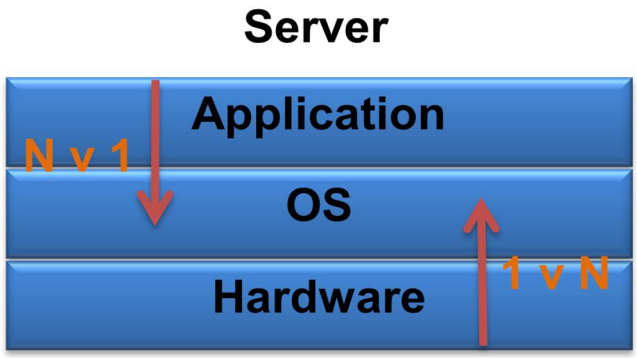
从半空看虚拟化

题解

打个比方，云计算是天上的云，虚拟化技术就是地上的湖泊水塘。以前也整理过一些对虚拟化和云计算技术的学习理解，感觉那时候就是站在云端，最多算是从云层中探个头出来瞅瞅下面，对虚拟化技术的理解似是而非。现在较之前有了一定进步，但也仍处在半空之中，期望有天真能落到水面上扑腾两下，但看来这辈子希望不大。在此向那些真正的泳者们致敬。

引言

书归正传，服务器虚拟化技术粗略的可分为一虚多和多虚一两大类。从下面这张图理下这两类技术的方向，注意这里提到的 Server 是针对云计算的 X86 服务器系统。

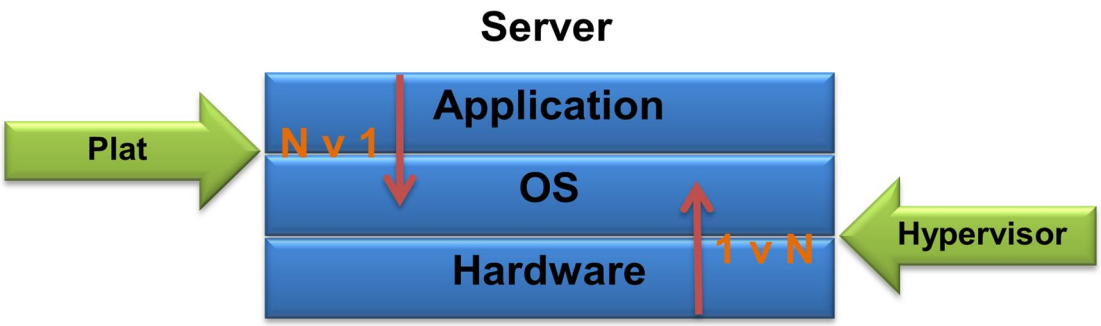


云计算的根本是服务器，服务器可简单分为硬件资源、操作系统和应用程序三个层面。一虚多技术主要面对硬件层面与操作系统层面，总的来说就是在一台物理服务器上虚拟出多套逻辑资源来运行多套操作系统。需注意前半句物理资源虚拟化只是手段不是重点，真正的目的是要搞多套能够同时运行且互不干扰的虚拟操作系统出来。这块技术是当前云计算 IaaS 的基础，也是本文后面要介绍的重点部分。

而多虚一技术主要针对应用程序层面，操作系统层面也有如 Cluster 和 Load Balance 等技术手段，但其重点是要在应用层面上将底层资源整合，达到对外统一服务的目的。上亿的用户在使用搜索引擎或浏览新闻的时候，实际的任务是由后端成千上万的服务器完成，但对我们这些使用者来说，看到的就是那几个应用页面。

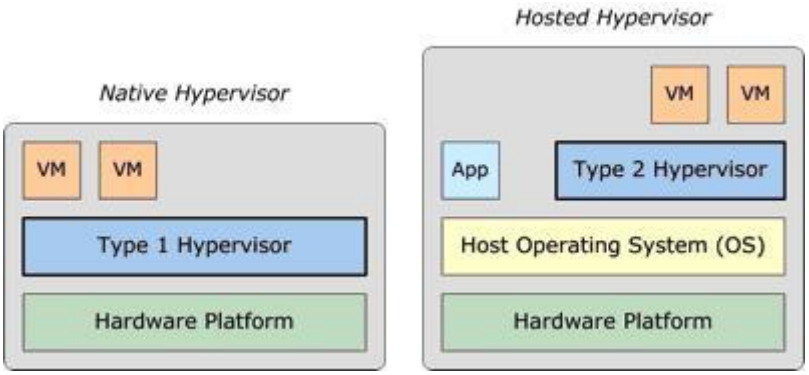
当我们想要在尽量少改动甚至不改动原有对象的条件下，对整体结构做调整时，可以在其中间插入新的一个对象，专门来处理结构调整与任务调度。虚拟化技术很好的阐述了

这种设计思路，如下图中 Plat 位置对应的 Hadoop 技术和 Hypervisor 位置对应的 VMware ESX/ESXi、Xen、Linux KVM 以及 Microsoft Hyper-V 等技术。



相对来说，底层的一虚多技术更加收敛一些，上层的多虚一技术则要更加开放，毕竟主流的硬件和操作系统就那么几种，而应用软件数不胜数。因此从简单的入手，本文主要分析的是基于硬件到操作系统层面的一虚多技术，对应云计算中的 IaaS 部分。

传统上有两种类型的 Hypervisor 应用如下图所示，本文主要针对的是 Type1，但个人理解这两种类型从技术上讲区别并不是很大，Hypervisor 不可能脱离 OS（Operating System 操作系统）独立去进行硬件的处理， Type1 中的 Hypervisor 也包含了一个最简单的 OS 内核，无非就是在这个内核上不能跑其他的应用罢了。因此可以将 Type1 的 Hypervisor 理解为 Mini OS+VMM(Virtual Machine Manager)，而 Type2 的 Hypervisor 理解为单独的 VMM。最新的主流 Hypervisor 产品之一 KVM 已经打破了这种简单划分，我们无法将其明确的定位为 Type1 或 Type2，因此以后对虚拟化技术应该会有更加先进准确的分类方式出现。后面的文章中会再详细的分析当前主流 Hypervisor 产品中 OS 与 VMM 之间的关系架构。



正文以介绍服务器相关虚拟化技术为主，按照硬件、Hypervisor 和 OS 三个层面自底向上展开，其中会针对典型的技术与产品进行分析讨论。

本文的主要目的是帮助那些在云上或者云中的同学们往下降一降。但毕竟作者现在也还是在半空中，因此很多内容无法讲细讲透，甚至可能会有一些理解误区，仅供参考，欢迎指正。

虚拟化技术介绍

虚拟化管理平台

讲虚拟化技术之前，先提两句虚拟化管理平台。

当通过虚拟化技术将物理资源模拟为逻辑资源后，还需要一个平台将其统一管理起来，并通过接口提供给终端用户使用，这个管理平台也是 IaaS 必不可少的另一支柱。从用户角度出发，申请虚拟机时最基本的四个要素是 CPU、内存、硬盘和带宽，其中 CPU 和内存代表了虚拟机的计算能力，属于单台硬件服务器一虚多分离出来的子集（你不可能要求一个 2 核的虚拟机使用的是两台物理服务器上各自分出来的一个核，同样也不可能同时使用多台物理服务上的物理内存来构建单台虚拟机的虚拟内存）；硬盘则代表数据存储能力，这个可以通过 NAS/SAN 等技术连接到磁盘阵列上来无限扩充；带宽则是网络服务能力的表现，同时网络还需要为不同租户的虚拟机之间提供隔离控制能力，为相同租户的虚拟机之间提供通道互访能力，而这些虚拟机可能在相同或不同的物理服务器上。

虚拟化管理平台并不具备什么先进的技术内容，但是其在推动云计算发展进程中占据着重要地位。如果说虚拟化技术是树根，则虚拟化管理平台可以看作树干，云计算这棵大树想要枝繁叶茂则二者缺一不可。虚拟化管理平台通过对硬件资源进行统筹管理，形成统一的资源池供用户使用。用户可以定制并管理自己的虚拟机和软件程序，不需要经过管理员和设备厂商的中间传递，其需求能够更加准确简便的得到实现。

再多废话两句，以 Apple 的例子来说明一下平台为王的现状，Apple Phone 的具体技术其实都算不上先进，无论是触屏还是模块化应用都是很早就出现的技术，真正帮助其王者归来的是 Apple Store。通过这种新的市场模式，手机成为了一个平台，人们看重的不再是其硬件功能本身，而是聚焦于其上的应用程序。在传统的手机制造销售中，程序的开发者与实际用户之间隔着厂商这堵墙，对开发者而言厂商让编什么就编什么，而使用者则只能是厂商给什么就用什么。一款手机产品的成功与否只能依赖于厂商对用户使用喜好的猜测和硬件与价格等低级竞争手段。Apple Store 则打破了那堵墙，用户可以直面开发者去选择喜爱的程序，开发者也可以根据下载量和排行榜更准确的摸清用户喜好，从而进行有针对性的开发，并直接获益。从目前来看唯一会对 Apple 造成发展制约的就是其硬件了，当满世界人们都举着 Apple Phone 时，审美疲劳会导致大家倾向于选择更新鲜更与众不同的款式。如果 Apple 将来的 IOS 可安装于任意手机硬件之上，那么现在

如日中天的 Apple 帝国也将如 Microsoft 般延续出属于自己的不朽王朝。另外，今日的 IOS 与 Android 之争与 90 年代的 Windows 与 Linux 之争是多么相似，同样从那场纷争可以看出，IOS 今后的主要对手不会是 Android，而只有 Microsoft 的 WP。

硬件层面虚拟化

首先来说硬件，一虚多的根本目的就是把一台物理服务器虚拟成多台虚拟服务器来使用。就服务器的 CPU、内存、存储和网卡等核心物理资源来说，真正的虚拟化难点在于 CPU。因此下面以 CPU 虚拟化技术介绍为主，还会简单说下北桥芯片和网卡相关的虚拟化技术。

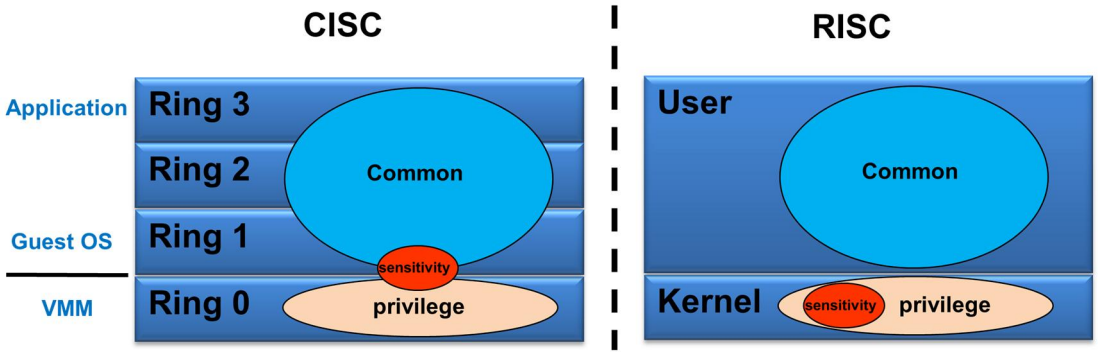
CPU 虚拟化技术

现代计算机的 CPU 技术有个核心特点，就是指令分级运行，这样做的目的是为了避免用户应用程序层面的错误导致整个系统的崩溃。需要注意这里“指令分级”中的级别专门指 CPU 指令运行级别，而和我们操作系统里面的进程运行优先级没有关系。不同类型的 CPU 会分成不同的级别，如 IBM PowerPC 和 SUN SPARC 分为 Core 与 User 两个级别，MIPS 多了个 Supervisor 共三个级别。本文针对的 X86 系统则分为 Ring0-Ring3 共 4 个级别，而在这里我们不需要考虑那么多，只需关注核心级别（Ring0）和用户级别（Ring1-Ring3）两个层面即可。

对于非虚拟化的普通操作系统而言，应用程序和系统下发的普通指令都运行于如 Ring1 到 Ring3 的用户级别中，只有特权指令会运行在如 Ring0 的核心级别中。而当进行虚拟化之后，实质上出现了两层操作系统，底层 OS（VMM 或 Hypervisor，下文都以 VMM 简称）负责虚拟机（VM）到硬件资源的调用和协调，所有的业务应用都运行在上层 OS（Guest OS）中。

在非 X86 系统中经典的虚拟化做法是令 VMM 拥有超级特权，其指令都运行在类似 Ring0 的核心级别；令 Guest OS 运行在类似 Ring1 到 Ring3 的用户级别，取消其特权指令在核心级别直接运行的权利。当 Guest OS 运行到特权指令时，这些指令产生异常并被 VMM 捕获到，VMM 会在核心级别中模拟执行，然后再将运行结果返回给 Guest OS。这种经典的虚拟化技术被称为特权解除和陷入模拟，以 IBM 的 Power 系列为代表。

我们将操作系统中会涉及系统底层公共资源调用的一些运行指令称为敏感指令，显然这些敏感指令在虚拟化的结构中也需要陷入到 Ring0 的核心级别执行，否则会导致不同 Guest OS 之间的资源调用冲突。大型服务器如 PowerPC 和 SPARC 运行的 RISC 指令集中，所有的敏感指令都属于特权指令，因此可以采用上面说的特权解除和陷入模拟技术完美的进行虚拟化实现。但对于 X86 的 CISC 指令集而言，存在 17 条非特权指令的敏感指令，这些指令被 Guest OS 在 Ring1 级别执行时，会被直接执行，无法产生异常从而陷入 Ring0 处理，也就导致无法采用经典技术进行虚拟化，因此下文将介绍的一系列方案都是为了解决此问题而设计的。



