# 虚拟化技术

## 概念描述

虚拟化就是由位于下层的模块，通过向上一层的模块提供一个与他原先所期待的运行环境一致的接口方法，抽象出一个虚拟的软件或硬件接口，使得上层软件可以直接运行在虚拟环境上。虚拟化用一个通熟易懂的词语解释，就是“欺骗”，通过模访下层原有的功能模块，来欺骗上层。可以联想“中间人攻击”还有现实中的皮包公司来理解。

虚拟化指的是对某些事物进行虚拟化创建的行为，例如对硬件计算平台、操作系统、存储设备、网络资源等进行虚拟化，其通过抽象出一个虚拟的软件或硬件接口，从而向上层软件提供一个与它原先所期待的运行环境完全一致的接口，最终使得上层软件可以直接运行在虚拟环境上。

VMM全称是Virtual Machine Monitor，虚拟机监控系统，也叫Hypervisor，是虚拟化层的具体实现。

在虚拟化中，物理资源通常有一个定语称为宿主机（Host Machine），其上如果运行操作系统，则称为宿主操作系统（Host OS）。虚拟出来的资源通常有一个定语称为客户机（Guest Machine）或虚拟机（Virtual Machine），其上运行的操作系统称为客户操作系统（Guest OS）。虚拟机可以看作是物理机的一种高效隔离的复制，其蕴含三层含义：

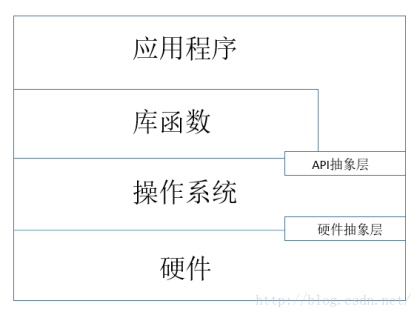
（1）同质。同质指的是虚拟机的运行环境和物理机的运行环境在本质上是相同的，但是在表现上可以有一些差异。例如，虚拟机所看到的处理器个数可以和物理机上实际的处理器个数不同，但是物理机上和虚拟机中看到的处理器必须是统一类型的。

（2）高效。高效指的是虚拟机中运行的软件有接近在物理机上直接运行的性能。

（3）资源受控。资源受控指的是虚拟化中间件需对系统资源有完全控制能力和管理权限，包括资源的分配、监控和回收。

理解虚拟化的不同内涵，需要从首先回顾计算机的层次结构。几乎所有的计算机遵守着相同的层级结构。分别是：

|  |
| --- |
| 应用程序层 |
| 函数库层 |
| API抽象层 |
| 操作系统层 |
| 硬件抽象层 |
| 硬件层 |



自底向上，形成依赖关系，这种分层的做法带来如下好处：

1）每一层都向上提供了接口，降低系统的复杂性和软件的可移植性。

2）每一层只需要知道下一层的抽象接口即可，不需要知道其内部运作机制

3）降低系统设计的复杂性

4）提高软件可移植性

## 基本原理

**硬件虚拟化**

硬件虚拟化就是硬件物理平台本身提供了对特殊指令的截获和重定向的支持。支持虚拟化的硬件，也是一些基于硬件实现软件虚拟化技术的关键。在基于硬件实现软件虚拟化的技术中，在硬件是实现虚拟化的基础，硬件(主要是CPU)会为虚拟化软件提供支持，从而实现硬件资源的虚拟化。支持虚拟化的硬件有：

* Intel-VT(Intel Virtualization Technology)，Intel公司为解决纯软件虚拟化解决方案在可靠性、安全性和性能上的不足而引进的技术。它可以让一个CPU工作起来像多个CPU在并行运行，从而使得在一部电脑内同时运行多个操作系统成为可能
* AMD-V(AMD Virtualization)，是AMD公司的虚拟化技术。它是对x86处理器系统架构的一组硬件扩展和硬件辅助虚拟化技术，可以简化纯软件的虚拟化解决方案，改进VMM（虚拟机监视器）的设计，更充分地利用硬件资源，提高服务器和数据中心的虚拟化效率

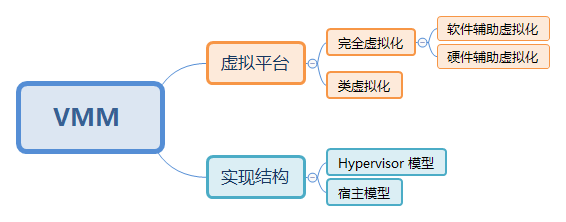
**软件虚拟化**

软件虚拟化就是利用软件技术，在现有的物理平台上实现对物理平台访问的截获和模拟。在软件虚拟化技术中，有些技术不需要硬件支持，如：QEMU；而有些软件虚拟化技术，则依赖硬件支持，如：VMware、KVM。对软件虚拟化进行细分，又可以分为以下几类：

* 完全虚拟化：（Full Virtualization）虚拟机模拟完整的底层硬件环境和特权指令的执行过程，使客户机操作系统可以独立运行。支持完全虚拟化的软件有：Parallels Workstation、VirtualBox、Virtual Iron、Oracle VM、Virtual PC、Virtual Server、Hyper-V、VMware Workstation、**QEMU**等
* 硬件辅助虚拟化：（Hardware-assisted Virtualization）是指通过硬件辅助支持模拟运行环境，使客户机操作系统可以独立运行，实现完全虚拟化的功能。支持硬件辅助虚拟化的软件有：**Linux KVM**、VMware Workstation、VMware Fusion、Virtual PC、**Xen**、VirtualBox、Parallels Workstation等
* 部分虚拟化：（Partial Virtualization）只针对部分硬件资源进行虚拟化，虚拟机模拟部分底层硬件环境，特别是地址空间。这样的环境支持资源共享和线程独立，但是不允许建立独立的客户机操作系统。
* 平行虚拟化：（Para-Virtualization）虚拟机不需要模拟硬件，而是将部分硬件接口以软件的形式提供给客户机操作系统。如：早期的Xen。
* 操作系统层虚拟化：（OS-level virtualization）这种技术将操作系统内核虚拟化，可以允许使用者空间软件实例被分割成几个独立的单元，在内核中运行，而不是只有一个单一实例运行。这个软件实例，也被称为是一个容器（containers）、虚拟引擎（Virtualization engine）、虚拟专用服务器（virtual private servers）。每个容器的进程是独立的，对于使用者来说，就像是在使用自己的专用服务器。**Docker**容器技术就是属于操作系统层虚拟化的范畴。

## 运行模式

VMM 根据平台类型和实现结构有两种不同的分类，按平台类型可以分为完全虚拟化和类虚拟化。



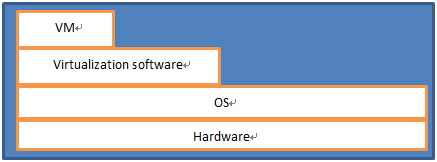
**软件辅助虚拟化**是通过优先级压缩（Ring Compression）和二进制代码翻译（Binary Translation）这两个技术来完成的。简单讲，RC 基于CPU特权级的原理，也就是guest、VMM和host分别处于不同的特权级上（这个后面讲CPU虚拟化的时候会详述），guest要访问host就属于越级访问，会抛异常，这时VMM会截获这个异常，并模拟出其可能的行为，从而进行相应处理。但这个问题很明显，就是由于硬件设计的缺陷，有些指令并不能截获，从而导致“漏洞”。

BT可以弥补这个缺陷，它通过去扫描guest的二进制的代码，将难以虚拟化的指令转为支持虚拟化的指令，从而可以配合VMM完成虚拟化功能。这两种方式都是通过「打补丁」的方式来辅助虚拟化，很难再架构上保证完整性。

所以，后期的硬件厂商就在硬件上对虚拟化提供了支持，有了硬件辅助的虚拟化。通过对硬件本身加入更多的虚拟化功能，就可以截获更多的敏感指令，填补上漏洞。在这一块，Intel 的VT-x/d技术和 AMD的AMD-V技术是其中的代表。

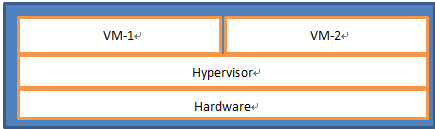
而**类虚拟化**则是另外一种通过软件来避免漏洞的方式，就是通过修改 guest 操作系统内核代码（API 级）来避免漏洞，这种方式好处就是可以自定义内核的执行行为，某种程度上对性能进行优化。

Type-I：直接运行在操作系统之上的虚拟化，模式如下图：

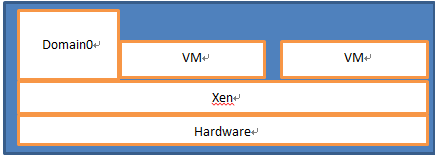


如：Vmware workstations、**Kvm**等。

Type-II：直接运行在硬件之上的（提供各种硬件驱动），模式如下图：



如：Vmware EXSI、Xen等。但是Xen有点特别，虽然也是直接安装在硬件之上，提供Hypervisor，但是只负责CPU、内存、中断，不提供I/O驱动，需要额外安装一个虚拟机再安装一个Linux系统用来管理I/O设备，如下图：



Type-III：其他类型

当然，除了上面提到的基于操作系统或直接基于硬件的虚拟化外，还有如下常见的类型。

**容器虚拟化**

基于内核的虚拟化，所有的虚拟机都是一个独立的容器，但共同运行硬件之上，使用着同一个内核。优点就是速度快，部署容易，缺点就是相互间的资源相互隔离比较麻烦，但现在市场也都有了相对成熟的解决方案。如，如今大火的**Docker**，网上都有人说Docker具有取代虚拟化的势头。

**模拟器虚拟化**

通过模拟器模拟所有的硬件，如**QEMU**，KVM就是使用QEMU。

**库虚拟化**

通过在操作系统之上模拟出不同系统的库，如Linux上运行Wine就可以支持Windows上的软件运行，Windows上运行Cywin就可以支持Linux上的软件运行。因为现在操作系统都是遵循POSIX标准，所以各自提供的库接口都是同一个标准，只需要在对应的平台上运行一个可以提供对方库的软件，然后在此软件之上运行针对对方系统编译好的软件即可。为什么要运行针对对方平台编译好的软件，因为虽然库统一了，但是各自的ABI（应用二进制接口）接口还是不同的。

出现过以下主要虚拟化技术或产品：

* **Xen** - 由剑桥大学开发的，一款开源的虚拟机监视器。采用ICA协议，它通过一种叫做准虚拟化的技术来获取高性能，甚至在一些与传统虚拟技术极度不友好的架构上（如：x86），Xen也有极佳的表现。Xen属于半虚拟化的技术，所以其性能损失非常小。Xen没有指令翻译，其或者使用使能理解和翻译虚拟操作系统发出的未修改指令的CPU（即：完全虚拟化）；或者修改操作系统，使它发出的指令最优化，便于在虚拟化环境中执行（即：准虚拟化）。
* **KVM** - KVM是一个Linux kernel模块，可以使用modprobe来加载KVM，加载后还需要通过其他工具创建虚拟机。**KVM是一个全虚拟化的解决方案**，但**需要CPU支持虚拟化功能**。相比Xen来说，KVM可以更加方便的整合进Linux内核，但它还需要其它虚拟化软件（如：QEMU）才能实现虚拟化功能。
* **LXC** - 即：Linux Container，Linux容器，是一种轻量级的虚拟化的手段。它可以提供轻量级的虚拟化，以隔离进程和资源，而且不需要提供指令解释机制以及全虚拟化的其他复杂性。容器会有效地将由单个操作系统管理的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求。
* **OpenVZ** - 是SWsoft公司开发的开源软件，是该公司Virtuozzo软件的基础产品，是基于Linux平台的操作系统级服务器虚拟化解决方案。通过OpenVZ，可以在单个物理服务器上创建多个相互隔离的虚拟专用服务器(VPS)并以最大的效率共享硬件和管理资源。其上运行虚拟服务器被称为VPS（Virtual Private Serve），每个VPS的运行环境和独立服务器完全一致。OpenVZ基于Linux系统内核及作业系统提供操作系统级虚拟化，在虚拟化过程中资源消耗非常小，官方宣称约1-2%。
* **Hyper-V**- 是微软件推出的一种虚拟化技术，可以采用半虚拟化或全虚拟的方式创建虚拟机。虽然它可以创建Windows或Linux操作系统，但其本身只能运行在Windows系统下，使用范围较为有限。
* **Oracle VM** - Oracle推出的服务器虚拟化软件，基于开源的Xen技术，包括Oracle VM Server和Oracle VM Manager两部分。
* **VMWare** - 是一家非常出名虚拟化软件公司，其产品涵盖服务器、桌面等各种虚拟化领域，如：
  + **VMware Workstation**- 是一款桌面虚拟机软件，可以在一台实体机器上模拟完整的网络环境，并可运行多个Windows、DOS、Linux或Mac系统，是非常好的开发、测试、部署解决方案。从技术角度来说，VMware Workstation是一款完全虚拟化产品，可借助硬件辅助在不修改用户操作系统的情况下完整虚拟化操作系统。
  + **VMware ESX Server**- 是一款适用于任何系统环境的企业级的虚拟机软件，可以认为是VMware Server的升级版。相比VMware Workstation来说，其功能更加强大，可以用于构建高伸缩和高可靠企业级服务器，并可实现远程管理、高级资源管理控制等高级功能。

**虚拟机**：存在于硬件层和操作系统层间的虚拟化技术。

虚拟机通过“伪造”一个硬件抽象接口，将一个操作系统以及操作系统层以上的层嫁接到硬件上，实现和真实物理机几乎一样的功能。

**容器**：存在于操作系统层和函数库层之间的虚拟化技术。

容器通过“伪造”操作系统的接口，将API抽象层，函数库层以上的功能置于操作系统上，以Docker为例，就是基于Linux操作系统的Namespace和CGroup功能实现的隔离容器，模拟操作系统的功能，因为它比虚拟机高了一层，也就需要少一层东西，所以容器占用资源少。

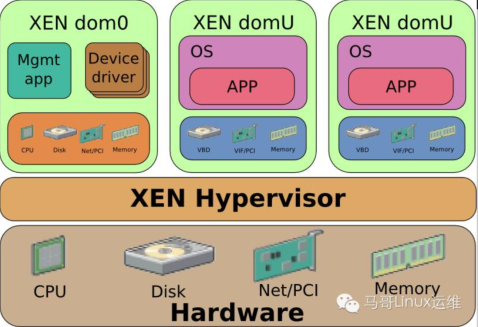
**JVM**：存在于函数库层和应用程序之间的虚拟化技术。

Java虚拟机具有跨平台特性，所谓跨平台特性实际上就是虚拟化的功劳，我们Java语言是调用操作系统函数库的，然而不同操作系统的函数库互不相同，JVM就是建立一个虚拟化层，对下通过不同的版本适应不同的操作系统，对上提供统一的运行环境交给程序和开发者。通过虚拟化实现了如今蔚为壮观的Java生态圈以及建立在JVM基础上的其他语言，如Scala，Clojure，Kotlin等。

## 主流虚拟化

### XEN

<http://blog.51cto.com/wzlinux/1727106>



第一部分：Xen Hypervisor，又称虚拟机监控程序(Virtual Machien Monitor简称VMM），VMM工作原有linux系统内核位置，替代了linux系统内核，用于虚拟CPU、Memeory等；

第二部分：Xen Dom0，又称特殊区域；为vmm提供硬件驱动程序，用于协助vmm驱动各个底层硬件，同时又为Xen DomU提供模拟IO等功能；由于特殊原因Linux-2.6.37以后的内核才支持Xen Dom0，建议大家在使用xen时使用linxu-3.0以后的内核；

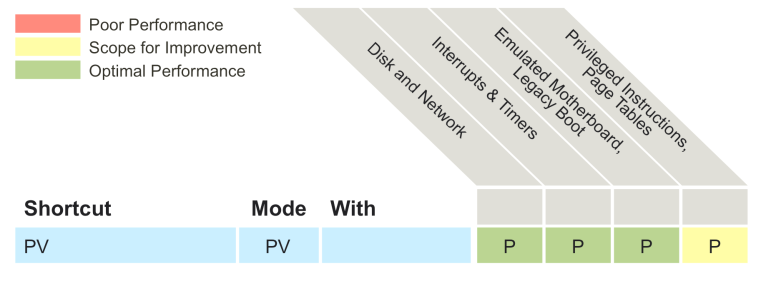
第三部分：Xen DomU，非特权区域；实际生产中的操作系统

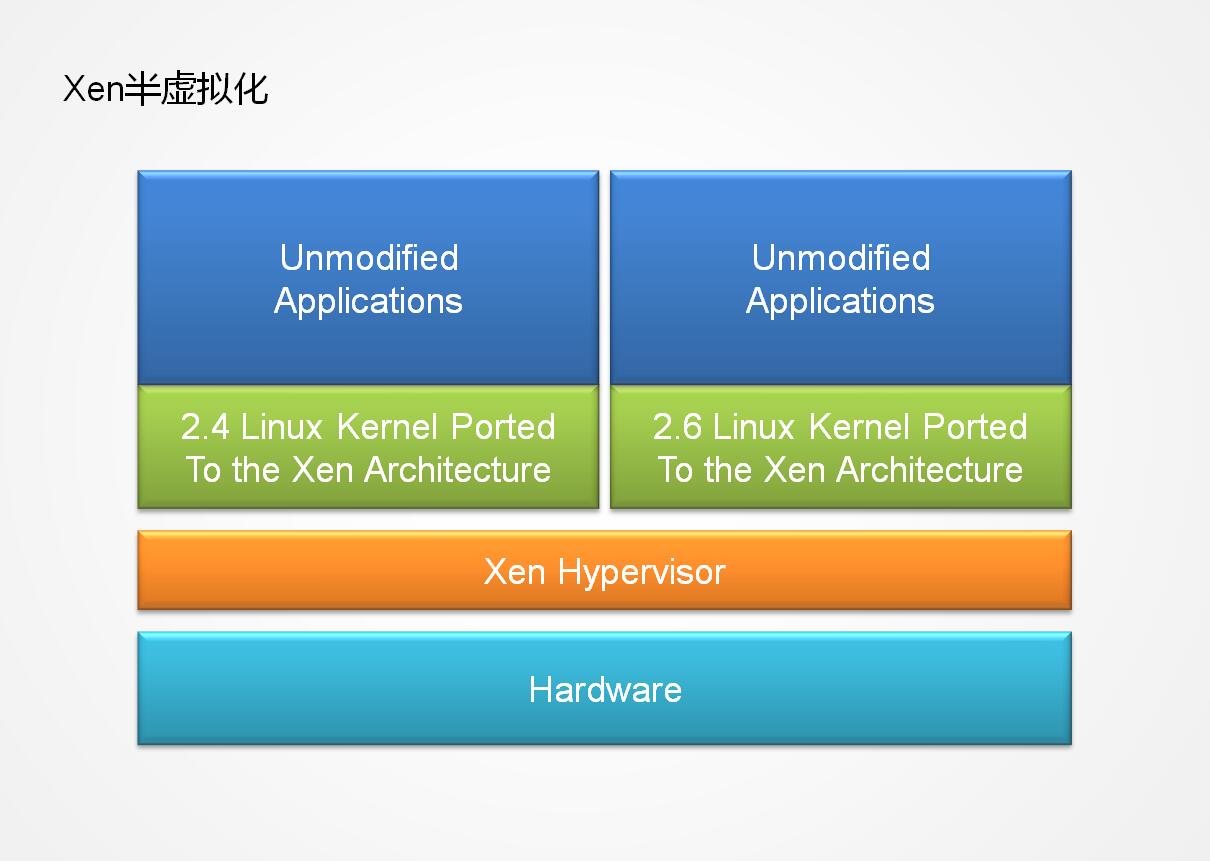
#### Xen虚拟化类型

Xen对虚拟机的虚拟化分为两大类，半虚拟化（Para virtualization）和完全虚拟化（Hardware VirtualMachine）。

**1、半虚拟化(PV)**

虚拟化（Paravirtualization）有些资料称为“超虚拟化”，简称为PV，是Xen主导的虚拟化技术。这种技术允许虚拟机操作系统感知到自己运行在Xen Hypervisor上而不是直接运行在硬件上，同时也可以识别出其他运行在相同环境中的客户虚拟机。

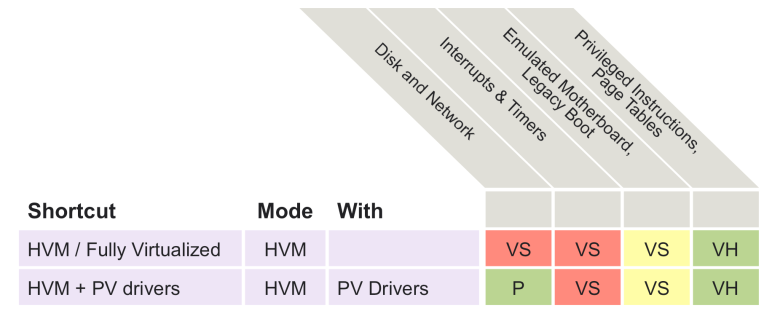


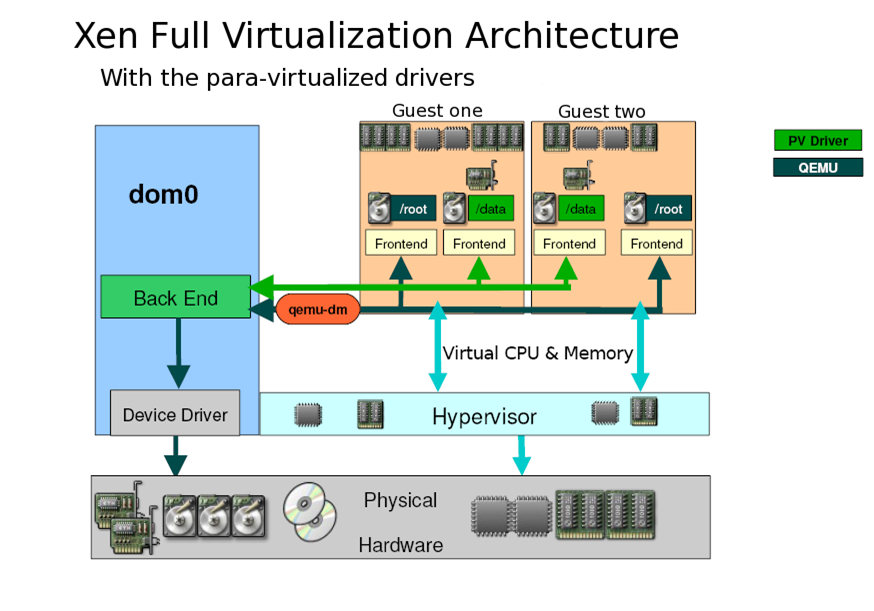


**2、完全虚拟化(HVM)**

完全虚拟化（Hardware Virtual Machine）又称“硬件虚拟化”，简称HVM，是指运行在虚拟环境上的虚拟机在运行过程中始终感觉自己是直接运行在硬件之上的，并且感知不到在相同硬件环境下运行着其他虚拟机的虚拟技术。

在Xen Hypervisor运行的完全虚拟化虚拟机，所运行的操作系统都是标准的操作系统，即：无需任何修改的操作系统版本。同时也需要提供特殊的硬件设备。 值的注意的是，在Xen上虚拟的Windows虚拟机必须采用完全虚拟化技术。

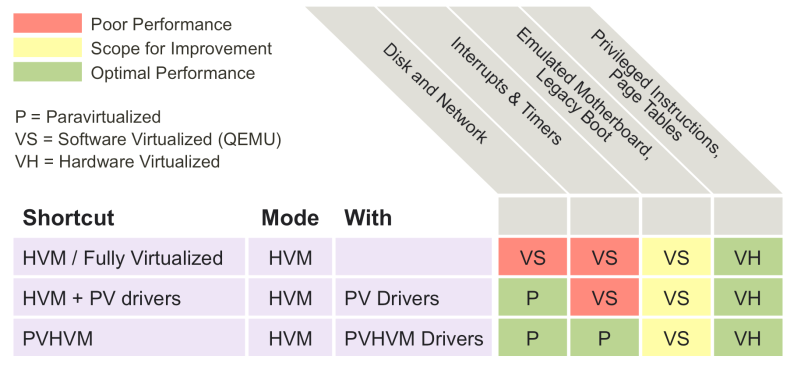


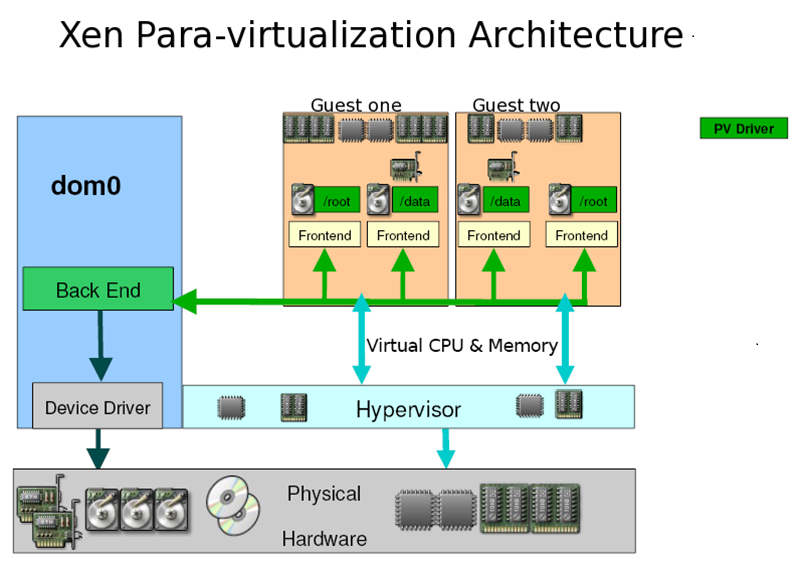


**3、CPU完全虚拟化，IO半虚拟化(PVHVM)**

为了提高性能，完全虚拟化的Guests可以使用特殊的半虚拟设备驱动程序（PVHVM或PV-on-HVM驱动）。这些驱动程序在HVM环境下优化你的PV驱动，模拟的磁盘和网络IO旁路运行，从而让你的PV在HVM中有更好的性能。这意味着你在用户窗口操作方面可以得到最佳的性能。

