首本Linux KVM虚拟化技术专著，由Intel虚拟化技术部门资深虚拟化技术专家和KVM研究者撰写

系统介绍KVM虚拟机的功能、特性和使用方法，深入剖析KVM虚拟机的核心技术和原理

實戰

任永杰 单海涛 著

KVM: Principles and Practices

KVM虚拟化技术实战与原理解析

机械工业出版社
China Machine Press

KVM虚拟化技术

——实战与原理解析

任永杰　单海涛　著

ISBN：978-7-111-43900-4

本书纸版由机械工业出版社于2013年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.bbbvip.com

新浪微博 @研发书局

腾讯微博 @yanfabook

目　录

9.5　本章小结

9.6　本章注释和参考阅读

前言

为什么要写这本书

近几年，云计算大潮风起云涌，Amazon的AWS是公共云计算平台的先驱和领导者，Google的GAE、GCE也取得了一定的成绩，国内大型互联网公司也分别推出自己的云计算服务，如阿里云、腾讯云、盛大云、新浪SAE、百度BAE等。云计算已成为IT技术领域最热门的词汇之一，现在很多互联网服务都声称自己是“云”服务。在云计算发展的大背景下，支撑着云计算服务的最底层、最基础的虚拟化技术也得到了快速发展，成为技术研究和应用的热点之一。

在虚拟化领域也有很多的技术和产品种类，包括传统的老牌虚拟化软件VMware、Microsoft的Hyper-V、Citrix的XenServer/XenClient、Oracle的VirtualBox等，还有在Linux平台上比较流行的开源虚拟化技术KVM、Xen等，甚至包括Linux上的轻量级虚拟化技术LXC。随着Linux服务器广泛地应用在互联网等热门行业，KVM作为Linux内核原生的虚拟化技术受到了开发者和用户广泛的关注，并且已经得到了较大规模的应用，如Google的公有云计算引擎GCE就是在底层完全使用KVM作为虚拟化技术来实现的。

KVM虚拟机最初是由以色列一家创业公司Qumranet员工Avi Kivity等人开发，于2006年8月完全开源并推向Linux内核社区，当年10月被Linux社区接受。2007年2月发布的Linux 2.6.20是第一个带有KVM模块的Linux内核正式发布版本。Red Hat公司于2008年9月正式将Qumranet公司收购，接着投入了较多的资源到KVM虚拟化的开发中。在2010年11月，Red Hat公司发布的RHEL 6中完全用KVM替换了RHEL 5中默认支持的Xen，让KVM成为RHEL操作系统默认的虚拟化方案。

KVM必须依赖CPU提供的硬件虚拟化技术，以Intel、AMD为代表的x86硬件平台在最近几年也推出了很多与虚拟化相关的硬件特性，包括最初基本的CPU的VT支持和EPT、VPID，以及I/O设备的VT-d、SR-IOV，还包括最新的APIC-v、Shadow VMCS等特性。随着x86硬件对虚拟化技术的支持越来越成熟，KVM在最新的硬件平台上的虚拟化效率也得到很大的提高。

除了硬件方面的虚拟化技术逐渐成熟之外，KVM作为Linux内核虚拟机，受到许许多多Linux软件开发者的青睐。Red Hat公司无疑是KVM开发中最大的一股力量，很多Red Hat工程师在KVM、QEMU、libvirt等开源社区中成为核心开发成员。除了Red Hat之外，还有许多知名公司都有不少软件开发者为KVM贡献自己的代码，如IBM、Intel、Novell、AMD、Siemens、华为等。除了这些大公司的开发者之外，还有许多小

公司和个人独立开发者活跃在KVM开源社区，为KVM开发代码或者做测试工作。

在硬件方面的虚拟化支持和软件方面的功能开发、性能优化的共同作用下，目前KVM虚拟化技术已经拥有非常丰富的功能和非常优秀的性能。而且，KVM虚拟化的上层应用软件和云计算平台（如libvirt、Ovirt、Virt-Manager、OpenStack等）也是日渐成熟。因此，KVM已经成为个人和企业在进行系统虚拟化或云平台建设的底层虚拟化时一个非常不错的选择。

在2012年，我们接受了机械工业出版社杨福川老师的邀请，决定写一本书来专门介绍KVM虚拟化的功能和原理，经过1年多的精心准备和努力写作，本书终于顺利出版。

本书所有作者在虚拟化技术领域有多年的学习和工作经验，积累了比较丰富的知识和经验。任永杰在2011年年初加入Intel的云计算与虚拟化部门，专注于Linux开源虚拟化的功能测试、性能分析以及与开源社区交流等工作。单海涛于2007年加入Intel虚拟化部门，一直从事KVM、Xen等虚拟化软件的开发及开发团队的管理。本书内容由任永杰统筹规划，单海涛编写了本书的第1、2两章，其余的第3至9章由任永杰编写。

在国内，本书是目前唯一一本专门介绍KVM虚拟化技术的书籍，而且在英文书籍中也没有任何一本书是在专门介绍KVM技术。本书可以让大家对Linux内核虚拟机——KVM的原理、技术、功能和性能有较为详细的了解，也希望能通过本书让更多的读者在学习和工作中实际应用KVM虚拟化技术，并且能够参与到Linux、KVM、QEMU等开源社区中去。

读者对象

本书适合对Linux下虚拟化或云计算基础技术感兴趣的读者，包括Linux运维工程师、KVM开发者、云平台开发者、虚拟化方案决策者、KVM的用户以及其他对KVM虚拟机感兴趣的计算机爱好者。希望本书对这些读者了解KVM提供以下帮助。

❑运维工程师：了解KVM的使用方法、功能和基本的性能数据，能够搭建高性能的KVM虚拟化系统并将其应用于生产环境中。

❑KVM开发者：了解KVM的基本原理和功能，了解其基本用法和一些调试方法，以及如何参与到KVM开源社区中去贡献代码。

❑云平台开发者：了解底层KVM虚拟化的基本原理和用法，以促进云平台上层应用的开发和调试的效率。

❑虚拟化方案决策者：了解KVM的硬件环境需求和它的功能、性能概况，以便在虚拟化技术选型时做出最优化的决策。

❑普通用户：了解KVM的功能和如何使用KVM，用KVM虚拟化技术来促进相应的学习、开发和测试。

如何阅读本书

本书分为9章，具体介绍如下内容：

第1章简单介绍了云计算的概念和技术，然后简单介绍了目前比较流行的包括KVM在内的几种虚拟化技术。

第2章简单介绍了Linux的基本历史、KVM的基本架构、QEMU的作用以及Intel的硬件虚拟化技术概况。

第3章详细介绍了KVM依赖的硬件环境配置、编译以及安装KVM和QEMU的步骤与方法，最后演示了如何启动一个KVM客户机。

第4章详细介绍了客户机系统的CPU、内存、存储、网络、图形显示等方面的配置和基本原理，以及CPU和内存的过载使用。

第5章详细介绍了除了第4章的基础功能之外的高级功能，包括半虚拟化驱动virtio、设备直接分配VT-d、热插拔、动态迁移、嵌套虚拟化、KSM、透明大页、KVM安全技术、QEMU监控器、QEMU命令行参数以及从物理机或其他虚拟机迁移到KVM的方法。在介绍每个功能时，都尽量对其背后的原理和技术做了简单的介绍。

第6章介绍了管理KVM虚拟化的上层软件，包括libvirt、virt-manager、virt-viewer、virt-install以及OpenStack云计算平台。

第7章分别介绍了RHEL、Fedora、SLES、OpenSuse、Ubuntu等Linux发行版中的KVM虚拟化功能的使用。

第8章首先简单介绍了虚拟化性能测试，然后详细介绍对KVM虚拟化的CPU、内存、网络、磁盘I/O等重要组件进行性能测试的方法、常见工具和具体测试步骤，并且分享了作者在实际性能测试实验中得到的一些数据供读者参考，以便读者对KVM的性能有基本的认识。

第9章简单介绍了Linux、KVM、QEMU等开源社区的基本情况，也简单介绍了KVM、QEMU和KVM单元测试代码的基本结构，接着详细介绍了如何向QEMU/KVM开源社区贡献自己的代码以及如何提交KVM相关的bug。

本书中每章后面都有“本章小结”，部分章节后面有“本章注释和参考阅读”。“本章注释”是本章正文中一些需要详细说明的地方，“参考阅读”中包含了笔者推荐的与本章内容相关的一些参考资料供有兴趣的读者对相关内容做更多的了解。

读者可以根据自己的兴趣和已经掌握的知识来有选择地阅读各个章节。如果对KVM基本没有什么了解，建议按本书顺序阅读，通读一遍之后再对感兴趣的章节进行仔细阅读。今后可能会经常使用KVM的读者，在阅读本书之时，可以根据书中示例进行实际操作，如果是开发

者也可以查看相应的源代码。另外，如果对某些章节中的内容比较感兴趣，建议阅读每章末尾笔者推荐的一些参考资料。

勘误和支持

KVM、QEMU等开源社区非常活跃，QEMU/KVM发展迅速，经常有新的功能加进去或者原有功能被改进，特别是QEMU命令行参数很可能会有所改动。因此本书中QEMU命令行参数只能适用于本书中提及的QEMU版本，若使用不同的版本，命令行参数可能并不完全相同。另外，本书中的QEMU命令行示例一般都没有添加"-enable-kvm"参数，这是由于笔者写作本书时使用的qemu-kvm是默认启用KVM加速的，如果读者使用普通的QEMU来启动KVM客户机，就请根据需要自行在QEMU命令行中添加"-enable-kvm"参数。

由于KVM和QEMU的发展变化比较快，加之作者的技术水平有限和编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者朋友批评指正。读者朋友对本书相关内容有任何的疑问、批评和建议，都可以通过作者之一（任永杰）的博客网站http://smilejay.com/进行讨论，作者会尽力回复每一条留言信息并帮助读者得到满意的答案。全书中涉及的示例代码程序（不包含单行的命令）和重要的配置文件，都可以从网站https://github.com/smilejay/kvm-book查看和下载。

如果读者朋友们有更多的宝贵意见或者任何关于KVM虚拟化技术的讨论，欢迎发送电子邮件至smile665@gmail.com，我们感谢并期待能

够得到朋友们的真挚反馈。

致谢

首先要感谢Linus Torvalds这样的“大神”级别的Linux程序员，是他们创造了一个伟大的操作系统内核——Linux。

其次要感谢Avi Kivity等创建KVM项目的程序员，如果没有KVM项目，自然就不会有本书的存在。

感谢Intel开源技术中心的云计算与虚拟化部门，我们在这里学到了很多与Linux和虚拟化相关的技术。特别感谢该部门的张献涛，他是KVM IA64架构的维护者，在本书写作过程中，给了我们很多帮助和建议。

非常感谢机械工业出版社华章公司的编辑杨福川和姜影老师，在这一年多的时间中始终支持本书的写作，在他们的鼓励和帮助下，我们才能顺利地完成本书的写作。

最后感谢我们的爸爸、妈妈，感谢你们将我们养育成人并给予最无私的爱！

谨以此书献给我们的家人，以及众多喜欢Linux、KVM等开源软件的朋友们！

任永杰　单海涛

第1章　虚拟化与云计算

本书是一本以KVM为例，讲解虚拟化技术的书。可是看到这一章的标题，有读者会问：虚拟化为什么会跟云计算扯到一起去呢？其实，作为两个时下比较热门的技术，它们之间有着密不可分的千丝万缕的联系。本章将试图为读者理清它们之间的关系，从简单介绍云计算的概念开始，逐步解释虚拟化技术在云计算中的作用，并对当前流行的虚拟化解决方案进行一些简单的对比。

1.1　云计算概念

云计算是一个本世纪初才方兴未艾的热门名词，其本身是由Google前首席执行官埃里克·施密特（Eric Schmidt）在2006年8月9日的搜索引擎大会（SES San Jose 2006）上首次提出的。在此之后，云计算变得炙手可热，很多公司趁势宣传，号称自己是先进的云计算公司。加之业界也一直没有对云计算形成一个统一的定义，各种各样的云变得“乱花渐欲迷人眼”。抛开形形色色的宣传，本书试图为读者一步步理清云计算的真实面目。

1.1.1　发展历史和现状

虽然云计算的提出距离现在还只是短短的几年，但是云计算并不算是个彻底的新事物，因为早在它之前已经有各种类似或关联的概念被先后提出。早在60年代麦卡锡（John McCarthy）就提出了把计算能力作为一种像水和电一样的公用事业提供给用户。1999年，Salesforce.com提出的通过一个网站向企业提供企业级的应用的概念，已经颇有云计算的雏形。另外，20世纪90年代，为了充分利用空闲的CPU资源，通过网络互联搭建平行分布式计算平台，诞生了网格计算，例如1999年出现的SETI@home[1] 科学研究项目。在网格计算中，将一个大型的计算任务拆分，分配给各个网格终端计算然后组合的思想，与云计算中的Map/Reduce技术不谋而合。

这些早期的概念虽然与云计算相关，但是要么碍于当时的计算机和网络技术，要么只是作为一种研究项目，始终没有能够被广泛推广并形成规模。然而，在本世纪初，计算机网络的大量普及和Web数据中心的建立，为云计算提供了必要的硬件准备。

在现有的被大家熟知并使用的共有云计算平台中，最著名的是由Amazon（亚马逊）和Google（谷歌）公司提供的服务。

Amazon使用弹性计算云（EC2）和简单存储服务（S3）为企业提供计算和存储服务。收费的服务项目包括存储服务器、带宽、CPU资源以及月租费。月租费与电话月租费类似，存储服务器、带宽按容量收费，CPU根据时长（小时）运算量收费。Amazon把云计算做成一个大生

意并没有花太长的时间：不到两年时间，在Amazon上注册的开发人员就达44万人，还有为数众多的企业级用户。云计算是Amazon增长最快的业务之一。Amazon的财报数据显示，在其2012年收入中，云计算相关业务的收入已经高达21亿美元。

Google则当数最大的云计算的使用者。Google搜索引擎就建立在分布在全世界200多个地点、超过100万台服务器的支撑之上，这些设施的数量正在迅猛增长。Google地球、地图、Gmail、Docs等也同样使用了这些基础设施。对于Google Docs[2] 之类的应用，用户数据会保存在互联网上的某个位置，可以通过任何一个与互联网相连的系统十分便利地访问这些数据。Google于2008年4月推出了Google App Engine（GAE）云计算服务，GAE是采用“平台即服务”（PaaS）的云计算模式，可以让第三方开发者在GAE平台上快速地开发自己的Web应用程序，而且GAE还将开发者的Web应用程序部署在Google全球的数据中心。Google还于2012年6月底发布了自己的“基础设施即服务”（IaaS）模式的云计算平台——Google Compute Engine（GCE），GCE的底层采用的是KVM虚拟化技术，它提供了与亚马逊EC2类似的云计算服务。

随着云计算的价值被不断发掘，越来越多的大公司开始加入到研究和部署云计算技术的阵营。

2008年2月1日，IBM宣布将在中国无锡的太湖新城科教产业园为中国的软件公司建立全球第一个云计算中心。

2008年7月29日，雅虎、惠普和英特尔宣布一项涵盖美国、德国和新加坡的联合研究计划，推出云计算研究测试床，推进云计算。该计划要与合作伙伴创建6个数据中心作为研究试验平台，每个数据中心配置1400个至4000个处理器。这些合作伙伴包括新加坡资讯通信发展管理局、德国卡尔斯鲁厄大学Steinbuch计算中心、美国伊利诺伊大学香宾分校、英特尔研究院、惠普实验室和雅虎。

2010年7月，美国国家航空航天局和包括Rackspace、AMD、Intel、戴尔等支持厂商共同宣布"OpenStack"开放源代码计划，微软在2010年10月表示支持OpenStack与Windows Server 2008 R2的集成，而Ubuntu已把OpenStack添加到Ubuntu 11.04版本中。

2011年2月，思科系统正式加入OpenStack，重点研发OpenStack的网络服务。

1.1.2 概念

云计算自从提出，一直没有一个明确而统一的定义。维基百科对云计算做了如下的描述[3]：云计算是一种通过因特网以服务的方式提供动态可伸缩的虚拟化的资源的计算模式（Cloud computing is a

style of computing in which dynamically scalable and often virtualized resources are provided as a service over the Internet）。美国国家标准与技术研究院（NIST）定义：云计算是一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问，进入可配置的计算资源共享池（资源包括网络、服务器、存储、应用软件和服务），这些资源能够被快速提供，只需投入很少的管理工作，或与服务供应商进行很少的交互（Cloud computing is a model for enabling ubiquitous,convenient,on-demand network access to a shared pool of configurable computing resources（e.g.,networks,servers,storage,applications,and services）that can be rapidly provisioned and released with minimal management effort or service provider interaction）。

也有人基于云端计算的实现方式，认为云计算是分布式计算技术的一种，其最基本的概念是，透过网络将庞大的计算处理程序自动分拆成无数个较小的子程序，再交给由多部服务器所组成的庞大系统经搜寻、计算分析之后将处理结果回传给用户。透过这项技术，网络服务提供者可以在数秒之内，处理数以千万计甚至亿计字节的信息，实现和“超级计算机”同样强大效能的网络服务。提供资源的网络被称为“云”。“云”中的资源在使用者看来是可以无限扩展的，并且可以随时获取，按需使用。

1.1.3　云计算模式

云计算是推动IT转向以业务为中心模式的一次重大变革。它着眼于运营效率、竞争力和快速响应等实际成果。这意味着IT的作用正在从提供IT服务逐步过渡到根据业务需求优化服务的交付和使用。这种全新的模式将以前的信息孤岛转化为灵活高效的资源池和具备自我管理能力的虚拟基础架构，从而以更低的成本和更好的服务的形式提供给用户。IT即服务将提供业务所需要的一切，并在不丧失对系统的控制力的同时，保持系统的灵活性和敏捷性。

云计算的表现形式多种多样，简单的云计算在日常网络应用中随处可见，比如腾讯QQ空间提供的在线制作Flash图片，Google的搜索服务，Google Docs和Google Apps等。目前，云计算的主要服务形式有：SaaS（Software as a Service，软件即服务）、PaaS（Platform as a Service，平台即服务）和IaaS（Infrastructure as a Service，基础设施即服务）。

1.软件即服务

SaaS提供商将应用软件统一部署在自己的服务器上，用户根据需求通过互联网向厂商订购应用软件服务，服务提供商根据客户所定软件的数量、时间的长短等因素收费，并且通过浏览器向客户提供软件的模式。这种服务模式的优势是，由服务提供商维护和管理软件，提

供软件运行的硬件设施，用户只需拥有能够接入互联网的终端，即可随时随地使用软件。在这种模式下，客户不再像传统模式那样花费大量资金在硬件、软件、维护人员上，只需要支出一定的租赁服务费用，通过互联网就可以享受到相应的硬件、软件和维护服务，这是网络应用最具效益的营运模式。对于小型企业来说，SaaS是采用先进技术的最好途径。

目前，Salesforce.com是提供这类服务最著名的公司，Google Docs、Google Apps和Zoho Office也属于这类服务。

2.平台即服务

PaaS把开发环境作为一种服务来提供。这是一种分布式平台服务，厂商提供开发环境、服务器平台、硬件资源等服务给客户，用户在其平台基础上定制开发自己的应用程序并通过其服务器和互联网传递给其他客户。PaaS能够为企业或个人提供研发的中间件平台，提供应用程序开发、数据库、应用服务器、试验、托管及应用服务。

Google App Engine、Salesforce的force.com平台、八百客的800APP是PaaS的代表产品。以Google App Engine为例，它是一个由python应用服务器群、BigTable数据库及GFS组成的平台，为开发者提供一体化主机服务器及可自动升级的在线应用服务。用户编写应用程

序并在Google的基础架构上运行就可以为互联网用户提供服务，Google提供应用运行及维护所需要的平台资源。

3.基础设施即服务

IaaS即把厂商的由多台服务器组成的“云端”基础设施，作为计量服务提供给客户。它将内存、I/O设备、存储和计算能力整合成一个虚拟的资源池为整个业界提供所需要的存储资源和虚拟化服务器等服务。这是一种托管型硬件方式，用户付费使用厂商的硬件设施。例如Amazon Web服务（AWS）、Google的Google Compute Engine、IBM的BlueCloud等均将基础设施作为服务出租。

IaaS的优点是用户只需低成本获得所需的硬件资源，按需租用相应计算能力、存储容量、网络带宽，而且省去了硬件运维方面的成本，大大降低了用户在硬件上的开销。

1.2　云计算技术

云计算的兴起，改变了现有的以本地计算为主的应用模型。用户不再需要付费购买软件并将其安装到本地的计算机中执行。取而代之，大量的计算任务，由客户端通过网络发起，在云计算提供商的数据中心的服务器集群上进行计算，其结果经由网络返回，在客户端进行呈现。新的计算模型的提出，必然伴随着新的问题需要解决。在云计算的环境下，不同厂商对如何有效地管理云端的资源，为用户提供快捷的计算进行了大量研究，提出和总结了一些行之有效的云计算技术。

1.2.1　Map/Reduce

Map/Reduce是Google开发的编程模型，它是一种简化的分布式编程模型和高效的任务调度模型，用于大规模数据集（大于1TB）的并行运算。严格的编程模型使云计算环境下的编程十分简单。MapReduce模式的思想是将要执行的问题分解成Map（映射）和Reduce（化简）的方式，先通过Map程序将数据切割成不相关的区块，分配（调度）给大量计算机处理，达到分布式运算的效果，再通过Reduce程序将结果汇总输出。

Map/Reduce编程模型适用于很多应用，例如，分布式搜索、分布式排序、机器学习、基于统计的机器翻译等。在Google公司，互联网网页的搜索索引也是用Map/Reduce技术计算生成的。

Map/Reduce目前已经有基于各种不同计算机编程语言的库实现，其中相当流行的是有Apache基金会的用Java语言开发的开源的Hadoop。

1.2.2 资源管理平台

云计算资源规模庞大，服务器数量众多并分布在不同的地点，同时运行着数百种应用，如何有效地管理这些服务器，保证整个系统提供不间断的服务是巨大的挑战。

云计算系统的平台管理技术能够使大量的服务器协同工作，方便进行业务部署和开通，快速发现和恢复系统故障，通过自动化、智能化的手段实现大规模系统的可靠运营。

当前比较流行的云计算平台，主要有思杰的CloudStack，开源的Eucalyptus，VMware公司的vCloud Director和开源的OpenStack等。除了VMware公司的vCloud Director没有免费版本和只支持VMware自由的虚拟化产品以外，其余几个都提供免费版本，而且支持多个虚拟化

产品。Eucalyptus和OpenStack还对亚马逊的API保持兼容，基于亚马逊API的所有的脚本和软件产品都可以轻松地进行私有云部署。

现在，CloudStack和OpenStack正处在激烈的竞争中，两者都希望自己能够成为开源社区云计算平台的事实标准。CloudStack在成熟度上面明显优于OpenStack，在培育客户方面也占了先机，但是背后的推手主要是思杰。OpenStack虽然是后起之秀，但是得到了诸如IBM、思科、英特尔、惠普和戴尔等大牌厂商的支持。究竟谁是未来的开源云平台老大，鹿死谁手，还未可知。

1.2.3 虚拟化

虚拟化是构建云基础架构不可或缺的关键技术之一。云计算的云端系统，其实质上就是一个大型的分布式系统。虚拟化通过在一个物理平台上虚拟出更多的虚拟平台，而其中的每一个虚拟平台则可以作为独立的终端加入云端的分布式系统。比起直接使用物理平台，虚拟化在资源的有效利用、动态调配和高可靠性方面有着巨大的优势。利用虚拟化，企业不必抛弃现有的基础架构即可构建全新的信息基础架构，从而更加充分地利用原有的IT投资。

可以说，虚拟化是云端系统部署必不可少的基础。正是因为虚拟化在云计算中的重要性，本书将集中篇幅以KVM为例，为各位读者讲解

虚拟化从技术理解到实践部署的方方面面。从下一节开始，本书将进入虚拟化技术的专题。

1.3　虚拟化技术

虚拟化是一个广义的术语，是指计算元件在虚拟的基础上而不是真实的基础上运行，是一个为了简化管理、优化资源的解决方案。

如图1-1所示，我们可以将一般的计算模型抽象成为一定的物理资源和运行于之上的计算元件，它们之间通过定义的物理资源接口进行交互。随着计算机硬件技术的发展，物理资源的容量越来越大而价格越来越低，在既有的计算元件架构下，物理资源不可避免地产生了闲置和浪费。为了充分利用新的物理资源，提高效率，一个比较直接的办法就是更新计算元件以利用更加丰富的物理资源。但是，人们往往出于对稳定性和兼容性的追求，并不情愿频繁地对已经存在的计算元件做大幅度的变更。虚拟化技术则是另辟蹊径，通过引入一个新的虚拟化层，对下管理真实的物理资源，对上提供虚拟的系统资源，从而实现了在扩大硬件容量的同时，简化软件的重新配置过程。

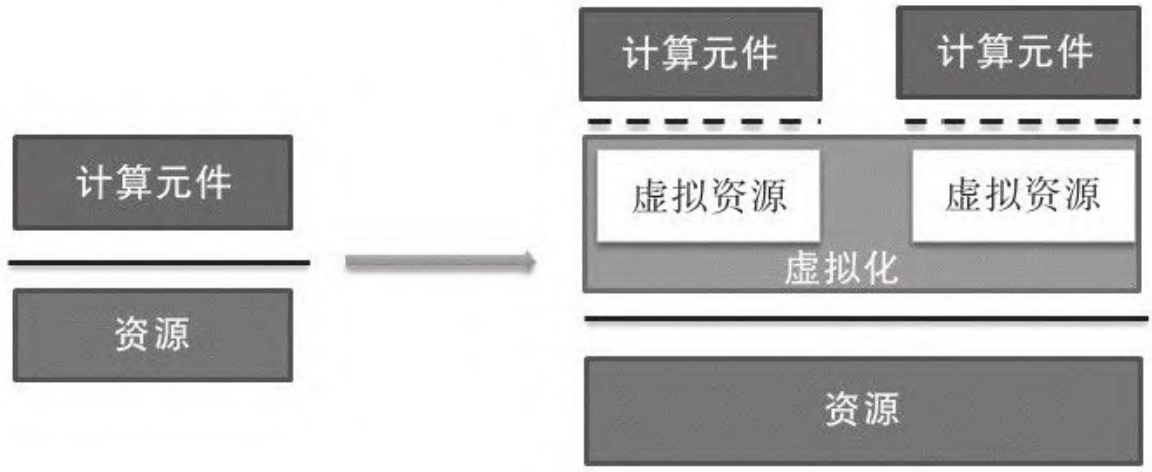


图　1-1　虚拟化

为了表述虚拟化的一般概念，在图1-1中使用了资源这个词。在实际应用中，资源可以表现为各种各样的形式。例如，如果把操作系统及其提供的系统调用作为资源，那么虚拟化就表现为操作系统虚拟化。Linux容器虚拟化技术就是在同样的一份Linux操作系统之上，虚拟出多个同样的操作系统。再例如，如果把整个X86平台包括处理器、内存和外设作为资源，那么对应的虚拟化技术就是平台虚拟化，在同一个X86平台上面，可以虚拟多个X86平台，每个虚拟平台都可以运行自己独立完整的操作系统。这也是我们在本书中将要重点讨论的虚拟化技术。

在X86平台虚拟化技术中，新引入的虚拟化层通常称为虚拟机监控器（Virtual Machine Monitor,VMM），也叫做Hypervisor。虚拟机监控器运行的环境，也就是真实的物理平台，称之为宿主机。而虚拟出

来的平台通常称为客户机，里面运行的系统对应地也称为客户机操作系统。

虚拟化技术有很多种实现方式，比如软件虚拟化和硬件虚拟化，再比如准虚拟化和全虚拟化。下面将针对每种实现方式做一个简单的介绍。

1.3.1 软件虚拟化和硬件虚拟化

通过前面的简单介绍，相信读者对虚拟化已经有了一个大致的了解。那么，什么是软件虚拟化和硬件虚拟化呢？在回答这个问题之前，读者需要思考一个问题：既然虚拟化这么美好，是不是物理资源都可以被虚拟化？

如图1-2所示，实现虚拟化的重要一步就在于，虚拟化层必须能够截获计算元件对物理资源的直接访问，并将其重新定向到虚拟资源池中。根据虚拟化层是通过纯软件的方法，还是利用物理资源提供的机制来实现这种“截获并重定向”，我们可以把虚拟化划分成为软件虚拟化和硬件虚拟化两种。

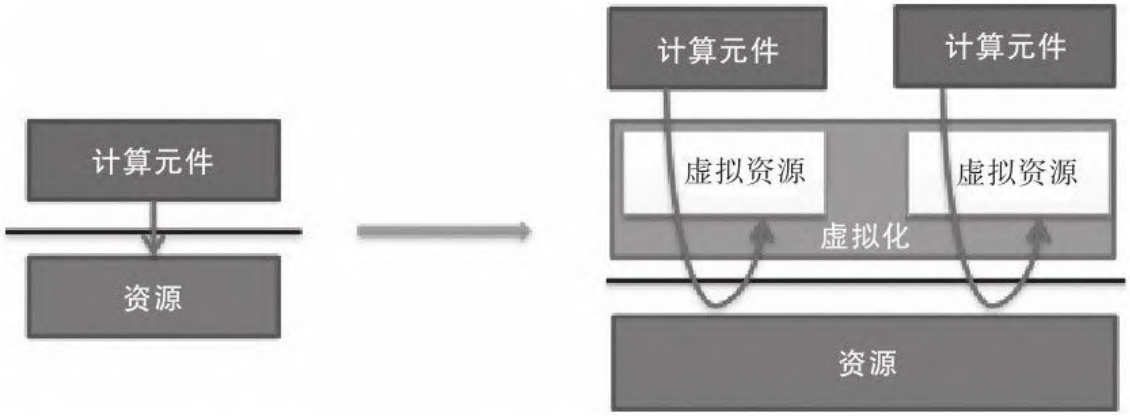


图　1-2　虚拟化的可实现性

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的技术。虚拟机监控机在可控制的范围内，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，但是也同时失去了跨平台虚拟化的能力。

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

硬件虚拟化，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。

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。

1.3.2 准虚拟化与全虚拟化

1.准虚拟化（para-virtualization）

如上一节所述，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，使它以为自己运行在虚拟环境下，能够与虚拟机监控机协同工作。这种方法就叫准虚拟化（para-virtualization），也叫半虚拟化。本质上，准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，准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[4] 。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。

2.全虚拟化（full virtualization）

与准虚拟化技术不同，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，是全虚拟化无可比拟的优势。

准虚拟化在诞生之初，运行在缺乏硬件虚拟化支持的平台上，操作系统和虚拟机监控器的密切配合，实现了针对全虚拟化软件方案的性能优势。这样的性能优势甚至一直保持到2006年左右（尽管那时已经出现了第一代硬件虚拟化平台）。但是随着硬件虚拟化技术的逐代演化，运行于Intel平台的全虚拟化的性能已经超过了准虚拟化产品，这一点在64位的操作系统上表现得更为明显。加之，全虚拟化不需要对客户机操作系统做任何修改的固有优势。可以预言，基于硬件的全虚拟化产品将是未来虚拟化技术的核心。

1.4 KVM简介

1.4.1 KVM的历史

KVM的全称是Kernel Virtual Machine，翻译成中文就是内核虚拟机。KVM虚拟机最初是由一个以色列的创业公司Qumranet开发的，作为他们的VDI产品的虚拟机。为了简化开发，KVM的开发人员并没有选择从底层开始新写一个Hypervisor，而是选择了基于Linux kernel，通过加载新的模块从而使Linux Kernel本身变成一个Hypervisor。2006年10月，在先后完成了基本功能、动态迁移以及主要的性能优化之后，Qumranet正式对外宣布了KVM的诞生。同年10月，KVM模块的源代码被正式接纳进入Linux Kernel，成为内核源代码的一部分。作为一个功能和成熟度都逊于Xen的项目，在这么快的时间内被内核社区接纳，主要原因在于：在虚拟化方兴未艾的当时，内核社区急于将虚拟化的支持包含在内，但是Xen取代内核由自身管理系统资源的架构引起了内核开发人员的不满和抵触。

在2008年9月4日，同内核社区保持着很深渊源的著名Linux发行版提供商——Redhat公司出人意料地出资1亿700百万美金，收购了Qumranet，从而成为了KVM开源项目的新东家。由于此次收购，Redhat公司有了自己的虚拟机解决方案，于是开始在自己的产品中用KVM替换

Xen。2010年11月，Redhat公司推出了新的企业版Linux——RHEL 6，在这个发行版中集成了最新的KVM虚拟机，而去掉了在RHEL 5.x系列中集成的Xen。

1.4.2 KVM功能概览

KVM是基于虚拟化扩展（Intel VT或AMD-V）的X86硬件，是Linux完全原生的全虚拟化解决方案。部分的准虚拟化支持，主要是通过准虚拟网络驱动程序的形式用于Linux和Windows客户机系统的。KVM目前设计为通过可加载的内核模块，支持广泛的客户机操作系统，比如Linux、BSD、Solaris、Windows、Haiku、ReactOS和AROS Research Operating System。

在KVM架构中，虚拟机实现为常规的Linux进程，由标准Linux调度程序进行调度。事实上，每个虚拟CPU显示为一个常规的Linux进程。这使KVM能够享受Linux内核的所有功能。

需要注意的是，KVM本身不执行任何模拟，需要用户空间程序通过/dev/kvm接口设置一个客户机虚拟服务器的地址空间，向它提供模拟的I/O，并将它的视频显示映射回宿主的显示屏。目前这个应用程序就是大名鼎鼎的QEMU。

图1-3显示了KVM的基本架构。

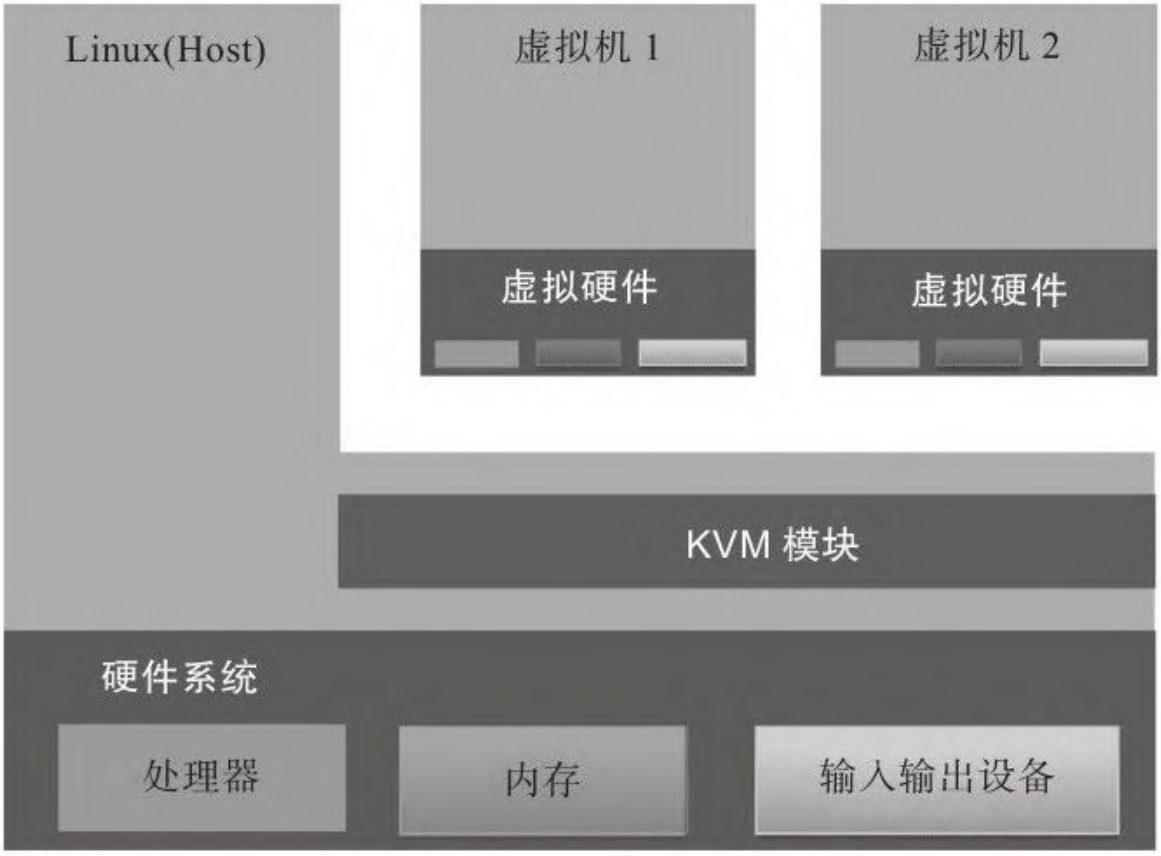


图　1-3　KVM架构

下面介绍一下KVM的功能特性。

（1）内存管理

KVM从Linux继承了强大的内存管理功能。一个虚拟机的内存与任何其他Linux进程的内存一样进行存储，可以以大页面的形式进行交换以实现更高的性能，也可以以磁盘文件的形式进行共享。NUMA支持（非一致性内存访问，针对多处理器的内存设计）允许虚拟机有效地访问大量内存。

KVM支持最新的基于硬件的内存虚拟化功能，支持Intel的扩展页表（EPT）和AMD的嵌套页表（NPT，也叫“快速虚拟化索引-RVI”），以实现更低的CPU利用率和更高的吞吐量。

内存页面共享通过一项名为内核同页合并（Kernel Same-page Merging，KSM）的内核功能来支持。KSM扫描每个虚拟机的内存，如果虚拟机拥有相同的内存页面，KSM将这些页面合并到一个在虚拟机之间共享的页面，仅存储一个副本。如果一个客户机尝试更改这个共享页面，它将得到自己的专用副本。

（2）存储

KVM能够使用Linux支持的任何存储来存储虚拟机镜像，包括具有IDE、SCSI和SATA的本地磁盘，网络附加存储（NAS）（包括NFS和SAMBA/CIFS），或者支持iSCSI和光纤通道的SAN。多路径I/O可用于改进存储吞吐量和提供冗余。由于KVM是Linux内核的一部分，它可以利用所有领先存储供应商都支持的一种成熟且可靠的存储基础架构，它的存储堆栈在生产部署方面具有良好的记录。

KVM还支持全局文件系统（GFS2）等共享文件系统上的虚拟机镜像，以允许虚拟机镜像在多个宿主之间共享或使用逻辑卷共享。磁盘镜像支持按需分配，仅在虚拟机需要时分配存储空间，而不是提前分

配整个存储空间，提高存储利用率。KVM的原生磁盘格式为QCOW2，它支持快照，允许多级快照、压缩和加密。

（3）设备驱动程序

KVM支持混合虚拟化，其中准虚拟化的驱动程序安装在客户机操作系统中，允许虚拟机使用优化的I/O接口而不使用模拟的设备，从而为网络和块设备提供高性能的I/O。KVM准虚拟化的驱动程序使用IBM和Red Hat联合Linux社区开发的VirtIO标准，它是一个与虚拟机管理程序独立的、构建设备驱动程序的接口，允许为多个虚拟机管理程序使用一组相同的设备驱动程序，能够实现更出色的虚拟机交互性。

（4）性能和可伸缩性

KVM也继承了Linux的性能和可伸缩性。KVM虚拟化性能在很多方面（如计算能力、网络带宽等）已经可以达到非虚拟化原生环境的95%以上的性能。KVM的扩展性也非常良好，客户机和宿主机都可以支持非常多的CPU数量和非常大量的内存。例如，Redhat官方文档[5] 就介绍过，RHEL 6.x系统中的一个KVM客户机可以支持160个虚拟CPU和多达2TB的内存，KVM宿主机支持4096个CPU核心和多达64TB的内存。

1.4.3 KVM的前景

尽管KVM是一个相对较新的虚拟机管理程序，但是诞生不久就被Linux社区接纳，成为随Linux内核发布的轻量型模块。与Linux内核集成，使KVM可以直接获益于最新的Linux内核开发成果，比如更好的进程调度支持、更广泛物理硬件平台的驱动、更高的代码质量，等等。

作为相对较新的虚拟化方案，KVM一直没有成熟的工具可用于管理KVM服务器和客户机，不过，现在随着libvirt、virt-manager等工具和OpenStack等云计算平台的逐渐完善，KVM管理工具在易用性方面的劣势已经逐渐被克服。另外，KVM仍然可以改进虚拟网络的支持、虚拟存储支持、增强的安全性、高可用性、容错性、电源管理、HPC/实时支持、虚拟CPU可伸缩性、跨供应商兼容性、科技可移植性等方面，不过，现在KVM开发者社区比较活跃，也有不少大公司的工程师参与开发，我们有理由相信很多功能都会在不远的将来得到完善。

1.5 Xen简介

1.5.1 Xen的历史

早在20世纪90年代，伦敦剑桥大学的Ian Pratt和Keir Fraser在一个叫做Xenoserver的研究项目中，开发了Xen虚拟机。作为Xenoserver的核心，Xen虚拟机负责管理和分配系统资源，并提供必要的统计功能。在那个年代，X86的处理器还不具备对虚拟化技术的硬件支持，所以Xen从一开始是作为一个准虚拟化的解决方案出现的。因此，为了支持多个虚拟机，内核必须针对Xen做出特殊的修改才可以运行。为了吸引更多开发人员参与，2002年Xen正式被开源。在先后推出了1.0和2.0版本之后，Xen开始被诸如Redhat、Novell和Sun的Linux发行版集成，作为其中的虚拟化解决方案。2004年，Intel的工程师开始为Xen添加硬件虚拟化的支持，从而为即将上市的新款处理器做必需的软件准备。在他们的努力下，2005年发布的Xen 3.0，开始正式支持Intel的VT技术和IA64架构，从而Xen虚拟机可以运行完全没有修改的操作系统。2007年10月，思杰公司出资5亿美金收购了XenSource，变成了Xen虚拟机项目的东家。

与Xen在功能开发上的快速进展形成对比的是，Xen在将它对内核的修改集成进入内核社区方面进展不大。有部分重要的内核开发人员

不喜欢Xen的架构和实现，多位内核维护人员公开声明不欢迎Xen。这样的形势一直持续到2010年，在基于内核的PVOPS对Xen做了大量重写之后，内核社区才勉强接纳了Xen。当然，目前从Linux 3.0版本开始的内核主干对Xen的支持还是越来越好了。

1.5.2 Xen功能概览

Xen是一个直接在系统硬件上运行的虚拟机管理程序。Xen在系统硬件与虚拟机之间插入一个虚拟化层，将系统硬件转换为一个逻辑计算资源池，Xen可将其中的资源动态地分配给任何操作系统或应用程序。在虚拟机中运行的操作系统能够与虚拟资源交互，就好像它们是物理资源一样。

图1-4显示了一个运行虚拟机的Xen系统。

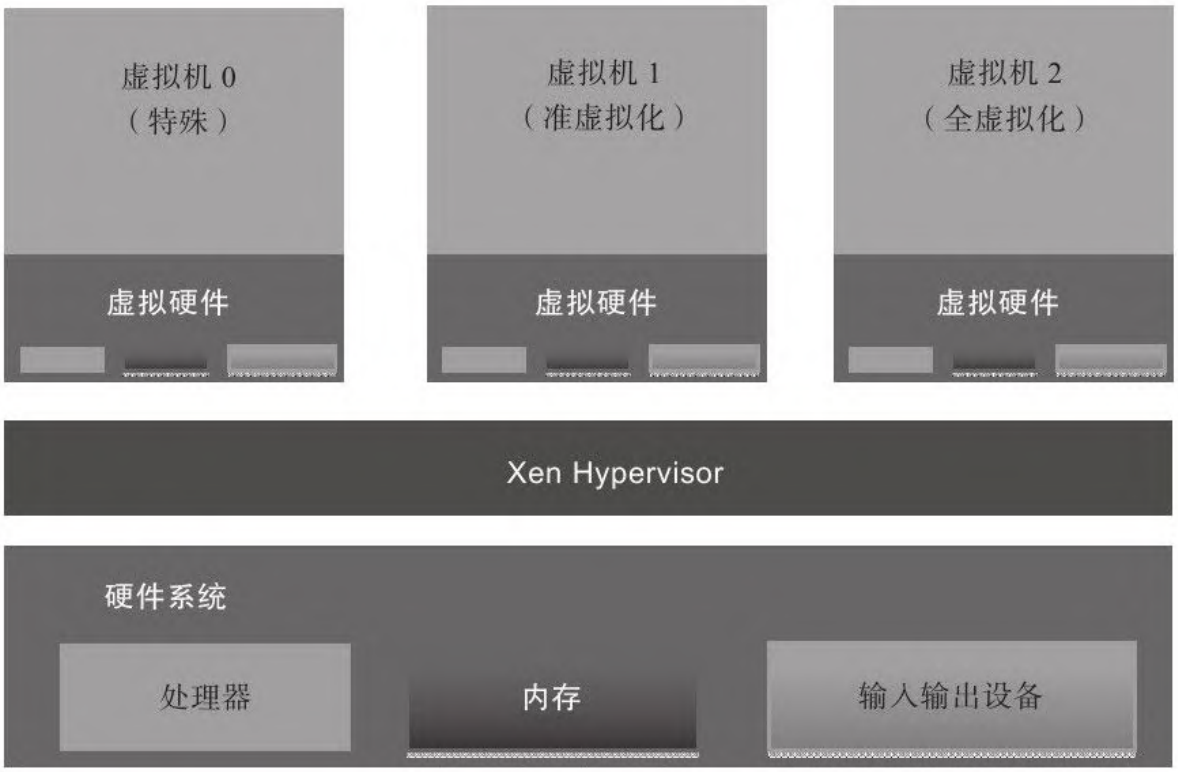


图　1-4　Xen架构

Xen被设计成为微内核的实现，其本身只是负责管理处理器和内存资源。Xen上面运行的所有虚拟机中，0号虚拟机是特殊的，其中运行的是经过修改的支持准虚拟化的Linux操作系统，大部分的输入输出设备都交由这个虚拟机直接控制，而Xen本身并不直接控制它们。这样做可以使基于Xen的系统可以最大程度地复用Linux内核的驱动程序。更广泛地说，Xen虚拟化方案在Xen Hypervisor和0号虚拟机的功能上做了聪明的划分，既能够重用大部分Linux内核的成熟代码，又可以控制系统之间的隔离型和针对虚拟机更加有效的管理和调度。通常，0号虚拟机也被视为是Xen虚拟化方案的一部分。

Xen上面运行的虚拟机，既支持准虚拟化，也支持全虚拟化，可以运行几乎所有可以在X86物理平台上运行的操作系统。此外，最新的Xen还支持ARM平台的虚拟化。

下面简单介绍一下Xen的功能特性。

Xen服务器（即思杰公司的Xen Server产品）构建于开源的Xen虚拟机管理程序之上，结合使用半虚拟化和硬件协助的虚拟化。操作系统与虚拟化平台之间的这种协作支持开发一个较简单的虚拟机管理程序来提供高度优化的性能。

Xen提供了复杂的工作负载平衡功能，可捕获CPU、内存、磁盘I/O和网络I/O数据，它提供了两种优化模式：一种针对性能，另一种针对密度。

Xen服务器利用一种名为Citrix Storage Link的独特的存储集成功能。使用Citrix Storage Link，系统管理员可直接利用来自HP、Dell Equal Logic、NetApp、EMC等公司的存储产品。

Xen服务器包含多核处理器支持、实时迁移、物理服务器到虚拟机转换（P2V）和虚拟到虚拟转换（V2V）工具、集中化的多服务器管理、实时性能监控，以及对Windows和Linux客户机的良好性能。

1.5.3 Xen的前景

Xen作为一个开发最早的虚拟化方案，对各种虚拟化功能的支持相对完善。Xen虚拟机监控程序，是一个专门为虚拟机开发的微内核，所以其资源管理和调度策略完全是针对虚拟机的特性而开发的。作为一个独立维护的微内核，Xen的功能明确，开发社区构成比较简单，所以更容易接纳专门针对虚拟化所做的功能和优化。

Xen比较难于配置和使用，部署会占用相对较大的空间，而且非常依赖于0号虚拟机中的Linux操作系统。Xen微内核直接运行于真实物理硬件之上，开发和调试都比基于操作系统的虚拟化困难。Xen最大的困难在于Linux内核社区的抵制，导致Xen相关的内核改动一直不能顺利进入内核源代码，从而无法及时得到内核最新开发成果的支持，这与KVM形成了鲜明的对比。

1.6 其他虚拟化方案简介

1.4节和1.5节分别介绍了两个最有名的开源虚拟机，开放源代码的好处在于可以有人数庞大的开发社区作为支撑，同时源代码公开也有利于人们研究和学习虚拟机的具体实现。但是，把虚拟机作为商品出售给终端用户，就需要一些商业化的虚拟机解决方案。下面简单介绍常见的一些商业化的虚拟机。

1.6.1 VMware

VMware公司创办于1998年，从公司的名字就可以看出，这是一家专注于提供虚拟化解决方案的公司。VMware公司很早就预见到了虚拟化在未来数据中心中的核心地位，有针对性地开发虚拟化软件，从而抓住了21世纪初虚拟化兴起的大潮，成为了虚拟化业界的标杆。
VMware公司从创建至今，一直占据着虚拟化软件市场的最大份额，是毫无争议的龙头老大。Vmware公司作为最成熟的商业虚拟化软件提供商，其产品线是业界覆盖范围最广的，下面会对VMware的主要产品进行简单的介绍。

（1）VMware Workstation

VMware Workstation是VMware公司销售的运行于台式机和工作站上的虚拟化软件，也是VMware公司第一个面市的产品（1999年5月）。该产品最早采用了VMware在业界知名的二进制翻译技术，在x86 CPU硬件虚拟化技术还未出现之前，为客户提供了纯粹的基于软件的全虚拟化解决方案。作为最初的拳头产品，VMware公司投入了大量的资源对二进制翻译进行优化，其二进制翻译技术带来的虚拟化性能甚至超过第一代的CPU硬件虚拟化产品。该产品如同KVM，是“类型二”虚拟机[6]，需要在宿主操作系统之上运行。

（2）VMware ESX Server

ESX服务器（一种能直接在硬件上运行的企业级的虚拟平台），虚拟的SMP，它能让一个虚拟机同时使用四个物理处理器，和VMFS一样，它能使多个ESX服务器分享块存储器。该公司还提供一个虚拟中心来控制和管理虚拟化的IT环境：VMotion让用户可以移动虚拟机器；DRS从物理处理器创造资源工具；HA提供从硬件故障自动恢复功能；综合备份可使LAN-free自动备份虚拟机器；VMotion存储器可允许虚拟机磁盘自由移动；更新管理器自动更新修补程序和更新管理；能力规划能使VMware的服务供应商执行能力评估；转换器把本地和远程物理机器转换到虚拟机器；实验室管理可自动化安装、捕捉、存储和共享多机软件配置；ACE允许桌面系统管理员对虚拟机应用统一的企业级IT安全策略，以防止不可控台式电脑带来的风险。虚拟桌面基础设施可主导个

人台式电脑在虚拟机运行的中央管理器；虚拟桌面管理，它是联系用户到数据库中的虚拟电脑的桌面管理服务器；WMware生命管理周期可通过虚拟环境提供控制权。

1.6.2 VirtualBox

Oracle VirtualBox是由德国InnoTek软件公司出品的虚拟机器软件，现在由甲骨文公司进行开发，是甲骨文公司xVM虚拟化平台技术的一部份。它提供使用者在32位或64位的Windows、Solaris及Linux操作系统上虚拟其他X86的操作系统。使用者可以在VirtualBox上安装并执行Solaris、Windows、DOS、Linux、OS/2 Warp、OpenBSD及FreeBSD等系统作为客户端操作系统。最新的VirtualBox还支持运行Android4.0系统。

与同性质的VMware及Virtual PC比较下，VirtualBox独到之处包括远端桌面协定（RDP）、iSCSI及USB的支援，VirtualBox在客户机操作系统上已可以支援USB 2.0的硬件装置。此外，VirtualBox还支持在32位宿主操作系统上运行64位的客户机操作系统。

VirtualBox既支持纯软件虚拟化，也支持Intel VT-x与AMD AMD-V硬件虚拟化技术。为了方便其他虚拟机用户向VirtualBox的迁移，VirtualBox可以读写VMware VMDK格式与VirtualPC VHD格式的虚拟磁盘文件。

1.6.3 Hyper-V

Hyper-V是微软提出的一种系统管理程序虚拟化技术。Hyper-V设计的目的是为广泛的用户提供更为熟悉及成本效益更高的虚拟化基础设施软件，这样可以降低运作成本、提高硬件利用率、优化基础设施并提高服务器的可用性。

Hyper-V的设计借鉴了Xen，采用微内核的架构，兼顾了安全性和性能的要求。Hyper-V底层的Hypervisor运行在最高的特权级别下，微软将其称为ring-1（而Intel也将其称为root mode），而虚拟机的操作系统内核和驱动运行在ring 0，应用程序运行在ring 3。

Hyper-V采用基于VMbus的高速内存总线架构，来自虚拟机的硬件请求（显卡、鼠标、磁盘、网络），可以直接经过VSC，通过VMbus总线发送到根分区的VSP，VSP调用对应的设备驱动，直接访问硬件，中间不需要Hypervisor的帮助。

从架构上讲，Hyper-V只有“硬件-Hyper-V-虚拟机”三层，本身非常小巧，代码简单，且不包含任何第三方驱动，所以安全可靠、执行效率高，能充分利用硬件资源，使虚拟机系统性能更接近真实系统性能。

1.7　本章小结

本章从云计算的概念和发展历史开始，逐渐为读者理清了云计算和虚拟化技术之间的联系：虚拟化技术是实现云计算的基础。

本章还简单介绍了虚拟化技术的本质，以及常见虚拟化技术的分类。作为重点，以KVM和Xen为例，重点介绍了这两个开源社区最有名的虚拟机。这些内容为第2章继续展开深入了解KVM的架构做了必要的知识铺垫。

最后，对常见的商业化虚拟解决方案做了简单的描述，供有兴趣的读者作为参考。

1.8 本章注释

[1]SETI@home科学实验，可参考其网站主页：http://setiathome.berkeley.edu/

[2]Google Docs服务从2012年4月开始已经被合并到新推出的Google Drive服务之中。Google Drive是Google提供的文件存储和同步服务，其网站地址为：https://drive.google.com/

[3]维基百科中对云计算的定义也在不断变化中，见：http://en.wikipedia.org/wiki/Cloud_computing.

[4]准确地讲，Xen最初是从准虚拟化发展起来的，但目前也能很好地利用硬件虚拟化技术实现全虚拟化，可以创建未经修改过的客户机。

[5]Redhat关于Redhat Linux中对于包括KVM在内的Hypervisor的限制，官方文档的网页为：http://www.redhat.com/resourcelibrary/articles/virtualization-limits-rhel-hypervisors.

[6]关于“类型一”和“类型二”虚拟机分类的标准，可参考2.3节中的介绍。

第2章　KVM原理简介

2.1　Linux操作系统简介

Linux是一种自由和开放源代码的类UNIX操作系统内核。目前存在着许多不同的Linux发布版，可安装在各种各样的电脑硬件设备上，从手机、平板电脑、路由器和影音游戏控制台，到台式机、大型机和超级电脑。Linux是一个领先的操作系统内核，世界上运算最快的10台超级电脑运行的都是基于Linux内核的操作系统。

Linux操作系统也是自由软件和开放源代码发展中最著名的例子。只要遵循GNU通用公共许可证，任何人和机构都可以自由地使用Linux的所有底层源代码，也可以自由地修改和再发布。严格来讲，Linux这个词本身只表示Linux内核，但实际上人们已经习惯了用Linux来表示整个基于Linux内核且使用GNU工程各种工具构建的操作系统（也被称为GNU/Linux）。在通常情况下，Linux被打包成供台式机和服务器使用的Linux发布版本。

1.Linux的历史

1991年，芬兰人Linus Torvalds在赫尔辛基大学上学，对操作系统很好奇，热衷于使用MINIX（一种教学使用的廉价UNIX）。但是，

Linus对MINIX只允许在教育上使用很不满（不允许任何商业使用），于是开始写他自己的操作系统，这就是后来的Linux内核。

Linus开始在MINIX上开发Linux内核，为MINIX写的软件也可以在Linux内核上使用。后来Linux成熟了，可以在Linux上面开发自身的Linux内核。为了让Linux可以在商业上使用，Linus Torvalds决定在Linux中改变MINIX的协议（这个协议会限制商业使用)，使用GNU GPL协议来代替。开发者致力于融合GNU元素到Linux中，做出一个有完整功能的、自由的操作系统。

Linux渐渐得以广泛使用，并赢得了众多的用户。Linux成功的一个重要因素是，它很快吸引了很多开发者，这些开发人员对其代码进行了修改和完善。直至今日，Linux仍然是一个基于因特网的协作开发项目。Linus作为Linux之父，现在仍是一位内核维护者，但是开发工作主要是由一个结构松散的内核社区上的开发人员共同完成的。随着Linux的商业前景逐渐被看好，计算机行业巨头（如IBM、Intel等）都投入自己的开发人员与内核社区一起开发和改进Linux。

2.模块化设计的Linux

操作系统内核设计一直分为两个阵营：微内核和单内核。

单内核是两大阵营中一种较为简单的设计，指的是整个内核从整体上作为一个单独的大过程来实现，并且同时运行在一个单独的地址

空间内。所有的内核服务都在这样一个大的内核空间运行，内核之间的通信可以简单地实现为函数调用。这样的设计具有简单高效的特点。但是，如果使用单内核设计，每次对内核作出修改（比如增加或者删除驱动程序），都必须重新编译源代码，生成新的二进制文件，造成了使用和部署上的麻烦。

微内核并不是作为一个单独的大过程来实现的，相反，内核的功能被划分成为多个独立的过程，每一个过程叫做一个服务器。多个服务器程序都运行在自己的地址空间，只有少量核心的服务器运行在特权模式下，服务器之间的通信采用了进程间通信机制。独立的服务器进程提高系统的健壮性，但是进程间通信由于涉及内核空间和用户空间的上下文切换，其开销远比函数调用大得多。

Linux采用了实用主义的设计。为了满足性能要求，Linux内核被设计成单内核。但是，Linux内核同时借鉴了微内核的精华：模块化设计以及动态装载内核模块的能力。除了诸如进程切换、内存管理等核心的内核功能，将大部分内核功能作为单独的内核模块设计并实现。这些内核模块编译好后以单独的二进制文件的形式存在，内核在运行过程中，按照需求，动态地加载并链接进入内核空间运行。不使用的模块还可以在运行的过程中动态卸载。这样的设计，既保证了内核的性能，也改进了传统单内核设计的灵活性。

Linux内核的开放源代码特性及模块化的设计，使其成为很多开发人员和计算机专业学生研究操作系统内核的典型范例。开发人员可以充分利用Linux内核已经实现的成熟功能，在其基础上设计、实现自己的内核模块，以此来扩充内核的功能，满足自己的要求。例如，KVM就是以内核模块的形式存在，为Linux内核增加了虚拟化的功能。

2.2 虚拟化模型

通过1.3节的阐述，大家应该已经建立了虚拟化的基本概念。在这一节，将针对当下最流行的X86平台的虚拟化做进一步的详述。这部分的知识也是理解KVM的必要准备。从本节开始，使用“虚拟化”一词时，如果没有特殊说明，都是指X86平台的虚拟化。

图2-1为大家展示了基本的虚拟化模型。

图 2-1 虚拟化模型

处于底层是整个物理系统，也就是我们平常看得见、摸得着的系统硬件，主要包括处理器、内存和输入输出设备（这一点相信有主机

DIY经验的读者是非常熟悉的）。

在物理系统之上，与以往熟悉的操作系统模型不同，运行的是虚拟机监控器（缩写为VMM或Hypervisor）。虚拟机监控器的主要职能是：管理真实的物理硬件平台，并为每个虚拟客户机提供对应的虚拟硬件平台。

图2-1中绘制了3个虚拟机的实例，每个虚拟机看起来就像是一个小的但是完整的计算机系统，具有自己的“系统硬件”，包括自己的处理器、内存和输入输出设备。在这个计算机系统上，运行着虚拟机自己的操作系统，例如Linux和Windows。

有读者或许会问，既然每一个虚拟机看起来就是一个小的“真实”计算机系统，那里面可以运行自己的虚拟机监控器吗？答案是肯定的。这种情况一般称为“嵌套虚拟化”。KVM支持嵌套虚拟化技术，只是嵌套虚拟化的实现还远远没有达到很稳定和成熟的状态。嵌套虚拟化并不是本章要讲述的内容。

一个X86平台的核心是其中的处理器，处理器运行程序代码，访问内存和输入输出设备。所以，X86平台虚拟化技术的核心部分是处理器的虚拟化。只要处理器虚拟化技术支持“截获并重定向”，内存和输入输出设备的虚拟化都可以基于处理器虚拟化技术之上实现。在处理器虚拟化技术的基础上，为了增强虚拟机的性能，内存虚拟化和IO虚

拟化的新技术也不断被加入到X86平台虚拟化技术中。X86平台虚拟化技术从开始单一的处理器开始，逐步牵涉芯片组、网卡、存储设备以及GPU的虚拟化。在2.6节，我们将以Intel硬件平台为例，详细阐述X86平台的硬件虚拟化相关技术的演进。

2.3 KVM架构

了解了虚拟化的基本模型之后，接下来，我们来看一下KVM的具体架构。

从虚拟机的基本架构上来区分，虚拟机一般分为两种，我们称之为类型一和类型二。

其中，“类型一”虚拟机是在系统上电之后首先加载运行虚拟机监控程序，而传统的操作系统则是运行在其创建的虚拟机中。类型一的虚拟机监控程序，从某种意义上说，可以视为一个特别为虚拟机而优化裁剪的操作系统内核。因为，虚拟机监控程序作为运行在底层的软件层，必须实现诸如系统的初始化、物理资源的管理等操作系统的职能；它对虚拟机的创建、调度和管理，与操作系统对进程的创建、调度和管理有共通之处。这一类型的虚拟机监控程序一般会提供一个具有一定特权的特殊虚拟机，由这个特殊虚拟机来运行需要提供给用户日常操作和管理使用的操作系统环境。著名的开源虚拟化软件Xen、商业软件VMware ESX/ESXi和微软的Hyper-V就是“类型一”虚拟机的代表。

与“类型一”虚拟机的方式不同，“类型二”虚拟机监控程序，在系统上电之后仍然运行一般意义上的操作系统（也就是俗称的宿主

机操作系统），虚拟机监控程序作为特殊的应用程序，可以视作操作系统功能的扩展。对于“类型二”的虚拟机来说，其最大的优势在于可以充分利用现有的操作系统。因为虚拟机监控程序通常不必自己实现物理资源的管理和调度算法，所以实现起来比较简洁。但是，正所谓“成也萧何，败也萧何”，这一类型的虚拟机监控程序既然依赖操作系统来实现管理和调度，就同样也会受到宿主操作系统的一些限制。例如，通常无法仅仅为了虚拟机的优化，而对操作系统作出修改。本书的主角KVM就是属于“类型二”虚拟机，另外，VMware Workstation、VirtualBox也是属于“类型二”虚拟机。

了解了基本的虚拟机架构之后，让我们来看一下如图2-2所示的KVM的基本架构。显而易见，KVM是一个基于宿主操作系统的类型二虚拟机。在这里，我们再一次看到了实用至上的Linux设计哲学，既然类型二的虚拟机是最简洁和容易实现的虚拟机监控程序，那么就通过内核模块的形式实现出来就好。其他的部分则尽可能充分利用Linux内核的既有实现，最大限度地重用代码。

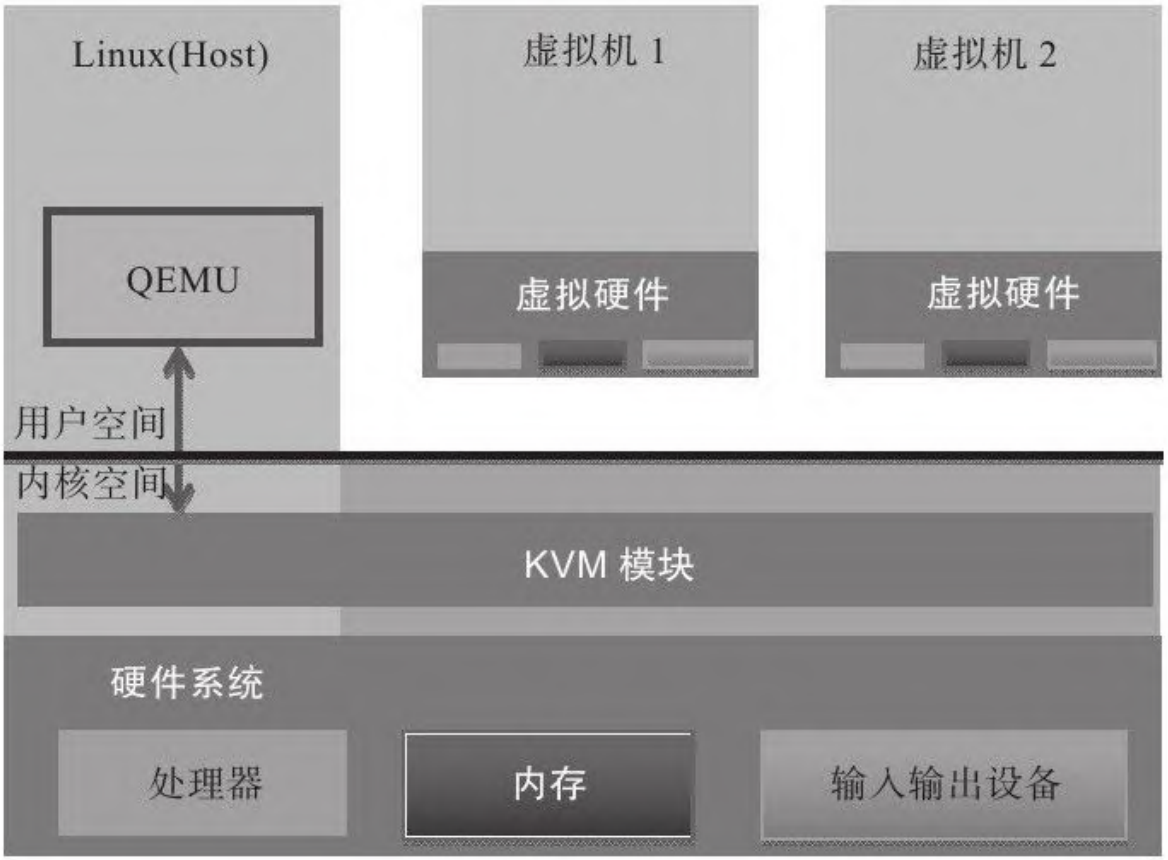


图　2-2　KVM架构

在图2-2中，左侧部分是一个标准的Linux操作系统，可以是RHEL、Fedora、Ubuntu等。KVM内核模块在运行时按需加载进入内核空间运行。KVM本身不执行任何设备模拟，需要用户空间程序QEMU通过/dev/kvm接口设置一个虚拟客户机的地址空间，向它提供模拟的I/O设备，并将它的视频显示映射回宿主机的显示屏。

下面两小节将专门对KVM模块和QEMU这两个组成部分做简单的介绍。

2.4 KVM模块

KVM模块是KVM虚拟机的核心部分。其主要功能是初始化CPU硬件，打开虚拟化模式，然后将虚拟客户机运行在虚拟机模式下，并对虚拟客户机的运行提供一定的支持。

为了软件的简洁和性能，KVM仅支持硬件虚拟化。自然而然，打开并初始化系统硬件以支持虚拟机的运行，是KVM模块的职责所在。以KVM在Intel公司的CPU上运行为例，在被内核加载的时候，KVM模块会先初始化内部的数据结构；做好准备之后，KVM模块检测系统当前的CPU，然后打开CPU控制寄存器CR4中的虚拟化模式开关，并通过执行VMXON指令将宿主操作系统（包括KVM模块本身）置于虚拟化模式中的根模式；最后，KVM模块创建特殊设备文件/dev/kvm并等待来自用户空间的命令。接下来虚拟机的创建和运行将是一个用户空间的应用程序（QEMU）和KVM模块相互配合的过程。

KVM模块与用户空间QEMU的通信接口主要是一系列针对特殊设备文件的IOCTL调用。

如上所述，KVM模块加载之初，只存在/dev/kvm文件，而针对该文件的最重要的IOCTL调用就是“创建虚拟机”。在这里，“创建虚拟机”可以理解成KVM为了某个特定的虚拟客户机（用户空间程序创建并

初始化）创建对应的内核数据结构。同时，KVM还会返回一个文件句柄来代表所创建的虚拟机。针对该文件句柄的IOCTL调用可以对虚拟机做相应的管理，比如创建用户空间虚拟地址和客户机物理地址及真实内存物理地址的映射关系，再比如创建多个可供运行的虚拟处理器（vCPU）。同样，KVM模块会为每一个创建出来的虚拟处理器生成对应的文件句柄，对虚拟处理器相应的文件句柄进行相应的IOCTL调用，就可以对虚拟处理器进行管理。

针对虚拟处理器的最重要的IOCTL调用就是“执行虚拟处理器”。通过它，用户空间准备好的虚拟机在KVM模块的支持下，被置于虚拟化模式中的非根模式下，开始执行二进制指令。在非根模式下，所有敏感的二进制指令都会被处理器捕捉到，处理器在保存现场之后自动切换到根模式，由KVM决定如何进一步处理（要么由KVM模块直接处理，要么返回用户空间交由用户空间程序处理）。

除了处理器的虚拟化，内存虚拟化也是由KVM模块实现的。

实际上，内存虚拟化往往是一个虚拟机实现中代码量最大、实现最复杂的部分（至少，在硬件支持二维地址翻译之前是这样的）。众所周知，处理器中的内存管理单元（MMU）是通过页表的形式将程序运行的虚拟地址转换成为物理内存地址。在虚拟机模式下，内存管理单元的页表则必须在一次查询的时候完成两次地址转换。这是因为，除了要将客户机程序的虚拟地址转换成为客户机物理地址以外，还必须

将客户机物理地址转换成为真实物理地址。KVM模块开始使用了影子页表的技术来解决这个问题：在客户机运行时候，处理器真正使用的页表并不是客户机操作系统维护的页表，而是KVM模块根据这个页表维护的另外一套影子页表。影子页表的机制比较复杂，感兴趣的读者可以自行翻阅相关材料，这里不再展开详述。

影子页表实现复杂，而且有时候开销很大。为了解决这个问题，新的处理器在硬件上做了增强（Intel的EPT技术）。通过引入第二级页表来描述客户机虚拟地址和真实物理地址的转换，硬件可以自动进行两级转换生成正确的内存访问地址。KVM模块将其称为二维分页机制。

处理器对设备的访问主要是通过IO指令和MMIO，其中IO指令会被处理器直接截获，MMIO会通过配置内存虚拟化来捕捉。但是，外设的模拟一般并不由KVM模块负责。一般来说，只有对性能要求比较高的虚拟设备才会由KVM内核模块来直接负责，比如虚拟中断控制器和虚拟时钟，这样可以大量减少处理器的模式切换的开销。大部分的输入输出设备还是会交给下一节将要介绍的用户态程序QEMU来负责。

2.5 QEMU设备模型

QEMU本身并不是KVM的一部分，其自身就是一个著名的开源虚拟机软件。与KVM不同，QEMU虚拟机是一个纯软件的实现，所以性能低下。但是，其优点是在支持QEMU本身编译运行的平台上就可以实现虚拟机的功能，甚至虚拟机可以与宿主机并不是同一个架构。作为一个存在已久的虚拟机，QEMU的代码中有整套的虚拟机实现，包括处理器虚拟化、内存虚拟化，以及KVM使用到的虚拟设备模拟（比如网卡、显卡、存储控制器和硬盘等）。

为了简化开发和代码重用，KVM在QEMU的基础上进行了修改。虚拟机运行期间，QEMU会通过KVM模块提供的系统调用进入内核，由KVM模块负责将虚拟机置于处理器的特殊模式运行。遇到虚拟机进行输入输出操作，KVM模块会从上次的系统调用出口处返回QEMU，由QEMU来负责解析和模拟这些设备。

从QEMU角度来看，也可以说QEMU使用了KVM模块的虚拟化功能，为自己的虚拟机提供硬件虚拟化的加速，从而极大地提高了虚拟机的性能。除此之外，虚拟机的配置和创建，虚拟机运行依赖的虚拟设备，虚拟机运行时的用户操作环境和交互，以及一些针对虚拟机的特殊技术（诸如动态迁移），都是由QEMU自己实现的。

从QEMU和KVM模块之间的关系可以看出，这是典型的开源社区在代码共用和开发项目共用上面的合作。诚然，QEMU可以选择其他的虚拟机或技术来加速，比如Xen或者KQEMU；KVM也可以选择其他的用户空间程序作为虚拟机实现，只要它按照KVM提供的API来设计。但是在现实中，QEMU与KVM两者的结合是最成熟的选择，这对一个新开发和后起的项目（KVM）来说，无疑多了一份未来成功的保障。

2.6 Intel虚拟化技术

Intel虚拟化技术其实是一系列硬件技术的集合，虚拟机监控机软件通过选择利用各项技术，从而提高虚拟化软件的性能或者实现各种不同的功能。

如图2-3所示，Intel虚拟化技术其实可以大致分为三类：第一类是处理器相关的，称为VT-x，是实现处理器虚拟化的硬件扩展，这也是硬件虚拟化的基础；第二类是芯片组相关的，成为VT-d，是从芯片组的层面为虚拟化提供必要支持，通过它，可以实现诸如直接分配物理设备给客户机的功能；第三类是输入输出设备相关的，主要目的是通过定义新的输入输出协议，使新一代的输入输出设备可以更好地支持虚拟化环境下的工作，比如Intel网卡自有的VMDq技术和PCI组织定义的单根设备虚拟化协议（SR-IOV）。

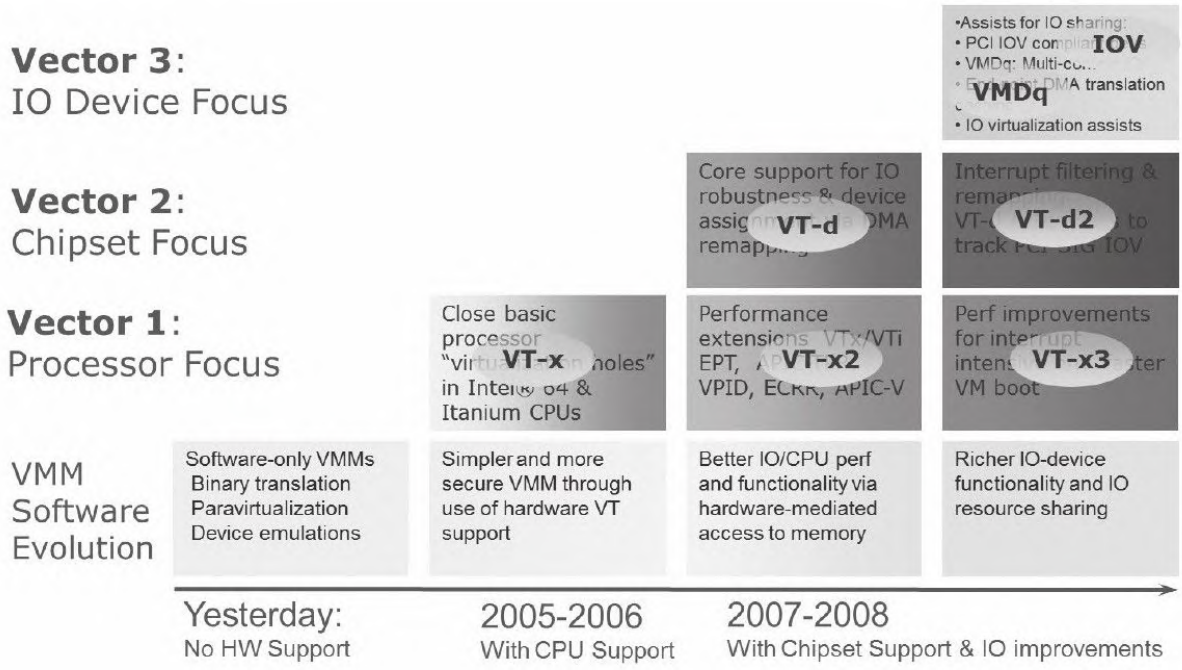


图　2-3　Intel虚拟化技术进化蓝图

从图2-3中可以看出，硬件虚拟化技术发展到今天，已经从最初的处理器虚拟化扩展变成了各种先进技术的集合。而且，Intel公司还在继续为这个集合添加新的成员。同时Intel公司的软件开发工程师也一直活跃在开源虚拟机社区，为KVM即时添加最新的硬件支持。对这方面感兴趣的读者，可以自己去下载KVM和QEMU的源代码，从而进行更深入的理解和应用。

2.7 本章小结

本章首先简单介绍了Linux操作系统和内核，接着引入了X86平台虚拟化的基本模型。在此基础之上，本章重点阐述了KVM的技术架构及组成要素，并对每一个组成要素做了简单介绍。最后，介绍了KVM实现所依赖的硬件虚拟化技术进化蓝图。

本章主要是为读者介绍KVM的架构、基本组成部分和功能，使读者对KVM建立一个初步的技术印象，以便读者可以更容易理解后面章节中关于KVM虚拟化的具体内容，另外也可以让在读者阅读QEMU/KVM等代码时有一个整体的概念。

第3章　构建KVM环境

在通过第2章了解KVM的基本原理之后，你是否迫不及待地想实践一下如何使用KVM来构建自己的虚拟化环境呢？那么，请仔细阅读本章吧！

本章将介绍如何通过整套的流程与方法来构建KVM环境，其中包括：硬件系统的配置、宿主机（Host）操作系统的安装、KVM的编译与安装、qemu-kvm的编译与安装、客户机（Guest）的安装，直到最后启动属于你的第一个KVM客户机。

3.1　硬件系统的配置

我们知道，KVM从诞生伊始就是需要硬件虚拟化扩展的支持，所以这里需要特别讲解一下硬件系统的配置。

KVM最初始的开发是基于x86和x86-64处理器架构上的Linux系统进行的，目前，KVM被移植到多种不同处理器架构之上，包括：Intel和HP的IA64（安腾）架构、AIM联盟（Apple-IBM-Motorola）的PowerPC架构、IBM的S/390架构、ARM架构（2012年开始[1]）。其中，在X86-64上面的功能支持最完善（主要原因是Intel/AMD的x86-64架构在桌面和服务器市场上的主导地位及其架构的开放性，以及它的开发者众

多），本书也采用基于Intel x86-64架构的处理器作为基本的硬件环境[2]。

在x86-64架构的处理器中，KVM必需的硬件虚拟化扩展分别为：Intel的虚拟化技术（Intel VT）和AMD的AMD-V技术。其中，Intel在2005年11月发布的奔腾四处理器（型号：662和672）第一次正式支持VT技术（Virtualization Technology），之后不久的2006年5月AMD也发布了支持AMD-V的处理器。现在比较流行的针对服务器和桌面的Intel处理器多数都是支持VT技术的，本节着重讲述英特尔的VT技术相关的硬件设置。

首先处理器（CPU）要在硬件上支持VT技术，还要在BIOS中将其功能打开，KVM才能使用到。目前，多数流行的服务器和部分桌面处理器的BIOS都默认将VT打开了。

一般在BIOS中，VT的选项通过"Advanced→Processor Configuration"来查看和设置，它的标识通常为"Intel(R)Virtualization Technology"或"Intel VT"等类似的文字说明。

除了支持必需的处理器虚拟化扩展以外，如果服务器芯片还支持VT-d，就建议在BIOS中将其打开，因为后面一些相对高级的设备的直接分配功能会需要硬件VT-d技术的支持。VT-d（Virtualization

Technology for Directed I/O）是对设备I/O的虚拟化硬件支持，在BIOS中的位置可能为"Advanced→Processor Configuration"或"Advanced→System Agent(SA)Configuration"，它一般在BIOS中的标志一般为"Intel(R)VTfor Directed I/O"或"Intel VT-d"等。

下面以一台Intel Romley-EP平台的服务器为例来说明在BIOS中的设置。

1）BIOS中的Advanced选项，如图3-1所示。

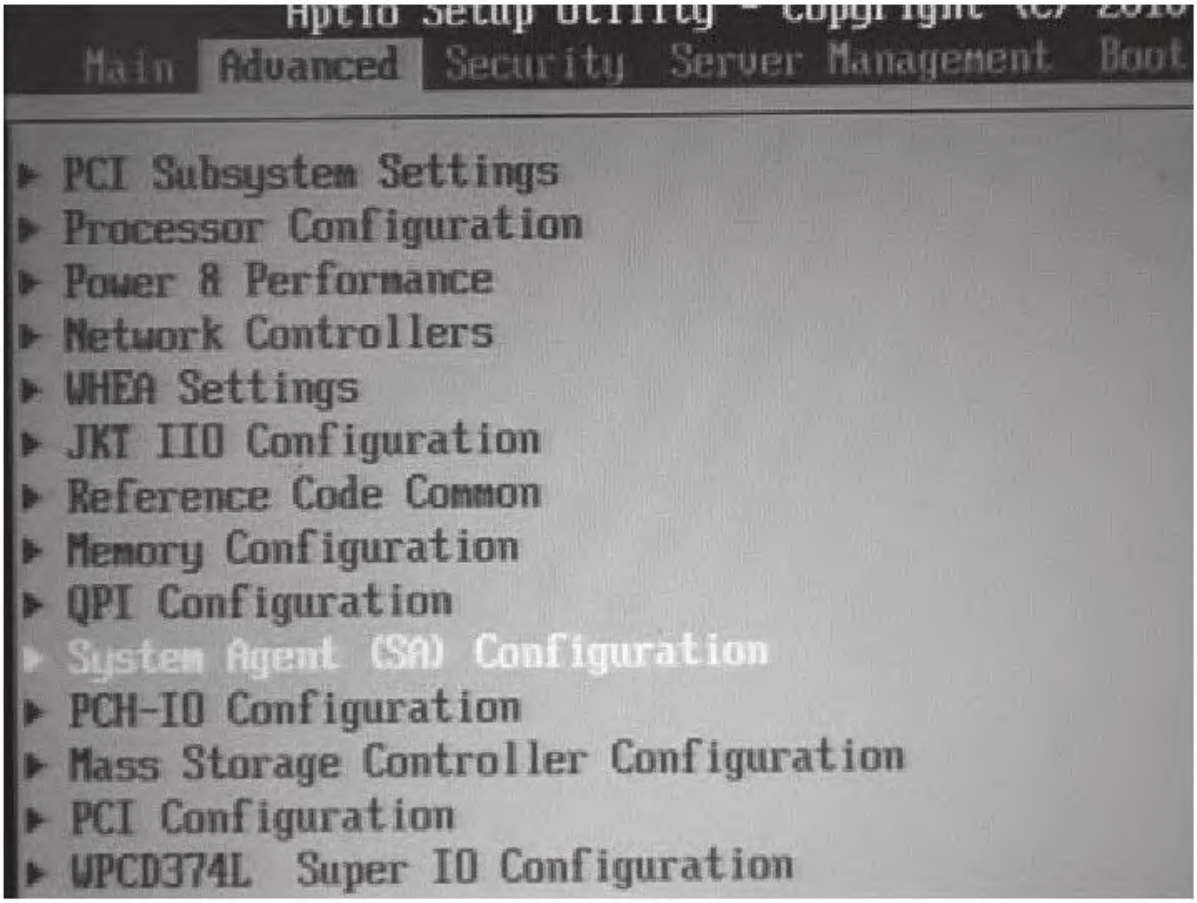


图　3-1　BIOS中的Advanced选项

2）BIOS中的VT和VT-d选项，如图3-2所示。

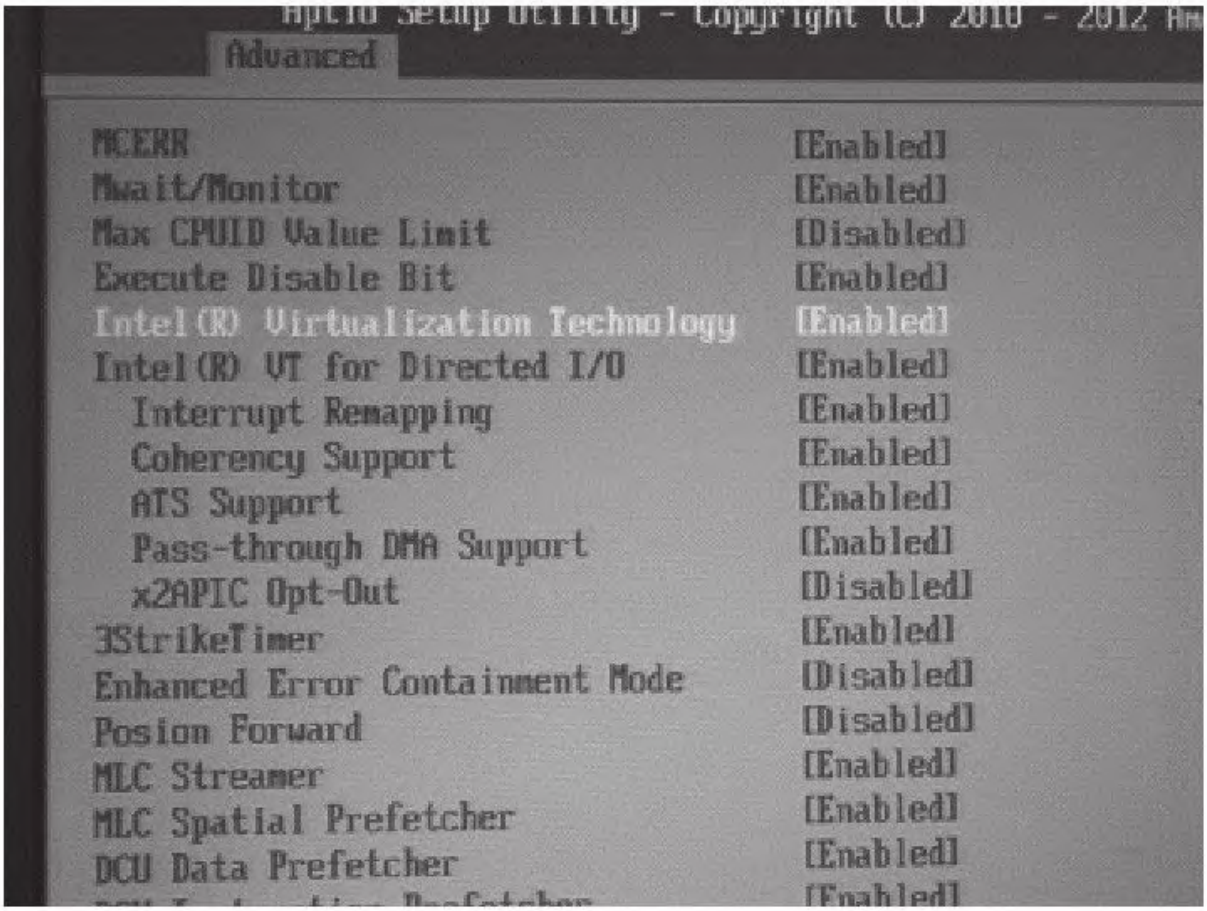


图　3-2　BIOS中Enabled的VT与VT-d选项

对于不同平台或不同厂商的BIOS，VT和VT-d等设置的位置可能是不一样的，需要根据实际的硬件情况和BIOS中的选项来灵活设置。

设置好了VT和VT-d的相关选项，保存BIOS的设置并退出，系统重启后生效。在Linux系统中，可以通过检查/proc/cpuinfo文件中的CPU

特性标志（flags）来查看CPU目前是否支持硬件虚拟化。在x86和x86-64平台中，Intel系列CPU支持虚拟化的标志为"vmx"，AMD系列CPU的标志为"svm"，所以可以用如下命令行查看"vmx"或者"svm"标志。

```
[root@jay-linux ~]# grep -E '(vmx|svm)' /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt
tsc_deadline_timer aes xsave avx lahf_lm ida arat epb xsaveopt pln pts
dts tpr_shadow vnmi flexpriority ept vpid
<!-- 此处省略多行其余 CPU 或 core 的 flags 输出信息 -->
```

3.2　安装宿主机Linux系统

KVM是基于内核的虚拟化技术，要运行KVM虚拟化环境，安装一个Linux操作系统的宿主机（Host）是必需的。由于Redhat公司是目前对KVM项目投入最多的企业之一，从RHEL6开始，其系统自带的虚拟化方案就采用了KVM，而且RHEL（Red Hat Enterprise Linux）也是最流行的企业级Linux发行版之一，所以本节选用RHEL来讲解Linux系统的安装步骤和过程，并且本章后面的编译和运行都是在这个系统上进行的。

当然，KVM作为流行的开源虚拟机之一，它可以在绝大多数流行的Linux系统上编译和运行，所以依然可以选择RHEL之外的其他Linux发行版，CentOS、Fedora、Ubuntu、Debian、OpenSuse等系统都是不错的选择。

本节内容基于目前最新的RHEL版本——RHEL6.3 Server版的系统来简单介绍，普通Linux安装的基本过程这里就不再详细描述，这里主要说明安装过程中一些值得注意的地方。

在选择哪种类型的服务器时，选择"Software Development Workstation"即可（如图3-3所示），然后选中当前页面的"Customize

now"，单击"Next"按钮进入下一步去选择具体需要安装的组件并设置所需要的各个RPM包。

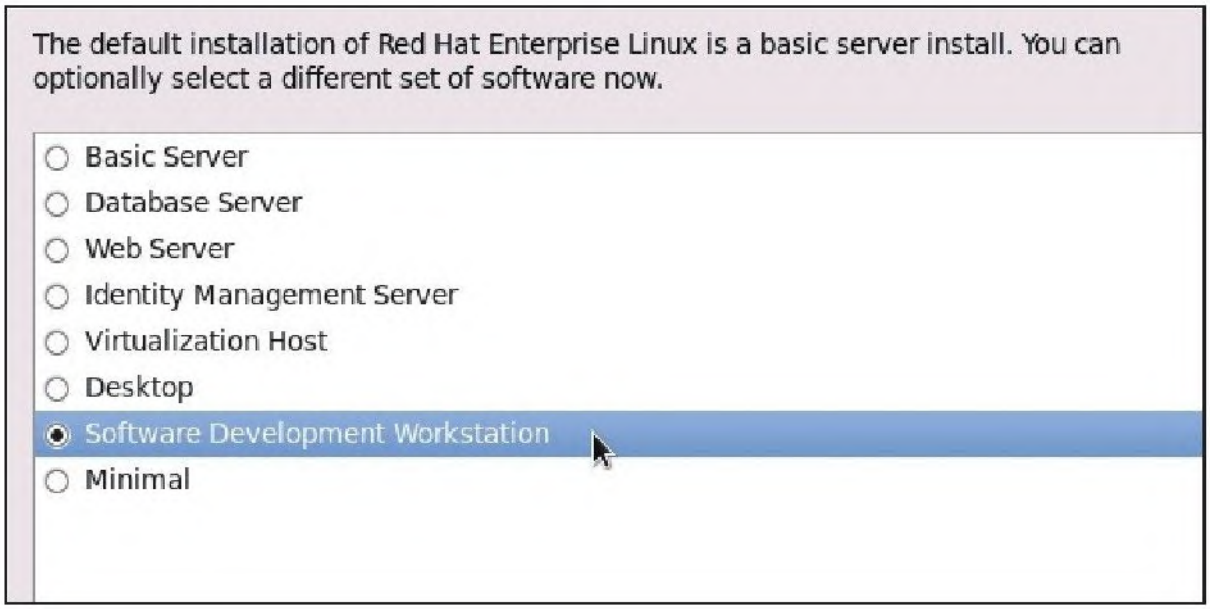


图　3-3　在RHEL 6.3安装中选择服务器类型

在选择了"Software Development Workstation"之后，在选择具体组件的界面就可以看到默认已经选择了很多的组件（如图3-4所示），这里主要需要检查一下Development这个选项中默认已经勾选了的很多的开发组件。其中，最好选中Development选项中的Development tools和Additional Development这两个组件，因为在本书的KVM编译过程中以及其他实验中可能会用到，其中包括一些比较重要的软件包，比如：gcc、git、make等（一般被默认选中）。另外可以单击下方的"Optional packages"按钮根据需要选择一些可选的软件包。在图3-4的Virtualization选项中，我们可以先不选中其中的任何

组件，因为在本章中会自己编译KVM和qemu-kvm，而在第7章介绍发行版中的KVM时，一般会安装Virtualization组件并使用发行版中自带的KVM Virtualization功能。

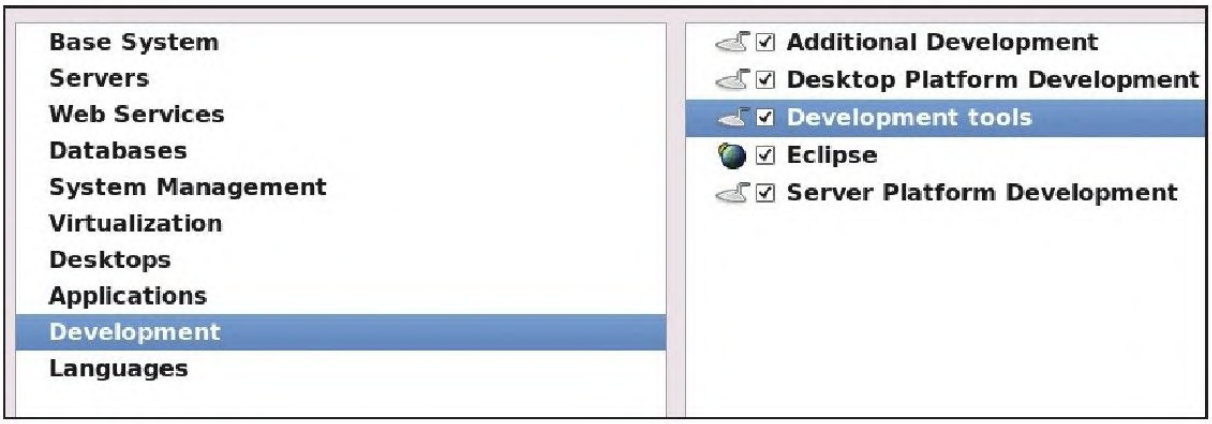


图　3-4　Software Development Workstation默认选择的具体组件

然后，继续进行后面安装流程，可以安装相应的软件包，安装过程的一个快照如图3-5所示。

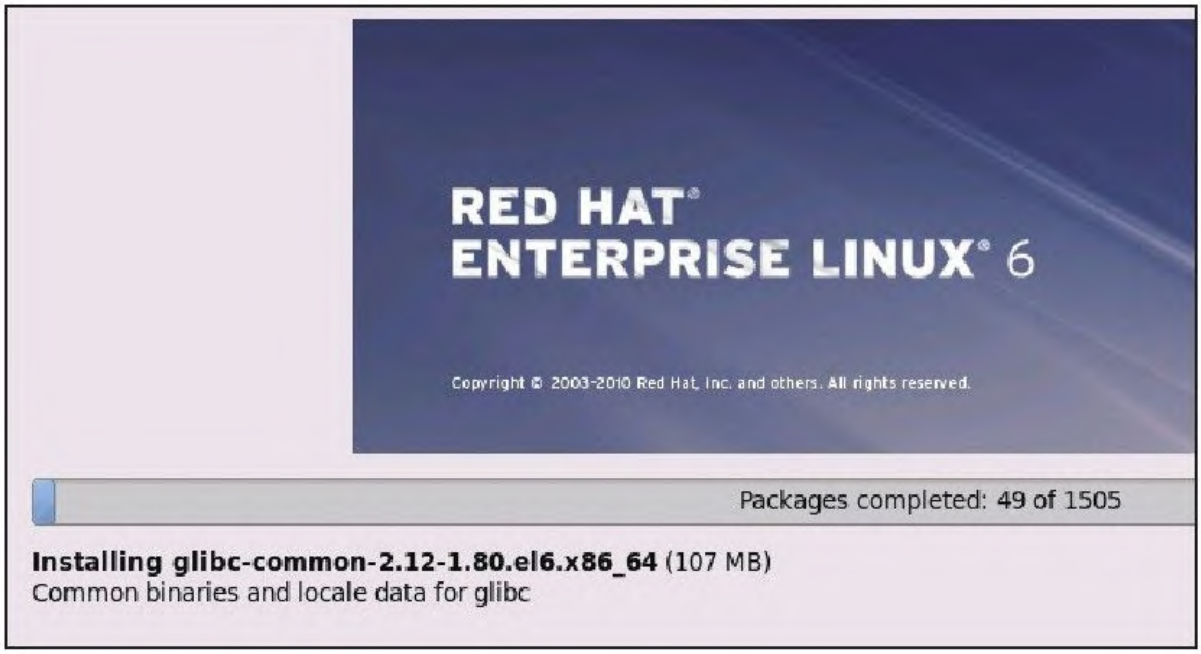


图　3-5　RHEL 6.3安装过程快照

在安装完所有软件包后，系统会提示安装完成需要重启系统，重启后即可进入到RHEL 6.3系统中。至此，Linux系统就安装完毕了，这就是在本书中作为宿主机（Host）的操作系统，后面的编译和实验都是在这个宿主机上进行的（当然，我们会使用本章讲述的自己编译的kernel和qemu-kvm来进行实验）。

3.3 编译和安装KVM

3.3.1 下载KVM源代码

KVM作为Linux kernel中的一个module而存在，是从Linux2.6.20版本开始被完全正式加入到内核的主干开发和正式发布代码中。所以，只需要下载2.6.20版本之后Linux kernel代码即可编译和使用KVM。当然，如果是学习KVM，推荐使用最新正式发布或者正在开发中的kernel版本，如果是实际部署到生产环境中，还需要自己选择适合的稳定版本进行详尽的功能和性能测试。如果你想使用最新的处于开发中的KVM代码，你需要自己下载KVM的代码仓库，本节就是以此为例来讲解的。

总的来说，下载最新KVM源代码，主要有如下三种方式：

- 下载KVM项目开发中的代码仓库kvm.git。
- 下载Linux内核的代码仓库linux.git。
- 打包下载Linux内核的源代码（Tarball[3] 格式）。

1.下载kvm.git

KVM项目的代码是托管在Linux内核官方源码网站http://git.kernel.org上的，可以到上面去查看和下载。该网页上virt/kvm/kvm.git即是KVM项目的代码，它是最新的功能最丰富的KVM源代码库（尽管并非最稳定的）。目前，kvm.git的最主要维护者（maintainer）是来自Redhat公司的Gleb Natapov和PaoloBonzini。从http://git.kernel.org/?p=virt/kvm/kvm.git网页可以看到，kvm.git下载链接有如下3个URL，可用于下载最新的KVM的开发代码仓库。

```
git://git.kernel.org/pub/scm/virt/kvm/kvm.git
http://git.kernel.org/pub/scm/virt/kvm/kvm.git
https://git.kernel.org/pub/scm/virt/kvm/kvm.git
```

从这3个URL下载的内容完全一致，根据自己实际情况选择其中任一个下载即可。Linux内核相关的项目一般都使用Git[4] 作为源代码管理工具，KVM当然也是用Git管理源码的。可以使用git clone命令来下载KVM的源代码，也可以使用Git工具的其他命令对源码进行各种管理，这里不详述Git的各种命令。

kvm.git的下载方式和过程，为如下命令行所示：

```
[root@jay-linux ~] cd kvm_demo
[root@jay-linux kvm_demo]# git clone\
git://git.kernel.org/pub/scm/virt/kvm/kvm.git kvm.git
```

```
Initialized empty Git repository in /root/kvm_demo/kvm.git/.git/
remote: Counting objects: 2556717, done.
remote: Compressing objects: 100% (399423/399423), done.
Receiving objects: 100% (2556717/2556717), 517.22 MiB | 11.21 MiB/s, done.
remote: Total 2556717 (delta 2141777), reused 2546109 (delta 2131175)
Resolving deltas: 100% (2141777/2141777), done.
[root@jay-linux kvm_demo]# cd kvm.git
[root@jay-linux kvm.git]# pwd
/root/kvm_demo/kvm.git
```

2. 下载linux.git

Linux内核的官方网站为http://kernel.org，其中源代码管理网为http://git.kernel.org，可以在此网站上找到最新的linux.git代码。在源码管理网站上，我们看到有多个linux.git，我们选择Linus Torvalds[5] 的源码库（也即是Linux内核的主干）。在内核源码的网页http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git中可以看到，其源码仓库也有如下3个链接可用：

```
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

这3个URL中源码内容是完全相同的，可以用用git clone命令复制到本地，其具体操作方式与前一种（kvm.git）的下载方式完全一样。

3. 下载Linux的Tarball

在Linux官方网站（http://kernel.org/）上，也提供Linux内核的Tarball文件下载；除了在其首页上单击一些Tarball之外，也可以

到如下网址下载Linux内核的各个版本的Tarball。

❑ftp://ftp.kernel.org/pub/linux/kernel/

❑http://www.kernel.org/pub/linux/kernel/

kernel.org还提供一种rsync的方式下载，此处不详细叙述，请参见其官网首页的提示。

以用wget下载linux-3.4.1.tar.gz为例，有如下的命令行代码：

```
[root@jay-linux kvm_demo]# wget\
http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.1.tar.gz
--2012-06-05 23:06:26--
http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.1.tar.gz
Resolving www.kernel.org... 149.20.20.133, 149.20.4.69
Connecting to www.kernel.org|149.20.20.133|:80...connected
Length: 101055487 (96M) [application/x-gzip]
Saving to: "linux-3.4.1.tar.gz"

100%[=====================================================================
===============>] 101,055,487 9.13M/s   in 11s

2012-06-05 23:06:37 (8.79 MB/s) - "linux-3.4.1.tar.gz" saved
[101055487/101055487]
[root@jay-linux kvm_demo]# ls -l
total 98692


drwxr-xr-x 24 root root      4096 Jun  5 23:05 kvm.git
-rw-r--r--  1 root root 101055487 Jun  5 02:36 linux-3.4.1.tar.gz
[root@jay-linux kvm_demo]# tar -zxf linux-3.4.1.tar.gz
[root@jay-linux kvm_demo]# ls -l
total 98696
drwxr-xr-x 24 root root      4096 Jun  5 23:05 kvm.git
drwxrwxr-x 23 root root      4096 Jun  1 15:18 linux-3.4.1
-rw-r--r--  1 root root 101055487 Jun  5 02:36 linux-3.4.1.tar.gz
```

4.通过kernel.org的镜像站点下载

由于Linux的源代码量比较大，如果只有美国一个站点可供下载，那么速度会较慢，服务器压力也较大。所以，kernel.org在世界上多个国家和地区都有一些镜像站点，而且一些Linux开源社区的爱好者们也自发建立了不少kernel.org的镜像，在中国的镜像站点中，推荐大家从如下两个镜像站点下载Linux相关的代码及其他源码（在本书写作之时，它们并没有提供git形式的代码仓库kvm.git或linux.git的下载），访问速度比较快。

❑一个是清华大学开源镜像站：http://mirror.tuna.tsinghua.edu.cn/，其中的链接地址http://mirror.tuna.tsinghua.edu.cn/kernel/linux/kernel/与http://www.kernel.org/pub/linux/kernel/就是同步的，用起来比较方便。

❑另外一个是北京交通大学的一个开源镜像站：

http://mirror.bjtu.edu.cn/kernel/linux/kernel/

还有如下两个镜像站可以推荐给大家：

❑网易开源镜像站，http://mirrors.163.com/

❑搜狐开源镜像站，http://mirrors.sohu.com/

（在本书写作之时，网易和搜狐的两个镜像站主要提供了一些Linux发行版的ISO及其软件仓库，而没有提供linux kernel的源码下载。）

3.3.2 配置KVM

上面三种方式下载的源代码都是可以同样地进行配置和编译，本章以开发中的最新源代码仓库kvm.git来讲解KVM的配置和编译等。KVM是作为Linux内核中的一个module存在的，而kvm.git是一个包含了最新的KVM模块开发中代码完整的Linux内核源码仓库。它的配置方式，与普通的Linux内核配置完全一样，只是需要注意将KVM相关的配置选择为编译进内核或者编译为模块。

在kvm.git（Linux kernel）代码目录下，运行"make help"命令可以得到一些关于如何配置和编译kernel的帮助手册，命令行如下：

```
[root@jay-linux kvm.git]# make help
Cleaning targets:
  clean           - Remove most generated files but keep the config and
                  enough build support to build external modules
  mrproper        - Remove all generated files + config + various backup files
```

```
  distclean       - mrproper + remove editor backup and patch files
                  Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based program
  menuconfig      - Update current config utilising a menu based program
  xconfig         - Update current config utilising a QT based front-end
<!- 此处省略数十行帮助信息 ->
make W=n   [targets] Enable extra gcc checks, n=1,2,3 where
                1: warnings which may be relevant and do not occur too often
                2: warnings which occur quite often but may still be relevant
                3: more obscure warnings, can most likely be ignored
                Multiple levels can be combined with W=12 or W=123
Execute "make" or "make all" to build all targets marked with [*]
For further info see the ./README file
```

对KVM或Linux内核配置时常用的配置命令的一些解释如下：

（1）make config

基于文本的最为传统的也是最为枯燥的一种配置方式，但是它可以适用于任何情况之下，这种方式会为每一个内核支持的特性向用户提问，如果用户回答"y"，则把特性编译进内核；回答"m"，则该特性作为模块进行编译；回答"n"，则表示不对该特性提供支持。输入“？”则显示该特性的帮助信息，在了解之后再决定处理该特性的方式；在回答每个问题前，必须考虑清楚，如果在配置过程中因为失误而给了错误的回答，就只能按"ctrl+c"强行退出然后重新配置了。

（2）make oldconfig

make oldconfig和make config类似，但是它的作用是在现有的内核设置文件基础上建立一个新的设置文件，只会向用户提供有关新内

核特性的问题，在新内核升级的过程中，make oldconfig非常有用，用户将现有的配置文件.config复制到新内核的源码中，执行make oldconfig，此时，用户只需要回答那些针对新增特性的问题。

（3）make silentoldconfig

和上面make oldconfig一样，但在屏幕上不再出现已在.config中配置好的选项。

（4）make menuconfig

基于终端的一种配置方式，提供了文本模式的图形用户界面，用户可以通过光标移动来浏览所支持的各种特性。使用这用配置方式时，系统中必须安装有ncurses库，否则会显示"Unable to find the ncurses libraries"的错误提示。其中"y"、"n"、"m"、“？”键的选择功能与前面make config中介绍的一致。

（5）make xconfig

基于XWindow的一种配置方式，提供了漂亮的配置窗口，不过只有能够在X Server上运行X桌面应用程序时才能够使用，它依赖于QT，如果系统中没有安装QT库，则会出现"Unable to find any QT installation"的错误提示。

（6）make gconfig

与make xconfig类似，不同的是make gconfig依赖于GTK库。

（7）make defconfig

按照内核代码中提供的默认配置文件对内核进行配置（在Intel x86_64平台上，默认配置为arch/x86/configs/x86_64_defconfig），生成.config文件可以用作初始化配置，然后再使用make menuconfig进行定制化配置。

（8）make allyesconfig

尽可能多地使用"y"设置内核选项值，生成的配置中包含了全部的内核特性。

（9）make allnoconfig

除必须的选项外，其他选项一律不选(常用于嵌入式Linux系统的编译)。

（10）make allmodconfig

尽可能多地使用"m"设置内核选项值来生成配置文件。

（11）make localmodconfig

会执行lsmod命令查看当前系统中加载了哪些模块(Modules)，并将原来的.config中不需要的模块去掉，仅保留前面lsmod命令查出来的这些模块，从而简化了内核的配置过程。这样做确实方便了很多，但是也有个缺点：该方法仅能使编译出的内核支持当前内核已经加载的模块。因为该方法使用的是lsmod查询得到的结果，如果有的模块当前没有加载，那么就不会编到新的内核中。

下面以make menuconfig为例介绍一下如何选择KVM相关的配置。运行make menuconfig后显示的界面如图3-6所示。

```
        General setup  --->
    [*] Enable loadable module support  --->
    -*- Enable the block layer  --->
        Processor type and features  --->
        Power management and ACPI options  --->
        Bus options (PCI etc.)  --->
        Executable file formats / Emulations  --->
    [*] Networking support  --->
        Device Drivers  --->
        Firmware Drivers  --->
        File systems  --->
        Kernel hacking  --->
        Security options  --->
    -*- Cryptographic API  --->
    [*] Virtualization  --->
        Library routines  --->
    ---
        Load an Alternate Configuration File
        Save an Alternate Configuration File

                        <Select>    < Exit >    < Help >
```

图　3-6　make menuconfig命令的选择界面

选择了Virtualization之后，进入其中进行详细配置，包括选中KVM、选中对处理器的支持（比如：KVM for Intel processors support，KVM for AMD processors support）等，如图3-7所示。

```
[*] built-in  [ ] excluded  <M> module  < > module capable

 --- Virtualization
 <M>   Kernel-based Virtual Machine (KVM) support
 <M>     KVM for Intel processors support
 < >     KVM for AMD processors support
 [*]     Audit KVM MMU
 <M>   Host kernel accelerator for virtio net (EXPERIMENTAL)
```

图　3-7　Virtualization中的配置选项

在配置完成之后，就会在kvm.git的目录下面生成一个.config文件，最好检查一下KVM相关的配置是否正确。在本次配置中，与KVM直接相关的几个配置项的主要情况如下：

```
CONFIG_HAVE_KVM=y
CONFIG_HAVE_KVM_IRQCHIP=y
CONFIG_HAVE_KVM_EVENTFD=y
CONFIG_KVM_APIC_ARCHITECTURE=y
CONFIG_KVM_MMIO=y
CONFIG_KVM_ASYNC_PF=y
CONFIG_HAVE_KVM_MSI=y
CONFIG_VIRTUALIZATION=y
CONFIG_KVM=m
CONFIG_KVM_INTEL=m
# CONFIG_KVM_AMD is not set
CONFIG_KVM_MMU_AUDIT=y
```

3.3.3 编译KVM

在对KVM源代码进行了配置之后，编译KVM就是一件比较容易的事情了。它的编译过程就完全是一个普通Linux内核编译的过程，需要经过编译kernel、编译bzImage和编译module等三个步骤。编译bzImage这一步不是必须的，在本章示例中，config中使用了initramfs，所以这里需要这个bzImage用于生成initramfs image。另外，在最新的Linux kernel代码中，根据Makefile中的定义可以看出，直接执行"make"或"make all"命令就可以将这里提及的3个步骤全部包括在内。本节是为了更好地展示编译的过程，才将编译的步骤分为这三步来来解释。

第一步，编译kernel的命令为"make vmlinux"，其编译命令和输出如下：

```
[root@jay-linux kvm.git]# makevmlinux -j 20
make[1]: Nothing to be done for 'all'.
HOSTCC  arch/x86/tools/relocs
  CHK     include/linux/version.h
HOSTCC  scripts/basic/fixdep


  CHK     include/generated/utsrelease.h
HOSTCC  scripts/kallsyms
HOSTCC  scripts/pnmtologo
HOSTCC  scripts/genksyms/genksyms.o
  CC      scripts/mod/empty.o
<!- 此处省略数千行编译时的输出信息 ->
LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux #这里就是编译、链接后生成了启动所需的Linux kernel文件
SORTEX  vmlinux
sort done marker at 9725f0
SYSMAP  System.map
```

其中，编译命令中"-j"参数并非必须的，它是让make工具用多进程来编译，比如上面命令中提到的"-j 20"，会让make工具最多创建20个GCC进程同时来执行编译任务。在一个比较空闲的系统上面，有一个

推荐值作为-j参数的值，即大约为2倍于系统上的CPU的core的数量（CPU超线程也算core）。

第二步，执行编译bzImage的命令"make bzImage"，其输出如下：

```
[root@jay-linux kvm.git]# make bzImage
make[1]:  Nothing to be done for 'all'.
make[1]:  Nothing to be done for 'relocs'.
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
make[3]: 'arch/x86/realmode/rm/realmode.bin' is up to date.
  CC      arch/x86/boot/a20.o
  AS      arch/x86/boot/bioscall.o
<!- 此处省略数十行编译时的输出信息 ->
  CC      arch/x86/boot/video-bios.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  OBJCOPY arch/x86/boot/vmlinux.bin
  HOSTCC  arch/x86/boot/tools/build
  BUILD   arch/x86/boot/bzImage # 这里生成了我们需要的 bzImage 文件
Kernel: arch/x86/boot/bzImage is ready  (#4)
```

第三步，编译kernel和bzImage之后编译内核的模块，命令为"make modules"，其命令行输出如下：

```
[root@jay-linux kvm.git]# make modules -j 20
make[1]: Nothing to be done for 'all'.
make[1]: Nothing to be done for 'relocs'.
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
  CALL    scripts/checksyscalls.sh
  CC [M]  arch/x86/kvm/vmx.o
<!- 此处省略数千行编译时的输出信息 ->
  IHEX    firmware/mts_gsm.fw
  IHEX    firmware/mts_edge.fw
  H16TOFW firmware/edgeport/boot.fw
  H16TOFW firmware/edgeport/boot2.fw
  H16TOFW firmware/edgeport/down.fw
  H16TOFW firmware/edgeport/down2.fw
  IHEX    firmware/edgeport/down3.bin
  IHEX2FW firmware/whiteheat_loader.fw
  IHEX2FW firmware/whiteheat.fw
  IHEX2FW firmware/keyspan_pda/keyspan_pda.fw
  IHEX2FW firmware/keyspan_pda/xircom_pgs.fw
```

3.3.4 安装KVM

编译完KVM之后，下面介绍如何安装KVM。

KVM的安装包括两个步骤：module的安装、kernel与initramfs的安装。

（1）安装module

通过"make modules_install"命令可以将编译好的module安装到相应的目录之中，在默认情况下module被安装

到/lib/modules/$kernel_version/kernel目录之中。

```
[root@jay-linux kvm.git]# make modules_install
  INSTALL arch/x86/crypto/aes-x86_64.ko
  INSTALL arch/x86/kernel/microcode.ko
  INSTALL arch/x86/kvm/kvm-intel.ko
  INSTALL arch/x86/kvm/kvm.ko
  INSTALL arch/x86/oprofile/oprofile.ko
<!- 此处省略千余行安装时的输出信息 ->
  INSTALL /lib/firmware/edgeport/down3.bin
  INSTALL /lib/firmware/whiteheat_loader.fw
  INSTALL /lib/firmware/whiteheat.fw
  INSTALL /lib/firmware/keyspan_pda/keyspan_pda.fw
  INSTALL /lib/firmware/keyspan_pda/xircom_pgs.fw
DEPMOD  3.5.0-rc2+
```

安装好module之后，可以查看一下相应的安装路径，可看到kvm模块也已经安装，如下所示：

```
[root@jay-linux kvm.git]# ls -l /lib/modules/3.5.0-rc2+/kernel/
total 28
drwxr-xr-x  3 root root 4096 Jun 13 08:58 arch
drwxr-xr-x  3 root root 4096 Jun 13 08:58 crypto
drwxr-xr-x 36 root root 4096 Jun 13 08:58 drivers
drwxr-xr-x 25 root root 4096 Jun 13 08:58 fs
drwxr-xr-x  5 root root 4096 Jun 13 08:58 lib
drwxr-xr-x 22 root root 4096 Jun 13 08:58 net
drwxr-xr-x  9 root root 4096 Jun 13 08:58 sound
[root@jay-linux kvm.git]# ls -l
/lib/modules/3.5.0-rc2+/kernel/arch/x86/kvm/total 528
-rw-r--r-- 1 root root 176861 Jun 13 08:58 kvm-intel.ko
-rw-r--r-- 1 root root 357867 Jun 13 08:58 kvm.ko
```

（2）安装kernel和initramfs

通过"make install"命令可以安装kernel和initramfs，命令行输出如下：

```
[root@jay-linux kvm.git]# make install
sh /root/kvm_demo/kvm.git/arch/x86/boot/install.sh 3.5.0-rc2+
arch/x86/boot/bzImage \
             System.map "/boot"
[root@jay-linux kvm.git]# ls -l /boot/
-rw-r--r--  1 root root  2775664 Jun 13 08:45 vmlinuz-3.5.0-rc2+
-rw-r--r--  1 root root  1825462 Jun 13 08:45 System.map-3.5.0-rc2+
lrwxrwxrwx  1 root root       24 Jun 13 08:45 vmlinuz ->
/boot/vmlinuz-3.5.0-rc2+
lrwxrwxrwx  1 root root       27 Jun 13 08:45 System.map ->
/boot/System.map-3.5.0-rc2+
dr-xr-xr-x  4 root root     4096 Jun 13 08:45 ./
-rw-r--r--  1 root root  4672632 Jun 13 08:45 initramfs-3.5.0-rc2+.img
drwxr-xr-x  2 root root     4096 Jun 13 08:45 grub/
```

可见，在/boot目录下生成了内核（vmlinuz）和initramfs等内核启动所需的文件。

在运行make install之后，grub配置文件（如：/boot/grub/grub.conf）中也自动添加了一个grub选项，如下所示：

```
title Red Hat Enterprise Linux Server (3.5.0-rc2+)
        root (hd0,0)
      kernel /boot/vmlinuz-3.5.0-r2+ ro
root=UUID=1a65b4bb-cd9b-4bbf-97ff-7e1f7698d3db r
initrd /boot/initramfs-3.5.0-r2+.img
```

检查了grub之后，重新启动系统，选择刚才为了KVM而编译、安装的内核启动。

系统启动后，登录进入系统，在通常情况下，系统启动时默认已经加载了kvm和kvm_intel这两个模块；如果没有加载，请手动用modprobe命令依次加载kvm和kvm_intel模块。

```
[root@jay-linux ~]# modprobe kvm
[root@jay-linux ~]# modprobe kvm_intel
[root@jay-linux ~]# lsmod | grep kvm
kvm_intel              112487  0
kvm                    206544  1 kvm_intel
```

确认KVM相关的模块加载成功后，检查/dev/kvm这个文件，它是kvm内核模块提供给用户空间的qemu-kvm程序使用的一个控制接口，它提供了客户机（Guest）操作系统运行所需要的模拟和实际的硬件设备环境。

```
[root@jay-linux ~]# ls -l /dev/kvm
crw-rw---- 1 root root 10, 232 Jun  6 15:45 /dev/kvm
```

3.4 编译和安装qemu-kvm

除了在内核空间的kvm模块之外，在用户空间需要QEMU[6] 来模拟所需要CPU和设备模型以及用于启动客户机进程，这样才有了一个完整的KVM运行环境。而qemu-kvm是为了针对KVM专门做了修改和优化的QEMU分支[7] ，在本书写作的2012年，qemu-kvm分支里面的小部分特性还没有完全合并进入到qemu的主干代码之中，所以本书中采用qemu-kvm来讲解。

在编译和安装了KVM并且启动到编译的内核之后，下面来看一下qemu-kvm的编译和安装。

3.4.1 下载qemu-kvm源代码

目前的QEMU项目针对KVM的代码分支qemu-kvm也是由Redhat公司的Gleb Natapov和Paolo Bonzini作维护者（Maintainer），代码也是托管在kernel.org上。

qemu-kvm开发代码仓库的网页连接为：http://git.kernel.org/?p=virt/kvm/qemu-kvm.git.

其中，可以看到有如下3个URL连接可供下载开发中的最新qemu-kvm的代码仓库。

```
git://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git
http://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git
https://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git
```

可以根据自己实际需要选择3个中任一个用git clone命令下载即可，它们是完全一样的。

另外，可以到sourceforge.net的如下链接中根据需要下载qemu-kvm各个发布版本的代码压缩包（笔者建议使用最新的正式发布版本，因为它的功能更多，同时也比较稳定）：

http://sourceforge.net/projects/kvm/files/qemu-kvm/

http://qemu-project.org/Download（从2013年开始直接到这里下载普通的QEMU）

在本节后面讲解编译时，是以下载开发中的最新的qemu-kvm.git为例的，获取其代码仓库过程如下：

```
[root@jay-linux kvm_demo]# git clone\
git://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git qemu-kvm.git
Initialized empty Git repository in /root/kvm_demo/qemu-kvm.git/.git/
remote: Counting objects: 145222, done.
remote: Compressing objects: 100% (35825/35825), done.
remote: Total 145222 (delta 114656), reused 137663 (delta 107444)
Receiving objects: 100% (145222/145222), 40.83 MiB | 10.33 MiB/s, done.
```

```
Resolving deltas: 100% (114656/114656), done.
[root@jay-linux kvm_demo]# cdqemu-kvm.git
[root@jay-linux kvm.git]# pwd
/root/kvm_demo/qemu-kvm.git
```

3.4.2 配置和编译qemu-kvm

qemu-kvm的配置并不复杂，在通常情况下，可以直接运行代码仓库中configure文件进行配置即可。当然，如果对其配置并不熟悉，可以运行"./configure--help"命令查看配置的一些选项及其帮助信息。

显示配置的帮助手册信息如下：

```
[root@jay-linux qemu-kvm.git]# ./configure --help
Usage: configure [options]
Options: [defaults in brackets after descriptions]

Standard options:
  --help                   print this message
  --prefix=PREFIX           install in PREFIX [/usr/local]
  --interp-prefix=PREFIX    where to find shared libraries, etc.
                           use %M for cpu name [/usr/gnemul/qemu-%M]
  --target-list=LIST        set target list (default: build everything)
                            Available targets: i386-softmmu x86_64-softmmu
<!- 此处省略百余行帮助信息的输出 ->
  --disable-guest-agent     disable building of the QEMU Guest Agent
  --enable-guest-agent      enable building of the QEMU Guest Agent
  --with-coroutine=BACKEND coroutine backend. Supported options:
                            gthread, ucontext, sigaltstack, windows

NOTE: The object files are built at the place where configure is launched
```

执行 configure 文件进行配置的过程如下：

```
[root@jay-linux qemu-kvm.git]# ./configure
Install prefix    /usr/local
BIOS directory    /usr/local/share/qemu
binary directory  /usr/local/bin
library directory /usr/local/lib
include directory /usr/local/include
config directory  /usr/local/etc
Manual directory  /usr/local/share/man
ELF interp prefix /usr/gnemul/qemu-%M
Source path       /root/kvm_demo/qemu-kvm.git
C compiler        gcc
Host C compiler   gcc
<!-- 此处省略数十行 -->
VNC support       yes      # 通常需要通过 VNC 连接到客户机中
<!-- 此处省略十余行 -->
KVM support       yes      # 这是对 KVM 的支持
TCG interpreter   no
KVM device assig. yes      # 这是对 KVM 中 VT-d 功能的支持
<!-- 此处省略十余行 -->
OpenGL support    yes
libiscsi support  no
build guest agent yes
coroutine backend ucontext
```

需要注意的是，上面命令行输出的KVM相关的选项需要是配置为yes，另外，一般VNC的支持也是配置为yes的（因为通常需要用VNC连接到客户机中）。

经过配置之后，进行编译就很简单了，直接执行make即可进行编译，如下所示：

```
[root@jay-linux qemu-kvm.git]# make -j 20
  GEN   config-host.h
  GEN   trace.h
  GEN   qemu-options.def
  GEN   qmp-commands.h
  GEN   qapi-types.h
  GEN   qapi-visit.h
  GEN   tests/test-qapi-types.h
  GEN   tests/test-qapi-visit.h
  GEN   tests/test-qmp-commands.h
  CC    libcacard/cac.o
  CC    libcacard/event.o
<!-- 此处省略数百行的编译时输出信息 -->
  CC    x86_64-softmmu/target-i386/cpu.o
  CC    x86_64-softmmu/target-i386/machine.o
  CC    x86_64-softmmu/target-i386/arch_memory_mapping.o
  CC    x86_64-softmmu/target-i386/arch_dump.o
  CC    x86_64-softmmu/target-i386/kvm.o
  CC    x86_64-softmmu/target-i386/hyperv.o
LINK  x86_64-softmmu/qemu-system-x86_64
```

可以看到，最后有编译生成qemu-system-x86_64文件，它就是我们常用的qemu-kvm的命令行工具（在多数Linux发行版中自带的qemu-kvm软件包的命令行是qemu-kvm，只是名字不同而已）。

3.4.3 安装qemu-kvm

编译完成之后，运行"make install"命令即可安装qemu-kvm，其过程如下：

```
[root@jay-linux qemu-kvm.git]# make install | tee make-install.log
install -d -m 0755 "/usr/local/share/qemu"
install -d -m 0755 "/usr/local/etc/qemu"
install -c -m 0644
/root/kvm_demo/qemu-kvm.git/sysconfigs/target/target-x86_64.conf

"/usr/local/etc/qemu"
install -c -m 0644
/root/kvm_demo/qemu-kvm.git/sysconfigs/target/cpus-x86_64.conf
"/usr/local/share/qemu"
install -d -m 0755 "/usr/local/bin"
install -c -m 0755  vscclient qemu-ga qemu-nbd qemu-img qemu-io  "/usr/local/bin"
install -d -m 0755 "/usr/local/libexec"
<!-- 此处省略数行的安装时输出信息 -->
make[1]: Entering directory
'/root/kvm_demo/qemu-kvm.git/x86_64-softmmu'
install -m 755 qemu-system-x86_64 "/usr/local/bin"
strip "/usr/local/bin/qemu-system-x86_64"
make[1]: Leaving directory
'/root/kvm_demo/qemu-kvm.git/x86_64-softmmu'
```

qemu-kvm安装过程的主要任务有这几个：创建qemu的一些目录，复制一些配置文件到相应的目录下，复制一些firmware文件（如：sgabios.bin,kvmvapic.bin）到目录下以便qemu-kvm的命令行启动时可以找到对应的固件提供给客户机使用，复制keymaps到相应的目录下以便在客户机中支持各种所需键盘类型，复制qemu-system-x86_64、qemu-img等可执行程序到对应的目录下。下面的一些命令行检查了qemu-kvm被安装了之后的系统状态。

```
[root@jay-linux qemu-kvm.git]# which qemu-system-x86_64
/usr/local/bin/qemu-system-x86_64
[root@jay-linux qemu-kvm.git]# which qemu-img
/usr/local/bin/qemu-img
[root@jay-linux qemu-kvm.git]# ls /usr/local/share/qemu/
bamboo.dtb        mpc8544ds.dtb     petalogix-ml605.dtb
pxe-pcnet.rom    slof.bin           vgabios-vmware.bin
bios.bin          multiboot.bin     petalogix-s3adsp1800.dtb
pxe-rtl8139.rom  spapr-rtas.bin
cpus-x86_64.conf  openbios-ppc      ppc_rom.bin
pxe-virtio.rom   vgabios.bin
keymaps           openbios-sparc32  pxe-e1000.rom
qemu-icon.bmp    vgabios-cirrus.bin
kvmvapic.bin      openbios-sparc64  pxe-eepro100.rom
s390-zipl.rom    vgabios-qxl.bin
linuxboot.bin     palcode-clipper   pxe-ne2k_pci.rom
sgabios.bin      vgabios-stdvga.bin
[root@jay-linux qemu-kvm.git]# ls /usr/local/share/qemu/keymaps/
ar    common  de     en-gb  es  fi  fr     fr-ca  hr  is  ja  lv
modifiers  nl-be  pl  pt-br  sl  th
bepo  da      de-ch  en-us  et  fo  fr-be  fr-ch  hu  it  lt  mk  nl
no    pt  ru     sv  tr
```

由于qemu-kvm是用户空间的程序，安装之后不用重启系统，直接用qemu-system-x86_64、qemu-img这样的命令行工具即可使用qemu-kvm了。

3.5 安装客户机

安装客户机（Guest）之前，我们需要创建一个镜像文件或者磁盘分区等来存储客户机中的系统和文件，关于客户机镜像有很多种的制作和存储方式（将在后面的第4章中进行详细的介绍），本节只是为了快速地演示安装一个客户机，采用了本地创建一个镜像文件，然后让镜像文件作为客户机的硬盘，将客户机操作系统（以RHEL6.3为例）安装在其中。

首先，需要创建一个镜像文件，可以使用dd工具，如下的命令行创建了一个8GB大小的镜像文件rhel6u3.img：

```
[root@jay-linux kvm_demo]# dd if=/dev/zero of=rhel6u3.img bs=1M\ count=8192
8192+0 records in
8192+0 records out
8589934592 bytes (8.6 GB) copied, 76.9331 s, 112 MB/s
```

然后，准备一个RHEL6.3安装所需的ISO文件，如下所示：

```
[root@jay-linux kvm_demo]# ls RHEL6.3-Server-x86_64-DVD1.iso
RHEL6.3-Server-x86_64-DVD1.iso
```

启动客户机，并在其中用准备好的ISO安装系统，命令行如下：

```
[root@jay-linux kvm_demo]#qemu-system-x86_64 -m 2048 -smp 4-boot
order=cd -hda /root/kvm_demo/rhel6u3.img -cdrom /root/kvm_demo/
RHEL6.3-Server-x86_64-DVD1.iso
VNC server running on '::1:5900'
```

其中，-m 2048是给客户机分配2048MB内存，-smp 4是给客户机分配4个CPU，-boot order=cd是指定系统的启动顺序为光驱（c:CD-ROM）、硬盘(d:hard Disk)，-hda**是分配给客户机的IDE硬盘（即前面准备的镜像文件），-cdrom**是分配客户机的光驱。在默认情况下，QEMU会启动一个VNC server端口（如上面的::1:5900），可以用vncviewer[8] 工具来连接到QEMU的VNC端口查看客户机。

通过启动时的提示，这里可以使用"vncviewer:5900"命令连接到QEMU启动的窗口。根据命令行制定的启动顺序，当有CDROM时，客户机默认会从光驱引导，启动后即可进入到客户机系统安装界面如图3-8所示。

图　3-8　客户机安装的选择界面

可以选择Install安装客户机操作系统，和安装普通Linux系统类似，根据需要做磁盘分区、选择需要的软件包等，安装过程中的一个快照如图3-9所示。

图　3-9　客户机安装过程的快照

在系统安装完成后，客户机中安装程序提示，如图3-10所示。

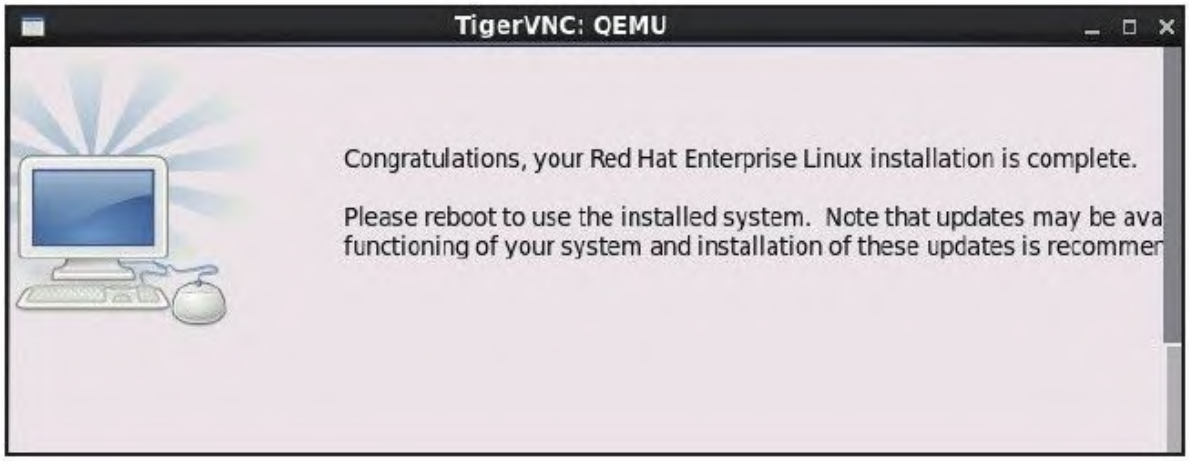


图　3-10　客户机安装完成后的提示信息

和安装普通的Linux系统一样，完成安装后，重启系统即可进入到刚才安装的客户机操作系统。

3.6 启动第一个KVM客户机

在安装好了系统之后，就可以使用镜像文件来启动并登录到自己安装的系统之中了。通过如下的命令行即可启动一个KVM的客户机。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -m 2048 -smp 4 -hda
/root/kvm_demo/rhel6u3.img
VNC server running on '::1:5900'
```

用vncviewer命令（此处命令为vncviewer:5900）查看客户机的启动情况，客户机启动过程中的状态，如图3-11所示。

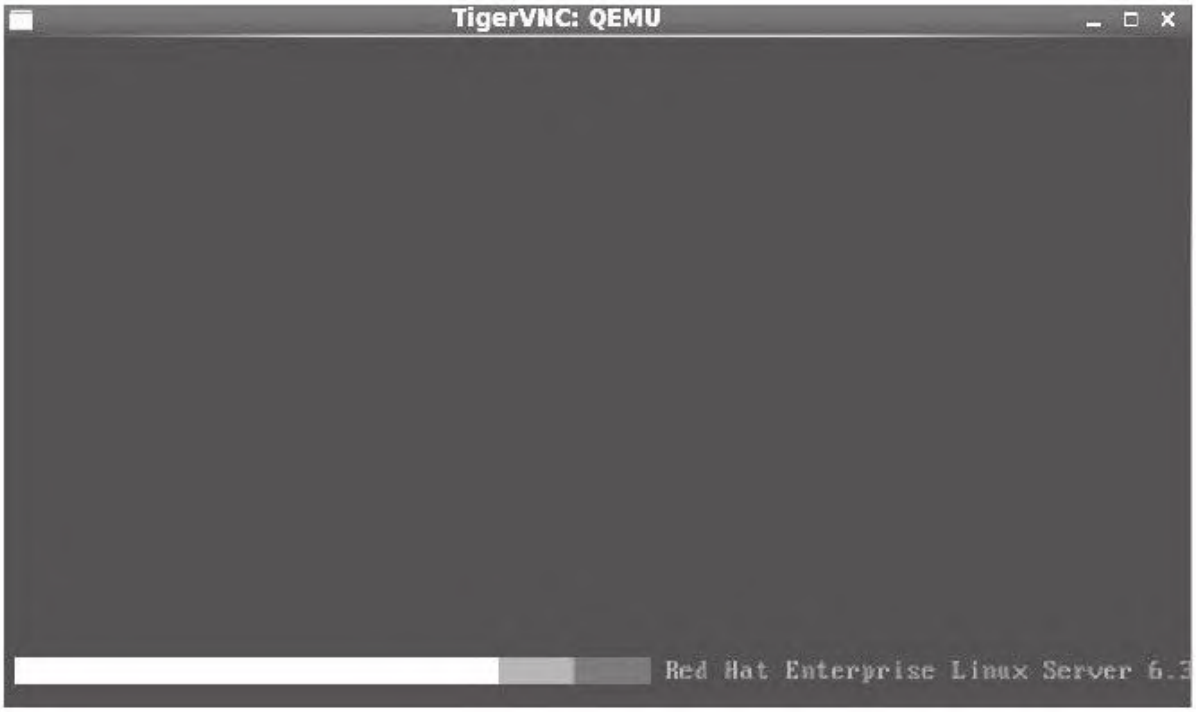


图 3-11 客户机启动中状态

客户机启动完成后的登录界面如图3-12所示。

图　3-12　客户机启动后的登录界面

在通过VNC链接到QEMU窗口后，可以按快捷键"Ctrl+Alt+2"切换到QEMU监视器窗口，在监视器窗口中可以执行一些命令，比如执行"info kvm"命令来查看当前QEMU是否使用着KVM，如图3-13所示（显示为kvm support:enabled）。

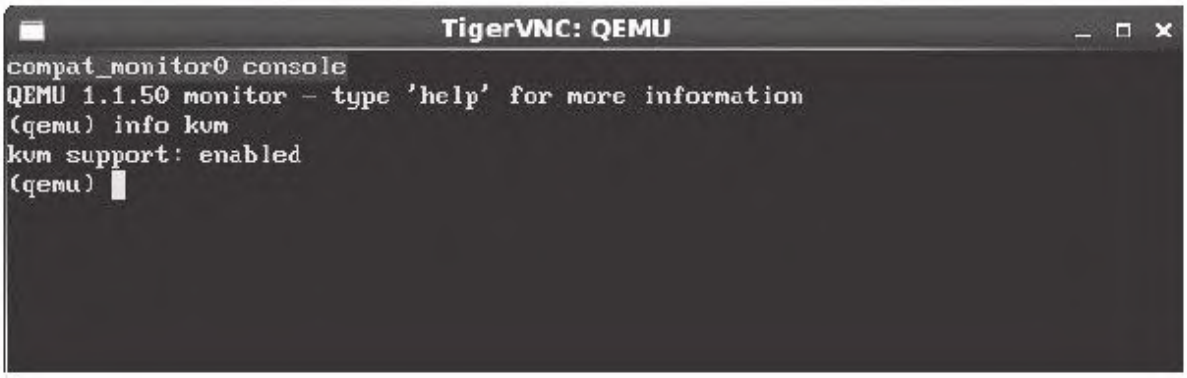


图　3-13　QEMU Monitor中"info kvm"命令

用快捷键"Ctrl+Alt+1"切换回普通的客户机查看窗口，就可以登录或继续使用客户机系统了。至此，你就已经启动了属于自己的第一

个KVM客户机，尽情享受KVM虚拟化带来的快乐吧！

3.7 本章小结

本章主要讲解了如何一步一步地构建属于自己的KVM虚拟化环境。硬件和BIOS中虚拟化技术的支持是KVM运行的先决条件，编译kvm和qemu-kvm是掌握KVM必须的最基础也是不可或缺的技能。其中的配置和编译kvm（或Linux kernel）是最复杂也是非常重要的环节，理解这环节对Linux kernel相关技术（包括KVM）的学习具有非常重要的价值。

通过本章，相信你已经对KVM的虚拟化环境的构建比较了解了，请自己动手构建一套自己的KVM虚拟化系统吧，也许在构建环境过程中你会碰到这样那样的问题，不过只有当你把它们都解决之后才能真正的理解。接下来，继续看后面的章节来学习KVM虚拟化都提供了那一些的功能、命令行以及它们的基本原理。

3.8 本章注释和参考阅读

1.本章注释

[1]KVM在ARM处理器上第一次正式的实现，可以参考http://comments.gmane.org/gmane.comp.emulators.kvm.devel/87136中主题为"[PATCH v6 00/12]KVM/ARM Implementation"的邮件。

[2]在本书中，若无特别注明，所有理论和实验都是基于Intel x86-64的处理器。

[3]Tarball是UNIX/Linux世界中的一个术语，用于指代tar档案（通过tar命令打包归档后的文件，如常见的以.tar、.tar.gz、.tar.bz2结尾的档案文件）。

[4]Git是一个高效、快速的分布式的版本控制工具和源代码管理工具，最初是Linus Torvalds为Linux内核开发而设计和开发的工具，是在Linux中最广泛使用的SCM工具。另外，一个基于Git的非常著名的源码托管网站https://github.com也值得推荐一下。

[5]Linus Torvalds，从芬兰移民的美国人，著名的软件工程师和黑客。1991年，在芬兰赫尔辛基大学读书的Linus发布了后来最流行的开源操作系统内核Linux的第一个版本。

[6]关于QEMU项目，可以参考其官方网站：http://wiki.qemu.org/Main_Page.

