先选择 usb-xhci 控制器（对应 USB 3.0 总线），以简化配置并支持全速率设备。

## 5.3.6.SATA 总线

全称：Serial ATA（串行 ATA）

作用：IDE 的继任者，KVM 模拟 SATA 控制器支持连接磁盘和光驱，平衡兼容性和性能。

特点：

性能优于 IDE（理论速率 600MB/s），支持热插拔和 Native Command Queuing（NCQ，优化 I/O 队列）。

兼容性较好（支持 Windows Vista+、Linux 2.6+），但不如 IDE 兼容老旧系统。

配置示例（XML）：

xml

```
<disk type="file" device="disk">

  <driver name="qemu" type="qcow2"/>

  <source file="/data/sata-disk.qcow2"/>

  <target dev="sda" bus="sata"/>  <!-- SATA 总线 -->

  <address type="drive" controller="0" bus="0" target="0" unit="0"/></disk><!-- SATA 控制器配置 --><controller type="sata" index="0"/>
```

适用场景：需要兼容 SATA 协议但不需要极致性能的场景（逐渐被 virtio 替代）。

总线选择建议

| 优先级 | 总线类型                  | 适用场景                           | 性能                 | 兼容性     |
|-----|-----------------------|--------------------------------|--------------------|---------|
| 1   | virtio                | 生产环境、高负载服务（数据库、Web 服务器）、现代操作系统 | ★★★★★              | 中等（需驱动） |
| 2   | SCSI<br>(virtio-scsi) | 多存储设备、需要 SCSI 命令支持的服务器         | ★★★★☆              | 良好      |
| 3   | SATA                  | 过渡场景，兼容较新但不支持 virtio 的系统       | ★★★☆☆              | 良好      |
| 4   | USB                   | 临时连接 USB 设备（U 盘、外设）            | ★★★☆☆<br>(USB 3.0) | 良好      |
| 5   | IDE                   | 老旧系统（Windows XP、Linux 2.4）     | ★★☆☆☆              | 极佳      |

总结

KVM 通过模拟多种总线类型实现设备连接的灵活性和性能平衡：

virtio 总线 是性能首选，适合绝大多数现代场景；

传统总线（IDE/SATA）用于兼容性需求；

PCI/PCIe 是设备连接的基础，支撑高级设备的高性能通信；

USB 总线 专注于外设连接。

实际使用中，应根据虚拟机操作系统版本、负载类型和性能需求选择合适的总线，并通过 libvirt XML 配置或 virsh 命令管理总线设备。

### 5.3.7.添加总线设备

#### virsh attach-device

`virsh attach-device` 是 KVM 虚拟化中用于动态为虚拟机添加设备的命令，支持在线（运行中）或离线（关闭状态）添加虚拟设备（如磁盘、网卡、USB 设备、PCI 设备等），无需重新定义虚拟机配置，灵活性极高。

#### 核心作用

动态添加设备：无需重启虚拟机即可为运行中的虚拟机添加新设备（部分设备支持热插拔）。

临时或永久生效：可选择设备仅在当前运行周期有效（重启后消失），或写入配置文件永久生效。

支持多种设备类型：包括磁盘、网卡、USB 设备、PCI 直通设备、显卡等。

#### 基本语法

##### bash

`virsh attach-device <域名> <设备 XML 文件> [选项]`

<域名>：虚拟机名称、ID 或 UUID。

<设备 XML 文件>：包含待添加设备配置的 XML 文件路径（需提前定义设备参数）。

#### 主要选项

| 选项                        | 作用                                                                 |
|---------------------------|--------------------------------------------------------------------|
| <code>--live</code>       | 对运行中的虚拟机生效（在线添加，支持热插拔的设备立即可用）。                                     |
| <code>--config</code>     | 仅修改配置文件，下次启动虚拟机时生效（离线添加）。                                          |
| <code>--current</code>    | 根据虚拟机当前状态生效（运行中则 <code>--live</code> ，关闭则 <code>--config</code> ）。 |
| <code>--persistent</code> | 同 <code>--config</code> ，添加后写入配置文件，永久生效。                           |
| <code>--verbose</code>    | 显示添加过程的详细信息。                                                       |

#### 典型使用场景

##### 1. 在线添加一块 virtio 磁盘（临时生效）

##### 步骤：

创建设备 XML 文件（如 `new-disk.xml`）：

xml

<disk type="file" device="disk">



```

<driver name="qemu" type="qcow2"/>

<source file="/data/new-disk.qcow2"/> <!-- 磁盘文件路径 -->

<target dev="vdb" bus="virtio"/> <!-- 设备名和总线类型 -->

<address type="pci" domain="0x0000" bus="0x06" slot="0x00" function="0x0"/> <!--
PCI 地址 --></disk>

```

在线添加设备（仅当前运行有效，重启后消失）：

```
bash
```

```
virsh attach-device vm1 new-disk.xml --live --verbose
```

## 2. 永久添加一块 USB 设备（写入配置）

步骤：

创建 USB 设备 XML 文件（如 usb-device.xml）：

```
xml
```

```

<hostdev mode="subsystem" type="usb" managed="yes">

  <source>

    <vendor id="0x1234"/> <!-- 设备厂商 ID -->

    <product id="0x5678"/> <!-- 设备产品 ID -->

  </source></hostdev>

```

（可通过 lsusb 命令在宿主机查询 USB 设备的 vendor id 和 product id）

添加并写入配置（永久生效）：

```
bash
```

```
virsh attach-device vm1 usb-device.xml --config --verbose
```

## 3. 为虚拟机添加 PCI 直通设备（如物理显卡）

步骤：

查看宿主机可用的 PCI 设备（获取 domain:bus:slot:function）：

```
bash
```

```
lspci | grep -i vga # 假设要直通显卡# 输出示例：01:00.0 VGA compatible controller:
```

NVIDIA Corporation ...

创建 PCI 设备 XML 文件（如 pci-gpu.xml）：

```
xml
```

```
<hostdev mode="subsystem" type="pci" managed="yes">
  <source>
    <address domain="0x0000" bus="0x01" slot="0x00" function="0x0"/>  <!-- 对应宿主
机的 01:00.0 -->
  </source></hostdev>
```

在线添加直通设备：

```
bash
```

```
virsh attach-device vm1 pci-gpu.xml --live --config # 同时生效于当前和配置文件
```

关键注意事项

设备 XML 格式：必须严格遵循 libvirt 设备定义规范，不同设备（磁盘、网卡等）的 XML 结构不同，错误格式会导致添加失败。

热插拔支持：

支持热插拔：virtio 磁盘 / 网卡、USB 设备、PCIe 设备（需 pcie-root-port 配置热插拔）。

不支持热插拔：IDE 磁盘、传统 PCI 设备（需关闭虚拟机添加）。

地址冲突：手动指定 <address> 时，需确保与虚拟机内已有设备的 PCI 地址不冲突（可通过 virsh domaddrlist <域名> 查看已用地址）。

权限问题：添加宿主机文件 / 设备（如磁盘文件、USB 设备）时，需确保 QEMU 进程（通常用户为 qemu）有访问权限。

移除设备：通过 virsh detach-device <域名> <设备 XML 文件> 命令移除已添加的设备。

总结

virsh attach-device 是 KVM 中动态管理设备的核心命令，通过 XML 配置文件灵活添加各类设备，支持在线操作和永久生效，极大简化了虚拟机的设备扩展流程。使用时需注意设备格式、热插拔兼容性和地址冲突问题，确保设备添加后正常工作。

### 5.3.8.分离总线设备

```
virsh detach-device
```

virsh detach-device 命令用于动态从虚拟机中移除设备，与 attach-device 相对应，支持在线（运行中）或离线移除虚拟设备（如磁盘、网卡、USB 设备等），无需重启虚拟机即可生效。

## 核心作用

动态移除设备：从运行中的虚拟机中安全移除设备（支持热插拔的设备可立即断开连接）。

临时或永久生效：可选择仅在当前运行周期移除（重启后设备恢复），或从配置文件中删除（永久移除）。

支持多种设备类型：包括磁盘、网卡、USB 设备、PCI 直通设备等。

## 基本语法

bash

**virsh detach-device** <域名> <设备 XML 文件> [选项]

<域名>：虚拟机名称、ID 或 UUID。

<设备 XML 文件>：包含待移除设备配置的 XML 文件（需与添加时的配置一致，至少包含设备类型和关键标识）。

## 主要选项

| 选项                  | 作用                                                     |
|---------------------|--------------------------------------------------------|
| <b>--live</b>       | 从运行中的虚拟机中移除设备（在线移除，支持热插拔的设备立即生效）。                      |
| <b>--config</b>     | 仅从配置文件中删除设备，下次启动后生效（不影响当前运行状态）。                        |
| <b>--current</b>    | 根据虚拟机当前状态移除（运行中则 <b>--live</b> ，关闭则 <b>--config</b> ）。 |
| <b>--persistent</b> | 同 <b>--config</b> ，从配置文件中永久删除设备。                       |
| <b>--verbose</b>    | 显示移除过程的详细信息。                                           |

## 典型使用场景

### 1. 在线临时移除一块 virtio 磁盘

步骤：

使用与添加时相同的 XML 配置（如 **remove-disk.xml**，至少保留关键标识）：

xml

```
<disk type="file" device="disk">
  <source file="/home/vm/.aaa.qcow2"/> <!-- 磁盘文件路径（关键标识） -->
  <target dev="vdd" bus="virtio"/> <!-- 设备名和总线（关键标识） --></disk>
```

在线移除（仅当前运行周期生效，重启后设备会恢复）：

bash

**virsh detach-device** fnos **--live --file** remove-disk.xml **--verbose**

### 2. 永久移除一块 USB 设备（从配置中删除）

步骤：

创建包含 USB 设备标识的 XML (remove-usb.xml) :

```
xml
```

```
<hostdev mode="subsystem" type="usb" managed="yes">
  <source>
    <vendor id="0x1234"/> <!-- 厂商 ID (关键标识) -->
    <product id="0x5678"/> <!-- 产品 ID (关键标识) -->
  </source></hostdev>
```

永久移除 (从配置文件中删除, 下次启动不再加载) :

```
bash
```

```
virsh detach-device fnos --config --file remove-usb.xml
```

3. 同时在线移除并永久删除设备

适用于确认不再使用该设备的场景:

```
bash
```

```
# 立即移除且从配置中删除 (彻底移除) virsh detach-device fnos --live --config --file
remove-nic.xml
```

关键注意事项

**XML 配置要求:** 移除设备时, XML 无需完整, 但必须包含唯一标识设备的关键信息 (如磁盘的 `source file` 和 `target dev`, USB 设备的 `vendor id` 和 `product id`), 否则 `libvirt` 无法识别要移除的设备。

热插拔支持:

支持在线移除: `virtio` 磁盘 / 网卡、USB 设备、PCIe 设备 (需 `pcie-root-port` 配置热插拔)。

不支持在线移除: IDE 磁盘、传统 PCI 设备 (需先关闭虚拟机)。

**数据安全:** 移除磁盘前, 需确保虚拟机内已卸载该磁盘 (如 Linux 中执行 `umount /dev/vdd`), 避免数据损坏。

验证移除结果:

```
bash
```

```
# 查看虚拟机当前设备列表 (确认设备已移除) virsh domblklist fnos # 磁盘设备 virsh
domiflist fnos # 网络设备 virsh domusbattach fnos # USB 设备
```

总结

`virsh detach-device` 是动态管理虚拟机设备的重要命令, 通过与添加时一致的 XML 标识, 可灵活移除临时或永久设备。使用时需注意设备热插拔兼容性和数据安全, 确保移除操作不会导致虚拟机异常。

5.4.控制器（网络接口）设备

| 内容             | 命令                     | 说明               |
|----------------|------------------------|------------------|
| 添加网络接口         | virsh attach-interface | 为虚拟机添加网络接口       |
| 分离网线接口         | virsh detach-interface | 从虚拟机移除网络接口       |
| 网络接口<br>链路状态控制 | virsh domif-setlink    | 设置网络接口的链路状态      |
| 网络接口<br>带宽控制   | virsh domiftune        | 查看 / 调整网络接口的带宽限制 |

```
<interface type='network'>

  <mac address='52:54:00:99:f6:ac'/>

  <source network='br0'/>

  <model type='virtio'/>

  <address type='pci' domain='0x0000' bus='0x01' slot='0x00' function='0x0'/>

</interface>
```

概述 ↔

XML 配置文件可以配置虚拟网络设备，包括 ethernet 模式、bridge 模式、vhostuser 模式等，本节介绍虚拟网卡设备的配置方法。

元素介绍 ↔

XML 配置文件中使用元素“interface”，其属性“type”表示虚拟网卡的模式，可选的值有“ethernet”、“bridge”、“vhostuser”等，下面以“bridge”模式虚拟网卡为例介绍其子元素以及对应的属性。

表 3 bridge 模式虚拟网卡常用子元素

| 子元素    | 子元素含义        | 属性及含义                           |
|--------|--------------|---------------------------------|
| mac    | 虚拟网卡的 mac 地址 | address: 指定 mac 地址，若不配置，会自动生成。  |
| target | 后端虚拟网卡名      | dev: 创建的后端 tap 设备的名称。           |
| source | 指定虚拟网卡后端     | bridge: 与 bridge 模式联合使用，值为网桥名称。 |
| boot   | 表示此网卡可以作     | order: 指定网卡的启动顺序。               |

| 子元素         | 子元素含义     | 属性及含义                                       |
|-------------|-----------|---------------------------------------------|
|             | 为远程启动     |                                             |
| model       | 表示虚拟网卡的类型 | type: bridge 模式网卡通常使用 virtio。               |
| virtualport | 端口类型      | type: 若使用 OVS 网桥，需要配置为 openvswitch。         |
| driver      | 后端驱动类型    | name: 驱动名称，通常取值为 vhost。<br>queues: 网卡设备队列数。 |

`type='network'`: 网络模式为“基于 libvirt 网络”，但实际通过 `<source network='br0'/>` 关联到主机的 br0 网桥，本质是桥接模式（虚拟机直接接入主机物理网络，获得与主机同网段的 IP）。

`<mac address='52:54:00:99:f6:ac'/>`: 虚拟网卡的 MAC 地址，52:54:00 是 libvirt 预留的前缀，避免与物理网卡 MAC 冲突。

`<source network='br0'/>`: 指定关联的主机网桥为 br0（需主机提前通过 brctl 或 nmcli 配置网桥，绑定物理网卡如 eth0）。

`<model type='virtio'/>`: 虚拟网卡型号为 virtio-net（高性能网卡，支持 TCP 分段卸载、巨帧等特性，Ubuntu 24.10 已默认支持驱动），比 e1000（Intel 千兆网卡模拟）性能提升更明显。

`<address.../>`: 虚拟网卡在 PCIe 总线上的地址（bus=0x01, slot=0x00），对应之前的 PCIe 根端口。

### 5.4.1.添加网络接口

`virsh attach-interface`

`virsh attach-interface` 命令用于为运行中的虚拟机或持久化配置添加网络接口，以下是各参数的详细说明：

核心参数

`--domain <名称/UUID>`

必需参数，指定要操作的虚拟机（通过名称或 UUID 标识）。

示例: `--domain centos7`

`--type <类型>`

必需参数, 指定接口类型, 常见值:

**bridge**: 桥接模式 (连接到宿主机的网桥)

**network**: 网络模式 (连接到 `libvirt` 定义的虚拟网络)

**direct**: 直接连接 (绑定到宿主机物理网卡)

**user**: 用户模式 (**NAT** 方式, 适合于简单测试)

示例: `--type bridge`

`--source <源>`

指定网络源, 值取决于 `--type`:

若 `--type bridge`: 指定网桥名称 (如 `br0`)

若 `--type network`: 指定虚拟网络名称 (如 `default`)

若 `--type direct`: 指定宿主机物理网卡 (如 `eth0`)

示例: `--source br0` (桥接至 `br0` 网桥)

接口属性配置

`--mac <MAC 地址>`

手动指定接口的 **MAC** 地址 (格式: `xx:xx:xx:xx:xx:xx`)。

若不指定, `libvirt` 会自动生成一个唯一 **MAC**。

示例: `--mac 52:54:00:ab:cd:ef`

`--model <网卡模型>`

指定虚拟机内部识别的网卡型号, 影响驱动兼容性:

**virtio**: 高性能半虚拟化驱动 (推荐, 需虚拟机内安装驱动)

**e1000**: 模拟 Intel 千兆网卡 (兼容性好)

**r8169**: 模拟 Realtek 网卡 (老旧系统兼容)

示例: `--model virtio`

`--target <接口名>`

指定虚拟机内部显示的接口名称 (如 `eth0`、`ens3`)。

若不指定, 由虚拟机操作系统自动分配。

示例: `--target eth1`

`--alias <别名>`

为接口设置一个别名（仅用于标识，不影响功能）。

示例：--alias "backup-network"

过滤规则

--inbound <规则>

配置入站流量过滤规则（基于 libvirt 网络过滤框架）。

需预先定义过滤规则，示例：--inbound "clean-traffic"

--outbound <规则>

配置出站流量过滤规则，用法同--inbound。

示例：--outbound "no-ip-spoofing"

--script <脚本路径>

指定接口启动 / 关闭时执行的脚本（用于自定义网络配置）。

示例：--script /etc/libvirt/hooks/if-up.sh

生效范围控制

--live

仅对运行中的虚拟机生效（临时添加，重启后消失）。

--config / --persistent

仅修改虚拟机的持久化配置（下次启动时生效，不影响当前运行状态）。

两者功能相同，--persistent 是别名。

--current

同时对运行状态和持久化配置生效（立即添加且下次启动保留）。

其他参数

--managed

由 libvirt 完全管理接口的生命周期（包括自动清理），推荐使用。

--print-xml

打印接口的 XML 配置（仅预览，不实际执行添加操作）。

用于验证配置是否正确，示例：

bash

virsh attach-interface --domain vm1 --type bridge --source br0 --print-xml

完整示例

bash



```
# 为运行中的 vm1 添加一个桥接接口（立即生效且持久化）virsh attach-interface \  
--domain vm1 \  
--type bridge \  
--source br0 \  
--model virtio \  
--mac 52:54:00:11:22:33 \  
--target eth1 \  
--current
```

此命令会为 vm1 添加一个连接到 br0 网桥的接口，使用 virtio 模型，MAC 地址为 52:54:00:11:22:33，在虚拟机内显示为 eth1，且配置立即生效并持久化。

分享

详细介绍下 virsh attach-interface 命令的参数——alias

除了上述介绍的参数，virsh attach-interface 命令还有其他参数吗？

如何使用 virsh attach-interface 命令为虚拟机添加桥接模式网络接口？

### 5.4.2.分离网线接口

virsh detach-interface

virsh detach-interface 命令用于从虚拟机中移除网络接口，其选项含义如下：

- config: 仅对持久化配置生效（下次启动后移除，不影响当前运行状态）
- domain: 指定目标虚拟机的名称或 UUID（必需参数）
- mac: 通过 MAC 地址指定要移除的接口（与--type 配合使用）
- type: 指定接口类型（如 bridge、network 等，需与--mac 配合定位接口）
- current: 对当前运行状态和配置同时生效（立即移除且下次启动不恢复）
- live: 仅对运行中的虚拟机生效（临时移除，重启后恢复）
- persistent: 同--config，表示操作将持久化到配置中

示例用法：

bash

```
# 从运行中的 vm1 虚拟机临时移除指定 MAC 的桥接接口 virsh detach-interface --domain  
vm1 --type bridge --mac 52:54:00:12:34:56 --live
```

```
# 从 vm1 中永久移除指定 MAC 的网络接口（立即生效且下次启动不恢复） virsh  
detach-interface --domain vm1 --type network --mac 52:54:00:aa:bb:cc --current
```

### 5.4.3.网络接口链路状态控制

```
virsh domif-setlink
```

virsh domif-setlink 命令用于设置虚拟机网络接口的链路状态（启用或禁用），以下是各参数的详细说明：

核心参数

**--domain <名称/UUID>**

必需参数，指定要操作的虚拟机（通过名称或 UUID 标识）。

示例：--domain ubuntu-vm

**--interface <接口标识>**

必需参数，指定要操作的网络接口，可通过以下方式标识：

接口名称（如虚拟机内的 eth0、ens3）

MAC 地址（如 52:54:00:ab:cd:ef）

示例：--interface eth0 或 --interface 52:54:00:ab:cd:ef

**--state <状态>**

必需参数，指定链路状态：

**up**：启用接口（激活链路）

**down**：禁用接口（关闭链路）

示例：--state down（禁用接口）

生效范围控制

**--config / --persistent**

仅修改虚拟机的持久化配置（下次启动时生效，不影响当前运行状态）。

两者功能相同，--persistent 是--config 的别名。

示例：--config（设置为下次启动后生效）

使用示例

```
bash
```

```
# 临时禁用运行中虚拟机 vm1 的 eth0 接口（重启后恢复）virsh domif-setlink --domain
vm1 --interface eth0 --state down

# 永久禁用虚拟机 vm1 的指定 MAC 接口（下次启动生效）virsh domif-setlink --domain
vm1 --interface 52:54:00:11:22:33 --state down --persistent

# 启用虚拟机 vm1 的 ens3 接口（立即生效）virsh domif-setlink --domain vm1 --interface
ens3 --state up
```

该命令常用于模拟网络故障、临时断开虚拟机网络连接等场景，操作效果类似在虚拟机内部执行 ifdown/ifup 命令，但通过宿主机层面直接控制。

#### 5.4.4.网络接口带宽控制

```
virsh domiftune
```

virsh domiftune 命令用于调整虚拟机网络接口的带宽限制（QoS 配置），可以控制入站和出站流量的速率，以下是各参数的详细说明：

核心参数

--domain <名称/UUID>

必需参数，指定要操作的虚拟机（通过名称或 UUID 标识）。

示例：--domain web-server-vm

--interface <接口标识>

必需参数，指定要调整的网络接口，可通过以下方式标识：

接口名称（如虚拟机内的 eth0、ens3）

MAC 地址（如 52:54:00:f1:2c:3b）

示例：--interface ens3 或 --interface 52:54:00:f1:2c:3b

带宽限制配置

--inbound <参数>

配置入站流量（进入虚拟机的流量）的带宽限制，格式为 rate,burst,peak:

rate: 平均速率（单位：kbps，必需）

burst: 突发速率上限（单位：kb，可选）

peak: 峰值速率（单位：kbps，可选）

示例：--inbound 10240,2048,20480（入站平均 10Mbps，突发 2048kb，峰值 20Mbps）

**--outbound <参数>**

配置出站流量（从虚拟机发出的流量）的带宽限制，格式同 **--inbound**：

示例：**--outbound 5120**（出站平均 5Mbps，不限制突发和峰值）

生效范围控制

**--live**

仅对运行中的虚拟机生效（临时调整，重启后失效）。

**--config**

仅修改虚拟机的持久化配置（下次启动时生效，不影响当前运行状态）。

**--current**

同时对运行状态和持久化配置生效（立即调整且下次启动保留）。

使用示例

**bash**

# 临时限制虚拟机 web-vm 的 eth0 接口：入站 10Mbps，出站 5Mbps  
**virsh domiftune**

**--domain web-vm --interface eth0 --inbound 10240 --outbound 5120 --live**

# 永久限制虚拟机 db-vm 的指定 MAC 接口：入站 20Mbps（突发 4096kb，峰值 30Mbps）

**virsh domiftune --domain db-vm --interface 52:54:00:a1:b2:c3 --inbound 20480,4096,30720**

**--config**

# 同时生效的配置（立即生效且持久化）  
**virsh domiftune --domain app-vm --interface ens3**

**--outbound 15360 --current**

该命令常用于对虚拟机进行网络流量控制，避免单个虚拟机占用过多带宽影响其他设备，尤其适合多虚拟机共享网络资源的场景。

### 5.4.5.网卡直通

允许虚拟机直接使用物理网卡，实现接近原生的网络性能，适合高带宽需求场景（如服务器、网络测试）。配置思路与显卡直通类似，但更简单（无需处理驱动兼容性问题），以下是详细步骤：

一、前提条件

硬件支持：

**CPU**：开启 Intel VT-d 或 AMD-Vi（同显卡直通，需在 BIOS 中启用）。

主板：支持 PCIe 直通（BIOS 中开启“PCIe 直通”“IOMMU”）。

网卡：任意 PCIe 网卡（有线网卡为主，无线网卡直通兼容性较差）。

软件环境：

宿主机已安装 KVM 环境（libvirt、qemu-kvm）。

宿主机已启用 IOMMU（参考显卡直通的 grub 配置步骤）。

## 二、配置步骤

### 1. 确认 IOMMU 已启用

若已配置过显卡直通，可跳过此步。否则执行：

```
bash
```

```
# 验证 IOMMU 是否开启 dmesg | grep -i iommu# 输出包含 "IOMMU enabled" 即为成功
```

### 2. 识别网卡 PCI 信息

查看网卡的 PCI 地址和设备 ID：

```
bash
```

```
# 列出所有网络设备（带 PCI 地址）
```

```
lspci | grep -iE "ethernet|network"
```

输出示例（记录网卡的 bus:slot.function）：

```
plaintext
```

```
02:00.0 Ethernet controller: Intel Corporation I210 Gigabit Network Connection (rev 03)
```

网卡 PCI 地址：02:00.0（后续配置用）

获取设备 ID（用于绑定驱动）：

```
bash
```

```
lspci -n -s 02:00.0 # 替换为你的网卡 PCI 地址
```

输出示例（8086:1533 为 VendorID:DeviceID）：

```
plaintext
```

```
02:00.0 0200: 8086:1533 (rev 03)
```

### 3. 绑定网卡到 vfio-pci 驱动

让宿主机放弃对网卡的控制：

```
bash
```

```
# 编辑驱动绑定配置（若已为显卡创建过，直接添加即可）sudo vim
/etc/modprobe.d/vfio.conf

添加网卡设备 ID:

bash

options vfio-pci ids=8086:1533 # 替换为你的网卡 VendorID:DeviceID

阻止宿主机加载网卡原生驱动（以 Intel 网卡为例）:

bash

sudo vim /etc/modprobe.d/blacklist.conf

添加:

bash

blacklist e1000e # Intel I210 对应的驱动，其他网卡需替换（如 igb、r8169 等）

更新 initramfs 并重启:

bash

sudo update-initramfs -u -k all # Ubuntu/Debian# 或 sudo dracut -f (CentOS/RHEL)sudo
reboot

验证驱动绑定:

bash

lspci -k -s 02:00.0 # 替换为网卡 PCI 地址

输出中出现 Kernel driver in use: vfio-pci 即为成功。

4. 修改虚拟机 XML 配置

通过 virsh edit 虚拟机名称 编辑配置，添加网卡直通:

（可选）删除原虚拟网卡：若不需要虚拟网卡，可删除 <interface> 标签段（如 bridge 或
nat 模式的网卡）。

添加网卡直通配置：在 <devices> 标签内添加:

xml

<hostdev mode="subsystem" type="pci" managed="yes">

  <source>

    <address domain="0x0000" bus="0x02" slot="0x00" function="0x0"/> <!-- 替换为网
卡 PCI 地址 -->

  </source>
```

`<address type="pci" domain="0x0000" bus="0x08" slot="0x00" function="0x0"/> <!--`  
虚拟机内地址，可自定义 --></hostdev>

5. 启动虚拟机并配置网络

`bash`

`virsh start` 虚拟机名称

在虚拟机内：

**Windows：**系统会自动识别网卡，若未安装驱动，需手动安装网卡官方驱动。

**Linux：**通常自动加载驱动，通过 `ip addr` 查看网卡是否识别，手动配置 IP 即可。

三、优化与注意事项

性能优化：

若网卡支持 **SR-IOV**，可创建虚拟功能（VF）分配给多个虚拟机，比直通物理网卡更灵活。

在 XML 中添加 `<rom bar="on" file="/path/to/rom.bin"/>`（需网卡支持），提升启动速度。

多网卡场景：

可同时直通多个物理网卡，分别用于不同网络（如公网、内网）。

保留一个虚拟网卡用于宿主机与虚拟机通信（若需要）。

常见问题：

虚拟机启动失败：检查网卡是否被宿主机占用（`lsof -i` 确认无进程使用）。

网络不通：确认虚拟机内 IP 配置与物理网络匹配（网关、子网掩码正确）。

网卡直通配置比显卡简单，兼容性更好，适合需要高性能网络的场景。如果是服务器级网卡（如 Intel X710、Mellanox ConnectX），还可结合 **SR-IOV** 实现多虚拟机共享，进一步提升资源利用率。

5.5.串口与控制台

| 内容         | 命令                             | 说明                           |
|------------|--------------------------------|------------------------------|
| console    | <code>virsh console</code>     | 连接到虚拟机的文本控制台（需配置 console 设备） |
| console 查询 | <code>virsh tty console</code> | console 查询                   |
| vnc 端口号    | <code>virsh vncdisplay</code>  | 查看虚拟机的 VNC 连接信息（如端口号）        |

| 内容     | 命令                            | 说明                                          |
|--------|-------------------------------|---------------------------------------------|
| 屏幕截图   | <code>virsh screenshot</code> | 捕获虚拟机的屏幕截图                                  |
| 虚拟键盘   | <code>virh send-key</code>    | 向虚拟机发送键盘事件（如 <code>send-key KEY_F1</code> ） |
| 注入 NMI | <code>virsh nject-nmi</code>  | 向虚拟机注入 NMI（非屏蔽中断，用于调试）                      |

发送命令      `virsh qemu-agent-command`    向 QEMU guest-agent 发送命令（需安装 agent）

内置用户密码      `virsh set-user-password`      通过 guest-agent 设置虚拟机内用户密码

```
<serial type='pty'>
  <target type='isa-serial' port='0'>
    <model name='isa-serial' />
  </target>
</serial>

<console type='pty'>
  <target type='serial' port='0' />
</console>
```

串行端口（Serial Port）和控制台（Console）的关键配置，主要作用是让物理机能够通过 `virsh console` 命令连接到虚拟机的串口终端。以下是每个字段和参数的详细解释：

一、<serial> 标签：定义串行端口设备

```
xml
<serial type='pty'>
  <target type='isa-serial' port='0'>
    <model name='isa-serial' />
  </target></serial>
```

1. <serial type='pty'>

<serial>：根标签，用于定义虚拟机的串行端口设备（类似物理机的 COM 口）。

type='pty'：指定串行端口的后端类型为 pty（伪终端，Pseudo Terminal）。

pty 是 KVM 中最常用的类型，它会在物理机上创建一个伪终端设备（如 `/dev/pts/0`），作为虚拟机串口与物理机的通信桥梁。

其他可选类型：file（输出到文件）、tcp（通过网络 TCP 传输）等，但 pty 是本地控制台连接的标准选择。



## 2. <target type='isa-serial' port='0'>

<target>: 定义虚拟机内部看到的串行端口属性。

type='isa-serial': 指定虚拟机内部的串口类型为 ISA 串行端口（传统的硬件接口标准）。

其他可选类型: pci-serial（PCI 接口的串口，现代虚拟机更常用），但 isa-serial 兼容性更好。

port='0': 指定虚拟机内部的串口编号，对应虚拟机中的设备文件 ttyS0（Linux）或 COM1（Windows）。

若添加多个串口，可递增编号（如 port='1' 对应 ttyS1 或 COM2）。

## 3. <model name='isa-serial' />

<model>: 指定虚拟机模拟的串口硬件型号。

name='isa-serial': 对应 type='isa-serial'，表示模拟传统 ISA 总线的串口设备，确保操作系统能识别。

若 type 为 pci-serial，name 可设置为 16550a（常见的 PCI 串口芯片型号）。

## 二、<console> 标签：关联控制台与串行端口

xml

```
<console type='pty'>
```

```
  <target type='serial' port='0' /></console>
```

### 1. <console type='pty'>

<console>: 根标签，定义虚拟机的控制台接口，用于 virsh console 命令的连接。

type='pty': 与 <serial> 的 type 保持一致，指定控制台通过伪终端与物理机通信。

### 2. <target type='serial' port='0' />

<target>: 定义控制台关联的目标设备。

type='serial': 指定控制台关联到虚拟机的串行端口（而非其他设备如 VGA 控制台）。

port='0': 指定关联的串口编号，需与 <serial> 中的 port='0' 一致，确保控制台连接到正确的串口。

总结：配置的作用

这两段配置的核心是：

为虚拟机添加一个串行端口（ttyS0/COM1），通过物理机的伪终端（pty）实现通信。

将 virsh console 命令与该串口绑定，使得物理机可以直接通过串口登录虚拟机。

配置后，需确保虚拟机内部已启用串口登录服务（如 `serial-getty@ttyS0` 或 `getty@ttyS0`），才能通过 `virsh console <虚拟机名>` 成功连接。