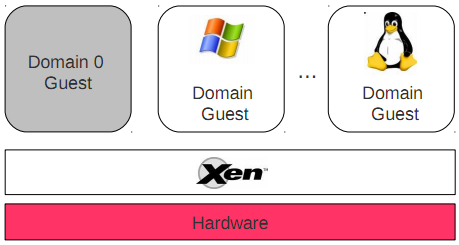
注意，Xen项目PV（半虚拟化）的Guest自动使用PV驱动：因此不需要这些驱动程序,因为你已经自动使用优化的驱动程序。PVHVM只会在HVM（全虚拟化）guest虚拟机中需要。





#### Xen基本组件

* Xen Hypervisor：直接运行于硬件之上是Xen客户操作系统与硬件资源之间的访问接口(如：)。通过将客户操作系统与硬件进行分类，Xen管理系统可以允许客户操作系统安全，独立的运行在相同硬件环境之上。
* Domain 0：运行在Xen管理程序之上，具有直接访问硬件和管理其他客户操作系统的特权的客户操作系统。
* DomainU：运行在Xen管理程序之上的普通客户操作系统或业务操作系统，不能直接访问硬件资源（如：内存，硬盘等），但可以独立并行的存在多个。



**1、Xen Hypervisor**

Xen Hypervisor是直接运行在硬件与所有操作系统之间的基本软件层。它负责为运行在硬件设备上的不同种类的虚拟机（不同操作系统）进行CPU调度和内存分配。Xen Hypervisor对虚拟机来说不单单是硬件的抽象接口，同时也控制虚拟机的执行，让他们之间共享通用的处理环境。

Xen Hypervisor不负责处理诸如网络、外部存储设备、视频或其他通用的I/O处理。

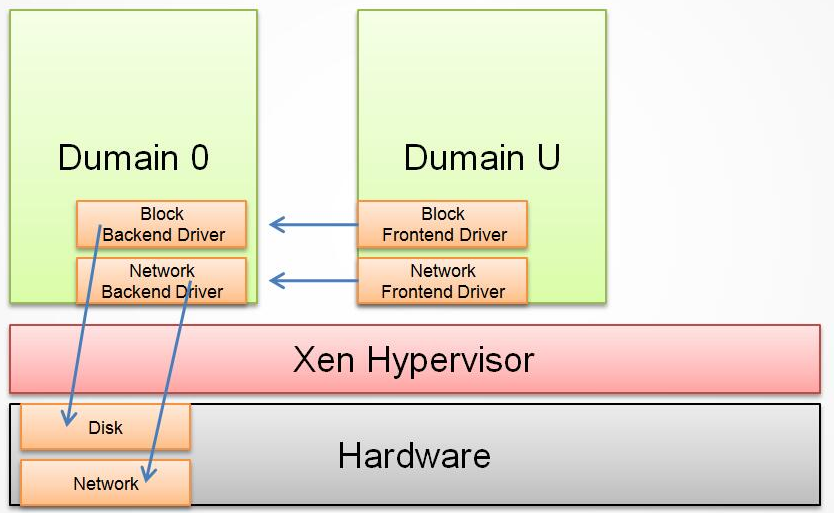
**2、Domain 0**

Domain 0 是经过修改的Linux内核，是运行在Xen Hypervisor之上独一无二的虚拟机，拥有访问物理I/O资源的特权，并且可以与其他运行在Xen Hypervisor之上的其他虚拟机进行交互。所有的Xen虚拟环境都需要先运行Domain 0，然后才能运行其他的虚拟客户机。

Domain 0 在Xen中担任管理员的角色，它负责管理其他虚拟客户机。

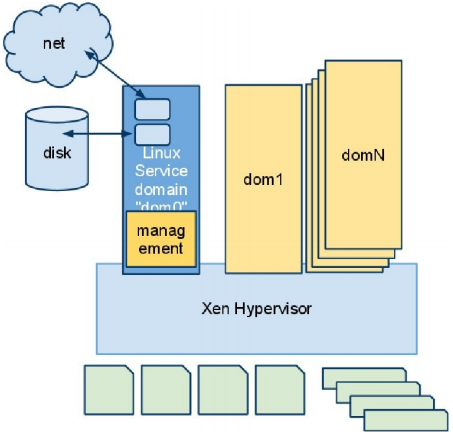
在Domain 0中包含两个驱动程序，用于支持其他客户虚拟机对于网络和硬盘的访问请求。这两个驱动分别是Network Backend Driver和Block Backend Driver。

Network Backend Driver直接与本地的网络硬件进行通信，用于处理来自Domain U客户机的所有关于网络的虚拟机请求。根据Domain U发出的请求Block Backend Driver直接与本地的存储设备进行通信然后，将数据读写到存储设备上。



Domain U客户虚拟机没有直接访问物理硬件的权限。所有在Xen Hypervisor上运行的半虚拟化客户虚拟机（简称：Domain U PV Guests）都是被修改过的基于Linux的操作系统、Solaris、FreeBSD和其他基于UNIX的操作系统。所有完全虚拟化客户虚拟机（简称：Domain U HVM Guests）则是标准的Windows和其他任何一种未被修改过的操作系统。

无论是半虚拟化Domain U还是完全虚拟化Domain U，作为客户虚拟机系统，Domain U在Xen Hypervisor上运行并行的存在多个，他们之间相互独立，每个Domain U都拥有自己所能操作的虚拟资源（如：内存，磁盘等）。而且允许单独一个Domain U进行重启和关机操作而不影响其他Domain U。



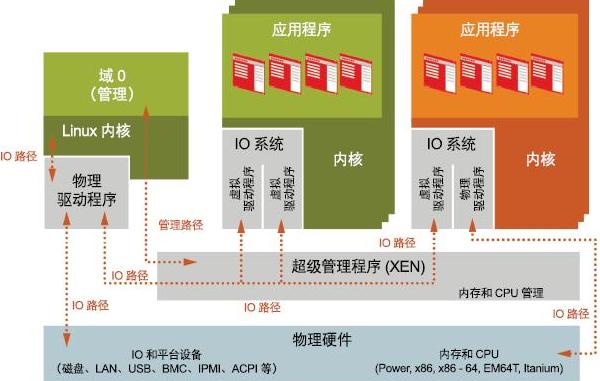
#### Xen体系架构

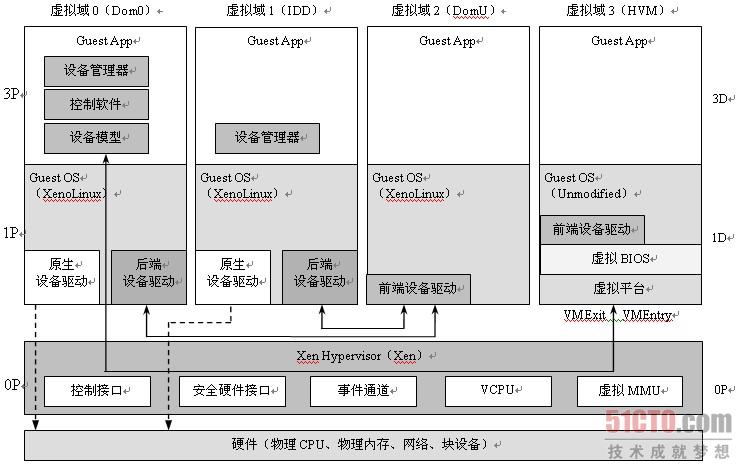
Xen 的 VMM ( Xen Hyperviso ) 位于操作系统和硬件之间，负责为上层运行的操作系统内核提供虚拟化的硬件资源，负责管理和分配这些资源，并确保上层虚拟机（称为域 Domain）之间的相互隔离。Xen采用混合模式，因而设定了一个特权域用以辅助Xen管理其他的域，并提供虚拟的资源服务，该特权域称为Domain 0，而其余的域则称为Domain U。

Xen向Domain提供了一个抽象层，其中包含了管理和虚拟硬件的API。Domain 0内部包含了真实的设备驱动（原生设备驱动），可直接访问物理硬件，负责与 Xen 提供的管理 API 交互，并通过用户模式下的管理工具来管理 Xen 的虚拟机环境。

Xen2.0之后，引入了分离设备驱动模式。该模式在每个用户域中建立前端（front end）设备，在特权域(Dom0)中建立后端(back end)设备。所有的用户域操作系统像使用普通设备一样向前端设备发送请求，而前端设备通过IO请求描述符(IO descripror ring)和设备通道（device channel）将这些请求以及用户域的身份信息发送到处于特权域中的后端设备。这种体系将控制信息传递和数据传递分开处理。

在Xen体系结构设计中，后端设别运行的特权域被赋予一个特有的名字---隔离设备域(Isolation Device Domain, IDD)，而在实际设计中，IDD 就处在Dom0中。所有的真实硬件访问都由特权域的后端设备调用本地设备驱动 (native device drive)发起。前端设备的设计十分简单，只需要完成数据的转发操作，由于它们不是真实的设备驱动程序，所以也不用进行请求调度操作。而运行在IDD中的后端设备，可以利用Linux的现有设备驱动来完成硬件访问，需要增加的只是IO请求的桥接功能---能完成任务的分发和回送。





Xen向Domain提供了VCPU、虚拟MMU（Memory Management Unit）等基本的虚拟硬件和基本机制（事件通道），并向Domain提供了虚拟域管理API（控制接口）。对Domain设备的访问，Xen也提供了相应的硬件接口（安全硬件接口），以保证设备访问得以安全进行。

在Xen体系架构中涉及到的基本概念包括：

**制接口**：仅能被Dom0使用，用于帮助Dom0控制和管理其他的Domain。控制接口提供的具体功能包括Domain的创建、销毁、暂停、恢复及迁移，对其他Domain的CPU调度、内存分配及设备访问等。

**全硬件接口**：提供除虚拟CPU、MMU之外的所有硬件虚拟工作，包括DMA/IO、驱动程序、虚拟的PCI地址配置、虚拟硬件中断等。该接口只能被具有原生设备驱动的Domain使用，而向其他Domain则仅通过设备通道提供虚拟硬件服务。

**vCPU**：Xen为每个Domain建立了VCPU结构，用以接收Guest OS中传递的指令，其中大部分的指令被VCPU直接提交到物理CPU执行，而对于特权指令则需要经过确认后交由Xen代为执行。

**虚拟MMU**：用于帮助Guest OS完成虚拟地址到机器地址的转换。Xen系统中增加了客户物理地址层，因而地址由原来的二层结构变为三层结构。Xen通过虚拟MMU仍能使用硬件MMU来完成地址转换。

**事件通道**：是用于Domain和Xen之间、Domain相互之间的一种异步事件通知机制，用于处理Guest OS中的虚拟中断、物理中断以及Domain之间的通信。

**设备管理器**：位于Dom0和IDD（Independent Device Domain）中，可作为系统BIOS的扩展，用于向所有的设备提供通用的管理接口，并负责在Domain启动时加载特定的设备驱动、建立管理设备通道、提供硬件设备配置接口并处理设备访问错误。

<http://book.51cto.com/art/201003/188412.htm>

#### Domain管理和控制

开源社区中将一系列的Linux精灵程序分类为“管理”和“控制”两大类。这些服务支撑着整个虚拟环境的管理和控制操作，并且存在于Domain 0虚拟机中。

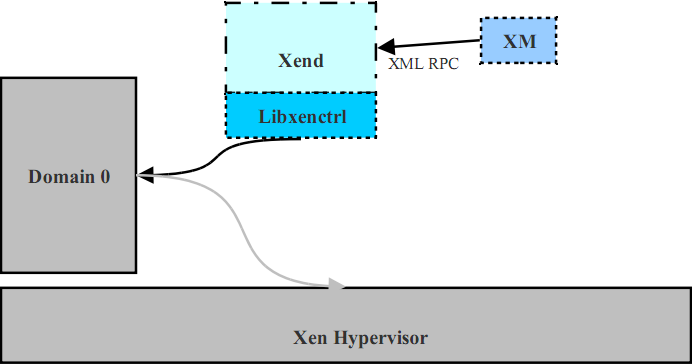
下面将对直接服务进行详细的描述。

注：为了清晰的描述Xen的运行流程，画图时将精灵程序放在Domain 0外部来描述，但事实上所有精灵程序都存在于Domain 0 之中。

**3.1、Xend**

Xend精灵线程是一个Python应用程序，它作为Xen环境的系统管理员。它利用Libxenctrl类库向Xen Hypervisor发出请求。

所有Xend处理的请求都是由XM工具使用XML RPC接口发送过来的。



**3.2、Xm**

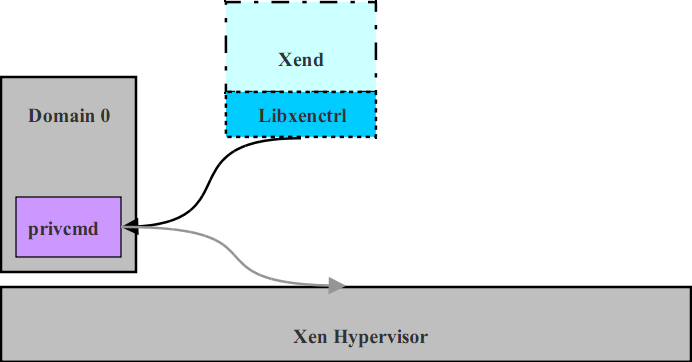
用于将用户输入通过XML RPC接口传递到Xend中的命令行工具。

**3.3、Xenstored**

Xenstored精灵程序用于维护注册信息，这些信息包括内存和在连接Domain 0和所有其他Domain U之间的事件通道。Domain 0虚拟机利用这些注册信息来与系统中其他虚拟机建立设备通道，即帮助Domain U虚拟机访问硬件资源。

3.4、Libxenctrl

Libxenctrl是C程序类库，用于让Xend具有通过Domain 0与Xen Hypervisor进行交互的能力。在Domain 0中存在一个特殊的驱动程序称作privcmd，它将请求发送给Hypervisor。



**3.5、Qemu-DM**

在Xen环境下，每个完全虚拟化虚拟机都需要拥有自己的Qemu精灵程序。Qemu-DM处理在Xen环境下完全虚拟化客户机所能允许执行的所有关于网络和磁盘请求和操作。Qemu程序必须存在于Hypervisor之外同时又需要访问网络和I/O，所以Qemu-DM必须存在于Domain 0 中（参见前面章节对Domain 0的描述）。

未来版本的Xen中，一种新的工具Stub-DM将会提供一系列对所有完全虚拟化客户机都可用的服务，以此来替代需要在每个虚拟机上都生成一个Qemu的逻辑。

**3.6、Xen Virtual Firmware**

Xen Virtual Firmware是被嵌入到所有完全虚拟化客户机中的虚拟的BIOS系统，来确保所有客户操作系统在正常启动操作中接收到标准的启动指令集并提供标准的软件兼容环境。

### KVM

**KVM 是基于CPU 辅助的全虚拟化方案，它需要CPU虚拟化特性的支持**

**Linux虚拟化KVM**

[**https://www.cnblogs.com/arnoldlu/category/1118000.html**](https://www.cnblogs.com/arnoldlu/category/1118000.html)

**KVM介绍**

<http://www.cnblogs.com/sammyliu/p/4543110.html>

<http://www.cnblogs.com/sammyliu/p/4543597.html>

KVM虚拟机源代码分析

<https://blog.csdn.net/sdulibh/article/details/51799150>

**KVM(Kernel-based Virtual Machine)基于内核的虚拟机**

[**http://www.cnblogs.com/sammyliu/p/4543110.html**](http://www.cnblogs.com/sammyliu/p/4543110.html)

[**http://zhangbin.junxilinux.com/?p=670**](http://zhangbin.junxilinux.com/?p=670) **KVM虚拟化详解**

**（1）介绍和安装**

**（2）CPU 和 内存虚拟化**

**（3）I/O QEMU 全虚拟化和准虚拟化（Para-virtulizaiton）**

**（4）I/O PCI/PCIe设备直接分配和 SR-IOV**

**（5）libvirt 介绍**

**（6）Nova 通过 libvirt 管理 QEMU/KVM 虚机**

**（7）快照 （snapshot）**

**（8）迁移 （migration）**

[**https://blog.opskumu.com/kvm-notes.html**](https://blog.opskumu.com/kvm-notes.html) **KVM 学习笔记**

**一、KVM 原理**

**二、KVM 基础功能**

**2.1 CPU**

**2.2 内存**

**2.3 存储**

**2.4 网络**

**三、KVM 高级功能**

**3.1 半虚拟化驱动**

**3.2 热插拔 (hot plugging)**

**3.3 动态迁移**

**四、KVM 管理工具**

**libvirt**

**五、参考文档**

[**KVM 介绍（1）：简介及安装**](http://www.cnblogs.com/sammyliu/p/4543110.html)

[**KVM 介绍（2）：CPU 和内存虚拟化**](http://www.cnblogs.com/sammyliu/p/4543597.html)

[**KVM 介绍（3）：I/O 全虚拟化和准虚拟化 [KVM I/O QEMU Full-Virtualizaiton Para-virtualization]**](http://www.cnblogs.com/sammyliu/p/4543657.html)

[**KVM 介绍（4）：I/O 设备直接分配和 SR-IOV [KVM PCI/PCIe Pass-Through SR-IOV]**](http://www.cnblogs.com/sammyliu/p/4548194.html)

[**KVM 介绍（5）：libvirt 介绍 [ Libvrit for KVM/QEMU ]**](http://www.cnblogs.com/sammyliu/p/4558638.html)

[**KVM 介绍（6）：Nova 通过 libvirt 管理 QEMU/KVM 虚机 [Nova Libvirt QEMU/KVM Domain]**](http://www.cnblogs.com/sammyliu/p/4568188.html)

[**KVM 介绍（7）：使用 libvirt 做 QEMU/KVM 快照和 Nova 实例的快照 （Nova Instances Snapshot Libvirt）**](http://www.cnblogs.com/sammyliu/p/4468757.html)

[**KVM 介绍（8）：使用 libvirt 迁移 QEMU/KVM 虚机和 Nova 虚机 [Nova Libvirt QEMU/KVM Live Migration]**](http://www.cnblogs.com/sammyliu/p/4572287.html)

[**KVM部署、使用、调优**](https://www.cnblogs.com/nmap/p/6369180.html)

**关于Linux虚拟化技术KVM的科普**

[**http://www.cnblogs.com/arnoldlu/p/6510618.html**](http://www.cnblogs.com/arnoldlu/p/6510618.html)

[**http://blog.chinaunix.net/uid-20201831-id-5775661.html**](http://blog.chinaunix.net/uid-20201831-id-5775661.html)

**KVM虚拟化技术之使用Qemu-kvm创建和管理虚拟机**

[**http://blog.51cto.com/sohudrgon/1613148**](http://blog.51cto.com/sohudrgon/1613148)

1）.KVM是开源软件，全称是kernel-based virtual machine（基于内核的虚拟机）。

2）.是x86架构且硬件支持虚拟化技术（如 intel VT 或 AMD-V）的Linux全虚拟化解决方案。

3）.它包含一个为处理器提供底层虚拟化 可加载的核心模块kvm.ko（kvm-intel.ko或kvm-AMD.ko）。

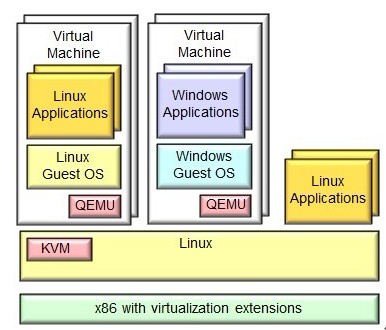
4）.KVM还需要一个经过修改的QEMU 软件（qemu-kvm），作为虚拟机上层控制和界面。

5）.KVM能在不改变linux或windows镜像的情况下同时运行多个虚拟机，（它的意思是多个虚拟机使用同一镜像）并为每一个虚拟机配置个性化硬件环境（网卡、磁盘、图形适配器……）。

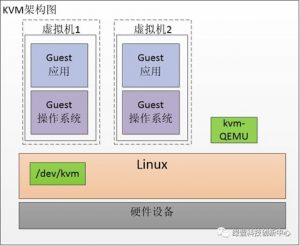
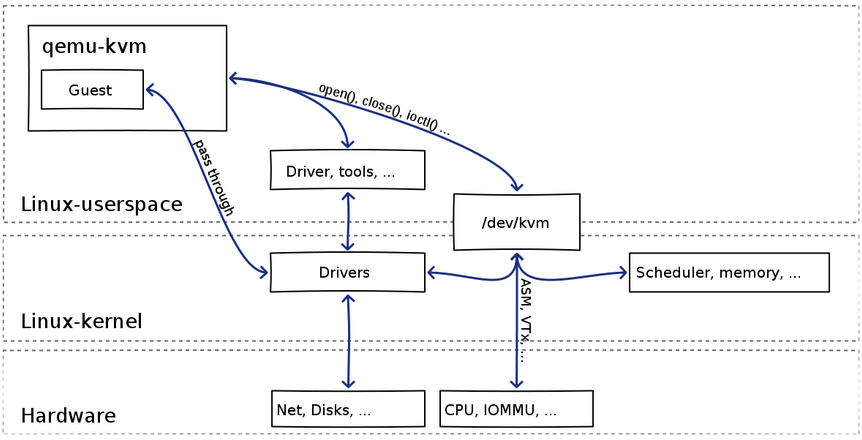
6）.在主流的Linux内核，如2.6.20以上的内核均已包含了KVM核心。

#### KVM基本概念

KVM（Kernel-based Virtual Machine）是支持虚拟化扩展（Intel VT 或 AMD-V技术）x86硬件的Linux完全虚拟化解决方案。它包括了一个可加载的内核模块kvm.ko，提供了核心的虚拟化架构以及一个处理器特定模块（kvm-intel.ko或kvm-amd.ko）。使用KVM，用户可以运行多个无需修改的Linux或Windows虚拟机。每个虚拟机有自己私有的虚拟硬件：网卡、磁盘、显卡等等。从Linux内核2.6.20开始，KVM就作为主线内核，KVM用户空间的组件包括在主线QEMU中。



**KVM 虚机的创建过程**



KVM 架构中，每个虚拟机其实就是一个常规的 Linux 进程，由 Linux 调度程序进行调度，因此，KVM使用Linux 内核的所有功能。如图所示，KVM就是Linux内核中的一个模块，用户空间通过使用修改的 QEMU 提供BIOS、PCI 总线、USB 总线和一组标准的设备（比如 IDE 和 SCSI 磁盘控制器、网络卡等）的模拟。

KVM是依赖于x86 CPU硬件支持基础之上的虚拟化技术， 目前KVM只能运行在Intel VT or AMD-V的硬件之上。查看是否支持KVM虚拟化，使用以下命令：

[root@localhost ~]# egrep '(vmx|svm)' /proc/cpuinfo

准确来说，KVM是Linux的一个模块。Linux系统使用modprobe等去加载KVM模块，但是用户无法直接控制内核模块去工作，还必须有一个用户空间的工具才行。这个工具就是已经成型的开源虚拟化软件QEMU。QEMU可虚拟化不同的CPU， 例如在x86的CPU上可虚拟一个Power的CPU，并利用他编译出可运行在Power上的程序。KVM使用了QEMU的一部分功能，并稍加改造就成了可控制KVM的用户空间工具。所以，官方提供的KVM下载有KVM模块、QEMU工具以及两者的合集。可以只升级KVM模块，也可只升级QEMU工具。