[7]关于QEMU与qemu-kvm的关系：http://wiki.qemu.org/KVM。在2012年底，QEMU的1.3.0版本发布时，qemu-kvm中针对KVM的修改已经完全加入到普通的QEMU代码库中了，从此之后可以完全使用纯QEMU来与KVM配合使用（命令行添加-enable-kvm参数），而不是需要专门使用qemu-kvm代码库了。下载普通QEMU的代码仓库为：git://git.qemu.org/qemu.git和http://git.qemu.org/git/qemu.git。

[8]在宿主机中需要安装包含vncserver和vncviewer工具的软件包，如在RHEL6.3系统中，可以安装tigervnc-server和tigervnc这两个RPM软件包。

2.参考阅读

（1）KVM官方网站的FAQ（常见问题与答案）列表：http://www.linux-kvm.org/page/FAQ.

（2）KVM宿主机或客户机安装RHEL6.x系统，也可参考如下链接中详细的英文文档：

http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/index.html.

第4章　KVM核心基础功能

KVM采用的完全虚拟化（Full Virtualizaiton）技术，在KVM环境中运行的客户机（Guest）操作系统是未经过修改的普通操作系统。在硬件虚拟化技术的支持下，内核的KVM模块与QEMU的设备模拟协同工作，就构成了一整套与物理计算机系统完全一致的虚拟化的计算机软硬件系统。

要运行一个完整的计算机系统，必不可少的最重要的子系统包括：处理器（CPU）、内存（Memory）、存储（Storage）、网络（Network）、显示（Display）等。本章将会介绍KVM环境中这些基本子系统的基本概念、原理、配置和实践。

4.1　硬件平台和软件版本说明

在本章及第5章的介绍和实验中，除非特别注明，否则默认硬件平台（CPU）是Intel(R)Xeon(R)CPU E5-4600（Romley[1] -EP 4S），软件系统中宿主机和客户机都是RHEL6.3系统，而宿主机内核是KVM内核3.5版本，用户态的QEMU是qemu-kvm 1.1.0版本，并且各个实验中在使用qemu-kvm时都开启了KVM加速的功能。在正式开始本章内容之前，先分别对软硬件环境进行较为详细的说明，其中涉及的编译和配置过程请参考第3章。

（1）硬件平台

一般来说使用支持硬件辅助虚拟化（如Intel的VT-x）的硬件平台即可，在第5章中介绍的一些特性（如AVX、SMEP、VT-d等）需要某些特定的CPU或芯片组的支持，在介绍具体特性时会进行说明。在第4、5章中，笔者默认使用CPU平台是Intel(R)Xeon(R)CPU E5-4600系列，在BIOS中打开VT-x和VT-d的支持，可参考3.1节中对硬件系统的配置说明。

（2）KVM内核

选取一个较新的又较稳定的正式发布版本，这里选择的是2012年7月发布的Linux 3.5版本。可以通过如下链接下载Linux 3.5版本：

http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.5.tar.gz.

如果是在linux.git的源代码仓库中，可以查询到v3.5这个发布标签，然后切换到当时正式发布时的状态，命令行如下：

```
[root@jay-linux linux.git]# git tag
<!-- 此处省略数百行输出 -->
v3.4-rc6
v3.4-rc7
```

```
v3.5
v3.5-rc1
v3.5-rc2
v3.5-rc3
v3.5-rc4
v3.5-rc5
v3.5-rc6
v3.5-rc7
[root@jay-linux linux.git]# git checkout v3.5
Note: checking out 'v3.5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

 git checkout -b new_branch_name

HEAD is now at 28a33cb... Linux 3.5
```

如果使用的是kvm.git，由于它没有Linux的v3.5版本标签，不能直接执行"git checkout v3.5"，因此需要在"git log"中查询到Linux发布3.5版本的信息，然后切换到对应的commit，命令如下：

```
[root@jay-linux kvm.git]# git log
<!-- 此处省略数百行输出信息 -->
commit 73bcc49959e4e40911dd0dd634bf1b353827df66
Merge: 8447c4d 28a33cb
Author: Mauro Carvalho Chehab <mchehab@redhat.com>
Date:   Sun Jul 29 21:09:39 2012 -0300

    Merge tag 'v3.5'

    Linux 3.5

    * tag 'v3.5': (1242 commits)
<!-- 此处省略数其余的输出信息 -->
[root@jay-linux kvm.git]# git checkout 73bcc499
Note: checking out '73bcc499'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

 HEAD is now at 73bcc49... Merge tag 'v3.5'
```

切换到合适的源码版本之后进行配置、编译、安装等操作，可参考3.3节中的内容。

（3）qemu-kvm

qemu-kvm的版本使用的是2012年7月初发布的qemu-kvm-1.1.0版本，下载链接为：

http://sourceforge.net/projects/kvm/files/qemu-kvm/1.1.0/qemu-kvm-1.1.0.tar.gz.

在qemu-kvm.git的GIT代码仓库中，可以先通过"git tag"命令查看有哪些标签，然后找到"qemu-kvm-1.1.0"标签，用"git checkout qemu-kvm-1.1.0"（或"git reset--hard qemu-kvm-1.1.0"）命令切换到1.1.0的qemu-kvm版本，此过程的命令行演示如下：

```
[root@jay-linux qemu-kvm.git]# git tag
<!-- 此处省略数十行输出 -->
qemu-kvm-1.1-rc3
qemu-kvm-1.1-rc4
qemu-kvm-1.1.0
<!-- 此处省略十余行输出 -->
[root@jay-linux qemu-kvm.git]# git checkout qemu-kvm-1.1.0
HEAD is now at bd11ac4... fdc: fix implied seek while there is no media in drive
```

对qemu-kvm进行编译和安装，参考3.4节中的内容。这里在编译qemu-kvm时，除了4.5.1小节对SDL进行介绍时需要SDL之外，默认没有将SDL的支持编译进去（在运行"configure"时加上"--disable-sdl"参数），而采用VNC的方式显示客户机的图形界面。

在使用qemu-kvm命令行启动客户机时，不一定需要超级用户（root）来操作，但是需要让当前用户对/dev/kvm这个接口具有可读可写的权限。另外，在涉及网络配置、设备分配等特权操作时，还是需要root的权限，所以为了简单起见，本书都采用root用户来进行操作。

（4）QEMU命令行开启KVM加速功能

如果使用的是qemu-kvm命令行，默认开启了对KVM的支持的，可以通过在QEMU monitor中的"info kvm"命令来看是否显示"kvm support:enabled"（详见3.6节中的介绍）。如果使用的并非qemu-kvm而是普通的QEMU，有可能KVM没有被默认打开，这时候需要在QEMU启动的命令行加上"-enable-kvm"这个参数。

另外，如果已经安装了支持KVM的Linux发行版，则不一定需要自己重新编译内核（包括KVM模块）和用户态程序qemu-kvm。如果已经安装了RHEL6.3系统且选择了其中的虚拟化组件，则只需检查当前内核是否支持KVM（查看/boot/config-xx文件中的KVM相关配置），以及kvm和kvm_intel模块是否正确加载（命令为lsmod|grep kvm），然后找到qemu-kvm的命令行工具（/usr/libexec/qemu-kvm），就用这个qemu-kvm命令行工具来进行后面的具体实践以便了解KVM即可。只需将本书中使用"qemu-system-x86_64"命令的地方替换为系统中实际的qemu-kvm的路径即可。关于QEMU的命令行参数基本都是一致的，不需要做特别的改变，如果遇到参数错误的提示，可查阅当前版本的qemu-kvm的帮助手册。

4.2 CPU配置

CPU作为计算机系统的“大脑”，是最重要的部分，负责计算机程序指令的执行。在QEMU/KVM中，QEMU提供对CPU的模拟，展现给客户机一定的CPU数目和CPU的特性；在KVM打开的情况下，客户机中CPU指令的执行由硬件处理器的虚拟化功能（如Intel VT-x和AMD AMD-V）来辅助执行，具有非常高的执行效率。

4.2.1 vCPU的概念

QEMU/KVM为客户机提供一套完整的硬件系统环境，在客户机看来其所拥有的CPU即是vCPU（virtual CPU）。在KVM环境中，每个客户机都是一个标准的Linux进程（QEMU进程），而每一个vCPU在宿主机中是QEMU进程派生的一个普通线程。

在普通的Linux系统中，进程一般有两种执行模式：内核模式和用户模式。而在KVM环境中，增加了第三种模式：客户模式。vCPU在三种执行模式下的不同分工如下：

（1）用户模式（User Mode）

主要处理I/O的模拟和管理，由QEMU的代码实现。

（2）内核模式（Kernel Mode）

主要处理特别需要高性能和安全相关的指令，如处理客户模式到内核模式的转换，处理客户模式下的I/O指令或其他特权指令引起的退出（VM-Exit），处理影子内存管理（shadow MMU）。

（3）客户模式（Guest Mode）

主要执行Guest中的大部分指令，I/O和一些特权指令除外（它们会引起VM-Exit，被hypervisor截获并模拟）。

vCPU在KVM中的这三种执行模式下的转换如图4-1所示。

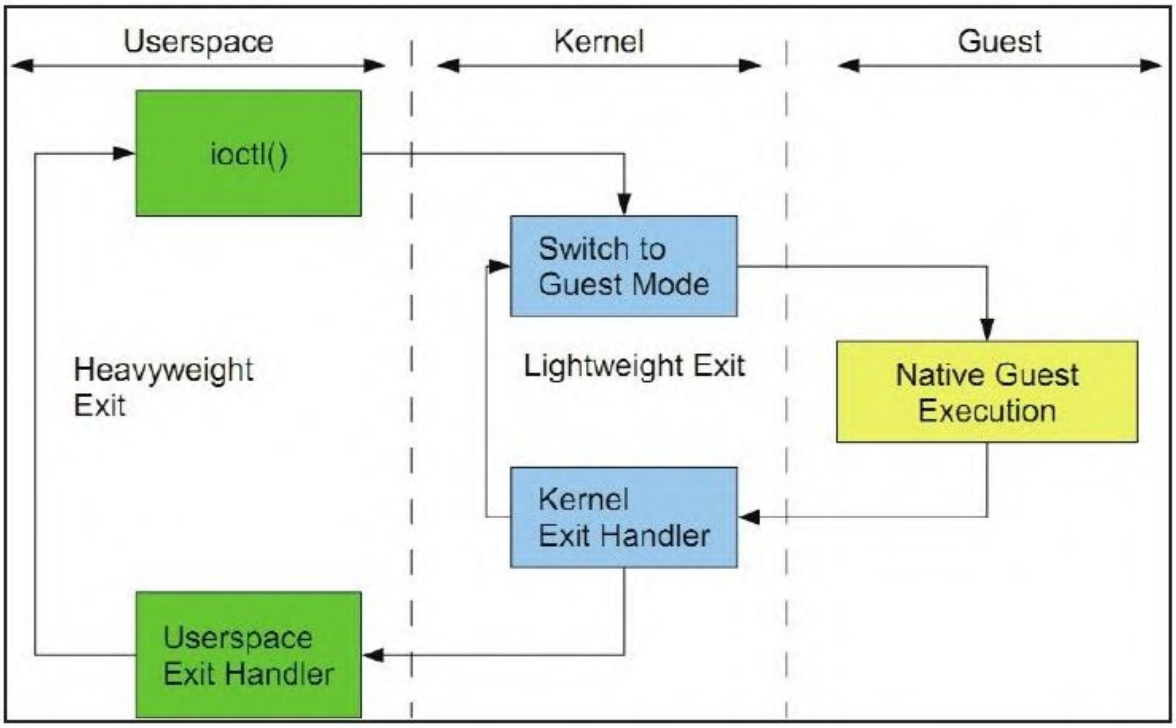


图　4-1　vCPU在KVM中的三种执行模式

在KVM环境中，整个系统的基本架构如图4-2所示。

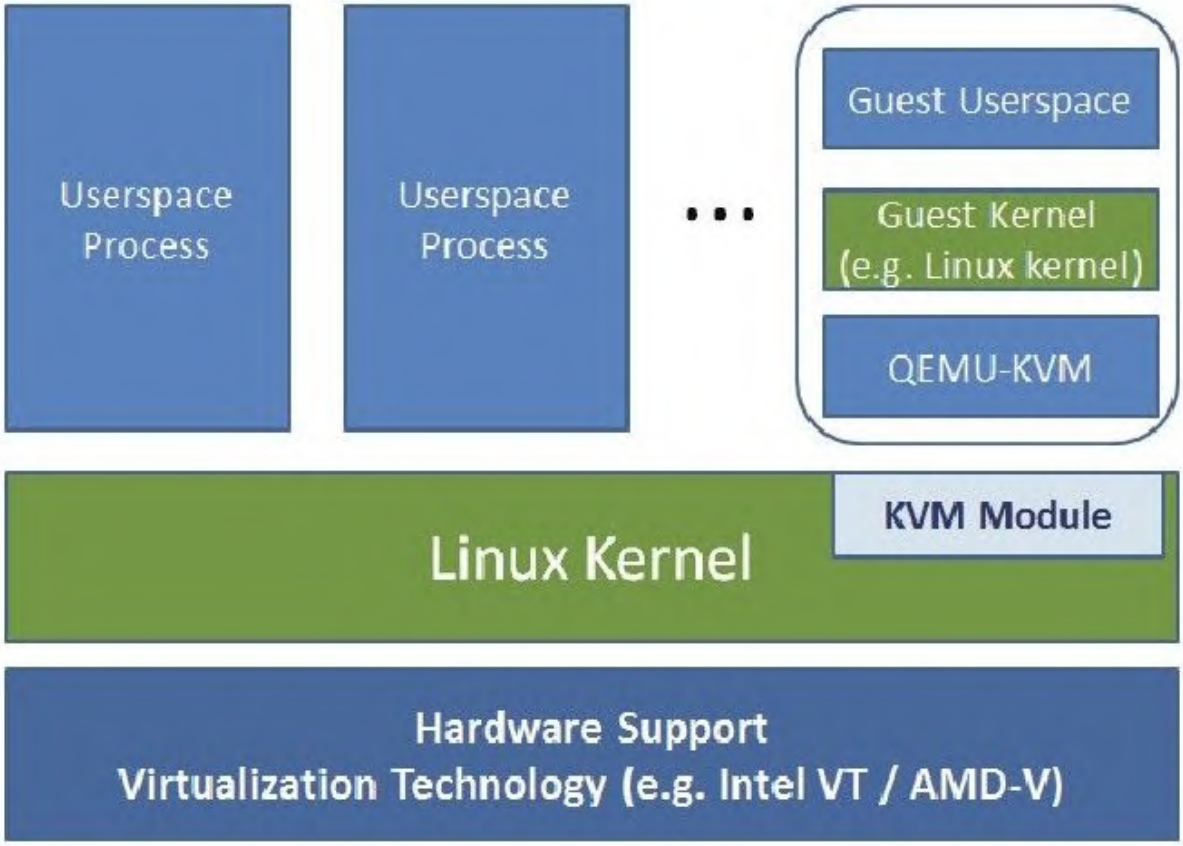


图 4-2 KVM系统的分层架构

在系统的底层CPU硬件中需要有硬件辅助虚拟化技术的支持（Intel VT或AMD-V等），宿主机就运行在硬件之上，KVM的内核部分是作为可动态加载内核模块运行在宿主机中的，其中一个模块是和硬件平台无关的实现虚拟化核心基础架构的kvm模块，另一个是硬件平台相关的kvm_intel（或kvm_amd）模块。而KVM中的一个客户机是作为一个用户空间进程（qemu-kvm）运行的，它和其他普通的用户空间进程（如gnome、kde、firefox、chrome等）一样由内核来调度使其运行在

物理CPU上，不过它由KVM模块的控制，可以在前面介绍的三种执行模式下运行。多个客户机就是宿主机中的多个QEMU进程，而一个客户机的多个vCPU就是一个QEMU进程中的多个线程。和普通操作系统一样，在客户机系统中，同样分别运行着客户机的内核和客户机的用户空间应用程序。

4.2.2 SMP的支持

在计算机技术非常普及和日益发达的今天，以Intel、IBM为代表的一些大公司推动着中央处理器（CPU）技术的飞速发展和更新换代，在现代计算机系统中，多处理器、多核、超线程等技术得到了广泛的应用。无论是在企业级和科研应用的服务器领域中，还是个人消费者使用的台式机、笔记本甚至智能手机上，随处可见SMP（Symmetric Multi-Processor，对称多处理器）系统。在SMP系统中，多个程序（进程）可以做到真正的并行执行，而且单个进程的多个线程也可以得到并行执行，这极大地提高了计算机系统并行处理能力和整体性能。

在硬件方面，早期的计算机系统更多的是在一个主板上拥有多个物理的CPU插槽来实现SMP系统，后来随着多核技术、超线程（Hyper-Threading）技术的出现，SMP系统就会使用多处理器、多核、超线程等技术中的一个或多个。多数的现代CPU都支持多核或超线程[2] 技

术，如Intel的Xeon（至强）、Pentium D（奔腾D）、Core Duo（酷睿双核）、Core2 Duo（酷睿二代双核）等系列的处理器和AMD的Athlon64 X2、Quad FX、Opteron 200、Opteron 2000等系列的处理器。

在操作系统软件方面，多数的现代操作系统都提供了对SMP系统的支持，如主流的Linux操作系统（内核2.6及以上对SMP的支持比较完善）、微软的Windows NT系列（包括：Windows 2000、Windows XP、Windows 7、Windows 8等）、Mac OS系统、BSD系统、HP-UX系统、IBM的AIX系统，等等。

在Linux中，例如，下面的Bash脚本（cpu-info.sh）可以根据/proc/cpuinfo文件来检查当前系统中的CPU数量、多核及超线程的使用情况。

```
#!/bin/bash
#filename: cpu-info.sh
#this script only works in a Linux system which has one or more
identical physical CPU(s).

echo -n "logical CPU number in total: "

# 逻辑 CPU 个数

cat /proc/cpuinfo | grep "processor" | wc -l

# 有些系统没有多核也没有打开超线程，就直接退出脚本

cat /proc/cpuinfo | grep -qi "core id"
if [ $? -ne 0 ]; then
      echo "Warning. No multi-core or hyper-threading is enabled."
      exit 0;
fi

echo -n "physical CPU number in total: "

# 物理 CPU 个数

cat /proc/cpuinfo | grep "physical id" | sort | uniq | wc -l

echo -n "core number in a physical CPU: "

# 每个物理 CPU 上 Core 的个数（未计入超线程）

core_per_phy_cpu=$(cat /proc/cpuinfo | grep "core id" | sort | uniq | wc -l)
echo $core_per_phy_cpu

echo -n "logical CPU number in a physical CPU: "

# 每个物理 CPU 中逻辑 CPU（可能是 core, threads 或 both）的个数

logical_cpu_per_phy_cpu=$(cat /proc/cpuinfo | grep "siblings" |
sort | uniq | awk -F: '{print $2}')
echo $logical_cpu_per_phy_cpu

# 是否打开有超线程

# 如果在同一个物理 CPU 上两个逻辑 CPU 具有相同的 "core id"，那么超线程是打开的


# 此处根据前面计算的 core_per_phy_cpu 和 logical_core_per_phy_cpu 的比较来查看超线程

if [ $logical_cpu_per_phy_cpu -gt $core_per_phy_cpu ]; then
       echo "Hyper threading is enabled."
elif [ $logical_cpu_per_phy_cpu -eq $core_per_phy_cpu ]; then
       echo "Hyper threading is NOT enabled."
else
       echo "Error. There's something wrong."
fi
```

SMP是如此的普及和被广泛使用，而QEMU在给客户机模拟CPU时，也可以提供对SMP（Symmetric Multi-processing，对称多处理）架构的模拟，让客户机运行在SMP系统中，充分利用物理硬件的SMP并行处理优势。由于每个vCPU在宿主机中都是一个线程，并且宿主机Linux系统是支持多任务处理的，因此可以通过两种操作来实现客户机的SMP，一是将不同的vCPU的进程交换执行（分时调度，即使物理硬件非SMP，也可以为客户机模拟出SMP系统环境），二是将在物理SMP硬件系统上同时执行多个vCPU的进程。

在qemu-kvm命令行中，"-smp"参数即是为了配置客户机的SMP系统，其具体参数如下：

-smp n[,maxcpus=*cpus*][,cores=*cores*][,threads=*threads*][,sockets=*sockets*]

其中：

❑n用于设置客户机中使用的逻辑CPU数量（默认值是1）。

❑maxcpus用于设置客户机中最大可能被使用的CPU数量，包括启动时处于下线（offline）状态的CPU数量（可用于热插拔hot-plug加入CPU，但不能超过maxcpus这个上限）。

❑cores用于设置每个CPU socket上的core数量（默认值是1）。

❑threads用于设置每个CPU core上的线程数（默认值是1）。

❑sockets用于设置客户机中看到的总的CPU socket数量。

下面通过KVM中的几个QEMU命令行示例来看一下如何将SMP应用于客户机中。

1.示例1

```
qemu-system-x86_64 /root/kvm_demo/rhel6u3.img
```

不加smp参数，使用其默认值1，在客户机中查看CPU情况，如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 2
model name      : QEMU Virtual CPU version 1.1.0
stepping        : 3
cpu MHz         : 3292.520
cache size      : 4096 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
```

```
wp              : yes
flags           : fpu de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov pse36 clflush mmx fxsr sse sse2 syscall nx lm up rep_good
unfair_spinlock pni cx16 popcnt hypervisor lahf_lm
bogomips        : 6585.04
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
[root@kvm-guest ~]# ls /sys/devices/system/cpu/
cpu0  cpufreq  cpuidle  kernel_max  offline  online  possible  present
```

由上面的输出信息可知，客户机系统识别到1个QEMU模拟的CPU（cpu0），并且在qemu monitor（默认通过Alt+Ctrl+2切换到monitor）中用"info cpus"命令可以看到客户机中的CPU数量及其对应qemu-kvm线程的ID，如下所示：

```
(qemu) info cpus
* CPU #0: pc=0xffffffff810375ab thread_id=19668
```

在宿主机中看到相应的QEMU进程和线程如下：

```
[root@jay-linux kvm_demo]# ps -efL | grep qemu
root     19666  4196 19666  0    2 17:17 pts/0    00:00:00
qemu-system-x86_64 /root/kvm_demo/rhel6u3.img
root     19666  4196 19668  0    2 17:17 pts/0    00:00:13
qemu-system-x86_64 /root/kvm_demo/rhel6u3.img
root     19728 15210 19728  0    1 17:20 pts/3    00:00:00 grep
--color=auto qemu
```

由以上信息可以看出，PID（Process ID）为19666的进程是客户机的进程，它产生了1个线程（线程ID为19668）作为客户机的vCPU运行在宿主机中。其中，ps命令的-L参数是为了指定打印出线程的ID和线程的个数，-e参数指定选择所有的进程，-f参数指定选择打印出完全的各列。

2.示例2

```
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img
```

"-smp 8"表示分配了8个虚拟的CPU给客户机，在客户机中用前面提到的"cpu-info.sh"脚本查看CPU情况，如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
<! --  此处省略 /proc/cpuinfo 的输出细节 -->
[root@kvm-guest ~]# ./cpu-info.sh          # 这个是前面提到的 Bash 脚本
logical CPU number in total: 8
Warning. No multi-core or hyper-threading is enabled.
[root@kvm-guest ~]# ls /sys/devices/system/cpu/
cpu0  cpu1  cpu2  cpu3 cpu4  cpu5  cpu6  cpu7 cpufreq  cpuidle
kernel_max  offline  online  possible  present
```

由上面的输出信息可知，客户机使用了8个CPU（cpu0～cpu7），系统识别的CPU数量也总共是8个，是8个物理CPU而没有多核、超线程之类的架构。

在QEMU monitor中查询CPU状态，如下：

```
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) info cpus
* CPU #0: pc=0xffffffff810387cb (halted) thread_id=22290
  CPU #1: pc=0xffffffff810387cb (halted) thread_id=22291
  CPU #2: pc=0xffffffff810387cb (halted) thread_id=22292
  CPU #3: pc=0xffffffff810387cb (halted) thread_id=22293
  CPU #4: pc=0xffffffff810387cb (halted) thread_id=22294
  CPU #5: pc=0xffffffff810387cb (halted) thread_id=22295
  CPU #6: pc=0xffffffff810387cb (halted) thread_id=22296
  CPU #7: pc=0xffffffff810387cb (halted) thread_id=22297
```

在宿主机中看到相应的QEMU进程和线程如下：

```
[root@jay-linux kvm_demo]# ps  -efL | grep qemu
root     22288  4196 22288  0    9 21:10 pts/0    00:00:00
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22290  4    9 21:10 pts/0    00:00:09
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22291  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22292  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22293  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22294  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22295  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22296  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22288  4196 22297  0    9 21:10 pts/0    00:00:01
qemu-system-x86_64 -smp 8 /root/kvm_demo/rhel6u3.img -monitor stdio
root     22341 21654 22341  0    1 21:13 pts/3    00:00:00
grep --color=auto qemu
```

由以上信息可知，PID为22288的进程是QEMU启动客户机的进程，它产生了8个线程（ID为22290～22297）作为客户机的8个vCPU在运行。

3.示例3

```
qemu-system-x86_64 -smp 8,sockets=2,cores=2,threads=2
/root/kvm_demo/rhel6u3.img
```

通过-smp参数详细定义了客户机中SMP的架构，在客户中得到的CPU信息如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
<! --  此处省略 /proc/cpuinfo 的输出细节 -->
[root@kvm-guest ~]# ./cpu-info.sh
logical CPU number in total: 8
physical CPU number in total: 2

core number in a physical CPU: 2
logical CPU number in a physical CPU: 4
Hyper threading is enabled.
[root@kvm-guest ~]# ls /sys/devices/system/cpu/
cpu0  cpu2  cpu4  cpu6  cpufreq  kernel_max  online    present
sched_smt_power_savings
cpu1  cpu3  cpu5  cpu7  cpuidle  offline     possible  sched_mc_power_savings
```

从上面的输出信息可知，客户机中共有8个逻辑CPU（cpu0～cpu7），2个CPU socket，每个socket有2个核，每个核有2个线程（超线程处于打开状态）。

4.示例4

```
qemu-system-x86_64 -smp 4,maxcpus=8 /root/kvm_demo/rhel6u3.img
```

通过-smp参数详细定义了客户机中最多有8个CPU可用，在系统启动之时有4个处于开启状态，在客户中得到的CPU信息如下：

```
[root@kvm-guest ~]# ./cpu-info.sh
logical CPU number in total: 4
Warning. No multi-core or hyper-threading is enabled.
```

如果要对客户机进行CPU的热插拔（hot-plug），则需要在启动客户机的qemu-kvm命令行参数中加上"maxcpus=num"这个选项。不过，目前qemu-kvm中CPU的hot-plug功能有一些bug，处于不可用状态，可参考http://www.linux-kvm.org/page/CPUHotPlug了解详情。

4.2.3 CPU过载使用

KVM允许客户机过载使用（over-commit）物理资源，即允许为客户机分配的CPU和内存数量多于物理上实际存在的资源。

物理资源的过载使用能带来资源充分利用方面的好处。试想在一台强大的硬件服务器中运行Web服务器、图片存储服务器、后台数据统计服务器等作为虚拟客户机，但是它们不会在同一时刻都负载很高，如Web服务器和图片服务器在白天工作时间负载较重，而后台数据统计服务器主要在晚上工作，所以对物理资源进行合理的过载使用，向这几个客户机分配的系统资源总数多于实际拥有的物理资源，就可能在白天和夜晚都充分利用物理硬件资源，而且由于几个客户机不会同时对物理资源造成很大的压力，它们各自的服务质量(QoS)也能得到保障。

CPU的过载使用，是让一个或多个客户机使用vCPU的总数量超过实际拥有的物理CPU数量。QEMU会启动更多的线程来为客户机提供服务，这些线程也是被Linux内核调度运行在物理CPU硬件上。

关于CPU的过载使用，推荐的做法是对多个单CPU的客户机使用over-commit，比如，在拥有4个逻辑CPU的宿主机中，同时运行多于4个（如8个、16个）客户机，其中每个客户机都分配一个vCPU。这时，如果每个宿主机的负载不是很大，宿主机Linux对每个客户机的调度是非常有效的，这样的过载使用并不会带来客户机的性能损失。

关于CPU的过载使用，最不推荐的做法是让某一个客户机的vCPU数量超过物理系统上存在的CPU数量。比如，在拥有4个逻辑CPU的宿主机中，同时运行一个或多个客户机，其中每个客户机的vCPU数量多于4个（如16个）。这样的使用方法会带来比较明显的性能下降，其性能反而不如为客户机分配2个（或4个）vCPU的情况，而且如果客户机中负载过重，可能会让整个系统运行不稳定。不过，在并非100%满负载的情况下，一个（或多个）有4个vCPU的客户机运行在拥有4个逻辑CPU的宿主机中并不会带来明显的性能损失。

总的来说，KVM允许CPU的过载使用，但是并不推荐在实际的生产环境（特别是负载较重的环境）中过载使用CPU。在生产环境中过载使用CPU，有必要在部署前进行严格的性能和稳定性测试。

4.2.4 CPU模型

每一种虚拟机管理程序（Virtual Machine Monitor，简称VMM或Hypervisor）都会定义自己的策略，让客户机看起来有一个默认的CPU

类型。有的Hypervisor会简单地将宿主机中CPU的类型和特性直接传递给客户机使用，而QEMU/KVM在默认情况下会向客户机提供一个名为qemu64或qemu32的基本CPU模型。QEMU/KVM的这种策略会带来一些好处，如可以对CPU特性提供一些高级的过滤功能，还可以将物理平台根据提供的基本CPU模型进行分组（如几台IvyBridge和Sandybridge硬件平台分为一组，都提供相互兼容的SandyBridge或qemu64的CPU模型），从而让客户机在同一组硬件平台上的动态迁移更加平滑和安全。

通过如下的命令行可以查看当前的QEMU支持的所有CPU模型。

```
[root@jay-linux ~]# qemu-system-x86_64 -cpu ?
x86       Opteron_G4
x86       Opteron_G3
x86       Opteron_G2
x86       Opteron_G1
x86      SandyBridge
x86         Westmere
x86          Nehalem
x86           Penryn
x86           Conroe
x86           [n270]
x86         [athlon]
x86       [pentium3]
x86       [pentium2]
x86        [pentium]
x86            [486]
x86        [coreduo]
x86          [kvm32]
x86         [qemu32]


x86           [kvm64]
x86        [core2duo]
x86          [phenom]
x86          [qemu64]
```

其中，加了方括号的"qemu64"、"core2duo"、"kvm64"、"kvm32"等CPU模型是QEMU命令中原生自带（built-in）的，而未加方括号的"SandyBridge"、"Westmere"、"Nehalem"等CPU模型是在配置文件中

配置的。原生自带的CPU模型是在源代码qemu-kvm.git/target-i386/cpu.c中的结构体数组builtin_x86_defs[]中定义的，而用于自定义配置CPU模型的文件在源代码仓库中为qemu-kvm.git/sysconfigs/target/cpus-x86_64.conf（安装后的路径一般为/usr/local/share/qemu/cpus-x86_64.conf）。不过，在qemu-kvm-1.3.0版本中，QEMU开发者删除了cpus-x86_64.conf这个文件，而将所有CPU模型的定义都放在了target-i386/cpu.c文件中。

在x86-64平台上编译和运行的QEMU，在不加"-cpu"参数启动时，采用"qemu64"作为默认的CPU模型，命令行演示如下：

```
qemu-system-x86_64 /root/kvm_demo/rhel6u3.img
```

在客户机中看到的CPU信息如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 2
model name      : QEMU Virtual CPU version 1.1.0
stepping        : 3
cpu MHz         : 3292.520
cache size      : 4096 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 4
wp              : yes
flags           : fpu de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov pse36 clflush mmx fxsr sse sse2 syscall nx lm up rep_good
unfair_spinlock pni cx16 popcnt hypervisor lahf_lm
bogomips        : 6585.04
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

由上面信息可知，在客户机中看到的CPU模型的名称为"QEMU Virtual CPUversion 1.1.0"这就是当前版本QEMU的"qemu64"CPU模型的名称。

在QEMU命令行中，可以用"-cpu cpu_model"来指定在客户机中的CPU模型。如下的命令行就在启动客户机时指定了CPU模型为SandyBridge。

```
qemu-system-x86_64 /root/kvm_demo/rhel6u3.img -cpu SandyBridge
```

在客户机中查看CPU信息如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel Xeon E312xx (Sandy Bridge)
stepping        : 1
cpu MHz         : 3292.520
cache size      : 4096 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm
constant_tsc up arch_perfmon rep_good unfair_spinlock pni
pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt aes xsave avx
hypervisor lahf_lm xsaveopt
bogomips        : 6585.04
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

由上面的信息可知，在客户机中看到的CPU是基于SandyBridge的Intel Xeon处理器。其中的vendor_id、cpu family、flags、cpuid level等都是在cpus-x86_64.conf文件中配置好的。这里的SandyBridge这个CPU模型在cpus-x86_64.conf中的配置如下：

```
[cpudef]
name = "SandyBridge"
level = "0xd"
vendor = "GenuineIntel"
family = "6"
model = "42"
stepping = "1"
   feature_edx = " sse2 sse fxsr mmx clflush pse36 pat cmov mca
pge mtrr sep apic cx8 mce pae msr tsc pse de fpu"
   feature_ecx = "avx xsave aes tsc-deadline popcnt x2apic sse4.2
sse4.1 cx16 ssse3 pclmulqdq sse3"
   extfeature_edx = "i64 rdtscp nx syscall "
   extfeature_ecx = "lahf_lm"
xlevel = "0x8000000A"
   model_id = "Intel Xeon E312xx (Sandy Bridge)"
```

4.2.5 进程的处理器亲和性和vCPU的绑定

通常在SMP系统中，Linux内核的进程调度器根据自有的调度策略将系统中的一个进程调度到某个CPU上执行。一个进程在前一个执行时间是在cpuM（M为系统中的某CPU的ID）上运行，而在后一个执行时间是在cpuN（N为系统中另一CPU的ID）上运行。这样的情况在Linux中是很可能发生的，因为Linux对进程执行的调度采用时间片法则（即用完自己的时间片立即暂停执行），而在默认情况下，一个普通进程或线程的处理器亲和性体现在所有可用的CPU上，进程或线程有可能在这些CPU之中的任何一个（包括超线程）上执行。

进程的处理器亲和性（Processor Affinity），即CPU的绑定设置，是指将进程绑定到特定的一个或多个CPU上去执行，而不允许将进

程调度到其他的CPU上。Linux内核对进程的调度算法也是遵守进程的处理器亲和性设置的。设置进程的处理器亲和性带来的好处是可以减少进程在多个CPU之间交换运行带来的缓存命中失效（cache missing），从该进程运行的角度来看，可能带来一定程度上的性能提升。换个角度来看，对进程亲和性的设置也可能带来一定的问题，如破坏了原有SMP系统中各个CPU的负载均衡（load balance），这可能会导致整个系统的进程调度变得低效。特别是在多处理器、多核、多线程技术使用的情况下，在NUMA（Non-Uniform Memory Access）[3]结构的系统中，如果不能基于对系统的CPU、内存等有深入的了解，对进程的处理器亲和性进行设置可能导致系统的整体性能的下降而非提升。

每个vCPU都是宿主机中的一个普通的QEMU线程，可以使用taskset工具对其设置处理器亲和性，使其绑定到某一个或几个固定的CPU上去调度。尽管Linux内核的进程调度算法已经非常高效了，在多数情况下不需要对进程的调度进行干预，不过，在虚拟化环境中有时有必要将客户机的QEMU进程或线程绑定到固定的逻辑CPU上。下面举一个云计算应用中需要绑定vCPU的示例。

作为IAAS（Infrastructure As A Service）类型的云计算提供商的A公司（如Amazon、Google、阿里云、盛大云等），为客户提供一个有两个逻辑CPU计算能力的一个客户机。要求CPU资源独立被占用，不

受宿主机中其他客户机的负载水平的影响。为了满足这个需求，可以分如下两个步骤来实现。

第一步，启动宿主机时隔离出两个逻辑CPU专门供一个客户机使用。在Linux内核启动的命令行加上"isolcpus="参数，可以实现CPU的隔离，使得在系统启动后普通进程默认都不会调度到被隔离的CPU上执行。例如，隔离了cpu2和cpu3的grub的配置文件如下：

```
title Red Hat Enterprise Linux Server (3.5.0)
        root (hd0,0)
        kernel /boot/vmlinuz-3.5.0 ro
root=UUID=1a65b4bb-cd9b-4bbf-97ff-7e1f7698d3db isolcpus=2,3
        initrd /boot/initramfs-3.5.0.img
```

在系统启动后，在宿主机中检查是否隔离成功，命令行如下：

```
[root@jay-linux ~]# ps -eLo psr | grep 0 | wc -l
106
[root@jay-linux ~]# ps -eLo psr | grep 1 | wc -l
107
[root@jay-linux ~]# ps -eLo psr | grep 2 | wc -l
```

```
4
[root@jay-linux ~]# ps -eLo psr | grep 3 | wc -l
4
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | awk
'{if($5==2) print $0}'
root        10     2    10   2 [migration/2]
root        11     2    11   2 [kworker/2:0]
root        12     2    12   2 [ksoftirqd/2]
root       245     2   245   2 [kworker/2:1]
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | awk
'{if($5==3) print $0}'
root        13     2    13   3 [migration/3]
root        14     2    14   3 [kworker/3:0]
root        15     2    15   3 [ksoftirqd/3]
root       246     2   246   3 [kworker/3:1]
```

由上面的命令行输出信息可知，cpu0和cpu1上分别有106和107个线程在运行，而cpu2和cpu3上分别只有4个线程在运行。而且，根据输出信息中cpu2和cpu3上运行的线程信息（也包括进程在内），分别有migration进程（用于进程在不同CPU间迁移）、两个kworker进程（用于处理workqueues）、ksoftirqd进程（用于调度CPU软中断的进程），这些进程都是内核对各个CPU的一些守护进程。没有其他的普通进程在cup2和cpu3上运行，说明对它们的隔离是生效的。

这里简单解释一下上面的一些命令行工具及其参数的意义。ps命令显示当前系统的进程信息的状态，它的"-e"参数用于显示所有的进程，"-L"参数用于将线程（LWP，light-weight process）也显示出来，"-o"参数表示以用户自定义的格式输出（其中"psr"这列表示当前分配给进程运行的处理器编号，"lwp"列表示线程的ID，"ruser"表示运行进程的用户，"pid"表示进程的ID，"ppid"表示父进程的

ID，"args"表示运行的命令及其参数）。结合ps和awk工具的使用，是为了分别将在处理器cpu2和cpu3上运行的进程打印出来。

第二步，启动一个拥有两个vCPU的客户机并将其vCPU绑定到宿主机中两个CPU上。此操作过程的命令行如下：

```
# 启动一个客户机
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 512 -daemonize
VNC server running on '::1:5900'
# 查看代表 vCPU 的 QEMU 线程
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | grep
qemu | grep -v grep
root      3963     1  3963   0 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
root      3963     1  3967   0 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
root      3963     1  3968   1 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
```

绑定代表整个客户机的 QEMU 进程，使其运行在 cpu2 上

```
[root@jay-linux ~]# taskset -p 0x4 3963
pid 3963's current affinity mask: 3
pid 3963's new affinity mask: 4
```

绑定第一个 vCPU 的线程，使其运行在 cpu2 上

```
[root@jay-linux ~]# taskset -p 0x4 3967
pid 3967's current affinity mask: 3
pid 3967's new affinity mask: 4
```

绑定第二个 vCPU 的线程，使其运行在 cpu3 上

```
[root@jay-linux ~]# taskset -p 0x8 3968
pid 3968's current affinity mask: 4
pid 3968's new affinity mask: 8
```

查看 QEMU 线程的绑定是否生效，下面的第 5 列为处理器亲和性

```
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | grep
qemu | grep -v grep
root      3963     1  3963   2 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
root      3963     1  3967   2 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
root      3963     1  3968   3 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
```

执行 vCPU 的绑定后，查看在 cpu2 上运行的线程

```
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | awk
'{if($5==2) print $0}'
root        10     2    10   2 [migration/2]
root        11     2    11   2 [kworker/2:0]
root        12     2    12   2 [ksoftirqd/2]
root       245     2   245   2 [kworker/2:1]
root      3963     1  3963   2 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
root      3963     1  3967   2 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
```

执行 vCPU 的绑定后，查看在 cpu3 上运行的线程

```
[root@jay-linux ~]# ps -eLo ruser,pid,ppid,lwp,psr,args | awk
'{if($5==3) print $0}'
root        13     2    13   3 [migration/3]
root        14     2    14   3 [kworker/3:0]
root        15     2    15   3 [ksoftirqd/3]
root       246     2   246   3 [kworker/3:1]
root      3963     1  39683 qemu-system-x86_64 rhel6u3.img
-smp 2 -m 512 -daemonize
```

由上面的命令行及其输出信息可知，在CPU进行绑定之前，代表这个客户机的QEMU进程和代表各个vCPU的QEMU线程分别被调度到cpu0和cpu1上。使用taskset命令将QEMU进程和第一个vCPU的线程绑定到cpu2，将第二个vCPU线程绑定到cpu3上。在绑定之后即可查看到绑定结果生效，代表两个vCPU的QEMU线程分别运行在cpu2和cpu3上（即使再过一段时间，它们也不会被调度到其他CPU上去）。

对于taskset命令，此处使用的语法是：taskset-p[mask]pid。其中，mask是一个代表了处理器亲和性的掩码数字，转化为二进制表示后，其值从最低位到最高位分别代表了第一个逻辑CPU到最后一个逻辑CPU，进程调度器可能将该进程调度到所有标志为“1”的位代表的逻辑CPU上去运行。根据上面的输出，在运行taskset命令之前，QEMU线程的处理器亲和性mask值是0x3（其二进制值为0011），可知其可能会被调度到cpu0和cpu1上运行；而在运行"taskset-p 0x4 3967"命令后，提示新的mask值被设为0x4（其二进制值为0100），所以该进程就只能被调度到cpu2上去运行，即通过taskset工具实现了将vCPU进程绑定到特定的CPU上。

在上面命令行中，根据ps命令可以看到QEMU的线程和进程的关系，但如何查看vCPU与QEMU线程之间的关系呢？可以切换（使用"Ctrl+Alt+2"快捷键）到QEMU monitor中进行查看，运行"info

cpus"命令即可（还记得3.6节中运行过的"info kvm"命令吧），其输出结果如下：

```
(qemu) info cpus
* CPU #0: pc=0xffffffff810375ab thread_id=3967
CPU #1: pc=0xffffffff812b2594 thread_id=3968
```

由上面的输出信息可知，客户机中的cpu0对应的线程ID为3967，cpu1对应的线程ID为3968。另外，"CPU#0"前面有一个星号（*），是标识cpu0是BSP（Boot Strap Processor，系统最初启动时在SMP生效前使用的CPU）。

4.3 内存配置

在计算机系统中，内存是一个非常重要的部件，它是与CPU沟通的一个桥梁，其作用是暂时存放CPU中将要执行的指令和数据，所有程序的运行都必须先载入到内存中才能够执行。内存的大小及其访问速度也直接影响整个系统性能，所以在虚拟机系统中，对内存的虚拟化处理和配置也是比较关键的。本节主要介绍KVM中内存的配置。

4.3.1 内存设置基本参数

在通过QEMU命令行启动客户机时设置内存大小的参数如下：

-m megs #设置客户机的内存为megs MB大小

默认的单位为MB，也支持加上"M"或"G"作为后缀来显式指定使用MB或GB作为内存分配的单位。如果不设置-m参数，QEMU对客户机分配的内存大小默认值为128MB。

下面通过3个示例用来进一步说明"-m"参数设置内存的具体用法。

1.示例1

```
qemu-system-x86_64 rhel6u3.img
```

在客户机中，查看到的内存信息如下：

```
[root@kvm-guest ~]# free -m
             total       used       free     shared    buffers
cached
Mem:           112        103          8          0          9
33
-/+ buffers/cache:         60         52
Swap:          510          0        510
[root@kvm-guest ~]# dmesg
<!-- 此处省略百余行输出信息  -->
PID hash table entries: 512 (order: 0, 4096 bytes)
Checking aperture...
No AGP bridge found
Memory: 95744k/131064k available (5151k kernel code, 392k absent,
34928k reserved, 7166k data, 1260k init)
Hierarchical RCU implementation.
NR_IRQS:33024 nr_irqs:256
Console: colour VGA+ 80x25
console [tty0] enabled
<!-- 此处省略数百行输出信息  -->
```

其中，free命令用于查看内存的使用情况，"-m"参数是内存大小以MB为单位来显示，以上信息中显示总的内存为112MB，这个值与128MB有一定差距，其原因是free命令显示的总内存是除去了内核执行文件占用内存和一些系统保留的内存之后能使用的内存。而通过dmesg命令显示的内核打印的信息可以看出，内核检测到总的内存为131064 KB，几乎是完完整整的128MB内存了（128*1024=131072，与131064非常接近）。

2.示例2

```
qemu-system-x86_64 rhel6u3.img -m 1024
```

在客户机中，查看内存信息如下：

```
[root@kvm-guest ~]# dmesg
<!-- 此处省略百余行输出信息  -->
PID hash table entries: 4096 (order: 3, 32768 bytes)
Checking aperture...
No AGP bridge found
Memory: 1000932k/1048568k available (5151k kernel code, 392k
absent, 47244k reserved, 7166k data, 1260k init)
Hierarchical RCU implementation.
NR_IRQS:33024 nr_irqs:256
```

```
Console: colour VGA+ 80x25
console [tty0] enabled
<!-- 此处省略数百行输出信息  -->
[root@kvm-guest ~]# cat /proc/meminfo
MemTotal:        1020752 kB
MemFree:          849204 kB
Buffers:           40872 kB
Cached:            58372 kB
<!-- 此处省略数十行输出信息  -->
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        8184 kB
DirectMap2M:     1040384 kB
```

由上面输出信息可知，dmesg中看到的内存总量为1048568kB，约等于1024*1024kB，而通过/proc/meminfo看到的"MemTotal"的大小为1048568kB，比1024MB稍小，其原因与前面free命令输出的总内存是一样的。

3.示例3

```
qemu-system-x86_64 rhel6u3.img -m 2G
```

在客户机中，查看内存信息如下：

```
[root@kvm-guest ~]# dmesg
<!-- 此处省略数百行输出信息  -->
PID hash table entries: 4096 (order: 3, 32768 bytes)
Checking aperture...
No AGP bridge found
Memory: 1903072k/2097144k available (5151k kernel code, 392k
absent, 193680k reserved, 7166k data, 1260k init)
Hierarchical RCU implementation.
NR_IRQS:33024 nr_irqs:256
Console: colour VGA+ 80x25
console [tty0] enabled
<!-- 此处省略数百行输出信息  -->
[root@kvm-guest ~]# free -m
             total       used       free     shared    buffers
cached
Mem:          1877        346       1531          0        140
77
-/+ buffers/cache:        128       1749
Swap:          510          0        510
```

由上面输出信息可知，在dmesg中看到的内存总量为2097144KB，约等于2GB，说明使用"G"作为-m参数的内存的单位已经生效。

4.3.2 EPT和VPID简介

EPT（Extended Page Tables，扩展页表），属于Intel的第二代硬件虚拟化技术，它是针对内存管理单元（MMU）的虚拟化扩展。EPT降低了内存虚拟化的难度（与影子页表相比），也提升了内存虚拟化的性能。从基于Intel的Nehalem[4] 架构的平台开始，EPT就作为CPU的一个特性加入到CPU硬件中去了。

和运行在真实物理硬件上的操作系统一样，在客户机操作系统看来，客户机可用的内存空间也是一个从零地址开始的连续的物理内存空间。为了达到这个目的，Hypervisor（即KVM）引入了一层新的地址空间，即客户机物理地址空间，这个地址空间不是真正的硬件上的地址空间，它们之间还有一层映射。所以，在虚拟化环境下，内存使用就需要两层的地址转换，即客户机应用程序可见的客户机虚拟地址（Guest Virtual Address,GVA）到客户机物理地址（Guest Physical Address，GPA）的转换，再从客户机物理地址（GPA）到宿主机物理地址（Host Physical Address，HPA）的转换。其中，前一个转换由客户机操作系统来完成，而后一个转换由Hypervisor来负责。

在硬件EPT特性加入之前，影子页表（Shadow Page Tables）是从软件上维护了从客户机虚拟地址（GVA）到宿主机物理地址（HPA）之间的映射，每一份客户机操作系统的页表也对应一份影子页表。有了影子页表，在普通的内存访问时都可实现从GVA到HPA的直接转换，从而避免了上面前面提到的两次地址转换。Hypervisor将影子页表载入到物理上的内存管理单元（Memory Management Unit，MMU）中进行地址翻译。图4-3展示了，GVA、GPA、HPA之间的转换，以及影子页表的作用。

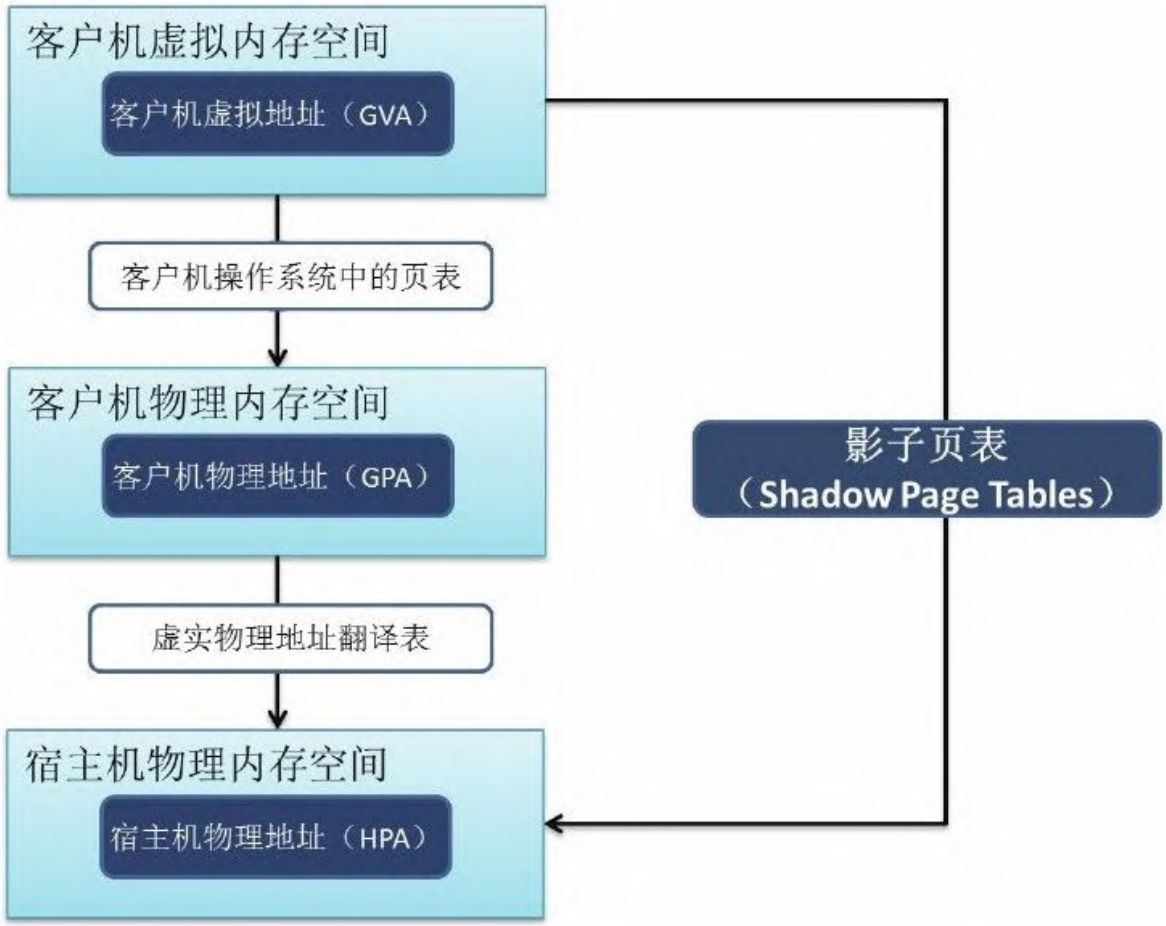


图　4-3　影子页表的作用

尽管影子页表提供了在物理MMU硬件中能使用的页表，但是其缺点也是比较明显的。首先影子页表实现非常复杂，导致其开发、调试和维护都比较困难。其次，影子页表的内存开销也比较大，因为需要为每个客户机进程对应的页表的都维护一个影子页表。

为了解决影子页表存在的问题，Intel的CPU提供了EPT技术（AMD提供的类似技术叫做NPT，即Nested Page Tables），直接在硬件上支持GVA-->GPA-->HPA的两次地址转换，从而降低内存虚拟化实现的复杂度，也进一步提升了内存虚拟化的性能。图4-4展示了Intel EPT技术的基本原理。

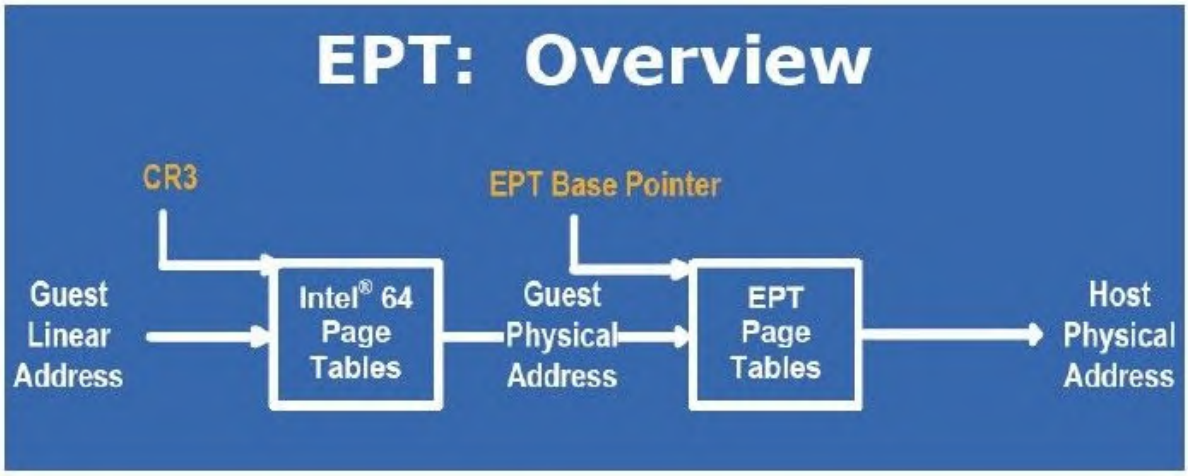


图　4-4　EPT基本原理

CR3（控制寄存器3）将客户机程序所见的客户机虚拟地址（GVA）转化为客户机物理地址（GPA），然后在通过EPT将客户机物理地址

（GPA）转化为宿主机物理地址（HPA）。这两次转换地址转换都是由CPU硬件来自动完成的，其转换效率非常高。在使用EPT的情况下，客户机内部的Page Fault、INVLPG（使TLB[5] 项目失效）指令、CR3寄存器的访问等都不会引起VM-Exit，因此大大减少了VM-Exit的数量，从而提高了性能。另外，EPT只需要维护一张EPT页表，而不需要像“影子页表”那样为每个客户机进程的页表维护一张影子页表，从而也减少了内存的开销。VPID（VirtualProcessor Identifiers，虚拟处理器标识），是在硬件上对TLB资源管理的优化，通过在硬件上为每个TLB项增加一个标识，用于不同的虚拟处理器的地址空间，从而能够区分开Hypervisor和不同处理器的TLB。硬件区分了不同的TLB项分别属于不同虚拟处理器，因此可以避免每次进行VM-Entry和VM-Exit时都让TLB全部失效，提高了VM切换的效率。由于有了这些在VM切换后仍然继续存在的TLB项，硬件减少了一些不必要的页表访问，减少了内存访问次数，从而提高了Hypervisor和客户机的运行速度。VPID也会对客户机的实时迁移（Live Migration）有很好的效率提升，会节省实时迁移的开销，会提升实时迁移的速度，降低迁移的延迟（Latency）。VPID与EPT是一起加入到CPU中的特性，也是Intel公司在2009年推出Nehalem系列处理器上新增的与虚拟化相关的重要功能。

在Linux操作系统中，可以通过如下命令查看/proc/cpuinfo中的CPU标志来确定当前系统是否支持EPT和VPID功能。

```
[root@jay-linux kvm_demo]# grep ept /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr

pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni dtes64
monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1
sse4_2 x2apic popcnt lahf_lm ida arat epb dtherm tpr_shadow vnmi
flexpriority ept vpid
[root@jay-linux kvm_demo]# grepvpid /proc/cpuinfo
```

在宿主机中，可以根据sysfs[6] 文件系统中kvm_intel模块的当前参数值来确定KVM是否打开EPT和VPID特性。在默认情况下，如果硬件支持了EPT、VPID，则kvm_intel模块加载时默认开启EPT和VPID特性，这样KVM会默认使用它们。

```
[root@jay-linux kvm_demo]# cat /sys/module/kvm_intel/parameters/ept
Y
[root@jay-linux kvm_demo]# cat /sys/module/kvm_intel/parameters/vpid
Y
```

在加载kvm_intel模块时，可以通过设置ept和vpid参数的值来打开或关闭EPT和VPID。当然，如果kvm_intel模块已经处于加载状态，则需要先卸载这个模块，在重新加载之时加入所需的参数设置。当然，一般不要手动关闭EPT和VPID功能，否则会导致客户机中内存访问的性能下降。

```
[root@jay-linux ~]# modprobe kvm_intel ept=0,vpid=0
[root@jay-linux ~]# rmmod kvm_intel
[root@jay-linux ~]# modprobe kvm_intelept=1,vpid=1
```

4.3.3 大页（Huge Page）

x86（包括x86-32和x86-64）架构的CPU默认使用4KB大小的内存页面，但是它们也支持较大的内存页，如x86-64系统就支持2MB大小的大页（huge page）。Linux2.6及以上的内核都支持huge page。如果在系统中使用了huge page，则内存页的数量会减少，从而需要更少的页表（page table），节约了页表所占用的内存数量，并且所需的地址转换也减少了，TLB缓存失效的次数就减少了，从而提高了内存访问的性能。另外，由于地址转换所需的信息一般保存在CPU的缓存中，huge page的使用让地址转换信息减少，从而减少了CPU缓存的使用，减轻了CPU缓存的压力，让CPU缓存能更多地用于应用程序的数据缓存，也能够在整体上提升系统的性能。

在KVM中，也可以将huge page的特性应用到客户机中，qemu-kvm就提供了"-mem-path FILE"参数选项用于使用huge page。另外，还有一个参数"-mem-prealloc"可以让宿主机在启动客户机时就全部分配好客户机的内存，而不是在客户机实际用到更多内存时才按需分配。-mem-prealloc必须在有"-mem-path"参数时才能使用。提前分配好内存的好处是客户机的内存访问速度更快，缺点是客户机启动时就得到了所有的内存，从而让宿主机的内存很快减少了（而不是根据客户机的需求而动态调整内存分配）。

可以通过在宿主机中的如下几个操作让客户机使用huge page。

（1）检查宿主机目前状态，检查了默认的内存大小和内存使用情况，如下：

```
[root@jay-linux ~]# getconf PAGESIZE
4096
[root@jay-linux ~]# cat /proc/meminfo
<!-- 此处省略了数十行输出信息 -->
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        8484 kB
DirectMap2M:      915456 kB
DirectMap1G:    48234496 kB
```

（2）挂载hugetlbfs文件系统，命令为"mount-t hugetlbfs hugetlbfs/dev/hugepages"，如下：

```
[root@jay-linux ~]# mount -t hugetlbfs hugetlbfs /dev/hugepages
[root@jay-linux ~]# mount
/dev/sda1 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sda2 on /home type ext4 (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
gvfs-fuse-daemon on /root/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev)
hugetlbfs on /dev/hugepages type hugetlbfs (rw)
```

（3）设置hugepage的数量，命令为"sysctl vm.nr_hugepages=num"，如下：

```
[root@jay-linux ~]# sysctl vm.nr_hugepages=1024
vm.nr_hugepages = 1024
[root@jay-linux ~]# cat /proc/meminfo
<!-- 此处省略了数十行输出信息 -->
HugePages_Total:    1024
HugePages_Free:     1024
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        8484 kB
DirectMap2M:      915456 kB
DirectMap1G:    48234496 kB
```

（4）启动客户机让其使用hugepage的内存，使用"-mem-path"参数，如下：

```
[root@jay-linux kvm_demo]#qemu-system-x86_64 -m 1024 -smp 2
rhel6u3.img -mem-path /dev/hugepages
```

（5）查看宿主机中huge page的使用情况，可以看到"HugePages_Free"数量减少，因为客户机使用了一定数量的hugepage。在如下的输出中，"HugePages_Free"数量的减少没有512（512*2MB=1024MB）那么多，这是因为启动客户机时并没有实际分配1024MB内存，qemu-kvm命令行中加上前面提到的"-mem-prealloc"参

数就会让meminfo文件中"HugePages_Free"的数量减少和分配给客户机的一致。

```
[root@jay-linux ~]# cat /proc/meminfo
<!-- 此处省略了数十行输出信息 -->
HugePages_Total:     1024
HugePages_Free:       897
HugePages_Rsvd:       393
HugePages_Surp:         0
Hugepagesize:        2048 kB
DirectMap4k:         8484 kB
DirectMap2M:       915456 kB
DirectMap1G:     48234496 kB
```

至此，如果在客户机中运行的应用程序具备使用huge page的能力，那么就可以在客户机中使用huge page带来性能的提升。

总的来说，对于内存访问密集型的应用，在KVM客户机中使用huge page是可以较明显地提高客户机性能的，不过，它也有一个缺点，使用huge page的内存不能被换出（swap out），也不能使用ballooning方式自动增长。

4.3.4 内存过载使用

同4.2.3节中介绍的CPU过载使用类似，在KVM中内存也是允许过载使用（over-commit）的，KVM能够让分配给客户机的内存总数大于实际可用的物理内存总数。由于客户机操作系统及其上的应用程序并非一直100%地利用其分配到的内存，并且宿主机上的多个客户机一般也不会同时达到100%的内存使用率，所以内存过载分配是可行的。一般来说，有如下三种方式来实现内存的过载使用。

1）内存交换（swapping）：用交换空间（swap space）来弥补内存的不足。

2）气球（ballooning）：通过virio_balloon驱动来实现宿主机Hypervisor和客户机之间的协作。

3）页共享（page sharing）：通过KSM（Kernel Samepage Merging）合并多个客户机进程使用的相同内存页。

其中，第一种内存交换的方式是最成熟的（Linux中很早就开始应用），也是目前广泛使用的，不过，相比KSM和ballooning的方式效率较低一些。ballooning和KSM将在第5章中详细介绍，本章主要介绍利用swapping这种方式实现内存过载使用。

KVM中客户机是一个QEMU进程，宿主机系统没有特殊对待它而分配特定的内存给QEMU，只是把它当做一个普通Linux进程。Linux内核在进程请求更多内存时才分配给它们更多的内存，所以也是在客户机操

作系统请求更多内存时，KVM才向其分配更多的内存。用swapping方式来让内存过载使用，要求有足够的交换空间（swap space）来满足所有的客户机进程和宿主机中其他进程所需内存。可用的物理内存空间和交换空间的大小之和应该等于或大于配置给所有客户机的内存总和，否则，在各个客户机内存使用同时达到较高比率时可能会有客户机（因内存不足）被强制关闭。

下面通过一个实际的例子来说明如何计算应该分配的交换空间大小以满足内存的过载使用。

某个服务器有32GB的物理内存，想在其上运行64个内存配置为1GB的客户机。在宿主机中，大约需要4GB大小的内存来满足系统进程、驱动、磁盘缓存及其他应用程序所需内存（不包括客户机进程所需内存）。计算过程如下：

客户机所需交换分区为：64 x 1GB+4GB-32GB=36GB。

根据Redhat的建议[7]，对于32GB物理内存的RHEL系统，推荐使用8GB的交换分区。

所以，在宿主机中总共需要建立44GB（36GB+8GB）的交换分区来满足安全实现客户机内存的过载使用。

下面是在一台Ivy Bridge桌面级的硬件平台上进行的简单实验，可以看出客户机并非一开始就在宿主机中占用其启动时配置的内存。

在宿主机中，在启动客户机之前和之后查看到的系统内存情况如下：

```
[root@jay-linux ~]# free -m
total       used       free     shared    buffers     cached
Mem:          3851       1006       2845          0        172        444
-/+ buffers/cache:        388       3462
Swap:         4095          0       4095
[root@jay-linux ~]# qemu-system-x86_64-m 2048 -smp 4 -hda
/root/kvm_demo/rhel6u3.img -daemonize
[root@jay-linux ~]# free -m
             total       used       free     shared    buffers
cached
Mem:          3851       1454       2397          0        172
531
-/+ buffers/cache:        749       3101
Swap:         4095          0       4095
```

在客户机中，查看内存使用情况如下：

```
[root@kvm-guest ~]# free -m
             total       used       free     shared    buffers
cached
Mem:          1877        246       1630          0         25
130
-/+ buffers/cache:         90       1786
Swap:          510          0        510
```

从理论上来说，供客户机过载使用的内存可以达到实际物理内存的几倍甚至几十倍，不过除非特殊情况，一般不建议过多地过载使用内存。一方面，交换空间通常是由磁盘分区来实现的，其读写速度比物理内存读写速度慢得多，性能并不好；另一方面，过多的内存过载

使用也可能导致系统稳定性降低。所以，KVM允许内存过载使用，但在生产环境中配置内存的过载使用之前，仍然应该根据实际应用进行充分的测试。

4.4 存储配置

在计算机系统中，存储设备存储着系统内的数据、程序等内容，是系统中不可或缺的一部分。特别是处于海量数据时代的今天，一些大型公司的数据量非常巨大，动辄以PB（1PB=10^15 Bytes）来计量，所以对磁盘的存储容量和存取速度都有越来越高的要求。和内存相比，磁盘存取速度要慢得多，不过磁盘的容量一般要比内存大得多，而且磁盘上的数据是永久存储，不像内存中的内容在掉电后就会消失。本节主要以磁盘、光盘等为例介绍KVM中的存储配置。

4.4.1 存储配置和启动顺序

QEMU提供了对多种块存储设备的模拟，包括IDE设备、SCSI设备、软盘、U盘、virtio磁盘等，而且对设备的启动顺序提供了灵活的配置。

1.存储的基本配置选项

在qemu-kvm命令行工具中，主要有如下的参数来配置客户机的存储。

（1）-hda file

将file镜像文件作为客户机中的第一个IDE设备（序号0），在客户机中表现为/dev/hda设备（若客户机中使用PIIX_IDE驱动）或/dev/sda设备（若客户机中使用ata_piix驱动）。如果不指定-hda或-hdb等参数，那么在前面一些例子中提到的"qemu-system-x86_64/root/kvm_demo/rhel6u3.img"就与加上-hda参数来指定镜像文件的效果一样的。另外，也可以将宿主机中的一个硬盘（如/dev/sdb）作为-hda的file参数来使用，从而让整个硬盘模拟为客户机的第一个IDE设备。如果file文件的文件名中包含有英文逗号（“,”），则在书写file时应该使用两个逗号（因为逗号是qemu-kvm命令行中的特殊间隔符，例如用于"-cpu qemu64,+vmx"这样的选项），如使用"-hda my,,file"将my,file这个文件作为客户机的第一个IDE设备。

（2）-hdb file

将file作为客户机中的第二个IDE设备（序号1），在客户机中表现为/dev/hdb或/dev/sdb设备。

（3）-hdc file

将file作为客户机中的第三个IDE设备（序号2），在客户机中表现为/dev/hdc或/dev/sdc设备。

（4）-hdd file

将file作为客户机中的第四个IDE设备（序号3），在客户机中表现为/dev/hdd或/dev/sdd设备。

（5）-fda file

将file作为客户机中的第一个软盘设备（序号0），在客户机中表现为/dev/fd0设备。也可以将宿主机中的软驱（/dev/fd0）作为-fda的file来使用。

（6）-fdb file

将file作为客户机中的第二个软盘设备（序号1），在客户机中表现为/dev/fd1设备。

（7）-cdrom file

将file作为客户机中的光盘CD-ROM，在客户机中通常表现为/dev/cdrom设备。也可以将宿主机中的光驱（/dev/cdrom）作为-cdrom的file来使用。注意，-cdrom参数不能和-hdc参数同时使用，因为"-cdrom"就是客户机中的第三个IDE设备。在通过物理光驱中的光盘或磁盘中ISO镜像文件安装客户机操作系统时（参见3.5节安装客户机），一般会使用-cdrom参数。

（8）-mtdblock file

使用file文件作为客户机自带的一个Flash存储器（通常说的闪存）。

（9）-sd file

使用file文件作为客户机中的SD卡（Secure Digital Card）。

（10）-pflash file

使用file文件作为客户机的并行Flash存储器（Parallel Flash Memory）。

2.详细配置存储驱动器的-driver参数

较新版本的qemu-kvm还提供了"-driver"参数来详细定义一个存储驱动器，该参数的具体形式如下：

```
-drive option[,option[,option[,...]]]
```

为客户机定义一个新的驱动器，它有如下一些选项：

（1）file=file

使用file文件作为镜像文件加载到客户机的驱动器中。

（2）if=interface

指定驱动器使用的接口类型，可用的类型有：ide、scsi、sd、mtd、floopy、pflash、virtio，等等。其中，除了virtio、scsi之外，其余几种类型都在本节的前面介绍过了。virtio将在第5章中介绍，而关于scsi类型的接口，QEMU目前还不支持从SCSI类型设备启动客户机系统，Launchpad上也有一个bug[8] 记录着这个问题。

（3）bus=bus,unit=unit

设置驱动器在客户机中的总线编号和单元编号。

（4）index=index

设置在同一种接口的驱动器中的索引编号。

（5）media=media

设置驱动器中媒介的类型，其值为"disk"或"cdrom"。

（6）snapshot=snapshot

设置是否启用"-snapshot"选项，其可选值为"on"或"off"。当snapshot启用时，QEMU不会将磁盘数据的更改写回到镜像文件中，而是写到临时文件中，当然可以在QEMU monitor中使用"commit"命令强制将磁盘数据的更改保存回镜像文件中。

（7）cache=cache

设置宿主机对块设备数据（包括文件或一个磁盘）访问中的cache情况，可以设置为"none"（或"off"）、"writeback"、"writethrough"等。其默认值是"writethrough"，即“直写模式”，它是在调用write写入数据的同时将数据写入磁盘缓存（disk cache）和后端块设备（block device）中，其优点是操作简单，其缺点是写入数据速度较慢。而"writeback"即“回写模式”，在调用write写入数据时只将数据写入到磁盘缓存中即返回，只有在数据被换出缓存时才将修改的数据写到后端存储中，其优点是写入数据速度较快，其缺点是一旦更新数据在写入后端存储之前遇到系统掉电，数据会无法恢复。"writethrough"和"writeback"在读取数据时都尽量使用缓存，若设置了"cache=none"关闭缓存的方式，QEMU将在调用open系统调用打开镜像文件时使用"O_DIRECT"的标识，所以其读写数据都是绕过缓存直接从块设备中读写的。一些块设备文件（如后面即将介绍的qcow2格式文件）在"writethrough"模式下性能表现很差，如果这时对性能要求比正确性更高，建议使用"writeback"模式。

（8）aio=aio

选择异步IO（Asynchronous IO）的方式，有"threads"和"native"两个值可选。其默认值为"threads"，即让一

个线程池去处理异步IO；而"native"只适用于"cache=none"的情况，就是使用Linux原生的AIO。

（9）format=format

指定使用的磁盘格式，在默认情况下是QEMU自动检测磁盘格式的。

（10）serial=serial

指定分配给设备的序列号。

（11）addr=addr

分配给驱动器控制器的PCI地址，该选项只有在使用virtio接口时才适用。

（12）id=name

设置该驱动器的ID，这个ID可以在QEMU monitor中用"info block"看到。

（13）readonly=on|off

设置该驱动器是否只读。

3.配置客户机启动顺序的参数

前面介绍了各种存储设备的使用参数，它们在客户机中的启动顺序可以用如下的参数设定：

-boot [order=*drives*][,once=*drives*][,menu=*on*|*off*] [,splash=*splashfile*] [,splash-time=*sp-time*]

在QEMU模拟的x86 PC平台中，用"a"、"b"分别表示第一和第二个软驱，用"c"表示第一个硬盘，用"d"表示CD-ROM光驱，用"n"表示从网络启动。其中，默认从硬盘启动，要从光盘启动可以设置"-boot order=d"。"once"表示设置第一次启动的启动顺序，在系统重启（reboot）后该设置即无效，如"-boot once=d"设置表示本次从光盘启动，但系统重启后从默认的硬盘启动。"memu=on|off"用于设置交互式的启动菜单选项（前提是使用的客户机BIOS支持），它的默认值是"menu=off"，表示不开启交互式的启动菜单选择。"splash=splashfile"和"splash-time=sp-time"选项都是在"menu=on"时才有效，将名为splashfile的图片作为logo传递给BIOS来显示，而sp-time是BIOS显示splash图片的时间，其单位是毫秒（ms）。图4-5展示了在使用"-boot order=dc,menu=on"设置后，在客户机启动窗口中按F12进入的启动菜单。

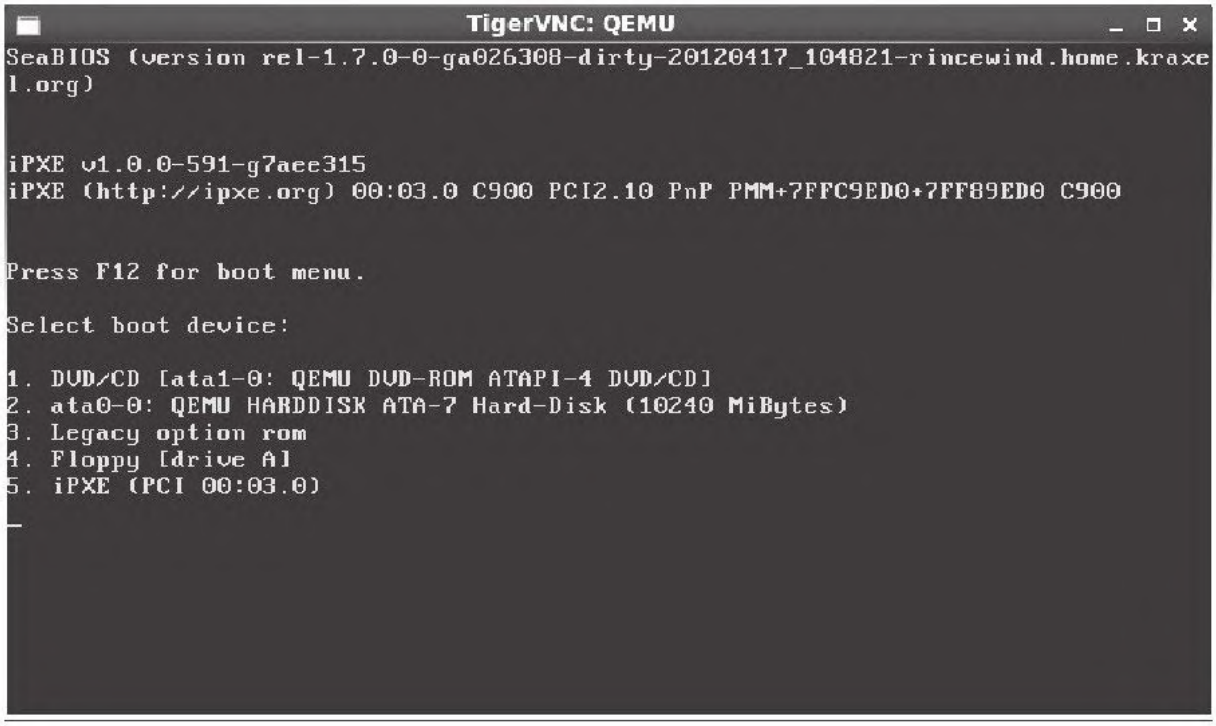


图　4-5　客户机中的启动菜单选项

4. 存储配置的示例

在介绍完基本的参数与启动顺序后，通过示例来看一下磁盘实际配置和在客户机中的效果，通过如下的3个等价命令之一启动一个客户机。

1）qemu-system-x86_64-m 1024-smp 2 rhel6u3.img

2）qemu-system-x86_64-m 1024-smp 2-hda rhel6u3.img

3）qemu-system-x86_64-m 1024-smp 2-drive file=rhel6u3.img,if=ide,cache=writethrough

然后在客户机中查看磁盘情况，如下：

```
[root@kvm-guest ~]# fdisk -l

Disk /dev/hda: 8589 MB, 8589934592 bytes

255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000d0e23

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *           1         980     7864320   83  Linux
Partition 1 does not end on cylinder boundary.
/dev/hda2             980        1045      523264   82  Linux swap
/Solaris[root@kvm-guest ~]# lspci | grep IDE
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
[root@kvm-guest ~]# lspci -vvv -s 00:01.1
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE
[Natoma/Triton II] (prog-if 80 [Master])
        Subsystem: Red Hat, Inc Qemu virtual machine
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV-
VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx-
        Status: Cap- 66MHz- UDF- FastB2B+ ParErr-
DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Region 0: [virtual] Memory at 000001f0 (32-bit,
non-prefetchable) [size=8]
        Region 1: [virtual] Memory at 000003f0 (type 3,
non-prefetchable) [size=1]
        Region 2: [virtual] Memory at 00000170 (32-bit,
non-prefetchable) [size=8]
        Region 3: [virtual] Memory at 00000370 (type 3,
non-prefetchable) [size=1]
        Region 4: I/O ports at c100 [size=16]
        Kernel driver in use: PIIX_IDE
```

由于这个客户机使用的是piix_ide驱动，因此看到的是/dev/hda这样的磁盘，而对于同样一个磁盘，如果客户机操作系统使用的是ata_piix驱动（此时无piix_ide驱动），则看到的是/dev/sda这样的磁盘，效果如下：

```
[root@kvm-guest ~]# lspci -vv -s 00:01.1
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE
[Natoma/Triton II] (prog-if 80 [Master])
        Subsystem: Red Hat, Inc Qemu virtual machine
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV-
VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx-
        Status: Cap- 66MHz- UDF- FastB2B+ ParErr-
DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Region 0: [virtual] Memory at 000001f0 (32-bit,
non-prefetchable) [size=8]
        Region 1: [virtual] Memory at 000003f0 (type 3,
non-prefetchable) [size=1]
        Region 2: [virtual] Memory at 00000170 (32-bit,


non-prefetchable) [size=8]
        Region 3: [virtual] Memory at 00000370 (type 3,
non-prefetchable) [size=1]
        Region 4: I/O ports at c100 [size=16]
        Kernel driver in use: ata_piix
        Kernel modules: ata_generic, pata_acpi, ata_piix

[root@kvm-guest ~]# fdisk -l

Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000d0e23

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1         980     7864320   83  Linux
Partition 1 does not end on cylinder boundary.
/dev/sda2             980        1045      523264   82  Linux swap / Solaris
```

4.4.2 qemu-img命令

qemu-img是QEMU的磁盘管理工具，在完成qemu-kvm源码编译后就会默认编译好qemu-img这个二进制文件。qemu-img也是QEMU/KVM使用

过程中一个比较重要的工具，本节对其用法进行介绍。

qemu-img工具的命令行基本用法如下：

qemu-img command[command options]

它支持的命令分为如下几种。

（1）check[-f fmt]filename

对磁盘镜像文件进行一致性检查，查找镜像文件中的错误，目前仅支持对"qcow2"、"qed"、"vdi"格式文件的检查。其中，qcow2是QEMU 0.8.3版本引入的镜像文件格式，也是目前使用最广泛的格式。qed（QEMU enhanced disk）是从QEMU 0.14版开始加入的增强磁盘文件格式，避免了qcow2格式的一些缺点，也提高了性能，不过目前还不够成熟。而vdi（Virtual Disk Image）是Oracle的VirtualBox虚拟机中的存储格式。参数-f fmt是指定文件的格式，如果不指定格式，qemu-img会自动检测。filename是磁盘镜像文件的名称（包括路径）。

如下命令行演示了qemu-img的check命令的使用方法。

```
[root@jay-linux kvm_demo]# qemu-img check rhel6u3.qcow2
No errors were found on the image.
```

（2）create[-f fmt][-o options]filename[size]

创建一个格式为fmt，大小为size，文件名为filename的镜像文件。根据文件格式fmt的不同，还可以添加一个或多个选项（options）来附加对该文件的各种功能设置，可以使用"-o?"来查询某种格式文件支持哪些选项，在"-o"选项中各个选项用逗号来分隔。

如果在"-o"选项中使用了backing_file这个选项来指定其后端镜像文件，那么这个创建的镜像文件仅记录与后端镜像文件的差异部分。后端镜像文件不会被修改，除非在QEMU monitor中使用"commit"命令或使用"qemu-img commit"命令去手动提交这些改动。在这种情况下，size参数不是必须需的，其值默认为后端镜像文件的大小。另外，直接使用"-b backfile"参数也与"-o backing_file=backfile"效果相同。

size选项用于指定镜像文件的大小，其默认单位是字节（bytes），也可以支持k（即K）、M、G、T来分别表示KB、MB、GB、TB大小。另外，镜像文件的大小（size）也并非必须写在命令的最后，也可以写在"-o"选项中作为其中一个选项。

对create命令的演示如下所示，其中包括查询qcow2格式支持的选项、创建有backing_file的qcow2格式的镜像文件、创建没有backing_file的10GB大小的qcow2格式的镜像文件。

```
[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o ? temp.qcow
Supported options:
size             Virtual disk size
compat           Compatibility level (0.10 or 1.1)
backing_file     File name of a base image
backing_fmt      Image format of the base image
encryption       Encrypt the image
cluster_size     qcow2 cluster size
preallocation    Preallocation mode (allowed values: off, metadata)

[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -b
rhel6u3.img  rhel6u3.qcow2
Formatting 'rhel6u3.qcow2', fmt=qcow2 size=8589934592
backing_file='rhel6u3.img' encryption=off cluster_size=65536

[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o
backing_file=rhel6u3.img  rhel6u3-1.qcow2
Formatting 'rhel6u3-1.qcow2', fmt=qcow2 size=8589934592
backing_file='rhel6u3.img' encryption=off cluster_size=65536


[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o
backing_file=rhel6u3.img,size=20G  rhel6u3-2.qcow2
Formatting 'rhel6u3-2.qcow2', fmt=qcow2 size=21474836480
backing_file='rhel6u3.img' encryption=off cluster_size=65536

[root@jay-linux kvm_demo]# qemu-img create -f qcow2 ubuntu.qcow2 10G
Formatting 'ubuntu.qcow2', fmt=qcow2 size=10737418240
encryption=off cluster_size=65536
```

（3）commit[-f fmt]filename

提交filename文件中的更改到后端支持镜像文件（创建时通过backing_file指定的）中。

（4）convert[-c][-f fmt][-O output_fmt][-o options]flename[flename2[...]]output_flename

将fmt格式的filename镜像文件根据options选项转换为格式为output_fmt的名为output_filename的镜像文件。这个命令支持不同格式的镜像文件之间的转换，比如可以用VMware使用的vmdk格式文件转换为qcow2文件，这对从其他虚拟化方案转移到KVM上的用户非常有用。一般来说，输入文件格式fmt由qemu-img工具自动检测到，而输出文件格式output_fmt根据自己需要来指定，默认会被转换为raw文件格式（且默认使用稀疏文件的方式存储以节省存储空间）。

其中，"-c"参数表示对输出的镜像文件进行压缩，不过只有qcow2和qcow格式的镜像文件才支持压缩，并且这种压缩是只读的，如果压缩的扇区被重写，则会被重写为未压缩的数据。同样可以使用"-o options"来指定各种选项，如后端镜像、文件大小、是否加密，等等。使用backing_file选项来指定后端镜像，使生成的文件成为copy-on-write的增量文件，这时必须让在转换命令中指定的后端镜像与输入文件的后端镜像的内容相同，尽管它们各自后端镜像的目录和格式可能不同。

如果使用qcow2、qcow、cow等作为输出文件格式来转换raw格式的镜像文件（非稀疏文件格式），镜像转换还可以起到将镜像文件转化为更小的镜像，因为它可以将空的扇区删除使之在生成的输出文件中不存在。

下面的命令行演示了两个转换：将VMware的vmdk格式镜像转换为KVM可以使用的raw格式的镜像，将一个raw镜像文件转化为qcow2格式的镜像。

```
[root@jay-linux kvm_demo]# qemu-img convertmy-vmware.vmdk my-kvm.img
# 此处并无实际存在 vmdk 文件，仅演示其命令行操作
[root@jay-linux kvm_demo]# qemu-img convert -O qcow2 rhel6u3.img rhel6u3-a.gcow2
```

（5）info[-f fmt]filename

展示filename镜像文件的信息。如果文件使用的是稀疏文件的存储方式，也会显示出它本来分配的大小及实际已占用的磁盘空间大小。如果文件中存放有客户机快照，快照的信息也会被显示出来。下面的命令行演示了前面进行文件转换的输入、输出文件的信息。

```
[root@jay-linux kvm_demo]# qemu-img info rhel6u3.img
image: rhel6u3.img
file format: raw
virtual size: 8.0G (8589934592 bytes)
disk size: 8.0G
[root@jay-linux kvm_demo]# qemu-img info rhel6u3-a.img
image: rhel6u3-a.img
file format: qcow2
virtual size: 8.0G (8589934592 bytes)
disk size: 6.8G
cluster_size: 65536
```

（6）snapshot[-l|-a snapshot|-c snapshot|-d snapshot]filename

"-l"选项表示查询并列出镜像文件中的所有快照，"-a snapshot"表示让镜像文件使用某个快照，"-c snapshot"表示创建一个快照，"-d"表示删除一个快照。

（7）rebase[-f fmt][-t cache][-p][-u]-b backing_file[-F backing_fmt]filename

改变镜像文件的后端镜像文件，只有qcow2和qed格式支持rebase命令。使用"-b backing_file"中指定的文件作为后端镜像，后端镜像也被转化为"-F backing_fmt"中指定的后端镜像格式。

这个命令可以工作于两种模式之下，一种是安全模式（Safe Mode），这是默认的模式，qemu-img会根据比较原来的后端镜像与现在的后端镜像的不同进行合理的处理；另一种是非安全模式（Unsafe Mode），是通过"-u"参数来指定的，这种模式主要用于将后端镜像重命名或移动位置后对前端镜像文件的修复处理，由用户去保证后端镜像的一致性。

（8）resize filename[+|-]size

改变镜像文件的大小，使其不同于创建之时的大小。“+”和“-”分别表示增加和减少镜像文件的大小，size也支持K、M、G、T等单位的使用。缩小镜像的大小之前，需要在客户机中保证其中的文件系统有空余空间，否则会数据丢失，另外，qcow2格式文件不支持缩小

镜像的操作。在增加了镜像文件大小后，也需启动客户机在其中应用"fdisk"、"parted"等分区工具进行相应的操作才能真正让客户机使用到增加后的镜像空间。不过使用resize命令时需要小心（做好备份），如果失败，可能会导致镜像文件无法正常使用而造成数据丢失。

如下命令行演示了两个镜像的大小改变：将一个8GB的qcow2镜像增加2GB的空间，也将一个8GB大小的raw镜像减少1GB空间。

```
[root@jay-linux kvm_demo]# qemu-img resize rhel6u3-a.img +2G
Image resized.
[root@jay-linux kvm_demo]# qemu-img info rhel6u3-a.img
image: rhel6u3-a.img
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 6.8G
cluster_size: 65536

[root@jay-linux kvm_demo]# qemu-img resize rhel6u3-b.img -1G
Image resized.
[root@jay-linux kvm_demo]# qemu-img info rhel6u3-b.img
image: rhel6u3-b.img
file format: raw
virtual size: 7.0G (7516192768 bytes)
disk size: 6.5G
```

4.4.3 QEMU支持的镜像文件格式

qemu-img支持非常多种的文件格式，可以通过"qemu-img-h"查看其命令帮助得到，它支持二十多种格式：blkdebug、blkverify、

bochs、cloop、cow、tftp、ftps、ftp、https、http、dmg、nbd、parallels、qcow、qcow2、qed、host_cdrom、host_floppy、host_device、file、raw、sheepdog、vdi、vmdk、vpc、vvfat。

下面对其中的几种文件格式做简单的介绍。

（1）raw

原始的磁盘镜像格式，也是qemu-img命令默认的文件格式。这种格式的文件的优势在于它非常简单且非常容易移植到其他模拟器（emulator，QEMU也是一个emulator）上去使用。如果客户机文件系统（如Linux上的ext2/ext3/ext4、Windows的NTFS）支持“空洞”（hole），那么镜像文件只有在被写有数据的扇区才会真正占用磁盘空间，从而起到节省磁盘空间的作用，就如前面用"qemu-img info"命令查看镜像文件信息中看到的那样。qemu-img默认的raw格式的文件其实是稀疏文件[9]（sparse file），而3.5节“安装客户机”中使用"dd"命令创建的镜像也是raw格式，不过那是一开始就让镜像实际占用了分配的空间，而没有使用稀疏文件的方式对待空洞来节省磁盘空间。尽管一开始就实际占用磁盘空间的方式没有节省磁盘的效果，不过这种方式在写入新的数据时不需要宿主机从现有磁盘空间中分配，因此在第一次写入数据时，这种方式的性能会比稀疏文件的方式更好一点。

（2）host_device

在需要将镜像转化到不支持空洞的磁盘设备时需要用这种格式来代替raw格式。

（3）qcow2

qcow2是QEMU目前推荐的镜像格式，它是功能最多的格式。它支持稀疏文件（即支持空洞）以节省存储空间，它支持可选的AES加密以提高镜像文件安全性，支持基于zlib的压缩，支持在一个镜像文件中有多个虚拟机快照。

在qemu-img命令中qcow2支持如下几个选项：

❑backing_file，用于指定后端镜像文件。

❑backing_fmt，设置后端镜像的镜像格式。

❑cluster_size，设置镜像中簇的大小，取值在512B到2MB之间，默认值为64KB。较小的簇可以节省镜像文件的空间，而较大的簇可以带来更好的性能，需要根据实际情况来平衡，一般采用默认值即可。

❑preallocation，设置镜像文件空间的预分配模式，其值可为"off"、"metadata"之一。"off"模式是默认值，设置了不为镜像文件预分配磁盘空间。而"metadata"模式用于设置为镜像文件预分配

metadata的磁盘空间，所以这种方式生成的镜像文件稍大一点，不过在其真正分配空间写入数据时效率更高。另外，一些版本的qemu-img（如RHEL6.3自带的）还支持"full"模式的预分配，它表示在物理上预分配全部的磁盘空间，它将整个镜像的空间都填充零以占用空间，当然它所花费的时间较长，不过使用时性能更好。

❑encryption用于设置加密，当它等于"on"时，镜像被加密。它使用128位密钥的ASE加密算法，故其密码长度可达16个字符（每个字符8位），可以保证加密的安全性较高。在将"qemu-img convert"命令转化为qcow2格式时，加上"-o encryption"即可对镜像文件设置密码，而在使用镜像启动客户机时需要在QEMU monitor中输入"cont"或"c"（是continue的意思）命令来唤醒客户机输入密码后继续执行（否则客户机将不会真正启动），命令行演示如下：

```
[root@jay-linux kvm_demo]# qemu-img convert -o encryption -O
qcow2 rhel6u3.img rhel6u3.qcow
Disk image 'rhel6u3.qcow' is encrypted.
password: ****** # 此处输入你需要设置的密码
[root@jay-linux kvm_demo]# qemu-system-x86_64 -m 1024
rhel6u3.qcow -vnc :0 -monitor stdio
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) cont
ide0-hd0 (rhel6u3.qcow) is encrypted.
Password: ******
(qemu)
```

（4）qcow

较旧的QEMU镜像格式，现在已经很少使用了，一般用于兼容比较老版本的QEMU。它支持backing_file（后端镜像）和encryption（加密）两个选项。

（5）cow

用户模式Linux（User-Mode Linux）的Copy-On-Write的镜像文件格式。

（6）vdi

兼容Oracle（Sun）VirtualBox1.1的镜像文件格式（Virtual Disk Image）。

（7）vmdk

兼容VMware 4版本以上的镜像文件格式（Virtual Machine Disk Format）。

（8）vpc

兼容Microsoft的Virtual PC的镜像文件格式（Virtual Hard Disk format）。

（9）sheepdog

Sheepdog项目是由日本NTT实验室发起的，为QEMU/KVM做的一个开源的分布式存储系统，为KVM虚拟化提供块存储。它无单点故障（无类似于元数据服务器的中央节点），方便扩展（已经支持上千的节点数量），其配置简单、运维成本较低，总的来说，具有高可用性、易扩展性、易管理性等优势。

sheepdog项目的官方网站为：http://www.osrg.net/sheepdog/。

目前，国内的淘宝公司对该项目也有较大的贡献，详情可见淘宝sheepdog项目的网站：http://sheepdog.taobao.org/

4.4.4 客户机存储方式

前面介绍了存储的配置和qemu-img工具来管理镜像，在QEMU/KVM中，客户机镜像文件可以由很多种方式来构建，其中几种如下：

1）本地存储的客户机镜像文件。

2）物理磁盘或磁盘分区。

3）LVM（Logical Volume Management），逻辑分区。

4）NFS（Network File System），网络文件系统。

5）iSCSI（Internet Small Computer System Interface），基于Internet的小型计算机系统接口。

6）本地或光纤通道连接的LUN（Logical Unit Number）。

7）GFS2（Global File System 2）.

本地存储的客户机镜像文件是最常用的一种方式，它有预分配空间的raw文件、稀疏文件类型的raw文件、qcow2等多种格式。预分配空间的raw文件不随着镜像的使用而增长，而是在创建之初即完全占用磁盘空间，其消耗较多磁盘空间，不过运行效率较高。稀疏文件（包括raw和qcow2格式）在一开始时并不占用多的磁盘空间，而是随着实际写入数据才占用物理磁盘，比较灵活且节省磁盘空间，不过其在第一次写入数据时需要额外在宿主机中分配空间，因此其效率较低一些。而qcow2具有加密的安全性，所以在对磁盘IO性能要求并非很高时建议选择qcow2类型的镜像文件。

使用文件来做镜像有很多的优点，例如：

1）存储方便，在一个物理存储设备上可以存放多个镜像文件；

2）易用性，管理多个文件比管理多个磁盘、分区、逻辑分区等都要方便；

3）可移动性，可以非常方便地将镜像文件移动到另外一个本地或远程的物理存储系统中去；

4）可复制性，可以非常方便地将一个镜像文件复制或修改，从而供另一个新的客户机使用；

5）稀疏文件可以节省磁盘空间，仅占用实际写入过数据的空间；

6）网络远程访问，镜像文件可以方便地存储在通过网络连接的远程文件系统（如NFS）中。

不仅一个文件可以分配给客户机作为镜像文件系统，而且一个完整的磁盘或LVM分区也可以作为镜像分配给客户机使用。不过，磁盘分区、LVM分区由于没有磁盘的MBR引导记录，不能作为客户机的启动镜像，只能作为客户机附属的非启动块设备。一般来说，磁盘或LVM分区会有较好的性能，读写的延迟较低、吞吐量较高。不过为了防止客户机破坏物理磁盘分区，一般不将整个磁盘作为镜像由客户机使用。使用磁盘或LVM分区的劣势在于管理和移动性方面都不如镜像文件方便，而且不方便通过网络远程使用。

而NFS作为使用非常广泛的分布式文件系统，可以使客户端挂载远程NFS服务器中的共享目录，然后像使用本地文件系统一样使用NFS远程文件系统。如果NFS服务器端向客户端开放了读写的权限，那么可以直接挂载NFS，然后使用其中的镜像文件作为客户启动磁盘。如果没有

向客户端开放写权限，也可以在NFS客户端系统将远程NFS系统上的镜像文件作为后端镜像（backing file），以建立qcow2格式Copy-On-Write的本地镜像文件供客户机使用，这样做还有一个好处是保持NFS服务器上的镜像一致性、完整性，从而可以供给多个客户端同时使用。而且由于NFS的共享特性，因此NFS方式为客户机的动态迁移（第5章会介绍）提供了非常方便的共享存储系统。下面的命令行演示了NFS作为后端镜像的应用，在本地用qcow2格式镜像文件启动一个客户机。

在宿主机中，挂载NFS文件系统、建立qcow2镜像，然后启动客户机，如下所示：

```
[root@jay-linux kvm_demo]# mount vt-nfs:/images /images/
[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o
backing_file=/images/linux/ia32e_rhel6u3.img,size=20G rhel6u3.qcow2
Formatting 'rhel6u3.qcow2', fmt=qcow2 size=21474836480
backing_file='/images/linux/ia32e_rhel6u3.img' encryption=off
cluster_size=65536
[root@jay-linux kvm_demo]# qemu-system-x86_64 -smp 2 -m 1024 -net
nic -net tap -hda rhel6u3.qcow2 -vnc :0 -daemonize
[root@jay-linux kvm_demo]#
```

在客户机中，查看磁盘文件系统，如下所示：

```
[root@rhel6u3-ga ~]# fdisk -l

Disk /dev/sda: 21.5 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000726b0

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1         931     7471104   83  Linux
Partition 1 does not end on cylinder boundary.
/dev/sda2             931        1045      916480   82  Linux swap / Solaris
```

iSCSI是一套基于IP协议的网络存储标准，真正的物理存储放在初始端（initiator），而使用iSCSI磁盘的是目标端（target），它们之间实现了SCSI标准的命令，让目标端使用起来就和使用本地的SCSI硬盘一样，只是数据是在网络上进行读写操作的。光纤通道（Fibre Channel）也可以实现与iSCSI类似的存储区域网络（storage area network，SAN），不过它需要光纤作为特殊的网络媒介。而GFS2是由Redhat公司主导开发的主要给Linux计算机集群使用的共享磁盘文件系统，一般在Redhat的RHEL系列系统中有较多使用，它也可被用做QEMU/KVM的磁盘存储系统。

另外，如果需要获得更高性能的磁盘IO，可以使用半虚拟化的virtio作为磁盘驱动程序，第5章中将会详细介绍virtio的相关内容。

4.5 网络配置

在现代计算机系统中，网络功能是一个非常重要的功能，特别在这个互联网、云计算盛行的时代，一个计算机如果没有网络连接，它就与外界隔离起来，失去了一半的价值。在一个普通的现代企业中，不管是员工的办公电脑，还是存储客户数据、处理工作流的服务器，都存在于计算机网络之中，网络瘫痪将会给企业带来较大的经济损失。而一些大型互联网公司（如Google、Amazon、百度、淘宝、腾讯等）拥有成千上万台各种服务器，它们都是基于计算机网络（不管是内部网络还是外部网络）为客户提供服务。因此在使用KVM部署虚拟化解决方案时，关于网络的配置也是非常重要的环节。

4.5.1 QEMU支持的网络模式

网络是现代计算机系统不可或缺的一部分，QEMU也对虚拟机提供了丰富的网络支持。qemu-kvm主要向客户机提供了如下4种不同模式的网络。

1）基于网桥（bridge）的虚拟网卡。

2）基于NAT（Network Addresss Translation）的虚拟网络。

3）QEMU内置的用户模式网络（user mode networking）。

4）直接分配网络设备的网络（包括VT-d和SR-IOV）。

这里主要讲述前三种模式，第4种网络设备的直接分配将在第5章中详细讲述。除了特别的需要iptables配置端口映射、数据包转发规则的情况，一般默认将防火墙所有规则都关闭，以避免妨碍客户机中的网络畅通，在实际生产环境中，可根据实际系统的特点进行配置。

在QEMU命令行中，对客户机网络的配置（除了网络设备直接分配之外）都是用"-net"参数进行配置的，如果没有设置任何的"-net"参数，则默认使用"-net nic-net user"参数，进而使用完全基于QEMU内部实现的用户模式下的网络协议栈（将在4.5.4节详细介绍）。

qemu-kvm提供了对一系列主流和兼容性良好的网卡的模拟，通过"-net nic,model=?"参数可以查询到当前的qemu-kvm工具实现了哪些网卡的模拟。如下命令行显示了qemu-kvm-1.1.0中能模拟的网卡种类。

```
[root@jay-linux ~]# qemu-system-x86_64 -net nic,model=?
qemu: Supported NIC models:
ne2k_pci,i82551,i82557b,i82559er,rtl8139,e1000,pcnet,virtio
```

其中，"rtl8139"这个网卡模式是qemu-kvm默认的模拟网卡类型。RTL8139是Realtek半导体公司的一个10/100M网卡系列，是曾经非常流行（当然现在看来有点古老）且兼容性好的网卡。几乎所有的现代操作系统都对RTL8139网卡驱动提供支持。"e1000"是提供Intel e1000系

列的网卡模拟，纯的QEMU（非qemu-kvm）默认就是提供Intel e1000系列的虚拟网卡。而virtio类型是qemu-kvm对半虚拟化IO（virtio）驱动的支持（将会在第5章中详细介绍virtio的基本原理、配置和使用）。

qemu-kvm命令行在不加任何网络相关的参数启动客户机后，在客户机中可以看到它有一个默认的RTL8139系列的网卡（如下所示），当然由于没有进行更多的网络配置，这个模拟的网卡虽然在客户机中可见，但是它使用的是用户模式的网络，其功能非常有限（将在后面的4.5.4节中详述）。

```
[root@kvm-guest ~]# lspci | grep Eth
00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd.
RTL-8139/8139C/8139C+ (rev 20)
```

如下的命令行会模拟一个Intel e1000系列的网卡供客户机使用。

```
qemu-system-x86_64 rhel6u3.img -net nic,model=e1000
```

在客户机中看到的e1000系列网卡如下所示，默认是Intel 82540EM系列的网卡。

```
[root@kvm-guest ~]# lspci | grep Eth
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit
Ethernet Controller (rev 03)
```

qemu-kvm命令行中基本的"-net"参数的细节如下：

-net nic[,vlan=*n*][,macaddr=*mac*][,model=*type*][,name=*name*][,addr=*addr*][,vectors=*v*]

执行这个命令行会让QEMU建立一个新的网卡并将其连接到n号VLAN上。

其中：

❑"-net nic"是必需的参数，表明这是一个网卡的配置。

❑vlan=n，表示将网卡放入到编号为n的VLAN，默认为0。

❑macaddr=mac，设置网卡的MAC地址，默认会根据宿主机中网卡的地址来分配。若局域网中客户机太多，建议自己设置MAC地址，以防止MAC地址冲突。

❑model=type，设置模拟的网卡的类型，在qemu-kvm中默认为rtl8139。

❑name=name，为网卡设置一个易读的名称，该名称仅在QEMU monitor中可能用到。

❑addr=addr，设置网卡在客户机中的PCI设备地址为addr。

❑vectors=v，设置该网卡设备的MSI-X向量的数量为n，该选项仅对使用virtio驱动的网卡有效。设置为"vectors=0"是关闭virtio网卡的MSI-X中断方式。

如果需要向一个客户机提供多个网卡，可以多次使用"-net"参数。

在宿主机中用如下的命令行启动一个客户机，并使用上面的一些网络参数。

```
[root@jay-linuxkvm-demo]# qemu-system-x86_64 -m 1024
rhel6u3.img-net
nic,vlan=0,macaddr=52:54:00:12:34:22,model=e1000,addr=08 -net user
```

在客户机中用一些工具查看网卡相关的信息如下（这里使用了用户模式的网络栈，其详细介绍可参考4.5.4节），由此可知上面的网络设置都已生效。

```
[root@kvm-guest ~]# lspci | grep Eth
00:08.0 Ethernet controller: Intel Corporation 82540EM Gigabit
Ethernet Controller (rev 03)
[root@kvm-guest ~]# ethtool -i eth1
driver: e1000
version: 7.3.21-k8-NAPI
firmware-version:
bus-info: 0000:00:08.0
[root@kvm-guest ~]# ifconfig
eth1      Link encap:Ethernet  HWaddr 52:54:00:12:34:22
          inet addr:10.0.2.15  Bcast:10.0.2.255
```

```
Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3422/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:47 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1890 (1.8 KiB)  TX bytes:6380 (6.2 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:720 (720.0 b)  TX bytes:720 (720.0 b)
```

在QEMU monitor中查看网络的信息，如下：

```
(qemu) info network
VLAN 0 devices:
  user.0: type=user,net=10.0.2.0,restrict=off
  e1000.0: type=nic,model=e1000,macaddr=52:54:00:12:34:22
Devices not on any VLAN:
```

本节只介绍了网络设置的基本参数，没有详细配置具体的网络工作模式，所以这里得到的虚拟网卡在客户机中可能并不能连接到外部网络，接下来的3个小节将详细介绍各个网络工作模式的原理和配置方法。

4.5.2 使用网桥模式

在QEMU/KVM的网络使用中，网桥（bridge）模式可以让客户机和宿主机共享一个物理网络设备连接网络，客户机有自己的独立IP地址，可以直接连接与宿主机一模一样的网络，客户机可以访问外部网络，外部网络也可以直接访问客户机（就像访问普通物理主机一样）。即使宿主机只有一个网卡设备，使用bridge模式也可知让多个客户机与宿主机共享网络设备，bridge模式使用非常方便，应用也非常广泛。

在qemu-kvm的命令行中，关于bridge模式的网络参数如下：

```
-nettap[,vlan=n][,name=str][,fd=h][,ifname=name][,script=file]
[,downscript=dfile][,helper=helper][,sndbuf=nbytes][,vnet_hdr=
on|off][,vhost=on|off][,vhostfd=h][,vhostforce=on|off]
```

该配置表示连接宿主机的TAP网络接口到n号VLAN中，并且使用file和dfile两个脚本在启动客户机时配置网络和在关闭客户机时取消网络配置。

❑tap参数，表明使用TAP设备。TAP是虚拟网络设备，它仿真了一个数据链路层设备（ISO七层网络结构的第二层），它像以太网的数据帧一样处理第二层数据报。而TUN与TAP类似，也是一种虚拟网络设备，它是对网络层设备的仿真。TAP用于创建一个网络桥，而TUN与路由相关。

❑vlan=n，设置该设备VLAN编号，默认值为0。

❑name=name，设置名称，在QEMU monior中可能用到，一般由系统自动分配即可。

❑fd=h，连接到现在已经打开着的TAP接口的文件描述符，一般不要设置该选项，而是让QEMU自动创建一个TAP接口。在使用了fd=h的选项后，ifname、script、downscript、helper、vnet_hdr等选项都不可使用了（不能与fd选项同时出现在命令行中）。

❑ifname=name，设置在宿主机中添加的TAP虚拟设备的名称（如tap1、tap5等），当不设置这个参数时，QEMU会根据系统中目前的情况，产生一个TAP接口的名称。

❑script=file，设置宿主机在启动客户机时自动执行的网络配置脚本。如果不指定，其默认值为"/etc/qemu-ifup"这个脚本，可指定自己的脚本路径以取代默认值；如果不需要执行脚本，则设置为"script=no"。

❑downscript=dfile，设置宿主机在客户机关闭时自动执行的网络配置脚本。如果不设置，其默认值为"/etc/qemu-ifdown"；若客户机关闭时宿主机不需要执行脚本，则设置为"downscript=no"。

❑helper=helper，设置启动客户机时在宿主机中运行的辅助程序，包括建立一个TAP虚拟设备，默认值为/usr/local/libexec/qemu-bridge-helper。一般不用自定义，采用默认值即可。

❑sndbuf=nbytes，限制TAP设备的发送缓冲区大小为n字节，当需要流量进行流量控制时可以设置该选项。其默认值为"sndbuf=0"，即不限制发送缓冲区的大小。

其余几个选项都是与virtio相关的，这里不做过多介绍。

上面介绍了使用TAP设备的一些选项，接下来通过在宿主机中执行如下步骤来实现网桥方式的网络配置。

1）要采用bridge模式的网络配置，首先需要安装两个RPM包，即bridge-utils和tunctl，它们提供所需的brctl和tunctl命令行工具。可以用yum工具安装这两个RPM包，如下：

```
[root@jay-linux ~]# yum install bridge-utils tunctl
```

2）查看tun模块是否加载，如下：

```
[root@jay-linux ~]# lsmod | grep tun
tun                    12197  2
```

如果tun模块没有加载，则运行"modprobe tun"命令来加载。当然，如果已经将tun编译到内核（可查看内核config文件中是否有"CONFIG_TUN=y"选项），则不需要加载了。如果内核完全没有配置TUN模块，则需要重新编译内核才行。

3）检查/dev/net/tun的权限，需要让当前用户拥有可读写的权限。

```
[root@jay-linux ~]# ll /dev/net/tun
crw-rw-rw- 1 root root 10, 200 Jul 20 16:23 /dev/net/tun
```

4）建立一个bridge，并将其绑定到一个可以正常工作的网络接口上，同时让bridge成为连接本机与外部网络的接口。主要的配置命令如下：

```
[root@jay-linux ~]# brctl addbr br0                    # 添加 br0 这个 bridge
[root@jay-linux ~]# brctl addif br0 eth0               # 将 br0 与 eth0 绑定起来
[root@jay-linux ~]# brctl stp br0 on                   # 将 br0 设置为启用 STP 协议
[root@jay-linux ~]# ifconfig eth0 0                    # 将 eth0 的 IP 设置为 0
[root@jay-linux ~]#dhclient br0                        # 将 br0 的网络配置好
[root@jay-linux ~]#route                               # 参看路由表是否正常配置
Kernel IP routing table
DestinationGateway                Genmask     Flags Metric Ref    Use
Iface
192.168.0.0  *                255.255.0.0     U     0      0        0
br0
default    sqa-gate.tsp.or        0.0.0.0     UG    0      0        0
br0
[root@jay-linux ~]#ping 192.168.199.99 -c 1            # 用 ping 测试网络通畅
PING 192.168.199.99 (192.168.199.99) 56(84) bytes of data.
64 bytes from 192.168.199.99: icmp_seq=1 ttl=64 time=4.16 ms
--- 192.168.199.99 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 4ms
rtt min/avg/max/mdev = 4.164/4.164/4.164/0.000 ms
[root@jay-linux ~]#dmesg
<! -- .....  -->
device eth0 entered promiscuous mode
br0: port 1(eth0) entered forwarding state
```

建立bridge后的状态是让网络接口eth0进入混杂模式（promiscuous mode，接收网络中所有数据包），网桥br0进入转发状

态（forwarding state），并且与eth0有相同的MAC地址，一般也会得到和eth0相同的IP。"brctl stp br0 on"是打开br0的STP协议，STP是生成树协议（Spanning Tree Protocol），在这里使用STP主要是为了避免在建有bridge的以太网LAN中出现环路。如果不打开STP，则可能出现数据链路层的环路，从而导致建有bridge的主机网络不畅通。

这里默认通过DHCP方式动态获得IP。在绑定了bridge之后，也可以使用"ifconfig"和"route"等命令设置br0的IP、网关、默认路由等，需要将bridge设置为与其绑定的物理网络接口一样的IP和MAC地址，并且让默认路由使用bridge（而不是ethX）来连通。这个步骤可能导致宿主机的网络断掉，之后重新通过bridge建立网络连接，因此建立bridge这一步骤最好不要通过SSH连接远程配置。另外，在RHEL系列系统中最好将NetworkManager这个程序结束掉，因为它并不能管理bridge的网络配置，相反，它在后台运行则可能对网络设置有些干扰。

5）准备qemu-ifup和qemu-ifdown脚本。

在客户机启动网络前会执行的脚本是由"script"选项配置的（默认为/etc/qemuif-up）。一般在该脚本中创建一个TAP设备并将其与bridge绑定起来。如下是qemu-ifup脚本的示例，其中“$1”是qemu-kvm命令工具传递给脚本的参数，它是客户机使用的TAP设备名称（如tap0、tap1等，或者是前面提及的ifname选项的值）。另外，其中

的"tunctl"命令这一行是不需要的，因为qemu-bridge-helper程序已经会创建好TAP设备，这里列出来只是因为在一些版本较旧的qemu-kvm中可能没有自动创建TAP设备。

```
#!/bin/bash
#This is a qemu-ifup script for bridging.
#You can use it when starting a KVM guest with bridge mode network.

#set your bridge name
switch=br0

if [ -n "$1" ]; then
        #create a TAP interface; qemu will handle it automatically.
        #tunctl -u $(whoami) -t $1

        #start up the TAP interface
        ip link set $1 up
        sleep 1
        #add TAP interface to the bridge
        brctl addif ${switch} $1
        exit 0
else
        echo "Error: no interface specified"
        exit 1
fi
```

由于qemu-kvm工具在客户机关闭时会解除TAP设备的bridge绑定，也会自动删除已不再使用的TAP设备，所以qemu-ifdown这个脚本不是必需的，最好设置为"downscript=no"。如下列出一个qemu-ifdown脚本的示例，是为了说明清理bridge模式网络环境的步骤，在qemu-kvm没有自动处理时可以使用。

```
#!/bin/bash
#This is a qemu-ifdown script for bridging.
#You can use it when starting a KVM guest with bridge mode network.
#Don't use this script in most cases; QEMU will handle it automatically.

#set your bridge name
switch=br0

if [ -n "$1" ]; then
        # Delete the specified interfacename
        tunctl -d $1
        #release TAP interface from bridge
        brctl delif ${switch} $1
        #shutdown the TAP interface
        ip link set $1 down
        exit 0
else

        echo "Error: no interface specified"
        exit 1
fi
```

6）用qemu-kvm命令启动bridge模式的网络。

在宿主机中，用命令行启动客户机并检查bridge的状态，如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic -net
tap,ifname=tap1,script=/etc/qemu-ifup,
downscript=no -vnc :0 -daemonize
[root@jay-linux kvm_demo]# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.60eb692129b7       no              eth0
                                                        tap1
[root@jay-linux kvm_demo]# ls /sys/devices/virtual/net/
lo  br0  tap1
```

由上面信息可知，在创建客户机后，添加了一个名为tap1的TAP虚拟网络设备，将其绑定在br0这个bridge上。查看到的3个虚拟网络设

备依次为：网络回路设备lo（就是一般IP为127.0.0.1的设备）、前面建立好的bridge设备br0、为客户机提供网络的TAP设备tap1。

在客户机中，如下的几个命令用于检查网络是否配置好。

```
[root@kvm-guest ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:192.168.63.144  Bcast:192.168.255.255
Mask:255.255.0.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2893 errors:0 dropped:0 overruns:0 frame:0
          TX packets:102 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:465871 (454.9 KiB)  TX bytes:18099 (17.6 KiB)
          Interrupt:11
[root@kvm-guest ~]# ping vt-snb9 -c 1
PING vt-snb9.tsp.org (192.168.199.99) 56(84) bytes of data.
64 bytes from 192.168.199.99: icmp_seq=1 ttl=64 time=3.51 ms

--- vt-snb9.tsp.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 4ms
rtt min/avg/max/mdev = 3.515/3.515/3.515/0.000 ms
[root@kvm-guest ~]# route
Kernel IP routing table
Destination  Gateway          Genmask      Flags Metric Ref    Use
Iface
192.168.0.0  *             255.255.0.0     U     0      0        0
eth0
default    sqa-gate.tsp.org 0.0.0.0        UG    0      0        0
eth0
```

然而在将客户机关机后，在宿主机中再次查看bridge状态和虚拟网络设备的状态，如下：

```
[root@jay-linux kvm_demo]# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.60eb692129b7       no              eth0
[root@jay-linux kvm_demo]# ls /sys/devices/virtual/net/
lo  br0
```

由上面的输出信息可知，qemu-kvm工具已经将tap1设备删除了。

4.5.3 使用NAT模式

NAT（Network Addresss Translation，网络地址转换），属于广域网接入技术的一种，它将内网地址转化为外网的合法IP地址，它被广泛应用于各种类型的Internet接入方式和各种类型的网络之中。NAT将来自内网IP数据包的包头中的源IP地址转换为一个外网的IP地址。众所周知，IPv4的地址资源已几近枯竭，而NAT使内网的多个主机可以共用一个IP地址接入网络，这样有助于节约IP地址资源，这也是NAT最主要的作用。另外，通过NAT访问外部网络的内部主机，其内部IP对外是不可见的，这就隐藏了NAT内部网络拓扑结构和IP信息，也就能够避免内部主机受到外部网络的攻击。客观事物总是有正反两面性的，没有任何技术是十全十美的。NAT技术隐藏了内部主机细节从而提高了安全性，但是如果NAT内的主机作为Web或数据库服务器需接受来自外部网络的主动连接，这时NAT就表现出了局限性，不过，可以在拥有外网IP的主机上使用iptables等工具实现端口映射，从而让外网对这个外网IP的一个端口的访问被重新映射到NAT内网的某个主机的相应端口上去。

在QEMU/KVM中，默认使用IP伪装的方式去实现NAT，而不是使用SNAT（Source-NAT）或DNAT（Destination-NAT）的方式。图4-6展示

了KVM中的NAT模式网络的结构图，宿主机在外网的IP是10.10.10.190，其上运行的各个客户机的IP属于内网的网络段192.168.122.0/24。

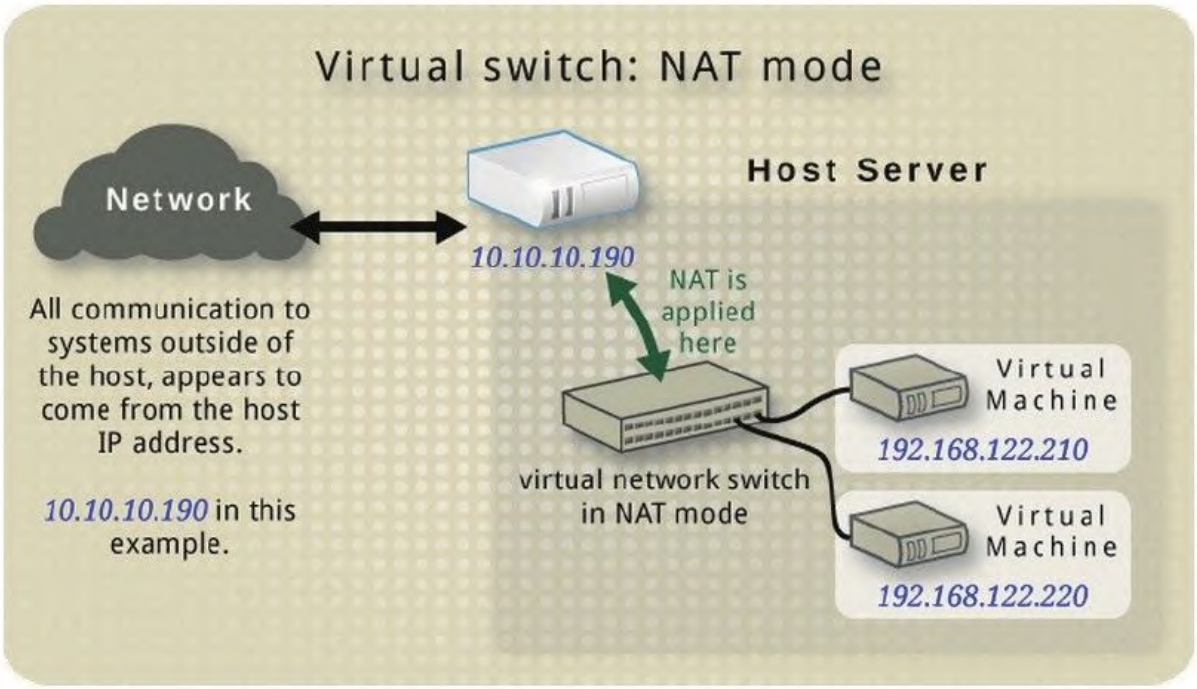


图　4-6　KVM中的NAT模式网络

在KVM中配置客户机的NAT网络方式，需要在宿主机中运行一个DHCP服务器给宿主机分配NAT内网的IP地址，可以使用dnsmasq工具来实现。在KVM中，DHCP服务器为客户机提供服务的基本架构如图4-7所示。

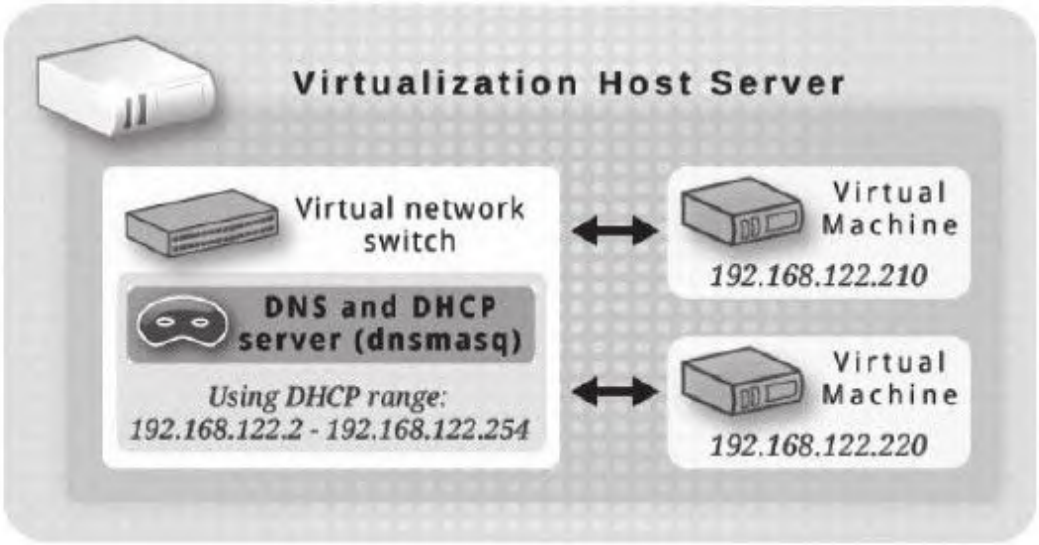


图　4-7　宿主机中的dnsmasq为客户机提供DHCP服务

通过下面几步可以使客户机启动并以NAT方式配置好它的网络。

1)检查配置宿主机内核编译的配置，将网络配置选项中与NAT相关的选项配置好，否则在启动客户机使用NAT网络配置时可能会遇到如下错误提示，因为无法按需加载"iptable_nat"和"nf_nat"等模块。

```
iptables v1.4.7: can't initialize iptables table `nat': Table does
not exist (do you need to insmod?)
```

遇到这样的情况，只能重新配置和编译内核了。下面截取的一小段内核配置，是一般情况下NAT的部分相关配置。

```
#
# IP: Netfilter Configuration
#
CONFIG_NF_DEFRAG_IPV4=m
CONFIG_NF_CONNTRACK_IPV4=m
CONFIG_NF_CONNTRACK_PROC_COMPAT=y
CONFIG_IP_NF_QUEUE=m
CONFIG_IP_NF_IPTABLES=m
CONFIG_IP_NF_MATCH_AH=m
CONFIG_IP_NF_MATCH_ECN=m
CONFIG_IP_NF_MATCH_RPFILTER=m
CONFIG_IP_NF_MATCH_TTL=m
CONFIG_IP_NF_FILTER=m
CONFIG_IP_NF_TARGET_REJECT=m
CONFIG_IP_NF_TARGET_ULOG=m
CONFIG_NF_NAT=m
CONFIG_NF_NAT_NEEDED=y
CONFIG_IP_NF_TARGET_MASQUERADE=m
CONFIG_IP_NF_TARGET_NETMAP=m
CONFIG_IP_NF_TARGET_REDIRECT=m
```

2)安装必要的软件包：bridge-utils、iptables和dnsmasq等。其中bridge-utils包含管理bridge的工具brctl（在4.5.2节中已使用过），iptables是对内核网络协议栈中IPv4包的过滤工具和NAT管理工具，dnsmasq是一个轻量级的DHCP和DNS服务器软件。当然，如有其他

满足类似功能的软件包，也可以选用。在宿主机中，查看所需软件包情况，如下：

```
[root@jay-linux kvm_demo]# rpm -qa | grep bridge
bridge-utils-1.2-9.el6.x86_64
[root@jay-linux kvm_demo]# rpm -qa | grep iptables
iptables-ipv6-1.4.7-4.el6.x86_64
iptables-1.4.7-4.el6.x86_64
[root@jay-linux kvm_demo]# rpm -qa | grep dnsmasq
dnsmasq-2.48-5.el6.x86_64
```

3)准备一个为客户机建立NAT用的qemu-ifup脚本及关闭网络用的qemu-ifdown脚本。这两个脚本中的$1（传递给它们的第一个参数）就是在客户机中使用的网络接口在宿主机中的虚拟网络名称（如tap0、tap1等）。

其中，在启动客户机时建立网络的脚本示例（/etc/qemu-ifup-NAT）如下，主要功能是：建立bridge，设置bridge的内网IP（此处为192.168.122.1），并且将客户机的网络接口与其绑定，然后打开系统中网络IP包转发的功能，设置iptables的NAT规则，最后启动dnsmasq作为一个简单的DHCP服务器。

```
#!/bin/bash
# qemu-ifup script for QEMU/KVM with NAT netowrk mode

# set your bridge name
BRIDGE=virbr0

# Network information
NETWORK=192.168.122.0
NETMASK=255.255.255.0
# GATEWAY for internal guests is the bridge in host
GATEWAY=192.168.122.1
DHCPRANGE=192.168.122.2,192.168.122.254

# Optionally parameters to enable PXE support
TFTPROOT=
BOOTP=

function check_bridge()
{
       if brctl show | grep "^$BRIDGE" &> /dev/null; then
              return 1
       else
              return 0
       fi
}

function create_bridge()
{
          brctl addbr "$BRIDGE"
```

```
          brctl stp "$BRIDGE" on
          brctl setfd "$BRIDGE" 0
          ifconfig "$BRIDGE" "$GATEWAY" netmask "$NETMASK" up
}

function enable_ip_forward()
{
      echo 1 > /proc/sys/net/ipv4/ip_forward
}

function add_filter_rules()
{
       iptables -t nat -A POSTROUTING -s "$NETWORK"/"$NETMASK" \
              ! -d "$NETWORK"/"$NETMASK" -j MASQUERADE
}

function start_dnsmasq()
{
       # don't run dnsmasq repeatedly
       ps -ef | grep "dnsmasq" | grep -v "grep" &> /dev/null
       if [ $? -eq 0 ]; then
               echo "Warning:dnsmasq is already running."
               return 1
       fi

       dnsmasq \
              --strict-order \
              --except-interface=lo \
              --interface=$BRIDGE \
              --listen-address=$GATEWAY \
              --bind-interfaces \
              --dhcp-range=$DHCPRANGE \
              --conf-file="" \
              --pid-file=/var/run/qemu-dhcp-$BRIDGE.pid \
              --dhcp-leasefile=/var/run/qemu-dhcp-$BRIDGE.leases \
              --dhcp-no-override \
              ${TFTPROOT:+"--enable-tftp"} \
              ${TFTPROOT:+"--tftp-root=$TFTPROOT"} \
              ${BOOTP:+"--dhcp-boot=$BOOTP"}
}

function setup_bridge_nat()
{
        check_bridge "$BRIDGE"
        if [ $? -eq 0 ]; then
                create_bridge
        fi
        enable_ip_forward
        add_filter_rules "$BRIDGE"
        start_dnsmasq "$BRIDGE"
```

```
}

# need to check $1 arg before setup
if [ -n "$1" ]; then
        setup_bridge_nat
        ifconfig "$1" 0.0.0.0 up
        brctl addif "$BRIDGE" "$1"
        exit 0
else
        echo "Error: no interface specified."
        exit 1
fi
```

关闭客户机时调用的网络脚本示例（/etc/qemu-ifdown-NAT）如下，它主要完成解除bridge绑定、删除bridge和清空iptables的NAT规则。

```
#!/bin/bash
# qemu-ifdown script for QEMU/KVM with NAT network mode

# set your bridge name
BRIDGE="virbr0"

if [ -n "$1" ]; then
      echo "Tearing down network bridge for $1"
      ip link set $1 down
      brctl delif "$BRIDGE" $1
      ip link set "$BRIDGE" down
      brctl delbr "$BRIDGE"
      iptables-t nat -F
      exit 0
else
      echo "Error: no interface specified"
      exit 1
fi
```

当然，对于这两个脚本中实现的功能，可以根据实际情况进行修改，另外，手动来完成这样的功能而不依赖于这两个脚本一样是可行的。

4)当启动客户机时，使用上面提到的启动脚本。创建客户机的qemu-kvm命令行如下：

```
qemu-system-x86_64 rhel6u3.img -m 1024 -smp 2 -net nic -net \
tap,script=/etc/qemu-ifup-NAT,downscript=/etc/qemu-ifdown-NAT
```

在启动客户机后，检查脚本中描述的宿主机中的各种配置生效的情况，如下：

```
[root@jay-linux kvm_demo]# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.60eb692129b7       yes              eth0
virbr0          8000.4a826f3a76d5       yes             tap0
```

#注意区别这两个bridge的不同：br0是前面提到的网桥模式使用的，它与一个物理上的网络接口eth0绑定，而virbr0是这里介绍的NAT方式的bridge，它没有绑定任何物理网络接口，只是绑定了tap0这个客户机使用的虚拟网络接口。

```
[root@jay-linux kvm_demo]# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.122.0/24    !192.168.122.0/24

[root@jay-linux ~]# ifconfig  virbr0
virbr0    Link encap:Ethernet  HWaddr 4A:82:6F:3A:76:D5
        inet addr:192.168.122.1  Bcast:192.168.122.255
Mask:255.255.255.0
          inet6 addr: fe80::4882:6fff:fe3a:76d5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:59 errors:0 dropped:0 overruns:0 frame:0
          TX packets:49 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:7537 (7.3 KiB)  TX bytes:6471 (6.3 KiB)
[root@jay-linux ~]# ps -ef | grep dnsmasq | grep -v grep
nobody   22507     1  0 Jul27 ?        00:00:08 dnsmasq
--strict-order --except-interface=lo --interface=virbr0
--listen-address=192.168.122.1 --bind-interfaces
--dhcp-range=192.168.122.2,192.168.122.254 --conf-file=
--pid-file=/var/run/qemu-dhcp-virbr0.pid
--dhcp-leasefile=/var/run/qemu-dhcp-virbr0.leases
--dhcp-no-override
```

5)在客户机中，通过DHCP动态获得IP，并且检查网络是否畅通，如下：

```
[root@kvm-guest ~]# dhclient eth0
eth0: link up, 100Mbps, full-duplex, lpa 0x05E1
[root@kvm-guest ~]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:192.168.122.140  Bcast:192.168.122.255
Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:112 errors:0 dropped:0 overruns:0 frame:0
          TX packets:108 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12073 (11.7 KiB)  TX bytes:18654 (18.2 KiB)
          Interrupt:11 Base address:0xa000
[root@kvm-guest ~]# route
Kernel IP routing table
Destination     Gateway     Genmask       Flags Metric Ref    Use Iface


192.168.122.0   *           255.255.255.0   U     0      0       0 eth0
default     192.168.122.1   0.0.0.0         UG    0      0       0 eth0
[root@kvm-guest ~]# ping 192.168.122.1 -c 1
PING 192.168.122.1 (192.168.122.1) 56(84) bytes of data.
64 bytes from 192.168.122.1: icmp_seq=1 ttl=64 time=0.373 ms

--- 192.168.122.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.373/0.373/0.373/0.000 ms
[root@kvm-guest ~]# ping 192.168.199.103 -c 1
PING 192.168.199.103 (192.168.199.103) 56(84) bytes of data.
64 bytes from 192.168.199.103: icmp_seq=1 ttl=63 time=0.947 ms

--- 192.168.199.103 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 1ms
rtt min/avg/max/mdev = 0.947/0.947/0.947/0.000 ms
```

从上面的命令行输出可知，客户机可以通过DHCP获得网络IP（192.168.122.0/24子网中），其默认网关是宿主机的bridge的IP（192.168.122.1），并且可以ping通网关（192.168.122.1）和子网外的另外一个主机（192.168.199.103），说明其与外部网络的连接正常。

另外，客户机中的DNS服务器默认配置为宿主机（192.168.122.1），如果宿主机没有启动DNS服务，则可能导致在客户机中无法解析域名。这时需要将客户机中/etc/resolv.conf修改为与宿主机中一致可用的DNS配置，然后就可以正常解析外部的域名（主机名）了，如下：

```
[root@kvm-guest ~]# vi /etc/resolv.conf
[root@kvm-guest ~]# cat /etc/resolv.conf
; generated by /sbin/dhclient-script
search tsp.org
nameserver 192.168.199.3
[root@kvm-guest ~]# nslookup vt-snb9
Server:         192.168.199.3
Address:        192.168.199.3#53

Name:   vt-snb9.tsp.org
Address: 192.168.199.99
[root@kvm-guest ~]# ping vt-snb9 -c 1
PING vt-snb9.tsp.org (192.168.199.99) 56(84) bytes of data.
64 bytes from 192.168.199.99: icmp_seq=1 ttl=63 time=0.741 ms

--- vt-snb9.tsp.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 0.741/0.741/0.741/0.000 ms
```

6)添加iptables规则进行端口映射，让外网主机也能访问客户机。

到步骤5）为止，客户机已可以正常连通外部网络，但是外部网络（除宿主机外）无法直接连接到客户机。其中一个解决方案是，在宿主机中设置iptables的规则进行端口映射，使外部主机对宿主机IP的一个端口的请求转发到客户机中的某一个端口。

在宿主机中，查看网络配置情况，然后iptables设置端口映射将如下，将宿主机的80端口（常用于HTTP服务）映射到客户机的80端口。

```
[root@jay-linux kvm_demo]# route
Kernel IP routing table
Destination     Gateway     Genmask     Flags Metric Ref    Use Iface
default     sqa-gate.tsp.or 0.0.0.0      UG    0      0        0 br0
192.168.0.0     *          255.255.0.0     U     0      0        0 br0
192.168.122.0   *         255.255.255.0   U     0      0        0 virbr0
[root@jay-linux kvm_demo]# ifconfig br0
br0       Link encap:Ethernet  HWaddr 60:EB:69:21:29:B7
          inet addr:192.168.82.0  Bcast:192.168.255.255
Mask:255.255.0.0
          inet6 addr: fe80::ccc7:84ff:fe41:280c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:608315 errors:0 dropped:0 overruns:0 frame:0
          TX packets:49539 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:72662298 (69.2 MiB)  TX bytes:79644268 (75.9 MiB)
[root@jay-linux kvm_demo]# iptables -t nat -A PREROUTING -p tcp
-d \ 192.168.82.0 --dport 80 -j DNAT --to 192.168.122.140:80
[root@jay-linux kvm_demo]# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       tcp  --  anywhere             192.168.82.0        tcp
dpt:http to:192.168.122.140:80

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.122.0/24    !192.168.122.0/24
```

在客户机中，编辑一个在HTTP服务中被访问的示例文件（/var/www/html/index.html，Apache默认根目录

为/var/www/html），然后启动Apache服务。

```
[root@kvm-guest ~]# cat /var/www/html/index.html
<b>This an index page demo running on apache in kvm-guest.</b>
[root@kvm-guest ~]# service httpd start
Starting httpd: httpd: apr_sockaddr_info_get() failed for kvm-guest
httpd: Could not reliably determine the server's fully qualified
domain name, using 127.0.0.1 for ServerName
                                                          [  OK  ]
```