在上述问题中，涉及到三个主要对象，Guest OS、VMM 和硬件 CPU 的指令集，其中 VMM 是新插入的对象，修改起来很方便，但 OS 和 CPU 改起来就难一些了。解决方案的思路也由此分为三个方向：

1、 只变动 VMM。好处是兼容性最强，OS 和 CPU 都不用动，但效率肯定是最低的。这种方案也被称为 CPU Full-Virtualization。
2、 改动 Guest OS。好处是效率较高，但缺点是 Windows 肯定不愿意干，只能在 Linux 上做些文章，而且使用特制的 OS，会带来一些可扩展性方面的隐患。这种方案也被称为 CPU Para-Virtualization。
3、 改动 CPU 指令集。这个改动就只有 Intel/AMD 能做了，好处是对 Guest OS 可以不需变动，兼容 Linux 和 Windows，VMM 的使用效率也较高。缺点也有，就是增加了一些虚拟化指令和结构，导致对 CPU 的利用率下降，在部分应用场景下的性能表现不如前面的 CPU Para-Virtualization 方案。这种方案也被称为硬件辅助虚拟化技术 HVM（Hardware-assisted Virtualization Machine）。随着 Intel/AMD 的服务器 CPU 全部更新换代对其提供支持，HVM 已经成为当前虚拟化技术应用的主流。

CPU Full-Virtualization

首先说 CPU Full-Virtualization，这种思路又被细化分为三种主要方案 Emulation、Scan-and-Patch 和 Binary Translation。其中 Emulation 是根本解决方案，而 Scan-and-Patch 和 Binary Translation 可以理解为是 Emulation 在 X86 体系上使用的扩展实现方案。CPU Full-Virtualization 由于实现较为简单，早在上世纪末就已经出现，是最早期的 X86 虚拟化技术。

基本的 Emulation 主要应用在跨平台进行虚拟化模拟，Guest OS 与底层系统平台不同，尤其是指令集区别很大的场景，比如在 X86 系统上模拟 PowerPC 或 ARM 系统。其主要思路就是 VMM 将 Guest OS 指令进行读取，模拟出此指令的执行效果返回，周而复始，逐条执行，不区分用户指令和敏感指令，由于每条指令都被通过模拟陷入到 Ring0 了，因此也就可以解决之前的敏感指令问题。代表产品就是 Linux 上的 QEMU 和 Bochs，其中 QEMU 可以通过开源加速器软件 KQEMU 来提升处理速度。它们作为开发工具，在嵌入式平台的开发工作中应用较多。

Scan-and-Patch 主要思路是将 Guest OS 的每个指令段在执行前先扫描一遍，找出敏感指令，在 VMM 中生成对应的补丁指令，同时将敏感指令替换为跳转指令，指向生成的补丁指令。这样当指令段执行到此跳转时会由 VMM 运行补丁指令模拟出结果返回给 Guest OS，然后再顺序继续执行。代表产品是 Oracle 的开源虚拟化系统 VirtualBox，目前主要应用于在主机上进行虚拟机的模拟，服务器使用较少。

Binary Translation 主要思路是将 Guest OS 的指令段在执行之前进行整段翻译，将其中的敏感指令替换为 Ring0 中执行的对应特权指令，然后在执行的同时翻译下一节指令段，交叉处理。代表产品为 VMware Workstation、Microsoft Virtual PC 主机虚拟化工具，以及早期 VMware 的 ESX/GSX 系列服务器虚拟化系统，目前的服务器上已经很少使用了。

CPU Full-Virtualization 受性能影响，在服务器上目前被逐渐淘汰。主要代表产品如 VirtualBox 和 VMware Workstation 大都应用于主机虚拟化的一些开发测试环境中。有个了解即可，可不必对其技术进行深究。只有 QEMU 作为基础虚拟化技术工具，在其他的虚拟化产品中被广泛实用。

CPU Para-Virtualization

Para-Virtualization 有半虚拟化、类虚拟化和超虚拟化等多种译法，这也可以看出人们对此技术褒贬不一的矛盾心理。此技术以 Xen 和 Hyper-V 为代表，但 VMware 的 ESX Server 和 Linux 的 KVM 两种当前主流虚拟化产品也都支持 Para-Virtualization，另外还有基于 Linux 的 Lguest 和基于 Windows 的 CoLinux 等各具特色的小型产品也都通过部分 Para-Virtualization 技术来实现操作系统的虚拟化。Para-Virtualization 技术实际上是一类技术总称，下面先要谈的是 CPU 的 Para-Virtualization（CPU PV）。

CPU PV 技术实现的主要原理如下，首先 VMM 公布其一些称为 Hypercalls 的接口函数出来，然后在 Guest OS 中增加根据这些接口函数修改内核中的代码以替代有问题的 17 条敏感指令执行系统调用操作。修改后的指令调用通常被称为 Hypercalls，Guest OS 可以通过 Hypercalls 直接调用 VMM 进行系统指令执行，相比较前面提到的陷入模拟方式极大的提升了处理效率。

然而 CPU PV 修改操作系统内核代码的方式带来了对 Guest OS 的很多使用限制，如只有 Hyper-V 可以支持 Para-Virtualization 方式的 Windows Server 作为 Guest OS，另外由于 KVM/Xen/VMware VMI/Hyper-V 各自 Hypercalls 代码进入 Linux 内核版本不同，因此采用 Linux 作为 Guest OS 时也必须关注各个发行版的 Linux 内核版本情况。KVM 是 2.6.20，VMI 是 2.6.22，Xen 是 2.6.23，Hyper-V 是 2.6.32。

CPU PV 方式由于对 Guest OS 的限制，应用范围并不很广，但由于其技术上的系统调用效率提升，仍然被部分开发与使用者所看好，在某些特定场景中也存在一定需求。另外由于 2002 年 Xen 既有采用 CPU PV 的产品面世，目前部分大型的数据中心服务器已经进行了部署，基于技术成熟度和利旧的考虑，短时间内仍然会拥有一定的生存竞争能力。

硬件辅助虚拟化技术 HVM

前面的 Full-Virtualization 和 Para-Virtualization 两种方案是在 VMM 和 Guest OS 的软件层面解决 CPU 的敏感指令问题，而最后这个方案则是从硬件出发，由 CPU 自己搞定。当前 X86 的 CPU 厂商就是 Intel 和 AMD 两家，个人觉得从虚拟化技术发展上来看，AMD 更多的属于跟随者的角色。Intel 推出的 CPU 虚拟化技术是在 Xeon CPU 上的 VT-x，和 Itanium CPU 上的 VT-i，AMD 的类似技术则是 AMD-V。下面以 VT-x 为例对 HVM 技术进行剖析，其他两种类似也就不多说了。

VT-x 在 CPU 操作方面有两个主要特性，一是增加了 VMCS（Virtual-Machine Control Structure）数据结构和 13 条专门针对 VM 的处理指令，用于提升 VM 切换时的处理效率，二是通过引入 Root/Non-Root 操作模式解决前面遇到的 Guest OS 敏感指令无法陷入问题。

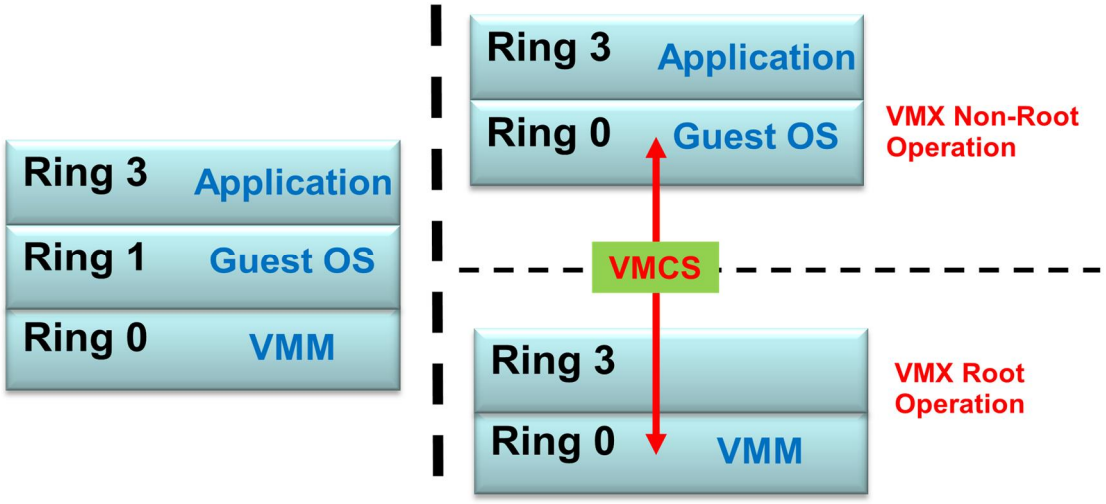
13 条新指令中包括 5 条用于 VMCS 维护，4 于 VMX 管理，2 条用于 VMX 对 TLB（Translation Look aside Buffer）的管理，2 条用于 Guest OS 调用。详见下表，说的啥大家应该也都能看明白，犯下懒就不翻译了。

Behavior	Instruction	Description
VMCS-maintenance	VMPTRLD	This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the

		current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.
	VMPTRST	This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
	VMCLEAR	This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to “clear”, renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
	VMREAD	This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.
	VMWRITE	This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.
VMX management	VMLAUNCH	This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
	VMRESUME	This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
	VMXOFF	This instruction causes the processor to leave VMX operation.
	VMXON	This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.
VMX-specific	INVEPT	This instruction invalidates entries in the TLBs and paging-structure caches that were derived from extended page tables (EPT).

TLB-management	INVVPID	This instruction invalidates entries in the TLBs and paging-structure caches based on a Virtual-Processor Identifier (VPID).
Guest-available	VMCALL	This instruction allows software in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
	VMFUNC	This instruction allows software in VMX non-root operation to invoke a VM function (processor functionality enabled and configured by software in VMX root operation) without a VM exit.

Root/Non-Root 操作模式将原有的 CPU 操作区分为 VMM 所在的 Root 操作与 VM 的 Non-Root 操作，每个操作都拥有 Ring0-Ring3 的所有级别。通过 VMCS 数据结构指向 Guest OS 到 VMM 的操作切换。



VMM 在初始化的时候会在物理内存中为每个 VM 开辟一段空间用于存储对应的一个 VMCS 数据结构，可以理解一个 VMCS 就对应一个 vCPU。VM 切换时，VMM 可以通过 VMCS 指针方便的在不同 VMCS 间进行跳转，有效的提升了 CPU 使用效率。

VT-x 技术还包含以下三个重要内容：

1、 EPT（Extended Page Table）：由 VMM 控制的一种新页表结构，主要目的为了在保证多个 VM 访问物理内存相互隔离的同时，提升大块内存的读写效率。

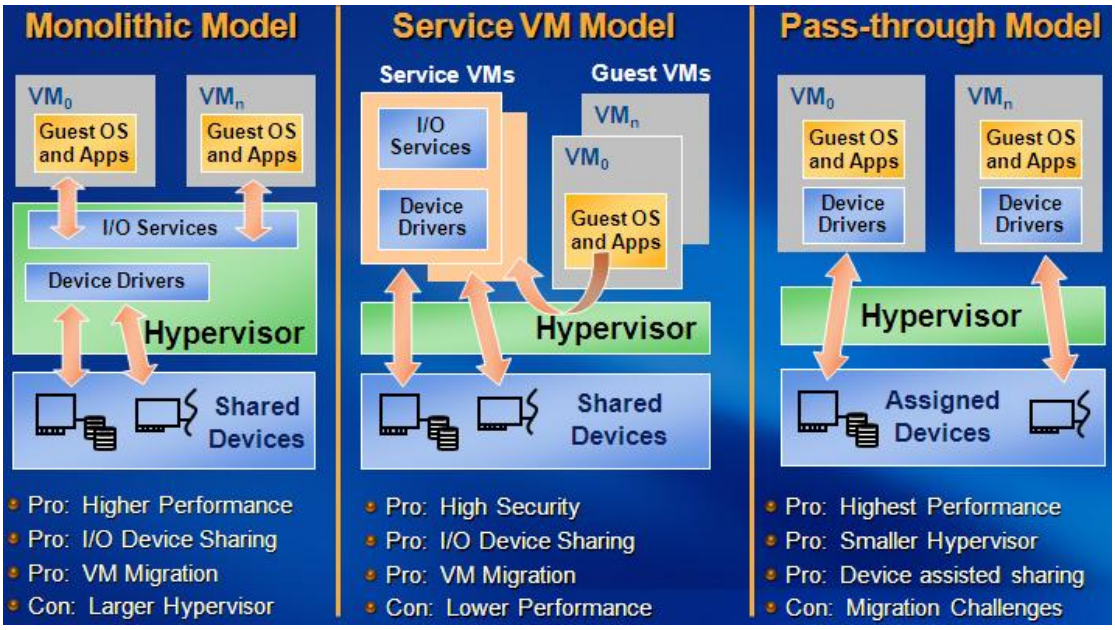
2、 FlexMigration：灵活迁移技术，用于 VM 在使用不同型号 CPU 的物理服务器间进行迁移，促成了类似 vMotion 的 VM 迁移技术的发展。

3、 FlexPriority：灵活优先级技术，用于优化 CPU 中断处理过程中对 TPR（Task-Priority Register）的使用，提高中断处理效率。

VT-x 和 AMD-V 等技术的出现，解决了前面两种纯软件方案进行 X86 虚拟化时，CPU Full-Virtualization 性能低和 Para-Virtualization 的 Guest OS 兼容性差问题。随着服务器 CPU 两三年一换代的更新速度，当前的主流 X86 服务器已经都可以支持 VT-x/AMD-V 等技术，因此 HVM 成为当前云计算 IaaS 服务器虚拟化的主流。主要的几款 VMM 产品 Xen/VMware ESXi/KVM/Hyper-V 都已经能够支持 HVM 功能。