### 5.5.1.console

`virsh console` 命令用于连接到虚拟机的串行控制台（serial console），实现与虚拟机的文本交互，以下是各参数的详细说明：

#### 核心参数

`--domain <名称/UUID>`

必需参数，指定要连接的虚拟机（通过名称或 UUID 标识）。

示例：`--domain centos-vm` 或 `--domain 8f3e7b2a-1c4d-5e6f-7g8h-9i0j1k2l3m4n`

#### 连接控制参数

`--devname <设备名>`

指定要连接的串行控制台设备名称（适用于虚拟机有多个串行控制台的情况）。

默认通常为 `console0`，一般无需指定，仅在虚拟机配置了多个串行设备时使用。

示例：`--devname console1`

`--force`

强制连接到控制台，即使该控制台已被其他会话占用。

注意：可能会中断现有连接，谨慎使用。

示例：`virsh console --domain ubuntu-vm --force`

`--resume`

连接控制台时自动恢复处于暂停（`paused`）状态的虚拟机。

若虚拟机当前是暂停状态，使用此参数会先唤醒虚拟机再连接控制台。

示例：`virsh console --domain test-vm --resume`

`--safe`

启用安全模式连接，避免某些可能导致终端异常的控制字符传递。

适用于终端兼容性问题的场景，可能会限制部分高级终端功能。

#### 使用示例

`bash`

# 连接到名为 `web-server` 的虚拟机控制台 `virsh console --domain web-server`

```
# 强制连接到已有会话的虚拟机控制台 virsh console --domain db-vm --force
```

```
# 恢复暂停的虚拟机并连接控制台 virsh console --domain app-vm --resume
```

连接成功后，可通过 **Ctrl+] 组合键**退出控制台。使用该命令前，需确保虚拟机已配置并启用串行控制台（多数 Linux 发行版默认支持，Windows 需额外配置）。

虚拟机需要打开串口登录服务：**serial-getty@ttyS0.service**

### 5.5.2.vnc 端口号

```
virsh vncdisplay
```

**virsh screenshot** 命令用于捕获虚拟机的屏幕图像（截图），以下是各参数的详细说明：

核心参数

```
--domain <名称/UUID>
```

必需参数，指定要截图的虚拟机（通过名称或 UUID 标识）。

示例：**--domain windows-vm**

```
--file <文件路径>
```

必需参数，指定截图保存的文件路径及文件名（支持常见图像格式如 PNG、JPG 等）。

示例：**--file /tmp/vm-screenshot.png**（保存为 PNG 格式）

```
--screen <屏幕编号>
```

可选参数，指定要截图的屏幕（适用于多显示器的虚拟机）。

默认为 0（主屏幕），多屏幕时可指定 1、2 等。

示例：**--screen 1**（截取第二个屏幕的图像）

使用示例

```
bash
```

```
# 截取名为 ubuntu-vm 的虚拟机屏幕，保存为 PNG 格式 virsh screenshot --domain ubuntu-vm --file ~/vm-snapshot.png
```

```
# 截取 windows-vm 的第二个屏幕，保存为 JPG 格式 virsh screenshot --domain windows-vm --file /tmp/win-screen2.jpg --screen 1
```

注意事项

虚拟机需处于运行状态（**running**）才能截图，关闭或暂停状态无法使用。

截图支持 VNC 或 SPICE 等图形显示协议的虚拟机才能使用此命令，纯文本控制台（如串行控制台）无法截图。

截图格式由文件名指定的文件名后缀决定（如 .png、.jpg），推荐使用 PNG 格式以保证图像质量。

如果执行命令后提示错误，可能是虚拟机未配置图形显示设备，或当前用户权限不足（需 root 或 libvirt 组权限）。

### 5.5.3.屏幕截图

### 5.5.4.虚拟键盘

你可能是想了解 `virsh send-key` 命令，这是用于向虚拟机发送键盘按键的工具，常用于模拟虚拟机内部的键盘操作（尤其适用于无图形界面或无法直接交互的场景）。

bash

基本语法

`virsh send-key <虚拟机名称/UUID> [--codeset <编码集>] [--holdtime <按住时间>] <按键序列>`

主要参数说明

虚拟机名称 / UUID 必需参数，指定接收按键的目标虚拟机。示例：ubuntuvml 或虚拟机的 UUID。

`--codeset <编码集>`指定按键编码标准，常用值：

linux：适用于 Linux 系统（默认）

xorg：适用于 X11 图形环境

win32：适用于 Windows 系统

`--holdtime <毫秒>`指定按键按住的时间（默认 100 毫秒），用于模拟长按效果。

常用常用按键表示方法

需使用扫描码别名表示按键，常见示例：

字母 / 数字：a、b、1、2 等

功能键：F1、F2、Esc、Enter

修饰键：LeftCtrl、RightAlt、Shift、Tab

方向键：Up、Down、Left、Right

特殊键：BackSpace（退格）、Delete、Insert

使用示例

向虚拟机发送 Enter 键

```
bash
```

```
virsh send-key ubuntuvm Enter
```

发送组合键 Ctrl+Alt+Del（常用于重启）

```
bash
```

```
virsh send-key ubuntuvm LeftCtrl LeftAlt Delete
```

输入文本 "hello" 并按回车

```
bash
```

```
virsh send-key ubuntuvm h e l l o Enter
```

向 Windows 虚拟机发送 Win+R 组合键

```
bash
```

```
virsh send-key win10-vm --codeset win32 LeftWin r
```

注意事项

虚拟机需处于运行状态，且已配置图形界面或支持键盘输入的环境。

复杂组合键需按顺序发送（如 Ctrl+C 需先发送 LeftCtrl，再发送 c，最后释放 LeftCtrl）。

若按键无响应，可尝试更换 --codeset 编码集（如 Linux 用 linux，Windows 用 win32）。

通过 virsh send-key 可以远程模拟各种键盘操作，适合自动化脚本或虚拟机故障排查场景。

### 5.5.5.注入 NMI

你可能想了解的是 virsh inject-nmi 命令，这是用于向虚拟机注入非屏蔽中断（NMI, Non-Maskable Interrupt）的工具。NMI 是一种无法被软件屏蔽的硬件中断，通常用于处理严重错误（如系统崩溃、硬件故障）或触发特定调试流程。

基本语法

```
bash
```

```
virsh inject-nmi <虚拟机名称/UUID>
```

## 功能说明

作用：向指定虚拟机发送 NMI 信号，触发虚拟机内部的 NMI 处理机制。

典型用途：

强制 Linux 系统进入内核崩溃转储（kdump）流程，生成崩溃日志用于调试。

在系统无响应时，触发紧急调试模式（如 sysrq 组合键的底层机制依赖 NMI）。

测试虚拟机对硬件级中断的响应能力。

使用示例

bash

```
# 向名为 "server-vm" 的虚拟机注入 NMIvirsh inject-nmi server-vm
```

```
# 通过 UUID 向虚拟机注入 NMIvirsh inject-nmi 8f3e7b2a-1c4d-5e6f-7g8h-9i0j1k2l3m4n
```

注意事项

虚拟机配置要求：

需确保虚拟机的 CPU 配置支持 NMI（现代 x86/x86\_64 架构通常支持）。

部分虚拟化平台可能需要在虚拟机 XML 配置中启用 NMI 支持（默认通常已启用）。

操作系统处理：

Linux 系统默认可能需要配置 `kernel.panic_on_unrecovered_nmi=1` 等内核参数，才能对 NMI 做出响应（如触发崩溃转储）。

Windows 系统会将 NMI 视为严重错误，可能直接进入蓝屏（BSOD）或重启。

谨慎使用：

NMI 是低级别硬件中断，可能导致虚拟机数据丢失或非正常关机，仅在调试或紧急情况下使用。

总结

`virsh inject-nmi` 是一个用于调试和故障处理的高级命令，通过向虚拟机注入非屏蔽中断，触发系统的紧急处理机制，主要用于内核级问题排查。

## 5.5.6.发送命令

`virsh qemu-agent-command` 命令用于通过 QEMU Guest Agent 与虚拟机内部进行交互，实现宿主机对虚拟机的高级管理操作（如获取系统信息、修改配置等）。QEMU Guest Agent 是运行在虚拟机内部的服务程序，需提前在虚拟机中安装并启动（如 `qemu-guest-agent` 包）。

## 基本语法

bash

virsh qemu-agent-command <虚拟机名称/UUID> '<JSON 命令>' [--pretty]

## 核心参数说明

虚拟机名称/UUID 必需参数，指定目标虚拟机。示例：ubuntuvn 或虚拟机的 UUID。

'<JSON 命令>'必需参数，以 JSON 格式指定要执行的命令（具体命令由 QEMU Guest Agent 支持）。

--pretty 可选参数，使输出的 JSON 结果格式化（更易读）。

## 常用命令示例

QEMU Guest Agent 支持的命令较多，以下是常见用法：

### 1. 获取虚拟机系统信息

bash

# 获取基本系统信息（如操作系统类型、主机名等）virsh qemu-agent-command ubuntuvn

```
'{"execute": "guest-info"}' --pretty
```

### 2. 获取网络接口信息

bash

# 列出虚拟机内所有网络接口及 IP 地址 virsh qemu-agent-command ubuntuvn

```
'{"execute": "guest-network-get-interfaces"}' --pretty
```

### 3. 操作文件系统

bash

# 创建文件（需指定路径和权限）virsh qemu-agent-command ubuntuvn '{

```
  "execute": "guest-file-open",
```

```
  "arguments": {
```

```
    "path": "/tmp/test.txt",
```

```
    "mode": "w"
```

```
  }
```

```
}' --pretty
```

### 4. 重启 / 关闭虚拟机（安全操作）

bash

```
# 优雅重启虚拟机（相当于内部执行 shutdown -r now）virsh qemu-agent-command  
ubuntuvml '{"execute": "guest-shutdown", "arguments": {"mode": "reboot"}}'
```

```
# 优雅关闭虚拟机（相当于内部执行 shutdown -h now）virsh qemu-agent-command  
ubuntuvml '{"execute": "guest-shutdown", "arguments": {"mode": "poweroff"}}'
```

## 5. 获取磁盘信息

bash

```
# 列出虚拟机内的磁盘分区及挂载点 virsh qemu-agent-command ubuntuvml  
'{"execute": "guest-fsinfo"}' --pretty
```

前提条件

虚拟机内必须安装 `qemu-guest-agent` 包（如 Ubuntu/Debian: `sudo apt install qemu-guest-agent`；CentOS/RHEL: `sudo yum install qemu-guest-agent`）。

启动服务：`sudo systemctl enable --now qemu-guest-agent`。

虚拟机 XML 配置中需启用 `agent`（默认通常已配置）：

xml

```
<channel type='unix'>  
  <target type='virtio' name='org.qemu.guest_agent.0' /></channel>
```

注意事项

命令的执行依赖虚拟机内的 `agent` 服务，若服务未运行，会返回错误。

不同操作系统支持的命令可能略有差异（如 Windows 需安装对应的 QEMU Guest Agent）。

所有操作在虚拟机内部执行，权限受限于虚拟机内的用户（通常是 `root` 权限）。

通过 `qemu-agent-command` 可以实现比 `virsh` 基础命令更深入的虚拟机内部管理，是自动化运维和高级监控的重要工具。

分享

有哪些方法可以安装 QEMU Guest Agent？

如何使用 `virsh qemu-agent-command` 命令修改虚拟机配置？

除了 `virsh qemu-agent-command` 命令，还有哪些工具可以实现宿主机对虚拟机的高级管理操作？



### 5.5.7.内置用户密码

virsh 中并没有 set-user-password 这个原生命令，不过可以通过 QEMU Guest Agent 配合 qemu-agent-command 来实现修改虚拟机内部用户密码的功能（前提是虚拟机已安装并运行 qemu-guest-agent）。

实现步骤：

#### 1. 确保前提条件

虚拟机已安装 qemu-guest-agent 并启动服务：

```
bash
```

```
# 虚拟机内部执行（以 Ubuntu 为例）sudo apt install qemu-guest-agent -y
```

```
sudo systemctl enable --now qemu-guest-agent
```

虚拟机 XML 配置中已启用 agent 通道（默认通常已配置）。

#### 2. 通过 qemu-agent-command 修改密码

使用 guest-set-user-password 命令（JSON 格式），语法如下：

```
bash
```

```
virsh qemu-agent-command <虚拟机名称/UUID> '{
```

```
  "execute": "guest-set-user-password",
```

```
  "arguments": {
```

```
    "username": "<用户名>",
```

```
    "password": "<新密码>",
```

```
    "crypted": false
```

```
  }
```

```
}'
```

参数说明：

username：虚拟机内的目标用户名（如 root、ubuntu）。

password：要设置的新密码。

crypted：false 表示密码是明文（agent 会自动加密）；true 表示密码已加密（需提前用 crypt() 等工具处理）。

示例：修改 Ubuntu 虚拟机的 ubuntu 用户密码

```
bash
```

```
virsh qemu-agent-command ubuntuvm '{
  "execute": "guest-set-user-password",
  "arguments": {
    "username": "ubuntu",
    "password": "NewSecurePass123!",
    "crypted": false
  }
}'
```

如果执行成功，命令会返回空结果（无报错即表示成功）。

注意事项：

部分操作系统可能限制通过 **agent** 修改密码（如 **Windows** 需特定配置，**Linux** 某些版本可能要求密码复杂度）。

若虚拟机未安装 **agent**，需通过其他方式修改密码（如 **virsh console** 登录后修改、**SSH** 远程修改等）。

密码会以明文形式通过 **JSON** 传输，建议在安全环境中使用，或提前加密（**crypted: true**）。这种方式是 **virsh** 环境下修改虚拟机用户密码的推荐方法，无需直接登录虚拟机即可完成操作。

## 5.6.通道设备

```
<channel type='unix'>
  <target type='virtio' name='org.qemu.guest_agent.0'/>
  <address type='virtio-serial' controller='0' bus='0' port='1'/>
</channel>
<channel type='spicevmc'>
  <target type='virtio' name='com.redhat.spice.0'/>
  <address type='virtio-serial' controller='0' bus='0' port='2'/>
</channel>
```

通道设备（Channel Device）是一种实现 Host（物理机）与 Guest（虚拟机）之间高效通信的机制，它允许两者在不依赖网络的情况下进行数据交换，常用于传递配置信息、执行命令或实现特定服务交互。

### 一、通道设备的核心作用

通道设备本质上是 Host 与 Guest 之间的一个“私有通信链路”，主要优势在于：

低延迟：无需经过网络协议栈，直接通过虚拟化层通信。

安全性：通信数据不经过外部网络，仅在物理机与虚拟机内部流转。

便捷性：可在虚拟机启动 / 运行时动态传递数据，无需提前配置网络。

常见应用场景：

向虚拟机传递初始化配置（如 IP 地址、密钥）。

物理机远程控制虚拟机执行脚本（无需 SSH）。

虚拟机向物理机上报运行状态或日志。

### 二、通道设备的类型与配置

KVM 支持多种通道类型，最常用的是 unix 类型（基于 Unix 域套接字）和 spicevmc 类型（SPICE 协议专用），以下以 unix 类型为例说明配置方法。

#### 1. 在虚拟机 XML 中定义通道设备

通过 `virsh edit <虚拟机名>` 编辑配置，在 `<devices>` 标签内添加通道配置：

xml

```
<channel type='unix'>
  <!-- 物理机端的套接字路径 -->
  <source mode='bind' path='/var/lib/libvirt/qemu/channel/target/<虚拟机名>.channel' />
  <!-- 虚拟机端的设备标识 -->
  <target type='virtio' name='org.qemu.guest_agent.0' />
  <!-- 自动连接（虚拟机启动时自动激活通道） -->
  <address type='virtio-serial' controller='0' bus='0' port='1' /></channel>
```

#### 2. 关键参数解释

`<channel type='unix'>`:

`type='unix'`: 指定通道基于 Unix 域套接字（仅支持本地 Host 与 Guest 通信）。

`<source mode='bind' path='...' />`:

`mode='bind'`: Host 端创建并绑定到指定路径的套接字。

path: Host 端套接字文件的绝对路径, 需确保 libvirt 有权限读写 (通常放在 /var/lib/libvirt/qemu/channel/target/ 目录)。

<target type='virtio' name='org.qemu.guest\_agent.0'/>:

type='virtio': 使用 virtio 驱动 (高性能虚拟化驱动)。

name: 虚拟机内部识别通道的唯一名称, 通常遵循 org.qemu.guest\_agent.<N> 格式 (与 QEMU Guest Agent 配合使用时固定为此名称)。

<address .../>:

指定通道在虚拟机 virtio-serial 总线上的地址, controller='0'、bus='0'、port='1' 表示连接到第 0 个控制器、第 0 条总线、第 1 个端口 (避免与其他设备冲突)。

### 三、Guest 端配置 (以 Linux 为例)

安装 QEMU Guest Agent (通道通信的常用工具, 提供标准接口):

# CentOS/RHEL

yum install qemu-guest-agent -y# Ubuntu/Debianapt install qemu-guest-agent -y

启动并启用服务:

systemctl enable --now qemu-guest-agent

该服务会自动识别通道设备 (/dev/virtio-ports/org.qemu.guest\_agent.0), 并通过通道与 Host 通信。

### 四、Host 与 Guest 通信示例

通过 virsh qemu-agent-command 命令, Host 可向 Guest 发送指令 (基于 QEMU Guest Agent 接口):

获取 Guest 的 IP 地址:

virsh qemu-agent-command <虚拟机名> '{"execute": "guest-network-get-interfaces"}'  
--pretty

让 Guest 执行关机:

virsh qemu-agent-command <虚拟机名> '{"execute": "guest-shutdown"}'

传递文件到 Guest (需 Guest 端支持):

```
# Host 端创建文件 echo "test data" > /tmp/host-data.txt# 通过通道发送到 Guest 的  
/tmp/guest-data.txtvirsh qemu-agent-command <虚拟机名> '{"execute":"guest-file-write",  
"arguments":{"path":"/tmp/guest-data.txt", "content":"'$(base64 /tmp/host-data.txt)'"}}
```

## 五、注意事项

兼容性: 需确保 Guest 系统安装了 virtio-serial 驱动(Linux 通常默认支持, Windows 需手动安装 virtio 驱动)。

权限: Host 端套接字文件需设置正确权限(属主 root:libvirt, 权限 0660), 否则 Guest 可能无法连接。

安全性: 通道通信默认不加密, 敏感数据需在应用层自行加密。

通过通道设备, Host 与 Guest 可以实现轻量级、高效的交互, 尤其适合虚拟化管理平台(如 OpenStack、Proxmox)对虚拟机的精细化控制。

实际用途: 主机可通过该通道执行虚拟机内操作, 如:

读取虚拟机 IP 地址(无需登录)。

优雅关闭 / 重启虚拟机(即使无图形 / 串口登录)。

同步主机与虚拟机时间。

挂载 / 卸载虚拟机内的磁盘镜像。

## SPICE 通道(图形交互辅助)

```
<channel type='spicevmc'>
```

```
  <target type='virtio' name='com.redhat.spice.0' />
```

```
  <address type='virtio-serial' controller='0' bus='0' port='2' /></channel>
```

核心作用: 配合 SPICE 图形协议, 实现主机与虚拟机的高级图形交互功能(如剪贴板共享、文件传输)。

关键参数解析:

<channel type='spicevmc'>: 通道类型为 spicevmc (SPICE Virtual Machine Channel), 是 SPICE 协议的专用辅助通道。

<target type='virtio' name='com.redhat.spice.0' />:

name='com.redhat.spice.0': SPICE 协议的标准通道名称, 虚拟机内的 SPICE 客户端(如 spice-vdagent)会自动识别。

需虚拟机内安装 `spice-vdagent` 包（Ubuntu 24.10 可通过 `apt install spice-vdagent` 安装），否则无法使用高级功能。

`<address... port='2'/>`：关联 `virtio-serial` 控制器的第 2 个端口（与上一个通道的 `port='1'` 区分）。

实际用途：

主机与虚拟机之间双向剪贴板共享（复制主机文本粘贴到虚拟机，反之亦然）。

支持自动分辨率适配（调整 SPICE 客户端窗口大小时，虚拟机桌面分辨率同步变化）。

部分 SPICE 客户端（如 `virt-viewer`）可通过该通道传输文件。

## 5.7. 输入设备

配置了 3 类输入设备，覆盖图形界面和文本模式的交互需求：

USB 平板输入（图形界面精准定位）

```
<input type='tablet' bus='usb'>
  <address type='usb' bus='0' port='1'/>
</input>
```

核心作用：用于图形界面下的鼠标精准定位，解决传统 PS/2 鼠标的“偏移问题”。

关键参数解析：

`<input type='tablet' bus='usb'>`：

`type='tablet'`：模拟“平板输入设备”，能直接映射主机鼠标坐标到虚拟机，无分辨率适配偏差（如虚拟机窗口缩放时，鼠标点击位置与显示位置完全一致）。

`bus='usb'`：通过 USB 总线连接（依赖配置中的 `<controller type='usb' index='0' model='qemu-xhci'>` USB 控制器）。

`<address type='usb' bus='0' port='1'/>`：关联 USB 控制器的第 0 条总线、第 1 个端口（端口号唯一）。

必要性：若仅用 PS/2 鼠标，在 SPICE/VNC 客户端缩放窗口时，鼠标点击会出现“偏移”（点击位置与实际触发位置不一致），`tablet` 类型输入设备可完全避免该问题，是图形界面的必需配置。

PS/2 鼠标与键盘（兼容性保障）

```
<input type='mouse' bus='ps2'/>
```

```
<input type='keyboard' bus='ps2'/>
```

核心作用：提供 PS/2 总线的鼠标和键盘，保障兼容性（如虚拟机启动阶段、无 USB 驱动的系统、文本模式）。

关键参数解析：

**type='mouse'/'keyboard'**：设备类型为鼠标 / 键盘。

**bus='ps2'**：通过 PS/2 总线连接（无需额外地址配置，PS/2 是传统低速总线，兼容性极强，所有 x86 系统均支持）。

必要性：

USB 设备依赖虚拟机内的 USB 驱动，若系统未安装驱动（如早期 Linux 版本、Windows 安装阶段），USB 鼠标 / 键盘无法使用，PS/2 设备可正常工作。

文本模式（如 Ubuntu 命令行界面）下，PS/2 设备响应更稳定。

## 5.8.图形显示设备

### 5.8.1.consoleVNC 协议

```
<graphics type="vnc" port="5906" autoport="no" listen="0.0.0.0">
  <listen type="address" address="0.0.0.0"/>
</graphics>
```

### 5.8.2.SPICE 协议

```
<graphics type="spice" port="5906" autoport="no" listen="0.0.0.0">
  <listen type="address" address="0.0.0.0"/>
</graphics>
```

SPICE（Simple Protocol for Independent Computing Environments）是一种高性能的远程桌面协议，专为虚拟化场景设计，支持高清显示、音频传输和 USB 重定向等功能。

一、字段与参数详解

```
<graphics type="spice" port="5906" autoport="no" listen="0.0.0.0">
  <listen type="address" address="0.0.0.0"/>
</graphics>
```

## 1. <graphics> 根标签

**type="spice":** 指定图形显示协议为 SPICE（必填），其他可选协议有 vnc、sdl 等，但 SPICE 在画质和性能上更优。

**port="5906":** 指定 SPICE 服务监听的端口号（5900-5999 是默认的 VNC/SPICE 端口段）。

若 **autoport="yes"**，此参数会被忽略，系统会自动分配一个空闲端口。

**autoport="no":** 是否自动分配端口。

**no:** 使用 port 指定的固定端口（适合需要固定连接地址的场景）。

**yes:** 自动从 5900 开始分配空闲端口（避免端口冲突，但连接地址不固定）。

**listen="0.0.0.0":** 指定 SPICE 服务监听的 IP 地址（全局配置）。

**0.0.0.0:** 监听物理机所有网络接口（允许远程主机通过任意 IP 连接）。

若指定具体 IP（如 192.168.1.100），则仅允许通过该 IP 连接（增强安全性）。

## 2. <listen type="address" address="0.0.0.0"/>

这是对 listen 参数的补充配置（优先级高于 <graphics> 标签中的 listen），用于更精确地指定监听地址。

**type="address":** 表示监听类型为具体 IP 地址（固定写法）。

**address="0.0.0.0":** 与外层 listen 含义一致，指定监听所有网络接口。

若内外层 listen 冲突，内层配置优先生效（例如外层 listen="127.0.0.1"，内层 address="192.168.1.100"，则实际监听 192.168.1.100）。

## 二、SPICE 显示效果调优配置

SPICE 的默认配置可能无法发挥最佳性能，可通过添加以下参数优化显示效果（尤其是高清分辨率、视频流畅度和色彩表现）：

### 1. 基础性能优化

在 <graphics> 标签中添加以下参数：

**<graphics type="spice" port="5906" autoport="no" listen="0.0.0.0"**

**passwd="your\_password" <!-- 连接密码（可选，增强安全性） -->**

**keymap="en-us" <!-- 键盘布局（如 "zh-cn" 对应中文） -->**

**sharePolicy="allow-exclusive" <!-- 允许多用户连接时独占输入 -->**

**defaultMode="desktop" <!-- 默认显示模式（desktop/fullscreen） -->**

**clipboard="both" <!-- 剪贴板共享**

**(both/guest-to-host/host-to-guest/none) -->**



```

        filetransfer="on">        <!-- 允许文件传输（需客户端支持） -->

<listen type="address" address="0.0.0.0"/>

<!-- 图像压缩与编码优化 -->

<image compression="auto_glz"        <!-- 图像压缩算法
(auto_glz/auto_lz4/glz/lz4/quic) -->

        jpeg_quality="90"        <!-- JPEG 压缩质量（1-100，越高画质越好） -->
        zlib_level="3"/>        <!-- zlib 压缩级别（1-9，越高压缩率越高但耗
CPU） -->

<!-- 视频流优化（适合播放视频） -->

<streaming mode="all"        <!-- 视频流模式（all/filter/none，all 启用所有
视频优化） -->

        debayer="fast"/>        <!-- 视频降噪算法（fast/quality，平衡速度与
画质） -->

<!-- USB 重定向（可选，允许虚拟机使用物理机 USB 设备） -->

<usb_redirection enabled="yes"/></graphics>

```

## 2. 关键调优参数说明

| 参数             | 作用                     | 推荐值                    |
|----------------|------------------------|------------------------|
| compression    | 图像压缩算法，影响传输效率          | auto_glz（自动选择，平衡速度与画质） |
|                | 和画质                    | 或 lz4（速度优先，适合低带宽）      |
| jpeg_quality   | 静态图像 JPEG 压缩质量         | 90（兼顾画质和传输量，100 为无损）   |
|                | 量                      |                        |
| zlib_level     | 文本 / UI 元素的 zlib 压缩级别  | 3（级别过高会增加 CPU 消耗）      |
|                |                        |                        |
| streaming mode | 视频流优化，对动态画面（如视频播放）效果显著 | all（启用所有视频优化）          |
|                |                        |                        |
| clipboard      | 剪贴板共享，提升交互便捷性          | both（双向共享）             |
|                |                        |                        |
| filetransfer   | 文件传输功能（需 SPICE 客户端支持）  | on（开启，方便文件交换）          |
|                |                        |                        |

## 3. 虚拟机内部配置配合

安装 SPICE 驱动（Linux 通常默认支持，Windows 需手动安装）：

Windows：下载 VirtIO 驱动包，安装 spice-guest-tools（包含显卡、鼠标、剪贴板驱动）。

调整虚拟机显示分辨率：

Linux：通过桌面环境设置分辨率（如 1920x1080）。

Windows：在“显示设置”中设置与客户端匹配的分辨率。

#### 4. 客户端配置优化

使用 virt-viewer（推荐）或 remote-viewer 连接 SPICE 服务时，可在客户端设置中优化：

启用“自适应质量”（Adaptive Quality）：根据网络状况自动调整画质。

开启“硬件加速”：利用显卡加速图像渲染（需客户端支持）。

调整“带宽限制”：若网络带宽有限，可限制最大传输速率（如 10Mbps）。

### 三、安全与兼容性注意事项

端口与密码：

若 listen="0.0.0.0"（允许外部访问），务必设置 passwd 密码，避免未授权访问。

建议使用防火墙限制端口访问（如 firewall-cmd --add-port=5906/tcp --permanent，仅允许信任的 IP 段连接）。

兼容性：

SPICE 对网络质量较敏感，建议在局域网（LAN）中使用（带宽 ≥100Mbps）。

老旧操作系统（如 Windows XP）可能不支持高分辨率和视频流优化。

通过以上配置，SPICE 可提供接近本地桌面的显示效果，尤其适合需要高清画质、流畅视频播放的场景（如图形化管理、多媒体应用）。

```
<graphics type='spice' autoport='yes'>
```

```
<listen type='address'/></graphics>
```

核心作用：启用 SPICE 图形协议（Simple Protocol for Independent Computing Environments），提供高性能的图形远程访问（优于 VNC，适合桌面级虚拟机）。

关键参数解析：

```
<graphics type='spice' autoport='yes'>:
```

type='spice'：图形协议类型为 SPICE（Red Hat 开发，专为虚拟化优化，支持 3D 加速、视频流、低带宽适配）。

`autoport='yes'`: 自动分配端口（避免手动指定端口冲突），libvirt 会从 5900 开始扫描未占用端口，分配后可通过 `virsh vncdisplay ubuntu` 查看（虽命令含 `vnc`，但 SPICE 端口也会显示，格式如 `:4` 表示端口 5904）。

`<listen type='address'/>`: 默认监听 主机所有网卡的 IP（即 0.0.0.0），允许局域网内其他设备通过 SPICE 客户端连接（若需限制访问，可添加 `address='192.168.1.100'` 指定仅监听主机特定 IP）。

使用方式：需通过 SPICE 客户端连接，如：

Linux: `virt-viewer ubuntu`（直接通过虚拟机名连接，自动识别端口）、`spicy` 主机 IP:端口。

Windows: `virt-viewer.exe`（需下载安装，支持图形界面操作）。

优势对比 VNC:

支持 动态分辨率调整（窗口缩放时桌面同步变化）。

支持 3D 加速（需配合 `qxl` 显卡，适合运行图形密集型应用）。

带宽占用更低（视频流压缩优化，适合远程访问）。

## 5.9.声音设备

```
<sound model='ich9'>
```

```
  <address type='pci' domain='0x0000' bus='0x00' slot='0x1b' function='0x0'/>
```

```
</sound>
```

核心作用：模拟声卡，实现虚拟机内的声音输出（如音乐、系统提示音）。

关键参数解析：

`<sound model='ich9'>`: 模拟 Intel ICH9 芯片组的声卡（ICH9 是主流的桌面级声卡芯片，兼容性强，Ubuntu、Windows 等系统均默认支持驱动）。

其他常见型号：`ac97`（老旧声卡，兼容性一般）、`hda`（高清声卡，支持多声道，但部分系统驱动适配差），`ich9` 是平衡兼容性和性能的首选。

`<address type='pci'... bus='0x00' slot='0x1b' function='0x0'/>`: 声卡在 PCI 总线上的地址：

`bus='0x00'`: 关联第 0 条 PCI 总线（对应 `pcie-root` 根控制器）。

`slot='0x1b'`: 总线的第 27 个插槽（`0x1b` 是 ICH9 声卡的标准插槽地址，符合硬件规范）。

使用注意：需虚拟机内安装声卡驱动（Ubuntu 24.10 已默认集成，Windows 会自动识别并安装驱动），且 SPICE/VNC 客户端需开启声音转发（如 virt-viewer 默认开启）。

## 5.10.显卡视频设备

5.10.1.virtio 显卡模拟

```
<video>

  <model type="virtio" heads="1" primary="yes">

    <acceleration accel3d="no"/>

  </model>

  <address type="pci" domain="0x0000" bus="0x00" slot="0x01" function="0x0"/>

</video>
```