[root@localhost ~]# lsmod | grep kvm

kvm\_intel 162153 4

kvm 525409 1 kvm\_intel

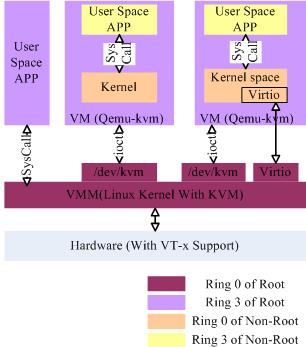
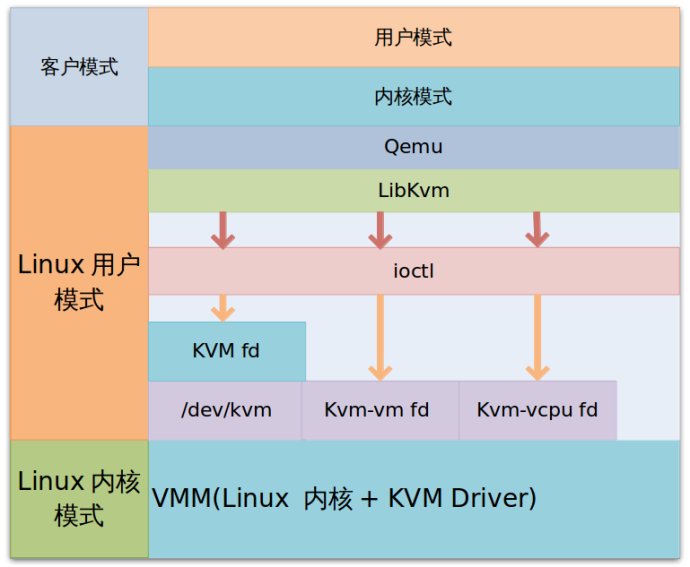
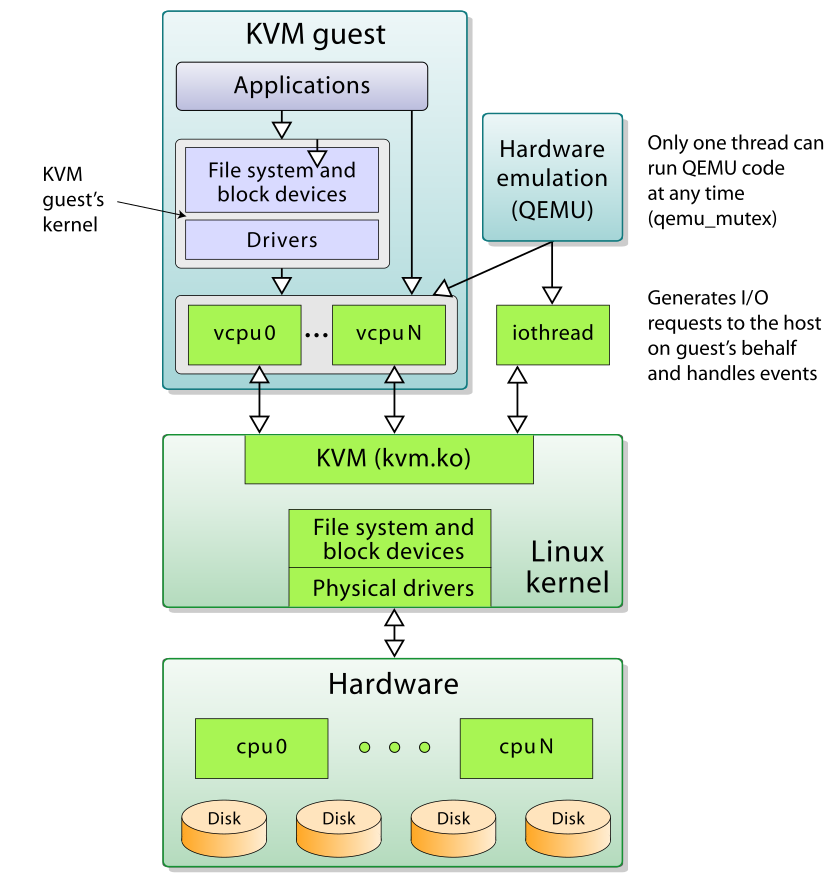
[root@localhost ~]# ps -ef | grep test-vm

KVM的基本结构由两个部分构成:

1) KVM驱动，作为linux kernel的一个模块，主要负责虚拟机的创建、虚拟内存的分配，vCPU寄存器的读写和运行

2）Qemu, 用于模拟虚拟机的用户空间组件，提供I/O设备模型，创建外设的途径 。如下图所示，kvm内核模块被看作是一个标准的linux字符集设备(/dev/kvm)。Qemu通过libkvm应用程序接口，使用fd通过ioctl向设备驱动来发送创建、运行虚拟机命令。

KVM是基于硬件辅助虚拟化技术(如Intel VT-x)的全虚拟化解决方案，如上图所示，说明如下：

VMM(即KVM内核)运行于根模式下的Ring0；

主机上的用户态进程运行于根模式下的Ring3；

虚拟机中的Kernel运行于非根模式下的Ring0；

虚拟机中的用户态进程运行于非根模式下的Ring3；

Qemu-kvm是KVM官方提供并维护的改进后的Qemu，针对KVM解决方案，将标准Qemu做了针对性的改造，使其具有更好的性能，并与KVM进行了完美的融合；

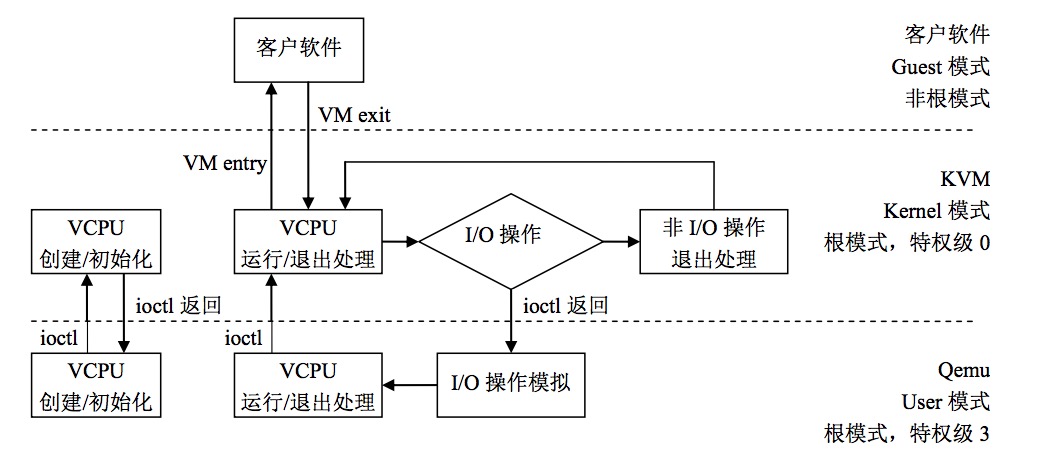
一个VM(虚拟机)就是一个传统的Linux进程，VM运行于Qemu-KVM进程的地址空间中；

VMM向上层提供/dev/kvm接口，/dev/kvm是一个标准的字符设备，通过ioctl接口控制；Qemu-kvm通过调用/dev/kvm设备的ioctl接口，对虚拟机进行相关控制，比如创建虚拟机、创建VCPU、运行虚拟机等；**为提升KVM虚拟机中的IO性能，KVM还提供了Virtio驱动，相当于Xen环境中的半虚拟化驱动**。

#### KVM工作原理

用户模式的qume利用libkvm 通过ioctl进入内核模式，kvm 模块为虚拟机创建虚拟内存，虚拟cpu后执行VMLUACH指令进入客户模式。加载guest os并执行。如果guest os 发生外部中断，或者影子页表缺页之类的情况，会暂停guest os的执行，退出客户模式出行异常处理，之后重新进入客户模式，执行客户代码，如果发生iO事件或者信号队列中有信号到达，就会进入用户模式处理。

处理情况如下图



流程描述：

1.运行在用户态的Qemu-kvm通过ioctl系统调用操作/dev/kvm字符设备，创建VM和VCPU

2.内核KVM模块负责相关数据结构的创建即初始化，然后返回用户态

3. Qemu-kvm通过ioctl调用运行VCPU，即调度相应的VM运行

4.内核进行相关处理后，执行VMLAUNCH指令，通过VM-Entry进入Guest OS运行，Guest OS运行于非根模式下。

5. Guest OS执行相应的虚拟机代码，非敏感指令可直接在物理CPU上运行

6.当Guest OS中执行到敏感指令、发生外部中断、或Guest OS发生内部异常时，将产生VM-Exit，并将相关信息记录到VMCS结构中

7. VM-Exit使CPU退回到根模式下，由VMM读取VMCS结构判断VM-Exit的原因

8.如是IO操作或是其他外设指令，则返回到用户态Qemu-kvm(即根模式下的Ring3)，由Qemu-kvm完成对相关指令的模拟。

9.如果不是，则由VMM自行处理

10.处理完成后，重新VM-entry进入到Guest OS运行。

#### KVM内核模块

KVM主要由3个内核模块组成：

- Kvm.ko

- Kvm-intel.ko

- Kvm-amd.ko

Kvm.ko是KVM的核心公共模块，kvm-intel.ko和kvm-amd.ko分别是针对Intel和AMD平台架构的独立模块。在KVM核心公共模块中，包含了IOMMU、中断控制、KVM arch、设备管理等部分代码，这些代码构成了虚拟机管理的核心功能，从这些模块的大致信息，也可以看出KVM自身并没有实现一个完整的PC系统的虚拟化，而只是实现了最核心的CPU虚拟化、内存虚拟化和IO虚拟化等部分功能并向上层提供了相应的API，其余虚拟化和管理工作主要交给了Qemu-kvm负责。

#查看宿主机CPU是否在硬件上支持虚拟化扩展特性

cat /proc/cpuinfo | grep -E "(vmx|svm)"

从Linux内核版本2.6.20开始，kvm模块就包含在Linux内核中，只需加载此模块即可

#查看kvm模块

modprobe -l | grep kvm

#加载kvm模块（Intel VT）

modprobe kvm

modprobe kvm-intel

#注意：如果加载失败，说明服务器硬件不支持或BIOS中未开启虚拟化扩展

#### 安装

<https://opengers.github.io/virtualization/kvm-libvirt-qemu-1/>

<http://www.diting0x.com/20160702/kvmqemuintro/>

KVM的官方地址：http://www.linux-kvm.org/page/Main\_Page

KVM的Howto文档：http://www.linux-kvm.org/page/HOWTO

Kqemu源码地址：http://sourceforge.net/projects/kqemu/

Qemu下载地址：http://wiki.qemu.org/Main\_Page

总的来说，**下载最新KVM源代码**，主要有如下三种方式：<https://www.linux-kvm.org/page/Main_Page>

a. 下载KVM项目开发中的代码仓库kvm.git

git clone git://git.kernel.org/pub/scm/virt/kvm/kvm.git

b. 下载Linux内核的代码仓库linux.git

git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

c. 打包下载Linux内核的源代码

wget [http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.109.tar.gz](http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.1.tar.gz)

git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

**下载qemu-kvm**源代码qemu.git（包括了 kvm）: <https://www.qemu.org/>

git clone git://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git

git clone git://git.qemu.org/qemu.git

<https://git.qemu.org/?p=qemu.git;a=summary>

下面从git安装qemu最新版qemu官网的git地址:

<http://git.qemu-project.org/qemu.git>

#### 内核KVM

<https://www.cnblogs.com/arnoldlu/p/6527496.html>

**KVM内核文档阅读笔记**

KVM在内核中有丰富的文档，位置在Documentation/virtual/kvm/。

00-INDEX：整个目录的索引及介绍文档。

api.txt：KVM用户空间API，所谓的API主要是通过ioctl来实现的。

cpuid.txt：KVM的cpuid相关API。

devices/：各种平台相关设备API。

hypercalls.txt：KVM的hypercall介绍，介绍了X86和S390的支持的hypercall详细信息。

locking.txt：介绍了KVM用到的锁、互斥量。

mmu.txt：介绍了Guest X86 MMU功能。

msr.txt：X86架构下的MSR用途。

nested-vmx.txt：使用X86特有的VMX来更简洁高效的运行Guest OS。

ppc-pv.txt和说90-diag.txt：针对PPC和S390架构的特殊用途，忽略。

review-checklist.txt：KVM相关patch的review检查列表。

timekeeping.txt：X86架构下时间虚拟化设备。

**下载libvirt**

git项目的位置http://libvirt.org/git/?p=libvirt.git

新建一个文件夹,存放git的文件

<https://libvirt.org/index.html>

git clone git://libvirt.org/libvirt.git

编译安装

这部分是参考http://libvirt.org/compiling.html文章

###个人不推荐这个方法，见下面

$ ./autogen.sh --prefix=$HOME/usr

$ make

$ sudo make install

<https://wiki.qemu.org/Hosts/Linux>

 wget https://libvirt.org/sources/libvirt-4.4.0.tar.xz

CentOS上安装Kvm步骤

<https://www.jianshu.com/p/03d3afff3b5f>

kvm安装包

libvirt-client # libvirt的客户端,最重要的功能之一就是在宿主机关机时可以通知虚拟机也关机

gpxe-roms-qemu # 虚拟机 iPXE 的启动固件,支持虚拟机从网络启动

libvirt-python # libvirt 为 python 提供的 API

python-virtinst # 一套 Python 的虚拟机安装工具

qemu-kvm # KVM 在用户空间运行的程序

libvirt # 用于管理虚拟机,它提供了一套虚拟机操作 API

virt-manager # 基于 libvirt 的图像化虚拟机管理软件

virt-viewer # 显示虚拟机的控制台 Console

virt-top # 类似于 top 命令,查看虚拟机的资源使用情况

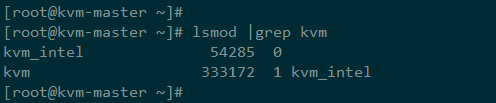
virt-what # 在虚拟机内部执行,查看虚拟机运行的虚拟化平台

virt-install #基于libvirt服务的虚拟机创建命令

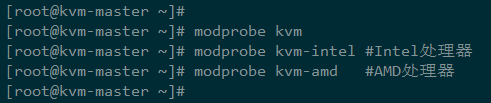
qemu-img # 用于操作虚拟机硬盘镜像的创建、查看和格式转化

bridge-utils #创建和管理桥接设备的工具（安装上述包会依赖此包。自动安装上）

模块加载



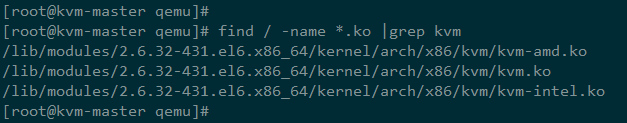
**如果没有检测到模块，则执行（一般用rpm安装kvm包，系统会在启动时自动加载模块。）：**



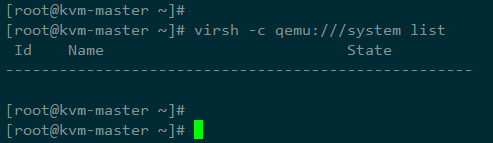
kvm：该模块实现虚拟化核心基础架构，与硬件平台无关

kvm\_intel、kvm\_amd：该模块主要涉及到硬件虚拟化

**模块文件在系统该位置：**



**校验kvm是否安装成功，如果下面为空，则表示没问题**



单元测试代码网址：

<https://git.kernel.org/cgit/virt/kvm/kvm-unit-tests.git/>

自动化测试源码网址：

<https://github.com/autotest/virt-test.git>

#### KVM API

**关于KVM API定义的文档(api.txt)**

**1.概述**

KVM API是一系列用来控制VM各方面的ioctl。它包括三个方面：

**System ioctls**：这些ioctl用来查询或者设置影响整个KVM子系统的属性。并且有一个system ioctl用来创建VM。

**VM ioctls**：这些ioctl用来查询或者设置影响整个VM的属性，其中一个VM ioctl用来创建vcpu。这些VM ioctl只能在用来创建此VM的进程中使用。

**vcpu ioctls**：这些ioctl用来查询或者设置能够控制单个vcpu的属性。这些vcpu ioctl只能在创建此vcpu的线程中使用。

从上面描述，可以清晰看出三者的层级关系：System ioctls(整个KVM子系统) > VM ioctls(单个VM实体) > vcpu ioctls(单个vcpu)。

**2.文件描述符**

KVM API是围绕文件描述符展开的。从打开/dev/kvm开始，获得操作整个KVM子系统的句柄，这个句柄用来执行System ioctls。基于此System句柄执行KVM\_CREATE\_VM可以创建一个VM的文件句柄，VM句柄用来执行VM ioctls。基于VM句柄的KVM\_CREATE\_VCPU用来创建一个vcpu句柄。vcpu句柄执行系列vcpu ioctls用来控制vcpu。

常见的文件句柄的特性在KVM并不适用，比如fork操作等。KVM里面只支持一个VM一个进程，一个vcpu一个线程。

**3.API描述**

每个API，即ioctl都包含一些信息，比如能力、所属架构、类型(System、VM、vcpu)、参数和返回值。这些ioctl非常多，并且庞杂，根据类型分为3类。有一些特殊架构专有的ioctl活在Architecture中列出，如果通用则是all。

下面重点分析Architecture为all或者x86，Capability为basic类型的ioctl。

**System ioctls**

|  |  |
| --- | --- |
| API值 | 说明 |
| KVM\_GET\_API\_VERSION | 目前情况下返回值，只有12。如果返回12，表示所有能力为basic的ioctl都能否使用。其他值是不被允许的。 |
| KVM\_CREATE\_VM | 创建一个VM，返回的句柄可以用来控制创建的VM。但此时VM还没有vcpu和memory。 |
| KVM\_GET\_MSR\_INDEX\_LIST | 返回Guest支持的MSRs |
| KVM\_CHECK\_EXTENSION |  |
| KVM\_GET\_VCPU\_MMAP\_SIZE | 返回运行vcpu的KVM\_RUN使用的共享Memory Region大小。 |
|  |  |

**VM ioctls**

|  |  |
| --- | --- |
| API值 | 说明 |
| KVM\_CAP\_CHECK\_EXTENSION\_VM |  |
| KVM\_CREATE\_VCPU | 在VM里添加一个vcpu，但是总数不会超过max\_cpus。vcpu的id在[0, max\_vcpu\_id)之间。 |
| KVM\_GET\_DIRTY\_LOG | 返回脏内存页的位图 |
| KVM\_SET\_MEMORY\_ALIAS |  |
| KVM\_CREATE\_IRQCHIP | 创建一个虚拟机的APIC，并且随后创建的vCPU都将连接到APIC |
| KVM\_GET\_IRQCHIP | 读取APIC的中断标志 |
| KVM\_SET\_IRQCHIP | 写入 |
| KVM\_GET\_CLOCK |  |
| KVM\_SET\_CLOCK |  |
| KVM\_GET\_VCPU\_EVENTS |  |
| KVM\_SET\_VCPU\_EVENTS |  |
| KVM\_SET\_USER\_MEMORY\_REGION |  |
| KVM\_CAP\_ENABLE\_CAP\_VM | 不是所有的能力都被默认打开，可以使用此ioctl来扩展。 |
|  |  |

**vcpu ioctls**

|  |  |
| --- | --- |
| API值 | 说明 |
| KVM\_RUN | 用于运行一个Guest的vcpu。 |
| KVM\_GET\_REGS | 返回vcpu的通用寄存器值，通过struct kvm\_regs结构体返回，根据不同架构有所不同。 |
| KVM\_SET\_REGS | 设置vcpu的通用寄存器。 |
| KVM\_GET\_SREGS | 读取不同架构vcpu的特殊寄存器。 |
| KVM\_SET\_SREGS | 设置不同架构vcpu的特殊寄存器。 |
| KVM\_TRANSLATE |  |
| KVM\_INTERRUPT |  |
| KVM\_DEBUG\_GUEST |  |
| KVM\_GET\_MSRS |  |
| KVM\_SET\_MSRS |  |
| KVM\_SET\_CPUID |  |
| KVM\_SET\_SIGNAL\_MASK |  |
| KVM\_GET\_FPU | 获取vcpu的FPU状态。 |
| KVM\_SET\_FPU | 设置vcpu的FPU状态。 |
| KVM\_ENABLE\_CAP | 不是所有的能力都被默认打开，可以使用此ioctl来扩展。 |
|  |  |

下面一段代码很好的诠释了KVM->VM->vcpu之间的关系：

[image](http://images2015.cnblogs.com/blog/1083701/201703/1083701-20170310140906076-2047849915.png)

|  |
| --- |
| int ret, kvmfd = -1, vmfd = -1, cpufd = -1;  kvmfd = qemu\_open("/dev/kvm", O\_RDWR); if (kvmfd < 0) {     goto err; } vmfd = ioctl(kvmfd, KVM\_CREATE\_VM, 0); if (vmfd < 0) {     goto err; } cpufd = ioctl(vmfd, KVM\_CREATE\_VCPU, 0); if (cpufd < 0) {     goto err; } |

#### 主要数据结构

kvm结构体

kvm\_vcpu结构体

kvm\_x86\_ops结构体

kvm api主要结构体

**4.kvm\_run结构体**

**KVM Hypercall(hypercalls.txt)**

这里重点关注x86架构下的Hypercall。

KVM hypercall最多支持四个参数，通过rbx、rcx、rdx、rsi。Hypercall调用号通过rax传递，返回时rax存放返回值。

一般情况下，其他寄存器不参与Hypercall。

适用于X86架构的Hypercall有三个：KVM\_HC\_VAPIC\_POLL\_IRQ、KVM\_HC\_MMU\_OP、KVM\_HC\_KICK\_CPU。

Hypercall的定义在include/uapi/linux/kvm\_para.h中：

|  |
| --- |
| #define KVM\_HC\_VAPIC\_POLL\_IRQ        1 #define KVM\_HC\_MMU\_OP            2 #define KVM\_HC\_FEATURES            3 #define KVM\_HC\_PPC\_MAP\_MAGIC\_PAGE    4 #define KVM\_HC\_KICK\_CPU            5 #define KVM\_HC\_MIPS\_GET\_CLOCK\_FREQ    6 #define KVM\_HC\_MIPS\_EXIT\_VM        7 #define KVM\_HC\_MIPS\_CONSOLE\_OUTPUT    8 |

KVM\_HC\_VAPIC\_POLL\_IRQ：Host检查关起的中断。

KVM\_HC\_MMU\_OP：支持MMU操作，比如PTE、flushing TLB、release PT。

KVM\_HC\_KICK\_CPU：唤醒HLT状态下的vcpu。如果Guest内核模式下一个vcpu等待时间超时而执行HLT指令，另一个同Guest下vcpu可以通过触发KVM\_HC\_KICK\_CPU来唤醒。

这些Hypercall都在kvm\_emulate\_hypercall中处理：

|  |
| --- |
| int kvm\_emulate\_hypercall(struct kvm\_vcpu \*vcpu) {     unsigned long nr, a0, a1, a2, a3, ret;     int op\_64\_bit, r = 1;      kvm\_x86\_ops->skip\_emulated\_instruction(vcpu);      if (kvm\_hv\_hypercall\_enabled(vcpu->kvm))         return kvm\_hv\_hypercall(vcpu);      nr = kvm\_register\_read(vcpu, VCPU\_REGS\_RAX);  Hypercall调用号     a0 = kvm\_register\_read(vcpu, VCPU\_REGS\_RBX);  依次是四个参数     a1 = kvm\_register\_read(vcpu, VCPU\_REGS\_RCX);     a2 = kvm\_register\_read(vcpu, VCPU\_REGS\_RDX);     a3 = kvm\_register\_read(vcpu, VCPU\_REGS\_RSI);  …      switch (nr) {     case KVM\_HC\_VAPIC\_POLL\_IRQ:         ret = 0;         break;     case KVM\_HC\_KICK\_CPU:         kvm\_pv\_kick\_cpu\_op(vcpu->kvm, a0, a1);  真正用到的也就是唤醒vcpu，a1表示vcpu的apicid。         ret = 0;         break;     default:         ret = -KVM\_ENOSYS;         break;     } out:     if (!op\_64\_bit)         ret = (u32)ret;     kvm\_register\_write(vcpu, VCPU\_REGS\_RAX, ret);  返回值放在RAX中。     ++vcpu->stat.hypercalls;     return r; } |

struct kvm\_memslots \*memslots; #KVM虚拟机所分配到的内存slot，以数组形式存储这些slot的地址信息。

struct kvm\_vcpu \*vcpus[KVM\_MAX\_VCPUS]; #KVM虚拟机中包含的vCPU结构体，一个虚拟CPU对应一个vCPU结构体。

struct kvm\_io\_bus \*buses[KVM\_NR\_BUSES]; #KVM虚拟机中的I/O总线，一条总线对应一个kvm\_io\_bus结构体，如ISA总线、PCI总线。

struct kvm\_vm\_stat stat; #KVM虚拟机中的页表、MMU等运行时状态信息。

struct kvm\_arch arch; #KVM的软件arch方面所需要的一些参数

struct kvm\_run \*run; #vCPU的运行时参数，其中保存了寄存器信息、内存信息、虚拟机状态等各种动态信息。

struct kvm\_vcpu\_arch arch; #存储有KVM虚拟机的运行时参数，如定时器、中断、内存槽等方面的信息。

**支持Guest的MMU功能(mmu.txt)**

X86 KVM的shadow MMU功能提供一个标准的MMU功能给Guest，将Guest的物理地址转换成Host物理地址。

关于Nested VMX是一种嵌套式虚拟功能，能够使一台虚拟机具有物理机CPU特性，支持VMX/SVM硬件虚拟化。参考《[Nested VMX](https://www.kernel.org/doc/Documentation/virtual/kvm/nested-vmx.txt)》。这样虚拟机可以使自己成为一个Hypervisors，并在其上安装虚拟机。