在外部网络某主机上测试连接宿主机（192.168.82.0）的80端口，就会被映射到客户机（192.168.122.140）中的80端口，如图4-8所示，外部网络已经可以正常访问在NAT内网中的那台客户机80端口上的HTTP服务了。

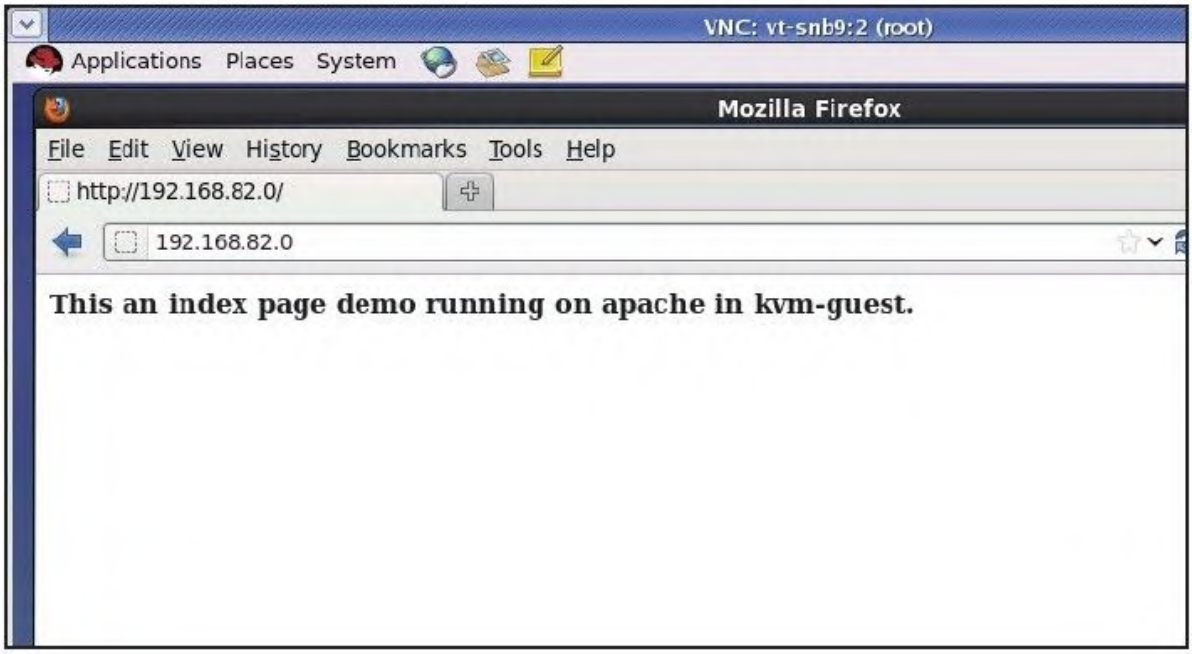


图 4-8 外部网络主机访问宿主机而被映射到客户机中相应的端口

在上面的示例中，NAT的配置涉及的一些iptables配置规则仅用于实验演示，在实际生产环境中需要根据实际情况进行更细粒度的配置，如果将访问规则和数据包转发规则设置得过于宽松可能会带来网络安全方面的隐患。

4.5.4 QEMU内部的用户模式网络

前面4.5.1节中提到，在没有任何"-net"参数时，qemu-kvm默认使用的是"-net nic-net user"的参数，提供了一种用户模式（user-mode）的网络模拟。使用用户模式的网络的客户机可以连通宿主机及外部的网络。用户模式网络是完全由QEMU自身实现的，不依赖于其他的工具（如前面提到的bridge-utils、dnsmasq、iptables等），而且不需要root用户权限（前面介绍过的bridge模式和NAT模式在配置宿主机网络和设置iptables规则时一般都需要root用户权限）。QEMU使用Slirp[10] 实现了一整套TCP/IP协议栈，并且使用这个协议栈实现了一套虚拟的NAT网络。

由于其使用简单、独立性好、不需root权限、客户机网络隔离性好等优势，用户模式网络是qemu-kvm的默认网络配置。不过，用户模式网路也有如下3个缺点：

1）由于其在QEMU内部实现所有网络协议栈，因此其性能较差。

2）不支持部分网络功能（如ICMP），所以不能在客户机中使用ping命令测试外网连通性。

3）不能从宿主机或外部网络直接访问客户机。

使用用户模式的网络，其qemu-kvm命令行参数为：

```
-net user[,option][,option][,...]
```

其中常见的选项（option）及其意义如下：

❑vlan=n，将用户模式网络栈连接到编号为n的VLAN中（默认值为0）。

❑name=name，分配一个在QEMU monitor中会用到的名字（如在monitor的"info network"命令中可看到这个网卡的name）。

❑net=addr[/mask]，设置客户机可以看到的IP地址（客户机所在子网），其默认值是10.0.2.0/24。其中，子网掩码（mask）有两种形式可选，一种是类似于255.255.255.0这样的地址，另一种是32位IP地址中前面被置位为1的位数（如10.0.2.0/24）。

❑host=addr，指定客户机可见宿主机的地址，默认值为客户机所在网络的第2个IP地址（如10.0.2.2）。

❑restrict=y|yes|n|no，如果将此选项打开（为y或yes），则客户机将会被隔离，客户机不能与宿主机通信，其IP数据包也不能通过宿主机而路由到外部网络中。这个选项不会影响"hostfwd"显示地指定的转发规则，"hostfwd"选项始终会生效。默认值为n或no，不会隔离客户机。

❑hostname=name，设置在宿主机DHCP服务器中保存的客户机主机名。

❑dhcpstart=addr，设置能够分配给客户机的第一个IP，在QEMU内嵌的DHCP服务器有16个IP地址可供分配。在客户机中IP地址范围的默认值是子网中的第15到第30个IP地址（如10.0.2.15～10.0.2.30）。

❑dns=addr，指定虚拟DNS的地址，这个地址必须与宿主机地址（在"host=addr"中指定的）不相同，其默认值是网络中的第3个IP地址（如10.0.2.3）。

❑tftp=dir，激活QEMU内嵌的TFTP服务器，目录dir是TFTP服务的根目录。不过，在客户机使用TFTP客户端连接TFTP服务后需要使用binary模式来操作。

❑hostfwd=[tcp|udp]:[hostaddr]:hostport-[guestaddr]:guestport，将访问宿主机的hostpot端口的TCP/UDP连接重定向到客户机（IP为guestaddr）的guestport端口上。如果没有设

置guestaddr，那么默认使用x.x.x.15（DHCP服务器可分配的第一个IP地址）。如果指定了hostaddr的值，则可以根据宿主机上的一个特定网络接口的IP端口来重定向。如果没有设置连接类型为TCP或UDP，则默认使用TCP连接。"hostfw=…"这个选项在一个命令行中可以多次重复使用。

❑guestfwd=[tcp]:server:port-dev，将客户机中访问IP地址为server的port端口的连接转发到宿主机的dev这个字符设备上。"guestfwd=…"这个选项也可以在一个命令行中多次重复使用。

❑bootfile=fle，让fle文件成为客户机可以使用的BOOTP[11] 启动镜像文件。它与"tfpt"选项联合使用，可以实现从网络（使用本地目录中的文件）启动客户机的功能。

smb=dir[,smbserver=addr]，激活Samba[12] 服务器，以便让Windows客户机很方便地透明地访问宿主机中的dir目录。addr设置了Samba服务器的IP地址，addr的默认值为用户模式网络中的第4个IP地址（如10.0.2.4）。值得注意的是，该选项要求宿主机中安装有Samba服务器软件，并且启动文件为"/usr/sbin/smbd"（当然这个smbd的路径可以在qemu-kvm编译时加上特定的配置使之变为用户特定的文件路径）。

下面用一个示例来介绍qemu-kvm中用户模式网络的使用。

首先，通过如下的命令行启动了一个客户机，为它配置用户模式网络，并且开启TFTP服务，还将宿主机的5022端口转发到客户机的22端口（SSH服务默认端口），将宿主机的5080端口转发到客户机的80端口（HTTP服务默认端口）。

```
qemu-system-x86_64 rhel6u3.img -m 1024 -smp 2 -net nic -net \ user,tftp=/
root/tftp,hostfwd=tcp::5022-:22,hostfwd=tcp::5080-:80
```

然后，在客户机中通过DHCP获得网络IP，检查其路由状态和默认网关，用ping命令来测试ICMP包的对外传输（如前面所说，ICMP在用户模式和网络中是不可用的），再用ssh工具测试客户机与宿主机（网关）的连通性，使用wget访问Google网站测试其外网网络连通性，并且访问宿主机中的TFTP服务（在其中测试了下载文件），还启动了客户机中的HTTP服务器。

```
[root@kvm-guest ~]# dhclient eth0
eth0: link up, 100Mbps, full-duplex, lpa 0x05E1
[root@kvm-guest ~]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:10.0.2.15  Bcast:10.0.2.255
Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:218 errors:0 dropped:0 overruns:0 frame:0
          TX packets:203 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20704 (20.2 KiB)  TX bytes:27661 (27.0 KiB)
          Interrupt:11
[root@kvm-guest ~]# route
Kernel IP routing table
Destination     Gateway    Genmask      Flags Metric Ref    Use Iface
10.0.2.0        *        255.255.255.0   U     0      0        0 eth0
default       10.0.2.2   0.0.0.0         UG    0      0        0 eth0
[root@kvm-guest ~]# ping vt-snb7 -c 1
PING vt-snb7.tsp.org (192.168.199.108) 56(84) bytes of data.

--- vt-snb7.tsp.org ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 10001ms
[root@kvm-guest ~]# ssh 10.0.2.2
The authenticity of host '10.0.2.2 (10.0.2.2)' can't be established.
RSA key fingerprint is 6b:69:03:c4:64:9d:c2:63:72:8b:34:eb:b5:c3:ad:f7.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.2' (RSA) to the list of known hosts.
root@10.0.2.2's password:
Last login: Mon Jul 30 23:50:19 2012 from 192.168.162.55
[root@jay-linux ~]# hostname
jay-linux
[root@jay-linux ~]# exit
logout
Connection to 10.0.2.2 closed.
[root@kvm-guest ~]# wget www.google.com
--2012-07-31 00:45:13--  http://www.google.com/
#......
awaiting response... 200 OK
```

```
Length: unspecified [text/html]
Saving to: "index.html"
   [ <=>            ] 14,405      --.-K/s   in 0.04s
2012-07-31 00:45:14 (387 KB/s) - "index.html" saved [14405]
[root@kvm-guest ~]# tftp 10.0.2.2
tftp>binary
tftp> verbose
Verbose mode on.
tftp> get tftp-demo.txt
getting from 10.0.2.2:tftp-demo.txt to tftp-demo.txt [octet]
Received 14 bytes in 0.0 seconds [230955 bit/s]
tftp>quit
[root@kvm-guest ~]#
[root@kvm-guest ~]# service httpd start
Starting httpd: httpd: apr_sockaddr_info_get() failed for kvm-guest
httpd: Could not reliably determine the server's fully qualified
domain name, using 127.0.0.1 for ServerName
                                                        [  OK  ]
```

最后，在外网中另外一台主机上测试前面配置的宿主机对客户机的端口转发。如下命令行是在某台主机上，通过ssh连接到宿主机的5022端口（使用-p参数指定ssh连接的端口），连接请求被自动转发到了客户机的22端口（ssh服务），然后可以登录到客户机。

```
[root@vt-snb9 ~]# ssh -p 5022 192.168.82.0
The authenticity of host '[192.168.82.0]:5022
([192.168.82.0]:5022)' can't be established.
RSA key fingerprint is a8:9d:c7:c8:bc:7a:a3:f4:13:5f:d3:8d:08:1e:56:14.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.82.0]:5022' (RSA) to the
list of known hosts.
root@192.168.82.0's password:
Last login: Tue Jul 31 00:15:36 2012
[root@kvm-guest ~]# hostname
kvm-guest
```

同样，在外部网络的一个主机上通过Firefox浏览器对宿主机的5080端口的访问，即被转发到了客户机的80端口，浏览器中显示了在

客户机的HTTP服务中测试网页内容，如图4-9所示。

图 4-9 在宿主机转发功能作用下，外部网络得以访问用户模式网络中的客户机

4.5.5 其他网络选项

关于网络的设置，还有其他几个并不太常用的选项，下面对其进行简单介绍。

（1）使用TCP socket连接客户机的VLAN

```
-net socket[,vlan=n][,name=name][,fd=h][,listen=[host]:
port][,connect=host:port]
```

使用TCP socket连接将n号VLAN连接到一个远程的qemu虚拟机的VLAN。如果有"listen=…"参数，那么QEMU会等待对port端口的连接，而host参数是可选的（默认值为本机回路IP地址127.0.0.1）。如果有"connect=…"参数，则表示去连接远端的已经使用"listen"参数的QEMU实例。可使用"fd=h"（文件描述符h）指定一个已经存在的TCP socket。

（2）使用UDP的多播socket建立客户机间的连接

```
-net socket[,vlan=n][,name=name][,fd=h][,mcast=maddr:port]
```

建立n号VLAN，使用UDP多播socket连接使其与另一个QEMU虚拟机共享，用相同的多播地址（maddr）和端口（port）为每个QEMU虚拟机建立同一个总线。多播的支持还与用户模式Linux（User Mode Linux[13] ）兼容。

（3）使用VDE swith的网络连接

```
-net vde[,vlan=n][,name=name][,sock=socketpath][,port=n]
[,group=groupname][,mode=octalmode]
```

连接n号VLAN到一个VDE[14] switch的n号端口，这个VDE switch运行在宿主机上并且监听着在socketpath上进来的连接。它使用groupname和octalmode（八进制模式的权限设置）去更改通信端口的

拥有组和权限。这个选项只有在QEMU编译时有了对VDE的支持后才可用。

（4）转存（dump）出VLAN中的网络数据

```
-net dump[,vlan=n][,file=file][,len=len]
```

将编号为n的VLAN中的网络流量转存（dump）出来保存到file文件中（默认为当前目中的qemu-vlan0.pcap文件）。最多截取并保存每个数据包中的前len（默认值为64）个字节的内容。保存的文件格式为libcap的格式，故可以使用tcpdump、Wireshark等工具来分析转存出来的文件。

（5）不分配任何网络设备

```
-net none
```

单独使用它时，表示不给客户机配置任何网络设备，可用于覆盖没有任何"-net"相关参数时的默认值"-net nic-net user"。

（6）一个新的网络配置参数"-netdev"

```
-netdev TYPE,sid=NAME,...
```

这是一种新的网络配置语法，它在宿主机中建立一个网络后端驱动，也可以实现前面的4.5.2～4.5.4节中介绍的网络配置，其中类型

TYPE的值可以为user、tap、bridge、socket等。简单列举它的两个示例配置："-netdev user,id=mynet0,hostfwd="（效果与4.5.4节中用户模式网络一样）、"-netdev bridge,id=mynet0"（效果与4.5.2节中的网桥模式网络一样）。

4.6 图形显示

在客户机中，特别是对于桌面级的Linux系统和所有的Windows系统来说，虚拟机中的图形显示是非常重要甚至必需的功能。本节主要介绍KVM中与图形界面显示相关配置。

4.6.1 SDL的使用

SDL（Simple DirectMedia Layer）是一个用C语言编写的、跨平台的、免费和开源的多媒体程序库，它提供了一个简单的接口用于操作硬件平台的图形显示、声音、输入设备等。SDL库被广泛应用于各种操作系统（如Linux、FreeBSD、Windows、Mac OS、iOS、Android等）上的游戏开发、多媒体播放器、模拟器（如QEMU）等众多应用程序之中。尽管SDL是用C语言编写的，但是其他很多流行的编程语言（如C++、C#、Java、Objective C、Lisp、Erlang、Pascal、Perl、Python、PHP、Ruby等）都提供了对SDL库的绑定，在这些编程语言中可以很方便地调用SDL的功能。

在QEMU模拟器中的图形显示默认就是使用SDL的。当然，需要在编译qemu-kvm时配置对SDL的支持，之后才能编译SDL功能到QEMU的命令行工具中，最后才能在启动客户机时使用SDL的功能。在编译qemu-kvm的系统中，需要有SDL的开发包的支持，例如，在RHEL 6.3系统中需要

安装SDL-devel这个RPM包。如果有了SDL-devel软件包，在3.4.2节中配置QEMU时默认会配置为提供SDL的支持，运行configure程序，在其输出信息中看到"SDL support yes"即表明SDL支持将会被编译进去。当然，如果不想将SDL的支持编译进去，那么在配置qemu-kvm时加上"--disable-sdl"的参数即可，在configure输出信息中会显示提示"SDL support no"。

在运行QEMU命令行的系统中必须安装SDL软件包，否则命令行在启动客户机时会有产生“无法加载libSDL”的错误。命令行的输入输出演示如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img
qemu-system-x86_64: error while loading shared libraries:
libSDL-1.2.so.0: cannot open shared object file: No such file or directory
[root@jay-linux kvm_demo]# yum install SDL    # 安装 SDL 软件包
```

SDL的功能很好用，也比较强大，不过它也有局限性，那就是在创建客户机并以SDL方式显示时会直接弹出一个窗口，所以SDL方式只能在图形界面中使用。如果在非图形界面中（如ssh连接到宿主机中）使用SDL会出现如下的错误信息：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img
Could not initialize SDL(No available video device) - exiting
```

在通过qemu-kvm命令启动客户机时，采用SDL方式，其效果如图4-10所示。

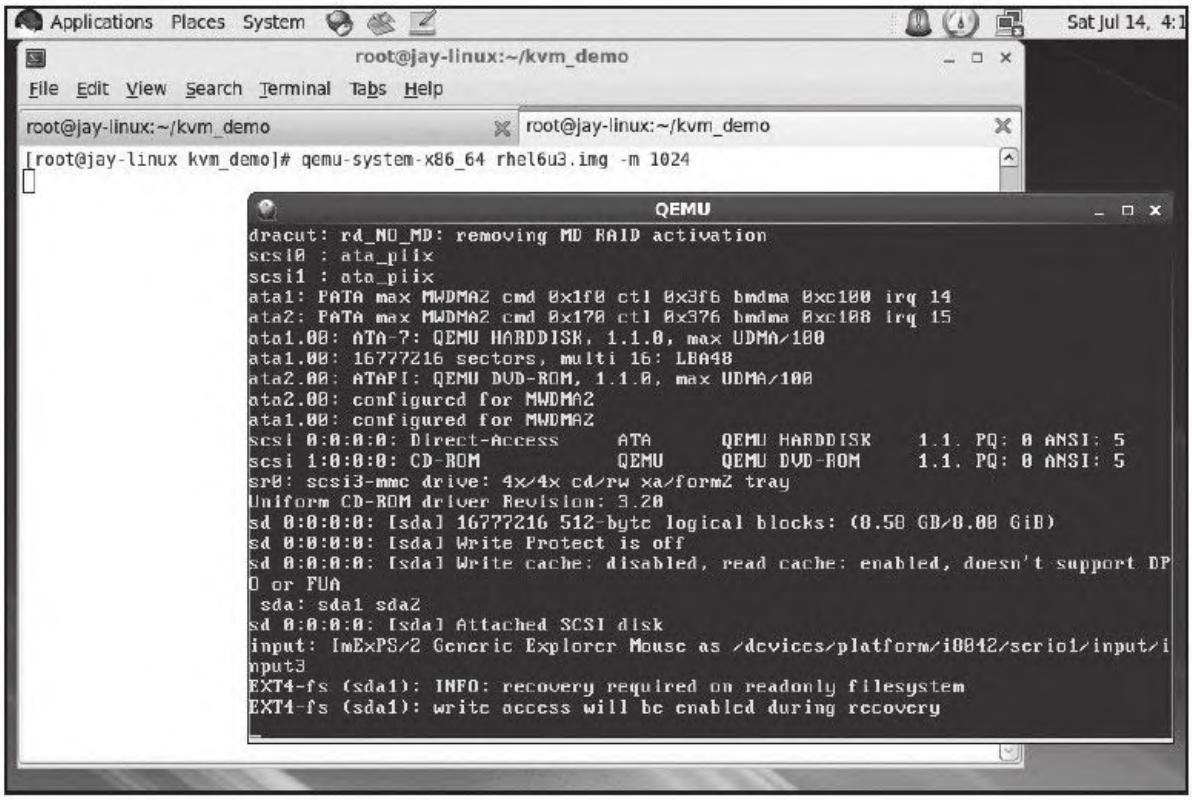


图 4-10 启动客户机时使用SDL，自动弹出客户机的显示窗口

在使用SDL时，如果将鼠标放入到客户机中进行操作会导致鼠标被完全抢占，此时在宿主机中不能使用鼠标进行任何操作。QEMU默认使用"Ctrl+Alt"组合键[15] 来实现鼠标在客户机与宿主机中切换。图4-11显示了客户机抢占了鼠标的使用场景，在QEMU monitor上部边框中提示了按哪个组合键可以释放鼠标。

图 4-11 使用SDL时客户机完全占用鼠标

使用SDL方式启动客户机时，弹出的QEMU窗口是一个普通的窗口，其右上角有最小化、最大化（或还原）和关闭等功能。其中，单击了“关闭”按钮会将QEMU窗口关闭，同时客户机也被直接关闭了，QEMU进程会直接退出。为了避免因误操作而关闭窗口从而使客户机直接退出的情况发生，QEMU命令行提供了"-no-quit"参数来去掉SDL窗口的直接关闭功能。在加了"-no-quit"参数后，SDL窗口中的“关闭”按钮的功能将会失效，而最小化、最大化（或还原）等功能正常。图4-12演示了启动客户机时加上了"-no-quit"参数的情况。

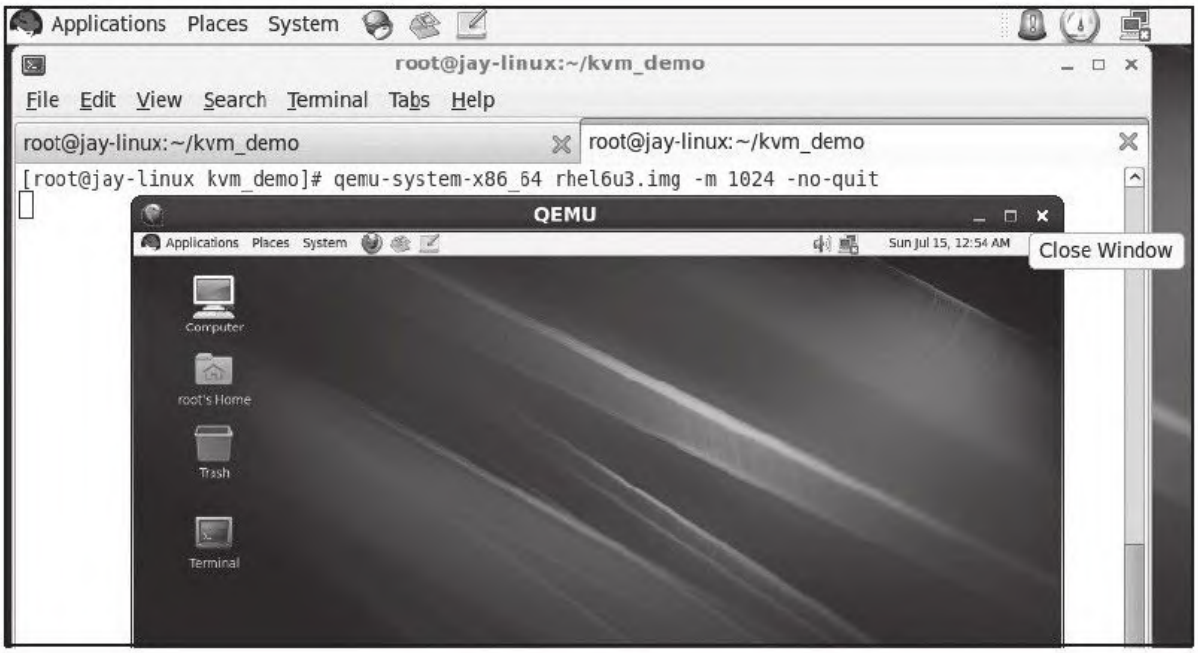


图 4-12 添加"-no-quit"参数后，SDL窗口中的关闭功能失效

4.6.2 VNC的使用

VNC（Virtual Network Computing）是图形化的桌面分享系统，它使用RFB（Remote FrameBuffer）协议来远程控制另外一台计算机系统。它通过网络将控制端的键盘、鼠标的操作传递到远程受控计算机中，而将远程计算机中的图形显示屏幕反向传输回控制端的VNC窗口中。VNC是不依赖于操作系统的，在Windows、Linux上都可以使用VNC，可以从Windows系统连接到远程的Linux VNC服务，也可以从Linux系统连接到远程的Windows系统，当然也可以在Windows对Windows系统之间、Linux对Linux系统之间使用VNC连接。

尽管qemu-kvm-1.1.0版本仍然采用SDL作为默认的图形显示方式，而使用VNC这种方式在虚拟化环境中对客户机进行管理是非常重要的，因为它克服了SDL的“只能在图形界面中使用”的局限性，而很多的Linux服务器系统通常启动到文本模式而不是图形界面模式。VNC中的操作在VNC窗口关闭或网络断开后，仍然会在服务端继续执行。另外，使用了VNC，可以在服务器端分别启动多个VNC Server进程，从而让多人通过各自VNC客户端同时连接到各自的VNC桌面并进行图形界面下的操作与维护。

下面分别讲述在宿主机中直接使用VNC和在通过qemu-kvm命令行创建客户机时采用VNC方式的图形显示。

1.宿主机中的VNC使用

笔者在本书采用的很多示例都是在KVM宿主机的远程VNC窗口中进行操作的。下面以RHEL 6.3系统为例来说明宿主机中VNC的配置。

第一步，在宿主机中安装VNC的服务器软件包（如tigervnc-server）。可以用如下命令查询vnc server的安装情况。

```
[root@jay-linux ~]# rpm -q tigervnc-server
tigervnc-server-1.0.90-0.17.20110314svn4359.el6.x86_64
```

```
[root@jay-linux ~]# rpm -qa | grep vnc
tigervnc-1.0.90-0.17.20110314svn4359.el6.x86_64
tigervnc-server-applet-1.0.90-0.17.20110314svn4359.el6.noarch
tigervnc-server-1.0.90-0.17.20110314svn4359.el6.x86_64
tigervnc-server-module-1.0.90-0.17.20110314svn4359.el6.x86_64
```

如果没有安装vnc server，则可以使用"yum install tigervnc-server"这样的命令来安装。

第二步，设置宿主机中的安全策略，使其允许VNC方式的访问，主要需要设置防火墙和SELinux的安全策略。这里为了简单起见，直接关闭了防火墙和SELinux，在实际生产环境中，需要根据特定的安全策略去设置。

使用"setup"命令来设置或关闭防火墙，如图4-13所示。

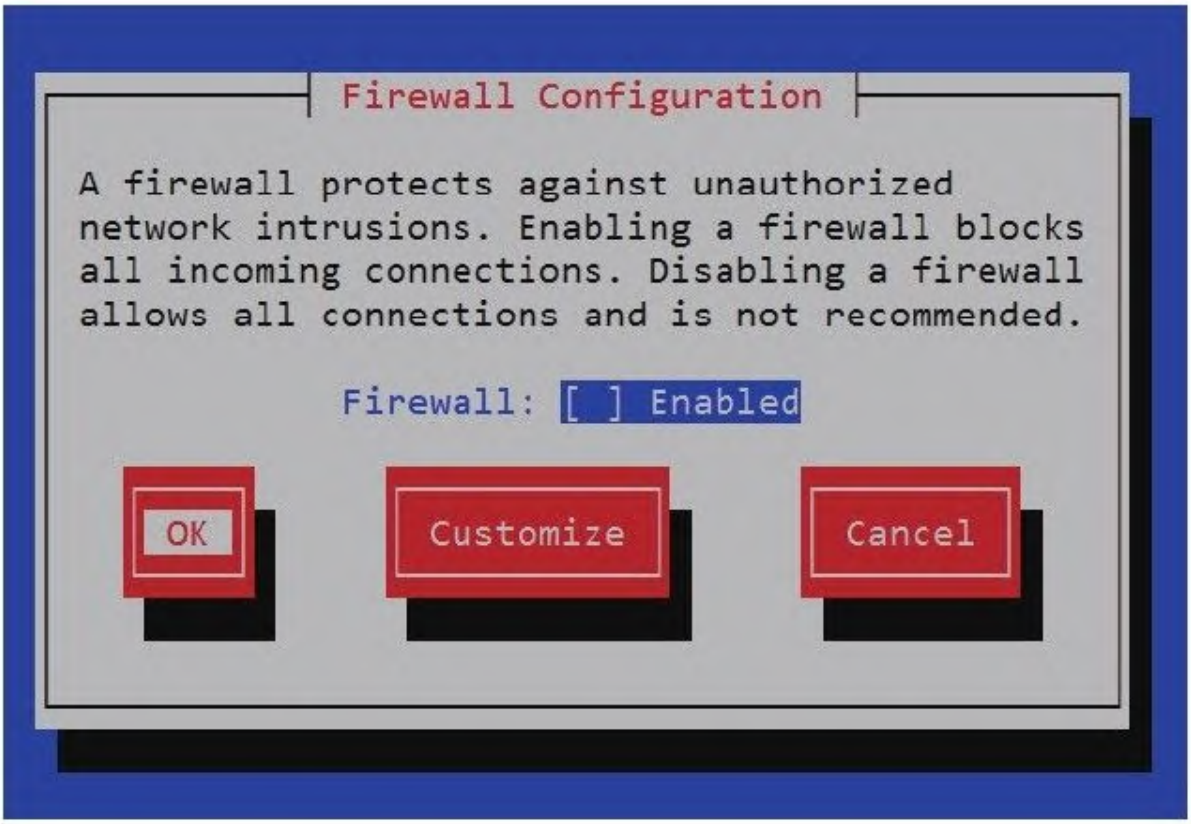


图　4-13　使用"setup"命令设置防火墙

对于SELinux的关闭，可以采取如下三种方式：

1）在运行时执行"setenforce"命令来设置，命令行如下：

```
[root@jay-linux ~]# setenforce 0
setenforce: SELinux is disabled
```

2）修改配置文件"/etc/selinux/config"，代码段如下：

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of these two values:
#     targeted - Targeted processes are protected,
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

3）设置系统启动时GRUB配置的kernel命令参数，加上"selinux=0"即可。/boot/grub/grub.conf配置文件中KVM启动条目的示例如下：

```
title Red Hat Enterprise Linux Server (3.5.0)
       root (hd0,0)
       kernel /boot/vmlinuz-3.5.0 ro
root=UUID=1a65b4bb-cd9b-4bbf-97ff-7e1f7698d3db selinux=0
       initrd /boot/initramfs-3.5.0.img
```

第三步，在宿主机中启动VNC服务端，运行命令"vncserver:1"即可启动端口为5901[16]（5900+1）的VNC远程桌面的服务器，示例如下。可以启动多个VNC Server，使用不同的端口供多个客户端使用。如果是第一次启动vncserver，系统会提示设置连接时需要输入的密码，根据需要进行设置即可。

```
[root@jay-linux ~]# vncserver :1

New 'jay-linux:1 (root)' desktop is jay-linux:1

Starting applications specified in /root/.vnc/xstartup
Log file is /root/.vnc/jay-linux:1.log
[root@jay-linux ~]# ps -ef | grep -i Xvnc
root      4877     1  0 Jul05 ?        00:02:48 /usr/bin/Xvnc :1
-desktop jay-linux:1 (root) -auth /root/.Xauthority -geometry
1024x768 -rfbwait 30000 -rfbauth /root/.vnc/passwd -rfbport 5901
-fp catalogue:/etc/X11/fontpath.d -pn
root     17239 16324  0 02:38 pts/0    00:00:00 grep --color=auto -i Xvnc
```

第四步，在客户端中，安装VNC的客户端软件。在RHEL 6.3中可以安装上面查询结果中列出的"tigervnc"这个RPM包；在Windows中，可以安装RealVNC的VNC Viewer软件。图4-14展示了在Windows7上运行VNC Viewer情形。

图　4-14　在Windows 7上的VNC Viewer的远程连接界面

第五步，连接到远程的宿主机服务器，使用的格式为"IP(hostname)：PORT"。在Windows中，在图4-14中输入需要访问的

IP或主机名加上端口即可连接；在RHEL 6.3中可以启动"TigerVNC Viewer"，其界面基本情况和图4-14类似，在命令行用"vncviewer jay-linux:1"这样的命令来连接到某台机器的某个VNC桌面。在连上VNC后，会要求输入密码验证（这个密码就是第一次启动vncserver时输入的密码），验证成功后即可正常连接到远程VNC桌面，如图4-15所示。

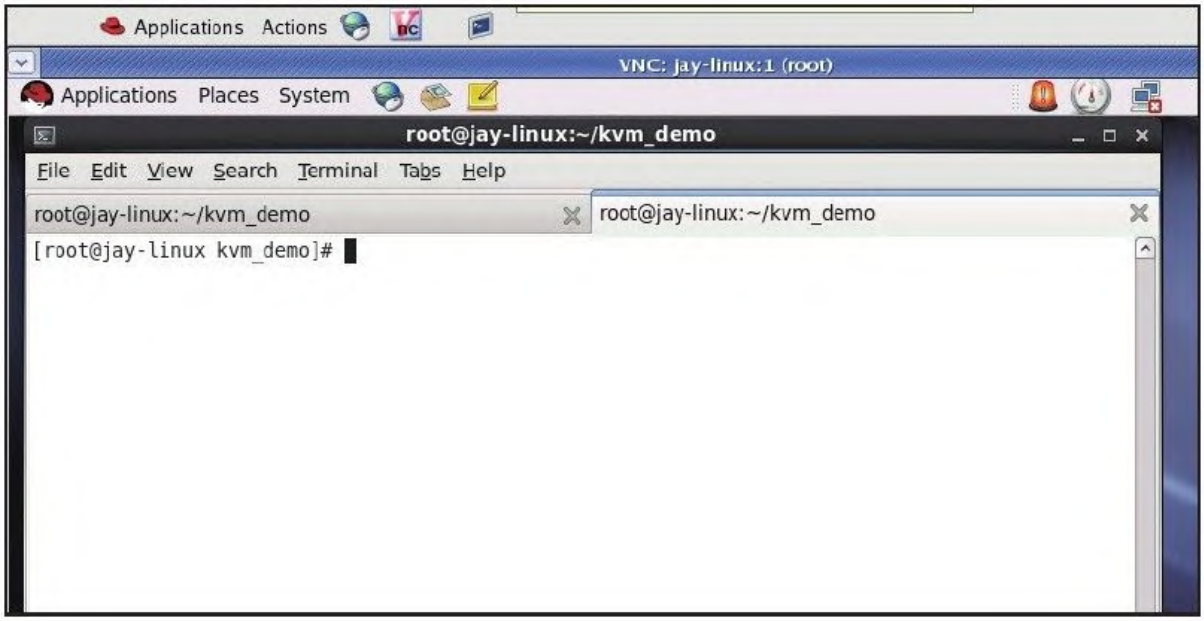


图　4-15　远程连接到宿主机的1号VNC桌面

2.QEMU使用VNC图形显示方式启动客户机

在qemu-kvm的命令行中，添加"-vnc display,option"参数就能让VGA显示输出到VNC会话中而不是SDL中。如果在进行qemu-kvm编译时没有SDL的支持，却有VNC的支持，则qemu-kvm命令行在启动客户机时不

需要"-vnc"参数也会自动使用VNC而不是SDL，在本书3.6节中用qemu-kvm命令行启动客户机时提示"VNC server running on '::1:5900'"就是这个原因。

在QEMU命令行的VNC参数中，display参数是必不可少的，它有如下3种具体的值。

（1）host:N

表示仅允许从host主机的N号显示窗口来建立TCP连接到客户机。在通常情况下，QEMU会根据数字N建立对应的TCP端口，其端口号为5900+N。而host值在这里是一个主机名或一个IP地址，是可选的，如果host值为空，则表示QEMU建立的Server端接受来自任何主机的连接。增加host参数值，可以阻止来自其他主机的VNC连接请求，从而在一定程度上提高了使用QEMU的VNC服务的安全性。

（2）unix:path

表示允许通过Unix domain socket[18] 来连接到客户机，而其中的path参数是一个处于监听状态的socket的位置路径。这种方式使用得不多，故不详细叙述。

（3）none

表示VNC已经被初始化，但是并不在开始时启动。而在需要真正使用VNC之时，可以在QEMU monitor中的change命令可以用来启动VNC连接。

作为可选参数的option则有如下几个可选值，每个option标志通过逗号隔开。

（1）reverse

表示“反向”连接到一个处于监听中的VNC客户端，这个客户端是由前面的display参数（host:N）来指定的。需要注意的是，在反向连接这种情况下，display中的端口号N是处于监听中的TCP端口，而不是现实窗口编号，即如果客户端（IP地址为IP_Demo）已经监听的命令为"vncviewer-listen:2"，则这里的VNC反向连接的参数为"-vnc IP_Demo:5902,reverse"，而不是用2这个编号。

（2）password

表示在客户端连接时需要采取基于密码的认证机制，但这里只是声明它使用密码验证，其具体的密码值必须到QEMU monitor中用change命令设置。

（3）"tls"、"x509=/path/to/certificate/dir"、"x509verify=/path/to/certificate/dir"、"sasl"和"acl"

这5个选项都是与VNC的验证、安全[17] 相关的选项，本书不对其详述。

上面已经简单介绍了QEMU命令行中关于VNC图形显示的一些参数及选项的意义和基本用法，下面几个示例是对上面内容的实践，主要介绍VNC的一些具体操作方法。

准备两个系统，一个是KVM的宿主机系统A（IP：192.168.185.229，主机名：jay-linux），另一个是安装有RHEL 6.3的Linux系统B（IP:192.168.199.99），这两个系统之间可以通过网络连通。

（1）示例1

在启动客户机时，带有一个不需要密码的对任何主机都可以连接的VNC服务。

在宿主机A系统中，运行如下命令即可启动服务。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m 1024 -vnc :2
```

在宿主机中，用如下命令连接到客户机中。

```
[root@jay-linux kvm_demo]# vncviewer :2   #或vncviewer localhost:2
```

而在B系统中，用如下命令中即可连接到A主机中对客户机开启的VNC服务。

```
[root@vt-snb9 ~]# vncviewer jay-linux:2          # 当然，不用主机名而用 IP 也是可以的
```

（2）示例2

在启动客户机时，带有一个需要密码的仅能通过本机连接的VNC服务。

在宿主机A系统中，运行如下命令即可将其启动，如前面提过的，VNC的密码需要到QEMU monitor中设置，所以这里加了"-monitor stdio"参数使monitor指向目前的标准输入输出，这样可以直接输入"change vnc password"命令来设置密码。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -vnc localhost:2,password -monitor stdio
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) change vnc password "123456"
(qemu) change vnc password
Password: ******
(qemu)
```

在QEMU monitor中，运行change vnc password"123456"命令可以将VNC密码设置为“123456”，而如果使用"change vnc password"命令不加具体密码，QEMU monitor会交互式地提示用户输入密码。这两个设置方法效果都是一样的。

设置好VNC密码后，在本机系统（A）中运行"vncviewer:2"或"vncviewer localhost:2"即可连接到客户机的VNC

显示上，系统会需要密码验证，输入之前设置的密码即可，如图4-16所示。

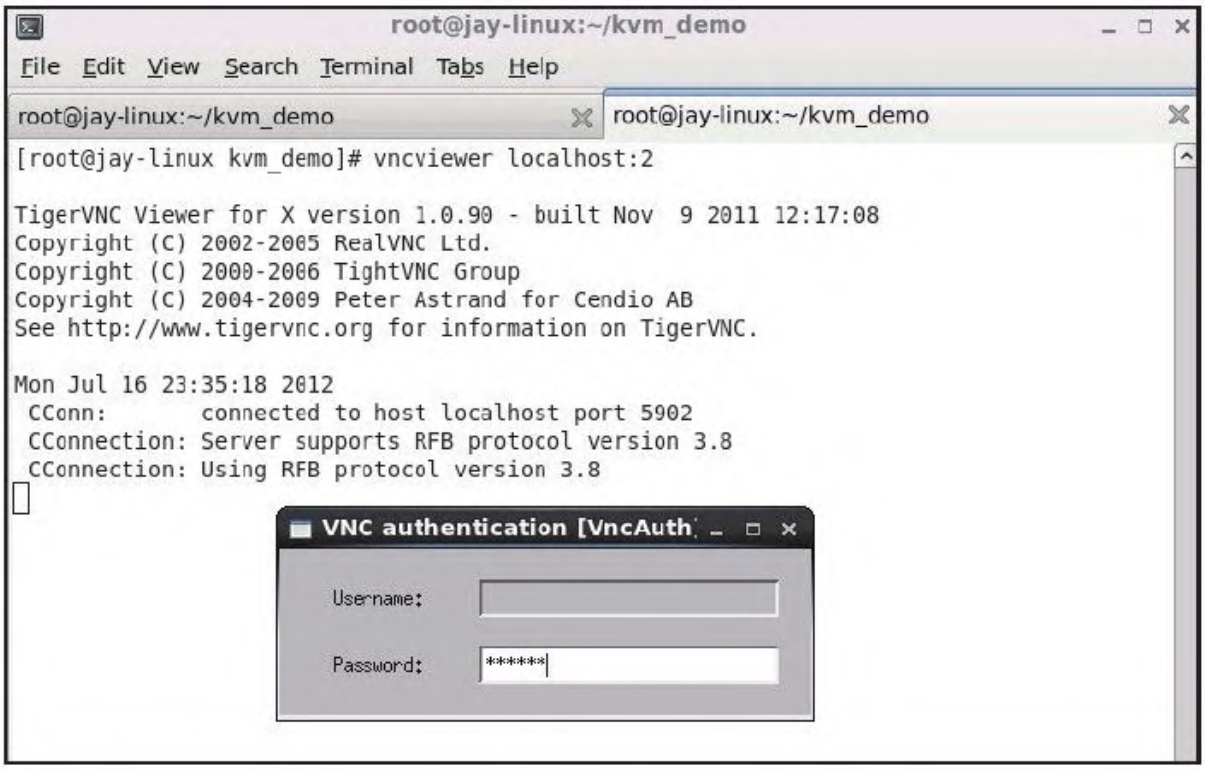


图 4-16 宿主机连接本机的VNC，需要输入密码

由于在QEMU启动时设置了只有通过主机名为"localhost"（即宿主机A系统）的主机的2号显示窗口才能连接到VNC服务，因此在B系统中无论是使用A主机的IP还是主机名去连接其2号VNC，都会提示“连接被拒绝”，如图4-17所示。

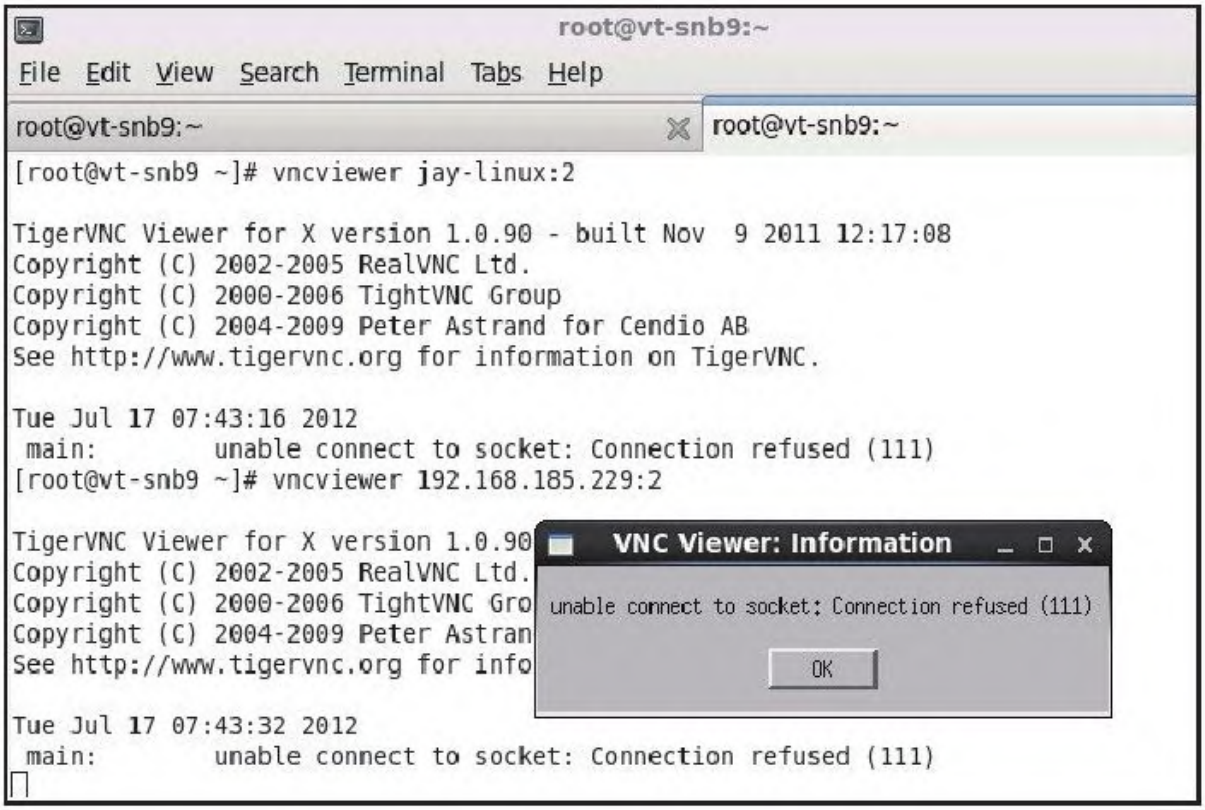


图 4-17 B系统连接A系统中QEMU的VNC服务，连接被拒绝

（3）示例3

启动客户机时并不启动VNC，启动后根据需要使用命令才真正开启VNC。

在宿主机A系统中，将VNC参数中display设置为none，然后在重定向到标准输入输出的QEMU monitor中使用"change vnc:2"这样的命令来开启VNC，操作命令如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -vnc none -monitor stdio
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) change vnc :2
(qemu)
```

（4）示例4

虽然在启动客户机时设置了VNC的参数，但是仍希望在guest启动后动态修改此参数。

这个需求可以通过QEMU monitor中的"change vnc XX"命令来实现。在客户机启动时，仅允许localhost:2连接到VNC，并且没有设置密码。但是在客户机启动后，根据实际的需求，改变VNC设置为：允许来自任意主机的对本宿主机上3号VNC端口的连接来访问客户机，还增加了访问时的密码验证。实现这个需求的命令行操作如下：

```
root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -vnc localhost:2 -monitor stdio
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) change vnc :3
(qemu) change vnc password "123456"
(qemu)
```

当然在本示例中，"-monitor stdio"这个参数不是必须的，只需要在启动客户机后切换到QEMU monitor中执行相关的change命令即可。

（5）示例5

在启动客户机时，将客户机的VNC反向连接到一个已经处于监听状态的VNC客户端。

本示例中的这种使用场景也是非常有用的，如某用户在KVM虚拟机中调试一个操作系统，经常需要多次重启客户机中的操作系统，而该用户并不希望每次都重新开启一个VNC客户端去连接到客户机。有了"reverse"参数，在实现反向连接后，该用户就可以先开启一个VNC客户端使其处于监听某个端口的状态，然后在每次用qemu-kvm命令行重启客户机的时候，自动地反向连接到那个处于监听状态的VNC客户端。

首先，在B系统中启动vncviewer处于listen状态，如图4-18所示。

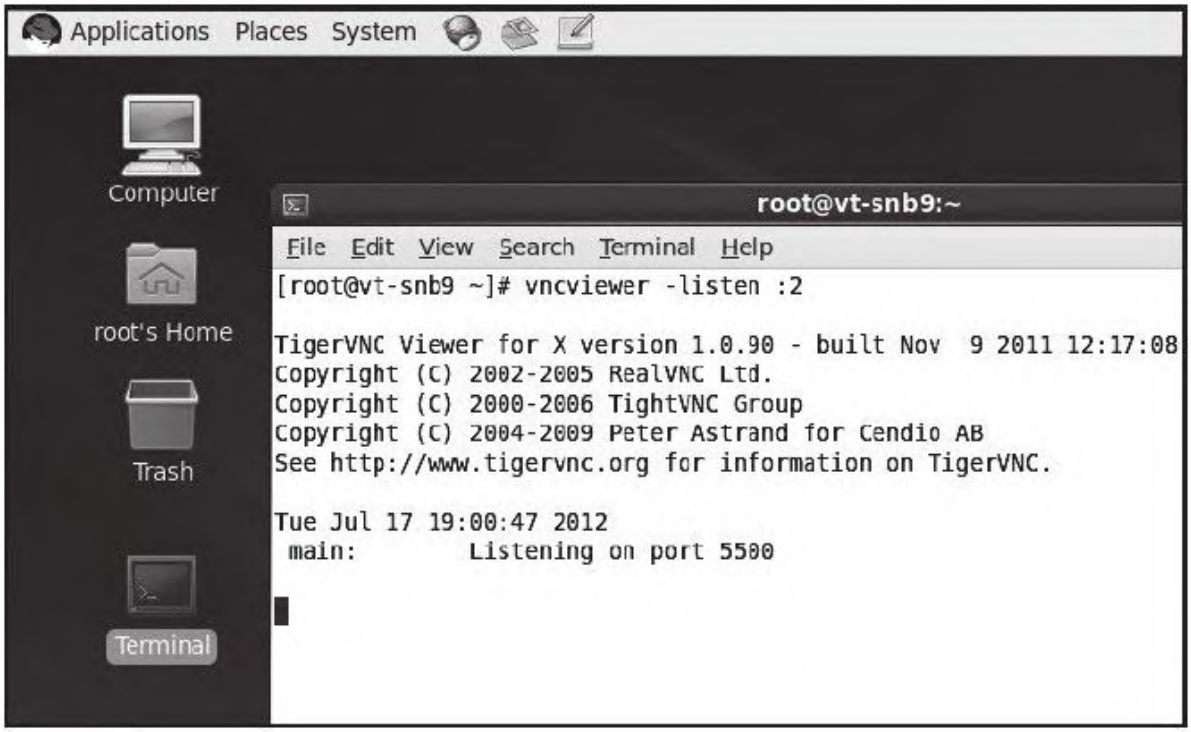


图 4-18 B在系统中启动一个VNC客户端处于监听状态

然后，在宿主机（A系统）中启动客户机，VNC中的参数包含B主机的IP（或主机名）、TCP端口，以及"reverse"选项。其中TCP端口的值根据B系统的监听端口来确定，从图4-18可以看到"Listening on port 5550"，所以监听的TCP端口是5500。在宿主机系统中，启动客户机的命令行操作如下：

```
qemu-system-x86_64 rhel6u3.img -vnc 192.168.199.99:5500,reverse
```

在客户机启动后，在B系统中监听中的VNC客户端就会自动连接上A系统的客户机，呈现客户机启动过程的界面，如图4-19所示。

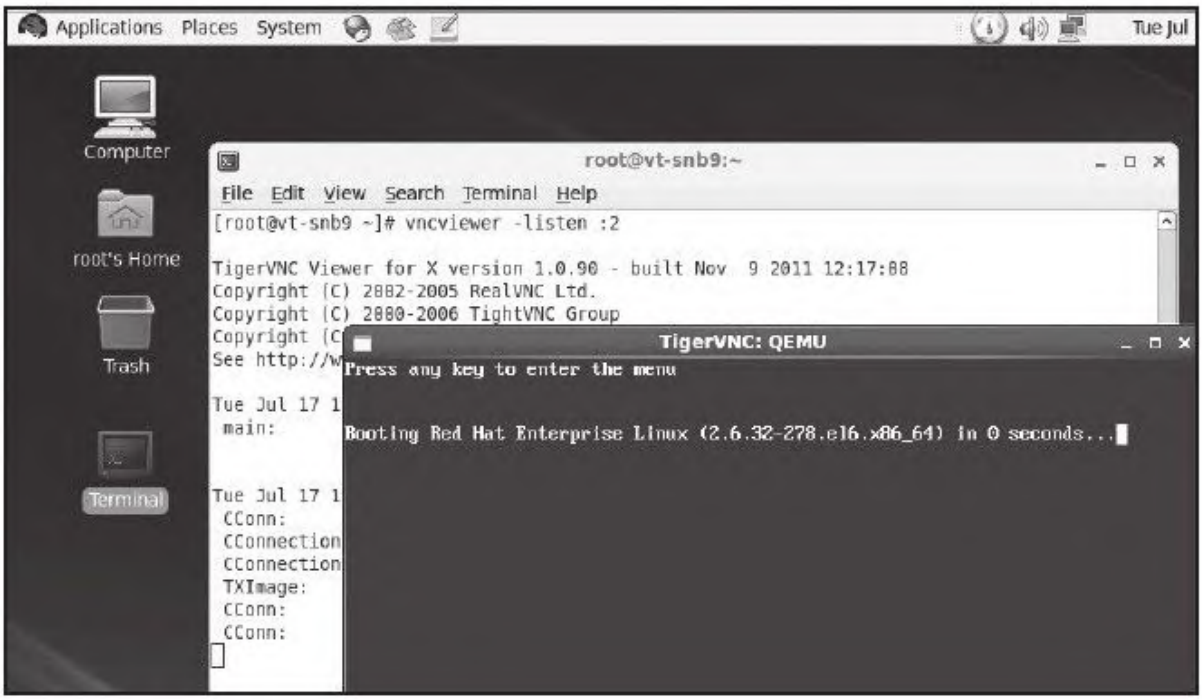


图 4-19 B系统中监听着的VNC客户端已经连接到了A系统中的客户机

当然，本示例在同一个系统中也是可以操作的，在宿主机系统上监听，然后在宿主机系统上启动客户机使其“反向”连接到本地处于监听状态的VNC客户端。

4.6.3 VNC显示中的鼠标偏移

在QEMU中的VNC显示客户机很方便使用，而且有较多可用选项，所以其功能比较强大。不过VNC显示客户机也有一个小的缺点，那就是在VNC显示中有鼠标偏移的现象。这里的偏移现象是指通过VNC连接到客户机中操作时，会看到两个鼠标，一个是客户机中的鼠标（这个是让

客户机操作实际生效的鼠标），另一个是连接到客户机VNC的客户端系统中的鼠标。这两个鼠标的焦点通常不重合，而且相差的距离还有点远（如图4-20所示），这样会导致在客户机中的鼠标移动非常不方便。特别是在Windows客户机系统中，通常运行在图形界面之下，可能多数的操作都需要移动和点击鼠标，如果鼠标操作不方便就会非常影响用户体验。

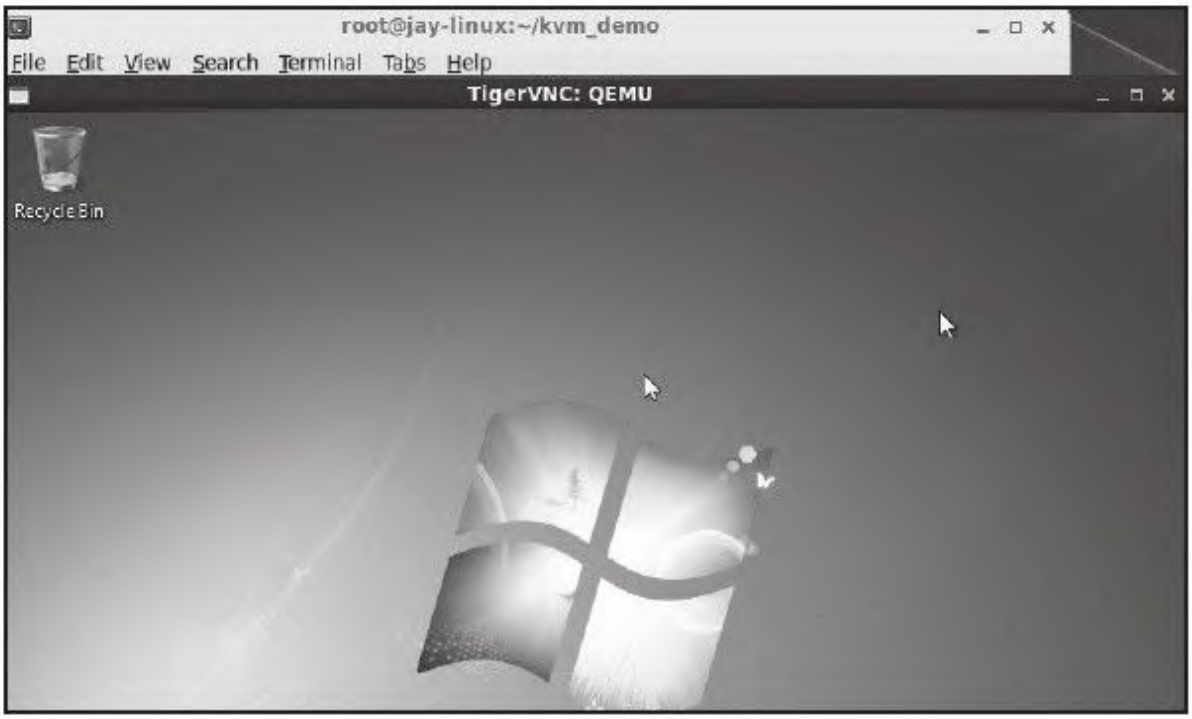


图 4-20 VNC中的Windows 7客户机的鼠标偏移现象

在图4-20中，右上方的鼠标是Windows 7客户机中实际使用的鼠标，而中间位置的鼠标为连接VNC的客户端所在系统的鼠标。

因为存在鼠标偏移的问题，所以在使用VNC方式启动客户机时，强烈建议"-usb"和"-usbdevice tablet"这两个USB的选项一起使用，从而解决上面提到的鼠标偏移问题。"-usb"参数开启为客户机USB驱动的支持（默认已经生效，可以省略此参数），而"-usbdevice tablet"参数表示添加一个"tablet"类型的USB设备。"tablet"类型的设备是一个使用绝对坐标定位的指针设备，就像在触摸屏中那样定位，这样可以让QEMU能够在客户机不抢占鼠标的情况下获得鼠标的定位信息。

在最新的QEMU中，与"-usb-usbdevice tablet"参数功能相同，也可以用"-device piix3-usb-uhci-device usb-tablet"参数。目前QEMU社区也主要推动使用功能丰富的"-device"参数来替代以前的一些参数（如：-usb等）。

用如下的命令启动客户机，解决了VNC中的鼠标漂移问题，如图4-21所示。

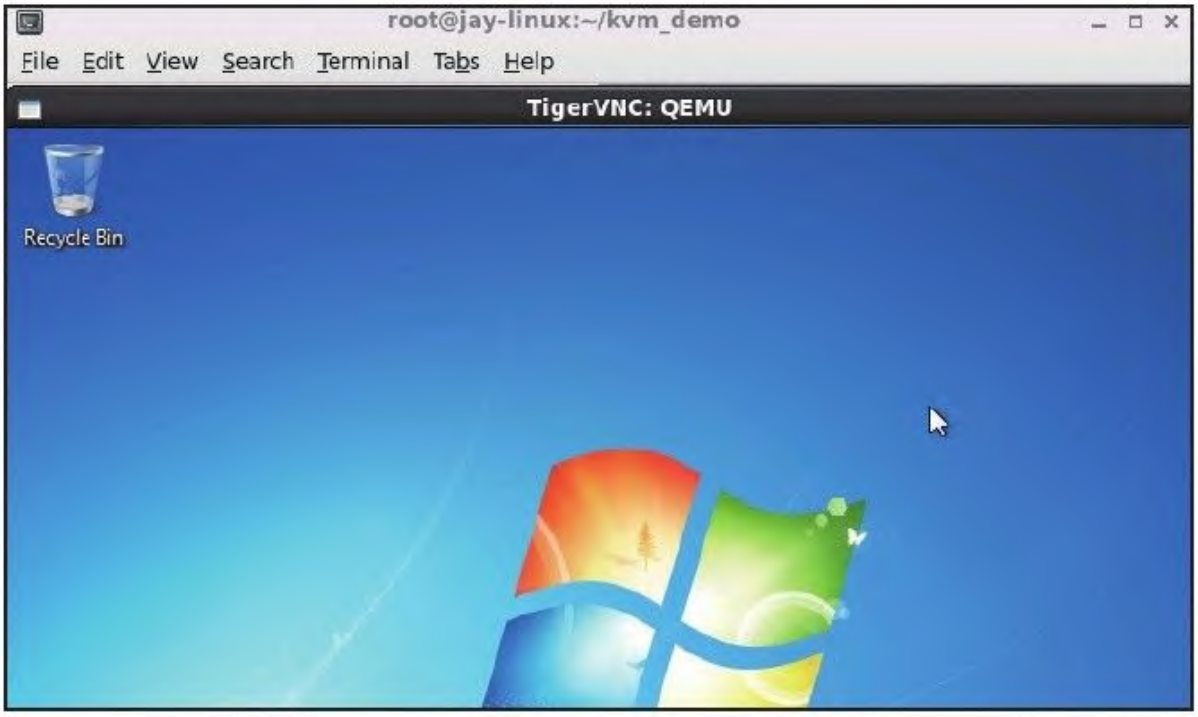


图 4-21 VNC连接到Windows 7客户机中，无鼠标偏移现象

```
qemu-system-x86_64 rhel6u3.img -vnc :2 -usb -usbdevice tablet
```

或者

```
qemu-system-x86_64 rhel6u3.img -vnc :2 -device piix3-usb-uhci
-device usb-tablet
```

4.6.4 非图形模式

在qemu-kvm命令行中，添加"-nographic"参数可以完全关闭QEMU的图形界面输出，从而让QEMU在该模式下完全成为简单的命令行工具。而在QEMU中模拟产生的串口被重定向到了当前的控制台（console）中，所以在客户机中对其内核进行配置使内核的控制台输

出重定向到串口后，依然可以在非图形模式下管理客户机系统或调试客户机的内核。

需要修改客户机的grub使其在kernel行加上将console输出重定向到串口ttyS0，如下为对一个客户机进行修改后的grub配置文件。

```
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux (2.6.32-279.el6.x86_64)
      root (hd0,0)
      kernel /boot/vmlinuz-2.6.32-279.el6.x86_64 ro
root=UUID=9a971721-db8f-4002c-a3f4-f4ae8b037ba3 3 console=ttyS0
initrd /boot/initramfs-2.6.32-279.el6.x86_64.img
```

用"-nographic"参数关闭图形输出，其启动命令行及客户机启动（并登录进入客户机）的过程如下所示，可见内核启动的信息就通过重定向到串口从而输出在当前的终端之中，而且可以通过串口登录到客户机系统（有的客户机Linux系统需要进行额外的设置才能允从串口登录）。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -nographic
Initializing cgroup subsys cpuset
Initializing cgroup subsys cpu
Linux version 2.6.32-278.el6.x86_64
(mockbuild@x86-010.build.bos.redhat.com) (gcc version 4.4.6
20120305 (Red Hat 4.4.6-4) (GCC) ) #1 SMP Fri Jun 1 14:17:12 EDT 2012
Command line: ro root=UUID=9a971721-db8f-402c-a3f4-f4ae8b037ba3 3 console=ttyS0
KERNEL supported cpus:
  Intel GenuineIntel
  AMD AuthenticAMD
  Centaur CentaurHauls
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009f400 (usable)
 BIOS-e820: 000000000009f400 - 00000000000a0000 (reserved)
 BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
<!-- 此处省略数百行启动时的串口输出信息 -->
Starting postfix: [  OK  ]
Starting abrt daemon: [  OK  ]
Starting ksm: [  OK  ]
Starting ksmtuned: [  OK  ]
Starting crond: [  OK  ]
Starting atd: [  OK  ]
Starting rhsmcertd 240 1440[  OK  ]
Starting certmonger: [  OK  ]
Red Hat Enterprise Linux Server release 6.3 (Santiago)
Kernel 2.6.32-278.el6.x86_64 on an x86_64
kvm-guest login: root      # 这里就是客户机的登录界面了
Password:
Last login: Tue Jul 17 15:56:21 on ttyS0
[root@kvm-guest ~]# pwd
/root
[root@kvm-guest ~]# date
Tue Jul 17 16:01:03 CST 2012
[root@kvm-guest ~]#
```

4.6.5 显示相关的其他选项

QEMU还有不少关于图形显示相关的其他选项，本节再介绍其中几个比较有用的。

1.-curses

让QEMU将VGA[19] 显示输出到使用curses/ncurses[20] 接口支持的文本模式界面，而不是使用SDL来显示客户机。与"-nographic"模式相比，它的好处在于，由于它是接收客户机VGA的正常输出而不是串口的输出信息，因此不需要额外更改客户机配置将控制台重定向到串口。当然，为了使用"-curses"选项，在宿主机中必须有“curses或ncurses”这样的软件包提供显示接口的支持。

通过Putty连接到宿主机，然后添加"-curses"参数的qemu-kvm命令行来启动客户机，启动到登录界面如图4-22所示。

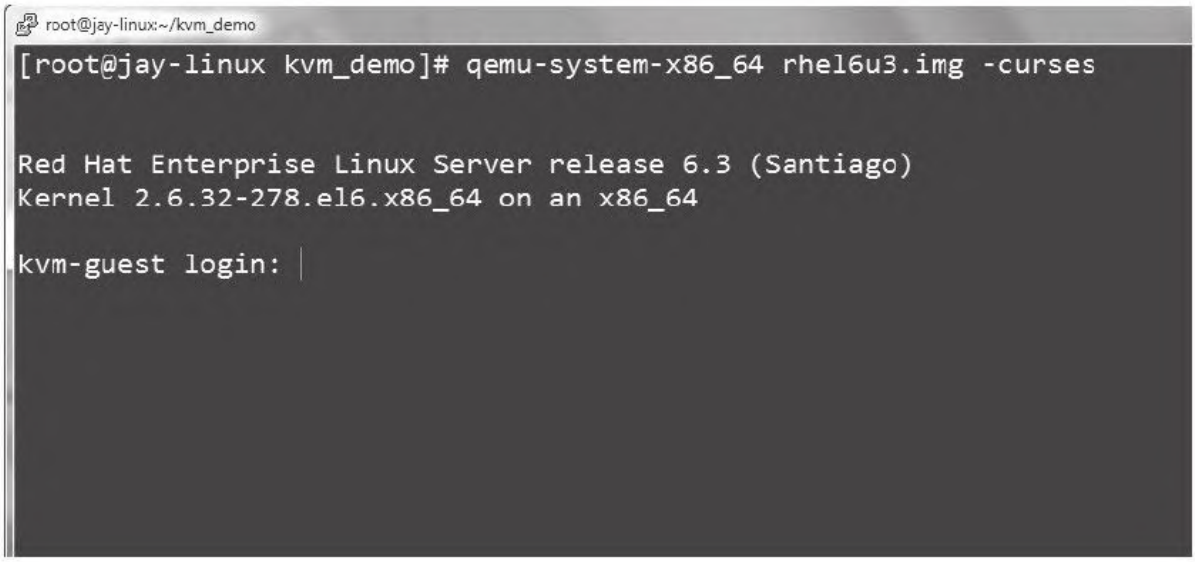


图 4-22 通过"ncurses"显示的文本模式下的客户机启动命令行和登录界面

2.-vga type

选择为客户机模拟的VGA卡的类别，可选类型有如下4种。

（1）cirrus

为客户机模拟出"Cirrus Logic GD5446"显卡，在客户机启动后，可以在客户机中看到VGA卡的型号，如在Linux中可以用"lspci"查看到VGA卡的信息。这个选项对图形显示的体验并不是很优秀，它的彩色是16位的，分辨率也很高，仅支持2D显示，不支持3D，不过绝大多数的系统（包括Windows95）都支持这个系列的显卡，所以这种类型是目前的QEMU对VGA的默认类型。

在Linux客户机中，查看VGA卡的类型，可以使用下面的命令行：

```
[root@kvm-guest ~]# lspci | grep VGA
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
```

（2）std

模拟标准的VGA卡，带有Bochs VBE扩展。当客户机支持VBE[21] 2.0及以上标准时（目前流行的操作系统多数都支持），如果需要支持更高的分辨率和彩色显示深度，就需要使用这个选项。

（3）vmware

提供对"VMWare SVGA-II"兼容显卡的支持。

（4）none

关闭VGA卡，使SDL或VNC窗口中无任何显示，一般不使用这个参数。

3.-no-frame

使SDL显示时没有边框。选择这个选项后，图4-10和4-11中SDL窗口就没有边框的修饰。

4.-full-screen

在启动客户机时，就自动使用全屏显示。

5.-alt-grab

使用"Ctrl+Alt+Shift"组合键去抢占和释放鼠标。从而"Ctrl+Alt+Shift"组合键也会成为QEMU中的一个特殊功能键。在QEMU中默认使用"Ctrl+Alt"组合键，所以本书常提到在SDL或VNC中用"Ctrl+Alt+2"组合键切换到QEMU monitor的窗口，而使用了"-alt-grab"选项后，应该相应改用"Ctrl+Alt-Shift+2"组合键切换到QEMU monitor窗口。

6.-ctrl-grab

使用“右Ctrl”键去抢占和释放鼠标，使其成为QEMU中的特殊功能键，这与前面的"-alt-grab"的功能类似。

4.7 本章小结

本章主要介绍了QEMU/KVM中关于CPU、内存、磁盘、网络、图形显示等计算机系统的最核心最基本的部件的简单原理、详细配置及实践操作，同时还提及了一些命令行工具（如ps、brctl、lspci、fdisk等），展示了几个配置脚本（如查看CPU信息的cpu-info.sh、建立网络链接的qemu-ifup等）。相信通过阅读本章的内容，读者可以创建自己的客户机，并且可以配置成功其中的CPU、内存、磁盘、网络等基本部件，从而满足普通的应用需求。在下一章，将会介绍KVM中一些提高性能的特性（如virtio、VT-d等）、其他高级功能（如KSM、动态迁移、热插拔、嵌套虚拟化等），以及一些并不太常用的特性（如SMEP功能、AVX、AES新指令等），使读者对KVM的各项功能有更完整的了解，并能根据实际情况选择适当的优化方法。

4.8 本章注释和参考阅读

1.本章注释

[1]Romley是Intel的一个平台的代号，在它里面配置有SandyBridge-EP的CPU和Patsburg芯片组，这里提及的Romley-EP 4S是有4个CPU socket的系统。关于Intel各种CPU、芯片组、网卡、主板等硬件的代号，可参见http://en.wikipedia.org/wiki/List_of_Intel_codenames。

[2]CPU支持了超线程（HT）技术，还需要在BIOS中打开它的开关才能使用。在BIOS中，超线程的设置可能会在"Advanced→CPU Configuration"下设置，通常标识为"Hyper-Threading"。另外，直到2012年笔者写本书时，AMD还没提供Hyper-Threading的支持（AMD目前走的是mult-core的路线），而Intel支持mult-core和hyper-threading一起使用。

[3]NUMA（Non-Uniform Memory Access，非一致性内存访问）是一种在多处理系统中的内存设计架构，在多处理器中，CPU访问系统上各个物理内存的速度可能不一样，一个CPU访问其本地内存的速度比访问（同一系统上）其他CPU对应的本地内存快一些。

[4]Nehaem是Intel的第二代45nm微处理器架构，通过如下的网址可以了解更多关于Nehalem平台的知识：http://www.realworldtech.com/nehalem/

[5]TLB（translation lookaside buffer，旁路转换缓冲）是内存管理硬件以提高虚拟地址转换速度的缓存。TLB是页表（page table）的缓存，保存了一部分页表。关于TLB的更多信息，可以查看wikipedia中的介绍：http://en.wikipedia.org/wiki/Translation lookaside buffer。

[6]sysfs是一个虚拟的文件系统，它存在于内存之中，它将Linux系统中设备和驱动的信息从内核导出到用户空间。sysfs也用于配置当前系统，此时其作用类似于sysctl命令。systfs通常是挂载在/sys目录上的，通过"mount"命令可以查看到sysfs的挂载情况。

[7]关于swap space，Redhat公司的文档曾有这样的推荐：在其RHEL5.x或RHEL6.x系列的Linux发行版中，物理内存小于4GB，推荐不少于2GB的交换空间；物理内存在4GB～16GB之间，推荐不少于4GB的交换空间；物理内存在16GB～64GB之间，推荐不少于8GB的交换空间；内存在64GB～256GB之间，推荐不少于16GB的交换空间。

[8]一个QEMU bug：Guest boot failed when the drive interface is scsi，网址是：

https://bugs.launchpad.net/qemu/+bug/485251

[9]稀疏文件（sparse file），是计算机系统块设备中能有效利用磁盘空间的文件类型，它用元数据（metadata）中的简要描述来标识哪些块是空的，只有在空间被实际数据占用时，才将数据实际写到磁盘中。可以参考wikipedia上的"sparse file"词条，网址是：http://en.wikipedia.org/wiki/Sparse_file。

[10]在QEMU的源代码中有一个专门的"slirp"目录使用了Slirp的实现。Slirp是通过普通的终端模拟PPP、SLIP等连接到Internet的开源软件程序，更多信息请参考其官方网站：http://slirp.sourceforge.net/

[11]BOOTP即Bootstrap Protocol，是一种比较早出现的远程启动协议，后来DHCP协议实现了BOOTP几乎所有的功能（增加了很多自有功能），已基本把BOOTP取代了，可参考如下网页上的描述：http://en.wikipedia.org/wiki/Bootstrap-Protocol。

[12]Samba是类Unix（包括Linux）系统与Windows系统进行相互操作的标准软件，主要用于文件的跨系统共享。其官方主页为：http://www.samba.org/

[13]Use-Mode Linux是一种安全地运行Linux各个版（单独的kernel）和Linux进程的方式。该项目主页为：http://user-mode-

linux.sourceforge.net/

[14]VDE（Virtual Distributed Ethernet)，详细内容参考其项目主页：http://vde.sourceforge.net/

[15]对于QEMU获取和释放鼠标的组合键（如"Alt+Ctrl"），一般来说是指位于键盘左边的键而不是键盘右边的键，当需要提到键盘右边的Ctrl、Alt、Shift键时，一般都会特别指明是右边的键（如“右Ctrl键”）。

[16]VNC一般采用TCP端口5900+N（N为在vncserver启动时指定的桌面端口号）作为服务端的端口，在VNC Viewer远程连接时，既可以用5900+N（如5901）这样的端口，也可以直接用桌面号N来连接，一般情况下VNC Viewer程序会自适应地做相应的转换。

[17]关于VNC的安全问题及其设置，如果对安全性要求很高，请参考如下网页：http://wiki.qemu.org/download/qemu-doc.html#vnc005fsecurity。

[18]想了解更多关于UNIX Domain Socket的内容，可以参考如下网页：http://beej.us/guide/bgipc/output/html/multipage/unixsock.html。

[19]VGA（Video Graphics Array）是IBM于1987年在其计算机中引入的显示硬件接口，后来也成为事实上的工业标准，目前绝大多数的PC和x86服务器都支持VGA接口及其显示标准。

[20]curses是类UNIX系统（包括：UNIX、Linux、BSD等）上终端控制的库，应用程序能够基于它去构建文本模式下的用户界面。而ncures是指"new curses"，发布于1993年，是目前curses最著名的实现，也是在目前类UNIX系统中使用非常广泛的。

[21]VBE（VESA BIOS Extensions）是VESA的一个标准，目前版本是3.0，请参考如下链接获取关于"VBE"的更多信息：http://en.wikipedia.org/wiki/VESA_BIOS_Extensions。

2.参考阅读

（1）关于Sandy Bridge处理器架构，可参考:

http://www.realworldtech.com/sandy-bridge/

（2）关于huge page的介绍，可参考：

http://lwn.net/Articles/374424/

（3）关于QEMU的网络，可参考：

http://people.gnome.org/～markmc/qemu-networking.html

（4）最好不要在对qcow2格式做镜像文件时使用"aio=native"，Redhat对其有说明，可参考：

https://access.redhat.com/knowledge/articles/41313

（5）关于QEMU的QED文件格式，可以参考：

http://www.linux-kvm.com/content/qed-qemu-enhanced-disk-format

http://wiki.qemu.org/Features/QED

（6）关于iSCSI，可参考：

http://en.wikipedia.org/wiki/iSCSI

（7）关于Fibre Channel，可参考：

http://en.wikipedia.org/wiki/Fibre_Channel

第5章　KVM高级功能详解

在第4章介绍了CPU、内存、存储、网络、图形等KVM的基本功能之后，本章将会介绍更多KVM的更高级功能。这些功能包括半虚拟化驱动、VT-d、SR-IOV、热插拔、动态迁移、KSM、AVX、cgroups、从物理机或虚拟机中迁移到KVM，以及QEMU监控器和qemu-kvm命令行的各种选项的使用。

5.1　半虚拟化驱动

5.1.1　virtio概述

KVM是必须使用硬件虚拟化辅助技术（如Intel VT-x、AMD-V）的Hypervisor，在CPU运行效率方面有硬件支持，其效率是比较高的；在有Intel EPT特性支持的平台上，内存虚拟化的效率也较高。QEMU/KVM提供了全虚拟化环境，可以让客户机不经过任何修改就能运行在KVM环境中。不过，KVM在I/O虚拟化方面，传统的方式是使用QEMU纯软件的方式来模拟I/O设备（如第4章中提到模拟的网卡、磁盘、显卡等），其效率并不非常高。在KVM中，可以在客户机中使用半虚拟化驱动（Paravirtualized Drivers，PV Drivers）来提高客户机的性能（特

别是I/O性能）。目前，KVM中实现半虚拟化驱动的方式是采用virtio这个Linux上的设备驱动标准框架。

1.QEMU模拟I/O设备的基本原理和优缺点

QEMU以纯软件方式模拟现实世界中的I/O设备的基本过程模型如图5-1所示。

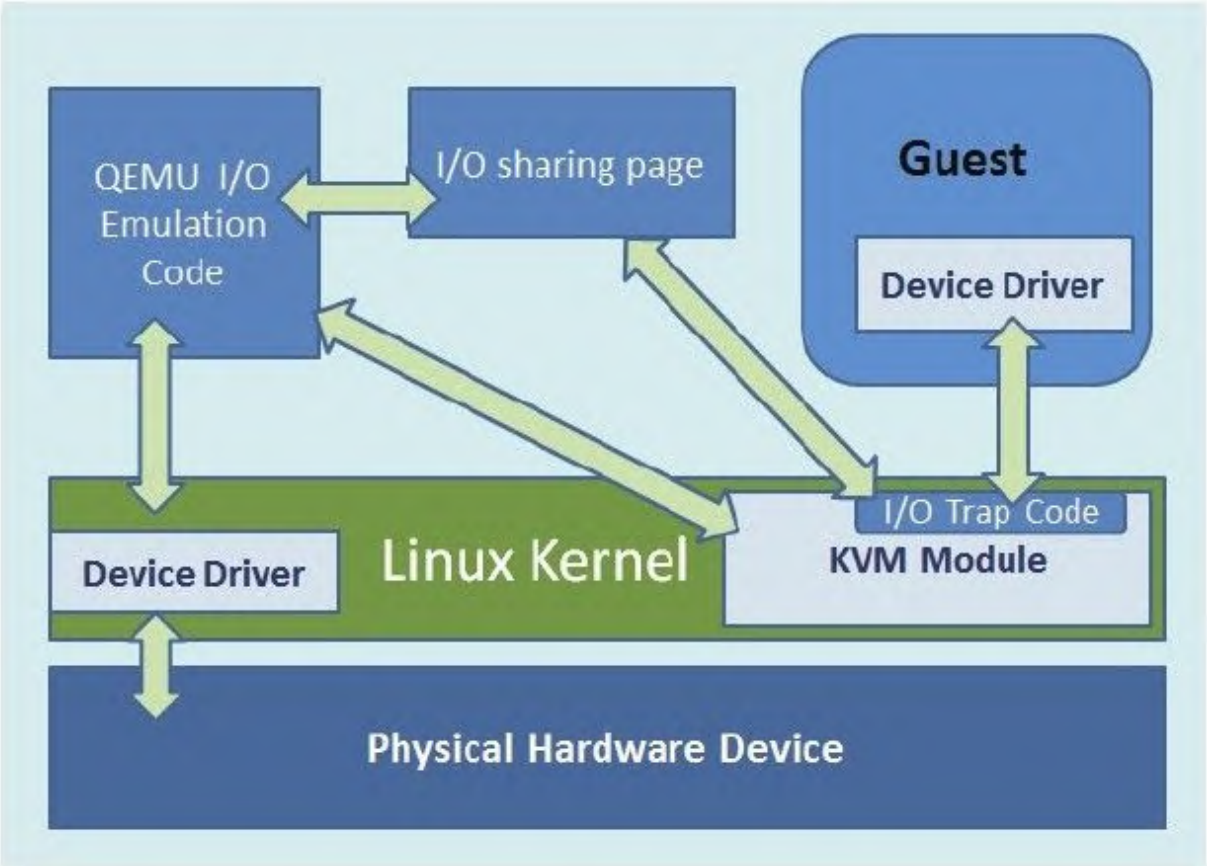


图　5-1　QEMU模拟I/O设备

在使用QEMU模拟I/O的情况下，当客户机中的设备驱动程序（Device Driver）发起I/O操作请求之时，KVM模块（Module）中的

I/O操作捕获代码会拦截这次I/O请求，然后在经过处理后将本次I/O请求的信息存放到I/O共享页（sharing page），并通知用户控件的QEMU程序。QEMU模拟程序获得I/O操作的具体信息之后，交由硬件模拟代码（Emulation Code）来模拟出本次的I/O操作，完成之后，将结果放回到I/O共享页，并通知KVM模块中的I/O操作捕获代码。最后，由KVM模块中的捕获代码读取I/O共享页中的操作结果，并把结果返回到客户机中。当然，在这个操作过程中客户机作为一个QEMU进程在等待I/O时也可能被阻塞。另外，当客户机通过DMA（Direct Memory Access）访问大块I/O之时，QEMU模拟程序将不会把操作结果放到I/O共享页中，而是通过内存映射的方式将结果直接写到客户机的内存中去，然后通过KVM模块告诉客户机DMA操作已经完成。

QEMU模拟I/O设备的方式，其优点是可以通过软件模拟出各种各样的硬件设备，包括一些不常用的或很老很经典的设备（如4.5节中提到的rtl8139网卡），而且该方式不用修改客户机操作系统，就可以实现模拟设备在客户机中正常工作。在KVM客户机中使用这种方式，对于解决手上没有足够设备的软件开发及调试有非常大的好处。而QEMU模拟I/O设备的方式的缺点是，每次I/O操作的路径比较长，有较多的VMEntry、VMExit发生，需要多次上下文切换（context switch），也需要多次数据复制，所以它的性能较差。

2.virtio的基本原理和优缺点

virtio最初由澳大利亚的一个天才级程序员Rusty Russell编写，是一个在Hypervisor之上的抽象API接口，让客户机知道自己运行在虚拟化环境中，进而根据virtio标准与Hypervisor协作，从而在客户机中达到更好的性能（特别是I/O性能）。目前，有不少虚拟机都采用了virtio半虚拟化驱动来提高性能，如KVM和Lguest[1] 。

在QEMU/KVM中，virtio的基本结构框架如图5-2所示。

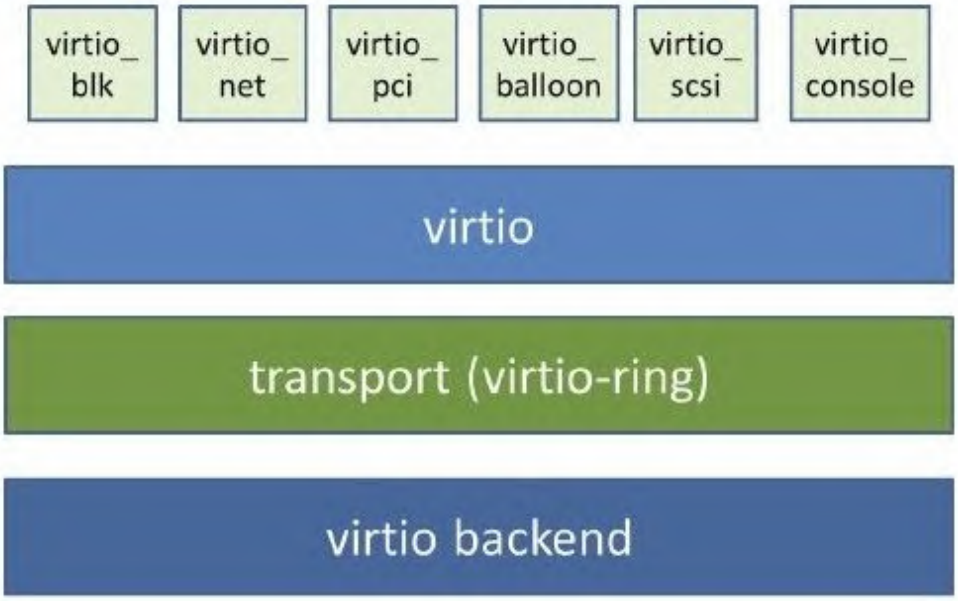


图　5-2　KVM中virtio基本架构

其中前端驱动（frondend，如virtio-blk、virtio-net等）是在客户机中存在的驱动程序模块，而后端处理程序（backend）是在QEMU中实现的[2] 。在前后端驱动之间，还定义了两层来支持客户机与QEMU之间的通信。其中，"virtio"这一层是虚拟队列接口，它在概念上将

前端驱动程序附加到后端处理程序。一个前端驱动程序可以使用0个或多个队列，具体数量取决于需求。例如，virtio-net网络驱动程序使用两个虚拟队列（一个用于接收，另一个用于发送），而virtio-blk块驱动程序仅使用一个虚拟队列。虚拟队列实际上被实现为跨越客户机操作系统和Hypervisor的衔接点，但该衔接点可以通过任意方式实现，前提是客户机操作系统和virtio后端程序都遵循一定的标准，以相互匹配的方式实现它。而virtio-ring实现了环形缓冲区（ring buffer），用于保存前端驱动和后端处理程序执行的信息，并且该环形缓冲区可以一次性保存前端驱动的多次I/O请求，并且交由后端驱动去批量处理，最后实际调用宿主机中设备驱动实现物理上的I/O操作，这样做就可以根据约定实现批量处理而不是客户机中每次I/O请求都需要处理一次，从而提高客户机与Hypervisor信息交换的效率。

virtio半虚拟化驱动的方式，可以获得很好的I/O性能，其性能几乎可以达到和native（即非虚拟化环境中的原生系统）差不多的I/O性能。所以，在使用KVM之时，如果宿主机内核和客户机都支持virtio，一般推荐使用virtio达到更好的性能。当然，virtio也是有缺点的，它需要客户机必须安装特定的virtio驱动使其知道是运行在虚拟化环境中，并且按照virtio的规定格式进行数据传输，不过客户机中可能有一些老的Linux系统不支持virtio，还有一些主流的Windows系统需要安装特定的驱动才支持virtio。不过，较新的一些Linux发行版（如RHEL 6.3、Fedora 17等）默认都将virtio相关驱动编译为模块，可直

接作为客户机使用，而主流Windows系统都有对应的virtio驱动程序可供下载使用。

5.1.2 安装virtio驱动

virtio已经是一个比较稳定成熟的技术了，宿主机中比较新的KVM中都支持它，Linux 2.6.24及以上的Linux内核版本都是支持virtio的。由于virtio的后端处理程序是在位于用户空间的QEMU中实现的，所以，在宿主机中只需要比较新的内核即可，不需要特别地编译与virtio相关的驱动。

客户机需要有特定的virtio驱动的支持，以便客户机处理I/O操作请求时调用virtio驱动而不是其原生的设备驱动程序。下面分别介绍Linux和Windows中virtio相关驱动的安装和使用。

1.Linux中的virtio驱动

在一些流行的Linux发行版（如RHEL 6.x、Ubuntu、Fedora）中，其自带的内核一般都将virtio相关的驱动编译为模块，可以根据需要动态地加载相应的模块。其中，对于RHEL系列来说，RHEL 4.8及以上版本、RHEL 5.3及以上版本、RHEL 6.x的所有版本都默认自动安装有virtio相关的半虚拟化驱动。可以查看内核的配置文件来确定某发行

版是否支持virtio驱动。以RHEL 6.3中的内核配置文件为例，其中与virtio相关的配置有如下几项：

```
CONFIG_VIRTIO=m
CONFIG_VIRTIO_RING=m
CONFIG_VIRTIO_PCI=m
CONFIG_VIRTIO_BALLOON=m
CONFIG_VIRTIO_BLK=m
CONFIG_SCSI_VIRTIO=m
CONFIG_VIRTIO_NET=m
CONFIG_VIRTIO_CONSOLE=m
CONFIG_HW_RANDOM_VIRTIO=m
CONFIG_NET_9P_VIRTIO=m
```

根据这样的配置选项，在编译安装好内核之后，在内核模块中就可以看到virtio.ko、virtio_ring.ko、virtio_net.ko这样的驱动，如下所示：

```
[root@kvm-guest ~]# find /lib/modules/2.6.32-279.el6.x86_64/ -name "virtio*.ko"
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/net/virtio_net.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/virtio/virtio_pci.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/virtio/virtio.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/virtio/virtio_balloon.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/virtio/virtio_ring.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/scsi/virtio_scsi.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/char/hw_random/virtio-rng.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/char/virtio_console.ko
/lib/modules/2.6.32-279.el6.x86_64/kernel/drivers/block/virtio_blk.ko
```

在一个正在使用virtio_net网络前端驱动的KVM客户机中，已自动加载的virtio相关模块如下：

```
[root@kvm-guest ~]# lsmod | grep virtio
virtio_net              16760  0
virtio_pci               7113  0
virtio_ring              7729  2 virtio_net,virtio_pci
virtio                   4890  2 virtio_net,virtio_pci
```

其中virtio、virtio_ring、virtio_pci等驱动程序提供了对virtio API的基本支持，是使用任何virtio前端驱动都必需使用的，而且它们的加载还有一定的顺序，应该按照virtio、virtio_ring、virtio_pci的顺序加载，而virtio_net、virtio_blk这样的驱动可以根据实际需要进行选择性的编译和加载。

2.Windows中的virtio驱动

由于Windows这样的操作系统不是开源操作系统，而且微软也并没有在其操作系统中默认提供virtio相关的驱动，因此需要另外安装特定的驱动程序以便支持virtio。可以通过Linux系统发行版自带软件包安装（如果有该软件包），也可以到网上下载Windows virtio驱动自行安装。

以RHEL 6.3为例，它有一个名为virtio-win的RPM软件包，能为如下版本的Windows提供virtio相关的驱动：

- Windows XP(仅32位版本)
- Windows Server 2003(32位和64位版本)

❑Windows Server 2008(32位和64位版本)

❑Windows 7(32位和64位版本)

可以通过yum来安装virtio-win这个软件包，代码如下：

```
[root@jay-linux kvm_demo]# yum install virto-win
[root@jay-linux kvm_demo]# rpm -q virtio-win
virtio-win-1.5.2-1.el6.noarch
[root@jay-linux kvm_demo]# ls /usr/share/virtio-win/
drivers  virtio-win-1.5.2.iso  virtio-win-1.5.2.vfd  virtio-win.iso  virtio-win.vfd
```

对于RHEL注册用户，也可以到如下的链接下载：

https://rhn.redhat.com/rhn/software/packages/details/Overview.do?pid=602010。

在virtio-win软件包安装完成后，可以看到/usr/share/virtio-win/目录下有一个virtio-win.iso文件，其中包含了所需要的驱动程序。可以将virtio-win.iso文件通过网络共享到Windows客户机中使用，或者通过qemu-kvm命令行的"-cdrom"参数将virtio-win.iso文件作为客户机的光驱。下面以Windows 7客户机为例来介绍在Windows中如何安装virtio驱动。

1）启动Windows 7客户机，将virio-win.iso作为客户机的光驱，命令行操作如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 win7.img -smp 2 -m
2048 -cdrom /usr/share/virtio-win/virtio-win.iso -vnc :0 -usbdevice tablet
```

在Windows 7客户机中打开CD-ROM（virtio-win.iso）可看到其中的目录，如图5-3所示，其中有4个子目录分别表示Windows的4个驱动：Balloon目录是内存气球相关的virtio_balloon驱动，NetKVM目录是网络相关的virtio_net驱动，vioserial目录是控制台相关的virtio_serial驱动，viostor是磁盘块设备存储相关的virtio_scsi驱动。

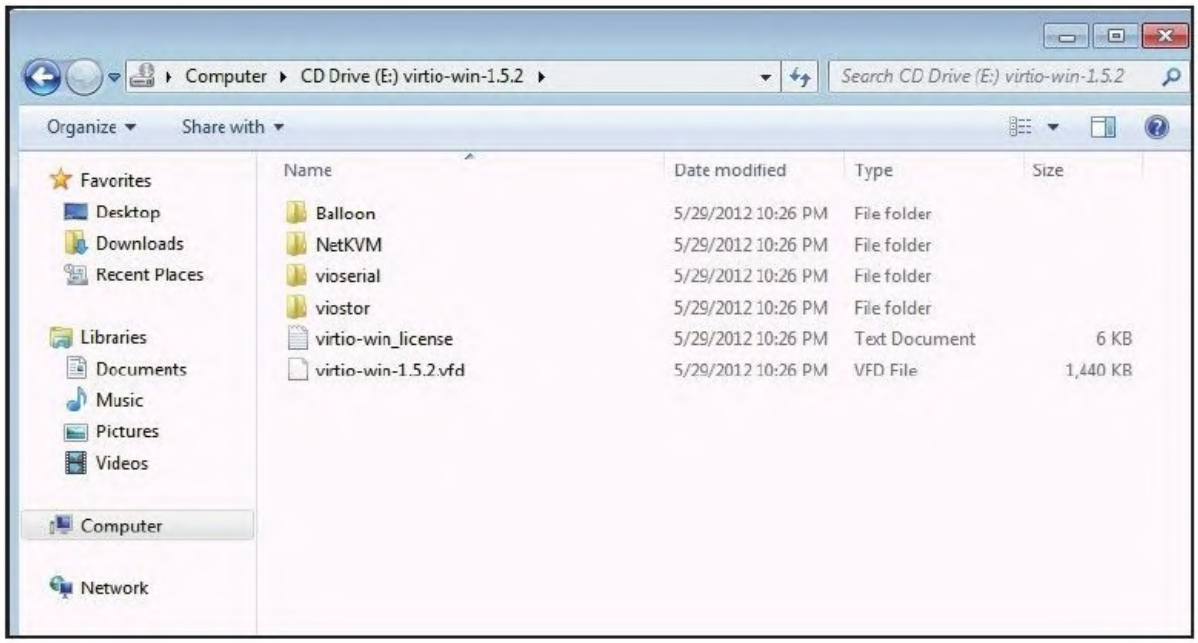


图　5-3　virtio-win中的目录结构

以NetKVM目录为例，其中又包含了各个Windows版本各自的驱动，分别对应Windows XP、Windows 2003、Windows 7、Windows 2008等4个不同的Windows版本。每个Windows版本的目录下又包含"amd64"和"x86"两个版本，分别对应Intel/AMD的x86-64架构和x86-

32架构，即64位的Windows系统应该选择amd64中的驱动，而32位Windows选择x86中的驱动。

在安装驱动前，在Windows（此处示例为英文版）客户机中通过"Computer"右键单击选择"Manage"，在选中的Device Manager（设备管理器）中查看磁盘和网卡驱动，如图5-4所示，磁盘驱动是QEMU模拟的IDE硬盘，网卡是QEMU模拟的rtl8139系列网卡。

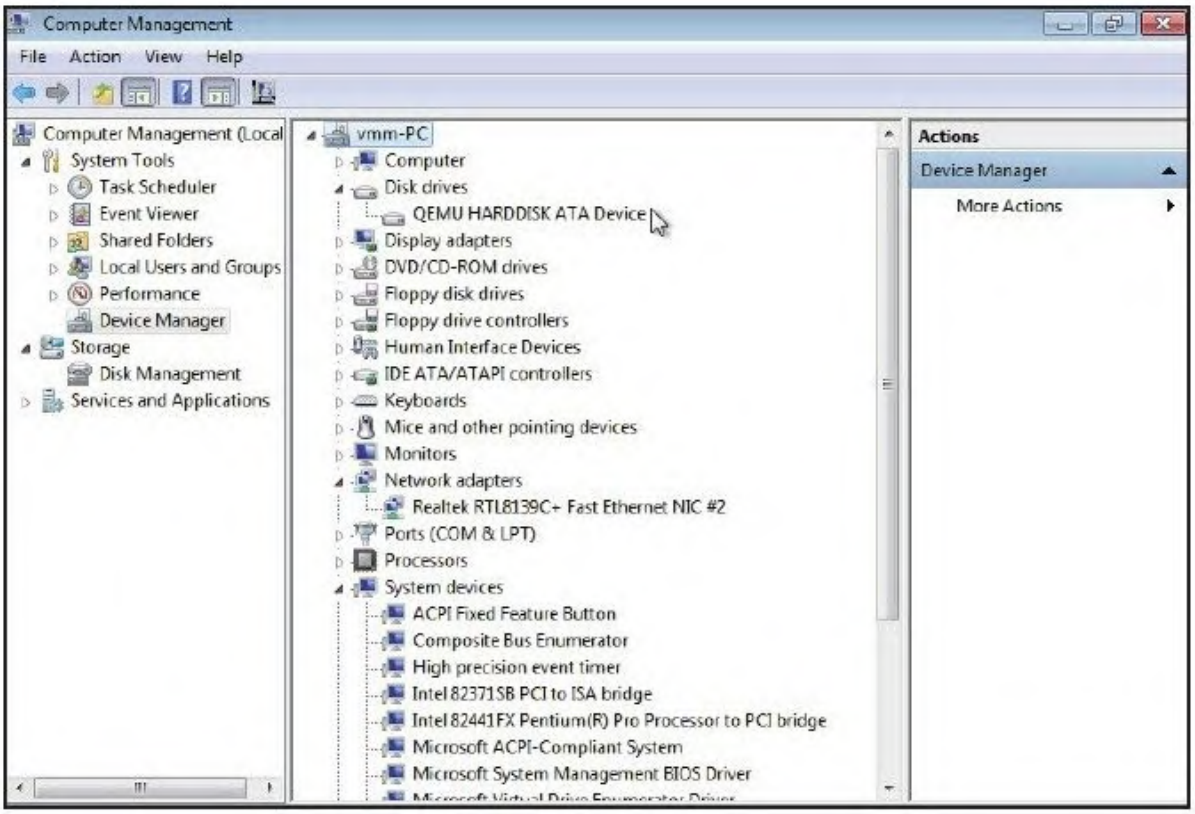


图　5-4　在安装virtio前，Windows客户机中的设备管理器

2）在启动客户机时，向其分配一些virtio相关的设备，在Windows中根据设备安装驱动。

这一步安装的驱动包括virtio_balloon、virtio_net、virtio_serial这3个驱动，而virtio_scsi是磁盘相关的驱动，相关内容将在步骤3）中单独介绍。

qemu-kvm命令行中有如下的参数分别对应这3个驱动：

❑"-balloon virtio"提供了virtio_balloon相关的设备；

❑"-net nic,model=virtio"提供了virtio_net相关的设备；

❑"-device virtio-serial-pci"提供了virtio_serial相关的设备。

启动客户机的命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 win7.img -smp 2 -m
2048 -cdrom /usr/share/virtio-win/virtio-win.iso -vnc :0 -usbdevice tablet
-net nic,model=virtio -net tap -balloon virtio -device virtio-serial-pci
```

在启动后，在Windows客户机的"Device Manager"的"Other devices"项目中会有3个设备没有找到合适的驱动，如图5-5所示，而且网卡处于不可用状态，这是因为使用virtio模型的网卡，没有驱动可用。

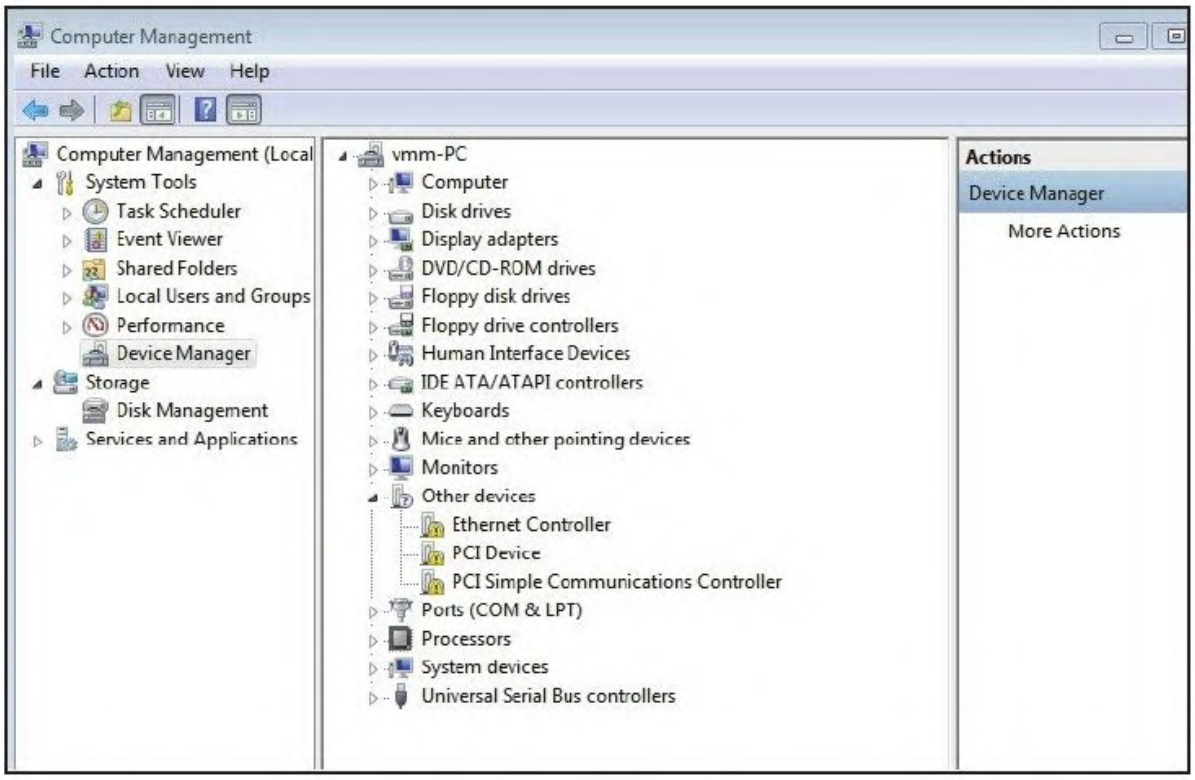


图　5-5　Windows客户机中未安装virtio的驱动程序

在这3个设备中，"Ethernet Controller"是使用virtio的网卡、"PCI Device"是内存balloon的virtio设备、"PCI Simple Communication Controller"是使用virtio的控制台设备。

在未安装virtio驱动的设备上，右键单击，选择"update driver software"更新驱动程序，然后选择"Browser my computer for driver software"，选择在virtio-win.iso中相对应的virtio驱动目录，单击"Next"按钮，就安装对应的virtio驱动了。安装或更新完驱动后，系统会提示"Windows has successfully updated your driver

software″。以virtio的网卡为例，选择驱动程序的操作如图5-6所示。

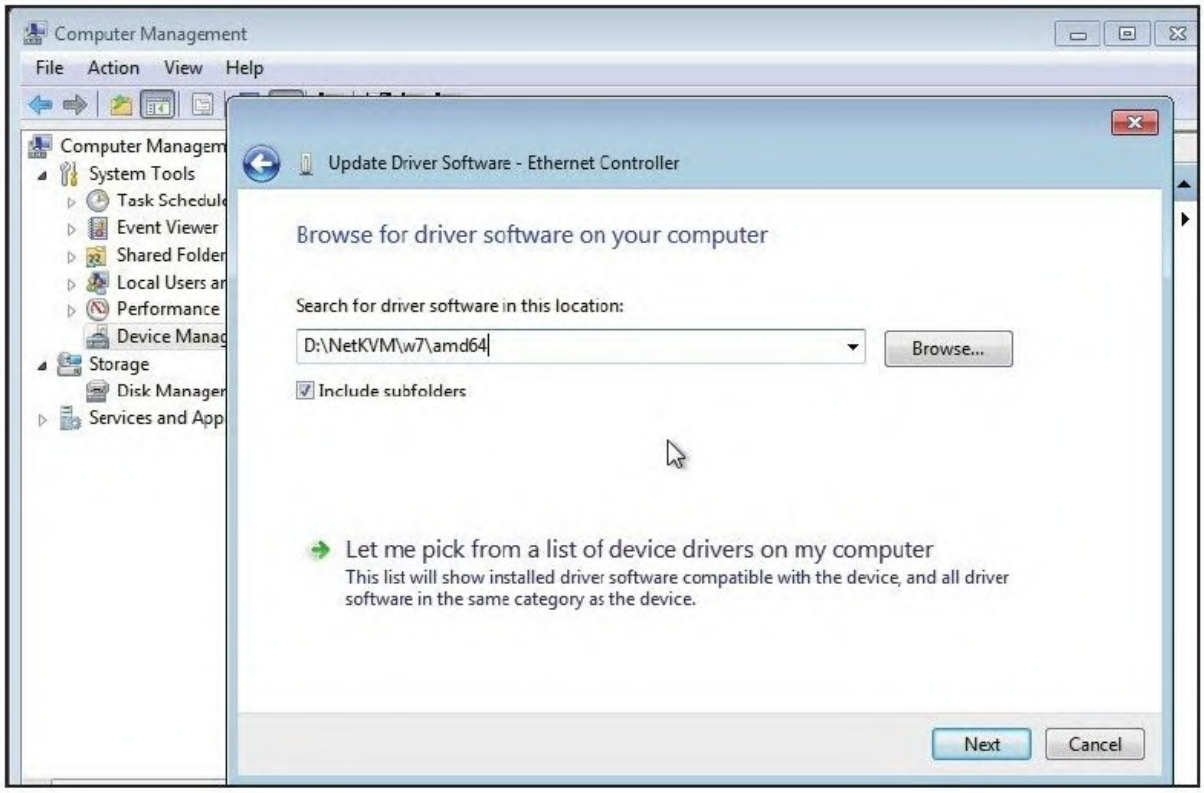


图　5-6　选择网卡相关的virtio驱动程序目录

在依次安装完这3个virtio驱动程序后，在″Device Manager″中的″Network adapter″项目中有了″Red Hat VirtIO Ethernet Adapter″设备，在″System devices″项目中增加了″VirtIO Balloon Driver″和″VirtIO-Serial Driver″这两个设备。网卡驱动安装好后即可生效，网络连接已经正常，而内存的balloon驱动和控制台驱动需要在重启客户机Windows系统后才能生效。

3）安装磁盘virtio驱动程序，其过程与之前的驱动安装略有不同，因为系统中没有virtio_scsi驱动就不能识别硬盘，系统就不能启动，所以可以通过两种方式来安装。其中一种是，在系统启动前从带有virtio驱动的可启动的光盘或软盘将驱动安装好，然后再重启系统从virtio的硬盘进系统。另一种方法是，使用一个非启动硬盘，将其指定为使用virtio驱动，在Windows客户机系统中会发现该非启动硬盘没有合适的驱动，像前面安装其他驱动那样安装即可，然后重启系统将启动硬盘的镜像文件也设置为virtio方式即可使用virtio驱动启动客户机系统。

这里选择第二种方式来演示磁盘virtio驱动的安装，代码如下，其目的是建立一个伪镜像文件，然后将其作为Windows客户机的一个非启动硬盘。

```
[root@jay-linux kvm_demo]#qemu-img create -f qcow2 fake.qcow2 10M
Formatting 'fake.qcow2', fmt=qcow2 size=10485760 encryption=off
cluster_size=65536
[root@jay-linux kvm_demo]# qemu-system-x86_64 win7.img -drive
file=fake.qcow2,if=virtio -smp 2 -m 2048 -cdrom
/usr/share/virtio-win/virtio-win.iso -vnc :0 -usbdevice tablet
```

在Windows客户机的"Device Manager"中会看到"Other devices"项目下有一个没有驱动程序的"SCSI Controller"，如图5-7所示。

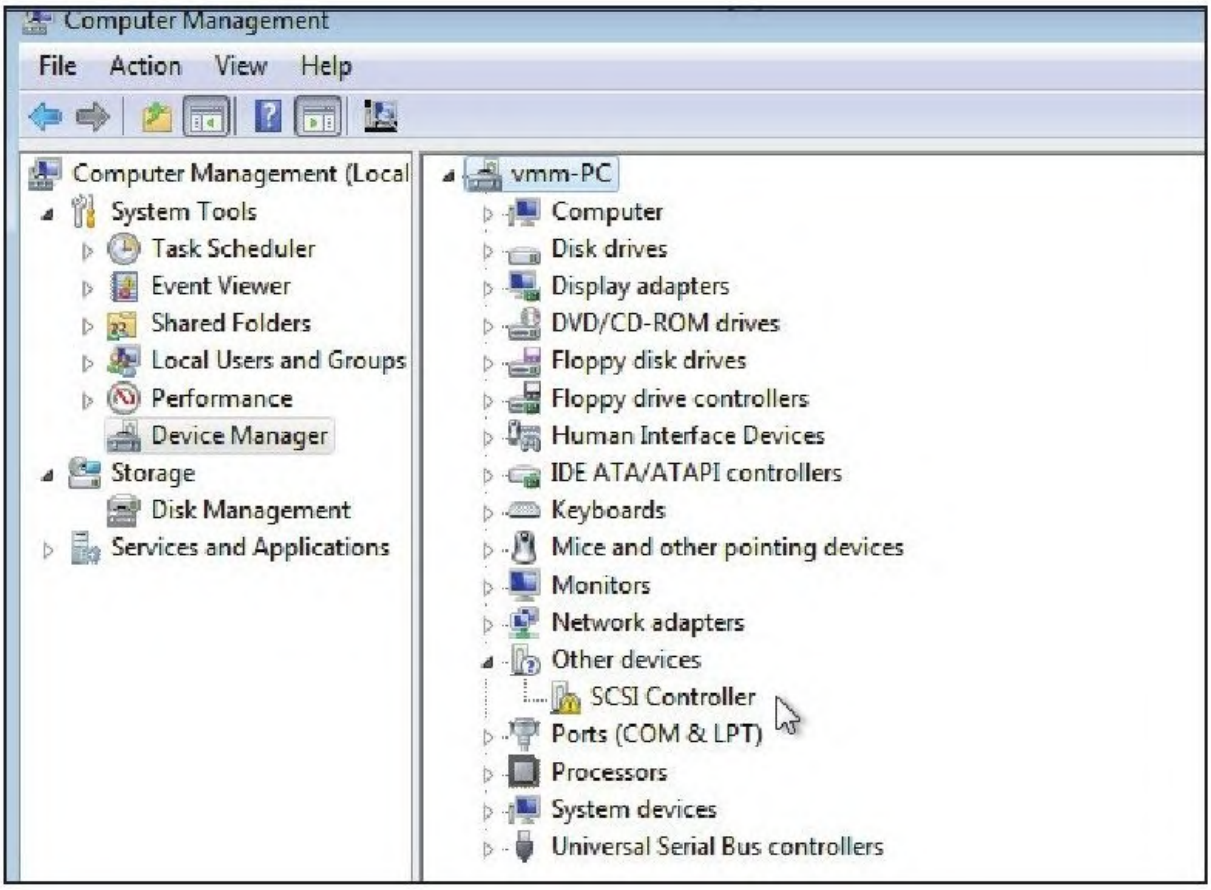


图　5-7　Windows客户机在安装virtio_scsi驱动之前

同前面步骤2）中更新驱动程序一样，选择virtio-win中的viostor目录下的对应驱动进行安装即可。安装完成后，用如下命令重启系统，使用virtio驱动的磁盘镜像。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -drive file=
win7.img,if=virtio -smp 2 -m 2048 -vnc :0 -usbdevice tablet
```

启动系统后，在Windows客户机的"Device Manager"的"Disk drives"项目下可看到"Red Hat VirtIO SCSI Disk

Device"，"Storage Controller"项目下有"Red Hat VirtIO SCSI Controller"即表明正在使用virtio_scsi驱动。

4）在安装好了这4个virtio驱动后，用下面的命令行重新启动这个客户机，使这4个virtio驱动全部处于使用状态。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -drive file=win7.img,
if=virtio -smp 2 -m 2048 -net nic,model=virtio -net tap -balloon virtio
-device virtio-serial-pci -vnc :0 -usbdevice tablet
```

在"Device Manager"中查看已经安装并使用的virtio驱动，如图5-8所示。

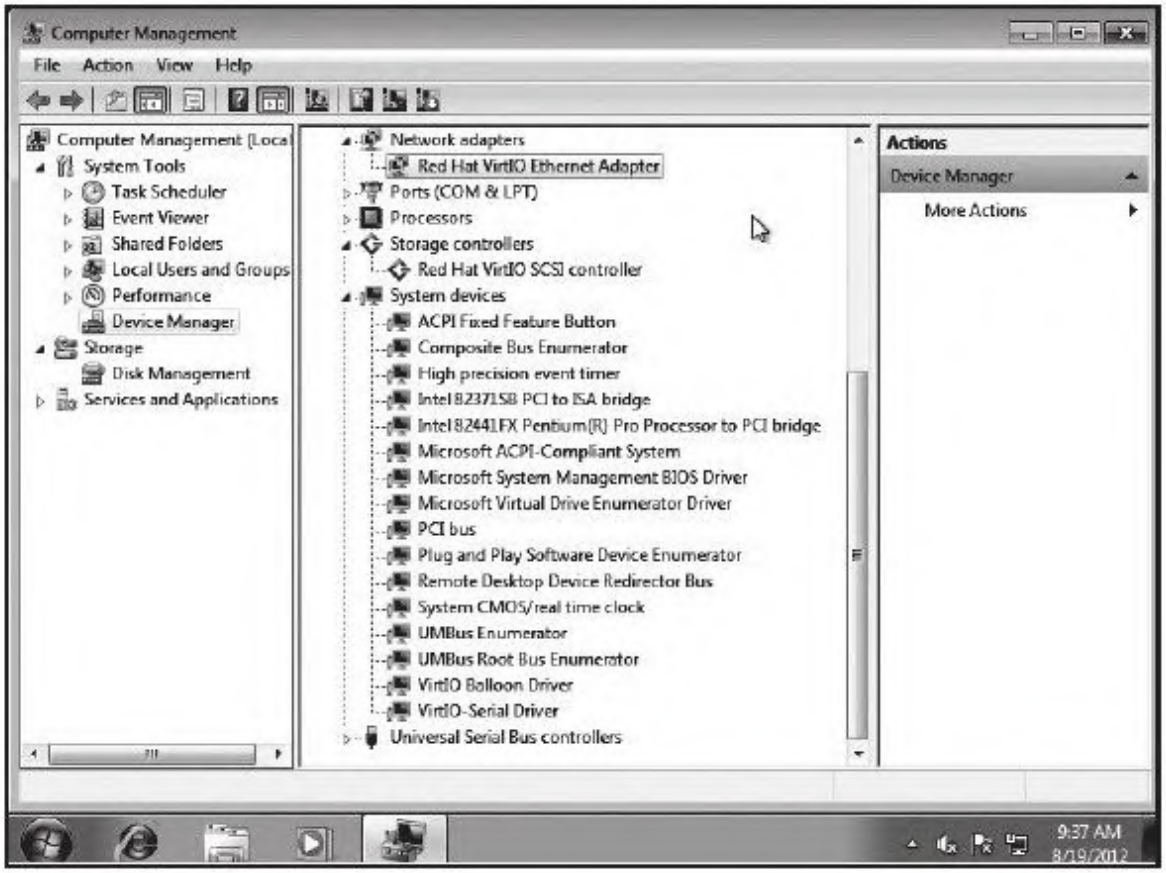


图　5-8　在Windows的“设备管理器”中查看virtio驱动

除了RHEL提供的virtio-win ISO文件之外，在github中也有最新KVM中Windows客户机的virtio驱动源代码仓库。访问https://github.com/YanVugenfirer/kvm-guest-drivers-windows，可以通过下载、编译得到所需要的驱动。当然编译过程可能比较复杂，可进行阅读代码仓库中的一些编译说明进行学习。

在64位的Windows系统中，从Windows Vista开始（如Windows 7、Windows 2008等），所安装的驱动就要求有数字签名。如果使用的发行版并没有提供Windows virtio驱动的二进制文件，或者没有对应的数字签名，则可以选择从Fedora项目中下载二进制ISO文件，它们都进行了数字签名，并且通过了在Windows系统上的测试。从http://alt.fedoraproject.org/pub/alt/virtio-win/latest/images/bin/可以下载Fedora项目提供的Windows virtio驱动，之后的安装过程与前面讲述的过程完全一致。

另外，关于KVM中Windows客户机的驱动程序，可以查看本章末尾的“参考阅读”部分了解更多信息。

5.1.3　使用virtio_balloon

1.ballooning简介

通常来说，要改变客户机占用的宿主机内存，要先关闭客户机，修改启动时的内存配置，然后重启客户机才能实现。而内存的ballooning（气球）技术可以在客户机运行时动态地调整它所占用的宿主机内存资源，而不需要关闭客户机。

ballooning技术形象地在客户机占用的内存中引入气球（balloon）的概念。气球中的内存是可以供宿主机使用的（但不能被客户机访问或使用），所以，当宿主机内存紧张，空余内存不多时，可以请求客户机回收利用已分配给客户机的部分内存，客户机就会释放其空闲的内存，此时若客户机空闲内存不足，可能还会回收部分使用中的内存，可能会将部分内存换出到客户机的交换分区（swap）中，从而使内存气球充气膨胀，进而使宿主机回收气球中的内存用于其他进程（或其他客户机）。反之，当客户机中内存不足时，也可以让客户机的内存气球压缩，释放出内存气球中的部分内存，让客户机使用更多的内存。

目前很多虚拟机，如KVM、Xen、VMware等，都对ballooning技术提供支持。关于内存balloon的概念，其示意图如图5-9所示。

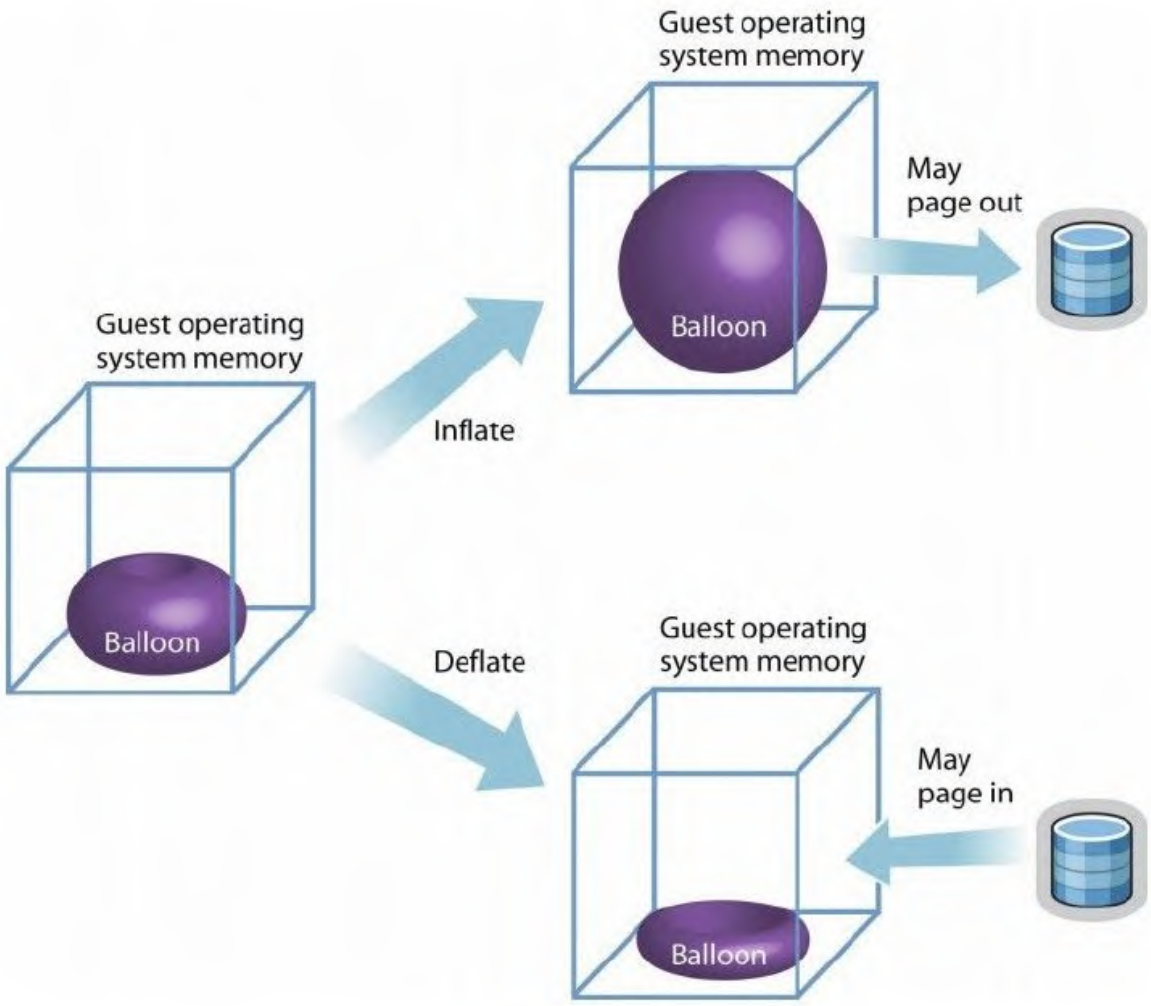


图　5-9　内存balloon的概念

2.KVM中ballooning的原理及优劣势

KVM中ballooning的工作过程主要有如下几步：

1）Hypervisor（即KVM）发送请求到客户机操作系统让其归还一定数量的内存给Hypervisor。

2）客户机操作系统中的virtio_balloon驱动接收到Hypervisor的请求。

3）virtio_balloon驱动使客户机的内存气球膨胀，气球中的内存就不能被客户机访问。如果此时客户机中内存剩余量不多（如某应用程序绑定/申请了大量的内存），并且不能让内存气球膨胀到足够大以满足Hypervisor的请求，那么virtio_balloon驱动也会尽可能多地提供内存使气球膨胀，尽量去满足Hypervisor的请求中的内存数量（即使不一定能完全满足）。

4）客户机操作系统归还气球中的内存给Hypervisor。

5）Hypervisor可以将从气球中得来的内存分配到任何需要的地方。

6）即使从气球中得到的内存没有处于使用中，Hypervisor也可以将内存返还到客户机中，这个过程为：Hypervisor发请求到客户机的virtio_balloon驱动；这个请求使客户机操作系统压缩内存气球；在气球中的内存被释放出来，重新由客户机访问和使用。

ballooning在节约内存和灵活分配内存方面有明显的优势，其好处有如下3点。

1）因为ballooning能够被控制和监控，所以能够潜在地节约大量的内存。它不同于内存页共享技术（KSM是内核自发完成的，不可控），客户机系统的内存只有在通过命令行调整balloon时才会随之改变，所以能够监控系统内存并验证ballooning引起的变化。

2）ballooning对内存的调节很灵活，既可以精细地请求少量内存，又可以粗犷地请求大量的内存。

3）Hypervisor使用ballooning让客户机归还部分内存，从而缓解其内存压力。而且从气球中回收的内存也不要求一定要被分配给另外某个进程（或另外的客户机）。

从另一方面来说，KVM中ballooning的使用不方便、不完善的地方也是存在的，其缺点如下。

1）ballooning需要客户机操作系统加载virtio_balloon驱动，然而并非每个客户机系统都有该驱动（如Windows需要自己安装该驱动）。

2）如果有大量内存需要从客户机系统中回收，那么ballooning可能会降低客户机操作系统运行的性能。一方面，内存的减少可能会让客户机中作为磁盘数据缓存的内存被放到气球中，从而使客户机中的磁盘I/O访问增加；另一方面，如果处理机制不够好，也可能让客户机中正在运行的进程由于内存不足而执行失败。

3）目前没有比较方便的、自动化的机制来管理ballooning，一般都采用在QEMU monitor中执行balloon命令来实现ballooning。没有对客户机的有效监控，没有自动化的ballooning机制，这可能会使在生产环境中实现大规模自动化部署不是很方便。

4）内存的动态增加或减少，可能会使内存被过度碎片化，从而降低内存使用时的性能。另外，内存的变化会影响到客户机内核对内存使用的优化，比如，内核起初根据目前状态对内存的分配采取了某个策略，而后突然由于balloon的原因使可用内存减少了很多，这时起初的内存策略就可能不是太优化了。

3.KVM中ballooning使用示例

KVM中的ballooning是通过宿主机和客户机协同实现的，在宿主机中应该使用Linux 2.6.27及以上版本的Linux内核（包括KVM模块），使用较新的qemu-kvm（如0.13版本以上），在客户机中也使用Linux 2.6.27及以上版本的Linux内核且将"CONFIG_VIRTIO_BALLOON"配置为模块或编译到内核。在很多Linux发行版中都已经配置有"CONFIG_VIRTIO_BALLOON=m"，所以用较新的Linux作为客户机系统，一般不需要额外配置virtio_balloon驱动，使用默认内核配置即可。

在QEMU命令行中可用"-balloon virtio"参数来分配balloon设备给客户机使其调用virtio_balloon驱动来工作，而默认值为没有分配balloon设备（与"-balloon none"效果相同）。

-balloon virtio[,addr=addr]#使用virtio balloon设备，addr可配置客户机中该设备的PCI地址

在QEMU monitor中，有两个命令用于查看和设置客户机内存的大小。

```
(qemu) info balloon    # 查看客户机内存占用量（balloon 信息）
(qemu) balloon num     # 设置客户机内存占用量为 numMB
```

下面介绍在KVM中使用ballooning的操作步骤。

1）QEMU启动客户机时分配balloon设备，命令行如下。也可以使用较新的"-device"的统一参数来分配balloon设备，如"-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x4"。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 2048-balloon virtio
```

2）在启动后的客户机中查看balloon设备及内存使用情况，命令行如下：

```
[root@kvm-guest ~]# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd.
RTL-8139/8139C/8139C+ (rev 20)
00:04.0 Unclassified device [00ff]: Red Hat, Inc Virtio memory balloon
[root@kvm-guest ~]# grep VIRTIO_BALLOON \
/boot/config-2.6.32-279.el6.x86_64
CONFIG_VIRTIO_BALLOON=m
[root@kvm-guest ~]# lsmod | grep virtio
virtio_balloon          4856  0
virtio_pci              7113  0
virtio_ring             7729  2 virtio_balloon,virtio_pci
virtio                  4890  2 virtio_balloon,virtio_pci
[root@kvm-guest ~]# lspci -s 00:04.0 -v
00:04.0 Unclassified device [00ff]: Red Hat, Inc Virtio memory balloon
        Subsystem: Red Hat, Inc Device 0005
        Physical Slot: 4
        Flags: fast devsel, IRQ 10
        I/O ports at c100 [size=32]
        Kernel driver in use: virtio-pci
        Kernel modules: virtio_pci
[root@kvm-guest ~]# free -m
             total       used       free     shared    buffers     cached
Mem:          1877        166       1711          0         21         59
-/+ buffers/cache:         85       1792
Swap:          508          0        508
```

根据上面输出可知，客户机中已经加载virtio_balloon模块，有一个名为"Red Hat,Inc Virtio memory balloon"的PCI设备，它使用了virtio_pci驱动。如果是Windows客户机，则可以在“设备管理器”看到使用virtio balloon设备。

3）在QEMU monitor中查看和改变客户机占用的内存，命令如下：

```
(qemu) info balloon
```

```
balloon: actual=2048
(qemu) balloon 512
(qemu) info balloon
balloon: actual=512
```

如果没有使用balloon设备，则在monitor中使用"info balloon"命令查看会得到"Device 'balloon' has not been activated"的警告提示。而"balloon 512"命令将客户机内存设置为512MB。

4）设置了客户机内存为512 MB后，再到客户机中检查，命令如下：

```
[root@kvm-guest ~]# free -m
             total       used       free     shared    buffers     cached
Mem:           341        166        175          0         21         59
-/+ buffers/cache:         85        256
Swap:          508          0        508
```

对于Windows客户机（如Windows 7），当balloon使其可用内存从2GB降低到512MB时，在其“任务管理器”中看到的内存总数依然是2GB，但是看到它的内存已使用量会增大1536MB（例如从原来使用量350MB变为1886MB），这里占用的1536MB内存正是balloon设备占用的，Windows客户机系统的其他程序已不能使用这1536 MB内存，这时宿主机系统可以再次分配这里的1536MB内存用于其他用途。

另外，值得注意的是，当通过"balloon"命令使客户机内存增加时，其最大值不能超过QEMU命令行启动时设置的内存，例如在命令行中将内存设置为2048MB，如果在Monitor中执行"balloon 4096"，则设置的4096MB内存不会生效，该值将会被设置为启动命令行中的最大值（即2048MB）。

4.通过ballooning过载使用内存

在4.3.4节“内存过载使用”中提到，内存过载使用主要有三种方式：swapping、ballooning和page sharing。在多个客户机运行时动态地调整其内存容量，ballooning是一种让内存过载使用得非常有效的机制。使用ballooning可以根据宿主机中对内存的需求，通过"balloon"命令调整客户机内存占用量，从而实现内存的过载使用。

在实际环境中，客户机系统的资源的平均使用率一般并不高，通常是一段时间负载较重，一段时间负载较轻。可以在一个物理宿主机上启动多个客户机，通过ballooning的支持，在某些客户机负载较轻时减少其内存使用，将内存分配给此时负载较重的客户机。例如，在一个物理内存为8GB的宿主机上，可以在一开始就分别启动6个内存为2GB的客户机（A、B、C、D、E、F等6个），根据平时对各个客户机中资源使用情况的统计可知，在当前一段时间内，A、B、C的负载很轻，就可以通过ballooning降低其内存为512 MB，而D、E、F的内存保持2 GB不变。内存分配的简单计算为：

512MB×3+2GB×3+512MB（用于宿主机中其他进程）=8GB

而在某些其他时间段，当A、B、C等客户机负载较大时，也可以增加它们的内存量（同时减少D、E、F的内存量）。这样就在8GB物理内存上运行了看似需要大于12GB内存才能运行的6个2GB内存的客户机，从而较好地实现了内存的过载使用。

如果客户机中有virtio_balloon驱动，则使用ballooning来实现内存过载使用是非常方便的。而前面提到“在QEMU monitor中使用balloon命令改变内存的操作执行起来不方便”的问题，如果使用第6章将介绍的libvirt工具来使用KVM，则对ballooning的操作会比较方便，在libvirt工具的"virsh"管理程序中就有"setmem"这个命令来动态更改客户机的可用内存容量，该方式的完整命令为"virsh setmem＜domain-id or domain-name＞＜Amount of memory in KB＞"。

5.1.4 使用virtio_net

1.配置和使用virtio_net

在选择KVM中的网络设备时，一般来说优先选择半虚拟化的网络设备而不是纯软件模拟的设备。使用virtio_net半虚拟化驱动，可以提高网络吞吐量（thoughput）和降低网络延迟（latency），从而让客

户机中网络达到几乎和非虚拟化系统中使用原生网卡的网络差不多的性能。

要使用virtio_net，需要两部分的支持，宿主机中的QEMU工具的支持和客户机中virtio_net驱动的支持。较新的qemu-kvm都对virtio网卡设备提供支持，且较新的流行Linux发行版中都已经将virtio_net作为模块编译到系统之中了，所以使用起来还是比较方便的。

可以通过如下步骤来使用virtio_net。

1)检查QEMU是否支持virtio类型的网卡。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -net nic,model=?
qemu: Supported NIC models: ne2k_pci,i82551,i82557b,i82559er,
rtl8139,e1000,pcnet,virtio
```

从输出信息中支持网卡类型可知，当前qemu-kvm支持virtio网卡模型。

2)启动客户机时，指定分配virtio网卡设备。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic,model=virtio,macaddr=00:16:3e:22:22:22 -net tap
VNC server running on '::1:5900'
```

3)在客户机中查看virtio网卡的使用情况。

```
[root@kvm-guest ~]# grep VIRTIO_ /boot/config-2.6.32-279.el6.x86_64
CONFIG_VIRTIO_BLK=m
CONFIG_VIRTIO_NET=m
CONFIG_VIRTIO_CONSOLE=m
CONFIG_VIRTIO_RING=m
CONFIG_VIRTIO_PCI=m
CONFIG_VIRTIO_BALLOON=m
[root@kvm-guest ~]# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
```

```
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Ethernet controller: Red Hat, Inc Virtio network device
[root@kvm-guest ~]# lspci -vv -s 00:03.0
00:03.0 Ethernet controller: Red Hat, Inc Virtio network device
        Subsystem: Red Hat, Inc Device 0001
        Physical Slot: 3
        Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV-
VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr-
DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Interrupt: pin A routed to IRQ 11
        Region 0: I/O ports at c000 [size=32]
        Region 1: Memory at febf1000 (32-bit, non-prefetchable) [size=4K]
        Expansion ROM at febe0000 [disabled] [size=64K]
        Capabilities: [40] MSI-X: Enable+ Count=3 Masked-
                Vector table: BAR=1 offset=00000000
                PBA: BAR=1 offset=00000800
        Kernel driver in use: virtio-pci
        Kernel modules: virtio_pci
[root@kvm-guest ~]# lsmod | grep virtio
virtio_net             16760  0
virtio_pci              7113  0
virtio_ring             7729  2 virtio_net,virtio_pci
virtio                  4890  2 virtio_net,virtio_pci
[root@kvm-guest ~]# ethtool -i eth1
driver: virtio_net
version:
firmware-version:
bus-info: virtio0
[root@kvm-guest ~]# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:16:3E:22:22:22
          inet addr:192.168.156.200  Bcast:192.168.255.255
Mask:255.255.0.0
          inet6 addr: fe80::216:3eff:fe22:2222/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:500 errors:0 dropped:0 overruns:0 frame:0
          TX packets:103 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:65854 (64.3 KiB)  TX bytes:19057 (18.6 KiB)
[root@kvm-guest ~]# ping 192.168.199.98 -c 1
PING 192.168.199.98 (192.168.199.98) 56(84) bytes of data.
64 bytes from 192.168.199.98: icmp_seq=1 ttl=64 time=0.313 ms

--- 192.168.199.98 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.313/0.313/0.313/0.000 ms
```

根据上面的输出信息可知，网络接口eth1使用了virtio_net驱动，并且当前网络连接正常工作。如果启动Windows客户机使用virtio类型的网卡，则在Windows客户机的“设备管理器”中看到的一个名为"Red Hat VirtIO Ethernet Adapter"的设备即是客户机中的网卡。

2.宿主机中TSO和GSO的设置

据Redhat文档[3] 的介绍，如果在使用半虚拟化网络驱动（即virtio_net）时得到的依然是较低的性能，可以检查宿主机系统中对GSO和TSO[4] 特性的设置。关闭GSO和TSO可以使半虚拟化网络驱动的性能更加优化。

通过如下命令可以检查宿主机中GSO和TSO的设置，其中eth0是建立bridge供客户机使用的网络接口。

```
[root@jay-linux ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.001320fb4fa8       yes             eth0
                                                        tap0
[root@jay-linux ~]# ethtool -k eth0
Offload parameters for eth0:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp- segmentation-offload: on           # 这个就是 TSO 的状态
udp-fragmentation-offload: off
generic-segmentation-offload: on        # 这是 GSO 的状态
generic-receive-offload: on
large-receive-offload: off
```

通过如下命令可以关闭宿主机的GSO和TSO功能。

```
[root@jay-linux ~]# ethtool -K eth0 gso off
[root@jay-linux ~]# ethtool -K eth0 tso off
[root@jay-linux ~]# ethtool -k eth0
Offload parameters for eth0:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp-segmentation-offload: off
udp-fragmentation-offload: off
generic-segmentation-offload: off
generic-receive-offload: on
large-receive-offload: off
```

3.用vhost_net后端驱动

前面提到virtio在宿主机中的后端处理程序（backend）一般是由用户空间的QEMU提供的，然而，如果对于网络IO请求的后端处理能够在在内核空间来完成，则效率会更高，会提高网络吞吐量和减少网络延迟。在比较新的内核中有一个叫做"vhost-net"的驱动模块，它作为一个内核级别的后端处理程序，将virtio-net的后端处理任务放到内核空间中执行，从而提高效率。

在4.5.2节“使用网桥模式”介绍网络配置时，提到有几个选项和virtio相关的，这里也介绍一下。