1. <model> 标签核心参数

| 参数      | 可选值 / 含义                                                                                 | 你的配置说明                                           |
|---------|------------------------------------------------------------------------------------------|--------------------------------------------------|
|         | - virtio: 高性能半虚拟化显卡（推荐）                                                                  |                                                  |
| type    | - qxl: 兼容型 2D 加速显卡 - vga: 传统 VGA（性能差，仅兼容旧系统） - cirrus: 拟显卡类型，需配合 virtio 驱动使用。老旧显卡模型（已淘汰） | 配置为 virtio，属于性能最优的虚拟显卡（性能差，仅兼容旧系统）。              |
| heads   | 正整数（如 1、2），表示虚拟显示器数量                                                                     | 配置为 1，即只启用 1 个虚拟显示器。                             |
| primary | yes/no，是否设为虚拟主机的主显卡                                                                      | 配置为 yes，表示此显卡为虚拟机默认输出设备（系统启动画面、默认显示器）。           |
| vram    | 显存大小（单位 KB，如 16384 = 16MB），未显式配置时取默认值（通常 16MB，建议根据需求手动指定 16MB）                           | 未配置时，virtio 显卡默认分配（如 vram="65536" = 64MB）。       |
| vgamem  | 显卡专用内存（单位 KB），用于 2D/3D 渲染缓存                                                              | 未配置时默认较小（如 4MB），高负载场景需加大（如 vgamem="8192" = 8MB）。 |

2. <acceleration> 子标签

| 参数      | 可选值 / 含义          | 你的配置说明                                                   |
|---------|-------------------|----------------------------------------------------------|
| accel3d | yes/no，是否启用 3D 加速 | 配置为 no，即关闭 3D 加速。若需运行 3D 应用（如 CAD、轻度游戏），需改为 yes 并安装对应驱动。 |

3. <address> 标签（PCI 地址配置）

| 参数       | 含义                                    | 你的配置说明                         |
|----------|---------------------------------------|--------------------------------|
| domain   | PCI 域（通常为 0x0000，单主机场景固定）             | 标准配置 0x0000，无需修改。              |
| bus      | PCI 总线号（0x00~0xFF），虚拟机内的虚拟总线          | 配置为 0x00，即连接到虚拟机的第 0 条 PCI 总线。 |
| slot     | 插槽号（0x00~0x1F），虚拟总线上的位置               | 配置为 0x01，表示在第 0 条总线上的第 1 个插槽。  |
| function | 功能号（0x00~0x07），单设备多功能时区分（如显卡 + 音频控制器） | 配置为 0x0，表示此为显卡的主功能。            |

针对 virtio 显卡，通过以下配置进一步提升性能：

### 1. 基础优化（必做）

加大显存与内存：编辑 XML，在 <model> 标签中添加：

```
<model type="virtio" heads="1" primary="yes" vram="65536" vgamem="8192">
```

vram=65536（64MB）：满足 1080P 分辨率需求，4K 需设为 262144（256MB）。

vgamem=8192（8MB）：提升 2D 渲染流畅度（如窗口拖动、视频播放）。

启用 3D 加速：将 <acceleration accel3d="no"/> 改为 yes，并安装对应驱动：

Windows 虚拟机：安装 virtio-win 驱动（含 virtio-gpu 3D 驱动）。

Linux 虚拟机：安装 virtio-gpu-drivers 和 spice-vdagent，支持 3D 硬件加速。

使用 SPICE 协议：配合 virt-viewer 客户端，启用 SPICE 协议（比 VNC 更流畅），在虚拟机 XML 中添加：

```
<graphics type="spice" port="-1" autoport="yes">
```

```
<listen type="address"/>
```

```
<image compression="auto_glz"/>
```

```
<streaming mode="filter"/></graphics>
```

### 2. 高级优化（按需配置）

启用 VRAM 气球技术：允许虚拟机动态调整显存（需宿主机支持）：

```
<model type="virtio" ...>
```

```
<balloon vramsize="32768"/> <!-- 最大可动态扩展到 32MB --></model>
```

关闭不必要的显示功能：如无多显示器需求，heads=1 即可（多显示器会占用额外资源）。

### 5.10.2.qxl 显卡模拟

```
<video>

  <model type="qxl" ram="65536" vram="65536" vgamem="16384" heads="1"
primary="yes"/>

  <address type="pci" domain="0x0000" bus="0x00" slot="0x01" function="0x0"/>
</video>
```

其中：

ram: qxl 设备的专用内存（单位 KB，65536=64MB）

vram: 视频显存（单位 KB，65536=64MB）

vgamem: 显卡驱动使用的内存（单位 KB，16384=16MB）

根据物理机配置优化参数

1. 物理机内存较小（ $\leq 16\text{GB}$ ）：优先控制资源占用

若物理机内存紧张（如用于轻量办公、服务器场景）：

降低 ram 和 vgamem，避免占用过多宿主机内存：

```
<model type="qxl" ram="32768" vram="32768" vgamem="8192" heads="1"
primary="yes"/>
```

适用场景：虚拟机分辨率 $\leq 1080\text{P}$ 、仅运行浏览器 / 文档编辑（无图形渲染）。

理由：qxl 的 ram 主要用于驱动缓存，低负载场景无需过大。

2. 物理机内存充足（ $\geq 32\text{GB}$ ）：提升图形性能

若物理机内存充裕（如用于设计、多任务）：

增加 vram 和 vgamem，支持更高分辨率和图形负载：

```
<model type="qxl" ram="131072" vram="131072" vgamem="32768" heads="1"
primary="yes"/>
```

适用场景：虚拟机分辨率 2K/4K、运行轻度设计软件（如 GIMP、LibreOffice Draw）。

理由：vram 直接影响最大分辨率（4K 需至少 128MB），vgamem 提升 2D 渲染流畅度（如窗口拖动、图表绘制）。

3. 物理机 CPU 性能较强（多核高频，如 $\geq 8$  核 16 线程）：启用多显示器

若物理机 CPU 性能较好（如用于多任务办公）：

增加 heads 数量（最多支持 4 个），但需同步提升资源：

```
<model type="qxl" ram="131072" vram="131072" vgamem="32768" heads="2"
primary="yes"/>
```

注意：多显示器（heads>1）会增加 CPU 负载，需物理机 CPU 有足够冗余。

#### 4. 物理机有独立显卡（但不做直通）：配合 SPICE 协议优化

若物理机有中高端独立显卡（如 RTX 3050/AMD RX 6500），即使不直通，也可通过协议优化提升体验：

在 XML 中添加 SPICE 图形配置（与<video>同级，位于<devices>内）：

```
<graphics type="spice" port="-1" autoport="yes">
  <listen type="address"/>
  <image compression="auto_glz"/> <!-- 自动压缩图像，平衡画质和速度 -->
  <streaming mode="all"/> <!-- 启用视频流优化，适合视频播放 -->
  <clipboard copypaste="yes"/> <!-- 支持剪贴板共享 --></graphics>
```

配合 virt-viewer 客户端连接，利用物理机显卡的解码能力加速图像渲染。

### 三、特殊场景优化

#### 1. 虚拟机运行 Linux 系统：启用动态分辨率

Linux 对 qxl 支持更好，可添加 vgamem 并安装工具：

xml

```
<model type="qxl" ram="65536" vram="65536" vgamem="16384" heads="1"
primary="yes"/>
```

虚拟机内安装：sudo apt install spice-vdagent（Debian/Ubuntu）或 sudo dnf install

spice-vdagent（Fedora），支持窗口大小自适应。

#### 2. 虚拟机运行 Windows 系统：限制资源 + 安装驱动

Windows 对 qxl 的兼容性略差，建议：

保持 vram≤128MB（避免驱动异常），并安装专用驱动：

```
x<model type="qxl" ram="65536" vram="131072" vgamem="16384" heads="1"
primary="yes"/>
```

下载并安装 virtio-win 驱动中的 qxl 驱动，提升 2D 性能。

### 四、避坑提醒

qxl 不支持 3D 加速：若需运行 3D 应用（如 CAD、轻度游戏），建议换成 virtio 显卡并开启 accel3d="yes"。



避免参数过大：ram 和 vram 总和不宜超过物理机内存的 10%（如 32GB 物理机，最大分配 3GB），否则会导致宿主机内存不足。

多虚拟机场景：若物理机运行多个虚拟机，需分摊资源（如总 vram 之和 $\leq$ 物理机显存的 50%）。

### 5.10.3.显卡直通

显卡直通（PCIe 直通）能让虚拟机直接使用物理显卡，实现接近原生的性能，适合游戏、渲染等场景。以下是详细配置步骤：

#### 一、前提条件

硬件支持：

CPU：Intel 需支持 VT-d，AMD 需支持 AMD-Vi（在 BIOS 中开启）。

主板：需在 BIOS 中开启“PCIe 直通”“SR-IOV”“ACS 支持”（不同主板选项名称可能不同）。

显卡：

AMD 显卡：大部分型号原生支持（如 RX 6000/7000 系列）。

NVIDIA 显卡：专业卡（如 Quadro、Tesla）原生支持；消费级卡（如 RTX 30/40 系列）需关闭 Secure Boot 或刷写修改版 VBIOS。

软件环境：

宿主机系统：推荐 Linux（如 Ubuntu Server、CentOS），KVM 已正确安装（libvirt、qemu-kvm）。

#### 二、配置步骤

##### 1. 启用宿主机内核支持

编辑 grub 配置，开启 IOMMU 支持：

```
bash
```

```
# 编辑 grub 配置文件（不同系统路径可能不同）sudo vim /etc/default/grub
```

找到 GRUB\_CMDLINE\_LINUX 行，添加参数：

Intel 处理器：intel\_iommu=on iommu=pt

AMD 处理器：amd\_iommu=on iommu=pt

示例：

```
bash
```

```
GRUB_CMDLINE_LINUX="quiet splash intel_iommu=on iommu=pt" # Intel 示例
```

更新 grub 并重启:

```
bash
```

```
sudo update-grub # Ubuntu/Debian# 或 sudo grub2-mkconfig -o /boot/grub2/grub.cfg
```

(CentOS/RHEL) sudo reboot

重启后验证 IOMMU 是否启用:

```
bash
```

```
dmesg | grep -i iommu# 出现 "IOMMU enabled" 即为成功
```

## 2. 识别显卡 PCI 信息

查看显卡的 PCI 地址和设备 ID:

```
bash
```

```
# 列出所有 PCI 设备，筛选显卡
```

```
lspci | grep -iE "vga|3d|display"
```

输出示例 (需记录显卡的 bus:slot.function 和设备 ID):

```
plaintext
```

```
01:00.0 VGA compatible controller: NVIDIA Corporation GA104 [GeForce RTX 3070] (rev a1)
```

```
01:00.1 Audio device: NVIDIA Corporation GA104 High Definition Audio Controller (rev a1)
```

显卡主设备: 01:00.0 (后续配置用)

音频控制器 (若有): 01:00.1 (需同时直通, 否则可能无声音)

获取设备 ID (用于绑定驱动):

```
bash
```

```
lspci -n -s 01:00.0 # 替换为你的显卡 PCI 地址
```

输出示例 (10de:2484 为 VendorID:DeviceID):

```
plaintext
```

```
01:00.0 0300: 10de:2484 (rev a1)
```

## 3. 绑定显卡到 vfio-pci 驱动

让宿主机放弃对显卡的控制, 交由 vfio 驱动管理:

```
bash
```

```
# 创建驱动绑定配置 sudo vim /etc/modprobe.d/vfio.conf
```

添加显卡设备 ID（多个设备用逗号分隔）：

```
bash
```

```
options vfio-pci ids=10de:2484,10de:228b # 替换为你的显卡和音频控制器 ID
```

阻止宿主机加载显卡原生驱动：

```
bash
```

```
sudo vim /etc/modprobe.d/blacklist.conf
```

添加（根据显卡类型选择）：

```
bash
```

```
# NVIDIA 显卡
```

```
blacklist nouveau
```

```
blacklist nvidia
```

```
blacklist nvidia-modeset
```

```
blacklist nvidia-uvm
```

```
blacklist nvidia-drm
```

```
# AMD 显卡
```

```
blacklist radeon
```

```
blacklist amdgpu
```

更新 initramfs 并重启：

```
bash
```

```
sudo update-initramfs -u -k all # Ubuntu/Debian# 或 sudo dracut -f (CentOS/RHEL) sudo
```

reboot

验证驱动是否绑定成功：

```
bash
```

```
lspci -k -s 01:00.0 # 替换为你的显卡 PCI 地址
```

输出中出现 Kernel driver in use: vfio-pci 即为成功。

#### 4. 修改虚拟机 XML 配置

通过 virsh edit 编辑虚拟机配置，替换原虚拟显卡配置：

删除原虚拟显卡：找到并删除 <video> 标签段（如 qxl 或 virtio 显卡）。

添加显卡直通配置：在 <devices> 标签内添加以下内容（替换为你的 PCI 地址）：

xml

```
<!-- 显卡主设备直通 --><hostdev mode="subsystem" type="pci" managed="yes">
```

```
<source>
```

```
<address domain="0x0000" bus="0x01" slot="0x00" function="0x0"/> <!-- 替换为显
```

卡 PCI 地址 -->

```
</source>
```

```
<address type="pci" domain="0x0000" bus="0x06" slot="0x00" function="0x0"/> <!--
```

虚拟机内地址，可自定义 --></hostdev>

```
<!-- 音频控制器直通（若有） --><hostdev mode="subsystem" type="pci" managed="yes">
```

```
<source>
```

```
<address domain="0x0000" bus="0x01" slot="0x00" function="0x1"/> <!-- 替换为音
```

频控制器 PCI 地址 -->

```
</source>
```

```
<address type="pci" domain="0x0000" bus="0x07" slot="0x00"
```

function="0x0"/></hostdev>

（NVIDIA 消费级卡）添加防检测配置：在 <features> 标签内添加，避免驱动报错（代码 43）：

xml

```
<features>
```

```
...
```

```
<kvm>
```

```
<hidden state="on"/> <!-- 隐藏虚拟化环境 -->
```

```
</kvm></features>
```

## 5. 启动虚拟机并验证

bash

```
# 启动虚拟机 virsh start 虚拟机名称
```

```
# 连接虚拟机（如通过 VNC 或远程桌面）
```

在虚拟机内：

**Windows:** 打开 “设备管理器”，确认显卡已识别（无黄色感叹号），安装官方驱动。

**Linux:** 执行 `nvidia-smi`（NVIDIA）或 `lspci -k` 确认驱动加载。

三、常见问题解决

虚拟机启动失败：

检查 BIOS 中 VT-d/AMD-Vi 是否开启。

确认显卡驱动已绑定到 `vfio-pci`（`lspci -k` 检查）。

**NVIDIA** 显卡代码 43：

关闭宿主机和虚拟机的 Secure Boot。

确保 XML 中添加了 `<hidden state="on"/>`。

刷写修改版 VBIOS（针对部分消费级卡，需搜索对应型号教程）。

显卡性能不佳：

检查 PCIe 带宽：`lspci -vv | grep -i "LnkSta:"`，确保为 x16 或 x8。

关闭宿主机不必要的服务，减少资源占用。

按照以上步骤，即可完成显卡直通配置。若遇到具体型号（如 RTX 4090、RX 7900 XTX）的问题，可以告诉我，提供针对性解决方案。

5.11.USB 重定向

```
<redirdev bus='usb' type='spicevmc'>
  <address type='usb' bus='0' port='2' />
</redirdev>
```

1. 顶层标签：<redirdev>

这是整个 USB 重定向配置的根标签，用于声明 “重定向设备” 的类型和基础协议。

| 参数              | 取值       | 作用详解                                            |
|-----------------|----------|-------------------------------------------------|
| bus='usb'       | 固定为      | 明确当前重定向的设备总线类型是 USB，告诉虚拟机                       |
|                 | usb      | 这是一个 USB 类设备。                                   |
| type='spicevmc' | 固定为      | 指定重定向使用的协议是 spicevmc，即 SPICE 虚拟机                |
|                 | spicevmc | 通道（SPICE VM Channel），这是 SPICE 协议实现 USB 共享的专用通道。 |

2. 子标签: <address>

用于定义这个虚拟 USB 设备在虚拟机内部的“硬件地址”，确保虚拟机操作系统能识别并与它通信，类似物理设备的“端口号”。

| 参数         | 取值示           | 作用详解                                                                                                                          |
|------------|---------------|-------------------------------------------------------------------------------------------------------------------------------|
|            | 例             |                                                                                                                               |
| type='usb' | 固定为 usb       | 声明该地址的类型是 USB 总线地址，与上层 bus='usb'对应，确保地址格式匹配。                                                                                  |
| bus='0'    | 非负整数（如 0、1）   | 指定该虚拟 USB 设备挂载在虚拟机内部的哪条 USB 总线上。- 通常 bus='0'表示虚拟机的“主 USB 总线”，是默认且最常用的选择。- 若虚拟机有多个 USB 控制器（如 USB 2.0 和 USB 3.0），会对应不同的 bus 编号。 |
| port='2'   | 非负整数（如 1、2、3） | 指定该虚拟 USB 设备在对应总线上的“端口号”。- 每个 bus 下的 port 编号唯一，用于区分同一总线上的不同设备。- 取值只要不与虚拟机内其他 USB 设备的 port 冲突即可，常见从 1 开始递增。                    |

配置的实际作用流程

虚拟机启动时，QEMU 会根据<redirdev>标签创建一个基于 SPICE 协议的虚拟 USB 重定向设备。

该设备通过<address>标签的 bus 和 port 参数，在虚拟机内部“注册”一个唯一的硬件地址，让虚拟机 OS（如 Windows、Linux）能检测到它。

当你在 SPICE 客户端（如 virt-viewer）中选择“共享本地 USB 设备”时，物理机的 USB 设备数据会通过 spicevmc 通道，传输到这个虚拟 USB 设备上，最终被虚拟机识别为“本地 USB 设备”。

使用方式：

虚拟机运行时，通过 SPICE 客户端（如 virt-viewer ubuntu）连接。

在客户端菜单中选择“USB 设备”，勾选需重定向的主机 USB 设备（如 U 盘）。

虚拟机内会自动识别该 USB 设备，可直接读写或使用。

注意：重定向的 USB 设备在主机端会暂时“不可用”，断开虚拟机连接后恢复主机使用。

## 5.12.内存气球设备

```
<memballoon model='virtio'>
  <address type='pci' domain='0x0000' bus='0x05' slot='0x00' function='0x0' />
</memballoon>
```

核心作用：实现虚拟机的动态内存调整（即“内存气球”技术），在主机内存紧张时回收虚拟机空闲内存，分配给其他需要的虚拟机 / 进程。

关键参数解析：

**model='virtio'**: 使用 virtio 总线的内存气球驱动（高性能，需虚拟机内安装驱动，Ubuntu 24.10 已默认集成 virtio-balloon 模块）。

若禁用该设备，虚拟机内存会被“独占”（即使空闲也无法回收，浪费主机资源）。

**<address... bus='0x05' slot='0x00' />**: 关联 PCI 总线的第 5 条总线、第 0 个插槽（对应配置中的 **<controller type='pci' index='5' model='pcie-root-port'>**）。

工作原理：

主机内存不足时，libvirt 向虚拟机发送指令，触发“气球膨胀”。

虚拟机内的 virtio-balloon 驱动申请空闲内存，并标记为“不可用”，将其“还给”主机。

主机内存充足时，触发“气球收缩”，将内存重新分配给虚拟机。

实际用途：在多虚拟机环境中，避免单虚拟机占用过多空闲内存，提升主机内存利用率（如虚拟机配置 2GiB 内存，实际仅用 500MiB，气球可回收 1.5GiB 给其他虚拟机）。

## 5.13.随机数设备

```
<rng model='virtio'>
  <backend model='random'>/dev/urandom</backend>
  <address type='pci' domain='0x0000' bus='0x06' slot='0x00' function='0x0' /></rng>
```

核心作用：为虚拟机提供高质量随机数，解决虚拟机“随机数不足”问题（虚拟机无法直接访问主机硬件随机数生成器，易导致依赖随机数的应用卡顿）。

关键参数解析：

**model='virtio'**: 使用 virtio 总线的随机数设备（高性能，需虚拟机内安装 virtio-rng 驱动，Ubuntu 24.10 已默认集成）。

backend model='random': 指定随机数来源为“主机随机数设备”。

## 6.虚拟机日常管理

### 6.1.虚拟机登录

#### 6.1.1.虚拟机图形化登录

remote-viewer spice://192.168.3.60:5906

remote-viewer vnc://192.168.3.60:5905

spice 协议显示效果好，vnc 协议兼容性相对好。

#### 6.1.2.虚拟机 console 登录

virsh console xxx

需要虚拟机打开串口登录相关内容，见设备管理（串口与控制台）相关内容。

#### 6.1.3.客户端安全登录

### 6.2.虚拟机在线配置

虚拟机配置的非常多的命令都有 live 选项，就是在线调整配置，重启后不保存，详见具体章节。

### 6.3.虚拟机信息查询与监控

| 内容        | 命令             | 说明                         |
|-----------|----------------|----------------------------|
| 虚拟机全局信息   | virsh list     | 物理机上所有虚拟机的状态信息             |
| 虚拟机基本信息   | virsh dominfo  | 包括虚拟机 ID、UUID，虚拟机规格等信息。    |
| 虚拟机运行状态信息 | virsh domstate | 可以使用--reason 选项查询虚拟机变为当前状态 |



| 内容            | 命令                  | 说明                       |
|---------------|---------------------|--------------------------|
| 虚拟机全局信息       | virsh list          | 物理机上所有虚拟机的状态信息           |
|               |                     | 的原因。                     |
| 虚拟机统计信息       | virsh domstats      |                          |
| 虚拟机内存统计信息     | virsh dommemstat    | 查看虚拟机内存统计信息              |
| 虚拟机 CPU 统计信息  | virsh cpu-stats     |                          |
| 虚拟机 VCPU 调度信息 | virsh schedinfo     | 包括 vCPU 份额等信息            |
| 虚拟机 vCPU 数目信息 | virsh vcpucount     | 查询虚拟机 vCPU 的个数           |
| 虚拟机 vCPU 负载信息 | virsh vcpuinfo      | 看每个 vCPU 的详细信息（负载 / 状态等） |
| 虚拟机 I/O 线程信息  | virsh iothreadinfo  | 查询虚拟机 I/O 线程及其 CPU 亲和性信息 |
| 虚拟机虚拟块设备列表信息  | virsh domblklist    | 虚拟机虚拟块设备列表信息             |
| 虚拟机块设备信息      | virsh domblkinfo    | 虚拟机块设备信息                 |
| 虚拟机块设备统计信息    | virsh domblkstat    | 虚拟机块设备统计信息               |
| 虚拟机块设备错误信息    | virsh domblkerror   | 虚拟机块设备中显示错误              |
| 虚拟机控制接口状态信息   | virsh domcontrol    | 控制接口状态信息                 |
| 虚拟机接口链接状态信息   | virsh domif-getlink | 接口链接状态信息                 |
| 虚拟机网络接口列表信息   | virsh domiflist     | 网络接口列表信息                 |

| 内容          | 命令                | 说明                                |
|-------------|-------------------|-----------------------------------|
| 虚拟机全局信息     | virsh list        | 物理机上所有虚拟机的状态信息                    |
| 虚拟机网络接口运行信息 | virsh domifaddr   | 网络接口运行信息                          |
| 虚拟机网络接口统计信息 | virsh domifstat   | 查询网络接口统计信息                        |
| 虚拟机名称信息     | virsh domname     | 通过 uuid 查询虚拟机名称                   |
| 虚拟机 UUID 信息 | virsh domuuid     | 查询虚拟机的 uuid                       |
| 虚拟机 ID 信息   | virsh domid       | 查询活动的虚拟机 id                       |
| 虚拟机时间信息     | virsh time        | 查询虚拟机时间                           |
| 虚拟机图形接口信息   | virsh domdisplay  | 查询虚拟机的登录地址                        |
| 虚拟机操作系统信息   | virsh guestinfo   | 查询有关客户机的信息（需 guest-agent 支持）      |
| 虚拟机操作系统主机名  | virsh domhostname | 获取虚拟机的 hostname（需 guest-agent 支持） |

### 6.3.1.虚拟机全局信息

virsh list

你提供的是 virsh list 命令的可选参数列表，这些参数用于过滤和显示不同状态的虚拟机。

以下是主要参数的说明：

--all: 显示所有虚拟机（运行中、已关闭、暂停等所有状态）

--inactive: 只显示非活跃状态的虚拟机（已关闭的）

--state-running: 只显示正在运行的虚拟机

--state-shutoff: 只显示已关闭的虚拟机

--state-paused: 只显示暂停状态的虚拟机

--name: 只显示虚拟机名称  
--uuid: 只显示虚拟机的 UUID  
--table: 以表格形式显示（默认格式）

常用示例:

查看所有虚拟机: `virsh list --all`

查看正在运行的虚拟机: `virsh list --state-running`

查看已关闭的虚拟机: `virsh list --state-shutoff`

这些参数可以帮助你快速筛选出需要查看的虚拟机状态信息。

### 6.3.2.虚拟机基本信息

`virsh dominfo`

用于查看虚拟机（domain）详细信息的命令，能够显示虚拟机的基本配置、状态、资源分配等关键信息。

基本用法:

`bash`

`virsh dominfo <虚拟机名称或 UUID>`

示例输出及字段说明:

以名为 `ubuntu24` 的虚拟机为例，运行 `virsh dominfo ubuntu24` 可能会显示类似以下内容:

plaintext

|              |                                      |
|--------------|--------------------------------------|
| Id:          | 5                                    |
| Name:        | ubuntu24                             |
| UUID:        | 1b9d6bcd-bbfd-4b2d-9b5d-ab8dfbbd4bed |
| OS Type:     | hvm                                  |
| State:       | running                              |
| CPU(s):      | 2                                    |
| CPU time:    | 1234.567s                            |
| Max memory:  | 4194304 KiB                          |
| Used memory: | 4194304 KiB                          |
| Persistent:  | yes                                  |

Autostart: enable

Managed save: no

Security model: apparmor

Security label: libvirt-1b9d6bcd-bbfd-4b2d-9b5d-ab8dfbbd4bed (enforcing)

主要字段含义：

**Id:** 虚拟机运行时的临时 ID（关闭时为 -）

**Name:** 虚拟机名称

**UUID:** 虚拟机唯一标识符（永久不变）

**OS Type:** 虚拟化类型（如 hvm 全虚拟化）

**State:** 当前状态（running 运行中、shut off 已关闭、paused 暂停等）

**CPU(s):** 分配的虚拟 CPU 数量

**Max memory:** 最大可使用内存（KiB）

**Used memory:** 当前使用的内存（KiB）

**Persistent:** 是否为持久化虚拟机（yes 表示配置文件永久保存，no 为临时虚拟机）

**Autostart:** 是否启用开机自动启动（enable/disable）

常见用法：

查看指定虚拟机信息：

```
bash
```

```
virsh dominfo myvm
```

结合 grep 筛选特定信息（如查看内存）：

```
bash
```

```
virsh dominfo myvm | grep -i memory
```

通过 dominfo 可以快速了解虚拟机的资源配置和运行状态，是管理 KVM 虚拟机时的常用命令。

### 6.3.3.虚拟机运行状态信息

```
virsh domstate
```

查看虚拟机的运行状态

虚拟机主要有如下几种状态：

未定义（**undefined**）：虚拟机未定义或未创建，即 **libvirt** 认为该虚拟机不存在。

关闭状态（**shut off**）：虚拟机已经被定义但未运行，或者虚拟机被终止。

运行（**running**）：虚拟机处于运行状态。

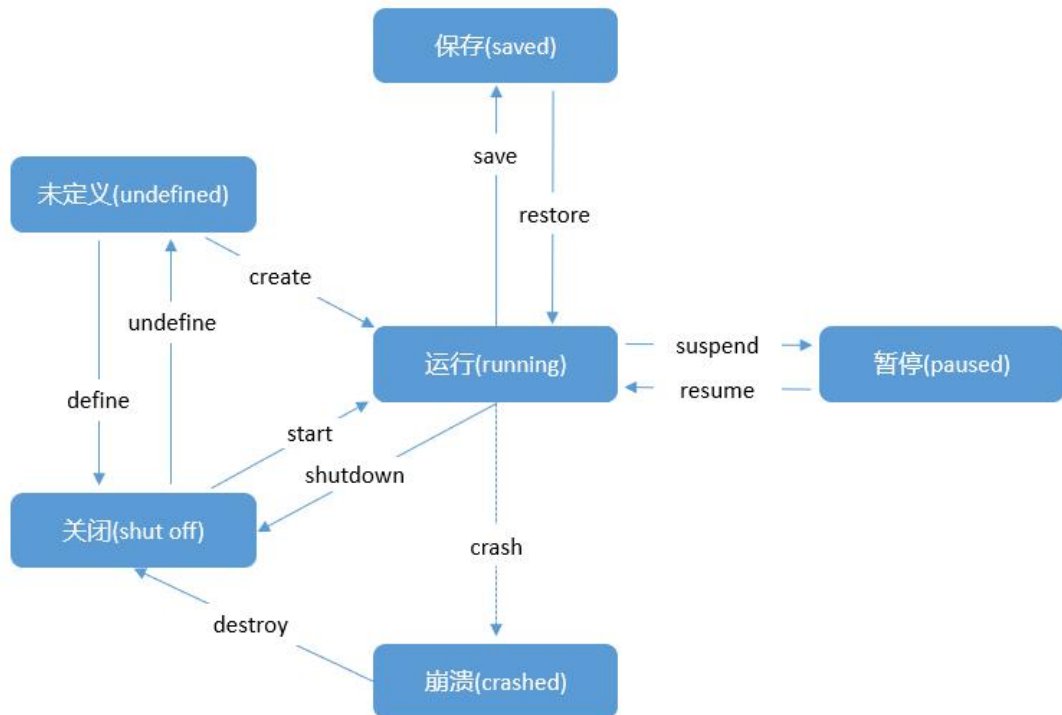
暂停（**paused**）：虚拟机运行被挂起，其运行状态被临时保存在内存中，可以恢复到运行状态。

保存（**saved**）：与暂停（**paused**）状态类似，其运行状态被保存在持久性存储介质中，可以恢复到运行状态。

崩溃（**crashed**）：通常是由于内部错误导致虚拟机崩溃，不可恢复到运行状态。

状态转换 ⇄

虚拟机不同状态之间可以相互转换，但必须满足一定规则。虚拟机不同状态之间的转换常用规则如图 1 所示。



#### 6.3.4.虚拟机统计信息

`virsh domstats`

于查看虚拟机（**domain**）的详细统计信息，包括资源使用、性能数据等。你列出的是该命令的常用参数，用于筛选特定类型的统计信息。以下是这些参数的详细说明：

常用参数说明：

- backing: 显示镜像文件的后端存储信息（如 backing file 路径、格式等）
- balloon: 显示内存气球（balloon）相关统计（如当前内存分配、可调整范围）
- block: 显示磁盘设备的 I/O 统计（读写次数、字节数、延迟等）
- cpu-total: 显示总 CPU 使用率统计（如用户态 / 内核态时间、总运行时间）
- domain <名称/UUID>: 指定单个虚拟机（不指定则显示所有）
- interface: 显示网络接口的统计（收发数据包数、字节数、错误数等）
- iothread: 显示 I/O 线程的统计信息
- memory: 显示内存使用详情（如实际使用、共享内存、交换空间等）
- vcpu: 显示每个虚拟 CPU 的单独统计（如每个 vCPU 的使用率、运行时间）
- state: 显示虚拟机状态相关信息（如运行状态、切换次数等）
- perf: 显示性能监控相关数据
- raw: 以原始格式输出（未格式化的键值对，适合脚本处理）

筛选虚拟机范围的参数：

- list-active: 仅统计活跃（运行中）的虚拟机
- list-inactive: 仅统计非活跃（已关闭）的虚拟机
- list-running: 仅统计运行中的虚拟机
- list-shutoff: 仅统计已关闭的虚拟机
- list-paused: 仅统计暂停的虚拟机
- list-persistent: 仅统计持久化虚拟机
- list-transient: 仅统计临时虚拟机

使用示例：

查看指定虚拟机的磁盘 I/O 统计：

```
bash
```

```
virsh domstats --block myvm
```

查看所有运行中虚拟机的网络接口统计：

```
bash
```

```
virsh domstats --list-running --interface
```

查看单个虚拟机的内存使用详情：

```
bash
```

```
virsh domstats --memory myvm
```

查看所有虚拟机的 vCPU 统计（按原始格式输出）：

```
bash
```

```
virsh domstats --vcpu --raw
```

这些参数可以组合使用，帮助精确获取需要的虚拟机性能和资源使用数据，常用于监控、调优或故障排查。

你提供的是 `virsh domstats --domain fnos` 命令的输出结果，展示了名为 `fnos` 的虚拟机的详细统计信息。这些数据反映了虚拟机的运行状态、资源使用情况等关键指标。以下是主要字段的解读：