北桥芯片 Chipset 虚拟化技术

北桥在服务器中主要是用来连接 CPU、内存、AGP 和南桥的各种 I/O 设备，处理 DMA 与中断请求，北桥芯片的虚拟化技术是为了支持 I/O 虚拟化技术。先介绍下常见的 I/O 虚拟化方案，有以下三种。



第一种将 I/O 服务和设备驱动都直接装载在 Hypervisor 的 Mini OS 上，优点是性能较高，缺点是 Hypervisor 会变得很大，代表产品就是 VMware 的 ESX 和 ESXi，最新的 ESXi 5.0 安装包简化后也达到 290M，另外一个缺点就是对新设备的驱动加载不够方便，有时需要升级整个 Hypervisor，比如 ESXi 直到 5.0 版本才能识别出 Qlogic 驱动的 CNA 网卡。如果希望采用此种方案，一定要先确认好 Hypervisor 的支持驱动列表再进行安装。

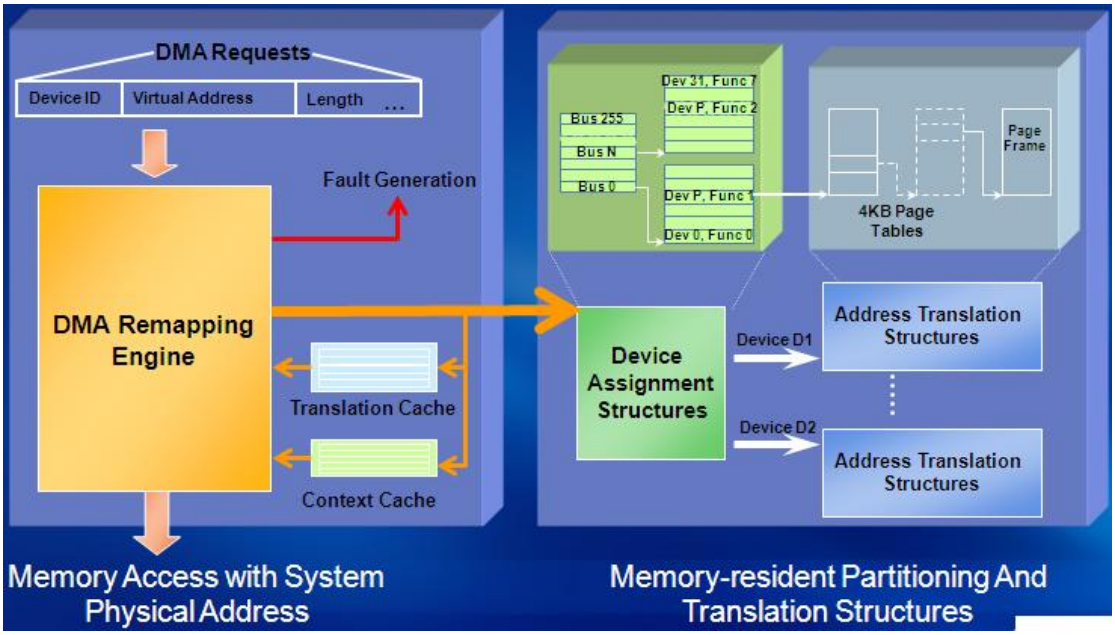
第二种是通过加载一个服务系统（如 Domain0）来减小 Hypervisor 的负担，所有的产品驱动都安装在服务系统上，这样便于其他外设硬件的管理和增加变化，同时 Hypervisor 越小意味着越安全，降低了故障的风险。缺点是资源调用需要经过服务系统，性能会稍低。典型代表就是 Xen 和 Hyper-V。KVM 有些类似，但不完全一样，因为其 Hypervisor 和服务系统使用了同一套 Linux 内核，而 Xen/Hyper-V 的 Hypervisor 各使用了一套最简化的新内核，与服务系统不同，下文会有更清楚的介绍。

另外这里还有个概念需要说明，前面两种方式在一些公开资料中也被称为 Full-Virtualization（FV）和 Para-Virtualization（PV），但个人认为这个是 I/O 的虚拟化，而 FV 和 PV 的更主要区别是在于 CPU 的虚拟化技术。CPU 的 FV/PV 和 I/O 的 FV/PV 并不是紧密联系的，例如早期的 VMware ESX 使用 CPU FV + I/O FV，早期的 Xen 使用 CPU PV + I/O PV，但在 HVM 技术出现之后，其各自主流结构已经变成 VMware ESXi 使用 VT-x + I/O FV，Xen 使用 VT-x + I/O PV 了，所以目前的 Xen 也都没有必须使用修改过内核的 Guest OS 的限制了。

第三种则是从硬件层面已经将各个设备进行了划分，分配给不同的 VM 使用，由每个 VM 自行维护其使用的 I/O 和设备驱动。Hypervisor 只提供通道，不再对 I/O 和设备驱动进行管理。这样做的好处是可以达到最高的性能和最小的代码量（Hypervisor），但缺点是设备非完全共享且对 VM 迁移提出了一定挑战。这种方案只有在对 I/O 操作要求较高的特定场景下使用，不具备普适性。部署时 I/O 设备支持逻辑划分技术如 SR IOV（Single Root I/O Virtualization）等即可，对 Hypervisor 没有什么特殊要求，前面说的几款主流产品都提供了此种工作方式。

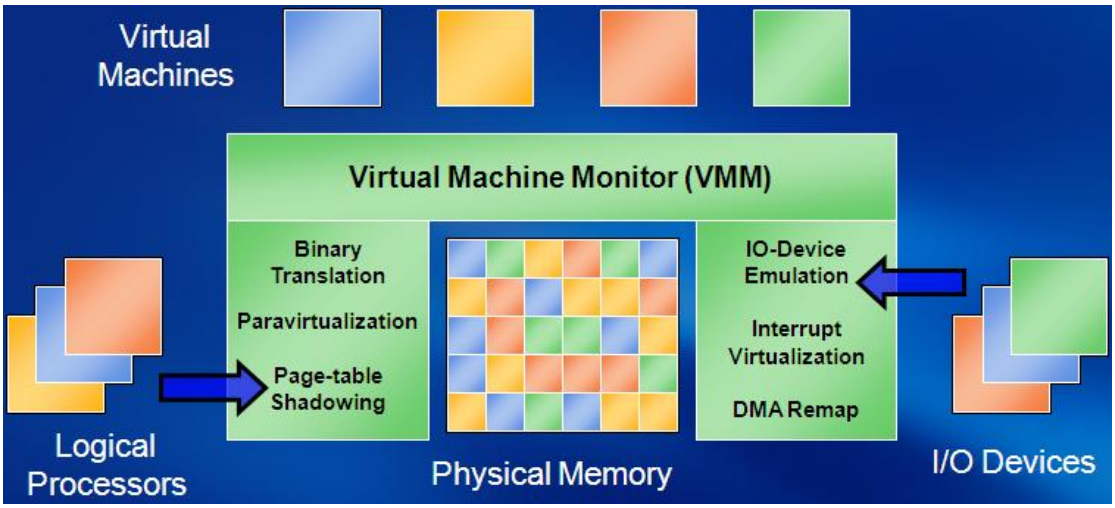
Intel 的 VT-d 技术作为一种平台结构在硬件层面可以有效的对上面三种方案全部提供支持。其根本目的就是为了多个 VM 在对虚拟设备进行 I/O 操作时，能够对其 DMA 和中断处理进行隔离保护与提升性能。

VT-d 的主要内容就是 DMA Remapping，用下面这张网上摘来的截图帮助有兴趣的同学进行理解，作者水平有限，细节就不解释了。

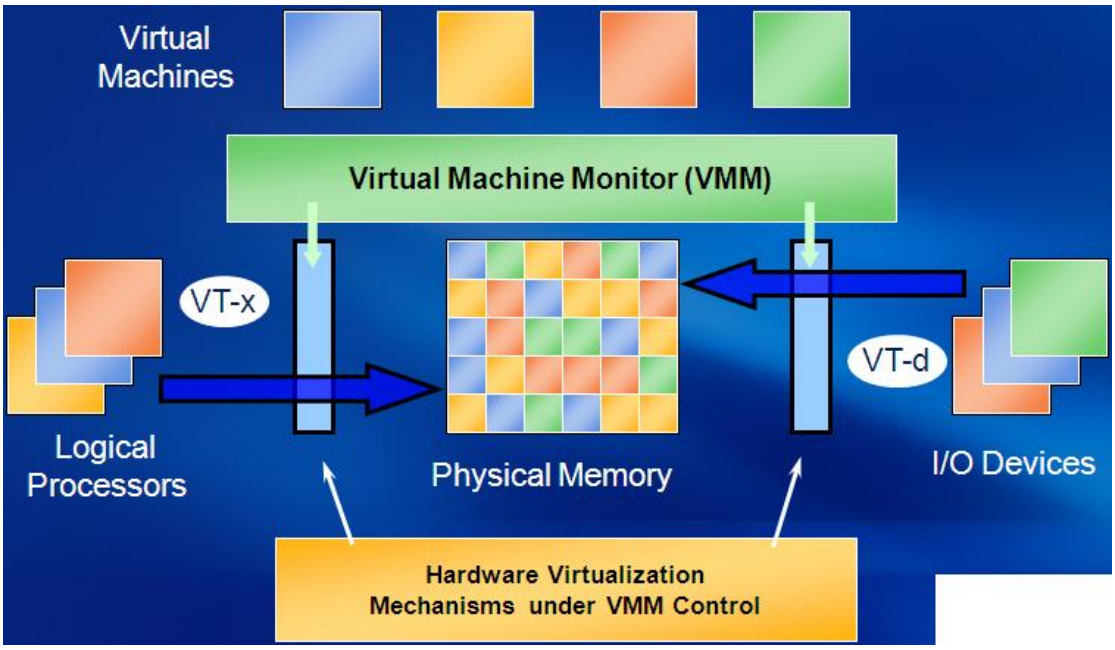


VT-x 和 VT-d 等技术内容很多，细说起来可以整理很多内容，本文深度有限，就不做详细探讨了，最后再用两张图来比较 VT-x 和 VT-d 出现前后的虚拟化结构区别。

软件虚拟化结构



HVM 结构



（上述四幅截图均来自于 Microsoft WinHEC 2006 大会宣讲胶片资料）

网卡虚拟化技术

服务器内部如 CPU、内存、芯片等设备虚拟化后都是希望让不同的 VM 能够达到资源共享，使用隔离的效果，但网络方面就有所区别了，除隔离外还得考虑一个 VM 互联互通的问题。

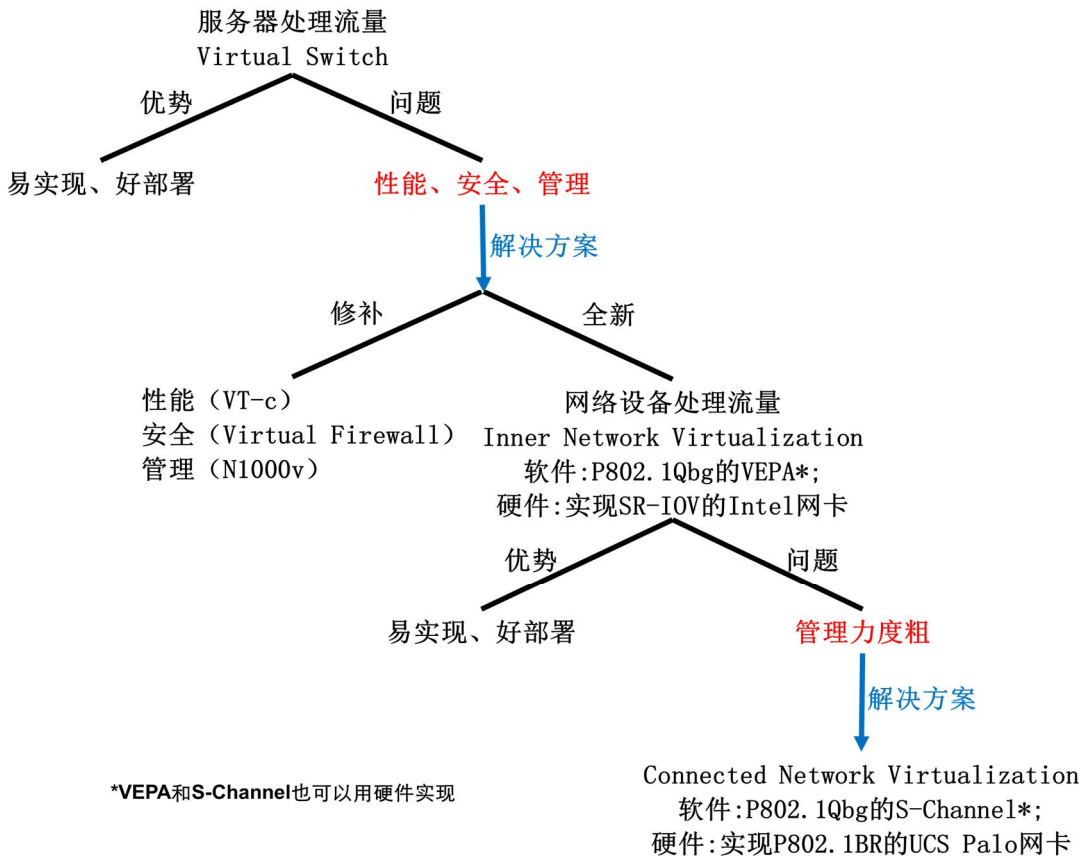
每个 VM 发出的流量会有两种去向，类型一流量是去往物理服务器外面的世界，这部分流量好办，按照其他设备的思路进行虚拟化隔离使用，给每个 VM 一个专用通道，共用物理网卡资源即可。VM 还有可能发出访问本物理服务器内部其他 VM 的类型二流量，

