虚拟化技术实现 — QEMU-KVM & Libvirt 原创

云物互联 2021-07-14 16:51:01

文章目录

前文列表

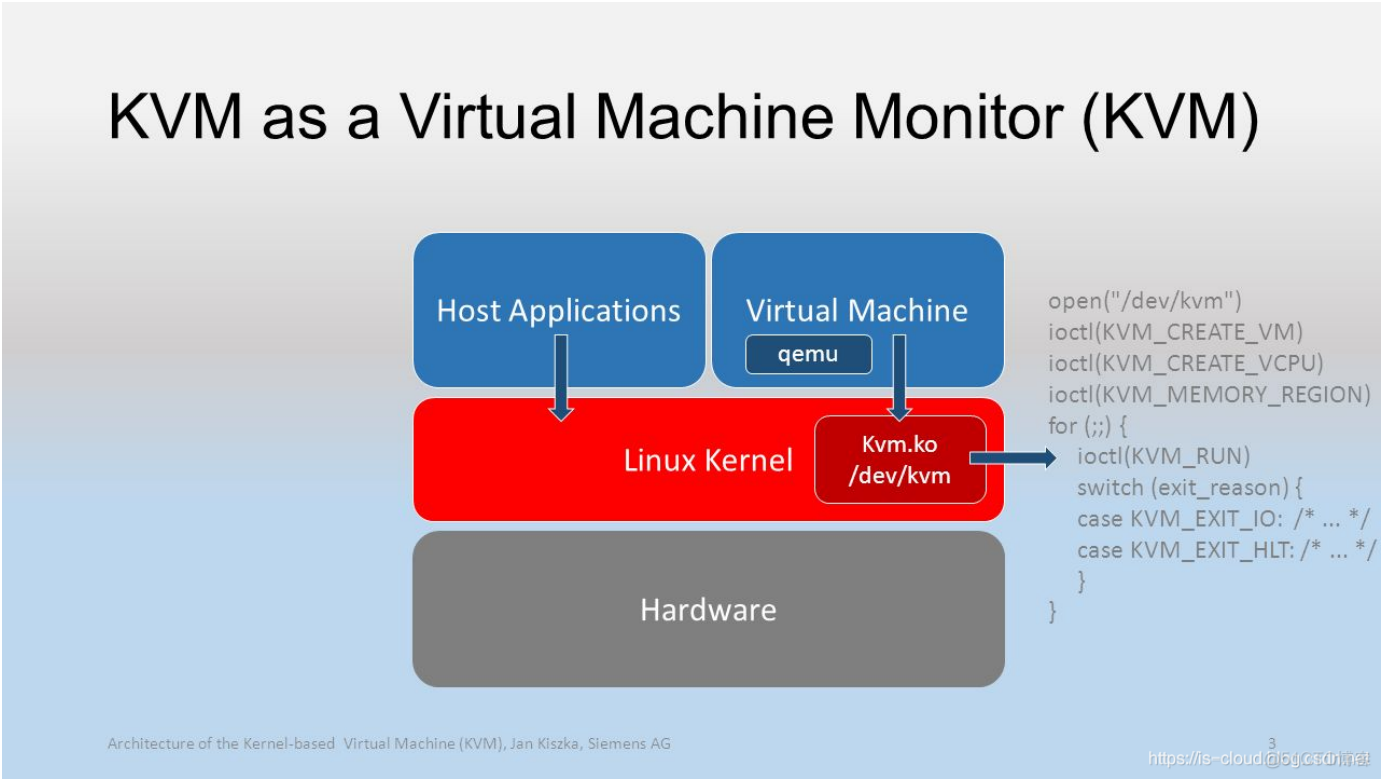
《虚拟化技术实现 — 虚拟化技术发展编年史》

KVM

KVM（Kernel-based Virtual Machine，基于内核的虚拟机）是一种用于 Linux 内核中的虚拟化基础设施。本质是一个嵌入到 Linux 内核中的虚拟化功能模块 kvm.ko（kvm-intel.ko/kvm-AMD.ko），该模块在利用 Linux 内核所提供的部分操作系统能力（e.g. 任务调度、内存管理、硬件设备交互）的基础上，再加入了处理器和内存虚拟化的能力，使得 Linux 内核具备了成为 VMM 的条件。