#### 1. 基本状态信息

`state.state=1`：虚拟机当前状态为 运行中（1 表示 `running`，5 表示 `shut off`）

`state.reason=1`：启动原因（1 通常表示正常启动）

#### 2. CPU 统计

`cpu.time=2450907762660`：总 CPU 运行时间（纳秒，约 2450 秒）

`cpu.user=1547500000000`：用户态 CPU 时间（约 1547 秒）

`cpu.system=454760000000`：内核态 CPU 时间（约 454 秒）

`vcpu.current=1`：当前启用的虚拟 CPU 数量（1 核）

`vcpu.maximum=2`：最大可配置的虚拟 CPU 数量（2 核）

`vcpu.0.state=1`：第 0 个 vCPU 状态为运行中

`vcpu.0.time=1961120000000`：第 0 个 vCPU 的运行时间（约 1961 秒）

#### 3. 内存统计（`balloon` 相关）

`balloon.current=2097152`：当前分配的内存（KiB，约 2GiB）

`balloon.maximum=10485760`：最大可分配的内存（KiB，约 10GiB）

`balloon.rss=1260444`：实际使用的物理内存（KiB，约 1.2GiB）

`balloon.unused=123196`：未使用的内存（KiB，约 120MiB）

`balloon.major_fault=6281`：主要页面错误次数（需要从磁盘加载数据）

`balloon.minor_fault=15276634`：次要页面错误次数（内存页已在内存中但未映射）

#### 4. 网络统计

**net.count=1:** 网络接口数量 (1 个)

**net.0.name=vnet0:** 网络接口名称 (宿主机侧为 vnet0)

接收 (rx):

**net.0.rx.bytes=8460547:** 接收总字节数 (约 8.1MiB)

**net.0.rx.pkts=78379:** 接收总数据包数

**net.0.rx.errs=0 / net.0.rx.drop=0:** 接收错误 / 丢弃数 (无异常)

发送 (tx):

**net.0.tx.bytes=4866272:** 发送总字节数 (约 4.6MiB)

**net.0.tx.pkts=13263:** 发送总数据包数

**net.0.tx.errs=0 / net.0.tx.drop=0:** 发送错误 / 丢弃数 (无异常)

#### 5. 磁盘 (块设备) 统计

共 4 个块设备 (**block.count=4**), 以 **vda** (第一个磁盘) 为例:

**block.0.name=vda:** 设备名称

**block.0.path=/home/vm/.fnos.qcow2:** 镜像文件路径

读操作 (rd):

**block.0.rd.reqs=23774:** 读请求次数

**block.0.rd.bytes=1062681600:** 读总字节数 (约 1.0GiB)

**block.0.rd.times=21451261206:** 读操作总耗时 (纳秒, 约 21 秒)

写操作 (wr):

**block.0.wr.reqs=45107:** 写请求次数

**block.0.wr.bytes=469556736:** 写总字节数 (约 448MiB)

**block.0.wr.times=9423352656:** 写操作总耗时 (纳秒, 约 9 秒)

刷新操作 (fl):

**block.0.fl.reqs=19637:** 刷新请求次数 (将缓存数据写入磁盘)

**block.0.fl.times=25753184584:** 刷新总耗时 (纳秒, 约 25 秒)

存储容量:

**block.0.capacity=107374182400:** 总容量 (约 100GiB)

**block.0.allocation=85973815296:** 已分配空间 (约 80GiB)



block.0.physical=115967873024: 实际占用宿主机磁盘空间（约 108GiB，因 qcow2 格式有额外开销）

#### 总结

从统计结果来看，fnos 虚拟机状态正常：

运行中，使用 1 核 CPU，内存占用约 1.2GiB（未达上限）

网络无错误或丢包

磁盘 I/O 操作正常，主要读写集中在 vda 设备

这些数据可用于监控虚拟机性能、排查资源瓶颈（如 CPU 过高、内存不足、磁盘 I/O 延迟等）。

### 6.3.5.虚拟机内存统计信息

```
virsh dommemstat
```

virsh dommemstat 命令用于查看虚拟机的内存使用详细统计信息，是监控虚拟机内存资源状态的重要工具。它能展示内存分配、实际使用量、页面错误等关键指标，帮助判断内存配置是否合理。

#### 基本用法

```
bash
```

```
virsh dommemstat <虚拟机名称或 UUID>
```

#### 示例输出及字段解读

以查询名为 fnos 的虚拟机为例，输出可能如下：

```
plaintext
```

```
actual:      2097152 KiB
swap_in:     56 KiB
swap_out:    1244 KiB
major_fault: 6281
minor_fault: 15276634
unused:      123196 KiB
```

```
available:    1811668 KiB
usable:       706652 KiB
rss:          1260444 KiB
```

各字段含义:

`actual`: 当前分配给虚拟机的总内存 (KiB) , 即虚拟机可使用的内存上限

`swap_in`: 从交换分区换入到内存的数据量 (KiB) , 数值增长可能表示内存不足

`swap_out`: 从内存换出到交换分区的数据量 (KiB) , 持续增加说明内存压力大

`major_fault`: 主要页面错误次数, 指需要从磁盘加载数据的内存访问失败, 对性能影响较大

`minor_fault`: 次要页面错误次数, 指内存页已在物理内存中但未建立映射, 性能影响较小

`unused`: 虚拟机内部未使用的内存 (KiB) , 数值高说明内存分配可能过剩

`available`: 虚拟机内部可用的总内存 (KiB) , 包含未使用和可回收的内存

`usable`: 可通过气球 (balloon) 技术回收的内存量 (KiB) , 供动态调整参考

`rss`: 虚拟机实际使用的物理内存 (KiB, 宿主机视角) , 反映真实内存消耗

实用场景

若 `rss` 远小于 `actual`, 说明内存分配过多, 可适当缩减

若 `swap_in/swap_out` 持续增长, 可能需要增加内存分配

结合 `virsh setmem` 命令可动态调整运行中虚拟机的内存 (需支持热调整)

通过这些指标, 能有效评估虚拟机的内存使用效率, 为资源优化提供依据。

### 6.3.6.虚拟机 CPU 统计信息

virsh cpu-stats

你提到的 `virsh cpu-stats` 命令用于查看虚拟机的 CPU 统计信息，你列出的是该命令的常用参数。以下是这些参数的详细说明及用法：

常用参数说明：

`--domain <名称/UUID>`：指定要查询的虚拟机（必选参数，需替换为实际虚拟机名称或 UUID）

`--total`：显示总 CPU 统计（默认行为，汇总所有 vCPU 的数据）

`--count <次数>`：指定统计的采样次数（如 `--count 5` 表示采样 5 次）

`--start <时间>`：指定开始统计的时间（格式为秒，相对于虚拟机启动时间）

使用示例：

查看指定虚拟机的总 CPU 统计：

bash

virsh cpu-stats --domain fnos

（等价于 `virsh cpu-stats fnos`，默认显示总统计）

查看单个 vCPU 的详细统计（不加 `--total` 时会按 vCPU 拆分）：

bash

virsh cpu-stats --domain fnos

输出会包含每个 vCPU 的单独数据（如 `vcpu0`、`vcpu1` 等）。

多次采样统计（例如采样 3 次）：

bash

virsh cpu-stats --domain fnos --count 3

会连续输出 3 次 CPU 统计数据，便于观察变化。

典型输出解读：

以 `virsh cpu-stats fnos` 为例，输出可能类似：

plaintext

CPU statistics for domain: fnos

```
user: 1547500000000 ns
system: 454760000000 ns
total: 2450907762660 ns

vcpu0:

    user: 1200000000000 ns
    system: 300000000000 ns
    total: 1500000000000 ns
```

user: 用户态 CPU 时间

system: 内核态 CPU 时间

total: 总 CPU 时间

vcpu0: 单个虚拟 CPU 的详细数据

通过 `cpu-stats` 可以监控虚拟机的 CPU 资源使用情况，判断是否存在 CPU 瓶颈或负载不均衡问题。

### 6.3.7.虚拟机 vCPU 调度信息

```
virsh schedinfo
```

### 6.3.8.虚拟机 vCPU 数目信息

```
virsh vcpucount
```

`virsh vcpucount` 是用于查询虚拟机虚拟 CPU（vCPU）数量配置的命令，能直观展示虚拟机的 vCPU 相关信息，包括当前使用数、最大限制数等。

基本语法

```
bash
```

```
virsh vcpucount <虚拟机名称或 UUID> [选项]
```

主要选项及功能

无选项：默认显示该虚拟机的 vCPU 配置概览，包含最大数量、配置文件中定义的数量、当前运行中的数量等。

--maximum：仅显示该虚拟机允许配置的最大 vCPU 数量。

--current：仅显示当前实际活跃的 vCPU 数量。

**--config:** 显示虚拟机配置文件中定义的 vCPU 数量（该配置需重启虚拟机后生效）。

**--live:** 针对运行中的虚拟机，显示当前实时生效的 vCPU 数量。

示例说明

假设虚拟机名称为 **test-vm**：

执行 **virsh vcpucount test-vm**，可能输出：

plaintext

| maximum | config | live | current |
|---------|--------|------|---------|
| 4       | 2      | 2    | 2       |

各字段含义：

**maximum:** 最多可配置 4 个 vCPU

**config:** 配置文件中定义了 2 个 vCPU

**live:** 当前运行中使用 2 个 vCPU

**current:** 当前活跃的 vCPU 数量为 2

执行 **virsh vcpucount test-vm --maximum**，输出：4（表示该虚拟机最多可配置 4 个 vCPU）。

执行 **virsh vcpucount test-vm --live**，输出：2（表示当前运行中的虚拟机使用 2 个 vCPU）。

通过该命令，可快速了解虚拟机的 vCPU 配置情况，为资源调整（如增加 vCPU 数量）提供参考。若需修改 vCPU 数量，可使用 **virsh setvcpus** 命令（部分场景支持动态调整，无需重启虚拟机）。

### 虚拟机 vCPU 负载信息

**virsh vcpuinfo** 命令用于查看虚拟机中每个虚拟 CPU（vCPU）的详细信息，包括运行状态、使用率、关联的物理 CPU（pCPU）等，帮助分析 vCPU 的运行情况和负载分布。

基本用法：

bash

**virsh vcpuinfo** <虚拟机名称或 UUID>

示例输出及字段说明：

以名为 **fnos** 的虚拟机为例，运行 **virsh vcpuinfo fnos** 可能显示：

plaintext

VCPU: 0  
CPU: 3  
State: running  
CPU time: 1961.12s  
CPU Affinity: yyyyyyyy

各字段含义:

VCPU: 0: 虚拟 CPU 的编号 (从 0 开始, 若有多个 vCPU 会依次显示 0、1、2...)

CPU: 3: 当前 vCPU 绑定的物理 CPU 编号 (表示该 vCPU 正在宿主机的第 3 个物理核心上运行)

State: running: vCPU 状态 (running 运行中、idle 空闲、paused 暂停等)

CPU time: 1961.12s: 该 vCPU 累计运行的时间 (秒)

CPU Affinity: yyyyyyyy: vCPU 可调度的物理 CPU 范围 (y 表示允许调度到对应的物理核心, 长度等于宿主机物理核心数)

多 vCPU 场景示例:

若虚拟机有 2 个 vCPU, 输出可能类似:

plaintext

VCPU: 0  
CPU: 1  
State: running  
CPU time: 1200.50s  
CPU Affinity: yyyyyyyy  
VCPU: 1  
CPU: 5  
State: idle  
CPU time: 850.20s  
CPU Affinity: yyyyyyyy

表示 2 个 vCPU 分别运行在宿主机的第 1 和第 5 个物理核心上, 其中 vCPU1 处于空闲状态。

实用场景:

查看 vCPU 与物理 CPU 的绑定关系 (判断是否需要调整 CPU 亲和性以优化性能)

分析单个 vCPU 的负载（如某个 vCPU 持续高负载可能需要增加 vCPU 数量）

检查 vCPU 状态是否正常（如异常 paused 可能需要排查原因）

### 6.3.9.虚拟机 I/O 线程信息

virsh iothreadinfo

virsh iothreadinfo 命令用于查看虚拟机的 I/O 线程（IOThreads）配置信息，包括已配置的 I/O 线程数量及关联的设备等。I/O 线程是 KVM 中用于专门处理虚拟机 I/O 操作（如磁盘、网络）的线程，可减轻 vCPU 负担，提升 I/O 性能。

基本用法：

bash

virsh iothreadinfo --domain <虚拟机名称或 UUID>

输出说明：

未配置 I/O 线程的情况（最常见）：

plaintext

No IOThreads found for the domain

表示该虚拟机未启用 I/O 线程（默认状态），所有 I/O 操作由 vCPU 线程处理。

已配置 I/O 线程的情况：若虚拟机配置了 I/O 线程，输出可能类似：

plaintext

IOThreads: 2

IOThread 1:

devices: vda, vnet0

IOThread 2:

devices: vdb

各字段含义：

IOThreads: 2: 共配置了 2 个 I/O 线程

IOThread 1/IOThread 2: 线程编号（从 1 开始）

devices: vda, vnet0: 绑定到该 I/O 线程的设备（如磁盘 vda、网卡 vnet0）

实用场景：

确认虚拟机是否启用了 I/O 线程优化（针对 I/O 密集型场景）。

检查 I/O 设备与线程的绑定关系，判断是否存在负载集中在单个线程的情况。

结合 virsh domblkstat 或 virsh domifstat，分析 I/O 线程的负载是否均衡。

注意事项：

I/O 线程需手动配置（默认不启用），适合数据库、文件服务器等 I/O 密集型虚拟机。

配置方法: 通过 `virsh edit` 修改虚拟机 XML, 添加 `<iothreads>` 定义并为设备绑定线程（详见前文 `iothreadinfo` 输出解读中的配置步骤）。

对于计算密集型虚拟机（如无大量磁盘 / 网络操作），无需启用 I/O 线程，默认配置即可满足需求。

通过该命令可快速了解虚拟机的 I/O 线程配置，为性能优化提供参考。

### 6.3.10. 虚拟机块设备列表信息

`virsh domblklist` 命令用于列出虚拟机的块设备（磁盘、光驱等存储设备）信息，包括设备名称、关联的镜像文件路径等。你列出的是该命令的常用参数，以下是详细说明：

常用参数说明：

`--domain <名称/UUID>`: 指定要查询的虚拟机（必选参数，需替换为实际虚拟机名称或 UUID）

`--details`: 显示详细信息，包括设备类型（如 `disk/cdrom`）、设备总线（如 `virtio/sata`）、只读状态等

`--inactive`: 显示虚拟机配置文件中定义的块设备（即使虚拟机未运行，也能查看配置）

使用示例：

列出指定虚拟机的块设备（基础信息）：

bash

`virsh domblklist fnos`

可能输出：

plaintext

| Target | Source |
|--------|--------|
|--------|--------|

-----

|     |                      |
|-----|----------------------|
| vda | /home/vm/.fnos.qcow2 |
|-----|----------------------|

|     |                         |
|-----|-------------------------|
| vdb | /usb1/.fnosmaster.qcow2 |
|-----|-------------------------|

|     |                         |
|-----|-------------------------|
| vdc | /usb2/.fnosbackup.qcow2 |
|-----|-------------------------|

|     |   |
|-----|---|
| hdb | - |
|-----|---|

Target: 虚拟机内部看到的设备名（如 `vda`、`vdb`）

Source: 宿主机上对应的存储文件路径（- 表示无实际文件，可能是光驱等）



显示详细信息（包括设备类型、总线等）：

```
bash
```

```
virsh domblklist fnos --details
```

可能输出：

```
plaintext
```

| Target | Source                  | Type | Device |
|--------|-------------------------|------|--------|
| -----  |                         |      |        |
| vda    | /home/vm/.fnos.qcow2    | disk | virtio |
| vdb    | /usb1/.fnosmaster.qcow2 | disk | virtio |
| vdc    | /usb2/.fnosbackup.qcow2 | disk | virtio |
| hdb    | -                       | disk | ide    |

Type: 设备类型（disk 表示磁盘，cdrom 表示光驱）

Device: 设备使用的总线类型（virtio 性能较好，ide 兼容性强）

查看未运行虚拟机的块设备配置：

```
bash
```

```
virsh domblklist fnos --inactive
```

即使虚拟机处于关闭状态，也能显示配置文件中定义的块设备信息。

实用场景：

快速确认虚拟机挂载的磁盘及对应宿主机文件路径

检查磁盘使用的总线类型（如是否启用高性能的 virtio 驱动）

排查存储相关问题（如镜像文件路径是否正确、是否存在未使用的冗余设备）

### 6.3.11.虚拟机块设备信息

virsh domblkinfo 命令用于查看虚拟机中块设备（如磁盘、光驱）的详细存储信息，包括容量、已分配空间、实际占用空间等，帮助了解存储资源的使用情况。

基本用法：

```
bash
```

```
virsh domblkinfo <虚拟机名称或 UUID> <目标设备名>
```

<目标设备名>：指虚拟机内部的设备标识（如 **vda**、**vdb**，可通过 **virsh domblklist** 命令获取）

示例输出及字段说明：

以查询 **fnos** 虚拟机的 **vda** 设备为例：

```
bash
```

```
virsh domblkinfo fnos vda
```

可能输出：

```
plaintext
```

```
Capacity:      107374182400
```

```
Allocation:    85973815296
```

```
Physical:      115967873024
```

各字段含义：

**Capacity**：设备总容量（字节），即虚拟机看到的磁盘大小（示例中为 **100GiB**）

**Allocation**：已分配的空间（字节），指虚拟机实际使用的逻辑空间（示例中约 **80GiB**）

**Physical**：实际占用的宿主机物理磁盘空间（字节，示例中约 **108GiB**），因 **qcow2** 等稀疏格式可能包含元数据或快照，此值可能大于 **Allocation**

实用场景：

检查磁盘容量是否符合预期（如是否达到预设上限）

评估存储实际占用（**Physical** 反映宿主机真实消耗）

结合 **domblklist** 排查存储相关问题（如磁盘空间不足）

例如，若需查看所有块设备的信息，可先通过 **domblklist** 获取设备名，再逐个查询：

```
bash
```

```
virsh domblklist fnos --name | xargs -l {} virsh domblkinfo fnos {}
```

该命令对于监控虚拟机存储使用、规划存储资源非常有用。

### 6.3.12.虚拟机块设备统计信息

```
virsh domblkstat
```

`virsh domblkstat` 命令用于查看虚拟机块设备（如磁盘、光驱）的实时 I/O 统计信息，包括读写请求次数、字节数、操作延迟等，是监控虚拟机存储性能、排查磁盘 I/O 瓶颈的核心工具。

基本用法

bash

`virsh domblkstat <虚拟机名称或 UUID> [目标设备名] [选项]`

<目标设备名>: 可选，指定虚拟机内部的块设备（如 `vda`、`vdb`，通过 `virsh domblklist` 可获取）；不指定则显示所有块设备的统计。

选项: `--period <秒数>`（设置统计周期，默认实时输出）、`--total`（仅显示累计统计，不实时刷新）。

示例输出及字段解读

以查询 `fnos` 虚拟机的 `vda` 设备为例，运行 `virsh domblkstat fnos vda`，输出可能如下（实时刷新）：

plaintext

```
vda:  rd_req=23774, rd_bytes=1062681600, rd_time=21451261206
      wr_req=45107, wr_bytes=469556736, wr_time=9423352656
      fl_req=19637, fl_time=25753184584
      busy=46.62%
```

若不指定设备（`virsh domblkstat fnos`），则会按设备分组显示所有块设备（如 `vda`、`vdb`、`vdc`）的统计。

核心字段含义（单位说明：req= 次数，bytes= 字节，time= 纳秒）

| 字段                    | 含义                                                        |
|-----------------------|-----------------------------------------------------------|
| <code>rd_req</code>   | 累计读请求次数（磁盘读取操作的总次数）                                       |
| <code>rd_bytes</code> | 累计读字节数（从磁盘读取的总数据量，可换算为 GiB/MiB，1GiB=1024 <sup>3</sup> 字节） |
| <code>rd_time</code>  | 累计读操作总耗时（纳秒，换算为秒需除以 10 <sup>9</sup> ，反映读操作的整体延迟）          |
| <code>wr_req</code>   | 累计写请求次数（磁盘写入操作的总次数）                                       |
| <code>wr_bytes</code> | 累计写字节数（写入磁盘的总数据量）                                         |
| <code>wr_time</code>  | 累计写操作总耗时（纳秒，反映写操作的整体延迟）                                   |

| 字段      | 含义                                     |
|---------|----------------------------------------|
| fl_req  | 累计刷新请求次数（flush 操作，将缓存数据强制写入磁盘，保障数据持久化） |
| fl_time | 累计刷新操作总耗时（纳秒，延迟高可能影响数据写入效率）            |
| busy    | 磁盘设备繁忙率（百分比，反映磁盘 I/O 负载，100% 表示完全饱和）   |

#### 常用场景与示例

查看单个设备的实时 I/O 负载监控 vda 磁盘的读写情况，观察是否有 I/O 瓶颈：

```
bash
virsh domblkstat fnos vda
```

若 busy 长期接近 100%，且 rd\_time/wr\_time 持续增长，说明磁盘 I/O 饱和，需优化存储（如更换 SSD、拆分磁盘）。

查看所有设备的累计统计（不实时刷新）仅输出一次所有块设备的累计 I/O 数据，便于记录或脚本处理：

```
bash
virsh domblkstat fnos --total
```

按周期刷新统计（如每 2 秒更新一次）设置统计周期，持续监控 I/O 变化趋势：

```
bash
virsh domblkstat fnos --period 2
```

#### 关键指标分析建议

读 / 写延迟：rd\_time/rd\_req 可计算单次读请求平均延迟（纳秒），若数值远高于正常范围（如机械硬盘 > 20ms、SSD>1ms），需排查存储介质或 I/O 队列问题。

繁忙率：busy 超过 80% 时，可能导致虚拟机读写卡顿，需考虑扩容存储或优化应用 I/O 逻辑。

刷新操作：fl\_req/fl\_time 高可能因应用频繁调用 fsync，需平衡数据安全性与 I/O 性能。

### 6.3.13.虚拟机块设备错误信息

```
virsh domblkerror
```

`virsh domblkerror` 命令用于查询虚拟机块设备（如磁盘、光驱）的错误信息，帮助排查存储设备相关的故障（如 I/O 错误、连接失败等）。

基本用法：

`bash`

`virsh domblkerror <虚拟机名称或 UUID> [目标设备名]`

`<目标设备名>`：可选参数，指定虚拟机内部的块设备（如 `vda`、`vdb`，可通过 `virsh domblklist` 获取）；不指定则显示所有块设备的错误信息。

示例输出及说明：

无错误情况若块设备正常，运行 `virsh domblkerror fnos vda` 可能输出：

plaintext

No errors found for block device 'vda'

表示该设备当前无错误记录。

有错误情况若设备存在错误（如磁盘文件损坏或权限问题），可能输出类似：

plaintext

vda: I/O error, cannot read sector 12345

或更详细的错误描述，如权限不足、文件不存在等。

实用场景：

当虚拟机出现磁盘读写异常（如无法启动、文件访问失败）时，可通过该命令快速检查块设备是否有错误。

结合 `virsh domblklist` 和 `virsh domblkinfo` 定位具体故障设备，进一步排查宿主机上的存储文件是否正常（如路径是否存在、权限是否正确、磁盘空间是否充足）。

注意事项：

该命令主要显示虚拟机运行过程中记录的块设备错误，并非实时监控工具。

若输出无错误但虚拟机仍有存储问题，需结合日志（如宿主机的 `/var/log/libvirt/qemu/<虚拟机名>.log`）进一步分析。

通过 `domblkerror` 可快速定位块设备的已知错误，是存储故障排查的第一步。

### 6.3.14.虚拟机控制接口状态信息

```
virsh domcontrol
```

```
[root@efenye-euler ~]# virsh domcontrol --domain fnos
```

确定

### 6.3.15.虚拟机接口链接状态信息

```
virsh domif-getlink
```

`virsh domif-getlink` 命令用于查询虚拟机网络接口（网卡）的链路状态（如是否连接、启用等）。你列出的是该命令的主要参数，以下是详细说明及用法：

常用参数说明：

`--domain <名称/UUID>`：必选参数，指定要查询的虚拟机（如 `fnos`）。

`--interface <接口名>`：必选参数，指定虚拟机的网络接口名称（如 `vnet0`，可通过 `virsh domiflist` 命令获取）。

`--persistent`：可选参数，查询配置文件中定义的持久化链路状态（而非当前运行状态，重启后生效的配置）。

基本用法：

查询运行中的虚拟机网络接口状态（实时状态）：

```
bash
```

```
virsh domif-getlink --domain fnos --interface vnet0
```

可能输出：

```
plaintext
```

```
Link state: up
```

表示接口 `vnet0` 当前链路状态为 启用（`up`）。

查询持久化配置中的链路状态（配置文件中定义的状态）：

```
bash
```

```
virsh domif-getlink --domain fnos --interface vnet0 --persistent
```

可能输出：

plaintext

Link state: up

表示配置文件中定义该接口默认链路状态为 启用（up）。

链路状态含义：

up：链路已启用，网络接口可正常收发数据。

down：链路已禁用，网络接口无法通信。

6.3.16.虚拟机网络接口列表信息

virsh domiflist

virsh domiflist 命令用于列出虚拟机的网络接口信息，包括接口名称、类型、源网络 / 网桥、MAC 地址等，是查看虚拟机网络配置的常用工具。

基本用法：

bash

virsh domiflist <虚拟机名称或 UUID>

示例输出及字段说明：

以名为 fnos 的虚拟机为例，运行 virsh domiflist fnos 可能显示：

plaintext

| Interface | Type    | Source  | Model  | MAC               |
|-----------|---------|---------|--------|-------------------|
| -----     |         |         |        |                   |
| vnet0     | bridge  | br0     | virtio | 52:54:00:12:34:56 |
| vnet1     | network | default | e1000  | 52:54:00:aa:bb:cc |

各字段含义：

Interface：宿主机上对应的虚拟接口名称（如 vnet0，通常与虚拟机内部接口一一对应）

Type：网络连接类型（bridge 表示桥接模式，network 表示虚拟网络模式）

Source：关联的宿主机网络资源（桥接模式下为网桥名如 br0，虚拟网络模式下为网络名如 default）

Model：虚拟机内部看到的网卡型号（virtio 性能最优，e1000 兼容性好）

MAC：虚拟机网卡的 MAC 地址（唯一标识，用于网络通信）

实用场景：

快速确认虚拟机的网络连接方式（桥接 / 虚拟网络）

查看网卡型号是否为高性能的 `virtio`（若为其他型号可能影响网络性能）

排查网络问题时，确认接口与宿主机网桥 / 虚拟网络的关联是否正确

获取接口名称（如 `vnet0`）用于后续操作（如 `virsh domif-getlink` 查看链路状态）

扩展用法：

结合其他命令可深入分析网络状态：

查看接口链路状态：`virsh domif-getlink fnos vnet0`

查看接口 IP 地址（需 QGA 支持）：`virsh domifaddr fnos`

临时添加网络接口：`virsh attach-interface`

通过 `domiflist` 可全面了解虚拟机的网络接口配置，是网络故障排查和优化的基础。

分享

如何使用 `virsh domiflist` 命令查询虚拟机的网络接口信息？

`virsh domiflist` 命令的返回值有哪些？

除了 `virsh domiflist` 命令，还有哪些命令可以查看虚拟机的网络配置？

### 6.3.17.虚拟机网络接口运行信息

`virsh domifaddr`

`virsh domifaddr` 命令用于查看虚拟机网络接口的 IP 地址信息，是获取虚拟机内部网络配置的便捷工具。不过该命令依赖 QEMU 虚拟机代理（QGA）才能正常工作（如未配置 QGA，会提示错误）。

基本用法：

bash

`virsh domifaddr <虚拟机名称或 UUID> [选项]`

常用选项：

无选项：默认显示所有网络接口的 IP 地址

`--interface <接口名>`：仅显示指定接口（如 `vnet0`）的 IP 地址

`--full`：显示更详细的信息（包括接口类型、MAC 地址等）

示例输出及说明：

当 QGA 配置正常时，查询 `fnos` 虚拟机的 IP 地址：

bash

`virsh domifaddr fnos`

可能输出：



plaintext

| Name  | MAC address       | Protocol | Address                    |
|-------|-------------------|----------|----------------------------|
| ----- |                   |          |                            |
| vnet0 | 52:54:00:9c:ce:30 | ipv4     | 192.168.1.105/24           |
| vnet0 | 52:54:00:9c:ce:30 | ipv6     | fe80::5054:ff:fe9c:ce30/64 |

各字段含义：

**Name：**宿主机侧的虚拟接口名（如 vnet0）

**MAC address：**网卡的 MAC 地址

**Protocol：**IP 协议版本（ipv4 或 ipv6）

**Address：**IP 地址及子网掩码（如 192.168.1.105/24）

注意事项：

依赖 QGA：若未配置 QEMU 虚拟机代理，会提示类似错误：

plaintext

错误：获取接口地址失败

错误：不支持的参数：未配置 QEMU 虚拟机代理

解决方法：参考之前提到的 QGA 配置步骤，在虚拟机内安装并启用 qemu-guest-agent。

与 domiflist 的区别：

domiflist 显示接口的基础配置（类型、源网络、MAC 等）

domifaddr 显示接口实际获取的 IP 地址（需 QGA 支持）

实用场景：

快速获取运行中虚拟机的 IP 地址（无需登录虚拟机内部）

检查虚拟机网络配置是否正确（如 IP 是否在预期网段）

结合 ping 或 ssh 命令远程连接虚拟机（直接使用获取的 IP）

通过 domifaddr 可高效管理虚拟机的网络连接，尤其适合批量管理多台虚拟机时快速定位网络地址。

### 6.3.18.虚拟机网络接口统计信息

virsh domifstat

你提供的是 virsh domifstat 命令查询 fnos 虚拟机 vnet0 网络接口的统计结果，展示了该接口的网络流量和错误情况。以下是各字段的详细解读：

字段含义（单位：字节 / 数据包）

| 字段 | 数值 | 含义 |
|----|----|----|
|----|----|----|

| 字段         | 数值      | 含义                                    |
|------------|---------|---------------------------------------|
| rx_bytes   | 9195619 | 累计接收的总字节数（约 8.77 MiB），反映虚拟机接收的数据量     |
| rx_packets | 88969   | 累计接收的数据包总数，反映网络通信的频繁程度                |
| rx_errs    | 0       | 接收错误的数据包数（0 表示无错误，若大于 0 可能是网络链路或配置问题） |
| rx_drop    | 0       | 被丢弃的接收数据包数（0 表示无丢弃，若持续增长可能是带宽饱和或缓存不足） |
| tx_bytes   | 5046642 | 累计发送的总字节数（约 4.81 MiB），反映虚拟机发送的数据量     |
| tx_packets | 14800   | 累计发送的数据包总数                            |
| tx_errs    | 0       | 发送错误的数据包数（0 表示无错误，非 0 可能是网络拥塞或目标不可达）  |
| tx_drop    | 0       | 被丢弃的发送数据包数（0 表示无丢弃，非 0 可能是本地网络缓冲区不足）  |

#### 结果分析

从当前统计来看，vnet0 接口状态正常：

接收和发送的错误数（rx\_errs/tx\_errs）均为 0，说明网络链路通畅，无通信错误。

丢弃的数据包（rx\_drop/tx\_drop）均为 0，表明网络带宽充足，未出现因过载导致的丢包。

接收数据量（rx\_bytes）大于发送量（tx\_bytes），符合多数虚拟机的网络行为（如接收更多数据）。

#### 实用场景

若 rx\_errs 或 tx\_errs 非 0，需检查网络配置（如 VLAN、防火墙规则）或物理链路（如网线、交换机端口）。

若 rx\_drop 或 tx\_drop 持续增长，可能是网络带宽不足，需升级网络或限制虚拟机带宽。

结合 --period <秒数> 选项（如 virsh domifstat fnos vnet0 --period 2），可实时监控流量变化，判断是否有突发网络负载。

该命令是监控虚拟机网络性能、排查网络故障的重要工具，能帮助快速定位网络瓶颈或异常。

### 6.3.19.虚拟机时间

`virsh domtime` 是 `libvirt` 工具集中用于管理虚拟机（`domain`）时间相关设置的命令。它主要用于查看和调整虚拟机的当前时间、时区以及时钟相关配置。