这部分流量目前的主流处理方式是在 Hypervisor 建立一个 Virtual Switch，内部软件交换，不需要走到服务器外面去。另外一个将要实现的思路是物理服务器内部仍然按照流量类型一的方式去处理，将流量隔离后都扔到服务器外面，由外界的网络设备确认流量终点位置，如果目的地址属于同一物理服务器的其他 VM，则通过相同的路径返回给服务器即可。

Virtual Switch 思路有两个主要问题，一是性能很差，二是对 VM 来说类型一与类型二流量通常是会同时存在的，而 Virtual Switch 在处理类型二流量的时候会存在较大的安全性问题。但由于其易于实现，因而 Virtual Switch 成为了当前使用的主流技术，几款主要的 Hypervisor 都有自己的 Virtual Switch，再如 IEEE P802.1Qbg 定义了 VEB（Virtual Ethernet Bridge），Cisco 也推出独立 Virtual Switch 产品 N1000v。同时为了解决前面所说的性能与安全问题，其他厂商也陆续做出了一些新的东西对其修修补补，像 Intel 的 VT-c 使用 VMDq（Virtual Machine Device Queues）技术通过网卡芯片硬件处理一些 Virtual Switch 的工作，这样可以有效提高 VM 数据交换性能，部分安全厂商则推出了 Virtual Firewall 产品来提高 Hypervisor 内部层面的数据转发安全性。

网络设备处理的 VM 互访流量思路从服务器来看也分为两种处理方式，一是只在服务器内部对 VM 的流量进行区分识别，从物理网卡扔出去时就是普通报文，外部网络设备根据 MAC/IP 再查表转发，不了解服务器内部的任何 VM 网络信息，作者给起个名字叫做 Inner Network Virtualization（INV）。其技术代表，软件的是 P802.1Qbg 中的 VEPA（Virtual Ethernet Port Aggregation），硬件的是 PCI-SIG 组织推的 SR-IOV（Single Root I/O Virtualization）标准。而第二种处理方式就是服务器将 VM 发到外面的流量都加个 Tag，外部网络设备根据此 Tag 识别来自不同 VM（的 vNIC）的流量进行细化处理。作者再给起个名字叫做 Connected Network Virtualization（CNV）。（只为了描述方便，作者可不是什么命名控）其技术代表，软件的是 P802.1Qbg 的 S-Channel（Multichannel），硬件的就是 Cisco UCS 服务器中 Palo 网卡支持的 P802.1BR 技术。由于 CNV 明显是 INV 的扩展，也可以说是为了解决 INV 控制力度太粗的问题而出的方案，所以从技术产品上来看，S-Channel 是 VEPA 的超集，而 Palo 网卡实现 P802.1BR 时也用到了 SR-IOV 的内容。

通过下面的图表，希望能够帮助读者更清晰的理解上述文字内容。



有些跑题，真正和本章主题“网卡硬件虚拟化技术”相关的内容其实主要就是 VT-c 和 SR-IOV 两项，一个是 Intel 实现的，一个是 Intel 主导的，但研究不深这里也不再展开了，有兴趣的同学请自行查阅相关资料吧。

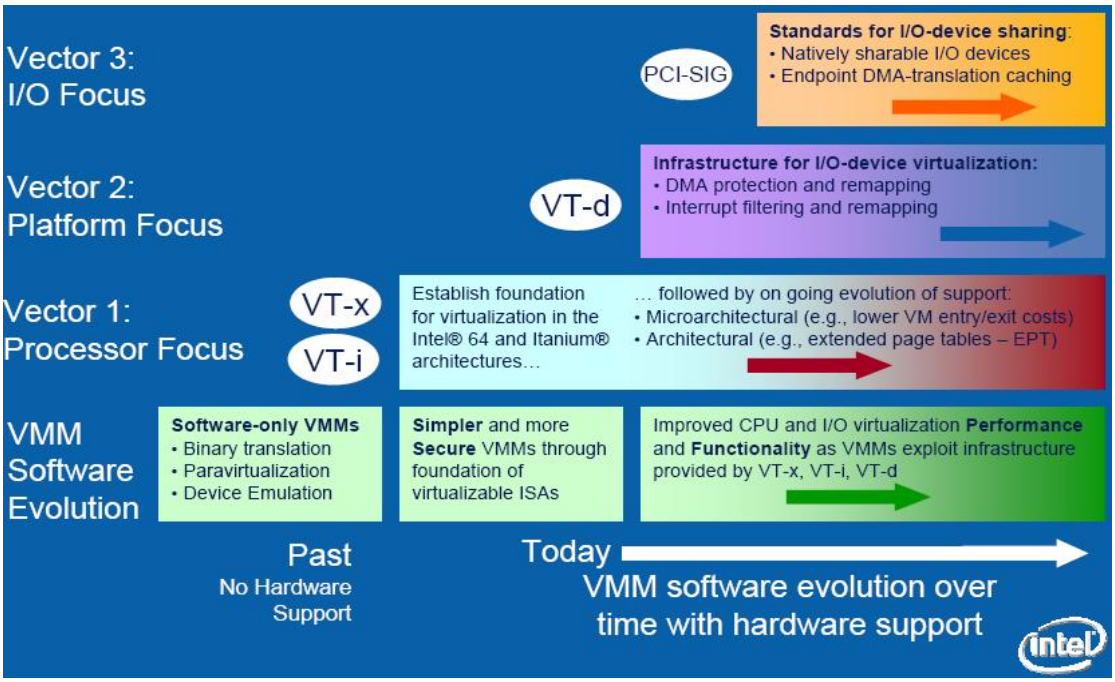
硬件虚拟化技术小结

前面章节从服务器硬件的角度对服务器一虚多技术进行了一些分析，其中两个关键方向就是 CPU 和 I/O 的虚拟化。CPU 虚拟化经历了早期的 CPU FV/PV 软件虚拟化技术发展，目前已经进入了硬件辅助技术 HVM 为主的阶段。而 I/O 虚拟化目前仍然由 I/O FV/PV 的软件虚拟化技术主导，SR IOV 等硬件辅助技术刚刚起步，存在很多使用上的局限，但预计在今后几年内随着技术的发展完善，也会逐步替代软件方式成为技术主流。

这种技术发展过程其实在网络技术方面更加常见，一个新的数据传输特性做出来后往往是先由软件实现，然后用各种方式优化改进效率，但最终总是由于性能需求推动，通过 ASIC、Fabric 或 NP 等硬件手段来实现大规模商用。

X86 系统 HVM 的最大推手和赢家就是 Intel，当然对 DELL、HP 等服务器集成商，Microsoft、Redhat 等系统厂商，VMware、Citrix 等 VMM 厂商以及 Google、Amazon 等云计算服务提供商来说，做大虚拟化这块蛋糕符合大家的利益，属于共赢的局面，因此可以看到所

有人都对 HVM 技术拍手称好。下面以一张来自 Intel 的胶片截图将其几个主要技术汇总作为本段的结尾。



Hypervisor 虚拟化

前面从硬件的角度来分析了虚拟化技术的构成，下面来看看 Hypervisor，主要针对 Type1 Hypervisor，介绍几款主流产品的构成与区别。

开源与社区

首先来说几句题外话。查看国外软件的时候，经常会看到 XX 软件开源、XX 开源软件由 XX 社区维护，XX 项目隶属于 XX 基金会等内容。感觉整个软件业的生态环境和国内有巨大差别，有必要先介绍一些背景。

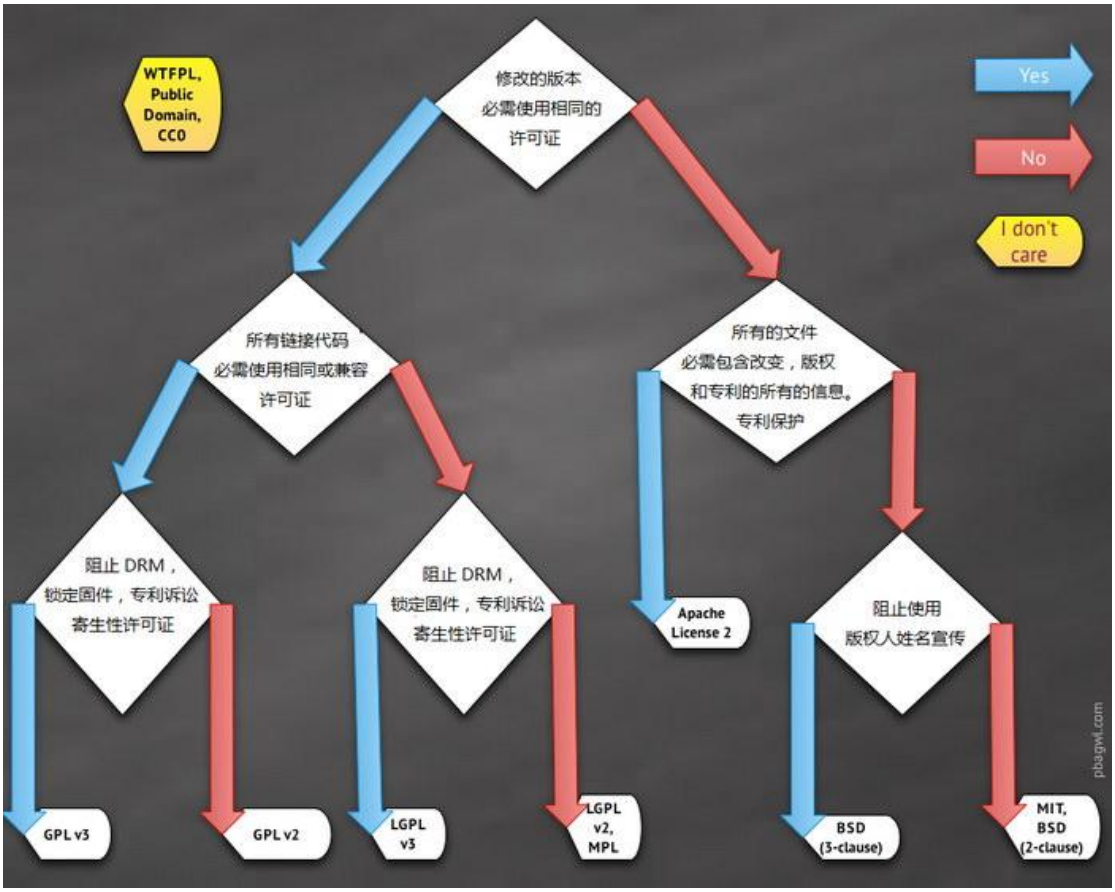
开源指开放源码 Open Source，最早是在 1998 年 2 月份美国加州 Palo Alto 一个会议上由 Christine Peterson 提出的。早先的自由软件中 Free 一词同时拥有自由和免费的含混意思，容易使人产生误解，而 Open 一词可以更好的对自由软件进行阐释。为了改善自由软件的生态环境，Open Source 通过制定完善的开发使用规范使其更容易被个人与企业所接受，更具有生命力。 同年 2 月底开源计划组织 OSI(Open Source Initiative)成立，对开源理念推广提供支持，至今此组织仍然是开源软件的中坚力量。

开源对作者来说主要是为了使软件能够得到更广阔的发展前景，同时通过更多的人参与进来对源码进行修改补充，也可以使作品更加完善。开源作为一种软件开发方式已经成为大势所趋，可以说当前的软件行业没有哪家公司会站出来说和开源一点儿关系都没有。从编程语言 JAVA、C、PHP 到操作系统 Linux、Android；从 HTTP 服务器 Apache 到数据库 MySQL；从浏览器 Firefox 到 Blog 系统 Wordpress；从多虚一分布式技术 Hadoop 到后面要介绍的一虚多 Hypervisor 系统 Xen 和 KVM，这些都是开源软件中大家耳熟能详的重量级产品。

开源软件最大的特点就是制定了很多规范用来确定使用者的权利和义务。这些规范一般都会被放置在代码的最前面或单独成章使人一目了然，我们通常称其为软件使用许可证。在 GNU 网站上记录了几乎所有的开源软件许可证，达到上百个之多，但我们平时主要会用到的就是 BSD（Berkeley Software Distribution）、MIT（Massachusetts Institute of Technology）、Apache、GPL（GNU General Public License）和 LGPL（GNU Lesser General Public License）几个。其中 BSD/MIT 和 Apache 相对宽松一些，重点要求保证作者对原始程序的版权，要求每个修改版本中都必须保留原始程序的作者信息。GPL 是当前应用最广的许可证，由 FSF（Free Software Foundation）发布，其赋予接受者对软件程序自由使用、修改、复制和发布改进的权利，并通过 Copyleft（针对 Copyright）规则，要求修改继承遵循 GPL 的软件得到的所有演绎版本都仍然必须遵循 GPL。鼎鼎大名的 Linux 就是使用 GPL，帮助我们可以免费使用所有版本的 Linux OS。当然企业和个人在遵循 GPL 开放源码和免费使用的基础上，仍可以对服务进行适当收费，这也是当前很多 Linux 厂商的存活之道。LGPL 对 GPL 进行了一定放宽，当发布程序对遵循 LGPL 的软件进行调用、