所谓Nested就是运行在Guest上的嵌套Guest，Guest作为Hypervisor。

|  |  |
| --- | --- |
| 术语 | 解释 |
| PFN | Host page frame number |
| HPA | Host physical address |
| HVA | Host virtual address |
| GFN | Guest page frame number |
| GPA | Guest physical address |
| GVA | Guest virtual address |
| NGPA | Nested guest physical address |
| NGVA | Nested guest virtual address |
| PTE | Page table entry |
| GPTE | Guest page table entry |
| SPTE | Shadow page table entry |
| TDP | Two dimensional paging |

此处KVM MMU功能主要工作就是配置处理器的MMU以达到将Guest地址转变到Host。有下面三种不同的转变需求：

-当Guest关闭分页功能时，将Guest物理地址转变成Host物理地址。GPA->HPA

-当Guesst使能分页功能时，将Guest虚拟地址，转变成Guest物理地址，进而Host物理地址。GVA->GPA->HPA

-当Guest又虚拟化一个嵌套Guest时，将嵌套的Guest虚拟地址转变成嵌套的物理地址，进而Guest物理地址，最后是Host物理地址。 NGVA->NGPA->GPA-HPA

GPA是运行KVM进程的用户地址空间的一部分。用户空间定义了Guest地址到用户空间地址的转变(GPA->HVA)。两个不同的GPA可以映射到一个HVA，但反之不成立。

#### 主要流程

初始化流程

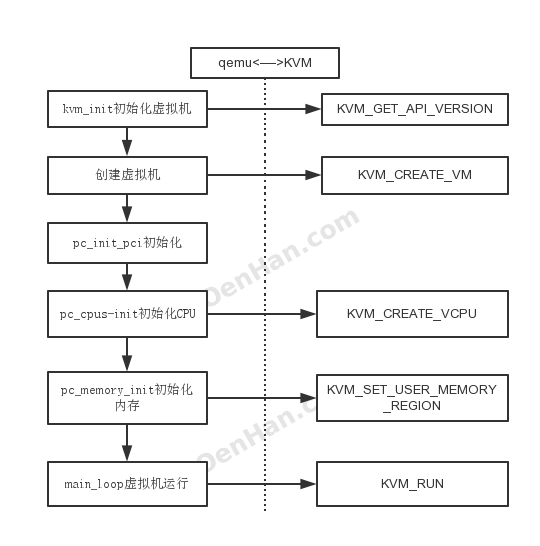
虚拟机创建

vCPU的创建

vCPU的运行

kvm虚拟化由用户态程序Qemu和内核态驱动kvm配合完成，qemu负责HOST用户态层面进程管理，IO处理等，KVM负责把qemu的部分指令在硬件上直接实现，从虚拟机的创建和运行上看，qemu的代码占了流程上的主要部分

QEMU和KVM的配合流程如下：

[](http://www.oenhan.com/wp-content/uploads/2014/12/qemu_create_kvm_vm.png)

#### vCPU调度

Guest和Host之间的切换

X86体系结构CPU虚拟化技术的称为Intel VT-x 技术，引入了VMX，提供了两种处理器的工作环境。 VMCS结构实现两种环境之间的切换。 VM Entry使虚拟机进去guest模式，VM Exit使虚拟机退出guest模式。

VMM调度guest执行时，qemu通过ioctl系统调用进入内核模式，在KVM Driver中获得当前物理CPU的引用。之后将guest状态从VMCS中读出， 并装入物理CPU中。执行 VMLAUCH 指令使得物理处理器进入非根操作环境，运行guest OS代码。

当guest OS执行一些特权指令或者外部事件时， 比如I/O访问，对控制寄存器的操作，MSR的读写等， 都会导致物理CPU发生VMExit， 停止运行Guest OS，将Guest OS保存到VMCS中，Host状态装入物理处理器中，处理器进入根操作环境，KVM取得控制权，通过读取VMCS中 VM\_EXIT\_REASON字段得到引起VM Exit的原因。 从而调用kvm\_exit\_handler处理函数。 如果由于I/O获得信号到达，则退出到userspace模式的 Qemu处理。处理完毕后，重新进入guest模式运行虚拟CPU。

#### KVM内存管理

KVM继承了Linux系统管理内存的诸多特性，比如，分配给虚拟使用的内存可以被交换至交换空间、能够使用大内存页以实现更好的性能，以及对NUMA的支持能够让虚拟机高效访问更大的内存空间等。

KVM基于Intel的EPT（ExtendedPage Table）或AMD的RVI（Rapid Virtualization Indexing）技术可以支持更新的内存虚拟功能，这可以降低CPU的占用率，并提供较好的吞吐量。

此外，KVM还借助于KSM（Kernel Same-pageMerging）这个内核特性实现了内存页面共享。KSM通过扫描每个虚拟机的内存查找各虚拟机间相同的内存页，并将这些内存页合并为一个被各相关虚拟机共享的单独页面。在某虚拟机试图修改此页面中的数据时，KSM会重新为其提供一个新的页面副本。实践中，运行于同一台物理主机上的具有相同GuestOS的虚拟机之间出现相同内存页面的概率是很的，比如共享库、内核或其它内存对象等都有可能表现为相同的内存页，因此，KSM技术可以降低内存占用进而提高整体性能。

OS对于物理内存主要有两点认识：1.物理地址从0开始；2.内存地址是连续的。VMM接管了所有内存，但guest OS的对内存的使用就存在这两点冲突了，除此之外，一个guest对内存的操作很有可能影响到另外一个guest乃至host的运行。VMM的内存虚拟化就要解决这些问题。

在OS代码中，应用也是占用所有的逻辑地址，同时不影响其他应用的关键点在于有线性地址这个中间层；解决方法则是添加了一个中间层：guest物理地址空间；guest看到是从0开始的guest物理地址空间（类比从0开始的线性地址），而且是连续的，虽然有些地址没有映射；同时guest物理地址映射到不同的host逻辑地址，如此保证了VM之间的安全性要求。

这样MEM虚拟化就是GVA->GPA->HPA的寻址过程，传统软件方法有影子页表，硬件虚拟化提供了EPT支持。

可能GVA->GPA->HVA->HPA更全面一点。

GVA: Guest Virtual Address

GPA: Guest Physical Address

HVA: Host Virtual Address

HPA: Host Physical Address

#### KVM设备管理

#### KVM功能列表

KVM所支持的功能包括：

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）

支持对称多处理（Symmetric Multi-Processing，缩写为 SMP ）

支持实时迁移（Live Migration）

支持PCI设备直接分配和单根I/O虚拟化（SR-IOV）

支持内核同页合并（KSM ）

支持NUMA（Non-Uniform Memory Access，非一致存储访问结构 ）

#### KVM工具集

libvirt：操作和管理KVM虚机的虚拟化 API，使用 C 语言编写，可以由 Python,Ruby, Perl, PHP, Java 等语言调用。可以操作包括 KVM，vmware，XEN，Hyper-v, LXC 等 Hypervisor。

Virsh：基于libvirt的命令行工具 （CLI）

Virt-Manager：基于libvirt的GUI工具

virt-v2v：虚机格式迁移工具

virt-\* 工具：包括 Virt-install（创建KVM虚机的命令行工具），Virt-viewer（连接到虚机屏幕的工具），Virt-clone（虚机克隆工具），virt-top 等

sVirt：安全工具

#### KVM命令集

想使用virsh命令必须开启acpid服务

yum -y install acpid

/etc/init.d/acpid start

chkconfig acpid on

虚拟机重要的两个文件目录

/etc/libvirt/qemu/ //虚拟机配置文件目录

/var/lib/libvirt/images/ //虚拟机磁盘文件目录

修改虚拟机的配置

vim /etc/libvirt/qemu/benet.xml //修改虚拟机配置信息（用来修改系统内存大小、磁盘文件等信息）

virsh edit benet //通过virsh命令修改虚拟机配置信息

基本命令

1 virsh -h //查看命令帮助

2 virsh list //查看正在运行的虚拟机

3 virsh list --all //查看所有虚拟机

4 virsh create /etc/libvirt/qemu/benet.xml //通过配置文件启动虚拟机系统实例

5 virsh start benet //启动，需要确认acpid服务安装并运行

6 virsh shutdown benet(主机名) //关机，需要确认acpid服务安装并运行

7 virsh destroy benet(主机名) //强制关机

8 virsh autostart benet(主机名) //随宿主机自动启动，此命令将创建/etc/libvirt/qemu/autostart/目录

9 virsh suspend benet(主机名) //挂起

10 virsh resume benet(主机名) //恢复

11 virsh dumpxml benet > /etc/libvirt/qemu/benet2.xml //导出虚拟机配置

12 virsh undefine benet //删除虚拟机，会发现配置文件被清除，但是磁盘文件不会被删除

13 mv /etc/libvirt/qemu/benet2.xml/ /etc/libvirt/qemu/benet.xml //把备份的配置文件改回去，以便重新定义虚拟机

14 virsh define benet.xml //重新定义虚拟机（因为磁盘文件没有被删除）

15 virsh edit CentOS-7-x86\_64 //编辑虚拟机

16 ps -aux |grep kvm //查看正在运行的虚拟机：

17 kvm是以进程的方式运行的。也可以kill -9 杀掉这个虚拟机

**克隆与快照**

1 [root@localhost ~]# virt-clone -o benet -n benet2 -f /var/lib/libvirt/images/benet2.img //虚拟机克隆

2 [root@localhost ~]# virsh snapshot-create benet //创建新快照

3 Domain snapshot 1382572463 created

4 [root@localhost ~]# virsh snapshot-delete benet 1382572463 //删除快照

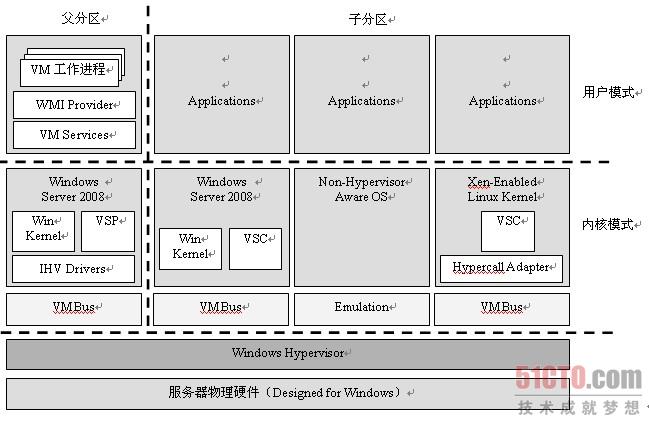
5 Domain snapshot 1382572463 deleted

6 [root@localhost ~]# virsh snapshot-revert benet 1382572463 //恢复虚拟机状态

7 [root@localhost ~]# virsh snapshot-current benet //查看快照版本号

8 [root@localhost ~]# virsh snapshot-list benet //查看快照信息

### Hyper-V



Hyper-V采用了VSP/VSC架构，其中VSP是虚拟化服务提供方，运行在宿主操作系统之上，而VSC是虚拟化服务客户端，运行在虚拟机操作系统之上，两者之间将通过VMBus组件进行连接。位于子分区的应用程序发出I/O操作请求后，将通过本地虚拟机操作系统平台的I/O堆栈，调用虚拟的设备驱动（VSC）。VSC接收到I/O请求后，通过VMBus将请求转发到父分区的对应VSP，VSP对操作请求进行验证和处理后，最终通过本地的I/O堆栈和硬件设备驱动访问物理设备完成操作。

Hyper-V可支持4 CPU的Windows Server 2008虚拟机和2 CPU的Windows Server 2003虚拟机。当然该特性并非Hyper-V所独有。多处理器虚拟机可支持更高的工作负载，但管理更多的处理器也将带来额外的处理负荷，因而应该按需使用多处理器虚拟机。

### VMware

## 虚拟化实现技术

CPU的虚拟化技术可以单CPU模拟多CPU并行虚拟机(Virtual machine)可像真实机器一样运行程序的软件。

虚拟内存，将不相邻的物理存储区虚拟成统一连续的内存地址；

存储虚拟化，将实体存储空间分隔成不同的逻辑存储空间；

网络虚拟化，将不同网络的硬件和软件资源结合成虚拟整体；

I/O虚拟化，如USB接口上的虚拟串口；

网卡虚拟化，如一张物理网卡上虚拟N个网络接口，AN，将局域网设备从逻辑上划分成一个个网段；

X86操作系统是设计在直接运行在裸硬件设备上的，因此它们自动认为它们完全占有计算机硬件。x86 架构提供四个特权级别给操作系统和应用程序来访问硬件。 Ring是指CPU的运行级别，Ring0是最高级别，Ring1次之，Ring2更次之…… 就 Linux+x86 来说，操作系统（内核）需要直接访问硬件和内存，因此它的代码需要运行在最高运行级别Ring0上，这样它可以使用特权指令，控制中断、修改页表、访问设备等等。

应用程序的代码运行在最低运行级别上Ring3上，不能做受控操作。如果要做，比如要访问磁盘，写文件，那就要通过执行系统调用（函数），执行系统调用的时候，CPU的运行级别会发生从ring3到ring0的切换，并跳转到系统调用对应的内核代码位置执行，这样内核就为你完成了设备访问，完成之后再从ring0返回ring3。这个过程也称作用户态和内核态的切换。



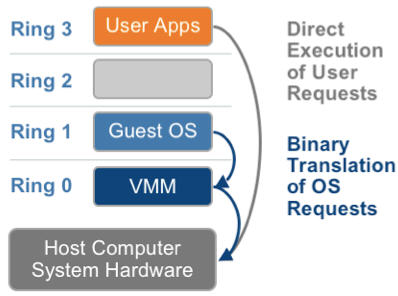
那么，虚拟化在这里就遇到了一个难题，因为宿主操作系统是工作在 ring0 的，客户操作系统就不能也在 ring0 了，但是它不知道这一点，以前执行什么指令，现在还是执行什么指令，但是没有执行权限是会出错的。所以这时候虚拟机管理程序（VMM）需要避免这件事情发生。 虚机怎么通过 VMM 实现 Guest CPU 对硬件的访问，根据其原理不同有三种实现技术：

1. 全虚拟化

2. 半虚拟化

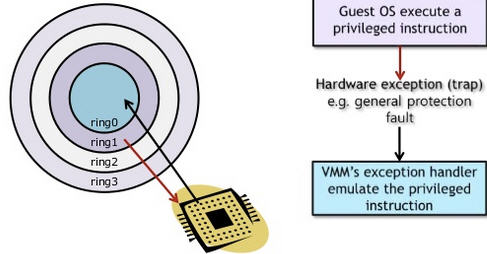
3. 硬件辅助的虚拟化

**基于二进制翻译的全虚拟化（Full Virtualization with Binary Translation）**



用户操作系统运行在 Ring 1，它在执行特权指令时，会触发异常（CPU的机制，没权限的指令会触发异常），然后 VMM 捕获这个异常，在异常里面做翻译，模拟，最后返回到客户操作系统内，客户操作系统认为自己的特权指令工作正常，继续运行。但是这个性能损耗，就非常的大，简单的一条指令，执行完，了事，现在却要通过复杂的异常处理过程。

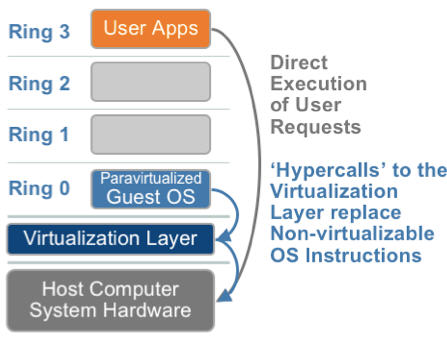
异常 “捕获（trap）-翻译（handle）-模拟（emulate）” 过程：



**超虚拟化（或者半虚拟化/操作系统辅助虚拟化 Paravirtualization）**

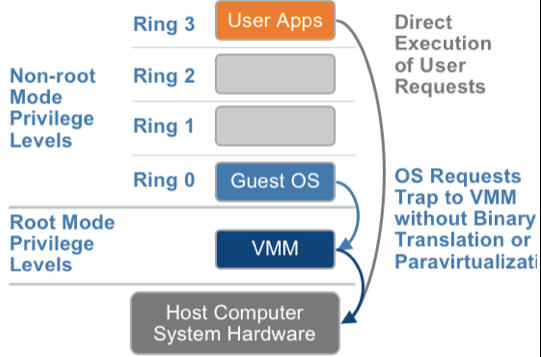
半虚拟化的思想就是，修改操作系统内核，替换掉不能虚拟化的指令，通过超级调用（hypercall）直接和底层的虚拟化层hypervisor来通讯，hypervisor 同时也提供了超级调用接口来满足其他关键内核操作，比如内存管理、中断和时间保持。

这种做法省去了全虚拟化中的捕获和模拟，大大提高了效率。所以像XEN这种半虚拟化技术，客户机操作系统都是有一个专门的定制内核版本，和x86、mips、arm这些内核版本等价。这样以来，就不会有捕获异常、翻译、模拟的过程了，性能损耗非常低。这就是XEN这种半虚拟化架构的优势。这也是为什么XEN只支持虚拟化Linux，无法虚拟化windows原因，微软不改代码啊。



**硬件辅助的全虚拟化**

2005年后，CPU厂商Intel 和 AMD 开始支持虚拟化了。 Intel 引入了 Intel-VT （Virtualization Technology）技术。 这种 CPU，有 VMX root operation 和 VMX non-root operation两种模式，两种模式都支持Ring 0 ~ Ring 3 共 4 个运行级别。这样，VMM 可以运行在 VMX root operation模式下，客户 OS 运行在VMX non-root operation模式下。



而且两种操作模式可以互相转换。运行在 VMX root operation 模式下的 VMM 通过显式调用 VMLAUNCH 或 VMRESUME 指令切换到 VMX non-root operation 模式，硬件自动加载 Guest OS 的上下文，于是 Guest OS 获得运行，这种转换称为 VM entry。Guest OS 运行过程中遇到需要 VMM 处理的事件，例如外部中断或缺页异常，或者主动调用 VMCALL 指令调用 VMM 的服务的时候（与系统调用类似），硬件自动挂起 Guest OS，切换到 VMX root operation 模式，恢复 VMM 的运行，这种转换称为 VM exit。VMX root operation 模式下软件的行为与在没有 VT-x 技术的处理器上的行为基本一致；而VMX non-root operation 模式则有很大不同，最主要的区别是此时运行某些指令或遇到某些事件时，发生 VM exit。

也就说，硬件这层就做了些区分，这样全虚拟化下，那些靠“捕获异常-翻译-模拟”的实现就不需要了。而且CPU厂商，支持虚拟化的力度越来越大，靠硬件辅助的全虚拟化技术的性能逐渐逼近半虚拟化，再加上全虚拟化不需要修改客户操作系统这一优势，全虚拟化技术应该是未来的发展趋势。

|  |  |  |  |
| --- | --- | --- | --- |
|  | **利用二进制翻译的全虚拟化** | **硬件辅助虚拟化** | **操作系统协助/半虚拟化** |
| **实现技术** | BT和直接执行 | 遇到特权指令转到root模式执行 | Hypercall |
| **客户操作系统修改/兼容性** | 无需修改客户操作系统，最佳兼容性 | 无需修改客户操作系统，最佳兼容性 | 客户操作系统需要修改来支持hypercall，因此它不能运行在物理硬件本身或其他的hypervisor上，兼容性差，不支持Windows |
| **性能** | 差 | 全虚拟化下，CPU需要在两种模式之间切换，带来性能开销；但是，其性能在逐渐逼近半虚拟化。 | 好。半虚拟化下CPU性能开销几乎为0，虚机的性能接近于物理机。 |
| **应用厂商** | VMware Workstation/QEMU/Virtual PC | VMware ESXi/Microsoft Hyper-V/Xen 3.0/KVM | Xen |

### 处理器虚拟化

#### Intel VT

#### AMD-V

**一、软件虚拟化和硬件虚拟化**

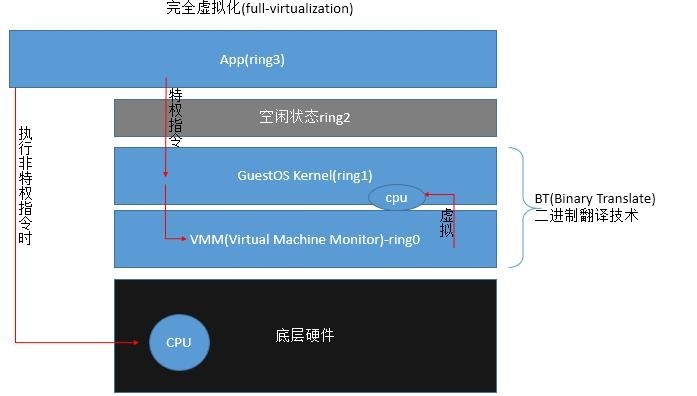
**1）虚拟化—软件方案**

纯软件虚拟化， 顾名思义， 就是用纯软件的方法在现有的物理平台上（往往并不支持硬件虚拟化） 实现对物理平台访问的截获和模拟。常见的软件虚拟机例如QEMU，它是通过纯软件来仿真X86平台处理器的取指、解码和执行，客户机的指令并不在物理平台上直接执行。由于所有的指令都是软件模拟的，因此性能往往比较差，但是可以在同一平台上模拟不同架构平台的虚拟机。

VMWare的软件虚拟化则使用了**动态二进制翻译（BT）**的技术，与QEMU这种模拟的方式不同，BT是一种加速虚拟化的方案之一，另一种常见的虚拟化加速方案就是**硬件辅助虚拟化技术**。BT就是在虚拟机监控机可控制的范围内，允许客户机的指令在物理平台上直接运行。但是，客户机指令在运行前会被虚拟机监控机扫描，其中突破虚拟机监控机限制的指令会被动态替换为可以在物理平台上直接运行的安全指令，或者替换为对虚拟机监控器的软件调用。这样做的好处是比纯软件模拟性能有大幅的提升（模拟其根本就是通过一个软件做出一个假的，可以是不存在的；而虚拟是把这个设备通过某种切割或其他方式虚拟出去提供一定程度的服务）， 但是也同时失去了跨平台虚拟化的能力。

有了BT技术后，Guest的用户空间运行在CPU ring 3上，而Guest的内核空间运行在了CPU ring 1上，Host的内核空间运行在CPU ring 0上。BT就监控在CPU ring 1上，随时将Guest内核的调用给转换为特权指令调用。当然CPU ring 1并没有被使用，BT这种技术让虚拟化性能得到了大大的提升。但是BT有一个大大的缺点就是无法跨平台，使用QEMU这种模拟器不管底层硬件是什么，能模拟各种CPU架构平台，如PowerPC、ARM等；但是BT却无法做到这点，BT强烈依赖底层架构，比如底层是X86的那么只能创建X86 CPU的虚拟机。

在纯软件虚拟化解决方案中，VMM在软件套件中的位置是传统意义上操作系统所处的位置，而操作系统的位置是传统意义上应用程序所处的位置， 这种转换必然会增加系统的复杂性。软件堆栈的复杂性增加意味着，这些环境难于管理，因而会加大确保系统可靠性和安全性的困难。

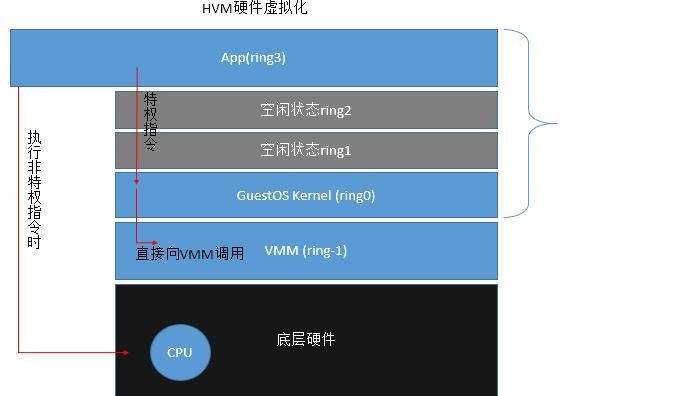


**2） 虚拟化—硬件方案**

**硬件辅助虚拟化（HVM）**，简而言之，就是物理平台本身提供了对特殊指令的截获和重定向的硬件支持，甚至，新的硬件会提供额外的资源来帮助软件实现对关键硬件资源的虚拟化，从而提升性能。可以理解为**CPU额外增加了一个ring -1环专门提供给虚拟机运行的**。以X86平台的虚拟化为例，支持虚拟技术的X86 CPU带有特别优化过的指令集来控制虚拟过程，通过这些指令集，VMM会很容易将客户机置于一种受限制的模式下运行，一旦客户机试图访问物理资源，硬件会暂停客户机的运行，将控制权交回给VMM处理。VMM还可以利用硬件的虚拟化增强机制，将客户机在受限模式下对一些特定资源的访问，完全由硬件重定向到VMM指定的虚拟资源，整个过程不需要暂停客户机的运行和VMM软件的参与。

由于虚拟化硬件可提供全新的架构，支持操作系统直接在上面运行，无需进行二进制转换，减少了相关的性能开销，极大简化了VMM 设计，进而使VMM能够按通用标准进行编写， 性能更加强大。