```
-net tap,[,vnet_hdr=on|off][,vhost=on|off][,vhostfd=h][,vhostforce=on|off]
```

vnet_hdr=on|off，设置是否打开TAP设备的"IFF_VNET_HDR"标识："vnet_hdr=off"表示关闭这个标识，而"vnet_hdr=on"表示强制开启这个标识，如果没有这个标识的支持，则会触发错误。IFF_VNET_HDR是tun/tap的一个标识，打开这个标识则允许发送或接受大数据包时仅做部分的校验和检查。打开这个标识，还可以提高virtio_net驱动的吞吐量。

vhost=on|off，设置是否开启vhost-net这个内核空间的后端处理驱动，它只对使用MIS-X[5] 中断方式的virtio客户机有效。

vhostforce=on|off，设置是否强制使用vhost作为非MSI-X中断方式的virtio客户机的后端处理程序。

vhostfs=h，设置去连接一个已经打开的vhost网络设备。

用如下命令行启动一个客户机，就可以在客户机中使用virtio-net作为前端驱动程序，而在后端处理程序时使用vhost-net（当然需要当前宿主机内核支持vhost-net模块）。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic,model=virtio,macaddr=00:16:3e:22:22:22 -net
tap,vnet_hdr=on,vhost=on
VNC server running on '::1:5900'
```

启动客户机后，检查客户机网络，应该是可以正常连接的。

而在宿主机中可以查看vhost-net的使用情况，如下：

```
[root@jay-linux kvm_demo]# grep VHOST /boot/config-3.5.0
CONFIG_VHOST_NET=m
[root@jay-linux kvm_demo]# lsmod | grep vhost
vhost_net              17161  1
tun                    13220  3 vhost_net
[root@jay-linux kvm_demo]# rmmod vhost-net
ERROR: Module vhost_net is in use
```

可见，宿主机中内核将vhost-net编译为module，此时vhost-net模块处于使用中状态（试图删除它时会报告“在使用中”的错误）。

一般来说，使用vhost-net作为后端处理驱动可以提高网络的性能。不过，对于一些使用vhost-net作为后端的网络负载类型，可能使其性能不升反降。特别是从宿主机到其客户机之间的UDP流量，如果客户机处理接受数据的速度比宿主机发送的速度要慢，这时就容易出现性能下降。在这种情况下，使用vhost-net将会使UDP socket的接受缓冲区更快地溢出，从而导致更多的数据包丢失。因此在这种情况下不使用vhost-net，让传输速度稍微慢一点，反而会提高整体的性能。

使用qemu-kvm命令行时，加上"vhost=off"（或不添加任何vhost选项）就会不使用vhost-net作为后端驱动，而在使用libvirt时，如果要选QEMU作为后端驱动，则需要对客户机的XML配置文件中的网络配置部分进行如下的配置，指定后端驱动的名称为"qemu"（而不是"vhost"）。

```
<interface type="network">
    ...
    <model type="virtio"/>
    <driver name="qemu"/>
    ...
</interface>
```

5.1.5 使用virtio_blk

virtio_blk驱动使用virtio API为客户机提供了一个高效访问块设备I/O的方法。在QEMU/KVM中对块设备使用virtio，需要在两方面进行配置：客户机中的前端驱动模块virtio_blk和宿主机中的QEMU提供后端处理程序。目前比较流行的Linux发行版都将virtio_blk编译为内核模块，可以作为客户机直接使用virtio_blk，而Windows中的virtio驱动的安装方法已在5.1.2节中做了介绍。并且较新的qemu-kvm都是支持virtio block设备的后端处理程序的。

启动一个使用virtio_blk作为磁盘驱动的客户机，其qemu-kvm命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -smp 2 -m 1024 -net
nic-net tap-drive file=rhel6u3.img,if=virtio
VNC server running on '::1:5900'
```

在客户机中，查看virtio_blk生效的情况如下所示：

```
[root@kvm-guest ~]#grep VIRTIO_BLK \
/boot/config-2.6.32-279.el6.x86_64
CONFIG_VIRTIO_BLK=m
[root@kvm-guest ~]#lsmod | grep virtio
virtio_blk              7292  3
virtio_pci              7113  0
virtio_ring             7729  2 virtio_blk,virtio_pci
virtio                  4890  2 virtio_blk,virtio_pci
[root@kvm-guest ~]# lspci | grep -i block
00:04.0 SCSI storage controller: Red Hat, Inc Virtio block device
[root@kvm-guest ~]# lspci -vv -s 00:04.0
00:04.0 SCSI storage controller: Red Hat, Inc Virtio block device
        Subsystem: Red Hat, Inc Device 0002
        Physical Slot: 4
        Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV-
VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr-
DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Interrupt: pin A routed to IRQ 11

        Region 0: I/O ports at c100 [size=64]
        Region 1: Memory at febf2000 (32-bit, non-prefetchable) [size=4K]
        Capabilities: [40] MSI-X: Enable+ Count=2 Masked-
                Vector table: BAR=1 offset=00000000
                PBA: BAR=1 offset=00000800
        Kernel driver in use: virtio-pci
        Kernel modules: virtio_pci
[root@kvm-guest ~]#fdisk -l
Disk /dev/vda: 8589 MB, 8589934592 bytes
16 heads, 63 sectors/track, 16644 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000726b0

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1   *           3       14826     7471104   83  Linux
/dev/vda2           14826       16645      916480   82  Linux swap / Solaris
```

可知客户机中已经加载了virtio_blk等驱动，QEMU提供的virtio块设备使用virtio_blk驱动（以上查询结果中显示为virtio_pci，因为它是任意virtio的PCI设备的一个基础的、必备的驱动）。使用

virtio_blk驱动的磁盘显示为"/dev/vda"，这不同于IDE硬盘的"/dev/hda"或SATA硬盘的"/dev/sda"这样的显示标识。

而"/dev/vd*"这样的磁盘设备名称可能会导致从前分配在磁盘上的swap分区失效，因为有些客户机系统中记录文件系统信息的"/etc/fstab"文件中有类似如下的对swap分区的写法。

```
/dev/sda2  swap swap defaults 0 0
```

或

```
/dev/hda2  swap swap defaults 0 0
```

原因就是"/dev/vda2"这样的磁盘分区名称未被正确识别，解决这个问题的方法就很简单了，只需要修改它为如下形式并保存"/etc/fstab"文件，然后重启客户机系统即可。

```
/dev/vda2  swap swap defaults 0 0
```

如果启动的是已安装virtio驱动的Windows客户机，那么在客户机的“设备管理器”中的“存储控制器”中看到的是正在使用"Red Hat VirtIO SCSI Controller"设备作为磁盘。

5.1.6 kvm_clock配置

在保持时间的准确性方面，虚拟化环境似乎天生就面临几个难题和挑战。由于在虚拟机中的中断并非真正的中断，而是通过宿主机向客户机注入的虚拟中断，因此中断并不总是能同时且立即传递给一个客户机的所有虚拟CPU（vCPU）。在需要向客户机注入中断时，宿主机的物理CPU可能正在执行其他客户机的vCPU或在运行其他一些非QEMU进程，这就是说中断需要的时间精确性有可能得不到保障。

而在现实使用场景中，如果客户机中时间不准确，就可能导致一些程序和一些用户场景在正确性上遇到麻烦。这类程序或场景，一般是Web应用程序或基于网络的应用场景，如Web应用中的Cookie或Session有效期计算、虚拟机的动态迁移（Live Migration），以及其他一些依赖于时间戳的应用等。

而QEMU/KVM通过提供一个半虚拟化的时钟，即kvm_clock，为客户机提供精确的System time和Wall time，从而避免客户机中时间不准确的问题。kvm_clock使用较新的硬件（如Intel SandyBridge平台）提供的支持，如不变的时钟计数器（Constant Time Stamp Counter）。Constant TSC的计数的频率，即使当前CPU核心改变频率（如使用了一些省电策略），也能保持恒定不变。CPU有一个不变的constant TSC频率是将TSC作为KVM客户机时钟的必要条件。

物理CPU对constant TSC的支持，可以查看宿主机中CPU信息的标识，有"constant_tsc"的即是支持constant TSC的，如下所示（信息

来源于运行在SandyBridge硬件平台的系统）。

```
[root@jay-linux kvm_demo]# grep constant_tsc /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid
```

一般来说，在较新的Linux发行版的内核中都已经将kvm_clock相关的支持编译进去了，可以查看如下的内核配置选项：

```
[root@kvm-guest ~]# grep PARAVIRT_GUEST \
/boot/config-2.6.32-279.el6.x86_64
CONFIG_PARAVIRT_GUEST=y
[root@kvm-guest ~]# grep KVM_CLOCK \
/boot/config-2.6.32-279.el6.x86_64
CONFIG_KVM_CLOCK=y
```

而在用QEMU命令行启动客户机时，已经会默认让其使用kvm_clock作为时钟来源。用最普通的命令启动一个Linux客户机，然后查看客户机中与时钟相关的信息如下，可知使用了kvm_clock和硬件的TSC支持。

```
[root@kvm-guest ~]# dmesg | grep -i clock
kvm-clock: Using msrs 4b564d01 and 4b564d00
kvm-clock: cpu 0, msr 0:1c1f601, boot clock
kvm-clock: cpu 0, msr 0:2216601, primary cpu clock
hpet clockevent registered
kvm-clock: cpu 1, msr 0:2316601, secondary cpu clock
Switching to clocksource kvm-clock
rtc_cmos 00:01: setting system clock to 2012-09-17 03:27:16 UTC (1347852436)
Refined TSC clocksource calibration: 3292.523 MHz.
```

另外，Intel的一些较新的硬件还向时钟提供了更高级的硬件支持，即TSC Deadline Timer，在前面查看一个SandyBridge平台的CPU信息时已经有"tsc_deadline_timer"的标识了。TSC deadline模式，不是使用CPU外部总线的频率去定时减少计数器的值，而是软件设置了一个"deadline"（最后期限）的阀值，当CPU的时间戳计数器的值大于或等于这个"deadline"时，本地的高级可编程中断控制器（Local APIC）就产生一个时钟中断请求（IRQ）。正是由于这个特点（CPU的时钟计数器运行于CPU的内部频率而不依赖于外部总线频率），TSC Deadline Timer可以提供更精确的时间，也可以更容易避免或处理竞态条件（race condition[6] ）。

KVM模块对TSC Deadline Timer的支持开始于Linux 3.6版本，QEMU对TSC Deadline Timer的支持开始于qeum-kvm 0.12版本。而且在启动客户机时，在qemu-kvm命令行使用"-cpu host"参数才能将这个特性传递给客户机，使其可以使用TSC Deadline Timer。

5.2 设备直接分配（VT-d）

5.2.1 VT-d概述

在QEMU/KVM中，客户机可以使用的设备大致可分为如下3种类型。

1）Emulated device：QEMU纯软件模拟的设备。

2）Virtio device：实现VIRTIO API的半虚拟化驱动的设备。

3）PCI device assignment：PCI设备直接分配。

其中，前两种类型都在5.1.1节“virtio概述”中已经进行了比较详细的介绍，这里再简单回顾一下它们的优缺点和适用场景。

模拟I/O设备方式的优点是对硬件平台依赖性较低、可以方便模拟一些流行的和较老久的设备、不需要宿主机和客户机的额外支持，因此兼容性高；而其缺点是I/O路径较长、VM-Exit次数很多，因此性能较差。一般适用于对I/O性能要求不高的场景，或者模拟一些老旧遗留（legacy）设备（如RTL8139的网卡）。

virtio半虚拟化设备方式的优点是实现了VIRTIO API，减少了VM-Exit次数，提高了客户机I/O执行效率，比普通模拟I/O的效率高很多；而其缺点是需要客户机中与virtio相关驱动的支持（较老的系统

默认没有自带这些驱动，Windows系统中需要额外安装virtio驱动），因此兼容性较差，而且I/O频繁时的CPU使用率较高。

而第三种方式叫做PCI设备直接分配（Device Assignment，或PCI pass-through），它允许将宿主机中的物理PCI（或PCI-E）设备直接分配给客户机完全使用，这正是本节要介绍的重点内容。较新的x86架构的主要硬件平台（包括服务器级、桌面级）都已经支持设备直接分配，其中Intel定义的I/O虚拟化技术规范为"Intel(R)Virtualization Technology for Directed I/O"（VT-d），而AMD的I/O虚拟化技术规范为"AMD-Vi"（也叫做IOMMU）。本节以在KVM中使用Intel VT-d技术为例来进行介绍（当然AMD IOMMU也是类似的）。

KVM虚拟机支持将宿主机中的PCI、PCI-E设备附加到虚拟化的客户机中，从而让客户机以独占方式访问这个PCI（或PCI-E）设备。通过硬件支持的VT-d技术将设备分配给客户机后，在客户机看来，设备是物理上连接在其PCI（或PCI-E）总线上的，客户机对该设备的I/O交互操作和实际的物理设备操作完全一样，不需要（或者很少需要）Hypervisor（即KVM）的参与。在KVM中通过VT-d技术使用一个PCI-E网卡的系统架构示例如图5-10所示。

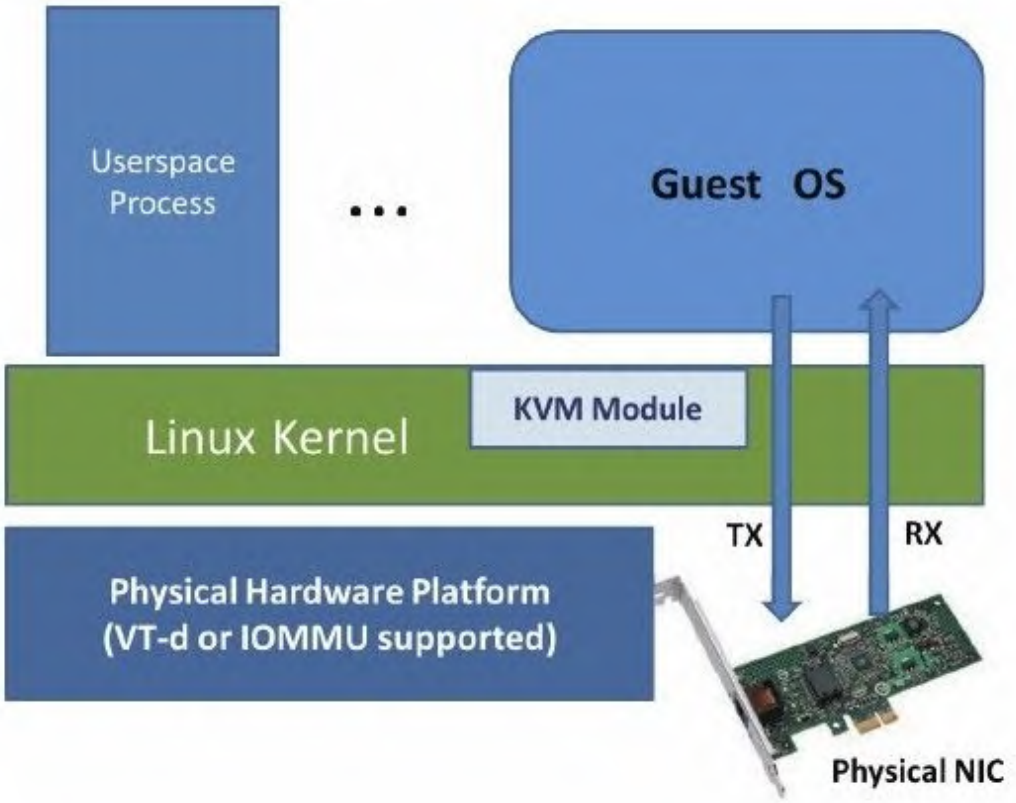


图 5-10 KVM客户机直接分配PCI-E设备（以网卡为例）

运行在支持VT-d平台上的QEMU/KVM，可以分配网卡、磁盘控制器、USB控制器、VGA显卡等供客户机直接使用。而为了设备分配的安全性，还需要中断重映射（interrupt remapping）的支持。尽管在使用QEMU命令行进行设备分配时并不直接检查中断重映射功能是否开启，但是在通过一些工具使用KVM时（如RHEL 6.3中的libvirt）默认需要有中断重映射的功能支持，才能使用VT-d分配设备供客户机使用。

设备直接分配让客户机完全占有PCI设备，这样在执行I/O操作时大量地减少甚至避免了VM-Exit陷入到Hypervisor中，极大地提高了I/O性能，可以达到几乎和Native系统中一样的性能。尽管virtio的性能也不错，但VT-d克服了其兼容性不够好和CPU使用率较高的问题。不过，VT-d也有自己的缺点，一台服务器主板上的空间比较有限，允许添加的PCI和PCI-E设备是有限的，如果一台宿主机上有较多数量的客户机，则很难向每台客户机都独立分配VT-d的设备。另外，大量使用VT-d独立分配设备给客户机，让硬件设备数量增加，这会增加硬件投资成本。为了避免这两个缺点，可以考虑采用如下两个方案：一是，在一台物理宿主机上，仅少数的对I/O（如网络）性能要求较高的客户机使用VT-d直接分配设备（如网卡），而其余的客户机使用纯模拟（emulated）或使用virtio以达到多个客户机共享同一个设备的目的；二是，对于网络I/O的解决方法，可以选择SR-IOV使一个网卡产生多个独立的虚拟网卡，将每个虚拟网卡分别分配给一个客户机使用，这也正是后面5.2.4“SR-IOV技术”这一小节要介绍的内容。另外，设备直接分配还有一个缺点是，对于使用VT-d直接分配了设备的客户机，其动态迁移功能将会受限，不过也可以用热插拔或libvirt工具等方式来缓解这个问题，详细内容将在“动态迁移”小节介绍此方法。

5.2.2 VT-d环境配置

在KVM中使用VT-d技术进行设备直接分配，需要以下几方面的环境配置。

1.硬件支持和BIOS设置

目前市面上有很多的x86硬件平台都有VT-d的支持，以笔者使用过的几台机器为例，其中服务器平台Intel Xeon X5670、Xeon E5-2680、Xeon E5-4650等和SandyBridge、IvyBridge平台的桌面台式机都是支持VT-d的。

除了在硬件平台层面对VT-d支持之外，还需要在BIOS将VT-d功能打开使其处于"Enabled"状态。由于各个BIOS和硬件厂商的标识的区别，VT-d在BIOS中设置选项的名称也有所不同。笔者见到BIOS中VT-d设置选项一般为"Intel(R)VT for Directed I/O"或"Intel VT-d"等，在图3-2中已经演示了在BIOS设置中打开VT-d选项的情况。

2.宿主机内核的配置

在宿主机系统中，内核也需要配置相应的选项。在较新的Linux内核（如3.5、3.6）中，应该配置如下几个VT-d相关的配置选项。在Fedora 17（使用3.3或3.4内核）中，也都已经使这些类似的配置处于打开状态，不需要重新编译内核即可直接使用VT-d。

```
CONFIG_IOMMU_SUPPORT=y
# CONFIG_AMD_IOMMU is not set   #AMD 平台的 IOMMU 设置
CONFIG_DMAR_TABLE=y
CONFIG_INTEL_IOMMU=y            #Intel 平台的 VT-d 设置
CONFIG_INTEL_IOMMU_DEFAULT_ON=y
CONFIG_INTEL_IOMMU_FLOPPY_WA=y
CONFIG_IRQ_REMAP=y
```

而在较旧的Linux内核（3.0及以下，如2.6.32）中，应该配置如下几个VT-d相关的配置选项。与上面较新内核的配置有些不同，大约在发布Linux内核3.0、3.1时进行了一次比较大的选项调整，名称和代码结构有所改变。以RHEL 6.3的内核为例，默认就已经配置了如下的选项，其内核已经支持VT-d技术的使用。

```
CONFIG_DMAR=y
# CONFIG_DMAR_DEFAULT_ON is not set    # 本选项可设置为 y，也可不设置
CONFIG_DMAR_FLOPPY_WA=y
CONFIG_INTR_REMAP=y
```

另外，为了配合接下来的第3步设置（用于隐藏设备），还需要配置pci-stub这个内核模块，相关的内核配置选项如下。在RHEL 6.3和Fedora 17系统的默认内核中，都将PCI_STUB配置为y（直接编译到内核），不作为模块来加载。

```
CONFIG_PCI_STUB=m       # 如果配置为 y（编译进内核），则不需作为模块来加载
```

在启动宿主机系统后，可以通过内核的打印信息来检查VT-d是否处于打开可用状态，如下所示。

```
[root@jay-linux kvm_demo]# dmesg | grep DMAR -i
ACPI: DMAR 00000000bdf96c98 00210 (v01 INTEL   S4600LH 06222004 INTL 20090903)
DMAR: Host address width 46
DMAR: DRHD base: 0x000000ebffe000 flags: 0x0

  DMAR: DRHD base: 0x000000f7ffe000 flags: 0x0
  DMAR: DRHD base: 0x000000fbffe000 flags: 0x0
  DMAR: DRHD base: 0x000000dfffc000 flags: 0x1
  DMAR: RMRR base: 0x000000bdd07000 end: 0x000000bdd1dfff
  DMAR: ATSR flags: 0x0
  [root@jay-linux kvm_demo]# dmesg | grep IOMMU -i
  IOMMU 0: reg_base_addr ebffe000 ver 1:0 cap d2078c106f0462 ecap f020fe
  IOMMU 1: reg_base_addr f7ffe000 ver 1:0 cap d2078c106f0462 ecap f020fe
  IOMMU 2: reg_base_addr fbffe000 ver 1:0 cap d2078c106f0462 ecap f020fe
  IOMMU 3: reg_base_addr dfffc000 ver 1:0 cap d2078c106f0462 ecap f020fe
  IOAPIC id 4 under DRHD base  0xfbffe000 IOMMU 2
  IOAPIC id 3 under DRHD base  0xf7ffe000 IOMMU 1
  IOAPIC id 2 under DRHD base  0xebffe000 IOMMU 0
  IOAPIC id 0 under DRHD base  0xdfffc000 IOMMU 3
  IOAPIC id 1 under DRHD base  0xdfffc000 IOMMU 3
  IOMMU 2 0xfbffe000: using Queued invalidation
  IOMMU 1 0xf7ffe000: using Queued invalidation
  IOMMU 0 0xebffe000: using Queued invalidation
  IOMMU 3 0xdfffc000: using Queued invalidation
  IOMMU: Setting RMRR:
  IOMMU: Setting identity map for device 0000:00:1d.0 [0xbdd07000 - 0xbdd1dfff]
  IOMMU: Setting identity map for device 0000:00:1a.0 [0xbdd07000 - 0xbdd1dfff]
  IOMMU: Prepare 0-16MiB unity mapping for LPC
  IOMMU: Setting identity map for device 0000:00:1f.0 [0x0 - 0xffffff]
```

如果内核的IOMMU默认没有打开，也可以在GRUB的kernel行中加入"intel_iommu=on"这个内核启动选项。

3.在宿主机中隐藏设备

使用pci_stub这个内核模块来对需要分配给客户机的设备进行隐藏，从而让宿主机和未被分配该设备的客户机都无法使用该设备，达到隔离和安全使用的目的，需要通过如下三步来隐藏一个设备。

1）加载pci_stub驱动（前面“2.宿主机内核的配置”中已提及将"CONFIG_PCI_STUB=m"作为内核编译的配置选项），如下所示。

```
[root@jay-linux kvm_demo]# modprobe pci_stub
[root@jay-linux kvm_demo]# lsmod | grep stub
pci_stub                1253  0
[root@jay-linux kvm_demo]# ls /sys/bus/pci/drivers/pci-stub/
0000:06:00.1  bind  module  new_id  remove_id  uevent  unbind
```

如果pci_stub已被编译到内核而不是作为module，则仅需最后一个命令来检查/sys/bus/pci/drivers/pci-stub/目录存在即可。

2）查看设备的vendor ID和device ID，如下所示（假设此设备的BDF为08:00.0）。

```
[root@jay-linux kvm_demo]#lspci -Dn -s 08:00.0
0000:08:00.0 0200: 8086:10b9 (rev 06)
```

在上面lspci命令行中，-D选项表示在输出信息中显示设备的domain，-n选项表示用数字的方式显示设备的vendor ID和device ID，-s选项表示仅显示后面指定的一个设备的信息。在该命令的输出信息中，“0000:08:00.0”表示设备在PCI/PCI-E总线中的具体位置，依次是设备的domain（0000）、bus（08）、slot（00）、function（0），其中domain的值一般为0（当机器有多个host bridge时，其取值范围是0～0xffff），bus的取值范围是0～0xff，slot取值范围是0～0x1f，function取值范围是0～0x7，其中后面3个值一般用BDF（即bus:device:function）来简称。在输出信息中，设备的vendor ID是“8086”（“8086”ID代表Intel Corporation），device ID是"10b9"（代表82572网卡）[7]。

3）绑定设备到pci_stub驱动，命令行操作如下所示。

```
echo  -n "8086 10b9" > /sys/bus/pci/drivers/pci-stub/new_id
echo 0000:08:00.0 > /sys/bus/pci/devices/0000:08:00.0/driver/unbind
echo 0000:08:00.0 > /sys/bus/pci/drivers/pci-stub/bind
```

在绑定前，用lspci命令查看BDF为08:00.0的设备使用的驱动是Intel的e1000e驱动，而绑定到pci_stub后，通过如下命令可以可查看到它目前使用的驱动是pci_stub而不是e1000e了，其中lspci的-k选项表示输出信息中显示正在使用的驱动和内核中可以支持该设备的模块。

```
[root@vt-nhm9 ~]# lspci -k -s 08:00.0
08:00.0 Ethernet controller: Intel Corporation 82572EI Gigabit
Ethernet Controller (Copper) (rev 06)
        Subsystem: Intel Corporation PRO/1000 PT Desktop Adapter
        Kernel driver in use: pci-stub
        Kernel modules: e1000e
```

而在客户机不需要使用该设备后，让宿主机使用该设备，则需要将其恢复到使用原本的驱动。

本节的隐藏和恢复设备，手动操作起来还是有点效率低和容易出错，利用如下一个Shell脚本（命名为pcistub.sh脚本）可以方便地实现该功能，并且使用起来非常简单，仅供大家参考。

```
#!/bin/bash
# A script to hide/unhide PCI/PCIe device for KVM. (using 'pci_stub' driver)

hide_dev=0
unhide_dev=0
driver=0

# check if the device exists
function dev_exist()
{
        local line_num=$(lspci -s "$1" 2>/dev/null | wc -l)
        if [ $line_num = 0 ]; then
                echo "Device $pcidev doesn't exists. Please check
your system or your command line."
                exit 1
```

```
        else
                return 0
        fi
}

# output a format "<domain>:<bus>:<slot>.<func>" (e.g. 0000:01:10.0) of device
function canon()
{
        f='expr "$1" : '.*\.\(.\)'`
        d='expr "$1" : ".*:\(.*\).$f"'
        b='expr "$1" : "\(.*\):$d\.$f"'

        if [ 'expr "$d" : '..'' == 0 ]
        then
                d=0$d
        fi
        if [ 'expr "$b" : '.*:'' != 0 ]
        then
        if [ 'expr "$b" : '.*:'' != 0 ]
        then
                p='expr "$b" : '\(.*\):''
                b='expr "$b" : '.*:\(.*\)''
        else
                p=0000
        fi
        if [ 'expr "$b" : '..'' == 0 ]
        then
                b=0$b
        fi
        echo $p:$b:$d.$f
}

# output the device ID and vendor ID
function show_id()
{
        lspci -Dn -s "$1" | awk '{print $3}' | sed "s/:/ /" > /dev/null 2>&1
        if [ $? -eq 0 ]; then
                lspci -Dn -s "$1" | awk '{print $3}' | sed "s/:/ /"
        else
                echo "Can't find device id and vendor id for device $1"
                exit 1
        fi
}

# hide a device using 'pci_stub' driver/module
function hide_pci()
```

```
{
        local pre_driver=NULL
        local pcidev=$(canon $1)
        local pciid=$(show_id $pcidev)

        dev_exist $pcidev

        if [ -h /sys/bus/pci/devices/"$pcidev"/driver ]; then
                pre_driver=$(basename $(readlink
/sys/bus/pci/devices/"$pcidev"/driver))
                echo "Unbinding $pcidev from $pre_driver"
                echo -n "$pciid" > /sys/bus/pci/drivers/pci-stub/new_id
                echo -n "$pcidev" >
/sys/bus/pci/devices/"$pcidev"/driver/unbind
        fi

        echo "Binding $pcidev to pci-stub"
        echo -n "$pcidev" > /sys/bus/pci/drivers/pci-stub/bind
        return $?
}

# unhide a device from 'pci_stub' driver and bind to a new driver
function unhide_pci()
{
        local driver=$2
        local pcidev=$(canon $1)
        local pciid=$(show_id $pcidev)
        local pre_driver=NULL

        dev_exist $pcidev

        if [ $driver != 0 -a ! -d /sys/bus/pci/drivers/$driver ]; then
                echo "No $driver interface under sys, return fail"
                exit 1
        fi

        if [ -h /sys/bus/pci/devices/"$pcidev"/driver ]; then
                pre_driver=$(basename $(readlink
/sys/bus/pci/devices/"$pcidev"/driver))
                if [ "$pre_driver" = "$driver" ]; then
                        echo "$1 has been already bind with $driver,
no need to unhide and bind."
                        exit 1
                elif [ "$pre_driver" != "pci-stub" ]; then
                        echo "$1 is not bind with pci-stub, it is bind
with $pre_driver, no need to unhide"
```

```
                        exit 1
                else
                        echo "Unbinding $pcidev from $pre_driver"
                        if [ $driver != 0 ]; then
                                echo -n "$pciid" >
/sys/bus/pci/drivers/$driver/new_id
                        fi
                        echo -n "$pcidev" >
/sys/bus/pci/drivers/pci-stub/unbind
                        if [ $? -ne 0 ]; then
                                return $?
                        fi
                fi
        fi

        if [ $driver != 0 ]; then
                echo "Binding $pcidev to $driver"
                echo -n "$pcidev" > /sys/bus/pci/drivers/$driver/bind
        fi

        return $?
}

# show the usage of this script
function usage()
{
        echo "Usage: pcistub -h pcidev [-u pcidev] [-d driver]"
        echo " -h pcidev: <pcidev> is BDF number of the device you
want to hide"
        echo " -u pcidev: Optional. <pcidev> is BDF number of the
device you want to unhide."
        echo " -d driver: Optional. When unhiding the device, bind
the device with <driver>. The option should be used together with '-u' option"
        echo ""
        echo "Example1: sh pcistub.sh -h 06:10.0         # Hide
device 01:10.0 to 'pci_stub' driver"
        echo "Example2: sh pcistub.sh -u 08:00.0 -d e1000e  # Unhide
device 08:00.0 and bind the device with 'e1000e' driver"
        exit 1
}

if [ $# -eq 0 ] ; then
        usage
fi

# parse the options in the command line
```

```
OPTIND=1
while getopts ":h:u:d:" Option
do
        case $Option in
                h ) hide_dev=$OPTARG;;
                u ) unhide_dev=$OPTARG;;
                d ) driver=$OPTARG;;
                * ) usage ;;
        esac
done

if [ ! -d /sys/bus/pci/drivers/pci-stub ]; then
        modprobe pci_stub
        if [ ! -d /sys/bus/pci/drivers/pci-stub ]; then
                echo "There's no 'pci-stub' module? Please check
your kernel config."
                exit 1
        fi
fi

if [ $hide_dev != 0 -a $unhide_dev != 0 ]; then
        echo "Do not use -h and -u option together."
        exit 1
fi

if [ $unhide_dev = 0 -a $driver != 0 ]; then
        echo "You should set -u option if you want to use -d option
to unhide a device and bind it with a specific driver"
        exit 1
fi

if [ $hide_dev != 0 ]; then
        hide_pci $hide_dev
elif [ $unhide_dev != 0 ]; then
        unhide_pci $unhide_dev $driver
fi
exit $?
```

本节后面的VT-d/SR-IOV的操作实例就是使用这个脚本对设备进行隐藏和恢复的。

4.通过QEMU命令行分配设备给客户机

利用qemu-kvm命令行中"-device"选项可以为客户机分配一个设备，配合其中的"pci-assign"作为子选项可以实现设备直接分配。

-device *driver*[,*prop*[=*value*][,...]]

其中driver是设备使用的驱动，有很多种类，如pci-assign表示PCI设备直接分配、virtio-balloon-pci（又为virtio-balloon）表示ballooning设备（这与第4章中提到的"-balloonvirtio"的意义相同）。prop[=value]是设置驱动的各个属性值。

"-device?"可以查看有哪些可用的驱动，"-device driver,?"可查看某个驱动的各个属性值，如下面命令行所示。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -device ?
name "usb-kbd", bus USB
name "virtio-serial-pci", bus PCI, alias "virtio-serial"
name "sysbus-ohci", bus System, desc "OHCI USB Controller"
<! -此处省略数十行输出信息 -->
name "virtio-balloon-pci", bus PCI, alias "virtio-balloon"
name "pci-assign", bus PCI
[root@jay-linux kvm_demo]# qemu-system-x86_64 -device pci-assign,?
kvm-pci-assign.host=pci-host-devaddr   #qemu-kvm 1.1.0 没这行
host 属性输出，算是个小 bug（qemu-kvm1.2 中已经 fix 了），但功能上正常
pci-assign.prefer_msi=on/off
pci-assign.share_intx=on/off
pci-assign.bootindex=int32
pci-assign.configfd=string
pci-assign.addr=pci-devfn
pci-assign.romfile=string
pci-assign.rombar=uint32
pci-assign.multifunction=on/off
pci-assign.command_serr_enable=on/off
pci-assign.id=string   #qemu-kvm 1.1.0 中也没有这行提示，但可正常使用
```

在-device pci-assign的属性中，host属性指定分配的PCI设备在宿主机中的地址（BDF号），addr属性表示设备在客户机中的PCI的slot编号（即BDF中的D-device的值），id属性表示该设备的唯一标识（可以在QEMU monitor中用"info pci"命令查看到）。

qemu-kvm命令行工具在启动时分配一个设备给客户机，命令行如下所示。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -device pci-assign,host=08:00.0,id=mydev0,addr=0x6
```

如果要一次性分配多个设备给客户机，只需在qemu-kvm命令行中重复多次"-device pci-assign,host=$BDF"这样的选项即可。由于设备直接分配是客户机独占该设备，因此一旦将一个设备分配给客户机使用，就不能再将其分配给另外的客户机使用了，否则在通过命令行启动另一个客户机也分配这个设备时，会遇到如下的错误：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u2.img -m
1024 -device pci-assign,host=08:00.0
qemu-system-x86_64: -device pci-assign,host=08:00.0: Failed to
assign device "(null)" : Device or resource busy
qemu-system-x86_64: -device pci-assign,host=08:00.0: *** The
driver 'pci-stub' is occupying your device 0000:08:00.0.
qemu-system-x86_64: -device pci-assign,host=08:00.0: ***
qemu-system-x86_64: -device pci-assign,host=08:00.0: ***You can

try the following commands to free it:
qemu-system-x86_64: -device pci-assign,host=08:00.0: ***
qemu-system-x86_64: -device pci-assign,host=08:00.0: *** $ echo
"8086 10b9" > /sys/bus/pci/drivers/pci-stub/new_id
qemu-system-x86_64: -device pci-assign,host=08:00.0: *** $ echo
"0000:08:00.0" > /sys/bus/pci/drivers/pci-stub/unbind
qemu-system-x86_64: -device pci-assign,host=08:00.0: *** $ echo
"0000:08:00.0" > /sys/bus/pci/drivers/pci-stub/bind
qemu-system-x86_64: -device pci-assign,host=08:00.0: *** $ echo
"8086 10b9" > /sys/bus/pci/drivers/pci-stub/remove_id
qemu-system-x86_64: -device pci-assign,host=08:00.0: ***
qemu-system-x86_64: -device pci-assign,host=08:00.0: Device
'kvm-pci-assign' could not be initialized
```

除了在客户机启动时就直接分配设备之外，QEUM/KVM还支持设备的热插拔（hot-plug）在客户机运行时添加所需的直接分配的设备，

这需要在QEMU monitor中运行相应的命令，相关内容将在5.3节“热插拔”中详细介绍。

5.2.3 VT-d操作示例

1.网卡直接分配

在如今的网络时代，有很多服务器对网卡性能要求较高，在虚拟客户机中也不例外。通过VT-d技术将网卡直接分配给客户机使用，会让客户机得到和native环境中使用网卡几乎一样的性能。本节通过示例来演示将一个Intel 82572型号的PCI-E网卡分配给一个RHEL 6.3客户机使用的过程，这里省略BIOS配置、宿主机内核检查等操作步骤（详见5.2.2小节的“VT-d环境配置”）。

1）选择需要直接分配的网卡。

```
[root@jay-linux kvm_demo]# lspci -s 08:00.0
08:00.0 Ethernet controller: Intel Corporation 82572EI Gigabit
Ethernet Controller (Copper) (rev 06)
        Subsystem: Intel Corporation PRO/1000 PT Desktop Adapter
        Kernel driver in use: e1000e
        Kernel modules: e1000e
```

2）隐藏该网卡（使用了前面介绍的pcistub.sh脚本）。

```
[root@jay-linux kvm_demo]#./pcistub.sh -h 08:00.0
Unbinding 0000:08:00.0 from e1000e
Binding 0000:08:00.0 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 08:00.0
08:00.0 Ethernet controller: Intel Corporation 82572EI Gigabit
Ethernet Controller (Copper) (rev 06)
        Subsystem: Intel Corporation PRO/1000 PT Desktop Adapter
        Kernel driver in use: pci-stub
        Kernel modules: e1000e
```

3）启动客户机时分配网卡。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -device pci-assign,host=08:00.0,id=mynic -net none
VNC server running on '::1:5900'
```

命令行中的"-net none"表示不使其他的网卡设备（除了直接分配的网卡之外），否则在客户机中将会出现一个直接分配的网卡和另一个emulated的网卡。

在QEMU monitor中，可以用"info pci"命令查看分配给客户机的PCI设备的情况。

```
(qemu) info pci
  Bus  0, device   0, function 0:
    Host bridge: PCI device 8086:1237
      id ""
  Bus  0, device   1, function 0:
    ISA bridge: PCI device 8086:7000
<! -此处省略其余一些PCI设备的信息 -->
  Bus  0, device   6, function 0:
    Ethernet controller: PCI device 8086:10b9
    IRQ 10.
    BAR0: 32 bit memory at 0xfeba0000 [0xfebbffff].
    BAR1: 32 bit memory at 0xfebc0000 [0xfebdffff].
    BAR2: I/O at 0xc000 [0xc01f].
    id "mynic"
```

4）在客户机中查看网卡的工作情况。

```
[root@kvm-guest ~]#lspci -s 00:06.0
00:06.0 Ethernet controller: Intel Corporation 82572EI Gigabit
[root@kvm-guest ~]# ethtool -i eth2
driver: e1000e
version: 1.9.5-k
firmware-version: 5.11-10
bus-info: 0000:00:06.0
[root@kvm-guest ~]# ping 192.168.199.98 -I eth2 -c 1
PING 192.168.199.98 (192.168.199.98) from 192.168.199.69 eth2:
56(84) bytes of data.
64 bytes from 192.168.199.98: icmp_seq=1 ttl=64 time=0.367 ms

--- 192.168.199.98 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.367/0.367/0.367/0.000 ms
```

由上面输出信息可知，在客户机中看到的网卡是使用e1000e驱动的Intel 82572网卡（和宿主机隐藏它之前看到的是一样的），eth2就是该网卡的网络接口，通过ping命令查看其网络是畅通的。

5）关闭客户机后，在宿主机中恢复前面被隐藏的网卡。

在客户机关闭或网卡从客户机中虚拟地“拔出”来之后，如果想让宿主机继续使用该网卡，则可以使用pcistub.sh脚本来恢复其在宿主机中的驱动绑定情况，操作过程如下所示。

```
[root@jay-linux kvm_demo]#./pcistub.sh -u 08:00.0 -d e1000e
Unbinding 0000:08:00.0 from pci-stub
Binding 0000:08:00.0 to e1000e
[root@jay-linux kvm_demo]# lspci -k -s 08:00.0
08:00.0 Ethernet controller: Intel Corporation 82572EI Gigabit
Ethernet Controller (Copper) (rev 06)
       Subsystem: Intel Corporation PRO/1000 PT Desktop Adapter
       Kernel driver in use: e1000e
       Kernel modules: e1000e
[root@jay-linux kvm_demo]#ethtool -i eth6
driver: e1000e
version: 2.0.0-k
firmware-version: 5.11-10
bus-info: 0000:08:00.0
```

其中，在pcistub.sh脚本中，"-u$BDF"是指定需要取消隐藏（unhide）的设备，"-d$driver"是指将从pci_stub的绑定中取消隐藏的设备绑定到另外一个新的驱动（driver）中。由上面的输出信息可知，08:00.0设备使用的驱动从pci_stub变回了e1000e，而且以太网络接口eth6正是该设备。

2.硬盘直接分配

在现代计算机系统中，一般SATA或SAS等类型硬盘的控制器（Controller）都是接入到PCI（或PCIe）总线上的，所以也可以将硬盘作为普通PCI设备直接分配给客户机使用。不过当SATA或SAS设备作为PCI设备直接分配时，实际上将其控制器作为一个整体分配到客户机中，如果宿主机使用的硬盘也连接在同一个SATA或SAS控制器上，则不能将该控制器直接分配给客户机，而是需要硬件平台中至少有两个或以上的SATA或SAS控制器。宿主机系统使用其中一个，然后将另外的一个或多个SATA/SAS控制器完全分配给客户机使用。下面以一个SATA硬盘为实例介绍对硬盘的直接分配过程。

1）先在宿主机中查看硬盘设备，然后隐藏需要直接分配的硬盘，其命令行操作如下所示。

```
[root@jay-linux kvm_demo]#ll
/dev/disk/by-path/pci-0000\:16\:00.0-sas-0x1221000000000000-lun-0
lrwxrwxrwx 1 root root 9 Sep 24 11:17
/dev/disk/by-path/pci-0000:16:00.0-sas-0x1221000000000000-lun-0 -> ../../sda
[root@jay-linux kvm_demo]# ll
/dev/disk/by-path/pci-0000\:00\:1f.2-scsi-0\:0\:0\:0
lrwxrwxrwx 1 root root 9 Sep 24 11:17
/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 -> ../../sdb

[root@jay-linux kvm_demo]#lspci -k -s 16:00.0
16:00.0 SCSI storage controller: LSI Logic / Symbios Logic SAS1078
PCI-Express Fusion-MPT SAS (rev 04)
        Subsystem: Intel Corporation Device 3505
        Kernel driver in use: mptsas
        Kernel modules: mptsas
```

```
[root@jay-linux kvm_demo]#lspci -k -s 00:1f.2
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
        Subsystem: Intel Corporation Device 34f8
        Kernel driver in use: ahci
        Kernel modules: ahci

[root@jay-linux kvm_demo]#fdisk -l /dev/sdb

Disk /dev/sdb: 164.7 GB, 164696555520 bytes
255 heads, 63 sectors/track, 20023 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0003e001

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           1        6528    52428800   83  Linux
/dev/sdb2            6528        7050     4194304   82  Linux swap / Solaris
/dev/sdb3            7050        9600    20480000   83  Linux

[root@jay-linux kvm_demo]#df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             197G   13G  175G   7% /
tmpfs                  12G   76K   12G   1% /dev/shm

[root@jay-linux kvm_demo]# ./pcistub.sh -h 00:1f.2
Unbinding 0000:00:1f.2 from ahci
Binding 0000:00:1f.2 to pci-stub

[root@jay-linux kvm_demo]#lspci -k -s 00:1f.2
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
        Subsystem: Intel Corporation Device 34f8
        Kernel driver in use: pci-stub
        Kernel modules: ahci
```

由上面的命令行输出可知，在宿主机中有两块硬盘sda和sdb，分别对应一个SAS Controller（16:00.0）和一个SATA Controller（00:1f.2），其中sdb大小为160GB，而宿主机系统安装在sda的第一个分区（sda1）上。在用pcistub.sh脚本隐藏SATA

Controller之前，它使用的驱动是ahci驱动，之后，将其绑定到pci-stub驱动，为设备直接分配做准备。

2）用如下命令行将STAT硬盘分配（实际是分配STAT Controller）给客户机使用。

```
[root@jay-linux kvm_demo]#qemu-system-x86_64 rhel6u3.img -m
1024 -device pci-assign,host=00:1f.2,addr=0x6-net nic -net tap
VNC server running on '::1:5900'
```

3）在客户机启动后，在客户机中查看直接分配得到的SATA硬盘，命令行如下所示。

```
[root@kvm-guest ~]#fdisk -l /dev/sdb

Disk /dev/sdb: 164.7 GB, 164696555520 bytes
255 heads, 63 sectors/track, 20023 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0003e001

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           1        6528    52428800   83  Linux
/dev/sdb2            6528        7050     4194304   82  Linux swap / Solaris
/dev/sdb3            7050        9600    20480000   83  Linux

[root@kvm-guest ~]#ll
/dev/disk/by-path/pci-0000\:00\:04.0-scsi-2\:0\:0\:0
lrwxrwxrwx 1 root root 9 Sep 23 23:47
/dev/disk/by-path/pci-0000:00:04.0-scsi-2:0:0:0 -> ../../sdb

[root@kvm-guest ~]#lspci -k -s 00:06.0
00:06.0 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
        Subsystem: Intel Corporation Device 34f8
        Kernel driver in use: ahci
        Kernel modules: ahci
```

由客户机中的以上命令行输出可知，宿主机中的sdb硬盘（BDF为00:06.0）就是设备直接分配的那个160GB大小的SATA硬盘。在SATA硬盘成功直接分配到客户机后，客户机中的程序就可以像普通硬盘一样对其进行读写操作（也包括磁盘分区等管理操作）。

3.USB直接分配

与SATA和SAS控制器类似，在很多现代计算机系统中，USB主机控制器（USB Host Controller）也是接入到PCI总线中去的，所以也可以对USB设备做设备直接分配。同样，这里的USB直接分配，也是指对

整个USB Host Controller的直接分配，而并不一定仅分配一个USB设备。常见的USB设备，如U盘、键盘、鼠标等都可以作为设备直接分配到客户机中使用。这里以U盘为例来介绍USB直接分配，而USB键盘、鼠标的直接分配也与此类似。在后面介绍VGA直接分配的示例时，也会将鼠标、键盘直接分配到客户机中。

1）在宿主机中查看U盘设备，并将其隐藏起来以供直接分配，命令行操作如下所示。

```
[root@jay-linux kvm_demo]# fdisk -l /dev/sdb

Disk /dev/sdb: 16.0 GB, 16008609792 bytes
21 heads, 14 sectors/track, 106349 cylinders
```

```
Units = cylinders of 294 * 512 = 150528 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xcad4ebea

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           7      106350    15632384    c  W95 FAT32 (LBA)
[root@jay-linux kvm_demo]# ls -l
/dev/disk/by-path/pci-0000\:00\:1d.0-usb-0\:1.2\:1.0-scsi-0\:0\:0\:0
lrwxrwxrwx 1 root root 9 Sep 24 06:47
/dev/disk/by-path/pci-0000:00:1d.0-usb-0:1.2:1.0-scsi-0:0:0:0 -> ../../sdb
[root@jay-linux kvm_demo]# lspci -k -s 00:1d.0
00:1d.0 USB controller: Intel Corporation C600/X79 series chipset
USB2 Enhanced Host Controller #1 (rev 06)
        Subsystem: Intel Corporation Device 35a0
        Kernel driver in use: ehci_hcd
        Kernel modules: ehci-hcd
[root@jay-linux kvm_demo]# ./pcistub.sh -h 00:1d.0
Unbinding 0000:00:1d.0 from ehci_hcd
Binding 0000:00:1d.0 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 00:1d.0
00:1d.0 USB controller: Intel Corporation C600/X79 series chipset
        USB2 Enhanced Host Controller #1 (rev 06)
        Subsystem: Intel Corporation Device 35a0
        Kernel driver in use: pci-stub
        Kernel modules: ehci-hcd
```

由宿主机中的命令行输出可知，sdb就是那个使用USB 2.0协议的U盘，它的大小为16GB，其PCI的ID为00:1d.0，在pci-stub隐藏之前使用的是ehci-hcd驱动，然后被绑定到pci-stub驱动隐藏起来以供后面直接分配给客户机使用。

2）将U盘直接分配给客户机使用的命令行如下所示，与普通PCI设备直接分配的操作完全一样。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -device pci-assign,host=00:1d.0,addr=0x5 -net nic -net tap
VNC server running on '::1:5900'
```

3）在客户机中，查看通过直接分配得到的U盘，命令行如下。

```
[root@kvm-guest ~]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             7.4G  6.3G  734M  90% /
tmpfs                 499M     0  499M   0% /dev/shm
[root@kvm-guest ~]# fdisk -l /dev/sdb

Disk /dev/sdb: 16.0 GB, 16008609792 bytes
21 heads, 14 sectors/track, 106349 cylinders

Units = cylinders of 294 * 512 = 150528 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xcad4ebea

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           7      106350    15632384    c  W95 FAT32 (LBA)
[root@kvm-guest ~]# lspci -k -s 00:05.0
00:05.0 USB controller: Intel Corporation C600/X79 series chipset
USB2 Enhanced Host Controller #1 (rev 06)
       Subsystem: Intel Corporation Device 35a0
       Kernel driver in use: ehci_hcd
```

由客户机中的命令行输出可知，sdb就是那个16GB的U盘（BDF为00:05.0），目前使用ehci_hcd驱动。在U盘直接分配成功后，客户机就可以像普通系统使用U盘一样直接使用它了。

另外，对于USB2.0设备，也有其他的命令行参数（-usbdevice）来支持USB设备的分配。不同于前面介绍的对USB Host Controller的直接分配，-usbdevice参数用于分配单个USB设备。在宿主机中不要对USB Host Controller进行隐藏（如果前面已经隐藏了，可以

用"pcistub.sh-u 00:1d.0-d ehci_hcd"命令将其释放出来），用"lsusb"命令查看需要分配的USB设备的信息，然后在要启动客户机的命令行中使用"-usbdevice host:xx"这样的参数启动客户机即可，其操作过程如下。

```
[root@jay-linux kvm_demo]# lsusb
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0781:5567 SanDisk Corp. Cruzer Blade
# 用于分配的SandDisk的U盘设备
root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m 1024
-usbdevice host:0781:5667 -net nic -net tap
VNC server running on '::1:5900'
```

4.VGA显卡直接分配

在计算机系统中，显卡也是作为一个PCI或PCIe设备接入到系统总线之中的。在KVM虚拟化环境中，如果有在客户机中看一些高清视频和玩一些高清游戏的需求，也可以将显卡像普通PCI设备一样完全分配给某个客户机使用。目前，市面上显卡的品牌很多，有Nvidia、ATI等独立显卡品牌，也包括Intel等公司在较新的CPU中集成的GPU模块（具有3D显卡功能）。显卡也有多种的接口类型，如VGA（Video Graphics Array）、DVI（Digital Visual Interface）、HDMI（High-Definition Multimedia Interface）等多种标准的接口。下面以一台服务器上的集成VGA显卡为例，介绍显卡设备的直接分配过程，在此过程中也将USB鼠标和键盘一起分配给客户机，以方便用服务器上直接连接的物理鼠标、键盘操作客户机。

1）查看USB键盘和鼠标的PCI的BDF，查看VGA显卡的BDF，命令行操作如下所示。

```
[root@jay-linux ~]# dmesg | grep -e Keyboard -e Mouse
hid-generic 0003:0557:2217.0001: input: USB HID v1.10 Keyboard
[ATEN ATEN  CS-1716/08/04] on usb-0000:00:1a.0-1.5/input0
hid-generic 0003:0557:2217.0002: input: USB HID v1.10 Mouse [ATEN
ATEN  CS-1716/08/04] on usb-0000:00:1a.0-1.5/input1
input: American Megatrends Inc. Virtual Keyboard and Mouse as
/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.4/2-1.4:1.0/input/input2
hid-generic 0003:046B:FF10.0003: input: USB HID v1.10 Keyboard
[American Megatrends Inc. Virtual Keyboard and Mouse] on
usb-0000:00:1d.0-1.4/input0
input: American Megatrends Inc. Virtual Keyboard and Mouse as
/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.4/2-1.4:1.1/input/input3
hid-generic 0003:046B:FF10.0004: input: USB HID v1.10 Mouse
[American Megatrends Inc. Virtual Keyboard and Mouse] on
usb-0000:00:1d.0-1.4/input1
[root@jay-linux kvm_demo]# lsusb
Bus 003 Device 002: ID 046b:ff10 American Megatrends, Inc. Virtual
Keyboard and Mouse
Bus 005 Device 002: ID 0557:2217 ATEN International Co., Ltd
Bus 008 Device 002: ID 05e3:0719 Genesys Logic, Inc. SATA adapter
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 008 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
[root@jay-linux kvm_demo]# lspci -k -s 00:1a.0
00:1a.0 USB controller: Intel Corporation C600/X79 series chipset
USB2 Enhanced Host Controller #2 (rev 06)
       Subsystem: Intel Corporation Device 35a0
       Kernel driver in use: ehci_hcd
       Kernel modules: ehci-hcd
[root@jay-linux ~]# lspci | grep -i VGA
09:00.0 VGA compatible controller: Matrox Electronics Systems
Ltd. MGA G200e [Pilot] ServerEngines (SEP1) (rev 05)
[root@jay-linux ~]# lspci -k -s 09:00.0
09:00.0 VGA compatible controller: Matrox Electronics Systems
Ltd. MGA G200e [Pilot] ServerEngines (SEP1) (rev 05)
       Subsystem: Intel Corporation Device 0103
```

由上面命令行的输出信息可知，USB鼠标和键盘的USB控制器的PCI设备BDF为00:1a.0（笔者实验是用的一个USB转接器，然后在它上面接了一个键盘和一个鼠标，所以宿主机只能看到一个USB设备），宿主机VGA显卡是Matrox公司的G200e型号显卡，其BDF为09:00.0。

2）分别将鼠标、键盘和VGA显卡隐藏起来，以便分配给客户机，命令行操作如下所示。

```
[root@jay-linux kvm_demo]# ./pcistub.sh -h 00:1a.0
Unbinding 0000:00:1a.0 from ehci_hcd
Binding 0000:00:1a.0 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 00:1a.0
00:1a.0 USB controller: Intel Corporation C600/X79 series chipset
USB2 Enhanced Host Controller #2 (rev 06)
       Subsystem: Intel Corporation Device 35a0
       Kernel driver in use: pci-stub
       Kernel modules: ehci-hcd
[root@jay-linux kvm_demo]# ./pcistub.sh -h 09:00.0
0000:09:00.0 wasn't bound to any drirver
Binding 0000:09:00.0 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 09:00.0
09:00.0 VGA compatible controller: Matrox Electronics Systems
Ltd. MGA G200e [Pilot] ServerEngines (SEP1) (rev 05)
       Subsystem: Intel Corporation Device 0103
       Kernel driver in use: pci-stub
```

3）qemu-kvm命令行启动一个客户机，将USB鼠标键盘和VGA显卡都分配给它，其命令行操作如下所示。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -device pci-assign,host=09:00.0 -device
pci-assign,host=00:1a.0 -net nic -net tap
VNC server running on '::1:5900'
```

4）在客户机中查看分配的VGA显卡和USB键盘鼠标，命令行操作如下所示。

```
[root@kvm-guest ~]# lspci | grep USB
00:05.0 USB controller: Intel Corporation 82801JI (ICH10 Family)
USB UHCI Controller #2
[root@kvm-guest ~]# lsusb
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 002: ID 0557:2217 ATEN International Co., Ltd
[root@kvm-guest ~]# lspci | grep VGA
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:04.0 VGA compatible controller: Matrox Electronics Systems
Ltd. MGA G200e [Pilot] ServerEngines (SEP1) (rev 05)
[root@kvm-guest ~]# dmesg | grep -i VGA
Console: colour VGA+ 80x25
vgaarb: device added:
PCI:0000:00:02.0,decodes=io+mem,owns=io+mem,locks=none
vgaarb: device added:
PCI:0000:00:04.0,decodes=io+mem,owns=io+mem,locks=none
vgaarb: loaded
vgaarb: bridge control possible 0000:00:04.0
vgaarb: no bridge control possible 0000:00:02.0
```

由上面输出可以看出，客户机中这个USB设备与宿主机中的是一样的，客户机有两个VGA显卡，其中BDF 00:02.0是4.6节提到的QEMU纯软件模拟的Cirrus显卡，而另外的BDF 00:04.0就是设备直接分配得到的GMA G200e显卡，它的信息和在宿主机中查看到的是一样的。从dmesg信息可以看到系统启动后，00:04.0显卡才是最后真正使用的显卡，而00:02.0是不可用的（处于"no bridge control possible"状态）。另外，本示例在客户机中也启动了图形界面，对使用的显卡进行检查还

可以在客户机中查看Xorg的日志文件：/var/log/Xorg.0.log，其中部分内容如下：

```
[    29.220]
X.Org X Server 1.10.6
Release Date: 2012-02-10
[    29.221] X Protocol Version 11, Revision 0
[    29.221] Build Operating System: x86-007 2.6.18-308.1.1.el5
[    29.221] Current Operating System: Linux kvm-guest
2.6.32-279.el6.x86_64 #1 SMP Wed Jun 13 18:24:36 EDT 2012 x86_64
<!-- 省略数十行输出信息  -->
(--) PCI: (0:0:2:0) 1013:00b8:1af4:1100 Cirrus Logic GD 5446 rev
0, Mem @ 0xfa000000/33554432, 0xfe834000/4096, BIOS @ 0x????????/65536
(--) PCI:*(0:0:4:0) 102b:0522:8086:0101 Matrox Graphics, Inc.
MGA G200e [Pilot] ServerEngines (SEP1) rev 2, Mem @ 0xfc000000/
16777216, 0xfe830000/16384, 0xfe000000/8388608, BIOS @ 0x????????/65536
(==) Using default built-in configuration (30 lines)
<!-- 省略数十行输出信息  -->
(II) MGA: driver for Matrox chipsets: mga2064w, mga1064sg, mga2164w,
        mga2164w AGP, mgag100, mgag100 PCI, mgag200, mgag200 PCI,
        mgag200 SE A PCI, mgag200 SE B PCI, mgag200 EV Maxim,
        mgag200 ER SH7757, mgag200 eW Nuvoton, mgag200eH, mgag400, mgag550
(II) VESA: driver for VESA chipsets: vesa
(II) FBDEV: driver for framebuffer: fbdev
(II) Primary Device is: PCI 00@00:04:0
<!-- 省略后面的其他输出信息  -->
```

由上面Xorg.0.log中的日志信息可知，X窗口程序检测到两个VGA显卡，最后使用的是BDF为00:04.0的显卡，使用了VESA程序来驱动该显卡。在客户机内核的配置中，对VESA的配置已经变异到内核中去了，因此可以直接使用。

```
[root@rhel6u3-ga ~]# grep VESA
/boot/config-2.6.32-279.el6.x86_64
CONFIG_FB_BOOT_VESA_SUPPORT=y
# CONFIG_FB_UVESA is not set
CONFIG_FB_VESA=y
```

在本示例中，在RHEL 6.3客户机启动的前期默认使用的是QEMU模拟的Cirrus显卡，而在系统启动完成后打开用户登录界面（启动了X-window图形界面），客户机就自动切换到使用直接分配的设备GMA G200e显卡了，在连接物理显卡的显示器上就出现了客户机的界面。

对于不同品牌的显卡及不同类型的客户机系统，KVM对它们的支持有所不同，其中也存在部分bug，在使用显卡设备直接分配时，可能有的显卡在某些客户机中并不能正常工作，这就需要根据实际情况来操作。另外，在Windows客户机中，如果在“设备管理器”中看到了分配给它的显卡，但是并没有使用和生效，可能需要下载合适的显卡驱动，并且在“设备管理器”中关闭纯软件模拟的那个显卡，而且需要开启设置直接分配得到的显卡，这样才能让接VGA显卡的显示器能显示Windows客户机中的内容。

5.2.4 SR-IOV技术

1.SR-IOV概述

前面介绍的普通VT-d技术实现了设备直接分配，尽管其性能非常好，但是它的一个物理设备只能分配给一个客户机使用。为了实现多个虚拟机能够共享同一个物理设备的资源，并且达到设备直接分配的性能，PCI-SIG[8] 组织发布了SR-IOV（Single Root I/O Virtualization and Sharing）规范，该规范定义了一个标准化的机制用以原生地支持实现多个共享的设备（不一定是网卡设备）。不过，目前SR-IOV（单根I/O虚拟化）最广泛的应用还是在以太网卡设备的虚拟化方面。QEMU/KVM在2009年实现了对SR-IOV技术的支持，其他一些虚拟化方案（如Xen、VMware、Hyper-V等）也都支持SR-IOV了。

在详细介绍SR-IOV之前，先介绍一下SR-IOV中引入的两个新的功能（function）类型。

1）Physical Function（PF，物理功能）：拥有包含SR-IOV扩展能力在内的所有完整的PCI-e功能，其中SR-IOV能力使PF可以配置和管理SR-IOV功能。简言之，PF就是一个普通的PCI-e设备（带有SR-IOV功能），可以放在宿主机中配置和管理其他VF，它本身也可以作为一个完整独立的功能使用。

2）Virtual Function（VF，虚拟功能）：由PF衍生而来的“轻量级”的PCI-e功能，包含数据传送所必需的资源，但是仅谨慎地拥有最小化的配置资源。简言之，VF通过PF的配置之后，可以分配到客户机中作为独立功能使用。

SR-IOV为客户机中使用的VF提供了独立的内存空间、中断、DMA流，从而不需要Hypervisor介入数据的传送过程。SR-IOV架构设计的目的是允许一个设备支持多个VF，同时也尽量减小每个VF的硬件成本。Intel有不少高级网卡可以提供SR-IOV的支持，图5-11展示了Intel以太网卡中的SR-IOV的总体架构。

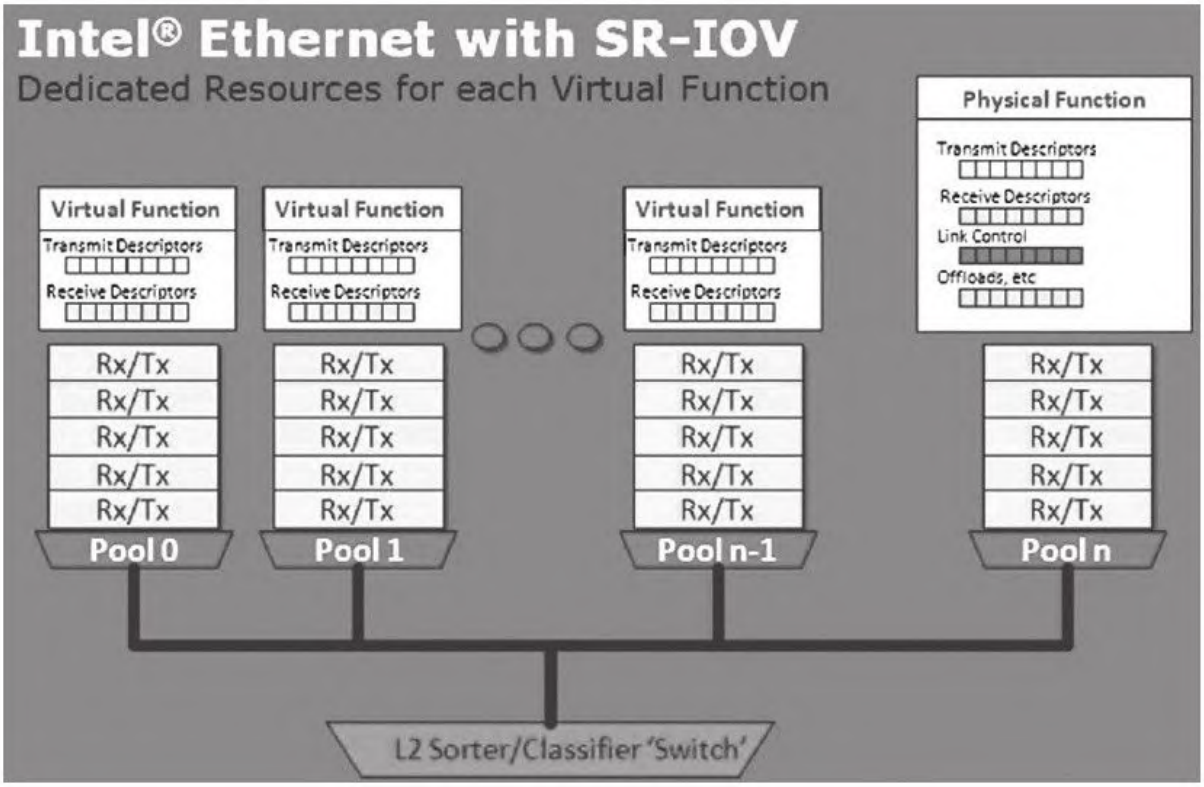


图　5-11　Intel以太网卡的SR-IOV示例

一个具有SR-IOV功能的设备能够被配置为在PCI配置空间（configuration space）中呈现出多个Function（包括一个PF和多个VF），每个VF都有自己独立的配置空间和完整的BAR（Base Address Register，基址寄存器）。Hypervisor通过将VF实际的配置空间映射

到客户机看到的配置空间的方式实现将一个或多个VF分配给一个客户机。通过Intel VT-x和VT-d等硬件辅助虚拟化技术提供的内存转换技术，允许直接的DMA传输去往或来自一个客户机，从而绕过了Hypervisor中的软件交换机（software switch）。每个VF在同一个时刻只能被分配到一个客户机中，因为VF需要真正的硬件资源（不同于emulated类型的设备）。在客户机中的VF，表现给客户机操作系统的就是一个完整的普通的设备。

在KVM中，可以将一个或多个VF分配给一个客户机，客户机通过自身的VF驱动程序直接操作设备的VF而不需要Hypervisor（即KVM）的参与，其总体架构示意图如图5-12所示。

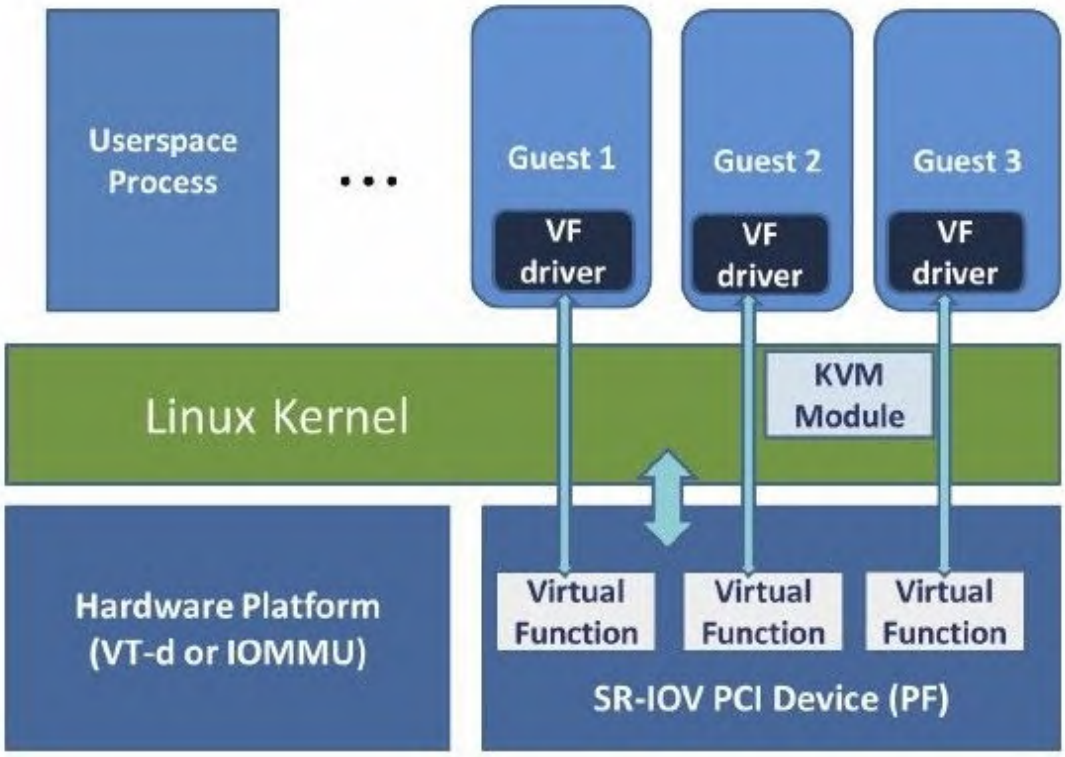


图　5-12　在KVM中使用SR-IOV功能

为了让SR-IOV工作起来，需要硬件平台支持Intel VT-x和VT-d（或AMD的SVM和IOMMU）硬件辅助虚拟化特性，还需要有支持SR-IOV规范的设备，当然也需要QEMU/KVM的支持。支持SR-IOV的设备较多，其中Intel有很多中高端网卡支持SR-IOV特性，如Intel 82576网卡（代号"Kawella"，使用igb驱动）、I350网卡（igb驱动）、82599网卡（代号"Niantic"，使用ixgbe驱动）、X540（使用ixgbe驱动）等。在宿主机Linux环境中，可以通过"lspci-v-s$BDF"的命令来查看网卡PCI信息的"Capabilities"项目，以确定设备是否具备SR-IOV的能力，如下命令行：

```
[root@jay-linux ~]# lspci -v -s 10:00.0

10:00.0 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)
      Subsystem: Intel Corporation Gigabit ET2 Quad Port Server Adapter
      Flags: bus master, fast devsel, latency 0, IRQ 39
      Memory at b2420000 (32-bit, non-prefetchable) [size=128K]
      Memory at b2000000 (32-bit, non-prefetchable) [size=4M]
        I/O ports at 2020 [size=32]
      Memory at b2444000 (32-bit, non-prefetchable) [size=16K]
      Capabilities: [40] Power Management version 3
      Capabilities: [50] MSI: Enable- Count=1/1 Maskable+ 64bit+
      Capabilities: [70] MSI-X: Enable+ Count=10 Masked-
      Capabilities: [a0] Express Endpoint, MSI 00
      Capabilities: [100] Advanced Error Reporting
      Capabilities: [140] Device Serial Number 00-1b-21-ff-ff-66-d8-ee
      Capabilities: [150] Alternative Routing-ID Interpretation (ARI)
      Capabilities: [160] Single Root I/O Virtualization (SR-IOV)
      Kernel driver in use: igb
      Kernel modules: igb
```

一个设备可支持多个VF，PCI-SIG的SR-IOV规范指出每个PF最多能拥有256个VF，而实际支持的VF数量是由设备的硬件设计及其驱动程序共同决定的。前面举例的几个网卡，其中使用"igb"驱动的82576、I350等千兆（1G）以太网卡的每个PF支持最多7个VF，而使用"ixgbe"驱动的82599、X540等万兆（10G）以太网卡的每个PF支持最多63个VF。在宿主机系统中可以用"modinfo"命令来查看某个驱动的信息，其中包括驱动模块的可用参数，如下命令行演示了常用igb和ixgbe驱动的信息。

```
[root@jay-linux kvm_demo]# modinfo igb
filename:       /lib/modules/3.5.0/kernel/drivers/net/ethernet/intel/igb/igb.ko
version:        3.4.7-k
license:        GPL
description:    Intel(R) Gigabit Ethernet Network Driver
author:         Intel Corporation, <e1000-devel@lists.sourceforge.net>
srcversion:     B905A7E217BFBA4F95FAEA8
alias:          pci:v00008086d000010D6sv*sd*bc*sc*i*
<!-- 这里省略了数十行输出信息 -->
alias:          pci:v00008086d00001539sv*sd*bc*sc*i*
depends:
intree:         Y
vermagic:       3.5.0 SMP mod_unload modversions
parm:           max_vfs:Maximum number of virtual functions to
allocate per physical function (uint)
parm:           debug:Debug level (0=none,...,16=all) (int)

[root@jay-linux kvm_demo]# modinfo ixgbe
filename:
/lib/modules/3.5.0/kernel/drivers/net/ethernet/intel/ixgbe/ixgbe.ko
```

```
version:        3.9.15-k
license:        GPL
description:    Intel(R) 10 Gigabit PCI Express Network Driver
author:         Intel Corporation, <linux.nics@intel.com>
srcversion:     BF985DC310192E0F0B49F50
alias:          pci:v00008086d0000154Asv*sd*bc*sc*i*
<!-- 这里省略了数十行输出信息 -->
alias:          pci:v00008086d000010B6sv*sd*bc*sc*i*
depends:        mdio,hwmon
intree:         Y
vermagic:       3.5.0 SMP mod_unload modversions
parm:           max_vfs:Maximum number of virtual functions to
allocate per physical function - default is zero and maximum value is 63 (uint)
parm:           allow_unsupported_sfp:Allow unsupported and
untested SFP+ modules on 82599-based adapters (uint)
parm:           debug:Debug level (0=none,...,16=all) (int)
```

在宿主机中，在加载支持SR-IOV的PCI设备的驱动时，一般还需要加上相应的参数来指定启用多少个VF，比如利用上面"modinfo"命令查看的igb和ixgbe驱动，其关于启用的VF个数的参数为"max_vfs"。如果当前系统还没有启用VF，则需要卸载掉驱动后重新加载驱动（加上VF个数的参数）来开启VF。如下命令行演示了开启igb驱动中VF个数的参数的过程及在开启VF之前和之后的系统中网卡的状态。

```
[root@jay-linux kvm_demo]# lspci | grep Eth
0d:00.0 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)
0d:00.1 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)
[root@jay-linux kvm_demo]# modprobe -r igb     #(或rmmod igb)
[root@jay-linux kvm_demo]# modprobe igb max_vfs=7
[root@jay-linux kvm_demo]# lspci | grep Eth
0d:00.0 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)
0d:00.1 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)
0e:10.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.1 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.3 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.5 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.6 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:10.7 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:11.0 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:11.1 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:11.2 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)

0e:11.3 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:11.4 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
0e:11.5 Ethernet controller: Intel Corporation 82576 Virtual Function (rev 01)
```

由上面的演示可知，BDF 0d:00.0和0d:00.1是PF，而在通过加了"max_vfs=7"的参数重新加载igb驱动后，对应的VF被启用了，每个PF启用了7个VF。为了让宿主机在启动时就能够默认自动开启VF，可以修改modprobe命令的配置文件，对于igb和ixgbe驱动，示例如下所示。

```
[root@jay-linux kvm_demo]#vim /etc/modprobe.d/igb.conf
option igb max_vfs=7
option ixgbe max_vfs=63
```

而通过如下的命令行可以查看到PF和VF的对应关系，以便清楚哪个VF是由哪个PF衍生而来的。

```
[root@jay-linux kvm_demo]# ls -l /sys/bus/pci/devices/0000\:0d\:00.0/virtfn*
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn0 -> ../0000:0e:10.0
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn1 -> ../0000:0e:10.2
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn2 -> ../0000:0e:10.4
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn3 -> ../0000:0e:10.6
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn4 -> ../0000:0e:11.0
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn5 -> ../0000:0e:11.2
lrwxrwxrwx 1 root root 0 Sep 26 17:13
/sys/bus/pci/devices/0000:0d:00.0/virtfn6 -> ../0000:0e:11.4
[root@jay-linux kvm_demo]# ls -l /sys/bus/pci/devices/0000\:0e\:10.0/physfn
lrwxrwxrwx 1 root root 0 Sep 26 17:14
/sys/bus/pci/devices/0000:0e:10.0/physfn -> ../0000:0d:00.0
```

其中，0d:00.0的VF为0e:10.0～0e:11.4之间的7个Function号为偶数的Virtual Function（还记得前面提及过的BDF的function位取值范围是0～7）。

另外，值得注意的是，由于VF还是共享和使用对应PF上的部分资源，因此要使SR-IOV的VF能够在客户机中工作，必须保证其对应的PF在宿主机中处于正常工作状态。

使用SR-IOV主要有如下3个优势：

1）真正实现了设备的共享（多个客户机共享一个SR-IOV设备的物理端口）；

2）接近于原生系统的高性能（比纯软件模拟和Virtio设备的性能都要好）。

3）相比于VT-d，SR-IOV可以用更少的设备支持更多的客户机，可以提高数据中心的空间利用率。

而SR-IOV的不足之处有如下两点：

1）对设备有依赖，只有部分PCI-E设备支持SR-IOV（如前面提到的Intel 82576、82599网卡）。

2）使用SR-IOV时，不方便动态迁移客户机（在5.4节“动态迁移”会介绍一种绕过这个问题的方法）。

2.SR-IOV操作示例

在了解了SR-IOV的基本原理及优劣势之后，本节将以一个完整的示例来介绍在KVM中使用SR-IOV的各个步骤。这给例子是这样的，在一个SandyBridge平台上，有一个两口的Intel 10G以太网卡（X540系列，使用ixgbe驱动），使用SR-IOV技术将其中的一个VF分配给一个RHEL 6.3的客户机使用。

1）重新加载ixgbe驱动，产生一定数量的VF，命令行操作如下：

```
[root@jay-linux kvm_demo]# modprobe -r ixgbe
[root@jay-linux kvm_demo]# modprobe ixgbe max_vf=3
[root@jay-linux kvm_demo]# ifconfig eth14
eth14   Link encap:Ethernet  HWaddr 00:1E:67:2B:5E:CC
        inet addr:192.168.97.45  Bcast:192.168.255.255  Mask:255.255.0.0
        inet6 addr: fe80::21e:67ff:fe2b:5ecc/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:380086 errors:0 dropped:1 overruns:0 frame:0
        TX packets:1524 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:49172444 (46.8 MiB)  TX bytes:435180 (424.9 KiB)

[root@jay-linux kvm_demo]# ethtool -i eth14
driver: ixgbe
version: 3.9.15-k
firmware-version: 0x8000032e
bus-info: 0000:06:00.1
[root@jay-linux kvm_demo]# lspci | grep Eth
06:00.0 Ethernet controller: Intel Corporation Ethernet
Controller 10-Gigabit X540-AT2 (rev 01)
06:00.1 Ethernet controller: Intel Corporation Ethernet
Controller 10-Gigabit X540-AT2 (rev 01)
06:10.0 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
06:10.1 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
06:10.2 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
06:10.3 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
06:10.4 Ethernet controller: Intel Corporation X540 Ethernet

Controller Virtual Function (rev 01)
06:10.5 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
[root@jay-linux kvm_demo]# ls -l/sys/bus/pci/devices/0000\:06\:00.1/virtfn*
lrwxrwxrwx 1 root root 0 Sep 27 17:22
/sys/bus/pci/devices/0000:06:00.1/virtfn0 -> ../0000:06:10.1
lrwxrwxrwx 1 root root 0 Sep 27 17:22
/sys/bus/pci/devices/0000:06:00.1/virtfn1 -> ../0000:06:10.3
lrwxrwxrwx 1 root root 0 Sep 27 17:22
/sys/bus/pci/devices/0000:06:00.1/virtfn2 -> ../0000:06:10.5
```

由以上输出信息可知，两个PF分别为06:00.0和06:00.1，它们对应的VF分别为06:10.0、06:10.2、06:10.4和06:10.1、06:10.3、06:10.5。06:00.1这个PF在宿主机中是正常工作的，可以看到它有对应的IP地址。

2）将其中一个VF（06:10.1）隐藏，以供客户机使用，命令行操作如下：

```
[root@jay-linux kvm_demo]# lspci -k -s 06:10.1
06:10.1 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
       Subsystem: Intel Corporation Device 35a0
[root@jay-linux kvm_demo]# ./pcistub.sh -h 06:10.1
0000:06:10.1 wasn't bound to any drirver
Binding 0000:06:10.1 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 06:10.1
06:10.1 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
      Subsystem: Intel Corporation Device 35a0
      Kernel driver in use: pci-stub
```

这里隐藏的06:10.1这个VF对应的PF是06:00.1，该PF处于可用状态，才能让VF能在客户机中正常工作。其实这里的06:10.1 VF在使用pcistub.sh脚本前也没有在系统中加载ixgbevf驱动，它没有处于使用中状态（lspci-k-s$BDF命令的输出中没有"Kernel driver in use"这一行），这时不需要用pci_stub模块去隐藏它，也是可以直接将它分配给客户机使用的。而笔者依然使用pci_stub去隐藏它，其中一个好

处是标识它处于隐藏中，可能已被分配给客户机使用，另一个好处是宿主机加载ixgbevf驱动也对这个已被隐藏的VF没有影响。

3）在命令行启动客户机时分配一个VF网卡，命令行操作如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -device pci-assign,host=08:00.0 -net none
VNC server running on '::1:5900'
```

4）在客户机中，查看VF的工作情况，命令行操作如下：

```
[root@kvm-guest ~]# lspci -k -s 00:03.0
00:03.0 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
        Subsystem: Intel Corporation Device 35a0
```

```
        Kernel driver in use: ixgbevf
        Kernel modules: ixgbevf
[root@kvm-guest ~]# lsmod | grep ixgbevf
ixgbevf                37948  0
[root@kvm-guest ~]# ethtool -i eth2
driver: ixgbevf
version: 2.2.0-k
firmware-version:
bus-info: 0000:00:03.0
[root@kvm-guest ~]# ifconfig eth2
eth2      Link encap:Ethernet  HWaddr D2:15:94:28:83:09
          inet addr:192.168.124.229  Bcast:192.168.255.255  Mask:255.255.0.0
          inet6 addr: fe80::d015:94ff:fe28:8309/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1041459 errors:0 dropped:0 overruns:0 frame:0
          TX packets:264534 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1556040201 (1.4 GiB)  TX bytes:19054060 (18.1 MiB)

[root@kvm-guest ~]# ping 192.168.199.98-c 1
PING 192.168.199.98 (192.168.199.98) 56(84) bytes of data.
64 bytes from 192.168.199.98: icmp_seq=1 ttl=64 time=0.589 ms

--- 192.168.199.98 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 676ms
rtt min/avg/max/mdev = 1.794/1.794/1.794/0.000 ms
```

由上面的输出信息可知，00:03.0就是那个VF，它的PCI信息与在宿主机中看到的是一致的（X540 Ethernet Controller Virtual Function），使用的是ixgbevf驱动，而且它获得了IP地址，网络连接也是畅通的。不过，有时可能遇到客户机中VF的网络并没有连通的情况，可能需要重新加载对应的驱动程序（如igbvf、ixgbevf等）。

3.SR-IOV使用问题解析

在使用SR-IOV时，可能也会遇到各种小问题，本节笔者根据经验来介绍一些可能会遇到的问题及其解决方法。

（1）VF在客户机中MAC地址全为零

如果使用Linux 3.9及之后的版本作为宿主机的内核，则可能会在使用igb或ixgbe驱动的网卡（如Intel 82576、I350、82599等）的VF进行SR-IOV时，在客户机中看到igbvf或ixgbevf网卡的MAC地址全为零（即00:00:00:00:00:00），从而导致VF不能正常工作。比如，在一个Linux客户机的dmesg命令的输出信息中，可能会看到如下的错误信息：

```
igbvf 0000:00:03.0: irq 26 for MSI/MSI-X
igbvf 0000:00:03.0: Invalid MAC Address: 00:00:00:00:00:00
igbvf: probe of 0000:00:03.0 failed with error -5
```

关于这个问题，笔者曾向Linux/KVM社区报过一个bug，其网页链接为：https://bugzilla.kernel.org/show_bug.cgi?id=55421。

这个问题的原因是，从Linux 3.9开始内核代码中的igb或ixgbe驱动程序在进行SR-IOV时，会将VF的MAC地址设置为全是零，而不是像之前那样使用一个随机生成的MAC地址。这样调整主要也是为了解决两个问题：一是随机的MAC地址对Linux内核中的设备管理器udev很不友好，多次使用VF可能会导致VF在客户机中的以太网络接口名称持续变化（如可能变为eth100）；二是随机生成的MAC地址并不能完全保证其唯一性，有很小的概率可能与其他网卡的MAC地址重复而产生冲突。

对于VF的MAC地址全为零的问题，可以通过如下两种方法之一来解决。

❑在分配VF给客户机之前，在宿主机中用ip命令来设置需要使用的VF的MAC地址，命令行操作实例如下：

```
[root@jay-linux ~]# ip link set eth0 vf 0 mac 00:1E:67:65:93:01
```

在上面的命令中，eth0为宿主机中PF对应的以太网接口名称，0代表设置的VF是该PF的编号为0的VF（即第一个VF）。那么，如何确定这个VF编号对应的PCI-E设备的BDF编号呢？可以使用如下的两个命令来查看PF和VF的关系。

```
[root@jay-linux ~]# ethtool -i eth0
driver: igb
version: 4.1.2-k
firmware-version: 1.64, 0x800006fc
bus-info: 0000:0a:00.0
supports-statistics: yes
supports-test: yes
supports-eeprom-access: yes
supports-register-dump: yes
supports-priv-flags: no
[root@jay-linux ~]# ls -l \ /sys/bus/pci/devices/0000:0a:00.0/virtfn*
lrwxrwxrwx 1 root root 0 Apr 23 15:09 /sys/bus/pci/devices/0000:0a:00.0/virtfn0
-> ../0000:0b:10.0
lrwxrwxrwx 1 root root 0 Apr 23 15:09 /sys/bus/pci/devices/0000:0a:00.0/virtfn1
-> ../0000:0b:10.4

<!-- 此处省略其余 VF 的对应关系 -->
```

❑可以升级客户机系统的内核或VF驱动程序，比如可以将Linux客户机升级到使用Linux 3.9之后的内核及其对应的igbvf驱动程序。最新的igbvf驱动程序可以处理VF的MAC地址为零的情况。

（2）Windows客户机中关于VF的驱动程序

对于Linux系统，宿主机中PF使用的驱动（如igb、ixgbe等）与客户机中VF使用的驱动（如igbvf、ixgbevf等）是不同的。当前流行的Linux发行版（如RHEL、Fedora、Ubuntu等）中都默认带有这些驱动模块。而对于Windows客户机系统，Intel网卡（如82576、82599等）的PF和VF驱动是同一个，只是分为32位和64位系统两个版本。有少数的最新Windows系统（如Windows 8、Windows 2012 Server等）默认带有这些网卡驱动，而多数的Windows系统（如Windows 7、Windows 2008 Server等）都没有默认带有相关的驱动，需要自行下载安装，例如前面提及的Intel网卡驱动可以到其官方网站（http://downloadcenter.intel.com/Default.aspx）下载。

（3）少数网卡的VF在少数Windows客户机中不工作

读者在进行SR-IOV的实验时，可能遇到VF在某些Windows客户机中不工作的情况。笔者就遇到这样的情况，在用默认的qemu-kvm命令行（如步骤3所示）启动客户机后，Intel的82576、82599网卡的VF在32位Windows 2008Server版的客户机中不能正常工作，而在64位客户机中的工作正常。该问题的原因不在于Intel的驱动程序也不在于KVM中SR-IOV的逻辑不正确，而是在于默认的CPU模型是qemu64，它不支持MSI-X这种中断方式，而32位的Windows 2008 Server版本中的82576、82599网卡的VF只能用MSI-X中断方式来工作。所以，需要在通过命令

行启动客户机时指定QEMU模拟的CPU的类型（在4.2.4节中介绍过），从而可以绕过这个问题。可以用"-cpu SandyBridge"、"-cpu Westmere"等参数来指定CPU类型，也可以用"-cpu host"参数来尽可能多地将物理CPU信息暴露给客户机，还可以用"-cpu qemu64,model=13"这样来改变qemu64类型CPU的默认模型。通过命令行启动一个32位Windows 2008 Server客户机，并分配一个VF给它使用，命令行操作如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 32bit-win2k8.img
-smp 2 -cpu SandyBridge -m 1024 -device pci-assign,host=08:00.0 -net none
```

在客户机中，网卡设备正常显示，网络连通状态正常，如图5-13所示。

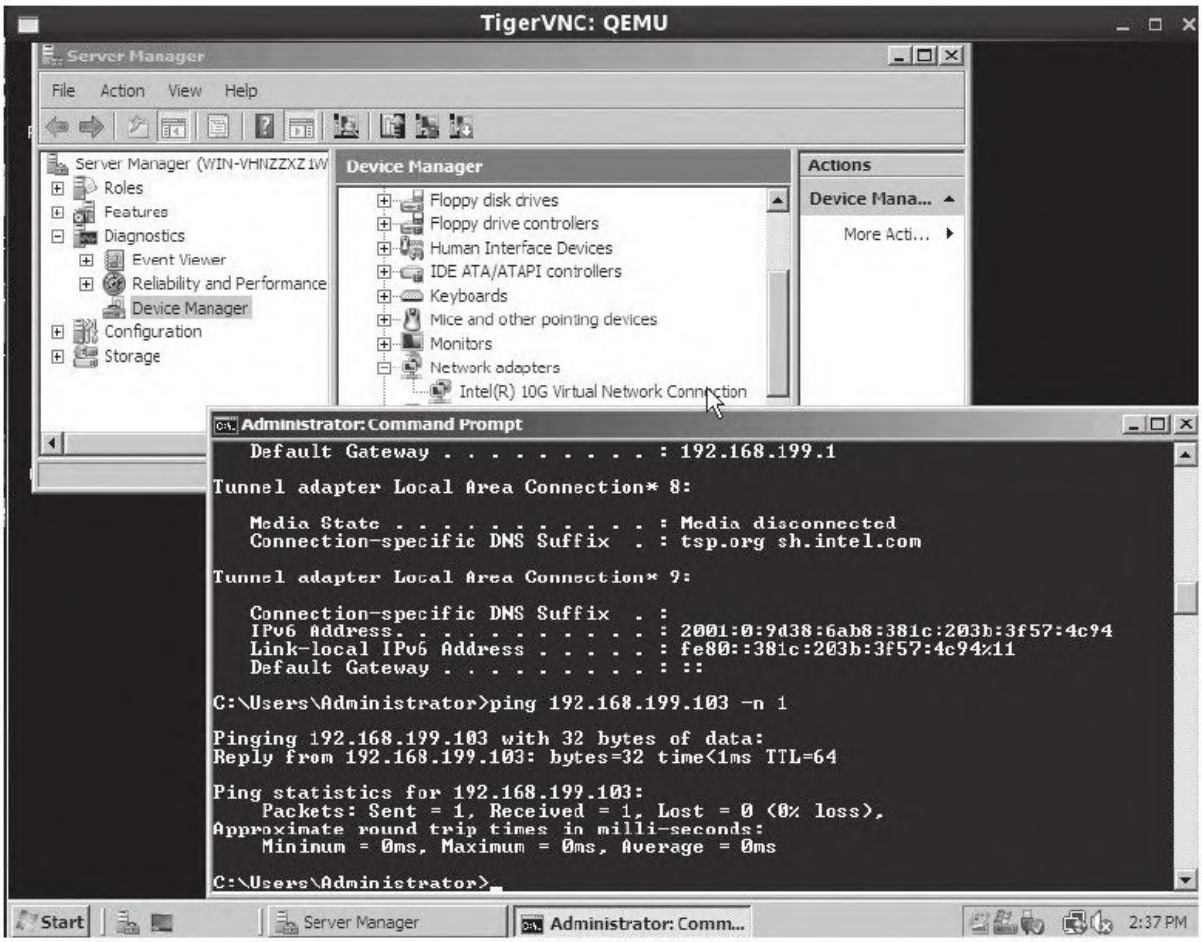


图 5-13 32位Windows 2008 Server客户机中82599 VF的网络状态

5.3 热插拔

热插拔（hot plugging）即“带电插拔”，指可以在电脑运行时（不关闭电源）插上或拔除硬件。热插拔最早出现在服务器领域，是为了提高服务器扩展性、灵活性和对灾难的及时恢复能力。实现热插拔需要有几方面支持：总线电气特性、主板BIOS、操作系统和设备驱动。目前，在服务器硬件中，可实现热插拔的部件主要有SATA硬盘（IDE不支持热插拔）、CPU、内存、风扇、USB、网卡等。在KVM虚拟化环境中，在不关闭客户机的情况下，也可以对客户机的设备进行热插拔。目前，KVM对热插拔的支持还不够完善，主要支持PCI设备和CPU的热插拔，也可以通过ballooning间接实现内存的热插拔。

5.3.1 PCI设备热插拔

前面5.2中介绍的VT-d设备直接分配和SR-IOV技术时都是在客户机启动时就分配相应的设备，本节将介绍可以通过热插拔来添加或删除这些PCI设备。QEMU/KVM不仅支持动态添加和动态移除设备，而且在启动客户机的qemu-kvm命令行中分配的普通VT-d设备或SR-IOV的VF设备也可以被动态移除。

PCI设备的热插拔，主要需要如下几个方面的支持。

（1）BIOS

QEMU/KVM默认使用SeaBIOS[9] 作为客户机的BIOS，该BIOS文件路径一般为/usr/local/share/qemu/bios.bin，目前默认的BIOS已经可以支持PCI设备的热插拔。

（2）PCI总线

物理硬件中必须有VT-d的支持，且现在的PCI、PCIe总线都支持设备的热插拔。

（3）客户机操作系统

多数流行的Linux和Windows操作系统都支持设备的热插拔。可以在客户机的Linux系统的内核配置文件中看到一些相关的配置，如下是RHEL 6.3系统中的部分相关配置。

```
CONFIG_HOTPLUG=y
CONFIG_HOTPLUG_PCI_PCIE=y
CONFIG_HOTPLUG_PCI=y
CONFIG_HOTPLUG_PCI_FAKE=m
CONFIG_HOTPLUG_PCI_ACPI=y
CONFIG_HOTPLUG_PCI_ACPI_IBM=m
```

（4）客户机中的驱动程序

一些网卡驱动（如Intel的e1000e、igb、ixgbe、igbvf、ixgbevf等）、SATA或SAS磁盘驱动、USB2.0、USB3.0驱动都支持设备的热插拔。注意，在一些较旧的Linux系统（如RHEL 5.5）中需要加载"acpiphp"（使用"modprobe acpiphp"命令）这个模块后才支持设备的热插拔，否则热插拔完全不会对客户机系统生效；而较新内核的Linux系统（如RHEL 6.3、Fedora 17等）中已经没有该模块，不需要加载该模块，默认启动的系统就支持设备热插拔。

有了BIOS、PCI总线、客户机操作系统和驱动程序的支持后，热插拔功能只需要在QEMU monitor中的两个命令即可完成热插拔功能。将一个BDF为02:00.0的PCI设备动态添加到客户机中（设置id为mydevice），在monitor中的命令如下：

```
device_add pci-assign,host=02:00.0,id=mydevice
```

将一个设备（id为mydevice）从客户机中动态移除，在monitor中的命令如下：

```
device_del mydevice
```

这里的mydevice是在添加设备时设置的唯一标识，可以通过"info pci"命令在QEMU monitor中查看到当前的客户机中的PCI设备及其id值。在5.2.2节中也已经提及，在命令行启动客户机时分配设备也可以

设置这个id值，如果这样，那么也就可以用"device_del id"命令将该PCI设备动态移除。

如果在动态添加PCI设备后，在客户机中用"lspci"命令可以查看到该设备，但是该设备实际并不可以使用，这有可能会是宿主机内核的一个bug。这需要在客户机卸载驱动后重新加载该设备的驱动才能让设备正常工作，笔者就曾经遇到个这样的一个bug（https://bugzilla.kernel.org/show_bug.cgi?id=47451）。

5.3.2 PCI设备热插拔示例

在介绍了PCI设备热插拔所需的必要条件和操作命令之后，本节分别以网卡、U盘、SATA硬盘的热插拔为例来演示具体的操作过程。

1.网卡的热插拔

1）启动一个客户机，不向它分配任何网络设备，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -smp 2 -net none
VNC server running on '::1:5900'
```

2）选择并用pci-stub隐藏一个网卡设备供热插拔使用，命令行如下：

```
[root@jay-linux kvm_demo]# lspci -s 06:10.1
06:10.1 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
[root@jay-linux kvm_demo]# ./pcistub.sh -h 06:10.1
Unbinding 0000:06:10.1 from ixgbevf
Binding 0000:06:10.1 to pci-stub
```

这里选取了Intel X540网卡的一个SR-IOV VF作为热插拔的设备。

3）切换到QEMU monitor中，将网卡动态添加到客户机中，命令如下所示。一般可以用"Alt+Ctrl+2"快捷键进入到monitor中，也可以在启动时添加参数"-monitor stdio"将monitor定向到当前终端的标准输入输出中直接进行操作。

```
(qemu) device_add pci-assign,host=06:10.1,id=mynic
```

4）在QEMU monitor中查看客户机的PCI设备信息，命令如下：

```
(qemu) info pci
  Bus  0, device   0, function 0:
    Host bridge: PCI device 8086:1237
      id ""
<!-- 此处省略多行其他 PCI 设备的信息输出  -->
  Bus  0, device   3, function 0:
    Ethernet controller: PCI device 8086:1515
      BAR0: 32 bit memory at 0x40000000 [0x40003fff].
      BAR3: 32 bit memory at 0x40004000 [0x40007fff].
      id "mynic"
```

由以上信息可知，"Bus 0,device 3,function 0"的设备就是动态添加的网卡设备。

5）在客户机中检查动态添加和网卡工作情况，命令行如下：

```
[root@kvm-guest ~]# lspci | grep Eth
00:03.0 Ethernet controller: Intel Corporation X540 Ethernet
Controller Virtual Function (rev 01)
[root@kvm-guest ~]# ethtool -i eth2
driver: ixgbevf
version: 2.2.0-k
firmware-version:
bus-info: 0000:00:03.0
[root@kvm-guest ~]# ifconfig eth2
eth2    Link encap:Ethernet  HWaddr D2:15:94:28:83:09
        inet addr:192.168.124.229  Bcast:192.168.255.255  Mask:255.255.0.0
        inet6 addr: fe80::d015:94ff:fe28:8309/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:644 errors:0 dropped:0 overruns:0 frame:0
        TX packets:107 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:116378 (113.6 KiB)  TX bytes:19689 (19.2 KiB)

[root@kvm-guest ~]# ping 192.168.199.103 -c 1 -I eth2
PING 192.168.199.103 (192.168.199.103) from 192.168.124.229
eth2: 56(84) bytes of data.
64 bytes from 192.168.199.103: icmp_seq=1 ttl=64 time=1.34 ms

--- 192.168.199.103 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 1ms
rtt min/avg/max/mdev = 1.348/1.348/1.348/0.000 ms
```

由以上输出信息可知，动态添加的网卡是客户机中唯一的网卡设备，其网络接口名称为"eth2"，它的网络连接是通畅的。

6）将刚添加的网卡动态地从客户机中移除，命令行如下：

```
(qemu) device_del mynic
```

将网卡动态移除后，在monitor中用"info pci"命令将没有刚才的PCI网卡设备信息，在客户机中"lspci"命令也不能看到客户机中有网卡设备的信息。

2.USB设备的热插拔

USB设备是现代计算机系统中比较重要的一类设备，包括USB的键盘和鼠标、U盘，还有现在网上银行经常可能用到的USB认证设备（如工商银行的“U盾”）。如本小节前面讲到的那样，USB设备也可以像普通PCI设备那样进行VT-d设备直接分配，而在热插拔方面也是类似的。下面以U盘的热插拔为例来介绍一下操作过程。

U盘的热插拔操作步骤和前面介绍网卡热插拔的步骤基本是一致的，只是需要注意：qemu-kvm默认没有向客户机提供USB总线，需要在启动客户机的qemu-kvm命令行中添加"-usb"参数（或"-device piix3-usb-uhci"参数）来提供客户机中的USB总线。另外，对于USB设备，在QEMU monitor中除了可以用"device_add"和"device_del"命令之外，也有两个专门的命令（usb_add和usb_del）用于对USB进行热插拔操作。

1）查看宿主机中的USB设备情况，然后启动一个带有USB总线控制器的客户机，命令行如下：

```
[root@jay-linux kvm_demo]# lsusb
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 004: ID 0781:5567 SanDisk Corp. Cruzer Blade
Bus 002 Device 003: ID 046b:ff10 American Megatrends, Inc. Virtual
Keyboard and Mouse
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -smp 2 -net none
VNC server running on '::1:5900'
```

2）切换到QEMU monitor窗口，动态添加SanDisk的U盘给客户机，使用"usb_add"命令行如下：

```
(qemu)usb_add host:002.004
```

或者

```
(qemu)usb_add host:0781:5567
```

而像5.2.3节中“VT-d操作示例”中那样查找到宿主机中USB controller对应的PCI BDF，对其进行隐藏，然后使用device_add命令动态添加设备的命令如下：

```
(qemu) device_add pci-assign,host=00:1d.0,id=myusb
```

解释一下"usb_add"这个用于动态添加一个USB设备的命令，在monitor中命令格式如下：

usb_add *devname*

其中devname是对该USB设备的唯一标识，该命令支持两种devname的格式：一种是USB hub中的Bus和Device号码的组合，一种是USB的vendor ID和device ID的组合（在5.2.2小节“VT-d环境配置”中也曾提及过）。举个例子，对于该宿主机中的一个SanDisk的U盘设备（前一步的lsusb命令），devname可以设置为“002.004”和“0781:5567”两种格式。另外，需要像上面命令行操作的那样，用"host:002.004"或"host:0781:5567"来指定分配宿主机中的USB设备给客户机。

3）在客户机中，查看动态添加的USB设备，命令行如下：

```
[root@kvm-guest ~]# lsusb
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 002: ID 0781:5567 SanDisk Corp. Cruzer Blade
[root@kvm-guest ~]# fdisk -l /dev/sdb

Disk /dev/sdb: 16.0 GB, 16008609792 bytes
21 heads, 14 sectors/track, 106349 cylinders
Units = cylinders of 294 * 512 = 150528 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xcad4ebea

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           7      106350    15632384    c  W95 FAT32 (LBA)
```

可见，USB设备已经添加成功了，在客户机中可以正常使用该U盘了。

4）在QEMU monitor中查看USB设备，然后动态移除USB设备命令行操作如下：

```
(qemu) info usb
  Device 0.2, Port 1, Speed 480 Mb/s, Product Cruzer Blade
(qemu) usb_del 0.2
(qemu) info usb
```

由上面的输出信息可知，移除前，"info usb"命令可以看到USB设备，在用"usb_del"命令移除后，"info usb"就没有查看到任何USB设备了。注意，usb_del命令后的参数是"info usb"命令查询出来的"Device"后的地址标识，这里为“0.2”。笔者发现在qemu-kvm 1.1版本中有个bug，在"usb_del 0.2"命令执行后，客户机qemu-kvm进程发生core dumped的错误，然后客户机进程被杀掉，换用较新的qemu-kvm1.2版本就正常操作了。

当然，如果使用device_add命令动态添加的USB设备，则使用如下device_del命令将其移除：

```
(qemu) device_del myusb
```

3.SATA硬盘的热插拔

与5.2.3小节“VT-d操作示例”类似，在本小节的示例中，宿主机从一台机器上的SAS硬盘启动，然后将SATA硬盘动态添加给客户机使用，接着动态移除该硬盘。

1）检查宿主机系统，得到需要动态热插拔的SATA硬盘（实际上用的是整个SATA控制器），并将其用pci-stub模块隐藏起来以供热插拔使用，命令行操作如下：

```
[root@jay-linux kvm_demo]# lspci | grep SATA
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family)

SATA AHCI Controller
[root@jay-linux kvm_demo]# lspci | grep SAS
16:00.0 SCSI storage controller: LSI Logic / Symbios Logic SAS1078
PCI-Express Fusion-MPT SAS (rev 04)
[root@jay-linux kvm_demo]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             197G   76G  112G  41% /
tmpfs                  12G   76K   12G   1% /dev/shm
[root@jay-linux kvm_demo]# ll
/dev/disk/by-path/pci-0000\:16\:00.0-sas-0x1221000000000000-lun-0
lrwxrwxrwx 1 root root 9 Oct 29 15:28
/dev/disk/by-path/pci-0000:16:00.0-sas-0x1221000000000000-lun-0 -> ../../sda
[root@jay-linux kvm_demo]# ll
/dev/disk/by-path/pci-0000\:00\:1f.2-scsi-0\:0\:0\:0
lrwxrwxrwx 1 root root 9 Oct 29 15:28
/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 -> ../../sdb
[root@jay-linux kvm_demo]# lspci -k -s 00:1f.2
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
       Subsystem: Intel Corporation Device 34f8
       Kernel driver in use: ahci
       Kernel modules: ahci
[root@jay-linux kvm_demo]#./pcistub.sh -h 00:1f.2
Unbinding 0000:00:1f.2 from ahci
Binding 0000:00:1f.2 to pci-stub
[root@jay-linux kvm_demo]# lspci -k -s 00:1f.2
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
       Subsystem: Intel Corporation Device 34f8
       Kernel driver in use: pci-stub
       Kernel modules: ahci
```

2）启动一个客户机，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -smp 2 -net nic -net tap
VNC server running on '::1:5900'
```

3）在QEMU monitor中，动态添加该SATA硬盘，命令行如下：

```
(qemu) device_add pci-assign,host=00:1f.2,id=sata,addr=0x06
(qemu) info pci    # 查看客户机中 pci 设备，可以看到动态添加的 SATA 控制器
  Bus  0, device   6, function 0:
    SATA controller: PCI device 8086:3a22
      IRQ 9.
      BAR0: I/O at 0x1020 [0x1027].
      BAR1: I/O at 0x1030 [0x1033].
      BAR2: I/O at 0x1028 [0x102f].
      BAR3: I/O at 0x1034 [0x1037].

 BAR4: I/O at 0x1000 [0x101f].
 BAR5: 32 bit memory at 0x40000000 [0x400007ff].
 id "sata"
```

4）在客户机中查看动态添加的SATA硬盘，命令行如下：

```
[root@kvm-guest ~]#fdisk -l /dev/sdb

Disk /dev/sdb: 164.7 GB, 164696555520 bytes
255 heads, 63 sectors/track, 20023 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0003e001

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *           1        6528    52428800   83  Linux
/dev/sdb2            6528        7050     4194304   82  Linux swap / Solaris
/dev/sdb3            7050        9600    20480000   83  Linux

[root@kvm-guest ~]#lspci -k -s 00:06.0
00:06.0 SATA controller: Intel Corporation 82801JI (ICH10 Family)
SATA AHCI Controller
       Subsystem: Intel Corporation Device 34f8
       Kernel driver in use: ahci
       Kernel modules: ahci
```

由以上信息可知，客户已经能够获取到SATA硬盘（/dev/sdb）的信息，然后就可以正常使用动态添加的该硬盘了。

5）在客户机中使用完SATA硬盘后，可以动态移除SATA硬盘，在QEMU monitor中命令行如下：

```
(qemu) device_del sata
```

在动态移除SATA硬盘后，客户机中将没有SATA硬盘的设备，宿主机又可以控制SATA硬盘，将其用于其他用途（包括分配给另外的客户机使用）。

5.3.3 CPU和内存的热插拔

CPU和内存的热插拔是RAS（Reliability、Availability和Serviceability）的一个重要特性，在非虚拟化环境中，只有较少的x86服务器硬件支持CPU和内存的热插拔（笔者曾在Intel的Westmere-EX平台上做过物理CPU和内存的热插拔）。在操作系统方面，拥有较新内核的Linux系统（如RHEL6.3）等已经支持CPU和内存的热插拔，在其内核配置文件中可以看到类似如的下选项。

```
CONFIG_HOTPLUG=y
CONFIG_MEMORY_HOTPLUG=y
CONFIG_HOTPLUG_CPU=y

CONFIG_ARCH_ENABLE_MEMORY_HOTPLUG=y
CONFIG_ARCH_ENABLE_MEMORY_HOTREMOVE=y
CONFIG_ACPI_HOTPLUG_CPU=y
CONFIG_ACPI_HOTPLUG_MEMORY=y
```

而在QEMU/KVM虚拟化环境中，对CPU和内存的热插拔支持也不完善。CPU的热插拔功能有一段时间在QEMU/KVM中是可以工作的，不过笔者写作本书时（Linux 3.6和qemu-kvm 1.2），该功能依然是不可用状态[10] 。下面简单介绍一下CPU热插拔的操作步骤。

1）在qemu-kvm命令行中启动客户机时，使用"-smp n,maxvcpus=N"的参数，如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -smp 2,maxvcpus=8 -net nic -net tap
VNC server running on '::1:5900'
```

这就是在客户机启动时使用的两个vCPU，而最多支持客户机动态添加到8个vCPU。

2）在客户机中检查CPU的状态，如下：

```
[root@kvm-guest ~]# ls /sys/devices/system/cpu/
cpu0  cpu1  cpufreq  cpuidle  kernel_max  offline  online  possible  present
```

3）通过QEMU monitor中的"cpu_set n online"命令为客户机添加n个vCPU，如下：

```
(qemu) cpu_set 4 online
```

而动态移除n个vCPU的命令为"cpu_set n offline"。

4）检查客户机中vCPU的数量是否与预期的相符，如果看到"/sys/devices/system/cpu/"目录下CPU的数量增加（或减少）了n个，则表示操作成功了。另外，如果是动态添加CPU，客户机中新增的CPU没有自动上线工作，可以用"echo 1 ＞/sys/devices/system/cpu/cpu2/online"命令使其进入可用状态。

对于内存的热插拔，尽管在2012年有社区贡献者发了一些补丁试图去支持它，还是没有被完全接受，在笔者写作本书时QEMU/KVM还没

有支持该功能。而使用ballooning的方式（详见5.1.3小节），也是可以动态改变客户机中可用的内存大小，可以算是间接地实现内存热插拔功能。

另外，据笔者了解，QEMU/KVM社区也考虑将CPU和内存的热插拔做成普通PCI/PCI-e设备热插拔的形式，但没有具体的时间点实现该功能，目前也没有太多的补丁去实现这个功能。

5.4 动态迁移

5.4.1 动态迁移的概念

本节将会从迁移的基本概念入手，然后介绍有无虚拟化环境下的迁移，最后详细介绍动态迁移。

迁移（migration）包括系统整体的迁移和某个工作负载的迁移。系统整体迁移，是将系统上的所有软件（也包括操作系统）完全复制到另一台物理硬件机器之上。而工作负载的迁移，是将系统上的某个工作负载转移到另一台物理机器上继续运行。服务器系统迁移的作用在于简化了系统维护管理，提高了系统负载均衡，增强了系统容错性并优化了系统电源管理。

虚拟化的概念和技术的出现，给迁移带来了更丰富的含义和实践。在传统应用环境中，没有虚拟化技术的支持，系统整体的迁移主要都是静态迁移。这种迁移主要依靠系统备份和恢复技术，将系统的软件完全复制到另一台机器上，可以通过先做出系统的镜像文件，然后复制到其他机器上，或者通过直接的硬盘相互复制来实现迁移的目的。在非虚拟化环境中也有动态迁移的概念，但都是对某个（或某一组）工作负载的迁移，需要特殊系统的支持才能实现，而且技术也不够成熟，如哥伦比亚大学的Zap[11] 系统，它通过在操作系统上提供了

一个很薄虚拟化层（这和现在主流的虚拟化技术不一样），可以实现将工作负载迁移到另一台机器上。

在虚拟化环境中的迁移，又分为静态迁移（static migration）和动态迁移（live migration），也有少部分人称之为冷迁移（cold migration）和热迁移（hot migration），或者离线迁移（offline migration）和在线迁移（online migration）。静态迁移和动态迁移最大的区别就是，静态迁移有明显一段时间客户机中的服务不可用，而动态迁移则没有明显的服务暂停时间。虚拟化环境中的静态迁移，也可以分为两种，一种是关闭客户机后，将其硬盘镜像复制到另一台宿主机上然后恢复启动起来，这种迁移不能保留客户机中运行的工作负载；另一种是两台宿主机共享存储系统，只需要在暂停（而不是完全关闭）客户机后，复制其内存镜像到另一台宿主机中恢复启动，这种迁移可以保持客户机迁移前的内存状态和系统运行的工作负载。

动态迁移，是指在保证客户机上应用服务正常运行的同时，让客户机在不同的宿主机之间进行迁移，其逻辑步骤与前面静态迁移几乎一致，有硬盘存储和内存都复制的动态迁移，也有仅复制内存镜像的动态迁移。不同的是，为了保证迁移过程中客户机服务的可用性，迁移过程仅有非常短暂的停机时间。动态迁移允许系统管理员将客户机在不同物理机上迁移，同时不会断开访问客户机中服务的客户端或应用程序的连接。一个成功的动态迁移，需要保证客户机的内存、硬盘

存储、网络连接在迁移到目的主机后依然保持不变，而且迁移过程的服务暂停时间较短。

另外，对于虚拟化环境的迁移，不仅包括相同Hypervisor之间的客户机迁移（如KVM迁移到KVM、Xen迁移到Xen），还包括不同Hypervisor之间的客户机迁移（如Xen迁移到KVM、VMware迁移到KVM等，5.11节会详细介绍）。

5.4.2 动态迁移的效率和应用场景

虚拟机迁移主要增强了系统的可维护性，其主要目标是在客户没有感觉的情况下，将客户机迁移到了另一台物理机器上，并保证其各个服务都正常使用。可以从如下几个方面来衡量虚拟机迁移的效率。

1）整体迁移时间：从源主机（source host）中迁移操作开始到客户机被迁移到目的主机（destination host）并恢复其服务所花费的时间。

2）服务器停机时间（service down-time）：在迁移过程中，源主机和目的主机上客户机的服务都处于不可用状态的时间，此时源主机上客户机已暂停服务，目的主机上客户机还未恢复服务。

3）对服务的性能影响：不仅包括迁移后的客户机中应用程序的性能与迁移前相比是否有所降低，还包括迁移后对目的主机上的其他服

务（或其他客户机）的性能影响。

动态迁移的整体迁移时间受诸多因素的影响，如Hypervisor和迁移工具的种类、磁盘存储的大小（如果需要复制磁盘镜像）、内存大小及使用率、CPU的性能及利用率、网络带宽大小及是否拥塞等，整体迁移时间一般为几秒钟到几十分钟不等。动态迁移的服务停机时间，也是受Hypervisor的种类、内存大小、网络带宽等因素的影响，服务停机时间一般在几毫秒到几秒钟不等。其中，服务停机时间在几毫秒到几百毫秒，而且在终端用户毫无察觉的情况下实现迁移，这种动态迁移也被称为无缝的动态迁移。而静态迁移的服务暂停时间一般都较长，少则几秒钟，多则几分钟，需要依赖于管理员的操作速度和CPU、内存、网络等硬件设备。所以说，静态迁移一般适合于对服务可用性要求不高的场景，而动态迁移的停机时间很短，适合对服务可用性要求较高的场景。动态迁移一般对服务的性能影响不大，这与两台宿主机的硬件配置情况、Hypervisor是否稳定等因素相关。

动态迁移的好处是非常明显的了，下面来看一下动态迁移的几个应用场景。

1）负载均衡：当一台物理服务器的负载较高时，可以将其上运行的客户机动态迁移到负载较低的宿主机服务器中，以保证客户机的服务质量（QoS）。而前面提到的，CPU、内存的过载使用可以解决某些客户机的资源利用问题，之后当物理资源长期处于超负荷状态时，对

服务器稳定性能和服务质量都有损害的，这时需要动态迁移来进行适当的负载均衡。

2）解除硬件依赖：当系统管理员需要在宿主机上升级、添加、移除某些硬件设备的时候，可以将该宿主机上运行的客户机非常安全高效地动态迁移到其他宿主机上。在系统管理员升级硬件系统之时，使用动态迁移，可以让终端用户完全感知不到服务有任何暂停时间。

3）节约能源：在目前的数据中心的成本支出中，其中有一项重要的费用是电能的开销。当有较多服务器的资源使用率都偏低时，可以通过动态迁移将宿主机上的客户机集中迁移到其中几台服务器上，而在某些宿主机上的客户机完全迁移走之后，就可以将关闭电源以节省电能消耗，从而降低数据中心的运营成本。

4）实现客户机地理位置上的远程迁移：假设某公司的运行某类应用服务的客户机本来仅部署在上海电信的IDC中，然后发现来自北京及其周边地区的网通用户访问量非常大，但是由于距离和网络互联带宽拥堵（如电信与网通之间的带宽），北方用户使用该服务的网络延迟较大，这时系统管理员可以将上海IDC中的部分客户机通过动态迁移部署到位于北京的北京网通的IDC中，从而让终端用户使用该服务的质量更高。

5.4.3　KVM动态迁移原理和实践

在KVM中，既支持离线的静态迁移，又支持在线的动态迁移。对于静态迁移，可以在源宿主机上某客户机的QEMU monitor中，用"savevm my_tag"命令来保存一个完整的客户机镜像快照（标记为my_tag），然后在源宿主机中关闭或暂停该客户机，然后将该客户机的镜像文件复制到另外一台宿主机中，用于源宿主机中启动客户机时以相同的命令启动复制过来的镜像，在其QEMU monitor中用"loadvm my_tag"命令来恢复刚才保存的快照即可完全加载保存快照时的客户机状态。这里的"savevm"命令保存的完整客户机状态包括CPU状态、内存、设备状态、可写磁盘中的内容。注意，这种保存快照的方法需要qcow2、qed等格式的磁盘镜像文件，因为只有它们才支持快照这个特性（可参见前面4.4节“存储配置”中对镜像文件格式的介绍）。

本节主要介绍KVM中比静态迁移更实用、更方便的动态迁移。如果源宿主机和目的宿主机共享存储系统，则只需要通过网络发送客户机的vCPU执行状态、内存中的内容、虚拟设备的状态到目的主机上。否则，还需要将客户机的磁盘存储发送到目的主机上去。KVM中一个基于共享存储的动态迁移过程如图5-14所示。

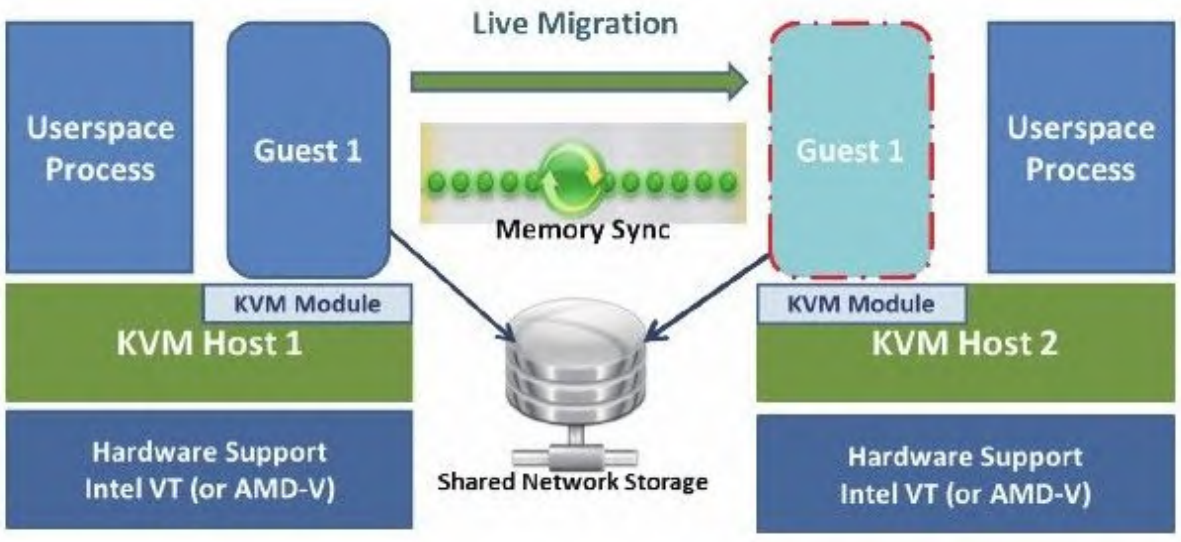


图　5-14　基于共享存储的KVM动态迁移

在不考虑磁盘存储复制的情况下（基于共享存储系统），KVM动态迁移的具体迁移过程为：在客户机动态迁移开始后，客户机依然在源宿主机上运行，与此同时，客户机的内存页被传输到目的主机之上。QEMU/KVM会监控并记录下迁移过程中所有已被传输的内存页的任何修改，并在所有的内存页都被传输完成后即开始传输在前面过程中内存页的更改内容。QEMU/KVM也会估计迁移过程中的传输速度，当剩余的内存数据量能够在一个可设定的时间周期（目前qemu-kvm中默认为30毫秒）内传输完成之时，QEMU/KVM将会关闭源宿主机上的客户机，再将剩余的数据量传输到目的主机上去，最后传输过来的内存内容在目的宿主机上恢复客户机的运行状态。至此，KVM的一个动态迁移操作就完成了。迁移后的客户机状态尽可能与迁移前一致，除非目的主机上缺少一些配置，例如，在源宿主机上有给客户机配置好网桥类型的网

络，但目的主机上没有网桥配置会导致迁移后客户机的网络不通。而当客户机中内存使用量非常大且修改频繁，内存中数据被不断修改的速度大于KVM能够传输的内存速度之时，动态迁移过程是不会完成的，这时要进行迁移只能进行静态迁移。笔者就曾遇到这样的情况，KVM宿主机上一个拥有4个vCPU、4GB内存的客户机中运行着一个SPECjbb2005（一个基准测试工具），使客户机的负载较重且内存会频繁更新，这时进行动态迁移无论怎样也不能完成，直到客户机中SPECjbb2005测试工具停止运行后，迁移过程才能真正完成。

在KVM中，动态迁移服务停机时间会与实际的工作负载和网络带宽等诸多因素有关，一般在数十毫秒到几秒钟之间，当然如果网络带宽过小或网络拥塞，会导致服务停机时间变长。必要的时候，服务器端在动态迁移时暂停数百毫秒或数秒钟后恢复服务，对于一些终端用户来说是可以接受的，可能会表现为：浏览器访问网页速度会慢一点，或者ssh远程操作过程中有一两秒不能操作但ssh连接并没有断开（不需要重新建立连接）。笔者曾经测试过KVM动态迁移过程中的服务停机时间，当时在客户机中运行了一个UnixBench（一个基准测试工具），然后进行动态迁移，经过测量，某次动态迁移中服务停机时间约为900毫秒。这次测试是通过在客户机中运行netperf（一个测试网络的基准测试工具）的服务端，在另外一台机器上运行netperf的客户端，通过查看netperf客户端收到服务端响应数据包在迁移过程中中断的间隔时间来粗略地估算客户机迁移过程中服务停机时间。当次实验的粗略结

果如图5-15所示，可以看出时间从4.6秒到5.5秒为动态迁移过程中大致的服务停机时间。

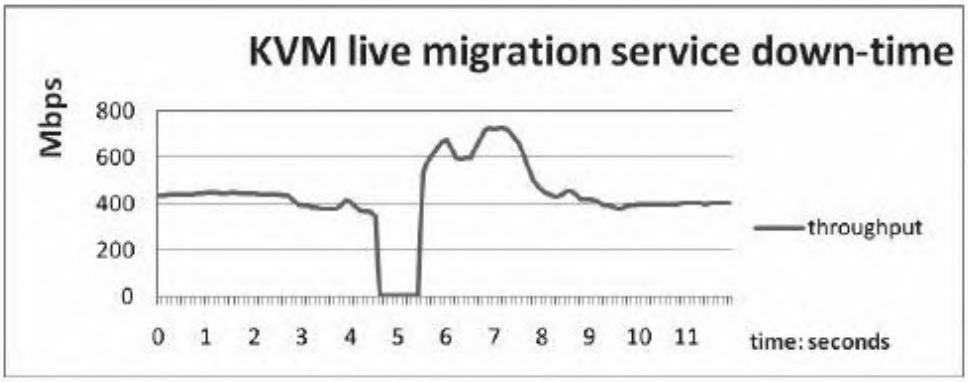


图　5-15　KVM动态迁移中服务停机时间的某次测量结果

从上面的介绍可知，KVM的动态迁移是比较高效也是很有用处的功能，在实际测试中也是比较稳定的（而且在多数情况下，就算迁移不成功，源宿主机上的客户机依然在运行）。不过，对于KVM动态迁移，也有如下几点建议和注意事项。

1）源宿主机和目的宿主机之间尽量用网络共享的存储系统来保存客户机磁盘镜像，尽管KVM动态迁移也支持连同磁盘镜像一起复制（加上一个参数即可，后面会介绍到）。共享存储（如NFS）在源宿主机和目的宿主机上的挂载位置必须完全一致。

2）为了提高动态迁移的成功率，尽量在同类型CPU的主机上面进行动态迁移，尽管KVM动态迁移也支持从Intel平台迁移到AMD平台（或者反向）。不过在Intel的两代不同平台之间进行动态迁移一般是比较

稳定的，如后面介绍实际操作步骤时，就是从一台Intel Westmere平台上运行的KVM中将一个客户机动态迁移到SandyBridge平台上。

3）64位的客户机只能在64位宿主机之间迁移，而32位客户机可以在32位宿主机和64位宿主机之间迁移。

4）动态迁移的源宿主机和目的宿主机对NX[12]（Never eXecute）位的设置是相同，要么同为关闭状态，要么同为打开状态。在Intel平台上的Linux系统中，用"cat/proc/cpuinfo|grep nx"命令可以查看是否有NX的支持。

5）在进行动态迁移时，被迁移客户机的名称是唯一的，在目的宿主机上不能有与源宿主机中被迁移客户机同名的客户机存在。另外，客户机名称可以包含字母、数字和“_”、“.”、“-”等特殊字符。

6）目的宿主机和源宿主机的软件配置尽可能的相同，例如，为了保证动态迁移后客户机的中的网络依然正常工作，需要在目的宿主机上配置和源宿主机相同名称网桥，并让客户机以桥接的方式使用网络（参见4.5.2小节“使用网桥模式”）。

下面详细介绍在KVM上进行动态迁移的具体操作步骤，这里的客户机镜像文件存放在NFS的共享存储上面，源宿主机（vt-nhm9）和目的宿主机（vt-snb9）都对NFS上的镜像文件具有可读写权限。

1）在源宿主机挂载NFS的上客户机镜像，并启动客户机，命令行操作如下：

```
[root@vt-nhm9 jay]# mount my-nfs:/rw-images/ /mnt/
[root@vt-nhm9 jay]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             197G   73G  115G  39% /
tmpfs                  12G   76K   12G   1% /dev/shm
my-nfs:/rw-images/    5.4T  2.3T  2.9T  45% /mnt
[root@vt-nhm9 jay]# qemu-system-x86_64 /mnt/ia32e_rhel6u3.img
-smp 2 -m 2048 -net nic -net tap
VNC server running on '::1:5900'
```

这里的特别之处是没有指定客户机中的CPU模型，默认是qemu64这个基本的模型，当然也可自行设置为"-cpu SandyBridge"或"-cpu Westmere"等，不过要保证在目的主机上也用相同的命令。在4.2.4小节“CPU模型”中已提及，同时都指定相同的某个CPU模型是可以让客户机在不同的几代平台上迁移更加稳定。

另外，还在客户机中运行一个程序（这里执行了"top"命令），以便在动态迁移后检查它是否仍然正常地继续执行。在动态迁移前，客户机运行了"top"命令的状态如图5-16所示。

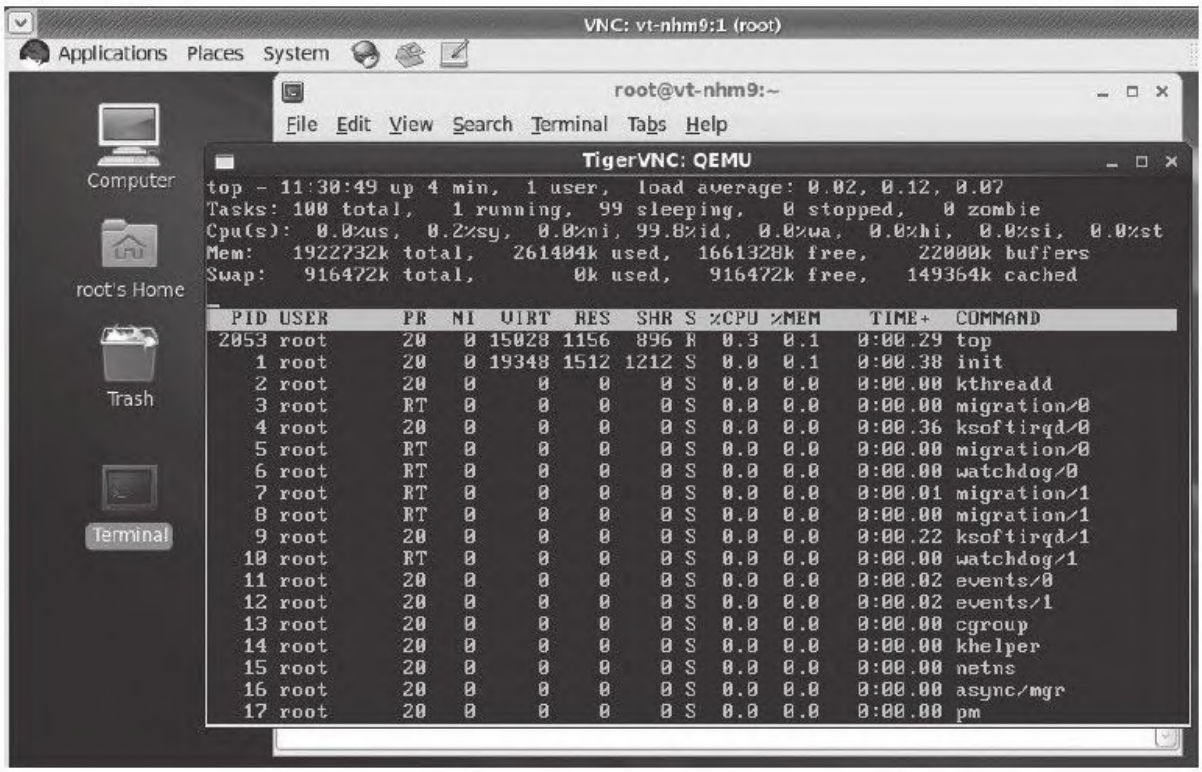


图 5-16 动态迁移前，客户机中运行着"top"命令

2）目的宿主机上也挂载NFS上的客户机镜像的目录，并且启动一个客户机用于接收动态迁移过来内存内容等，命令行操作如下：

```
[root@vt-snb9 jay]# mount vt-nfs:/rw-images/ /mnt/
[root@vt-snb9 jay]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             296G  172G  109G  62% /
tmpfs                  16G  232K   16G   1% /dev/shm
my-nfs:/rw-images/    5.4T  2.3T  2.9T  45% /mnt
[root@vt-snb9 jay]# qemu-system-x86_64 /mnt/ia32e_rhel6u3.img
-smp 2 -m 2048 -net nic -net tap -incoming tcp:0:6666
VNC server running on '::1:5901'
```

在这一步骤中，目的宿主机上的操作，有两个值得注意的地方：一是NFS的挂载目录必须与源宿主机上保持完全一致；二是启动客户机的命令与源宿主机上的启动命令一致，但是需要增加"-incoming"的选项。

这里在启动客户机的qemu-kvm命令行中添加了"-incoming tcp:0:6666"这个参数，它表示在6666端口建立一个TCP Socket连接用于接收来自源主机的动态迁移的内容，其中“0”表示允许来自任何主机的连接。"-incoming"这个参数使这里的qemu-kvm进程进入到迁移监听（migration-listen）模式，而不是真正以命令行中的镜像文件运行客户机，从VNC中看到客户机是黑色的没有任何显示，没有像普通客户机一样启动，而是在等待动态迁移数据的传入，如图5-17所示。

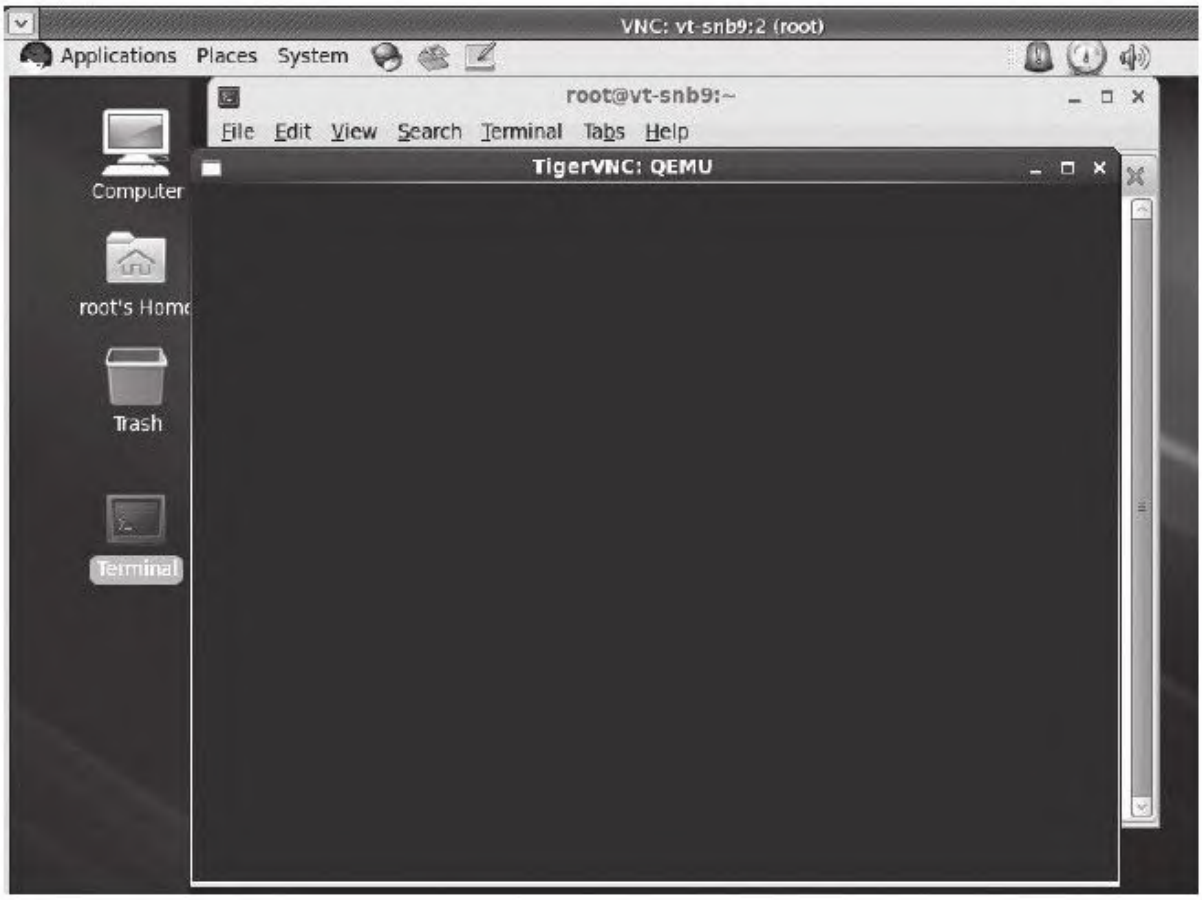


图　5-17　目的宿主机上带有"-incoming"参数的客户机

3）在源宿主机的客户机的QEMU monitor（默认用Ctrl+Alt+F2快捷键进入monitor）中，使用命令"migrate tcp:vt-snb9:6666"即可进入动态迁移的流程，如图5-18所示。这里的vt-snb9为目的宿主机的主机名（写为IP也是可以的），tcp协议和6666端口号与目的宿主机上qemu-kvm命令行的"-incoming"参数中的值保持一致。

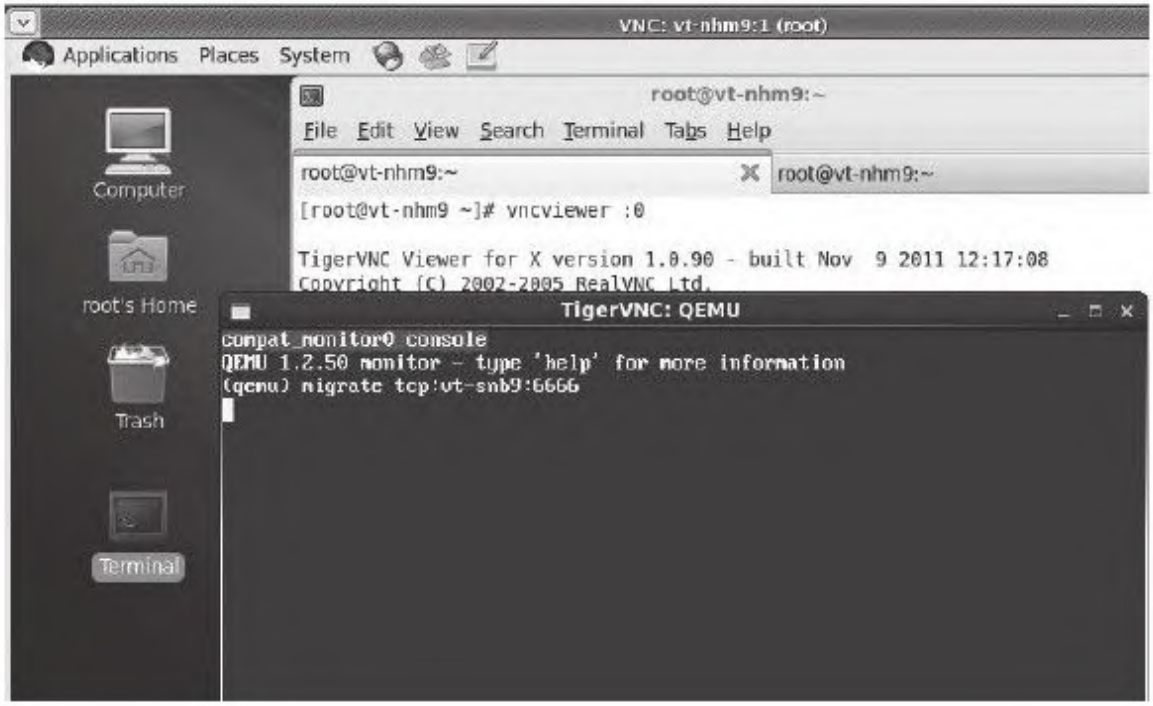


图 5-18 在QEMU monitor中使用"migrate"命令进行动态迁移

4）在本示例中，migrate命令从开始到执行完成，大约用了十秒钟。在执行完迁移后，在目的主机上，之前处于迁移监听模式的客户机就开始正常运行了，其中运行的正是动态迁移过来的客户机，可以看到客户机中的"top"命令在迁移后继续运行，如图5-19所示。