连接而非包含时，允许封闭源码。LGPL 最早是针对类库进行的设计，因此早期也曾被称为 Library General Public License。

对于主要开源许可证的使用选择可以通过下面这张网上截图来查看。



其中 WTFPL（What The Fuck Public License）具体内容如下，从名字就能看出其 NB 之处。

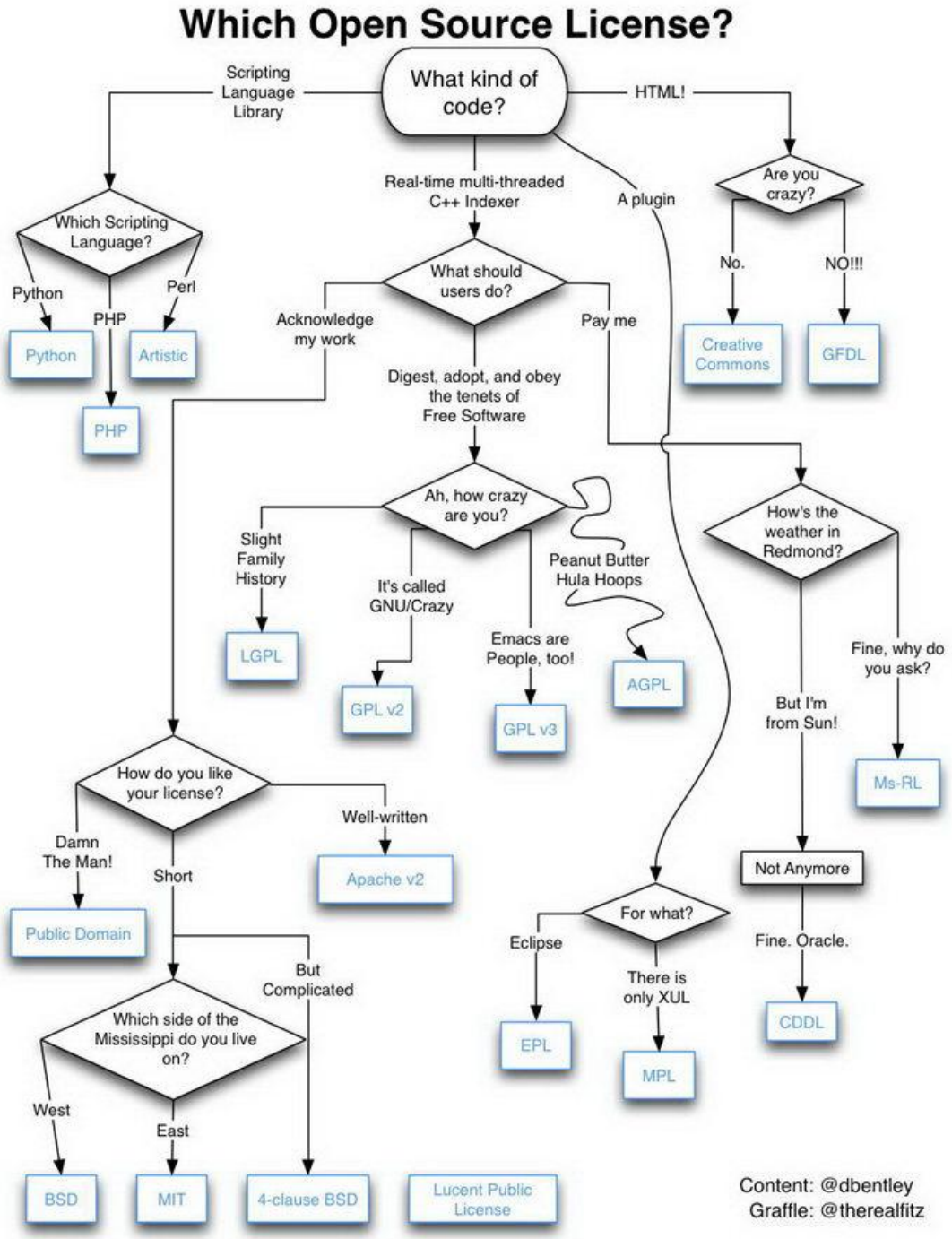
```
DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE

                    Version 2, December 2004


 Copyright (C) 2004 Sam Hocevar <sam@hocevar.net>


 Everyone is permitted to copy and distribute verbatim or modified copies
of this license document, and changing it is allowed as long as the name
is changed.
```

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. You just DO WHAT THE FUCK YOU WANT TO.

另外还有张截图更详细也更调侃一些。此处关于开源软件的理解描述大部分源自于开源中国社区（www.oschina.net），有兴趣的同学可以自行登陆查阅详细内容。



关于开源软件再补充下其与免费软件的关系，开源软件不一定就免费，而免费软件也不一定会开源，一切均取决于开发者的意愿并通过许可证 License 进行控制。

那么开源软件如何发布和管理呢，这就要说到社区了。社区有多种多样的组织形式，其主要作用有两个，一是作为代码仓库为开发者们提供修改程序管理版本的平台，二是作为发布窗口为使用者们提供获取软件程序及源码的平台。

从服务的开源软件对象类型可以将社区简单分为三大类。首先是服务无数小型软件的社区，此类社区为开源软件提供最基本的版本管理与发布平台，典型代表就是SourceForge.net，也被称为SF.net。其作为全球最大的开源软件开发平台和代码仓库，为近30万软件项目和几百万注册用户提供服务。SourceForge既包含如Wiki百科使用的Media Wiki这种知名开源程序，也包含了大量个人开发及目前已经停止开发的软件项目。类似的社区还有采用分布式版本控制系统Git的GitHub和采用Mercurial的BitBucket；由Microsoft维护专注于Windows周边程序的CodePlex；以及由Google提供服务资源的Google code（最初专注于Google产品及其周边的开源软件，目前已经成为更加开放的软件托管平台，代表软件为Android）。

第二类社区专门服务于某个大型开源软件，典型代表如Linux、Mozilla、OpenBSD、OpenOffice、Eclipse、Xen和OpenStack等社区。此类社区还要对其服务的开源软件进行推广、维护和提供法律保护等工作，比第一类社区做得更多，也需要更大的投入，往往会以基金会的形式运作以确保持续运营的能力。

第三类社区为多个大型开源软件项目提供管理运维服务，对项目运作及版本发布使用进行更严格的监管和控制。如果将第一类社区比作大型集市，将第二类社区比作专卖店，那么第三类社区就是精品百货商场了。典型代表目前看来好像也就ASF(Apache Software Foundation)一家。ASF是由第二类社区发展来的，最初只是针对开源软件Apache HTTP Server的维护，后来在启动并行项目研究时，子项目由于发展得很好就被提升为新的顶级项目，同时很多优秀的项目也加入到ASF，由ASF统一管理，将其项目家族打造成今天的庞然大物。现今ASF拥有93个直接管理的顶级项目，同时还托管着如OpenOffice等其他开源社区。其知名顶级项目有Apache HTTP Server、Tomcat、Ant、Struts、Lucene、Nutch、Logging、Geronimo、XML、Perl、Hadoop、Hbase等。在自建维护社区有顾虑的情况下，将项目托管给ASF管理和运作已经成为大型开源软件的主要选择之一。ASF管理的开源项目全都遵循许可证Apache License2.0。

最后再简单介绍下这些社区的运作资金来源。支出方面虽然部分工作人员可以是自愿的投入，但是必要的损耗仍然是避免不了的，如服务器、网络等物理资源及法律支持和日常维护的人员开销。从收入方面来看主要是三个方面，首先是广告收入，通过网站上的广告位出售获取资金，这块是小头。然后是项目收入，通过特定License授权开源软件给第三方企业进行商业化修改包装销售来获取部分收入，ASF就允许其管理项目以此方式受益，但明显这种行为只能适合于部分易于包装为产品的项目，而且个人觉得与开源软件的初衷有悖。最后一项实际上开源软件的最大头收入是来自于募捐，也就是我们看到社区经常被命名为XX基金会，其背后经常会看到一个或多个主要的金主，当然这些财神们也会对开源软件的发展产生一些影响和利益纠葛，置于背后的博弈就不是我这篇文章里面要分析的了。

关于基金会募捐再多提两个有趣的案例。一个是 2006 年时，OpenBSD 的管理人员向部分厂商发出了对 OpenSSH 产品项目的捐款请求，用于 OpenSSH 的开发和维护工作。这些厂商在其自身产品中都集成和使用了 OpenSSH，并因此产生了部分获益，对象包括 HP、IBM、Redhat 和 Cisco 等大公司。最终的募捐结果没有找到，但据说收到的最快回应是来自于 Mozilla 基金会捐出的 1 万美元，因为 FireFox 浏览器中也使用 OpenSSH 的部分功能。另一个案例也与 Mozilla 有关，由于开源基金会大多定位于公益性的非盈利组织，在 M 国是免税的。而 2007 财年发布的公告中，Mozilla 从 Google 收入了一笔 6600 万美元，来源于其在 FireFox 浏览器中默认集成的 Google 搜索引擎。此收入已经占到了 Mozilla 当年收入总和的 88%，因此被 M 国政府税务局进行审查，考虑要免除其非盈利组织的的免税待遇，并要求为 2007 年补税 10 万美元。几经波折之后，Mozilla 还是最终逃过了一劫。

上述两个案例均来自于网上资料，作者没有任何的主观倾向性与褒贬意图，只是摘录来和读者分享，如引起任何人的不适，请望谅解，如有不正确之处请知会作者，会立即进行修改删除。

Hypervisor 与操作系统内核

回归正题，继续讲技术，当前有四种主流的 Hypervisor 产品，VMware ESXi、Xen、KVM 和 Hyper-V。其中 Hyper-V 未开源， VMware 部分开源（从 VMware 网站上可以自由下载到包括 ESX/ESXi/vCenter 等所有发售产品的部分源码），Xen 和 KVM 则完全开源。下面将从操作系统内核的角度分析下这几款产品的差异。

操作系统与内核

先介绍一些背景技术。从批处理到分时处理，到实时处理再到分布式处理，从单进程到多进程处理，推动操作系统出现与发展的根本需求都来源于对多工作任务的排序处理。其核心作用就是在不同应用程序运行操作硬件资源时，进行统一安排调度。

在最早的计算机时代，根本没有操作系统的事，应用程序直接去调用操作硬件进行任务处理，一个任务结束，后面人再上去安装运行新的任务。随着时间的发展和硬件性能的提升，大家觉的可以安排些自动化的工作，就不用操作人员天天站在设备前面排队了。于是最早的操作系统诞生，其只需要进行任务分时调度即可，反正所有的工作都是排队顺序执行。此时的应用和硬件也就那么几种，仍然由应用程序编写时通过代码直接操作硬件就够了。但是随着硬件和应用的发展，到了今天估计没有人能数清计算机到底包含多少种应用程序和多少种硬件组成单元（包括不同品牌），在应用程序中直接集成硬件操作指令也就成为不可能完成的任务。因此现代的操作系统就必须提供任务调度外的另一项必要功能，对应用程序代码和硬件操作命令之间进行转换，简单来说就是在应用程序和硬件之间做一个翻译器，帮助两边进行交流。

由于服务对象众多，这种翻译往往不是一步到位的，首先需要由各个硬件设备厂商提供驱动 Driver，将最基本的硬件操作指令翻译给操作系统，然后操作系统需要将这些操作指令抽象为应用程序可以理解的高级对象（如文件系统等），最后由编译接口 API（如 C 库等）将这些抽象的对象元素提供给应用程序调用。我们通常将执行硬件抽象动作的部分称为操作系统的内核 Kernel，同时内核还要负责任务调度排序处理的执行，因而其成为了不同操作系统之间的根本区别之处。

当然完整的操作系统还必须提供人机交互通道（命令行或者图形方式）以及一些其他的系统服务。虚拟化技术应用场景中，监控管理 VM 运行的 VMM 可以理解为一个应用程序，也必须通过内核才能去操作硬件，因此前面也说过 Type1 的 Hypervisor 可以理解为 Mini OS+VMM。程序设计的原则永远是简单为美，这个 Mini OS 更需要小而稳定，必要的最简人机交互通道和系统服务大家能做的都差不多，而 VMM 需要提供的基本功能也同样类似，那么 Hypervisor 真正的差异化设计思路就剩下内核了。

内核分类与 Hypervisor

内核最基本的任务有以下几项：

1、 进程管理：对应用程序运行时在内存中存放的代码执行进行管理。比如执行的先后顺序，硬件操作冲突了怎么办（多核或多进程情况下），应用程序间互相通信调用怎么办（IPC）等等。说白了都是由最开始的多任务批处理扩充演变而来的。

2、 内存管理：由于进程（应用程序）的代码都被放到内存里面执行，那么对内存的地址分配就需要统一管理了，需要将物理内存的空间抽象为虚拟内存元素（如 Page 或 Segment）方便应用程序使用。另外既然虚拟化了就可以将硬盘上的空间以链接形式扩充为虚拟内存，提供更高的内存的可用性（如休眠时将内存都拷贝到硬盘上用于快速恢复）。需要注意，由于内核也是一段代码要在内存中运行，因此通常将所有的虚拟内存地址划分为专门跑内核代码的内核空间（Kernel Space）和应用程序代码的用户空间（User Space），应用程序不能直接访问和操作内核空间干扰内核代码的运行已经成为当前内核设计的基本准则。下文的内核分类会涉及到这两个代码空间的划分。

3、 I/O 设备管理：就是前面讲的硬件抽象等设备管理工作。是内核最重要的工作之一。

4、 系统调用：操作系统通过 C 库等 API 提供给应用程序的一组专用代码，用于改变 CPU 模式，执行访问设备和内存的一系列基础动作。包含 close、open、read、write 和 wait。

操作系统内核目前有很多分类，最根本的是单内核与微内核类别，其他的都可以看作是由这两种内核设计思路衍生而来。单内核 Monolithic kernel，内核的所有功能代码都运行在同一个内核空间内，包括模块化设计，所有的模块也是运行于同一空间。此种方式设计起来会比较复杂，任何一个功能模块的问题都会导致整个系统的崩溃，当然优势是性能较高。典型代表是 Linux 内核。