主要功能和用法如下：

查看虚拟机时间信息

bash

`virsh domtime <虚拟机名称或 ID>`

该命令会显示虚拟机的当前时间、时区偏移、是否使用 UTC 时间等信息。

设置虚拟机时间

bash

`virsh domtime <虚拟机名称或 ID> --set <时间字符串>`

可以通过特定格式的时间字符串来设置虚拟机的时间，例如："`2023-10-05 14:30:00`"

同步虚拟机时间到主机

bash

`virsh domtime <虚拟机名称或 ID> --sync`

该选项会将虚拟机的时间与宿主机的时间进行同步。

设置虚拟机时区可以通过指定 UTC 偏移来设置虚拟机的时区：

bash

`virsh domtime <虚拟机名称或 ID> --offset <秒数>`

其中正数表示 UTC+，负数表示 UTC-

使用这个命令需要注意：

某些操作可能需要虚拟机处于运行状态

对于某些虚拟机类型（如 KVM），时间同步也可以通过 `guest agent` 或其他机制实现

修改虚拟机时间可能会影响虚拟机内运行的应用程序和服务

### 6.3.20.虚拟机图形接口信息

`virsh domdisplay` 是用于获取虚拟机图形显示接口信息的命令，它能返回连接虚拟机图形界面的方式（如 **VNC**、**SPICE** 等）及连接地址，方便管理员通过图形工具（如 **virt-viewer**、**VNC** 客户端）直接操作虚拟机桌面。

#### 一、基本用法

##### 1. 语法格式

bash

```
virsh domdisplay [--domain] <虚拟机名/ID> [--type <显示类型>]
```

**--domain**: 可选参数，显式指定虚拟机（通常可省略，直接跟虚拟机名 / ID）。

**--type**: 可选参数，指定显示类型（如 **vnc**、**spice**，默认自动识别虚拟机配置的显示类型）。

##### 2. 基础示例

获取名为 **ubuntuvvm** 的虚拟机的显示信息：

bash

```
virsh domdisplay ubuntuvvm
```

常见输出场景：

**VNC** 显示（最常见，尤其是默认配置）：

plaintext

```
vnc://127.0.0.1:0
```

含义：通过 **VNC** 协议连接，地址为 **127.0.0.1**（宿主机本地），端口为 **5900**（**VNC** 默认基础端口 **5900** + 显示编号 **0**，即 **5900 + 0 = 5900**）。

**SPICE** 显示（高性能图形协议，常用于需要高清 / 高帧率的场景）：

plaintext

```
spice://127.0.0.1:5900
```

含义：通过 **SPICE** 协议连接，地址 **127.0.0.1**，端口 **5900**（**SPICE** 端口通常与 **VNC** 类似，按虚拟机编号递增）。

本地 **UNIX** 套接字（仅宿主机本地访问）：

plaintext

```
vnc+unix:///var/run/libvirt/qemu/domain-ubuntuvvm/vnc.sock
```

含义：仅允许宿主机本地通过 **UNIX** 套接字连接，无法远程访问（安全性更高，适合无需远程图形连接的场景）。

## 二、关键参数与扩展用法

### 1. 指定显示类型（--type）

若虚拟机配置了多种显示类型（如同时支持 **VNC** 和 **SPICE**），可显式指定类型获取对应地址：

```
bash
```

```
# 获取 SPICE 显示地址 virsh domdisplay ubuntuvm --type spice
```

```
# 获取 VNC 显示地址 virsh domdisplay ubuntuvm --type vnc
```

### 2. 显示原始 XML 配置（--print-xml）

若需要查看虚拟机显示接口的完整 **XML** 配置（而非仅连接地址），可添加 **--print-xml** 参数：

```
bash
```

```
virsh domdisplay ubuntuvm --print-xml
```

输出示例（**VNC** 显示的 **XML** 配置）：

```
xml
```

```
<graphics type='vnc' port='5900' autoport='yes' listen='127.0.0.1'>
```

```
  <listen type='address' address='127.0.0.1' /></graphics>
```

关键配置解析：

**type='vnc'**：显示类型为 **VNC**。

**port='5900'**：固定端口 5900（若 **autoport='yes'**，则会自动分配未占用的端口，如 5901、5902）。

**listen='127.0.0.1'**：仅允许宿主机本地访问（若需远程访问，需改为宿主机的外网 IP 或 0.0.0.0，但需注意安全风险）。

## 三、如何通过 domdisplay 结果连接虚拟机

获取 **domdisplay** 的输出后，可通过对应的图形工具连接虚拟机桌面：

### 1. 宿主机本地连接（推荐）

工具：**virt-viewer**（**libvirt** 官方图形工具，兼容性最好）。

用法：直接将 **domdisplay** 的输出作为参数传入，或通过虚拟机名启动：

```
bash
```

# 方式 1: 直接使用 domdisplay 结果

```
virt-viewer $(virsh domdisplay ubuntuvm)
```

# 方式 2: 直接指定虚拟机名 (更简单)

```
virt-viewer ubuntuvm
```

2. 远程连接 (宿主机外的设备)

若 domdisplay 输出的地址是 127.0.0.1 (仅本地访问), 需先修改虚拟机的显示配置, 允许远程访问:

编辑虚拟机 XML 配置:

```
bash
```

```
virsh edit ubuntuvm
```

找到 <graphics> 标签, 修改 listen 地址为宿主机的外网 IP (如 192.168.3.100) 或 0.0.0.0 (允许所有设备访问):

```
xml
```

```
<graphics type='vnc' port='5900' autoport='yes' listen='0.0.0.0'>
```

```
  <listen type='address' address='0.0.0.0' /></graphics>
```

重启虚拟机使配置生效:

```
bash
```

```
virsh reboot ubuntuvm
```

远程设备使用 VNC 客户端 (如 RealVNC、TigerVNC) 连接, 地址为 宿主机 IP:端口 (如 192.168.3.100:5900)。

#### 四、常见问题排查

1. 报错「No graphics found for domain」

原因: 虚拟机未配置图形显示接口 (如纯命令行虚拟机, 未启用 VNC/SPICE)。

解决:

编辑虚拟机 XML, 添加图形配置 (以 VNC 为例):

```
xml
```

```
<devices>
```

```
  <!-- 其他设备配置 -->
```

```
  <graphics type='vnc' autoport='yes' listen='127.0.0.1' /></devices>
```

重启虚拟机: `virsh reboot ubuntuvm`。

## 2. 连接超时（VNC/SPICE 客户端无法连接）

排查步骤：

确认虚拟机已启动：`virsh list --all`（确保状态为 `running`）。

检查端口是否被占用：`ss -tuln | grep 5900`（替换为 `domdisplay` 显示的端口）。

检查宿主机防火墙：若远程连接，需开放对应端口（如 5900）：

`bash`

```
# 临时开放 5900 端口（CentOS/RHEL）sudo firewall-cmd --add-port=5900/tcp
```

```
--permanent sudo firewall-cmd --reload
```

```
# 临时开放 5900 端口（Ubuntu/Debian）sudo ufw allow 5900/tcp
```

## 五、总结

`virsh domdisplay` 的核心作用是快速获取虚拟机图形连接方式，配合 `virt-viewer` 或 VNC/SPICE 客户端，可便捷地管理虚拟机桌面。关键注意点：

若仅需本地访问，默认配置（`listen='127.0.0.1'`）足够安全。

若需远程访问，需修改 `listen` 地址并开放防火墙端口，同时注意 VNC 协议无加密（敏感场景建议用 SPICE 或搭配 SSH 隧道）。

### 6.3.21.虚拟机名称信息

`virsh domname` 是用于通过虚拟机的 ID 或 UUID，反向查询其名称的命令。在管理多个虚拟机时，若你只记得虚拟机的 ID（如 1、2）或 UUID

（如 123e4567-e89b-12d3-a456-426614174000），而忘记了名称，可通过该命令快速定位。

#### 一、基本用法与语法

##### 1. 核心语法

`bash`

```
virsh domname <虚拟机 ID 或 UUID>
```

参数说明：仅需传入一个参数 —— 虚拟机的「ID」（数字，如 1）或「UUID」（36 位字符串，含连字符）。

返回结果：直接输出对应的虚拟机名称（若存在）；若不存在，会提示错误。

##### 2. 关键前提

该命令对「运行中」和「已关闭」的虚拟机均有效（只要虚拟机在 `virsh list --all` 中能查到）。

虚拟机的 ID 是动态的（每次启动可能变化，如第一次启动是 1，重启后可能变为 2），而 UUID 是唯一且固定的（创建虚拟机时自动生成，除非手动修改）。

## 二、实际示例

### 1. 通过「虚拟机 ID」查询名称

首先通过 `virsh list --all` 查看虚拟机的 ID（以 ID 列为准）：

bash

# 查看所有虚拟机的 ID、名称、状态 `virsh list --all`

输出示例：

plaintext

| Id | 名称        | 状态       |
|----|-----------|----------|
| 1  | ubuntuvvm | running  |
| -  | centosvm  | shut off |

若已知 `ubuntuvvm` 的 ID 是 1，用 `domname` 查询名称（验证场景，或忘记名称时查询）：

bash

`virsh domname 1`

输出结果（直接返回名称）：

plaintext

ubuntuvvm

### 2. 通过「虚拟机 UUID」查询名称

首先通过 `virsh domuuid` 获取虚拟机的 UUID（若忘记 UUID，可先查名称对应的 UUID）：

bash

# 先获取 `ubuntuvvm` 的 UUID `virsh domuuid ubuntuvvm`

输出示例（UUID 字符串）：

plaintext

123e4567-e89b-12d3-a456-426614174000

再用该 UUID 通过 `domname` 查询名称：

bash



```
virsh domname 123e4567-e89b-12d3-a456-426614174000
```

输出结果：

```
plaintext
```

```
ubuntuvvm
```

3. 错误场景（无对应 ID/UUID）

若传入不存在的 ID 或 UUID，命令会提示错误：

```
bash
```

```
# 传入不存在的 ID 100virsh domname 100
```

错误输出：

```
plaintext
```

错误：找不到域 '100'

三、与相关命令的对比

domname 是「ID/UUID → 名称」的反向查询工具，常与以下「名称 → ID/UUID」的命令配合使用，形成完整的虚拟机标识转换：

| 命令               | 功能                      | 示例                            | 输出结果                                 |
|------------------|-------------------------|-------------------------------|--------------------------------------|
| virsh<br>domname | ID/UUID → 名称            | virsh<br>domname 1            | ubuntuvvm                            |
| virsh<br>domid   | 名称 / UUID<br>→ ID（仅运行中） | virsh<br>domid<br>ubuntuvvm   | 1                                    |
| virsh<br>domuuid | 名称 / ID →<br>UUID（永久唯一） | virsh<br>domuuid<br>ubuntuvvm | 123e4567-e89b-12d3-a456-426614174000 |

注意：virsh domid 仅对「运行中」的虚拟机有效（关闭的虚拟机无 ID），而 domname 和 domuuid 对所有状态的虚拟机均有效。

四、常见使用场景

批量管理时定位虚拟机：若通过脚本或批量命令获取到虚拟机 ID（如 virsh list --name 仅输出名称，若需 ID 需配合 domid），但需名称进行后续操作（如 virsh start 名称），可通过 domname 转换。

忘记名称时快速查询：仅记得虚拟机的 ID 或 UUID（如之前记录过 UUID），但忘记名称，无需遍历 `virsh list --all`，直接用 `domname` 定位。

脚本自动化：在 Shell 脚本中，若变量存储的是虚拟机 ID/UUID，可通过 `domname` 将其转换为名称，方便后续调用依赖名称的命令（如 `virsh domfsinfo 名称`）。

#### 总结

`virsh domname` 是一个轻量但实用的工具，核心价值是「通过唯一标识（ID/UUID）反向获取虚拟机名称」，尤其适合多虚拟机管理或自动化场景。使用时只需确保传入的 ID/UUID 正确，即可快速得到结果。

### 6.3.22.虚拟机 UUID 信息

`virsh domuuid` 是用于获取虚拟机唯一标识符（UUID）的命令。UUID（通用唯一识别码）是一个 36 位的字符串（格式如 `123e4567-e89b-12d3-a456-426614174000`），在虚拟机创建时自动生成，且永久唯一（除非手动修改），不受虚拟机状态（运行 / 关闭）或名称变化的影响，是识别虚拟机的可靠标识。

#### 一、基本用法与语法

##### 1. 核心语法

bash

`virsh domuuid <虚拟机名称 或 ID>`

参数：需传入虚拟机的「名称」（如 `ubuntuvvm`）或「运行时 ID」（如 `1`，仅对运行中的虚拟机有效）。

返回结果：输出该虚拟机的 UUID 字符串；若虚拟机不存在，返回错误。

##### 2. 关键特性

永久性：UUID 一旦生成，除非手动修改，否则不会改变（即使虚拟机重命名、重启、迁移，UUID 保持不变）。

唯一性：同一宿主机上的虚拟机 UUID 绝对唯一，可用于精准区分虚拟机（即使名称重复，UUID 也不同）。

全状态有效：对运行中（`running`）、关闭（`shut off`）、暂停（`paused`）等所有状态的虚拟机均有效。

#### 二、实际示例

### 1. 通过「虚拟机名称」获取 UUID

bash

# 获取名为 ubuntuvm 的虚拟机的 UUID  
virsh domuuid ubuntuvm

输出示例：

plaintext

123e4567-e89b-12d3-a456-426614174000

### 2. 通过「虚拟机 ID」获取 UUID（仅运行中）

先通过 virsh list 确认运行中虚拟机的 ID：

bash

virsh list # 仅显示运行中的虚拟机

输出示例：

plaintext

| Id | 名称 | 状态 |
|----|----|----|
|----|----|----|

-----

|   |          |         |
|---|----------|---------|
| 1 | ubuntuvm | running |
|---|----------|---------|

再用 ID 查询 UUID：

bash

virsh domuuid 1 # 通过 ID 1 获取对应虚拟机的 UUID

输出示例（与通过名称查询的结果一致）：

plaintext

123e4567-e89b-12d3-a456-426614174000

### 3. 错误场景（虚拟机不存在）

若传入不存在的名称或 ID，命令会报错：

bash

virsh domuuid invalid-vm # 传入不存在的虚拟机名称

错误输出：

plaintext

错误：找不到域 'invalid-vm'

## 三、UUID 的重要用途

唯一标识虚拟机：在脚本自动化、批量管理或虚拟机迁移场景中，用 UUID 定位虚拟机比名称更可靠（避免名称重复或变更导致的错误）。

修改虚拟机配置：编辑虚拟机 XML 配置时，UUID 是核心标识之一（位于 <domain> 标签的 uuid 属性）：

```
bash
```

```
virsh edit ubuntuvm # 编辑配置，可看到 <domain type='kvm' id='1' uuid='...'>
```

跨工具兼容：几乎所有虚拟化管理工具（如 virt-manager、ovirt）都支持通过 UUID 操作虚拟机，是标准化的标识方式。

查询反向映射：结合 virsh domname 命令，可通过 UUID 反查虚拟机名称（virsh domname <UUID>），实现标识转换。

#### 四、修改 UUID（特殊场景）

通常不建议修改 UUID，但在克隆虚拟机等特殊场景（避免 UUID 冲突）下，可通过以下步骤修改：

确保虚拟机处于关闭状态：

```
bash
```

```
virsh shutdown ubuntuvm
```

生成新 UUID（可通过 uuidgen 命令）：

```
bash
```

```
uuidgen # 生成新 UUID，如 550e8400-e29b-41d4-a716-446655440000
```

编辑虚拟机配置，替换旧 UUID：

```
bash
```

```
virsh edit ubuntuvm
```

找到 <domain> 标签的 uuid 属性，替换为新值：

```
xml
```

```
<domain type='kvm' uuid='550e8400-e29b-41d4-a716-446655440000'> <!-- 替换为新  
UUID -->
```

保存配置并启动虚拟机：

```
bash
```

```
virsh start ubuntuvm
```

总结

`virsh domuuid` 是获取虚拟机唯一、永久标识的核心命令，UUID 因其稳定性和唯一性，在自动化管理、跨工具操作等场景中比名称或 ID 更可靠。日常使用中，通过名称或 ID 即可快速获取 UUID，满足大多数管理需求。

### 6.3.23.虚拟机 ID 信息

`virsh domid` 是用于通过虚拟机的名称或 UUID，查询其当前运行时 ID 的命令。虚拟机的 ID 是一个动态生成的数字（通常从 1 开始递增），仅在虚拟机运行期间有效，关闭后会释放，下次启动可能变化。

#### 一、基本用法与语法

##### 1. 核心语法

```
bash
```

```
virsh domid <虚拟机名称 或 UUID>
```

参数：需传入虚拟机的「名称」（如 `ubuntuvm`）或「UUID」（36 位字符串）。

返回结果：若虚拟机正在运行，输出其当前 ID（数字）；若未运行或不存在，返回错误。

##### 2. 关键特性

仅对运行中虚拟机有效：关闭（`shut off`）或暂停（`paused` 部分状态）的虚拟机没有 ID，会提示错误。

ID 动态性：同一虚拟机每次启动可能获得不同的 ID（取决于宿主机当时的虚拟机启动顺序）。

#### 二、实际示例

##### 1. 通过「虚拟机名称」查询 ID

先确认虚拟机处于运行状态：

```
bash
```

```
virsh list --all # 查看状态，确保目标虚拟机显示为 running
```

查询运行中的 `ubuntuvm` 的 ID：

```
bash
```

```
virsh domid ubuntuvm
```

输出示例（运行中）：

```
plaintext
```

1 # 表示当前 ID 是 1

2. 通过「虚拟机 UUID」查询 ID

先获取虚拟机的 UUID（若已知名称）：

```
bash
```

```
virsh domuuid ubuntuvm # 获取 UUID
```

输出示例：

```
plaintext
```

```
123e4567-e89b-12d3-a456-426614174000
```

再用 UUID 查询 ID：

```
bash
```

```
virsh domid 123e4567-e89b-12d3-a456-426614174000
```

输出示例（运行中）：

```
plaintext
```

```
1
```

3. 错误场景（虚拟机未运行）

若虚拟机处于关闭状态（shut off），执行命令会报错：

```
bash
```

```
virsh domid ubuntuvm # 假设 ubuntuvm 已关闭
```

错误输出：

```
plaintext
```

错误：获取域 'ubuntuvm' 的 ID 失败

错误：域不是运行中的

三、与相关命令的配合

domid 常与其他命令配合，实现「名称 / UUID → ID」的转换，在脚本或批量操作中很有用：

| 场景需求              | 组合命令示例                                                                                               |
|-------------------|------------------------------------------------------------------------------------------------------|
| 先查 ID，再用 ID 执行操作  | <code>id=\$(virsh domid ubuntuvm) &amp;&amp; virsh shutdown \$id # 用 ID 关闭虚拟机</code>                 |
| 验证虚拟机是否在运行（通过 ID） | <code>if virsh domid ubuntuvm &gt;/dev/null 2&gt;&amp;1; then echo "运行中"; else echo "未运行"; fi</code> |

#### 四、使用场景

**快速定位运行中的虚拟机：**在 `virsh list` 输出中，ID 是最左侧的标识，通过 `domid` 可快速关联名称与 ID。

**脚本自动化：**部分 `virsh` 命令支持用 ID 操作虚拟机（如 `virsh shutdown 1`），在脚本中通过 `domid` 将名称转换为 ID 更灵活。

**区分同名虚拟机：**理论上虚拟机名称可重复（不推荐），但 UUID 唯一，此时可用 `domid <UUID>` 精准查询特定虚拟机的 ID。

#### 总结

`virsh domid` 的核心作用是查询运行中虚拟机的动态 ID，需注意它仅对运行状态的虚拟机有效。结合 `domname`（ID→名称）和 `domuuid`（名称→UUID），可实现虚拟机标识（名称、ID、UUID）之间的完整转换，方便各种管理场景。

### 6.3.24.虚拟机操作系统信息

### 6.3.25.虚拟机操作系统主机名

`virsh domhostname`

出现 "不支持的参数：未配置 QEMU 虚拟机代理" 错误，是因为 `virsh domhostname` 命令依赖 QEMU 虚拟机代理（QEMU Guest Agent，简称 QGA）来获取虚拟机内部的主机名，而你的虚拟机未安装或未启用该代理。

解决方法：

#### 1. 什么是 QEMU 虚拟机代理？

QGA 是一个运行在虚拟机内部的服务程序，用于在宿主机和虚拟机之间传递信息（如主机名、IP、磁盘信息等），支持宿主机通过 `virsh` 命令管理虚拟机内部状态。

#### 2. 如何配置 QGA？

需要在虚拟机内部和宿主机配置两方面操作：

步骤 1：在虚拟机内部安装 QGA

进入 fnos 虚拟机系统，根据操作系统类型安装 QGA：

Debian/Ubuntu：

```
bash
```

```
sudo apt-get install qemu-guest-agent  
sudo systemctl enable --now qemu-guest-agent #
```

启动并设置开机自启

RHEL/CentOS/Fedora：

```
bash
```

```
sudo yum install qemu-guest-agent  
sudo systemctl enable --now qemu-guest-agent
```

其他系统：参考对应发行版的包管理工具安装（通常包名为 qemu-guest-agent）。

步骤 2：在宿主机启用 QGA 配置

关闭虚拟机（若运行中）：

```
bash
```

```
virsh shutdown fnos
```

编辑虚拟机 XML 配置，添加 QGA 设备：

```
bash
```

```
virsh edit fnos
```

在 <devices> 标签内添加以下内容（通常放在其他设备后面）：

```
xml
```

```
<channel type='unix'>
```

```
  <target type='virtio' name='org.qemu.guest_agent.0' />
```

```
  <address type='virtio-serial' controller='0' bus='0' port='1' /></channel>
```

保存退出，启动虚拟机：

```
bash
```



virsh start fnos

3. 验证 QGA 是否生效

待虚拟机启动后，在宿主机执行：

bash

virsh domhostname fnos

若配置成功，会返回虚拟机内部的主机名。

也可通过以下命令检查 QGA 连接状态：

bash

virsh qemu-agent-command fnos '{"execute":"guest-ping"}'

若返回 {"return":{}}，表示 QGA 工作正常。

总结

domhostname 等依赖虚拟机内部信息的命令必须通过 QGA 实现。配置完成后，不仅能获取主机名，还能使用 virsh domifaddr（获取虚拟机 IP）等更强大的功能。

6.4.虚拟机生命周期管理

| 内容          | 命令             | 说明                                          |
|-------------|----------------|---------------------------------------------|
| 注册虚拟机       | virsh define   | 定义持久化虚拟机，定义完成后虚拟机处于关闭状态，虚拟机被看作为一个 domian 实例 |
| 注销虚拟机       | virsh undefine | 销毁持久性虚拟机，虚拟机生命周期完结，不能继续对该虚拟机继续操作            |
| 临时创建虚拟机     | virsh create   | 创建一个临时性虚拟机，创建完成后虚拟机处于运行状态                   |
| 启动虚拟机       | virsh start    | 启动虚拟机                                       |
| 关闭虚拟机       | virsh shutdown | 关闭虚拟机。启动虚拟机关机流程，若关机失败可使用强制关闭                |
| 强制关闭（断电）虚拟机 | virsh destroy  | 强制关闭虚拟机                                     |
| 重启虚拟机       | virsh reboot   | 重启虚拟机                                       |
| 暂停虚拟机（T 文件） | virsh save     | 将虚拟机的运行状态转储到文件中                             |
| 恢复虚拟机（F 文件） | virsh restore  | 从虚拟机状态转储文件恢复虚拟机                             |
| 待机虚拟机（T 内存） | virsh suspend  | 暂停虚拟机的运行，使虚拟机处于 paused 状态                   |

| 内容                 | 命令                          | 说明                                                                  |
|--------------------|-----------------------------|---------------------------------------------------------------------|
| 恢复待机虚拟机（F 内存）      | virsh resume                | 唤醒虚拟机，将处于 <b>paused</b> 状态的虚拟机恢复到运行状态                               |
| 休眠虚拟机<br>（自动保存到文件） | virsh managedsave           | 休眠虚拟机，虚拟机状态保存在硬盘中<br>解除休眠需要用 <b>virsh start</b> 命令，解除后，自动保存的文件会自动删除 |
| 删除休眠状态文件           | virsh<br>managedsave-remove | 删除虚拟机休眠状态的状态文件，相当于关机，数据会丢失                                          |
| 虚拟机自动启动            | virsh autostart             | 虚拟机自动启动                                                             |
| 虚拟机电源挂起            | virsh dompmsuspend          | 使用电源管理功能挂起域                                                         |
| 虚拟机电源唤醒            | virsh dompmwakeup           | 从 <b>Pmsuspended</b> 状态唤醒域                                          |

### 6.4.1.注册虚拟机

virsh define 命令详解

命令功能

将指定的 XML 配置文件“注册”为 libvirt 管理的持久性虚拟机域(Persistent Domain)。该操作仅完成“配置注册”，不会启动虚拟机，注册后的虚拟机将长期保留在 libvirt 管理列表中，即使主机或 libvirtd 服务重启，配置也不会丢失，后续可通过 virsh start 命令启动。

命令语法

bash

virsh define FILE [--validate]

参数说明

参数

功能描述

注意事项

|      |                                                                             |                                                    |
|------|-----------------------------------------------------------------------------|----------------------------------------------------|
| FILE | 必选参数，指定包含虚拟机配置的 XML 文件路径（需符合 libvirt 域配置格式，如 /etc/libvirt/qemu/ubuntu.xml）。 | XML 文件需正确定义域的核心信息（如名称、UUID、内存、CPU、磁盘、网络等），否则注册会失败。 |
|------|-----------------------------------------------------------------------------|----------------------------------------------------|

| 参数                                                                                              | 功能描述                                                                     | 注意事项                                                           |
|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|----------------------------------------------------------------|
|                                                                                                 |                                                                          | 等效于 <code>virt-xml-validate</code>                             |
|                                                                                                 | 可选参数，注册前先验证 XML 文件格式                                                     | FILE 工具，可提前发现 XML 的                                            |
| <code>--validate</code>                                                                         | 是否符合 libvirt 内部的 RNG (RELAX NG) schema 规范。                               | 语法错误（如标签缺失、属性无效、值超出范围等），避免注册后启动失败。                             |
| 核心特性：为何需要 <code>virsh define</code> ？                                                           |                                                                          |                                                                |
| <code>virsh define</code> 的核心价值是实现“虚拟机配置的持久化”，其关键特性与 <code>virsh create</code> （创建临时虚拟机）形成鲜明对比： |                                                                          |                                                                |
| 特性                                                                                              | <code>virsh define</code> （注册持久性虚拟机）                                     | <code>virsh create</code> （创建临时虚拟机）                            |
| 配置                                                                                              | 注册后，XML 配置会被持久化到 libvirt 配置目录（默认 <code>/etc/libvirt/qemu/</code> ），长期保留。 | 不保存配置，仅基于 XML 临时构造实例，实例销毁后配置“随实例消失”。                           |
| 虚拟机状态                                                                                           | 仅完成“注册”，虚拟机处于“未运行（inactive）”状态，需手动用 <code>virsh start</code> 启动。         | 创建后直接启动虚拟机（默认运行状态，除非加 <code>--paused</code> ），是“注册 + 启动”的组合操作。 |
| 生命周期                                                                                            | 配置长期有效，即使执行 <code>virsh destroy</code> （停止虚拟机），配置仍保留，可重复启动。              | 实例为“会话级”，销毁 / 重启后实例消失，需重新通过 XML 创建。                            |
| 适用场景                                                                                            | 用于长期使用的虚拟机（如生产环境服务、日常办公虚拟机）。                                             | 用于短期测试、临时验证 XML 配置的场景。                                         |

## 常见使用场景与示例

### 场景 1：注册新的持久性虚拟机

若有一个新编写的 XML 配置文件（如 `centos.xml`），需将其注册为长期可用的虚拟机：  
（可选）验证 XML 格式：先确保 XML 无语法错误，避免注册后启动失败：

```
bash
```

```
virsh define centos.xml --validate
```

若 XML 有效，输出类似：

```
plaintext
```

定义域 `centos`（从 `centos.xml`）

若 XML 无效（如缺少 <name> 标签），会输出错误信息（如 error: 解析 XML 时出错: 元素类型 "domain" 必须声明）。

注册虚拟机：若无需提前验证，直接执行注册：

bash

virsh define /etc/libvirt/qemu/centos.xml

注册成功后，可通过 virsh list --all 查看（状态为 inactive）：

plaintext

| Id    | Name   | State    |
|-------|--------|----------|
| ----- |        |          |
|       | centos | shut off |

启动注册后的虚拟机：注册后需手动启动，后续可重复启动 / 停止：

bash

# 启动虚拟机 virsh start centos

# 停止虚拟机（配置仍保留） virsh shutdown centos

场景 2：更新已注册虚拟机的配置

若已注册的虚拟机（如 ubuntu）需要修改配置（如增加内存、更换磁盘），步骤如下：

导出当前配置：先导出已注册的 XML 配置（避免直接修改原始文件导致格式错误）：

bash

virsh dumpxml ubuntu > ubuntu-new.xml

编辑 XML 配置：修改 ubuntu-new.xml 中的目标配置（如将内存从 2G 改为 4G）：

xml

<!-- 修改前 --><memory unit='KiB'>2097152</memory> <!-- 2G = 2\*1024\*1024 KiB

--><currentMemory unit='KiB'>2097152</currentMemory>

<!-- 修改后 --><memory unit='KiB'>4194304</memory> <!-- 4G = 4\*1024\*1024 KiB

--><currentMemory unit='KiB'>4194304</currentMemory>

重新注册（覆盖旧配置）：执行 define 命令用新 XML 覆盖旧配置（无需先删除旧注册，define 会自动更新）：

bash

```
virsh define ubuntu-new.xml --validate
```

生效配置：若虚拟机已运行，需重启后新配置才生效；若未运行，下次启动直接使用新配置：

```
bash
```

```
# 重启运行中的虚拟机（使新配置生效）virsh destroy ubuntu  
virsh start ubuntu
```

注意事项

域名唯一性：XML 中 `<name>` 标签指定的域名必须唯一，不可与已注册的虚拟机（包括 `inactive` 状态的）重名，否则注册会失败（报错“域 [名称] 已存在”）。若需重名，需先执行 `virsh undefine` 旧域名 删除旧注册。

UUID 建议：XML 中 `<uuid>` 标签建议指定唯一 UUID（可通过 `uuidgen` 命令生成），若省略，`define` 会自动生成 UUID；若手动指定，需确保与其他虚拟机不重复（否则可能导致管理冲突）。

配置目录默认路径：注册后，`libvirt` 会将 XML 配置文件自动保存到默认目录（如 QEMU/KVM 为 `/etc/libvirt/qemu/`，Xen 为 `/etc/libvirt/xen/`），后续修改配置建议通过 `virsh dumpxml` 导出后编辑，再重新 `define`，避免直接修改默认目录的文件（可能因格式错误导致虚拟机无法启动）。

权限要求：需以 `root` 或具有 `libvirt` 管理权限的用户执行（非 `root` 用户需在 `libvirt` 用户组中，且配置了访问控制），否则会因权限不足无法写入配置目录。

与 `virsh undefine` 的配合：若需删除已注册的持久性虚拟机，需执行 `virsh undefine` 域名（仅删除配置，不删除磁盘文件），删除后虚拟机从 `libvirt` 管理列表中消失，无法再用 `virsh start` 启动。

### 6.4.2.注销虚拟机

`virsh undefine` 是用于删除 `libvirt` 中已注册的持久性虚拟机域（Persistent Domain）配置的核心命令，作用是从 `libvirt` 管理列表中移除虚拟机的定义（包括 XML 配置文件、元数据等），但默认不会删除虚拟机的磁盘文件或快照数据。该命令仅针对“已定义但未运行”或“已停止运行”的持久性虚拟机，是彻底清理无用虚拟机配置的关键操作。

一、基本语法

bash

virsh undefine 域名/域 UUID [选项]

操作对象：必须是持久性虚拟机（通过 `virsh define` 注册的虚拟机），且状态为“已停止（shut off）”或“已暂停（paused）”（若虚拟机正在运行，需先执行 `virsh destroy` 停止），通过“域名”或“域 UUID”指定（推荐用域名，更直观）。

核心作用：删除虚拟机在 `libvirt` 中的注册信息（默认删除 `/etc/libvirt/qemu/` 等配置目录下的 XML 文件），使虚拟机不再被 `libvirt` 管理，后续无法通过 `virsh start` 启动（需重新 `define` 才能恢复）。