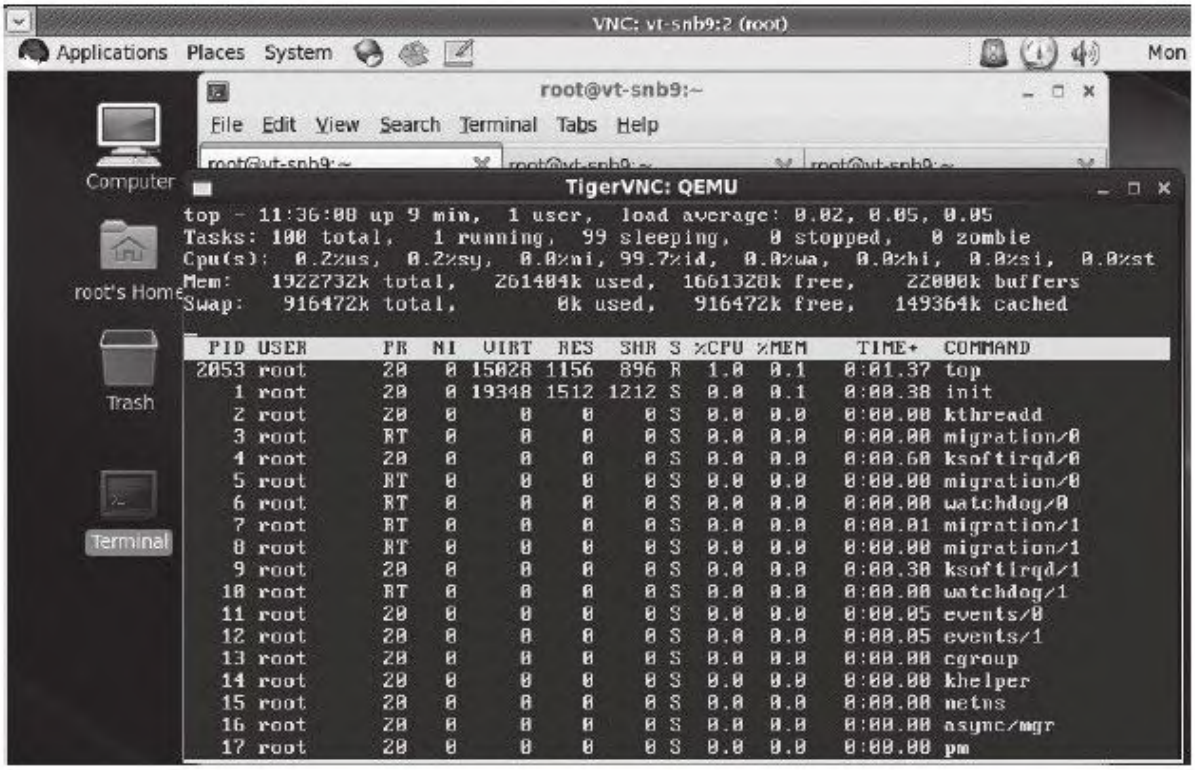


图 5-19 动态迁移后，客户机中的"top"命令依然在继续运行

至此，使用NFS作为共享存储的动态迁移就已经正确完成了。当然，QEMU/KVM中也支持增量复制磁盘修改部分数据（使用相同的后端镜像时）的动态迁移，以及直接复制整个客户机磁盘镜像的动态迁移。使用相同后端镜像文件的动态迁移过程如下，与前面直接使用NFS共享存储非常类似。

1）在源宿主机上，根据一个后端镜像文件，创建一个qcow2格式的镜像文件，并启动客户机，命令行如下：

```
[root@vt-nhm9 jay]# qemu-img create -f qcow2 -o
backing_file=/mnt/ia32e_rhel6u3.img,size=20G rhel6u3.qcow2
Formatting 'rhel6u3.qcow2', fmt=qcow2 size=21474836480
backing_file='/mnt/ia32e_rhel6u3.img' encryption=off
cluster_size=65536 lazy_refcounts=off
[root@vt-nhm9 jay]# qemu-system-x86_64 rhel6u3.qcow2 -smp 2 -m
2048 -net nic -net tap
VNC server running on '::1:5900'
```

这里使用前面挂载的NFS上的镜像文件作为qcow2的后端镜像。关于qemu-img命令的详细使用方法，可以参考4.4.2小节“qemu-img命令”。

2）在目的宿主机上，也建立相同的qcow2格式的客户机镜像，并带有"-incoming"参数来启动客户机使其处于迁移监听状态，命令行如下：

```
[root@vt-snb9 jay]# qemu-img create -f qcow2 -o
backing_file=/mnt/ia32e_rhel6u3.img,size=20G rhel6u3.qcow2
Formatting 'rhel6u3.qcow2', fmt=qcow2 size=21474836480
backing_file='/mnt/ia32e_rhel6u3.img' encryption=off
cluster_size=65536 lazy_refcounts=off
[root@vt-snb9 jay]# qemu-system-x86_64 rhel6u3.qcow2 -smp 2 -m
2048 -net nic -net tap -incoming tcp:0:6666
VNC server running on '::1:5901'
```

3）在源宿主机上的客户机的QEMU monitor中，运行"migrate-i tcp:vt-snb9:6666"命令即可进行动态迁移（"-i"表示increasing，增量的），在迁移过程中，还有实时的迁移百分比显示，如图5-20所示，提示为"Completed 100%"即表示迁移完成。

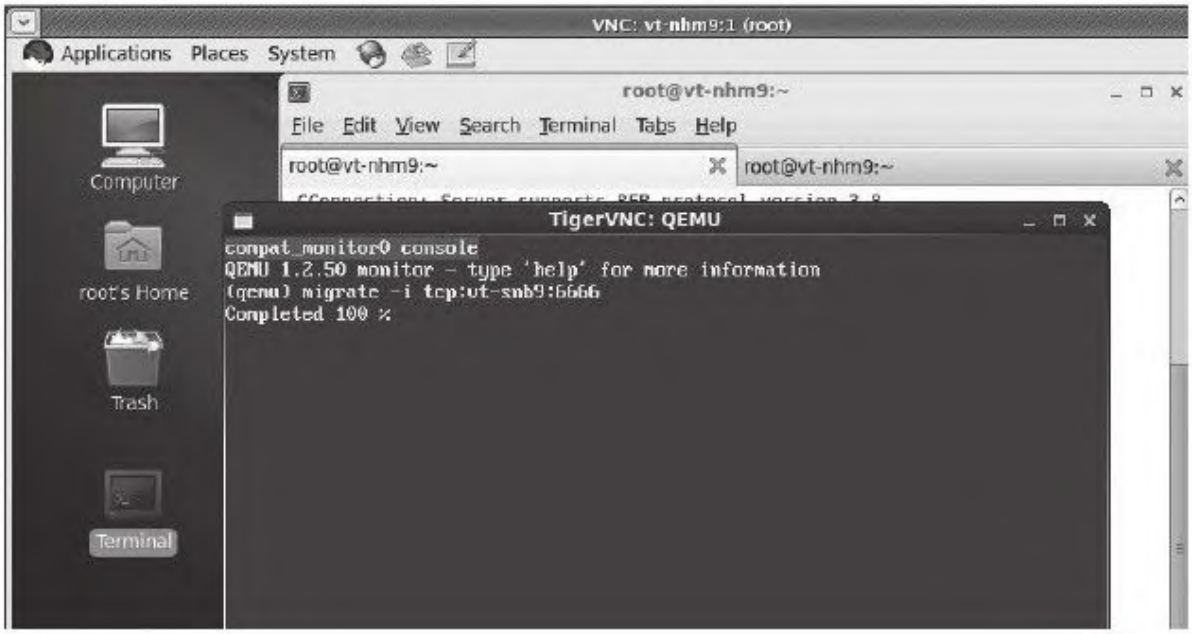


图　5-20　动态迁移"-i"参数显示传输增量部分的客户机镜像

与此同时，在目的宿主机上，在启动迁移监听状态的客户机的命令行所在标准输出中，也会提示正在传输的（增量的）磁盘镜像百分比，当传输完成时也会提示"Completed 100%"，如下：

```
[root@vt-snb9 jay]# qemu-system-x86_64 rhel6u3.qcow2 -smp 2 -m
2048 -net nic -net tap -incoming tcp:0:6666
VNC server running on '::1:5901'
Receiving block device images
Completed 100 %
Completed 100 %
```

至此，基于相同后端镜像的磁盘增量动态迁移就已经完成，在目的宿主机上可以看到迁移过来的客户机已经处于正常运行状态。在本示例中，由于qcow2文件中记录的增量较小（小于1GB），因此整个迁移过程花费了约20秒钟的时间。

如果不使用后端镜像的动态迁移，将会传输完整的客户机磁盘镜像（可能需要更长的迁移时间），其步骤与上面类似，只有两点需要修改：一是不需要用"qemu-img"命令创建qcow2格式的增量镜像整个步骤；二是QEMU monitor中的动态迁移的命令变为"migrate-b tcp:vt-snb9:6666"（-b参数意为block，传输块设备）。

最后介绍在QEMU monitor中与动态迁移相关的几个命令，可以用"help command"来查询命令的用法，如下：

```
(qemu) help migrate
migrate [-d] [-b] [-i] uri -- migrate to URI (using -d to not wait
for completion)
       -b for migration without shared storage with full copy of disk
       -i for migration without shared storage with incremental
copy of disk (base image shared between src and destination)
(qemu) help migrate_cancel
migrate_cancel  -- cancel the current VM migration
(qemu) help migrate_set_speed
migrate_set_speed value -- set maximum speed (in bytes)
for migrations. Defaults to MB if no size suffix is specified, ie. B/K/M/G/T
(qemu) help migrate_set_downtime
migrate_set_downtime value -- set maximum tolerated downtime (in
seconds) for migrations
(qemu) info migrate    ##show migration status
```

对于"migrate[-d][-b][-i]uri"命令，其中uri为uniform resource identifier（统一资源标识符），在上面的示例中就是"tcp:vt-snb9:6666"这样的字符串，在不加"-b"和"-i"选项的情况下，默认是共享存储下的动态迁移（不传输任何磁盘镜像内容）。"-b"和"-i"选项前面也都提及过，"-b"选项表示传输整个磁盘镜像，"-i"选项是在有相同的后端镜像的情况下增量传输qcow2类型的磁盘镜

像，而"-d"选项是不用等待迁移完成就让QEMU monior处于可输入命令的状态（在前面的示例中都没有使用"-d"选项，所以在动态迁移完成之前"migrate"命令会完全占有monitor操作界面，而不能输入其他命令）。

"migrate_cancel"命令，是在动态迁移进行过程中取消迁移（在"migrate"命令中需要使用"-d"选项才能有机会在迁移完成前操作"migrate_cancel"命令）。

"migrate_set_speed value"命令，设置动态迁移中的最大传输速度，可以带有B、K、G、T等单位，表示每秒钟传输的字节数。在某些生产环境中，如果动态迁移消耗了过大的带宽，可能会让网络拥塞，从而降低其他服务器的服务质量，这时也可以设置动态迁移用的合适的速度。

"migrate_set_downtime value"命令，设置允许的最大停机时间，单位是秒，value的值可以是浮点数（如0.5）。qeum-kvm会预估最后一步的传输需要花费的时间，如果预估时间大于这里设置的最大停机时间，则不会做最后一步迁移，直到预估时间小于等于设置的最大停机时间时才会完成最后迁移，暂停源宿主机上的客户机，然后传输内存中改变的内容。

5.4.4 VT-d/SR-IOV的动态迁移

前面已经介绍过，使用VT-d、SR-IOV等技术，可以让设备（如网卡）在客户机中获得非常良好的、接近原生设备的性能。不过，当QEMU/KVM中有设备直接分配到客户机中时，就不能对该客户机进行动态迁移，所以说VT-d、SR-IOV等的使用会破坏动态迁移的特性。

QEMU/KVM并没有直接解决这个问题，不过，可以使用热插拔设备来避免动态迁移的失效。比如，想使用VT-d或SR-IOV方式的一个网卡（包括虚拟功能VF），可以在qemu-kvm命令行启动客户机时并不分配这个网卡（而是使用网桥等方式为客户机分配网络），当在客户机启动后，再动态添加该网卡到客户机中使用。当该客户机需要动态迁移时，就动态移除该网卡，让客户机在迁移前后这一小段时间内使用启动时分配的网桥方式的网络，待动态迁移完成后，如果迁移后的目的主机上也有可供直接分配的网卡设备，就再重新动态添加一个网卡到客户机中。这样既满足了使用高性能网卡的需求，又没有损坏动态迁移的功能。

另外，如果客户机使用较新的Linux内核，还可以使用“以太网绑定驱动”（Linux Ethernet Bonding Driver），该驱动可以将多个网络接口绑定为一个逻辑上的单一接口。当在网卡热插拔场景中使用该绑定驱动时，可以提高网络配置的灵活性和网络切换时的连续性。关于Linux中的“以太网绑定驱动”，参考如下网页中Linux内核文档对该驱动的描述：

http://www.kernel.org/doc/Documentation/networking/bonding.txt。

如果使用libvirt来管理QEMU/KVM，则在libvirt0.9.2及更新的版本中已经开始支持直接使用VT-d的普通设备和SR-IOV的VF且不丢失动态迁移的能力。在libvirt中直接使用宿主机网络接口需要KVM宿主机中macvtap驱动的支持，要求宿主机的Linux内核是2.6.38或更新的版本。在libvirt的客户机的XML配置文件中，关于该功能的配置示例如下：

```
  ...
<devices>
  ...
  <interface type='direct'>
  <source dev='eth0' mode='passthrough'/>
  <model type='virtio'/>
 </interface>
</devices>
  ...
```

其中，dev='eth0'表示使用宿主机中的eth0这个网络接口，如果需要在客户机中使用高性能的网络，则eth0可以是一个高性能的网络接口（当然eth0可以根据实际情况换为ethX）。

如果系统管理员对于客户机中网卡等设备的性能要求非常高，而且不愿意丢失动态迁移功能，那么可以考虑尝试使用本节介绍的

PCI/PCI-e设备的动态热插拔或使用libvirt中提供的方法直接将宿主机中某个网络接口给客户机使用。

5.5 嵌套虚拟化

5.5.1 嵌套虚拟化的基本概念

嵌套虚拟化（nested virtualization、recursive virtualization）是指在虚拟化的客户机中运行一个Hypervisor，从而再虚拟化运行一个客户机。嵌套虚拟化不仅包括相同Hypervisor的嵌套（如KVM嵌套KVM、Xen嵌套Xen、VMware嵌套VMware等），也包括不同Hypervisor的相互嵌套（如VMware嵌套KVM、KVM嵌套Xen、Xen嵌套KVM等）。根据嵌套虚拟化这个概念可知，不仅包括两层嵌套（如KVM嵌套KVM），还包括多层的嵌套（如KVM嵌套KVM再嵌套KVM）。

嵌套虚拟化的使用场景是非常多的，至少包括如下五个较大的应用：

1）IaaS（Infrastructure as a Service）类型的云计算提供商，如果有了嵌套虚拟化功能的支持，就可以为其客户提供让客户可以自己运行所需Hypervisor和客户机的能力。对于由于这类需求的客户来说，这样的嵌套虚拟化能力会成为吸引他们购买云计算服务的因素。

2）为测试和调试Hypervisor带来了非常大的便利，有了嵌套虚拟化功能的支持，被调试Hypervisor运行在更底层的Hypervisor之上，就算遇到被调试Hypervisor的系统崩溃，也只需要在底层的Hypervisor上重启被调试系统即可，而不需要真实地与硬件打交道。

3）在一些为了起到安全作用的带有Hypervisor的固件（firmware）上，如果有嵌套虚拟化的支持，则在它上面不仅可以运行一些普通的负载，还可以运行一些Hypervisor启动另外的客户机。

4）嵌套虚拟化的支持，对虚拟机系统的动态迁移也提供了新的功能，从而可以将一个Hypervisor及其上面运行的客户机作为一个单一的节点进行动态迁移。这对服务器的负载均衡及灾难恢复等方面也有积极意义。

5）嵌套虚拟化的支持，对于系统隔离性、安全性方面也提供更多的实施方案。

对于不同的Hypervisor，嵌套虚拟化的实现方法和难度都相差很大。对于完全纯软件模拟CPU指令执行的模拟器（如QEMU），实现嵌套虚拟化相对来说并不复杂；而对于QEMU/KVM这样的必须依靠硬件虚拟化扩展的方案，就必须在客户机中模拟硬件虚拟化特性（如vmx、svm）的支持，并且对上层KVM Hypervisor的操作指令进行模拟。据笔者所知，目前，Xen方面已经支持Xen on Xen和KVM on Xen，而且在某

些平台上已经可以运行"KVM on Xen on Xen"的多级嵌套虚拟化；VMware已经支持VMware on VMware和KVM on VMware这两类型的嵌套。在KVM方面，KVM已经性能较好地支持KVM on KVM的情况，目前Xen on KVM还不能正常运行，KVM社区也会逐渐修复这些问题。

5.5.2 KVM嵌套KVM

KVM嵌套KVM，即是在KVM上面运行的第一级客户机中再加载kvm和kvm_intel（或kvm_amd）模块，然后在第一级的客户机中用qemu-kvm启动带有kvm加速的第二级客户机。“KVM嵌套KVM”的基本架构示意图如图5-21所示，其中底层是具有Intel VT或AMD-V特性的硬件系统，硬件层之上就是底层的宿主机系统（我们称之为L0，即Level 0），在L0宿主机中可以运行加载有KVM模块的客户机（我们称之为L1，即Level 1，第一级），在L1客户机中通过QEMU/KVM启动一个普通的客户机（我们称之为L2，即Level 2，第二级）。如果KVM还可以做多级的嵌套虚拟化，各个级别的操作系统被依次称为：L0、L1、L2、L3、L4……，其中L0向L1提供硬件虚拟化环境（Intel VT或AMD-V），L1向L2提供硬件虚拟化环境，依此类推。而最高级别的客户机Ln（如图5-21中的L2）可以是一个普通客户机，不需要下面的Ln-1级向Ln级中的CPU提供硬件虚拟化支持。

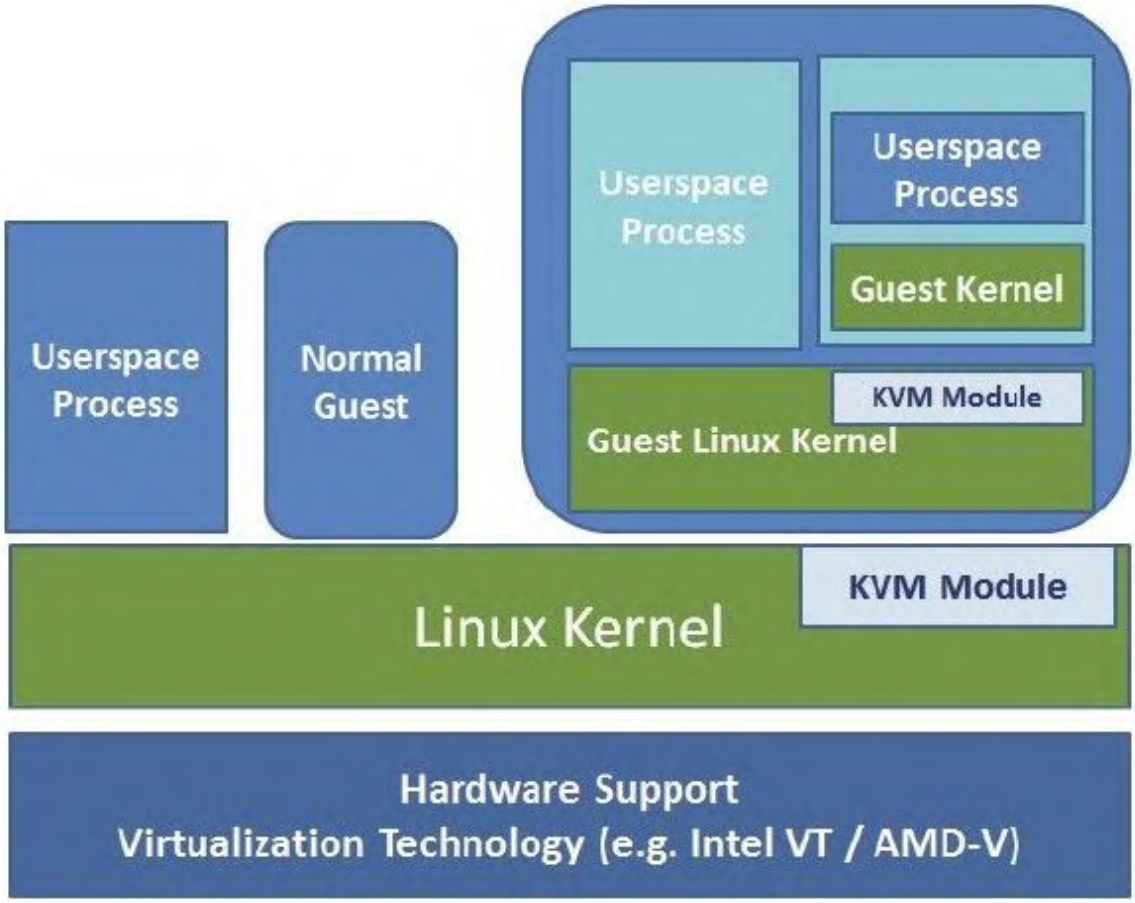


图　5-21　KVM嵌套KVM的基本架构图

KVM对“KVM嵌套KVM”的支持从2010年就开始了，目前已经比较成熟了。“KVM嵌套KVM”功能的配置和使用，有如下几个步骤。

1）在L0中，加载kvm-intel（或kvm-amd）模块时需要添加"nested=1"的选项以打开“嵌套虚拟化”的特性，如下：

```
[root@jay-linux kvm_demo]# modprobe kvm
[root@jay-linux kvm_demo]# modprobe kvm_intel nested=1
[root@jay-linux kvm_demo]# cat \
/sys/module/kvm_intel/parameters/nested
Y
```

如果kvm_intel模块已经处于使用中，则需要用"rmmod kvm_intel"命令移除kvm_intel模块后重新加载即可，然后要检查"/sys/module/kvm_intel/parameters/nested"这个参数是否为"Y"。对于AMD平台上的kvm-amd模块的操作也是一模一样的。

2）启动L1客户机时，在qemu-kvm命令中加上"-cpu host"或"-cpu qemu64,+vmx"选项以便将CPU的硬件虚拟化扩展特性暴露给L1客户机，如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
4096 -smp 2 -net nic -net tap -cpu host
VNC server running on '::1:5900'
```

这里的"-cpu host"参数的作用是尽可能将宿主机L0的CPU特性暴露给L1客户机，而"-cpu qemu64,+vmx"以qemu64这个CPU模型为基础，然后加上Intel VMX特性（即CPU的VT-x支持）。当然，以其他CPU模型为基础再加上VMX特性，如"-cpuSandyBridge,+vmx"、"-cpu Westmere,+vmx"都是可以的。在AMD平台上，则需要对应的CPU模型（"qemu64"是通用的），再加上AMD-V特性即可，如"-cpu qemu64,+svm"。

3）在L1客户机中，查看CPU的虚拟化支持，然后加载kvm和kvm_intel模块，启动一个L2客户机，如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo | grep vmx
flags           : fpu de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc
arch_perfmon rep_good unfair_spinlock pni pclmulqdq vmx ssse3 cx16
sse4_1 sse4_2 x2apic popcnt aes xsave avx hypervisor lahf_lm xsaveopt
flags           : fpu de pse tsc msr pae mce cx8 apic mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc
arch_perfmon rep_good unfair_spinlock pni pclmulqdq vmx ssse3 cx16
sse4_1 sse4_2 x2apic popcnt aes xsave avx hypervisor lahf_lm xsaveopt
[root@kvm-guest ~]# modprobe kvm
[root@kvm-guest ~]# modprobe kvm_intel
[root@kvm-guest ~]# lsmod | grep kvm
kvm_intel              52570  0
kvm                   314739  1 kvm_intel
[root@kvm-guest ~]# qemu-system-x86_64 rhel6u3.img -m 1024 -smp 2
VNC server running on '::1:5900'
```

如果L0没有向L1提供硬件虚拟化的CPU环境，则加载kvm_intel模块时会有错误，kvm_intel模块会加载失败。在L1中启动客户机，就与在普通KVM环境中的操作完全一样。不过对L1系统的内核要求并不高，一般选取较新Linux内核即可，如笔者选用了RHEL 6.3系统自带的内核和Linux 3.5的内核，这都是可以的。

4）在L2客户机中查看是否正常运行，图5-22展示了“KVM嵌套KVM”虚拟化的运行环境，L0启动了L1，然后在L1中启动了L2系统。

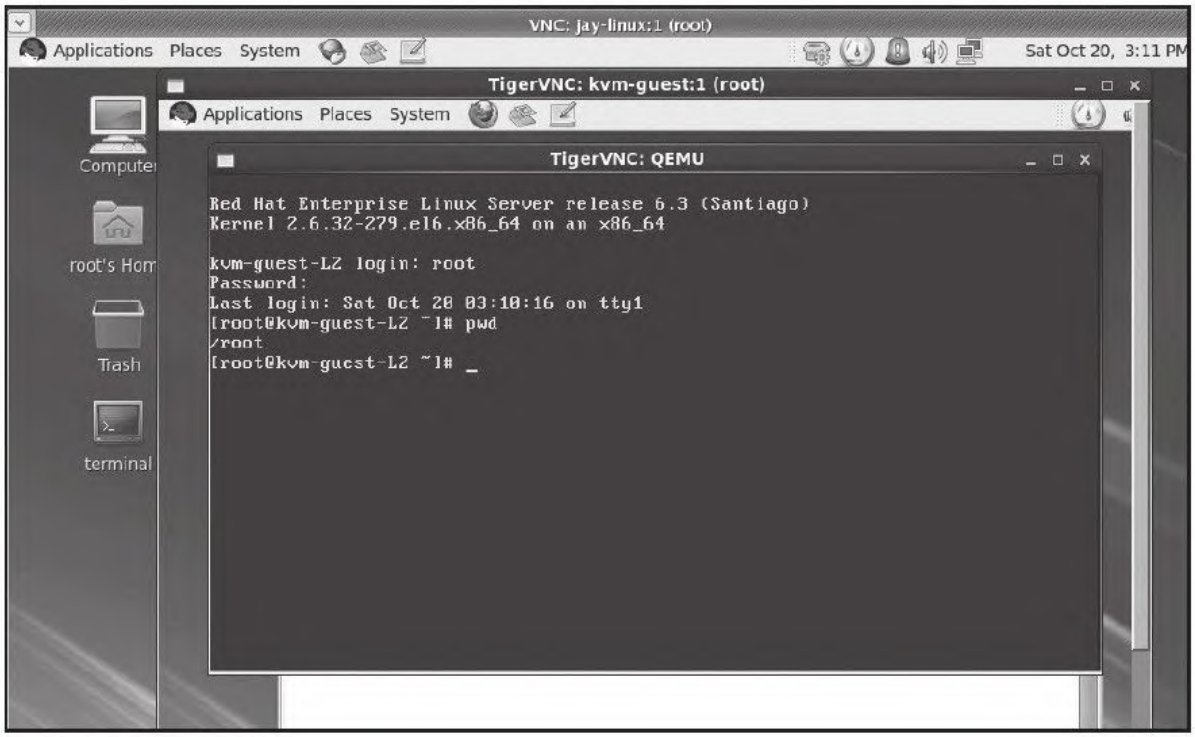


图　5-22　KVM嵌套KVM的L0、L1、L2的示例

由于KVM是全虚拟化Hypervisor，对于其他L1 Hypervisor（如Xen）嵌套运行在KVM上情况，在L1中启动L2客户机的操作就完全与在普通的Hypervisor中的操作步骤完全一样，因为KVM为L1提供了有硬件辅助虚拟化特性的透明的硬件环境。

5.6 KSM技术

5.6.1 KSM基本原理

在现代操作系统中，共享内存一个很普遍应用的概念。如在Linux系统中，当使用fork函数创建一个进程之时，子进程与其父进程共享全部的内存，而当子进程或父进程试图修改它们的共享内存区域之时，内核会分配一块新的内存区域，并将试图修改的共享内存区域复制到新的内存区域上，然后让进程去修改复制的内存。这就是著名的“写时复制”（copy-on-write，COW）技术。而本节介绍的KSM技术却与这种内存共享概念有点相反。

KSM是"Kernel SamePage Merging"的缩写，中文可称为“内核同页合并”。KSM允许内核在两个或多个进程（包括虚拟客户机）之间共享完全相同的内存页。KSM让内核扫描检查正在运行中的程序并比较它们的内存，如果发现他们有内存区域或内存页是完全相同的，就将多个相同的内存合并为一个单一的内存页，并将其标识为“写时复制”。这样可以起到节省系统内存使用量的作用。之后，如果有进程试图去修改被标识为“写时复制”的合并的内存页时，就为该进程复制出一个新的内存页供其使用。

在QEMU/KVM中，一个虚拟客户机就是一个QEMU进程，所以使用KSM也可以实现多个客户机之间的相同内存合并。而且，如果在同一宿主机上的多个客户机运行的是相同的操作系统或应用程序，则客户机之间的相同内存页的数量就可能还比较大，这种情况下KSM的作用就更加显著。在KVM环境下使用KSM，KSM还允许KVM请求哪些相同的内存页是可以被共享而合并的，所以KSM只会识别并合并那些不会干扰客户机运行，不会影响宿主机或客户机的安全内存页。可见，在KVM虚拟化环境中，KSM能够提高内存的速度和使用效率，具体可以从以下两个方面来理解。

1）在KSM的帮助下，相同的内存页被合并了，减少了客户机的内存使用量，一方面，内存中的内容更容易被保存到CPU的缓存当中；另一方面，有更多的内存可用于缓存一些磁盘中的数据。因此，不管是内存的缓存命中率（CPU缓存命中率），还是磁盘数据的缓存命中率（在内存中命中磁盘数据缓存的命中率）都会提高，从而提高了KVM客户机中操作系统或应用程序的运行速度。

2）正如在4.3.4节“内存过载使用”中提及的那样，KSM是内存过载使用的一种较好的方式。KSM通过减少每个客户机实际占用的内存数量，就可以让多个客户机分配的内存数量之和大于物理上的内存数量。而对于使用相同内存量的客户机，在物理内存量不变的情况，可

以在一个宿主机中创建更多的客户机，提高了虚拟化客户机部署的密度，提高了物理资源的利用效率。

KSM是在Linux内核2.6.32中被加入到内核主干代码中去的。目前多数流行的Linux发型版都已经将KSM的支持编译到内核中了，其内核配置文件中有"CONFIG_KSM=y"项。Linux系统的内核进程ksmd负责扫描后合并进程的相同内存页，从而实现KSM功能。root用户可以通过"/sys/kernel/mm/ksm/"目录下的文件来配置和监控ksmd这个守护进程。KSM只会去扫描和试图合并那些应用程序建议为可合并的内存页，应用程序（如qemu-kvm）通过调用如下的madvise系统调用来告诉内核哪些页可合并。目前较新的qemu-kvm都是支持KSM的，也可以通过查看其代码中对madvise函数的调用情况来确定是否支持KSM，qemu-kvm中的关键函数简要分析如下：

```
/* 将地址标志为 KSM 可合并的系统调用 */
/* int madvise(addr, length, MADV_MERGEABLE) */
/* madvise 系统调用的声明在 <sys/mman.h>  中 */
/* int  madvise(  void  *start,  size_t  length,  int  advice  );  */

/* qemu-kvm 代码的 exec.c 文件中，开启内存可合并选项 */
static int memory_try_enable_merging(void *addr, size_t len)
{
    QemuOpts *opts;

    opts = qemu_opts_find(qemu_find_opts("machine"), 0);
    if (opts && !qemu_opt_get_bool(opts, "mem-merge", true)) {
        /* disabled by the user */
        return 0;
    }
    return qemu_madvise(addr, len, QEMU_MADV_MERGEABLE);
}

/* qemu-kvm 代码的 osdep.c 文件中对 qemu_madvise() 函数的定义 */
int qemu_madvise(void *addr, size_t len, int advice)
{
    if (advice == QEMU_MADV_INVALID) {
        errno = EINVAL;
        return -1;
    }
#if defined(CONFIG_MADVISE)
    return madvise(addr, len, advice);
#elif defined(CONFIG_POSIX_MADVISE)
   return posix_madvise(addr, len, advice);
#else
   errno = EINVAL;
   return -1;
#endif
}
```

KSM最初就是为KVM虚拟化中的使用而开发的，不过它对非虚拟化的系统依然非常有用。KSM可以在KVM虚拟化环境中非常有效地降低内存使用量，据笔者在网上看到的资料显示，在KSM的帮助下，有人在物

理内存为16GB的机器上，用KVM成功运行了多达52个1GB内存的Windows XP客户机。

由于KSM对KVM宿主机中的内存使用有较大的效率和性能的提高，所以一般建议打开KSM功能。不过，“金无足赤，人无完人”，KSM必须有一个或多个进程去检测和找出哪些内存页是完全相同可以用于合并的，而且需要找到那些不会经常更新的内存页，这样的页才是最适合于合并的。因此，KSM让内存使用量降低了，但是CPU使用率会有一定程度的升高，也可能会带来隐蔽的性能问题，需要在实际使用环境中进行适当配置KSM的使用，以便达到较好的平衡。

KSM对内存合并而节省内存的数量与客户机操作系统类型及其上运行的应用程序有关，如果宿主机上的客户机操作系统相同且其上运行的应用程序也类似，节省内存的效果就会很显著，甚至节省超过50%的内存都有可能的。反之，如果客户机操作系统不同，且运行的应用程序也大不相同，KSM节省内存效率就不好，可能连5%都不到。

另外，在使用KSM实现内存过载使用时，最好保证系统的交换空间（swap space）足够大。因为KSM将不同客户机的相同内存页合并而减少了内存使用量，但是客户机可能由于需要修改被KSM合并的内存页，从而使这些被修改的内存被重新复制出来占用内存空间，因此可能会导致系统内存的不足，这是需要足够的交换空间来保证系统的正常运行。

5.6.2 KSM操作实践

内核的KSM守护进程是ksmd，配置和监控ksmd的文件在"/sys/kernel/mm/ksm/"目录下。通过如下命令行可以查看该目录下的几个文件：

```
[root@jay-linux kvm_demo]# ls -l /sys/kernel/mm/ksm/*
-r--r--r-- 1 root root 4096 Oct 29 17:28 /sys/kernel/mm/ksm/full_scans
-r--r--r-- 1 root root 4096 Oct 29 17:28 /sys/kernel/mm/ksm/pages_shared
-r--r--r-- 1 root root 4096 Oct 29 17:28 /sys/kernel/mm/ksm/pages_sharing
-rw-r--r-- 1 root root 4096 Oct 29 18:06 /sys/kernel/mm/ksm/pages_to_scan
-r--r--r-- 1 root root 4096 Oct 29 17:28 /sys/kernel/mm/ksm/pages_unshared
-r--r--r-- 1 root root 4096 Oct 29 17:28 /sys/kernel/mm/ksm/pages_volatile
-rw-r--r-- 1 root root 4096 Oct 29 23:01 /sys/kernel/mm/ksm/run
-rw-r--r-- 1 root root 4096 Oct 29 18:06 /sys/kernel/mm/ksm/sleep_millisecs
```

这里面的几个文件对于了解KSM的实际工作状态来说是非常重要的，下面简单介绍各个文件的作用。

❑full_scans：记录着已经对所有可合并的内存区域扫描过的次数。

❑pages_shared：记录着正在使用中的共享内存页的数量。

❑pages_sharing：记录着有多少数量的内存页正在使用被合并的共享页，不包括合并的内存页本身。这就是实际节省的内存页数量。

❑pages_unshared：记录了守护进程去检查并试图合并，却发现了并没有重复内容而不能被合并的内存页数量。

❑pages_volatile：记录了因为其内容很容易变化而不被合并的内存页。

❑pages_to_scan：在ksmd进程休眠之前会去扫描的内存页数量。

❑sleep_millisecs：ksmd进程休眠的时间（单位：毫秒），ksmd的两次运行之间的间隔。

❑run：控制ksmd进程是否运行的参数，默认值为0，要激活KSM必须要设置其值为1（除非内核关闭了sysfs的功能）。设置为0，表示停止运行ksmd但保持它已经合并的内存页；设置为1，表示马上运行ksmd进程；设置为2表示停止运行ksmd，并且分离已经合并的所有内存页，但是保持已经注册为可合并的内存区域给下一次运行使用。

通过前面查看这些sysfs中的ksm相关的文件可以看出，只有pages_to_scan、sleep_millisecs、run这3个文件对root用户是可读可写的，其余5个文件都是只读的。可以向pages_to_scan、sleep_millisecs、run这3个文件中写入自定义的值以便控制ksmd的运行。例如，"echo 1200＞/sys/kernel/mm/ksm/pages_to_scan"用来调整每次扫描的内存页数量，"echo 10＞/sys/kernel/mm/ksm/sleep_millisecs"用来设置ksmd两次运行的时间间隔，"echo 1＞/sys/kernel/mm/ksm/run"用来激活ksmd的运行。

pages_sharing的值越大，说明KSM节省的内存越多，KSM效果越好，如下命令计算了节省的内存数量。

```
[root@jay-linux ~]# echo "KSM saved: $(( $(cat
/sys/kernel/mm/ksm/pages_sharing) * $(getconf PAGESIZE) / 1024 / 1024 ))MB"
KSM saved: 1375MB
```

而pages_sharing除以pages_shared得到的值越大，说明相同内存页重复的次数越多，KSM效率就是越高。pages_unshared除以pages_sharing得到的值越大，说明ksmd扫描不能合并的内存页越多，KSM的效率越低。可能有多种因素影响pages_volatile的值，不过较高的page_voliatile值预示着很可能有应用程序过多地使用了madvise(addr,length,MADV_MERGEABLE)系统调用来将其内存标志为KSM可合并。

在通过"/sys/kernel/mm/ksm/run"等修改了KSM的设置之后，系统默认不会再修改它的值，这样可能并不能更好地使用后续的系统状况，或者经常需要人工动态调节是比较麻烦的。Redhat系列系统（如RHEL 6.3）中提供了两个服务ksm和ksmtuned来动态调节KSM的运行情况，RHEL 6.3中ksm和ksmtuned两个服务都包含在qemu-kvm这个RPM安装包中。在RHEL 6.3上，如果不运行ksm服务程序，则KSM默认只会共享2000个内存页，这样一般很难起到较好的效果。而在启动ksm服务后，KSM能够共享最多达到系统物理内存一半的内存页。而ksmtuned服

务一直保持循环执行，以调节ksm服务的运行，其配置文件在/etc/ksmtuned.conf，配置文件的默认内容如下：

```
[root@jay-linux kvm_demo]# cat /etc/ksmtuned.conf
# Configuration file for ksmtuned.

# How long ksmtuned should sleep between tuning adjustments
# KSM_MONITOR_INTERVAL=60

# Millisecond sleep between ksm scans for 16Gb server.
# Smaller servers sleep more, bigger sleep less.
# KSM_SLEEP_MSEC=10

# KSM_NPAGES_BOOST=300
# KSM_NPAGES_DECAY=-50
# KSM_NPAGES_MIN=64
# KSM_NPAGES_MAX=1250

# KSM_THRES_COEF=20
# KSM_THRES_CONST=2048

# uncomment the following if you want ksmtuned debug info

# LOGFILE=/var/log/ksmtuned
# DEBUG=1
```

下面演示一下KSM带来的节省内存的实际效果。在物理内存为4GB的系统上，使用了Linux 3.5内核的RHEL 6.3系统作为宿主机，开始时将ksm和ksmtuned服务暂停，"/sys/kernel/mm/ksm/run"的默认值为0，KSM不生效，然后启动每个内存为1GB的4个Windows 7客户机（没有安装virtio-balloon驱动，没有以ballooning方式节省内存），启动ksm和ksmtuned服务，10分钟后检查系统内存的使用情况以确定KSM的效果。实现该功能的一个示例脚本（ksm-test.sh）如下：

```
#!/bin/bash
# file: ksm-test.sh

echo "----stoping services: ksm and ksmtuned ..."
service ksm stop
service ksmtuned stop

echo "----'free -m' command output before starting any guest."
free -m

# start 4 Win7 guest
for i in $(seq 1 4)
do
        qemu-img create -f qcow2 -o
backing_file="/images/Windows/ia32e_win7_ent.img" win7-${i}.qcow2
        echo "starting the No. ${i} guest..."
        qemu-system-x86_64 win7-${i}.qcow2 -smp 2 -m 1024 -net nic
-net tap -daemonize
        sleep 20
done

echo "----'free -m' command output with several guests running ."
free -m

echo "----starting services: ksm and ksmtuned ..."
service ksm start
service ksmtuned start

sleep 600

echo "----'free -m' command output with ksm and ksmtuned running."
free -m
```

执行该脚本，其命令行输出如下：

```
[root@jay-linux kvm_demo]# ./ksm-test.sh
----stoping ksmd and ksmtuned
Stopping ksm:                                              [  OK  ]
Stopping ksmtuned:                                         [  OK  ]
----'free -m' command output before starting any guest.
             total       used       free     shared    buffers     cached
Mem:          3979        424       3554          0         34        180
-/+ buffers/cache:        209       3769
Swap:         4095          0       4095
Formatting 'win7-1.qcow2', fmt=qcow2 size=21474836480
backing_file='/images/Windows/ia32e_win7_ent.img'
encryption=off cluster_size=65536
starting the No. 1 guest...
VNC server running on '::1:5900'
Formatting 'win7-2.qcow2', fmt=qcow2 size=21474836480 backing_file=
'/images/Windows/ia32e_win7_ent.img' encryption=off cluster_size=65536
starting the No. 2 guest...
VNC server running on '::1:5901'
Formatting 'win7-3.qcow2', fmt=qcow2 size=21474836480 backing_file=
'/images/Windows/ia32e_win7_ent.img' encryption=off cluster_size=65536
starting the No. 3 guest...
VNC server running on '::1:5902'
Formatting 'win7-4.qcow2', fmt=qcow2 size=21474836480 backing_file=
'/images/Windows/ia32e_win7_ent.img' encryption=off cluster_size=65536
starting the No. 4 guest...
VNC server running on '::1:5903'
----'free -m' command output with service guests running .
             total       used       free     shared    buffers     cached
Mem:          3979       3892         86          0          0         39
-/+ buffers/cache:       3853        126
Swap:         4095        246       3849
----starting ksm and ksmtuned ...
Starting ksm:                                              [  OK  ]


Starting ksmtuned:                                         [  OK  ]
----'free -m' command output after running ksm and ksmtuned.
             total       used       free     shared    buffers     cached
Mem:          3979       3861        117          0         18        965
-/+ buffers/cache:       2877       1101
Swap:         4095        423       3672
```

在以上输出中，从ksm、ksmtuned服务开始运行之前和之后的"free-m"命令的"-/+buffers/cache:"这一行的输出数值可以看出：

由于KSM的作用，系统的空闲内存从126 MB提高到了1101MB，获得了差不多1GB的空闲可用内存。此时查看"/sys/kernel/mm/ksm/"目录中KSM的状态，如下：

```
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/full_scans
4
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/pages_shared
109395
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/pages_sharing
572100
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/pages_to_scan
1200
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/pages_unshared
233589
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/run
1
[root@jay-linux kvm_demo]# cat /sys/kernel/mm/ksm/sleep_millisecs
41
```

可见，KSM已经为系统提供了不少的共享内存页，而当前KSM的运行标志（run）设置为1。当然，查看到的run值也可能为0，因为RHEL 6.3中的ksm和ksmtuned这两个服务会根据系统状况，按照既定的规则来修改"/sys/kernel/mm/ksm/run"文件中的值，从而调节KSM的运行。

5.7 KVM其他特性简介

前面几节介绍了KVM的一些重要的并且较大的特性，本节再介绍几个较小但同样非常有用的特性，如1GB大页内存的使用、透明大页、AVX和XSAVE指令的支持、AES新指令的支持、完全暴露宿主机CPU的特性。

5.7.1 1GB大页

4.3.3小节介绍的大页（huge page）主要是针对x86-64 CPU架构上的2MB大小的大页，本节将介绍更大的大页——1GB大页，这两者的基本用法和原理是大同小异的。总的来说，1GB大页与2MB大页一样，使用了hugetlbfs（一个基于内存的特殊文件系统）来直接利用硬件提供的大页支持，以便创建共享的或私有的内存映射，这样减少了内存页表数量，提高了TLB缓存的效率，从而提高了系统的内存访问的性能。而且在Intel EPT（或AMD NTP）技术的辅助下，KVM的客户机中内存访问性能还会有更明显的提高。在KVM虚拟化环境中，一个客户机就是一个qemu-kvm进程，Linux宿主机系统可以向qemu-kvm进程分配大页，而且当客户机中有应用程序或程序库使用到大页时，就能很好地提高客户机中内存访问的性能。1GB和2MB大页的不同点，主要在于它们的内存页大小不同，且1GB大页的分配只能在Linux系统的内核启动

参数中指定，而2MB大页既可以在启动时内核参数配置，又可以在系统运行中用命令行操作来配置。

下面介绍一下在KVM环境中使用1GB大页的具体操作步骤。

1）检查硬件和内核配置对1GB大页的支持，命令行如下：

```
[root@jay-linux ~]# cat /proc/cpuinfo | grep pdpe1gb
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni
pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr
pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes
xsave avx lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow
vnmi flexpriority ept vpid
[root@jay-linux ~]# grep HUGETLB /boot/config-3.5.0
CONFIG_HUGETLBFS=y
CONFIG_HUGETLB_PAGE=y
```

Intel的一些新的服务器平台（如Westmere、SandyBridge、IvyBridge等）都是支持1GB大页特性的，部分桌面级平台可能不支持1GB大页。如上面的命令所示，在Linux系统中，可以查看"pdpe1gb"这个标志在/proc/cpuinfo中的存在来确定硬件和内核是否有1GB的支持，另外2MB大页在/proc/cpuinfo中的标志为"pse"。而在内核配置中，对大页（1GB和2MB）的支持主要查看"HUGETLB"相关的支持。

2）在宿主机的内核启动参数中配置1GB，例如，在启动时使用6个1GB大页的grub配置文件如下：

```
title KVM Demo
        root (hd0,0)
        kernel (hd0,0)/boot/vmlinuz-3.5.0 ro root=/dev/sda1 \
hugepagesz=1GB hugepages=6 default_hugepagesz=1GB
        initrd (hd0,0)/boot/initrd-3.5.0.img
```

在上面的内核启动选项中，与大页相关的几个选项如下：

❑hugepagesz表示HugeTLB内存页的大小，在x86-64平台上其值为"2MB"或"1GB"。

❑hugepages表示在启动时大页分配的数量。

❑default_hugepagesz表示在挂载hugetlb文件系统时，没有设置大页的大小时默认使用的大页的大小。如果不设置这个选项，在x86-64平台上，其默认值为2MB。

对于这3个选项，有如下两点需要注意：

❑hugepagesz×hugepages的值一定不能大于系统的物理内存。

❑"hugepagesz"和"hugepages"选项可以成对地多次使用，可以让系统在启动时同时保留多个大小不同的大页（在x86-64上有2MB和1GB两种）。例如，"hugepagesz=1GB hugepages=6 default_hugepagesz=1GB hugepagesz=2MB hugepages=512"选项表示启动时系统保留6个1GB大页和512个2MB大页的内存，在挂载hugetlb时默认使用1GB大小的大页。

3）在启动宿主机后，在宿主机中查看内存信息和大页内存信息，命令行如下：

```
[root@jay-linux ~]# cat /proc/meminfo
MemTotal:       33033956 kB
MemFree:        26107172 kB
Buffers:           37280 kB
Cached:           174316 kB
<!-- 此处省略数十行输出信息 -->
HugePages_Total:       6
HugePages_Free:        6
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
<!-- 此处省略其余行的输出信息 -->
[root@jay-linux ~]# hugeadm --pool-list
      Size  Minimum  Current  Maximum  Default
1073741824        6        6        6        *
```

在上面的输出信息中，宿主机启动后，/proc/meminfo文件中的"MemFree"和"MemTotal"相差大于6GB，从侧面说明了系统启动时即保留了那6个1GB大页的内存。/proc/meminfo中的"Hugepagesize:1048576 kB"表示大页的大小为1GB，"HugePages_Total:6"表示总共有6个大页，"HugePages_Free:6"表示未被使用的大页有6个，"HugePages_Rsvd:0"表示因为别的程序使用而保留的大页有0个。"hugeadm"命令行工具是RHEL 6.3中"libhugetlbfs-utils" RPM包提供的配置大页资源池的一个工具，"hugeadm--pool-list"命令显示

了系统当前大页资源池的状态。如果使用前面提到的在启动时同时保留2MB和1GB大小大页的设置，则用"hugeadm"工具查看到的大页资源池状态的示例为：

```
[root@jay-linux ~]# hugeadm --pool-list
      Size  Minimum  Current  Maximum  Default
   2097152      512      512      512
1073741824        6        6        6        *
```

4）挂载hugetlbfs文件系统，命令行如下：

```
[root@jay-linux ~]#  mount -t hugetlbfs hugetlbfs /dev/hugepages
```

如果使用了两种大小的大页，可以在挂载hugetlbfs文件系统时，通过"pagesize"选项来指定挂载hugetlbfs的大页的大小，如下命令行指定了使用2MB大页（而不是1GB）：

```
mount -t hugetlbfs hugetlbfs /dev/hugepages -o pagesize=2MB
```

5）使用qemu-kvm命令启动客户机，"-mem-path"参数为其提供1GB大页的支持，命令行操作如下：

```
[root@jay-linux ~]# qemu-system-x86_64 -m 6G -smp 2 rhel6u3.qcow
-net nic -net tap -mem-path /dev/hugepages/
VNC server running on '::1:5900'
```

在使用1GB大页时，笔者发现，启动客户机的内存不能超过"-mem-path"指定的目录的1GB大页的内存量（这里为6GB），否则可能会出

现"can't mmap RAM pages:Cannot allocate memory"的错信息，从而不会为客户机提供任何大页的实际支持。

6）此时，在宿主机中再次查看内存情况，命令行如下：

```
[root@jay-linux ~]# cat /proc/meminfo
MemTotal:        33033956 kB
MemFree:         25133412 kB
Buffers:            39284 kB
Cached:           1073284 kB
<!-- 此处省略数十行输出信息 -->
HugePages_Total:        6
HugePages_Free:         3
HugePages_Rsvd:         3
HugePages_Surp:         0
Hugepagesize:     1048576 kB
<!-- 此处省略其余行的输出信息 -->
```

从/proc/meminfo中的"HugePages_Free:3"可以看出，只有3个1GB大页空闲了，说明已经分配3个1GB大页给客户机qemu-kvm进程了。从此，当客户机中应用程序真正请求分配1GB大页时，宿主机就可以帮它在物理内存上分配1GB大页了。（目前，由于对大页的支持需要显式调用libhugetlb库的函数，因此正式发布的应用程序中能使用大页的并不多，据笔者了解，Oracle、Sybase等数据库这样的大程序可以配置使用大页。）

和2MB大页一样，使用1GB的大页时，存在的问题也是一开始就需要预留大页的内存（不能分配给普通4KB页使用），不能被交换（swap）到交换分区，也不能使用ballooning方式使用大页内存。不过，在硬件、宿主机和客户机都支持1GB大页时，在客户机中使用1GB大页的应用程序对内存访问效率的提升是比较明显的，比2MB大页的效果更好。总之，在了解了它的利弊之后，对于对内存性能需求很高的应用，在KVM中可以选择使用1GB大页的特性。

5.7.2 透明大页

在4.3.3和5.7.1中已经介绍过，使用大页（huge page）可以提高系统内存的使用效率和性能，不过大页有如下几个缺点：

1）大页必须在使用前就预留下来（1GB大页还只能在启动时分配）。

2）应用程序代码必须显式使用大页（一般是调用libhugetlbfs API来分配大页）。

3）大页必须常驻物理内存中，不能给交换到交换分区中。

4）需要超级用户权限来挂载hugetlbfs文件系统。

5）如果预留了大页内存但没实际使用就会造成物理内存的浪费。

透明大页（Transparent Hugepage）正是发挥了大页的一些优点，又能避免了上述缺点。透明大页（THP）是Linux内核的一个特性，由Redhat的工程师Andrea Arcangeli在2009年实现的，然后在2011年的Linux内核版本2.6.38中被正式合并到内核的主干开发树中。目前一些流行的Linux发行版（如RHEL 6.3、Ubuntu 12.04等）的内核都默认提供了透明大页的支持。

透明大页，如它的名称描述的一样，对所有应用程序都是透明的（transparent），应用程序不需要任何修改即可享受透明大页带来的好处。在使用透明大页时，普通的使用hugetlbfs大页依然可以正常使用，而在没有普通的大页可供使用时，才使用透明大页。透明大页是可交换的（swapable），当需要交换到交换空间时，透明的大页被打碎为常规的4KB大小的内存页。在使用透明大页时，如果因为内存碎片导致大页内存分配失败，这时系统可以优雅地使用常规的4KB页替换，而且不会发生任何错误、故障或用户态的通知。而当系统内存较为充裕、有很多的大页面可用时，常规的页分配的物理内存可以通过khugepaged进程自动迁往透明大页内存。内核进程khugepaged的作用是，扫描正在运行的进程，然后试图将使用的常规内存页转换到使用大页。目前，透明大页仅仅支持匿名内存（anonymous memory）的映射，对磁盘缓存（page cache）和共享内存（shared memory）的透明大页支持还处于开发之中。

下面看一下使用透明大页的步骤。

1）在编译Linux内核时，配置好透明大页的支持，配置文件中的示例如下：

```
CONFIG_TRANSPARENT_HUGEPAGE=y
CONFIG_TRANSPARENT_HUGEPAGE_ALWAYS=y
# CONFIG_TRANSPARENT_HUGEPAGE_MADVISE is not set
```

这表示默认对所有应用程序的内存分配都尽可能地使用透明大页。当然，还可以在系统启动时修改Linux内核启动参数"transparent_hugepage"来调整这个默认值，其取值为如下3个值之一：

```
transparent_hugepage=[always|madvise|never]
```

2）在运行的宿主机中配置透明大页的使用方式，命令行如下：

```
[root@jay-linux ~]# cat
/sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
[root@jay-linux ~]# cat
/sys/kernel/mm/transparent_hugepage/defrag
[always] madvise never
[root@jay-linux ~]# cat \
```

```
/sys/kernel/mm/transparent_hugepage/khugepaged/defrag
1
[root@jay-linux ~]# echo "never" >\
/sys/kernel/mm/transparent_hugepage/defrag
[root@jay-linux ~]# cat /sys/kernel/mm/transparent_hugepage/defrag
always madvise [never]
```

在本示例的系统中，"/sys/kernel/mm/transparent_hugepage/enabled"接口的值为"always"，表示尽可能地在内存分配中使用透明大页；若将该值设置为"madvise"，则表示仅在"MADV_HUGEPAGE"标识的内存区域使用透明大页（在嵌入式Linux系统中内存资源比较珍贵，为了避免使用透明大页可能带来的内存浪费，如申请一个2MB内存页但只写入了1 Byte的数据，可能选择"madvise"方式使用透明大页）；若将该值设置为"never"则表示关闭透明内存大页的功能。当设置为"always"或"madvise"之时，系统会自动启动"khugepaged"这个内核进程去执行透明大页的功能；当设置为"never"时，系统会停止"khugepaged"进程的运行。"transparent_hugepage/defrag"这个接口是表示系统在发生页故障（page fault）时同步地做内存碎片的整理工作，其运行的频率较高（在某些情况下可能会带来额外的负担）；而"transparent_hugepage/khugepaged/defrag"接口表示在khugepaged进程运行时进行内存碎片的整理工作，它运行的频率较低。

当然还可以在KVM客户机中也使用透明大页，这样在宿主机和客户机同时使用的情况下，更容易提高内存使用的性能。

关于透明大页的详细配置可以阅读本章末尾“参考阅读”中的资料。另外，一些Linux发型版对透明大页的sys文件系统的接口可能有些不一致，在RHEL 6.3系统默认的对透明大页的设置中，各个选项的取值也和Linux 3.5上的有点不一样。下面就是RHEL 6.3中关于透明大页的默认设置：

```
# cat /sys/kernel/mm/redhat_transparent_hugepage/enabled
[always] never
# cat /sys/kernel/mm/redhat_transparent_hugepage/defrag
[always] never
# cat /sys/kernel/mm/redhat_transparent_hugepage/khugepaged/defrag
[yes] no
```

3）查看系统使用透明大页的效果，可以通过查看"/proc/meminfo"文件中的"AnonHugePages"这行来看系统内存中透明大页的大小，命令行如下：

```
[root@jay-linux ~]# cat /proc/meminfo | grep -i AnonHugePages
AnonHugePages:    688128 kB
[root@jay-linux ~]# echo $((688128/2048))
336
```

由上面的输出信息可知，当前系统使用了336个透明大页，透明大页内存大小为688128 KB。

在KVM 2010年论坛（KVM Forum 2010）[13] 上，透明大页的作者Andrea Arcangeli发表的一个题为"Transparent Hugepage Support"的演讲，其中展示了不少透明大页对性能提升的数据。图5-23展示了在是否打开宿主机和客户机的透明大页（THP）、是否打开Intel EPT特性时，内核编译（kernel build）所需时间长度的对比，时间越短说明效率越高。

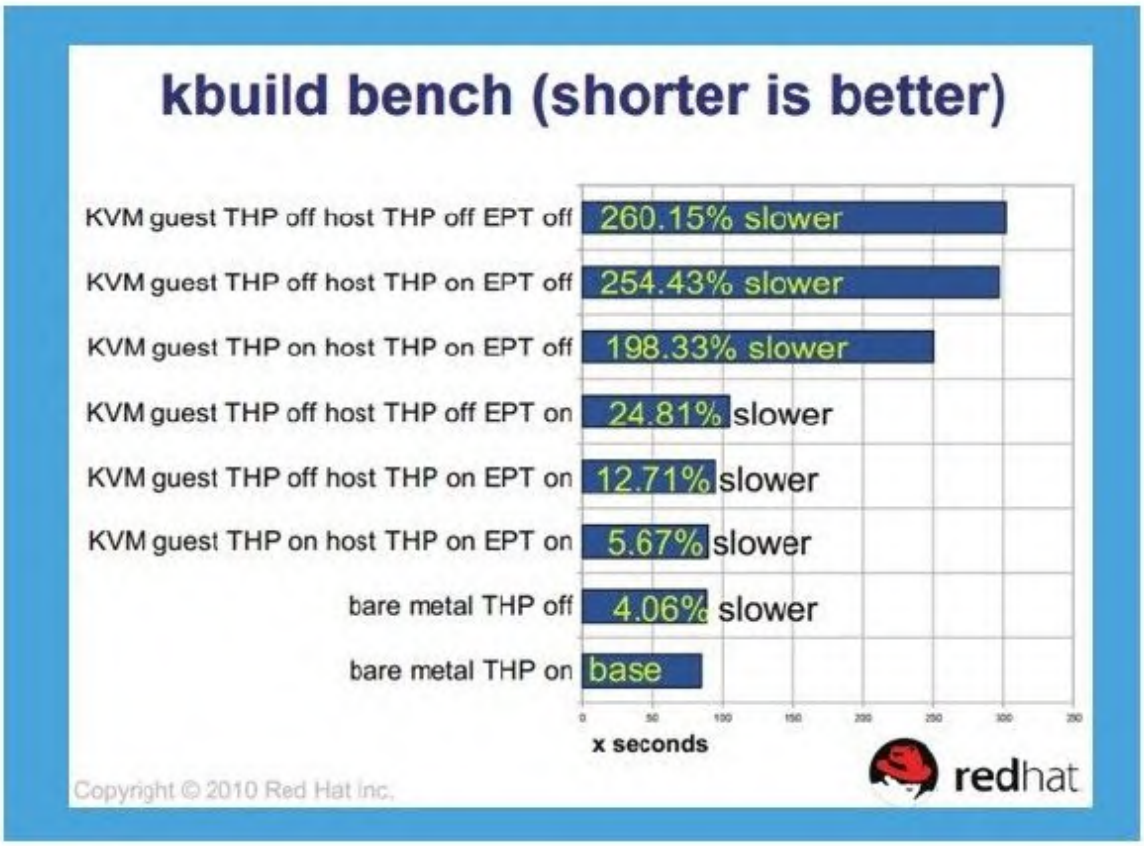


图　5-23　打开EPT、THP对内核编译效率的提高

由图5-23的数据可知，当打开EPT的支持且在宿主机和客户机中同时打开透明大页的支持时，内核编译的效率只比原生的非虚拟化环境

中的系统降低了5.67%；而打开EPT但关闭宿主机和客户机的透明大页时，内核编译效率比原生系统降低了24.81%；EPT和透明大页都全部关闭时，内核编译效率比原生系统降低了260.15%。这些数据说明了，EPT对内存访问效率有很大的提升作用，透明大页对内存访问效率也有较大的提升作用。

5.7.3 AVX和XSAVE

AVX（Advanced Vector Extensions，高级矢量扩展）是Intel和AMD的x86架构指令集的一个扩展，它最早是Intel在2008年3月提出的指令集，在2011年Intel发布Sandy Bridge处理器时开始第一次正式支持AVX，随后AMD最新的处理器也支持AVX指令集。Sandy Bridge是Intel处理器微架构从2009年发布的Nehalem架构后的革新，而引入AVX指令集是Sandy Bridge处理中引入的一个较大的特性，甚至有人将AVX指令认为是Sandy Bridge中最大的亮点，其重要性堪比1999年Pentium Ⅲ处理中引入的SSE（流式SIMD扩展）指令集。AVX中的新特性有：将向量化宽度从128为提升到256位，且将XMM0～XMM15寄存器重命名为YMM0～YMM15；引入了三操作数、四操作数的SIMD指令格式；弱化了对SIMD指令中对内存操作对齐的要求，支持灵活的不对齐内存地址访问。

向量就是多个标量的组合，通常意味着SIMD（单指令多数据），就是一个指令同时对多个数据进行处理，达到很大的吞吐量。早期的超级计算机大多都是向量机，而随着图形图像、视频、音频等多媒体的流行，PC处理器也开始向量化。X86上最早出现的是1996年的MMX（多媒体扩展）指令集，之后是1999年的SSE（流式SIMD扩展）指令集，它们分别是64位向量和128位向量，比超级计算机用的要短得多，所以叫做“短向量”。SandyBridge的AVX指令集将向量化宽度扩展到了256位，原有的16个128位XMM寄存器扩充为256位的YMM寄存器，可以同时处理8个单精度浮点数和4个双精度浮点数。不过，目前AVX的256位向量仅支持浮点，不像128位的SSE那样能支持整数运算。

对于AVX指令的支持，在CPU硬件方面，Intel的Sandy Bridge、Ivy Bridge、Haswell（将于2013年发布）及AMD的Bulldozer处理都提供AVX指令集。在编译器方面，GCC（the GNU Compiler Collection）4.6版本、ICC（Intel C++Compiler）11.1版本、Visual Studio 2010都已经支持AVX了。在操作系统方面，Linux内核2.6.30、Window 7 SP1、Windows 2008 R2 SP1、Mac OS 10.6.8等操作系统都提供了对AVX的支持。不过在应用程序方面，目前只有较少的软件提供了对AVX的支持，因为新的指令集的应用需要一定的时间，就像当年SSE指令集出来后，也是过了几年后才被多数软件支持。据了解，Prime95/MPrime（用于GIMPS[14] 项目中搜索梅森素数的分布式网络计算的软件）在其27.7版本中提供了对AVX稳定的支持，相比于不支持

AVX的26版本，27.7版本提升了大约30%的整体计算性能。AVX指令对浮点运算的性能提升是明显的，可主要应用于多媒体（音视频解码、3D图像渲染、3D游戏、HTML5展示等）、科学计算、金融分析计算及一些并行计算的领域。

另外，XSAVE指令（包括XSAVE、XRSTOR等）是在Intel Nehalem处理器中开始引入的，是为了保存和恢复处理器扩展状态的，在AVX引入后，XSAVE也要处理YMM寄存器状态。在KVM虚拟化环境中，客户机的动态迁移需要保存处理器状态，然后在迁移后恢复处理器的执行状态，如果有AVX指令要执行，在保存和恢复时也需要XSAVE/XRSTOR指令的支持。

下面介绍一下如何在KVM中为客户机提供AVX、XSAVE特性。

1）检查宿主机中AVX、XSAVE的支持，Intel Sandy Bridge之后的硬件平台都支持，较新的Linux内核（如3.x）也支持，命令行如下：

```
[root@jay-linux ~]# cat /proc/cpuinfo | grep avx | grep xsave
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid
```

2）启动客户机，将AVX、XSAVE特性提供给客户机使用，命令行操作如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -smp 2 -m 1024
rhel6u3.img -cpu host -net nic -net tap
VNC server running on '::1:5900'
```

3）在客户机中，查看QEMU提供的CPU信息中是否支持AVX和XSAVE，命令行如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo | grep avx | grep xsave

flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb
rdtscp lm constant_tsc arch_perfmon rep_good unfair_spinlock pni
pclmulqdq vmx ssse3 cx16 sse4_1 sse4_2 x2apic popcnt aes xsave
avx hypervisor lahf_lm xsaveopt
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb
rdtscp lm constant_tsc arch_perfmon rep_good unfair_spinlock pni
pclmulqdq vmx ssse3 cx16 sse4_1 sse4_2 x2apic popcnt aes xsave
avx hypervisor lahf_lm xsaveopt
```

由上面输出可知，客户机已经检测到CPU有AVX和XSAVE的指令集支持了，如果客户机中有需要使用到它们的程序，就可正常使用，从而提高程序执行的性能了。

另外，Intel将于2013年发布的Haswell处理器平台将会引入新的指令集AVX2（AVX指令集的扩展），它将会提供包括支持256位向量的

整数运算在内的更多功能。在qemu-kvm命令行中，用"-cpu host"参数也可以将AVX2的特性提供给客户机使用。

5.7.4 AES新指令

AES（Advanced Encryption Standard，高级加密标准）是一种用于对电子数据进行加密的标准，它在2001年就被美国政府正式接纳和采用。软件工业界广泛采用AES用于对个人数据、网络传输数据、公司内部IT基础架构等进行加密保护。AES的区块长度固定为128位，密钥长度则可以是128、192或256位。随着大家对数据加密越来越重视，以及AES应用越来越广泛，并且AES算法用硬件实现的成本并不高，一些硬件厂商（包括Intel、AMD等）都在自己的CPU中直接实现了针对AES算法的一系列指令，从而提高AES加解密的性能。

AES-NI（Advanced Encryption Standard new instructions，AES新指令）是Intel在2008年3月提出的在x86处理器上的指令集扩展。它包含了7条新指令，其中6条指令是在硬件上对AES的直接支持，另外一条是对进位乘法的优化，从而在执行AES算法的某些复杂的、计算密集型子步骤时使程序能更好地利用底层硬件，减少计算所需的CPU周期，提升AES加解密的性能。Intel公司从Westmere平台开始就支持AES-NI，目前Westmere、SandyBridge、IvyBridge、Haswell等平台的服务器多数都支持AES-NI。

目前有不少的软件都已经支持AES新指令了，如OpenSSL（1.0.1版本以上）、Oracle数据库（11.2.0.2版本以上）、7-Zip（9.1版本以上）、Libgcrypt（1.5.0以上）、Linux的Crypto API等。在KVM虚拟化环境中，如果客户机支持AES新指令（如RHEL6.0以上版本的内核都支持AES-NI），而且在客户机中用到AES算法加解密，那么将AES新指令的特性提供给客户机使用，会提高客户机的性能。

笔者在KVM的客户机中对AES-NI进行了测试，对比在使用AES-NI新指令和不使用AES-NI的情况对磁盘进行加解密的速度。下面介绍一下AES新指令的配置和测试过程及测试结果。

1）在进行测试之前，检查硬件平台是否支持AES-NI，一般来说如果CPU支持AES-NI，则会默认暴露到操作系统中去。而有一些BIOS中的CPU configuration下有一个"AES-NI Intel"这样的选项，也需要查看并且确认打开AES-NI的支持。不过，在设置BIOS时需要注意，笔者在一台Romley-EP的BIOS中设置了"Advanced"→"Processor Configuration"→"AES-NI Defeature"的选项，需要看清楚这里是设置"Defeature"而不是"Feature"，所以这个"AES-NI Defeature"应该设置为"disabled"（其默认值也是"disabled"），表示打开AES-NI功能。而BIOS中没有AES-NI相关的任何设置之时，就需要到操作系统中加载"aesni_intel"等模块来确认硬件是否提供了AES-NI的支持。

2）需要保证在内核中将AES-NI相关的配置项目编译为模块或直接编译进内核。当然，如果不是通过内核来使用AES-NI而是直接应用程序的指令使用它，则该步对内核模块的检查来说不是必要的。RHEL 6.3的内核关于AES的配置如下：

```
CONFIG_CRYPTO_AES=m
CONFIG_CRYPTO_AES_X86_64=m
CONFIG_CRYPTO_AES_NI_INTEL=m
```

3）在宿主机中，查看/proc/cpuinfo中的AES-NI相关的特性，并加载"aseni_intel"这个模块，命令行操作如下：

```
[root@jay-linux ~]# cat /proc/cpuinfo | grep aes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx lahf_lm
ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
[root@jay-linux ~]# lsmod | grep aes
aes_x86_64              7316  0
aes_generic            25994  1 aes_x86_64
[root@jay-linux ~]# modprobe aesni_intel
[root@jay-linux ~]# lsmod | grep aes
aesni_intel            39274  0
cryptd                  6667  1 aesni_intel
aes_x86_64              7316  1 aesni_intel
aes_generic            25994  2 aesni_intel,aes_x86_64
```

在加载"aesni_intel"模块的过程中，可能遇到如下的错误。这种情况是硬件不支持AES-NI或BIOS屏蔽了AES-NI特性造成的。

```
[root@jay-linux ~]# modprobe aesni_intel
FATAL: Error inserting aesni_intel
(/lib/modules/3.5.0/kernel/arch/x86/crypto/aesni-intel.ko): No such device
```

如果是"aesni_intel"模块没有正确编译，则会出现如下的错误提示信息。

```
[root@jay-linux ~]# modprobe aesni_intel
FATAL: Module aesni_intel not found.
```

4）启动KVM客户机，默认qemu-kvm启动客户机时，没有向客户机提供AES-NI的特性，可以用"-cpu host"或"-cpu qemu64,+aes"选项来暴露AES-NI特性给客户机使用。当然，由于前面提及一些最新的CPU系列是支持AES-NI的，所以也可用"-cpu Westmere"、"-cpu SandyBridge"这样的参数提供相应的CPU模型，从而提供对AES-NI特性的支持。

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -smp 4 -m 4096
rhel6u3.img -cpu host
VNC server running on '::1:5900'
```

5）在客户机中可以看到aes标志在/proc/cpuinfo中也是存在的，然后像宿主机中那样加载"aesni_intel"模块使其能够用到AES-NI，再执行使用AES-NI测试程序，即可得出使用了AES-NI的测试结果。当然，为了衡量AES-NI带来的性能提升，还需要做对比测试，即不添加"-cpu host"等参数启动客户机从而没有AES-NI特性的测试结果。注意，如果在qemu-kvm启动命令行启动客户机时带了AES-NI参数，一些

程序使用AES算法时可能会自动加载"aesni_intel"模块，所以，为了确保没有AES-NI的支持，也可以用"rmod aesni_intel"命令移除该模块，再找到aesni-intel.ko文件并将其删除以防被自动加载。

笔者某次进行ASE-NI测试的硬件平台是一个IvyBridge桌面级平台的PC，宿主机内核是Linux3.5.0，qemu-kvm版本是1.1.1，客户机使用的是RHEL 6.3原生系统。测试工具是使用一个特定测试脚本来测试对ramdisk（内存磁盘）进行AES加密后的读写速度。在测试脚本中调用"cryptsetup"这个命令行工具来对ramdisk建立设备映射（device mapper），然后用AES算法加密，最后cryptsetup会调用到Linux内核的Crypto API（前面已提到它是支持AES-NI的）。接着，用"dd"命令来分别读写ramdisk以得到读写速度的数据。测试脚本的内容如下：

```
#!/bin/bash

# create a crypt device using cryptsetup (dmsetup)
create_dm()
{
        echo 123456 | cryptsetup create $1 $2 -c aes-ecb-plain
        # echo 123456 | cryptsetup create $1 $2 -c aes-cbc-plain
        # echo 123456 | cryptsetup create $1 $2 -c aes-ctr-plain
        # echo 123456 | cryptsetup create $1 $2 -c aes-lrw-plain
        # echo 123456 | cryptsetup create $1 $2 -c aes-pcbc-plain
        # echo 123456 | cryptsetup create $1 $2 -c aes-xts-plain
}

# remove the device-mapper using cryptsetup
remove_dm()
```

```
{
        if [ -d /sys/block/dm-0 ]; then
                cryptsetup remove crypt0 &> /dev/null
        fi
}

remove_dm

# try to write some data to ram-disk /dev/ram0; just a pre-test before the main test.
dd if=/dev/zero of=/dev/ram0 bs=1M count=128 >& /dev/null
if [ $? -ne 0 ]; then
        echo "there may be no enough ramdisk space on /dev/ram0 ."
        echo "you may add 'ramdisk_size=154112' in kernel option
to extend ram-disk size and reboot the system."
        echo "Or, you can decrease the value of 'count=xx' option in the 'dd' command."
fi

create_dm crypt0 /dev/ram0

# check /dev/dm-0 ; if the previous command is executed
successfully, /dev/md-0 should exists.
while ! [ -e /dev/dm-0 ];
do
        echo "/dev/dm-0 doesn't exist."
        echo "please check if the command 'cryptsetup create xx' is executed successfully."
        sleep 1
done

# do read or write action for several times, so that you can
calculate their average value.
for i in $(seq 10); do
       # the following line is for read action
       dd if=/dev/dm-0 of=/dev/null bs=1M count=128

       # the following line is for write action
       #dd of=/dev/dm-0 if=/dev/zero bs=1M count=128
done

remove_dm
```

在该脚本中，选取"/dev/ram0"这个ramdisk作为示例，在RHEL 6.3的默认内核启动时，默认有16个ramdisk，每个ramdisk大小为16MB。这里在脚本中使用dd命令读取或写入的数据是128MB，所以需要调整ramdisk大小，可以在其Grub配置文件的内核的启动行加

上"ramdisk_size="参数来修改ramdisk大小，然后重启系统即可。"ramdisk_size="参数后数值的单位是KB，所以设置ramdisk大小为128MB，其参数就为"ramdisk_size=154112"（128*1024）。另外，也可以修改脚本，将"dd"命令的"count"参数进行调整（变小），以匹配当前系统中ramdisk的实际大小。在RHEL 6.3内核中，对ramdisk的默认配置项如下：

```
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_COUNT=16
CONFIG_BLK_DEV_RAM_SIZE=16384
```

另外，在执行脚本中"cryptsetup create"命令时，会出现如下的错误：

```
[root@kvm-guest ~]# echo 123456 | cryptsetup create crypt0
/dev/ram0 -c aes-ctr-plain
device-mapper: reload ioctl on  failed: No such file or directory
```

这有可能是内核配置的问题，缺少一些加密相关的模块，在本示例中，需要确保内核配置中有如下的内核配置项：

```
CONFIG_CRYPTO_CBC=m
CONFIG_CRYPTO_CTR=m
CONFIG_CRYPTO_CTS=m
CONFIG_CRYPTO_ECB=m
CONFIG_CRYPTO_LRW=m
CONFIG_CRYPTO_PCBC=m
CONFIG_CRYPTO_XTS=m
CONFIG_CRYPTO_FPU=m
```

笔者实验得到的测试结果如表5-1所示，其中，"AES-NI-Read"表示有AES-NI支持的情况下读取ramdisk的速度，"Soft-AES-Read"表示没有AES-NI支持（用软件实现AES算法）的情况下读取ramdisk的速度，同理，带有"Write"字样的为写入ramdisk的速度。所有的速度数值的单位是"MB/s"（兆字节每秒），最后一行是各种AES在加密情况下综合起来的读写速度的平均值。

表 5-1　使用 AES-NI 对 ramdisk 进行加密读写的测试数据

AES-Mode	AES-NI-Read	Soft-AES-Read	AES-NI-Write	Soft-AES-Write
aes-ecb-plain	664.1	151.7	290	175
aes-cbc-plain	718.5	139.8	268	209.65
aes-ctr-plain	269.8	135.2	229	195.5
aes-lrw-plain	314.3	167.5	270	208.5
aes-pcbc-plain	219.6	99	193	175
aes-xts-plain	304	161.2	273.5	230
Average（均值）	**415.05**	**142.4**	**253.92**	**198.94**

从表5-1可以看出，在使用AES-NI硬件实现AES算法情况下对ramdisk的读写速度，是仅用软件实现AES算法时读写速度的2～3倍。

本小节首先对AES-NI新特性的介绍，然后对如何让KVM客户机也使用AES-NI的步骤进行了介绍，最后展示了笔者曾经某次AES-NI在KVM客户机中的实验数据。相信读者对AES-NI新指令有了较好的认识，在实际应用环境中如果物理硬件平台支持ASE-NI且客户机中用到AES相关的加解密算法，可以考虑将AES-NI特性暴露给客户机使用以便提高加解密性能。

5.7.5 完全暴露宿主机CPU特性

在4.2.4节“CPU模型”中介绍过，qemu-kvm提供qemu64作为默认的CPU模型，对于部分CPU特性，也可以通过“+”号来添加一个或多个CPU特性到一个基础的CPU模型之上，如前面介绍AES新指令时，可以选用"-cpu qemu64,+aes"参数来让客户机支持AES新指令。

当需要客户机尽可能使用宿主机和物理CPU支持的特性时，qemu-kvm也提供了"-cpu host"参数来尽可能多地暴露宿主机CPU特性给客户机，从而在客户机中可以看到和使用CPU的各种特性（如果QEMU/KVM都支持该特性）。在一个Intel IvyBridge平台上，对"-cpu host"参数的效果演示如下。

1）在KVM宿主机中查看CPU信息，命令行如下：

```
[root@jay-linux ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz
stepping        : 9
microcode       : 0x8
cpu MHz         : 1600.000
cache size      : 6144 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c
rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow

 vnmi flexpriority ept vpid fsgsbase smep erms
 bogomips        : 6585.08
 clflush size    : 64
 cache_alignment : 64
 address sizes   : 36 bits physical, 48 bits virtual
 power management:
 <!-- 省略其余逻辑CPU信息的输出 -->
```

2）用"-cpu host"参数启动一个RHEL 6.3客户机系统，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -net nic -net tap -cpu host
VNC server running on '::1:5900'
```

3）在客户机中查看CPU信息，命令行如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz
stepping        : 9
cpu MHz         : 3292.520
cache size      : 4096 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx rdtscp
lm constant_tsc up arch_perfmon rep_good unfair_spinlock pni
pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt aes xsave avx
f16c rdrand hypervisor lahf_lm xsaveopt fsgsbase smep erms
bogomips        : 6585.04
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

由上面客户机中CPU信息中可知，客户机看到的CPU模型与宿主机中一致，都是"Intel(R)Core(TM)i5-3550"；CPUID等级信息（cpuid

level）也和宿主机一致，在CPU特性标识（flags）中，也有了"aes"、"xsave"、"avx"、"rdrand"、"smep"等特性，这些较高级的特性是默认的"qemu64"CPU模型中没有的。说明"-cpu host"参数成功地将宿主机的特性尽可能多地提供给客户机使用了。

当然，"-cpu host"参数也并没有完全让客户机得到与宿主机同样多的CPU特性，这是因为QEMU/KVM对于其中的部分特性没有模拟和实现，如"EPT"、"VPID"等CPU特性目前就不能暴露客户机使用。另外，尽管"-cpu host"参数尽可能多地暴露宿主机CPU特性给客户机，可以让客户机用上更多的CPU功能也能提高客户机的部分性能，与此同时，"-cpu host"参数还是可能会阻止客户机的动态迁移。例如，在Intel的SandyBridge平台上，用"-cpu host"参数让客户机使用了AVX新指令进行运算，此时试图将客户机机器迁移到没有AVX支持的Intel Westmere平台上去，就可能会导致动态迁移的失败。所以，"-cpu host"参数尽管向客户机提供了更多的功能和更好的性能，还是需要根据使用场景谨慎使用。

5.8 KVM安全

如今，计算机信息安全越来越受到人们的重视，在计算机相关的各种学术会议、论文、期刊中“安全”（security）一词被经常提及。因为KVM是Linux系统中的一个虚拟化相关的模块，qemu-kvm是Linux系统上一个普通的用户空间进程，所以Linux系统上的各种安全技术、策略对QEMU/KVM都是适用的。在本节中笔者根据经验选择了其中的一些技术来介绍一下KVM的安全技术。

5.8.1 SMEP

有一些安全渗透（exploitation）攻击，会诱导系统在最高执行级别（ring0）上访问在用户空间（ring3）的数据和执行用户空间的某段恶意代码，从而获得系统的控制权或使系统崩溃。

SMEP（Supervision Mode Execution Protection，监督模式执行保护），是Intel在2012年发布的代号为"Ivy Bridge"的新一代CPU上提供的一个安全特性。当控制寄存器CR4寄存器的20位（第21位）被设置为1时，表示SMEP特性是打开状态。SMEP特性让处于管理模式（supervisor mode，当前特权级别CPL＜3）的程序不能获取用户模式（user mode，CPL=3）可以访问的地址空间上的指令，如果管理模式的程序试图获取并执行用户模式的内存上的指令，则会发生一个错误

（fault），不能正常执行。在SMEP特性的支持下，在运行管理模式的CPU不能执行用户模式的内存页，可以较好地阻止前面提到的那种渗透攻击，而且由于SMEP是CPU的一个硬件特性，这种安全保护非常方便和高效，其带来的性能开销几乎可以忽略不计。

目前，Linux 3.0之后的linux系统和Windows 8系统都已经支持SMEP的特性，而且一些Linux发行版（如RHEL 6.2/6.3）尽管使用的是Linux 2.6.32为基础的内核，但是也都向后移植（backport）了SMEP的特性。而较新的KVM（如Linux 3.4）和qemu-kvm（如1.2.0）让KVM客户机中也支持了SMEP特性。下面介绍一下如何让KVM客户机也可以使用SMEP的步骤。

1）检查宿主机中的CPU对SMEP特性支持，命令行如下所示：

```
[root@jay-linux kvm_demo]# cat /proc/cpuinfo  | grep smep
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr

 pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
 tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts
 rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
 dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
 sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c
 rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow
 vnmi flexpriority ept vpid fsgsbase smep erms
 <!-- 省略其余逻辑CPU的输出信息 -->
```

2）加上"-cpu host"参数来启动客户机，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -m
1024 -net nic -net tap -cpu host
VNC server running on '::1:5900'
```

3）在客户机中检查其CPU是否有SMEP特性的支持，命令行如下：

```
[root@kvm-guest ~]# cat /proc/cpuinfo | grep smep
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx rdtscp
lm constant_tsc up arch_perfmon rep_good unfair_spinlock pni
pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt aes xsave avx
f16c rdrand hypervisor lahf_lm xsaveopt fsgsbase smep erms
```

由上面的输出信息可知，客户机中已经能够检测到SMEP特性了，从而使客户机成为一个有Intel CPU的SMEP特性支持的较为安全的环境了。

5.8.2 控制客户机的资源使用——cgroups

在KVM虚拟化环境中，每个客户机操作系统使用系统的一部分物理资源（包括处理器、内存、磁盘、网络带宽等）。当一个客户机对资源的消耗过大时（特别是qemu-kvm启动客户机时没有能够控制磁盘或网络I/O的选项），它可能会占用该系统的大部分资源，此时，其他的客户机对相同资源的请求就会受到严重影响，可能会导致其他客户机响应速度过慢甚至失去响应。为了让所有客户机都能够按照预先的比例来占用物理资源，我们需要对客户机能使用的物理资源做控制。第4章中介绍过，通过qmue-kvm命令行启动客户机时，可以用"-smp

num"参数来控制客户机的CPU个数，使用"-m size"参数来控制客户机的内存大小，不过，这些都是比较粗粒度的控制，例如，不能控制客户机仅使用1个CPU的50%的资源，而且对磁盘I/O、网络I/O等并没有直接的参数来控制。由于每个客户机就是宿主机Linux系统上的一个普通qemu-kvm进程，所以可以通过控制qemu-kvm进程使用的资源来达到控制客户机的目的。

1.cgroups简介

cgroups（即control groups，控制群组）是Linux内核中的一个特性，用于限制、记录和隔离进程组（process groups）对系统物理资源的使用。cgroups最初是由Google的一些工程师（Paul Menage、Rohit Seth等人）在2006年以“进程容器”（process container）的名字实现的，在2007年被重命名为“控制群组”（Control Groups），然后被合并到Linux内核的2.6.24版本之中。在加入到Linux内核的主干之后，cgroups越来越成熟，有很多新功能和控制器（controller）被键入进去，其功能也越来越强大了。cgroups为不同的用户场景提供了一个统一的接口，这些用场景包括对单一进程的控制，也包括像OpenVZ、LXC（Linux Containers）等操作系统级别的虚拟化技术。一些较新的比较流行的Linux发行版，如RHEL 6.0、RHEL 6.3、Fedora 17、Ubuntu 12.10等，都提供了对cgroups的支持。

cgroups提供了如下的一些功能：

1）资源限制（Resource limiting），让进程组被设置为使用不能超过某个界限的资源数量。如内存子系统可以为进程组设定一个内存使用的上限，一旦进程组使用的内存达到限额后，如果再申请内存就会发生缺乏内存的错误（即：OOM，out of memory）。

2）优先级控制（Prioritization），让不同的进程组有不同的优先级。可以让一些进程组占用较大的CPU或磁盘I/O吞吐量的百分比，另一些进程组占用较小的百分比。

3）记录（Accounting），衡量每个进程组（包括KVM客户机）实际占用的资源数量，可以用于对客户机用户进行收费等目的，如使用cpuacct子系统记录某个进程组使用的CPU时间。

4）隔离（Isolation），对不同的进程组使用不同的命名空间（namespace），不同的进程组之间不能看到相互的进程、网络连接、文件访问等信息，如使用ns子系统就可以使不同的进程组使用不同的命名空间。

5）控制（Control），控制进程组的暂停、添加检查点、重启等，如使用freezer子系统可以将进程组挂起和恢复。

cgroups中有如下几个重要的概念，理解它们是了解cgroups的前提条件。

1）任务（task）：在cgroups中，一个任务就是Linux系统上的一个进程或线程，可以将任务添加到一个或多个控制群组中。

2）控制群组（control group）：一个控制群组就是按照某种标准划分的一组任务（进程）。在cgroups中，资源控制都是以控制群组为基本单位来实现的。一个任务可以被添加到某个控制群组，也从一个控制群组转移到另一个控制群组。一个控制群组中的进程可以使用以控制群组为单位分配的资源，同时也受到以控制群组为单位而设定的资源限制。

3）层级体系（hierarchy）：简称“层级”，控制群组被组织成有层级关系的一棵控制群组树。控制群组树上的子节点控制群组是父节点控制群组的孩子，继承父节点控制群组的一些特定属性。每个层级需要被添加到一个或多个子系统中，受到子系统的控制。

4）子系统（subsystem）：一个子系统就是一个资源控制器（resource controller），如blkio子系统就是控制对物理块设备的I/O访问的控制器。子系统必须附加到一个层级上才能起作用，一个子系统附加到某个层级以后，该子系统会控制这个层级上的所有控制群组。

目前cgroups中主要有如下10个子系统可供使用。

❑blkio：这个子系统为块设备（如磁盘、固态硬盘、U盘等）设定读写I/O的访问设置限制。

❑cpu：这个子系统通过使用进程调度器提供了对控制群组中的任务在CPU上执行的控制。

❑cpuacct：这个子系统为控制群组中的任务所实际使用的CPU资源自动生成报告。

❑cpuset：这个子系统为控制群组中的任务分配独立CPU核心（在多核系统）和内存节点。

❑devices：这个子系统可以控制一些设备允许或拒绝来自某个控制群组中的任务的访问。

❑freezer：这个子系统用于挂起或恢复控制群组中的任务。

❑memory：这个子系统为控制群组中任务能使用的内存设置限制，并能自动生成那些任务所使用的内存资源的报告。

❑net_cls：这个子系统使用类别识别符（classid）标记网络数据包，允许Linux流量控制程序（traffic controller）识别来自某个控制群组中任务的数据包。

❑ns：名称空间子系统。用于命名空间的隔离，使不同命令空间中控制群组的任务不能“看到”其他命名空间中的任务的信息。

❑perf_event：这个子系统主要用于对系统中进程运行的性能监控、采样和分析等。

在Redhat系列的系统中，如RHEL 6.3系统的libcgroup这个RPM包提供了lssubsys工具在命令包中查看当前系统支持那些子系统，命令行操作如下：

```
[root@jay-linux ~]# uname -a
Linux jay-linux 3.5.0 #8 SMP Sun Nov 11 22:51:54 CST 2012 x86_64
x86_64 x86_64 GNU/Linux
[root@jay-linux ~]# lssubsys -am
cpuset
cpu
cpuacct
memory
devices
freezer
net_cls
blkio
perf_event
net_prio
```

cgroups中层级体系的概念，与Linux系统中的进程模型有相似之处。在Linux系统中，所有进程也是组成树状的形式（可以用"pstree"命令查看），除init外的所有进程都是一个公共父进程，即init进程的子进程。init进程是由Linux内核在启动时执行的，它会启动其他进程（当然普通进程也可以启动自己的子进程）。除init进

程外的其他进程从其父进程那里继承环境变量（如$PATH变量）和其他一些属性（打开的文件描述符）。与Linux进程模型类似，cgroups的层级体系也是树状分层结构的，子节点控制群组继承父节点控制群组的一些属性。尽管有类似的概念和结构，但它们之间也一些区别。最主要的区别是，在Linux系统中可以同时存在cgroups的一个或多个相互独立的层级，而且此时Linux系统中只有一个进程树模型（因为它们有一个相同的父进程init进程）。多个独立的层级的存在也是有其必然性的，因为每个层级都会给添加到一个或多个子系统下。

图5-24展示了cgroups模型的一个示例，"cpu"、"memory"、"blkio"、"net_cls"是4个子系统，"Cgroup Hierarchy A"～"Cgroup Hierarchy C"是3个相互独立的层级，含有"cg"两字的就是控制群组（包括cpu_mem_cg、blk_cg、cg1、cg4等），"qemu-kvm"是一个任务（其PID为8201）。"cpu"和"memory"子系统同时附加到了"Cgroup Hierarchy A"这个层级上，"blkio"、"net_cls"子系统分别附加到了"Cgroup Hierarchy B"和"Cgroup Hierarchy C"这两个层级上。qemu-kvm任务被添加到这三个层级之中，它分别在cg1、cg2、cg3这3个控制群组中。

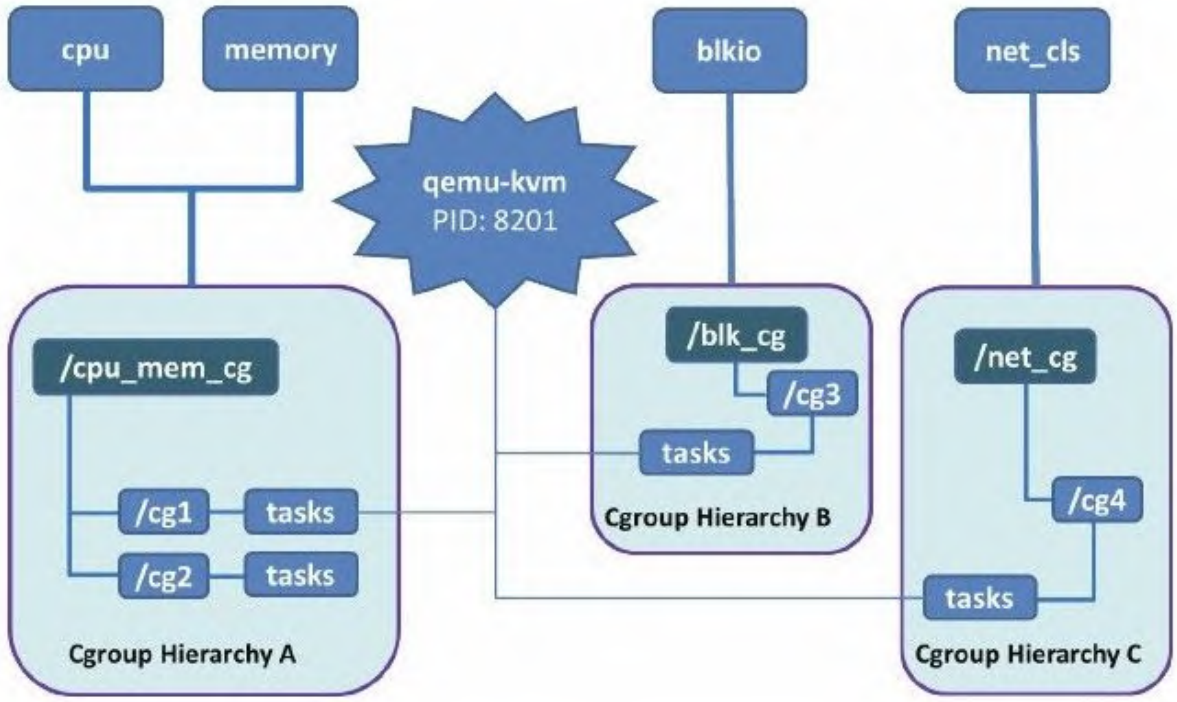


图　5-24　一个cgroups模型的示例

cgroups中的子系统、层级、控制群组、任务之间的关系，至少有如下几条规则需要遵循。

1）每个层级可以有一个或多个子系统附加上去。

在图5-24中，"Cgroup Hierarchy A"就有cpu、memory两个子系统，而"Cgroup Hierarchy B"只有blkio一个子系统。

2）只要其中的一个层级已经有一个子系统被附加上了，任何一个子系统都不能被添加到两个或多个层级上。

在图5-24中，memory子系统就不能同时添加到层级A和层级B之上。

3）一个任务不能同时是同一个层级中的两个或多个控制群组中的成员。一个任务一旦要成为同一个层级中的第二个控制群组中的成员，它必须先从第一个控制群组中移除。

在图5-24中，qemu-kvm任务就不能同时是层级A中的cg1和cg2这两个控制群组的共同成员。

4）派生（fork）出来的一个任务会完全继承它父进程的cgroups群组关系。当然，也可以再次调整派生任务的群组关系使其与它的父进程不同。

如图5-24所示，如果PID为8201的qemu-kvm进程派生出一个子进程，则该子进程也默认是cg1、cg3、cg4这三个控制群组的成员。

5）在每次初始化一个新的层级之时，该系统中所有的任务（进程或线程）都被默认添加该层级默认的控制群组中成为它的成员。该群组被称为根控制群组（root cgroup），在创建层级时自动创建，之后在该层级中创建的所有其他群组都是根控制群组的后代。

2.cgroups操作示例

在了解了cgroups的基本功能和原理后（关于cgroups更多描述可以参考本章末尾的“参考阅读”），本节将介绍一下如何实际操作cgroups来达到控制KVM客户机的资源使用。Linux内核提供了一个统一的cgroups接口来访问多个子系统（如cpu、memory、blkio等）。在编译Linux内核时，需要对cgroups相关的项目进行配置，如下是RHEL 6.3内核中与cgroups相关的一些重要配置。

```
CONFIG_CGROUP_SCHED=y
CONFIG_CGROUPS=y
# CONFIG_CGROUP_DEBUG is not set
CONFIG_CGROUP_NS=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
CONFIG_PROC_PID_CPUSET=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_RESOURCE_COUNTERS=y
CONFIG_CGROUP_MEM_RES_CTLR=y
CONFIG_CGROUP_MEM_RES_CTLR_SWAP=y
CONFIG_BLK_CGROUP=y
# CONFIG_DEBUG_BLK_CGROUP is not set
CONFIG_CGROUP_PERF=y
CONFIG_SCHED_AUTOGROUP=y
CONFIG_MM_OWNER=y
# CONFIG_SYSFS_DEPRECATED_V2 is not set
CONFIG_RELAY=y
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
CONFIG_NET_CLS_CGROUP=y
CONFIG_NETPRIO_CGROUP=y
```

在实际操作过程中，可以通过如下几个方式来使用cgroups。

1）手动地访问cgroups的虚拟文件系统。

2）使用libcgroup软件包中的cgcreate、cgexec、cgclassify等工具来创建和管理cgroups控制群组。

3）通过一些使用cgroups的规则引擎，通常是系统的一个守护进程来读取cgroups相关的配置文件，然后对任务进程相应的设置。例如，在RHEL 6.3系统中，如果安装了"libcgroup" RPM包，就会有cgconfig这个服务可以使用（"service cgconfig start"命令可以启动它），其配置文件默认为/etc/cgconfig.conf文件。

4）通过其他一些软件来间接使用cgroups，如使用操作系统虚拟化技术（LXC等）、libvirt工具等。

下面举个KVM虚拟化实际应用中的例子：一个系统中运行着两个客户机（它们的磁盘镜像文件都在宿主机的本地存储上），每个客户机中都运行着MySQL数据库服务，其中一个数据库的优先级很高（需要尽可能快地响应），另一个优先级不高（慢一点也无所谓）。我们知道数据库服务是磁盘I/O密集型服务，所以这里通过cgroups的blkio子系统来设置两个客户机对磁盘I/O读写的优先级，从而使优先级高的客户机能够占用更多的宿主机中的I/O资源。采用手动读写cgroups虚拟文件系统的方式，实现这个需求的操作步骤如下：

1）启动这两个客户机和其中的MySQL服务器，让其他应用开始使用MySQL服务。这里不再写出启动过程。假设需要优先级高的客户机在qemu-kvm命令行启动时加上了"-name high_prio"的参数来指定其名称，而优先级低的客户机有"-name low_prio"参数。在它们启动时为它们取不同的名称，仅仅是为了在后面操作中方便区别出两个客户机。

2）添加blkio子系统到/cgroup/blkio这个控制群组上，并创建高优先级和低优先级两个群组，命令行操作如下：

```
[root@jay-linux ~]# mkdir -p /cgroup/blkio
[root@jay-linux ~]# mount -t cgroup -o blkio blkio /cgroup/blkio
[root@jay-linux ~]# mkdir /cgroup/blkio/high_prio
[root@jay-linux ~]# mkdir /cgroup/blkio/low_prio
[root@jay-linux ~]# ls /cgroup/blkio/high_prio/
blkio.io_merged         blkio.reset_stats
blkio.throttle.write_bps_device   cgroup.event_control
blkio.io_queued         blkio.sectors
blkio.throttle.write_iops_device  cgroup.procs
blkio.io_service_bytes  blkio.throttle.io_service_bytes
blkio.time                      notify_on_release
blkio.io_serviced       blkio.throttle.io_serviced
blkio.weight                       tasks
blkio.io_service_time   blkio.throttle.read_bps_device
blkio.weight_device
blkio.io_wait_time      blkio.throttle.read_iops_device
cgroup.clone_children
```

其中"mount-t cgroup-o blkio blkio/cgroup/blkio"命令创建了名为blkio的层级，该层级被挂载到"/cgroup/blkio"目录上，并且将blkio子系统附加到该层级上。如果需要附加多个子系统到一个层级

上，在创建时选用多个子系统即可，如执行"mount-t cgroup-o cpu,cpuset,memorycpu_mem/cgroup/cpu_mem"命令就创建了名为cpu_mem的层级，并将cpu、cpuset、memory三个子系统附加到该层级上。

3）分别将高优先级和低优先级的客户机的qemu-kvm进程（及其子进程、子线程）移动到相应的控制群组下面去。可以使用下面的脚本执行，使高优先级客户机（用"high_prio"来标识）进程作为high_prio控制器群组的成员；同样修改一下该脚本也可以处理低优先级客户机。

```
#!/bin/bash

pid_list=$(ps -eLf | grep qemu | grep "high_prio" | awk '{print $4}')
for pid in $pid_list
do
       echo $pid >> /cgroup/blkio/high_prio/tasks
done
```

将控制群组正常分组后，分别查看high_prio和low_prio两个控制群组中的任务，命令行如下，其中7372是高优先级客户机qemu-kvm的进程ID，7374、7375、7377都是ID为7372进程的子线程的ID。低优先级客户机对应的low_prio群组中的信息也与此类似。

```
[root@jay-linux ~]# cat /cgroup/blkio/high_prio/tasks
7372
7374
7375
7377
[root@jay-linux ~]# cat /cgroup/blkio/low_prio/tasks
7398
7400
7401
7403
```

4）分别设置高低优先级的控制群组中块设备I/O访问的权重，这里假设高低优先级的比例为10:1，设置权重的命令行如下：

```
[root@jay-linux ~]# echo 1000 > /cgroup/blkio/high_prio/blkio.weight
[root@jay-linux ~]# echo 100 > /cgroup/blkio/low_prio/blkio.weight
```

顺便提一下，"blkio.weight"是在一个控制群组中设置块设备I/O访问相对比例（即权重）的参数，其取值范围是100～1000的整数。

5）块设备I/O访问控制的效果分析。假设宿主机系统中磁盘I/O访问的最大值是每秒写入66 MB，除客户机的qemu-kvm进程之外的其他所有进程的磁盘I/O访问可以忽略不计，那么在未设置两个客户机的块设备I/O访问权重之前，在满负载的情况下，高优先级和低优先级的客户机对实际磁盘的写入速度会同时达到约33 MB/s，而在通过10:1的比例设置了它们的权重之后，高优先级客户机对磁盘的写入速度可以达到约60 MB/s，而低优先级的客户机可达到约6 MB/s。

在KVM虚拟化环境中，通过使用cgroups，系统管理员可以对客户机进行细粒度的控制，包括资源分配、优先级调整、拒绝某个资源的访问、监控系统资源利用率。硬件资源可以根据不同的客户机进行“智能”的分组，从整体上提高了资源利用效率，从某种角度来说，也可以提高各个客户机之间的资源隔离性，提升了KVM虚拟机的安全性。

5.8.3 SELinux和sVirt

1.SELinux简介

SELinux（Security-Enhanced Linux）是一个Linux内核的一个安全特性，是通过使用Linux内核中的Linux安全模块（Linux Security Modules，LSM）提供了一种机制来支持访问控制的安全策略，包括美国国防部风格的强制访问控制（Mandatory Access Controls，MAC）。简单地说，SELinux提供了一种安全访问机制的体系，在该体系中进程只能访问那些其任务中所需要的文件。SELinux中的许多概念是来自美国国家安全局（National Security Agency，NSA）的一些早期项目。SELinux特性是从2003年发布Linux内核2.6版本开始进入到内核主干开发树中的。在一些Linux发型版中，可以通过使用SELinux配置的内核和在用户空间的管理工具来使用SELinux，目前RHEL 6.x、

Fedora、CentOS、OpenSuse、SLES、Ubuntu等发行版中都提供对SELinux的支持。

使用SELinux可以减轻恶意攻击、恶意软件等带来的灾难损失，并提供对机密性、完整性有较高要求的信息的安全保障。普通的Linux和传统的UNIX操作系统一样，在资源访问控制上采用自由访问控制（Discretionary Access Controls，DAC）策略，只要符合规定的权限（如文件的所有者和文件属性等），就可以访问资源。在这种传统的安全机制下，一些通过setuid或setgid的程序就可能产生安全隐患，甚至错误的配置可能引发很大的安全漏洞，让系统处于脆弱的容易被攻击的状态。而SELinux是基于强制访问控制（MAC）策略的，应用程序或用户必须同时满足传统的DAC和SELinux的MAC访问控制才能进行资源的访问操作，否则都会被拒绝或返回失败，但这个过程不影响其他的应用程序，保持了系统的安全性。

强制访问控制（MAC）模式为每一个应用程序都提供了一个虚拟“沙箱”，只允许应用程序执行它设计需要的且在安全策略中明确允许的任务，对每个应用程序只分配它正常工作所需的对应特权。例如，Web服务器可能只能够读取网站发布目录中的文件，并监听指定的网络端口。即使攻击者将其攻破，他们也无法执行在安全策略中没有明确允许的任何活动，即使这个进程在超级用户（root）下运行。传统Linux的权限控制仍然会在系统中存在，并且当文件被访问时，此权

限控制将先于SELinux安全策略生效。如果传统的权限控制拒绝本次访问，则该访问直接被拒绝，SELinux在整个过程中没有参与。但是，如果传统Linux权限允许访问，SELinux此时将进一步对其进行访问控制检查，并根据其源进程和目标对象的安全上下文来判断允本次许访问还是拒绝访问。

2.sVirt简介

在非虚拟化环境中，每台服务器是物理硬件上隔离的，一般通过网络来进行相互的通信，如果一台服务器被攻击了，它一般只会让该服务器不能正常工作，当然针对网络的攻击也可能影响到其他服务器。在KVM这样的虚拟化环境中，有多个客户机服务器运行在一个宿主机上，它们使用相同的物理硬件，如果一个客户机被攻击了，攻击者可能会利用被攻陷的服务器发起对宿主机的攻击，如果Hypervisor有bug，则可能会让攻击很容易从一个客户机蔓延到其他客户机上去。

在传统的文件权限管理中，一般是将用户分为所有者（owner）、与所有者相同的用户组（group）和其他用户（others）。在虚拟化应用中，如果使用一个用户账号启动了多个客户机，那么当其中一个客户机qemu-kvm进程有异常行为时，它可能会对其他客户机的镜像文件有读写权限，可能会让其他客户机也暴露在不安全的环境中。当然可以使一个客户机对应一个用户账号，使用多个账号来分别启动多个不同的客户机，以便使客户机镜像访问权限隔离，而当一个宿主机中有

多达数十个客户机时，就需要对应数十个用户账号，管理和使用起来都会非常不方便。

sVirt是Redhat公司开发的针对虚拟化环境的一种安全技术，它主要集成了SELinux和虚拟化。sVirt通过对虚拟化客户机使用强制访问控制来提高安全性，它可以阻止因为Hypervisor的bug而导致的从一台客户机向宿主机或其他客户机的攻击。sVirt让客户机之间相互隔离得比较彻底，而且即使某个客户机被攻陷了，也能够限制它发起进一步攻击的能力。

sVirt还是通过SELinux来起作用的，图5-25展示了sVirt阻止来自一个客户机的攻击。

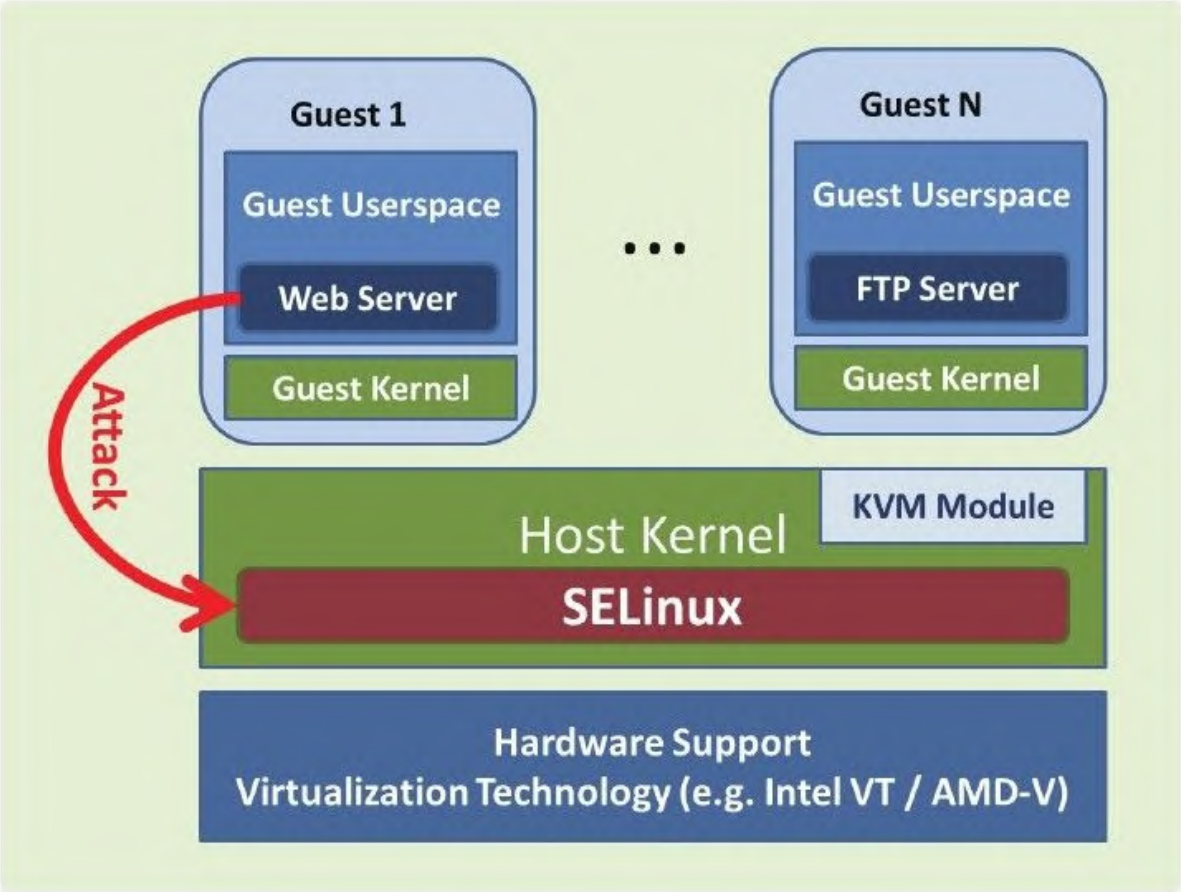


图　5-25　宿主机中的sVirt（SELinux）阻止来自客户机的攻击

sVirt框架允许将客户机及其资源都打上唯一的标签，如qemu-kvm进程有一个标签（tag1），其对应的客户机镜像也使用这个标签（tag1），而其他客户机的标签不能与这个标签（tag1）重复。一旦被打上标签，就可以很方便地应用规则来拒绝不同客户机之间的访问，SELinux可以控制仅允许标签相同的qemu-kvm进程可以访问某个客户机镜像。

3. SELinux和sVirt的配置和操作示例

由于Redhat公司是SELinux的主要开发者之一，更是sVirt最主要的开发者和支持者，因此SELinux和sVirt在Redhat公司支持的系统中使用得最为广泛，RHEL、Fedora的较新的版本都有对sVirt的支持。因为sVirt是以SELinux为基础，并通过SELinux来真正实现访问控制的，所以本节以RHEL 6.3系统为例介绍sVirt的配置和使用也完全和SELinux相关。

（1）SELinux相关的内核配置

查看RHEL 6.3系统内核配中SELinux相关的内容，命令行如下：

```
[root@rhel6u3-ga ~]# grep -i selinux /boot/config-2.6.32-279.el6.x86_64
CONFIG_SECURITY_SELINUX=y
CONFIG_SECURITY_SELINUX_BOOTPARAM=y
CONFIG_SECURITY_SELINUX_BOOTPARAM_VALUE=1
CONFIG_SECURITY_SELINUX_DISABLE=y
CONFIG_SECURITY_SELINUX_DEVELOP=y
CONFIG_SECURITY_SELINUX_AVC_STATS=y
CONFIG_SECURITY_SELINUX_CHECKREQPROT_VALUE=1
# CONFIG_SECURITY_SELINUX_POLICYDB_VERSION_MAX is not set
```

这里的"CONFIG_SECURITY_SELINUX_BOOTPARAM_VALUE=1"表示在内核启动时默认开启SELinux，也可以在Grub引导程序的内核启动行中添加"selinux="参数来修改内核启动时的SELinux状态："selinux=0"参数表示在内核启动时关闭SELinux，"selinux=1"参数表示在内核启动时开启SELinux。

（2）SELinux和sVirt相关的软件包

RHEL 6.3中用rpm命令查询与SELinux、sVirt相关的主要软件包，命令行如下：

```
[root@rhel6u3-ga ~]# rpm -qa | grep selinux
libselinux-devel-2.0.94-5.3.el6.x86_64
libselinux-python-2.0.94-5.3.el6.x86_64
libselinux-utils-2.0.94-5.3.el6.x86_64
libselinux-2.0.94-5.3.el6.x86_64
selinux-policy-targeted-3.7.19-154.el6.noarch
selinux-policy-3.7.19-154.el6.noarch
[root@rhel6u3-ga ~]# rpm -q policycoreutils
policycoreutils-2.0.83-19.24.el6.x86_64
```

其中，最重要的就是libselinux、selinux-policy、selinux-policy-targeted这3个RPM包，它们提供了SELinux在用户空间的库文件和安全策略配置文件。而libselinux-utils RPM包则提供了设置和管理SELinux的一些工具，如getenforce、setenforce等命令行工具；policycoreutils RPM包提供了一些查询和设置SELinux策略的工具，如setfiles、fixfiles、sestatus、setsebool等命令行工具。另外，因为RHEL中SELinux和sVirt的安全策略是根据libvirt来配置的，所以一般也需要安装libvirt软件包，这样才能更好的发挥sVirt对虚拟化安全的保护作用。

（3）SELinux和sVirt的配置文件

SELinux和sVirt的配置文件一般都在/etc/selinux目录之中，如下：

```
[root@rhel6u3-ga ~]# ls /etc/selinux/
config  restorecond.conf  restorecond_user.conf  semanage.conf  targeted
[root@rhel6u3-ga ~]# ls -l /etc/selinux/config
-rw-r--r--. 1 root root 457 Nov 19 11:14 /etc/selinux/config
```

其中最重要的配置文件是"/etc/selinux/config"文件，在该配置文件中修改设置之后需要重启系统才能生效。该配置文件的一个示例如下：

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of these two values:
#     targeted - Targeted processes are protected,
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

其中"SELINUX="项是用于设置SELinux的状态的，有三个值可选："enforcing"是生效状态，它会禁止违反规则策略的行为；"permissive"是较为宽松的状态，它会在发现违反SELinux策略的行为时发出警告（而不是阻止该行为），管理员在可以看到警告后可以考虑是否修改该策略来满足目前的安全需求；"disabled"是关闭状态，任何SELinux策略都不会生效。

"SELINUXTYPE="项是用于设置SELinux的策略，有两个值可选：一个是目标进程保护策略（targeted），另一个是多级安全保护策略（mls）。目标进程保护策略仅针对部分已经定义为目标的系统服务和进程执行SELinux策略，会执行"/etc/selinux/targeted"目录中各个具体策略；而多级安全保护策略是严格分层级定义的安全策略。一般来说，选择其默认的"targeted"目标进程保护策略即可。

SELinux的状态和策略的类型决定了SELinux工作的行为和方式，而具体策略决定具体的进程去访问文件时的安全细节。在目标进程工作模式（targeted）下，具体的策略配置在"/etc/selinux/targeted"目录中详细定义，执行如下命令行可查找和sVirt直接相关的配置文件：

```
[root@rhel6u3-ga targeted]# pwd
/etc/selinux/targeted
[root@rhel6u3-ga targeted]# grep svirt * -r
contexts/virtual_domain_context:system_u:system_r:svirt_t:s0
contexts/virtual_image_context:system_u:object_r:svirt_image_t:s0
contexts/customizable_types:svirt_image_t
Binary file modules/active/policy.kern matches
Binary file policy/policy.24 matches
```

"virtual_domain_context"文件配置了客户机的标签，而"virtual_image_context"文件配置了客户机镜像的标签，"customizable_types"中增加了"svirt_image_t"这个自定义类型。

（4）SELinux相关的命令行工具

在RHEL 6.3中，命令行查询和管理SELinux的工具主要是由"libselinux-utils"和"policycoreutils"这两个RPM包提供的。其中部分命令行工具的使用示例如下：

```
[root@rhel6u3-ga ~]# setenforce 1
[root@rhel6u3-ga ~]# getenforce
Enforcing
[root@rhel6u3-ga ~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
[root@rhel6u3-ga ~]# getsebool -a | grep httpd
allow_httpd_anon_write --> off
allow_httpd_mod_auth_ntlm_winbind --> off
allow_httpd_mod_auth_pam --> off
httpd_enable_homedirs --> off
<!-- 省略其他输出信息 -->
[root@rhel6u3-ga ~]# setsebool httpd_enable_homedirs on
[root@rhel6u3-ga ~]# ll -Z
drwxr-xr-x. root root unconfined_u:object_r:admin_home_t:s0 html
[root@rhel6u3-ga ~]# chcon -R -t httpd_sys_content_t /root/html
[root@rhel6u3-ga ~]# ll -Z
drwxr-xr-x. root root unconfined_u:object_r:httpd_sys_content_t:s0 html
```

下面对这几个命令进行简单介绍。

❑setenforce：修改SELinux的运行模式，其语法为setenforce[Enforcing|Permissive|1|0]，设置为1或Enforcing就是

让其处于生效状态，设置为0或Permissive是让其处于宽松状态（不阻止违反SELinux策略的行为，只是给一个警告提示）。

❑getenforce：查询获得SELinux当前的状态，可能会是生效（Enforcing）、宽松（Permissive）和关闭（Disabled）三个状态。

❑sestatus：获取运行SELinux系统的状态，通过此命令获取的SELinux信息更加详细。

❑getsebool：获取当前SELinux策略中各个配置项的布尔值，每个布尔值有开启（on）和关闭（off）两个状态。

❑setsebool：设置SELinux策略中配置项的布尔值。

❑chcon：改变文件或目录的SELinux安全上下文，"-t[type]"参数是设置目标安全上下文的类型（TYPE），"-R"参数表示递归地更改目录中所有子目录和文件的安全上下文。另外，"-u[user]"参数更改安全上下文中的用户（User），"-r[role]"参数更改安全上下文的角色（Role）。

（5）查看SELinux的安全上下文

在SELinux启动后，所有文件和目录都有各自的安全上下文，进程的安全上下文是域（domain）。关于安全上下文，一般遵守如下几个规则。

1）系统根据Linux-PAM（可插拔认证模块，Pluggable Authentication Modules）子系统中的pam_selinux.so模块来设定登录者运行程序的安全上下文。

2）RPM包安装时会根据RPM包内现有的记录来生成安全上下文。

3）手动创建的一个文件或者目录，会根据安全策略中的规则来设置其安全上下文。

4）如果复制文件或目录（用cp命令），会重新生成安全上下文。

5）如果移动文件或目录（用mv命令），其安全上下文保持不变。

安全上下文由用户（User）ID、角色（Role）、类型（Type）三部分组成，这里的用户和角色和普通系统的概念中的用户名和用户分组是没有直接关系的。

❑用户ID。与Linux系统中的UID类似，提供身份识别的功能，例如：user_u表示普通用户；system_u表示开机过程中系统进程的预设值；root表示超级用户root登录后的预设；unconfined_u为不限制的用户。在默认的目标（targeted）工作模式下，用户ID的设置并不重要。

❑角色。普通文件和目录的角色通常是object_r；用户可以具有多个角色，但是在同一时间内只能使用一角色；用户的角色，类似于普

通系统中的GID（用户组ID），不同的角色具备不同的权限。在默认的目标工作模式下，用户角色的设置也并不重要。

❑类型。将主体（程序）与客体（程序访问的文件）划分为不同的组，一个组内的各个主体和系统中的客体都定义了一个类型。类型是安全上下文中最重要的值，一般来说，一个主体程序能不能读取到这个文件资源，与类型这一栏的值有关。类型有时会有两个名称：在文件资源（Object）上面称为类型(Type)；而在主体程序（Subject）上称为域（Domain）。在域与类型匹配时，一个程序才能正常访问一个文件资源。

一般来说，有三种安全上下文：1）账号的安全上下文；2）进程的安全上下文；3）文件或目录的安全上下文。可以分别用"id-Z"、"ps-Z"、"ls-Z"等命令进行查看，示例如下：

```
[root@rhel6u3-ga ~]# id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[root@rhel6u3-ga ~]# ps -eZ | grep httpd
unconfined_u:system_r:httpd_t:s0 10999 ?        00:00:00 httpd
unconfined_u:system_r:httpd_t:s0 11002 ?        00:00:00 httpd
[root@rhel6u3-ga ~]# ls -Z /var/www/
drwxr-xr-x. root root system_u:object_r:httpd_sys_script_exec_t:s0 cgi-bin
drwxr-xr-x. root root system_u:object_r:httpd_sys_content_t:s0 error
drwxr-xr-x. root root system_u:object_r:httpd_sys_content_t:s0 html
drwxr-xr-x. root root system_u:object_r:httpd_sys_content_t:s0 icons
```

另外，chcon命令行工具可以改变文件或目录的SELinux安全上下文内容，对该工具的介绍和使用示例已经在本节前面部分提及了。

（6）sVirt提供的与虚拟化相关的标签

sVirt使用基于进程的机制和约束为虚拟化客户机提供了一个额外的安全保护层。在一般情况下，只要不违反约束的策略规则，用户是不会感知到sVirt在后台运行的。sVirt和SELinux将客户机用到的资源打上标签，也对相应的qemu-kvm进程打上标签，然后保证只允许具有与被访问资源相对应的类型（type）和相同分类标签的qemu-kvm进程可以访问某个磁盘镜像文件或其他存储资源。如果有违反策略规则的进程试图非法访问资源，SELinux会直接拒绝它的访问操作，返回一个权限不足的错误提示（通常为"Permission denied"）。

sVirt将SELinux与QEMU/KVM虚拟化进行了结合。sVirt一般的工作方式是：在RHEL系统中使用libvirt的守护进程（libvirtd）在后台管理客户机，在启动客户机之前，libvirt动态选择一个带有两个分类标志的随机的MCS标签（如s0:c1,c2），将该客户机使用到的所有存储资源（包括磁盘镜像、光盘等）都打上相应的标签（svirt_image_t:s0:c1,c2），然后用该相应的标签（svirt_t:c0:c1,c2）执行qemu-kvm进程，从而启动了客户机。

在客户机启动之前，查看一些关键的程序和磁盘镜像的安全上下文，示例如下：

```
# 依次查看 libvirtd、virsh、qemu-kvm 的安全上下文
[root@rhel6u3-ga ~]# ls -Z /usr/sbin/libvirtd
-rwxr-xr-x. root root system_u:object_r:virtd_exec_t:s0 /usr/sbin/libvirtd
[root@rhel6u3-ga ~]# ls -Z /usr/bin/virsh
-rwxr-xr-x. root root system_u:object_r:xm_exec_t:s0   /usr/bin/virsh
[root@rhel6u3-ga ~]# ls -Z /usr/libexec/qemu-kvm
-rwxr-xr-x. root root system_u:object_r:qemu_exec_t:s0 /usr/libexec/qemu-kvm

# 查看默认存放客户机镜像的目录中镜像文件的安全上下文
[root@rhel6u3-ga ~]# ls -Z /var/lib/libvirt/images/
-rw-r--r--. qemu qemu system_u:object_r:virt_content_t:s0 RHEL6.3.iso
-rw-------. root root system_u:object_r:virt_image_t:s0 rhel6u3.img
-rw-------. root root system_u:object_r:virt_image_t:s0 rhel6u3-new.img
```

然后分别使用上面看到的rhel6u3.img和rhel6u3-new.img两个磁盘镜像，用libvirt的命令行或者virt-manager图形界面工具（将会在第6章中将详细介绍libvirt、virt-manager等工具）启动两个客户机。在客户机启动后，查看磁盘镜像和qemu-kvm进程的安全上下文，命令行如下：

```
[root@rhel6u3-ga ~]# ls -Z /var/lib/libvirt/images/
-rw-r--r--. qemu qemu system_u:object_r:virt_content_t:s0 RHEL6.3.iso
-rw-------. qemu qemu system_u:object_r:svirt_image_t:s0:c259,c710 rhel6u3.img
-rw-------. qemu qemu
system_u:object_r:svirt_image_t:s0:c782,c824 rhel6u3-new.img
[root@rhel6u3-ga ~]# ps -eZ | grep qemu
system_u:system_r:svirt_t:s0:c259,c710 23271 ? 00:00:18 qemu-kvm
system_u:system_r:svirt_t:s0:c782,c824 23364 ? 00:00:06 qemu-kvm
```

由上面的输出信息可知，两个磁盘镜像已经被动态地标记上了不同的标签，相应的两个qemu-kvm进程也被标记上了对应的标签。其中，PID为23271的qemu-kvm进程对应的是rhel6u3.img这个磁盘镜像，由于SELinux和sVirt的配合使用，即使23271 qemu-kvm进程对应的客户机被黑客入侵，其qemu-kvm进程（PID 23172）也无法访问非它自己

所有的rhel6u3-new.img这个镜像文件，也就不会攻击到其他客户机（当然，通过网络发起的攻击除外）。

（7）sVirt根据标签阻止不安全访问的示例

前面已经提及，如果进程与资源的类型和标签不匹配，那么进程对该资源的访问将会被拒绝。真的这么有效吗？下面，笔者通过示例来证明它的有效性：模拟一些不安全的访问，然后查看sVirt和SELinux是否将其阻止。

首先，将bash这个Shell程序复制一份，并将其重命名为svirt-bash，以"svirt_t"类型和"s0:c1,c2"标签执行svirt-bash程序，进入这个Shell环境，用"id-Z"命令查看当前用户的安全上下文。然后创建/tmp/svirt-test.txt这个文件，并写入一些文字，可以看到该文件的安全上下文中类型和标签分别是：svirt_tmp_t和s0:c1,c2。该过程的命令行操作如下（包括部分命令行的注释）：

```
# 复制一个 Shell，重命名为 /bin/svirt-bash
[root@rhel6u3-ga ~]# cp /bin/bash /bin/svirt-bash

# 默认策略规定，使用 svirt_t 类型的入口程序必须标记为“qemu_exec_t”

# 类型，与 /usr/libexec/qemu-kvm 文件的安全上下文类似

[root@rhel6u3-ga ~]# chcon -t qemu_exec_t /bin/svirt-bash

# runcon 命令是以某个特定的 SELinux 安全上下文来执行某个命令
[root@rhel6u3-ga ~]# runcon -t svirt_t -l s0:c1,c2 /bin/svirt-bash
svirt-bash: /root/.bashrc: Permission denied
# 由于类型改变后，无法读取 .bashrc 配置文件，对本次实验无影响，忽略该错误提示

# 已经进入到使用 svirt_t:s0:c1,c2 这样的安全上下文的 Shell 环境中了

svirt-bash-4.1# id -Z
unconfined_u:unconfined_r:svirt_t:s0:c1,c2

# 可能会遇到“svirt: child setpgid (6962 to 6962): Permission denied”

# 这样的错误提示，Shell 试图为每个命令都设置进程的组 ID（setpgid），对本次实验
# 无影响，在本次实验中忽略 setpid 来带的错误提示

svirt-bash-4.1# echo "just for testing" >> /tmp/svirt-test.txt

svirt-bash-4.1# ls -Z /tmp/svirt-test.txt

-rw-r--r--. root
root unconfined_u:object_r:svirt_tmp_t:s0:c1,c2 /tmp/svirt-test.txt

svirt-bash-4.1# cat /tmp/svirt-test.txt
just for testing
```

使用"svirt_t"类型和"s0:c1,c3"（与前面s0:c1,c2不同）标签启动svirt-bash这个Shell程序，进到该Shell环境，用"id-Z"命令查看当前用户的安全上下文，用"ls-Z"查看上一步创建的临时测试文件的安全上下文，试着用"cat"命令去读取该测试文件（svirt-test.txt）。该过程的命令行操作如下：

```
[root@rhel6u3-ga ~]# runcon -t svirt_t -l s0:c1,c3 /bin/svirt-bash
svirt-bash: /root/.bashrc: Permission denied

svirt-bash-4.1# id -Z
unconfined_u:unconfined_r:svirt_t:s0:c1,c3

svirt-bash-4.1# ls -Z /tmp/svirt-test.txt
-rw-r--r--. root
root unconfined_u:object_r:svirt_tmp_t:s0:c1,c2 /tmp/svirt-test.txt

# 在 s0:c1,c3 的标签下，无法读取 s0:c1,c2 标签的测试文件
svirt-bash-4.1# cat /tmp/svirt-test.txt
cat: /tmp/svirt-test.txt: Permission denied
```

可知，在标签为"s0:c1,c3"的Shell环境中，不能访问具有"s0:c1,c2"标签的一个临时测试文件，sVirt拒绝本次非法访问，故sVirt基于标签的安全访问控制策略是生效的。

在使用Redhat相关系统（如RHEL、Fedora等）时，如果对KVM虚拟化安全性要求较高，可以选择安装SELinux和sVirt相关的软件包，使用sVirt来加强虚拟化中的安全性。不过，由于SELinux配置还是比较复杂，其默认策略对其他各种服务的资源访问限制还是较多的（为了安全性），也受到一些系统管理员的排斥，一些系统管理员经常会关闭SELinux。总之，一切从实际出发，根据实际项目中对安全性的需求对SELinux和sVirt进行相应的配置。

5.8.4 可信任启动——Tboot

1.TXT和Tboot简介

现在越来越多的公司开始使用虚拟化技术来提高物理资源的利用率和系统可管理性，客户机都运行在Hypervisor之上，所以通常会把Hypervisor作为安全可信的基础，这就要求Hypervisor本身是可信任的。Hhypervisor的可信启动是一个可信Hypervisor的基础，也是整个虚拟化环境的安全可信的基础。由于Hypervisor是和系统物理硬件接触最紧密的软件层，它直接依赖于系统的固件（如BIOS）和硬件（如CPU、内存），所以可以利用一些硬件技术来保证hypervisor的可信启动。

（1）TXT简介

TXT是Intel公司开发的Trusted Execution Technology（可信执行技术）的简称，是可信计算（Trusted Computing）在Intel平台上的实现，它是在PC或服务器系统启动时对系统的关键部件进行验证的硬件解决方案。TXT的主要目标在于：证明一个平台和运行于它之上的操作系统的可靠性；确保一个可靠的操作系统是启动在可信任的环境中的，进而成为一个可信的操作系统；提供给可信任操作系统比被证实的系统更多的安全特性。

TXT提供了动态可信根（dynamic root of trust）的机制以增强平台的安全性，为建立可信计算环境和可信链提供必要的支持。动态可信根，依靠一些已知正确的序列来检查系统启动时的配置和行为的

一致性。使用这样的基准测试，系统可以很快地评估当前这次启动过程中是否有改变或篡改已设置的启动环境的意图。

恶意软件是对IT基础架构的一个重要威胁，而且也是日益严重的威胁。尽管恶意软件的原理不尽相同，但是它们都试图去污染系统、扰乱正常商业活动、盗取数据、夺取平台的控制权，等等。由于现在的公司都越来越多地使用虚拟的、共享的、多租户的IT基础架构模型，因此传统的网络基础架构就看起来更加的分散，暴露了更多的安全漏洞。同样，传统的安全技术（反病毒和反蠕虫软件）采用的主要方式是对已知的病毒或蠕虫的攻击行为来进行防护和阻止，今天随着各种安全公司的规模扩大和复杂性的增强，传统安全技术也只能是部分的有效，效果并不尽如人意。Intel的TXT技术提供了一种不同的方法——使用已知是正确的数据和流程来检查系统或普通软件的启动。

Intel的TXT技术主要通过三个方面来提供安全和信任：首先将数字指纹软件保存在一个称为“可信用平台模块”（Trusted Platform Module，TMP）的受保护区域内。每次软件启动时，都会检测并确保数字指纹吻合，从而判断系统是否存在风险；其次，它能阻止其他应用程序、操作系统或硬件修改某个应用程序的专用内存区；再次，如果某个应用程序崩溃，TXT将清除它在内存中的数据和芯片缓存区，避免攻击软件嗅探其残余数据。

基于Intel TXT技术建立的可信平台，主要由三个部分组成：安全模式指令扩展（Safe Mode Externtions，SMX）、认证代码模块（Authenticated Code Module，AC模块）、度量过的安全启动环境（Measured Launched Environment，MLE）。安全模式指令扩展，是对现有指令集进行扩展，引入了一些与安全技术密切相关的指令，通过执行这些指令，系统能够进入和退出安全隔离环境。认证代码模块，是由芯片厂商提供的，经过数字签名认证的，与芯片完全绑定的一段认证代码。当系统接入安全隔离环境时，最先被执行的就是这段认证代码，它被认为是可信的，它的作用是检测后续安全启动环境的可信性。安全启动环境是用于检测内核（或Hypervisor）以及启动内核（或Hypervisor）的软件，它不仅需要对即将启动的内核（或Hypervisor）进行检测，同时还需要保证这种检测过程是受保护的，以及它自身所运行的代码不会被篡改。

（2）Tboot简介

Tboot即“可信启动”（Trusted Boot），是使用TXT技术的在内核或Hypervisor启动之前的一个软件模块，用于度量和验证操作系统或Hypervisor的启动过程。Tboot利用TXT技术能够对“动态可信根”在进入安全隔离环境时进行一系列可信检测，与保存在可信平台模块（TPM）的存储空间中的目标检测值进行比较。可信平台模块（通常是一个硬件芯片）利用自身的安全性使这些值不会被恶意代码获取和篡

改。Tboot能够利用可信平台模块中的平台配置寄存器（PCR）来保护和控制启动顺序，使非法应用不能跳过这些检测步骤直接进入到可信链的后续环节。同时，Tboot还能在系统关闭或休眠前清除敏感信息以防泄漏，使可信环境在安全可控制的过程中退出。除此之外，Tboot还利用支持TXT技术的芯片中自带的DMA受保护区域（DMA Protected Range，PTR）和VT-d的受保护内存区域（Protected Memory Region，PMR）来防止恶意代码通过DMA跳过检测过程而直接访问内存中的敏感区域。

关于TXT和Tboot的实现细节，这里并不做详细的介绍，读者可以查看本章参考阅读中提到的Intel关于TXT技术规范的手册。在使用Tboot的情况下，一个可信的内核或Hypervisor的启动过程大致如图5-26所示。

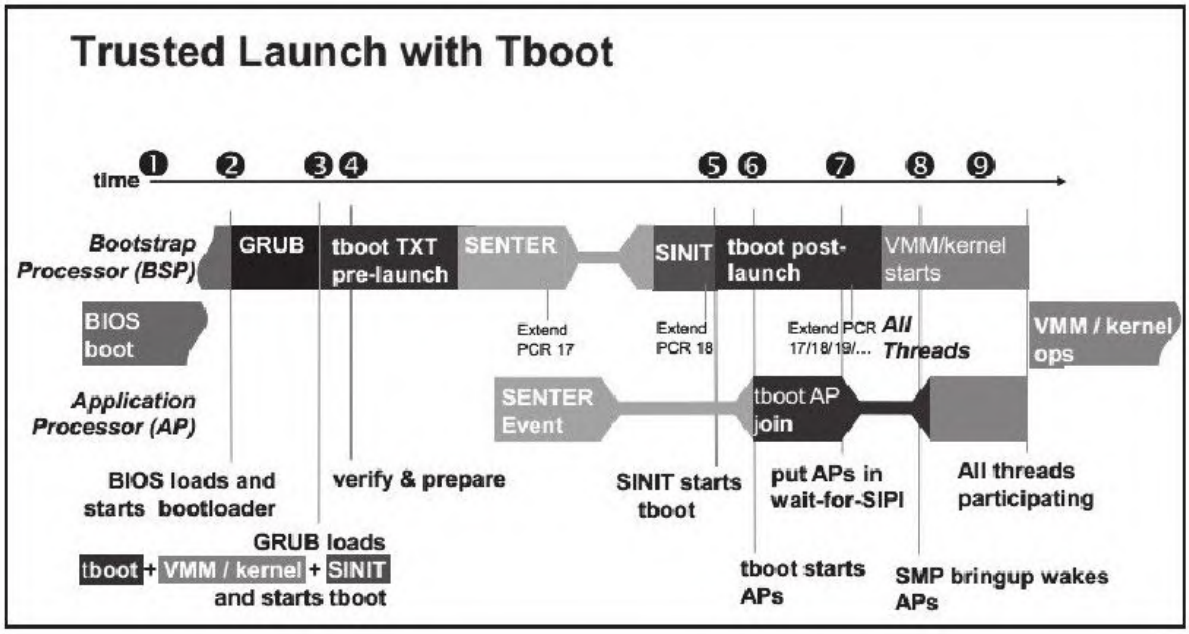


图　5-26　在Tboot支持下的可信任启动过程

Tboot目前也是由Intel的开源软件工程师发起的一个开源的项目，其项目主页是：http://sourceforge.net/projects/tboot/和http://tboot.sourceforge.net/

目前的一些开源操作系统内核和Hypervisor都对Tboot有较好支持，Xen从3.4版本开始支持Tboot，而Linux内核从2.6.35版本开始支持Tboot（在3.0版本中加入了一部分和VT-d相关的修复）。一些主流的Linux发行版（如RHEL、Fedora、Ubuntu等）也支持Tboot，并提供了Tboot的软件安装包。

2.使用Tboot的示例

在介绍了Intel TXT和Tboot技术之后，本节将介绍如何配置系统的软硬件来让Tboot真正地工作起来。关于配置和使用Tboot，笔者根据在一台使用Intel Westmere-EP处理器的Xeon(R)X5670服务器的实际操作来介绍如下几个操作步骤。

（1）硬件配置及BIOS设置

并非Intel平台的任何一个机器都支持TXT，在Intel中，有vPro标识的桌面级硬件平台一般都支持TXT，另外也有部分服务器平台支持TXT技术。因为Intel TXT技术依赖于Intel VT和VT-d技术，所以在

BIOS中不仅需要打开TXT技术的支持，还需要打开Intel VT和VT-d的支持。在BIOS中设置如下项目（不同BIOS在选项命令和设置位置上有些差别）。

打开TXT技术的支持，BIOS选项位于：Advanced→Processor Configuration→Intel(R)TXT或Intel(R)Trusted Excution Technology，需要将其设置为"[Enabled]"状态。

打开Intel VT和VT-d技术的支持，BIOS选项位于：Advanced→Processor Configuration→Intel(R)Virtualization Technology和Intel(R)VT-for Direct I/O，将这两者都设置为"[Enabled]"状态。

可信平台模块（TPM）需要在主板上由一个TPM模块芯片支持，打开TPM支持的BIOS选项位于：Security→TPM Administrative Control。有几个选项，分别是：No Operation、Turn ON、Turn OFF、Clear Ownership，应该选择"Turn ON"（打开）。重启系统后，可以看到BIOS设置中的：Security→TPM State的值为"Enabled＆Activated"。当然，如果TPM状态一开始就是"Enabled＆Activated"，那么说明TPM处于打开状态，不需要重复打开了。

（2）编译支持Tboot的Linux内核

要支持Tboot，需要在宿主机的Linux内核中配置TXT和TCG[15]（Trusted Computing Group）相关的配置。由于Tboot还需要Intel VT和VT-d技术的支持，在内核中也要有支持VT（配置关键字为VT）和VT-d（配置文件中关键字为IOMMU）的配置，查看这些配置命令行如下：