需要说明的是， 硬件虚拟化技术是一套解决方案。完整的情况需要CPU、主板芯片组、BIOS和软件的支持，例如VMM软件或者某些操作系统本身。即使只是CPU支持虚拟化技术，在配合VMM软件的情况下，也会比完全不支持虚拟化技术的系统有更好的性能。鉴于虚拟化的巨大需求和硬件虚拟化产品的广阔前景，Intel一直都在努力完善和加强自己的硬件虚拟化产品线。自2005年末，Intel便开始在其处理器产品线中推广应用Intel Virtualization Technology（IntelVT）虚拟化技术，发布了具有IntelVT虚拟化技术的一系列处理器产品，包括桌面的Pentium和Core系列，还有服务器的Xeon至强和Itanium安腾。Intel一直保持在每一代新的处理器架构中优化硬件虚拟化的性能和增加新的虚拟化技术。现在市面上，从桌面的Core i3/5/7，到服务器端的E3/5/7/9，几乎全部都支持Intel VT技术。可以说， 在不远的将来， Intel VT很可能会成为所有Intel处理器的标准配置。当然AMD的CPU也都支持虚拟化技术。



**总结**

硬件辅助虚拟化貌似比BT技术更好，如果BT技术能够让虚拟机性能达到物理机80%的性能的话，那么硬件辅助虚拟化（HVM）就能够让虚拟机性能达到物理机85%左右。当然这中间的转换还是需要的，只不过是由硬件直接完成了，仅此而已。

**二、全虚拟化和半虚拟化**

**Full-virtualization（全虚拟化)**

全虚拟化为客户机提供了完整的虚拟X86平台， 包括处理器、 内存和外设， 支持运行任何理论上可在真实物理平台上运行的操作系统， 为虚拟机的配置提供了最大程度的灵活性。不需要对客户机操作系统做任何修改即可正常运行任何非虚拟化环境中已存在基于X86平台的操作系统和软件，这也是全虚拟化无可比拟的优势。

在全虚拟化情况下，虚拟机并不知道自己运行在虚拟化环境下，是无感知的，安装使用时跟在物理机上没有什么区别。但是这种完全虚拟化中间需要软件做支撑的，需要软件去模拟提供所有的硬件资源，至少是这个CPU的特权指令需要用软件去模拟的，因为你要让各Guest并不知道自己运行在虚拟环境中，那么你就必须要提供一个带有特权指令的CPU。

在虚拟化环境中，通常虚拟跟模拟是两个概念，VMWare的动态二进制翻译技术（BT）是虚拟的，而QEMU软件技术是模拟的。最大的区别在于，模拟通过软件实现时需要模拟CPU ring 0-3，也就是需要转换CPU ring 0-3所有的指令，而虚拟只需要转换CPU ring 0特权指令即可。

当然不管上面说到的BT技术还是QEMU还是硬件辅助虚拟化技术都属于完全虚拟化技术，都是需要指令转换的，都是需要复杂的步骤才能完成的，如果我们能够精简这其中的步骤那么虚拟机的性能一定会有提升的。那么怎么精简呢？这就是下面说的半虚拟化技术。另外，在全虚拟化模式下：

CPU如果不支持硬件虚拟化技术：那么所有指令都是通过VMM虚拟的，通过VMM内的BT动态翻译技术把虚拟机要运行的特权指令转换为物理指令集，然后到CPU上运行。

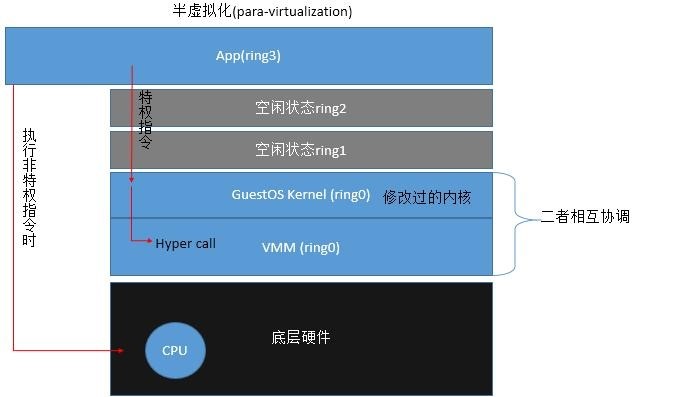
CPU如果支持硬件虚拟化技术：VMM运行ring -1，而GuestOS运行在ring 0。



**Para-virtualization（半虚拟化）**

软件虚拟化可以在缺乏硬件虚拟化支持的平台上完全通过VMM软件来实现对各个虚拟机的监控，以保证它们之间彼此独立和隔离。 但是付出的代价是软件复杂度的增加，和性能上的损失。减轻这种负担的一种方法就是，改动客户操作系统，使它知道自己运行在虚拟环境下，能够与虚拟机监控机协同工作。这种方法就叫半虚拟化（para-virtualization）。虚拟机内核明确知道自己是运行在虚拟化之上的，对于硬件资源的使用不再需要BT而是自己向VMM申请使用，如对于内存或CPU的使用是直接向VMM申请使用，直接调用而非翻译。就算对于I/O设备的使用它也可以通过Hyper Call（Hypervisor提供的系统调用）直接可以跟硬件打交道，减少了中间的翻译步骤自然性能就好了，据说这种半虚拟化方式能够让虚拟化达到物理机90%的性能。本质上，半虚拟化弱化了对虚拟机特殊指令的被动截获要求，将其转化成客户机操作系统的主动通知。但是，半虚拟化需要修改客户机操作系统的源代码来实现主动通知。

Xen是开源准虚拟化技术的一个例子，操作系统作为虚拟服务器在Xen Hypervisor上运行之前，它必须在内核层面进行某些改变。因此，Xen适用于BSD、Linux、Solaris及其他开源操作系统，但不适合对像Windows这些专有的操作系统进行虚拟化处理，因为它们不  
公开源代码，所以无法修改其内核。



**总结**

由于硬件辅助虚拟化的出现，使得完全虚拟化在性能上也得到了提升。并且相比半虚拟化而言，完全虚拟化使用上更加简化，虚拟过程对于Guest而言是透明的。所以完全虚拟化更加符合市场需求，比如后面说的KVM虚拟机。

### 中断虚拟化

### I/O虚拟化

从处理器的角度看，外设是通过一组I/O资源（端口I/O或者是MMIO）来进行访问的，所以设备的相关虚拟化被称为I/O虚拟化，如：

1）外存设备：硬盘、光盘、U盘。

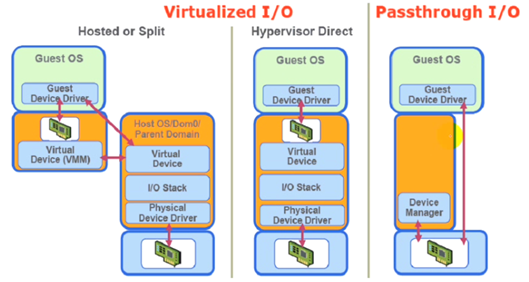
2）网络设备：网卡。

3）显示设备：VGA（显卡）。

4）键盘鼠标：PS/2、USB。

还有一些如串口设备、COM口等等设备统称IO设备，所谓IO虚拟化就是提供这些设备的支持，其思想就是VMM截获客户操作系统对设备的访问请求，然后通过软件的方式来模拟真实设备的效果。基于设备类型的多样化，I/O虚拟化的方式和特点纷繁复杂，我们挑一些常用IO设备说一说。

但一般IO虚拟化的方式有以下三种，如下图：



**第一种：模拟I/O设备**

完全使用软件来模拟，这是最简单但性能最低的方式，对于IO设备来说模拟和完全虚拟化没有太大意义上的区别。VMM给Guest OS模拟出一个IO设备以及设备驱动，Guest OS要想使用IO设备需要调内核然后通过驱动访问到VMM模拟的IO设备，然后到达VMM模拟设备区域。VMM模拟了这么多设备以及VMM之上运行了那么多主机，所以VMM也提供了一个I/O Stack（多个队列）用来调度这些IO设备请求到真正的物理IO设备之上。经过多个步骤才完成一次请求。

**举例：Qemu、VMware Workstation**

通过VMM模拟I/O设备（磁盘和网卡等）实现虚拟化。Guest OS所能看到就是一组统一的I/O设备。VMM截获Guest OS对I/O设备的访问请求，然后通过软件模拟真实的硬件。这种方式对Guest而言非常透明，无需考虑底层硬件的情况。

**第二种：半虚拟化**

半虚拟化比模拟性能要高，其通过系统调用直接使用I/O设备，跟CPU半虚拟化差不多，虚拟化明确知道自己使用的IO设备是虚拟出来的而非模拟。VMM给Guest OS提供了特定的驱动程序，在半虚拟化IO中我们也称为“**前端IO驱动**”；跟模拟I/O设备工作模式不同的是，Guest OS自己本身的IO设备不需要处理IO请求了，当Guest OS有IO请求时通过自身驱动直接发给VMM进行处理，而在VMM这部分的设备处理我们称之为“**后端IO驱动**”。

**举例：Xen、virtio**

通过前端（Front-End）/后端（Back-End）模拟实现虚拟化。Guest OS中的驱动程序为前端，VMM提供的与Guest通信的驱动程序为后端。前端驱动将Guest OS的请求通过与VMM间的特殊通信机制发送给VMM的后端驱动，后端驱动在处理完请求后再发送给物理驱动。

**第三种：I/O透传技术**

I/O透传技术（I/O through）比模拟和半虚拟化性能都好，几乎进阶于硬件设备，Guest OS直接使用物理I/O设备，操作起来比较麻烦。其思想就是提供多个物理I/O设备，如硬盘提供多块，网卡提供多个，然后规划好宿主机运行Guest OS的数量，通过协调VMM来达到每个Guest OS对应一个物理设备。另外，要想使用I/O透传技术，不光提供多个I/O设备还需要主板上的I/O桥提供支持透传功能才可以，一般Intel提供的这种技术叫VT-d，是一种基于北桥芯片的硬件辅助虚拟化技术，主要功能是由来提高I/O灵活性、可靠性和性能的。

为什么I/O透传还需要主板支持呢？每个虚拟机直接使用一个网卡不就可以了吗？主要是因为在我们传统的X86服务器架构上，所有的IO设备通常有一个共享或集中式的DMA（直接内存访问），DMA是一种加速IO设备访问的方式。由于是集中式的，所以在VMM上管理多块网卡时其实使用的还是同一个DMA，如果让第一个Guest OS直接使用了第一块网卡，第二个Guest OS直接使用第二块网卡，但使用的DMA还是同一个，而DMA是无法区分哪个Guest OS使用的是哪块网卡，这就变的麻烦了。而像Intel的VT-d就是用来处理这些问题的，以及处理各主机中断。

**举例：Intel VT-d**

**I/O硬件辅助虚拟化技术**

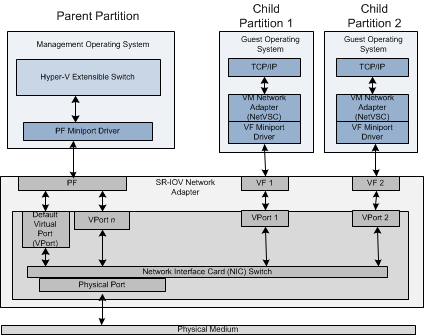
主要有Intel的VT-d、AMD的IOMMU和PCI-SIG的IOV这三种技术。前两种技术属于直接I/O技术，物理设备可以直接分配给虚拟机使用（而不需经过VMM），通过硬件芯片完成GPA到MA的翻译。IOV技术在其基础上，在硬件设备中增加了一个PCIe设备，用于呈现一个物理功能（PF）和多个虚拟功能（VF），从而使每个虚拟功能可单独分配给不同的客户机使用。

#### SR-IOV

SR-IOV（Single Root I/O Virtualization and Sharing Specification），用来解决虚拟最后一公里的问题，即多个虚机可以同时共享使用同一个PCI硬件。它需要专门支持SR-IOV的硬件网卡，它会在Hypervisor里注册成多个网卡（每个网卡都有独立的中断，收发队列，QoS等机制）,将虚拟网卡中的数据分类功能挪到了硬件SR-IOV网卡中实现。

SR-IOV（Single Root I/O Virtualization）是一个将PCIe共享给虚拟机的标准，通过为虚拟机提供独立的内存空间、中断、DMA流，来绕过VMM实现数据访问。SR-IOV基于两种PCIe functions：

* PF (Physical Function)： 包含完整的PCIe功能，包括SR-IOV的扩张能力，该功能用于SR-IOV的配置和管理。
* FV (Virtual Function)： 包含轻量级的PCIe功能。每一个VF有它自己独享的PCI配置区域，并且可能与其他VF共享着同一个物理资源

****

* [Intel SR-IOV Configuration Guide](http://www.intel.com/content/www/us/en/embedded/products/networking/xl710-sr-iov-config-guide-gbe-linux-brief.html)
* [OpenStack SR-IOV Passthrough for Networking](https://wiki.openstack.org/wiki/SR-IOV-Passthrough-For-Networking)
* [Redhat OpenStack SR-IOV Configure](https://access.redhat.com/documentation/zh-CN/Red_Hat_Enterprise_Linux_OpenStack_Platform/7/html/Networking_Guide/sec-sr-iov.html)
* [SDN Fundamentails for NFV, Openstack and Containers](http://www.slideshare.net/nyechiel/sdn-fundamentals-for-nfv-open-stack-and-containers-red-hat-summit-20161)
* [I/O设备直接分配和SRIOV](http://www.cnblogs.com/sammyliu/p/4548194.html)
* [Libvirt PCI passthrough of host network devices](http://wiki.libvirt.org/page/Networking#PCI_Passthrough_of_host_network_devices)
* [Story of Network Virtualization and its future in Software and Hardware](http://netdevconf.org/2.1/session.html?jain)

### 网络虚拟化

当前，主流的网络I/O虚拟化技术有三种：软件模拟、网卡直通和SR-IOV。这三种虚拟化技术在不同程度上实现了网络I/O设备的虚拟化功能。

其中，软件模拟是通过虚拟化Hypervisor层模拟虚拟网卡，实现与物理设备完全一样的接口，虚拟机操作系统无须修改就能直接驱动虚拟网卡，其最大的缺点是性能相对较差；

网卡直通支持虚拟机绕过Hypervisor层，直接访问物理I/O设备，具有最高的性能，但是，在同一时刻，物理I/O设备只能被一个虚拟机独享；

SR-IOV是Intel在2007年提出的解决虚拟化网络I/O的硬件技术方案，该技术不仅能够继承网卡直通的高性能优势，而且同时支持物理I/O设备的跨虚拟机共享，具有较好的应用前景。

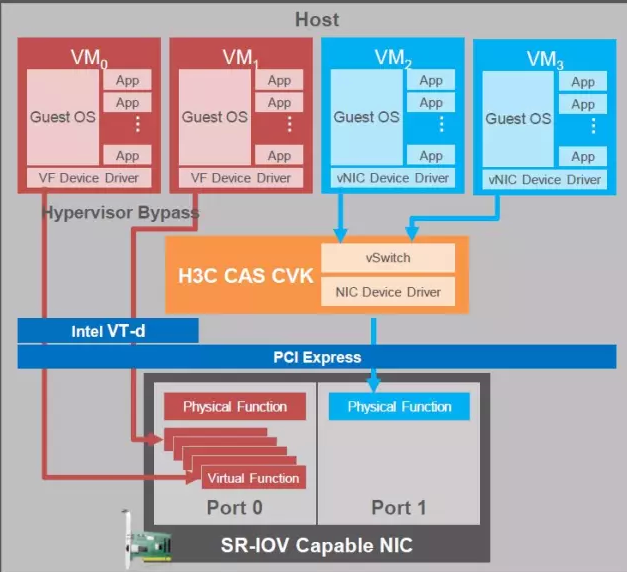
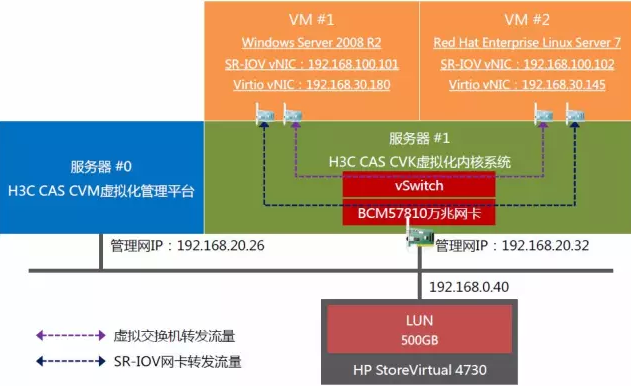
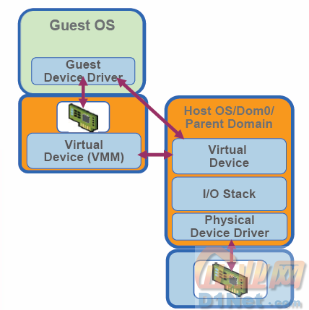
 

图2 硬件SR-IOV网卡逻辑组网图

**4个主流网络I/O模型分别是**

**1、Emulation**

原理：仿真(emulation)是一个完全通过软件程序来模拟硬件的技术。早期虚拟化都才采用这种方案来虚拟网络设备。常见仿真软件有QEMU、VMware WorkStation、VirtualBox。Emulation网络模型图如下：



不同虚拟化厂商的虚拟网卡产品都不尽相同。Microsoft Hyper-V Emulation类型网卡有：Intel/DEC 21140 100M网卡

优点：软件模拟不需要硬件支持，通过CPU计算来模拟，跟宿主机物理网卡隔离，没有平台要求。

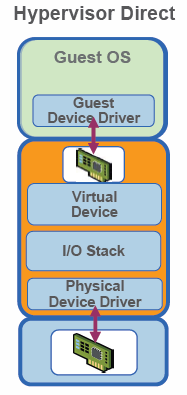
虚拟机操作系统不需要修改，模拟的都是常见网卡(比如：Intel E1000、RTL8139等)，主流操作系统都已经自带这些驱动，因此默认情况下虚拟机不需要再安装驱动。

缺点：CPU资源消耗大，尤其当虚拟机数量多的时候。网卡性能一般，由于是软件模拟，只能模拟常见的、功能比较简单的网卡。

虚拟机迁移支持：剥离了硬件要求，使用这类型可迁移性强。由于XEN和KVM都是使用qemu仿真，所以这类型虚拟机在XEN和KVM之间混合迁移实现难度也不大。

**2、para-virtualization**

原理：Para-virtualization又称半虚拟化，最早由Citrix的Xen提出使用。在半虚拟化模型中，物理硬件资源统一由Hypervisor管理，由Hypervisor提供资源调用接口。虚拟子机通过特定的调用接口与Hypervisor通信，然后完整I/O资源控制操作。Para-virtualization模型图如下：



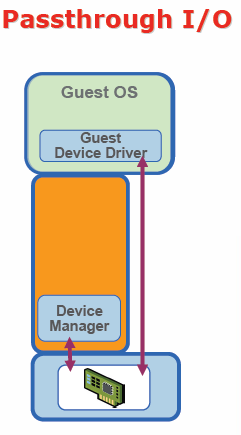
Para-virtualization又称半虚拟化，最开始由XEN提出的，XEN本身就是从虚拟化起家的。Para-virtualization模型下，虚拟子机的网卡驱动只能有Hypervisor厂商来开发，Redhat、VMware、Citrix、Microsoft这几大虚拟厂商都有各自的para-virtualization驱动。比如Redhat的KVM就叫virtio，VMware的有VMXNET2、VMXNET 3，Citrix的XEN叫xen-pv，Mircrosoft暂时没有找到(欢迎朋友们补充)。

优点：个人认为是一种改进版的emulation模型，但是由于子机和Hypervisor之间通信，性能比emulation要很多。

缺点：需要修改虚拟子机操作系统内核，添加不同Hypervisor厂商的网络驱动。比如Linux(Redhat和Novell)就在发行版里面添加了Mircosoft的para-virtualizaiton网络驱动，同样Microsoft也在自己发行版里面添加对KVM的virtio和xen-pv驱动支持。

虚拟机迁移支持：虽然不同虚拟化厂商的para-virtualization方案都不相同，由于主流操作系统都同时提供对这些方案的支持;所以这类型虚拟子机可迁移性也比较容易实现。

**3、pass-through**



原理：Hypervisor将一个PCI设备(可以是网卡、USB、光驱)直接分配给指定虚拟子机单独访问。为了安全和稳定性考虑，pass-through使用通常结合intel VT-D(AMD也有类似技术)来使用，通过iommu保证虚拟子机之间内存访问不冲突。这种技术在VMware上叫VMDirectPath I/O，其他方案中没有找到相关专门名词。

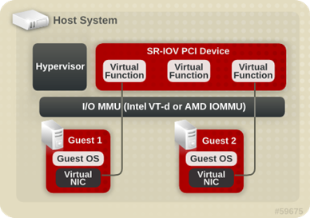
优点：性能好。单独PCI设备分配给虚拟子机，虚拟子机直接跟物理设备通信。

缺点：设备只能被一个虚拟子机使用，配置也比较复杂，首先需要在hypervisor将指定设备通过PCI id方式分配给指定虚拟子机，然后虚拟子机识别到设备再安装驱动来使用。

迁移性：迁移性方面待研究，有兴趣的朋友可以补充完善。

**4、SR-IOV**

背景：pass-through模型让虚拟子机直接使用物理设备，这样使得虚拟子机的网络性能达到最优。SR-IOV主要用来解决pass-through只能被一台虚拟子机访问的问题。SR-IOV标准由PCI-SIG，这个标准实现需要CPU、芯片组和PCI设备(主要是网卡等I/O资源)协同在硬件层面实现，SR-IOV被很多人认为是解决了虚拟化最后一公里的问题。模型图如下：



原理：SR-IOV需要网卡硬件支持，支持SR-IOV功能的网卡可以在Hypervior里面注册成多个网卡(每个网卡都独立的中断ID、收发队列、QOS管理机制)。每个设备可以通过pass-through方式分配给虚拟子机。

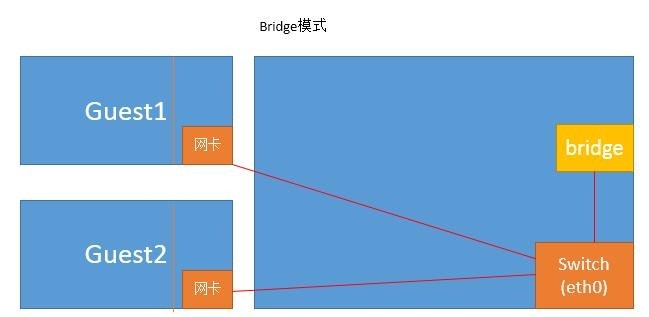
产品：常见就是基于intel 82599和82598芯片组的10Gb网卡。VMware、Redhat、Citrix和Microsoft都已经或者正在Hypervisor里面添加这个功能的支持。下面是一篇基于KVM的SR-IOV性能测试报告。(http://www.principledtechnologies.com/clients/reports/Red%20Hat/RHEL6\_SR-IOV\_DB0111.pdf)，不同厂商虚拟化方案都不尽相同，有兴趣可以在google里面搜索到更多资料。