四大 Hypervisor 中的 KVM 就是直接使用 Linux 内核，可以作为程序安装在任何一个开放的 Linux OS 上，更像传统中 Type2 类型的 Hypervisor，底层的开放 Linux OS 上可以安装任何驱动和应用程序，灵活但导致系统整体稳定性稍差。下文还会对 KVM 进行进一步分析。

另一类被称为微内核 Microkernel，其设计思路是类似 Client-Server 的分层模式，只有上述四项内核任务中最基本的部分代码会运行于内核空间作为 Server（注意四项任务一个都不能少），非关键部分和其他的操作系统相关内容（如 Network，File System 等）都运行于用户空间中作为 Client。这样的好处是任何一个 Client 有问题，都能够避免整个 OS 的崩溃。从理论上来说，虽然内核空间内的代码尽量简化，能够远小于单内核设计，但由于处于不同空间，服务调用时就需要更多的指针从而导致更大的内核代码总量，性能效率也会变低（更多次上下文切换）。Mach 和 QNU 是微内核的两个典型代表之作，目前更多的被应用于对稳定性有更高要求的一些特定领域中，如航天和医疗等。单内核与微内核孰优孰劣的争执由来已久，本文不做深入讨论。

微内核作为一种通过层次化将基本内核功能尽量简化的思路，实际上没有明确界定哪些功能代码就一定应该是最基本的，需要放在内核空间中，而哪些代码就要作为 Client 放在用户空间中。因此有了第三类内核，处于单内核与微内核之间，对设计难易程度、工作效率与稳定性的妥协产物，混合内核 Hybrid kernel。此类内核相比较微内核将更多的东西放入了内核空间中，但仍选择了一些非关键代码放置于用户空间中。由于归类划分完全由设计者自行定义，因此混合内核在封闭操作系统中很受欢迎，如鼎鼎大名的 Windows 和 Mac OS X。

VMware 的官方公开资料较少的提及其 Hypervisor ESX/ESXi 的具体内核技术结构，从种种蛛丝马迹中，作者主观的推断其应属于微内核或混合内核结构（这两者并没有明确区分界定），其微内核 VMkernel 有很大可能是从斯坦福大学的 SimOS 变化而来。下文也会对其进行更详细的分析。

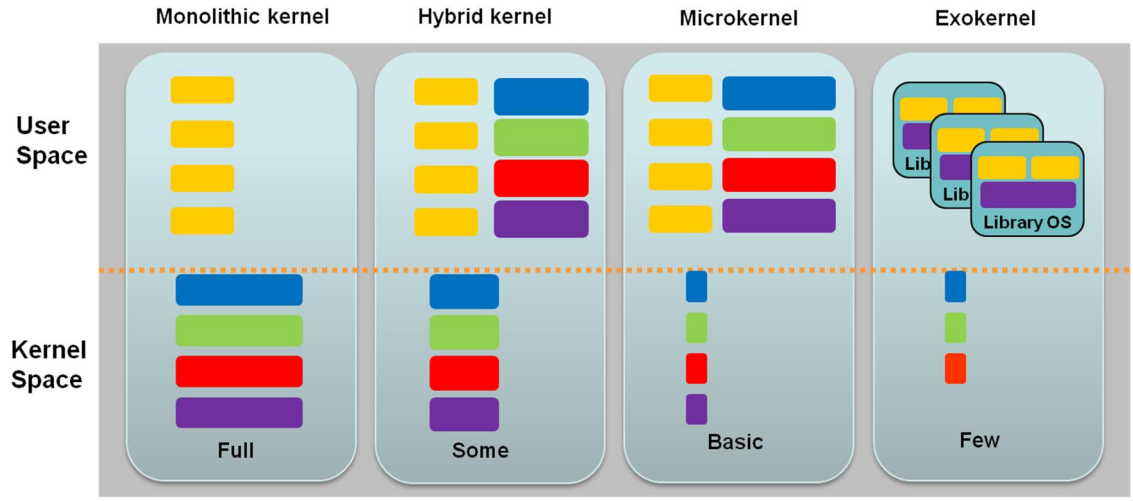
除了上述主流内核外，还有其他的很多类型，如微微内核 Nano kernel：将设备管理的中断和计时器等基本的控制工作都扔给设备驱动程序去做，以达到内核空间代码比微内核更简的目的；巨内核 Megalithic Kernel：将所有的操作系统相关内容甚至包括应用软件程序都写到内核空间中，以达到最快的运行效率目的，当然这也意味着安装修改软件时就只能重编整个内核了。

这里还有一款需要重点介绍的内核被称为外内核 Exokernel（也叫垂直结构操作系统），比微内核更加极端一些，外内核将硬件抽象工作也从内核空间中移除，丢回给用户空间的应用程序去完成，只确保应用程序访问资源时，资源是空闲可用的。当然为了处理硬件抽象工作以及维护设备驱动和 API，外内核往往需要另外一个 Library 操作系统进行硬件设备管理，而且这个 Library OS 的内核代码需要有所修改，以便与 Exokernel 协作处理任务。

感觉 Exokernel 的思路有些回归到计算机操作系统原始状态的意思，以处理任务为主，不负责维护硬件操作。这样做有三个明显的好处：一是内核空间代码量更小内核更稳定；二是应用程序可以直接做硬件抽象工作，软件开发人员对硬件使用拥有更高的自由度和处理效率；三是外内核之上理论上讲可以同时运行多个不同类型的 Library 操作系统如 Linux 和 Windows 进行基本的硬件抽象和管理工作，同时通过不同的 API 供不同的类型的应用程序使用。换而言之，我们可以做出个巨大的操作系统，这个操作系统以很小的 Exokernel 在内核空间内负责协调工作，然后在用户空间内运行一大堆各种操作系统作为其 Library OS，并跑上基于各种操作系统 API 的应用程序，如 Apache 的 WEB Server 和 Windows 的 SQL Server 等等。有人该说了，这不就是理想的服务器虚拟化么。Bingo，准确的说这个也许是将来的虚拟化结构，但是 Exokernel 现在还是处于试验阶段的技术，没有办法十全十美的实现上述模型。另外真这么搞的话需要 Library OS 都得改内核以适应 Exokernel，也是个不大不小的麻烦。典型的 Exokernel 如 University of Cambridge 的 Nemesis 和 MIT 的 ExOS。

OK，理想的虚拟化现在还实现不了，那么我们先搞搞折中的办法，Library OS 不好直接用来跑应用，可以先弄一个出来专门做做硬件设备管理，再通过类似 QEMU 等模拟器走主流路线去虚拟其他的 Guest OS 来运行应用程序。这就是 Xen 和 Hyper-V 的思路，由此我们看到其 Hypervisor 必须和一个叫做 Domain 0 或者 Parent Partition 的 Library OS 一起运行为 Guest OS 提供虚拟化服务。又因为这个 Library OS 必须是内核增加了相关代码的 Modified OS，导致一直迟迟未能进入 Linux 内核的 Xen 与 Redhat 等 Linux 集成商们之间不断演绎着悲欢离合，而 Microsoft 干脆将 Hyper-V 与改了内核的 Server 2008 捆在一起安装发售。下文还将详细介绍这两款主要的 Hypervisor 产品。

通过下面这张图再将前面四种主要内核的关系汇总一下。



OS	Linux	Windows	Mach	Nemesis
Hypervisor	KVM	ESX/ESXi		XEN Hyper-V

从代码量来说，这四种内核的内核空间代码是逐渐减少的，但就整个内核的代码总量而言，通常由大到小的顺序是 Hybrid kernel、Microkernel 和 Monolithic kernel，而 Exokernel 由于必须与 Library OS 共生，因此不好进行排位。

主流 Hypervisor 产品介绍

按照发布时间的早晚，下文依次详细介绍 VMware ESX/ESXi、Xen、KVM 和 Hyper-V 技术。

VMware ESX/ESXi

首先分析 VMware 的 ESX/ESXi，无论公司还是产品都是当前 Hypervisor 界当之无愧的重量级元老。先八卦一下 VMware 的精彩历史。

1998 年 VMware 在 M 国的 Palo Alto, California 由五位高人创建，其中的关键人物 Mendel Rosenblum 是 Stanford 的副教授，当时 SimOS 项目的主负责人；Edouard Bugnion 是他的在读博士学生，当时 SimOS 项目的主要研究员；Diane Greene 是 Rosenblum 的妻子（两人相识于 University of California, Berkeley）。

1999 年 5 月，VMware 发布其第一款产品 VMware Workstation。2001 年发布 GSX Server（Type2）和 ESX Server（Type1）两款 Hypervisor 产品正式进入服务器虚拟化市场。2003 年 VMware 发布 Virtual Center（后来的 vCenter）产品，集成 vMotion 和 VSMP 等技术。这些产品一直发展到现在已经成为虚拟化技术的典型代表。

2004 年，VMware 以$625 million 被 EMC 收购，截止目前仍以其全资子公司的形式独立运作。

2005 年 Bugnion 从 VMware 的 CTO 职位上辞职，参与创建了 Nuova System，此公司 2008 年被 Cisco 收购，而后这个哥们在 Cisco 一直干到 VP 的职位，于 2011 年辞职回 Stanford 继续完成其博士学业去了。

2006 年 6 月，VMware 收购了私有公司 Akimbi Systems，开始其成长为 IT 大鳄的吞噬之旅。

2007 年 8 月，EMC 将 VMware 的 10%股份拿出来在纽交所上市，发行当天从 29$上涨到收市时的 51$。

2008 年 5 月，VMware 收购了以色列的 Start-up 公司 B-Hive Network，并基于其场地设备和人力等资源在以色列建立了第一个海外研发中心。

2008 年 7 月，VMware 创始人之一 Greene 在 CEO 的位子上被董事会突然辞退， EMC 云计算部门主管 Paul Maritz，一位曾在微软工作 14 年的老兵取而代之。

2008 年 9 月 10 日，另一位重要创始人 Rosenblum，Greene 的丈夫，也从 VMware 首席科学家的职位上辞职，与妻子一同回归校园，专心于研究事业。上述人员变动导致 VMware 当时股价剧烈波动，市值骤降接近 25%。

紧跟着 2008 年 9 月 16 号，VMware 宣布与 Cisco 在数据中心市场进行深度合作，将其 N1000v 虚拟软件交换机集成到 VMware 架构中作为可选产品一同销售。

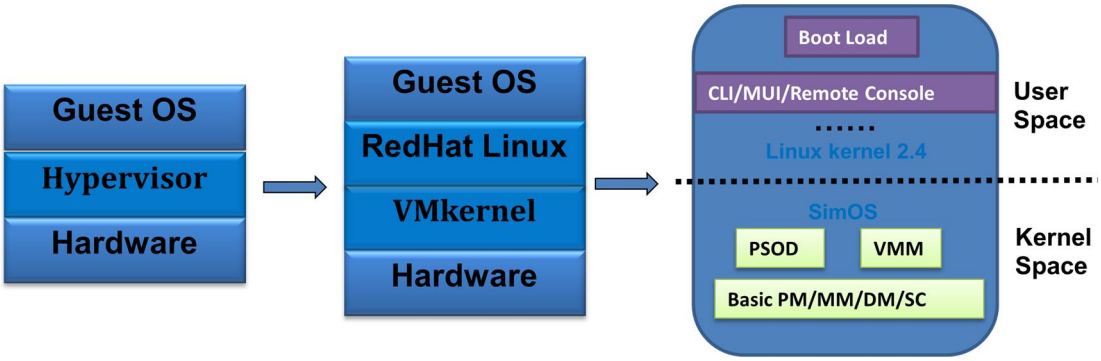
随后几年内，VMware 陆续收购了多家 PaaS 相关软件公司，并与 2011 年 4 月推出自己的 PaaS 开源服务平台系统 Cloud Foundry。

紧跟着在 2011 年 4、5、6 三个月，VMware 又连续出手收购了 3 家 SaaS 相关软件公司，其下一步动作昭然若揭。。。

从 VMware 的发展史中，看到了又一个典型的硅谷公司成长案例，和早年的 Cisco 起家颇有些相似之处。技术与资本之间错综复杂的关系总是使人滋味无穷，估计再过个几年高科技企业创业的题材应该会掀起一阵影视热潮，诸位有志于向编剧导演发面发展的同志们现在可以适当关注并抓紧搜集素材了。

好了，小憩之后让我们转回到技术，讲下 ESX/ESXi 是怎么一回事。ESX 名称来自于“ Elastic Sky X”，而 ESXi 则是从 ESX3.5 版本开始改的名，最早是 ESX 3i，后来就直接叫做 ESXi *.* 了。ESXi 比较 ESX 最大的变化是对 Hypervisor 进行了简化，去掉了服务控制台等功能，导致产品在体积缩小的同时从操作角度来看更加封闭，只能使用其 vCenter 或 Client 软件进行管理操作。最新的 ESXi5.0 安装包的大小为 290MB，远远小于早期的 ESX 版本。

本文开篇的 Hypervisor 典型分类其实就是从 VMware 的 ESX 和 GSX 产品分类而来的。GSX 为代表的 Type2 Hypervisor 由于其性能局限，目前的应用已经很少，不做细说。ESX/ESXi 一路发展至今，在服务器虚拟化市场已经独占鳌头。从结构上讲，VMware ESX 一直给人以紧密结合的一个整体的印象，由于其各种系统服务（包括系统启动和管理操作等）都是采用源于 Linux 2.4 内核的 Redhat 发布版进行修改后提供的，导致大家很容易误解其采用的也是 Linux 内核进行整个系统硬件的管理。但是其实不然，它还另有一颗不同的心 VMkernel。VMware 的官方发布资料一直有意无意的对其内核结构进行模糊宣传，但是 VMkernel 作为其不开源内核肯定和 Linux 无关是毋容置疑的，否则其必须得遵循 GPL 也完全开源。而事实上 VMware 只是对其产品中修改使用的各种 Linux 服务组件进行了开源，并没有将整个 ESX 完全开源。还有从 ESX 支持的所有硬件驱动必须集成在产品版本中发布，而不能进行自动安装修改这一表现推断，硬件抽象这一内核基本工作应该是由 VMkernel 完成的，而非其中的 Linux 内核。因此推断其结构应该是由 VMkernel 作为微内核，而 Linux 内核运行在用户空间，作为一个 Client 进程，管理整个系统其他服务特性（如 CLI、Boot Load 等）。这是典型的微内核或混合内核结构，区别就在于 VMkernel 在内核空间中集合的代码多少而定。再参考前面 VMware 创始人的经历，SimOS 的修改版本作为 VMkernel 微内核的推断基本上就十之八九成立了。注意这里说的是推断，只有等 VMware 的官方资料公布后才能确认对错。