- KVM 内核模块本身只能提供 CPU 和内存的虚拟化。
- KVM 需要在具备 Intel VT 或 AMD-V 功能的 x86 平台上运行，所以 KVM 也被称之为硬件辅助的虚拟化实现。
- KVM 包含一个提供给 CPU 底层虚拟化实现可加载的内核模块 kvm.ko（kvm-intel.ko、kvm-AMD.ko）。

KVM 于 2007 年 2 月 5 日被集成到 Linux 2.6.20 内核中。使用 KVM 的前提是宿主机必须拥有支持硬件虚拟化拓展特性（Intel VT 或者 AMD-V）的处理器。



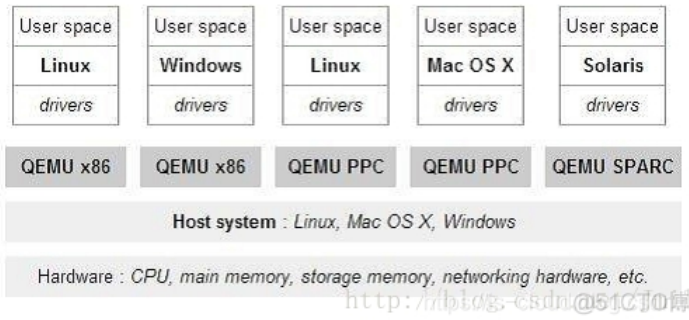
KVM 的功能清单：

- 支持 CPU 和 Memory 超分（Overcommit）
- 支持半虚拟化 I/O（virtio）
- 支持热插拔 （CPU、块设备、网络设备等）
- 支持 SMP（Symmetric Multi-Processing，对称多处理）处理器架构
- 支持 NUMA （Non-Uniform Memory Access，非一致存储访问）处理器架构
- 支持实时迁移（Live Migration）
- 支持 PCI 设备直接分配（Pass-through）和单根 I/O 虚拟化 （SR-IOV）
- 支持合并相同内存页 （KSM ）

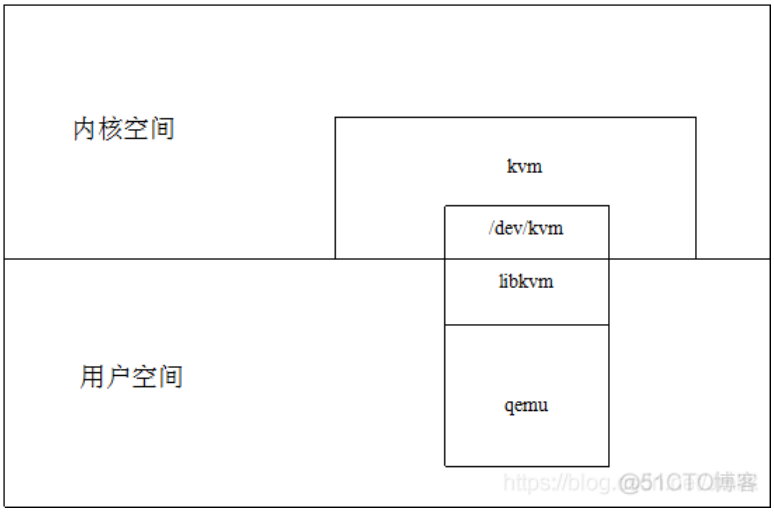
以 Intel VT 为例，当启动 Linux 操作系统并加载 KVM 内核模块时：