优点：优点不用说，X86虚拟化最新的IO虚拟化模型;虚拟机不但性能好，而且结合硬件功能，为虚拟机IO管理提出了一个新方案。

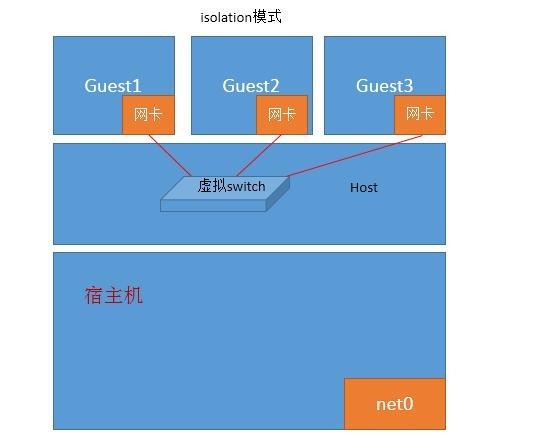
缺点：待定

迁移性：SR-IOV同时需要硬件和软件两个层面支持，虚拟子机在相同网卡主机之间迁移时理论上不会有问题。具体还要看虚拟化厂商实现。

bridge：把原宿主机上的网卡当交换机；然后虚拟出一个桥来接收发往宿主机的数据包。

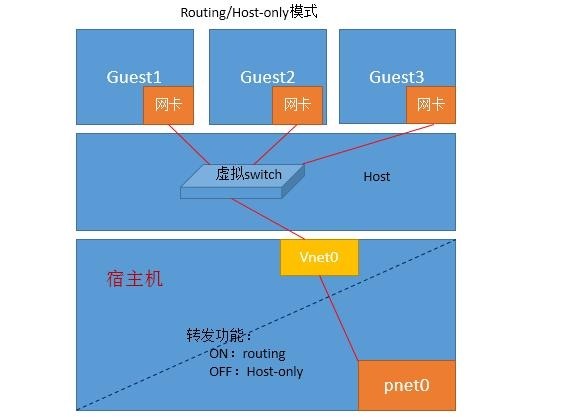


isolation mode：仅guest之间通信；不与外部网络和宿主机通信。

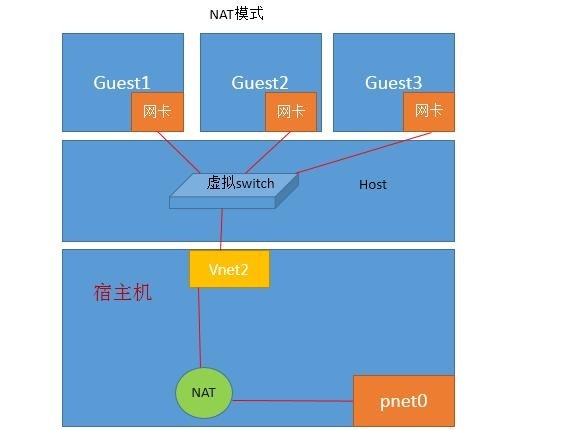


routed mode：与外部主机通信；依赖于静态路由指定到各Guest需经过pnet0。

   host-only：不与外部主机通信。

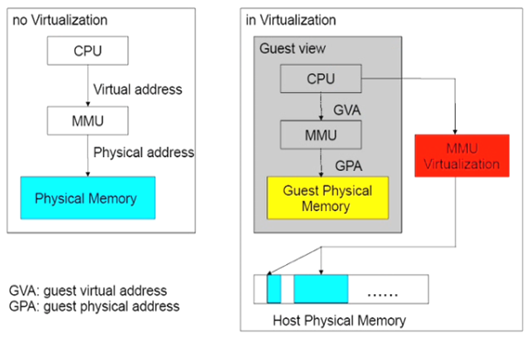


nat：地址转换；在虚拟网卡和物理网卡之间建立一个nat转发服务器；对数据包进行源地址转换。



### 内存虚拟化

首先我们知道内存本身就类似于虚拟化技术，其通过虚拟地址对外提供服务，所有的进程都以为自己可以使用所有的物理内存。如下图提供了在非虚拟化中和虚拟化中寻址方式。

****

**No Virtualation**

在非虚拟化中，系统把物理地址通过虚拟地址的方式（一个个页框）提供出去给进程使用，每个进程都以为自己可以使用所有的物理内存。本来在CPU上有个称为MMU（memory management unit）的东西，任何时候当某个进行想要访问数据自己的线性地址中的某段数据的时候，就是虚拟地址。这个进程就会传给CPU一个地址，并需要读取数据，但是CPU知道这个地址是无法真正访问到数据的，于是CPU要通过MMU将这段地址转换为对应物理地址的访问，从而这段数据就能访问到了。一般进程所得到的内存地址空间是一个连续的虚拟地址空间，而在真正的物理内存存储时一般都不会是连续的地址空间。

**In Virtualation**

为了实现内存虚拟化，让客户机使用一个隔离的、从零开始且具有连续的内存空间，像KVM虚拟机引入一层新的地址空间，即客户机物理地址空间 (Guest Physical Address, GPA)，这个地址空间并不是真正的物理地址空间，它只是宿主机虚拟地址空间在客户机地址空间的一个映射。对客户机来说，客户机物理地址空间都是从零开始的连续地址空间，但对于宿主机来说，客户机的物理地址空间并不一定是连续的，客户机物理地址空间有可能映射在若干个不连续的宿主机地址区间。

从上图我们看出，在虚拟化环境中，由于虚拟机物理地址不能直接用于宿主机物理MMU进行寻址，所以需要把虚拟机物理地址转换成宿主机虚拟地址 （Host Virtual Address, HVA）。运行在硬件之上的Hypervisor首先会对物理内存进行虚拟地址 （Host Virtual Address, HVA）转换，然后还需要对转换后的虚拟地址内存空间进行再次虚拟，然后输出给上层虚拟机使用，而在虚拟机中同样又要进行GVA转换到GPA操作。显然通过这种映射方式，虚拟机的每次内存访问都需要Hypervisor介入，并由软件进行多次地址转换，其效率是非常低的。

因此，为了提高GVA到HPA转换的效率，目前有两种实现方式来进行客户机虚拟地址到宿主机物理地址之间的直接转换。其一是基于纯软件的实现方式，也即通过**影子页表（Shadow Page Table）**来实现客户虚拟地址到宿主机物理地址之间的直接转换（KVM虚拟机是支持的）。其二是基于**硬件辅助MMU**对虚拟化的支持，来实现两者之间的转换。

其中Shadow Page Table（影子页表），其实现非常复杂，因为每一个虚拟机都需要有一个Shadow Page Table。并且这种情况会出现一种非常恶劣的结果，那就是TLB（Translation Lookaside Buffer，传输后备缓冲器）很难命中，尤其是由多个虚拟主机时，因为TLB中缓存的是GVA到GPA的转换关系，所以每一次虚拟主机切换都需要清空TLB，不然主机之间就会发生数据读取错误（因为各主机间都是GVA到GPA）。传输后备缓冲器是一个内存管理单元用于改进虚拟地址到物理地址转换后结果的缓存，而这种问题也会导致虚拟机性能低下。

此外，Intel的EPT(Extent Page Table) 技术和AMD的NPT(Nest Page Table) 技术都对内存虚拟化提供了硬件支持。这两种技术原理类似，都是在硬件层面上实现客户机虚拟地址到宿主机物理地址之间的转换。称为V**irtualation MMU**。当有了这种MMU虚拟化技术后，对于虚拟机进程来说还是同样把GVA通过内部MMU转换为GPA，并不需要改变什么，保留了完全虚拟化的好处。但是同时会自动把GVA通过Virtualation MMU技术转换为真正的物理地址（HPA）。很明显减少了由GPA到HPA的过程，提升虚拟机性能。

并且CPU厂商还提供了TLB硬件虚拟化技术，以前的TLB只是标记GVA到GPA的对应关系，就这两个字段，现在被扩充为了三个字段，增加了一个主机字段，并且由GVA到GPA以及对应变成了GVA到HPA的对应关系。明确说明这是哪个虚拟机它的GVA到HPA的映射结果。

**总结**

由此看出内存虚拟化，如果没有硬件做支撑，那么只能使用Shadow Page Table（影子页表），也就意味着TLB需要不断地进行清空。而有了内存虚拟机技术后，虚拟机的性能在某种程度上也得到了大大地提升。

### 时间虚拟化

## 虚拟化管理工具

<https://libvirt.org/sources/>

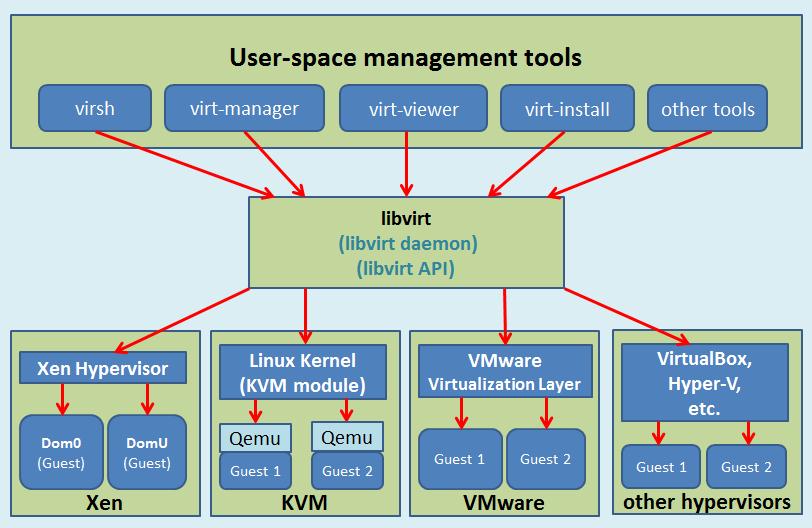
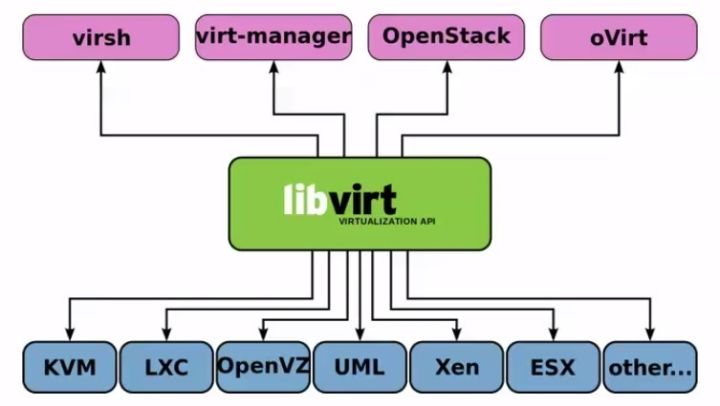
libvirt是目前使用最为广泛的对KVM虚拟机进行管理的工具和应用程序接口（API），而且一些常用的虚拟机管理工具（如virsh、virt-install、virt-manager等）和云计算框架平台（如OpenStack、OpenNebula、Eucalyptus等）都在底层使用libvirt的应用程序接口。

libvirt是为了更方便地管理平台虚拟化技术而设计的开放源代码的应用程序接口、守护进程和管理工具，它不仅提供了对虚拟化客户机的管理，也提供了对虚拟化网络和存储的管理。尽管libvirt项目最初是为Xen设计的一套API，但是目前对KVM等其他Hypervisor的支持也非常的好。 libvirt支持多种虚拟化方案，既支持包括KVM、QEMU、Xen、VMware、VirtualBox等在内的平台虚拟化方案，又支持OpenVZ、LXC等Linux容器虚拟化系统，还支持用户态Linux（UML）的虚拟化。libvirt是一个免费的开源的软件，使用的许可证是LGPL[1]（GNU宽松的通用公共许可证），使用libvirt库进行链接的软件程序不需要一定选择开源和遵守GPL许可证。和KVM、Xen等开源项目类似，libvirt也有自己的开发者社区，而且随着虚拟化、云计算等成为近年来的技术热点，libvirt项目的社区也比较活跃。目前，libvirt的开发主要由Redhat公司作为强大的支持，由于Redhat公司在虚拟化方面逐渐偏向于支持KVM（而不是Xen），故libvirt对QEMU/KVM的支持是非常成熟和稳定的。当然，IBM、Novell等公司以及众多的个人开发者，对libvirt项目的代码贡献量也是非大的。

libvirt本身提供了一套较为稳定的C语言应用程序接口，目前，在其他一些流行的编程语言中也提供了对libvirt的绑定，在Python、Perl、Java、Ruby、PHP、OCaml等高级编程语言中已经有libvirt的程序库可以直接使用。libvirt还提供了为基于AMQP（高级消息队列协议）的消息系统（如Apache Qpid）提供QMF代理，这可以让云计算管理系统中宿主机与客户机、客户机与客户机之间的消息通信变得更易于实现。libvirt还为安全的远程管理虚拟客户机提供了加密和认证等安全措施。正是由于libvirt拥有这些强大的功能和较为稳定的应用程序接口，而且它的许可证（license）也比较宽松，libvirt的应用程序接口已被广泛地用在基于虚拟化和云计算的解决方案中，主要作为连接底层Hypervisor和上层应用程序的一个中间适配层。

libvirt对多种不同的Hypervisor的支持是通过一种基于驱动程序的架构来实现的。libvirt对不同的Hypervisor提供了不同的驱动：对Xen有Xen的驱动，对QEMU/KVM有QEMU驱动，对VMware有VMware驱动。在libvirt源代码中，可以很容易找到qemu\_driver.c、xen\_driver.c、xenapi\_driver.c、vmware\_driver.c、vbox\_driver.c这样的驱动程序源代码文件。

libvirt作为中间适配层，让底层Hypervisor对上层用户空间的管理工具是可以做到完全透明的，因为libvirt屏蔽了底层各种Hypervisor的细节，为上层管理工具提供了一个统一的、较稳定的接口（API）。通过libvirt，一些用户空间管理工具可以管理各种不同的Hypervisor和上面运行的客户机，它们之间基本的交互框架如图6-1所示。



Libvirt 虚拟化库剖析 <https://www.ibm.com/developerworks/cn/linux/l-libvirt/>

在libvirt中涉及到几个重要的概念，解释如下：

1. 节点（Node）：一个物理机器，上面可能运行着多个虚拟客户机。Hypervisor和Domain都运行在Node之上。

2. Hypervisor：也称虚拟机监控器（VMM），如KVM、Xen、VMware、Hyper-V等，是虚拟化中的一个底层软件层，它可以虚拟化一个节点让其运行多个虚拟客户机（不同客户机可能有不同的配置和操作系统）。

3. 域（Domain）：是在Hypervisor上运行的一个客户机操作系统实例。域也被称为实例（instance，如亚马逊的AWS云计算服务中客户机就被称为实例）、客户机操作系统（guest OS）、虚拟机（virtual machine），它们都是指同一个概念。  
关于节点、Hypervisor和域的关系，可以简单地用图6-2来表示。

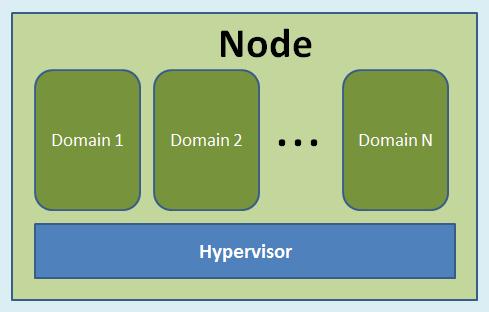
[](http://smilejay.b0.upaiyun.com/wp-content/uploads/2013/03/libvirt-node-hypervisor-domain.jpg)

图6-2 节点、Hypervisor和域 三者之间的关系

在了解了节点、Hypervisor和域的概念之后，用一句话概括libvirt的目标，就是：**为了安全高效的管理节点上的各个域，而提供一个公共的稳定的软件层**。当然，这里的管理，既包括本地的管理，也包含远程的管理。具体地讲，libvirt的管理功能主要包含如下五个部分：

**（1）域的管理**：包括对节点上的域的各个生命周期的管理，如：启动、停止、暂停、保存、恢复和动态迁移。也包括对多种设备类型的热插拔操作，包括：磁盘、网卡、内存和CPU，当然不同的Hypervisor上对这些热插拔的支持程度有所不同。

**（2）远程节点的管理**：只要物理节点上运行了libvirtd这个守护进程，远程的管理程序就可以连接到该节点进程管理操作，经过认证和授权之后，所有的libvirt功能都可以被访问和使用。libvirt支持多种网络远程传输类型，如SSH、TCP套接字、Unix domain socket、支持TLS的加密传输等。假设使用最简单的SSH，则不需要额外配置工作，比如：example.com节点上运行了libvirtd，而且允许SSH访问，在远程的某台管理机器上就可以用如下的命令行来连接到example.com上，从而管理其上的域。

virsh -c qemu+ssh://root@example.com/system

**（3）存储的管理**：任何运行了libvirtd守护进程的主机，都可以通过libvirt来管理不同类型的存储，如：创建不同格式的客户机镜像（qcow2、raw、qde、vmdk等）、挂载NFS共享存储系统、查看现有的LVM卷组、创建新的LVM卷组和逻辑卷、对磁盘设备分区、挂载iSCSI共享存储，等等。当然libvirt中，对存储的管理也是支持远程管理的。

**（4）网络的管理**：任何运行了libvirtd守护进程的主机，都可以通过libvirt来管理物理的和逻辑的网络接口。包括：列出现有的网络接口卡，配置网络接口，创建虚拟网络接口，网络接口的桥接，VLAN管理，NAT网络设置，为客户机分配虚拟网络接口，等等。

（5）提供一个稳定、可靠、高效的应用程序接口（API）以便可以完成前面的4个管理功能。

libvirt主要由三个部分组成，它们分别是：应用程序编程接口（API）库、一个守护进程（libvirtd）和一个默认命令行管理工具（virsh）。应用程序接口（API）是为了其他虚拟机管理工具（如virsh、virt-manager等）提供虚拟机管理的程序库支持。libvirtd守护进程负责执行对节点上的域的管理工作，在用各种工具对虚拟机进行管理之时，这个守护进程一定要处于运行状态中，而且这个守护进程可以分为两种：一种是root权限的libvirtd，其权限较大，可以做所有支持的管理工作；一种是普通用户权限的libvirtd，只能做比较受限的管理工作。virsh是libvirt项目中默认的对虚拟机管理的一个命令行工具，将在6.2节中详细介绍该工具。

**daemon进程（libvirtd ）**

该后台进程主要实现以下功能：

（1）远程代理

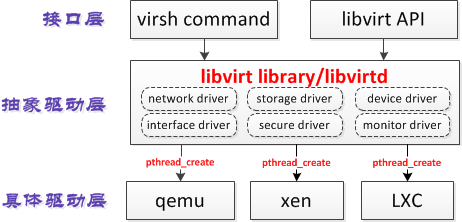
所有remote client发送来的命令，由该进程监测执行

（2）本地环境初始化

libvirt服务的启停，用户connection的响应等

（3）根据环境注册各种Driver（qemu, xen, storage…）的实现

不同虚拟化技术以Driver的形式实现，由于libvirt对外提供的是统一的接口，所以各个Driver就是实现这些接口， 即将Driver注册到libvirt中



参照上图，来理一下通过virsh命令或接口创建虚拟机实例的代码执行路径：

（1）virsh命令或API接口c创建虚拟机 -- 接口层

virsh create vm.xml 或者 virDomainPtr virDomainCreateXML (virConnectPtr conn, const char \* xmlDesc, unsigned int flags)

（2）调用libvirt提供的统一接口 -- 抽象驱动层

conn->driver->domainCreateXML(conn, xmlDesc, flags); //此处的domainCreateXML即抽象的统一接口，这里并不需要关心底层的driver是kvm，还是xen

（3）调用底层的相应虚拟化技术的接口 -- 具体驱动层

domainCreateXML = qemuDomainCreateXML; //如果driver=qemu，那么此处即调用的qemu注册到抽象驱动层上的函数qemuDomainCreateXML

（4）拼装shell命令，并执行

以qemu为例，qemuDomainCreateXML首先会拼装一条创建虚拟机的命令，比如qemu -hda disk.img，然后创建一个新的线程来执行

回过头来思考，libvirt通过4步，将最底层的直接在shell中输入命令来完成的操作进行了抽象封装，给应用程序开发人员提供了统一的，易用的接口。

**抽象驱动层**

目前，libvirt以下几种类型的抽象驱动，每一种类型的驱动代表某以功能模块的抽象封装：

（1）虚拟化驱动(virDriverPtr)

（2）虚拟网络驱动(virNetworkDriverPtr)

（3）物理网卡驱(virInterfaceDriverPtr)

（4）存储驱动(virStorageDriverPtr)

（5）监控驱动(virDeviceMonitorPtr)

（6）安全驱动(virSecretDriverPtr)

（7）过滤驱动(virNWFilterDriverPtr)

（8）状态驱动(virStateDriverPtr)

（1）连接 Hypervisor 相关的API：以virConnect 开头的一系列函数。

只有与 Hypervisor 建立了连接之后，才能进行虚拟机管理操作，所以连接 Hypervisor 的API是其他所有API使用的前提条件。与 Hypervisor 建立的连接是为其他API的执行提供了路径，是其他虚拟化管理功能的基础。通过调用 virConnectOpen 函数可以建立一个连接，其返回值是一个virConnectPtr 对象，该对象就代表到 Hypervisor 的一个连接；如果连接出错，则返回空值（NULL）。而 virConnectOpenReadOnly 函数会建立一个只读的连接，在该连接上可以使用一些查询的功能，而不使用创建、修改等功能。 virConnectOpenAuth 函数提供了更具认证建立的连接。 virConnectGetCapabilities 函数是返回对 Hypervisor 和驱动的功能的描述的 XML 格式的字符串。virConnectListDomains函数返回一列域标识符，它们代表该 Hypervisor 上的活动域。

（2）域管理的 API：以virDomain 开头的一系列函数。

虚拟机的管理，最基本的职能就是对各个节点上的域的管理，故 libvirt API 中实现了很多针对域管理的函数。要管理域，首先就要获取virDomainPtr 这个域对象，然后才能对域进行操作。有很多种方式来获取域对象，如 virDomainPtr virDomainLookupByID (virConnectPtr conn, int id) 函数是根据域的 id 值到 conn 这个连接上去查找相应的域。类似地，virDomainLookupByName、virDomainLookupByUUID 等函数分别是根据域的名称和 UUID 去查找相应的域。在得到了某个域的对象后，就可以进行很多的操作，可以是查询域的信息（如：virDomainGetHostname、virDomainGetInfo、virDomainGetVcpus、virDomainGetVcpusFlags、virDomainGetCPUStats，等等），也可以是控制域的生命周期（如：virDomainCreate 、virDomainSuspend 、virDomainResume 、virDomainDestroy 、virDomainMigrate，等等）。

（3）节点管理的 API：以virNode 开头的一系列函数。

域是运行在物理节点之上，libvirt也提供了对节点的信息查询和控制的功能。节点管理的多数函数都需要使用一个连接 Hypervisor 的对象作为其中的一个传入参数，以便可以查询或修改到该连接上的节点的信息。virNodeGetInfo函数是获取节点的物理硬件信息，virNodeGetCPUStats 函数可以获取节点上各个 CPU 的使用统计信息，virNodeGetMemoryStats 函数可以获取节点上的内存的使用统计信息，virNodeGetFreeMemory 函数可以获取节点上可用的空闲内存大小。也有一些设置或者控制节点的函数，如virNodeSetMemoryParameters 函数可以设置节点上的内存调度的参数，virNodeSuspendForDuration 函数可以让节点（宿主机）暂停运行一段时间。

（4）网络管理的 API：以 virNetwork 开头的一系列函数和部分以 virInterface 开头的函数。

libvirt 对虚拟化环境中的网络管理也提供了丰富的API。libvirt 首先需要创建virNetworkPtr 对象，然后才能查询或控制虚拟网络。一些查询网络相关信息的函数，如：virNetworkGetName 函数可以获取网络的名称，virNetworkGetBridgeName 函数可以获取该网络中网桥的名称，virNetworkGetUUID 函数可以获取网络的 UUID 标识，virNetworkGetXMLDesc 函数可以获取网络的以 XML 格式的描述信息，virNetworkIsActive 函数可以查询网络是否正在使用中。一些控制或更改网络设置的函数，有：virNetworkCreateXML 函数可以根据提供的 XML 格式的字符串创建一个网络（返回 virNetworkPtr 对象），virNetworkDestroy 函数可以销毁一个网络（同时也会关闭使用该网络的域），virNetworkFree 函数可以回收一个网络（但不会关闭正在运行的域），virNetworkUpdate 函数可根据提供的 XML 格式的网络配置来更新一个已存在的网络。另外，virInterfaceCreate、virInterfaceFree、virInterfaceDestroy、virInterfaceGetName、virInterfaceIsActive 等函数可以用于创建、释放和销毁网络接口，以及查询网络接口的名称和激活状态。

（5）存储卷管理的 API：以 virStorageVol 开头的一系列函数。

libvirt 对存储卷（volume）的管理，主要是对域的镜像文件的管理，这些镜像文件可能是 raw、qcow2、vmdk、qed等各种格式。libvirt 对存储卷的管理，首先需要创建virStorageVolPtr 这个存储卷的对象，然后才能对其进行查询或控制操作。libvirt 提供了3个函数来分别通过不同的方式来获取存储卷对象，如：virStorageVolLookupByKey 函数可以根据全局唯一的键值来获得一个存储卷对象，virStorageVolLookupByName 函数可以根据名称在一个存储资源池（storage pool）中获取一个存储卷对象，virStorageVolLookupByPath 函数可以根据它在节点上路径来获取一个存储卷对象。有一些函数用于查询存储卷的信息，如：virStorageVolGetInfo 函数可以查询某个存储卷的使用情况，virStorageVolGetName 函数可以获取存储卷的名称，virStorageVolGetPath 函数可以获取存储卷的路径，virStorageVolGetConnect 函数可以查询存储卷的连接。一些函数用于创建和修改存储卷，如：virStorageVolCreateXML 函数可以根据提供的 XML 描述来创建一个存储卷，virStorageVolFree 函数可以释放存储卷的句柄（但是存储卷依然存在），virStorageVolDelete 函数可以删除一个存储卷，virStorageVolResize 函数可以调整存储卷的大小。

（6）存储池管理的 API：以virStoragePool 开头的一系列函数。

libvirt 对存储池（pool）的管理，包括对本地的基本文件系统、普通网络共享文件系统、iSCSI共享文件系统、LVM分区等的管理。libvirt 需要基于 virStoragePoolPtr 这个存储池对象才能进行查询和控制操作。一些函数可以通过查询获取一个存储池对象，如：virStoragePoolLookupByName 函数可以根据存储池的名称来获取一个存储池对象，virStoragePoolLookupByVolume 可以根据一个存储卷返回其对应的存储池对象。virStoragePoolCreateXML 函数可以根据 XML 描述来创建一个存储池（默认已激活），virStoragePoolDefineXML 函数可以根据 XML 描述信息静态地定义个存储池（尚未激活），virStoragePoolCreate 函数可以激活一个存储池。virStoragePoolGetInfo、virStoragePoolGetName、virStoragePoolGetUUID等函数可以分别获取存储池的信息、名称和 UUID 标识。virStoragePoolIsActive函数可以查询存储池是否处于使用中状态。virStoragePoolFree 函数可以释放存储池相关的内存（但是不改变其在宿主机中的状态），virStoragePoolDestroy 函数可以用于销毁一个存储池（但并没有释放virStoragePoolPtr 对象，之后还可以用virStoragePoolCreate 函数重新激活它），virStoragePoolDelete 函数可以物理删除一个存储池资源（该操作不可恢复）。