多说一句，SimOS 项目最早研究的是在 MIPS 上运行 IRIX 系统，目前在 Stanford 网站上已经搜索不到，其衍生出的 SimOS-PPC 成为 IBM AIX 系统模拟的一个内部项目，在 AIX 4.3 License 下对外发布。还有一个衍生产品 SimBCM 用于模拟 Broadcom BCM1250 作为完全开源软件在 GPL 下发布。国内有些研究龙芯的兄弟在 2002 年左右就开始接触 SimOS，并研究通过其对 Linux/MIPS 进行模拟，有兴趣的可以去网上搜搜当时的一些文章，不过这两年也都销声匿迹了。

最后再介绍下 VMware 的最新服务器虚拟化产品家族，让大家对其盈利模式有所了解。首先是 ESXi，去年 8 月底发布的版本已经到了 5.0，免费使用。（早先的 ESX 版本都是收费的，但从 ESXi 开始全部免费使用）。单独的 ESXi 可以通过 vSphere Client 软件进行管理，也是免费的，但同时只能一次性管理一个物理 ESXi Server，不嫌累的话可以在管理 PC 上开多个进程同时管理多台物理 ESXi Server。当然不管谁真想用虚拟化，肯定不

会就只有一两台物理 Server，那么就需要 vCenter 软件对多台服务器进行集中管理了，vCenter 可以提供 convert、vMotion、HA 和 DRS 等 VM 扩展管理功能，而这些扩展功能是大型数据中心 VM 管理所必须的。vCenter 有 60 天的免费试用期，依靠 License 进行控制。这样做的好处是可以避免各种试用和测试等场景的 License 控制管理麻烦。同时只要是个正规点儿的企业使用 VM，谁也忍受不了 60 天就将整个系统重新折腾一遍，肯定要乖乖掏银子。

Xen

第二个 Hypervisor 界老牌重量级选手就是 Xen 了。Xen 最早诞生于剑桥大学电脑实验室的研究项目，2003 年 10 月 2 号由研究项目领导人 Ian Pratt 创建的 XenSource Inc.发布了第一个 Xen 的发行版本。2007 年 Citrix 收购了 XenSource，将 Xen 产品全部置于旗下，并将 Xen 的开源项目转移到 Xen.org 进行维护。同一时期 Citrix 推动成立了 Xen Project Advisory Board（简称 Xen AB，早期成员有 Citrix, IBM, Intel, Hewlett-Packard, Novell, Red Hat, Sun Microsystems 和 Oracle）对 Xen 进行代码管理和市场推广。2009 年，Citrix 宣布将其虚拟化管理平台 XenServer 也在 Xen.org 上全部开源，命名为 XCP（Xen Cloud Platform）。

目前，Xen.org 上面共包含 6 个相关项目，主要项目 Xen Hypervisor 和 XCP；孵化项目 Xen ARM Project（与三星合作在 ARM 上利用 Xen 达到虚拟化目标）和 XCI - Xen Hypervisor for Client Devices（目标减小 Xen Hypervisor 的代码规模，类似于 ESXi 之于 ESX）；归档项目 HXen - Hosted Xen（在 Windows 上和 Mac 上运行 Xen 的 Type2 类主机虚拟化）和 Project Satori（在 Hyper-V 上运行带有 Xen 代码 Para Virtualization Linux 作为 Guest OS）。

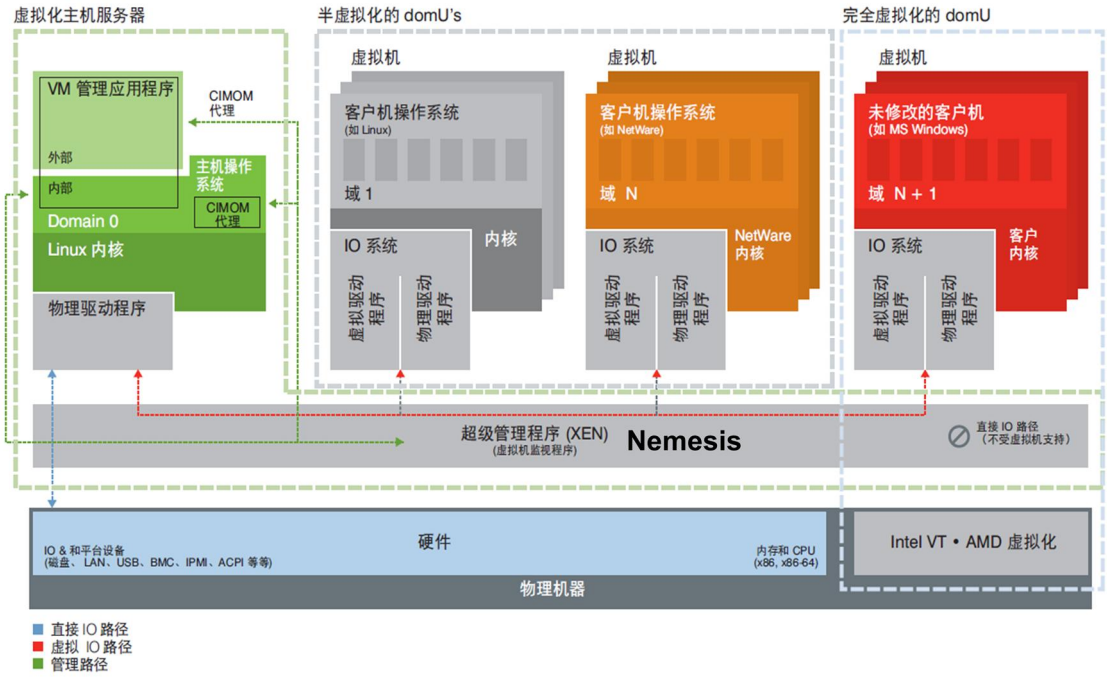
再八一八 Xen 现在东家 Citrix 的背景，这家由 IBM 工程师在 1989 年成立的公司，最初定位于操作系统（当时的 OS/2，Win32 等）多用户使用的软件市场，然而命运坎坷，从 1989 年到 1995 年连年亏损，甚至在 1991 年之前几乎没有任何收入。但从 1991 年开始至 1993 年 Citrix 融到了多笔风投，并收购了北电的一款软件产品 Netware Access Server，随后开始走上了上坡路，并于 1995 年上市。从 1997 年至今 Citrix 共收购了 27 家公司并将其产品融入到自己的产品体系中，如当前其主要产品 GotoMeeting、NetScaler、EdgeSight 和 Xen 等都是靠收购整合而来的。现如今 Citrix 已经发展成为超过 6000 人的全球性企业，2011 年总资产达到$3.75B，其中 2010 财年收入$327M。个人看完后感觉很励志，创业在于坚持，不要怕没收入，不要怕技术方向走错，山穷水尽疑无路，柳暗花明又一村。当然对 Citrix 来说得到 Microsoft 的贵人相助也是很重要的，更细节的内容有兴趣的同学自己去网上查看吧。

回来说 Xen Hypervisor 的实现，在前面已经介绍了部分内容，这里再对其进行一下汇总。首先从内核结构来说，Xen Hypervisor 采用的是 Exokernel 结构，其 kernel space 运行的

内核代码推断为剑桥大学电脑实验室 90 年代研究的 Nemesis 内核演变而来。Nemesis 在 99 年 4 月发布第二个版本之后就渺无声息了，时间上也与 Xen 的研究发布时期比较匹配。

然后是管理驱动的 Library OS，Xen 称其为 Domain0，可选的操作系统很显然是开源的 Linux 最为合适。为了和 Exokernel 配合进行系统调用等动作，必须对 Linux 内核代码进行适当修改才能作为 Domain0 系统。和 VMware 的 Linux 系统类似，Domain0 同样要负责 Bootload 和 CLI 等系统管理工作。Xen 与 Linux 的关系可谓一波三折，从诞生开始，开源的 Xen 就一直在受到市场的追捧，早期的 Amazon EC2、RackSpace 等大型 IaaS 运营商都是采用 Xen Hypervisor 来搭建其虚拟化平台，这些有实力的厂商自行修改和维护 Linux 内核代码，使 Xen 可以顺畅运行，同时推动了 Linux 集成厂商对 Xen 的支持，到 2009 年时，Suse、Redhat、Ubuntu、Debian、Gentoo 和 Arch 等发行版都集成 Xen domain0 的代码。但是随着 KVM 的雄起，以及 Xen Domain0 代码始终未能进入 Linux kernel，2009 年 KVM 的东家 Redhat 在其最新的 Enterprise 6 版本中去掉了支持 Xen Domain0 的代码（其另一款完全开源的知名产品 Fedora 则始终未支持 Xen Domain0），其盟友 Ubuntu 新的 8.10 版本也不再支持作为 Xen Domain0。这些变化导致 Xen 的前景一度不被人们所看好，甚至很多人开始预言 Xen 的末日来临，其地位很快会被 KVM 所取代。然而以 Citrix 为主导的 Xen 盟军们开始发力，2010 年 7 月 Linux kernel 2.6.31 发布版修改了部分代码，可以使用 PVOps 工具很简单的使其支持 Xen Domain0。2011 年 3 月 Xen Domain0 代码终于得偿所愿，进入 Linux kernel 2.6.37 发布版本。目前如 Redhat 和 Ubuntu 这些厂商们又需要在其还未发布的新版本中重新考虑对 Xen Domain0 的支持了。

最后再说下 Guest OS，Xen 早期的当家技术就是 Para Virtualization 的 Guest OS，虽然需要修改内核代码，但是相比较使用 QEMU 之类的模拟器技术，更高效的系统资源调用表现始终令人称赞。如 Amazon EC2 等 IaaS 厂商也大量采用这种 Para Virtualization Guest OS 来搭建自己的虚拟化服务器平台。而后随着硬件辅助虚拟化技术 HVM 的盛行，Xen 也开始推广其采用 QEMU 模拟器的硬件辅助技术以同时支持 Windows 系统作为 Guest OS。由于 Xen PV 代码早就进入了 Linux kernel 2.6.23 版本，因此基本上目前所有集成商的 Linux 发行版本已经都能够作为 Guest OS 支持 Para Virtualization。下面这张源自 XEN 发布资料的图可以很清楚的看出其各个部件的基本结构关系。



KVM

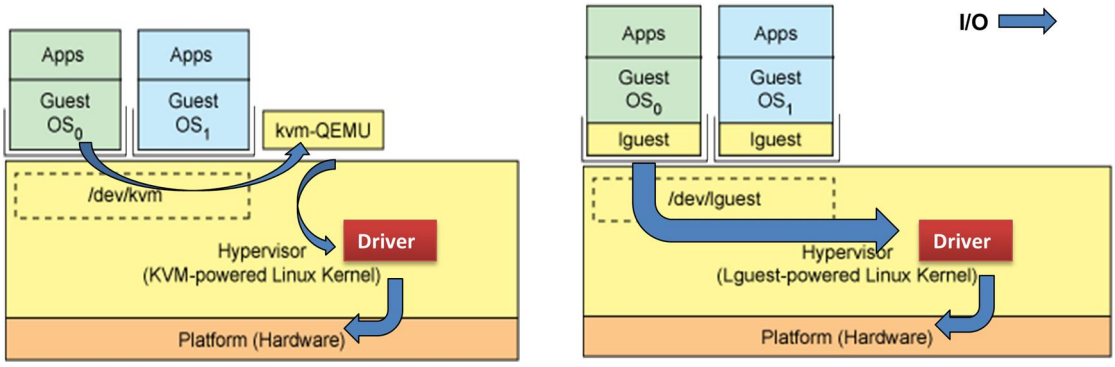
下面介绍一下使用 Linux 内核的 Hypervisor 技术 KVM（Kernel-based Virtual Machine）。这个可是最近几年比较火的明星选手，在 Redhat 等厂商的鼓吹之下，大有要在开源 Hypervisor 中取 Xen 而代之的意思。

首先说为什么会有 KVM 的出现。VMware 虽然部分开源，但是由于其 VMkernel 不公开，也没法做修改和二次开发，只有开源的 Xen 可以被广泛的开发者们折腾。而 Xen 虽然在 Linux（Domain0）上运行，但是采用的是独立的外内核 Nemesis，这点始终令 Linux 的狂信们不爽，而且也导致其无法进入早期的 Linux 内核，只能是在集成商修改过内核的 Linux 发行版中使用。（Linux 再开放也不会希望在其内核中再运行另一个内核，至于去年 Xen 成功进入 Linux 内核只能说是金主们博弈的结果，从技术上讲未必能有几人赞同）基于上述原因，Linux 的信徒决定要搞出一款真正基于 Linux 内核的开放性 Hypervisor 出来，于是有了 KVM。

KVM 是由一家以色列的 startup 公司 Qumranet 发布，2007 年 2 月进入 Linux kernel 2.6.20 版本后开始受到大家的关注，2008 年 9 月 Qumranet 被 Redhat 以$107M 收购。Qumranet 的创始人好像有以色列军方背景，应该也非常人。而后在 Redhat 的大力推动下，KVM 在 Hypervisor 技术领域的影响力扩张一发不可收拾。