1. 初始化 KVM 模块内部的数据结构；
2. KVM 模块检测当前的 CPU 体系结构，然后打开 CPU 控制器以及存取 CR4 寄存器的虚拟化模式开关，并通过执行 VMXON 指令将 Host OS/VMM（在 KVM 环境中，Host OS 即是 VMM）置于虚拟化模式的根模式（Root Mode）；
3. 最后，KVM 模块创建特殊的接口设备文件 /dev/kvm 并等待来自用户空间（QEMU）的指令。

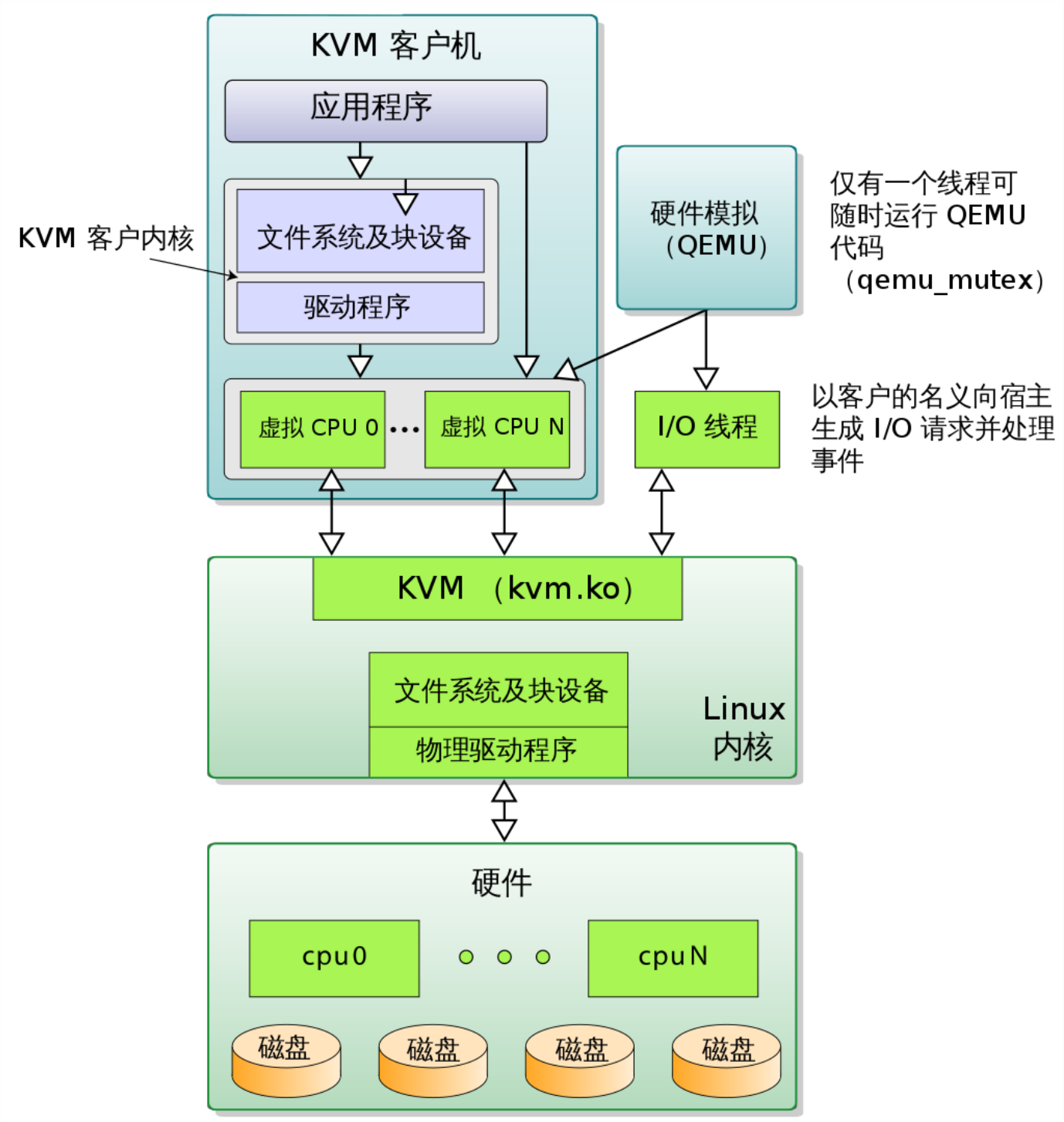
但需要注意的是，KVM 是运行在内核态的且本身不能进行任何设备的模拟。所以，KVM 还必须借助于一个运行在用户态的应用程序来模拟出虚拟机所需要的虚拟设备（e.g. 网卡、显卡、存储控制器和硬盘）同时为用户提供操作入口。目前这个应用程序的最佳选择就是 QEMU。
QEMU
QEMU（Quick Emulator）是一款免费的、开源的、纯软件实现的、可执行硬件虚拟化的 VMM。与 Bochs，PearPC 等模拟器相比，QEMU 具有高速（配合 KVM）以及跨平台的特性。