（7）事件管理的API：以virEvent 开头的一系列函数。

libvirt 支持事件机制，使用该机制注册之后，可以在发生特定的事件（如：域的启动、暂停、恢复、停止等）之时，得到自己定义的一些通知。

（8）数据流管理的API：以virStream 开头的一系列函数。

libvirt 还提供了一系列函数用于数据流的传输。对于 libvirt API 一些细节的使用方法和实现原理，可以参考其源代码或者本章末尾的参考阅读中介绍资料。

**依赖库**

#!/bin/bash

apt-get update

apt-get -f install

sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

apt-get install open-iscsi nfs-kernel-server sysstat ifupdown lvm2 build-essential automake gdb

apt-get install subversion git flex bison libtool shtool pkg-config gettext libpython2.7 libnl1

apt-get install libsepol1 expect libjna-java zlib1g-dev libxml2-dev libpci-dev libpciaccess-dev

apt-get install libgpg-error-dev libgcrypt11-dev libtasn1-3-dev libgnutls-dev libsepol1-dev libselinux1-dev

apt-get install libudev-dev libdevmapper-dev uuid-dev libblkid-dev libparted0-dev libssl-dev

apt-get install libexpat1-dev python2.7-dev python-dev libconfig-dev libglib2.0-data libglib2.0-0

apt-get install libglib2.0-bin libglib2.0-dev libaio-dev libreadline-dev libyajl-dev libbz2-dev dos2unix

apt-get install libsasl2-dev libalsa-ocaml-dev libnss3-dev bcc libpixman-1-dev libnuma-dev

# my dev depends

apt-get install ctags cscope samba unzip texinfo unrar kernel-package

apt-get install python-libxml2

# for libvirt-1.3.0

apt-get install libxml-xpath-perl w3c-dtd-xhtml xsltproc

# for autobuild

apt-get install autopoint

## 动态迁移

动态迁移是虚拟化特有的新特性，它将虚拟机从一个物理机快速迁移到另一个物理机，但是虚拟机里面的程序和网络都保持连接。从用户的角度来看，动态迁移对虚拟机的可用性没有任何影响，用户不会察觉任何的服务被中断。

动态迁移实现的方法是在目的服务器上建立一台同样配置的新虚拟机，然后不断地在两个虚拟机之间同步各种内部状态，比如内存、外设、CPU等。等状态同步完成后，关掉老的虚拟机，启动新的虚拟机。

## 参考文档

**虚拟机迁移技术漫谈**

[**https://www.ibm.com/developerworks/cn/linux/l-cn-mgrtvm1/index.html**](https://www.ibm.com/developerworks/cn/linux/l-cn-mgrtvm1/index.html)

KVM虚拟化技术：实战与原理解析

**KVM虚拟化网络优化技术总结**

[**http://www.10tiao.com/html/251/201505/205614320/2.html**](http://www.10tiao.com/html/251/201505/205614320/2.html)

[**http://blog.chinaunix.net/uid-20201831-id-5775661.html**](http://blog.chinaunix.net/uid-20201831-id-5775661.html)

[**http://oenhan.com/category/virtualization**](http://oenhan.com/category/virtualization)

# Docker

Docker使用Google公司推出的Go语言进行开发实现，基于Linux内核的cgroup，namespace，以及AUFS类的Union FS等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。最初实现是基于 LXC，从0.7以后开始去除LXC，转而使用自行开发的 libcontainer，从1.11开始，则进一步演进为使用 runC 和containerd。

Docker Engine改为Docker CE（社区版）

它包含了CLI客户端、后台进程/服务以及API。用户像以前以同样的方式获取。

Docker Data Center改为Docker EE（企业版）

在Docker三个定价层增加了额外的支付产品和支持

## 概念

Docker是一个开源的应用容器引擎，它让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到安装了任何 Linux 发行版本的机器上。Docker基于LXC来实现类似VM的功能，可以在更有限的硬件资源上提供给用户更多的计算资源。与同VM等虚拟化的方式不同，LXC不属于全虚拟化、部分虚拟化或半虚拟化中的任何一个分类，而是一个操作系统级虚拟化。

Docker是直接运行在宿主操作系统之上的一个容器，使用沙箱机制完全虚拟出一个完整的操作，容器之间不会有任何接口，从而让容器与宿主机之间、容器与容器之间隔离的更加彻底。每个容器会有自己的权限管理，独立的网络与存储栈，及自己的资源管理能，使同一台宿主机上可以友好的共存多个容器。

Docker借助Linux的内核特性，如：控制组（Control Group）、命名空间（Namespace）等，并直接调用操作系统的系统调用接口。从而降低每个容器的系统开销，并实现降低容器复杂度、启动快、资源占用小等特征。

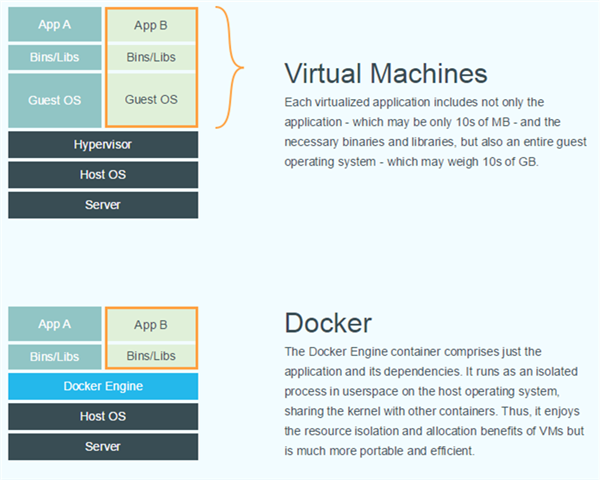
Docker定义的标准 + 服务应用

- 基础设施标准化（Docker Engine）：有了Docker Engine，可将 Docker 容器跑起来

- 应用交付的标准化（Docker Image）: 提供了一套应用快速打包为轻量级 Docker Image的方法。开发人员在代码完成之后，可以将其打包为镜像

- 运维管理的标准化（Docker container）：运维人员不在需要将应用准备系统、运行环境、组件和基础软件包,容器时代，应用都运行在一个个的 Docker container中。标准运维将关注容器，而不是复杂的系统环境。

- 分发部署标准化（Docker Registry）：指的是容器化之后不同版本的应用镜像都存储在镜像仓库中。



作为一种新兴的虚拟化方式， Docker 跟传统的虚拟化方式相比具有众多的优势。

1）更高效的利用系统资源

由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销， Docker 对系统资源的利用率更高。 无论是应用执行速度、 内存损耗或者文件存储速度， 都要比传统虚拟机技术更高效。 因此， 相比虚拟机技术， 一个相同配置的主机， 往往可以运行更多数量的应用。

2）更快速的启动时间

传统的虚拟机技术启动应用服务往往需要数分钟， 而 Docker 容器应用， 由于直接运行于宿主内核， 无需启动完整的操作系统， 因此可以做到秒级、 甚至毫秒级的启动时间。 大大的节约了开发、 测试、 部署的时间。

3）一致的运行环境

开发过程中一个常见的问题是环境一致性问题。 由于开发环境、 测试环境、 生产环境不一致， 导致有些 bug 并未在开发过程中被发现。 而 Docker 的镜像提供了除内核外完整的运行时环境， 确保了应用运行环境一致性， 从而不会再出现 “这段代码在我机器上没问题啊” 这类问题。

4）持续交付和部署

对开发和运维（ DevOps） 人员来说， 最希望的就是一次创建或配置， 可以在任意地方正常运行。使用 Docker 可以通过定制应用镜像来实现持续集成、 持续交付、 部署。 开发人员可以通过 Dockerfile 来进行镜像构建， 并结合 持续集成(Continuous Integration) 系统进行集成测试， 而运维人员则可以直接在生产环境中快速部署该镜像， 甚至结合持续部署(Continuous Delivery/Deployment) 系统进行自动部署。

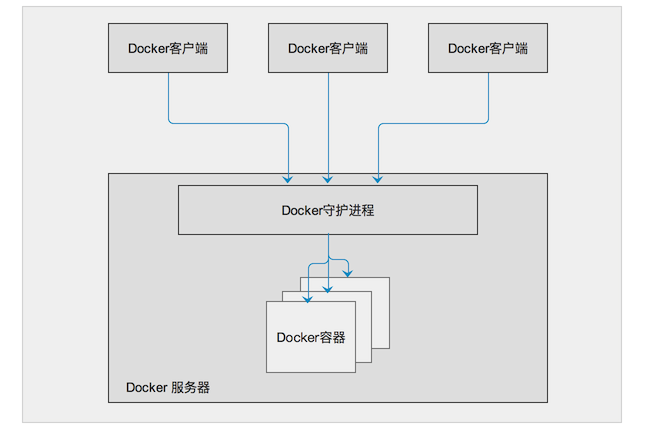
对比传统虚拟机总结

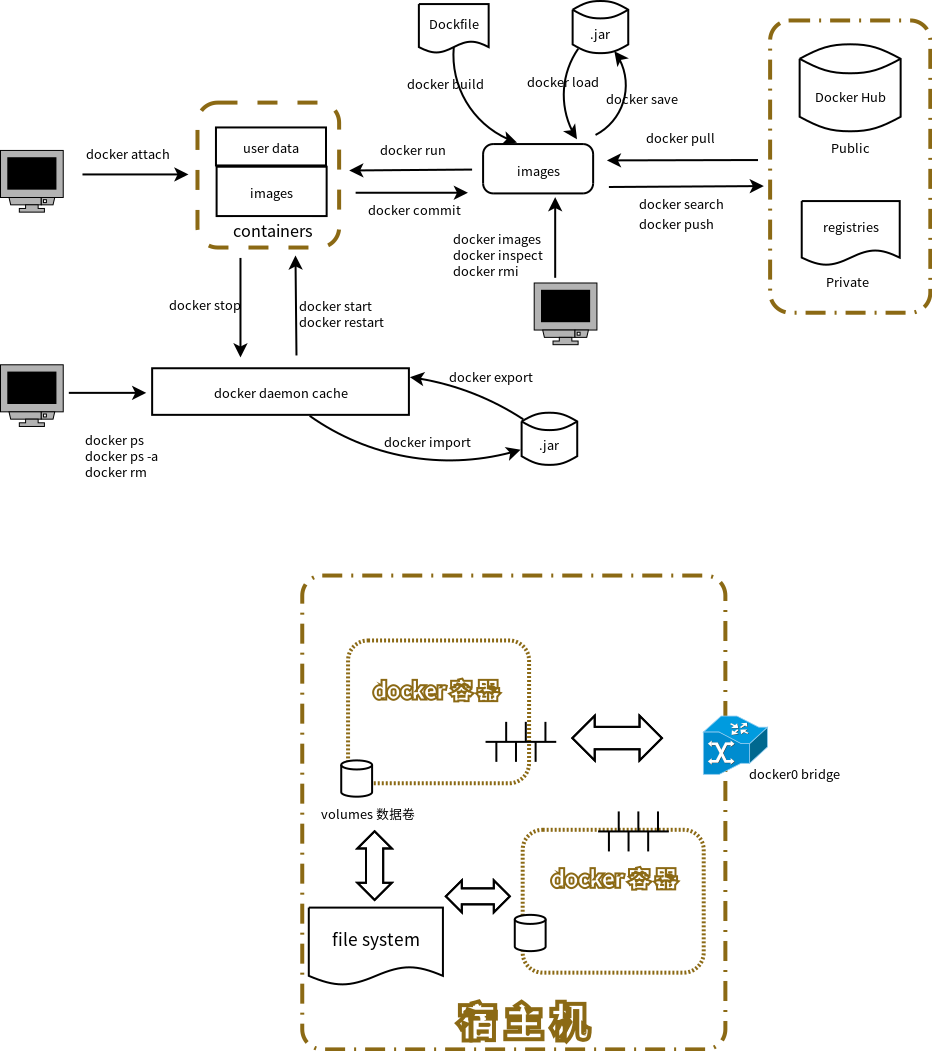
|  |  |  |
| --- | --- | --- |
| 特性 | 容器 | 虚拟机 |
| 启动 | 秒级 | 分钟级 |
| 硬盘使用 | 一般为 MB | 一般为 GB |
| 性能 | 接近原生 | 弱于 |
| 系统支持量 | 单机支持上千个容器 | 一般几十个 |

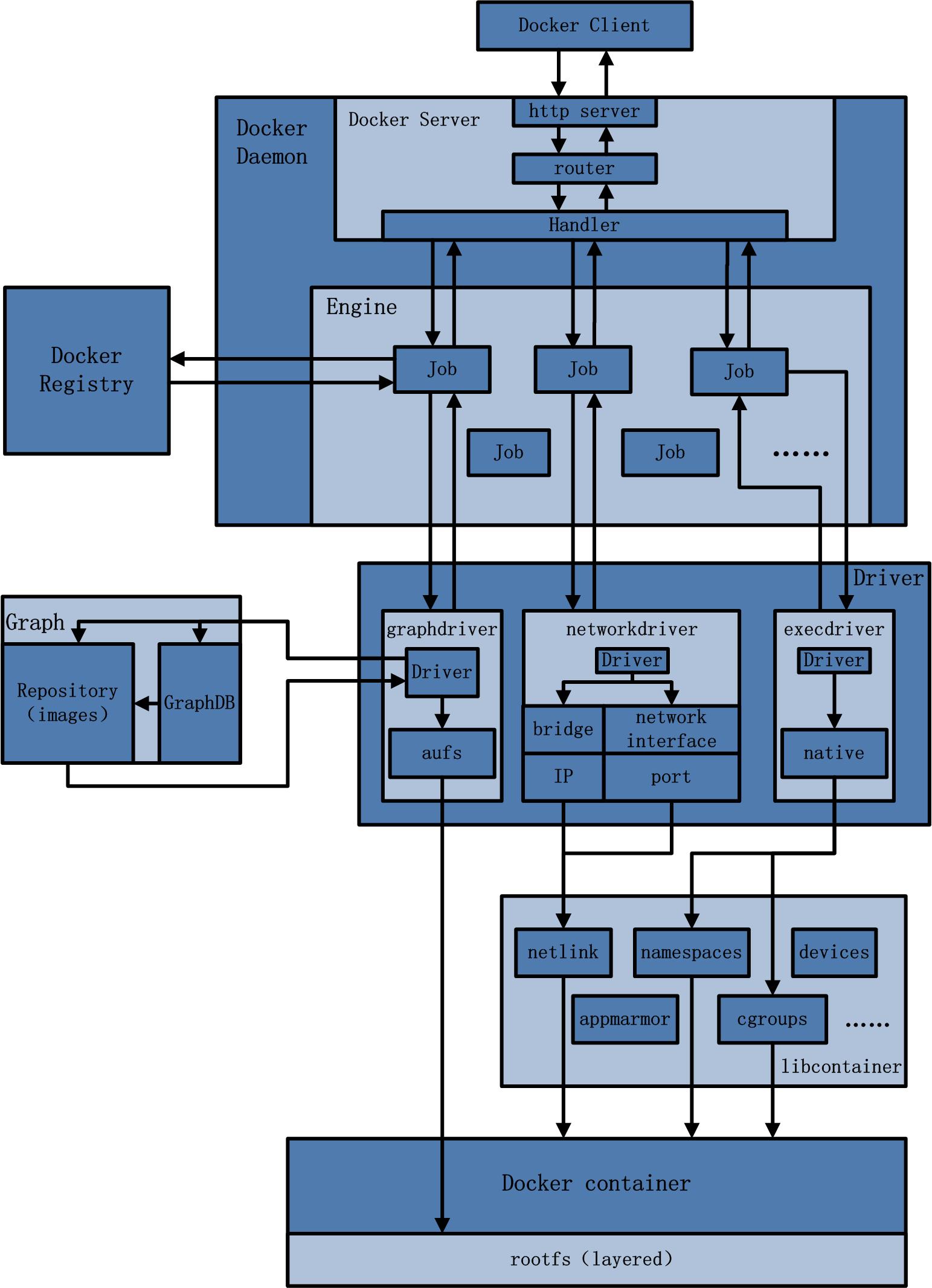
## 架构

Docker使用客户端-服务器 (C/S) 架构模式 使用远程API来管理和创建Docker容器。Docker 容器（Container）通过 Docker 镜像（Image）来创建，二者之间的关系类似于面向对象编程中的对象与类。

其中，Docker客户端用于向Docker服务器或守护进程发送请求。而服务器或守护进程会接受来自客户的请求（创建、运行、分发容器），并在处理完请求后返回响应结果。Docker提供了一个命令行工具docker，及一整套RESTful API。客户端和服务端既可以运行在一个机器上，也可通过 socket或者RESTful API来进行通信。







如图3.1，不难看出，用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。

　　而Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；而后Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。

　　Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。

　　而libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作。

　　当执行完运行容器的命令后，一个实际的Docker容器就处于运行状态，该容器拥有独立的文件系统，独立并且安全的运行环境等。

## 基本概念

Docker Image（镜像）、Registry（仓库）、Docker Container（容器）是Docker中包含几个核心组件（核心概念），这几个组件介绍如下：

**Docker Image - 镜像**

镜像用于构建Docker容器，是构成Docker的基础。它类似于虚拟机中的镜像，但它非常小，可以将它理解为容器的源代码。是一种层式结构的增量文件系统，由一系列命令一步步组成。

我们都知道， 操作系统分为内核和用户空间。 对于 Linux 而言， 内核启动后， 会挂载 root 文件系统为其提供用户空间支持。 而 Docker 镜像（ Image） ， 就相当于是一个 root 文件系统。 比如官方镜像 ubuntu:14.04 就包含了完整的一套Ubuntu 14.04 最小系统的 root 文件系统。Docker 镜像是一个特殊的文件系统， 除了提供容器运行时所需的程序、 库、 资源、 配置等文件外， 还包含了一些为运行时准备的一些配置参数（ 如匿名卷、 环境变量、 用户等） 。 镜像不包含任何动态数据， 其内容在构建之后也不会被改变。

**Registry - 仓库**

Registry是Docker用于存放镜像文件仓库，仓库分为开放（Public）和私有（Private），其设计理念非常类似于代码仓库Git。

[Docker Hub](https://hub.docker.com/)是Docker公司运营的Registry，也是最大的Docker仓库，用户可以在上面注册帐号分享并保存自己的镜像。除Docker Hub外，还有很多第三方镜像仓库。如果不希望自己的镜像被其它人看到，你甚至可以架设自己的私有仓库，以增强安全性或满足一些特殊需求。

镜像构建完成后， 可以很容易的在当前宿主上运行， 但是， 如果需要在其它服务器上使用这个镜像， 我们就需要一个集中的存储、 分发镜像的服务， Docker Registry就是这样的服务。

一个 Docker Registry 中可以包含多个仓库（ Repository） ； 每个仓库可以包含多个标签（ Tag） ； 每个标签对应一个镜像。通常， 一个仓库会包含同一个软件不同版本的镜像， 而标签就常用于对应该软件的各个版本。 我们可以通过 <仓库名>:<标签> 的格式来指定具体是这个软件哪个版本的镜像。 如果不给出标签， 将以 latest 作为默认标签。

仓库名经常以 两段式路径 形式出现， 比如 jwilder/nginx-proxy ， 前者往往意味着 Docker Registry 多用户环境下的用户名， 后者则往往是对应的软件名。 但这并非绝对， 取决于所使用的具体 Docker Registry 的软件或服务。

*注意：*也可以将Repository认为是Docker仓库，实际上二者区分并不严格。

**Docker Container - 容器**

容器是用于运行镜像的沙箱环境，它是从镜像创建的应用运行实例，可以将其启动、开始、停止、删除。而这些容器都是相互隔离、互不可见的。我们可以理解为：镜像是Docker生命周期的打包和构建阶段，而容器Docker的启动和执行阶段。

Docker借签了集装箱的设计概念，不同的是集装箱运送的货物，而Docker运送的是软件。和集装箱一样，我们并不需要关心里面装的是什么，只需要将应用按标准的形式装进去，然后上传到Registry，在需要使用它的服务器下载部署即可。Docker容器可以非常方便的替换，且各个容器都有良好的通用性，易于分发和叠加。

容器的实质是进程， 但与直接在宿主执行的进程不同， 容器进程运行于属于自己的独立的命名空间。 因此容器可以拥有自己的 root 文件系统、 自己的网络配置、自己的进程空间， 甚至自己的用户 ID 空间。 容器内的进程是运行在一个隔离的环境里， 使用起来， 就好像是在一个独立于宿主的系统下操作一样。 这种特性使得容器封装的应用比直接在宿主运行更加安全。

按照 Docker 最佳实践的要求， 容器不应该向其存储层内写入任何数据， 容器存储层要保持无状态化。 所有的文件写入操作， 都应该使用 数据卷（ Volume） 、 或者绑定宿主目录， 在这些位置的读写会跳过容器存储层， 直接对宿主(或网络存储)发生读写， 其性能和稳定性更高。

数据卷的生存周期独立于容器， 容器消亡， 数据卷不会消亡。 因此， 使用数据卷后， 容器可以随意删除、 重新 run ， 数据却不会丢失。

## 适用范围

正如前所述，Docker为容器提供了标准的隔离性，而各个容器本身就是一个标准的系统环境。因些，Docker非常适合构建服务块，以下是Docker的一些应用场景：

* 方便应用的自动化打包与部署 - 加速本地开发和构建流程，使其更加高效和轻量化。本地开发人员可以构建、运行并分享Docker容器。容器可以在开发环境中构建，然后轻松的提交到测试环境，并最终进入生产环境。
* 创建轻量、私密的PAAS环境
* 部署和扩展Web应用、数据库与后台服务 - 可以让独立服务在不同环境中得到相同的执行结果
* 连续的集成/部署、测试自动化 - 开发者可以在本地构建运行或测试环境，而不是直接在生产环境中部署、测试。

Docker也并不是万能的，其设计目的也并不是要成为KVM等的替代品。不可避免的，Docker也会有一些局限性，如：

* 基于Linux内核，所以只能运行于Linux环境中，且只能64位主机上
* 基于LXC实现容器，而LXC是基于Linux内核中的cgroup。因此，Docker容器使用的权限、物理资源等也受限于LXC

## 安装

官方网站上有各种环境下的安装指南，

<https://docs.docker.com/install/#desktop>

<https://docs.docker.com/install/linux/docker-ce/centos/#prerequisites>

<https://download.docker.com/linux/centos/7/x86_64/stable/Packages/>

添加内核配置参数以启用这些功能。

$ sudo tee -a /etc/sysctl.conf <<-EOF

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

EOF

然后重新加载 sysctl.conf 即可

$ sudo sysctl –p

解决方法：

[root@localhost ~]# modprobe br\_netfilter

启动 Docker 引擎

$ sudo systemctl enable docker

$ sudo systemctl start docker

建立 docker 组：

$ sudo groupadd docker

将当前用户加入 docker 组：

$ sudo usermod -aG docker $USER

## 使用Docker镜像

镜像是 Docker 的三大组件之一。Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地， Docker 会从镜像仓库下载（ 默认是 Docker Hub 公共注册服务器中的仓库） 。

本章将介绍更多关于镜像的内容，包括：

* 从仓库获取镜像；
* 管理本地主机上的镜像；
* 介绍镜像实现的基本原理。

**获取镜像**

Docker Hub上有大量的高质量的镜像可以用<https://hub.docker.com/explore/>

从 Docker Registry 获取镜像的命令是 docker pull 。 其命令格式为：

docker pull [选项] [Docker Registry地址]<仓库名>:<标签>

具体的选项可以通过 docker pull --help 命令看到， 这里我们说一下镜像名称的格式。

Docker Registry地址： 地址的格式一般是 <域名/IP>[:端口号] 。 默认地址是Docker Hub。

仓库名： 如之前所说， 这里的仓库名是两段式名称， 既 <用户名>/<软件名> 。

对于 Docker Hub， 如果不给出用户名， 则默认为 library ， 也就是官方镜像。

比如：

$ docker pull ubuntu:14.04

**运行**

有了镜像后， 我们就可以以这个镜像为基础启动一个容器来运行。 以上面的ubuntu:14.04 为例， 如果我们打算启动里面的 bash 并且进行交互式操作的话， 可以执行下面的命令。

$ docker run -it --rm ubuntu:14.04 bash

docker exec 命令进入容器， 修改其内容。

$ docker exec -it webserver bash

**列出镜像**

要想列出已经下载下来的镜像， 可以使用 docker images 命令。

$ docker images

**利用 commit 理解镜像构成**

镜像是容器的基础， 每次执行 docker run 的时候都会指定哪个镜像作为容器运行的基础。讲解如何定制镜像。

回顾一下之前我们学到的知识， 镜像是多层存储， 每一层是在前一层的基础上进行的修改； 而容器同样也是多层存储，是在以镜像为基础层， 在其基础上加一层作为容器运行时的存储层。现在让我们以定制一个 Web 服务器为例子， 来讲解镜像是如何构建的。

docker run --name webserver -d -p 80:80 nginx

这条命令会用 nginx 镜像启动一个容器，命名为 webserver ， 并且映射了 80端口， 这样我们可以用浏览器去访问这个 nginx 服务器。

要知道，当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。而Docker提供了一个docker commit命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候， 就会拥有原有容器最后的文件变化。

docker commit 的语法格式为：

docker commit [选项] <容器ID或容器名> [<仓库名>[:<标签>]]

docker commit \

--author "Tao Wang <twang2218@gmail.com>" \

--message "修改了默认网页" \

webserver \

nginx:v2 \

sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214

其中 --author 是指定修改的作者， 而 --message 则是记录本次修改的内容。

## 使用Dockerfile定制镜像

镜像的定制实际上就是定制每一层所添加的配置、文件。如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么之前提及的无法重复的问题、镜像构建透明性的问题、体积的问题就都会解决。这个脚本就是Dockerfile。 Dockerfile是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

**FROM 指定基础镜像**

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个nginx镜像的容器，再进行修改一样，基础镜像是必须指定的。而FROM就是指定基础镜像，因此一个Dockerfile中FROM是必备的指令，并且必须是第一条指令。

除了选择现有镜像为基础镜像外，Docker还存在一个特殊的镜像，名为scratch。这个镜像是虚拟的概念，并不实际存在，它表示一个空白的镜像。

FROM scratch

...