```
[root@jay-linux kvm_demo]# grep TXT /boot/config-3.5.0
CONFIG_HAVE_INTEL_TXT=y
CONFIG_INTEL_TXT=y
[root@jay-linux kvm_demo]# grep TCG /boot/config-3.5.0
CONFIG_TCG_TPM=m
CONFIG_TCG_TIS=m
CONFIG_TCG_NSC=m
CONFIG_TCG_ATMEL=m
CONFIG_TCG_INFINEON=m
[root@jay-linux kvm_demo]# grep -i IOMMU /boot/config-3.5.0
CONFIG_GART_IOMMU=y
CONFIG_CALGARY_IOMMU=y
# CONFIG_CALGARY_IOMMU_ENABLED_BY_DEFAULT is not set
CONFIG_IOMMU_HELPER=y
CONFIG_IOMMU_API=y
CONFIG_IOMMU_SUPPORT=y
# CONFIG_AMD_IOMMU is not set
CONFIG_INTEL_IOMMU=y
CONFIG_INTEL_IOMMU_DEFAULT_ON=y
[root@jay-linux ~]# grep -i virtualization /boot/config-3.5.0
CONFIG_VIRTUALIZATION=y
[root@jay-linux ~]# grep -i config_vt /boot/config-3.5.0
CONFIG_VT=y
```

```
CONFIG_VT_CONSOLE=y
CONFIG_VT_CONSOLE_SLEEP=y
CONFIG_VT_HW_CONSOLE_BINDING=y
```

如果缺少这些配置，则需要添加上这些配置，然后重新编译内核。这里不再重复编译内核的步骤，在3.3节“编译和安装KVM”中对内核的配置和编译已经做过详细的介绍。

（3）编译和安装Tboot

在一些主流的Linux发行版中，Tboot已经作为一个单独的软件包发布了，这些发行版包括：Fedora 14、RHEL 6.1、SELS 11 SP2、Ubuntu 12.04等，以及比它们更新的版本。在已经有Tboot支持的系统上，只需要安装tboot这个软件包即可，如在RHEL 6.3中可以用"yum install tboot"来安装Tboot，一般也会同时安装trousers软件包（可提供tcsd这个用于管理可信计算资源的守护程序）。

如果没有现成的软件包可用，可到sourceforge上tboot的项目页面下载其源代码的tar.gz包进行编译。Tboot项目主页是：http://sourceforge.net/projects/tboot/和http://tboot.sourceforge.net/，同时也可以在线查看该项目的修改记录和代码，见http://tboot.hg.sourceforge.net/hgweb/tboot/tboot。

tboot的编译和运行会依赖openssl-devel、zlib-devel、trousers-devel、trousers等软件包，在编译tboot之前需要将它们安装好（如果有的依赖软件包不存在，编译时会有报错提示，安装上相

应的软件包即可）。获取最新的tboot开发源代码仓库（目前是使用Mercurial[16] 工具管理的），并进行编译，然后查看编译后生成的所需要的二进制文件，命令行操作如下：

```
[root@jay-linux kvm_demo]# hg clone
http://tboot.hg.sourceforge.net:8000/hgroot/tboot/tboot  tboot

requesting all changes
adding changesets
adding manifests
adding file changes
added 318 changesets with 1194 changes to 197 files
updating to branch default
169 files updated, 0 files merged, 0 files removed, 0 files unresolved

[root@jay-linux kvm_demo]# cd tboot
[root@jay-linux tboot]# make

<!-- 此处省略编译过程的输出信息 -->

[root@jay-linux tboot]# make install

<!-- 此处省略安装过程的输出信息 -->

[root@jay-linux tboot]#ls -l/boot/tboot.gz
-rw-r--r--. 1 root root 70834 Nov 24 23:39 /boot/tboot.gz
[root@jay-linux tboot]#ls -l /usr/sbin/txt-stat
-rwxr-xr-x. 1 root root 18424 Nov 24 23:40 /usr/sbin/txt-stat
```

由上面的输出信息可知，通过源代码编译和安装Tboot及其用户空间管理工具的过程是比较简单的，默认会将Tboot的内核文件tboot.gz安装到/boot/目录下，将Tboot相关的用户空间管理工具安装到/usr/sbin/目录下。

（4）修改GRUB，让tboot.gz先于Linux内核启动

在修改GRUB之前，还需要获取当前系统的SINIT AC模块（它是芯片厂商提供的经过数字签名的一段认证代码），并将其配置到GRUB中使其对系统后续启动过程进行可信的认证。可以从硬件供应商那里去索取SINIT AC模块，也可以到Intel公司官方网站中关于TXT技术的网页[17] 去查看哪些平台支持TXT和下载对应的SINIT二进制代码模块。这里已经将SINIT模块下载放到/boot/WSM_SINIT_100407_rel.bin位置。

在使用Tboot之时，在GRUB等引导程序中，需要将tboot.gz文件设置为内核（kernel）以便最先启动，而原本的Linux内核（或Xen等Hypervisor）、初始化内存镜像（initramfs、initrd）和SINIT模块都设置为模块启动。一个配置有Tboot的GRUB入口条目示例如下：

```
title KVM host with Tboot
        root (hd0,0)
        kernel /boot/tboot.gz logging=vga,serial,memory
        module /boot/vmlinuz-3.5.0 ro
root=UUID=1a65b4bb-cd9b-4bbf-97ff-7e1f7698d3db intel_iommu=on
        module /boot/initramfs-3.5.0.img
        module /boot/WSM_SINIT_100407_rel.bin
```

其中，tboot.gz行的"logging=vga,serial,memory"配置表示将Tboot内核启动的信息同时记录在显示器、串口和内存中；WSM_SINIT_100407_rel.bin就是用于支持笔者使用的Westmere-EP平台的SINIT AC模块。

（5）获取可信平台模块（TPM）的所有权

通过上一步修改好的Tboot启动的GRUB入口条目重新启动系统，在系统中获取可信平台模块（TPM）的所有权，步骤如下：

1）加载tpm_tis内核模块。

tpm_tis模块是可信平台模块（TPM）硬件的驱动程序，加载和检查tpm_tis的命令行操作如下：

```
[root@jay-linux ~]#modprobe tpm_tis
[root@jay-linux ~]#lsmod | grep tpm
tpm_tis                 9580  0
tpm                   17031  1 tpm_tis
tpm_bios              10339  1 tpm
```

由上面的输出信息可知，tpm_tis模块依赖于tpm模块，而tpm模块又依赖于tpm_bios模块。如果加载tpm_tis时不成功，可以添加一些参数，通过用如下命令来加载tpm_tis模块：

```
[root@jay-linux ~]#modprobe tpm_tis force=1 interrupts=0
```

2）启动tcsd守护进程。

tcsd是trousers软件包中的一个命令行工具，是一个管理可信平台模块（TPM）资源的守护程序，它也负责处理来自本地和远程的可信服务提供者（TSP）的请求。tcsd应该是唯一的到达TPM设备驱动程序

的一个用户空间守护进程。启动和查询tcsd守护程序的命令行操作如下：

```
[root@jay-linux ~]#tcsd
[root@jay-linux ~]#ps -ef | grep tcsd | grep -v grep
tss       8440     1  0 14:24 ?        00:00:00 tcsd
```

3）设置系统的可信平台模块（TPM）的所有者。

tpm_takeownership也是trousers软件包中的一个命令行工具，它通过TPM_TakeOwnership应用程序接口在当前系统的可信平台模块（TPM）上建立一个所有者（owner）。该命令在执行时会要求设置所有者的密码和存储根密钥（Storage Root Key，SRK）的密码和确认。

```
[root@jay-linux ~]#tpm_takeownership
Enter owner password:
Confirm password:
Enter SRK password:
Confirm password:
```

另外，可以添加-y参数将所有者的密码设置为20个“0”而不需要交互式的输出，也可以用-z参数将存储根密钥（SRK）的密码设置为20个“0”，示例如下：

```
[root@jay-linux ~]#tpm_takeownership -y -z
```

（6）启动系统，检查Tboot的启动情况

从前面修改的Grub的Tboot入口条目启动系统，在显示器或串口信息上都可以看到Tboot启动时的信息，示例如下：

```
TBOOT: ******************* TBOOT *******************
TBOOT:    2012-11-17 23:44 +0800 301:950fec11ef90
TBOOT: *********************************************
TBOOT: command line: logging=vga,serial,memory
TBOOT: BSP is cpu 0
TBOOT: original e820 map:
TBOOT:  0000000000000000 - 000000000009b000  (1)
TBOOT:  000000000009b000 - 00000000000a0000  (2)
<!-- 省略中间的其余部分输出 -->
TBOOT:  0000000100000000 - 0000000670000000  (1)
TBOOT: TPM is ready
TBOOT: TPM nv_locked: TRUE
<!-- 省略中间的其余部分输出 -->
TBOOT: platform id: 0x0
TBOOT:   1 ACM chipset id entries:
TBOOT:       vendor: 0x8086, device: 0xc000, flags: 0x1, revision:
0x7, extended: 0x0
```

```
TBOOT: SINIT matches platform
TBOOT: TXT.SINIT.BASE: 0x8f700000
TBOOT: TXT.SINIT.SIZE: 0x20000 (131072)
TBOOT: copied SINIT (size=8c40) to 0x8f700000
TBOOT: AC mod base alignment OK
TBOOT: AC mod size OK
<!-- 省略中间的其余部分输出 -->
TBOOT: cpu 18 waking up from TXT sleep
TBOOT: .MSR for SMM monitor control on cpu 18 is 0x0
TBOOT: verifying ILP's MSR_IA32_SMM_MONITOR_CTL with cpu 18
         : succeeded.
TBOOT: enabling SMIs on cpu 18
TBOOT:
TBOOT: VMXON done for cpu 18
TBOOT: launching mini-guest for cpu 18
<!-- 省略中间的其余部分输出 -->
TBOOT: verifying e820 table against SINIT MDRs: verification succeeded.
TBOOT: verifying tboot and its page table (800000 - 96fe5f) in e820 table
         (range from 0000000000800000 to 000000000096fe60 is in E820_RAM)
TBOOT: : succeeded.
<!-- 省略中间的其余部分输出 -->
TBOOT: assuming kernel is Linux format
TBOOT: Initrd from 0x6f958000 to 0x7ffff200
TBOOT: Kernel (protected mode) from 0x100000 to 0x444450
TBOOT: Kernel (real mode) from 0x90000 to 0x93c00
TBOOT: transfering control to kernel @0x100000...
<!-- 省略中间的其余部分输出 -->
TBOOT: cpu 2 waking up, SIPI vector=95000
TBOOT: cpu 34 waking up, SIPI vector=95000
TBOOT: cpu 4 waking up, SIPI vector=95000
```

在系统启动之后，通过dmesg命令可以找到Tboot正在使用的一些信息，如下：

```
[root@jay-linux ~]# dmesg | grep -i tboot
tboot: found shared page at phys addr 0x826000:
tboot: Forcing Intel-IOMMU to enabled
```

（7）使用txt-stat工具检查本次启动的可信状态

对于Tboot是否正常工作，本次操作系统启动是否可信，可以使用Tboot源码中自带的txt-stat命令行工具来查看TXT当前的工作状态。如果在txt-stat中看到如下的输出信息，则表明本次是一次安全可信的启动。

```
************************************************************
         TXT measured launch: TRUE

         secrets flag set: TRUE
************************************************************
```

使用txt-stat工具查看TXT工作状态，该工具的输出结果如下：

```
[root@jay-linux ~]# txt-stat
Intel(r) TXT Configuration Registers:
        STS: 0x00014081
            senter_done: TRUE
            sexit_done: FALSE
            mem_config_lock: FALSE
            private_open: TRUE
            locality_1_open: FALSE
            locality_2_open: TRUE
        ESTS: 0x00
            txt_reset: FALSE
        E2STS: 0x000000020000001e
            secrets: TRUE
        ERRORCODE: 0x00000000
        DIDVID: 0x0000003fc0008086
            vendor_id: 0x8086
            device_id: 0xc000
            revision_id: 0x3f
        FSBIF: 0x0000000000000000
        QPIIF: 0x000000008c482000
        SINIT.BASE: 0x8f700000
        SINIT.SIZE: 131072B (0x20000)
        HEAP.BASE: 0x8f720000
        HEAP.SIZE: 917504B (0xe0000)
        DPR: 0x000000008f800031
            lock: TRUE
            top: 0x8f800000
            size: 3MB (3145728B)
        PUBLIC.KEY:
            08 77 7b 21 ec 4d 7f ce f7 68 2a 26 96 bc 5f 42
            a9 96 45 a4 21 81 10 7f 87 70 c2 24 37 fd e0 2c

**************************************************************
         TXT measured launch: TRUE
         secrets flag set: TRUE
**************************************************************
TBOOT log:
         max_size=7fe8
         curr_pos=66b5
         buf:
TBOOT: ******************* TBOOT *******************
TBOOT:    2012-11-17 23:44 +0800 301:950fec11ef90
```

```
TBOOT: ********************************************
TBOOT: command line: logging=vga,serial,memory
TBOOT: BSP is cpu 0
TBOOT: original e820 map:
TBOOT:  0000000000000000 - 000000000009b000  (1)
TBOOT:  000000000009b000 - 00000000000a0000  (2)
TBOOT:  00000000000e0000 - 0000000000100000  (2)
<!-- 省略 txt-stat 命令的其余部分输出信息 -->
[root@jay-linux ~]# txt-stat  | grep -i "secrets flag"
         secrets flag set: TRUE
TBOOT: set TXT.CMD.SECRETS flag
[root@jay-linux ~]# txt-stat  | grep -i "measured launch"
         TXT measured launch: TRUE
TBOOT: measured launch succeeded
```

如果看到在Tboot的保护下正常启动操作系统，也查询到当前系统的TXT是正常工作的，那么就可以证明本次启动是一次可信任的启动。

5.8.5 其他安全策略

1.镜像文件加密

随着网络与计算机技术的发展，数据的一致性和数据的完整性在信息安全中变得越来越重要，对数据进行加密处理对数据的一致性和完整性都有较好的保障。有一种类型的攻击叫做“离线攻击”，如果攻击者在系统关机状态下可以物理接触到磁盘或其他存储介质，就属于“离线攻击”的一种表现形式。另外，在一个公司内部，不同职位的人有不同的职责和权限，系统处于启动状态时的使用者是工作人员A，而系统关机后，会存放在另外的位置（或不同部门），而工作人员

B可以获得该系统的物理硬件。如果没有保护措施，那么B就可以轻易地越权获得系统中的内容。如果有了良好的加密保护，就可以防止这样的攻击或内部数据泄露事件的发生。

在KVM虚拟化环境中，存放客户机镜像的存储设备（如磁盘、U盘等）可以对整个设备进行加密，如果其分区是LVM，也可以对某个分区进行加密。而对于客户机镜像文件本身，也可以进行加密的处理，如“4.4.3 QEMU支持的镜像文件格式”节中已经介绍过qcow2镜像文件格式支持加密，这里再简单介绍一下。

"qemu-img convert"命令在"-o encryption"参数的支持下，可以将未加密或已经密的镜像文件转化为加密的qcow2的文件格式。先创建一个8 GB大小的qcow2格式镜像文件，然后用"qemu-img convert"命令将其加密，命令行操作如下：

```
[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o size=8G  guest.qcow2
Formatting guest.qcow2', fmt=qcow2 size=8589934592
encryption=off cluster_size=65536

[root@jay-linux kvm_demo]# qemu-img convert -o encryption -O
qcow2 guest.qcow2 encrypted.qcow2
Disk image encrypted.qcow2' is encrypted.
password: ******  # 此处输入你需要设置的密码，然后按回车键确认
```

生成的encryped.qcow2文件就是已经加密的文件，查看其信息如下所示，可以看到"encrypted:yes"的标志。

```
[root@jay-linux kvm_demo]# qemu-img info encrypted.qcow2
Disk image 'encrypted.qcow2' is encrypted.
password:
image: encrypted.qcow2
file format: qcow2
virtual size: 8.0G (8589934592 bytes)
disk size: 8.0G
encrypted: yes
cluster_size: 65536
```

在使用加密的qcow2格式的镜像文件启动客户机时，客户机会先不启动而暂停，需要在QEMU monitor中输入"cont"或"c"命令以便继续执行，然后会出现输入已加密qcow2镜像文件的密码，只有密码正确才可以正常启动客户机。在QEMU monitor中的命令行示例如下：

```
(qemu) c
ide0-hd0 (encryped.qcow2) is encrypted.
Password: ******          # 此处输入先前设置过的密码
(qemu)
```

当然，在执行"qemu-img create"创建镜像文件时就可以将其创建为加密的qcow2文件格式，但是不能交互式地指定密码，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-img create -f qcow2 -o
backing_file=rhel6u3.img,encryption encrypted.qcow2
```

这样创建的qcow2文件处于加密状态，但是其密码为空，在使用过程中提示输入密码时，直接按回车键即可。对于在创建时已设置为加

密状态的qcow2文件，仍然需要用上面介绍过的"qemu-img convert"命令来转换一次，这样才能设置为自己所需的非空密码。

当使用qcow2镜像文件的加密功能时，可能会觉得每次都要输入密码，可能会给大规模地自动化部署KVM客户机带来一定的障碍。其实，输入密码验证的问题也是很容易解决的，例如Linux上的expect工具就很容易解决需要输入密码这样的交互式会话。在使用expect进行ssh远程登录过程中输入密码交互的一段Bash脚本如下：

```
SSH_Time_Out() {
    local host=192.168.111.111
    local time_out=250
    local command="ifconfig"
    local exp_cmd=$(cat << EOF

  spawn ssh root@$host $command
  set timeout $time_out
  expect {
      "*assword:" { send "123456\r"; exp_continue}
      "*(yes/no)?" { send "yes\r"; exp_continue }
      eof           { exit 0 }
  }
  EOF)
  ??? expect -c "$exp_cmd"
  ??? return $?
  }
```

上面的脚本示例实现了用ssh远程登录到一台主机上，并执行了一个ifconfig命令。其中远程主机的root用户的密码的输入是通过

expect的脚本来实现自动输入的，expect的超时时间在这里被设置为250秒。

2.虚拟网络的安全

在KVM宿主机中，为了网络安全的目的，可以使用Linux防火墙——iptables工具。使用iptables工具（为IPv4协议）或ip6tables（为IPv6协议）可以创建、维护和检查Linux内核中IP数据报的过滤规则。

而对于客户机的网络，QEMU/KVM提供了多种网络配置方式（见“4.5节网络配置”）。例如：使用NAT方式让客户机获取网络，就可以对外界隐藏客户机内部网络的细节，对客户机网络的安全起到了保护作用。不过，在默认情况下，NAT方式的网络让客户机可以访问外部网络，而外部网络不能直接访问客户机。如果客户机中的服务需要被外部网络直接访问，就需要在宿主机中配置好iptables的端口映射规则，通过宿主机的某个端口映射到客户机的一个对应端口，其具体配置的示例已经在“4.5.3使用NAT模式”小节中详细介绍了。

如果物理网卡设备比较充足，而且CPU、芯片组、主板等都支持设备的直接分配技术（如Intel VT-d技术），那么选择使用设备直接分配技术（VT-d）为每个客户机分配一个物理网卡也是一个非常不错的选择。因为在使用设备直接分配使用网卡时，网卡工作效率非常高，

而且各个客户机中的网卡是物理上完全隔离的，提高了客户间的隔离性和安全性，即使一个客户机中网络流量很大（导致了阻塞）也不会影响到其他客户机中网络的质量。关于网卡设备的直接分配，可以参考“5.2.3 VT-d操作示例”这一小节中的详细配置。

3.远程管理的安全

在KVM虚拟化环境中，可以通过VNC的方式远程访问客户机，那么为了虚拟化管理的安全性，可以为VNC连接设置密码，并且可以设置VNC连接的TLS、X.509等安全认证方式。关于VNC的配置和密码设置等相关内容，可以参考“4.6.2 VNC的使用”这一小节。

如果使用libvirt的应用程序接口来管理虚拟机，包括使用virsh、virt-manager、virt-viewer等工具，为了远程管理的安全性考虑，最好只允许管理工具使用SSH连接或者带有TLS加密验证的TCP套接字来连接到宿主机的libvirt。关于libvirt API的简介及其配置、连接的方式，可以阅读"6.1 libvirt"中的内容。

4.普通Linux系统的安全准则

KVM宿主机及运行在KVM上Linux客户机都是普通的Linux操作系统。普通Linux系统的一些安全策略和准则都可以用于KVM环境的安全性的提高。

美国国家安全局（National Security Agency，NSA）的一份公开文档[18] 中谈及了Redhat Linux系统的安全设置应遵循的几个通用原则，对于任何Linux系统（甚至Windows系统）的安全配置都有一定的借鉴意义。这里简要分享一下，仅供读者参考。

（1）对传送的数据尽可能进行加密处理

无论是在有线网络还是无线网络上传输的数据都很容易被监听，所以只要对这些数据的加密方案存在，都应该使用加密方案。即使是预期仅在局域网中传输的数据，也应该做加密处理。对于一些身份认证相关的数据（如密码）等的加密有其非常重要。RHEL 5.x或6.x版本的Linux系统组成的网络中，可以配置为它们之间传输的全部都经过加密处理。

（2）安装尽可能少的软件，从而使软件漏洞数量尽可能少

避免一个软件漏洞最简单的方法就是避免安装那个软件。在RHEL系统中，有RPM软件包管理工具可以比较方便地管理系统中已经安装的软件包。随意安装的软件可能会以多种方式暴露出系统的漏洞：1）包含有setuid程序的软件包可能为本地的攻击者提供一种特权扩大的潜在途径。2）包含有网络服务的软件包可能为基于网络的攻击者提供攻击的机会。3）包含有被本地用户有预期地执行的程序（如图形界面登录后必须执行的程序）的软件可能会给特洛伊木马者其他隐藏执行的

攻击代码提供机会。通常，一个系统上安装的软件包数量可以被大量删减，直到只包含环境或者操作所真正必需的软件。

（3）不同的网络服务尽量运行在不同的系统上

一个服务器（当然也可以是虚拟客户机）应该尽可能地专注只提供一个网络服务。如果这样做，即使一个攻击者成功地渗透了一个网络服务上的软件漏洞，也不会危害到其他服务的正常运行。

（4）配置一些安全工具去提高系统的鲁棒性

现存的几个工具可以有效地提高系统的抵抗能力和检测未知攻击的能力。这些工具可以利用较低的配置成本去提高系统面对攻击时的鲁棒性。有一些实际的工具就具有这样的能力，如基于内核的防火墙——iptables，基于内核的安全保护软件——SELinux，以及其他一些日志记录和进程审计的软件基础设施。

（5）使用尽可能少的权限

只授予用户账号和运行的软件最少的必需的权限。例如，仅给真正需要用sudo管理员权限的用户开放sudo的功能（在Ubuntu中默认开放过多的sudo权限可能并不太安全）。限制系统的登录，仅向需要管理系统的用户开放登录的权限。另外，还可以使用SELinux来限制一个

软件对系统上其他资源的访问权限，合适的SELinux策略可以让某个进程仅有权限做应该做的事情。

5.9 QEMU监控器

QEMU监控器（monitor）是QEMU与实现用户交互的一种控制台，一般用于为QEMU模拟器提供较为复杂的功能，包括为客户机添加和移除一些媒体镜像（如CD-ROM、磁盘镜像等），暂停和继续客户机的运行，快照的建立和删除，从磁盘文件中保存和恢复客户机状态，客户机动态迁移，查询客户机当前各种状态参数……。在前面几章中，根据实际应用的具体场景已经多次提及一部分QEMU monitor中的命令了，本节将对qemu-kvm monitor的使用和其中的常见命令进行介绍和总结。

5.9.1 QEMU monitor的切换和配置

要使用QEMU monitor，首先需要切换到monitor窗口中，然后才能使用命令来操作。

在默认情况下，在显示客户机的QEMU窗口中，"Ctrl+Alt+2"组合键可以切换到QEMU monitor中，从monitor窗口中按"Ctrl+Alt+1"组合键可以回到客户机标准显示窗口。

当然，并非在所有情况下都使用"Ctrl+Alt+2"快捷键切换到monitor窗口，正如4.6.5节中介绍的那样，如果使用SDL显示，且在使

用qemu-kvm命令行启动客户机时添加了"-alt-grab"或"-ctrl-grab"参数，则会使该组合键被对应修改为"Ctrl+Alt+Shift+2"或“右Ctrl+2”组合键。

如果所有的情况都一定要到图形窗口（SDL或VNC）才能操作QEMU monitor，那么在某些完全不能使用图形界面的情况下将会受到一些限制。其实，QEMU提供了如下的参数来灵活地控制monitor的重定向。

```
-monitor dev
```

该参数的作用是将monitor重定向到宿主机的dev设备上。关于dev设备这个选项的写法有很多种，下面简单介绍其中的几种。

1）vc

即虚拟控制台（Virtual Console），不加"-monitor"参数就会使用"-monitor vc"作为默认参数。而且，还可以用于指定monitor虚拟控制台的宽度和长度，如"vc:800x600"表示宽度、长度分别为800像素、600像素，"vc:80Cx24C"则表示宽度、长度分别为80个字符宽和24个字符长，这里的C代表字符（character）。注意，只有选择这个"vc"为"-monitor"的选项时，利用前面介绍的"Ctrl+Alt+2"组合键才能切换到monitor窗口，其他情况下不能用这个组合键。

2）/dev/XXX

使用宿主机的终端（tty），如"-monitor/dev/ttyS0"是将monitor重定向到宿主机的ttyS0串口上去，而且QEMU会根据QEMU模拟器的配置来自动设置该串口的一些参数。

3）null

空设备，表示不将monitor重定向到任何设备上，无论怎样也不能连接上monitor。

4）stdio

标准输入输出，不需要图形界面的支持。"-monitor stdio"将monitor重定向到当前命令行所在标准输入输出上，可以在运行QEMU命令后直接就默认连接到monitor中，操作起来非常方便。当需要使用较多QEMU monitor的命令时（这是笔者经常使用的方式，在前面一些章中已提及这种使用方式），命令行示例如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic -net tap -monitor stdio
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) VNC server running on '::1:5900'
(qemu) help device_add
device_add driver[,prop=value][,...] -- add device, like -device
on the command line
```

上面的命令行中演示了通过qemu-kvm命令行启动客户机后，标准输入输出中显示了QEMU monitor，然后在monitor中运行了"help

device_add"命令来查看"device_add"命令的帮助手册。

5.9.2 常用命令介绍

前面一些章节中已经根据示例介绍和使用过qemu-kvm的monitor中不少的命令了，本节将系统性地选取其中一些重要的命令进行简单的介绍，以便读者对monitor中命令的功能有一个全局的认识。

1.help显示帮助信息

help命令可以显示其他命令的帮助信息，其命令格式为：

help 或 ? [*cmd*]

"help"与“?”命令是同一个命令，都是显示命令的帮助信息。它后面不加cmd命令作为参数时，help命令（单独的"help"或“?”）将显示该QEMU中支持的所有命令及其简要的帮助手册。当有cmd参数时，"help cmd"将显示cmd命令的帮助信息，如果cmd不存在，则帮助信息输出为空。

在monitor中使用help命令的几个示例，命令行操作如下：

```
(qemu) help migrate
migrate [-d] [-b] [-i] uri -- migrate to URI (using -d to not wait
for completion)
                        -b for migration without shared storage with
full copy of disk
                        -i for migration without shared storage with
incremental copy of disk (base image shared between src and destination)
(qemu) help device_add
device_add driver[,prop=value][,...] -- add device, like -device
on the command line
```

```
(qemu) help savevm
savevm [tag|id] -- save a VM snapshot. If no tag or id are provided,
a new snapshot is created
(qemu) ? savevm
savevm [tag|id] -- save a VM snapshot. If no tag or id are provided,
a new snapshot is created
```

2. info显示系统状态

info命令显示当前系统状态的各种信息，也是monitor中一个很常用的命令，其命令格式如下：

info *subcommand*

显示subcommand中描述的系统状态。如果subcommand为空，则显示当前可用的所有的各种info命令组合及其介绍，这与"help info"命令显示的内容相同。

在前面的章节中已经多次用到info命令来查看客户机系统的状态了，下面单独介绍一些常用的info命令的基本功能。

（1）info version

查看QEMU的版本信息。

（2）info kvm

查看当前QEMU是否有KVM的支持。

（3）info name

显示当前客户机的名称。

（4）info status

显示当前客户机的运行状态，可能为运行中（running）和暂停（paused）状态。

（5）info uuid

查看当前客户机的UUID[19] 标识。

（6）info cpus

查看客户机各个vCPU的信息。

（7）info registers

查看客户机的寄存器状态信息。

（8）info tlb

查看TLB信息，显示了客户机虚拟地址到客户机物理地址的映射。

（9）info mem

查看正在活动中的虚拟内存页。

（10）info numa

查看客户机中看到的NUMA结构。

（11）info mtree

以树状结构展示内存的信息。

（12）info balloon

查看ballooning的使用情况。

（13）info pci

查看PCI设备的状态信息。

（14）info qtree

以树状结构显示客户机中的所有设备。

（15）info block

查看块设备的信息，如硬盘、软盘、光盘驱动器等。

（16）info chardev

查看字符设备的信息，如串口、并口和这里的monitor设备等。

（17）info network

查看客户的网络配置信息，包括VLAN及其关联的网络设备。

（18）info usb

查看客户机中虚拟USB hub上的USB设备。

（19）info usbhost

查看宿主机中的USB设备的信息。

（20）info snapshots

显示当前系统中已保存的客户机快照的信息。

（21）info migrate

查看当前客户机迁移的状态。

（22）info roms

显示客户机使用的BIOS等ROM文件的信息。

（23）info vnc

显示当前客户机的VNC状态。

（24）info history

查看当前的QEMU monitor中命令行执行的历史记录。

在QEMUmonitor中实际执行其中的几个命令，命令行如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.qcow2
-smp 2 -m 1024 -net nic -net tap -monitor stdio
VNC server running on '::1:5900'
QEMU 1.1.0 monitor - type 'help' for more information
(qemu) info version
1.1.0 (qemu-kvm-devel)
(qemu) info kvm
kvm support: enabled
(qemu) info name
(qemu) info status
```

```
VM status: running
(qemu) info cpus
* CPU #0: pc=0xffffffff810387cb (halted) thread_id=21665
  CPU #1: pc=0xffffffff810387cb (halted) thread_id=21666
(qemu) info block
ide0-hd0: removable=0 io-status=ok file=rhel6u3.qcow2
backing_file=/images/linux/ia32e_rhel6u3.img ro=0 drv=qcow2
encrypted=0 bps=0 bps_rd=0 bps_wr=0 iops=0 iops_rd=0 iops_wr=0
ide1-cd0: removable=1 locked=0 tray-open=0 io-status=ok [not inserted]
floppy0: removable=1 locked=0 tray-open=0 [not inserted]
sd0: removable=1 locked=0 tray-open=0 [not inserted]
(qemu) info network
VLAN 0 devices:
  tap.0: type=tap,ifname=tap0,script=/etc/qemu-ifup,downscript=/etc/qemu-ifdown
  rtl8139.0: type=nic,model=rtl8139,macaddr=52:54:00:12:34:56
Devices not on any VLAN:
(qemu) info snapshots
There is no snapshot available.
(qemu) info vnc
Server:
     address: ::1:5900
        auth: none
Client: none
(qemu) info history
0: 'info version'
1: 'info kvm'
2: 'info name'
3: 'info status'
4: 'info cpus'
5: 'info block'
6: 'info network'
7: 'info snapshots'
8: 'info vnc'
9: 'info history'
```

3. 已使用过的命令

在前面的章节中，已经在示例中介绍了monitor中的一些命令，本节对它们进行简单的回顾。

（1）info

在上一小节已经详细介绍过info命令了，5.8节之前使用过的info命令包括：info kvm、info cpus、info block、info network、info pci、info balloon、info migrate等。

（2）commit

提交修改部分的变化到磁盘镜像中（在使用了"-snapshot"启动参数），或提交变化部分到使用后端镜像文件。

（3）cont或c

恢复QEMU模拟器继续工作。另外，"stop"是暂停QEMU模拟器的命令。

（4）change

改变一个设备的配置，如"change vnc localhost:2"改变VNC的配置，"change vnc password"更改VNC连接的密码，"change ide1-cd0/path/to/some.iso"改变客户机中光驱加载的光盘。

（5）balloon

改变分配给客户机的内存大小，如"balloon 512"表示改变分配给客户机的内存大小为512 MB。

（6）device_add和device_del

动态添加或移除设备，如"device_add pci-assign,host=02:00.0,id=mydev"将宿主机中的BDF编号为02:00.0的PCI设备分配给客户机，而"device_del mydev"移除刚才添加的设备。

（7）usb_add和usb_del

添加或移除一个USB设备，如"usb_add host:002.004"表示添加宿主机的002号USB总线中的004设备到客户机中，"usb_del 0.2"表示删除客户机中的某个USB设备。

（8）savevm、loadvm和delvm

创建、加载和删除客户机的快照，如"savevm mytag"表示根据当前客户机状态创建标志为"mytag"的快照，"loadvm mytag"表示加载客户机标志为"mytag"快照时的状态，而"delvm mytag"表示删除"mytag"标志的客户机快照。

（9）migrate和migrate_cancel

动态迁移和取消动态迁移，如"migrate tcp:des_ip:6666"表示动态迁移当前客户机到IP地址为"des_ip"的宿主机的TCP 6666端口上去，而"migrate_cancel"则表示取消当前进行中的动态迁移过程。

4.其他常见命令

除了前面章节中使用过的部分命令，QEMU monitor中还有很多非常有用的命令，本节选取其中一些常用的进行简单介绍。

（1）cpu index

设置默认的CPU为index数字指定的，在info cpus命令的输出中，星号（*）标识的CPU就是系统默认的CPU，几乎所有的中断请求都会优先发到默认CPU上去。如下命令行演示了"cpu index"命令的作用。

```
(qemu) info cpus
* CPU #0: pc=0xffffffff810387cb (halted) thread_id=23634
  CPU #1: pc=0xffffffff810387cb (halted) thread_id=23635
(qemu) cpu 1
(qemu) info cpus

  CPU #0: pc=0xffffffff810387cb (halted) thread_id=23634
* CPU #1: pc=0xffffffff810387cb (halted) thread_id=23635
```

在"cpu 1"命令后，系统的默认CPU变为CPU#1了。另外，利用"cpu_set num online|offline"命令可以添加或移除num数量的CPU，前面章节中已经提及过目前这个命令不生效，有bug存在。

（2）log和logfile

"log item1[,...]"将制定的item1项目的log保存到/tmp/qemu.log中；而"logfile filename"命令设置log文件输出到filename文件中而不是默认的/temp/qemu.log文件。

（3）sendkey keys

向客户机发送keys按键（或组合键），就如同非虚拟环境中那样的按键效果。如果同时发送的是多个按键的组合，则按键之间用“-”来连接。如"sendkey ctrl-alt-f2"命令向客户机发送"ctrl-alt-f2"键，将会切换客户机的显示输出到tty2终端；"sendkey ctrl-alt-delete"命令则会发送"ctrl-alt-delete"键，在文本模式的客户机Linux系统中该组合键会重启系统。

用"sendkey ctrl-alt-f1"、"sendkey ctrl-alt-f2"、"sendkey ctrl-alt-f5"切换到客户机的tty1、tty2、tty5等终端登录系统，然后ssh连接到系统中查看当前系统已登录用户的状态如下。

```
[root@kvm-guest ~]# who
root     tty2         2012-11-03 22:39
root     tty1         2012-11-03 22:35
root     tty5         2012-11-03 22:39
root     pts/0        2012-11-03 22:46 (192.168.162.55)
```

（4）system_powerdown、system_reset和system_wakeup

❑system_powerdown向客户机发送关闭电源的事件通知，一般会让客户机执行关机操作。

❑system_reset让客户机系统重置，相当于直接拔掉电源，然后插上电源，按开机键开机。

❑system_wakeup将客户机从暂停（suspend）中唤醒。

使用这几个命令要小心，特别是system_reset命令是很“暴力”的，可能会损坏客户机系统中的文件系统。

（5）x和xp

❑x/fmtaddr转存（dump）出从addr开始的虚拟内存地址。

❑xp/fmtaddr转存出从addr开始的物理内存地址。

在上面两个命令中，fmt指定如何格式化输出转存出来的内存信息。fmt格式的语法是:/{count}{format}{size}。其中，count表示被转存出来条目的数量，format可以是x（hex，十六进制）、d（有符号的十进制）、u（无符号的十进制）、o（八进制）、c（字符）、i（asm汇编指令），size可以是b（8位）、h（16位）、w（32位）、g（64位）。另外，在x86架构体系下，format中的i可以根据实际指令长度自动设置size为h（16位）或w（32位）。x和xp这两个命令可以用于对客户机或者QEMU开发过程中的调试。使用x和xp转存出一些内存信息，如下：

```
(qemu) x/10i $eip
0xffffffff810387cb:  leaveq
0xffffffff810387cc:  retq
0xffffffff810387cd:  nopl   (%rax)
0xffffffff810387d0:  push   %rbp
0xffffffff810387d1:  mov    %rsp,%rbp
0xffffffff810387d4:  nopl   0x0(%rax,%rax,1)
0xffffffff810387d9:  hlt
0xffffffff810387da:  leaveq
0xffffffff810387db:  retq
0xffffffff810387dc:  nopl   0x0(%rax)
(qemu) xp/80xh 0xb8000
00000000000b8000: 0x0753 0x0774 0x0761 0x0772 0x0774 0x0769 0x076e 0x0767
00000000000b8010: 0x0720 0x0772 0x0770 0x0763 0x0762 0x0769 0x076e 0x0764
00000000000b8020: 0x073a 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8030: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8040: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8050: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8060: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8070: 0x0720 0x0720 0x0720 0x075b 0x0220 0x0220 0x024f 0x024b
00000000000b8080: 0x0220 0x0220 0x075d 0x0720 0x0720 0x0720 0x0720 0x0720
00000000000b8090: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
```

（6）p或print fmt expr

按照fmt格式打印expr表达式的值，可以使用$reg来访问CPU寄存器。如"print 1+2"就是计算“1+2”表达式的值，而"p$cs"就是打印CS寄存器的值。使用p或print命令的示例如下：

```
(qemu) p 100+200
300
(qemu) print 100+200
300
(qemu) p $ecx
0
(qemu) p $cs
16
(qemu) p $eip
-2130475061
(qemu) p $eax
0
(qemu) p $ss
24
```

（7）q或quit

执行q或quit命令，直接退出QEMU模拟器，QEMU进程会被杀掉。

5.10 qemu-kvm命令行参数

用户使用QEMU/KVM之时，一般有两个途径与客户机进行交互和配置，其中一个途径是通过前一节介绍的QEMU monitor，另一个就是通过qemu-kvm命令行。用户通过qemu-kvm命令行启动个客户机，并通过qemu-kvm命令行的各种参数来配置客户机。前面各个章节在介绍KVM的某个功能时一般都会提及qemu-kvm命令行启动时使用什么参数来达到效果。本节将会简单总结一下之前用过的qemu-kvm命令行参数，然后介绍另外一些未曾介绍的重要参数的用法和功能。

5.10.1 回顾已用过的参数

1.qemu-kvm命令基本格式

一般来说x86_64平台上的qemu-kvm工具的命令行格式如下：

```
qemu-system-x86_64[options] [disk_image]
```

其中，options是各种选项、参数，disk_image是客户机的磁盘镜像文件（默认被挂载为第一个IDE磁盘设备）。而关于disk_image的写法也是多种多样的，如可以通过"-hda"参数使用IDE磁盘，也可以用"-driver"参数来提供磁盘镜像，在少数情况下也可以没有磁盘镜像参数。

2.CPU相关的参数

（1）-cpu参数

指定CPU模型，如"-cpu SandyBridge"参数指定给客户机模拟Intel的代号为SandyBridge的CPU。默认的CPU模型为qemu64，"-cpu?"可以查询当前qemu-kvm支持哪些CPU模型。

可以用一个CPU模型作为基础，然后用“+”号将部分的CPU特性添加到基础模型中去，如"-cpu qemu64,+avx"将在qemu64模型中添加对AVX支持的特性，"-cpu qemu64,+vmx"将在qemu64模型中添加Intel VMX特性。

如果想尽可能多地将宿主机的CPU特性暴露给客户机使用，则可以使用"-cpu host"参数。当然，使用"-cpu host"参数会带来动态迁移的限制，不让客户机在不同的CPU硬件上迁移。

（2）-smp参数

```
-smp n[,cores=cores][,threads=threads][,sockets=sockets]
```

设置客户机总共有n个逻辑CPU，并设置了其中CPU socket的数量、每个Socket上核心（core）的数量、每个核心上的线程（thread）数量。其中：n=sockets×cores×threads。

3.内存相关的参数

（1）-m megs参数

设置客户机内存大小为megs MB。默认单位为MB，如"-m 1024"就表示1024MB内存，也可以使用G来表示GB为单位的内存大小，如"-m 4G"表示4GB内存大小。

（2）-mem-path path参数

从path路径表示的临时文件中为客户机分配内存，主要是分配大页内存（如2 MB大页），如"-mem-path/dev/hugepages"，可以参考4.3节“内存配置”。

（3）-mem-prealloc参数

启动时即分配全部的内存，而不是根据客户机请求而动态分配，必须与"-mem-path"参数一起使用。

（4）-balloon开启内存气球的设置

"-balloon virtio"为客户机提供virtio_balloon设备，从而通过内存气球balloon，可以在QEMU monitor中用"balloon"命令来调节客户机占用内存的大小（在qemu-kvm命令行启动时的"-m"参数设置的内存范围内）。

4.磁盘相关的参数

（1）-hda、-hdb和-cdrom等参数

设置客户机的IDE磁盘和光盘设备。如"-hda rhel6u3.img"将rhel6u3.img镜像文件作为客户机的第一个IDE磁盘。

（2）-drive参数

详细地配置一个驱动器，如：在介绍半虚拟化驱动时，用到过"-drive file=rhel6u3.img,if=virtio"的参数配置使用virtio-block驱动来支持该磁盘文件。

（3）-boot参数

设置客户机启动时的各种选项（包括启动顺序等），如：在介绍客户机系统的安装时，使用到"-boot order=dc-hda rhel6u3.img-cdrom rhel6u3.iso"参数，让rhel6u3.img（未安装系统）文件作为IDE磁盘，安装光盘rhel6u3.iso作为IDE光驱，并且从光盘启动客户机，从而让客户机进入到系统安装的流程中。

5.网络相关的参数

（1）-net nic参数

为客户机创建一个网卡（NIC），凡是使用qemu-kvm模拟的网卡作为客户机网络设备的情况都应该使用该参数。当然，如果用VT-d方式将宿主机网卡直接分配给客户机使用，是不需要"-net nic"参数的。

（2）-net user参数

让客户机使用不需要管理员权限的用户模式网络（user mode network），如"-net nic-net user"（详见4.5.4节“QEMU内部的用户模式网络”）。

（3）-net tap参数

使用宿主机的TAP网络接口来帮助客户机建立网络。使用网桥连接和NAT模式网络的客户机都会使用到"-net tap"参数。如"-net nic-net tap,ifname=tap1,script=/etc/qemu-ifup,downscript=no"参数就是在4.5.2节中使用网桥模式网络的命令行参数。

（4）-net dump参数

转存（dump）出网络中的数据流量，之后可以用tcpdump或Wireshark工具来分析。

（5）-net none参数

当不需要配置任何网络设备时，需要使用"-net none"参数，因为如果不添加"-net"参数，则会被默认设置为"-net nic-net user"参数。

6.图形显示相关的参数

（1）-sdl参数

使用SDL方式显示客户机。如果在qemu-kvm编译时已经将SDL的支持编译进去了，则qemu-kvm命令行在默认情况下（不加"-sdl"）也会使用SDL方式来显示客户机。

（2）-vnc参数

使用VNC方式显示客户机。只有在进行qemu-kvm编译时没有添加SDL支持，但是编译了VNC相关的支持，才会默认开启VNC方式。在有SDL支持的qemu-kvm工具中，需要使用"-vnc"参数来让客户机显示在VNC中，如"-vnc localhost:2"就将客户机的显示放到本机的2号VNC窗口中，然后在宿主机上可以通过"vncviewer localhost:2"来连接到客户机。

（3）-vga参数

设置客户机中的VGA显卡类型，默认值为"-vga cirrus"，默认会为客户机模拟出"Cirrus Logic GD5446"显卡。可以使用"-vga std"参

数来模拟带有Bochs VBE扩展的标准VGA显卡，而"-vga none"参数是不为客户机分配VGA卡，会让VNC或SDL中都没有任何显示。

（4）-nographic参数

完全关闭QEMU的图形界面输出，从而让QEMU在该模式下完全成为简单的命令行工具。而QEMU中模拟产生的串口被重定向到了当前的控制台（console）中，所以如果在客户机中对其内核进行配置从而让内核的控制台输出重定向到串口后，就依然可以在非图形模式下管理客户机系统。

在"-nograhpic"非图形模式下，使用"Ctrl+a h"组合键（按Ctrl+a之后，再按h键）可以获得终端命令的帮助，如下所示：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic -net tap -nographic
Initializing cgroup subsys cpuset
Initializing cgroup subsys cpu
Linux version 2.6.32-279.el6.x86_64
<!-- 此处省略串口输出中的其他启动信息 -->
Starting atd: [  OK  ]
```

```
Starting rhsmcertd 240 1440[  OK  ]
Starting certmonger: [  OK  ]

Red Hat Enterprise Linux Server release 6.3 (Santiago)
Kernel 2.6.32-279.el6.x86_64 on an x86_64

kvm-guest login:       #此时按键"Ctrl+a h"。
C-a h    print this help
C-a x    exit emulator
C-a s    save disk data back to file (if -snapshot)
C-a t    toggle console timestamps
C-a b    send break (magic sysrq)
C-a c    switch between console and monitor
C-a C-a  sends C-a

kvm-guest login:      #此处按键"Ctrl+a t"，停顿一会儿后，按Enter键
[00:01:15.974] Red Hat Enterprise Linux Server release 6.3 (Santiago)
[00:01:15.975] Kernel 2.6.32-279.el6.x86_64 on an x86_64
[00:01:15.975]
[00:01:15.976] rhel6u3-ga login:

kvm-guest login:        #此处按键"Ctrl+a c"，切换到monitor
QEMU 1.1.0 monitor - type 'help' for more information
(qemu)
```

帮助手册中打印出了多个快捷键，上面演示了"Ctrl+a t"来控制是否显示控制台的时间戳，以及"Ctrl+a c"在控制台（串口也重定向到控制台）与QEMU monitor之间的切换。

7.VT-d和SR-IOV相关的参数

```
-device driver[,prop[=value][,...]]
```

添加一个设备驱动器（driver），其中prop=value是设置驱动器的各项属性。可以用"-device?"参数查看到有哪些可用的驱动器，可

以用"-device driver,?"查看到某个驱动器（driver）支持的所有属性。不管是KVM的VT-d还是SR-IOV特性，都是使用"device"参数将宿主机中的设备完全分配给客户机使用，如"-device pci-assign,host=08:00.0,id=mydev0,addr=0x6"参数就将宿主机的BDF号是08:00.0的设备分配给客户机使用。VT-d和SR-IOV在使用时的区别在于，VT-d中分配的设备是一个物理PCI/PCI-e设备，而SR-IOV使用的是虚拟设备（VF，Virtual Function）。

8.动态迁移的参数

-incoming port参数让qemu-kvm进程进入到迁移监听（migration-listen）模式，而不是真正以命令行中的镜像文件运行客户机。如启动客户机的qemu-kvm命令行中添加了"-incoming tcp:0:6666"参数，表示在6666端口建立一个TCP Socket连接用于接收来自源主机的动态迁移的内容，其中“0”表示允许来自任何主机的连接。

9.已使用过的其他参数

（1）-daemonize参数

在启动时让qemu-kvm作为守护进程在后台运行。如果没有该参数，默认qemu-kvm在启动客户机后就会占用标准输入输出，直到客户

机退出。"-daemonize"参数的使用，可以让一个qemu-kvm进程在后台运行，接着在当前位置进行其他的操作（如启动另一个客户机）。

（2）-usb参数

开启客户机中的USB总线，如"-usb-usbdevice tablet"就是在客户机中模拟USB而不是PS/2的键盘和鼠标，而且使用tablet这种类型的设备实现鼠标的定位。

（3）-enable-kvm参数

打开KVM虚拟化的支持，在qemu-kvm中，"-enable-kvm"默认就是打开的，默认支持KVM虚拟化，而在纯QEMU中，默认没有打开KVM的支持，需要"-enable-kvm"参数来配置。

5.10.2　其他常用参数

本节将介绍前面章节中未介绍的但也非常有用的qemu-kvm命令行参数。

1.-h显示帮助手册

```
[root@jay-linux kvm_demo]#qemu-system-x86_64 -h
QEMU emulator version 1.1.0 (qemu-kvm-devel), Copyright (c)
2003-2008 Fabrice Bellard
usage: qemu-system-x86_64 [options] [disk_image]

'disk_image' is a raw hard disk image for IDE hard disk 0

Standard options:
-h or -help     display this help and exit
-version        display version information and exit
-machine [type=]name[,prop[=value][,...]]
                selects emulated machine (-machine ? for list)
<!-- 以下省略数百行输出信息 -->
```

显示了当前qemu-kvm工具中支持的所有命令行参数。

2.-version显示qemu-kvm的版本信息

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -version
QEMU emulator version 1.1.0 (qemu-kvm-devel), Copyright (c)
2003-2008 Fabrice Bellard
```

显示了当前QEMU模拟器是qemu-kvm开发分支的1.1.0版本。

3.-k设置键盘布局的语言

默认值为en-us（美式英语键盘），一般不需要设置这个参数，除非客户机中键盘布局、按键不准确才需要设置，如"-k fr"表示客户机使用法语（French）的键盘布局。它所支持的键盘布局的语言一般在"/usr/local/share/qemu/keymaps/"目录中，如下：

```
[root@jay-linux kvm_demo]# ls /usr/local/share/qemu/keymaps/
ar    common  de     en-gb  es  fi  fr     fr-ca  hr  is  ja  lv
modifiers  nl-be  pl  pt-br  sl  th
bepo  da      de-ch  en-us  et  fo  fr-be  fr-ch  hu  it  lt  mk
nl         no     pt  ru     sv  tr
```

4.-soundhw开启声卡硬件的支持

可以通过"-soundhw?"查看有效的声卡的种类，在qemu-kvm命令行中添加"-soundhw ac97"参数即可在客户机中使用到Intel 82801AAAC97声卡。在宿主机中查看支持的声卡种类，然后选择使用"ac97"声卡类型启动一个客户机，如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -soundhw ?
Valid sound card names (comma separated):
sb16        Creative Sound Blaster 16
ac97        Intel 82801AA AC97 Audio
es1370      ENSONIQ AudioPCI ES1370
hda         Intel HD Audio
-soundhw all will enable all of the above
[root@jay-linux kvm_demo]# qemu-system-x86_64 rhel6u3.img -smp
2 -m 1024 -net nic -net tap -soundhw ac97
VNC server running on '::1:5900'
```

在客户机中通过"lspci"命令查看到的声卡如下：

```
[root@kvm-guest ~]# lspci | grep -i audio
00:04.0 Multimedia audio controller: Intel Corporation 82801AA
AC'97 Audio Controller (rev 01)
```

5.-display设置显示方式

选择客户机使用的显示方式，它是为了取代前面提及过的较旧的"-sdl"、"-curses"、"-vnc"等参数。如"-display sdl"表示通过SDL方式显示客户机图像输出，"-display curses"表示使用curses/ncurses方式来显示图像，"-display none"表示不显示任何的图像输出（它与"-nographic"参数的区别在于，"-nographic"会显示客户机的串口和并口的输出），"-display vnc=localhost:2"表示将客户机图像输出显示到本地的2号VNC显示端口中（与"-vnc localhost:2"的意义相同）。

6.-name设置客户机名称

设置客户机名称可用于在某宿主机上唯一标识该客户机，如"-name myname"参数就表示设置客户机的名称为"myname"，设置的名字将会在SDL窗口边框的标题中显示，或者在VNC窗口的标题栏中显示。

7.-uuid设置系统的UUID

客户机的UUID标识符，与名称类似，不过一般来说UUID是一个较大系统中唯一的标识符。在libvirt等虚拟机管理工具中就根据UUID来管理所有客户机的唯一标识。在通过qemu-kvm命令行启动客户时添加"-uuid 12345678-1234-1234-1234-123456789abc"参数（UUID是按照8-4-4-4-12个数分布的32个十六进制数字），就配置了客户机的

UUID，然后在QEMU monitor中也可以用"info uuid"命令来查询到该客户机的UUID值。

8.-rtc设置RTC开始时间和时钟类型

"-rtc"参数的完整形式为：

```
-rtc [base=utc|localtime|date][,clock=host|rt|vm][,driftfix=none|slew]
```

其中"base"选项设置客户机的实时时钟（RTC，real-time clock）开始的时间，默认值为"utc"。而当微软的DOS和Windows系统作为客户机时，应该将"base"选项设置为"base=localtime"，否则时钟非常不准确。也可以选择某个具体的时间作为"base"基础时间，如"2012-11-06T22:22:22"或“2012-11-06”这样的格式。

"clock"选项用于设置客户机实时时钟的类型。默认情况下是"clock=host"，表示由宿主机的系统时间来驱动，一般会让客户机使用的RTC时钟比较准确。特别是当宿主机的时间通过与外部时钟进行同步（如NTP方式）而保持准确的时候，默认"clock=host"会提供非常准确的时间。如果设置"clock=rt"，则表示将客户机和宿主机的时间进行隔离，而不进行校对。如果设置"clock=vm"，则当客户机暂停的时候，客户机时间将不会继续向前计时。

"driftfix"选项用于设置是否进行时间漂移的修复。默认值是"driftfix=none"，表示不进行客户机时间偏移的修复。

而"driftfix=slew"则表示当客户机中可能出现时间漂移的时能够自动修复，某些Windows作为客户机时可能会出现时间不太准确（时间漂移）的情况，这时qemu-kvm先计算出Windows客户机中缺少了多少个时间中断，然后重新将缺少的时间中断注入到客户机中。

9.存储设备URL的语法参数

如在4.4节“存储配置”中介绍过，QEMU/KVM除了可以使用本地的raw、qcow2等格式的镜像之外，还可以使用远程的如NFS上镜像文件，也可以使用iSCSI、NBD、Sheepdog等网络存储设备上面的存储。对于NFS，挂载后就和使用本地文件没有任何区别，而iSCSI等则在启动时需要使用一些特殊的URL语法以便标识存储的位置。本节并不详细讲解其语法，而是举例简要介绍其基本用法。

（1）iSCSI的URL语法

iSCSI支持qemu-kvm直接访问iSCSI资源和直接使用其镜像文件作为客户机存储，支持磁盘镜像和光盘镜像。qemu-kvm使用iSCSI LUNs的语法为：

```
iscsi://<target-ip>[:<port>]/<target-iqn>/<lun>
```

而iSCSI会话建立的参数为：

```
-iscsi [user=user][,password=password]
       [,header-digest=CRC32C|CR32C-NONE|NONE-CRC32C|NONE
       [,initiator-name=iqn]
```

一个使用iSCSI的示例如下：

```
qemu-system-x86_64 -iscsi \
initiator-name=iqn.2012-11.com.example:my-initiator \
        -cdrom iscsi://192.168.100.1/iqn.2012-11.com.example/2 \
        -drive file=iscsi://192.168.100.1/iqn.2012-11.com.example/1
```

不过需要注意的是，在qemu-kvm配置、编译时需要有libiscsi的支持才行（在运行./configure配置时，添加--enable-libiscsi参数），否则qemu-kvm可能不能支持iSCSI。

（2）NBD的URL语法

QEMU支持使用TCP协议的NBD(Network Block Devices)设备，也支持Unix Domain Socket的NBD设备。

使用TCP的NBD设备，在qemu-kvm中的语法为：

```
nbd:<server-ip>:<port>[:exportname=<export>]
```

而使用Unix Domain Socket的NBD设备，其语法为：

```
nbd:unix:<domain-socket>[:exportname=<export>]
```

在qemu-kvm命令中的示例为：

```
qemu-system-x86_64 -drive file=nbd:192.168.2.1:30000
qemu-system-x86_64 -drive file=nbd:unix:/tmp/nbd-socket
```

（3）Sheepdog的URL语法

第4章中已提及过，Sheepdog可以让QEMU使用分布式存储系统。qemu-kvm支持本地和远程网络的Sheepdog设备。Sheepdog在使用时的语法，可以有如下几种形式：

```
sheepdog:<vdiname>
sheepdog:<vdiname>:<snapid>
sheepdog:<vdiname>:<tag>
sheepdog:<host>:<port>:<vdiname>
sheepdog:<host>:<port>:<vdiname>:<snapid>
sheepdog:<host>:<port>:<vdiname>:<tag>
```

qemu-kvm命令行中的示例为：

```
qemu-system-x86_64 -drive
file=sheepdog:192.168.2.1:30000:MyGuest
```

关于Sheepdog更详细的使用，可以参考：http://www.osrg.net/sheepdog/。

10.-chardev配置字符型设备

```
-chardev backend ,id=id [,mux=on|off] [,options]
```

配置一个字符型设备，其中"backend"可以是"null、socket、udp、msmouse、vc、file、pipe、console、serial、pty、stdio、braille、tty、parport、spicevmc"之一。后端（backend）将会决定后面可用的选项。"id=id"选项设置了该设备的唯一标识，ID可以是包含最多127个字符的字符串。每个设备必须有一个唯一的ID，它可用于其他命令行参数识别该设备。"mux=on|off"选项表示该设备是否多路复用。当该字符型设备被用于多个前端（frontend）使用时，需要启用"mux=on"这个模式。

11.-bios指定客户机的BIOS文件

设置BIOS的文件名称。一般来说，qemu-kvm会到"/usr/local/share/qemu/"目录下去找BIOS文件。但也可以使用"-L path"参数来改变qemu-kvm查找BIOS、VGA BIOS、keymaps等文件的目录。

12.-no-reboot和-no-shutdown参数

"-no-reboot"参数，让客户机在执行重启（reboot）操作时，在系统关闭后就退出qemu-kvm进程，而不会再启动客户机。

"-no-shutdown"参数，让客户机执行关机（shutdown）操作时，在系统关闭后，不退出qemu-kvm进程（在正常情况下，系统关闭后就退出qemu-kvm进程），而是保持这个进程存在，它的QEMU monitor依

然可以使用。在需要的情况下，这就允许在关机后切换到monitor中将磁盘镜像的改变提交到真正的镜像文件中。

13.-loadvm加载快照状态

"-loadvm mysnapshot"在qemu-kvm启动客户机时即加载系统的某个快照，这与QEMU monitor中的"loadvm"命令的功能类似。

14.-pidfile保存进程ID到文件中

"-pidfile qemu-pidfile"保存qemu-kvm进程的PID文件到qemu-pidfile中，这对在某些脚本中对该进程继续做处理提供了便利（如设置该进程的CPU亲和性，监控该进程的运行状态）。

15.-nodefaults不创建默认的设备

在默认情况下，qemu-kvm会为客户机配置一些默认的设备，如串口、并口、虚拟控制台、monitor设备、VGA显卡等。使用了"-nodefaults"参数可以完全禁止默认创建的设置，而仅仅使用命令行中显式指定的设备。

16.-readconfig和-writeconfig参数

"-readconfig guest-confg"参数从文件中读取客户机设备的配置（注意仅仅是设备的配置信息，不包含CPU、内存之类的信息）。当

qemu-kvm命令行参数的长度超过系统允许的最长参数的个数时，qemu-kvm将会遇到错误信息"arg list too long"，这时如果将需要的配置写到文件中，使用"-readconfig"参数来读取配置，就可以避免参数过长的问题。在Linux系统中，可以用"getconf ARG_MAX"命令查看系统能支持的命令行参数的字符个数。

"-writeconfig guest-confg"参数表示将客户机中设备的配置写到文件中；"-writeconfig"参数则会将设备的配置打印在标准输出中。保存好的配置文件，可以用于刚才介绍的"-readconfig"参数。

笔者保存下来的一个示例设备配置文件如下：

```
[drive]
  media = "disk"
  index = "0"
  file = "rhel6u3.img"

[net]
  type = "nic"

[net]
  type = "tap"
[cpudef]
  name = "Conroe"
  level = "2"
  vendor = "GenuineIntel"
  family = "6"
<!-- 以下省略数十行信息  -->
```

17.-nodefconfig和-no-user-config参数

"-nodefconfig"参数使qemu-kvm不加载默认的配置文件。在默认情况下，qemu-kvm会加载/usr/local/share/qemu/目录下的配置文件（当然不同系统中可能目录不一致的），如cpus-x86_64.conf文件等。

"-no-user-config"参数使qemu-kvm不加载用户自定义的配置文件（其目录是在编译QEMU时指定的，默认还是

为"/usr/local/share/qemu/"），但是依然会加载qemu-kvm原本提供的配置文件（如cpus-x86_64.conf）。

18.Linux或多重启动相关的参数

qemu-kvm提供了一些参数，可以让用户不用安装系统到磁盘上即可启动Linux或多重启动的内核，这个功能可以用于进行早期调试或测试各种不同的内核。

（1）"-kernel bzImage"参数

使用"bzImage"作为客户机内核镜像。这个内核可以是一个普通Linux内核或多重启动的格式中的内核镜像。

（2）"-append cmdline"参数

使用"cmdline"作为内核附加的命令选项

（3）"-initrd file"参数

使用"file"作为初始化启动是的内存盘（ram disk）。

（4）"-initrd" file1 arg=foo,file2""参数

仅用于多重启动中，使用file1和file2作为模块，并将"arg=foo"作为参数传递给第一个模块（file1）。

（5）"-dtb file"参数

使用file文件作为设备树二进制（dtb，device tree binary）镜像，在启动时将其传递给客户机内核。

19.-serial串口重定向

"-serial dev"参数将客户机的串口重定向到宿主机的字符型设备dev上。可以重复多次使用"-serial"参数，以便为客户机模拟多个串口，最多可以达到4个串口（ttyS0～ttyS3）。

在默认情况下，如果客户机工作在图形模式，则串口被重定向到虚拟控制台（vc,virtual console），"Ctrl+Alt+3"组合键可以切换到该串口；如果客户机工作在非图形模式下（使用了"-nographics"参数），串口默认被重定向到标准输入输出（stdio）。还可以将串口重定向到一个文件中，如"-serial file:myserial.log"参数就将串口输出重定向到当前目录的myserial.log文件中，如果将客户机的内核输出也重定向到串口，那么这样就可以将内核打印的信息都保存到myserial.log文件中了，特别是当客户机系统崩溃时，这样保存的串口输出日志文件可以辅助我们分析系统崩溃时的具体状态。

对于串口重定向，可以选择很多种设备（dev）作为重定向的输出，下面简单介绍其中的几种。

（1）vc

虚拟控制台（virtual console），这是默认的选择，与"-monitor"重定向监控器输出类似，还可以指定串口重定向的虚拟控制台的宽度和长度，如"vc:800x600"表示宽度、长度分别为800像素、600像素。

（2）pty

重定向到虚拟终端（pty），系统默认自动创建一个新的虚拟终端。

（3）none

不重定向到任何设备。

（4）null

重定向到空的设备。

（5）/dev/XXX

使用宿主机系统更多终端设备（如/dev/ttyS0）。

（6）file:filename

重定向到filename这个文件中，只能保存串口输出，不能输入字符进行交互。

（7）stdio

重定向到当前的标准输入输出。

（8）pipe:filename

重定向到filename名字的管道。

（9）其他

还可以将串口重定向到TCP或UDP建立的网络控制台中，还可以重定向到Unix Domain Socket。

20.调试相关的参数

qemu-kvm中也有很多和调试相关的参数，下面简单介绍其中的几个参数。

（1）-singlestep

以单步执行的模式运行QEMU模拟器。

（2）-S

在启动时并不启动CPU，需要在monitor中运行"c"（或"cont"）命令才能继续运行。它可以配合"-gdb"参数一起使用，启动后，让gdb远程连接到qemu-kvm上，然后再继续运行。

（3）-gdb dev

运行GDB服务端（gdbserver），等待GDB连接到dev设备上。典型的连接可能是基于TCP协议的，当然也可能是基于UDP协议、虚拟终端（pty），甚至是标准输入输出（stdio）的。"-gdb"参数配置可以让内核开发者很方便地使用qemu-kvm运行内核，然后用GDB工具连接上去进行调试（debug）。

在qemu-kvm命令行中使用TCP方式的"-gdb"参数，示例如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -kernel
vmlinuz-3.6 -initrd initrd-3.6.img -gdb tcp::1234 -S
```

在本机的GDB中可以运行如下命令连接到qemu-kvm运行的内核上去，当然如果是远程调试就需要添加一些网络IP地址的参数：

```
(gdb) target remote :1234
```

而在使用标准输入输出（stdio）时，允许在GDB中执行qemu-kvm，然后通过管道连接到qemu-kvm的客户机中，例如可以用如下的方式来使用：

```
(gdb) target remote | exec qemu-system-x86_64 -gdb stdio -hda
rhel6u3.img -smp 2 -m 1024
```

（4）-s

"-s"参数是"-gdb tcp::1234"的简写表达方式，即在TCP 1234端口打开一个GDB服务器。

（5）-d

将QEMU的日志保存在/tmp/qemu.log中，以便调试时查看日志。

（6）-D logfile

将QEMU的日志保存到logfile文件中（而不是"-d"参数指定的/tmp/qemu.log）中。

（7）-watchdog model

创建一个虚拟的硬件看门狗（watchdog）设备，对于一个客户机而言，只能够启用一个看门狗。在客户机中必须要有看门狗的驱动程序，周期性地轮询这个看门狗，否则客户机将会被重启。"model"选项是QEMU模拟产生的硬件看门狗的模型，一般有两个可选"ib700"（iBASE 700）和"i6300esb"（Intel 6300ESB I/O controller hub）。使用"-watchdog?"可以查看到所有可用的硬件看门狗模型的列表，命令行演示如下：

```
[root@jay-linux kvm_demo]# qemu-system-x86_64 -watchdog ?
       i6300esb        Intel 6300ESB
       ib700   iBASE 700
```

查看客户机中内核是否支持这些看门狗，在客户机中命令行如下：

```
[root@jay-linux kvm_demo]# grep -i i6300esb \ /boot/config-2.6.32-279.el6.x86_64
CONFIG_I6300ESB_WDT=m
[root@jay-linux kvm_demo]# grep -i ib700 /boot/config-2.6.32-279.el6.x86_64
CONFIG_IB700_WDT=m
```

（8）-watchdog-action action

"action"选项控制qemu-kvm在看门狗定时器到期时的动作。默认动作是"reset"，它表示“暴力”重置客户机（让客户机掉电然后重启）。一些可选的动作包括："shutdown"表示正常关闭客户机系统，"poweroff"表示正常关闭系统后再关闭电源，"pause"表示暂停客户机，"debug"表示打印出调试信息然后继续运行，"none"表示什么也不做。看门狗相关的一个示例参数如下：

```
-watchdog i6300esb -watchdog-action pause
```

（9）-trace-unassigned

跟踪未分配的内存访问或未分配的I/O访问，并记录到标准错误输出（stderr）。

（10）-trace[events=file][,file=lofile]

指定一些跟踪的选项。其中"event=file"中的file文件的格式必须是每行包含一个事件（event）的名称，其中所有的事件名称都已在qemu-kvm源代码中的"trace-events"文件中列出来了。"event=file"这个选项只有在qemu-kvm编译时指定的跟踪后端（tracing backend）为"simple"或"stderr"时才可用，编译qemu-kvm的配置命令演示如下：

```
[root@jay-linux qemu-kvm.git]# ./configure --enable-trace-backend=simple
<!-- 省略其他输出信息 -->
Trace backend     simple
Trace output file trace-<pid>
```

"file=logfile"选项是将跟踪的日志输出到logfile文件中。该选项只有在qemu-kvm编译时选择了"simple"作为跟踪后端时才可用。

5.11 迁移到KVM虚拟化环境

前面的一些章节已经介绍过虚拟化带来的好处，也介绍了KVM虚拟化中的各种功能。那么，如何才能从其他虚拟化方案迁移到KVM虚拟化中呢？或者如何能将物理原生系统直接迁移到KVM虚拟化环境中去呢？本节将会介绍迁移到KVM虚拟化环境的方法和一些注意事项。

5.11.1 virt-v2v工具介绍

V2V是“从一种虚拟化迁移到另一种虚拟化”（Virtual to Virtual）过程的简称。不借助于其他的工具，使用纯QEMU/KVM能实现KVM到KVM的迁移，而不能实现从Xen等其他Hypervisor迁移到KVM上去。virt-v2v工具可用于将虚拟客户机从一些Hypervisor（也包括KVM自身）迁移到KVM环境中。它要求目的宿主机中的KVM是由libvirt管理的或者是由RHEV（Red Hat Enterprise Virtualization）管理的。virt-v2v是由Redhat的工程师Matthew Booth开发的命令行工具，它也是一个完全开源的项目，除了Matthew自己，也有一些其他开发者为该项目贡献过代码。可以用下面的命令克隆最新的virt-v2v开发源代码：

```
git clone git://git.fedorahosted.org/virt-v2v.git
```

virt-v2v默认会尽可能由转换过来的虚拟客户机使用半虚拟化的驱动（virtio）。根据Redhat官方对virt-v2v工具的的描述[20]，RHEL 6.x系统中的virt-v2v工具支持从KVM、Xen、VMware ESX等迁移到KVM上去（最新的virt-v2v还支持VirtualBox的转化）。当然从KVM到KVM的迁移，使用前面“5.4动态迁移”中的方法也是可以完成的。

与“5.4动态迁移”中介绍的不同，virt-v2v工具的迁移不是动态迁移，在执行迁移操作之前，必须要在源宿主机（Xen、VMware等）上关闭待迁移的客户机，所以，实际上，可以说virt-v2v工具实现的是一种转化，将Xen、VMware等Hypervisor的客户机转化为KVM客户机。一般来说，virt-v2v要依赖于libvirt（可参考“第6章KVM管理工具”中对libvirt的介绍），让libvirt为不同的Hypervisor提供一个公共的适配层，为向KVM转化提供了必要功能。它要求源宿主机中运行着libvirtd（当然VMware除外），迁移到的目标宿主机上也要运行着libvirtd。

根据Redhat官方文档的介绍，virt-v2v支持多种Linux和Windows客户机的迁移和转换，包括：RHEL 4、RHEL 5、RHEL 6、Windows XP、Windows Vista、Windows 7、Windows Server 2003、Windows Server 2008等。当然，尽管没有官方的声明和支持，通过实际测试还是可以发现，使用virt-v2v也能够使其他一些类型客户机迁移到KVM（如Fedora、Ubuntu等）。

virt-v2v的可执行程序已经在一些Linux发行版中发布了。Fedora 11和之后的Fedora都已经包含了virt-v2v工具，RHEL 5.x（x≥6）和RHEL 6.x也已经发布了virt-v2v软件包，可以使用"yum install virt-v2v"来安装。

与V2V的概念相对应，还有一个P2V的概念，P2V是“物理机迁移到虚拟化环境”（Physical to Virtual）的缩写。virt-p2v的代码也在virt-v2v的代码库中，在使用时，还需要一个用于P2V迁移的可启动的ISO（Redhat的注册用户可以通过RHN通道下载）。从Fedora 14和RHEL 6.3开始的Fedora和RHEL版本开始提供对virt-p2v的支持。

5.11.2 从Xen迁移到KVM

在Linux开源虚拟化领域中，Xen和QEMU/KVM是两个比较广泛使用的Hypervisor，Xen的部分用户可能会有将Hypervisor从Xen迁移到KVM的需求。virt-v2v工具支持在libvirt管理下在Xen客户机迁移到KVM虚拟化环境中，既支持本地转化，也支持远程迁移。

将Xen上的一个客户机迁移到KVM的过程，需要将客户机的配置转化为对应的KVM客户机配置，也需要将客户机镜像文件传输到目的宿主机上。磁盘镜像的传输需要SSH会话的支持，所以需要保证Xen宿主机支持SSH访问，另外，在每个磁盘进行传输时都会提示输入SSH用户名

和密码，可以考虑使用SSH密钥认证来避免多次的SSH用户名和密码交互。

使用virt-v2v工具将Xen客户机迁移到KVM中的命令示例如下：

```
virt-v2v -ic xen+ssh://root@xen0.demo.com -os pool -b brnamevm-name
```

-ic URI表示连接远程Xen宿主机中libvirt的URI（可以参考“6.1.6建立到Hypervisor的连接”中对URI的详细介绍）。-os pool表示迁移过来后，用于存放镜像文件的本地存储池。-b brname（即：--bridge brname）表示本地网桥的名称，用于建立与客户机的网络连接。如果本地使用虚拟网络，则使用-n network（或--network network）参数来指定虚拟网络。vm-name表示的是在Xen的源宿主机中将要被迁移的客户机的名称。

Xen中的全虚拟化类型客户机（HVM）和半虚拟化客户机（PV guest，XenU），都可以使用上面介绍的命令迁移到KVM。由于KVM是仅支持全虚拟化客户机（HVM）的，所以，支持对Xen半虚拟化客户机的迁移，virt-v2v还会涉及客户机内核修改、转化的过程，可以参考本章注释[20] 中介绍的文档。

下面通过一个示例来介绍一下使用virt-v2v工具将Xen上的HVM客户机迁移到KVM上的过程。其中，源宿主机（Xen的Dom0）使用的是经过升级的Fedora 17系统，Xen的版本是4.1.3，Dom0内核版本是

3.6.1，libvirt版本是0.9.11；KVM宿主机使用的是RHEL 6.3原生系统的内核、qemu-kvm和libvirt（0.9.10版本），并且用"yum install virt-v2v"安装了0.8.7版本的virt-v2v工具。

1）在源宿主机系统（IP地址为192.168.127.163）中，启动libvirtd服务，并且定义好一个名为xen-hvm1的客户机用于本次迁移，该客户机在Xen环境中是可以正常工作的，但需要在迁移之前将该客户及关闭。通过virsh工具查看Xen中libvirt管理的所有客户机的状态，命令行如下：

```
[root@F17-Xen ~]# virsh list --all
 Id    Name                           State
----------------------------------------------------
 0     Domain-0                       running
 -     xen-hvm1                       shut off
```

由上面信息可知，xen-hvm1客户机处于关机状态。

2）在KVM宿主机（目的宿主机）中，启动libvirtd服务，然后使用virt-v2v命令行工具进行Xen到KVM的迁移，命令行操作如下：

```
[root@rhel6u3-ga ~]# service libvirtd start

[root@rhel6u3-ga ~]# virt-v2v -ic
xen+ssh://root@192.168.127.163 -os  default --bridge br0 xen-hvm1
root@192.168.127.163's password:
root@192.168.127.163's password:
```

```
ia32e_rhel6u3.img: 100% [=========================================]D 0h02m09s
virt-v2v: WARNING: /boot/grub/device.map references unknown
device /dev/vda. This entry must be manually fixed after conversion.
virt-v2v: WARNING: The connected hypervisor does not support feature pae.
virt-v2v: xen-hvm1 configured with virtio drivers.
```

由以上信息可知，经过用户命名/密码的验证之后，virt-v2v将客户机的镜像文件传输到KVM宿主机中，磁盘镜像的传输过程可能需要一定的时间，所需的时间长度与磁盘镜像的大小及当前网络带宽有很大的关系。尽管最后有一些警告提示信息，该迁移过程还是正常完成了。

已经提及过，virt-v2v的转换需要先关闭客户机，如果Xen宿主机（Dom0）上的该客户机处于运行中状态，运行上面的virt-v2v命令会得到如下的错误提示：

```
virt-v2v: Guest XX is currently blocked. It must be shut down first.
```

3）在KVM宿主机中，查看迁移过来的客户机镜像和启动该客户机，命令行操作如下：

```
[root@rhel6u3-ga ~]# ls /var/lib/libvirt/images/ia32e_rhel6u3.img
/var/lib/libvirt/images/ia32e_rhel6u3.img

[root@rhel6u3-ga ~]# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     rhel6u3-1                      shut off
 -     rhel6u3-2                      shut off
 -     xen-hvm1                       shut off

[root@rhel6u3-ga ~]# virsh create /etc/libvirt/qemu/xen-hvm1.xml
Domain xen-hvm1 created from /etc/libvirt/qemu/xen-hvm1.xml

[root@rhel6u3-ga ~]# virsh list
 Id    Name                           State
----------------------------------------------------
 2     xen-hvm1                       running
```

由以上信息可知，从Xen上迁移过来的客户机，其镜像文件默认在/var/lib/libvirt/images/目录下，其XML配置文件默认在/etc/libvirt/qemu/目录下。从第一个"virsh list--all"命令可知，迁移过来的客户机默认处于关闭状态。在使用"virsh create"命令启动该客户后，该客户机就处于运行中（Running）状态了。

一般来说，从Xen到KVM迁移成功后，迁移过来的客户机就可以完全正常使用了。不过，由于一些命令和配置的不同，也可能会导致迁移后的客户机网络不通的情况，这就需要自行修改该客户机的XML配置文件；也可能出现磁盘不能识别或启动的问题（Windows客户机迁移时易出现该问题），这一方面需要检查和修改XML配置文件（可以直接改

为模拟IDE磁盘设备而不是virtio），另一方面可以在该客户机中安装virtio-blk相关的磁盘驱动。

由于Xen中也是使用QEMU来作为设备模型，因此Xen中的客户机一般使用raw、qcow2等格式的客户机镜像文件，这与QEMU/KVM中的磁盘镜像格式是完全一致（或兼容）的，不需要进行任何的格式化转换（除非它们的QEMU版本差异太大）。除了使用virt-v2v工具来实现Xen到KVM的迁移，也可以直接将Xen中客户机的磁盘镜像远程复制到KVM宿主机的存储池中，然后根据Xen上面客户机的需求，手动地使用相应的qemu-kvm命令行参数来启动该客户机即可，或者，将libvirt管理的Xen中该客户机的XML文件复制到KVM宿主机中，对该XML配置文件进行相应的修改后，通过libvirt启动该客户机即可。

5.11.3　从VMware迁移到KVM

VMware作为系统虚拟化领域的开拓者和市场领导者之一，其虚拟化产品功能比较强大、易用性也比较良好，所以被很多人了解并在生产环境中使用。不过，美中不足的是，其企业级虚拟化产品的许可证授权费用还是比较昂贵的，特别是有大批量的服务器需要部署WMware ESX/ESXi虚拟化产品时，许可证授权费用就真的不是一笔小数目了。不管是从KVM可以完全免费的角度，还是从KVM基于Linux内核且完全开

源的角度来看，如果考虑从VMware虚拟化中迁移到KVM虚拟化方案中，那么可以参考本小节即将介绍的一些方法。

从VMware迁移到KVM的方法，与前一节中从Xen迁移到KVM的完全类似。可以通过virt-v2v工具来做迁移，实现将VMware ESX中的一个客户机迁移到KVM上。利用virt-v2v迁移VMware客户机的命令行示例如下：

```
virt-v2v -ic esx://esx.demo.com/ -os pool --bridge brname vm-name
```

上面命令行中的命令和参数与前一节介绍的基本类似，只是这里使用了esx://esx.demo.com来表示连接到VMware ESX服务器，将命令vm-name的客户机迁移过来。在连接到VMware的ESX服务器时，一般需要认证和授权。virt-v2v工具支持连接ESX时使用密码认证，它默认读取$HOME/.netrc文件中的机器名、用户名、密码等信息，这与FTP客户端命令"ftp"类似。这个.netrc文件中的格式如下：

```
machine esx.demo.com login root password 123456
```

除了通过virt-v2v工具可以将VMware中运行着的客户机迁移到KVM，也可以采用直接复制VMware客户机镜像到KVM中的方法：先关闭VMware客户机，然后直接将VMware的客户机镜像（一般是.vmdk为后缀的文件）复制到KVM的客户机镜像存储系统系统上（可能是宿主机本地也可能是网络共享存储），接着通过qemu-kvm命令行工具启动该镜像

文件即可。qemu-kvm从0.12版本开始就支持直接使用VMware的vmdk镜像文件启动客户机，一个简单的命令行示例如下：

```
[root@jay-linux ~]# qemu-system-x86_64 -m 1024 win7.vmdk
```

如果qemu-kvm版本较旧，不支持直接使用vmkd镜像文件格式，那么可以将VMware的镜像格式转化为raw或qcow2等在QEMU/KVM中最常用的格式。在4.4.2节“qemu-img命令”中已经介绍了convert命令可以让不同格式的镜像文件进行相互转换。将vmdk格式的镜像文件转化为qcow2格式，然后用qemu-kvm命令行启动，命令行操作过程如下：

```
[root@jay-linux ~]# qemu-img convert win7.vmdk -O qcow2 win7.qcow2
[root@jay-linux ~]# qemu-system-x86_64 -m 1024 win7.qcow2
```

5.11.4 从VirtualBox迁移到KVM

virt-v2v工具从0.8.8版本开始也支持将VirtualBox客户机迁移到KVM上，其方法与从Xen迁移到KVM的方法完全类似，其转化命令示例如下：

```
virt-v2v -ic vbox+ssh://root@vbox.demo.com -os pool -b brnamevm-name
```

在该命令中，仅仅在连接到VirtualBox时使用的URI是vbox+ssh这样的连接方式，而不是用xen+ssh的方式。

除了使用virt-v2v工具来转换，也可以直接将VirtualBox中的客户机镜像文件（一般是以.vdi为后缀的文件）复制到KVM宿主机中使用。较新的qemu-kvm（如1.2.0版本）都支持直接用.vdi格式的镜像文件作为客户机镜像直接启动，命令行示例如下：

```
[root@jay-linux ~]# qemu-system-x86_64 -m 1024 ubuntu.vdi
```

也可以将VirtualBox客户机镜像文件转化为QEMU/KVM中最常用的qcow2或raw格式的镜像文件，然后在qemu-kvm命令行启动转化后的镜像文件，命令行操作如下：

```
[root@jay-linux ~]# qemu-img convert ubuntu.vdi -O qcow2 ubuntu.qcow2
[root@jay-linux ~]#qemu-system-x86_64 -m 1024 ubuntu.qcow2
```

5.11.5 从物理机迁移到KVM虚拟化环境（P2V）

virt-p2v由两部分组成：包含在virt-v2v软件包中的服务器端，可启动的ISO作为virt-p2v的客户端。使用virt-p2v工具的方法将物理机迁移到KVM虚拟化环境中，需要经过如下几个步骤：

1）在服务器端（一个KVM宿主机）安装virt-v2v、libvirt等软件，打开该宿主机的root用户SSH登录权限。

2）在服务器端，修改/etc/virt-v2v.conf配置文件，让其有类似如下的示例配置：

```
<virt-v2v>
  <!-- Target profiles -->

  <profile name="libvirt">
    <method>libvirt</method>
    <storage>default</storage>
    <network type="default">
      <network type="network" name="default"/>
    </network>


    </profile>
  </virt-v2v>
```

3）制作virt-p2v可以启动的设备。如果是Redhat的客户，可以到RHN中去下载virt-p2v的ISO镜像文件（如rhel-6.3-p2v.iso），然后将其烧录到一个光盘或U盘，使其作为物理机的启动设备。当然，可以下载virt-v2v的源代码，然后编译、制作ISO镜像文件。

4）在待迁移的物理机上，选择前一步中制作的virt-p2v启动介质（光盘或U盘）来启动系统。

5）在virt-p2v客户端启动后，根据其界面上的提示，填写前面准备的服务器端的IP或主机名、登录用户等信息，在连接上服务器端后，可以进一步填写转移后的虚拟客户机的名称、vCPU数量、内存大小等信息，最后点击"convert"（转化）按钮即可。virt-p2v客户端会

将物理机中的磁盘信息通过网络传输到服务器端，待传输完成后，选择关闭该物理机即可。

6）在virt-p2v服务器端（KVM宿主机）通过libvirt或qemu-kvm命令行启动前面转移过来的客户机即可。

因为使用virt-p2v工具进行转换的步骤还比较复杂，获得可启动的ISO文件可能还需要Redhat的授权，且virt-p2v并不太成熟可能导致迁移不成功，所以使用KVM的普通用户在实际环境中使用virt-p2v工具的情况还不多。

将物理机转化为KVM客户机，是一个公司或个人实施KVM虚拟化的基本过程。可以很简单地完成这个过程：安装一个和物理机上面相同系统的客户机，然后将物理机磁盘中的内容完全复制到对应客户机中即可。或者，更简单地，将物理机的磁盘物理上放到KVM宿主机中，直接使用该磁盘作为客户机磁盘启动即可（QEMU/KVM支持客户机使用一个完整的物理磁盘）。只是需要注意根据自己的需求来使用相应的qemu-kvm命令行参数来启动客户机，或者配置libvirt使用的相应客户机的XML配置文件。

5.12 本章小结

本章介绍了QEMU/KVM提供的各种各样的相对比较高级的功能。尽管QEMU/KVM是使用全虚拟化的工作方式，但是为了I/O设备性能或时钟准确度方面的考虑，也提供了让客户机使用一些半虚拟化驱动的方式，包括virtio_net、virtio_blk、kvm_clock等驱动。为了达到非常好的、接近原生设备的性能，可以考虑使用VT-d、SR-IOV等直接分配设备的功能，同时，可以对设备进行动态的热插拔，这样使用起来更加灵活。QEMU/KVM还提供了动态迁移、嵌套虚拟化、内存大页、SMEP、CPU的各种新指令支持等各种特性。KSM（内核同页合并）特性可以节省宿主机中分配给qemu-kvm进程使用的总内存。关于KVM安全方面的特性，可以有SMEP这个CPU特性的支持，还有cgroups用于控制各个qemu-kvm进程占用的资源，SELinux和sVirt保证了KVM虚拟化环境中对资源访问的安全，基于Intel TXT技术的Tboot可以确保宿主机系统的启动是安全可信的。qemu-kvm命令行中有许许多多的参数提供了各种各样的功能，QEMU monitor中也有很多的命令提供了非常有用的功能，本章也介绍其中一些常用功能的使用方法。本章最后还介绍如何从Xen、VMware、VirutalBox等Hypervisor迁移到KVM虚拟化方案时可以选择的工具和方法。

通过本章的详细介绍并结合了部分操作的命令行示例，相信读者可以对QEMU/KVM提供的许多（高级）功能都能了解其基本原理和操作步骤，以便可以在实际的生产环境中使用这些功能，也可以更好地规划KVM虚拟化环境的部署方案。

5.13 注释和参考阅读

1.本章注释

[1]Lguest是x86 32位Linux上一个简单的Hypervisor，它允许通过加载"lg"模块从而在同一个32位内核上运行多个副本。目前，该项目由Rusty Russell维护。它不依赖于硬件虚拟化技术（如Intel VT）的支持，其64位版本的移植已经由Redhat的一些工程师完成。可通过如下网站了解更多Lguest的信息：http://lguest.ozlabs.org/。

[2]QEMU中virtio相关的代码在qemu-kvm.git/hw/目录下，有一些带有"virtio"关键字作为文件名的C程序文件或.h头文件。

[3]Redhat文档：Red Hat Enterprise Linux 6Virtualization Host Configuration and Guest Installation Guide中的"10.4.Using the para-virtualized drivers with Red Hat Enterprise Linux 3.9 guests"，该文档目前对任何用户都开放访问权限，通过Google搜索一下即可访问到该文档。

[4]TSO（TCP segmentation offload），是一种利用网卡分割大数据包减小CPU负荷的一种技术，也被叫做LSO（Large segment offload），如果数据包的类型只是TCP，则称为TSO，如果硬件支持TSO功能，也需要同时支持硬件的TCP校验计算和分散-聚集（Scatter

Gather）功能。TSO使得网络协议栈能够将大块buffer推送至网卡，然后网卡执行分片工作，这样减轻了CPU的负荷，但TSO需要硬件来实现分片功能；而性能上的提高，主要是因为延缓分片而减轻了CPU的负载，所以可以考虑将TSO技术一般化，由于其本质是延缓分片，因此这种技术在Linux中被叫做GSO（Generic Segmentation Offload），它比TSO更通用，原因在于它不需要硬件的支持分片就可使用，对于支持TSO功能的硬件，则先经过GSO功能，然后使用网卡的硬件分片能力执行分片；而对于不支持TSO功能的网卡，将分片的执行放在了将数据推送至网卡的前一刻，也就是在调用驱动的xmit函数前。

[5]MSI（Message Signaled Interrupts）是触发中断的一种方式，它不同于传统的设备通过一个中断引脚（硬件）来产生中断，MSI中断方式是设备通过写少量特定的信息到一个特别的内存地址从而模拟产生一个中断注入到CPU。从PCI 2.2规范开始（包括之后的PCI-E规范）定义了MSI，而MSI-X是从PCI 3.0规范开始定义的。MSI-X允许一个设备分配最多2048个中断，比MSI中最多32个中断要多得多。在Linux中，可以通过"lspci-vv-s$BDF"命令查看某个设备的PCI配置空间信息，在其中的"Capabilities"的项目中可能会看到它支持MSI、MSI-X等中断类型。

[6]Race Condition是这样一种状态，在多线程（或多进程）情况下，对有些（只应该原子操作的）共享资源进行混乱操作而产生冲突

（collision），导致整个处理过程变得混乱，从而导致计算结果不正确或不精确。

[7]http://www.pcidatabase.com/网站提供了在线查询PCI信息的数据库，可以查询PCI设备的vendor ID和device ID。

[8]PCI-SIG（Peripheral Component Interconnect Special Interest Group），一个致力于定义PCI、PCI-X、PCI Express（PCIe）规范的电子工业协会，是成立于1992年的一个非营利性组织。其委员会理事代表来自于Intel、Microsoft、IBM、AMD、HP、Broadcom、Agilent和NVIDIA等大型IT公司。

[9]SeaBIOS是16位x86架构上BIOS的一种开源实现，它既能够在模拟器上运行，也能够在coreboot的帮助下在真实的x86硬件上运行。在SeaBIOS上可以运行GRUB、LILO等启动加载器，也支持Windows、Linux、FreeBSD等系统的运行。QEMU/KVM就是使用SeaBIOS作为客户机的默认BIOS。SeaBIOS官方网站为：www.seabios.org。

[10]笔者曾给QEMU社区报过一个CPU热插拔的Bug，如下：https://bugs.launchpad.net/qemu/+bug/1019179。

[11]Zap系统，可以参考哥伦比亚大学的网页（http://systems.cs.columbia.edu/projects/zap/），该系统也很长时间没有更新了。

[12]NX bit(Never eXecute)是CPU中的一种技术，用于在内存区域中对指令的存储和数据的存储进行标志以便区分。由于NX位技术的支持，操作系统可以将特定的内存区域标志为不可执行，处理器就不会执行该区域中的任何代码。这种技术在理论上可以防止“缓冲区溢出”（buffer overflow）类型的黑客攻击。在Intel处理器上被称为"XD Bit"（eXecute Disable），在AMD中被称为EVP（Enhanced Virus Protection），在ARM中被称为"XN"（eXecute Never）。目前主流的操作系统（如Windows、Linux、Mac OS等）都有对NX位这个技术的支持。

[13]KVM Forum是KVM社区每年一届的技术盛会，一般会讨论KVM中的最新功能和技术、未来发展方向等，http://www.linux-kvm.org/page/Kvm_Forum_2010网页是关于"KVM Forum 2010"的，其中有那次大会的PPT可以下载。

[14]GIMPS项目（Great Internet Mersenne Prime Search），可维基百科中的如下链接：

http://en.wikipedia.org/wiki/GIMPS

http://en.wikipedia.org/wiki/Prime95

http://en.wikipedia.org/wiki/Mersenne_prime

[15]TCG（Trusted Computing Group，可信计算组织）是一个制定国际工业标准的组织，一个非营利性组织，它的主要工作是基于硬件辅助的可信任根、为可信计算平台而开发、定义和推广一些开放的、对设备商中立的、全球工业标准。它曾经制定和推广了TPM（可信平台模块）标准的1.2版本。包括Intel、AMD、IBM、惠普、微软、甲骨文、ARM、高通、三星、联想等许多IT业界的大公司都是TCG的成员，它们一起参与制定可信计算相关的标准。

[16]Mercurial，和Git、SVN等类似，是一个跨平台的分布式软件代码管理工具。它主要是用Python语言开发的（也有小部分代码是C语言实现的），可以在Windows、Linux、Mac OS X等主流操作系统上使用。Mercurial在Linux上的命令行操作工具是hg程序，所以读者会看到本节示例代码中有"hg clone"命令。有一些著名的开源软件项目，如Mozilla、Python、OpenJDK、NetBeans等，是使用Mercurial工具进行源代码管理的，当然这里介绍的Tboot也使用Mercurial来管理tboot开发中的源代码。

[17]Intel TXT的介绍，哪些平台支持TXT，以及下载平台对应的SINIT AC模块，可参考如下链接：

http://software.intel.com/en-us/articles/intel-trusted-execution-technology/

[18]美国国家安全局的一份关于Linux系统安全配置的文档，名为"Guide to the Secure Configuration ofRed Hat Enterprise Linux 5"，可以通过如下链接下载：

www.nsa.gov/ia/_files/os/redhat/rhel5-guide-i731.pdf

[19]UUID（universally unique identifier，通用唯一标识符）是由开源软件基金会(Open Software Foundation,OSF)组织标准化的一个软件建构标准，通常应用在分布式计算环境（Distributed Computing Environment,DCE）中。UUID的目的是让分布式系统中的所有元素都能有唯一的辨识信息，而不需要透过中央控制端来做辨识信息的指定。UUID目前已经被软件业界广泛使用，其中包括：微软的Globally Unique Identifiers(GUIDs)、Linux上的ext2/ext3/ext4等文件系统、LUKS加密分割区、GNOME、KDE、Mac OS X等。标准的UUID由128位（32个十六进制数字）组成，格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx（8-4-4-4-12）。

[20]Redhat的关于RHEL 6.x（如RHEL 6.3）中virt-v2v工具的官方文档，可参考如下网页：

https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html-single/V2V_Guide/index.html

2.参考阅读

（1）关于KVM虚拟化中的Windows客户机驱动，可参考：

http://www.linux-kvm.org/page/WindowsGuestDrivers，其中提供了较为详细的文档。

（2）关于OS中的计时器（timer）的更多信息，可以参考如下链接了解：

❑APIC timer：http://wiki.osdev.org/APIC_timer

❑Programmable Interval Timer(PIT)：http://wiki.osdev.org/PIT

（3）IBM的郭晋兵和吴超凤两位工程师曾写过一篇文章“虚拟机迁移技术漫谈”，链接如下：

http://www.ibm.com/developerworks/cn/linux/l-cn-mgrtvml/index.html

（4）Intel AVX官方网址：

http://software.intel.com/en-us/avx/

关于AVX指令集的效果演示介绍，可以参考北京的Intel IDF（Intel Developer Forum）

2010大会上的一个视频（在视频的33到36分钟之间），其链接如下：

http://intelstudios.edgesuite.net/idf/2010/bj/keynote/100413_DP_CN/f.htm

（5）Intel的AES-NI新指令集的白皮书，可以到如下链接下载：

http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set

（6）关于cgroups的详细介绍，可以参考Linux内核代码中的文档，如下：

http://lxr.linux.no/#linux+v3.6/Documentation/cgroups/cgroups.txt

也可以参考Redhat的文档，链接如下：

https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html

（7）LWN.net网站上关于内存大页的介绍，可以参考如下3个链接：

❑http://lwn.net/Articles/374424/

❑http://lwn.net/Articles/375096/

❑http://lwn.net/Articles/376606/

（8）Linux内核代码中关于透明大页的文档，见如下网页：

http://lxr.linux.no/#linux+v3.6/Documentation/vm/transhuge.txt

（9）Intel官方关于TXT技术的软件开发指导手册，见如下网页：

http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html

第6章　KVM管理工具

通过第4章和第5章的介绍，相信读者已经基本理解了KVM的各个功能、简单原理和配置方法。在前面的所有章节中，几乎所有的示例都是直接用qemu-kvm命令行工具（qemu-system-x86_64）来配置和启动客户机的，其中有各种各样的配置参数，这些参数对于新手来说是很难记忆和熟练配置。本章将向大家介绍能够更加方便地配置和使用KVM的一些管理工具，它们一般都对qemu-kvm命令进行了封装和功能增强，从而提供了比原生的qemu-kvm命令行更加友好、更加高效的用户交互接口。

6.1　libvirt

6.1.1　libvirt简介

提到KVM的管理工具，首先不得不介绍的就是大名鼎鼎的libvirt，因为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，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一定要选择开源和遵守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常大的。

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ApacheQpid）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可以是完全透明的，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。

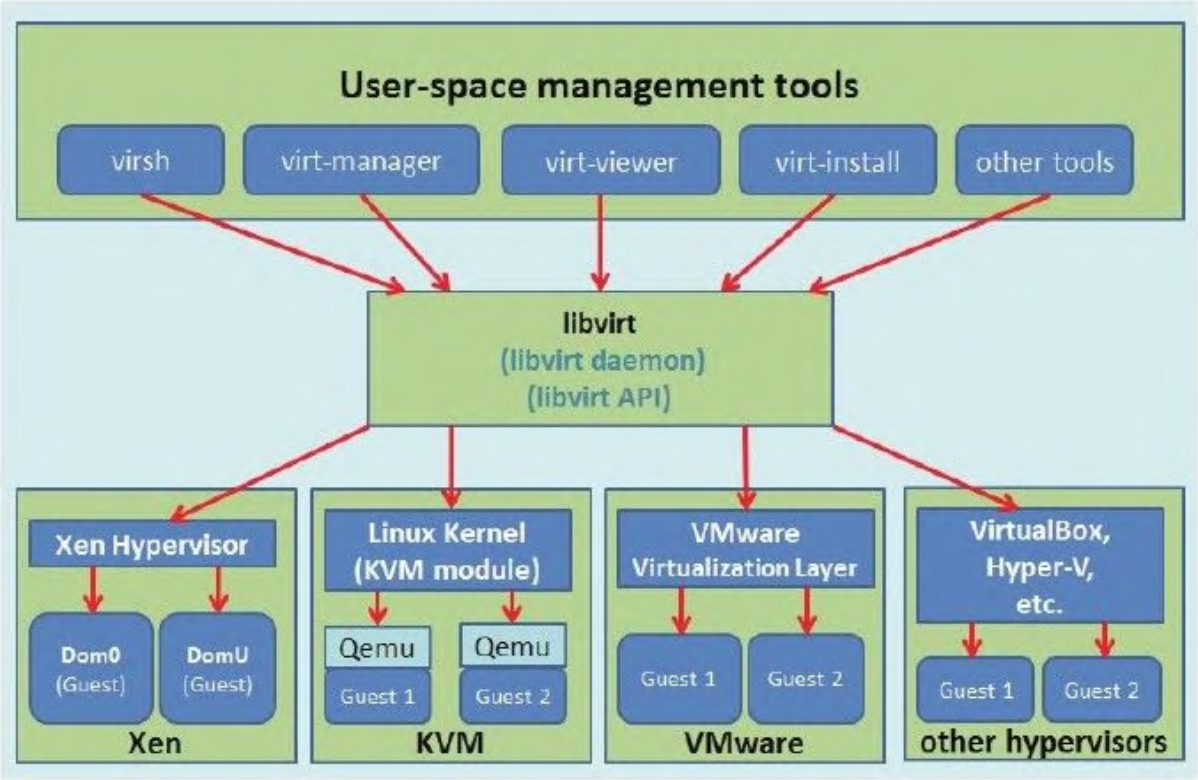


图　6-1　虚拟机管理工具通过libvirt管理各种类型的虚拟机

在libvirt中涉及几个重要的概念，解释如下：

❑节点（Node）是一个物理机器，上面可能运行着多个虚拟客户机。Hypervisor和Domain都运行在节点之上。

❑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）。

❑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云计算服务中的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。

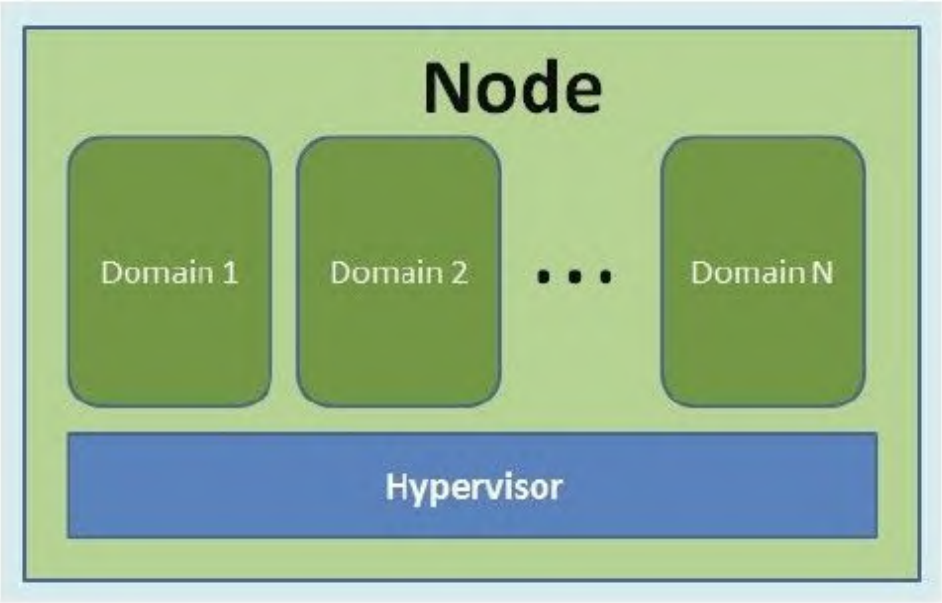


图　6-2　节点、Hypervisor和域之间的关系

在了解了节点、Hypervisor和域的概念之后，用一句话概括libvirt的目标，那就是：为了安全高效地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。假设使用了最简单的SSH，不需要额外配置工作，比如，在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。当然在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，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）。应用程序接口是为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，可以完成所有支持的管理工作；一种是普通用户权限的libvirtd，只能完成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。

6.1.2 libvirt的编译、安装和配置

1.从源代码编译和安装libvirt

如果只是普通用户使用libvirt，一般不需要从源码编译libvirt，只需要安装对应的Linux系统上libvirt软件包即可。一些高级用户或开发者，可能希望对libvirt进行更多的了解，甚至通过修改libvirt的源代码来实现自己的功能，因此还是需要了解从源代码编译和安装libvirt的过程。

下面以在一个RHEL 6.3系统上编译libvirt 1.0.0版本为例，介绍一下libvirt的编译和安装过程。

（1）检查和清理安装的libvirt

在真正开始编译之前，需要检查和清理系统上已经安装的libvirt（如果曾经安装过），命令行操作如下：

```
[root@jay-linux kvm_demo]# which libvirtd
/usr/sbin/libvirtd
[root@jay-linux kvm_demo]# rpm -q libvirt
libvirt-0.9.10-21.el6.x86_64
[root@jay-linux kvm_demo]# rpm -e libvirt
[root@jay-linux kvm_demo]# libvirtd --version
-bash: /usr/sbin/libvirtd: No such file or directory
```

（2）下载libvirt的源代码

可以选择从Linux系统发行版的源代码ISO中获取其libvirt的源代码安装包，也可以选择到libvirt官方网站下载libvirt源代码的

tar.gz压缩包，还可以通过git工具将开发中的libvirt源码仓库克隆到本地。

libvirt官方发布源代码的网页是：http://libvirt.org/sources/。该网页上提供了最原生的libvirt各个版本的源代码的tar.gz压缩包，以及已经编译好的libvirt-devel、libvirt-python、libvirt-java、libvirt-php等RPM包。libvirt官方还提供了下载源代码的FTP站点：ftp://libvirt.org/libvirt/。下载libvirt-1.0.0.tar.gz源码包并将其解压缩的命令行如下：

```
[root@jay-linux kvm_demo]# wget \
http://libvirt.org/sources/libvirt-1.0.0.tar.gz

--2012-11-25 22:58:24--  http://libvirt.org/sources/libvirt-1.0.0.tar.gz
HTTP request sent, awaiting response... 200 OK
Length: 21868950 (21M) [application/x-gzip]
Saving to: "libvirt-1.0.0.tar.gz"

100%[=======================>] 21,868,950  67.6K/s   in 2m 29s

2012-11-25 23:00:54 (143 KB/s) - "libvirt-1.0.0.tar.gz" saved
[21868950/21868950]
[root@jay-linux kvm_demo]# tar -zxf libvirt-1.0.0.tar.gz
```

libvirt处于开发中的最新的git代码仓库的地址为：git://libvirt.org/libvirt.git。还可以通过网页http://libvirt.org/git/以在线网页的方式浏览libvirt.git、libvirt-java.git、libvirt-php.git等开发中的代码仓库。另外，libvirt的python绑定的代码就存放在libvirt的源码仓库

（libvirt.git）中，没有单独的libvirt-python代码仓库。下载libvirt开发代码仓库的命令行如下：

```
[root@jay-linux kvm_demo]# git clone git://libvirt.org/libvirt.git
Initialized empty Git repository in /root/kvm_demo/libvirt/.git/
remote: Counting objects: 96578, done.
remote: Compressing objects: 100% (15214/15214), done.
Receiving objects:  52% (50926/96578), 51.28 MiB | 26 KiB/s
Receiving objects:  52% (50927/96578), 51.31 MiB | 24 KiB/s
remote: Total 96578 (delta 80940), reused 96578 (delta 80940)
Receiving objects: 100% (96578/96578), 91.22 MiB | 63 KiB/s, done.
Resolving deltas: 100% (80940/80940), done.
```

（3）配置和编译libvirt

配置和编译libvirt的方法与Linux上多数的开源项目的方法类似，都是先运行configure脚本进行编译环境配置的，然后用make命令进行编译，用make install命令进行安装。

查看有哪些配置选项可用的命令为"./configure--help"，命令行操作如下：

```
[root@jay-linux kvm_demo]# cd libvirt-1.0.0
[root@jay-linux libvirt-1.0.0]# ./configure --help
'configure' configures libvirt 1.0.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print 'checking ...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for '--cache-file=config.cache'
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or '..']

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX

                              [PREFIX]

  By default, 'make install' will install all the files in
  '/usr/local/bin', '/usr/local/lib' etc.  You can specify
  an installation prefix other than `/usr/local' using '--prefix',
  for instance '--prefix=$HOME'.

  <!-- 此处省略配置 libvirt 的数百行帮助选项信息 -->
```

根据上面的配置帮助信息可知，如果不用"--prefix=PREFIX"参数指定自定义的安装路径，那么libvirt相关的文件默认都会被安装到/usr/local/bin和/usr/local/lib等目录中。

配置libvirt编译环境的命令为"./configure"，命令行操作如下：

```
[root@jay-linux libvirt-1.0.0]# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking how to create a ustar tar archive... gnutar
<!-- 此处省略配置过程中的数百行输出信息 -->
configure:      Readline: yes
configure:        Python: yes
configure:        DTrace: no
configure:         numad: no
configure:   XML Catalog: /etc/xml/catalog
configure:   Init script: redhat
configure: Console locks: /var/lock
configure:
configure: Privileges
configure:
configure:       QEMU: root:root
configure:
```

在配置过程中，可能会由于缺少编译时需要依赖的软件包而导致配置失败，这时只需要按照提示安装对应的软件包，然后重新运行"./configure"命令配置即可。在默认情况下，libvirt会配置QEMU/KVM、vmware的驱动支持（如果能找到相关依赖库程序），也会配置libvirtd和virsh等，还会默认配置libvirt对python的绑定。对于Xen、Hyper-V等的支持，配置程序会自动检查当前系统是否含有与这些Hypervisor相关的程序，如果检测成功，就会编译对应的驱动。

真正编译libvirt的命令为"make"，命令行操作如下：

```
[root@jay-linux libvirt-1.0.0]# make -j 4
make  all-recursive
make[1]: Entering directory '/root/kvm_demo/libvirt-1.0.0'
Making all in gnulib/lib

 make[2]: Entering directory '/root/kvm_demo/libvirt-1.0.0/gnulib/lib'
   GEN    alloca.h
   GEN    c++defs.h
   GEN    warn-on-use.h
 <!-- 此处省略编译过程中的数百行输出信息 -->
 Making all in examples/systemtap
 make[2]: Entering directory '/root/kvm_demo/libvirt-1.0.0/examples/systemtap'
 make[2]: Nothing to be done for 'all'.
 make[2]: Leaving directory '/root/kvm_demo/libvirt-1.0.0/examples/systemtap'
 make[2]: Entering directory '/root/kvm_demo/libvirt-1.0.0'
 make[2]: Leaving directory '/root/kvm_demo/libvirt-1.0.0'
 make[1]: Leaving directory '/root/kvm_demo/libvirt-1.0.0'
```

（4）安装libvirt

在配置和编译时都不需要超级用户（root）权限，但是在安装libvirt时一般都需要root用户权限。执行make install命令即可完成libvirt安装，命令行操作如下：

```
[root@jay-linux libvirt-1.0.0]# make install
Making install in gnulib/lib
make[1]: Entering directory '/root/kvm_demo/libvirt-1.0.0/gnulib/lib'
make  install-am
make[2]: Entering directory '/root/kvm_demo/libvirt-1.0.0/gnulib/lib'
make[3]: Entering directory '/root/kvm_demo/libvirt-1.0.0/gnulib/lib'
if test yes = no; then \
          case 'linux-gnu' in \
            darwin[56]*) \
              need_charset_alias=true ;; \
<!-- 此处省略安装过程中的数百行输出信息 -->
make[2]: Entering directory '/root/kvm_demo/libvirt-1.0.0'
make[2]: Nothing to be done for 'install-exec-am'.
 /bin/mkdir -p '/usr/local/lib/pkgconfig'
 /usr/bin/install -c -m 644 libvirt.pc '/usr/local/lib/pkgconfig'
make[2]: Leaving directory '/root/kvm_demo/libvirt-1.0.0'
make[1]: Leaving directory '/root/kvm_demo/libvirt-1.0.0'
```

（5）检查已经安装的libvirt

libvirt的安装会为系统安装libvirtd、virsh等可执行程序，也会安装libvirt的API程序库，还会安装对python的绑定，检查这些安装后的文件，命令行操作如下：

```
[root@jay-linux libvirt-1.0.0]# which libvirtd
/usr/local/sbin/libvirtd
[root@jay-linux libvirt-1.0.0]# libvirtd --version
libvirtd (libvirt) 1.0.0
[root@jay-linux libvirt-1.0.0]# which virsh
/usr/local/bin/virsh
```

```
[root@jay-linux libvirt-1.0.0]# virsh --version
1.0.0

[root@jay-linux libvirt-1.0.0]# ls /usr/local/include/libvirt/
libvirt.h  libvirt-qemu.h  virterror.h

[root@jay-linux libvirt-1.0.0]# ls /usr/local/lib/libvirt*
/usr/local/lib/libvirt.a        /usr/local/lib/libvirt-qemu.so
/usr/local/lib/libvirt.so.0
/usr/local/lib/libvirt.la       /usr/local/lib/libvirt-qemu.so.0
/usr/local/lib/libvirt.so.0.1000.0
/usr/local/lib/libvirt-qemu.a   /usr/local/lib/libvirt-qemu.so.0.1000.0
/usr/local/lib/libvirt-qemu.la  /usr/local/lib/libvirt.so

/usr/local/lib/libvirt:
connection-driver

[root@jay-linux libvirt-1.0.0]# ls
/usr/local/lib64/python2.6/site-packages/*virt*
/usr/local/lib64/python2.6/site-packages/libvirtmod.la
/usr/local/lib64/python2.6/site-packages/libvirtmod.so
/usr/local/lib64/python2.6/site-packages/libvirtmod_qemu.la
/usr/local/lib64/python2.6/site-packages/libvirt.py
/usr/local/lib64/python2.6/site-packages/libvirtmod_qemu.so
/usr/local/lib64/python2.6/site-packages/libvirt_qemu.py
```

如果安装后立即使用libvirt这些程序库，会遇到找不到对应库文件的错误提示，这时可能需要运行ldconfig等工具来更新刚才安装的共享库。

（6）从libvirt的git代码仓库编译libvirt

从git源代码仓库编译和安装libvirt，与从libvirt的源码tar.gz包编译和安装的过程是完全类似的。这里只介绍一些在编译前进行配置（configure）时的一些不同之处。在使用libvirt.git源码仓库配置时，先运行其自带的autogen.sh这个脚本，它会默认会先下载

git://git.sv.gnu.org/gnulib.git，然后根据模板生成configure配置脚本和初始化一些Makefile文件，最后自动运行configure文件进行对编译环境的配置。

从git源码仓库安装libvirt的基本操作命令行如下（省略命令执行的输出信息）：

```
[root@jay-linux kvm_demo]# cd libvirt.git
[root@jay-linux libvirt.git]#./autogen.sh
[root@jay-linux libvirt.git]# make
[root@jay-linux libvirt.git]# make install
```

在默认状态下，配置和编译后安装的目录与Linux操作系统发行版提供的默认目录可能不一致，例如，在RHEL 6.3中通过系统的RPM包安装的libvirtd、virsh等可执行程序被安装在/usr/sbin/目录下，libvirt.so、libvirt-qemu.so等共享库文件被安装在/usr/lib64/目录下，而从前面步骤（5）可知，在编译安装时默认会将libvirtd、virsh等安装在/usr/local/sbin目录下，而libvirt.so、libvirt-qemu.so等被安装在/usr/local/lib/目录下。如果想保持对操作系统发行版中安装的可执行程序和共享库的目录的一致性，autogen.sh脚本提供了"--system"参数，通过这个参数来配置，就会尽可能保证安装目录与原生系统的一致性，其命令行操作如下（省略了部分命令执行的输出信息）：

```
[root@jay-linux libvirt.git]# ./autogen.sh --system
[root@jay-linux libvirt.git]# make
[root@jay-linux libvirt.git]# make install
[root@jay-linux libvirt.git]# which libvirtd
/usr/sbin/libvirtd
[root@jay-linux libvirt.git]# which virsh
/usr/bin/virsh
[root@jay-linux libvirt.git]# ls /usr/lib64/libvirt*
/usr/lib64/libvirt.a        /usr/lib64/libvirt-qemu.so
/usr/lib64/libvirt.so
/usr/lib64/libvirt.la       /usr/lib64/libvirt-qemu.so.0
/usr/lib64/libvirt.so.0
/usr/lib64/libvirt-qemu.a   /usr/lib64/libvirt-qemu.so.0.1000.0
/usr/lib64/libvirt.so.0.1000.0
/usr/lib64/libvirt-qemu.la  /usr/lib64/libvirt-qemu.so.0.9.10
/usr/lib64/libvirt.so.0.9.10

/usr/lib64/libvirt:
connection-driver
```

2.用软件包安装libvirt

很多流行的Linux发行版（如RHEL 6.x、Fedora 17、Ubuntu 12.10等）都提供了libvirt相关的软件包，按照安装普通软件包的方式安装libvirt相关的软件包即可。在笔者当前使用的RHEL 6.3中可以使用yum或rpm工具来安装对应的RPM包。查看某系统中已经安装的libvirt相关的RPM包，命令行如下：

```
[root@rhel6u3-ga ~]# rpm -qa | grep libvirt
libvirt-python-0.9.10-21.el6.x86_64
libvirt-java-javadoc-0.4.7-1.el6.noarch
libvirt-qmf-0.3.0-6.el6.x86_64
libvirt-client-0.9.10-21.el6.x86_64
libvirt-lock-sanlock-0.9.10-21.el6.x86_64
libvirt-java-devel-0.4.7-1.el6.noarch
libvirt-0.9.10-21.el6.x86_64
libvirt-snmp-0.0.2-3.el6.x86_64
libvirt-java-0.4.7-1.el6.noarch
libvirt-devel-0.9.10-21.el6.x86_64
libvirt-cim-0.6.1-3.el6.x86_64
```

当然，RHEL 6.3默认采用QEMU/KVM的虚拟化方案，所以应该安装QEMU相关的软件包，查看这些软件包的命令行操作如下：

```
[root@rhel6u3-ga ~]# rpm -qa | grep qemu
qemu-img-0.12.1.2-2.295.el6.x86_64
qemu-kvm-0.12.1.2-2.295.el6.x86_64
gpxe-roms-qemu-0.9.7-6.9.el6.noarch
```

由于libvirt是跨平台的，而且还支持微软公司的Hyper-V虚拟化，所以在Windows上也可以安装libvirt，甚至可以编译libvirt。可以到libvirt官方的网页（http://libvirt.org/sources/win32_experimental/）中查看和下载能在Windows上运行的libvirt安装程序。不过，由于libvirt主要还

是基于Linux开发的，而且支持它的公司（Redhat等）中的开发者和个人开发者大多数都是Linux程序员，故libvirt的Windows版本还是处于开发中的，开发进度并不如Linux上libvirt开发得快，前面提到的能下载的libvirt Windows版也是实验性的版本，而不是正式产品的发行版，其功能并不是非常完善。

6.1.3 libvirt和libvirtd的配置

1.libvirt的配置文件

以RHEL 6.3为例，libvirt相关的配置文件都在/etc/libvirt/目录之中，如下：

```
[root@rhel6u3-ga libvirt]# cd /etc/libvirt/
[root@rhel6u3-ga libvirt]# ls
cim  libvirt.conf  libvirtd.conf  lxc.conf  nwfilter  qemu
qemu.conf  qemu-sanlock.conf  storage
[root@rhel6u3-ga libvirt]# cd qemu
[root@rhel6u3-ga qemu]# ls
networks  rhel6u3-1.xml  rhel6u3-2.xml
```

下面简单介绍其中几个重要的配置文件和目录。

（1）/etc/libvirt/libvirt.conf

libvirt.conf文件用于配置一些常用libvirt连接（通常是远程连接）的别名，和Linux中的普通配置文件一样，在该配置文件中以井号

（#）开头的行是注释，如下：

```
[root@rhel6u3-ga kvm_demo]# cat /etc/libvirt/libvirt.conf
#
# This can be used to setup URI aliases for frequently
# used connection URIs. Aliases may contain only the
# characters  a-Z, 0-9, _, -.
#
# Following the '=' may be any valid libvirt connection
# URI, including arbitrary parameters

uri_aliases = [
  "remote1=qemu+ssh://root@192.168.93.201/system",
]
```

其中，配置了remote1这个别名用于指代qemu+ssh://root@192.168.93.201/system这个远程的libvirt连接，有这个别名后，就可以在用virsh等工具或自己写代码调用libvirt API时使用这个别名，而不需要写完整的、冗长的URI连接标识了。用virsh使用这个别名，连接到远程的libvirt上查询当前已经启动的客户机状态，然后退出连接，命令行操作如下：

```
[root@rhel6u3-ga kvm_demo]# service libvirtd reload
Reloading libvirtd configuration:                          [  OK  ]
[root@rhel6u3-ga kvm_demo]# virsh -c remote1
root@192.168.93.201's password:
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # list
 Id    Name                           State
----------------------------------------------------
 1     rhel6u3-remote                 running

virsh # quit

[root@rhel6u3-ga kvm_demo]#
```

在代码中调用libvirt API时也可以使用这个别名来建立连接，如下的python代码行就实现了使用这个别名来建立连接。

```
conn = libvirt.openReadOnly('remote1')
```

（2）/etc/libvirt/libvirtd.conf

libvirtd.conf是libvirt的守护进程libvirtd的配置文件，被修改后需要让libvirtd重新加载配置文件（或重启libvirtd）才会生效。在libvirtd.conf文件中，用井号（#）开头的行是注释内容，真正有用的配置在文件的每一行中使用“配置项=值”（如tcp_port="16509"）这样配对格式来设置。在libvirtd.conf中配置了libvirtd启动时的许多设置，包括是否建立TCP、UNIX domain socket等连接方式及其最大连接数，以及这些连接的认证机制，等等。

例如，下面的几个配置项表示关闭TLS安全认证的连接（默认值是打开的）、打开TCP连接（默认是关闭TCP连接的），设置TCP监听的端口，TCP连接不使用认证授权方式，设置UNIX domain socket的保存目录等。

```
listen_tls = 0
listen_tcp = 1
tcp_port = "16666"
unix_sock_dir = "/var/run/libvirt"
auth_tcp = "none"
```

注意　要让TCP、TLS等连接生效，需要在启动libvirtd时加上--listen参数（简写为-l）。而默认的service libvirtd start命令在启动libvirtd服务时并没带--listen参数，所以如果要使用TCP等连接方式，可以使用libvirtd--listen-d命令来启动libvirtd。

以上配置选项实现将UNIX socket放到/var/run/libvirt目录下，启动libvirtd并检验配置是否生效，命令行操作如下：

```
[root@rhel6u3-ga ~]# libvirtd --listen -d
2012-12-01 07:54:02.715+0000: 6788: info : libvirt version: 0.9.10,
package: 21.el6 (Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>,
2012-05-23-09:15:11, x86-003.build.bos.redhat.com)
[root@rhel6u3-ga ~]# virsh -c qemu+tcp://localhost:16666/system
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # quit

[root@rhel6u3-ga ~]# ls /var/run/libvirt/libvirt-sock*
/var/run/libvirt/libvirt-sock  /var/run/libvirt/libvirt-sock-ro
```

（3）/etc/libvirt/qemu.conf

qemu.conf是libvirt对QEMU的驱动的配置文件，包括VNC、SPICE等和连接它们时采用的权限认证方式的配置，也包括内存大页、SELinux、Cgroups等相关配置。

（4）/etc/libvirt/qemu/目录

在qemu目录下存放的是使用QEMU驱动的域的配置文件，查看qemu目录如下：

```
[root@rhel6u3-ga ~]# ls /etc/libvirt/qemu/
networks  rhel6u3-1.xml  rhel6u3-2.xml
```

其中包括了两个域的XML配置文件（rhel6u3-1.xml和rhel6u3-2.xml），这就是笔者用virt-manager工具创建的两个域，默认会将其

配置文件保存到/etc/libvirt/qemu/目录下。而其中的networks目录保存了创建一个域时默认使用的网络配置。

2.libvirtd的使用

libvirtd是一个作为libvirt虚拟化管理系统中的服务器端的守护程序，要让某个节点能够利用libvirt进行管理（无论是本地还是远程管理），都需要在这个节点上运行libvirtd这个守护进程，以便让其他上层管理工具可以连接到该节点，libvirtd负责执行其他管理工具发送给它的虚拟化管理操作指令。而libvirt的客户端工具（包括virsh、virt-manager等）可以连接到本地或远程的libvirtd进程，以便管理节点上的客户机（启动、关闭、重启、迁移等）、收集节点上的宿主机和客户机的配置和资源使用状态。

在RHEL 6.3中libvirtd是作为一个服务（service）配置在系统中的，所以可以通过service命令来对其进行操作（实际是通过/etc/init.d/libvirtd服务脚本来实现的）。常用的操作方式有："service libvirtd start"命令表示启动libvirtd，"service libvirtd restart"表示重启libvirtd，"service libvirtd reload"表示不重启服务但重新加载配置文件（即/etc/libvirt/libvirtd.conf配置文件）。对libvirtd服务进行操作的命令行如下：

```
[root@rhel6u3-ga ~]# service libvirtd
Usage: /etc/init.d/libvirtd {start|stop|status|restart|condrestart
|reload|force-reload|try-restart}

[root@rhel6u3-ga ~]# service libvirtd start
Starting libvirtd daemon: 2012-12-01 08:14:33.048+0000: 31202: info :
libvirt version: 0.9.10, package: 21.el6 (Red Hat, Inc.
<http://bugzilla.redhat.com/bugzilla>, 2012-05-23-09:15:11,
x86-003.build.bos.redhat.com)
                                                           [  OK  ]

[root@rhel6u3-ga ~]# service libvirtd status
libvirtd (pid  31327) is running...

[root@rhel6u3-ga ~]# service libvirtd restart
Stopping libvirtd daemon:                                  [  OK  ]
Starting libvirtd daemon: 2012-12-01 08:14:43.522+0000: 31325: info :
libvirt version: 0.9.10, package: 21.el6 (Red Hat, Inc.
<http://bugzilla.redhat.com/bugzilla>, 2012-05-23-09:15:11,
x86-003.build.bos.redhat.com)
                                                           [  OK  ]
```

在默认情况下，libvirtd在监听一个本地的UNIX domain socket，而没有监听基于网络的TCP/IP socket，需要使用“-l或--listen”的命令行参数来开启对libvirtd.conf配置文件中TCP/IP socket的配置。另外，libvirtd守护进程的启动或停止，并不会直接影响正在运行中的客户机。libvirtd在启动或重启完成时，只要客户机的XML配置文件是存在的，libvirtd会自动加载这些客户的配置，获取它们的信息；当然，如果客户机没有基于libvirt格式的XML文件来运行，libvirtd则不能发现它。

libvirtd是一个可执行程序，不仅可以使用"service"命令调用它作为服务来运行，而且可以单独地运行libvirtd命令来使用它。下面

介绍以下几种libvirtd命令行的参数。

（1）-d或--daemon

表示让libvirtd作为守护进程（daemon）在后台运行。

（2）-f或--config FILE

指定libvirtd的配置文件为FILE，而不是使用默认值（通常是/etc/libvirt/libvirtd.conf）。

（3）-l或--listen

开启配置文件中配置的TCP/IP连接。

（4）-p或--pid-file FILE

将libvirtd进程的PID写入到FILE文件中，而不是使用默认值（通常是/var/run/libvirtd.pid）。

（5）-t或--timeout SECONDS

设置对libvirtd连接的超时时间为SECONDS秒。

（6）-v或--verbose

执行命令输出详细的输出信息。特别是在运行出错时，详细的输出信息便于用户查找原因。

（7）--version

显示libvirtd程序的版本信息。

关于libvirtd命令的使用，几个简单的命令行操作如下：

```
# 使用 libvirtd 命令前，先停止已有的服务
[root@rhel6u3-ga ~]# service libvirtd stop
Stopping libvirtd daemon:                                  [  OK  ]

[root@rhel6u3-ga ~]# libvirtd --version
libvirtd (libvirt) 0.9.10
[root@rhel6u3-ga ~]# libvirtd
2012-12-01 08:57:38.475+0000: 2196: info : libvirt version: 0.9.10,
package: 21.el6 (Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>,
2012-05-23-09:15:11, x86-003.build.bos.redhat.com)
^C     # 没有以 daemon 的形式启动，标准输出被 libvirtd 占用；这里用 Ctrl-C
结束掉 libvirtd 进程，以便继续进行后续操作

[root@rhel6u3-ga ~]# libvirtd -l -d  -p /root/libvirtd.pid
2012-12-01 08:56:14.665+0000: 7683: info : libvirt version: 0.9.10,
package: 21.el6 (Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>,
2012-05-23-09:15:11, x86-003.build.bos.redhat.com)
[root@rhel6u3-ga ~]# cat /root/libvirtd.pid
7685
```

6.1.4 libvirt域的XML配置文件

在使用libvirt对虚拟化系统进行管理时，很多地方都是XML文件作为配置文件，包括客户机（域）的配置、宿主机网络接口配置、网

络过滤、各个客户机的磁盘存储配置、磁盘加密、宿主机和客户机的CPU特性，等等。本节只针对客户机的XML进行较详细的介绍，因为客户机的配置是最基本的和最重要的，了解了它之后就可以使用libvirt管理客户机了。

1.客户机的XML配置文件格式的示例

在libvirt中，客户机（即域）的配置是采用XML格式来描述的。下面展示了笔者使用virt-manager创建的一个客户机的配置文件（即在6.1.3节中看到的rhel6u3-1.xml文件），后面几节将会分析其中的主要配置项目。

```
<!--
WARNING: THIS IS AN AUTO-GENERATED FILE. CHANGES TO IT ARE LIKELY TO BE
OVERWRITTEN AND LOST. Changes to this xml configuration should be made using:
  virsh edit rhel6u3-1
or other application using the libvirt API.
-->
```

```
<domain type='kvm'>
  <name>rhel6u3-1</name>
  <uuid>e39a1025-87ec-b4ab-b08c-c02b5cebb5d0</uuid>
  <memory unit='KiB'>1048576</memory>
<currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type arch='x86_64' machine='rhel6.3.0'>hvm</type>
    <boot dev='hd'/>
    <boot dev='cdrom'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/libexec/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none'/>
      <source file='/var/lib/libvirt/images/rhel6u3-1.img'/>
      <target dev='vda' bus='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
    </disk>
    <controller type='usb' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
    </controller>
    <controller type='ide' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <interface type='bridge'>
      <mac address='52:54:00:e9:e0:3b'/>
      <source bridge='br0'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
```

```
    <input type='tablet' bus='usb'/>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='-1' autoport='yes'/>
    <sound model='ich6'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
    </sound>
    <video>
      <model type='cirrus' vram='9216' heads='1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
    </video>
    <memballoon model='virtio'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
    </memballoon>
  </devices>
</domain>
```

由上面的配置文件示例可以看到，在该域的XML文件中所有有效配置都在＜domain＞和＜/domain＞标签之间，这表明该配置文件是一个域的配置。（XML文档中注释在两个特殊的标签之间，如＜!--注释--＞。）

通过libvirt启动客户机，经过文件解析和命令参数的转换，最终也会调用qemu-kvm命令行工具来实际完成客户机的创建。用这个XML配置文件启动的客户机，它的qemu-kvm命令行参数是非常详细非常冗长的一行，查询qemu-kvm命令行参数的操作如下：

```
[root@rhel6u3-ga ~]# ps -ef | grep qemu | grep rhel6u3-1
qemu      8597     1  3 11:10 ?        00:13:57 /usr/libexec/qemu-kvm -S
-M rhel6.3.0 -enable-kvm -m 1024 -smp 2,sockets=2,cores=1,threads=1
-name rhel6u3-1 -uuid e39a1025-87ec-b4ab-b08c-c02b5cebb5d0
-nodefconfig -nodefaults -chardev
socket,id=charmonitor,path=/var/lib/libvirt/qemu/rhel6u3-1.monitor,
server,nowait -mon chardev=charmonitor,id=monitor,mode=control -rtc
base=utc -no-shutdown -device
piix3-usb-uhci,id=usb,bus=pci.0,addr=0x1.0x2 -drive
file=/var/lib/libvirt/images/rhel6u3-new-clone.img,if=none,id=drive-
virtio-disk0,format=raw,cache=none -device virtio-blk-pci,scsi=off,
bus=pci.0,addr=0x5,drive=drive-virtio-disk0,id=virtio-disk0,bootindex=1
-netdev tap,fd=22,id=hostnet0,vhost=on,vhostfd=23 -device
virtio-net-pci,netdev=hostnet0,id=net0,mac=52:54:00:e9:e0:3b,
bus=pci.0,addr=0x3 -chardev pty,id=charserial0 -device isa-serial,
chardev=charserial0,id=serial0 -device usb-tablet,id=input0 -vnc 127.0.0.1:0
-vga cirrus -device intel-hda,id=sound0,bus=pci.0,addr=0x4 -device
hda-duplex,id=sound0-codec0,bus=sound0.0,cad=0 -device virtio-balloon-pci,
id=balloon0,bus=pci.0,addr=0x6
```

这里RHEL 6.3系统中的qemu-kvm工具为/usr/libexec/qemu-kvm，与第3、4、5章中用的从源代码编译和安装的qemu-system-x86_64工具是类似的，它们的参数也基本一致（当然如果两者版本差异较大，参数和功能可能有一些不同）。对于qemu-kvm命令的这么多的参数，在前面的第4章和第5章中已经对其中比较重要的大多数参数都进行了详细介绍。本节主要是针对域的XML配置文件进行介绍和分析。

2.CPU、内存、启动顺序等基本配置

（1）CPU的配置

在前面介绍的rhel6u3-1.xml配置文件中，关于CPU的配置为：

```
<vcpu placement='static'>2</vcpu>
<features>
  <acpi/>
  <apic/>
  <pae/>
</features>
```

vcpu标签，表示客户机中vCPU的个数，这里为两个。features标签，表示Hypervisor为客户机打开或关闭CPU或其他硬件的特性，这里打开了ACPI、APIC、PAE等特性。当然，CPU的特性是在该客户机的CPU模型中定义的，如RHEL 6.3中的qemu-kvm默认该客户机的CPU模型是cpu64-rhel6，该CPU模型中的特性（如SSE2、LM、NX、TSC等）也是该客户机使用的。

对CPU的分配，可以有更细粒度的配置，例如：

```
<domain>
  ...
  <vcpu placement='static' cpuset="1-4,^3,6" current="1">2</vcpu>
  ...
</domain>
```

cpuset表示允许到哪些物理CPU上执行，这里表示客户机的两个vCPU被允许调度到1、2、4、6号物理CPU上执行（^3表示排除3号），而current表示启动客户机时只给一个vCPU，最多可以增加到使用2个vCPU。

当然libvirt还提供cputune标签来对CPU分配进行更多调节，如下：

```
<domain>
  ...
  <cputune>
    <vcpupin vcpu="0" cpuset="1"/>
    <vcpupin vcpu="1" cpuset="2,3"/>
    <vcpupin vcpu="2" cpuset="4"/>
    <vcpupin vcpu="3" cpuset="5"/>
    <emulatorpin cpuset="1-3"/>
    <shares>2048</shares>
    <period>1000000</period>
    <quota>-1</quota>
    <emulator_period>1000000</emulator_period>
    <emulator_quota>-1</emulator_quota>
  </cputune>
  ...
</domain>
```

这里只简单解释其中几个配置，vcpupin标签表示将虚拟CPU绑定到某一个或多个物理CPU上，如"＜vcpupin vcpu="2"cpuset="4"/＞"表示客户机2号虚拟CPU被绑定到4号物理CPU上运行。"＜emulatorpin cpuset="1-3"/＞"表示将QEMU emulator绑定到1～3号物理CPU上。在不设置任何vcpupin和cpuset的情况下，客户机的虚拟CPU默认会被调度到任何一个物理CPU上去运行。"＜shares＞2048＜/shares＞"表示客户机占用CPU时间的加权配额，一个配置为2048的

域，会获得的CPU执行时间是配置为1024的域的两倍。如果不设置shares值，就会默认使用宿主机系统提供的默认值。

（2）内存的配置

在该域的XML配置文件中，内存大小的配置如下：

```
<memory unit='KiB'>1048576</memory>
<currentMemory unit='KiB'>1048576</currentMemory>
```

可知，内存大小为1048576 KB（即1GB），memory标签中的内存表示客户机最大可使用的内存，currentMemory标签中的内存表示启动时即分配给客户机使用的内存。在使用QEMU/KVM时，一般将二者设置为相同的值。

另外，内存的ballooning相关的配置包含在devices这个标签的memballoon子标签中，该标签配置了该客户机的内存气球设备，如下：

```
<devices>
  <memballoon model='virtio'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
  </memballoon>
</devices>
```