事实上，QEMU 本身作为一套完整的 VMM 实现，包括了处理器虚拟化，内存虚拟化，以及模拟各类虚拟设备的功能。QEMU 4.0.0 版本甚至几乎可以模拟任何硬件设备，但由于这些模拟都是纯软件实现的，所以其性能低下。
在 KVM 开发者在对 QEMU 进行稍加改造后，QEMU 可以通过 KVM 对外暴露的 /dev/kvm 接口来对其进行调用。从 QEMU 角度来看，也可以说是 QEMU 使用了 KVM 的处理器和内存虚拟化功能，为自己的虚拟机提供了硬件辅助虚拟化加速。



此外，虚拟机的配置和创建、虚拟机运行所依赖的虚拟设备、虚拟机运行时的用户环境和用户交互，以及一些虚拟机的特定技术，比如：动态迁移，都是交由 QEMU 来实现的。



总的来说，QEMU 具有以下几种使用方式：

1. **纯软件（二进制翻译）实现的全虚拟化虚拟机**
2. **基于硬件辅助虚拟化（KVM）的全虚拟化虚拟机**
3. **仿真器**：为用户空间的进程提供 CPU 仿真，让在不同处理器结构体系上编译的程序得以跨平台运行。例如：让 SPARC 架构上编译的程序在 x86 架构上运行（借由 VMM 的形式）。

QEMU-KVM

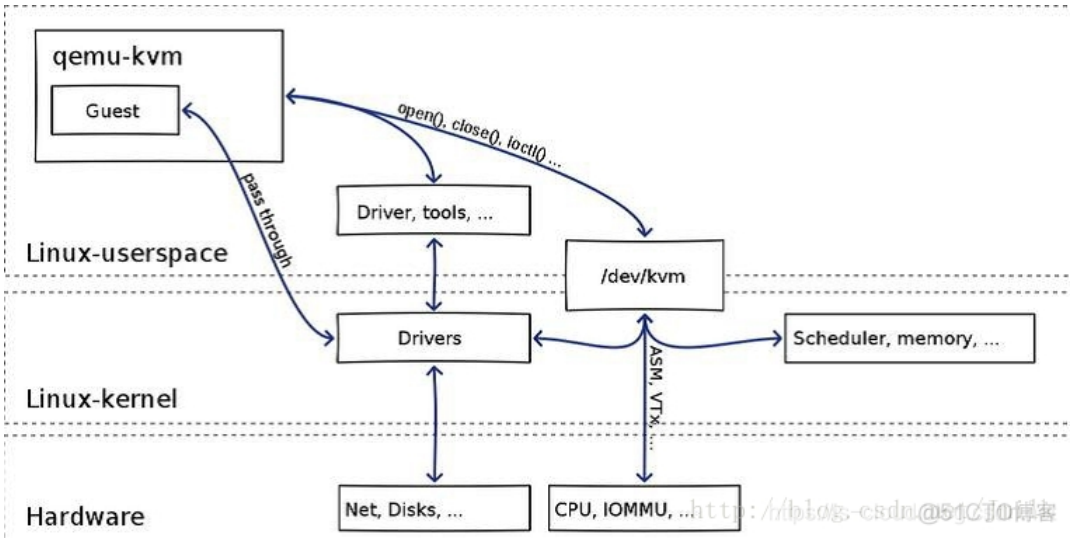
KVM 官方提供的软件包下载包含了 KVM 内核模块、QEMU、qemu-kvm 以及 virtio 四个文件。其中，qemu-kvm 本质是专门针对 KVM 的 QEMU 分支代码包（一个特殊的 QEMU 版本）。