二、关键选项说明（常用且风险相关）

`virsh undefine` 命令的选项主要围绕“是否删除关联资源”（如磁盘、快照、日志等），部分选项具有数据破坏性，需谨慎使用。以下是实际场景中高频使用的选项：

| 选项    | 完整语法                 | 功能描述                                                                                                      | 注意事项                                                                   |
|-------|----------------------|-----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| (无选项) | virsh<br>undefine 域名 | 基础删除：仅删除虚拟机的配置（XML 文件、libvirt 元配置，保留用户数据（如虚拟数据），不删除磁盘文件、快照镜像文件），后续可通过 XML 重新 define 恢复                    | 最安全的用法，仅清理                                                             |
|       |                      | 强制删除虚拟机的“托管保存（managed save）镜像”：若虚拟机存在未删除的托管保存状态（如通过 virsh managedsave 创建），默认 undefine 会失败，加此选项可强制删除该状态及配置 | 1. 托管保存镜像存储虚拟机的运行状态（如内存数据）；2. 仅当 undefine 因“存在托管保存状态”报错时使用，无该状态则无需加此选项 |
|       |                      | 删除虚拟机的快照元数据：若虚拟机存在快照（如通过 virsh snapshot-create 创建），默认 undefine 会失败，加此选项可强制删除快照元数据                         | 1. 快照元数据是 libvirt 无法识别快照，但快照磁盘文件仍存在（需手动删除）；2. 仅适用于“已停                  |

| 选项                            | 完整语法                            | 功能描述                                                                                                                | 注意事项                                                                |
|-------------------------------|---------------------------------|---------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|
|                               |                                 | 数据（不删除快照的磁盘文件）                                                                                                      | 止运行”的虚拟机，运行中虚拟机需先 destroy                                           |
|                               |                                 | 删除虚拟机的检查点元数据                                                                                                        |                                                                     |
|                               | virsh                           | 数据：若虚拟机存在检查点                                                                                                        | 与 --snapshots-metadata                                              |
| --checkpoints-undefine        |                                 | （如通过 virsh                                                                                                          | 逻辑类似，仅针对检查点，                                                        |
| metadata                      | --checkpoints-checkpoint-create | 创建），默认不删除检查点关联的磁盘数据                                                                                                 |                                                                     |
|                               | metadata 域名                     | undefine 会失败，加此选项可强制删除检查点元数据                                                                                        |                                                                     |
|                               |                                 |                                                                                                                     | 1. 高风险选项：直接删除磁盘文件，数据无法恢复，删除虚拟机配置时，同时需确认卷列表无误；2. 仅支持 libvirt 管理的存储池中 |
| --storage-undefine            | virsh                           | 删除指定的存储卷（磁盘）：                                                                                                       | 的卷，非管理卷需手动删除；                                                       |
| 列表                            | undefine                        | 卷列表 为逗号分隔的“卷目标名”（如 vda）或“卷路径”                                                                                       | 3. 示例：--storage                                                     |
|                               | --storage                       | （如 /var/lib/libvirt/images/usb.img 域名                                                                               | vda,/var/lib/libvirt/images/ubuntu.qcow2 会同时删除这两个卷                  |
|                               |                                 | buntu.qcow2）                                                                                                        |                                                                     |
|                               |                                 |                                                                                                                     | 1. 极高风险选项：批量删除所有关联卷，适用于                                             |
|                               | virsh                           | 删除虚拟机配置时，同时“彻底清理虚拟机，无任何数据保留”的场景；2. 部分存储卷（包括所有磁盘、快照卷驱动（如 rbd）支持结合 --delete-storage-volume-snapshots 删除卷关联的快照，需额外加该选项 |                                                                     |
| --remove-all-storage-undefine |                                 | 删除该虚拟机关联的所有存储卷（包括所有磁盘、快照卷驱动（如 rbd）支持结合 --delete-storage-volume-snapshots 删除卷关联的快照，需额外加该选项                           |                                                                     |
|                               | virsh                           | 控制是否删除虚拟机的                                                                                                          | 1. 若虚拟机配置了                                                          |
| --nvram / --keep-nvram        | undefine                        | NVRAM 文件（存储 BIOS/UEFI 配置，如启动顺序                                                                                      | NVRAM（如 UEFI 启动的虚拟机），默认 undefine 会因                                 |
|                               | --nvram 域名                      |                                                                                                                     |                                                                     |

| 选项           | 完整语法          | 功能描述                              | 注意事项                                                           |
|--------------|---------------|-----------------------------------|----------------------------------------------------------------|
| 名 / virsh    | 序) : --nvram: | 强制删除 “NVRAM 文件存在” 报错,             |                                                                |
| undefine     | NVRAM 文件;     |                                   | 需显式                                                            |
| --keep-nvram | --keep-nvram: | 保留 NVRAM加 --nvram 或 --keep-nvram; |                                                                |
| 域名           | 文件            |                                   | 2. NVRAM 文件默认路径:<br>/var/lib/libvirt/qemu/nvram/<br>域名_VARS.fd |

三、典型使用示例

示例 1：基础用法（仅删除配置，保留磁盘数据）

删除名为 ubuntu 的持久性虚拟机配置，不删除其磁盘文件

（如 /var/lib/libvirt/images/ubuntu.qcow2）：

```
bash
efenye@efenye-deepin:~$ sudo virsh undefine ubuntu # 非 root 用户需加 sudo
定义域 ubuntu 被取消定义
验证结果：
通过 virsh list --all 查看，ubuntu 已从列表中消失；
配置目录（如 /etc/libvirt/qemu/）下的 ubuntu.xml 文件被删除；
磁盘文件 /var/lib/libvirt/images/ubuntu.qcow2 仍保留，后续可基于该文件重新 define 虚拟机。
```

示例 2：强制删除含托管保存状态的虚拟机

若 ubuntu 虚拟机存在托管保存状态（virsh managedsave 创建），直接 undefine 会报错，需加 --managed-save 强制删除：

```
bash
# 直接删除报错（存在托管保存状态）sudo virsh undefine ubuntu
error: 无法取消定义域 ubuntu
error: 域有托管保存状态; 请先使用 --managed-save 选项删除它
# 加选项强制删除 sudo virsh undefine --managed-save ubuntu
定义域 ubuntu 被取消定义
```

示例 3：删除配置并清理关联磁盘（高风险）

彻底删除 test-vm 虚拟机的配置及关联的 vda 磁盘（需确认数据无需保留）：



```
bash
```

```
sudo virsh undefine --storage vda test-vm
```

执行后：

test-vm 的配置被删除；

vda 对应的磁盘文件（如 /var/lib/libvirt/images/test-vm.qcow2）被删除，数据无法恢复。

示例 4：删除 UEFI 虚拟机（处理 NVRAM 文件）

对于 UEFI 启动的 centos 虚拟机，undefine 时需显式控制 NVRAM 文件（否则报错）：

```
bash
```

```
# 保留 NVRAM 文件（后续重建虚拟机可复用 BIOS 配置）sudo virsh undefine  
--keep-nvram centos
```

```
# 或删除 NVRAM 文件（彻底清理）sudo virsh undefine --nvram centos
```

#### 四、关键注意事项

操作前提：虚拟机必须已停止运行若虚拟机处于“运行中（running）”状态，直接执行 virsh undefine 会报错，需先执行 virsh destroy 域名 停止虚拟机：

```
bash
```

```
# 先停止运行中的虚拟机 sudo virsh destroy ubuntu# 再删除配置 sudo virsh undefine  
ubuntu
```

区分“临时域”与“持久性域”

virsh undefine 仅对持久性域有效（通过 virsh define 注册的虚拟机）；

对于“临时域”（通过 virsh create 创建的虚拟机），无需 undefine，执行 virsh destroy 后实例和配置会自动消失，undefine 会报错“域未定义”。

数据安全：默认不删除磁盘，谨慎使用存储相关选项

无选项时，undefine 仅删除配置，磁盘、快照等数据仍保留，是最安全的方式；

若需删除磁盘，必须通过 --storage 或 --remove-all-storage 显式指定，且删除前需确认数据无需保留（建议先备份关键数据）。

与“destroy”的核心区别很多用户混淆 virsh undefine 与 virsh destroy，两者本质差异如下：

| 命令                    | 作用对象      | 核心操作                             | 对配置的影响                                                      | 对数据的影响                 |
|-----------------------|-----------|----------------------------------|-------------------------------------------------------------|------------------------|
| <code>virsh</code>    | 运行中 / 虚拟机 | 强制停止虚拟机，释放 CPU、内存资源              | 不删除任何配置，持久性域仍可通过 <code>start</code> 重新启动                    | 不删除磁盘、快照数据，仅丢失未保存的内存数据 |
| <code>destroy</code>  | 暂停的虚拟机    | 删除虚拟机在 <code>libvirt</code> 中的配置 | 彻底删除配置，无法通过 <code>start</code> 启动（需重新 <code>define</code> ） | 默认不删除数据，需显式选项才删除磁盘     |
| <code>virsh</code>    | 已停止的持久性域  |                                  |                                                             |                        |
| <code>undefine</code> |           |                                  |                                                             |                        |

恢复方式：删除后如何重建虚拟机若误执行 `virsh undefine`，只要磁盘文件未删除，可通过以下步骤重建虚拟机：

# 1. 基于保留的磁盘文件创建新的 XML 配置（或复用旧 XML 模板）# 2. 重新注册为持久性域 `sudo virsh define /path/to/new-ubuntu.xml` # 3. 启动虚拟机 `sudo virsh start ubuntu`

### 6.4.3.临时创建虚拟机

`virsh create` 命令详解

命令功能

从指定的 XML 文件创建并启动一个虚拟机域（domain）。创建的域可分为两种类型：

临时域（Transient Domain）：创建后仅在当前运行会话中存在，一旦销毁（如执行 `destroy` 命令或主机重启），域的配置和状态将丢失。

基于持久性域的临时运行实例：若 XML 文件对应的域是已定义的持久性域，使用此命令将以“一次性配置”启动该域，不会修改原始的持久性 XML 配置（适用于基于原始配置进行自动化测试等场景）。

命令语法

bash

```
virsh create FILE [--console] [--paused] [--autodestroy] [--pass-fds N,M,...] [--validate]
[--reset-nvram]
```

参数说明

| 参数   | 功能描述                 | 注意事项            |
|------|----------------------|-----------------|
| FILE | 必选参数，指定包含域配置的 XML 文件 | XML 文件需正确定义域的基本 |

| 参数                              | 功能描述                                                                                                        | 注意事项                                                                                                                        |
|---------------------------------|-------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
|                                 | XML 文件路径（XML 需符合 libvirt 信息（如名称、UUID、内存、CPU、域配置格式，可通过 <code>dumpxml</code> 命令磁盘、网络等），否则创建会失败。导出现有域的配置作为模板）。 |                                                                                                                             |
| <code>--console</code>          | 可选参数，创建域后自动附加到其虚拟串行控制台（类似物理机的串口终端）。                                                                         | 若需退出控制台，默认使用 <code>^J</code> （ <code>Ctrl + J</code> ）快捷键；可通过 <code>virsh console</code> 命令的 <code>--escape</code> 选项修改快捷键。 |
| <code>--paused</code>           | 可选参数，创建域后将其置于“暂停（ <code>paused</code> ）”状态，而非直接运行。                                                          | 需虚拟机监控程序（如 QEMU/KVM）支持，暂停后可通过 <code>virsh resume</code> 命令恢复运行。                                                             |
| <code>--autodestroy</code>      | 可选参数，当 <code>virsh</code> 关闭与 <code>libvirt</code> 的连接（如退出 <code>virsh</code> 交互模式）或异常退出时，自动销毁该域。后自动清理临时域）。 | 仅对当前创建的域生效，用于避免“孤儿域”残留（如脚本执行完毕式）。                                                                                           |
| <code>--pass-fds N,M,...</code> | 可选参数，将主机上指定的文件描述符（以逗号分隔，如 <code>3,5</code> ）传递到客户机中，客户机内从文件描述符 <code>3</code> 开始重新编号。                       | 仅支持基于容器的虚拟化（如 LXC），QEMU/KVM 等全虚拟化技术不支持此参数。                                                                                  |
| <code>--validate</code>         | 可选参数，创建前先验证 XML 文件的格式是否符合 <code>libvirt</code> 内部的具验证 RNG（RELAX NG） schema 规范。                              | 等效于使用 <code>virt-xml-validate</code> 工具验证 XML，可提前发现配置文件的语法错误（如标签缺失、属性无效等）。                                                  |
| <code>--reset-nvram</code>      | 可选参数，删除域对应的现有 NVRAM（非易失性随机存取存储器）BIOS/UEFI 配置（如启动顺序、硬件文件，并从原始模板重新初始化 NVRAM。                                  | NVRAM 通常存储虚拟机的参数），此参数适用于需重置固件配置的场景。                                                                                         |

## 常见使用场景与示例

### 场景 1：创建临时域（从现有域模板修改）

导出现有域的 XML 模板：若需基于已有的域（如名为 `fedora` 的域）创建临时域，先导出其配置作为模板：

```
bash
```

```
virsh dumpxml fedora > temp-domain.xml
```

修改 XML 模板：编辑 temp-domain.xml，必须修改 <name>（域名）和 <uuid>（可直接删除 <uuid> 标签，libvirt 会自动生成新 UUID），避免与现有域冲突：

```
xml
```

```
<!-- 修改前
```

```
--><name>fedora</name><uuid>cece4f9f-dff0-575d-0e8e-01fe380f12ea</uuid>
```

```
<!-- 修改后 --><name>fedora-temp</name><!-- 删除或保留 <uuid>（保留则需手动修改为新 UUID） -->
```

创建并启动临时域：使用修改后的 XML 创建域，若需自动附加控制台，添加 --console 参数：

```
bash
```

```
virsh create temp-domain.xml --console
```

输出示例（成功创建）：

```
plaintext
```

```
Domain 'fedora-temp' created from temp-domain.xml
```

```
Connected to domain 'fedora-temp'
```

```
Escape character is ^]
```

场景 2：创建暂停状态的域

若需创建域后暂不运行（如后续手动配置网络后再启动），使用 --paused 参数：

```
bash
```

```
virsh create test-domain.xml --paused
```

创建后可通过 virsh list --all 验证状态（状态为 paused）：

```
plaintext
```

| Id | Name | State |
|----|------|-------|
|----|------|-------|

-----

|   |             |        |
|---|-------------|--------|
| 5 | test-domain | paused |
|---|-------------|--------|

后续通过 virsh resume test-domain 恢复运行。

场景 3：验证 XML 并创建域

若需确保 XML 配置无语法错误，先验证再创建：

```
bash
```

`virsh create new-domain.xml --validate`

若 XML 格式无效，会输出错误信息（如标签缺失）；若有效，将直接创建域。

注意事项

权限要求：大多数虚拟化技术（如 QEMU/KVM、Xen）要求以 root 权限执行 `virsh create`，非 root 用户会因权限不足失败（可通过配置 `libvirt` 访问控制放宽权限，需谨慎操作）。

XML 配置规范：

域名（`<name>`）需唯一，不可与现有域（包括 `inactive` 状态的域）重名。

磁盘、网络等资源需在主机上真实存在（如 XML 中指定的磁盘文件路径需有效），否则域启动会失败。

临时域的生命周期：临时域销毁后，其配置和运行状态会丢失，但磁盘文件（如虚拟机镜像）不会被删除；若需长期保留域，需使用 `virsh define` 命令将其定义为持久性域。

NVRAM 重置的影响：`--reset-nvram` 会清除虚拟机的 BIOS/UEFI 配置（如启动顺序、密码），适用于解决固件配置异常的场景，正常创建无需使用。

6.4.4.启动虚拟机

你提供的是 `virsh start` 命令的帮助信息，这个命令用于启动一个已定义但处于非活跃状态的虚拟机（域）。以下是对各参数的详细解释，帮助你更好地使用这个命令：

| 参数                          | 作用与使用场景                                                                                                          |
|-----------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>&lt;domain&gt;</code> | 必须指定的参数，为虚拟机的名称、UUID 或 ID（非活跃状态的虚拟机 ID 通常为 -）。示例： <code>virsh start my-vm</code> （启动名为 <code>my-vm</code> 的虚拟机）。 |
| <code>--console</code>      | 启动虚拟机后立即连接到其控制台（类似物理机的显示器输出）适合需要查看启动过程日志（如 BIOS、内核启动信息）的场景。<br>示例： <code>virsh start my-vm --console</code> 。    |
| <code>--paused</code>       | 启动虚拟机后使其保持“暂停”状态（CPU 不运行，但内存和状态已加载），需后续用 <code>virsh resume</code> 恢复运行。适合需要预先加载虚拟机但暂不运行的场景（如资源预留）。              |
| <code>--autodestroy</code>  | 当 <code>virsh</code> 连接断开时（如关闭终端、退出 <code>virsh</code> 交互模式），自动销毁该虚拟机（强制关机）。适合临时启动的虚拟机，避免忘记清理残留进程。               |

| 参数                  | 作用与使用场景                                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------------------|
| --bypass-cache      | 启动时绕过文件系统缓存，直接从磁盘读取虚拟机镜像文件，适合大型镜像启动时减少对系统缓存的占用（可能略微降低启动速度，但不影响后续运行）。                                                  |
| --force-boot        | 强制从全新状态启动虚拟机，忽略之前的 managedsave 状态（managedsave 类似“休眠”，保存了虚拟机运行状态）。示例：若虚拟机上次通过 virsh managedsave 休眠，使用此参数会强制重启而非恢复休眠状态。 |
| --pass-fds <string> | 向虚拟机传递宿主机器的文件描述符（如特定设备、管道），通常用于高级调试或定制化场景（需配合虚拟机配置文件，一般用户很少用到）。示例：--pass-fds 3,4（传递文件描述符 3 和 4）。                      |
| --reset-nvram       | 从初始模板重新初始化虚拟机的 NVRAM（非易失性内存，类似物理机的 BIOS/UEFI 设置），适合解决因 NVRAM 配置错误导致的启动故障（如 UEFI 启动项损坏）。                               |

### 6.4.5.关闭虚拟机

```
[root@efenye-euler ~]# virsh shutdown --domain fnos --mode
acpi      agent      initctl    paravirt   signal
```

### 6.4.6.强制关闭（断电）虚拟机

| 项        | 完整语法                        | 功能描述                                                                                       | 注意事项                                                                   |
|----------|-----------------------------|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| --domain | virsh destroy               | 显式指定要销毁的虚拟机（与直接写“virsh destroy 域名”功能完全一致，属于冗余选项，仅用于明确标识操作对象，增强命令可读性）                       | 示例：virsh destroy --domain ubuntu 等效于 virsh destroy ubuntu，无特殊限制，所有场景通用 |
|          | --domain 域名/ID/UUID         |                                                                                            |                                                                        |
|          | --graceful                  | 启用“优雅销毁”模式：先尝试向客户机发送优雅关闭信号（类似 virsh shutdown），若超时（时间由虚拟机监控程序决定，通常几秒到十几秒）仍未关闭，项，直接强制销毁；2. 若 | 1. 仅部分虚拟机监控程序支持（如 QEMU/KVM、Xen），不支持                                    |
|          | virsh destroy --graceful 域名 |                                                                                            |                                                                        |

| 项 | 完整语法                                        | 功能描述                                                                                                                      | 注意事项                                                                                                                           |
|---|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
|   |                                             | 再强制终止                                                                                                                     | 客户机操作系统无响应（如死机），即使加此选项，最终仍会强制终止；                                                                                               |
|   |                                             |                                                                                                                           | 3. 与直接 <code>virsh shutdown</code> 不同：<br><code>shutdown</code> 仅发送优雅信号，超时后不会强制终止；<br><code>destroy --graceful</code> 超时后会强制终止 |
|   |                                             |                                                                                                                           | 1. 仅部分部署配置支持（如 QEMU 需使用 <code>virlogd</code> 服务处理日志，                                                                           |
|   |                                             | 销毁虚拟机的同时，删除该虚拟机对应的日志文件（日志路径由 <code>libvirt</code> 配置决定，默认如 <code>/var/log/libvirt/qemu/</code> 目录下，删除“虚拟机相关的文件名为“域名.log”） | 否则此选项会被忽略，不删除日志）；2. 谨慎使用：日志可用于排查虚拟机故障，删除后无法恢复；3. 仅 <code>libvirt</code> 日志”，不删除客户机内部的日志（如客户机操作系统的 <code>/var/log/</code> 日志）   |
|   | <code>virsh destroy --remove-logs 域名</code> |                                                                                                                           |                                                                                                                                |

### 三、常见使用示例

示例 1：基础用法（直接销毁虚拟机）

销毁名为 `ubuntu` 的运行中虚拟机，无额外操作：

```
bash
```

```
efenye@efenye-deepin:~$ sudo virsh destroy ubuntu # 非 root 用户需加 sudo（或有 libvirt 权限）
```

域 `ubuntu` 被销毁

执行后，虚拟机立即停止，通过 `virsh list` 查看，已无该域；通过 `virsh list --all` 查看，状态变为 `shut off`（仅持久性域会显示，临时域会直接消失）。

示例 2：显式指定域（`--domain` 选项）

场景：当命令参数较多时，用 `--domain` 明确标识虚拟机，增强可读性（如结合其他选项时）：

```
bash
```

```
sudo virsh destroy --graceful --domain ubuntu
```

功能：先尝试优雅关闭 `ubuntu`，超时后强制销毁，等效于 `sudo virsh destroy --graceful ubuntu`。

示例 3：优雅销毁 + 删除日志

场景：销毁虚拟机时，先尝试优雅关闭，同时删除日志（需监控程序支持）：

```
bash
```

```
sudo virsh destroy --graceful --remove-logs ubuntu
```

执行逻辑：先发送优雅关闭信号 → 超时未关则强制终止 → 终止后删除该虚拟机的 `libvirt` 日志文件。

#### 四、关键注意事项

权限要求：非 `root` 用户需以 `sudo` 执行（如你的 `efenye` 用户），或提前配置 `libvirt` 访问控制（将用户加入 `libvirt` 组，并确保组权限生效），否则会报错“权限被拒绝”。

对临时域和持久性域的影响不同：

临时域（`virsh create` 创建）：销毁后，实例和配置完全消失，需重新通过 XML 用 `virsh create` 重建；

持久性域（`virsh define` 注册）：销毁后，仅实例停止，配置仍保留，后续可通过 `virsh start` 域名重新启动。

数据风险：无论是否加 `--graceful`，`destroy` 本质是“强制终止”（`--graceful` 仅多一步优雅尝试），若客户机有未保存的文件（如编辑中的文档、未提交的数据库事务），可能导致数据丢失或损坏，生产环境优先使用 `virsh shutdown` 优雅关闭，仅在紧急情况（如客户机死机）使用 `destroy`。

日志删除的限制：若未使用 `virlogd` 服务（如默认的 `QEMU` 日志配置），`--remove-logs` 会被忽略，需手动删除日志（如 `sudo rm /var/log/libvirt/qemu/ubuntu.log`）。



### 6.4.7.重启虚拟机

```
[root@efenye-euler ~]# virsh reboot --domain fnos --mode  
acpi      agent      initctl    paravirt   signal
```

### 6.4.8.暂停虚拟机 (T 文件)

`virsh save` 是 KVM 虚拟化环境中用于手动保存虚拟机状态到指定文件的命令，与 `virsh restore` 配合形成“保存 - 恢复”工作流，适用于需要手动控制状态文件位置和管理场景。以下是详细说明：

#### 一、基本功能

将指定虚拟机的当前运行状态（包括内存数据、进程状态、网络连接等）完整保存到用户指定的文件中，同时关闭虚拟机（释放 CPU / 内存资源）。后续可通过 `virsh restore` 命令从该文件恢复虚拟机至保存时的状态。

#### 二、语法与常用参数

bash

```
virsh save <虚拟机名称/UUID> <状态文件路径> [选项]
```

常用参数：

`--bypass-cache`：保存时绕过文件系统缓存，直接写入磁盘，确保状态文件即时持久化（适合对数据安全性要求高的场景）。

`--running`：恢复时默认使虚拟机处于运行状态（此参数影响后续 `restore` 的默认行为）。

`--paused`：恢复时默认使虚拟机处于暂停状态（需手动 `resume` 启动）。

`--verbose`：显示保存过程的详细日志（如内存数据写入进度）。

`--live`：在不中断虚拟机运行的情况下保存状态（热备份，虚拟机持续运行直至保存完成）。

#### 三、使用示例

基础保存将名为 `centos7` 的虚拟机状态保存到 `/backup/centos7_state`：

bash

```
virsh save centos7 /backup/centos7_state
```

热备份（不中断运行）在虚拟机运行中后台保存状态，完成后自动关闭虚拟机：

bash

```
virsh save --live centos7 /backup/centos7_live_state
```

详细模式保存显示保存过程的详细信息，便于调试：

```
bash
```

```
virsh save --verbose centos7 /backup/centos7_state
```

指定恢复后状态保存时指定恢复后虚拟机默认处于暂停状态：

```
bash
```

```
virsh save --paused centos7 /backup/centos7_paused_state# 后续恢复时会自动处于暂停
```

```
状态 virsh restore /backup/centos7_paused_state
```

四、与 managedsave 的核心区别

| 对比维度   | virsh save                                      | virsh managedsave                                     |
|--------|-------------------------------------------------|-------------------------------------------------------|
| 状态文件位置 | 需手动指定路径<br>(如 /backup/xxx)                      | 由 libvirt 自动管理 (默认路径：<br>/var/lib/libvirt/qemu/save/) |
| 恢复方式   | 必须用 <code>virsh restore &lt;文件路径&gt;</code>     | 直接用 <code>virsh start &lt;vm&gt;</code> 即可自动恢复        |
| 适用场景   | 手动备份、迁移状态文件、自定义存储路径                             | 快速休眠 / 恢复，无需管理状态文件位置                                  |
| 文件管理责任 | 用户需手动删除状态文件<br>( <code>rm &lt;文件路径&gt;</code> ) | 自动由 <code>managedsave-remove</code> 命令管理              |

五、注意事项

状态文件大小约等于虚拟机当前占用的内存量，需确保目标路径有足够磁盘空间。

保存过程中虚拟机可能短暂卡顿（非 --live 模式），建议在低负载时执行。

若保存中断，可能导致状态文件损坏，此时需删除损坏文件并重新启动虚拟机。

状态文件与虚拟机绑定，不可用于恢复其他虚拟机。

### 6.4.9.恢复虚拟机 (F 文件)

`virsh restore` 是用于从之前保存的状态文件中恢复虚拟机运行状态的命令，通常与 `virsh save` 配合使用（`save` 保存状态到文件，`restore` 从文件恢复）。它适用于需要临时保存虚拟机状态并在后续精确恢复的场景，以下是详细说明：

#### 一、基本功能

从指定的状态文件（由 `virsh save` 生成）中加载虚拟机的完整运行状态（包括内存数据、进程、网络连接等），使虚拟机恢复到保存时的精确状态，而无需正常启动。

#### 二、语法与常用参数

bash

`virsh restore <状态文件路径> [选项]`

常用参数：

`--bypass-cache`：恢复时绕过文件系统缓存，直接从磁盘读取状态文件，确保数据即时生效（适合对数据一致性要求高的场景）。

`--running`：恢复后虚拟机直接进入运行状态（默认行为）。

`--paused`：恢复后虚拟机处于暂停状态，需手动执行 `virsh resume` 使其运行。

`--verbose`：显示恢复过程的详细日志，便于调试。

#### 三、使用示例

基础恢复（默认运行状态）从 `/tmp/ubuntu_save` 文件恢复虚拟机：

bash

`virsh restore /tmp/ubuntu_save`

恢复后保持暂停状态恢复后虚拟机处于暂停状态，需手动恢复运行：

bash

`virsh restore /tmp/ubuntu_save --paused# 恢复运行 virsh resume ubuntu-vm`

详细模式恢复显示恢复过程的详细信息：

bash

`virsh restore /tmp/ubuntu_save --verbose`

#### 四、注意事项

状态文件关联：恢复的状态文件必须是通过 `virsh save <vm> <文件路径>` 生成的，且与目标虚拟机一一对应，不可混用。

虚拟机状态要求：恢复前，目标虚拟机必须处于关闭（`shut off`）状态，否则会提示错误。

文件完整性：若状态文件损坏或不完整，恢复会失败，此时需重新启动虚拟机（`virsh start`）。

资源需求：恢复时需要足够的内存资源（与保存时虚拟机占用的内存相当），否则可能因内存不足导致失败。

与 `managedsave` 的区别：

`virsh restore` 依赖手动指定的状态文件，适用于临时备份 / 恢复场景；

`managedsave` 保存的状态由 `libvirt` 自动管理，恢复时直接用 `virsh start` 即可（无需指定文件）。

#### 6.4.10.待机虚拟机（T 内存）

`virsh suspend` 是用于暂停运行中虚拟机域（`domain`）的命令，暂停后虚拟机将停止占用 CPU 资源（不再被调度执行），但仍会保留已分配的内存、磁盘等资源，相当于物理机的“暂停”状态（并非关闭），后续可通过 `virsh resume` 命令恢复运行。以下是完整的命令用法、选项说明及注意事项

#### 6.4.11.恢复待机虚拟机（F 内存）

`virsh resume` 是用于恢复被暂停（`paused`）的虚拟机域（`domain`）的核心命令，作用是将处于“暂停”状态的虚拟机重新纳入虚拟机监控程序的调度队列，使其恢复 CPU 资源占用和正常运行，且恢复后虚拟机将完全保留暂停前的运行状态（如未保存的文件、打开的进程、网络连接上下文等），相当于物理机的“继续”操作。

#### 6.4.12.休眠虚拟机（自动保存到文件）

功能

将指定虚拟机（<vm> 为虚拟机名称或 UUID）的当前运行状态（包括内存数据、进程状态等）保存到持久化存储中，然后自动关闭虚拟机。这类似于“休眠”功能——状态保存后，虚拟机不再占用 CPU 和内存资源，但下次启动时可直接从保存的状态恢复，无需重新初始化系统和应用。

#### 特点

状态完整保留：所有运行中的程序、打开的文件、网络连接等都会被冻结并保存。

快速恢复：恢复时直接加载保存的状态，比正常开机快得多（时间取决于虚拟机内存大小）。

自动释放资源：保存完成后，虚拟机会自动关闭，释放物理机的 CPU 和内存资源。

#### 使用示例

bash

```
# 保存名为 "ubuntu-server" 的虚拟机状态并关闭 virsh managedsave ubuntu-server
```

#### 恢复方法

要从保存的状态恢复虚拟机，直接启动即可：

bash

```
virsh start ubuntu-server # 会自动检测并恢复到保存的状态
```

### 6.4.13.删除休眠虚拟机状态文件

#### 2. virsh managedsave-remove <vm>

##### 功能

删除通过 managedsave 命令保存的虚拟机持久化状态文件，使虚拟机回到“未保存状态”。执行后，下次启动虚拟机时将按照正常流程开机（而非恢复之前的保存状态），同时释放状态文件占用的磁盘空间。

##### 适用场景

保存的状态不再需要，想释放磁盘空间（状态文件大小约等于虚拟机已分配的内存）。

保存的状态文件损坏，导致无法正常恢复时，清除后可重新正常启动虚拟机。

希望虚拟机下次启动时“全新启动”，而非恢复之前的运行状态。

#### 使用示例

bash

```
# 取消名为 "ubuntu-server" 的虚拟机的持久化保存状态 virsh managedsave-remove
ubuntu-server
```

注意事项

执行 `managedsave` 前，确保虚拟机处于运行状态（`running`），否则命令会失败。

状态文件默认存储在 `libvirt` 的配置目录（通常为 `/var/lib/libvirt/qemu/save/`），`managedsave-remove` 会自动清理该文件。

若虚拟机已通过 `managedsave` 保存状态，使用 `virsh list --all` 查看时，其状态会显示为 `saved`。

通过这两个命令，可以灵活管理虚拟机的暂停与恢复，平衡资源占用和使用效率。

你列出的是 `virsh managedsave` 命令的可选参数，这些参数用于调整状态保存的行为。

以下是各参数的详细说明：

`--bypass-cache` 保存状态时绕过文件系统缓存，直接写入磁盘。适用于需要确保状态文件立即持久化到物理存储的场景（如避免系统崩溃导致状态文件损坏），但可能略微降低保存速度。