该配置将为客户机分配一个使用virtio-balloon驱动（已在5.1.3节中介绍过）的设备，以便实现客户机内存的ballooning调节，该设备在客户机中的PCI设备编号为0000:00:06.0。

（3）客户机系统类型和启动顺序

客户机系统类型及其启动顺序，在os标签中配置，如下：

```
<os>
  <type arch='x86_64' machine='rhel6.3.0'>hvm</type>
  <boot dev='hd'/>
  <boot dev='cdrom'/>
</os>
```

这样的配置表示客户机类型是HVM类型，HVM（硬件虚拟机，Hardware Virtual machine）原本是Xen虚拟化中的概念，它表示在硬件辅助虚拟化技术（Intel VT或AMD-V等）的支持下不需要修改客户机操作系统就可以启动客户机。因为KVM一定要依赖于硬件虚拟化技术的支持，所以在KVM中，客户机类型应该总是hvm，操作系统的架构是x86_64，机器类型是rhel6.3.0（这是RHEL 6.3系统的默认类型，也可以根据需要填写其他类型，如rhel6.2.0等）。boot选项用于设置客户机启动时的设备，这里有hd（即硬盘）和cdrom（光驱）两种，而且是按照硬盘、光驱的顺序启动的，它们在XML配置文件中的先后顺序即是启动时的先后顺序。

3.网络的配置

（1）桥接方式的网络配置

在示例的域的XML配置中，使用桥接方式的网络的相关配置如下：

```
<devices>
  ...
  <interface type='bridge'>
    <mac address='52:54:00:e9:e0:3b'/>
    <source bridge='br0'/>
    <model type='virtio'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
  </interface>
  ...
</devices>
```

type=’bridge’表示使用桥接方式使客户机获得网络，address用于配置客户机中网卡的MAC地址，＜source bridge=’br0’/＞表示使用宿主机中的br0网络接口来建立网桥，＜model type=’virtio’/＞表示在客户机中使用virtio-net驱动的网卡设备，也配置了该网卡在客户机中的PCI设备编号为0000:00:03.0。在“4.5网络配置”中已经详细介绍过几种不同的网络配置方式，而在libvirt中都有对它们的相应支持。

（2）NAT方式的虚拟网络配置

在域的XML配置中，NAT方式的虚拟网络的配置示例如下：

```
<devices>
  ...
  <interface type='network'>
    <mac address='52:54:00:32:7d:f6'/>
    <source network='default'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
  </interface>
  ...
</devices>
```

这里的设置为type=’network’和＜source network=’default’/＞表示使用NAT的方式，并使用默认的网络配置，客户机将会分配到

192.168.122.0/24网段中的一个IP地址。当然，使用NAT必须保证宿主机中运行着DHCP和DNS服务器，一般默认使用的dnsmasq软件，查询DHCP和DNS服务的运行的命令行如下：

```
[root@rhel6u3-ga ~]# ps -ef | grep dnsmasq
nobody    1863     1  0 Dec08 ?        00:00:03 /usr/sbin/dnsmasq
--strict-order --bind-interfaces
--pid-file=/var/run/libvirt/network/default.pid --conf-file=
--except-interface lo --listen-address 192.168.122.1 --dhcp-range
192.168.122.2,192.168.122.254
--dhcp-leasefile=/var/lib/libvirt/dnsmasq/default.leases
--dhcp-lease-max=253 --dhcp-no-override
```

由于配置使用了默认的NAT网络配置，可以到libvirt相关的网络配置中看到一个default.xml文件（/usr/share/libvirt/networks/default.xml），它具体配置了默认的连接方式，如下：

```
<network>
  <name>default</name>
  <bridge name="virbr0" />
  <forward/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254" />
      </dhcp>
  </ip>
</network>
```

在使用NAT时，查看宿主机中网桥的使用情况如下：

```
[root@rhel6u3-ga ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.525400b45ba5       yes             virbr0-nic
                                                        vnet0
```

其中vnet0这个网络接口就是客户机和宿主机网络连接的纽带。

（3）用户模式网络的配置

在域的XML文件中，如下的配置即实现了使用用户模式的网络：

```
<devices>
  ...
  <interface type='user'>
    <mac address="00:11:22:33:44:55"/>
  </interface>
  ...
</devices>
```

其中，type='user'表示该客户机的网络接口是用户模式网络，完全由qemu-kvm软件模拟的一个网络协议栈。在宿主机中，是没有一个虚拟的网络接口连接到virbr0这样的网桥的。

（4）网卡设备直接分配（VT-d）

在客户机的网络配置中，还可以采用PCI/PCI-e网卡将设备直接分配给客户机使用。关于设备直接分配的细节，可以参考5.2节的介绍，本节只介绍其在libvirt中的配置方式。对于设备直接分配的配置在域

的XML配置文件中有两种方式：一种是较新的方式，使用＜interface type='hostdev'/＞标签；另一种是较旧但支持设备很广泛的方式，直接使用＜hostdev＞标签。

＜interface type='hostdev'/＞标签是较新的配置方式，目前仅支持libvirt 0.9.11以上的版本，而且仅支持SR-IOV特性中的VF的直接配置。在＜interface type='hostdev'/＞标签中，用＜source＞标签来指示将宿主机中的哪个VF分配给宿主机使用，还可使用＜mac address='52:54:00:6d:90:02'＞来指定在客户机中看到的该网卡设备的MAC地址。一个示例配置如下所示，它表示将宿主机的0000:08:10.0这个VF网卡直接分配给客户机使用，并规定该网卡在客户机中的MAC地址为“52:54:00:6d:90:02”。

```
<devices>
  ...
  <interface type='hostdev'>
    <source>
      <address type='pci' domain='0x0000' bus='0x08' slot='0x10' function='0x0'/>
    </source>
    <mac address='52:54:00:6d:90:02'>
  </interface>
  ...
</devices>
```

在＜devices＞标签中直接使用＜hostdev＞标签来指定将网卡设备直接分配给客户机使用，这是较旧的配置方式，是libvirt 0.9.11版本之前对设备直接分配的唯一使用方式，而且对设备的支持较为广泛，既支持有SR-IOV功能的高级网卡的VF的直接分配，也支持无SR-IOV功能的普通PCI或PCI-e网卡的直接分配。这种方式并不支持对直接

分配的网卡在客户机中的MAC地址的设置，在客户机中网卡的MAC地址与宿主机中看到的完全相同。在域的XML配置文件中，使用＜hostdev＞标签配置网卡设备直接分配的示例如下，它表示将宿主机中的PCI 0000:08:00.0设备直接分配给客户机使用。

```
<devices>
  ...
  <hostdev mode='subsystem' type='pci' managed='yes'>
   <source>
    <address domain='0x0000' bus='0x08' slot='0x00' function='0x0'/>
   </source>
  </hostdev>
  ...
</devices>
```

4.存储的配置

在示例的域的XML配置文件中，关于客户机磁盘的配置如下：

```
<devices>
  ...
  <disk type='file' device='disk'>
    <driver name='qemu' type='raw' cache='none'/>
    <source file='/var/lib/libvirt/images/rhel6u3-1.img'/>
    <target dev='vda' bus='virtio'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
  </disk>
  ...
</devices>
```

上面的配置表示，使用raw格式的rhel6u3-1.img镜像文件作为客户机的磁盘，磁盘的缓存是关闭的，其在客户机中使用virtio总线（使用virtio-blk驱动），设备名称为/dev/vda，其PCI地址为0000:00:05.0。

＜disk＞标签是客户机磁盘配置的主标签，其中包含它的属性和一些子标签。它的type属性表示磁盘使用哪种类型作为磁盘的来源，其取值为file、block、dir或network中的一个，分别表示使用文件、块设备、目录或网络来作为客户机磁盘的来源。它的device属性表示让客户机如何来使用该磁盘设备，其取值为floppy、disk、cdrom或lun中的一个，分别表示软盘、硬盘、光盘和LUN（逻辑单元号），其默认值为disk（硬盘）。

在＜disk＞标签中可以配置许多的子标签，这里仅简单介绍一下上面示例中出现的几个重要的子标签。＜driver＞子标签用于定义Hypervisor如何为该磁盘提供驱动，它的name属性用于指定宿主机中使用的后端驱动名称，QEMU/KVM仅支持name='qemu'，但是它支持的类型type可以是多种，包括raw、qcow2、qed、bochs等。而这里的cache属性表示在宿主机中打开该磁盘时使用的缓存方式，可以配置为default、none、writethrough、writeback、directsync和unsafe等多种模式。在4.4.1节中已经详细地介绍过磁盘缓存的各种配置方式的区别。

＜source＞子标签表示磁盘的来源，当＜disk＞标签的type属性为file时，应该配置为＜source file='/var/lib/libvirt/images/rhel6u3-1.img'/＞这样的模式，而

type属性为block时，应该配置为＜source dev='/dev/sda'/＞这样的模式。

＜target＞子标签表示将磁盘暴露给客户机时的总线类型和设备名称。其dev属性表示在客户机中该磁盘设备的逻辑设备名称，而bus属性表示该磁盘设备被模拟挂载的总线类型，bus属性的值可以为ide、scsi、virtio、xen、usb、sata等。如果省略了bus属性，libvirt则会根据dev属性中的名称来“推测”bus属性的值，例如，sda会被推测是scsi，而vda被推测是virtio。

＜address＞子标签表示该磁盘设备在客户机中的PCI地址，这个标签在前面网络配置中也是多次出现的，如果该标签不存在，libvirt会自动分配一个地址。

5.其他配置简介

（1）域的配置

在域的整个XML配置文件中，＜domain＞标签是范围最大、最基本的标签，是其他所有标签的根标签。在示例的域的XML配置文件中，＜domain＞标签的配置如下：

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

在＜domain＞标签中可以配置两个属性：一个是type，用于表示Hypervisor的类型，可选的值为xen、kvm、qemu、lxc、kqemu、vmware等中的一个；另一个是id，其值是一个数字，用于在该宿主机的libvirt中唯一标识一个运行着的客户机，如果不设置id属性，libvirt会按顺序分配一个最小的可用ID。

（2）域的元数据配置

在域的XML文件中，有一部分是用于配置域的元数据（meta data）。元数据用于表示域的属性（用于区别其他的域）。在示例的域的XML文件中，元数据的配置如下：

```
<name>rhel6u3-1</name>
<uuid>e39a1025-87ec-b4ab-b08c-c02b5cebb5d0</uuid>
```

其中，name用于表示该客户机的名称，uuid是唯一标识该客户机的UUID。在同一个宿主机上，各个客户机的名称和UUID都必须是唯一的。

当然，域的元数据，还有其他很多的配置，例如Xen上的一个域的元数据配置如下：

```
<domain type='xen' id='3'>
  <name>fv0</name>
  <uuid>4dea22b31d52d8f32516782e98ab3fa0</uuid>
  <title>A short description - title - of the domain</title>
  <description>Some human readable description</description>
  <metadata>
    <app1:foo xmlns:app1="http://app1.org/app1/">..</app1:foo>
    <app2:bar xmlns:app2="http://app1.org/app2/">..</app2:bar>
  </metadata>
  ...
</domain>
```

（3）QEMU模拟器的配置

在域的配置文件中，需要制定使用的设备模型的模拟器，在emulator标签中配置模拟器的绝对路径。在示例的域的XML文件中，模拟器的配置如下：

```
<devices>
  <emulator>/usr/libexec/qemu-kvm</emulator>
...
  </devices>
```

假设自己编译了一个最新的qemu-kvm，要使用自己编译的qmeu-kvm作为模拟器，只需要将这里修改为/usr/local/bin/qemu-system-x86_64即可。不过，创建客户机时可能会遇到如下的错误信息：

```
[root@rhel6u3-ga ~]# virsh create rhel6u3-1.xml
error: Failed to create domain from rhel6u3-1.xml
error: internal error Process exited while reading console log output:
Supported machines are:
pc                   Standard PC (alias of pc-1.1)
pc-1.1               Standard PC (default)
pc-1.0               Standard PC
pc-0.15              Standard PC
pc-0.14              Standard PC
pc-0.13              Standard PC
```

这是因为qemu-system-x86_64并不支持配置文件中的rhel6.3.0机器类型，做如下修改即可解决这个问题：

```
<type arch='x86_64' machine='pc'>hvm</type>
```

（4）图形显示方式

在示例的域的XML文件中，对连接到客户机的图形显示方式的配置如下：

```
<devices>
  ...
  <graphics type='vnc' port='-1' autoport='yes'/>
...
  </devices>
```

这表示通过VNC的方式连接到客户机，其VNC端口为libvirt自动分配。

也可以支持其他多种类型的图形显示方式，以下就配置了SDL、VNC、RDP、SPICE等多种客户机显示方式。

```
<devices>
  ...
  <graphics type='sdl' display=':0.0'/>
  <graphics type='vnc' port='5904'>
    <listen type='address' address='1.2.3.4'/>
  </graphics>
  <graphics type='rdp' autoport='yes' multiUser='yes' />
  <graphics type='desktop' fullscreen='yes'/>
  <graphics type='spice'>
    <listen type='network' network='rednet'/>
  </graphics>
  ...
</devices>
```

（5）客户机声卡和显卡的配置

在示例的域的XML文件中，该客户机的声卡和显卡的配置如下：

```
<devices>
  ...
  <sound model='ich6'>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
  </sound>
  <video>
    <model type='cirrus' vram='9216' heads='1'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
  </video>
  ...
</devices>
```

＜sound＞标签表示的是声卡配置，其中model属性表示为客户机模拟出来的声卡的类型，其取值为es1370、sb16、ac97和ich6中的一个。

＜video＞标签表示的是显卡配置，其中＜model＞子标签表示为客户机模拟的显卡的类型，它的类型（type）属性可以为vga、cirrus、vmvga、xen、vbox、qxl等中的一个，vram属性表示虚拟显卡的显存容量（单位为KB），heads属性表示显示屏幕的序号。KVM客户机的显卡的默认配置为cirrus类型、显存为9216（即9 MB）、使用在第1号屏幕上。

（6）串口和控制台

串口和控制台是非常有用的设备，特别是在调试客户机的内核或遇到客户机宕机的情况下，一般都可以在串口或控制台中查看到一些利于系统管理员分析问题的日志信息。在示例的域的XML文件中，客户机串口和控制台的配置如下：

```
<devices>
  ...
  <serial type='pty'>
    <target port='0'/>
  </serial>
  <console type='pty'>
    <target type='serial' port='0'/>
  </console>
  ...
</devices>
```

设置了客户机的编号为0的串口（即/dev/ttyS0），使用宿主机中的伪终端（pty），由于这里没有指定使用宿主机中的哪个虚拟终端，因此libvirt会自己选择一个空闲的虚拟终端（可能为/dev/pts/下的任意一个）。当然也可以加上＜source path='/dev/pts/1'/＞配置来明确指定使用宿主机中的哪一个虚拟终端。在通常情况下，控制台（console）配置在客户机中的类型为'serial'，此时，如果没有配置串口（serial），则会将控制台的配置复制到串口配置中，如果已经配置了串口（本例即是如此），则libvirt会将控制台的配置项忽略。

当然为了让控制台有输出信息和能够与客户机交互，也需在客户机中配置将信息输出到串口，如在Linux客户机内核的启动行中添加"console=ttyS0"这样的配置。在“5.10.2其他常用参数”中，对-serial参数的介绍中有更多和串口配置相关的介绍。

（7）输入设备

在示例的XML文件中，在客户机图形界面下进行交互的输入设备的配置如下：

```
<devices>
  ...
  <input type='tablet' bus='usb'/>
  <input type='mouse' bus='ps2'/>
  ...
  </devices>
```

这里的配置会让QEMU模拟PS2接口的鼠标，还提供了tablet这种类型的设备，让光标可以在客户机获取绝对位置定位。在“4.6.3 VNC显示中的鼠标偏移”中已经介绍过tablet设备的使用及其带来的好处。

（8）PCI控制器

根据客户机架构的不同，libvirt默认会为客户机模拟一些必要的PCI控制器（而不需要在XML配置文件中指定），而一些PCI控制器需要显式地在XML配置文件中配置。在示例的域的XML文件中，一些PCI控制器的配置如下：

```
<controller type='usb' index='0'>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
  </controller>
  <controller type='ide' index='0'>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
</controller>
```

这里显式指定了一个USB控制器和一个IDE控制器。libvirt默认还会为客户机分配一些必要的PCI设备，如PCI主桥（Host bridge）、

ISA桥等。使用示例的域的XML配置文件启动客户机，在客户机中查看到的PCI信息如下：

```
[root@rhel6u3-1 ~]# lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.2 USB controller: Intel Corporation 82371SB PIIX3 USB [Natoma/Triton II] (rev 01)
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Cirrus Logic GD 5446
00:03.0 Ethernet controller: Realtek Semiconductor Co., Ltd.
RTL-8139/8139C/8139C+ (rev 20)
00:04.0 Audio device: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6
Family) High Definition Audio Controller (rev 01)
00:05.0 SCSI storage controller: Red Hat, Inc Virtio block device
00:06.0 Unclassified device [00ff]: Red Hat, Inc Virtio memory balloon
```

6.1.5 libvirt API简介

libvirt的核心价值和主要目标就是提供一套管理虚拟机的、稳定的、高效的应用程序接口（API）。libvirt API本身是用C语言实现的，本节以其提供的最核心的C语言接口的API为例进行简单的介绍。

libvirt API大致可划分为如下8个部分。

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

只有在与Hypervisor建立连接之后，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函数提供了根据认证建立的连接。virConnectGetCapabilities函数返回对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。

❑域管理的API：以virDomain开头的一系列函数。

虚拟机最基本的管理职能就是对各个节点上的域的管理，故在libvirt API中实现了很多针对域管理的函数。要管理域，首先要获取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virDomainPtrvirDomainLookupByID(virCo nnectPtr 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。在得到某个域的对象后，就可以进行很多的操作，可以查询域的信息（如virDomainGetHostname、virDomainGetInfo、virDomainGetVcpus、virDomainGetVcpusFlags、virDomainGetCPUStats，等等），也可以

控制域的生命周期（如virDomainCreate、virDomainSuspend、virDomainResume、virDomainDestroy、virDomainMigrate，等等）。

❑节点管理的API：以virNode开头的一系列函数。

域运行在物理节点之上，libvirt也提供了对节点进行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。

❑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也对虚拟化环境中的网络管理提供了丰富的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，例如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。

❑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的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。

❑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）。

❑事件管理的API：以virEvent开头的一系列函数。

libvirt支持事件机制，在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）时得到自己定义的一些通知。

❑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本章末尾的参考阅读中介绍的资料。

6.1.6 建立到Hypervisor的连接

要使用libvirt API进行虚拟化管理，就必须先建立到Hypervisor的连接，因为有了连接才能管理节点、Hypervisor、域、网络等虚拟化中的要素。本节就介绍一下建立到Hypervisor连接的一些方式。

对于一个libvirt连接，可以使用简单的客户端-服务器端（C/S）的架构模式来解释，一个服务器端运行着Hypervisor，一个客户端去连接服务器端的Hypervisor，然后进行相应的虚拟化管理。当然，如果通过libvirt API实现本地的管理，则客户端和服务器端都在同一个节点上，并不依赖于网络连接。一般来说（如基于QEMU/KVM的虚拟化方案），不管是基于libvirt API的本地管理还是远程管理，在服务器端的节点上，除了需要运行相应的Hypervisor，还需要让libvirtd这个守护进程处于运行中的状态，以便让客户端连接到libvirtd从而进行管理操作。不过，也并非所有的Hypervisor都需要运行libvirtd守护进程，比如VMware ESX/ESXi就不需要在服务器端运行libvirtd，依然可以通过libvirt客户端以另外的方式[2] 连接到VMware。

由于支持多种Hypervisor，libvirt需要通过唯一的标识来指定如何才能准确地连接到本地或远程的Hypervisor。为了达到这个目的，libvirt使用了在互联网应用中广泛使用的URI[3] （Uniform Resource Identifier，统一资源标识符）来标识到某个Hypervisor的连接。

libvirt中连接的标识符URI，其本地URI和远程URI有一些区别，下面分别介绍一下它们的使用方式。

1.本地URI

在libvirt的客户端使用本地的URI用于连接本系统范围内的Hypervisor，本地URI的一般格式如下：

```
driver[+transport]:///[path][?extral-param]
```

其中，driver是连接Hypervisor的驱动名称（如qemu、xen、xbox、lxc等），transport是选择该连接所使用的传输方式（可以为空，也可以是"unix"这样的值），path是连接到服务器端上的某个路径，?extral-param是可以用于添加额外的一些参数（如Unix domain sockect的路径）。

在libvirt中KVM使用QEMU驱动。QEMU驱动是一个多实例的驱动，它提供了一个系统范围内的特权驱动（即"system"实例）和一个用户相关的非特权驱动（即"session"实例）。通过"qemu:///session"这样的URI可以连接到一个libvirtd非特权实例，但是这个实例必须是与本地客户端的当前用户和用户组相同的实例，也就说，根据客户端的当前用户和用户组去服务器端寻找对应用户下的实例。在建立session连接后，可以查询和控制的域或其他资源都仅仅是在当前用户权限范围内的，而不是整个节点上的全部域或其他资源。而使

用"qemu:///system"这样的URI连接到libvirtd实例，是需要系统特权账号"root"权限的。在建立system连接后，由于它是具有最大权限的，因此以查询和控制整个节点范围内的域，还可以管理该节点上特权用户才能管理的块设备、PCI设备、USB设备、网络设备等系统资源。一般来说，为了管理的方便，在公司内网范围内建立到system实例的连接进行管理的情况比较常见，当然为了安全考虑，赋予不同用户不同的权限就可以使用建立到session实例的连接。

在libvirt中，本地连接QEMU/KVM的几个URI示例如下：

（1）qemu:///session

连接到本地的session实例，该连接仅能管理当前用户虚拟化资源。

（2）qemu+unix:///session

以Unix domain sockect的方式连接到本地的session实例，该连接仅能管理当前用户的虚拟化资源。

（3）qemu:///system

连接到本地的system实例，该连接可以管理当前节点的所有特权用户可以管理的虚拟化资源。

（4）qemu+unix:///system

以Unix domain sockect的方式连接到本地的system实例，该连接可以管理当前节点的所有特权用户可以管理的虚拟化资源。

2.远程URI

除了本地管理，libvirt还提供了非常方便的远程的虚拟化管理功能。libvirt可以使用远程URI来建立到网络上的Hypervisor的连接。远程URI和本地URI也是类似的，只是会增加用户名、主机名（或IP地址）和连接端口来连接到远程的节点。远程URI的一般格式如下：

```
driver[+transport]://[user@][host][:port]/[path][?extral-param]
```

其中，transport表示传输方式，其取值可以是ssh、tcp、libssh2等；user表示连接远程主机使用的用户名，host表示远程主机的主机名或IP地址，port表示连接远程主机的端口。其余的参数的意义，与本地URI中介绍的完全一样。

在远程URI连接中，也存在使用system实例和session实例两种方式，这二者的区别和用途，与本地URI中介绍的内容是完全一样的。

在libvirt中，远程连接QEMU/KVM的URI示例如下：

（1）qemu+ssh://root@example.com/system

通过ssh通道连接到远程节点的system实例，具有最大的权限来管理远程节点上的虚拟化资源。建立该远程连接时，需要经过ssh的用户名和密码验证或者基于密钥的验证。

（2）qemu+ssh://user@example.com/session

通过ssh通道连接到远程节点的使用user用户的session实例，该连接仅能对user用户的虚拟化资源进行管理，建立连接时同样需要经过ssh的验证。

（3）qemu://example.com/system

通过建立加密的TLS连接与远程节点的system实例相连接，具有对该节点特权管理权限。在建立该远程连接时，一般需要经过TLS x509安全协议的证书验证。

（4）qemu+tcp://example.com/system

通过建立非加密的普通TCP连接与远程节点的system实例相连接，具有对该节点特权管理权限。在建立该远程连接时，一般需要经过SASL/Kerberos认证授权。

3.使用URI建立到Hypervisor的连接

在某个节点启动libvirtd后，一般在客户端都可以通过ssh方式连接到该节点。而TLS和TCP等连接方式却不一定都处于开启可用状态，如RHEL 6.3系统中的libvirtd服务在启动时就默认没有打开TLS和TCP这两种连接方式。关于libvirtd的配置可以参考“6.1.3 libvirt和libvirtd的配置”中的介绍。而在服务器端的libvirtd打开了TLS和TCP连接方式，也需要一些认证方面的配置，当然也可直接关闭掉认证功能（这样不安全），可以参考libvirtd.conf配置文件。

我们看到URI这个标识还是比较复杂的，特别是在管理很多远程节点时，需要使用很多的URI连接。为了简化系统管理的复杂程度，可以在客户端的libvirt配置文件中，为URI命名别名以方便记忆，这在6.1.3节中也已经介绍过了。

在6.1.5节中已经介绍过，libvirt使用virConnectOpen函数来建立到Hypervisor的连接，所以virConnectOpen函数就需要一个URI作为参数。而当传递给virConnectOpen的URI为空值（NULL）时，libvirt会依次根据如下3条规则去决定使用哪一个URI。

1）试图使用LIBVIRT_DEFAULT_URI这个环境变量。

2）使用客户端的libvirt配置文件中的uri_default参数的值。

3）依次试图用每个Hypervisor的驱动去建立连接，直到能正常建立连接后即停止尝试。

当然，如果这3条规则都不能够让客户端libvirt建立到Hypervisor的连接，就会报出建立连接失败的错误信息（"failed to connect to the hypervisor"）。

在使用virsh这个libvirt客户端工具时，可以用"-c"或"--connect"选项来指定建立到某个URI的连接。只有连接建立之后，才能够操作。使用virsh连接到本地和远程的Hypervisor的示例如下：

```
[root@rhel6u3-ga ~]# virsh -c qemu:///system
Welcome to virsh, the virtualization interactive terminal.
virsh # list
 Id    Name                           State
----------------------------------------------------
 8     rhel6u3-2                      running
 10    rhel6u3-1                      running

virsh # quit

[root@rhel6u3-ga ~]# virsh -c qemu+ssh://root@192.168.158.31/system
root@192.168.158.31's password:
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit


virsh # list
 Id    Name                           State
----------------------------------------------------
 3     rhel6u3-remote                 running

virsh # quit

[root@rhel6u3-ga ~]#
```

其实，除了针对QEMU、Xen、LXC等真实Hypervisor的驱动之外，libvirt自身还提供了一个名叫"test"的傀儡Hypervisor及其驱动程序。test Hypervisor是在libvirt中仅仅用于测试和命令学习的目的，因为在本地的和远程的Hypervisor都连接不上（或无权限连接）时，test这个Hypervisor却一直都会处于可用状态。使用virsh连接到test Hypervisor的示例操作如下：

```
[root@rhel6u3-ga ~]# virsh -c test:///default list
 Id    Name                           State
----------------------------------------------------
 1     test                           running

[root@rhel6u3-ga ~]# virsh -c test:///default
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # list
 Id    Name                           State
----------------------------------------------------
 1     test                           running

virsh # quit

[root@rhel6u3-ga ~]#
```

6.1.7 libvirt API使用示例

经过前面几个小节对libvirt的配置、编译、API、建立连接等内容的介绍，相信大家对libvirt已经有了大致的了解。对API进行学习的最好方法就是通过代码来调用API实现几个小功能，所以本节主要通过两个示例来分别演示如何调用libvirt的由C语言和Python语言绑定的API。

1.libvirt的C API的使用

在使用libvirt API之前，必须要在远程或本地的节点上启动libvirtd守护进程。在使用libvirt的客户端前，先安装libvirt-devel软件包或使用6.1.2节中介绍的方法来自行编译和安装libvirt。本次示例中安装的是RHEL 6.3自带的libvirt-devel软件包，如下：

```
[root@rhel6u3-ga ~]# rpm -q libvirt-devel
libvirt-devel-0.9.10-21.el6.x86_64
```

如下一个简单的C程序（文件名为dominfo.c）就是通过调用libvirt的API来查询一些关于某个域的信息。该示例程序比较简单易懂，它仅仅是使用libvirt API的一个演示程序，这里不做过多的介绍。不过，这里有三点需要注意：1）需要在示例代码的开头引入<libvirt/libvirt.h>这个头文件；2）由于只是做查询信息的功能，所以可以使用virConnectOpenReadOnly来建立只读连接；3）这里使用了空值（NULL）作为URI，是让libvirt自动根据6.1.6小节中介绍的默认规则去建立到Hypervisor的连接。这里由于本地已经运行了

libvirtd守护进程，并启动了两个QEMU/KVM客户机，所以它默认会建立到QEMU/KVM的连接。

```
/**
 * Get domain information via libvirt C API.
 * Testedwithlibvirt-devel-0.9.10 on a RHEL 6.3 host system.
 */

#include <stdio.h>
#include <libvirt/libvirt.h>

int getDomainInfo(int id) {
        virConnectPtr conn = NULL; /* the hypervisor connection */
        virDomainPtr dom = NULL;   /* the domain being checked */
        virDomainInfo info;        /* the information being fetched */

        /* NULL means connect to local QEMU/KVM hypervisor */
        conn = virConnectOpenReadOnly(NULL);
        if (conn == NULL) {
                fprintf(stderr, "Failed to connect to hypervisor\n");
                return 1;
        }

        /* Find the domain by its ID */
        dom = virDomainLookupByID(conn, id);
        if (dom == NULL) {
                fprintf(stderr, "Failed to find Domain %d\n", id);
                virConnectClose(conn);
                return 1;
        }

        /* Get virDomainInfo structure of the domain */
        if (virDomainGetInfo(dom, &info) < 0) {
                fprintf(stderr, "Failed to get information for Domain %d\n", id);
                virDomainFree(dom);
```

```
                virConnectClose(conn);
                return 1;
        }

        /* Print some info of the domain */
        printf("Domain ID: %d\n", id);
        printf("    vCPUs: %d\n", info.nrVirtCpu);
        printf("   maxMem: %d KB\n", info.maxMem);
        printf("   memory: %d KB\n", info.memory);

        if (dom != NULL)
                virDomainFree(dom);
        if (conn != NULL)
                virConnectClose(conn);

        return 0;
}

int main(int argc, char **argv)
{
        int dom_id = 3;
        printf("-----Get domain info by ID via libvirt C API -----\n");
        getDomainInfo(dom_id);
        return 0;
}
```

在获得dominfo.c这个示例程序之后，用virsh命令查看当前节点中的情况，再编译和运行这个示例程序去查询一些域的信息，可以将二者得到的一些信息进行对比，可以发现得到的信息是匹配的，命令行操作如下：

```
[root@rhel6u3-ga kvm_demo]# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm-guest                      running
 6     kvm-guest-2                    running

[root@rhel6u3-ga kvm_demo]# virsh dommemstat 3
actual 1048576
rss 680228

[root@rhel6u3-ga kvm_demo]# virsh vcpucount 3
maximum      live           4
current      live           4

[root@rhel6u3-ga kvm_demo]# gcc dominfo.c -o dominfo -lvirt

[root@rhel6u3-ga kvm_demo]# ./dominfo

 -----Get domain info by ID via libvirt C API -----
 Domain ID: 3
     vCPUs: 4
    maxMem: 1048576 KB
    memory: 1048576 KB
```

这里需要注意的是，在使用GCC编译dominfo.c这个示例程序时，加上了"-lvirt"这个参数来指定程序链接时的依赖的库文件，如果不指定libvirt相关的共享库，则会发生链接时错误。在本次示例的RHEL 6.3系统中，需要依赖的libvirt共享库文件是/usr/lib64/libvirt.so，如下：

```
[root@rhel6u3-ga ~]# ls /usr/lib64/libvirt.so
/usr/lib64/libvirt.so
```

2.libvirt的Python API的使用

在6.1.1节中已经介绍过，许多种编程语言都提供了libvirt的绑定。Python作为一种在Linux上比较流行的编程语言，它也提供了libvirt API的绑定。在使用Python调用libvirt之前，需要安装libvirt-python软件包，或者使用6.1.2节介绍的方法来自行编译和安装libvirt及其Python API。

本次示例是基于RHEL 6.3系统自带的libvirt和libvirt-python软件包来进行的，对libvirt-python及Python中的libvirt API文件的查询，命令行如下：

```
[root@rhel6u3-ga ~]# rpm -q libvirt-python
libvirt-python-0.9.10-21.el6.x86_64
[root@rhel6u3-ga ~]# ls /usr/lib64/python2.6/site-packages/libvirt*
/usr/lib64/python2.6/site-packages/libvirtmod_qemu.so
/usr/lib64/python2.6/site-packages/libvirt.pyo
/usr/lib64/python2.6/site-packages/libvirtmod.so
/usr/lib64/python2.6/site-packages/libvirt_qemu.py
/usr/lib64/python2.6/site-packages/libvirt.py
/usr/lib64/python2.6/site-packages/libvirt_qemu.pyc
/usr/lib64/python2.6/site-packages/libvirt.pyc
/usr/lib64/python2.6/site-packages/libvirt_qemu.pyo
```

如下是本次示例使用的一个Python小程序（libvirt-test.py），用于通过调用libvirt的Python API来查询域的一些信息。该Python程序示例的源代码如下：

```
#!/usr/bin/python
# Get domain info via libvirt python API.
# Tested with python2.6 and libvirt-python-0.9.10 on a KVM host.

import libvirt
import sys

def createConnection():
    conn = libvirt.openReadOnly(None)
```

```
    if conn == None:
        print 'Failed to open connection to QEMU/KVM'
        sys.exit(1)
    else:
        print '-----Connection is created successfully-----'
        return conn

def closeConnnection(conn):
    print ''
    try:
        conn.close()
    except:
        print 'Failed to close the connection'
        return 1

    print 'Connection is closed'

def getDomInfoByName(conn, name):
    print ''
    print '----------- get domain info by name ----------"'
    try:
        myDom = conn.lookupByName(name)
    except:
        print 'Failed to find the domain with name "%s"' % name
        return 1

    print "Dom id: %d   name: %s" % (myDom.ID(), myDom.name())
    print "Dom state: %s" % myDom.state(0)
    print "Dom info: %s" % myDom.info()
    print "memory: %d MB" % (myDom.maxMemory()/1024)
    print "memory status: %s" % myDom.memoryStats()
    print "vCPUs: %d" % myDom.maxVcpus()

def getDomInfoByID(conn, id):
    print ''
    print '----------- get domain info by ID ----------"'
    try:
        myDom = conn.lookupByID(id)
    except:
        print 'Failed to find the domain with ID "%d"' % id
        return 1

    print "Domain id is %d ; Name is %s" % (myDom.ID(), myDom.name())

if __name__ == '__main__':
    name1 = "kvm-guest"
```

```
name2 = "notExist"
id1 = 3
id2 = 9999
print "---Get domain info via libvirt python API---"
conn = createConnection()
getDomInfoByName(conn, name1)
getDomInfoByName(conn, name2)
getDomInfoByID(conn, id1)
getDomInfoByID(conn, id2)
closeConnnection(conn)
```

该示例程序比较简单，只是简单地调用libvirt Python API获取一些信息，这里唯一需要注意的是"import libvirt"语句引入了libvirt.py这个API文件，然后才能够使用libvirt.openReadOnly、conn.lookupByName等libvirt中的方法。在本次示例中，必须被引入的libvirt.py这个API文件的绝对路径是/usr/lib64/python2.6/site-packages/libvirt.py，它实际调用的是/usr/lib64/python2.6/site-packages/libvirtmod.so这个共享库文件。

在获得该示例Python程序后，运行该程序（libvirt-test.py），查看其运行结果，命令行操作如下：

```
[root@rhel6u3-ga kvm_demo]# python libvirt-test.py 2>/dev/null
---Get domain info via libvirt python API---
-----Connection is created successfully-----

----------- get domain info by name ----------"
Dom id: 3   name: kvm-guest
Dom state: [1, 1]
Dom info: [1, 1048576L, 1048576L, 4, 257070000000L]
memory: 1024 MB
memory status: {'actual': 1048576L, 'rss': 680228L}
vCPUs: 4

----------- get domain info by name ----------"
Failed to find the domain with name "notExist"

----------- get domain info by ID ----------"
Domain id is 3 ; Name is kvm-guest

----------- get domain info by ID ----------"
Failed to find the domain with ID "9999"

Connection is closed
```

6.2 virsh

6.2.1 virsh简介

libvirt项目的源代码中就包含了virsh这个虚拟化管理工具的代码。virsh是用于管理虚拟化环境中的客户机和Hypervisor的命令行工具，与virt-manager等工具类似，它也是通过调用libvirt API来实现虚拟化的管理。virsh是完全在命令行文本模式下运行的用户态工具，它是系统管理员通过脚本程序实现虚拟化自动部署和管理的理想工具之一。

virsh是用C语言编写的一个使用libvirt API的虚拟化管理工具。virsh程序的源代码在libvirt项目源代码的tools目录下，实现virsh工具最核心的一个源代码文件是virsh.c，其路径如下：

```
[root@jay-linux libvirt-1.0.0]# cd tools/
[root@jay-linux tools]# pwd
/root/kvm_demo/libvirt-1.0.0/tools
[root@jay-linux tools]# ls virsh.c
virsh.c
```

在使用virsh命令行进行虚拟化管理操作时，可以使用两个工作模式：交互模式和非交互模式。交互模式，是连接到相应的Hypervisor

上，然后输入一个命令得到一个返回结果，直到用户使用"quit"命令退出连接。非交互模式，是直接在命令行中一个建立连接的URI之后添加需要执行的一个或多个命令，执行完成后将命令的输出结果返回到当前终端上，然后自动断开连接。

使用virsh的交互模式，命令行操作示例如下：

```
[root@rhel6u3-ga ~]# virsh -c qemu+ssh://root@192.168.158.31/system
root@192.168.158.31's password:
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # list
 Id    Name                           State
----------------------------------------------------
 3     rhel6u3-remote                 running

virsh # quit
```

使用virsh的非交互模式，命令行操作示例如下：

```
[root@rhel6u3-ga ~]# virsh -c qemu+ssh://root@192.168.158.31/system "list"
root@192.168.158.31's password:
 Id    Name                           State
----------------------------------------------------
 3     rhel6u3-remote                 running
```

另外，在某个节点上直接使用"virsh"命令，就默认连接节到本节点的Hypervisor之上，并且进入virsh的命令交互模式。而直接使用"virsh list"命令，就是在连接本节点的Hypervisor之后，使用virsh的非交互模式中执行了"list"命令操作。

6.2.2 virsh常用命令

virsh这个命令行工具使用libvirt API实现了很多命令来管理Hypervisor、节点和域，实现了前面第4章和第5章提到的qemu-kvm命令行中的多数参数和QEMU monitor中的多数命令。这里只能说，virsh实现了对QEMU/KVM中的多数而不是全部的功能的调用，这是和开发模式及流程相关的，libvirt中实现的功能和最新的QEMU/KVM中的功能相比有一定的滞后性。一般来说，一个功能都是先在QEMU/KVM代码中实现，然后再修改libvirt的代码来实现的，最后由virsh这样的用户空间工具添加相应的命令接口去调用libvirt来实现。当然，除了QEMU/KVM、libvirt和virsh还实现了对Xen、VMware等其他Hypervisor的支持，如果考虑到这个因素，virsh等工具中有部分功能也可能是QEMU/KVM中本身就没有实现的。

virsh工具有很多的命令和功能，本小节仅针对virsh的一些常见命令进行简单介绍，一些更详细的参考文档可以在Linux系统中通过"man virsh"命令查看帮助文档，也可以参考本章最后的参考阅读中的内容。这里将virsh常用命令划分为五个类别来分别进行介绍，在介绍virsh命令时，使用的时是RHEL 6.3系统中的libvirt0.9.10版本，假设已经通过交互模式连接到本地或远程的一个Hypervisor的system实例上了（拥有该节点上最高的特权），以在介绍交互模式中使用的命令作为本节的示例。对于与域相关的管理，一般都需要使用域的

ID、名称或UUID这样的唯一标识来指定是对某个特定的域进行的操作。为了简单起见，在本节中，一般使用"＜ID＞"来表示一个域的唯一标识（而不专门指定为"＜ID or Name or UUID＞"这样冗长的形式）。另外，介绍一个输入virsh命令的小技巧：在交互模式中输入命令的交互方式，与在终端中输入Shell命令进行的交互类似，可以使用＜Tab＞键根据已经输入的部分字符（在virsh支持的范围内）进行联想，从而找到匹配的命令。

1.域管理的命令

virsh的最重要的功能之一就是实现对域（客户机）的管理，当然其相关的命令也是最多的，而且后面的网络管理、存储管理也都有很多是算对域的管理。

表6-1列出了域管理中的一小部分常用的virsh命令。

表 6-1　virsh 中域管理相关的常用命令

命　　令	功能描述
list	获取当前节点上所有域的列表
domstate<ID or Name or UUID>	获取一个域的运行状态
dominfo<ID>	获取一个域的基本信息
domid <Name or UUID>	根据域的名称或 UUID 返回域的 ID 值
domname <ID or UUID>	根据域的 ID 或 UUID 返回域的名称
dommemstat<ID>	获取一个域的内存使用情况的统计信息
setmem <ID><mem-size>	设置一个域的内存大小（默认单位为 KB）
vcpuinfo<ID>	获取一个域的 vCPU 的基本信息

（续）

命　　令	功能描述
vcpupin<ID><vCPU><pCPU>	将一个域的 vCPU 绑定到某个物理 CPU 上运行
setvcpus<ID><vCPU-num>	设置一个域的 vCPU 的个数
vncdisplay<ID>	获取一个域的 VNC 连接 IP 地址和端口
create<dom.xml>	根据域的 XML 配置文件创建一个域（客户机）
suspend<ID>	暂停一个域名
resume <ID>	唤醒一个域名
shutdown<ID>	让一个域执行关机操作
reboot <ID>	让一个域重启
reset <ID>	强制重启一个域，相当于在物理机器上按电源“reset”按钮（可能会损坏该域的文件系统）
destroy<ID>	立即销毁一个域，相当于直接拔掉物理机器的电源线（可能会损坏该域的文件系统）
save<ID><file.img>	保存一个运行中的域的状态到一个文件中
restore <file.img>	从一个被保存的文件中恢复一个域的运行
migrate <ID><dest_url>	将一个域迁移到另外一个目的地址
dumpxml <ID>	以 XML 格式转存出一个域的信息到标准输出中
attach-device<ID><device.xml>	向一个域添加 XML 文件中的设备（热插拔）
detach-device<ID><device.xml>	将 XML 文件中的设备从一个域中移除
console<ID>	连接到一个域的控制台

对上面表格中提及域管理的几个命令，在virsh的交互模式中进行操作的示例如下：

```
virsh # list
 Id    Name                           State
----------------------------------------------------
 10    rhel6u3-1                      running
 11    rhel6u3-2                      running

virsh # domname 10
rhel6u3-1

virsh # vcpucount 10
maximum      config         2
maximum      live           2
current      config         2
current      live           2

virsh # setmem 10 1G

virsh # dommemstat 10
actual 1048576
rss 1183060
```

```
virsh # console 10
Connected to domain rhel6u3-1
Escape character is ^]
```

2.宿主机和Hypervisor的管理命令

一旦建立有特权的连接，virsh也可以对宿主机和Hypervisor进行管理，主要是对宿主机和Hypervisor信息的查询。

表6-2列出了对宿主机和Hypervisor进行管理的部分常用的virsh命令。

表 6-2　virsh 中宿主机和 Hypervisor 管理相关的常用命令

命　　令	功能描述
version	显示 libvirt 和 Hypervisor 的版本信息
sysinfo	以 XML 格式打印宿主机系统的信息
nodeinfo	显示该节点的基本信息
uri	显示当前连接的 URI
hostname	显示当前节点（宿主机）的主机名
capabilities	显示该节点宿主机和客户机的架构和特性
freecell	显示当前 MUMA 单元的可用空闲内存
nodememstats<cell>	显示该节点的（某个）内存单元使用情况的统计
connect <URI>	连接到 URI 指示的 Hypervisor
nodecpustats<cpu-num>	显示该节点的（某个）CPU 使用情况的统计
qemu-attach<pid>	根据 PID 添加一个 QEMU 进程到 libvirt 中
qemu-monitor-command domain [--hmp] command	向域的 QEMU monitor 中发送一个命令；一般需要“--hmp”参数，以便直接传入 monitor 中的命令而不需要转换

对于宿主机和Hypervisor管理的命令，选择其中的几个，命令行操作示例如下：

```
virsh # version
Compiled against library: libvir 0.9.10
Using library: libvir 0.9.10
Using API: QEMU 0.9.10
Running hypervisor: QEMU 0.12.1

virsh # nodeinfo
CPU model:           x86_64
CPU(s):              8
CPU frequency:       1200 MHz
CPU socket(s):       1
Core(s) per socket:  4
Thread(s) per core:  2
NUMA cell(s):        1
Memory size:         3870556 kB

virsh # uri

 qemu:///system

 virsh # hostname
 rhel6u3-ga

 virsh # qemu-monitor-command 11 --hmp "info kvm"
 kvm support: enabled
```

3.网络的管理命令

virsh可以对节点上的网络接口和分配给域的虚拟网络进行管理。

表6-3列出了网络管理中的一小部分常用的命令。

表 6-3　virsh 中网络管理相关的常用命令

命　　令	功能描述
iface-list	显示出物理主机的网络接口列表
iface-mac<if-name>	根据网络接口名称查询其对应的 MAC 地址
iface-name <MAC>	根据 MAC 地址查询其对应的网络接口名称
iface-edit<if-name-or-uuid>	编辑一个物理主机的网络接口的 XML 配置文件
iface-dumpxml<if-name-or-uuid>	以 XML 格式转存出一个网络接口的状态信息
iface-destroy<if-name-or-uuid>	关闭宿主机上一个物理网络接口
net-list	列出 libvirt 管理的虚拟网络
net-info<net-name-or-uuid>	根据名称查询一个虚拟网络的基本信息
net-uuid<net-name>	根据名称查询一个虚拟网络的 UUID
net-name <net-UUID>	根据 UUID 查询一个虚拟网络的名称
net-create<net.xml>	根据一个网络 XML 配置文件创建一个虚拟网络
net-edit<net-name-or-uuid>	编译一个虚拟网络的 XML 配置文件
net-dumpxml<net-name-or-uuid>	转存出一个虚拟网络的 XML 格式化的配置信息
net-destroy<net-name-or-uuid>	销毁一个虚拟网络

在virsh命令中关于网络管理的几个命令的命令行操作如下：

```
virsh # iface-list
Name                 State      MAC Address
---------------------------------------------
br0                  active     88:88:88:88:87:88
eth0                 active     88:88:88:88:87:88
lo                   active     00:00:00:00:00:00

virsh # iface-mac eth0
88:88:88:88:87:88

virsh # net-list
Name                 State      Autostart
-----------------------------------------
default              active     yes
```

```
virsh # net-info default
Name            default
UUID            6ac4e5e9-c351-414f-a6a7-9a45d8304ccb
Active:         yes
Persistent:     yes
Autostart:      yes
Bridge:         virbr0

virsh # net-uuid default
6ac4e5e9-c351-414f-a6a7-9a45d8304ccb
```

4. 存储池和存储卷的管理命令

virsh也可以对节点上的存储池和存储卷进行管理。

表6-4列出了对存储池和存储卷管理的部分常用命令。

表 6-4　virsh 中存储管理相关的常用命令

命　　令	功能描述
pool-list	显示出 libvirt 管理的存储池
pool-info <pool-name>	根据一个存储池名称查询其基本信息
pool-uuid<pool-name>	根据存储池名称查询其 UUID
pool-create<pool.xml>	根据 XML 配置文件的信息创建一个存储池
pool-edit<pool-name-or-uuid>	编辑一个存储池的 XML 配置文件
pool-destroy<pool-name-or-uuid>	关闭一个存储池（在 libvirt 可见范围内）
pool-delete<pool-name-or-uuid>	删除一个存储池（不可恢复）
vol-list<pool-name-or-uuid>	查询一个存储池中的存储卷的列表
vol-name<vol-key-or-path>	查询一个存储卷的名称
vol-path --pool <pool><vol-name-or-key>	查询一个存储卷的路径
vol-create<vol.xml>	根据 XML 配置文件创建一个存储池
vol-clone<vol-name-path><name>	克隆一个存储卷
vol-delete <vol-name-or-key-or-path>	删除一个存储卷

在virsh中关于存储池和存储卷管理的几个常用命令的命令行操作如下：

```
virsh # vol-list
error: command 'vol-list' requires <pool> option
virsh # pool-list
Name                 State      Autostart
-----------------------------------------
default              active     yes

virsh # pool-info default
Name:           default
UUID:           4ec0d276-4601-8b33-fe40-31882b3c1837
State:          running
Persistent:     yes
Autostart:      yes


  Capacity:       98.43 GB
  Allocation:     21.83 GB
  Available:      76.60 GB

  virsh # vol-list default
  Name                 Path
  -----------------------------------------
  RHEL6.3.iso          /var/lib/libvirt/images/RHEL6.3.iso
  rhel6u3-new-clone.img /var/lib/libvirt/images/rhel6u3-new-clone.img
  rhel6u3-new.img      /var/lib/libvirt/images/rhel6u3-new.img
  rhel6u3.qcow2        /var/lib/libvirt/images/rhel6u3.qcow2

  virsh # vol-info --pool default rhel6u3.qcow2
  Name:           rhel6u3.qcow2
  Type:           file
  Capacity:       20.00 GB
  Allocation:     1.96 GB

  virsh # vol-path --pool default rhel6u3.qcow2
  /var/lib/libvirt/images/rhel6u3.qcow2

  virsh # vol-name /var/lib/libvirt/images/rhel6u3.qcow2
  rhel6u3.qcow2
```

5.其他常用命令

除了对节点、Hypervisor、域、虚拟网络、存储池等的管理之外，virsh还有一些其他的命令。表6-5列出了部分其他的常用命令。

表 6-5　virsh 中部分命令

命　　令	功能描述
help	显示出 virsh 的命令帮助文档
pwd	打印出当前的工作目录
cd <your-dir>	改变当前工作目录
echo "test-content"	回显 echo 命令后参数中的内容
quit	退出 virsh 的交互终端
exit	退出 virsh 的交互终端（与 quit 命令功能相同）

表6-5中的部分命令的命令行操作如下：

```
virsh # pwd
/root

virsh # cd /root/kvm_demo/

virsh # pwd
/root/kvm_demo
```

```
virsh # help
Grouped commands:

 Domain Management (help keyword 'domain'):
    attach-device                  attach device from an XML file
<!-- 省略百余行输出信息 -->
virsh # help list
  NAME
    list - list domains

  SYNOPSIS
<!-- 省略十余行输出信息 -->
virsh # echo "just for fun"
just for fun
virsh # quit

[root@rhel6u3-ga ~]#        # 这里已经退出了 virsh 的交互终端了
```

6.3 virt-manager

6.3.1 virt-manager简介

virt-manager是虚拟机管理器（Virtual Machine Manager）这个应用程序的缩写，也是该管理工具的软件包名称。virt-manager是用于管理虚拟机的图形化的桌面用户接口，目前仅支持在Linux或其他类Unix系统中运行。和libvirt、oVirt等类似，virt-manager是由Redhat公司发起的项目，在RHEL 6.x、Fedora、CentOS等Linux发行版中有较广泛的使用，当然在Ubuntu、Debian、OpenSuse等系统中也可以正常使用virt-manager。为了实现快速开发而不太多地降低程序运行性能的需求，virt-manager项目选择使用Python语言开发其应用程序部分，使用GNU AutoTools（包括autoconf、automake等工具）进行项目的构建。virt-manager是一个完全开源的软件，使用Linux界广泛采用的GNU GPL许可证发布。virt-manager依赖的一些程序库主要包括Python（用于应用程序逻辑部分的实现）、GTK+PyGTK（用于UI界面）和libvirt（用于底层的API）。

virt-manager工具在图形界面中实现了一些易用的和比较丰富的虚拟化管理功能，已经为用户提供的功能如下：

1）对虚拟机（即客户机）生命周期的管理，如创建、编辑、启动、暂停、恢复和停止虚拟机，还包括虚拟快照、动态迁移等功能。

2）运行中客户机的实时性能、资源利用率等的监控，统计结果的图形化展示。

3）对创建客户机的图形化的引导，对客户机的资源分配和虚拟硬件的配置和调整等功能也提供了图形化的支持。

4）内置了一个VNC客户端，可以用于连接到客户机的图形界面进行交互。

5）支持本地或远程管理KVM、Xen、QEMU、LXC等Hypervisor上的客户机。

在没有成熟的图形化的管理工具之时，由于需要记忆大量的命令行参数，QEMU/KVM的使用和学习曲线比较陡峭，常常让部分习惯于GUI界面的初学者望而却步。不过现在情况有所改观，已经出现了一些开源的、免费的、易用的图形化管理工具，可以用于KVM虚拟化管理。virt-manager作为KVM虚拟化管理工具中最易用的工具之一，其最新的版本已经提供了比较成熟的功能、易用的界面和不错的性能。对于习惯于图形界面或不需要了解KVM原理和qemu-kvm命令细节的部分读者来说，通过virt-manager工具来使用KVM也是一个不错的选择。

6.3.2 virt-manager编译和安装

virt-manager的源代码开发仓库是用Linux世界中著名的版本管理工具Git进行管理，使用autoconf、automake等工具进行构建。如果想从源代码编译和安装virt-manager，可以到其官方网站（http://virt-manager.org/download.html）下载最新发布的virt-manager源代码。或者使用Git工具克隆其开发中的代码仓库：git://git.fedorahosted.org/git/virt-manager.git。

virt-manager源代码的编译，与Linux下众多的开源项目类似，主要运行"./configure"、"make"、"make install"等几个命令分别进行配置、编译和安装即可。在3.3节、3.4节、6.1.2小节中分别介绍了对KVM内核、qemu-kvm、libvirt等开源项目的编译，这里不再详细演示virt-manager从源代码编译和安装的过程。

许多流行的Linux发行版（如RHEL、Fedora、Ubuntu等）中都提供了virt-manager软件包供用户自行安装。例如，在RHEL 6.3系统中，使用"yum install virt-manager"命令即可安装virt-manager的RPM软件包了，当然YUM工具也会检查并同时安装它所依赖的一些软件包，包括python、pygtk2、libvirt-python、libxml2-python、python-virtinst等。

6.3.3 virt-manager使用

在本小节中，将以RHEL 6.3（英文版）系统中的virt-manager 0.9.0版本为例来简单地介绍它的一些基本用法和技巧，读者可以参考本章末尾的“参考阅读”部分的资料来进行深入学习。

1.在RHEL6.3中打开virt-manager

在本节的示例系统中，查看virt-manager的版本，命令行操作如下：

```
[root@rhel6u3-ga ~]# rpm -qa | grep virt-manager
virt-manager-0.9.0-14.el6.x86_64
```

登录到RHEL 6.3的图形用户界面中，用鼠标选择"Applications->System Tools->Virtual Machine Manager"即可打开virt-manager的使用界面。

也可以在桌面系统的终端（terminal）中直接运行"virt-manager"命令来打开virt-manager管理界面，而且使用该命令还可以像virsh那样添加"-c URI"参数来指定启动时连接到本地或远程的Hypervisor，在没有带"-c URI"参数时，默认连接到本地的Hypervisor。对于远程连接，当然需要用户名密码的验证或使用数字证书的验证后才能建立连接，实现远程管理。在图形界面的终端中用

命令行启动virt-manager并远程连接到某个Hypervisor，命令行示例如下：

```
virt-manager -c qemu+ssh://192.168.158.31/system
```

在RHEL 6.3中启动virt-manager，其管理界面如图6-3所示。在此图中，virt-manager默认连接到了本地的QEMU/KVM上，可以看到有两个客户机（rhel6u3-1、rhel6u3-2），在客户机名称的右边是客户机CPU使用率统计的图形展示。

图　6-3　RHEL6.3中的virt-manager界面

另外，还可以在本地图形界面终端上通过"ssh-X remoge-host"命令连接到远程主机并开启了ssh中的X11转发，然后可以在本机终端上

直接运行"virt-manager"命令来使用远程主机上的virt-manager工具。这种使用方式的命令行操作如下：

```
[root@rhel6u3-ga ~]# ssh -X 192.168.158.31
root@192.168.158.31's password:
Last login: Sat Dec 15 11:18:37 2012 from 192.168.185.145
[root@remote-host ~]# virt-manager
```

2.创建一个客户机

在如图6-3所示的virt-manager界面中，选择"File"菜单下方的那个带有“创建一个新的虚拟机”的标识的按钮，即可进入到创建客户机流程的向导之中。

根据virt-manager中的向导指引，可以比较方便地创建一个客户机。主要需要输入或选择一些必要的设置项目，包括：客户机名称、安装介质的选择（如本地的ISO文件）、客户机类型和版本、虚拟CPU个数、内存大小、创建的磁盘镜像文件的空间大小和是否立即分配空间，虚拟网络的配置，等等。在设置了这些必要的步骤之后，最后一步是一个确认的界面，如图6-4所示，点击其中的"Finish"按钮确认即可。确认之后，virt-manager会自动连接到客户机中，客户机系统启动进入到普通的安装流程，这之后的安装过程就和在非虚拟化环境中安装操作系统的过程完全一样了。

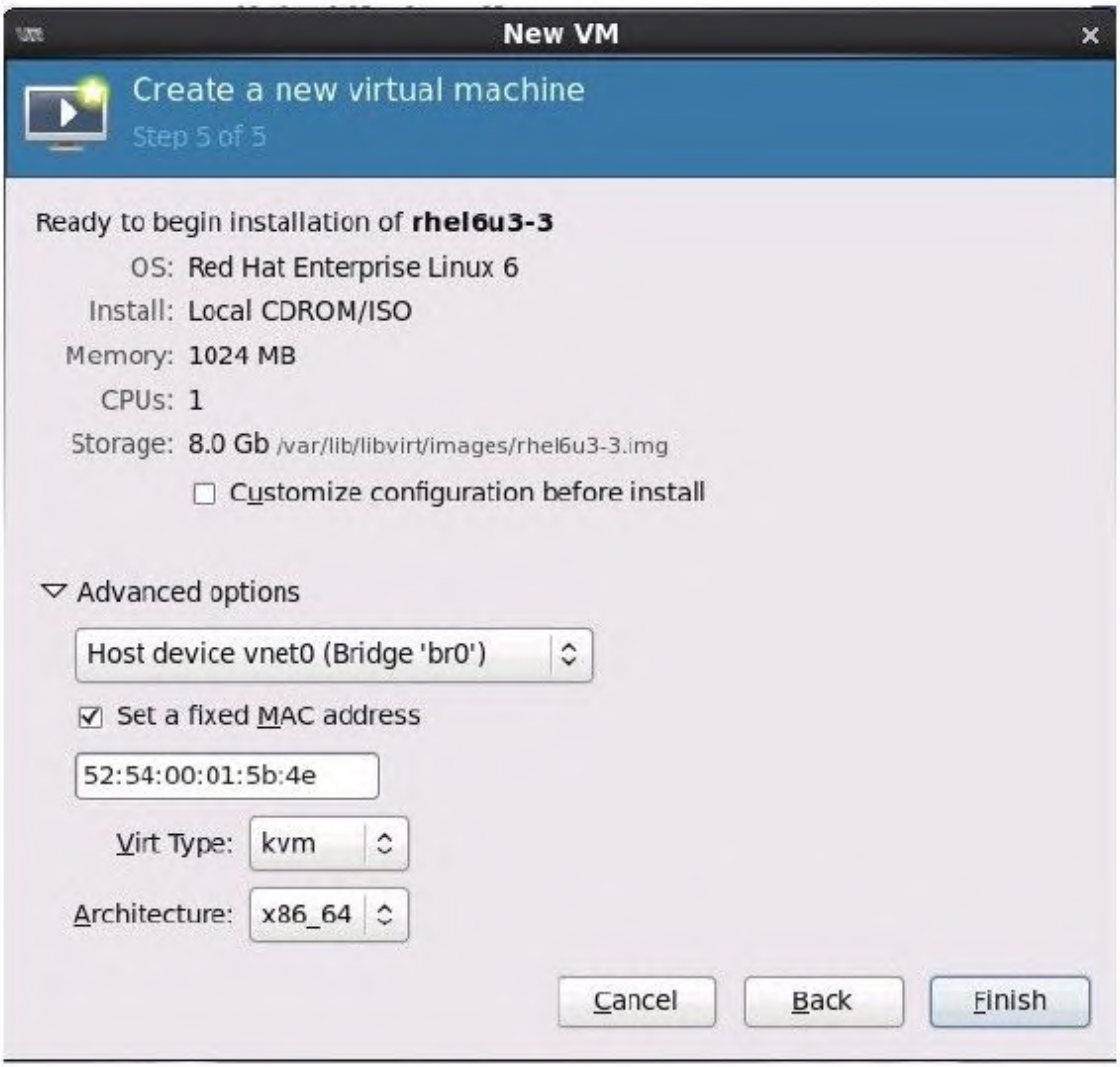


图 6-4 virt-manager中创建客户机的最后一个步骤

在图6-4中点击"Finish"完成配置之后，libvirt和virt-manager工具会默认创建以客户机名称来命令的客户机的XML配置文件和磁盘镜像文件，查看XML配置和镜像文件的命令行如下：

```
[root@rhel6u3-ga ~]# ls /etc/libvirt/qemu/rhel6u3-3.xml
/etc/libvirt/qemu/rhel6u3-3.xml
[root@rhel6u3-ga ~]# ls /var/lib/libvirt/images/rhel6u3-3.img
/var/lib/libvirt/images/rhel6u3-3.img
```

3. 启动、暂停和关闭一个客户机

在virt-manager中，处于关机状态的客户机的状态标识为"Shutoff"，如图6-5所示，选中了一个处于关机状态的客户机，然后点击管理界面上方的开机按钮（即图中的三角符号按钮）即可让该客户机开机。双击已经开启的客户机，可以连接到客户机的图形界面的控制台，然后可以登录到客户机中正常使用客户机系统。

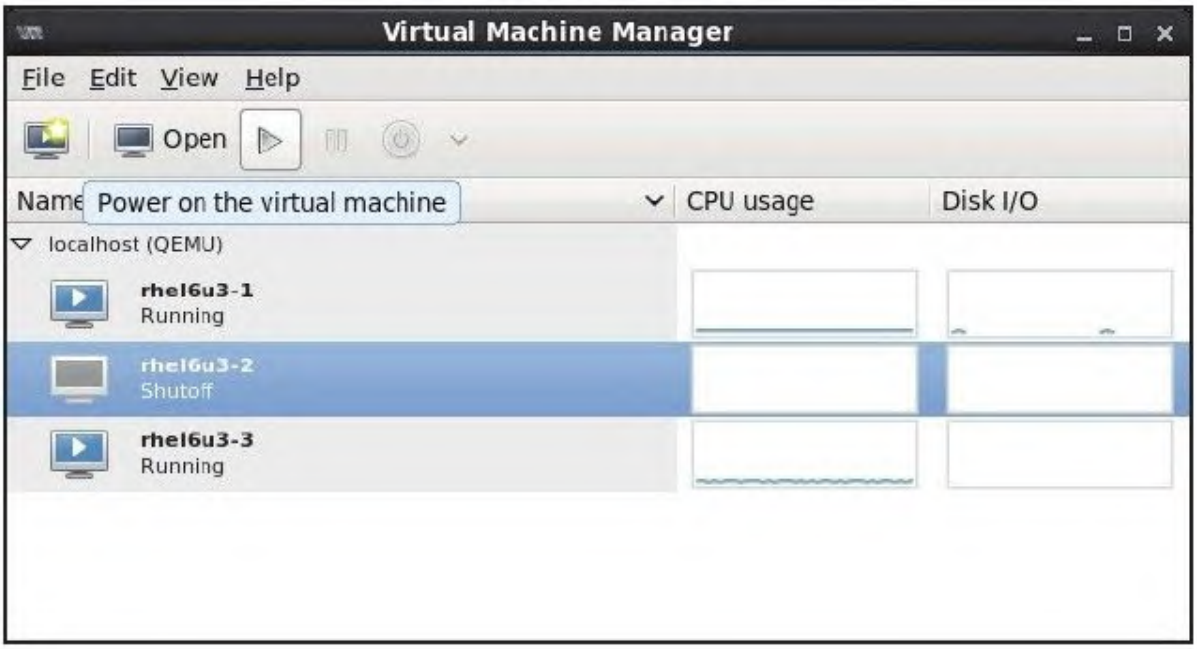


图　6-5　virt-manager中的开机按钮

除了上面的开机方法之外，还可以双击某个客户机进入到对该客户机的管理窗口中，这里同样有开机的按钮。如果一个客户机处于正常的开机运行状态，那么virt-manager可以选择对其进行暂停和关机等操作。在某个客户机的管理窗口中，如图6-6所示，可以选

择"shutdown"选项来关机，也可以选择"Reboot"来重启，选择"Force Off"来强制关机，选择"Save"来保存客户机的当前运行状态。注意，一般尽量避免使用"Force Off"来实施强制关机，因为它可能损坏客户机的磁盘镜像，一般是在客户机完全失去响应而不能正常关机时才采用强制关机的方式。

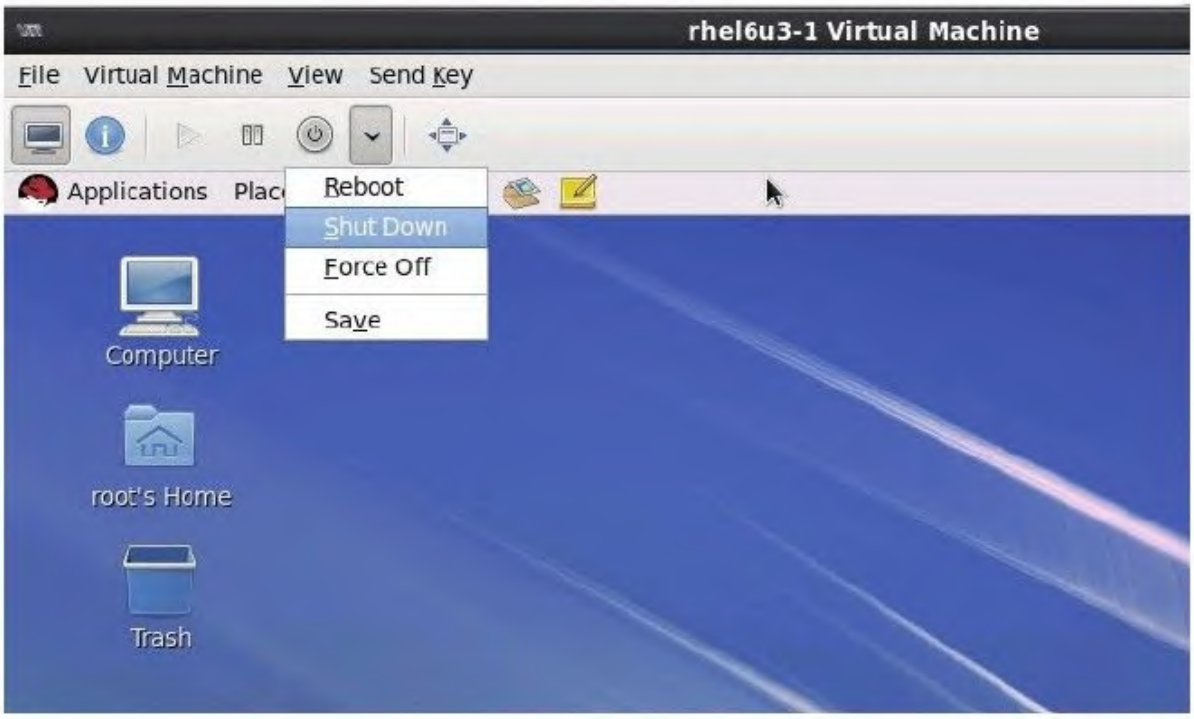


图　6-6　virt-manager中的关机选项

4.连接到本地和远程的Hypervisor

在一般情况下，启动virt-manager时会默认通过libvirt API试图连接本地的Hypervisor，如果libvirtd守护进程没有在运行，则会有连接失败的错误提示。在重启libvirtd服务之后，需要在virt-

manager中重新建立连接，否则连接处于未连接（Not Connected）状态。

在virt-manager系统界面中建立一个到本地或远程主机的连接，可以选择"File"-->"Add Connection"菜单即可。如图6-7所示，选择相应的Hypervisor（QEMU/KVM、Xen、LXC等），选择远程连接方式（SSH、TCP、TLS等），填写好用户名、主机名（或IP地址），然后点击"Connect"按钮，经过密码或证书验证之后，即可建立好到远程主机的连接。

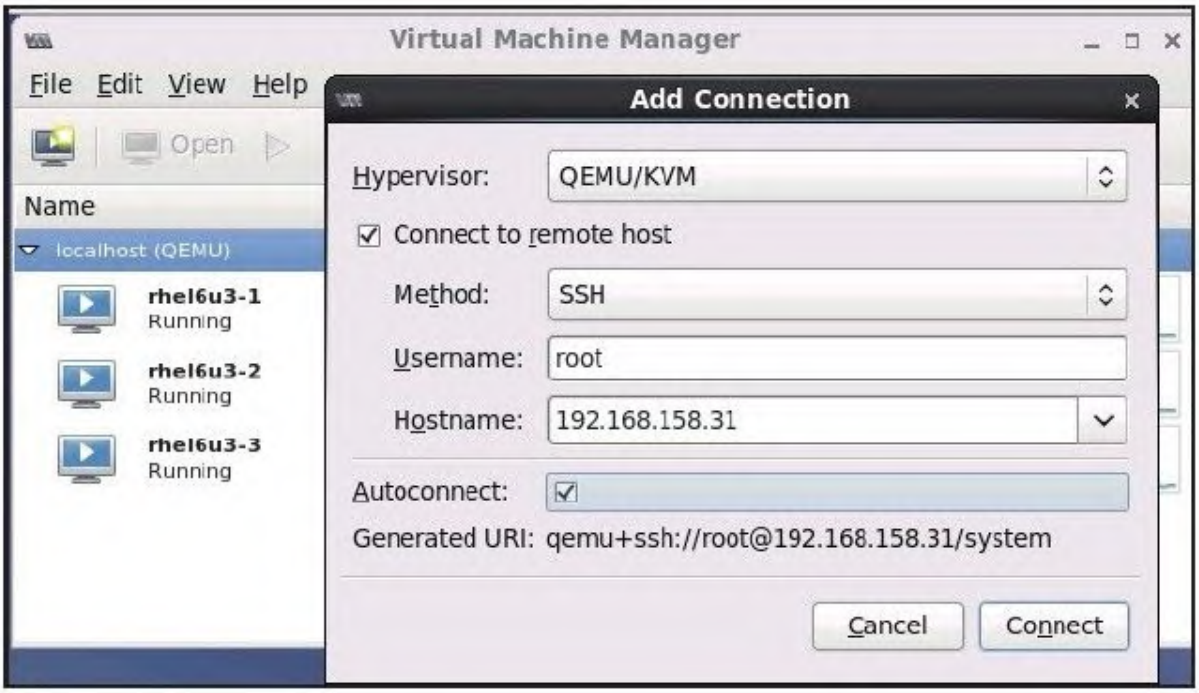


图　6-7　在virt-manager中建立到远程主机的连接

建立了远程连接后，在virt-manager的主界面中可以看到远程主机和本地主机上运行着的客户机，然后可以对其进行相应的管理，如图6-8所示。

图　6-8　在virt-manager中管理本地和远程主机

5. 查看和修改客户机的详细配置

在virt-manager的主界面中，双击某个客户机标识，可以进入到这个客户机的详细界面，在此处点击virt-manager窗口上工具栏中的提示信息为"Show virtual hardware details"的图标，即可进入到该客户机的虚拟硬件的详细设置界面。如图6-9所示，在客户机硬件详细配置界面中，可以查看也可以修改该客户机的配置参数。

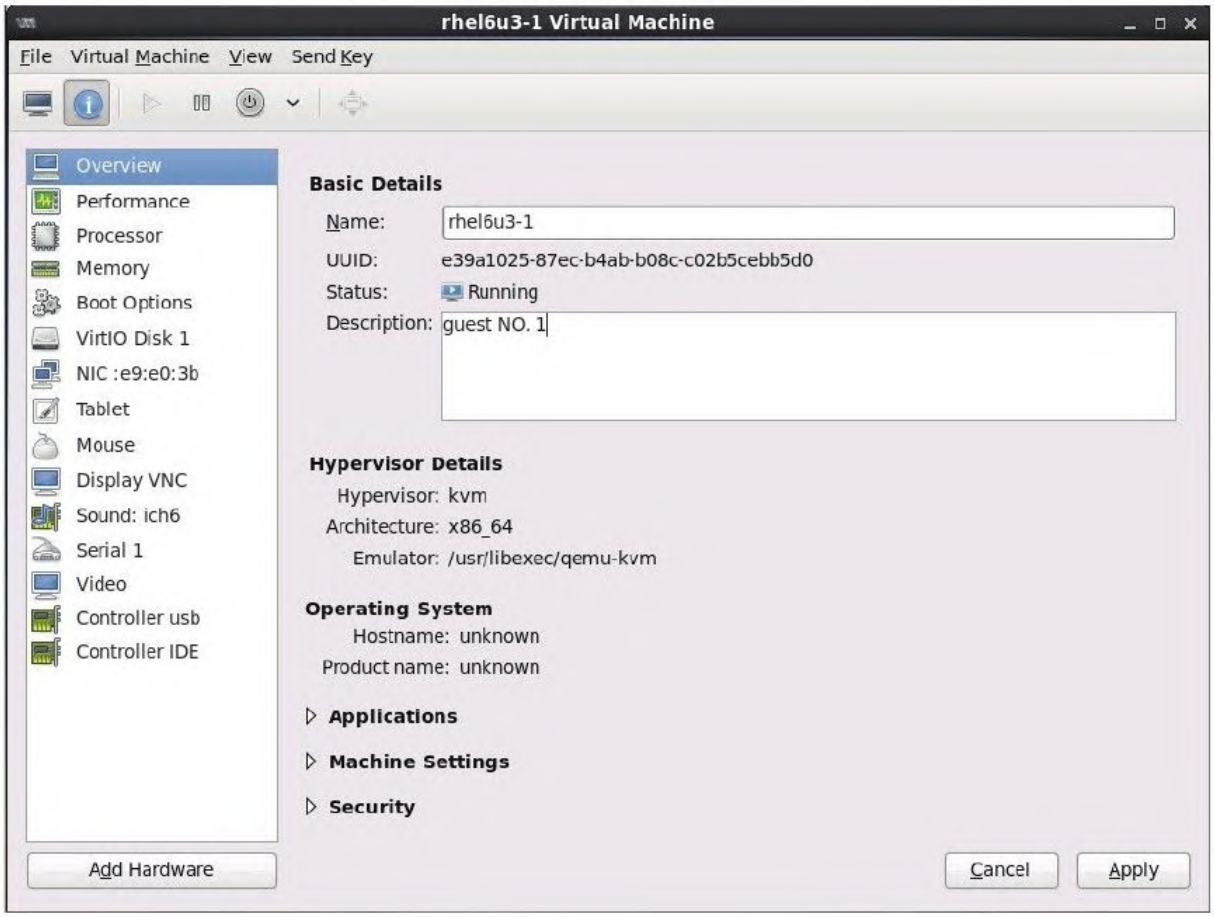


图 6-9 virt-manager中客户机的详细配置

图6-9中的对客户机详细配置的设置，包括对客户机的名称、描述信息、处理器、内存、启动选项、磁盘、网卡、鼠标、VNC显示、声卡、显卡、串口等许多信息的配置。根据界面上的提示，设置起来还是比较方便的，这里不再详细介绍它们的配置。

需要注意的是，这里的详细设置都会在libvirt管理的该客户机的XML配置文件中表现出来，对运行中的客户机的设置，并不能立即生效，而是在重启客户机后才会生效。

6.动态迁移

在KVM虚拟环境中，如果遇到宿主机负载过高或需要升级宿主机硬件等需求时，可以选择将部分或全部客户机动态迁移到其他的宿主机上去继续运行。virt-manager也提供了这个功能，而且可以方便使用图形界面的方式进行操作。不过，在做动态迁移之前，一般来说（如对RHEL 6.3系统中的libvirt和virt-manager来说），需要满足如下前提条件才能使动态迁移成功实施。

1）源宿主机和目的宿主机使用共享存储，如NFS、iSCSI、基于光纤通道的LUN、GFS2等，而且它们挂载共享存储到本地的挂载路径需要完全一致，被迁移的客户机就是使用该共享存储上的镜像文件。

2）硬件平台和libvirt软件的版本要尽可能的一致，如果软硬件平台差异较大，可能会增加动态迁移失败的概率。

3）源宿主机和目的宿主机的网络通畅并且打开了对应的端口。

4）源宿主机和目的宿主机必须要有相同的网络配置，否则可能出现动态迁移之后客户机的网络不能正常工作的情况。

5）如果客户机使用了和源宿主机建立桥接的方式获得网络，那么只能在同一个局域网（LAN）中进行迁移，否则客户机在迁移后，其网络将无法正常工作。

在准备配置环境且满足上面的所有前提条件后，在virt-manager中可以完成动态迁移。在本节示例中，源宿主机IP地址为192.168.185.145，目的主机IP地址为192.168.158.31，被动态迁移的客户机名称为"guest-demo"，下面介绍动态迁移过程中的关键操作步骤。

（1）在目的宿主机的virt-manager中查看当前目的宿主机上没有运行任何客户机，如图6-10所示。

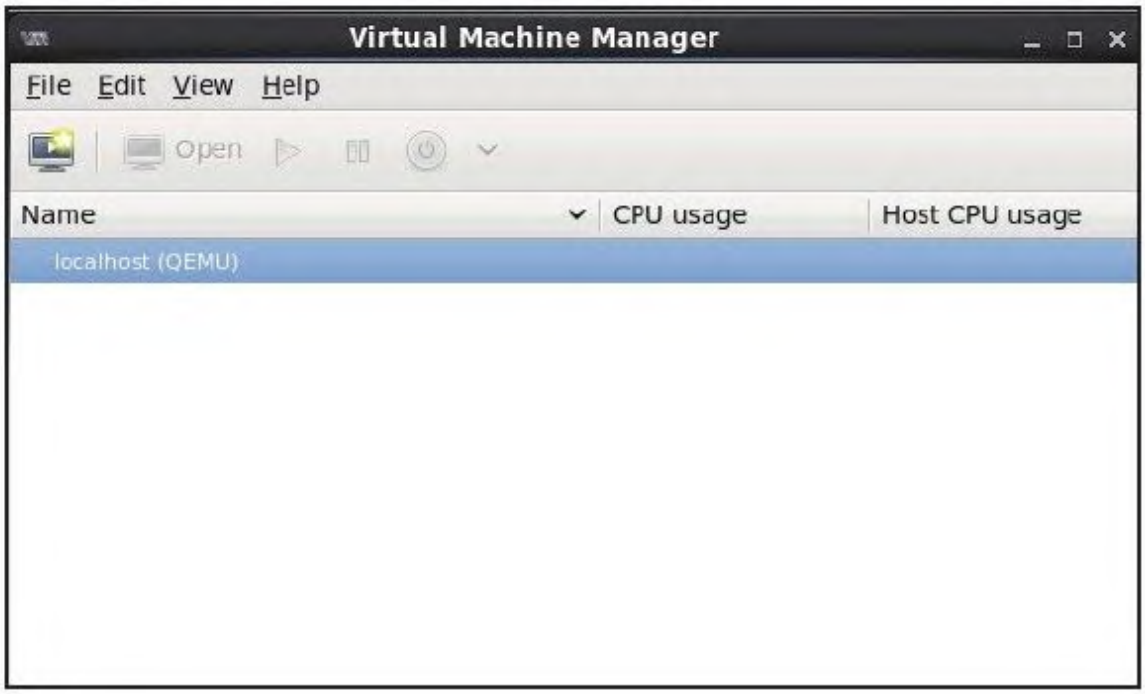


图 6-10 在动态迁移前，目的宿主机上没有运行客户机

（2）在源宿主机的virt-manager管理界面中，选中名为"guest-demo"的客户机，右键单击可以看到"Migrate"选项，如图6-11所示，

该选项就实现动态迁移的功能。

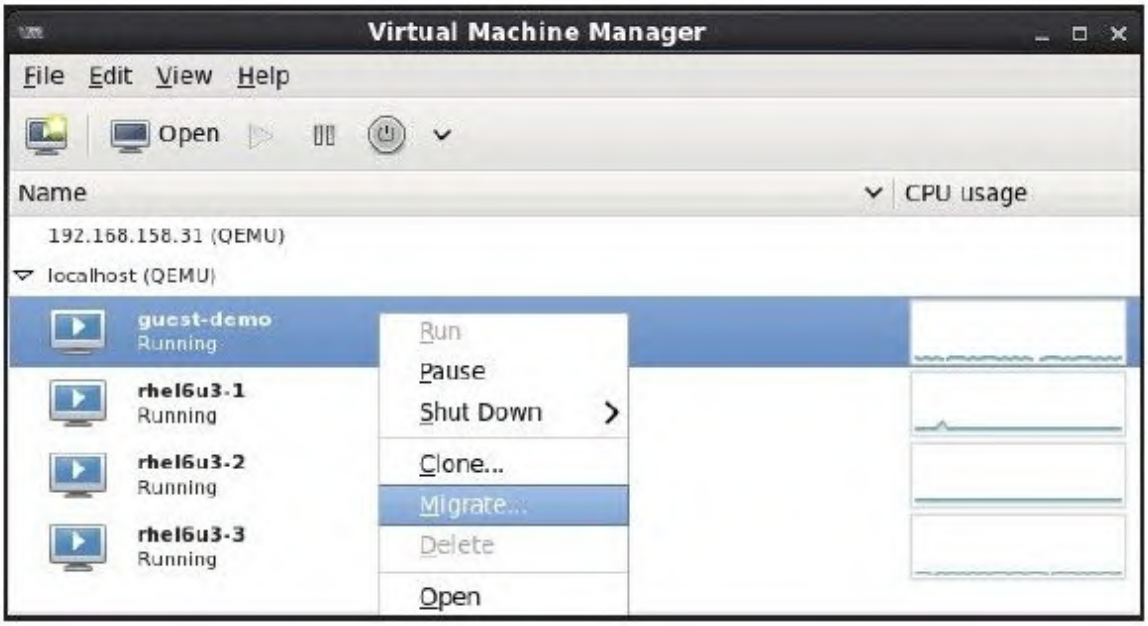


图　6-11　选择"Migrate"选项

选择"Migrate"选项进入到与动态迁移相关的选项设置界面，如图6-12所示，填写目的宿主机的IP或主机名，Max downtime、Port、Bandwith等选项，可以不填写，一般使用默认值即可。

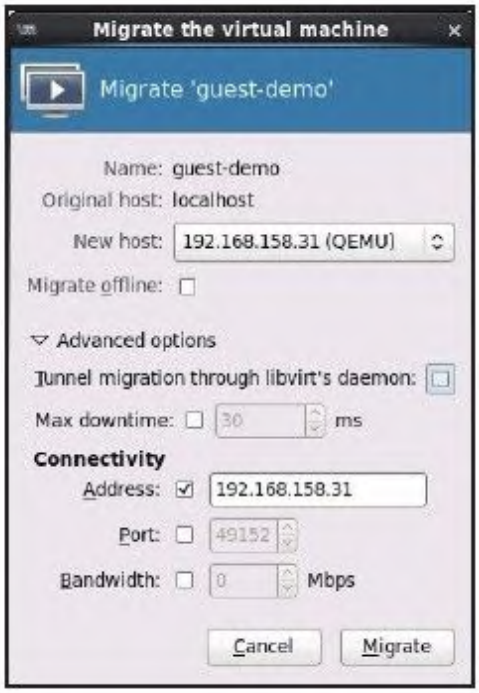


图　6-12　virt-manager中动态迁移选项的配置

填写好动态迁移的配置信息后，点击"Migrate"按钮即进入到正式的动态迁移过程，这里可能需要输入目的宿主机的root用户的密码，密码验证成功之后，将会出现如图6-13所示的动态迁移的进度图。

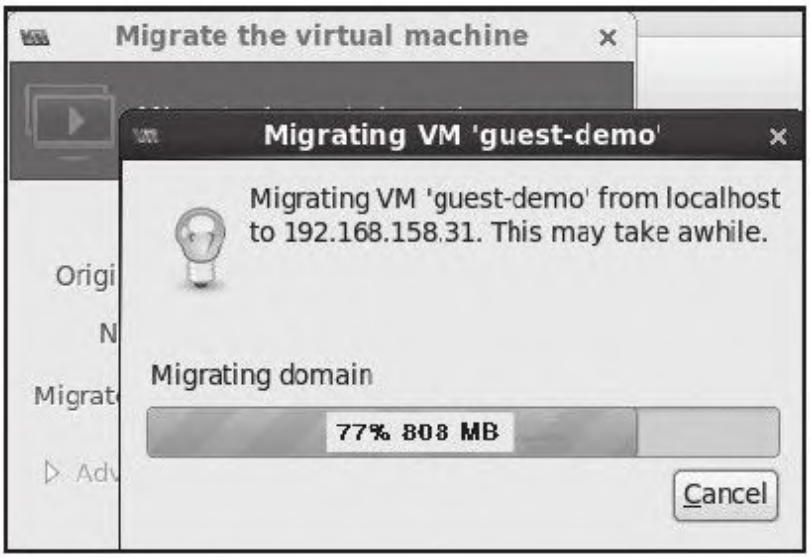


图　6-13　virt-manager中动态迁移的进度展示

（3）待动态迁移的进度完成之后，在目的客户机的virt-manager主界面中可以看到动态迁移过来的"guest-demo"客户机系统，如图6-14所示。如果登录到该客户机系统，该系统是可以正常使用的（包括网络配置），就说明了本示例的动态迁移是成功的。

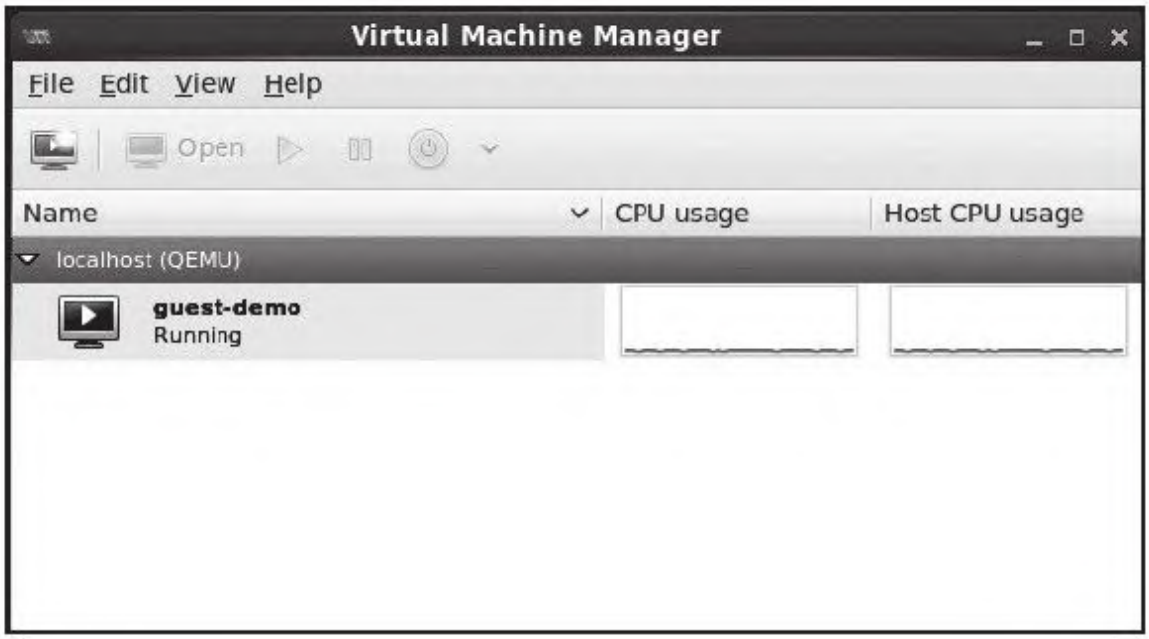


图 6-14 动态迁移完成后，在目的宿主机上查看客户机

7.性能统计图形界面

virt-manager还提供了对宿主机和客户机的资源使用的监控，如RHEL 6.3中的virt-manager就提供了对宿主机CPU利用率、客户机CPU利用率、客户机的磁盘I/O、客户机的网络I/O等项目的图形化监控。

在virt-manager中，对资源监控频率和保存的采样数量的设置位于主界面的"Edit"→"Preferences"→"Stats"标签中，如图6-15所示。

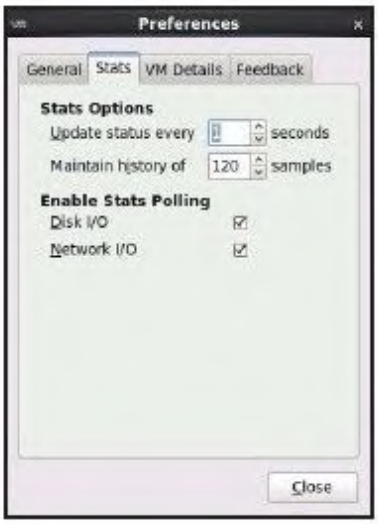


图　6-15　virt-manager中的监控参数的配置

在"View"→"Graph"中，可以选择哪些被监控的资源是需要显示在界面上的，如图6-16所示。如果发现"Disk I/O"和"Network I/O"两个项目不可选，则需要检查图6-15中的是否开启了对磁盘和网络I/O的轮询。

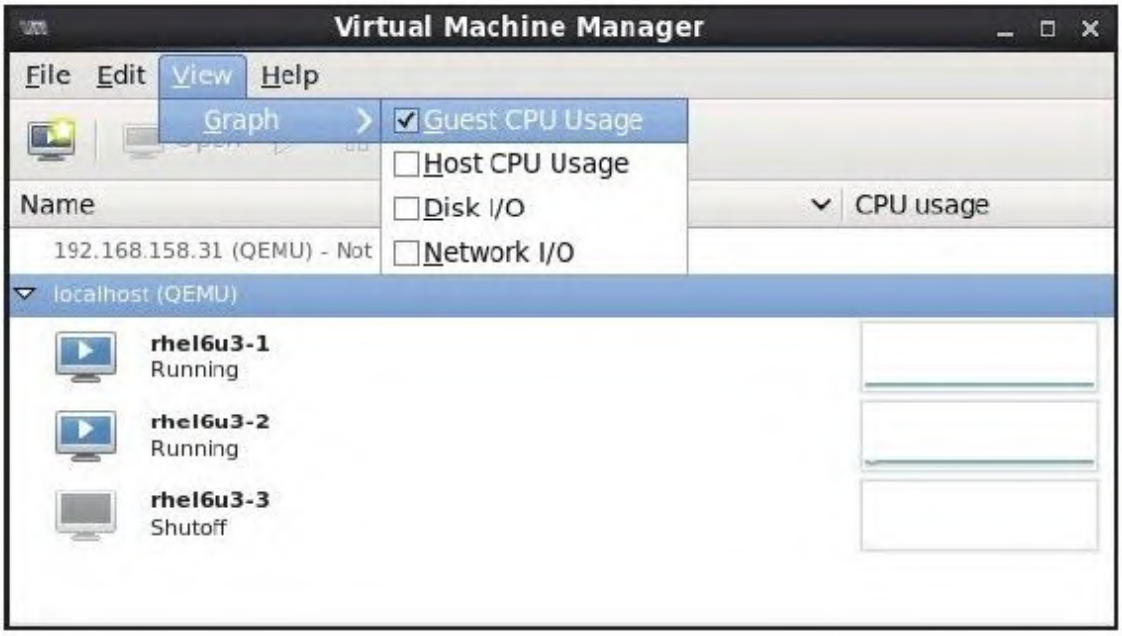


图　6-16　选择需要显示到资源监控图

选中了所有监控图，让其显示在virt-manager中，就可以尽可能多地通过图形化的方式了解客户机和宿主机的资源使用率，如图6-17所示。

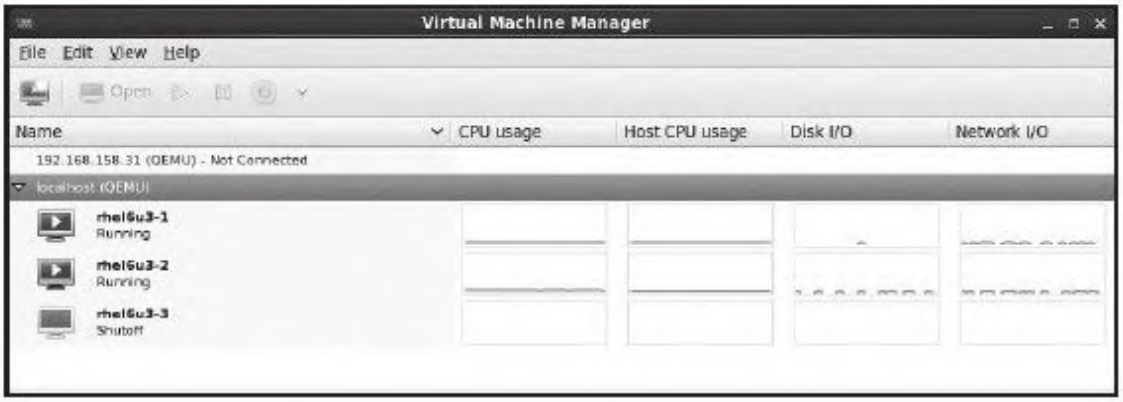


图　6-17　virt-manager中对客户机和宿主机资源的监控

6.4 virt-viewer、virt-install和virt-top

6.4.1 virt-viewer

virt-viewer是"Virtual Machine Viewer"（虚拟机查看器）工具的软件包和命令行工具名称，它是一个用于与虚拟化客户机的图形显示的轻量级的交互接口工具。virt-viewer使用GTK-VNC作为它的显示能力，使用libvirt API去查询客户机的VNC服务器端的信息。virt-viewer经常用于去掉传统的VNC客户端查看器，因为后者通常不支持x509认证授权的SSL/TLS加密，而对virt-viewer是支持的。在virt-manager中查看客户机图形界面进行交互之时，其实已经间接地使用过virt-viewer工具了。

在RHEL 6.3系统中查看virt-viewer的RPM包信息，命令行如下：

```
[root@rhel6u3-ga ~]# rpm -q virt-viewer
virt-viewer-0.5.2-9.el6.x86_64
```

virt-viewer的使用语法为：

```
virt-viewer [OPTIONS] DOMAIN-NAME|ID|UUID
```

virt-viewer连接到的客户机可以通过客户机的名称、域ID、UUID等表示来唯一指定。virt-viewer还支持"-c URI"或"--connection

URI"参数来指定连接到远程宿主机上的一个客户机，当然远程连接需要的一些必要的认证还是必须的。关于virt-viewer工具更多详细的参数和解释，可以通过"man virt-viewer"命令查看使用手册。

在图形界面的一个终端中，用"virt-viewer rhel6u3-1"连接到本地宿主机上名为"rhel6u3-1"的客户机，其显示效果如图6-18所示。

图　6-18　virt-viewer连接到本地的一个客户机

在virt-viewer打开的客户机窗口中（图6-18），其工具栏的"File"菜单下有保存屏幕快照的功能，"View"菜单下有使用全屏和放大（或缩小）屏幕的功能，"Send key"菜单下可以向客户机发送一些特殊的按键（如Ctrl+Alt+Del、Ctrl+Alt+F2等）。

6.4.2 virt-install

virt-install是"Virt Install"工具的命令名称，其软件包名为virtinst（或python-virtinst）。virt-install命令行工具为虚拟客户机的安装提供了一个便捷易用的方式，它也是libvirt API来创建KVM、Xen、LXC等上面的客户机，同时，它也为virt-manager的图形界面创建客户机提供了安装系统的API。virt-install工具使用文本模式的串口控制台和VNC（或SDL）图形接口，可以支持基于文本模式和图形界面的客户机安装。virt-install中使用到的安装介质（如光盘、ISO文件）可以存放在本地系统上，也可以存放在远程的NFS、HTTP、FTP服务器上。virt-install支持本地的客户机系统，也可以通过"--connect URI"（或"-c URI"）参数来支持在远程宿主机中安装客户机。使用virt-install中的一些选项（--initrd-inject、--extra-args等）和kickstart[4] 文件，可以实现无人值守的自动化安装客户机系统。

在RHEL中virt-install工具存在于"python-virtinst" RPM包中，查询的命令行如下：

```
[root@rhel6u3-ga ~]# rpm -q python-virtinst
python-virtinst-0.600.0-8.el6.noarch
```

使用virt-install命令启动一个客户机的安装过程，命令行操作如下：

```
[root@rhel6u3-ga kvm_demo]# virt-install --connect qemu:///system
--name rhel6u3-4 --ram 1024 --disk
path=/var/lib/libvirt/images/rhel6u3-4.img,size=8 --network
network:default --cdrom /var/lib/libvirt/images/RHEL6.3.iso
--os-type=linux --accelerate --vnc


Starting install...
Allocating 'rhel6u3-4.img'                                | 8.0 GB     00:00
Creating domain...                                        |    0 B     00:00
```

上面virt-install的命令行参数中，"--connect"用于连接到本地或远程的Hypervisor（无该参数时，默认连接本地Hypervisor）；"--disk"用于配置客户机的磁盘镜像文件，其中的path属性表示路径，size属性表示磁盘大小（默认单位为GB）；"--cdrom"用于指定用于安装的ISO光盘镜像文件；"--os-type=linux"表示客户机是Linux系统（virt-install会根据这个值自动优化一些安装配置）；"--accelerate"表示使用KVM来加速QEMU客户机的安装（一般都添加这个参数）；"--graphics vnc"表示使用VNC图形接口来连接到客户机的控制台。关于virt-install工具的更多更详细参数配置可以参考通过"man virt-install"命令查看到相应的帮助文档。

在示例中使用VNC接口连接到客户机，会默认用virt-viewer自动打开客户机的控制台，如图6-19所示。

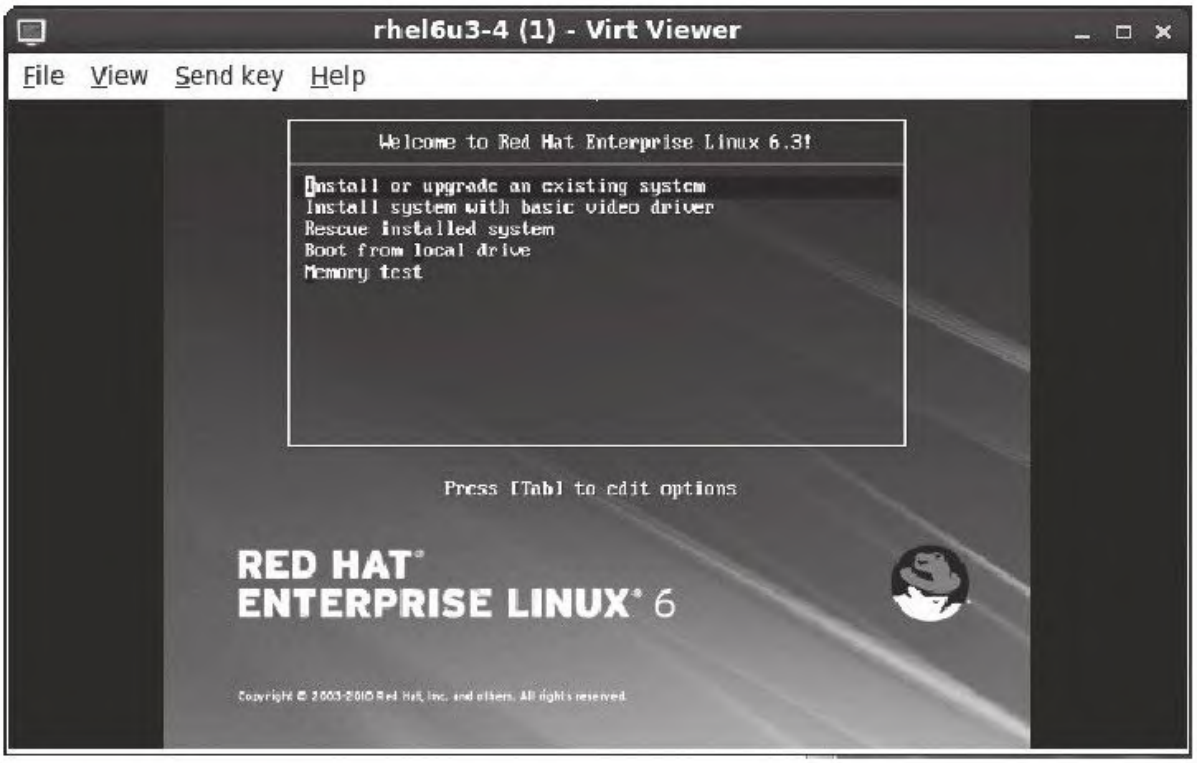


图 6-19 virt-install在安装一个RHEL6.3客户机

6.4.3 virt-top

virt-top是一个用于展示虚拟化客户机运行状态和资源使用率的工具，它和Linux系统上常用的"top"工具类似，而且它的许多快捷键和命令行参数的设置都与"top"工具的相同。virt-top也是使用libvirt API来获取客户机的运行状态和资源使用情况的，所以也是只要libvirt支持的Hypervisor，就可以用virt-top监控该Hypervisor上的客户机状态。

在RHEL 6.3系统上，virt-top命令就是在名为virt-top的RPM包中用命令行查看：

```
[root@rhel6u3-ga ~]# rpm -q virt-top
virt-top-1.0.4-3.13.el6.x86_64
```

直接运行"virt-top"命令后，将会显示出当前宿主机上各个客户机的运行情况，如图6-20所示，其中包括宿主机的CPU、内存的总数，也包括各个客户机的运行状态、CPU、内存的使用率。关于virt-top工具的更多更详细参数配置可以参考通过"man virt-top"命令查看到的相应的帮助文档。

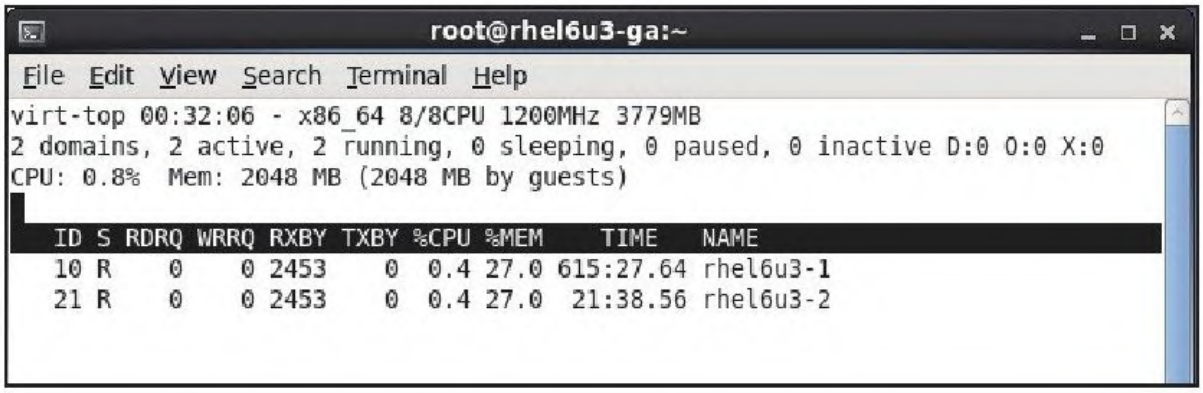


图 6-20 运行"virt-top"命令查看客户机的状态和资源利用率

6.5 OpenStack云计算平台

6.5.1 OpenStack简介

在计算设备（包括PC、智能手机、平板电脑等）、互联网技术非常普及的今天，从技术热点上看，云计算无疑是其中最热门的概念之一。公共云计算服务的主要优势是，向用户提供按需付费的弹性的计算能力，以及简化软硬件计算环境的搭建，让用户更专注于自身的计算任务或应用程序的开发。而私有云计算服务的主要优势是，让一个公司或组织的计算资源得到充分整合，从而实现按需分配计算资源而提升现有硬件资源的利用率，而且更加方便管理，减低资源管理成本。

云计算的强烈需求，一方面推动了包括VMware、KVM、Xen等虚拟化技术的迅速发展，另一方面也促进云计算管理平台的产生和发展。在众多的开源云计算平台中，OpenStack、CloudStack、CloudFoundry、OpenNebula、Eucalyptus等无疑是其中的佼佼者，本节将简单介绍其中的OpenStack云计算平台。

OpenStack是一个开源的基础架构即服务（IaaS）云计算平台，可以为共有云和私有云服务提供云计算基础架构平台。OpenStack使用的开发语言是Python，采用Apache许可证发布项目源代码。OpenStack支

持多种不同的Hypervisor（如QEMU/KVM、Xen、VMware、Hyper-V、LXC等），通过调用各个底层Hypervisor的API来实现对客户机的创建和关闭等操作，使用libvirt API来管理QEMU/KVM和LXC、使用XenAPI来管理XenServer/XCP、使用VMwareAPI来管理VMware，等等。OpenStack开源项目是在2010年由Rackspace公司和美国国家航空航天局（NASA）发起的云计算项目。OpenStack项目在这两年发展得非常快，目前，有超过150家公司和成千上万的个人开发者已经宣布加入到该项目的开发。在支持OpenStack开发的一些大公司中，包括了AT＆T、Canonical、IBM、HP、Redhat、Suse、Intel、Cisco、WMware、Yahoo!、新浪、华为等一批在IT业界非常知名的公司。

OpenStack的使命是为大规模的共有云和小规模的私有云都提供一个易于扩展的、弹性云计算服务，从而让云计算的实现更加简单、云计算架构具有更好的扩展性。也可以说，OpenStack是一个云计算操作系统，它给用于提供仅仅通过一个使用Web交互接口的控制面板（Dashboard）来管理一个或多个数据中心的所有计算资源池、存储资源池、网络资源池等硬件资源。OpenStack的作用是整合各种底层硬件硬件资源，为系统管理员提供Web界面的控制面板以方便资源管理，为开发者的应用程序提供统一管理接口，为终端用户提供无缝的透明的云计算服务。OpenStack在云计算软硬件架构的主要作用与一个操作系统类似，如图6-21所示（该图来源于OpenStack的官方网站）。

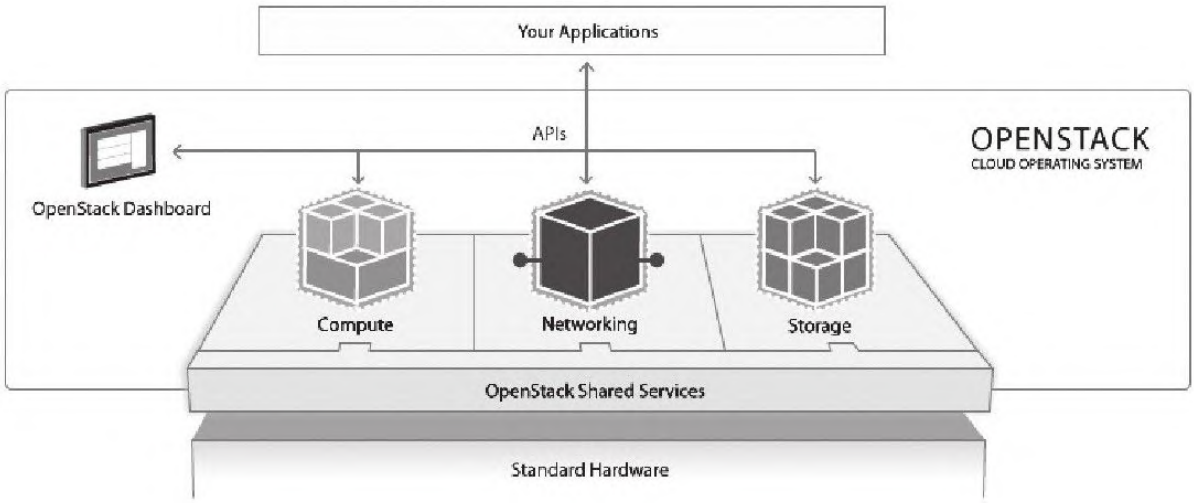


图　6-21　OpenStack在云计算架构中的位置

OpenStack项目从2010年10月开始，大约每半年正式发布一个版本，本书写作时，OpenStack的最新版本是在2012年9月27日发布的代号为"Folsom"的版本，下一个版本计划在2013年4月发布。在目前最新的OpenStack（代码"Folsom"）中，主要包含了7个核心组件：计算、对象存储、块存储、镜像、网络、身份认证、控制面板。在OpenStack中，这7个核心组件的逻辑架构，如图6-22所示（来源于OpenStack官方网站上的管理员手册[5]）。

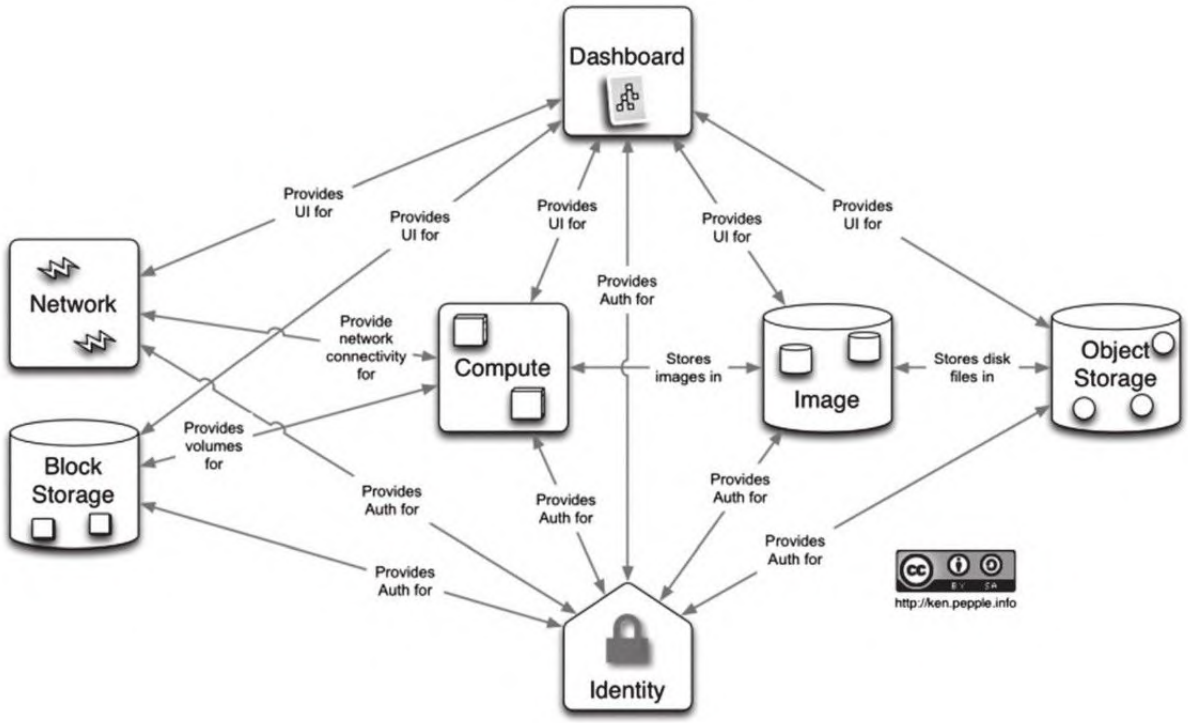


图　6-22　OpenStack中7个核心组件的逻辑架构

（1）计算（Compute，代号为"Nova"）

根据需求提供虚拟的服务器。Rackspace和HP公司提供商业云计算服务正是建立在Nova之上，在Mercado Libre和NASA（Nova项目的起源地）内部也是使用的Nova。

（2）对象存储（Object Storage，代号为"Swift"）

提供的对象存储服务，允许对文件进行存储或者检索（但不通过挂载文件服务器上目录的方式来实现）。目前已经有好几家公司开始提供基于Swift的商业存储服务，这些公司包括KT公司、Rackspace公

司（Swift项目的发源地）和Internap公司，而且，有很多大公司内部也使用Swift来存储数据。

（3）块存储（Block Storage，代号为"Cinder"）

为虚拟化的客户机提供持久化的块存储服务。该组件项目的很多代码最初来自于Nova之中（就是the nova-volume service）。不过要注意，这是块存储（或者volumes），而不是类似于NFS或CIFS文件系统，Cinder是在最新的"Folsom"版本OpenStack中才加入的一个全新的项目。

（4）镜像（Image，代号为"Glance"）

提供了一个虚拟磁盘镜像的目录和存储仓库，可以提供对虚拟机镜像的存储和检索。这些磁盘镜像常常广泛应用于OpenStack Compute组件之中。虽然这种服务在技术上是属于可选的，但任何规模的云都可能对该服务有需求。

（5）网络（Network，代号为"Quantum"）

在接口设备之间提供“网络连接即服务”的服务，而这些接口设备主要是由OpenStack的其他服务（如Nova）进行管理的。该服务允许用户创建自己的网络，然后添加网络接口设备。Quantum提供了一个可

插拔的体系架构，使其能够支持很多流行的网络供应商和新的网络技术。Quantum是在OpenStack的Folsom版本中才加入的新项目。

（6）身份认证（Identity，代号为"Keystone"）

为OpenStack上的所有服务提供身份验证和授权。它还提供了在特定OpenStack云服务上运行的服务的一个目录。

（7）控制面板（Dashboard，代号为"Horizon"）

为OpenStack的所有服务提供一个模块化的基于Web的用户界面。使用这个Web图形界面，可以完成云计算平台上的大多数的操作，如启动客户机、分配IP地址、设置访问控制权限等。

除了上面介绍的7个核心组件，OpenStack计划在2013年4月发布代号为"Grizzly"的版本中，加入下面的两个组件。其中，一个是代号为"Ceilometer"的项目，用于对用户实际使用资源的进行比较细粒度的度量，可以为计费系统提供非常详细的资源监控数据（包括CPU、内存、网络、磁盘等）。另一个是代号为"Heat"的项目，使用Amazon的AWS云格式（CouldFormation）模板来编排和描述OpenStack中的各种资源（包括客户机、动态IP、存储卷等），它提供了一套OpenStack故有的RESTful的API，和一套与AWS CloudFormation兼容的查询API。

6.5.2　使用DevStack安装和配置OpenStack开发环境

OpenStack的安装和配置有一点复杂，特别是对于初学者，第一次安装OpenStack时经常会碰到很多的问题。不过在Openstack社区中，一些开发者开发了一些自动化脚本来方便搭建OpenStack的开发环境，DevStack是其中相对比较完善的，也是OpenStack官方网站推荐的用于了解OpenStack时搭建环境的方法。

DevStack是由Rackspace公司开源的一套用Bash编写的自动化脚本，用于为OpenStack构建和配置一整套完整的开发环境。DevStack的官方网站为http://devstack.org/，可以到该网站上查看DevStack脚本的使用方法及其工作原理。

目前，DevStack对Ubuntu 12.04和Fedora 16及以上的Ubuntu和Fedora系统有比较完善的支持，选择在其他操作系统上使用DevStack可能需要做一些修改才能正常工作。所以，这里的示例以在Ubuntu 12.04系统上使用DevStack脚本来搭建OpenStack开发环境，其过程非常简单，只分为如下两个步骤。

1）下载DevStack的代码仓库，命令行如下：

```
git clone git://github.com/openstack-dev/devstack.git
```

2）运行stack.sh脚本，命令行操作如下：

```
jay@my-ubuntu:~$ cd devstack/
jay@my-ubuntu:~/devstack$ ./stack.sh
<!-- 此处省略其余上万行输出信息 -->
Horizon is now available at http://192.168.158.31/
Keystone is serving at http://192.168.158.31:5000/v2.0/
Examples on using novaclient command line is in exercise.sh
The default users are: admin and demo
The password: admin
This is your host ip: 192.168.158.31
stack.sh completed in 1663 seconds.
# 看到最后这样的信息，说明 OpenStack 已经成功安装和配置
```

由于stack.sh脚本会自动下载大量的被依赖的软件包和OpenStack源代码，还要进行各种配置文件和数据库的初始化，所以第一次执行stack.sh脚本比较花费时间，大约需要几十分钟到一个小时不等（根据处理器速度和网络带宽的不同而不同）。

在Ubuntu系统中运行stack.sh脚本时，因为需要经常使用"sudo"命令去安装软件包和写配置文件，所以可能会提示输入当前用户的密码；因为需要配置一些数据库、OpenStack的认证，所以可能会有一些数据库用户名、密码的确认需要与用户进行交互。在执行DevStack的stack.sh之前，用户也可以填写本地自己配置的文件名为localrc的配置文件，示例如下：

```
DATABASE_PASSWORD=123456          # 根据需要修改此配置文件
RABBIT_PASSWORD=123456
SERVICE_TOKEN=123456
SERVICE_PASSWORD=123456
ADMIN_PASSWORD=admin
HOST_IP=192.168.111.111
```

如果准备了这样的本地配置文件，那么在运行stack.sh脚本时就会减少这些密码和认证信息的确认过程。关于Hypervisor的配置，在DevStack源代码库的默认配置文件stackrc中有如下的两行配置：

```
VIRT_DRIVER=${VIRT_DRIVER:-libvirt}
LIBVIRT_TYPE=${LIBVIRT_TYPE:-kvm}
```

所以，在使用DevStack安装OpenStack时，默认设置libvirt作为Hypervisor的驱动，使用QEMU/KVM作为Hypervisor，只有在不能成功加载"kvm"模块时才会退回到使用QEMU纯软件模拟的虚拟化方式。在执行完stack.sh脚本安装好OpenStack之后，也可以查看"kvm"和"kvm_intel"（或"kvm_amd"）模块是否成功加载，还可以查看/etc/nova/nova.conf文件中有如下两项：

```
libvirt_type=kvm
compute_driver=libvirt.LibvirtDriver
```

一般来说，正常执行完DevStack中的stack.sh脚本后，OpenStack的各项服务（如计算、镜像、控制面板、网络等）都已经正常打开处

于可用状态。当然，如果安装过程中有一些错误出现，则可以根据其源代码进行相应的修改来解决或绕过一些问题。当需要管理这些OpenStack服务时，或者运行stack.sh前需要清理环境时，可以执行DevStack中的unstack.sh脚本来停止OpenStack相关的服务和清理一些中途安装出错的环境。

当然，DevStack这一套脚本一般用于部署的是OpenStack开发环境，对于在实际生产环境中大规模部署OpenStack云计算基础架构来说，还并不够完善。不过，可以根据OpenStack官方的一些部署参考文档（见本章末尾的“参考阅读”）和实际的应用场景，以DevStack为基础来构建一套个性化的OpenStack生产环境。

6.5.3　在OpenStack中使用KVM

在本次示例中，执行完DevStack中的脚本，就顺利地安装了OpenStack的安装，并自动启动了OpenStack的各种必要的服务。而且，在默认情况下，当前的OpenStack通过libvirt API使用的Ubuntu系统中的QEMU/KVM来管理云计算中的虚拟化客户机及其使用的各种资源。

在示例中，在Ubuntu宿主机的Firefox浏览器中输入"http://localhost/"即可连接到本地OpenStack的控制面板（Dashboard）中，对存储卷、镜像、客户机、用户、项目等进行管

理。选择一个名为Demo的演示项目，该项目在控制面板的Web界面中的总体概况如图6-23所示，其中展示了当前系统中总的资源使用情况（客户机个数、CPU资源、存储资源等）。

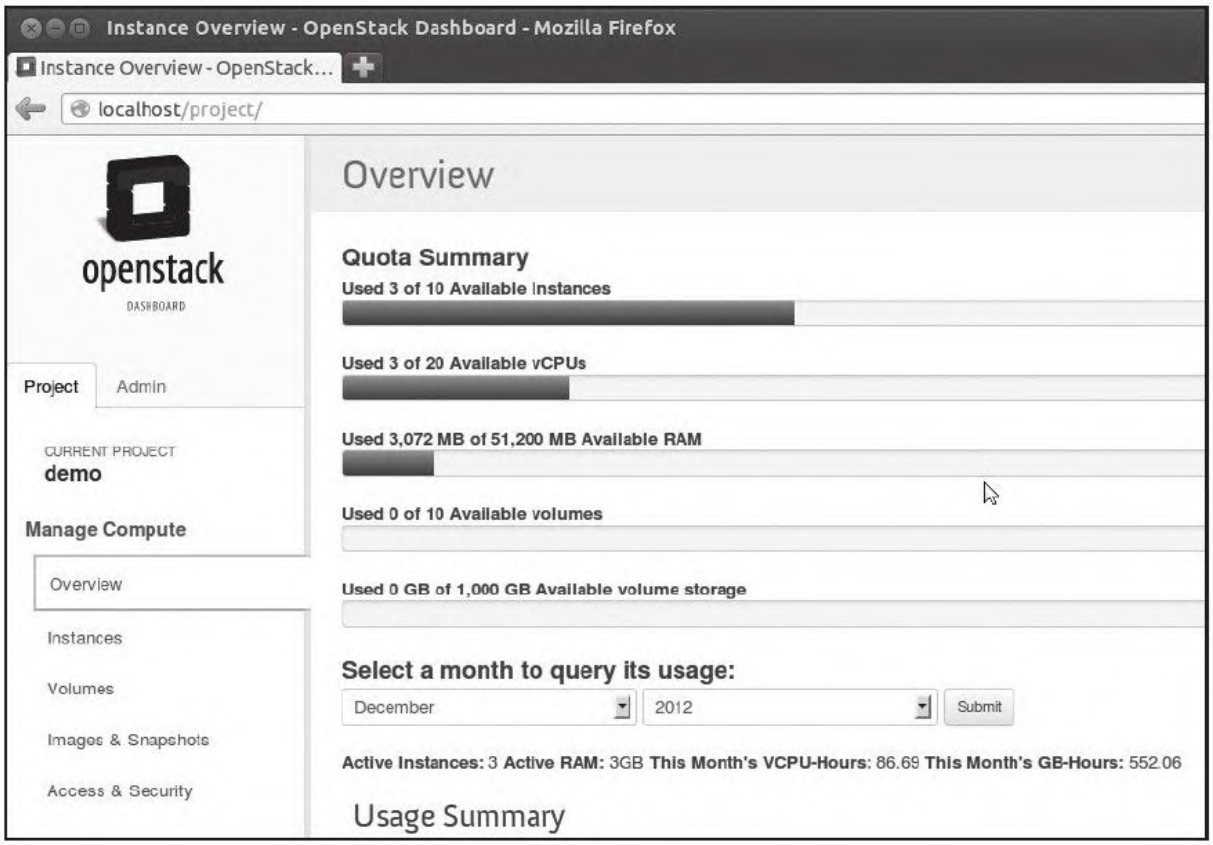


图　6-23　在OpenStack控制面板中查看Demo项目

在OpenStack中管理客户机镜像和快照，如图6-24所示，可以在该管理界面中查看各个镜像和快照的名称、状态、格式等，也可以创建、删除镜像和快照文件，还看到了启动镜像的按钮（"Launch"按钮）。镜像在启动后就成为一个客户机（也在这里称为示例Instance）。在使用DevStack安装OpenStack时，默认会下载一个

cirros镜像上传上去（图6-24中名为"cirros-0.3.0-x86_64-uec"的镜像），这里已经额外上传过3个镜像文件了。

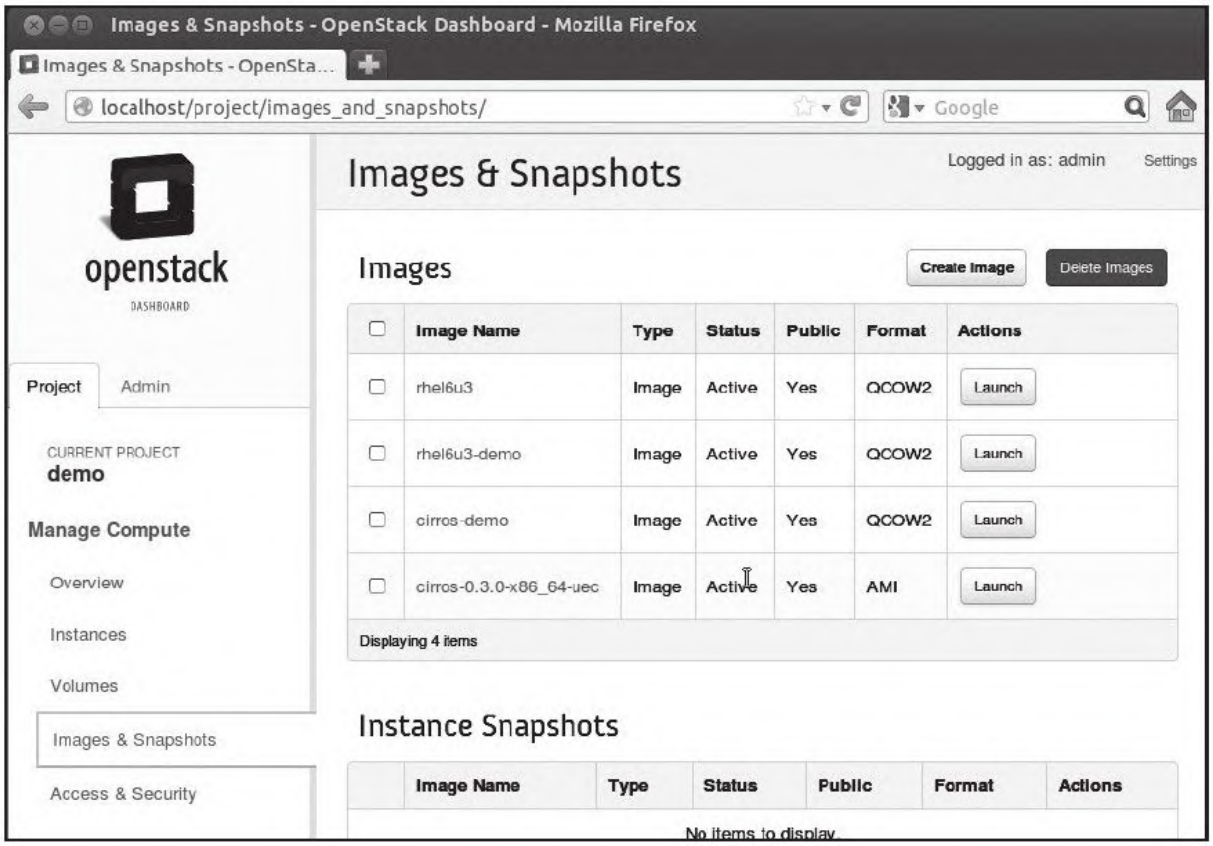


图　6-24　OpenStack中管理镜像和快照

在OpenStack中管理客户机（实例），如图6-25所示，可以查看到当前系统中实例的个数、名称、IP地址、大小、状态等信息，还可以创建、暂停、重启、删除实例，还有为某个实例创建快照的功能。由于这里分配的是内部局域网IP地址，只要在该OpenStack的实例的局域网之内，可以使用Linux上的SSH等工具，根据这里查看到的IP地址来访问到某个实例，然后可以到实例中执行所需的任务。

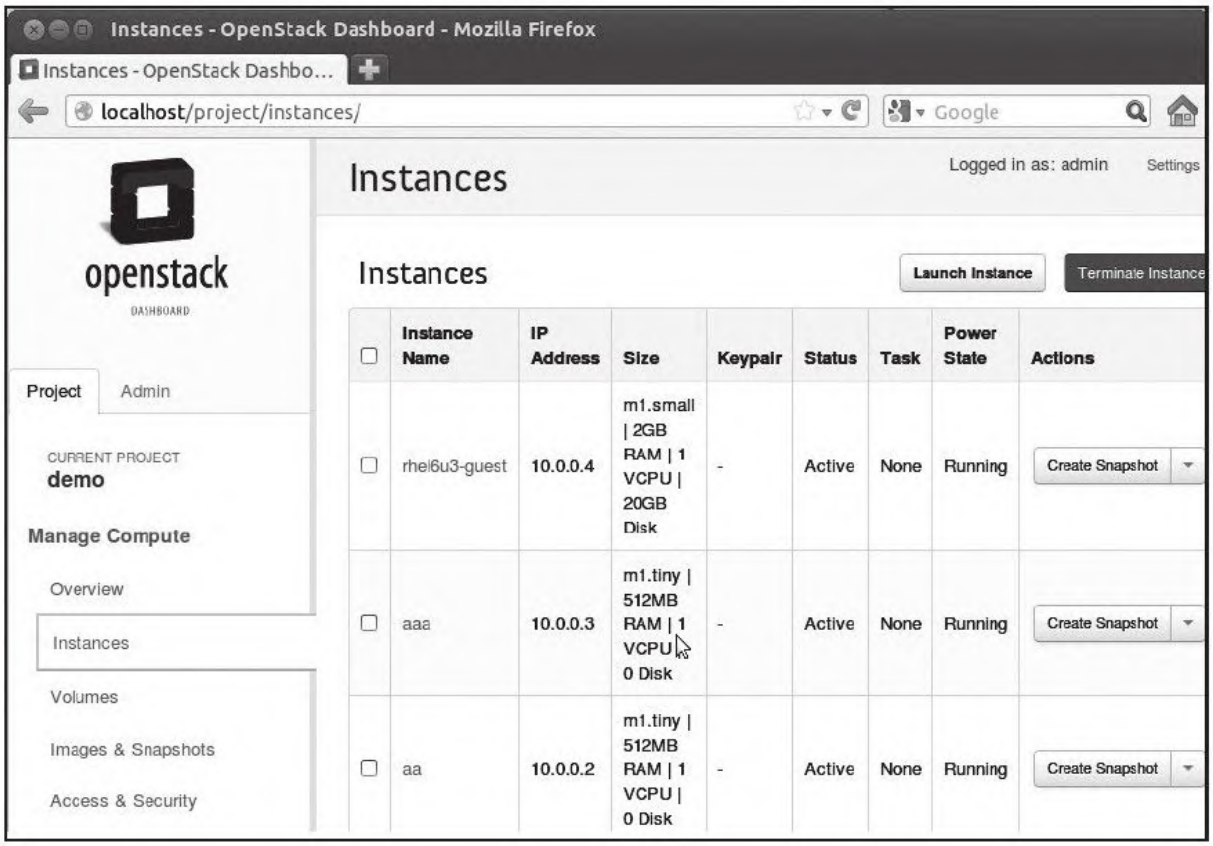


图　6-25　OpenStack中的实例管理

本小节只是粗略地介绍了通过OpenStack的Web管理界面（Dashboard）来管理云计算中的各种资源，让读者有一个基本的认识。除了通过Web的方式，还是可以通过一些命令行（如nova、glance、keystone等命令）来对资源进行查询和控制。关于OpenStack使用中可能遇到的其他问题，读者可以自行参考OpenStack官方网站的文档，或者参与它的邮件讨论列表中的讨论。

6.6 本章小结

经过这几年的发展，KVM虚拟化方案的功能完善、性能良好、架构清晰而简单，不过由于其用户空间的qemu-kvm命令行工具的参数有一定的复杂度，提高了初学者和系统管理员学习和部署KVM的难度。然而，在本章中，介绍了一些比较流行的虚拟化管理工具，让用户可以更加方便地使用KVM，也可使上层应用程序更方便来调用KVM的功能。在这些工具中，有libvirt API为其他虚拟化管理工具提供一套通用的API来管理包括QEMU/KVM在内的多种Hypervisor，有基于libvirt API的virsh这个命令行管理工具，有virt-manager这个图形化的虚拟机管理器，也有virt-install、virt-viewer、virt-top等分别用于安装客户机、查看客户机控制台、查看资源占用率的工具。本章还介绍了包括OpenStack在内的基于libvirt API来实现KVM虚拟化管理的云计算平台。虚拟化和云计算以在当今IT工业中的用途和优势，成为近年来的技术热点之一，其发展非常迅速。目前，已经出现了很多与KVM相关的管理工具和云计算平台，在本章没有提及的工具中，比较知名和流行的也有不少，如面向数据中心虚拟化的管理工具——oVirt，Redhat的基于其企业级Linux版本RHEL的虚拟化管理服务器——RHEV，访问和修改客户机磁盘镜像的工具库——libguestfs，与OpenStack类似的开源云计算平台——CloudStack、Eucalyptus、OpenNebula、Cloud

Foundry等。读者可以根据自己实际的生产环境和学习的需要，选择适当的虚拟化管理工具来使用KVM。

6.7　本章注释和参考阅读

1.本章注释

[1]LGPL（Lesser General Public License）是自由软件基金会（FSF）发布的一种自由软件许可证。GNU LGPL是介于完全公共版权（copyleft）的GNU GPL许可证和完全宽松的BSD、MIT许可证之间的一种折中许可证。在Linux世界中有很多的软件库（library）都采用GNU LGPL许可证发布。可以参考维基百科中对LGPL的介绍：http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License

[2]关于VMware的API、驱动和libvirt客户端连接VMWare的介绍，可以参考如下网页中的内容：http://libvirt.org/drvesx.html

[3]URI（Uniform Resource Identifier，统一资源标识符），是一个字符序列，它用于唯一标识Web上抽象的或物理的资源。读者比较熟悉的在浏览器中输入的URL就属于URI中的一种表现形式。URI包括统一资源名称（URN）和统一资源定位器（URL）。关于URI的语法标准，可以参考RFC3986规范文档：http://www.ietf.org/rfc/rfc3986.txt

[4]Kickstart配置文件是在Redhat公司的RHEL系统中用于无人值守的系统安装和配置的文件，当然其他一些系统（如Fedora、

Ubuntu）也支持使用Kickstart的方式安装系统。在需要方便地安装和配置大量操作系统之时，Kickstart的方式将会非常有用。可以通过手动方式编写Kickstart配置文件，也可以通过"system-config-kickstart"这个GUI工具来生成Kickstart配置文件，还可以使用Redhat的标准安装程序"Anaconda"来生成Kickstart配置文件

[5]本章中关于OpenStack的架构图来自于OpenStack官方网站上的管理员手册：http://docs.openstack.org/trunk/openstack-compute/admin/content/conceptual-architecture.html

2.参考阅读

（1）libvirt官方网站上关于libvirt AP的详细描述：http://libvirt.org/html/libvirt-libvirt.html

（2）libvirt官方网站上关于virsh命令行的参考手册：http://libvirt.org/sources/virshcmdref/html/

（3）QEMU中的命令行参数及其monitor中的命令，在virsh中的对应关系，可以参考：

http://wiki.libvirt.org/page/QEMUSwitchToLibvirt

（4）下面的网址是virt-manager官方网站，从中可以获取更多关于virt-manager的信息和参考文档。

http://virt-manager.org/

（5）虚拟化数据中心的管理工具——oVirt，其官方网站是：http://www.ovirt.org/

（6）OpenStack的官方参考文档：http://docs.openstack.org/

（7）Openstack的安装和配置的入门文档：http://www.openstack.org/software/start/

第7章　Linux发行版中的KVM

前面3章主要介绍的编译KVM内核和qemu-kvm工具，第4章和第5章主要介绍了KVM的一些各种功能并使用qemu-kvm命令行工具进行了必要的演示，第6章主要是介绍使用libvirt、virt-manager等工具来管理KVM，那么，目前的Linux发行版对KVM的支持力度怎么样呢？不用自己从源代码编译QEMU/KVM，如何在Linux发行版中直接使用KVM的功能呢？本章将会向大家介绍Linux发行版中的KVM的状况和基本使用方法。

7.1　RHEL和Fedora中的KVM

7.1.1　Red Hat、RHEL、Fedora和CentOS简介

Red Hat是一家总部位于美国北加州的专注于提供开源软件产品和服务的软件公司，也是美国纽约证券交易所的上市公司。Red Hat是目前对Linux内核贡献最大的公司，同时，也是对KVM源代码贡献量最大并提供软件产品和技术服务的公司。因为Red Hat公司有专门的虚拟化团队负责开发、测试和维护KVM源代码，并将其集成到Red Hat的虚拟化产品之中，所以Red Hat提供的系统可能是目前对KVM虚拟化功能和性能都支持的最好的系统之一。

大约在2003年时，原来的Red Hat Linux发行版被分裂为了两个分支版本：RHEL和Fedora。Red Hat公司发布的企业级Linux操作系统——RHEL（Red Hat Enterprise Linux）是目前市场上使用得最广泛的商业化的Linux发行版。RHEL是商业软件，需要付费向Red Hat购买许可证和服务才能使用，不过，RHEL也是开源的操作系统，可以从Red Hat公司获取RHEL的源代码。RHEL为了稳定，一般没有使用最新的Linux内核版本，如RHEL6.x系列都是统一用2.6.32版的内核作为基础，再加上Red Hat工程师向后移植（backport）了Linux内核很多新功能到RHEL的内核中。RHEL是经过充分测试的比较稳定的企业级Linux发行版，而且在高可用性、可扩展性、可管理性、负载均衡等方面都做了不少优化，是在生产环境中使用Linux系统的一个不错的选择。