从技术实现上看，KVM 必须安装在一个底层的 Linux 操作系统之上，通常也称之为宿主操作系统。KVM 包含了一个内核模块帮助宿主 Linux OS 成为一个 Hypervisor，同时使用

修改过的 QEMU 模拟器 KVM QEMU 来为其上的 Guest OS 提供进程空间，使每个 Guest OS 启动后，都会成为宿主 Linux OS 上的一个独立进程，并可以对其执行各种进程调度。与传统 Linux 的区别是，这些 Guest OS 运行在一个新的 Guest 状态，独立于标准的 Kernel 和 User 状态。Guest OS 的 I/O 请求都会先被映射到 KVM QEMU 独立进程中进行模拟处理，然后再下发给宿主 Linux OS 的 Driver 进行硬件操作。这个 KVM QEMU 进程有些类似 I/O PV 中的 Domain0 的作用，区别是 Driver 被安装在宿主 OS 上。当然我们可以不要这个 QEMU，直接将一些调用和模拟放到 Guest OS 上来做，这同样需要修改 Guest OS 内核，增加一个执行层次。这也是另一款基于 Linux 的开源 Hypervisor 软件 Lguest 的思路，如下图可以明显看出其实现的区别比较。



Lguest（也称 Lhype）是澳大利亚 IBM 的工程师 Rusty Russell 开发和维护的，在 2007 年就已经进入 Linux 2.6.23 版本内核，但由于其需要修改 Guest OS 的内核，同 CPU PV 一样，目前也只有 Linux 的版本支持。Lguest 最大的优势就是简单，Hypervisor 代码被简化到只有 5000 行左右，是 KVM 的 1/10，号称是最小的 Hypervisor，因而也被一些开发者所喜好。在 Guest OS 的 Lguest 代码瘦层中还做了很多其他虚拟化增强工作，如系统服务和中断处理等。

KVM 也曾有限的支持 I/O PV 技术（如 VirtIO），可以对部分修改过的 Guest OS 的 I/O 请求不经 QEMU 模拟直接下发到硬件，但最新的 KVM 社区介绍中已经摒弃掉了这部分内容，将其定位为一套运行在支持硬件虚拟化技术（Intel VT 和 AMD-V）的 X86 平台上的 Full Virtualization 解决方案。

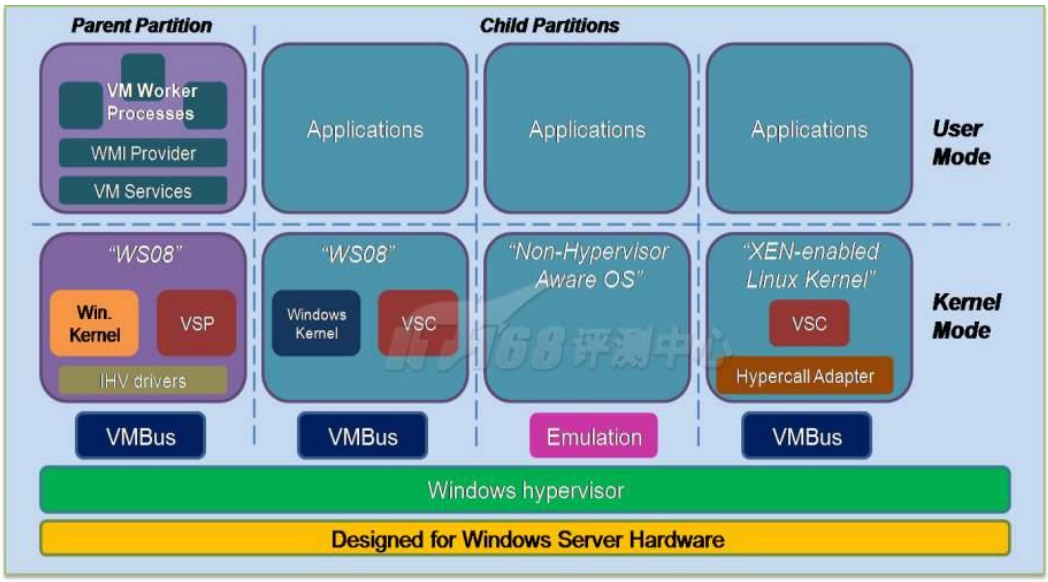
在当前的服务器市场，Linux 占据了绝对的优势，Windows 只是在中低端服务器有些份额，而像 AIX 等系统则只能在专用的高端服务器上有所斩获，开放性不够限制了其市场份额的发展。对高科技企业尤其是互联网公司来说，设备硬件的标准化（如 X86）可以降低采购成本和厂商捆绑风险，而软件开发能力又是其起家的根本，因此越大的公司越喜欢由自己的团队专门为业务应用系统定制开发和维护自己的底层平台，像 Linux 这种开源又免费的系统成为了最佳选择。得益于 Linux 的不断前进，KVM 同样在虚拟化方面拥有着光明的前景。如前所说，最大的 Linux 集成商 Redhat 已经在其 Enterprise 6.0 发布版本中取消了作为 Xen Domain0 的代码集成支持（Fedora 则始终没支持过 XEN），专心

推广其主导的 KVM 技术，导致很多使用 Redhat/Fedora 作为底层 OS 的用户也只得转投 KVM 的怀抱。同时 Ubuntu、Suse 和 Gentoo 等厂商也陆续在发行版中直接集成支持 KVM 虚拟化组件，推动了 KVM 更大范围的普及。显然在今后很长的一段时间内 KVM 和 Xen 还会在 Linux Server 平台上继续上演着夺面双雄这出大戏。

Hyper-V

最后来说说 Microsoft 的 Hyper-V。与其在个人终端市场所向披靡的境况截然相反，Windows 平台在服务器市场始终都未能成大气候。至于其原因业内分析众多，作者就不再献丑了。2008 年 7 月，Microsoft 推出了 Hyper-V 参与到服务器 Hypervisor 的战团中，虽然无论是名气还是市场份额都无法与前三款产品相比，但毕竟也在业内占据了一席之地，依靠背后 Windows 家族底蕴，在市场上也是小有斩获。

Hyper-V 有两种安装方式，一种是在现有的 Windows Server2008 系统上部署的安装包，另一种是包含了 Server2008 的完整安装包 Hyper-V Server，目前 Microsoft 推荐更多的是后一种安装方式。无论哪种方式，都需要一个 Server2008 作为 Parent Partition 与 Hyper-V 一起运行，管理硬件驱动程序。很明显 Hyper-V 也是以 Exokernel 方式运行，而 Server2008 则扮演着 Library OS 的角色，同时传说中 Hyper-V 500KB 的代码大小，也几乎只有 Exokernel 能够搞定。Hyper-V 与 Xen 架构大同小异，这里就不再多介绍了，贴张网上搞到的截图给大家简单看看。另外 Microsoft 的不开放也导致对其分析的无从下手。



Detailed architecture of Windows Server virtualization

在 Hyper-V 之上的 Guest OS 肯定是能跑几乎所有的 Windows 系统，但是 Linux 就有些麻烦了。目前经过 Windows 认证可以跑的就是 Suse10 SP3、Redhat Enterprise5 和 CentOS5.2

及其后续版本的几个发行版。而采用 Para Virtualization 方式时，理论上讲，由于 Hyper-V 的代码被合入了 Linux kernel 2.6.32 版本，因此采用其及后续内核版本的 Linux 发行版都可以作为 Guest OS 在 Hyper-V 上运行。另外通过 Xen 的 Satori 项目，还可以将修改过内核的支持 Xen PV 的操作系统在 Hyper-V 上作为 Guest OS 运行。

最后再提一下 Hyper-V 的限制，这可能也是其当前无法挤进主流 Hypervisor 的原因之一。基于 Windows Server2008（R2）版本的 Hyper-V 在 USB/声卡/光驱/显卡都存在着诸多使用限制和问题，同时在非 R2 版本上不支持迁移，总之给人以不够成熟稳定的感觉，与微软第一 OS 厂商的市场地位实不副名。只能期待着其基于下一代 Windows8 平台的 Hyper-V 能够解决上述问题限制，令人耳目一新。

OS 虚拟化技术

在上述虚拟化场景中 Guest OS 的运行可以有两种选择，一是无意识虚拟化，就是说安装后 Guest OS 仍然认为自己是工作在一台独立的物理服务器上。二是有意识虚拟化，既通过对 OS 内核的一些变动，使 Guest OS 了解自己是个虚拟化操作系统，可以执行一些虚拟化相关指令和系统调用。例如前面各种 PV 技术或 Lguest 技术中提到的 Modified OS 等。

无意识虚拟化 Guest OS 明显较有意识虚拟化 Guest OS 具有更广泛的应用场景，但二者都对整个计算机运行体系提出了更多的要求，导致了复杂的 Hypervisor 的出现。那么我们换个思路，看看能不能省些事，抛开 Hypervisor 搞下虚拟化。

前面的虚拟化场景尽量都要求 Guest OS 能够为 Windows 或 Linux 等任意 OS，这样可以更灵活的在物理服务器上进行虚拟机部署。但是从实际项目来说，我们完全可以要求在同一台物理服务器上部署的 Guest OS 都采用相同的操作系统，甚至是同一种发布版本的 OS，将不同类型的 Guest OS 分布部署在不同的物理服务器之上，尤其是在大型的数据中心项目中，这种方式更加有利于整体规划。例如在一台物理服务器上只部署多个 Redhat Linux 作为 Guest OS，而另一台物理服务器只部署 Windows Server 的 Guest OS。那么这时我们除了传统虚拟化方案外就有了新的选择，OS Level Virtualization（OS LV）技术。

OS LV 技术的思路是在操作系统之上，将其进程和资源进行隔离处理，同时提供给不同的用户使用，让不同的用户感觉自己是在操作一个独立的 VM，但其实各种调用和处理都是由底层 OS 完成的，只有程序进程的执行空间是相互独立的。这种技术提供的 VM 通常被形容为“容器 Container”，个人觉得这个词用得很贴切。有点儿类似 Windows Server 系统的多用户同时登陆操作，但 OS LV 更进一步，每个用户都拥有独立的文件系统与进程空间，相互隔离操作互不干扰。目前的 OS LV 还只有基于 Linux OS 的产品，如

Open VZ、Linux-VServer 和 FreeVPS，都能够支持 CPU、内存、I/O、存储和网络的配额独立分配，其中 Open VZ 还能提供如 vMotion 的 VM 迁移功能。相信 Microsoft 应该也有相似的 Windows Server 平台技术在开发中，估计发布也就是这两年的事情。

OS LV 的优点和缺点同样突出，优点是 Guest OS 的应用程序执行过程中不需要像其他虚拟化技术那样多经过 Guest OS 和 Hypervisor 两层的翻译调用，效率可以很高。据 Open VZ 称单物理服务器可以支持多达 300-500 个 VPS（就是 VM，其实这个数值可以简单理解为一个操作系统能够支持的进程总量），这是当前主流 Hypervisor 虚拟化技术所无法想象的。缺点则是目前能够支持的 Guest OS 操作系统有限，只有 Linux，而且还需要注意 Guest OS 和底层 OS 的版本需要尽量一致，导致其部署适用范围较窄。另外就是安全性与稳定性方面的顾虑，由于资源共享，单个 VPS 的进程故障是否真的如其宣传的不会影响到其他 VPS 甚至底层 OS，还需要更多更广泛的应用案例来验证。

其他虚拟化技术

再多介绍几个虚拟化技术，从实现上看类似于 OS LV 或 Type1 Hypervisor 模型，这些技术由于种种明显的限制目前大多应用在研究试验中，都没有知名商用产品流传于世。

首先是 UML（User-Mode Linux），同样是在 Linux 上运行 Linux，但是作为 Guest OS 的 Linux 需要像 Lguset 一样做些内核方面的改动，因此适用范围更窄，但其特点是可以在 Guest OS 上继续以 UML 运行更上层的 Guest OS。

然后是 CoLinux，专门用于在 Windows 上运行 Linux 的技术，可以使它们共享底层硬件资源，同样需要修改 Linux Guest OS 的内核，但每个 Windows 上只能运行一个 Guest OS。

其次是 WINE（Wine Is Not an Emulator），从名称就可以看出其非同一般的设计思路。WINE 是在 Linux 系统上运行 Windows 系统的一种方案，其没有对硬件操作层面做指令模拟，而是在 API 层面对 Guest Windows OS 的指令通过动态链接库 DLL 的形式翻译执行。很显然庞大的 Windows API 指令会对其运行造成严重的负担。

最后是 Cygwin，相信这个熟悉的同学更多一些，它是由 Redhat 开发的软件工具，用于在 Windows 上模拟 Linux 环境，以便调试类 Unix 软件，更类似于一种编程开发用环境。

结束语与声明

服务器虚拟化技术就介绍这么多了，其实还有很多如 QEMU、VirtualBox 等重要的技术产品都可以拿出来再深入讲讲，但是个人能力有限就不多献丑了。本文的套路还是横向平铺，纵向深入实在是力有不及。

如果希望更好的理解服务器虚拟化技术，就需要对现代计算机技术的方方面面都有所涉猎，软的硬的最好都要看看。无可否认，计算机技术的出现已经对人类种群的发展产生了根本性的影响，再丰富的想象力也已跟不上时代日新月异的变化，无数几十甚至十几年前的梦想现在都已实现，今后还会迎来更多。身处在这个波澜壮阔的时代，身为一名技术人员，永远追不上技术更新的脚步，永远处于饥渴的学习状态，但任何不经意的发现都可能改变整个技术生态的发展，进而改变人类的历史。悲哉？幸哉？