如果你以scratch为基础镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。

**RUN执行命令**

RUN指令是用来执行命令行命令的。由于命令行的强大能力，RUN指令在定制镜像时是最常用的指令之一。其格式有两种：

shell格式：RUN <命令>，就像直接在命令行中输入的命令一样。刚才写的Dockrfile中的RUN指令就是这种格式。

exec格式：RUN ["可执行文件", "参数1", "参数2"]，这更像是函数调用中的格式。

Dockerfile中每一个指令都会建立一层，RUN也不例外。每一个RUN的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，commit这一层的修改，构成新的镜像。

Union FS是有最大层数限制的，比如 AUFS，曾经是最大不得超过42层，现在是不得超过127层。

Docker制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

**构建镜像**

FROM nginx

RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html

在 Dockerfile 文件所在目录执行：

$ docker build -t nginx:v3 .

docker build 命令进行镜像构建。 其格式为：

docker build [选项] <上下文路径/URL/->

在这里我们指定了最终镜像的名称-t nginx:v3，构建成功后，我们可以像之前运行nginx:v2那样来运行这个镜像，其结果会和 nginx:v2一样。

**镜像构建上下文（Context）**

如果注意，会看到 docker build 命令最后有一个 . 。 . 表示当前目录， 而Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定Dockerfile所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

首先我们要理解docker build的工作原理。Docker在运行时分为Docker引擎（也就是服务端守护进程）和客户端工具。Docker的引擎提供了一组REST API，被称为Docker Remote API， 而如docker命令这样的客户端工具， 则是通过这组API与Docker引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种docker功能，但实际上，一切都是使用的远程调用形式在服务端（Docker引擎）完成。也因为这种C/S设计，让我们操作远程服务器的Docker引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过RUN指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD指令等。而 docker build命令构建镜像， 其实并非在本地构建，而是在服务端，也就是Docker引擎中构建的。**那么在这种客户端/服务端的架构中， 如何才能让服务端获得本地文件呢？这就引入了上下文的概念**。**当构建的时候，用户会指定构建镜像上下文的路径 docker build命令得知这个路径后，会将路径下的所有内容打包，然后上传给Docker引擎。这样Docker引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。**

如果在Dockerfile中这么写：

COPY ./package.json /app/

这并不是要复制执行docker build命令所在的目录下的package.json，也不是复制Dockerfile所在目录下的package.json，而是复制上下文（context）目录下的 package.json 。

COPY 这类指令中的源文件的路径都是相对路径。

现在就可以理解刚才的命令docker build -t nginx:v3 . 中的这个 . ，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给Docker 引擎以帮助构建镜像。

一般来说，应该会将Dockerfile置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker引擎， 那么可以用 .gitignore 一样的语法写一个 .dockerignore ， 该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

其它docker build 的用法

直接用Git repo进行构建，或许你已经注意到了，docker build还支持从URL构建，比如可以直接从 Git repo中构建：

## Dockerfile指令详解

**COPY 复制文件**

格式：

COPY <源路径>... <目标路径>

COPY ["<源路径1>",... "<目标路径>"]

和RUN指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。

COPY指令将从构建上下文目录中 <源路径> 的文件/目录复制到新的一层的镜像内的 <目标路径> 位置。比如：

COPY package.json /usr/src/app/

CMD 指令的格式和 RUN 相似， 也是两种格式：

shell 格式：CMD <命令>

exec 格式：CMD ["可执行文件", "参数1", "参数2"...]

参数列表格式：CMD ["参数1", "参数2"...]。在指定了ENTRYPOINT 指令后，用CMD指定具体的参数。

Docker不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候，需要指定所运行的程序及参数。CMD指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令， 比如， ubuntu镜像默认的CMD是/bin/bash，如果我们直接 docker run -it ubuntu的话， 会直接进入bash。我们也可以在运行时指定运行别的命令，如docker run -it ubuntu cat /etc/os-release。这就是用 cat /etc/os-release命令替换了默认的 /bin/bash 命令了，输出了系统版本信息。在指令格式上，一般推荐使用exec格式，这类格式在解析时会被解析为 JSON数组，因此一定要使用双引号 " ，而不要使用单引号。

Docker不是虚拟机，容器中的应用都应该以前台执行，而不是像虚拟机、物理机里面那样， 用upstart/systemd去启动后台服务，容器内没有后台服务的概念。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的，主进程退出，容器就失去了存在的意义，从而退出，其它辅助进程不是它需要关心的东西。

ENTRYPOINT 入口点

ENTRYPOINT 的格式和 RUN 指令格式一样， 分为 exec 格式和 shell 格式。

ENTRYPOINT 的目的和 CMD 一样， 都是在指定容器启动程序及参数。 ENTRYPOINT 在运行时也可以替代， 不过比 CMD 要略显繁琐， 需要通过docker run 的参数 --entrypoint 来指定。当指定了 ENTRYPOINT 后， CMD 的含义就发生了改变， 不再是直接的运行其命

令， 而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令， 换句话说实际执行

时， 将变为：

<ENTRYPOINT> "<CMD>"

ENV设置环境变量

格式有两种：

ENV <key> <value>

ENV <key1>=<value1> <key2>=<value2>...

这个指令很简单，就是设置环境变量而已，无论是后面的其它指令，如RUN，还

是运行时的应用，都可以直接使用这里定义的环境变量。

ENV VERSION=1.0 DEBUG=on \

NAME="Happy Feet"

ARG构建参数

格式： ARG <参数名>[=<默认值>]

构建参数和ENV的效果一样，都是设置环境变量。所不同的是，ARG所设置的构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。

VOLUME 定义匿名卷

格式为：

VOLUME ["<路径1>", "<路径2>"...]

VOLUME <路径>

之前我们说过， 容器运行时应该尽量保持容器存储层不发生写操作， 对于数据库类需要保存动态数据的应用，其数据库文件应该保存于卷(volume)中。

EXPOSE声明端口

格式为EXPOSE <端口1> [<端口2>...] 。

EXPOSE指令是声明运行时容器提供服务端口，这只是一个声明， 在运行时并不

会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有

两个好处，一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映

射；另一个用处则是在运行时使用随机端口映射时，也就是docker run -P时，会自动随机映射EXPOSE的端口。

要将EXPOSE和在运行时使用-p <宿主端口>:<容器端口> 区分开来。-p，是映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

WORKDIR指定工作目录

格式为WORKDIR <工作目录路径> 。

使用WORKDIR指令可以来指定工作目录（ 或者称为当前目录），以后各层的当前目录就被改为指定的目录，该目录需要已经存在， WORKDIR 并不会帮你建立目录。

**USER指定当前用户**

格式： USER <用户名>

USER指令和WORKDIR相似，都是改变环境状态并影响以后的层。WORKDIR是改变工作目录，USER则是改变之后层的执行RUN , CMD以及ENTRYPOINT 这类命令的身份。当然，和 WORKDIR一样，USER只是帮助你切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

HEALTHCHECK健康检查

格式：

HEALTHCHECK [选项] CMD <命令> ：设置检查容器健康状况的命令

HEALTHCHECK NONE ：如果基础镜像有健康检查指令，使用这行可以屏蔽掉其健康检查指令

## Docker网络功能

Docker允许通过外部访问容器或容器互联的方式来提供网络服务。

外部访问容器

容器中可以运行一些网络应用， 要让外部也可以访问这些应用， 可以通过 -P 或-p 参数来指定端口映射。

当使用 -P 标记时， Docker 会随机映射一个 49000~49900 的端口到内部容器开放的网络端口。

使用 docker ps 可以看到， 本地主机的 49155 被映射到了容器的 5000 端口。

此时访问本机的 49155 端口即可访问容器内 web 应用提供的界面。

$ sudo docker run -d -P training/webapp python app.py

$ sudo docker ps -l

CONTAINER ID IMAGE COMMAND CREATED

STATUS PORTS NAMES

bc533791f3f5 training/webapp:latest python app.py 5 seconds ago Up 2 seconds 0.0.0.0:49155->5000/tcp nostalgic\_morse

同样的， 可以通过 docker logs 命令来查看应用的信息。

$ sudo docker logs -f nostalgic\_morse

\* Running on http://0.0.0.0:5000/

10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -

10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1"

404 -

-p（ 小写的） 则可以指定要映射的端口， 并且， 在一个指定端口上只可以绑定一个容器。 支持的格式有 ip:hostPort:containerPort | ip::containerPort |hostPort:containerPort 。

**映射所有接口地址**

使用 hostPort:containerPort 格式本地的 5000 端口映射到容器的 5000 端

口， 可以执行$ sudo docker run -d -p 5000:5000 training/webapp python app.py

此时默认会绑定本地所有接口上的所有地址。

**映射到指定地址的指定端口**

可以使用 ip:hostPort:containerPort 格式指定映射使用一个特定地址， 比如localhost 地址 127.0.0.1

$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py

**映射到指定地址的任意端口**

使用 ip::containerPort 绑定 localhost 的任意端口到容器的 5000 端口， 本地主机会自动分配一个端口。

$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py

还可以使用 udp 标记来指定 udp 端口

$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py

**查看映射端口配置**

使用 docker port 来查看当前映射的端口配置， 也可以查看到绑定的地址

$ docker port nostalgic\_morse 5000 127.0.0.1:49155.

注意：容器有自己的内部网络和 ip 地址（ 使用 docker inspect 可以获取所有的变量， Docker 还可以有一个可变的网络配置。 ）

-p 标记可以多次使用来绑定多个端口

例如

$ sudo docker run -d -p 5000:5000 -p 3000:80 training/webapp python app.py

**容器互联**

容器的连接（ linking） 系统是除了端口映射外， 另一种跟容器中应用交互的方式。该系统会在源和接收容器之间创建一个隧道， 接收容器可以看到源容器指定的信息。

**自定义容器命名**

连接系统依据容器的名称来执行。 因此， 首先需要自定义一个好记的容器命名。虽然当创建容器的时候， 系统默认会分配一个名字。 自定义命名容器有2个好处：自定义的命名， 比较好记， 比如一个web应用容器我们可以给它起名叫web

当要连接其他容器时候， 可以作为一个有用的参考点， 比如连接web容器到db容器

使用 --name 标记可以为容器自定义命名。

$ sudo docker run -d -P --name web training/webapp python app.py

使用 docker ps 来验证设定的命名。

$ sudo docker ps –l

也可以使用 docker inspect 来查看容器的名字

$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde

**容器互联**

使用 --link 参数可以让容器之间安全的进行交互。

下面先创建一个新的数据库容器。

$ sudo docker run -d --name db training/postgres

删除之前创建的 web 容器

$ docker rm -f web

然后创建一个新的 web 容器， 并将它连接到 db 容器

$ sudo docker run -d -P --name web --link db:db training/webapp python app.py

此时， db 容器和 web 容器建立互联关系。

--link 参数的格式为 --link name:alias ， 其中 name 是要链接的容器的名称， alias 是这个连接的别名。

使用 docker ps 来查看容器的连接

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NA MES 349169744e49 training/postgres:latest su postgres -c '/usr Ab

out a minute ago Up About a minute 5432/tcp db, web/db aed84ee21bde training/webapp:latest python app.py 16 hours ago Up 2 minutes 0.0.0.0:49154->5000/tcp web

可以看到自定义命名的容器， db 和 web， db 容器的 names 列有 db 也有web/db。 这表示 web 容器链接到 db 容器， web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道， 而且不用映射它们的端口到宿主主机上。 在启动 db 容器的时候并没有使用 -p 和 -P 标记， 从而避免了暴露数据库端口到外部网络上。

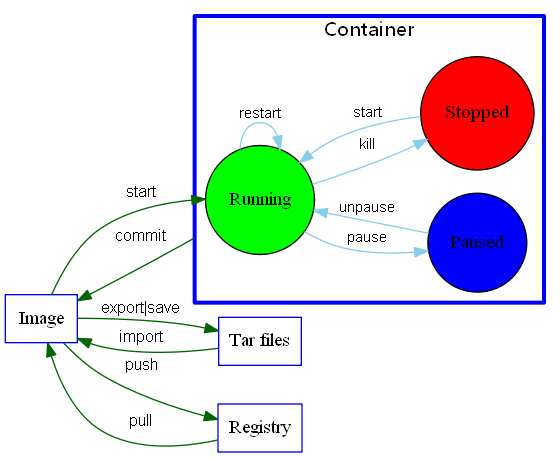
Docker 通过 2 种方式为容器公开连接信息：

环境变量更新 /etc/hosts 文件

使用 env 命令来查看 web 容器的环境变量

$ sudo docker run --rm --name web2 --link db:db training/webapp

## docker命令



### Docker数据管理

Docker内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

* 数据卷（Data volumes）
* 数据卷容器（Data volume containers）

数据卷是一个可供一个或多个容器使用的特殊目录， 它绕过 UFS， 可以提供很多有用的特性：

* 数据卷可以在容器之间共享和重用
* 对数据卷的修改会立马生效
* 对数据卷的更新， 不会影响镜像
* 数据卷默认会一直存在， 即使容器被删除

\*注意： 数据卷的使用， 类似于 Linux 下对目录或文件进行 mount， 镜像中的被指定为挂载点的目录中的文件会隐藏掉， 能显示看的是挂载的数据卷。

**创建一个数据卷**

在用 docker run 命令的时候， 使用 -v 标记来创建一个数据卷并挂载到容器里。 在一次 run 中多次使用可以挂载多个数据卷。

下面创建一个名为 web 的容器， 并加载一个数据卷到容器的 /webapp 目录。

$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py

\*注意： 也可以在 Dockerfile 中使用 VOLUME 来添加一个或者多个新的卷到由该镜像创建的任意容器。

**挂载一个主机目录作为数据卷**

使用 -v 标记也可以指定挂载一个本地主机的目录到容器中去。

$ sudo docker run -d -P --name web -v /tmp/webapp:/opt/webapp training/webapp python app.py

上面的命令加载主机的 / tmp /webapp 目录到容器的 /opt/webapp 目录。 这个功能在进行测试的时候十分方便， 比如用户可以放置一些程序到本地目录中， 来查看容器是否正常工作。 本地目录的路径必须是绝对路径， 如果目录不存在 Docker会自动为你创建它。

\*注意： Dockerfile 中不支持这种用法， 这是因为 Dockerfile 是为了移植和分享用的。 然而， 不同操作系统的路径格式不一样， 所以目前还不能支持。Docker 挂载数据卷的默认权限是读写， 用户也可以通过 :ro 指定为只读。

**挂载一个本地主机文件作为数据卷**

-v 标记也可以从主机挂载单个文件到容器中

$ sudo docker run --rm -it -v ~/.bash\_history:/.bash\_history ubu

ntu /bin/bash

这样就可以记录在容器输入过的命令了

**数据卷容器**

如果你有一些持续更新的数据需要在容器之间共享， 最好创建数据卷容器。

数据卷容器， 其实就是一个正常的容器， 专门用来提供数据卷供其它容器挂载的。

首先， 创建一个名为 dbdata 的数据卷容器：

$ sudo docker run -d -v /dbdata --name dbdata training/postgres echo Data-only container for postgres

然后， 在其他容器中使用 --volumes-from 来挂载 dbdata 容器中的数据卷。

$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres

$ sudo docker run -d --volumes-from dbdata --name db2 training/postgres

可以使用超过一个的 --volumes-from 参数来指定从多个容器挂载不同的数据卷。 也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

$ sudo docker run -d --name db3 --volumes-from db1 training/postgres

\*注意： 使用 --volumes-from 参数所挂载数据卷的容器自己并不需要保持在运行状态。如果删除了挂载的容器（ 包括 dbdata、 db1 和 db2） ， 数据卷并不会被自动删除。如果要删除一个数据卷， 必须在删除最后一个还挂载着它的容器时使用docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。

**利用数据卷容器来备份、 恢复、 迁移数据卷**

可以利用数据卷对其中的数据进行进行备份、 恢复和迁移。

**备份**

首先使用 --volumes-from 标记来创建一个加载 dbdata 容器卷的容器， 并从主

机挂载当前目录到容器的 /backup 目录。 命令如下：

$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

容器启动后， 使用了 tar 命令来将 dbdata 卷备份为容器中 /backup/backup.tar文件， 也就是主机当前目录下的名为 backup.tar 的文件。

**恢复**

如果要恢复数据到一个容器， 首先创建一个带有空数据卷的容器 dbdata2。

$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash

然后创建另一个容器， 挂载 dbdata2 容器卷中的数据卷， 并使用 untar 解压备份文件到挂载的容器卷中。

$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox tar xvf /backup/backup.tar

为了查看/验证恢复的数据， 可以再启动一个容器挂载同样的容器卷来查看

$ sudo docker run --volumes-from dbdata2 busybox /bin/ls /dbdata

### 环境相关

docker info

docker version

docker inspect [images Id/container Id]

查看数据卷的具体信息，在主机里使用以下命令可以查看指定容器的信息

$ docker inspect web

sudo docker inspect --format='{{.NetworkSettings.IPAddress}}' cb84c980357d

### 运维相关

docker attach

进入容器

在使用-d参数时，容器启动后会进入后台。某些时候需要进入容器进行操作，有很多种方法，包括使用docker attach命令或nsenter工具等。

docker build

docker commit

docker cp CONTAINER:PATH HOSTPATH

docker diff CONTAINER

docker export CONTAINER > xxx.tar

**导出容器**

如果要导出本地某个容器， 可以使用 docker export 命令。

$ sudo docker ps -a

$ sudo docker export 7691a814370e > ubuntu.tar

这样将导出容器快照到本地文件。

docker images

docker import

cat xxx.tar | docker import - xxx:xxx

**导入容器快照**

可以使用docker import从容器快照文件中再导入为镜像，例如

$ cat ubuntu.tar | sudo docker import - test/ubuntu:v1.0

$ sudo docker images

docker save IMAGE

docker load

docker CONTAINER

docker port CONTAINER PRIVATE\_PORT

docker pause/pause CONTAINER

docker ps

docker rm

删除容器

可以使用 docker rm 来删除一个处于终止状态的容器。

清理所有处于终止状态的容器

用docker ps -a 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦， 用 docker rm $(docker ps -a -q) 可以全部清理掉。

docker rmi

如果要删除本地的镜像，可以使用 docker rmi 命令，其格式为：docker rmi [选项] <镜像1> [<镜像2> ...]

注意docker rm命令是删除容器，不要混淆。

用ID、镜像名、摘要删除镜像,其中 <镜像> 可以是 镜像短ID、镜像长ID、镜像名或者镜像摘要。也可以用 镜像名 ， 也就是 <仓库名>:<标签> ， 来删除镜像。

镜像的唯一标识是其ID和摘要，而一个镜像可以有多个标签。

用docker images命令来配合

像其它可以承接多个实体的命令一样，可以使用docker images -q 来配合使用docker rmi， 这样可以成批的删除希望删除的镜像。

docker run

**启动容器**

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一个是将在终止状态（ stopped）的容器重新启动。

$ sudo docker run -t -i ubuntu:14.04 /bin/bash

root@af8bae53bdd3:/#

其中，-t选项让Docker分配一个伪终端（ pseudo-tty）并绑定到容器的标准输入上，-i 则让容器的标准输入保持打开。

当利用 docker run来创建容器时，Docker在后台运行的标准操作包括：

* 检查本地是否存在指定的镜像，不存在就从公有仓库下载
* 利用镜像创建并启动一个容器
* 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
* 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
* 从地址池配置一个 ip 地址给容器
* 执行用户指定的应用程序
* 执行完毕后容器被终止

docker start/stop/restart

**启动已终止容器**

可以利用docker start命令，直接将一个已经终止的容器启动运行。

容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。 除此之外， 并没有其它的资源。

**后台(background)运行**

更多的时候，需要让Docker在后台运行而不是直接把执行命令的结果输出在当前

宿主机下。此时，可以通过添加 -d参数来实现。

**终止容器**

可以使用docker stop来终止一个运行中的容器。此外，当Docker容器中指定的应用终结时，容器也自动终止。启动了一个终端的容器， 用户通过exit命令或Ctrl+d来退出终端时，所创建的容器立刻终止。终止状态的容器可以用 docker ps -a命令看到

docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

docker top CONTAINER [ps OPTIONS]

docker wait CONTAINER [CONTAINER…]

### 日志相关

docker events [OPTIONS]

docker history [OPTIONS] IMAGE

### Hub服务相关

访问仓库

仓库（ Repository）是集中存放镜像的地方。

一个容易混淆的概念是注册服务器（ Registry）。实际上注册服务器是管理仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像。从这方面来说，仓库可以被认为是一个具体的项目或目录。例如对于仓库地址dl.dockerpool.com/ubuntu来说， dl.dockerpool.com 是注册服务器地址，ubuntu是仓库名。

私有仓库

有时候使用 Docker Hub 这样的公共仓库可能不方便， 用户可以创建一个本地仓库供私人使用。

docker-registry 是官方提供的工具，可以用于构建私有的镜像仓库。

docker login [OPTIONS] [SERVER]

docker logout

docker search

docker pull/push

## 数据卷

数据卷是一个可供一个或多个容器使用的特殊目录， 它绕过 UFS， 可以提供很多有用的特性：

* 数据卷可以在容器之间共享和重用
* 对数据卷的修改会立马生效
* 对数据卷的更新， 不会影响镜像
* 数据卷默认会一直存在， 即使容器被删除

\*注意： 数据卷的使用， 类似于Linux下对目录或文件进行mount， 镜像中的被指定为挂载点的目录中的文件会隐藏掉， 能显示看的是挂载的数据卷。

创建一个数据卷

在用docker run命令的时候， 使用 -v 标记来创建一个数据卷并挂载到容器里。在一次run中多次使用可以挂载多个数据卷。下面创建一个名为web的容器，并加载一个数据卷到容器的/webapp目录。

$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py

\*注意： 也可以在 Dockerfile 中使用 VOLUME 来添加一个或者多个新的卷到由该镜像创建的任意容器。

挂载一个主机目录作为数据卷

使用 -v 标记也可以指定挂载一个本地主机的目录到容器中去。

$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py

上面的命令加载主机的 /src/webapp 目录到容器的 /opt/webapp 目录。 这个功能在进行测试的时候十分方便， 比如用户可以放置一些程序到本地目录中， 来查看容器是否正常工作。 本地目录的路径必须是绝对路径， 如果目录不存在 Docker会自动为你创建它。

\*注意：Dockerfile 中不支持这种用法，这是因为Dockerfile是为了移植和分享用的。然而，不同操作系统的路径格式不一样， 所以目前还不能支持。Docker 挂载数据卷的默认权限是读写，用户也可以通过 :ro 指定为只读。

挂载一个本地主机文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中

$ sudo docker run --rm -it -v ~/.bash\_history:/.bash\_history ubuntu /bin/bash

这样就可以记录在容器输入过的命令了。

数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

## 参考文档

参考文档

Dockerfie 官方文档： <https://docs.docker.com/engine/reference/builder/>

Dockerfile 最佳实践文档： <https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/>

Docker 的安装 <https://itbilu.com/linux/docker/NkbHFYXDM.html>

Docker专题 <http://udn.yyuap.com/portal.php?mod=topic&topicid=5>

Docker源码分析

<http://udn.yyuap.com/forum.php?mod=viewthread&tid=28085&from=portal>

<https://github.com/hangyan/docker-resources/blob/master/README_zh.md>

<https://github.com/liuzhen9327/Docker-Tutorials>

<https://github.com/y123456yz/reading-and-annotate-docker-17.05.0>

<https://docs.docker-cn.com/>