QEMU-KVM 相比原生 QEMU 的改动：

- 原生的 QEMU 通过指令翻译实现 CPU 的完全虚拟化，但是修改后的 QEMU-KVM 会调用 ICOTL 命令来调用 KVM 模块。
- 原生的 QEMU 是单线程实现，QEMU-KVM 是多线程实现。

然而在 QEMU 1.3 版本之后两者又保持一致了，但我们能仍习惯在 KVM 语境中将其称之为 QEMU-KVM。

NOTE：在 RHEL6/CentOS6 中，qemu-kvm 存放在 /usr/libexec 目录下。不过 PATH 环境变量缺省是不包含此目录的，所以用户无法直接使用 qemu-kvm，这样做是为了防止 QEMU 替代了 KVM 作为 VMM 的角色。如果希望启用 QEMU 作为 VMM 的话，可以通过将 /usr/libexec/qemu-kvm 链接为 /usr/bin/qemu 来完成。



在 QEMU-KVM 中，KVM 运行在内核空间，提供 CPU 和内存的虚级化，以及 Guest OS 的 I/O 拦截。QEMU 运行在用户空间，提供硬件 I/O 虚拟化，并通过 ioctl 调用 /dev/kvm 接口将 KVM 模块相关的 CPU 指令传递到内核中执行。当 Guest OS 的 I/O 被 KVM 拦截后，就会将 I/O 请求交由 QEMU 处理。例如：

```
open("/dev/kvm", O_RDWR|O_LARGEFILE)    = 3
ioctl(3, KVM_GET_API_VERSION, 0)        = 12
ioctl(3, KVM_CHECK_EXTENSION, 0x19)     = 0
ioctl(3, KVM_CREATE_VM, 0)              = 4
ioctl(3, KVM_CHECK_EXTENSION, 0x4)      = 1
ioctl(3, KVM_CHECK_EXTENSION, 0x4)      = 1
ioctl(4, KVM_SET_TSS_ADDR, 0xfffbd000)  = 0
ioctl(3, KVM_CHECK_EXTENSION, 0x25)     = 0
ioctl(3, KVM_CHECK_EXTENSION, 0xb)      = 1
ioctl(4, KVM_CREATE_PIT, 0xb)           = 0
ioctl(3, KVM_CHECK_EXTENSION, 0xf)      = 2
ioctl(3, KVM_CHECK_EXTENSION, 0x3)      = 1
ioctl(3, KVM_CHECK_EXTENSION, 0)        = 1
ioctl(4, KVM_CREATE_IRQCHIP, 0)         = 0
ioctl(3, KVM_CHECK_EXTENSION, 0x1a)     = 0
```

QEMU-KVM 调用 KVM 内核模块启动虚拟机的流程概要

1. 获取 /dev/kvm fd（文件描述符）

```
kvmfd = open("/dev/kvm", O_RDWR);
```

2. 创建虚拟机，获取虚拟机的句柄。KVM_CREATE_VM 时，可以理解成 KVM 为虚拟机创建了对应的数据结构，然后，KVM 会返回一个文件句柄来代表该虚拟机。针对这个句柄执行 ioctl 调用即可完成对虚拟机执行相应的管理，比如：创建用户空间虚拟地址（Virtual Addr

ess）、客户机物理地址（Guest Physical Address）以及主机物理地址（Host Physical Address）之间的映射关系；

```
vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);
```