示例：

```
bash
```

```
virsh managedsave --bypass-cache ubuntu-vm
```

`--domain <vm>` 显式指定要操作的虚拟机（`<vm>` 为名称或 UUID），与直接在命令后写虚拟机名功能相同，主要用于脚本中参数化操作。

示例：

```
bash
```

```
virsh managedsave --domain ubuntu-vm
```

`--paused` 保存状态后，虚拟机恢复时默认处于暂停状态（而非直接运行），需手动执行 `virsh resume` 恢复运行。适用于需要先检查状态再启动的场景。

示例：

```
bash
```

```
virsh managedsave --paused ubuntu-vm# 恢复后虚拟机处于暂停状态 virsh start
ubuntu-vm # 此时状态为 pausedvirsh resume ubuntu-vm # 手动恢复运行
```

`--running` 保存状态后，虚拟机恢复时默认处于运行状态（默认行为），与直接执行 `virsh start` 效果一致，可省略。

示例：

bash

`virsh managedsave --running ubuntu-vm` # 等效于不加此参数

`--verbose` 显示详细的操作过程日志，便于调试或了解保存状态的进度（如内存数据写入情况）。

示例：

bash

`virsh managedsave --verbose ubuntu-vm`

这些参数可组合使用，例如：

bash

# 详细模式保存状态，绕过缓存，且恢复后默认暂停 `virsh managedsave --verbose --bypass-cache --paused ubuntu-vm`

通过 `virsh help managedsave` 可查看完整的参数说明。

#### 6.4.14.虚拟机自动启动

在 `virsh` 中，`autostart` 是用于设置虚拟机开机自启动的核心功能（也可用于取消自启动），即宿主机启动时，指定的虚拟机自动随之启动，无需手动执行 `virsh start`。以下是完整用法、参数说明及操作示例：

一、基本语法

`autostart` 是 `virsh` 的独立子命令，核心语法分两类：

设置虚拟机自启动

bash

`virsh autostart <虚拟机名称/UUID>`

取消虚拟机自启动

bash

```
virsh autostart --disable <虚拟机名称/UUID>
```

## 二、关键参数说明

| 参数 | 功能描述 |
|----|------|
|----|------|

|              |                                                                   |
|--------------|-------------------------------------------------------------------|
| <虚拟机名称/UUID> | 必需参数，指定目标虚拟机（通过名称或 UUID 标识，可通过 <code>virsh list --all</code> 查看）。 |
|--------------|-------------------------------------------------------------------|

|                        |                                        |
|------------------------|----------------------------------------|
| <code>--disable</code> | 可选参数，用于取消自启动（若虚拟机已设置自启动，添加此参数后会关闭自启动）。 |
|------------------------|----------------------------------------|

|                     |                                              |
|---------------------|----------------------------------------------|
| <code>--help</code> | 查看 <code>autostart</code> 命令的帮助信息（包含所有可选参数）。 |
|---------------------|----------------------------------------------|

## 三、常用操作示例

### 1. 查看虚拟机当前自启动状态

先确认目标虚拟机是否已开启自启动，可通过 `virsh dominfo` 查看：

```
bash
```

```
# 查看 "ubuntuv" 的详细信息，包含自启动状态 virsh dominfo ubuntuv
```

输出中会有一行 `Autostart: yes/no`，`yes` 表示已开启自启动，`no` 表示未开启。

### 2. 开启虚拟机自启动

为指定虚拟机配置宿主机开机自启：

```
bash
```

```
# 让 "ubuntuv" 随宿主机自动启动 virsh autostart ubuntuv
```

执行成功后，无额外输出（`virsh` 多数命令“无报错即成功”）。此时通过 `virsh dominfo ubuntuv` 会看到 `Autostart: yes`。

### 3. 取消虚拟机自启动

若需关闭已配置的自启动（宿主机启动时不再自动启动该虚拟机）：

```
bash
```

```
# 取消 "ubuntuv" 的自启动 virsh autostart --disable ubuntuv
```

同样无额外输出，通过 `virsh dominfo` 可验证 `Autostart` 已变为 `no`。

### 4. 查看所有已设置自启动的虚拟机

若想知道宿主机上有哪些虚拟机开启了自启动，可通过以下命令（本质是查看自启动配置文件目录）：

```
bash
```



# 自启动虚拟机的配置文件默认存放在 `/etc/libvirt/qemu/autostart/ls`  
`/etc/libvirt/qemu/autostart/`

输出的 .xml 文件名（去掉后缀）即为已开启自启动的虚拟机名称。

#### 四、原理说明

`virsh autostart` 的本质是在宿主机的 `libvirt` 配置目录中创建符号链接：

开启自启动时：将虚拟机的配置文件（默认路径 `/etc/libvirt/qemu/<虚拟机名>.xml`）链接到自启动目录 `/etc/libvirt/qemu/autostart/`。

取消自启动时：删除 `/etc/libvirt/qemu/autostart/` 中对应的符号链接。

宿主机启动时，`libvirt` 服务会扫描 `autostart` 目录，自动启动所有存在配置链接的虚拟机。

#### 五、注意事项

虚拟机状态不影响配置：无论虚拟机当前是“运行中(`running`)”还是“关闭(`shut off`)”，都可执行 `autostart` 配置（配置仅影响宿主机下次启动时的行为）。

权限要求：需使用 `root` 用户或属于 `libvirt` 组的用户执行（普通用户无权限修改自启动配置）。

兼容性：所有主流虚拟化类型（`KVM/QEMU`、`Xen` 等）均支持此命令，无虚拟化平台限制。

#### 总结

`autostart` 是虚拟机运维的常用命令，核心作用是“一键配置 / 取消开机自启”，配合 `virsh dominfo` 和目录查看，可完整管理虚拟机的自启动行为，适合需要长期稳定运行的服务型虚拟机（如数据库、Web 服务虚拟机）。

### 6.4.15. 虚拟机电源挂起

`virsh dompm_suspend` 命令用于将虚拟机（域）置于电源管理挂起状态，类似物理机的休眠或睡眠模式，可节省资源同时保持虚拟机状态。其参数说明如下：

#### 参数详细说明

1. `--domain <虚拟机名/ID/UUID>`

必填参数，指定要挂起的目标虚拟机。示例：

`bash`

`virsh dompm_suspend --domain ubuntuvm # 对 ubuntuvm 执行电源挂起`

## 2. --target <挂起类型>

指定虚拟机的挂起模式（电源管理目标状态），不同模式对应不同的电源状态，常用值包括：

**mem**：内存挂起（类似休眠），将虚拟机状态保存到内存，功耗低，恢复速度快（断电会丢失状态）。

**disk**：磁盘挂起（类似休眠到硬盘），将虚拟机状态保存到磁盘，断电不丢失状态，但恢复速度较慢。

**hybrid**：混合挂起，同时保存状态到内存和磁盘，兼顾速度和安全性（部分系统支持）。

示例：

bash

```
# 将虚拟机挂起至内存（快速恢复）virsh dompm_suspend --domain ubuntuvm --target mem

# 将虚拟机挂起至磁盘（断电安全）virsh dompm_suspend --domain ubuntuvm --target disk
```

## 3. --duration <秒数>

指定挂起的持续时间，到期后虚拟机会自动恢复运行（部分系统和挂起模式支持）。若不指定，虚拟机将保持挂起状态，直至手动恢复（`virsh dompm_wakeup`）。

示例：

bash

```
# 挂起虚拟机至内存，10 分钟（600 秒）后自动恢复 virsh dompm_suspend --domain ubuntuvm --target mem --duration 600
```

使用前提

虚拟机必须处于运行状态（`running`）。

虚拟机操作系统需支持 **ACPI** 电源管理（现代系统通常默认支持）。

部分挂起模式（如 **disk**）可能需要虚拟机内安装 **qemu-guest-agent** 以确保状态正确保存。

恢复挂起的虚拟机

挂起后，可通过以下命令手动唤醒：

bash

注意事项

**--target mem** 依赖内存供电，若宿主机断电，虚拟机状态会丢失；**--target disk** 更安全但恢复慢。

并非所有虚拟机系统都支持所有挂起模式（如某些轻量 OS 可能不支持 hybrid），需测试验证。

挂起期间，虚拟机的网络连接会中断，恢复后需重新建立连接。

总结

`virsh dompm_suspend` 适合需要临时释放资源但保留虚拟机状态的场景，通过 `--target` 可选择不同的挂起策略，`--duration` 则支持自动恢复，配合 `dompm_wakeup` 可灵活管理虚拟机的电源状态。

#### 6.4.16. 虚拟机电源唤醒

```
virsh dompm_wakeup <虚拟机名/ID/UUID> # 唤醒挂起的虚拟机
```

### 6.5. 虚拟机镜像管理

**qemu-img:** QEMU 的磁盘镜像管理工具，支持创建、转换、检查、修改磁盘镜像文件等操作。常用功能包括创建不同格式的镜像（如 `qcow2`、`raw`）、调整镜像大小、合并快照等。

**qemu-io:** 用于对磁盘镜像或块设备进行底层 I/O 操作的工具，可模拟读写请求，常用于测试磁盘性能或调试存储相关问题。

**qemu-nbd:** 将 QEMU 磁盘镜像通过 NBD（Network Block Device）协议导出为网络块设备，允许远程主机像访问本地磁盘一样使用该镜像。

**qemu-pr-helper:** 处理 SCSI Persistent Reservations（持久预留）的辅助程序，用于支持集群环境中对共享存储的访问控制。

**qemu-storage-daemon:** QEMU 的存储守护进程，提供块设备服务，支持多种存储后端和协议，可作为独立的存储服务器使用。

**qemu-system-i386:** 用于模拟 32 位 x86 架构的虚拟机监控程序，可直接启动 32 位的操作系统镜像。

**qemu-system-amd64** 和 **qemu-system-x86\_64:** 均为模拟 64 位 x86 架构（AMD64/x86\_64）的虚拟机监控程序，用于运行 64 位操作系统，两者功能基本一致（不同发行版可能命名不同）。

**qemu-system-x86\_64-microvm:** 针对 x86\_64 架构的轻量级虚拟机模式（microvm），启动速度更快，资源占用更低，适用于边缘计算或快速部署场景。

**virt-clone:** 用于克隆现有虚拟机

**virt-xml:** 用于编辑虚拟机的 XML 配置文件

**virt-host-validate:** 检查主机系统是否满足虚拟化要求

**virt-host-validate:** 检查主机系统是否满足虚拟化要求

## 7.网络资源池管理

| 命令         | 用途                               |
|------------|----------------------------------|
| event      | 监控 libvirt 事件（如虚拟机状态变化）          |
| nwfilter-* | 网络过滤器管理（如 nwfilter-list 列出过滤器）   |
| secret-*   | 加密密钥管理（如 secret-set-value 设置密钥值） |
| perf <vm>  | 查看虚拟机的性能计数器                      |

| 命令                 | 用途                |
|--------------------|-------------------|
| net-list [--all]   | 列出虚拟网络（默认显示运行中）   |
| net-start <net>    | 启动虚拟网络            |
| net-destroy <net>  | 停止虚拟网络            |
| net-create <xml>   | 从 XML 临时创建虚拟网络    |
| net-define <xml>   | 从 XML 定义虚拟网络（持久化） |
| net-undefine <net> | 删除虚拟网络的定义         |
| net-dumpxml <net>  | 导出虚拟网络的 XML 配置    |
| net-edit <net>     | 编辑虚拟网络的 XML 配置    |
| net-info <net>     | 查看虚拟网络的基本信息       |
| net-name <uuid>    | 通过 UUID 查询虚拟网络名称  |

| 命令                                     | 用途                |
|----------------------------------------|-------------------|
| net-uuid <name>                        | 通过名称查询虚拟网络 UUID   |
| net-autostart <net>                    | 设置虚拟网络开机自启        |
| net-dhcp-leases <net>                  | 查看虚拟网络的 DHCP 租约信息 |
| net-update <net> <action> <section>... | 动态更新虚拟网络配置        |
| net-port-create <net> <xml>            | 为虚拟网络创建端口         |
| net-port-delete <net> <port>           | 删除虚拟网络的端口         |
| net-port-list <net>                    | 列出虚拟网络的端口         |

**7.1.1.net-list [--all]**

**7.1.2.net-start <net>**

**7.1.3.net-destroy <net>**

**7.1.4.net-create <xml>**

**7.1.5.net-define <xml>**

**7.1.6.net-undefine <net>**

**7.1.7.net-dumpxml <net>**

**7.1.8.net-edit <net>**

**7.1.9.net-info <net>**

**7.1.10.net-name <uuid>**

**7.1.11.net-uuid <name>**

**7.1.12.net-autostart <net>**

**7.1.13.net-dhcp-leases <net>**

**7.1.14.net-update <net> <action> <section>...**

**7.1.15.net-port-create <net> <xml>**

**7.1.16.net-port-delete <net> <port>**

### 7.1.17.net-port-list <net>

## 8.存储资源池管理

| 命令                                 | 用途                 |
|------------------------------------|--------------------|
| pool-list [--all]                  | 列出存储池（默认显示运行中）     |
| pool-start <pool>                  | 启动存储池              |
| pool-destroy <pool>                | 停止存储池              |
| pool-create <xml>                  | 从 XML 临时创建存储池      |
| pool-define <xml>                  | 从 XML 定义存储池（持久化）   |
| pool-undefine <pool>               | 删除存储池的定义           |
| pool-dumpxml <pool>                | 导出存储池的 XML 配置      |
| pool-edit <pool>                   | 编辑存储池的 XML 配置      |
| pool-info <pool>                   | 查看存储池的基本信息         |
| pool-name <uuid>                   | 通过 UUID 查询存储池名称    |
| pool-uuid <name>                   | 通过名称查询存储池 UUID     |
| pool-autostart <pool>              | 设置存储池开机自启          |
| pool-build <pool>                  | 初始化存储池（如格式化分区）     |
| pool-delete <pool>                 | 删除存储池中的数据（谨慎使用）    |
| pool-refresh <pool>                | 刷新存储池（检测新增 / 删除的卷） |
| vol-list <pool>                    | 列出存储池中的卷（磁盘文件）     |
| vol-create <pool> <xml>            | 从 XML 在存储池中创建卷     |
| vol-create-as <pool> <name> <size> | 快速创建卷（指定名称和大小）     |

| 命令                                  | 用途                 |
|-------------------------------------|--------------------|
| vol-create-from <pool> <name> <src> | 从现有卷复制创建新卷         |
| vol-clone <src> <dst>               | 克隆卷                |
| vol-delete <vol>                    | 删除卷                |
| vol-dumpxml <vol>                   | 导出卷的 XML 配置        |
| vol-info <vol>                      | 查看卷的基本信息（大小 / 路径等） |
| vol-resize <vol> <size>             | 调整卷的大小             |
| vol-upload <vol> <file>             | 将本地文件上传到卷          |
| vol-download <vol> <file>           | 将卷内容下载到本地文件        |
| vol-wipe <vol>                      | 擦除卷数据（防止数据泄露）      |
| find-storage-pool-sources           | 自动检测可用的存储池源（如磁盘分区） |

## 9.虚拟机迁移

|                        |                     |
|------------------------|---------------------|
| Migrate-setmaxdowntime | 设定最大可耐受故障时间         |
| Migrate-getmaxdowntime | 获得最大可容忍停机时间         |
| Migrate-compcache      | 获取/设定压缩缓存大小         |
| Migrate-setspeed       | 设定迁移带宽的最大值          |
| Migrate-getspeed       | 获取最长迁移带宽            |
| Migrate-postcopy       | 将运行迁移从 Pre-copy 切换到 |
| Post-copy              |                     |
| Migrate                | 将域迁移到另一个主机中         |
| Domjobabort            | 忽略活跃域任务             |
| Domjobinfo             | 域任务信息               |



## 9.1.虚拟机迁移准备

### 9.1.1.物理机 cpu 基线管理 (cpu 兼容性排查)

### 9.1.2.虚拟机 cpu 基线管理 (cpu 兼容性排查)

### 9.1.3.virsh hypervisor-cpu-baseline

用于生成一个 CPU 特性基线，该基线是多个 CPU 配置的交集，可作为虚拟机在不同宿主机间迁移时的兼容 CPU 配置，确保迁移兼容性。其主要参数含义如下：

--arch <架构>: 指定 CPU 架构（如 x86\_64、aarch64），用于限定基线的架构范围

--features: 显式指定需要包含的 CPU 特性（可覆盖自动计算的基线）

--machine <机器类型>: 指定虚拟机的机器类型（如 pc-i440fx-4.2），影响兼容特性集

--virttype <虚拟化类型>: 指定虚拟化技术（如 kvm、qemu），不同类型支持的特性可能不同

--emulator <路径>: 指定模拟器二进制文件路径（如 /usr/bin/qemu-system-x86\_64），基于该模拟器能力生成基线

--file <文件>: 从指定文件读取多个 CPU 配置（每行一个 CPU 描述），用于计算基线

--migratable: 生成适合迁移的 CPU 基线（优先保留迁移兼容的特性）

示例用法：基于当前宿主机 CPU 和文件中定义的其他 CPU 配置，生成 x86\_64 架构下可迁移的 KVM 虚拟机 CPU 基线：

```
bash
```

```
virsh hypervisor-cpu-baseline --arch x86_64 --virttype kvm --migratable --file  
/path/to/cpu-configs.txt
```

输出为 XML 格式的 CPU 配置，包含所有输入 CPU 共有的特性，可直接用于虚拟机的 CPU 配置，确保虚拟机在这些宿主机间迁移时的兼容性。

#### 9.1.4.virsh cpu-baseline

`virsh cpu-baseline` 命令用于计算多个虚拟机（或 CPU 配置）的“基准 CPU 配置”，即所有指定 CPU 配置的交集功能集，常用于确保虚拟机在不同主机间迁移时的 CPU 兼容性。你列出的参数说明如下：

**--features:** 在输出的基准 CPU 配置中，显式包含所有 CPU 特性（默认可能省略部分默认特性）

**--file <文件路径>:** 从指定文件读取多个 CPU 配置（XML 格式），用于计算基准配置（文件中需包含多个<cpu>定义）

**--migratable:** 生成适合迁移的基准配置（会排除主机特定的 CPU 特性，确保兼容性）

常用示例

从文件读取多个 CPU 配置并生成基准配置：

```
bash
```

```
virsh cpu-baseline --file cpu_configs.xml
```

（cpu\_configs.xml 中需包含多个虚拟机的 CPU 配置，如不同主机的 CPU 信息）

生成可迁移的基准配置：

```
bash
```

```
virsh cpu-baseline --migratable --file cpu_configs.xml
```

输出结果为 XML 格式的 CPU 配置，代表所有输入 CPU 配置的共同功能集，可用于配置虚拟机的 CPU，确保其能在不同主机间迁移时正常运行。

#### 9.1.5.virsh hypervisor-cpu-compare

`virsh hypervisor-cpu-compare` 命令用于比较两个或多个宿主机（hypervisor）的 CPU 特性，判断它们之间的兼容性，尤其用于验证虚拟机在不同宿主机间迁移的可行性。其主要参数含义如下：

**--arch <架构>:** 指定 CPU 架构（如 x86\_64），限定比较的架构范围

**--emulator <路径>:** 指定模拟器二进制文件路径（如 /usr/bin/qemu-system-x86\_64），基于该模拟器能力进行比较

**--error:** 若存在不兼容特性，以错误码而非警告形式输出

**--file <文件>:** 从指定文件读取待比较的 CPU 配置（通常是其他宿主机的 CPU 信息）

**--machine <机器类型>:** 指定虚拟机的机器类型（如 `pc-i440fx-4.2`），影响特性兼容性判断

**--virttype <虚拟化类型>:** 指定虚拟化技术（如 `kvm`），不同类型的兼容性规则不同

示例用法：比较当前宿主机与文件中定义的另一宿主机的 CPU 兼容性（针对 KVM 虚拟化的 `x86_64` 架构）：

```
bash
```

```
virsh hypervisor-cpu-compare --arch x86_64 --virttype kvm --file
```

```
/path/to/other-hypervisor-cpu.xml
```

命令输出会提示两者的 CPU 特性差异：

若完全兼容，会显示 “CPU is compatible”

若存在不兼容特性（如目标宿主机缺少源宿主机的关键特性），会列出具体差异，帮助判断是否需要调整虚拟机 CPU 配置以支持迁移。

该命令是虚拟机跨宿主机迁移前的重要检查工具，可提前发现潜在的 CPU 兼容性问题。

编辑分享

除了 `x86_64` 架构，`virsh hypervisor-cpu-compare` 还支持哪些架构？

如何使用 `virsh hypervisor-cpu-compare` 命令？

除了 `virsh hypervisor-cpu-compare`，还有哪些类似的命令？

### 9.1.6.virsh cpu-compare

`virsh cpu-compare` 是用于对比两个 CPU 配置兼容性的命令，核心作用是判断 “目标 CPU” 是否支持 “参考 CPU” 的所有功能（常用于验证虚拟机迁移到目标主机的 CPU 兼容性）。你列出的参数具体说明如下：

一、核心参数解析

| 参数 | 作用与说明 |
|----|-------|
|----|-------|

|                                |                                                                                                 |
|--------------------------------|-------------------------------------------------------------------------------------------------|
| <b>--file &lt;XML 文件路径&gt;</b> | 指定一个包含 CPU 配置的 XML 文件（作为 “参考 CPU”），与当前主机的 CPU 进行对比。XML 文件需符合 <code>libvirt</code> 的 CPU 配置格式（可通 |
|--------------------------------|-------------------------------------------------------------------------------------------------|

参数                      作用与说明

过 `virsh cpu-baseline` 或 `virsh dumpxml` 提取）。

当“目标 CPU”不支持“参考 CPU”的功能时，直接以错误码（非 0）

`--error`              退出，而非仅在终端打印警告信息。常用于脚本自动化场景（例如迁移前检查，不兼容则终止后续流程）。

## 二、命令本质与对比逻辑

`virsh cpu-compare` 的对比逻辑是“参考 CPU → 目标 CPU”：

参考 CPU：通常是虚拟机当前运行主机的 CPU 配置（或通过 `--file` 指定的 CPU XML），代表虚拟机依赖的 CPU 功能集。

目标 CPU：默认是执行命令所在主机的 CPU（即计划迁移到的主机）。

命令执行后会返回 3 种核心结果：

兼容：目标 CPU 支持参考 CPU 的所有功能，虚拟机可正常迁移 / 运行。示例输出：  
CPU model 'Intel(R) Xeon(R) CPU E5-2680 v3' is compatible with CPU model 'Intel(R) Xeon(R) CPU E5-2670 v3'

部分兼容（缺功能）：目标 CPU 缺少参考 CPU 的部分非关键功能，虚拟机可能运行但存在风险（会列出缺失的特性）。示例输出：CPU model 'Intel(R) Xeon(R) CPU E5-2680 v3' is incompatible with CPU model 'Intel(R) Xeon(R) CPU E5-2670 v3': missing features: avx2, bmi1, bmi2

不兼容：目标 CPU 缺少参考 CPU 的关键功能，虚拟机无法正常运行。

## 三、常用示例

### 1. 基础对比：文件中的参考 CPU vs 当前主机 CPU

将“源主机 CPU 配置”保存为 `source_cpu.xml`，在“目标主机”上执行对比，检查迁移兼容性：

```
bash
```

```
# 在目标主机执行：对比 source_cpu.xml（参考 CPU）与当前主机 CPU
virsh cpu-compare --file source_cpu.xml
```

### 2. 自动化对比：不兼容则报错退出

在迁移脚本中使用 `--error`，若 CPU 不兼容则直接终止脚本（避免后续无效操作）：

```
bash
```

```
# 若对比不兼容，脚本立即退出（错误码非 0）；兼容则继续执行后续命令 if ! virsh  
cpu-compare --error --file source_cpu.xml; then
```

```
    echo "CPU 不兼容，无法迁移虚拟机！"
```

```
    exit 1fi echo "CPU 兼容，开始迁移..."
```

#### 四、关键用途

虚拟机迁移前检查：确保目标主机 CPU 支持虚拟机当前依赖的所有 CPU 特性（避免迁移后虚拟机启动失败或运行异常）。

集群 CPU 一致性验证：检查集群内所有主机的 CPU 是否兼容，确保虚拟机可在集群内任意主机间迁移。

脚本自动化：配合--error 参数，将 CPU 兼容性检查集成到自动化运维脚本中，提升可靠性。

### 9.1.7.

#### 9.1.8.virsh cpu-models

```
virsh cpu-models --arch
```

virsh cpu-models --arch <架构> 命令用于查询指定 CPU 架构下，当前系统支持的预定义 CPU 模型列表（这些模型是 libvirt 预定义的标准 CPU 配置，包含特定的指令集和特性组合）。

基本用法

```
bash
```

```
virsh cpu-models --arch <架构名称>
```

常用示例

查询 x86\_64 架构支持的 CPU 模型：

```
bash
```

```
virsh cpu-models --arch x86_64
```

输出示例（部分）：

```
plaintext
```

```
486
```

pentium  
pentium2  
pentium3  
pentiumpro  
coreduo  
core2duo  
nehalem  
westmere  
sandybridge  
ivybridge  
haswell  
broadwell  
skylake  
# ... 更多模型

查询 ARM 架构（如 aarch64）支持的 CPU 模型：

```
bash  
virsh cpu-models --arch aarch64
```

用途说明

这些预定义 CPU 模型是 libvirt 为不同架构和代际的 CPU 预设的标准化配置，包含特定的指令集和特性（如 haswell 模型对应 Intel Haswell 架构的典型特性）。

在创建或配置虚拟机时，可以指定这些模型（如<cpu mode='custom' model='haswell'/>），确保虚拟机在不同主机间的兼容性。

结合 cpu-baseline 和 cpu-compare 命令，可以更精准地管理跨主机的 CPU 兼容性。

## 10.快照与备份

| 命令                   | 用途       |
|----------------------|----------|
| snapshot-create <vm> | 为虚拟机创建快照 |

| 命令                                    | 用途             |
|---------------------------------------|----------------|
| snapshot-create-as <vm> --name <name> | 自定义名称创建快照      |
| snapshot-current <vm>                 | 查看当前激活的快照      |
| snapshot-delete <vm> <snap>           | 删除虚拟机的快照       |
| snapshot-dumpxml <vm> <snap>          | 导出快照的 XML 配置   |
| snapshot-edit <vm> <snap>             | 编辑快照的 XML 配置   |
| snapshot-info <vm> <snap>             | 查看快照的详细信息      |
| snapshot-list <vm>                    | 列出虚拟机的所有快照     |
| snapshot-parent <vm> <snap>           | 查看快照的父快照       |
| snapshot-revert <vm> <snap>           | 将虚拟机恢复到指定快照状态  |
| backup-begin <vm> <xml>               | 开始虚拟机的备份任务     |
| backup-dumpxml <vm>                   | 查看备份任务的 XML 配置 |

10.1.1.virsh snapshot-create <vm>

10.1.2.virsh snapshot-create-as <vm> --name <name>

10.1.3.virsh snapshot-current <vm>

10.1.4.virsh snapshot-delete <vm> <snap>

10.1.5.virsh snapshot-dumpxml <vm> <snap>

10.1.6.virsh snapshot-edit <vm> <snap>

10.1.7.virsh snapshot-info <vm> <snap>

10.1.8.virsh snapshot-list <vm>

10.1.9.virsh snapshot-parent <vm> <snap>

10.1.10.virsh snapshot-revert <vm> <snap>

10.1.11.virsh backup-begin <vm> <xml>

10.1.12.virsh backup-dumpxml <vm>

## 11.虚拟机 xml 操作

| 内容           | 命令            | 说明                                          |
|--------------|---------------|---------------------------------------------|
| 虚拟机 XML 文件导出 | virsh dumpxml | 定义持久化虚拟机，定义完成后虚拟机处于关闭状态，虚拟机被看作为一个 domian 实例 |



| 内容                  | 命令                           | 说明                               |
|---------------------|------------------------------|----------------------------------|
| 虚拟机 XML 文件编辑        | virsh edit                   | 销毁持久性虚拟机，虚拟机生命周期完结，不能继续对该虚拟机继续操作 |
| 原始配置转 XML 文件        | virsh<br>domxml-from-native  | 将原始配置转换为域 Xml                    |
| XML 文件转原始配置         | virsh<br>domxml-to-native    | 将域 Xml 转换为原始配置                   |
| 虚拟机休眠保存文件<br>XML 导出 | virsh<br>Managedsave-dumpxml | Xml 中受管休眠状态文件的域信息                |
| 虚拟机休眠文件<br>XML 编辑   | virsh<br>Managedsave-edit    | 虚拟机休眠文件<br>XML 编辑                |
| 虚拟机休眠文件<br>注册       | virsh<br>Managedsave-define  | 虚拟机休眠文件<br>注册                    |
| 虚拟机暂停保存文件<br>XML 导出 | virsh<br>save-image-dumpxml  | 虚拟机暂停保存文件<br>XML 导出              |
| 虚拟机暂停文件<br>XML 编辑   | virsh<br>save-image-edit     | 虚拟机暂停文件<br>XML 编辑                |
| 虚拟机暂停文件<br>注册       | virsh<br>save-image-define   | 虚拟机暂停文件<br>注册                    |

## 11.1.虚拟机 XML 文件导出

virsh dumpxml 命令的这些选项用于控制 XML 配置的导出方式和内容，以下是各参数的详细说明及用法：

核心参数说明

1. --domain <虚拟机名/ID/UUID>

指定要导出配置的虚拟机（必填参数）。示例：

bash

virsh dumpxml --domain ubuntuvm # 导出 ubuntuvm 的配置

## 2. --inactive

导出非活跃状态的配置（即虚拟机配置文件中保存的静态配置），忽略当前运行时的动态修改（如临时调整的内存、CPU 等）。适用场景：查看或备份虚拟机的「原始配置」，而非运行中的临时状态。示例：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --inactive # 导出静态配置
```

## 3. --migratable

导出「可迁移」的配置格式，移除与特定宿主机相关的信息（如本地文件路径、宿主机硬件依赖等），使配置可用于跨主机迁移。适用场景：虚拟机迁移前，生成通用配置文件。示例：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --migratable # 生成可迁移的配置
```

## 4. --security-info

在导出的 XML 中包含安全相关信息，如 SELinux 上下文、安全标签、访问控制列表(ACL) 等。默认情况下，这些信息会被隐藏，仅当需要调试安全相关问题时使用。示例：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --security-info # 包含安全信息
```

## 5. --update-cpu

更新 XML 中的 CPU 配置，使其与当前宿主机的 CPU 特性兼容（自动调整 CPU 型号或特性列表）。适用场景：当虚拟机在不同 CPU 型号的宿主机间迁移时，确保 CPU 配置兼容。示例：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --update-cpu # 自动更新 CPU 配置
```

## 6. --xpath <XPath 表达式>

仅导出 XML 中与指定 XPath 表达式匹配的部分（而非完整配置），用于快速提取特定配置项。适用场景：精准查询某类配置（如磁盘路径、网络接口 MAC 等）。示例：

```
bash
```

```
# 提取所有磁盘设备的源文件路径 virsh dumpxml --domain ubuntuvm --xpath //disk/source  
  
# 提取网络接口的 MAC 地址 virsh dumpxml --domain ubuntuvm --xpath //interface/mac
```

## 7. --wrap

在导出的 XML 中，对长文本内容（如证书、复杂配置）进行换行处理，使输出更易读。

默认情况下，长文本可能显示为一行，--wrap 可优化格式。示例：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --wrap # 格式化长文本内容
```

常用组合示例

备份静态配置并保存到文件：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --inactive > ubuntuvm_static_backup.xml
```

提取特定配置（如 VNC 端口）：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --xpath //graphics/@port
```

生成可迁移的配置文件：

```
bash
```

```
virsh dumpxml --domain ubuntuvm --migratable --update-cpu > migrate_config.xml
```

总结

这些参数扩展了 dumpxml 的灵活性，可根据需求选择：

备份配置用 --inactive；

迁移虚拟机用 --migratable + --update-cpu；

提取特定配置用 --xpath；

调试安全问题用 --security-info。

通过组合参数，能更高效地管理和分析虚拟机配置。

## 11.2.虚拟机 XML 文件编辑