3. 为虚拟机映射内存和其他的 PCI 设备，以及信号处理的初始化。

```
ioctl(kvmfd, KVM_SET_USER_MEMORY_REGION, &mem);
```

4. 将虚拟机镜像数据映射到内存，相当于物理机的 boot 过程，把操作系统内核映射到内存。
5. 创建 vCPU，并为 vCPU 分配内存空间。KVM_CREATE_VCPU 时，KVM 为每一个 vCPU 生成对应的文件句柄，对其执行相应的 ioctl 调用，就可以对 vCPU 进行管理。

```
ioctl(kvmfd, KVM_CREATE_VCPU, vcpuid);
vcpu->kvm_run_mmap_size = ioctl(kvm->dev_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
```

6. 创建 vCPU 个数的线程并运行虚拟机。

```
ioctl(kvm->vcpus->vcpu_fd, KVM_RUN, 0);
```

7. 线程进入循环，监听并捕获虚拟机退出原因，做相应的处理。这里的退出并不一定指的是虚拟机关机，虚拟机如果遇到 I/O 操作，访问硬件设备，缺页中断等都会执行退出。执行退出可以理解为将 CPU 执行上下文返回到 QEMU。

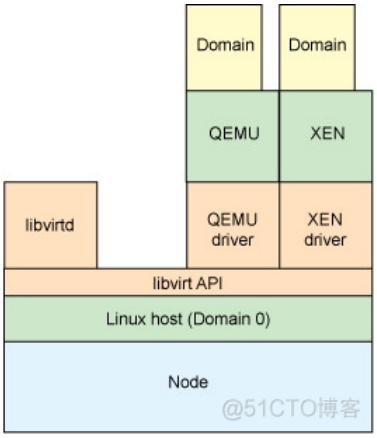
```
open("/dev/kvm")

ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)

for (;;) {
     ioctl(KVM_RUN)
     switch (exit_reason) {  /* 分析退出原因，并执行相应操作 */
     case KVM_EXIT_IO:  /* ... */
     case KVM_EXIT_HLT:  /* ... */
     }
}
```

Libvirt

Libvirt 是目前使用最为广泛的异构虚拟化管理工具及 API，由应用程序编程接口库、Libvirtd Daemon、virsh CLI 组成。



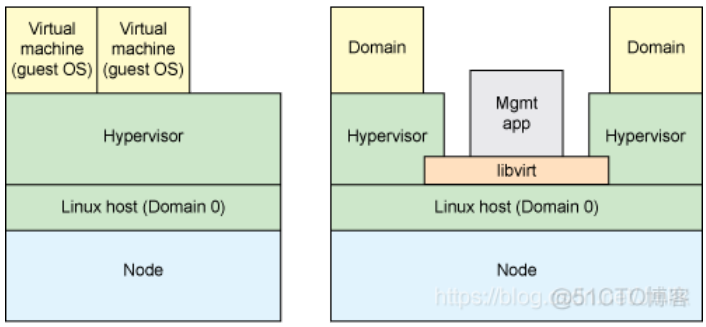
其中 Libvirtd Daemon 负责调度和管理虚拟机，而且这个 Daemon 可以分为 root 权限和普通用户权限的两种 libvirtd。root 权限，可以虚拟出物理主机的各种设备。

开启 root 权限 libvirtd：

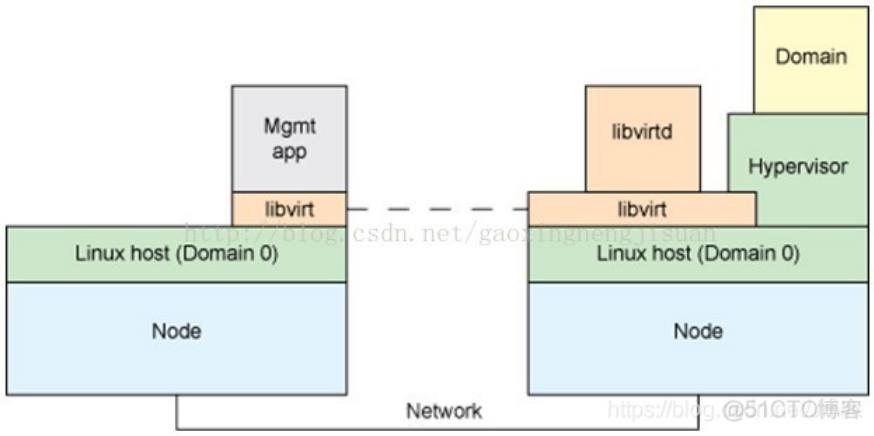
```
1. sudo libvirtd --daemon
```

Libvirtd Daemon 还可供本地或远程的 virsh 调用，所以 Libvirt 具有两种控制方式：

1. 本地控制管理：Management Application 和 Domain 在同一个 Node 上。

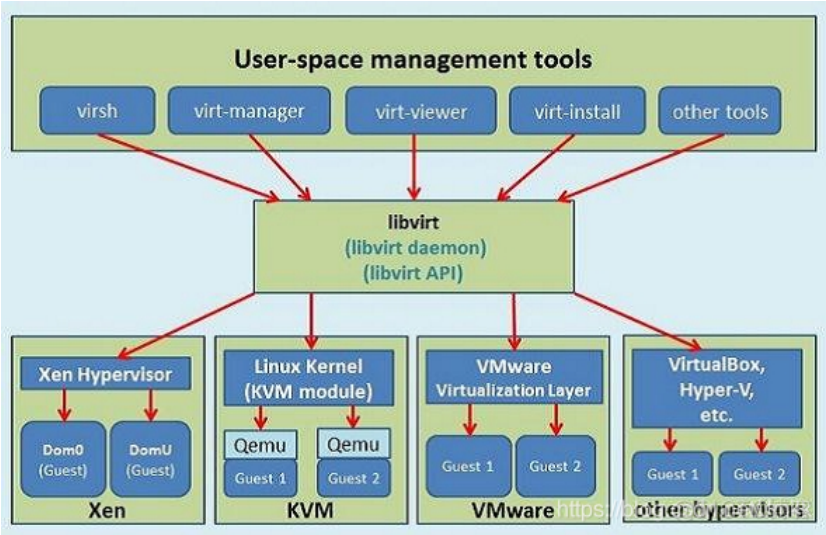


2. 远程控制管理：Management Application 和 Domain 不再同一个 Node 上。该模式使用了运行于远程节点上的 libvirtd 守护进程，当在其他节点上安装 Libvirt 时 Management Application 会自动启动，并且会自动确定本地虚拟机的监控程序和为虚拟机安装驱动程序。Management Application 通过一种通用协议从本地 Libvirt 连接到远程 libvirtd。



此外，Libvirt 引用了面向驱动的架构设计，对所有的虚拟化技术都提供了相应的驱动和统一的接口，所以 Libvirt 支持异构的虚拟化技术。同时 Libvirt 提供了多种语言的编程接口，可以通过程序调用这些接口来实现对虚拟机的操作。由此可见，Libvirt 具有非常强的可扩展性，OpenStack 与 Libvirt 关系密切。

例如：Libvirt 将 XML 格式的配置转换为 qemu-cli 调用。它还提供了一个管理守护进程来配置子进程（e.g. QEMU），因此 QEMU 可以不使用 root 权限。Openstack Nova 在启动虚机时，Nova 会使用 Libvirt 为每个虚机启动一个 QEMU 子进程。



在 KVM 中，可以使用 virsh CLI 来调用 libvirtd，libvirtd 再通过调用 qemu-kvm 来操作 KVM 虚拟机。