```
efenye@efenye-deepin:~$ virsh edit --
```

```
--domain
```

```
--skip-validate
```

### 11.3.原始配置转 XML 文件

`virsh domxml-from-native` 是 `domxml-to-native` 的反向命令，用于将其他格式的配置转换为 `libvirt` 兼容的 XML 配置文件。你列出的参数说明如下：

`--config`: 将转换后的 XML 直接用于定义虚拟机的配置（配合输出重定向可保存为文件，或直接用于 `virsh define`）

`--format <格式>`: 指定源配置的格式（与 `domxml-to-native` 支持的格式对应，如 `qemu-argv`、`lxc` 等）

### 11.4.XML 文件转原始配置

`virsh domxml-to-native` 命令用于将虚拟机的 XML 配置文件转换为其他格式（如 `QEMU` 命令行参数格式），方便查看或直接用于启动虚拟机。你列出的是该命令的核心参数，具体说明如下：

`--domain <名称/UUID>`: 指定要转换配置的虚拟机（通过名称或 `UUID` 标识）

`--format <格式>`: 指定目标转换格式，最常用的是 `qemu-argv`（转换为 `QEMU` 命令行参数格式）

`--xml <XML 文件路径>`: 直接指定 XML 配置文件的路径（替代 `--domain` 参数，适用于未定义的虚拟机配置）

常用示例

将名为 `vm1` 的虚拟机配置转换为 `QEMU` 命令行格式：

```
bash
```

```
virsh domxml-to-native --domain vm1 --format qemu-argv
```

将某个 XML 文件（如 `vm2.xml`）转换为 `QEMU` 命令行：

```
bash
```

```
virsh domxml-to-native --xml /path/to/vm2.xml --format qemu-argv
```

转换后的输出会展示该虚拟机对应的 `QEMU` 启动命令（包含硬件参数、磁盘路径、网络配置等），主要用于调试虚拟化配置或手动启动虚拟机时参考。

## 11.5.虚拟机保存文件 XML 导出

`virsh managedsave-dumpxml` 命令，这是用于导出虚拟机的托管保存（managed save）状态的 XML 配置的命令。托管保存状态是指虚拟机被暂停并将其状态保存到磁盘后，用于描述该保存状态的配置信息。

基本语法

bash

`virsh managedsave-dumpxml <虚拟机名/ID/UUID>`

参数：指定已进行托管保存（`virsh managedsave`）的虚拟机（名称、ID 或 UUID）。

功能：导出该虚拟机保存状态的 XML 配置，包含保存时的设备状态、内存信息等元数据（非完整内存数据，仅状态描述）。

使用场景

查看托管保存状态的配置细节：了解虚拟机在被保存时的设备状态、配置参数等。

备份或迁移保存状态：结合 `managedsave-define` 命令，可在另一台宿主机上恢复保存状态（需配合磁盘文件）。

示例

假设已对 `ubuntuvvm` 执行托管保存：

bash

# 先执行托管保存（将虚拟机状态保存到磁盘并暂停）`virsh managedsave ubuntuvvm`

# 导出该保存状态的 XML 配置 `virsh managedsave-dumpxml ubuntuvvm`

输出的 XML 片段示例（包含保存时间、设备状态等）：

xml

```
<domainsnapshot>
```

```
  <name>managedsave</name>
```

```
  <state>paused</state>
```

```
  <creationTime>1697000000</creationTime>
```

```
  <memory snapshot='internal'/'>
```

```
  <disks>
```

```
    <disk name='vda' snapshot='external'/'>
```

```
  </disks>
```

```
<!-- 其他设备状态信息 --></domainsnapshot>
```

#### 相关命令

**virsh managedsave:** 将虚拟机状态保存到磁盘并暂停（创建托管保存状态）。

**virsh managedsave-define:** 通过 XML 配置恢复托管保存状态。

**virsh managedsave-remove:** 删除虚拟机的托管保存状态。

#### 注意事项

仅当虚拟机存在托管保存状态时（即已执行 **managedsave**），该命令才有输出，否则会报错。

导出的 XML 是状态描述信息，不包含完整内存数据，完整状态恢复还需依赖原始磁盘文件。

通过 **managedsave-dumpxml** 可查看和管理虚拟机的托管保存状态配置，是虚拟化环境中状态备份与恢复的辅助工具。

在 **virsh** 命令中，**--security-info** 是一个常用选项，用于在输出中包含与安全相关的详细信息。它通常与查询或导出虚拟机配置的命令配合使用（如 **dumpxml**、**managedsave-dumpxml** 等），默认情况下这些安全信息会被隐藏。

#### 作用与包含的信息

当使用 **--security-info** 时，命令输出会额外包含以下安全相关内容：

**SELinux 上下文：**虚拟机进程和文件的 SELinux 安全标签

（如 **system\_u:system\_r:svirt\_t:s0:c123,c456**）。

**安全标签：**与虚拟机相关的访问控制标签（如 **svirt** 标签）。

**安全驱动配置：**虚拟机使用的安全驱动类型及参数。

**敏感设备权限：**如 **USB** 设备、**PCI** 设备的访问控制列表（**ACL**）。

#### 典型使用场景

最常用于 **virsh dumpxml** 命令，查看虚拟机的完整安全配置：

```
bash
```

```
# 导出虚拟机 ubuntuvm 的配置并包含安全信息 virsh dumpxml ubuntuvm
```

```
--security-info
```

输出中会新增类似以下的安全相关片段：

```
xml
```

```
<seclabel type='dynamic' model='selinux' relabel='yes'>
```

```
<label>system_u:system_r:svirt_t:s0:c123,c456</label>
```

```
<imagelabel>system_u:object_r:svirt_image_t:s0:c123,c456</imagelabel></seclabel><seclabel
```

```
type='dynamic' model='dac' relabel='yes'>
```

```
<label>107:107</label>
```

```
<imagelabel>107:107</imagelabel></seclabel>
```

适用命令

**virsh dumpxml:** 导出虚拟机配置时包含安全信息。

**virsh managedsave-dumpxml:** 导出托管保存状态的配置时包含安全信息。

**virsh snapshot-dumpxml:** 导出快照配置时包含安全信息。

使用价值

**安全审计:** 检查虚拟机的 SELinux 上下文、权限配置是否符合安全策略。

**问题排查:** 当虚拟机因权限问题无法访问设备或文件时，通过安全信息定位原因（如标签不匹配）。

**合规性验证:** 确保虚拟机的安全配置符合企业或行业的安全规范。

注意事项

仅管理员用户（如 **root**）能查看完整的安全信息，普通用户可能被限制。

输出内容会增加一定复杂度，非安全相关场景通常无需使用该选项。

总之，**--security-info** 是深入了解虚拟机安全配置的关键选项，主要用于安全审计和权限问题排查。

## 11.6.虚拟机保存文件 XML 编辑

## 11.7.虚拟机保存文件注册

## 11.8.虚拟机暂停保存文件 XML 导出

## 11.9.虚拟机暂停文件 XML 编辑

## 11.10.虚拟机暂停文件注册

# 12.物理机管理

## 12.1.物理机网络设置

|                                        |                                                          |
|----------------------------------------|----------------------------------------------------------|
| Interface (使用关键字 'interface' 来显示帮助信息): |                                                          |
| iface-begin                            | 生成当前接口设置快照，可在今后用于提交 (iface-commit) 或者恢复 (iface-rollback) |
| iface-bridge                           | 生成桥接设备并为其附加一个现有网络设备                                      |
| iface-commit                           | 提交 iface-begin 后的更改并释放恢复点                                |
| iface-define                           | 定义不活跃的持久性物理主机接口，或者从 XML 文件中修改现有的持久性物理主机接口                |
| iface-destroy                          | 删除物理主机接口（启用它请执行 "if-down"）                               |
| iface-dumpxml                          | XML 中的接口信息                                               |
| iface-edit                             | 为物理主机界面编辑 XML 配置                                         |
| iface-list                             | 物理主机接口列表                                                 |
| iface-mac                              | 将接口名称转换为接口 MAC 地址                                        |
| iface-name                             | 将接口 MAC 地址转换为接口名称                                        |
| iface-rollback                         | 恢复到之前保存的使用 iface-begin 生成的更改                             |
| iface-start                            | 启动物理主机接口（启用它请执行 "if-up"）                                 |
| iface-unbridge                         | 在分离其设备后取消网桥设备的定义                                         |
| iface-undefine                         | 取消定义物理主机接口（从配置中删除）                                       |

## 12.2.物理机信息查询与监控

### 12.2.1.virsh hostname

显示 KVM 主机的名字



### 12.2.2.virsh uri

显示登录的虚拟机是本地的，还是远程的：

本地：qemu:///session

远程：<qemu:///root@192.168.3.60/system>

### 12.2.3.virsh sysinfo

virsh sysinfo 是 libvirt 工具集中的一个命令，用于获取宿主机的系统信息。它会返回关于宿主机硬件和软件的详细信息，包括：

处理器信息（型号、核心数、线程数等）

内存信息（总内存、可用内存等）

主板信息（制造商、型号等）

BIOS 信息

操作系统信息

其他硬件相关信息（如存储控制器、网络控制器等）

### 12.2.4.virsh nodeinfo

virsh nodeinfo 是 libvirt 工具集中用于查询宿主机（节点）基本信息的命令，它提供了宿主机的关键硬件和虚拟化能力摘要，输出格式比 virsh sysinfo 更简洁易读。

执行该命令后，通常会返回以下信息：

宿主机的 CPU 架构（如 x86\_64）

物理 CPU 总数（sockets）

每个物理 CPU 的核心数（cores per socket）

每个核心的线程数（threads per core）

总逻辑 CPU 数量（= sockets × cores × threads）

宿主机的内存总量（以 KiB 为单位）

### 12.2.5.virsh nodecpustats

virsh nodecpustats 命令用于查询宿主机（节点）的 CPU 统计信息，提供关于物理 CPU 使用率、负载等详细指标，帮助管理员监控宿主机的 CPU 资源状态。

主要输出信息及含义

|             |       |                            |
|-------------|-------|----------------------------|
| user:       | 15.2% | # 用户态进程占用 CPU 的百分比         |
| system:     | 3.8%  | # 内核态占用 CPU 的百分比           |
| idle:       | 79.5% | # 空闲 CPU 的百分比              |
| iowait:     | 1.2%  | # 等待 I/O 的 CPU 百分比         |
| irq:        | 0.1%  | # 硬中断占用 CPU 的百分比           |
| softirq:    | 0.2%  | # 软中断占用 CPU 的百分比           |
| steal:      | 0.0%  | # 被其他虚拟机占用的 CPU 百分比（虚拟化环境） |
| guest:      | 5.0%  | # 所有虚拟机占用的 CPU 百分比         |
| guest_nice: | 0.0%  | # 虚拟机中低优先级进程的 CPU 百分比      |

主要输出信息及含义

执行命令后，通常会返回类似以下的统计数据（不同系统可能略有差异）：

|             |            |                               |
|-------------|------------|-------------------------------|
| plaintext   |            |                               |
| user:       | 123456789  | # 用户态进程使用的 CPU 时间（纳秒）         |
| system:     | 987654321  | # 内核态使用的 CPU 时间（纳秒）           |
| idle:       | 1234567890 | # 空闲 CPU 时间（纳秒）               |
| iowait:     | 123456     | # 等待 I/O 的 CPU 时间（纳秒）         |
| irq:        | 1234       | # 处理硬中断的 CPU 时间（纳秒）           |
| softirq:    | 5678       | # 处理软中断的 CPU 时间（纳秒）           |
| steal:      | 0          | # 被虚拟化层偷走的 CPU 时间（纳秒，通常用于虚拟机） |
| guest:      | 0          | # 虚拟机使用的 CPU 时间（纳秒）           |
| guest_nice: | 0          | # 虚拟机中低优先级进程使用的 CPU 时间（纳秒）    |

关键指标说明

user: 用户空间程序消耗的 CPU 时间，反映应用程序的 CPU 负载。

system: 内核处理系统调用等消耗的 CPU 时间，过高可能表示系统内核有较多操作（如频繁的进程调度、I/O 处理）。

**idle:** CPU 空闲时间，占比越高说明 CPU 资源越充足。

**iowait:** CPU 等待磁盘或网络 I/O 完成的时间，过高可能表示 I/O 成为性能瓶颈。

**steal:** 在虚拟化环境中，物理 CPU 被其他虚拟机占用的时间，过高可能表示宿主机 CPU 资源紧张。

**guest:** 当前宿主机上所有虚拟机消耗的总 CPU 时间，可用于评估虚拟机对物理 CPU 的占用情况。

实际用途

实时监控宿主机 CPU 负载，判断是否存在资源紧张。

分析 CPU 时间的分布（用户态 / 内核态 / 空闲等），定位性能瓶颈（如 I/O 等待过高、内核态消耗过大等）。

结合虚拟机的 CPU 使用率，评估宿主机 CPU 资源的分配合理性。

示例

直接执行命令查看当前宿主机的 CPU 统计信息：

```
bash
```

```
virsh nodecpustats
```

该命令输出的是累计时间值，若需观察动态变化，可结合 **watch** 命令定期执行：

```
bash
```

```
watch -n 1 virsh nodecpustats # 每 1 秒刷新一次统计信息
```

```
virsh nodememstats
```

**virsh nodememstats** 是用于查询宿主机（节点）内存使用情况的命令，输出系统物理内存的总量、空闲量及各类缓存等详细统计信息，帮助管理员掌握宿主机内存资源状态。

典型输出及含义

执行命令后，通常会显示以下内存指标（单位通常为 KiB）：

plaintext

**total** :<数值> # 宿主机总物理内存容量

**free** :<数值> # 完全空闲的内存（未被任何进程使用）

**used** :<数值> # 已使用的内存（total - free - buffers - cached）

**buffers**:<数值> # 内核用于磁盘 I/O 缓冲区的内存（可回收）

**cached**:<数值> # 用于缓存文件内容的内存（可回收，提升文件访问速度）

**swap\_total**:<数值> # 交换分区总容量（若配置）

**swap\_free:** <数值> # 空闲交换分区容量（若配置）

关键指标说明

**total:** 物理内存总大小，反映宿主机的内存硬件配置。

**free:** 真正未被使用的内存，可直接分配给新进程或虚拟机。

**buffers:** 缓存磁盘元数据（如目录结构、文件属性）的内存，系统需要时可释放。

**cached:** 缓存文件内容的内存，进程再次访问时可直接从内存读取，减少磁盘 I/O，同样可被系统回收重用。

**swap\_\*:** 交换分区相关信息，当物理内存不足时，系统会将部分内存数据写入交换分区（性能较低）。

实际用途

快速评估宿主机内存整体使用情况，判断是否有足够内存新建或扩容虚拟机。

通过 **free + buffers + cached** 计算“实际可灵活分配的内存”（Linux 中这部分内存可被重新利用）。

监控缓存内存占比，若 **cached** 过大且系统内存紧张，可能需要清理缓存（如 **sync; echo 3 > /proc/sys/vm/drop\_caches**）。

示例用法

查看整体内存状态：

**bash**

**virsh nodememstats**

查看指定 NUMA 节点的内存状态（如第 0 个节点）：

**bash**

**virsh nodememstats --cell 0**

该命令是虚拟化环境中内存资源监控的基础工具，常用于日常运维和性能调优。

分享

哪些操作会导致 **buffers** 和 **cached** 的值发生变化？

如何根据这些指标优化宿主机内存使用？

如何查看特定进程或虚拟机的内存使用情况？

## 12.2.6.virsh nodecpumap

`virsh nodecpumap --pretty` 是 `libvirt` 工具中用于查看宿主机 CPU 映射关系的命令，它以易读的格式展示物理 CPU 与逻辑 CPU 之间的对应关系，帮助理解宿主机的 CPU 拓扑结构。

执行该命令后，通常会输出类似以下的信息：

plaintext

Sockets: 2

Cores per socket: 4

Threads per core: 2

Total CPUs: 16

Socket 0:

Core 0: 0,8

Core 1: 1,9

Core 2: 2,10

Core 3: 3,11

Socket 1:

Core 0: 4,12

Core 1: 5,13

Core 2: 6,14

Core 3: 7,15

上述输出的含义：

宿主机有 2 个物理 CPU（Sockets）

每个物理 CPU 有 4 个核心（Cores per socket）

每个核心有 2 个线程（Threads per core）

总逻辑 CPU 数量为 16（ $2 \times 4 \times 2$ ）

每个数字代表一个逻辑 CPU 的 ID，同一行的 ID 表示它们属于同一个物理核心（超线程技术）

这个命令对于虚拟机的 CPU 亲和性配置、性能优化非常有用，能帮助管理员了解如何合理分配 CPU 资源给虚拟机，避免资源冲突或性能瓶颈

### 12.2.7.virsh freecell

`virsh freecell` 命令用于查询宿主机上 NUMA 节点的空闲内存信息，帮助管理员了解各 NUMA 节点的内存使用情况，以便进行内存资源分配规划。

其参数含义如下：

`--all`：显示所有 NUMA 节点的空闲内存信息

`--cellno <num>`：指定要查询的 NUMA 节点编号（如 `--cellno 0` 表示查询第 0 个 NUMA 节点）

### 12.2.8.virsh freepages

`virsh freepages` 命令用于查询宿主机上特定 NUMA 节点中不同大小内存页面的空闲数量，主要用于监控大页内存的可用情况，对优化虚拟机内存分配（尤其是大页内存配置）非常有价值。

其参数含义如下：

`--all`：显示所有 NUMA 节点的空闲页面信息

`--cellno <num>`：指定要查询的 NUMA 节点编号（如 `--cellno 0` 表示查询第 0 个 NUMA 节点）

`--pagesize <size>`：指定要查询的页面大小（单位为 KiB，如 4096 表示 4KB 页面，2048 表示 2MB 页面等）

示例用法：

查看所有 NUMA 节点的空闲页面情况：

```
bash
```

```
virsh freepages --all
```

查看第 0 个 NUMA 节点中 2MB 页面的空闲数量：

```
bash
```

```
virsh freepages --cellno 0 --pagesize 2048
```

命令输出通常包含指定条件下的页面大小、总页数、空闲页数等信息，帮助管理员判断是否有足够的特定大小内存页面可供虚拟机使用，尤其适合需要配置大页内存的高性能场景（如数据库、虚拟化等）。

### 12.2.9.virsh node-memory-tune

用于配置和查看宿主机的共享内存（shared memory）相关调优参数，这些参数主要影响系统对共享内存的管理策略，进而影响虚拟机的内存使用效率。

从你的输出和参数来看，主要涉及以下共享内存相关配置：

主要参数含义：

`--shm-pages-to-scan`

含义：每次扫描共享内存时检查的页面数量（默认 100）。

作用：系统会定期扫描共享内存页面，寻找可合并的重复页面（通过 KSM 机制，即内核 Same Page Merging），此参数控制每次扫描的范围。

`--shm-sleep-millisecs`

含义：两次扫描 扫描间隔时间（毫秒，默认 20）。

作用：控制两次扫描之间的等待时间，间隔越短，内存 合并越及时，但会增加 CPU 开销。

`--shm-merge-across-nodes`

含义：是否允许跨 NUMA 节点合并共享内存页面（1 = 允许，0 = 禁止，默认 1）。

作用：在多 NUMA 节点的服务器上，允许跨节点合并可减少总体内存占用，但可能增加跨节点内存访问延迟。

输出信息解读：

plaintext

共享内存：

|                          |                      |
|--------------------------|----------------------|
| shm_pages_to_scan 100    | # 每次扫描 100 个页面       |
| shm_sleep_millisecs 20   | # 扫描间隔 20 毫秒         |
| shm_pages_shared 0       | # 已合并的共享内存页面数        |
| shm_pages_sharing 0      | # 正在被共享的页面数（每个共享页被多个 |
| 进程使用）                    |                      |
| shm_pages_unshared 0     | # 未被共享的页面数           |
| shm_pages_volatile 0     | # 易变页面数（频繁修改，不适合合并）  |
| shm_full_scans 0         | # 完整扫描的次数            |
| shm_merge_across_nodes 1 | # 允许跨 NUMA 节点合并      |

这些指标反映了当前共享内存的使用和合并状态，`shm_pages_shared` 为 0 通常表示暂时没有可合并的重复页面，或 KSM 机制未活跃工作。

实际用途：

优化内存使用率：通过调整扫描参数（如增加 `shm-pages-to-scan` 或缩短 `shm-sleep-millisecs`），可让系统更积极地合并重复内存页面（如多个虚拟机的相同系统镜像页），减少总内存占用。

平衡性能开销：若服务器 CPU 压力大，可增大 `shm-sleep-millisecs` 减少扫描频率，降低 CPU 消耗（代价是内存合并效率下降）。

多 NUMA 场景：若对延迟敏感，可关闭 `shm-merge-across-nodes`（设为 0），避免跨节点内存访问。

示例：修改参数

```
bash
# 增加每次扫描的页面数为 200
virsh node-memory-tune --shm-pages-to-scan 200
# 延长扫描间隔到 50 毫秒(降低 CPU 开销)
virsh node-memory-tune --shm-sleep-millisecs 50
```

调整后可通过 `virsh node-memory-tune` 再次查看生效情况。

## 12.2.10.virsh capabilities

`virsh capabilities` 是 `libvirt` 工具集中用于查询宿主机虚拟化能力的重要命令，它会以 XML 格式输出宿主机支持的虚拟化技术、CPU 特性、设备支持等详细信息，是了解宿主机虚拟化能力的核心命令。

主要输出内容：

虚拟化技术支持：

宿主机支持的虚拟化类型（如 KVM、QEMU、Xen 等）

虚拟化架构（如 x86\_64、arm64 等）

CPU 特性：

物理 CPU 的型号、架构

支持的 CPU 指令集（如 VMX、SVM 等虚拟化扩展）

可向虚拟机暴露的 CPU 特性（用于虚拟机 CPU 配置）



内存与存储支持：

内存页面大小支持

存储虚拟化相关能力

设备与总线支持：

支持的虚拟设备类型（如网卡、磁盘控制器等）

总线类型（如 PCI、USB 等）

安全特性：

支持的安全模块（如 SELinux、AppArmor 集成）

虚拟化安全技术（如 Intel SGX、AMD SEV 等）

使用方法：

```
bash
```

```
virsh capabilities
```

由于输出是 XML 格式，可结合 xmllint 美化显示：

```
bash
```

```
virsh capabilities | xmllint --format -
```

典型用途：

验证宿主机是否支持特定虚拟化技术

规划虚拟机配置（如 CPU 特性、设备类型）

排查虚拟化兼容性问题

自动化部署中获取宿主机能力信息

该命令输出内容较为专业，主要供系统管理员或虚拟化平台开发者参考，用于了解宿主机的虚拟化潜力和限制。

### 12.2.11.virsh domcapabilities

`virsh domcapabilities` 命令用于查询特定条件下虚拟机（域）的能力和 supports 的功能，你列出的是该命令的主要参数，用于指定查询的条件。以下是各参数的说明：

`--arch <架构>`：指定 CPU 架构（如 `x86_64`、`aarch64` 等），查询该架构下的虚拟机能力

`--emulatorbin <路径>`：指定模拟器二进制文件的路径，查询该模拟器支持的功能

**--machine <机器类型>**: 指定虚拟机的机器类型（如 `pc-i440fx-rhel7.0.0`），查询该类型的支持能力

**--virttype <虚拟化类型>**: 指定虚拟化类型（如 `kvm`、`qemu`、`xen` 等），查询该类型下的功能支持

```
virsh maxvcpus
```

`c` 命令用于查询宿主机或特定虚拟机类型支持的最大虚拟 CPU（vCPU）数量，`--type` 是其核心参数，用于指定查询的对象类型。

语法与参数说明

```
bash
```

```
virsh maxvcpus --type <类型>
```

其中 `--type` 可指定的类型包括：

**domain**: 查询单个虚拟机（`domain`）最多可配置的 vCPU 数量（受宿主机硬件和虚拟化层限制）。

**host**: 查询宿主机最多可支持的总 vCPU 数量（所有虚拟机的 vCPU 总和上限）。

示例用法

查询单个虚拟机最多可配置的 vCPU 数量：

```
bash
```

```
virsh maxvcpus --type domain
```

输出示例：256（表示单台虚拟机最多可配置 256 个 vCPU）

查询宿主机支持的总 vCPU 数量上限：

```
bash
```

```
virsh maxvcpus --type host
```

输出示例：1024（表示宿主机上所有虚拟机的 vCPU 总和最多为 1024）

实际意义

该命令帮助管理员了解虚拟化环境的 vCPU 配置上限，避免因过度分配 vCPU 导致的配置失败或性能问题。

结果受宿主机物理 CPU 核心数、虚拟化技术（如 KVM）限制、内核参数等因素影响。例如，32 核物理 CPU 的宿主机，通常支持的总 vCPU 数量会高于物理核心数（超配），但存在上限。

kvm: KVM 虚拟化类型的虚拟机

qemu: 纯 QEMU 模拟的虚拟机

xen: Xen 虚拟化类型的虚拟机（若系统支持）

### 12.2.12.virsh nodesuspend

`virsh nodesuspend` 命令用于将宿主机（物理机）置于挂起（suspend）状态，也称为“节点暂停”。该命令会冻结宿主机的运行状态，将其内存数据保存到磁盘，然后关闭物理机电源；后续可通过 `virsh noderesume` 命令恢复到之前的运行状态，类似于物理机的“休眠”功能。

基本语法

bash

`virsh nodesuspend [--target <目标状态>] [--reason <原因>]`

主要参数

`--target <状态>`: 指定宿主机暂停后的目标状态，常见值为 `disk`（将内存数据写入磁盘后关机，默认值）。

`--reason <原因>`: 可选参数，用于记录暂停宿主机的原因（如维护、节能等），便于后续追溯。

作用与特性

工作原理:

执行命令后，宿主机会将当前内存中的所有数据（包括运行中的虚拟机状态、系统进程等）写入磁盘的休眠镜像文件。

完成数据写入后，宿主机会自动关闭电源，进入断电状态（但休眠镜像保留在磁盘中）。

恢复方式:

需通过 `virsh noderesume` 命令或物理启动宿主机，系统会从休眠镜像中恢复内存数据，回到暂停前的运行状态（包括所有虚拟机的运行状态）。

适用场景:

临时关闭宿主机以节省电力（如夜间或低负载时段），同时保留当前工作状态。

宿主机需要短时间维护（如硬件检查），但希望避免重启后重新部署虚拟机。

注意事项

兼容性：该命令依赖宿主机的硬件和固件支持（如 ACPI 休眠功能），部分服务器可能不支持此特性。

数据安全：休眠镜像文件存储在磁盘中，若磁盘损坏可能导致数据丢失，建议执行前确保数据已备份。

虚拟机影响：暂停宿主机时，所有运行中的虚拟机也会被一并冻结，恢复后将继续运行（状态与暂停前一致）。

与虚拟机暂停的区别：`virsh suspend` 是暂停单个虚拟机，而 `virsh nodesuspend` 是暂停整个宿主机（包括所有虚拟机和系统进程）。

示例

将宿主机暂停并将内存数据写入磁盘：

```
bash
```

```
virsh nodesuspend --target disk --reason "夜间节能"
```

恢复被暂停的宿主机（通常在物理启动后执行）：

```
bash
```

```
virsh noderesume
```

该命令在需要临时停用宿主机但保留工作状态的场景中非常实用，尤其适合对服务中断时间敏感的环境。

分享

如何在 KVM 中使用 `virsh noderesume` 命令恢复宿主机？

哪些操作系统支持 `virsh nodesuspend` 命令？

如何配置 `virsh nodesuspend` 命令以实现特定的暂停行为？

从你的命令提示来看，`virsh nodesuspend` 命令支持 `--duration` 和 `--target` 两个主要参数，具体含义和用法如下：

1. `--target <状态>`（核心参数）

指定宿主机暂停后的目标状态，即“以何种方式暂停宿主机”，常见取值：

**disk**：将宿主机内存数据写入磁盘（休眠镜像），然后断电（默认值，即“休眠”状态）。

**mem**：将内存数据保留在物理内存中，仅关闭其他硬件电源（即“待机”状态，依赖持续供电）。

示例：

```
bash
```

# 以“休眠”方式暂停（数据写入磁盘，支持断电）`virsh nodesuspend --target disk`

# 以“待机”方式暂停（数据存内存，不可断电）`virsh nodesuspend --target mem`

2. `--duration <秒数>`（可选参数）

指定宿主机暂停的时长，到期后自动恢复运行（仅部分系统支持，依赖硬件和 `libvirt` 版本）。

示例：

`bash`

# 暂停宿主机 3600 秒（1 小时）后自动恢复 `virsh nodesuspend --target disk --duration 3600`

注意事项

若未指定 `--target`，默认使用 `disk`（休眠），数据写入磁盘，支持断电后通过物理启动恢复。

`--duration` 功能并非所有系统都支持，若执行后未自动恢复，需手动启动宿主机。

操作前确保宿主机有足够磁盘空间（`--target disk` 需存储内存镜像），且虚拟机状态正常。

使用时根据需求选择暂停方式：需要断电选 `disk`，短期快速恢复选 `mem`。

### 12.2.13.virsh allocpages

用于管理宿主机内存页面分配的命令，主要用于预留特定大小的内存页面给虚拟机使用，有助于优化内存性能，特别是针对需要大页内存的场景。

从你列出的选项来看，常用的参数组合及含义如下：

`--add`：添加内存页面分配（与其他参数配合使用）

`--all`：对所有 NUMA 节点操作

`--cellno <num>`：指定 NUMA 节点编号（如 `--cellno 0` 表示操作第 0 个 NUMA 节点）

`--pagecount <num>`：指定要分配的页面数量

`--pagesize <size>`：指定页面大小（如 4096 表示 4KB，2048 表示 2MB 等，需宿主机支持）

## 13.virsh 其它命令

### 13.1.help 帮助关键字

virsh 有 261 条命令，如何记忆，需要理出一个思路来，好在 virsh 给出了一个大致的思路，就是用 help+关键字来显示每个虚拟化管理领域的命令群。

virsh help virsh 显示内建的相关命令

virsh help domain 显示虚拟机相关操作的命令

virsh help monitor 获得统计信息相关的命令

virsh help host 显示 host 与 hypervisor 相关的命令

virsh help checkpoint 检查点相关命令

virsh help interface 网卡相关命令

virsh help network 网络池相关命令

virsh help nodedev 节点相关命令

virsh help secret 加密相关命令

virsh help pool 存储池相关的命令

virsh help volume 卷相关的命令

virsh help backup 备份相关的命令

### 13.2.virsh 短命令

-c, --connect URI: 连接到指定的 URI，就像执行 “connect” 命令一样，而不是使用默认连接。

-d, --debug LEVEL: 启用级别为整数 LEVEL 及以上的调试信息。LEVEL 的取值范围为 0 到 4（默认值为 4）。有关每个级别的详细说明，请参见下文对 VIRSH\_DEBUG 环境变量的介绍。

-e, --escape string: 为 “console” 命令设置替代的转义序列。默认使用 telnet 的 “^]”。使用脱字符（^）表示法时，允许使用的字符包括字母字符、@、[、]、,、^、\_。

-h, --help: 忽略所有其他参数，且等效于执行 “help” 命令。

-k, --keepalive-interval INTERVAL: 设置发送保活消息的时间间隔（以秒为单位），用于检查与服务器的连接是否仍然有效。将间隔设置为 0 会禁用客户端保活机制。

-K, --keepalive-count COUNT: 设置在未收到服务器响应的情况下，可发送保活消息的次数，超过该次数则判定连接失效。如果 INTERVAL（保活间隔）设置为 0，此设置将不起作用。

-l, --log FILE: 将日志详细信息输出到指定的文件 FILE 中。

-q, --quiet: 避免输出额外的信息性消息。

-r, --readonly: 创建只读的初始连接，就像执行“connect”命令时使用“--readonly”选项一样。

-t, --timing: 输出每个命令的执行时间信息。

-v, --version [=short]: 忽略所有其他参数，并打印 virsh 所依赖的 libvirt 库的版本信息。

-V, --version=long: 忽略所有其他参数，打印 virsh 所依赖的 libvirt 库的版本信息，以及编译时包含的选项和驱动程序。

### 13.3.virsh 基本命令

**virsh 正常模式:** 在 shell 中执行 virsh 子命令 参数等,用途多为脚本使用或者习惯用 virsh 开头

**virsh 交互模式:** 在 shell 输入 virsh 回车，进入了 virsh 交互模式，交互模式的好处是不需要每次都要带 virsh，减少重复输入。

virsh 基本命令就是在交互模式中使用。

|         |                      |
|---------|----------------------|
| cd      | 更改当前目录               |
| echo    | 对参数进行编码              |
| exit    | 退出当前的交互式终端           |
| help    | 显示帮助信息               |
| pwd     | 显示当前目录               |
| quit    | 退出当前的交互式终端           |
| connect | 连接（或重新连接）管理程序（即虚拟化层） |