与RHEL不同的是，Fedora是完全由Fedora社区而不是某一个公司开发和维护的完全免费和开源的Linux发行版。当然，Red Hat公司对Fedora项目和社区都有比较大力度的支持，Redhat的很多工程师也是Fedora社区中非常活跃的开发者。Fedora系统大约每六个月发布一次正式版本，而且Fedora社区更愿意直接升级内核而不是为老的内核打补丁，所以使用最新版本的Fedora很容易直接使用到比较新的Linux内核，从而获得新的内核带来的新功能和性能提升。当然，有时候最新的并不一定是最好的，因为使用最新的内核也可能因此而遇到最新的bug。另外，一般来说，Linux内核中的很多公共的新特性一般都是先进入最新的Fedora发行版中，然后Red Hat工程师们再将这些新特性添

加到RHEL发行版中。对于想尝鲜的个人用户或想使用免费Linux发行版的企业用户来说，Fedora是一个非常不错的选择。

另外，还有一个与Red Hat关系密切的Linux发行版，那就是CentOS。CentOS系统在目前的Web服务器领域有较高的市场份额。CentOS项目收集了Redhat公司为了遵循各种软件许可证（如GNU GPL）而必须公开的关于RHEL发行版的绝大部分源代码，然后将这些源代码重新编译后发布了自己的CentOS操作系统。一般来说，RHEL发布后的几个月内，CentOS就会发布与之对应的CentOS系统，例如，Red Hat在2012年6月21日发布了RHEL 6.3系统，CenOS社区在2012年7月9日就发布了CentOS 6.3系统。CentOS项目宣称，CentOS系统是与对应版本的RHEL在二进制可执行文件的层面上是100%完全兼容的。由于CentOS是完全免费和开源的Linux发行版，且CentOS还包含RHEL的绝大部分源代码，能获得与RHEL差不多的功能和性能，因此对于不想花钱买Red Hat许可证又想用RHEL系统的部分用户来说，CentOS是一个不错的选择。

7.1.2 RHEL中的KVM

本书前面的示例，一般都是根据在RHEL 6.3系统上自己编译的QEMU/KVM来介绍的，所以关于RHEL中KVM的使用已经提及过了。本节主要以x86_64架构的RHEL 6.3系统为例来介绍一下如何使用它自带的KVM虚拟化功能。

1.安装虚拟化相关的软件包

在安装RHEL 6.3系统的过程中，如图3-3所示，在安装时选择为"Virtualization Host"即可将该系统安装为KVM虚拟化宿主机。如果没有在安装时选择虚拟化宿主机选项，那么也可以自己手动用YUM或RPM工具安装所需要的RPM软件包。

在RHEL 6.3系统中，与KVM虚拟化相关的RPM包括主要包括如下几个（在具体系统中版本号可能与下面提示中的并不完全相同）：

```
kernel-2.6.32-279.el6.x86_66
qemu-kvm-0.12.1.2-2.295.el6.x86_64
qemu-img-0.12.1.2-2.295.el6.x86_64
libvirt-0.9.10-21.el6.x86_64
libvirt-client-0.9.10-21.el6.x86_64
libvirt-python-0.9.10-21.el6.x86_64
virt-manager-0.9.0-14.el6.x86_64
virt-viewer.x86_64 0:0.5.2-9.el6
```

2.支持KVM虚拟化的内核与用户空间的工具

RHEL系统从6.0版本开始默认的内核就支持KVM虚拟化，而且在安装系统时可以选择与虚拟化相关的软件包。查看RHEL 6.3系统的内核配置文件（一般为/boot/config-2.6.32*），会发现它已经将KVM虚拟化配置到内核中了，具体配置可以参考3.3.2节“配置KVM”中的介

绍。用户空间的虚拟化工具qemu-kvm是由qemu-kvm软件包提供的，一般为/usr/libexec/qemu-kvm可执行程序。尽管本书前面的多数章节都使用自己编译的qemu-system-x86_64这个命令行工具来使用KVM，但RHEL 6.3中的qemu-kvm命令行工具的参数与介绍过的qemu-system-x86_64的参数几乎完全一样，当然如果QEMU版本差异较大，可能有少数的几个参数用法不兼容。另外，qemu-img、libvirt、virt-manager、virt-viewer等工具都有对应的RPM软件包。

3.在RHEL中使用KVM

在RHEL 6.3中，可以用qemu-kvm命令行直接使用KVM启动客户机，也可以使用virsh、virt-manager等工具来管理客户机。利用qemu-kvm命令行工具可以很容易启动一个客户机。一个简单的命令行操作如下：

```
/usr/libexec/qemu-kvm -m 1024 rhel6u3.img
```

要利用virsh工具启动一个客户机，需要事先准备好一个libvirt格式的客户机XML配置文件，命令行操作如下：

```
virsh create rhel6u3-guest.xml
```

关于virsh工具的更多功能和命令操作，可以参考6.2.2节“virsh常用命令”的介绍。由于第6章介绍virt-manager工具时，就是以RHEL

6.3系统中的virt-manager为例来介绍的，所以此处不再重复介绍了，参考6.3.3节“virt-manager使用”中的详细介绍即可。

7.1.3 Fedora中的KVM

由于Fedora最初也是从Redhat Linux中衍生出来的，因此Fedora的使用方法与RHEL有非常多的相似之处。本节以目前较新的Fedora 17版本为例，简单介绍在Intel x86_64硬件平台Fedora中如何使用KVM。

Fedora 17中与KVM虚拟化相关的部分软件包如下，使用KVM之前需要安装它们。

```
kernel-3.5.6-1.fc17.x86_64
qemu-system-x86-1.0-17.fc17.x86_64
qemu-common-1.0-17.fc17.x86_64
qemu-kvm-1.0-17.fc17.x86_64
qemu-img-1.0-17.fc17.x86_64
libvirt-daemon-0.9.11.3-1.fc17.x86_64
libvirt-0.9.11.3-1.fc17.x86_64
libvirt-python-0.9.11.3-1.fc17.x86_64
libvirt-client-0.9.11.3-1.fc17.x86_64
virt-manager-common-0.9.1-3.fc17.noarch
virt-manager-0.9.1-3.fc17.noarch
virt-viewer-0.5.2-1.fc17.x86_64
```

安装相应的软件包后，可以使用qemu-kvm或qemu-system-x86_64命令行工具来是使用KVM，也可以用virsh、virt-manager等工具通过libvirt API来使用KVM虚拟化功能。

图7-1展示了在Fedora 17的virt-manager中运行着一个KVM客户机。

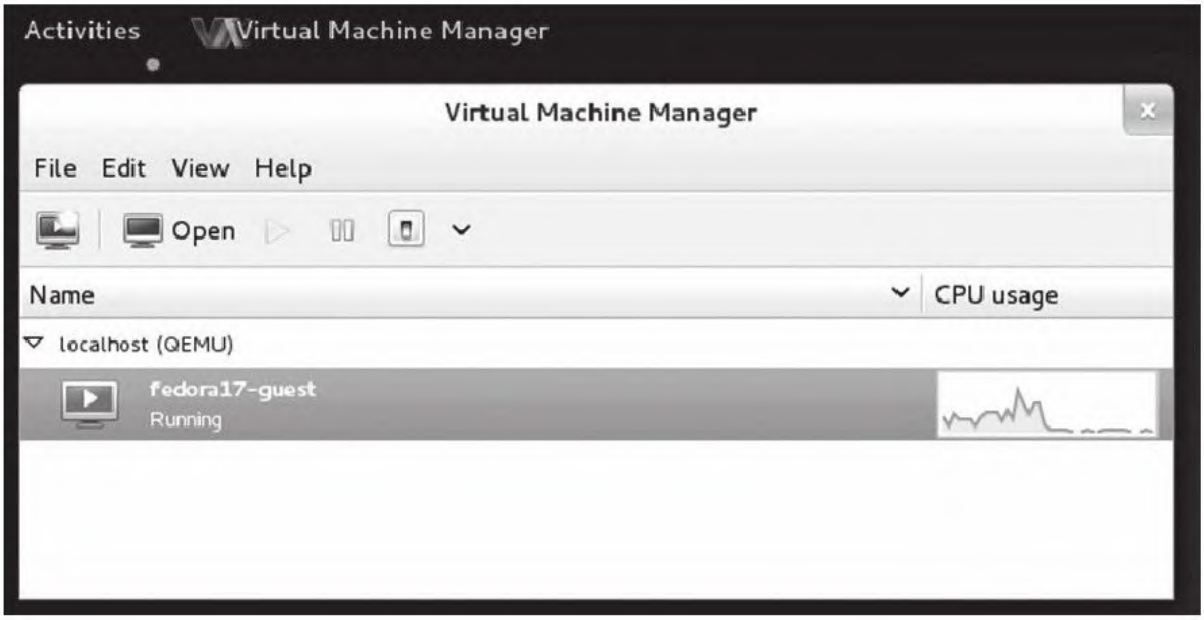


图　7-1　Fedora 17系统的virt-manager工具中运行着KVM客户机

7.2 SLES和openSUSE中的KVM

7.2.1 SLES中的KVM

SLES（SUSE Linux Enterprise Server）是由Attachmate集团[1]开发的一个企业级Linux发行版，主要用于服务器、工作站等领域，不过也可以在桌面PC上使用SLES。SLES的大版本一般是3～4年发布一次，小版本（也称为“服务包”，service pack）大约18个月一个发布周期。目前，SLES的最新版本是在2012年2月发布的SLES11 SP2。本节正是以SLES11 SP2为例来简单介绍SLES中KVM的使用方法的。

SLES11 SP2从官方发布版本中支持Xen和KVM两种虚拟化技术。在SLES11 SP2中，可以使用YaST工具选择"Virtualization"中的"Hypervisor and Tools"，选中"KVM"，然后确定即可安装KVM虚拟化相关的软件包。SLES11 SP2中的内核已经将KVM的支持编译进去了，KVM用户态工具相关的软件包如下：

```
kvm-0.15.1-0.17.3
libvirt-python-0.9.6-0.13.42
libvirt-0.9.6-0.13.42
libvirt-client-0.9.6-0.13.42
virt-manager-0.9.0-3.15.15
virt-viewer-0.4.1-1.13.34
virt-utils-1.1.7-0.9.9
```

其中kvm软件包主要包含qemu-kvm命令行工具及KVM虚拟化必须的文件，如下：

```
/usr/bin/qemu-img-kvm      # 用于创建 KVM 客户机镜像
/usr/bin/qemu-kvm          #qemu-kvm 命令行工具
/usr/share/qemu-kvm/       # 含有客户机 BIOS 文件、网卡 ROM、显卡 ROM 文件等
```

在SLES11 SP2中也是使用libvirt、virt-manager、virt-viewer等工具来管理KVM虚拟化的，这些工具在第6章“KVM的管理工具”中已经详细介绍，在SLES中的用法与之非常类似的，本节不再详细介绍其细节。安装好KVM虚拟化工具后，在SLES的YaST管理中心中查看虚拟化相关的工具和配置，如图7-2所示。

图　7-2　SLES11　SP2的YaST管理中心查看虚拟化相关的配置

图7-3展示了在SLES11 SP2中运行这一个KVM客户机的virt-manager主界面。

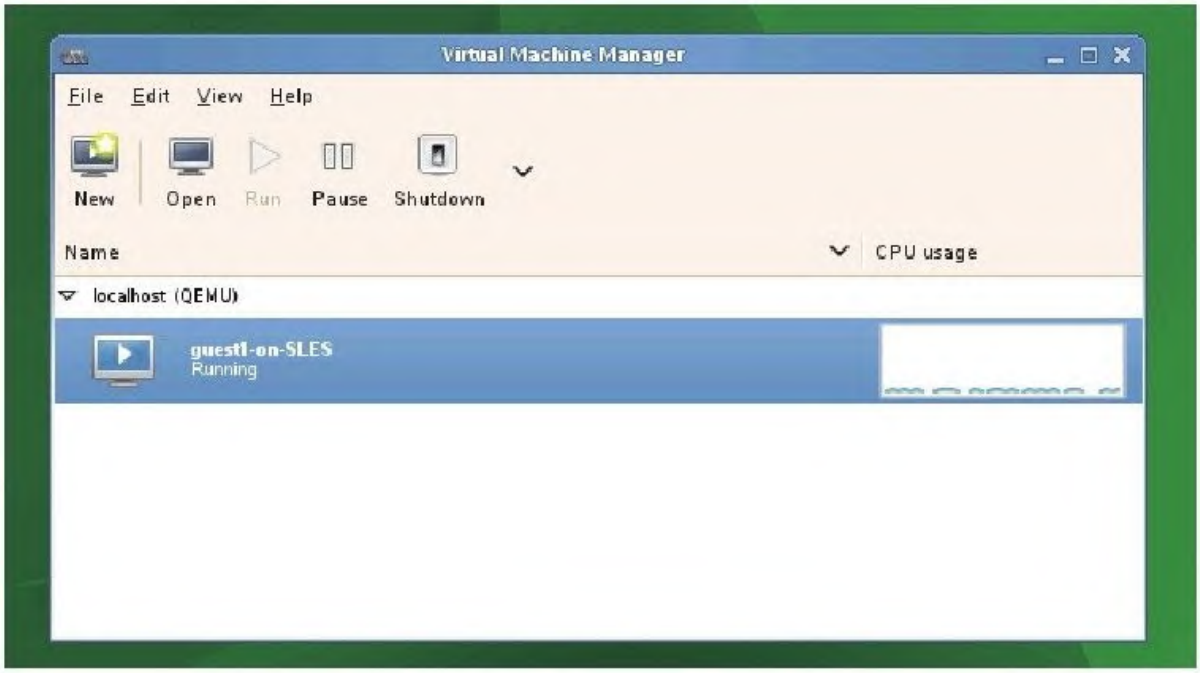


图 7-3 SLES11 SP2中的virt-manager运行了一个KVM客户机

当然，除了一些封装好的工具之外，也可以直接使用qemu-kvm命令行工具来启动KVM客户机，启动一个1024MB内存的客户机，让其显示在2号VNC窗口中，命令行如下：

```
qemu-kvm -m 1024 kvm-guest.img -vnc :2
```

7.2.2 openSUSE中的KVM

openSUSE和SLES的关系，与Fedora和RHEL的关系类似，SLES是需要购买许可证才能使用的商业化的企业级Linux系统，而openSUSE是由社区开发和维护的完全免费和开源的项目。目前，openSUSE系统最新

的版本是12.2，本节就是以openSUSE 12.2为例来介绍openSUSE中KVM的使用方法。

在openSUSE 12.2桌面版的系统中，与KVM虚拟化相关的部分软件包如下：

```
kernel-desktop-3.4.6-2.10.1.x86_64
kvm-1.1.1-1.1.1.x86_64
qemu-tools-1.1.0-3.6.1.x86_64
libvirt-0.9.11.4-1.2.1.x86_64
libvirt-python-0.9.11.4-1.2.1.x86_64
libvirt-client-0.9.11.4-1.2.1.x86_64
vm-install-0.6.3-3.1.1.x86_64
virt-manager-0.9.3-1.3.1.x86_64
virt-viewer-0.5.3-2.1.2.x86_64
```

openSUSE中的内核已经默认将KVM支持编译进去了，所以不需重新编译内核来使用KVM了。与SLES类似，在openSUSE中也是使用qemu-kvm、libvirt、virt-manager、virt-viewer等工具来使用和管理KVM客户机。

在openSUSE 12.2系统中，包含一个运行中的KVM客户机的virt-manager管理主界面，如图7-4所示。

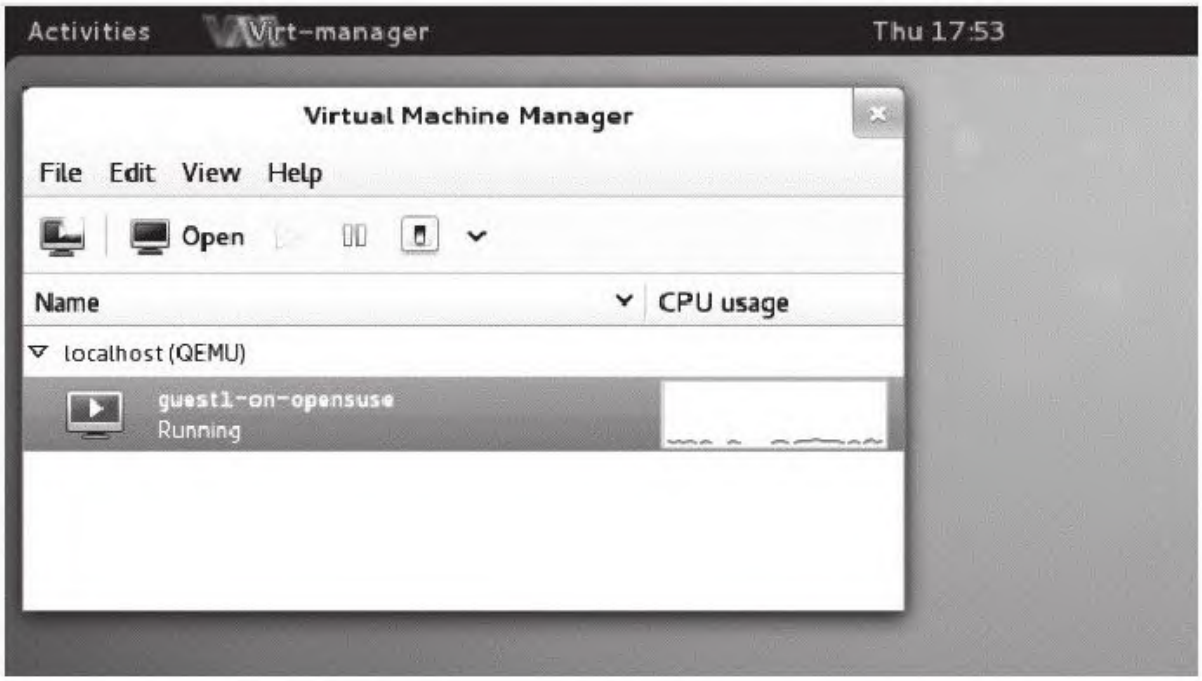


图　7-4　openSUSE 12.2中的virt-manager

7.3 Ubuntu中的KVM

Ubuntu是一个基于Debian[2] 发行版的免费和开源的Linux操作系统。Ubuntu系统的开发是由Ubuntu基金会组织一些大公司的工程师及许多个人开发者共同完成的，其背后领导Ubuntu开发的公司是一家总部位于英国伦敦的名叫Canonical[3] 的软件公司。由于Ubuntu的自由和开放性及功能的易用性，Ubuntu是目前在台式机和笔记本上最流行的桌面级Linux发行版。除了桌面应用，Ubuntu也发布了其Server版本用于服务器领域，而且在云计算领域也有Ubuntu的一些应用（如6.5节“OpenStack云计算平台”就是在Ubuntu环境中演示的）。

Ubuntu基金会计划每6个月发布一次新版本，目前一般是在每年的4月和10月分别发布两个版本，对普通版本提供18个月的技术支持。Ubuntu采用发布版本时的年份和月份来作为发布的版本号，如最新的12.10版本是在2012年10月18日发布的。另外，Ubuntu一般会每隔两年发布一次长期支持版（long-term support，LTS），对于目前的LTS版本提供支持的时间为5年。目前最新的LTS版本是2012年4月26日发布的Ubuntu 12.04 LTS版本，本节就是选用Ubuntu 12.04 LTS版本来介绍KVM在Ubuntu中的使用方法的。

Ubuntu 12.04首选KVM作为其虚拟化技术方案，同时也使用libvirt作为KVM虚拟化管理的API，使用virsh、virt-manager等工具

来调用libvirt以管理KVM虚拟化。在Ubuntu中，可以使用apt-get命令来安装KVM相关的软件包，命令行操作如下：

```
sudo apt-get install qemu-kvmlibvirt-bin bridge-utils
sudo apt-get install ubuntu-vm-builder
sudo apt-get install virt-managervirtinst
```

其中，qemu-kvm提供了KVM必需的用户空间管理工具，libvirt-bin就是提供libvirt API和virsh管理工具，bridge-untils提供了管理网桥的brctl工具，ubuntu-vm-builder提供了一个Ubuntu优化过的构建客户机的强大的命令行工具，virt-manager提供了图形界面下管理KVM的工具（virt-manager），virtinst提供了命令行下安装客户机的工具（virt-install）。

在Ubuntu 12.04中，qemu-kvm的版本是1.0版本，由于kvm命令是一个到qemu-system-x86_64工具的软连接，因此可以直接使用kvm命令来启动客户机。在使用KVM前，可以使用kvm-ok命令（它是一个Shell脚本）来检查当前系统是否支持KVM。命令行操作如下：

```
root@jay-ubuntu:~# kvm --version
QEMU emulator version 1.0 (qemu-kvm-1.0), Copyright (c) 2003-2008 FabriceBellard
root@jay-ubuntu:~# ls -l /usr/bin/kvm
lrwxrwxrwx 1 root root 18 Apr 13  2012 /usr/bin/kvm -> qemu-system-x86_64
root@jay-ubuntu:~# kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

在Ubuntu 12.04的virt-manager中启动了一个Ubuntu客户机，如图7-5所示。

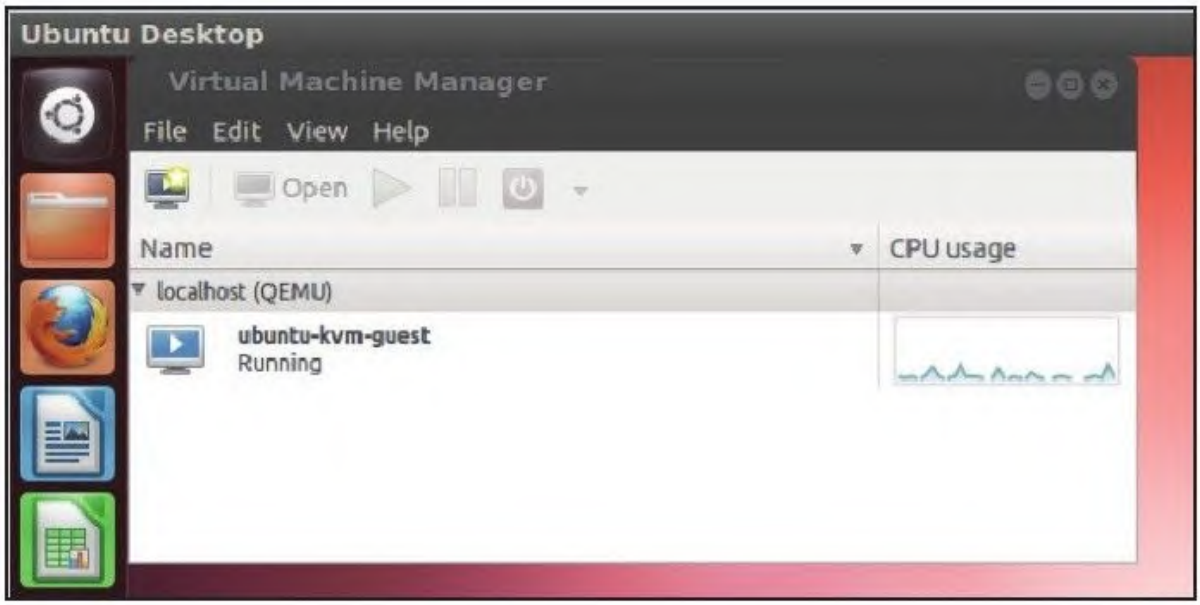


图 7-5 运行着一个KVM客户机的virt-manager

双击运行着的客户机标识可以进入到客户机的图形化控制台中，如图7-6所示。

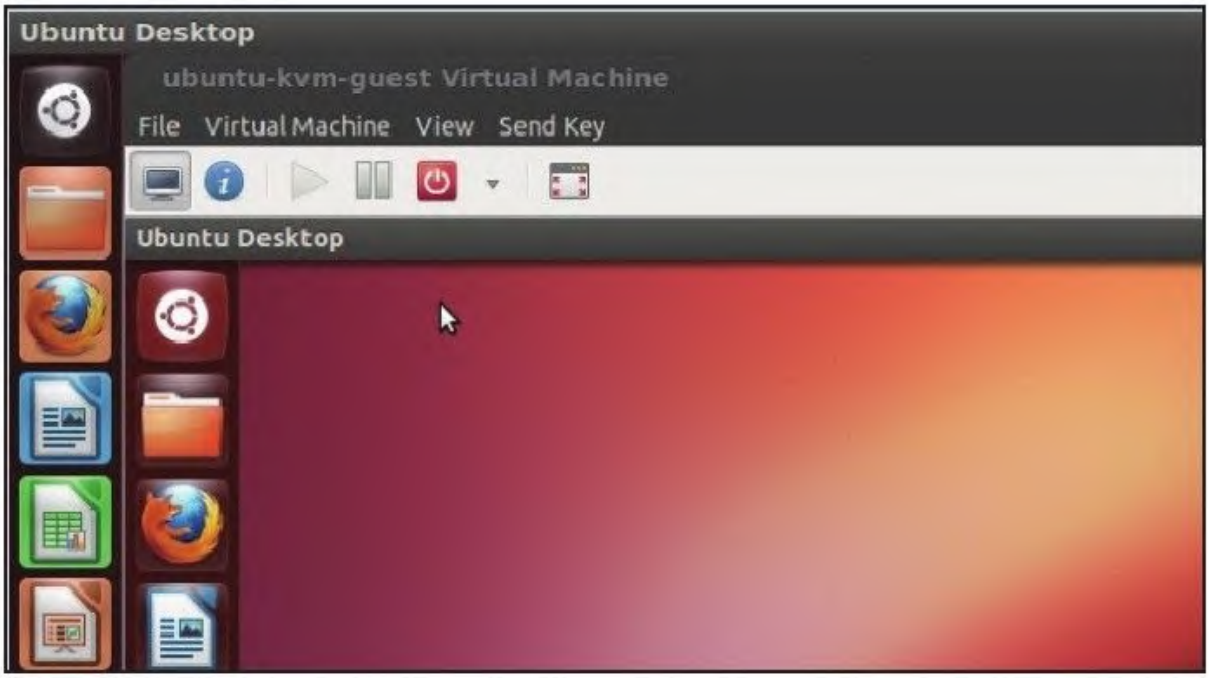


图 7-6 Ubuntu12.04的virt-manager中运行着一个Ubuntu客户机

7.4 本章小结

本章简单介绍和演示了RHEL、Fedora、SLES、openSUSE、Ubuntu等几个比较流行的Linux发行版中KVM虚拟化的使用。由于Linux发行版数量众多，而且在其他的Linux发行版（如Arch Linux、Gentoo、CentOS、Oracle Enterprise Linux、Mint、Mandriva等）中，KVM的使用方法也跟本章介绍的这些类似，因此没有对它们进行一一介绍。简单来说，KVM的使用，只需要三个方面的支持：一是CPU支持硬件虚拟化（Intel VT或AMD-V）；二是内核编译时配置KVM的支持；三是安装必要的用户空间的qemu-kvm工具。在各个Linux发行版中的qemu-kvm工具的名称和命令参数，除了版本差异带来的少量不同之处，它们的使用方法都与第3章、第4章和第5章中自己进行编译时使用的qemu-system-x86_64命令行工具是相同的。在Linux发行版中，使用libvirt、virsh和virt-manager等工具来管理KVM，这与第6章“KVM的管理工具”中介绍的使用方法是完全类似的，因此本章没有再重复详细介绍这些工具的使用。

7.5　本章注释和参考阅读

1.本章注释

[1]Attachmate集团是一家总部位于美国的软件公司，它于2011年收购了Novell公司，而Novell公司早在在2003年收购了SUSE公司。SUSE是一家创立于德国的软件公司，它开发了SUSE Linux系统。所以，目前，是Attachemate集团拥有SLES系统和SUSE商标。

[2]Debian（Debian GNU/Linux）发行版包含了Linux内核和一套GNU管理操作系统的工具，它是基于GNU GPL许可证发布的免费和开源的Linux操作系统。Debian也是比较流行和具有影响力的Linux发行版。为开源社区所称道的是，Debian非常严格地坚持了UNIX和自由软件的哲学。由于Debian的稳定性和安全性做得非常好，因此有不少其他发行版（如Ubuntu）以Debian作为自己发行版的基础。

[3]Canonical公司是由南非企业家Mark Shuttleworth创立的，总部位于英国伦敦的软件公司。Canonical主要依靠为Ubuntu等软件提供商业化的技术支持和服务来实现盈利。

2.参考阅读

（1）RHEL官方网站：www.redhat.com/products/enterprise-linux/

（2）Fedora项目主页：http://fedoraproject.org/index.html

（3）Fedora官方的虚拟化手册：http://docs.fedoraproject.org/en-US/Fedora/13/html/Virtualization_Guide/

（4）SLES官方网站：https://www.suse.com/products/server/

（5）openSUSE项目主页：http://www.opensuse.org/

（6）SLES中KVM虚拟化使用手册：http://doc.opensuse.org/products/draft/SLES/SLES-kvm_sd_draft/book.kvm.html

（7）Ubuntu官方主页：http://www.ubuntu.com/

（8）Ubuntu社区中的KVM帮助文档：https://help.ubuntu.com/community/KVM

第8章　KVM性能测试及参考数据

系统虚拟化有很多的好处，如提高物理资源利用率、让系统资源更方便监控和管理、提高系统运维的效率、节约硬件投入的成本，等等。那么，在真正实施生产环境的虚拟化时，到底选择哪种虚拟化方案呢？选择商业软件VMware ESXi、开源的KVM和Xen，还是微软的Hyper-V，或者有其他的虚拟化方案？在进行虚拟化方案的选择时，需要重点考虑的因素中至少有两个至关重要：虚拟化方案的功能和性能，这二者缺一不可。功能是实现虚拟化的基础，而性能是虚拟化效率的关键指标。即便是功能非常丰富的虚拟化技术，如果它的性能非常不好，我们也很难想象将其应用到生产环境中的效果到底是“利大于弊”还是“弊大于利”。

在前面的章节中，已经介绍了QEMU/KVM虚拟化中的很多概念、功能及基本原理，本章将介绍如何测试KVM虚拟化的性能及分享一些测试数据供读者参考。

8.1　虚拟化性能测试简介

虚拟化性能测试包括的范围比较广泛，可能包含CPU、内存、网络、磁盘的性能，也可能包含虚拟客户机动态迁移时的性能，也可能

需要考虑多种物理平台上的性能，也可能需要考虑很多个虚拟客户机运行在同一个宿主机上时的性能。目前，有一些针对各个虚拟化软件的性能分析工具（profiling），也有一些衡量虚拟化系统中单个方面性能的基准测试工具（benchmark），不过还没有一个能集成所有这些性能测试于一体的比较权威的专门针对虚拟化的性能测试工具。由于虚拟化性能测试涉及计算机系统的方方面面，而且没有一个标准化的测试工具，因此，虚拟化性能测试与性能分析也是一个比较具有挑战性的工程研究领域。

虚拟化性能测试，初看起来是比较复杂和难以操作的，不过，只要细心研究并从用户的角度出发，会发现虚拟化性能测试也并不是多么的高深。对于绝大多数普通用户来说，他们所接触到的无非是一些应用软件（如微软的Office办公套件、杀毒软件等）和互联网中的网页（如Google、百度等），所以，不管是否使用虚拟化，终端用户最关心的还是实际使用的应用软件和互联网站点的性能。应用软件、网络站点的性能才是直接地真正关系到用户体验。从这个角度来看，在虚拟化环境中，只要能保证普通应用程序的性能良好，自然就能为用户带来良好的性能体验。

评价一个系统的性能标准，一般可以用响应时间（response time）、吞吐量（throughput）、并发用户数（concurrent users）

和资源占用率（utilization）等几个指标来衡量。下面简单介绍一下这几个指标的含义。

1）响应时间：指的是客户端从发出请求到得到响应的整个过程所花费的时间。响应时间是用户能感受到的最直接、最关键的性能指标，试想，如果一个网页，尽管其中内容质量比较良好，但是每次在浏览器中输入URL后需要10分钟才能得到响应打开网页，这样的网页你愿意再次访问吗？

2）吞吐量：指的是在一次性能测试过程中网络上传输的数据量的总和。在一定的时间长度内，系统能达到的吞吐量当然是越大越好，因为吞吐量越大越可能为用户传输更多的数据。

3）并发用户数：指的是同时使用一个系统服务的用户数量。对于一个系统来说，能支持的并发用户数当然是越多越好。2011年6月推出的12306铁路购票网站[1]，在2012年春节之前不久，由于很多人同时登录网站购票，而且没买到票的人会不停刷新网页，并发用户数量达到很大的数量级别，从而导致12306网站几乎瘫痪。

4）资源利用率：指的是在使用某项服务时，客户端和服务器端物理资源占用情况，包括CPU、内存等的利用率。在达到同样的响应时间、吞吐量和并发用户数的指标时，系统的资源利用率当然是越小越

好。例如，我们在使用一个应用程序时，都不希望它将我们宝贵的CPU和内存全都占用，因为同一个系统还需要并行运行其他的程序。

系统中应用程序的数量和类型都非常多，如Office等办公软件、数据库服务器软件、文件存储系统、Web服务、缓存服务、邮件服务、科学计算服务、各种单机的或网络版的游戏等。应用程序不但数量众多，而且它们对使用系统的使用特点也不同，有CPU密集型的（如科学计算），有网络I/O密集型的（如Web服务），也有磁盘I/O密集型的（如数据库服务），也有内存密集型的（如缓存服务）。功能相似的应用程序，使用不同编程语言来开发，其性能差别可能很大，而且，选择不同的中间件（middle ware）来部署同一套应用程序，其性能也很可能大不相同。所以，衡量KVM虚拟化的性能最直接的方法就是：将准备实施虚拟化的系统中运行的应用程序迁移到KVM虚拟客户机中试运行，如果性能良好且稳定，则可以考虑真正实施该系统的虚拟化。

尽管系统中运行的应用程序可能是数量繁多，种类也千差万别，但是它们几乎都会使用CPU、内存、网络、磁盘等基本的子系统。

在本章中，主要对KVM虚拟化中的几个最重要的子系统进行性能对比测试，具体方法是：在非虚拟化的原生系统（native）中执行某个基准测试程序，然后将该测试程序放到与原生系统配置相近的虚拟客户机中执行，接着对比在虚拟化和非虚拟化环境中该测试程序执行的性能。由于QEMU/KVM的性能测试与硬件配置、测试环境参数、宿主机

和客户机系统的种类和版本等都有千丝万缕的联系，而且性能测试本身也很可能有一定的误差存在，本章展示的部分测试结果可能在不同的测试环境中并不能重现，本章中的所有测试数据和结论都仅供读者参考。在实施KVM虚拟化之前，请以实际应用环境中的测试数据为准。

8.2 CPU性能测试

CPU是计算机系统中最核心的部件，CPU的性能直接决定了系统的计算能力，故对KVM虚拟化进行性能测试首先选择对客户机中CPU的性能进行测试。任何程序的执行都会消耗CPU资源，所以任何程序几乎都可以作为衡量CPU性能的基准测试工具，不过最好是选择CPU密集型的测试程序。有很多的测试程序可用于CPU性能的基准测试，包括SPEC组织的SPEC CPU和SPECjbb系列、UnixBench、SysBench、PCMark、PC内核编译、Super PI等，下面对其中的几种工具进行简单的介绍。

8.2.1 CPU性能测试工具

（1）SPECCPU2006

SPEC（Standard Performance Evaluation Corporation）是一个非营利组织，专注于创建、维护和支持一系列标准化的基准测试程序（benchmark），让这些基准测试程序可以应用于高性能计算机的性能测试。许多公司，如IBM、Microsoft、Intel、HP、Oracle、Cisco、EMC、华为、联想、中国电信等，都是SPEC组织的成员。SPEC系列的基准测试工具，针对不同的测试重点，有不同的测试工具，如测试CPU的SPEC CPU、测试Java应用的SPECjbb、测试电源管理的SPECpower、测试Web应用的SPECweb、测试数据中心虚拟化服务器整合的SPECvirt_sc

等。相对来说，SPEC组织的各种基准测试工具在业界的口碑都比较良好，也具有一定的权威性。

SPEC CPU2006是SPEC CPU系列的最新版本，之前的版本有CPU2000、CPU95等，其官方主页是http://www.spec.org/cpu2006/。SPEC CPU2006既支持在Linux系统上运行又支持在Windows系统上运行，是一个非常强大的CPU密集型的基准测试集合，里面包含分别针对整型计算和浮点型计算的数十个基准测试程序[2] 。在SPEC CPU2006的测试中，有bzip2数据压缩测试（401.bzip2）、人工智能领域的象棋程序（458.sjeng）、基于隐马尔可夫模型的蛋白质序列分析（456.hmmer）、实现H.264/AVC标准的视频压缩（464.h264ref）、2D地图的路径查找（473.astar）、量子化学中的计算（465.tonto）、天气预报建模（481.wrf）、来自卡内基梅隆大学的一个语音识别程序（482.sphinx3），等等。当然，其中一些基准测试也是内存密集型的，如429.mcf的基准测试既是CPU密集型又是内存密集型的。在测试完成后，可以生成HTML、PDF等格式的测试报告。测试报告中有分别对整型计算和浮点型计算的总体分数，并且有各个具体的基准测试程序的分数。分别在非虚拟化原生系统和KVM虚拟化客户机系统中运行SPEC CPU2006，然后对比它们的得分即可大致衡量虚拟化中CPU的性能。

（2）SPECjbb2005

SPECjbb2005是SPEC组织用于评估服务器端Java应用性能的基准测试程序，其官方主页为http://www.spec.org/jbb2005/。该基准测试主要测试Java虚拟机（JVM）、JIT编译器、垃圾回收、Java线程等各个方面，也可以对CPU、缓存、内存结构的性能进行度量。SPECjbb2005既是CPU密集型也是内存密集型的基准测试程序，它利用Java应用能够比较真实地反映Java程序在某个系统上的运行性能。

（3）UnixBench

UnixBench（即曾经的BYTE基准测试）为类UNIX系统提供了基础的衡量指标，其官方主页为http://code.google.com/p/byte-unixbench/。它并不是专门测试CPU的基准测试，而是测试系统的许多方面，测试结果不仅会受系统CPU、内存、磁盘等硬件的影响，也会受操作系统、程序库、编译器等软件系统的影响。UnixBench中包含许多测试用例，如文件复制、管道的吞吐量、上下文切换、进程创建、系统调用、基本的2D和3D图形测试等。

（4）SysBench

SysBench是一个模块化的、跨平台的、支持多线程的基准测试工具，主要评估的是系统在模拟的高压力的数据库应用中的性能，其官方主页为http://sysbench.sourceforge.net/。其实，SysBench并非

一个完全CPU密集型的基准测试，它主要用来衡量CPU调度器、内存分配和访问、文件系统I/O操作、线程创建等多方面的性能。

（5）PCMark

PCMark由Futuremark公司开发，是针对一个计算机系统整体及其部件进行性能评估的基准测试工具，其官方网站是http://www.futuremark.com/benchmarks/pcmark。在PCMark的测试结果中，会对系统整体和各个测试组件进行评分，得分的高低直接反映其性能的好坏。目前，PCMark只能在Windows系统中运行。PCMark分为几个不同等级的版本，其中基础版可以免费下载和使用，而高级版和专业版都需要支付一定的费用才能合法使用。

（6）内核编译

内核编译（kernel build或kernel compile）就是以固定的配置文件对Linux内核代码进行编译，它是Linux开发者社区（特别是内核开发者社区）中最常用的系统性能测试方法，可以算作是一个典型的基准测试。内核编译是CPU密集型，也是内存密集型，而且是磁盘I/O密集型的基准测试，在使用make命令进行编译时，可以添加"-j N"参数来使用N进程协作编译，所以它也可以评估系统在多处理器（SMP）系统中多任务并行执行的可扩展性。只要使用相同的内核代码，使用相同的内核配置，使用相同的命令进行编译，然后对比编译时间的长

短即可评价系统之间的性能差异。另外，关于Linux内核编译步骤可以参考3.3.3节“编译KVM”中的详细介绍。

（7）Super PI

Super PI是一个典型的CPU密集型基准测试工具，最初是1995年日本数学家金田康正[3] 用于计算圆周率π的程序，当时他将圆周率计算到了小数点后4G（2的32次方）个数据位。Super PI基准测试程序的原理非常简单，它根据用户的设置计算圆周率π小数点后N个位数，然后统计消耗的时间，根据时间长度的比较就能初步衡量CPU计算能力的优劣。Super PI最初是一个Windows上的应用程序，可以从http://www.superpi.net/网站下载，目前支持计算小数点后32M（2的25次方）个数据位。不过，目前也有Linux版本的Super PI，可以从http://superpi.ilbello.com/网站下载，它也支持计算到小数点后32M个数据位。目前的Super PI都支持单线程程序，可以执行多个实例从而实现多个计算程序同时执行，另外，也有一些测试程序实现了多线程的Super PI，如Hyper PI（http://virgilioborges.com.br/hyperpi/）。

在实际生产环境中，运行实际的CPU密集型程序（如可以执行MapReduce的Hadoop）当然是测试CPU性能较好的方法，不过，为了体现更普通而不是特殊的应用场景，本节选择了3个基准测试程序用于测

试KVM虚拟化中的CPU性能，包括比较权威的SPEC CPU2006、Linux社区中常用的内核编译和非常简单的Super PI。

8.2.2 测试环境配置

本次对CPU性能测试的硬件环境为一台使用Intel Xeon E5-4650处理器的服务器，在BIOS中打开Intel VT和VT-d技术的支持，默认开启Intel CPU的超线程（Hyper-threading）技术。

本次测试使用的宿主机内核是根据手动下载的Linux 3.7.0版本的内核源代码然后自己编译的，qemu-kvm使用的是1.2.0版本。用于对比测试的原生系统和客户机系统使用完全相同的操作系统，都是使用默认配置的RHEL 6.3 Linux系统。更直观的测试环境基本描述如表8-1所示。

表 8-1 CPU 性能测试的软硬件环境

测试环境	描　述
硬件系统	一个使用 Intel Xeon E5-4650 处理器的服务器，VT-x、VT-d、超线程等是打开的
宿主机内核	Linux kernel 3.7.0（x86_64 架构，EPT/VPID 打开，THP 也处于使用中）
qemu-kvm	qemu-kvm 1.2.0
非虚拟化原生系统	RHEL 6.3（kernel 2.6.32-279.el6.x86_64）
客户机系统	RHEL 6.3（kernel 2.6.32-279.el6.x86_64）

在KVM宿主机中，EPT、VPID等虚拟化特性是默认处于打开状态的（可参考4.3.2节），透明大页（THP）的特性也默认处于打开状态，这几个特性对本次测试的结果的影响是比较大的。对于本章的所有性

能测试，若没有特别注明，就是使用表8-1所示的软硬件测试环境。注意：本次CPU性能测试并没有完全使用Xeon E5-4650处理器中的CPU资源，而是对服务器上的CPU和内存资源都进行了限制，KVM宿主机限制使用了4个CPU线程和12 GB内存，原生系统使用4个CPU线程和10GB内存。为了防止图形桌面对结果的影响，原生系统、KVM宿主机、KVM客户机系统的运行级别都是3（带有网络的多用户模式，不启动图形界面）。在本次测试中，对各个Linux系统的内核选项添加的额外配置如表8-2所示，它们既可以设置在GRUB配置文件中，又可以在系统启动到GRUB界面时进行编辑。

表 8-2　CPU 性能测试的内核选项配置情况

内核选项	配　　置
host kernel option	maxcpus=4 nr_cpus=4 mem=12G 3
native kernel option	maxcpus=4 mem=10G 3
guest kernel option	3 (i.e. run level is 3 in guest.)

在本次测试中，为客户机分配了4个vCPU和9.5GB内存，与原生系统基本保持一致以便进行性能对比（将客户机内存设置为9.5GB而不是10GB是为了给宿主机多留一点内存，分配给客户机9GB或是10GB内存，对本次CPU性能测试结果的影响不大）。由于SPEC CPU2006的部分基准测试消耗较多的内存，例如在429.mcf执行时每个执行进程就需要大约2GB内存，所以这里设置的内存数量是比较大的。将客户机的磁盘驱动设置为使用virtio-blk驱动，启动客户机的qemu-kvm命令行如下：

```
qemu-system-x86_64 -smp 4 -m 9728 -drive rhel6u3.img,if=virtio
-net nic -net tap -vnc :0 --daemonize
```

从上面命令可以看出，本次测试并没有指定虚拟CPU的模型，默认使用了qemu64这个CPU模型，如果应用程序中需要更多CPU特性，可以用-cpu SandyBridge或-cpu host参数设置CPU模型（可参考4.2.4节）。

8.2.3 性能测试方法

本节的CPU性能测试选取SPEC CPU2006、内核编译和Super PI这三个基准测试来对比KVM客户机与原生系统的性能。下面分别介绍在本次性能测试中使用的具体测试方法。

（1）SPEC CPU2006

在获得SPEC CPU2006的测试源代码并进入其主目录后，运行install.sh脚本即可安装SPEC CPU2006，然后通过source命令执行shrc脚本来配置运行环境，最后执行bin/runspec这个Perl脚本即可正式开始运行基准测试。SPEC CPU2006还提供了在Windows系统中可以执行的对应的.BAT脚本文件。在Linux系统中，将这些基本执行步骤整合到一个Shell脚本中，代码如下：

```
#!/bin/bash
cd /root/cpu2006/
./install.sh
echo "starting SPECCPU2006 at $(date)"
source shrc
bin/runspec --action=validate -o all -r 4 -c \
Example-linux64-amd64-gcc43.cfg all
echo "SPECCPU2006 ends at $(date)"
```

在本示例中，runspec脚本用到的参数有：--action=validate表示执行validate这个测试行为（包括编译、执行、结果检查、生成报告等步骤），-o all表示输出测试报告的文件格式为尽可能多的格式（包括html、pdf、text、csv、raw等），-r 4（等价于--rate--copies 4）表示本次将会使用4个并发进程执行rate类型的测试（这样可以最大限度地消耗分配的4个CPU线程资源），--config xx.cfg表示使用xx.cfg配置文件来运行本次测试，最后的all表示执行整型（int）和浮点型（fp）两种测试类型。runspec的参数比较多也比较复杂，可以参考其官方网站的文档[4] 了解各个参数的细节。

执行完上面整合的测试脚本后，在SPEC CPU2006主目录下的result目录中，就会出现关于本次运行测试的各种测试报告，本次示例使用的报告是HTML格式的CINT2006.001.ref.html（对整型的测试报告）和CFP2006.001.ref.html（对浮点型的测试报告）。这两个报告文件中，报告的第一部分中有总体的测试分数，报告中部的结果表格中记录了各个具体的基准测试的得分情况。分别在非虚拟化的原生系

统和KVM客户机系统执行SPEC CPU2006，然后对比它们的测试报告中的分数即可得到对KVM虚拟化环境中CPU虚拟化性能的评估。

（2）内核编译

本次内核编译的基准测试中采用的方法是：对Linux 3.7.0正式发布版本的内核进行编译，并用time命令对编译过程进行计时。关于内核编译测试中的内核配置，可以随意进行选择，只是需要注意：不同的内核配置，它们的编译时间长度可能会相差较大。内核编译过程可以参考3.3.3节的详细介绍。执行make命令进行编译，用time命令计时，命令行操作如下：

```
[root@kvm-guest linux.git]# time make -j 4
<!-- 省略编译过程的输出信息；下面的时间只是演示需要，并非编译用的真实时间 -->
real    1m0.259s
user    0m18.103s
sys     0m3.825s
```

在time输出信息中，第一行real的时间标识表示实际感受到的从程序开始执行到程序终止所经过的时间长度，第二行user的时间表示CPU在用户空间执行的时间长度，第三行sys表示CPU在内核空间执行的时间长度。在本次内核编译测试中，统计的时间是time命令输出信息的第一行（用real标识）中的时间长度。

（3）Super PI

从http://superpi.ilbello.com/网页下载Linux版本的Super PI，然后运行super_pi可执行程序，本次Super PI的基准测试中选择执行了计算圆周率π的小数点后1048576（2的20次方）个数据位。在计算完成后，程序会输出本次计算花费的时间，命令行操作如下：

```
[root@kvm-guest super-pi]# ./super_pi 20
<!--省略其余输出信息 -->
Start of PI calculation up to 1048576 decimal digits
<!--省略其余输出信息 -->
Total calculation(I/O) time=       10.236(       0.284) Sec.
```

在x86-64架构的系统上运行Super PI执行程序，可能会提示找不到ld-linux.so.2共享库，这是由于Super PI程序比较老，是用32位的glibc链接而生成的，所以只需要在该64位系统中安装32位的glibc库即可，命令行操作如下：

```
[root@kvm-guest super-pi]# ./super_pi 20
./super_pi: ./pi: /lib/ld-linux.so.2: bad ELF interpreter: No
such file or directory
[root@kvm-guest super-pi]# yum install glibc-2.12-1.80.el6.i686
```

8.2.4 性能测试数据

注意，由于使用的硬件平台、操作系统、内核、qemu-kvm等对本次CPU性能测试都有较大影响，而且本次仅仅使用了Intel Xeon E5-4650处理器上的4个CPU线程，所以本次CPU性能测试数据并不代表该处

理器的实际处理能力，测试数据中绝对值的参考意义不大，读者主要参考其中的相对值（即KVM客户机中的测试结果占原生系统中测试结果的百分比）。

（1）SPEC CPU2006

在非虚拟化的原生系统和KVM客户机中，分别执行了SPEC CPU2006的rate base类型的整型和浮点型测试，总体结果如图8-1所示，测试的分数越高表明性能越好。由图8-1中的数据可知，通过SPEC CPU2006基准测试的度量，KVM虚拟化客户机中CPU执行整型计算的性能达到原生系统的97.04%，浮点型计算的性能达到原生系统的96.89%。

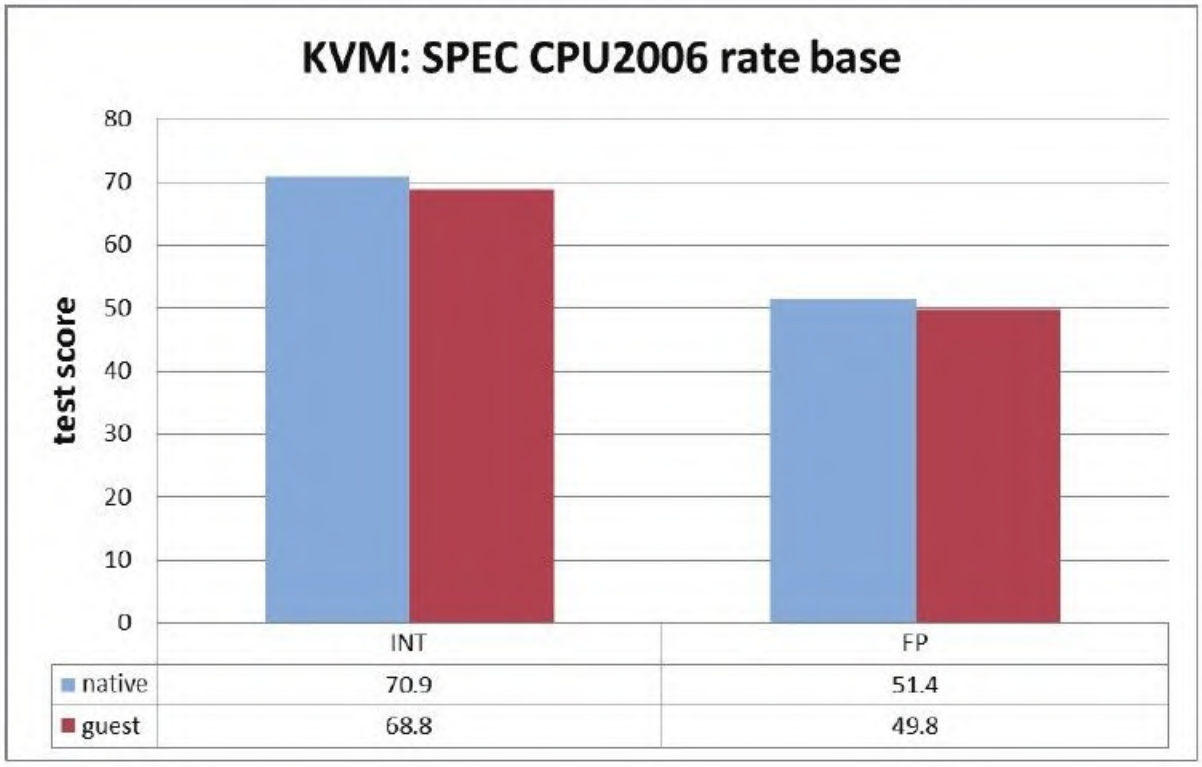


图　8-1　SPEC CPU2006整型和浮点型测试的总体结果对比[5]

在SPEC CPU2006的整型计算测试中，各个基准测试的性能得分对比如图8-2所示，各个基准测试的结果都比较稳定，波动较小。

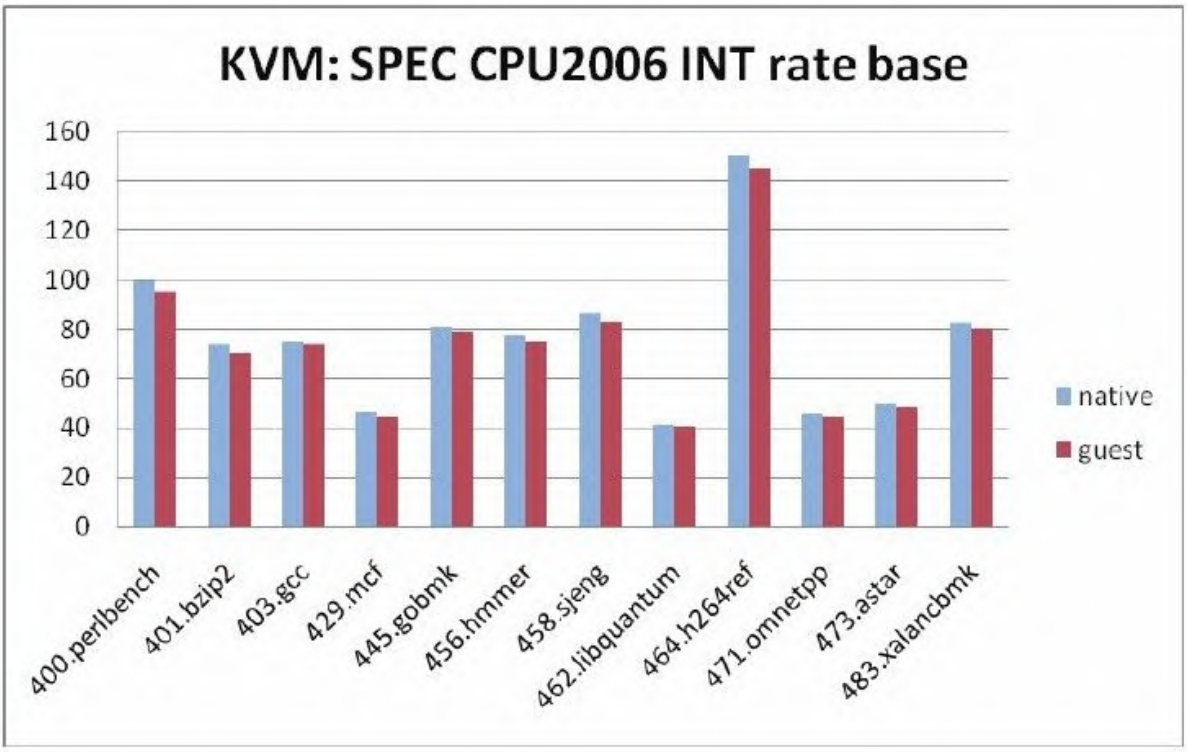


图　8-2　SPEC CPU2006整型测试数据的对比

在SPEC CPU2006的浮点型计算测试中，各个基准测试的性能得分对比如图8-3所示，其中有两项测试（410.bwaves和470.lbm）的对比结果为100%，也有一项测试（416.gamess）的对比性能稍微差一点（为78.40%）。

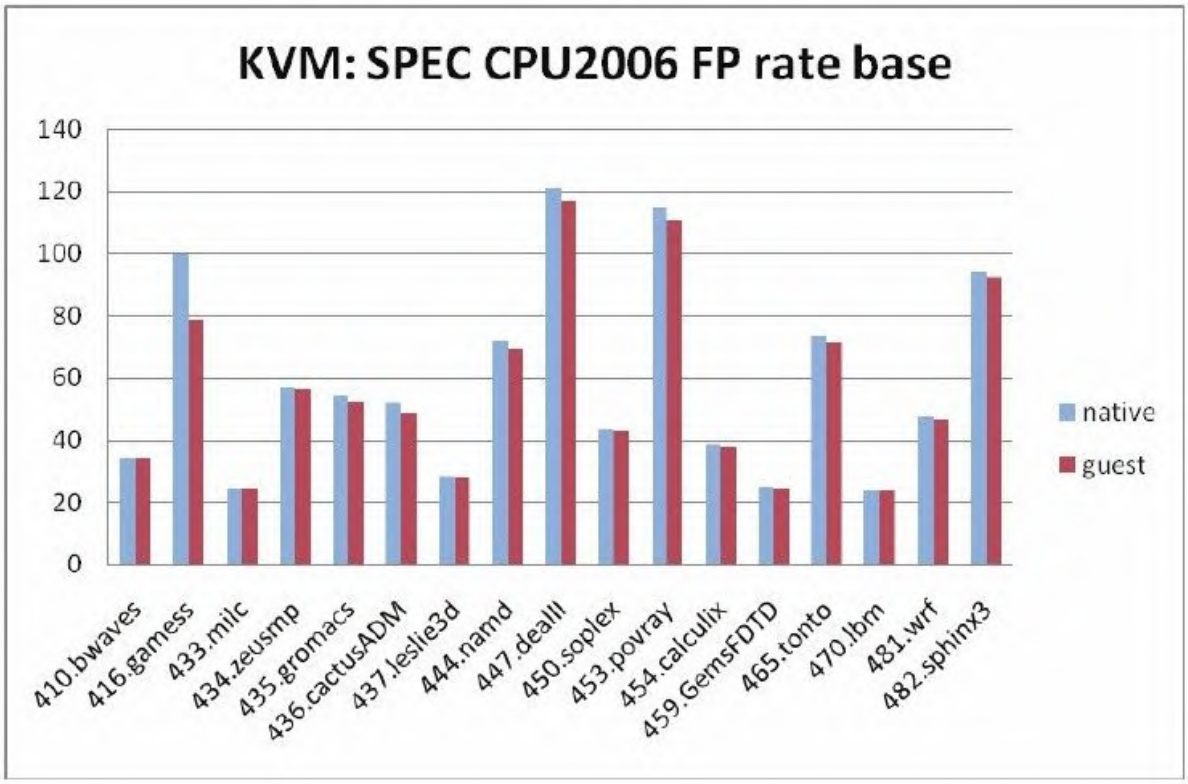


图　8-3　SPEC CPU2006浮点型测试数据的对比

（2）内核编译

分别在原生系统和KVM客户机中编译Linux内核，记录所花费的时间。这里为了测试结果的准确性，图8-4展示的内核编译时间都是测试3次后计算的平均值。

如图8-4所示，时间越短说明CPU性能越好，在本次示例中，总体来说，KVM客户机中编译内核的性能为同等配置原生系统的94.15%左右。

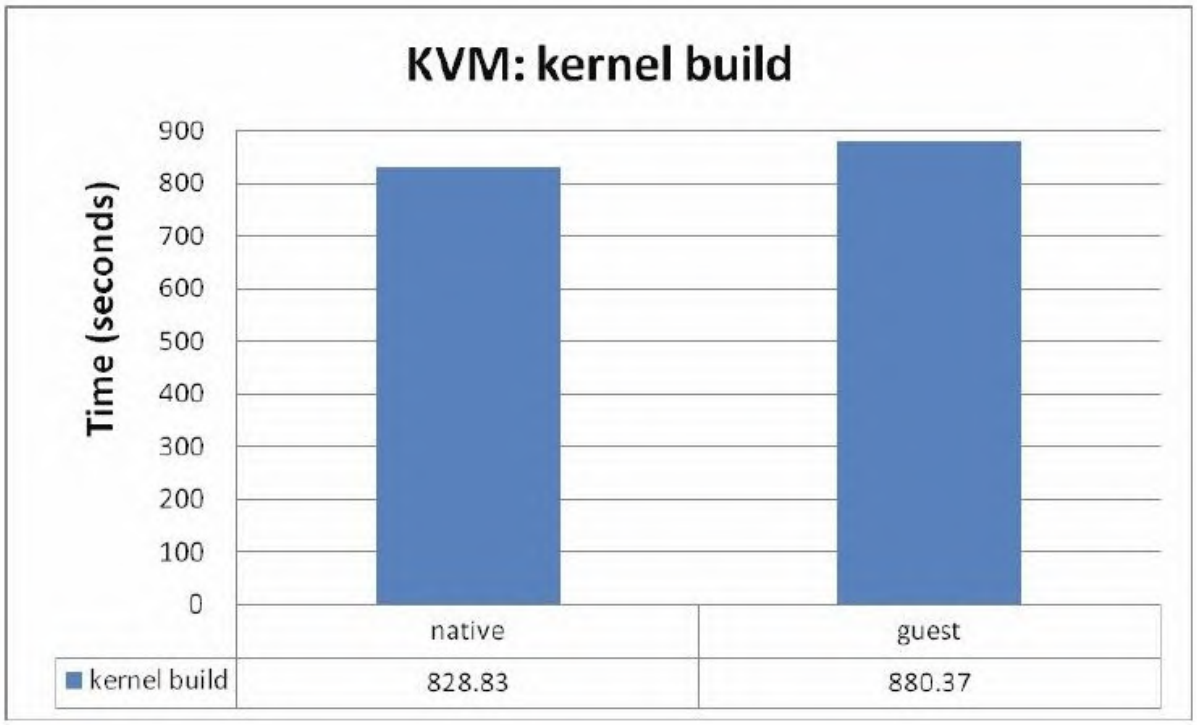


图　8-4　内核编译所需时间的对比

（3）Super PI

分别在原生系统和KVM客户机中执行Super PI基准测试程序，分别重复执行3次取平均值，一次计算所花费的平均时间对比如图8-5所示，其中时间越短说明性能越好。从图8-5看来，本次Super PI测试示例中，客户机CPU性能约为原生系统的97.17%。

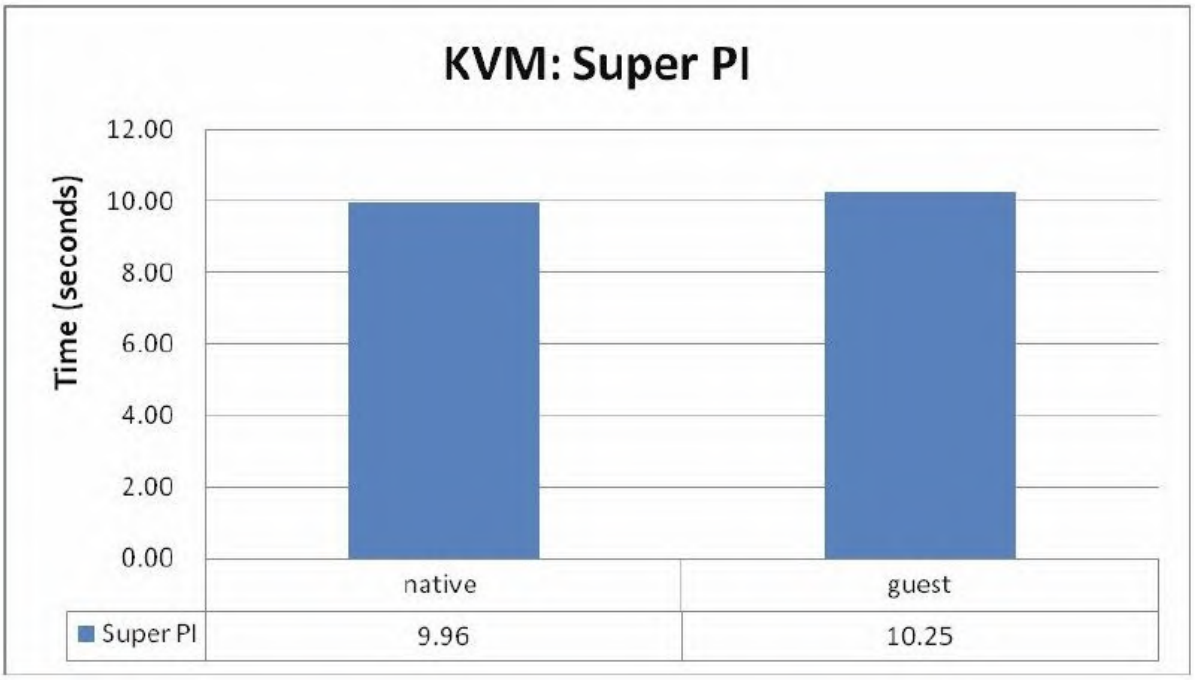


图　8-5　Super PI基准测试数据对比

从SPEC CPU2006、内核编译、Super PI这3个基准测试的数据看来，其中有两个测试（CPU2006和Super PI）显示KVM虚拟化中CPU性能为原生系统的97%左右，另一个测试（内核编译）显示KVM中CPU性能为原生系统的94%。由于内核编译也同时是磁盘I/O密集型的计算，所以可能是由于磁盘I/O的性能差异而导致效果与另外两个测试的结果稍微有一点差距。

8.3 内存性能测试

与CPU的重要性类似，内存也是一个计算机系统中最基本、最重要的组件，因为任何应用程序的执行都需要用到内存。将内存密集型的应用程序分别在非虚拟化的原生系统和KVM客户机中运行，然后根据它们的运行效率就可以粗略评估KVM的内存虚拟化性能。对于内存的性能测试，可以选择8.2.1节中的SPECjbb2005、SysBench、内核编译等基准测试（因为它们同时也是内存密集型的测试），还可以选择LMbench、Memtest86+、STREAM等测试工具。下面简单介绍几种内存性能测试工具。

8.3.1 内存性能测试工具

（1）LMbench

LMbench是一个使用GNU GPL许可证发布的免费和开源的自由软件，可以运行在类UNIX系统中以便比较它们的性能，其官方网址是http://www.bitmover.com/lmbench/。LMbench是一个用于评价系统综合性能的可移植性良好的基准测试工具套件，主要关注两个方面：带宽（bandwidth）和延迟（latency）。LMbench中包含很多简单的基准测试，覆盖了文档读写、内存操作、管道、系统调用、上下文切换、进程创建和销毁、网络等多方面的性能测试。

另外，LMbench能够对同级别的系统进行比较测试，反映不同系统的优劣势，通过选择不同的库函数就能够比较库函数的性能；更重要的是，作为一个开源软件，LMbench提供一个测试框架，假如测试者对测试项目有更高的测试需要，能够通过修改少量的源代码达到目的（比如现在只能评测进程创建、终止的性能和进程转换的开销，通过修改部分代码即可实现线程级别的性能测试）。

（2）Memtest86+

Memtest86+是基于Chris Brady所写的著名的Memtest86改写的一款内存检测工具，其官方网址为http://www.memtest.org/。该软件的目标是提供一个可靠的软件工具，进行内存故障检测。Memtest86+同Memtest86一样是基于GNU GPL许可证进行开发和发布的，它也是免费和开源的。

Memtest86+对内存的测试不依赖于操作系统，它提供了一个可启动文件镜像（如ISO格式的镜像文件），将其烧录到软盘、光盘或U盘中，启动系统时就从软驱、光驱或U盘中的Memtest86+启动，之后就可以对系统的内存进行测试。在运行Memtest86+时，操作系统都还没有启动，所以此时的内存基本上是未使用状态（除了BIOS等可能占用小部分内存）。一些高端计算机主板甚至将Mestest86+默认集成到BIOS中。

（3）STREAM

STREAM是一个基准测试程序，用于衡量系统在运行一些简单矢量计算内核时能达到的最大内存带宽和相应的计算速度，其官方网址为http://www.cs.virginia.edu/stream/。STREAM可以运行在DOS、Windows、Linux等系统上。另外，STREAM的作者还开发了对STREAM进行扩充和功能增强的工具STREAM2，可以参考其主页http://www.cs.virginia.edu/stream/stream2/。

本节对KVM内存虚拟化性能的测试，使用LMbench基准测试工具。

8.3.2 测试环境配置

对KVM的内存虚拟化性能测试的测试环境配置，与8.2.2节的环境配置基本相同，下面仅说明不同之处和需要强调的地方。

本次对内存的性能测试中，使用的CPU资源为两个CPU线程，非虚拟化原生系统使用2GB内存，KVM宿主机拥有4GB内存，这是通过GRUB配置文件中的内核选项来控制的，如表8-3所示。

表 8-3 内存性能测试的内核选项配置情况

内核选项	配　置
host kernel option	maxcpus=2 nr_cpus=2 mem=4G 3
native kernel option	maxcpus=2 mem=2G 3
guest kernel option	3 (i.e. run level is 3 in guest.)

KVM客户机使用两个vCPU和2GB内存，在宿主机中启动客户机的命令如下：

```
qemu-system-x86_64 -smp 2 -m 2G -drive rhel6u3.img,if=virtio -net
nic -net tap -vnc :0 --daemonize
```

值得注意的是，本次测试的Intel平台是支持EPT和VPID的，对于内存测试，只要硬件支持，KVM也默认使用EPT、VPID。EPT和VPID（特别是EPT）对KVM中内存虚拟化的加速效果是非常明显的，所以要保证能使用它们（可参考4.3.2节）。

8.3.3 性能测试方法

本节选取LMbench工具进行内存的性能测试，分别在原生系统和KVM客户机系统中执行LMbench基准测试，然后对比各项测试的得分即可评估KVM内存虚拟化的性能。

（1）下载LMbench

从其官方网址http://www.bitmover.com/lmbench/get_lmbench.html下载LMbench测试程序的第3版。

（2）编译LMbench

对下载的lmbench3.tar.gz文件进行解压，然后运行make命令即可进行编译。在编译过程中，可能遇到如下编译错误提示。

```
[root@kvm-guest lmbench3]# make
<!-- 此处省略输出信息 -->
gmake[2]: *** No rule to make target '../SCCS/s.ChangeSet', needed by 'bk.ver'.  Stop.
gmake[2]: Leaving directory '/root/memory/lmbench3/src'
make[1]: *** [lmbench] Error 2
make[1]: Leaving directory '/root/memory/lmbench3/src'
make: *** [build] Error 2
```

该编译错误并不影响实际的测试功能，有两个方法用来绕过这个编译错误。一个方法是建立SCCS目录，然后在SCCS目录中建立s.ChangeSet文件，命令行操作如下：

```
[root@kvm-guest lmbench3]# mkdir SCCS
[root@kvm-guest lmbench3]# touch SCCS/s.ChangeSet
```

另一个方法是修改src/Makefile，将其中的下面一行的bk.ver字符删除，从而不需要编译bk.ver这个目标。

```
$O/lmbench : ../scripts/lmbench
```

（3）执行LMbench基准测试

运行make results命令即可执行LMbench中的默认测试，命令行操作如下：

```
[root@kvm-guest lmbench3]# make results
```

运行make results后，在正式运行测试之前，会有一些交互式的操作以便确认测试时使用的具体配置，多数的提示只需要按Enter键选择默认值即可。在本次测试中，没有使用默认值的配置有3个，分别是LMbench测试的内存值、处理器时钟频率，以及是否将测试结果发到LMbench3的官方邮箱，如下：

```
MB [default 1393] 1024          # 这个值如果过大，需要很长的测试时间

Processor mhz [default 3289 MHz, 0.3040 nanosec clock] 2700
# 这里默认的时钟频率不准确，所以根据 /proc/cpuinfo 中的如下值来填写
# "model name      : Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz"

Mail results [default yes] no     # 不将结果发 Email 给 LMbench 开发者
```

（4）查看测试结果

LMbench根据配置文档执行完所需要的测试项之后，在results目录下根据系统类型、系统名和操作系统类型等生成一个子目录，测试结果文档按照“主机名+序号”的命令方式存放于该目录下。运行make see命令可以查看测试结果报告及其说明。

```
[root@kvm-guest lmbench3]# make see
```

在本次测试中，分别在原生系统和KVM客户机中执行了LMbench基准测试，将测试结果文档统一放在lmbench3/results/x86_64-linux-gnu/目录中，然后运行make see命令即可查看到非常直观的两次测试结果进行对比的报告。

8.3.4 性能测试数据

分别在非虚拟化原生系统和KVM客户机的RHEL 6.3系统中执行LMbench基准测试，然后将测试结果文档放在同一目录下，运行make see命令即可查看到比较清晰的性能对比。make see命令的输出信息代表的测试结果的示例如下：

```
cd results && make summary percent 2>/dev/null | more
make[1]: Entering directory '/root/lmbench3/results'

                L M B E N C H  3 . 0   S U M M A R Y
```

```
                 -------------------------------------
                  (Alpha software, do not distribute)

Basic system parameters
----------------------------------------------------------------------------
Host                 OS Description              Mhz  tlb  cache  mem   scal
                                                 pages line   par   load
                                                 bytes
--------- ------------- ----------------------- ---- ----- ----- ------ ----
native    Linux 2.6.32-        x86_64-linux-gnu 2700    32   128 4.1100    1
kvm-guest Linux 2.6.32-        x86_64-linux-gnu 2700    32   128 4.2100    1

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                         call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
native    Linux 2.6.32- 2700 0.07 0.11 0.91 1.37 2.16 0.15 0.96 228. 671. 2745
kvm-guest Linux 2.6.32- 2700 0.07 0.11 0.90 1.34 2.18 0.15 0.89 324. 924. 2962

Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
Host                 OS  intgr intgr  intgr  intgr  intgr
                         bit   add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
native    Linux 2.6.32- 0.3100 0.0100 0.1200 6.8700 7.5200
kvm-guest Linux 2.6.32- 0.3100 0.0100 0.1200 6.9300 7.5900

Basic float operations - times in nanoseconds - smaller is better
-----------------------------------------------------------------
Host                 OS  float  float  float  float
                         add    mul    div    bogo
--------- ------------- ------ ------ ------ ------
native    Linux 2.6.32- 0.9100 1.5200 4.5600 4.3600
kvm-guest Linux 2.6.32- 0.9200 1.5300 4.5900 4.4000

Basic double operations - times in nanoseconds - smaller is better
------------------------------------------------------------------
Host                 OS  double double double double
                         add    mul    div    bogo
--------- ------------- ------  ------ ------ ------
native    Linux 2.6.32- 0.9100 1.5200 6.9900 6.6900
kvm-guest Linux 2.6.32- 0.9200 1.5300 7.0400 6.7400

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
```

```
Host                 OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
native    Linux 2.6.32- 3.6600 6.6800   10.5 7.3100   12.2 7.59000    12.5
kvm-guest Linux 2.6.32- 1.0100 0.9000 1.1400 1.4400   22.9 7.32000    11.8

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS  2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                         ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
native    Linux 2.6.32- 3.660  11.8 12.0  26.1  32.0  33.6  41.0  24.
kvm-guest Linux 2.6.32- 1.010 2.719 3.85 6.853 9.943 116.8  12.8  26.

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS  0K File      10K File     Mmap    Prot   Page   100fd
                         Create Delete Create Delete Latency Fault  Fault  selct
--------- ------------- ------ ------ ------ ------ ------- ----- ------- -----
native    Linux 2.6.32- 6.2500 4.7421   13.1 6.9634  4974.0 0.211 0.73250 0.905
kvm-guest Linux 2.6.32- 6.4565 4.9180   13.0 7.2052  5382.0 0.219 0.80160 0.901

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------------
Host                 OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                         UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
native    Linux 2.6.32- 2295 9769 798. 7711.7  11.0K 3634.1 3669.9 9895 6040.
kvm-guest Linux 2.6.32- 4994 5787 3872 7593.9  10.8K 3553.8 3595.5 9746 5811.

Memory latencies in nanoseconds - smaller is better
    (WARNING - may not be correct, check graphs)
-------------------------------------------------------------------------------
Host                 OS   Mhz   L1 $   L2 $    Main mem    Rand mem    Guesses
--------- -------------   ---   ----   ----    --------    --------    -------
native    Linux 2.6.32-  2700 1.2170 3.6630   24.6        83.4
kvm-guest Linux 2.6.32-  2700 1.2250 3.9200   25.6        87.2
make[1]: Leaving directory '/root/lmbench3/results'
```

从上面的测试结果可以看出，KVM虚拟化中内存的带宽和延迟，与原生系统相比都是比较接近的。所以，可以粗略地得出结论：在硬件提供的内存虚拟化技术（如Intel的EPT）支持下，QEMU/KVM的内存虚拟化性能比较良好，可以达到原生系统95%以上的性能。

8.4 网络性能测试

如果KVM客户机中运行网络服务器对外提供服务，那么客户机的网络性能也是非常关键的。只要是需要快速而且大量的网络数据传输的应用都可以作为网络性能基准测试工具，可以是专门用于测试网络带宽的Netperf、Iperf、NETIO、Ttcp等，也可以是常用的Linux上文件传输工具SCP。下面简单介绍几种常用的网络性能测试工具。

8.4.1 网络性能测试工具

（1）Netperf

Netperf是由HP公司开发的网络性能基准测试工具，是非常流行网络性能测试工具，其官方主页是http://www.netperf.org/netperf/。

Netperf工具可以运行在UNIX、Linux和Windows操作系统中。Netperf的源代码是开放的，不过它和普通开源软件使用的许可证协议不完全一样，如果想使用完全开源的软件，可以考虑采用GNU GPLv2许可证发布的netperf4工具（http://www.netperf.org/svn/netperf4/）。

Netperf可以测试网络性能的多个方面，主要包括使用TCP、UDP等协议的单向的批量数据传输模式和请求-响应模式的传输性能。

Netperf主要测试的项目包括：使用BSD Sockets的TCP和UDP连接（IPv4和IPv6）、使用DLPI接口的链路级别的数据传输、Unix Domain Socket、SCTP协议的连接（IPv4和IPv6）。Netperf采用客户机/服务器（Client/Server）的工作模式：服务端是netserver，用来侦听来自客户端的连接，客户端是netperf，用来向服务端发起网络测试。在客户端与服务端之间，首先建立一个控制连接，用于传递有关测试配置的信息和测试完成后的结果；在控制连接建立并传递了测试配置信息以后，客户端与服务端之间会另外建立一个测试数据连接，用来传递指定测试模式的所有数据；当测试完成数据连接就断开，控制连接会收集好客户端和服务端的测试结果然后让客户端展示给用户。为了尽可能模拟更多真实的网络传输场景，Netperf有非常多的测试模式供选择，包括TCP_STREAM、TCP_MAERTS、TCP_SENDFILE、TCP_RR、TCP_CRR、TCP_CC、UDP_STREAM、UDP_RR等。

（2）Iperf

Iperf是一个常用的网络性能测试工具，它是用C++编写的跨平台的开源软件，可以在Linux、UNIX和Windows系统上运行，其项目主页是http://sourceforge.net/projects/iperf/。

Iperf支持TCP和UDP的数据流模式的测试，用于衡量其吞吐量。与Netperf类似，Iperf也实现了客户机/服务器模式，Iperf有一个客户端和一个服务端，可以测量两端的单向和双向数据吞吐量。当使用TCP

功能时，Iperf测量有效载荷的吞吐带宽；当使用UDP功能时，Iperf允许用户自定义数据包大小，并最终提供一个数据包吞吐量值和丢包值。另外，有一个项目叫做Iperf3（项目主页为http://code.google.com/p/iperf/），它完全重新实现了Iperf，其目的是使用更小、更简单的源代码来实现相同的功能，同时也开发了可用于其他程序的一个函数库。

（3）NETIO

NETIO也是个跨平台的源代码公开的网络性能测试工具，它支持UNIX、Linux和Windows平台，其作者的关于NETIO的主页是http://www.ars.de/ars/ars.nsf/docs/netio。NETIO也是基于客户机/服务器的架构，它可以使用不同大小的数据报文来测试TCP和UDP网络连接的吞吐量。

（4）SCP

SCP是Linux系统上最常用的远程文件复制程序，它可以作为实际的应用来测试网络传输的效率。用SCP远程传输同等大小的一个文件，根据其花费时间的长短可以粗略评估出网络性能的好坏。

在本次网络性能测试中，采用Netperf基准测试工具和SCP工具来评估KVM虚拟化中客户机的网络性能。

8.4.2 测试环境配置

对KVM的网络虚拟化性能测试的测试环境配置，与8.2.2节的环境配置基本相同，下面仅说明不同之处和需要强调的地方。

本次对网络的性能测试中，使用的CPU资源为4个CPU线程，非虚拟化原生系统使用8GB内存，KVM宿主机拥有12GB内存，这是通过GRUB配置文件中的内核选项来控制的，如表8-4所示。

表 8-4 网络性能测试的内核选项配置情况

内核选项	配置情况
host kernel option	maxcpus=4 nr_cpus=4 mem=12G 3
native kernel option	maxcpus=4 mem=8G 3
guest kernel option	3 (i.e. run level is 3 in guest.)

网络测试需要增加相应的网卡，本次测试分别使用Intel 82576（代号Kawela）和Intel 82599（代号Niantic）两种型号的网卡进行测试。在测试SR-IOV类型的网络时，在加载这两种网卡的驱动程序时，打开SR-IOV的功能，使用的命令分别为modprobe igb max_vfs=2和modprobe ixgbe max_vfs=2。通过lspci命令查看已经打开SR-IOV功能后的网卡具体信息，示例如下：

```
02:00.0 Ethernet controller: Intel Corporation 82599EB
10-Gigabit SFI/SFP+ Network Connection (rev 01)     #82599 PF
02:10.0 Ethernet controller: Intel Corporation 82599 Ethernet
Controller Virtual Function (rev 01)  #82599 VF
<!-- 省略其他 PCI 设备的输出信息 -->
08:00.0 Ethernet controller: Intel Corporation 82576 Gigabit
Network Connection (rev 01)      #82576 PF
09:10.0 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)       #82576 VF
```

在本次测试中，分别对KVM客户机中使用默认rtl8139模式的网桥网络、使用e1000模式的网桥网络、使用virtio-net模式（QEMU做后端驱动）的网桥网络、使用virtio-net模式（vhost-net做后端驱动）的网桥网络、VT-d直接分配PF、SR-IOV直接分配VF等6种模式进行测试。为了实现这6种模式，启动客户机的qemu-kvm命令示例如下（使用82576网卡测试时没有使用第4种vhost=on配置）：

```
# 1. 默认 rtl8139 网卡的网桥网络
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -net nic -net tap

# 2. e1000 网卡的网桥网络
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -net nic,model=e1000 -net tap

# 3. virtio-net 模式的网桥网络（vhost=off）
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -netdev
tap,id=nic1,vhost=off,
vnet_hdr=on -device virtio-net-pci,netdev=nic1

# 4. virtio-net 模式的网桥网络（vhost=on）
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -netdev
tap,id=nic1,vhost=on,
vnet_hdr=on -device virtio-net-pci,netdev=nic1

# 5. VT-d 直接分配 PF
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -device pci-assign,host=02:00.0 -net none

# 6. SR-IOV 直接分配 VF
qemu-system-x86_64 -smp 4 -m 8G -drive
file=rhel6u3.img,if=virtio -device pci-assign,host=02:10.0 -net none
```

在本次测试中，网桥网络在宿主机中都绑定在82576或82599的PF上。另外，为了尽量排除交换机等瓶颈的影响，客户端和服务端是使用网线直连两块相同网卡来测试网络带宽的，在测试一块网卡在虚拟化中的性能时，客户端也使用同样型号的网卡，并且配置保持不变，同时调整服务器端KVM虚拟化启动客户机时使用的网卡参数，然后使用Netperf测试工具测试即可对比各种配置的网卡性能。使用SCP进行文件远程复制的测试时，需要在客户端准备一个20GB大小的文件，然后记录在服务端用SCP工具复制这个文件所需的时间。

本次实验中，KVM客户机网络配置的具体方法可以参考4.5.2节、5.1.4节和5.2节。

8.4.3 性能测试方法

在本次网络性能测试中，将被测试的原生系统或者KVM客户机作为服务端，分别测试服务端在各种配置方式下的网络性能。

（1）Netperf

首先下载Netperf最新的2.6.0版本，然后配置、编译、安装即可得到Netperf的客户端和服务端测试工具，分别是位于src目录下的netperf和netserver两个可执行文件，命令行操作示例如下：

```
[root@kvm-guest ~]#wget\
ftp://ftp.netperf.org/netperf/netperf-2.6.0.tar.gz
[root@kvm-guest ~]#tar -zxvf netperf-2.6.0.tar.gz
[root@kvm-guest ~]#cd netperf-2.6.0
[root@kvm-guest netperf-2.6.0]# ./configure
[root@kvm-guest netperf-2.6.0]# make
[root@kvm-guest netperf-2.6.0]# ls src/netperf
[root@kvm-guest netperf-2.6.0]# ls src/netserver
[root@kvm-guest netperf-2.6.0]#make install
```

在服务端运行netserver程序即可，命令行操作如下：

```
[root@kvm-guest ~]# netserver
Starting netserver with host 'IN(6)ADDR_ANY' port '12865' and family AF_UNSPEC
```

然后，在客户端运行netperf程序对服务端进行网络性能测试。假设服务端的IP地址为10.0.0.1，客户端IP地址为10.0.0.2。在客户端发起的Netperf测试的命令如下：

```
# 使用 Intel 82576 网卡时的 netperf 客户端命令
[root@net-client ~]#netperf -H 10.0.0.1 -l 60 -t TCP_STREAM -- -m 1500
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET
to 10.0.0.1 () port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

87380  16384   1500    60.03     940.73

# 使用 Intel 82599 网卡时的 netperf 客户端命令
[root@net-client ~]#netperf -H 10.0.0.1 -l 60 -t TCP_STREAM
<!-- 此处省略其余输出信息 -->
87380  16384  16384    60.00    9397.11
```

从上面的示例可以看出，使用Intel 82576千兆网卡时，某次测试的吞吐量（这里指带宽）为940.73 Mb/s，使用82599万兆网卡时带宽达到了9397.11 Mb/s。

在netperf命令中，-H name|IP表示连接到远程服务端的主机名或IP地址，-l testlen表示测试持续的时间长度（单位是秒），-t testname表示执行的测试类型（这里指定为TCP_STREAM这个很典型的类型）。“--”符号后面的是表示特定测试的一些选项，比如测试Intel 82576时用"-- -m 1500"指定客户端发送缓冲为1500字节。可以

使用netperf -h（或man netperf）命令查看netperf程序的帮助文档，更详细的解释可以参考Netperf 2.6.0的官方在线文档[6] 。

在Netperf测试过程的60秒内，同时使用sar命令记录KVM宿主机的40秒内的CPU使用率，命令行操作如下：

```
[root@jay-linux ~]# sar -u 2 20 &>sar-1.log
```

该命令表示每2秒采样一次CPU使用率，共采样20次。得到CPU利用率的采样日志sar-1.log的信息大致如下：

```
Linux 3.7.0 (jay-linux)  12/27/2012     _x86_64_       (4 CPU)

11:08:04 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
11:08:06 AM  all 13.03  0.00  5.78    0.00      0.00     71.19
11:08:08 AM  all  11.78  0.00 14.45   0.00      0.00     73.77
<!-- 此处省略其余信息 -->
Average:all 12.18  0.00  14.54   0.09      0.00     73.19
```

在本次实验中，主要观测了CPU空闲（idle）百分比的平均值（如上面信息的最后一行的最后一个数值），而将其余部分都视为已经被占用的CPU资源。在带宽能达到基本相同的情况下，CPU空闲的百分比越高，说明Netperf服务端消耗的CPU资源越少，这样的系统性能就越高。

另外，在本次Netperf的测试过程中，笔者发现在宿主机中KSM相关进程消耗了一些CPU资源，在内存资源充足的情况下，在KVM宿主机

中选择将KSM关闭能让82599网卡测试时得到的网络带宽数值稍微高一些，命令行操作如下：

```
[root@jay-linux ~]# service ksmtuned stop
Stopping ksmtuned:                                          [  OK  ]
[root@jay-linux ~]# echo 0 >/sys/kernel/mm/ksm/run
```

（2）SCP

在非虚拟化原生系统和KVM客户机中分别用SCP命令从一台用网卡直连的客户端机器远程复制一个20 GB的文件，复制完成后SCP命令会打印花费的时间。通过比较SCP时间的长短，即可以粗略地反映网络性能。SCP命令行操作如下：

```
[root@kvm-guest ~]# scp 10.0.0.2:/root/temp.img /tmp/
temp.img                              100%   20GB  64.4MB/s   05:18
```

8.4.4 性能测试数据

本次KVM虚拟化网络性能测试的Netperf和SCP基准测试都是分别收集了3次测试数据后计算的平均值。

（1）Netperf

使用Intel 82576、82599网卡对KVM网络虚拟化的性能进行测试得到的数据，与非虚拟化原生系统测试数据的对比，分别如图8-6和图8-

7所示。图中显示的吞吐量（throughput）越大越好，吞吐量相近时，CPU空闲百分比也是越大越好。

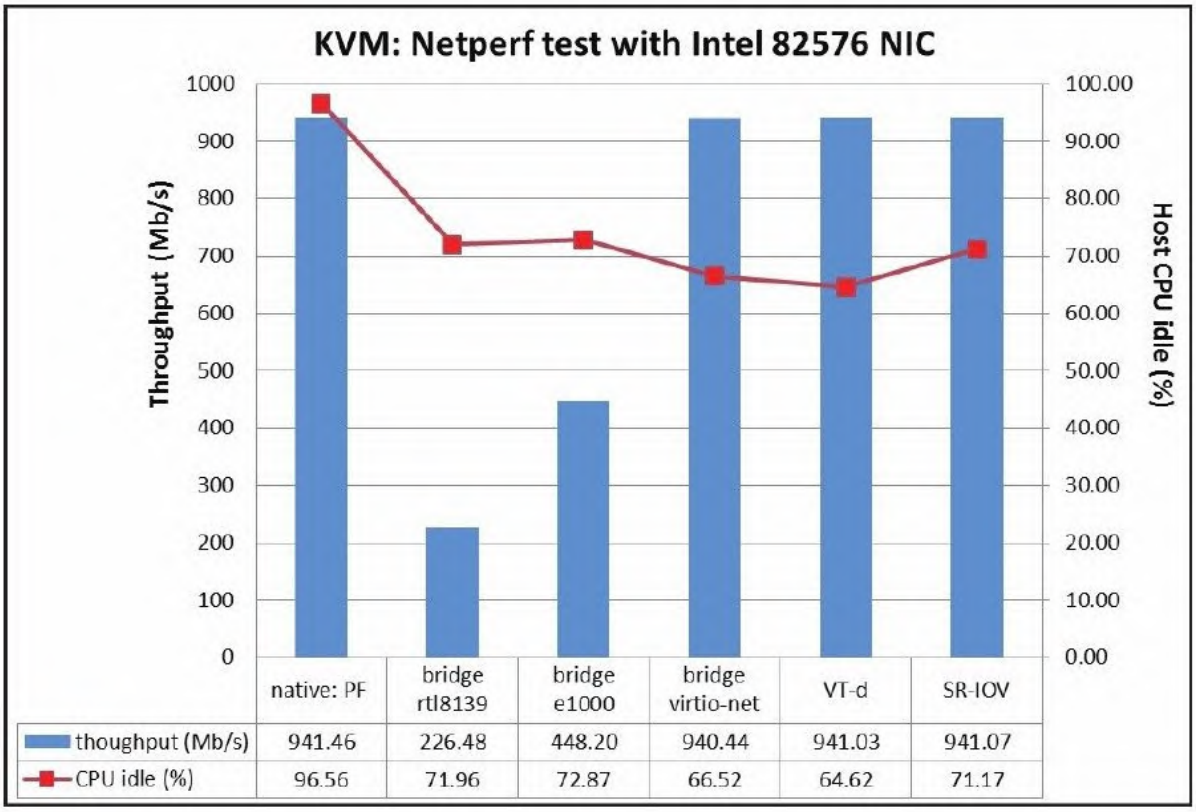


图　8-6　Netperf使用Intel 82576网卡的测试数据对比

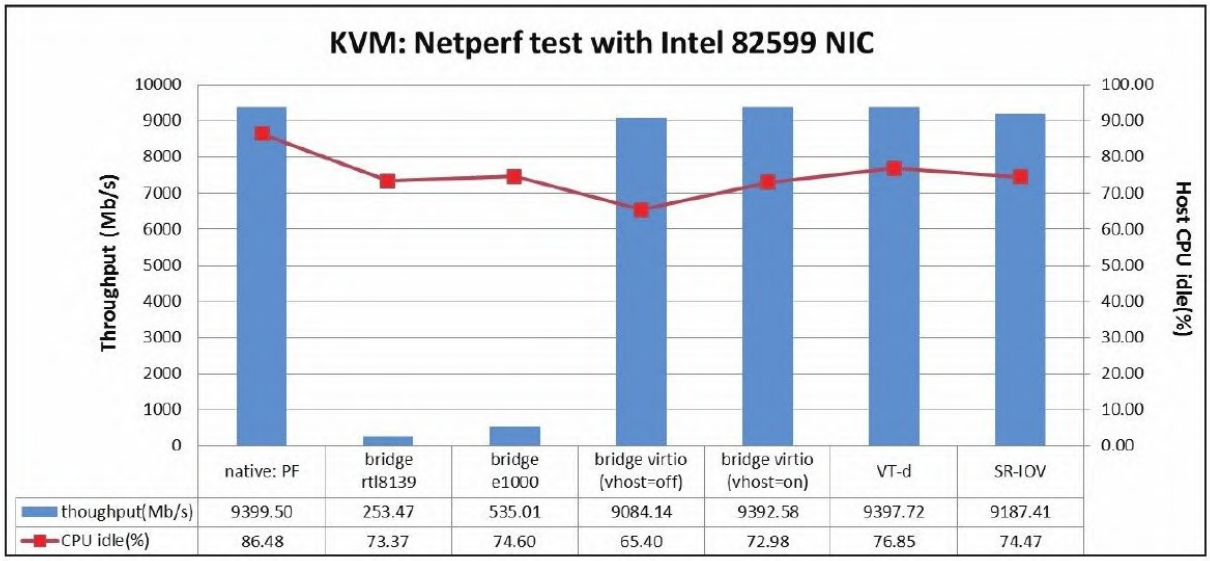


图　8-7　Netperf使用Intel 82599网卡的测试数据对比

从图8-6和图8-7中的信息可知，使用virtio、VT-d、SR-IOV等方式的网卡虚拟化可以达到和原生系统上的网卡差不多的高性能。其中，VT-d、SR-IOV方式直接分配网卡设备给客户机使用时，其CPU使用率比virtio的方式略低3%～5%（由于宿主机是4个CPU线程，CPU资源节省量相当于一个CPU线程的12%～20%），不过VT-d、SR-IOV的使用可能会丧失动态迁移的灵活性。virtio方式的网络设备比纯软件模拟的rtl8139、e1000型号网卡的性能要好得多，不过virtio方式要求客户机有virtio-net驱动才能使用网络；另外，在达到非常大的网络带宽时（如图8-7所示），使用vhost-net作为后端网络驱动的性能比使用QEMU作为后端驱动的virtio性能要好一些。在本次实验中纯软件模拟的rtl8139、e1000网卡的性能相对较差，吞吐量比较低，一般在500 Mb/s以内。不过笔者也曾使用其他型号的千兆以太网卡，比如采用

e1000模式的虚拟网卡在KVM虚拟化客户机中的吞吐量也能达到800 Mb/s。

（2）SCP

在原生系统和采用各种网络配置的KVM客户机中，使用SCP命令远程复制一个20GB的文件，记录其所花费时间的对比如图8-8和图8-9所示，其中所用的时间越短，表明网络性能越好。

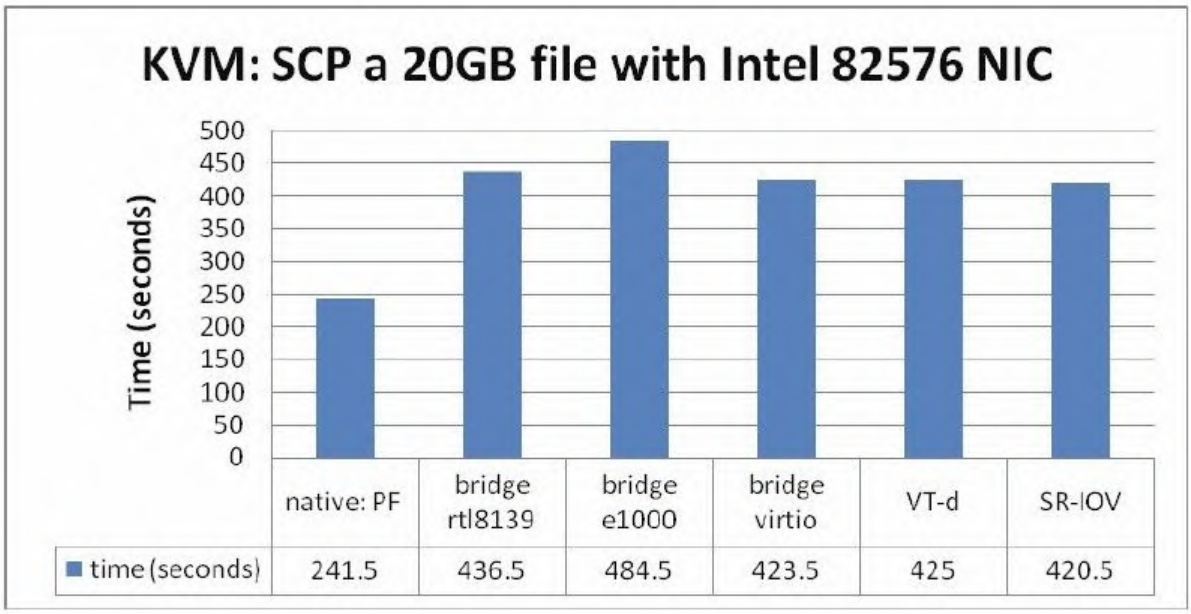


图　8-8　SCP使用Intel 82576网卡的测试数据对比

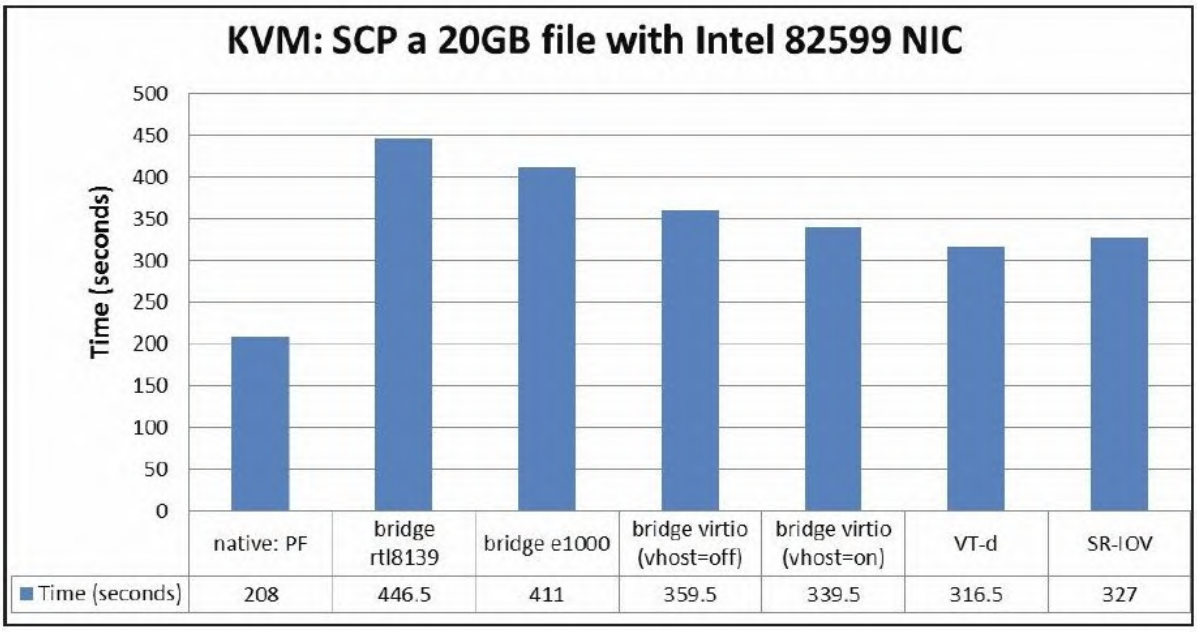


图　8-9　SCP使用Intel 82599网卡的测试数据对比

SCP测试结果对比的趋势与Netperf的测试结果相似。不过，客户机中采用几个不同的网络方式对它们传输时间的影响不是太大，这是因为SCP复制受到本地磁盘I/O的限制，即使网络性能很好也不能达到非常快的SCP复制速度。而在客户机中的SCP比原生系统的速度都要慢得多，这同样是因为原生系统的磁盘I/O速度比客户机中要快得多。

从本节对KVM的网络性能测试数据中，可以谨慎而粗略地得出如下推论：

1）virtio、VT-d和SR-IOV等方式的网络，可以达到和原生系统网络差不多的性能。

2）在达到相同网络带宽时，VT-d和SR-IOV方式占用的CPU资源比virtio略少。

3）使用vhost-net做后端驱动比使用QEMU做后端驱动的virtio网络性能略好。

4）纯软件模拟的rtl8139和e1000网卡的性能相对较差。

当然，这里的测试还不够完善（如Netperf工具仅测试了TCP_STREAM类型），与真正应用程序的网络使用情况并不完全相同。另外，由于QEMU纯软件模拟的rtl8139网卡的配置非常简单且兼容性很好，在对网络带宽不敏感的情况下，它依然是一个不错的选择，而且QEMU也默认提供模拟的rtl8139网卡。

8.5 磁盘I/O性能测试

在一个计算机系统中，CPU获取自身缓存数据的速度非常快，读写内存的速度也比较快，内部局域网速度也比较快（特别是使用万兆以太网），但是磁盘I/O的速度是相对比较慢的。很多日常软件的运行都会读写磁盘，而且大型的数据库应用（Oracle、MySQL等）都是磁盘I/O密集型的应用，所以在KVM虚拟化中磁盘I/O的性能也是比较关键的。测试磁盘I/O性能的工具有很多，如DD、IOzone、Bonnie++、Fio、iometer、hdparm等。下面简单介绍其中几个工具。

8.5.1 磁盘I/O性能测试工具

（1）DD

DD（命令为dd）是Linux上一个非常流行文件复制工具，在复制文件的同时可以根据其具体选项进行转换和格式化等操作。通过DD工具复制同一个文件（相同数据量）所需要的时间长短即可粗略评估磁盘I/O的性能。一般的Linux系统中都自带这个工具，用man dd命令即可查看DD工具的使用手册。

（2）IOzone

IOzone是一个常用的文件系统基准测试工具，它通过多种文件操作（如普通的读写、重读、重写、随机的读写）来衡量一个文件系统的性能。IOzone的官方主页是http://www.iozone.org/。IOzone可以运行在包括Linux、UNIX、Windows等在内的多个操作系统平台上。在测试完成之后，它还提供一个工具，可以根据测试结果绘制出3D图形以便看出I/O性能趋势。

（3）Bonnie++

Bonnie++是基于Bonnie[7] 代码编写的软件，使用一系列对硬盘驱动器和文件系统的简单测试来衡量其性能。Bonnie++可以模拟数据库去访问一个单一的大文件，也可以模拟Squid创建、读取和删除许多小文件。它可以实现有序地读写一个文件，也可以随机地查找一个文件中的某个部分，而且支持按字符方式和按块方式读写。

（4）hdparm

hdparm是一个用于获取和设置SATA和IDE设备的参数的工具，在RHEL 6.3中可以用yum install hdparm命令来安装hdparm工具。hdparm可以粗略地测试磁盘的I/O性能，通过如下的命令即可粗略评估sdb磁盘的读性能。

```
hdparm -tT /dev/sdb
```

本节选择DD、IOzone、Bonnie++这三种工具用于KVM虚拟化的磁盘I/O性能测试。

8.5.2 测试环境配置

对KVM的磁盘I/O虚拟化性能测试的测试环境配置，与8.2.2节的环境配置基本相同，下面仅说明不同之处和需要强调的地方。

在本次对磁盘I/O的性能测试中，尽量减少页面缓存的作用，使用较少内存。使用的CPU资源为两个CPU线程，非虚拟化原生系统使用2GB内存，KVM宿主机拥有4GB内存，这是通过GRUB配置文件中的内核选项来控制的，如表8-3所示（与8.3.2节相同）。

在实验中使用西部数据生产的1 TB的SATA硬盘，型号如下：

WDC-WD1002FAEX-00Z3A0（西部数据 1TB 硬盘）

由于文件系统的类型对磁盘I/O的性能是有影响的，因此本次磁盘I/O性能测试中，原生系统、KVM宿主机、KVM客户机系统中统一使用ext4格式的文件系统。

本次测试评估了QEMU/KVM中的纯软件模拟的IDE磁盘和使用virtio-blk驱动的磁盘（参考5.1.5节），启动客户机的qemu-kvm命令行分别如下：

```
#1. 使用 IDE 磁盘
qemu-system-x86_64 -smp 2 -m 2G -drive
file=rhel6u3.img,if=none,id=ide-disk0,format=raw,cache=none
-device ide-hd,drive=ide-disk0 -net nic -net tap

#2. 使用 virtio-blk 磁盘
qemu-system-x86_64 -smp 2 -m 2G -drive
file=rhel6u3.img,if=none,id=virtio-disk0,format=raw,cache=none
-device virtio-blk-pci,drive=virtio-disk0 -net nic -net tap
```

从上面的命令行中可以看出，客户机镜像文件都是raw格式的，并且配置"cache=none"用来绕过页面缓存。配置"cache=none"虽然绕过了页面缓存，但是没有绕过磁盘自身的磁盘缓存；如果要在宿主机中彻底绕过这两种缓存，可以在启动客户机时配置"cache=directsync"。不过由于"cache=directsync"配置会让客户机中磁盘I/O效率比较低，所以这种配置用得比较少，常用的配置一般为"cache=writethrough"、"cache=none"等。关于"cache=xx"选项的配置，可以参考4.4.1节中的详细介绍。

由于启动客户机时使用的磁盘配置选项"cache=xx"的设置对磁盘I/O测试结果的影响非常大，所以本次结果仅能代表"cache=none"这样配置下的一次基准测试。

8.5.3 性能测试方法

对非虚拟化的原生系统和KVM客户机都执行相同的磁盘I/O基准测试，然后对比其测试结果。

（1）DD

DD工具对读取磁盘上文件的测试，测试4种不同的块大小，使用的命令如下：

```
dd if=file.dat of=/dev/nulliflag=direct bs=1k count=100k
dd if=file.dat of=/dev/nulliflag=direct bs=8k count=100k
dd if=file.dat of=/dev/nulliflag=direct bs=1M count=10k
dd if=file.dat of=/dev/nulliflag=direct bs=8M count=2k
```

在上面命令中，if=xx表示输入文件（即被读取的文件），of=xx表示输出文件（即写入的文件），这里为了测试读磁盘的速度，所以读取一个磁盘上的文件，然后将其写到/dev/null[8] 这个空设备中。iflag=xx表示打开输入文件时的标志，此处设置为direct是为了绕过页面缓存，得到更真实的读取磁盘的性能。bs=xx表示一次读写传输的数据量大小，count=xx表示执行多少次数据的读写。

DD工具向磁盘上写入文件的测试，也测试4种不同的块大小，使用的命令如下：

```
dd if=/dev/zero of=dd1.dat conv=fsync oflag=direct bs=1k count=100k
dd if=/dev/zero of=dd1.dat conv=fsync oflag=direct bs=8k count=100k
dd if=/dev/zero of=dd1.dat conv=fsync oflag=direct bs=1M count=10k
dd if=/dev/zero of=dd1.dat conv=fsync oflag=direct bs=8M count=2k
```

在上面的命令中，为了测试磁盘写入的性能，使用了/dev/zero[9] 这个提供空字符的特殊设备作为输入文件。conv=fsync表示每次写入都要同步到物理磁盘设备后才返回，oflag=direct表示使用直写的方

式绕过页面缓存。conv=fsync和oflag=direct这两个配置都是为了写入数据是尽可能地绕过缓存，从而尽可能真实地反映磁盘的实际I/O性能。

关于dd命令的详细参数，可以用man dd命令查看其帮助文档。

（2）IOzone

下载IOzone源代码（http://www.iozone.org/src/current/iozone3_414.tar），解压后进入iozone3_414/src/current目录下运行make linux-AMD64命令即可编译。在编译完成后，当前目录就生成了iozone可执行程序。命令行操作示例如下：

```
[root@jay-linux ~]# cdiozone3_414
[root@jay-linuxiozone3_414]# cdsrc/current
[root@jay-linux current]# make linux-AMD64
[root@jay-linux current]# ls iozone
```

在RHEL 6.3系统上，也可以用yum install iozone命令直接安装IOzone。

在本次磁盘I/O性能测试中，使用IOzone工具的测试命令为：

```
iozone -s 512m -r 8k -S 20480 -L 64 -I -i 0 -i 1 -i 2 -Rab iozone.xls
```

在上面的命令参数中，-s 512m表示用于测试的文件大小为512 MB，-r 8k表示一条记录的大小（一次读写操作的数据大小）为8 KB，-S 20480表示本机的缓存大小是20480KB，-L 64表示缓存线路大小为64字节，-I表示使用直接I/O方式读写而绕过页面缓存，-i 0-i 1-i 2表示运行"0=write/rewrite,1=read/re-read,2=random-read/write"这三种测试，-Rab iozone.xls表示运行完整的自动模式进行测试并生成Excel格式的报告iozone.xls。其中-S、-L的值通过如下命令查询得到，当然，也可不填写而让IOzone自己决定。

```
[root@jay-linux ~]# cat /proc/cpuinfo | grep cache
cache size      : 20480 KB
cache_alignment : 64
```

IOzone是一个非常强大磁盘I/O测试工具，不同的参数可以得到不同的结果，关于其参数的详细信息，可以用man iozone命令查看，或者参考IOzone的官方文档[10] 。

（3）Bonnie++

从http://www.coker.com.au/bonnie++/网页下载bonnie++-1.03e.tgz文件，然后解压，对其进行配置、编译、安装的命令行操作如下：

```
[root@kvm-guest ~]#cdbonnie++-1.03e
[root@kvm-guest bonnie++-1.03e]# ./configure
[root@kvm-guest bonnie++-1.03e]# make
[root@kvm-guest bonnie++-1.03e]#make install
```

本次测试使用Bonnie++的命令如下：

```
bonnie++ -D -m kvm-guest -x 3 -u root
```

其中，-D表示在批量I/O测试时使用直接I/O的方式（O_DIRECT），-m kvm-guest表示Bonnie++得到的主机名为kvm-guest，-x 3表示循环执行3次测试，-u root表示以root用户运行测试。

在执行完测试后，默认会在当前终端上输出测试结果。可以将其CSV格式的测试结果通过Bonnie++提供的bon_csv2html转化为更容易读的HTML文档，命令行操作如下：

```
[root@kvm-guest bonnie++-1.03e]# echo
"native,4G,102817,88,58631,25,56712,4,108330,91,151383,7,299.
0,1,16,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++"
| perl bon_csv2html > native-bonnie-1.html
```

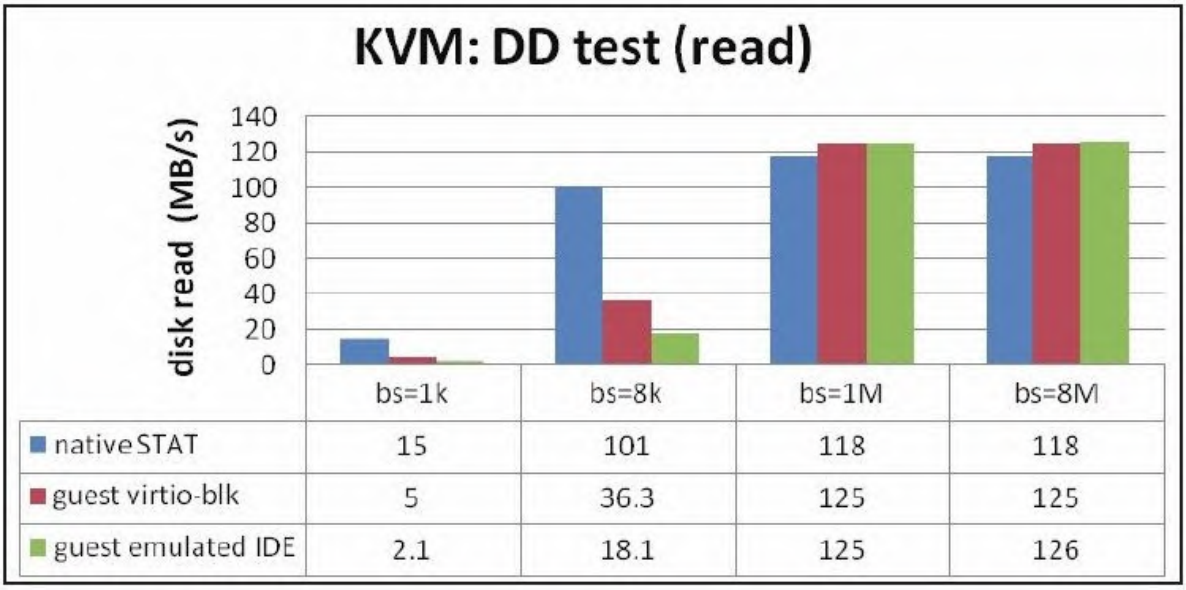
Bonnie++是一个强大的测试硬盘和文件系统的工具，关于Bonnie++命令的用法，可以用man bonnie++命令获取帮助手册，关于Bonnie++工具的原理及测试方法的简介，可以参考其源代码中的readme.html文档。

8.5.4 性能测试数据

分别用DD、IOzone、Bonnie++这3个工具在原生系统和KVM客户机中进行测试，然后对比其测试结果数据。为了尽量减小误差，每个测试项目都收集了3次测试数据，下面提供的测试数据都是根据3次测试计算出的平均值。

（1）DD

使用DD工具测试磁盘读写性能，将得到的测试数据进行对比，如图8-10和图8-11所示，读写速度越大，说明磁盘I/O性能越好。



	bs=1k	bs=8k	bs=1M	bs=8M
native STAT	15	101	118	118
guest virtio-blk	5	36.3	125	125
guest emulated IDE	2.1	18.1	125	126

图 8-10 DD测试的磁盘读取速度对比

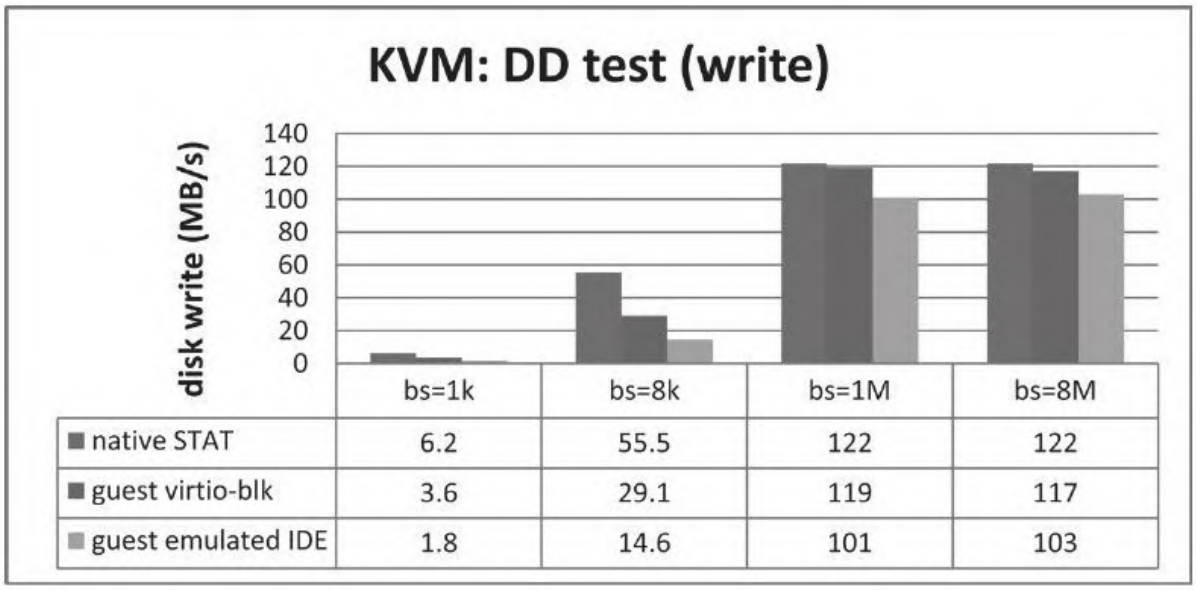


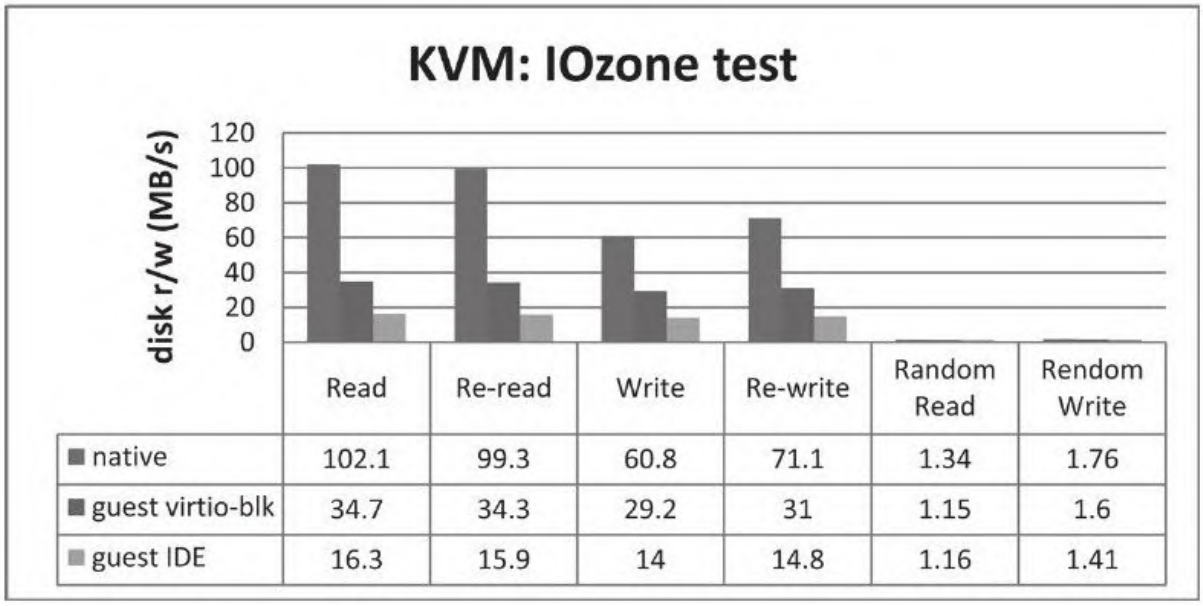
图　8-11　DD测试的磁盘写入速度对比

根据图8-10和图8-11中的数据可知，virtio方式的磁盘比纯模拟的IDE磁盘读写性能要好，特别是当一次读写的数据块较小时（bs=1k，bs=8k），virtio的读写速度大约是IDE的两倍。当一次读写的数据块较大时（如bs=1M，bs=8M），KVM客户机中virtio和IDE两种方式的磁盘的读写性能都与原生系统相差不大。另外，对于图8-10，当bs=1M和bs=8M时，客户机的磁盘读性能比原生系统略好，出现这种情况是因为宿主机使用的是3.7.0的Linux内核，其磁盘I/O读性能比原生系统RHEL 6.3的性能更好一些。

（2）IOzone

IOzone的测试数据对比如图8-12所示。从图8-12中的数据可知，virtio方式的磁盘顺序读写性能比纯模拟的IDE磁盘要高不少，不过与

原生系统的磁盘I/O性能相比仍然有不少的差距。在随机读写测试场景中，virtio和IDE方式都与原生系统的性能差别不大。另外，本次IOzone测试的是一次读写8 KB大小的数据块（-r 8k参数），本次测试结果与使用DD工具测试的图8-10和图8-11中bs=8k时的测试数据基本一致。



	Read	Re-read	Write	Re-write	Random Read	Rendom Write
native	102.1	99.3	60.8	71.1	1.34	1.76
guest virtio-blk	34.7	34.3	29.2	31	1.15	1.6
guest IDE	16.3	15.9	14	14.8	1.16	1.41

图　8-12　IOzone测试的磁盘读写速度对比

（3）Bonnie++

Bonnie++的测试结果对比如图8-13所示，其中的s-w表示顺序写（sequential write）、s-r表示顺序读（sequential read），s-w-per-char就表示按字符的顺序写，s-w-block表示按块的顺序写，s-rewrite表示顺序重写，s-r-per-char按字符的顺序读，s-r-block表

示按块的顺序读，Random Seeks表示随机改变文件读写指针偏移量（使用lseek()和random()函数）。

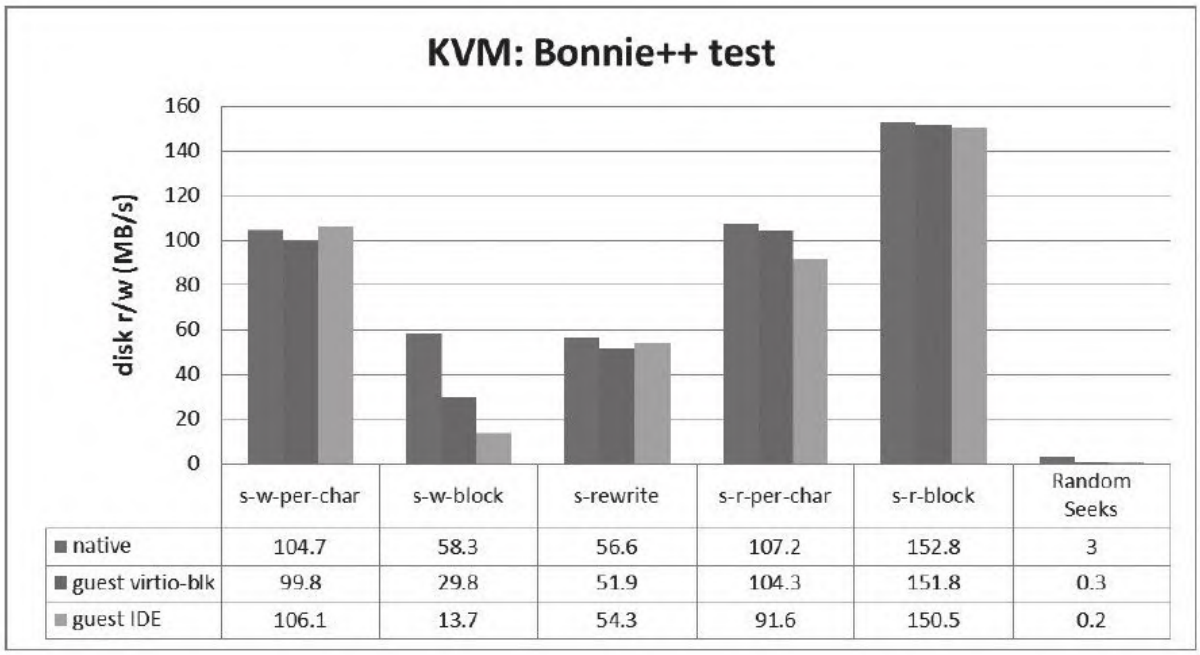


图　8-13　Bonnie++测试的磁盘I/O速度对比

本次Bonnie++测试没有指定一次读写数据块的大小，默认值是8 KB，所以图8-13的测试结果与前面DD、IOzone工具测试每次读写8 KB数据时的测试结果是基本一致的。

从图8-10到图8-13可以看出，当一次读写的数据块较小时，KVM客户机中的磁盘读写速度比非虚拟化原生系统慢得比较多，而一次读写数据块较大时，磁盘I/O性能差距不大。在一般情况下，用virtio方式的磁盘I/O性能比纯模拟的IDE磁盘要好一些。

8.6 本章小结

本章介绍了通过在KVM客户机与非虚拟化原生系统中运行相同的基准测试，然后将测试结果进行对比的方法来测试KVM虚拟化的性能，详细介绍了针对CPU、内存、网络、磁盘等各个部分的一些测试工具和测试方法，还介绍了笔者在测试KVM性能测试时的详细配置和最后得到的测试数据。在本章的所有测试过程中，笔者仅启动了1个客户机进行测试，这可能与实际虚拟化应用中启动很多个客户机的场景并不一样，所以本章提及的测试结果可能比实际应用时能达到的性能结果要好一些。另外，实际的CPU、内存、网卡、硬盘等硬件环境，KVM内核配置和版本，qemu-kvm版本，启动客户机的命令行参数，运行基准测试的命令行参数等都可能对KVM的性能测试结果产生较大的影响，所以本章的测试数据和结论都只是仅供参考，读者在实际测试中可能会得出不完全一致的结果。尽管数据和结论都比较粗略，但本章提供了一些较好的测试方法和工具可供读者在实际测试KVM虚拟化性能时参考，希望起到抛砖引玉的作用。

8.7　本章注释和参考阅读

1.本章注释

[1]12306是全国铁路统一客服电话号码，12306网站（http://www.12306.cn）是铁路客户服务官方网站，提供火车票查询、网上订票、铁路知识和新闻公告、货运信息查询等服务。

[2]SPEC CPU2006中针对整型计算的基准测试，见http://www.spec.org/cpu2006/CINT2006/，针对浮点型计算的基准测试，见http://www.spec.org/cpu2006/CINT2006/

[3]金田康正（Yasumasa Kanada）是日本的一个数学家，他因为多次打破计算圆周率π的小数点位数的记录而出名。2002年到2009年期间，他保持的计算圆周率π后小数点的位数的世界纪录为1.2411 T（1T=10的9次方）位。

[4]关于SPEC CPU2006的runspec的命令及其参数的细节，可以参考其官方网页http://www.spec.org/cpu2006/Docs/runspec.html

[5]图8-1中的CPU2006测试结果仅基于4个CPU线程，而Xeon E5-4650有64个CPU线程，图8-1中的测试结果并不是针对整个服务器的性能测试。图中的整型计算得分是70.9，将其乘以16会得到1134.4，这

个得分与Intel官方网页（http://www.intel.com/content/www/us/en/benchmarks/server/xeon-e5-4600/xeon-e5-4600-SPECrate-int-throughput.html）中公布的SPEC INT测试的得分1180比较接近。

[6]Netperf 2.6.0的官方在线文档，可参考网页http://www.netperf.org/svn/netperf2/tags/netperf-2.6.0/doc/netperf.html

[7]Bonnie是由Tim Bray编写的一个对UNIX文件系统进行性能测试的软件，其主页为http://www.textuality.com/bonnie/

[8]/dev/null是一个空设备，也称为位桶（bit bucket）或者黑洞（black hole）。我们可以向它输入任何数据，但任何写入它的数据都会被丢弃。它通常用于处理不需要的输出流。

[9]/dev/zero设备可以无穷尽地提供空字符（ASCII NUL，0x00），可以提供任何需要的数目。它通常用于向设备或文件写入多个空字符，用于初始化数据存储。

[10]IOzone的使用文档，可以从http://www.iozone.org/docs/IOzone_msword_98.pdf下载。

2.参考阅读

Linux上的一些基准测试工具：

http://www.gnutoolbox.com/linux-benchmark-tools/index.php

第9章　参与KVM开源社区

KVM是开源虚拟化软件的一个典范，前面的章节已经介绍了很多关于KVM虚拟化的功能、实例和原理，本章将介绍QEMU/KVM开源社区、基本的代码结构以及如何参与到QEMU/KVM开源社区中学习和贡献自己的力量。

9.1　开源社区介绍

“开源”一词在近几年的IT业界中是一个非常热门的词汇，还有一种精神叫做“开源精神”。开源即开放源代码（Open Source），需要保证任何人都可以根据“自由许可证”（Free Licence）获得软件的源代码，并且允许任何人对代码进行修改并重新发布。在英文中，Free有两个意思：一个是免费，一个是自由。当提及"Free Software"（自由软件）时，应该将"Free"理解为“自由的演讲”（Free Speech）中的自由，而不是“免费的啤酒”（Free Beer）中的免费。

在开源软件的发展历史上，诞生了许多优秀的开源软件，很多大大小小的IT企业基于多种开源软件来构建属于自己的计算机系统或服务。一般来说，每个开源软件都有一个对应的开源社区，在社区中可

以自由地讨论该软件的开发、设计、未来发展方向、用户使用时遇到的问题等。本节将简单介绍KVM相关的一些开源社区的基本情况。

9.1.1 Linux开源社区

KVM是Linux内核的一个模块，因此KVM社区也与Linux内核社区非常类似，也可以算作Linux内核社区的一部分。这里介绍的Linux开源社区主要是指Linux内核的社区而不是Linux系统的用户工具和发行版社区。

Linux是自由软件与开源软件中的经典代表作，也是其中最著名的项目。Linux是一个操作系统内核，由Linus Torvalds于1991年第一次正式发布。由于其开源的特性，通过全世界一些优秀程序员的参与，在过去的二十多年里Linux内核取得了突飞猛进的发展。从最初Linus一个人的操作系统项目，发展到如今Linux成为全世界最流行的操作系统内核，有数以千万计的服务器和个人电脑运行Linux系统，而且目前移动互联网时代使用最广泛的移动操作系统——Android系统也是基于Linux内核而开发，Android系统已经运行在数以十亿计的智能手机或平板电脑上。Linux内核的代码量，从最初发布时0.01版本的1万行代码，发展到最新3.9.0版本的1600多万行代码。现在有1万多人向Linux内核贡献过代码，其中长期活跃在社区的开发者超过1千人。

作为Linux内核的缔造者，Linus Torvalds是Linux世界中最著名的黑客，直到现在他依然是Linux内核代码最主要的维护者（maintainer）。不过，由于代码量和开发者的不断增长，Linus也没足够的时间和精力（也可能是能力）来审查到内核代码中的每一个细节，现在Linus不会亲自审查每一个新的补丁、每一行新的代码，而是将大部分的工作都交给各个子系统的维护者去处理，各个维护者下面可能还有一些驱动程序或特殊几个文件的维护者，Linus自己主要工作是把握一些大的方向，以及将各个子系统维护者的分支合并（merge）到Linux主干树。如图9-1所示，普通的Linux内核开发者一般都是向驱动程序或子系统的维护者提交代码，经过维护者和开发社区中的其他人审核之后，提交的代码才能进入子系统维护者的开发代码仓库，之后进入每个Linux内核的合并窗口（merge window），最后才能进入真正的Linux upstream[1] 代码仓库中去。

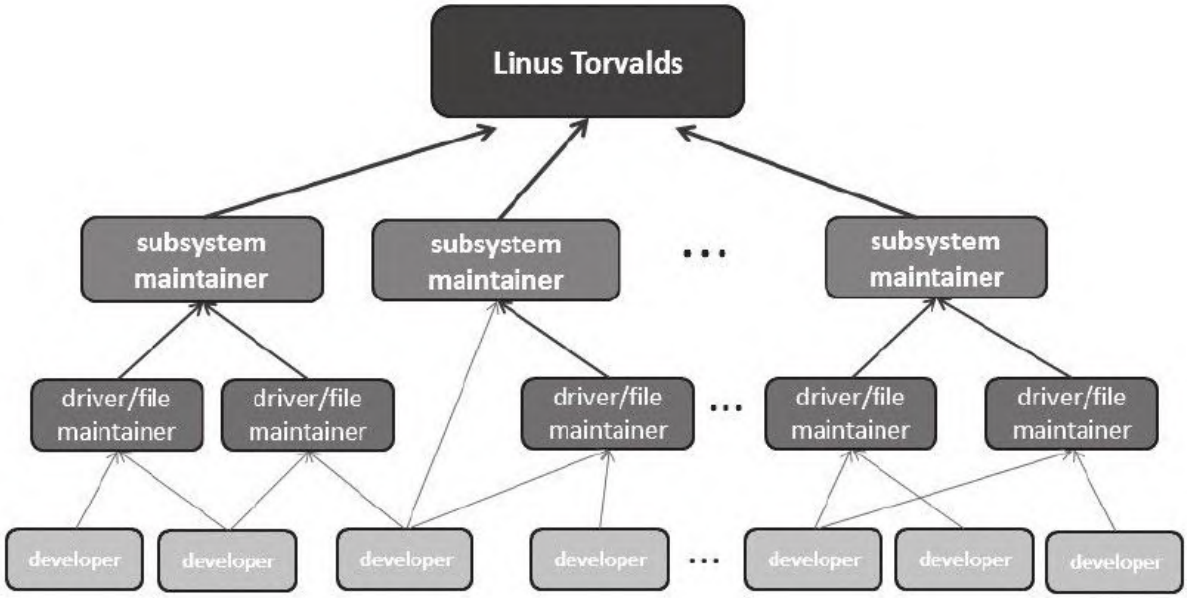


图　9-1　Linux内核代码维护层次结构

在Linux内核中，有许多的子系统（如内存管理、PCI总线、网络子系统、ext4文件系统、SCSI子系统、USB子系统、KVM模块等），也分为更细的许多驱动（如Intel的以太网卡驱动、USB XHCI驱动、Intel的DRM驱动等）。各个子系统和驱动分别由相应的开发者来维护，如PCI子系统的维护者是来自Google的Bjorn Helgaas，而Intel以太网卡驱动（如igb、ixgbe）的维护者是来自Intel的Jeff Kirsher、Greg Rose等，USB XHCI驱动（即USB 3.0）的维护者是Intel的Sarah Sharp。关于Linux内核中维护者相关的信息，可以查看内核代码中名为"MAINTAINERS"的文件。

从2011年5月开始（在Linux内核诞生20周年时），Linus舍弃了原本该使用2.6.40作为版本号的2.6内核命名方式，将其下一个Linux内

核版本命名为3.0，从而让Linux内核进入了3.x版本时代。在写作本章之时，Linux内核的最新版本是在2013年4月29日发布的3.9版本。目前，Linux内核版本的发布周期一般是2到3个月，在这段时间中，一般会发布7、8个RC[2] 版本，如2013年2月19日发布了3.8版本，随后4月29日发布了3.9版本，期间有Linux 3.9-rc1到3.9-rc8等8个RC版本。

总之，Linux开源社区主要工作是围绕Linux内核开发及其应用，其核心就是Linux操作系统，目前有非常多的优秀开发者活跃在Linux社区，Linux系统也得到了业界的极大认可，这使得社区发展速度非常的快。关于Linux社区的更多信息，可以参考9.6节推荐的一些资源。

9.1.2 KVM开源社区

KVM（Kernel Virtual Machine）是Linux内核中原生的虚拟机技术，KVM内核部分的代码全部都在Linux内核中，是Linux内核的一部分（相当于一个内核子系统）。

KVM最初是由以色列的Qumranet公司的Avi Kivity等人开发的虚拟机系统，在Linux内核2.6.20版本时加入到Linux upstream中，从此正式成为Linux内核的一部分。在2008年，Red Hat公司以1亿多美元的价格收购了Qumranet，并且在之后的RHEL 6操作系统中放弃了Xen，而使用KVM作为其虚拟化产品。KVM得到了Linux内核开发者喜爱，有不少独

立开发者在向KVM贡献代码，而且也得到了Red Hat、IBM、Intel、Novell、Google等公司的支持。

目前，KVM代码的维护者是来自Red Hat公司的Gleb Natapov和Paolo Bonzini，所有关于KVM虚拟机代码的改动都必须经过他们两个维护者的审核才能加入KVM开发仓库中去，进而进入Linus维护的Linux主干代码中。由于KVM支持不同的处理器架构（如x86、PowerPC、ARM等），对于各个架构也有不同的代码维护者，他们也负责审核处理器架构相关部分的KVM内核代码，不过最后还是需要由Gleb和Marcelo两位维护者将审核后的代码加入KVM代码仓库中。根据Linux内核代码中"MAINTAINERS"文件的描述，KVM AMD-V的维护者是Joerg Roedel，KVM PowerPC的维护者是来自Suse的Alexander Graf，KVM IA64（安腾）的维护者是来自Intel的Xiantao Zhang，KVM s390的维护者是来自IBM的Christian Borntraeger和Cornelia Huck，KVM ARM的维护者是Christoffer Dall。其中，值得一提的是，在Linux 3.9中第一次将KVM对ARM架构的支持正式加入Linux主干代码中。

除了平时在KVM邮件列表中的讨论之外，KVMForum是KVM开发者、用户等相互交流的一个重要会议。KVM Forum每年举行一届，会议内容主要涉及KVM的一些最新开发的功能、未来的发展思路以及管理工具的开发。关于KVM Forum，可以参考其官方网页http://www.linux-

kvm.org/page/KVM_Forum，该网页提供了各届KVM Forum议程和演讲文档的下载。

9.1.3 QEMU开源社区

QEMU（Quick EMUlator）是一个实现硬件虚拟化的开源软件，它通过动态二进制转换实现对中央处理单元（CPU）的模拟，并且提供了对一整套模拟的设备模型，从而可以使未经修改的各种操作系统得以在QEMU上运行。QEMU最初是由一个天才级的程序员Fabrice Bellard在2003年第一次正式发布的开源软件项目，如今QEMU已经是硬件模拟全虚拟化领域中最著名的开源软件。QEMU项目的官方网站为qemu.org或qemu-project.org。QEMU当前的最新版本是2013年5月发布的1.5.0版本。

QEMU自身就是一个独立的完整的虚拟机模拟器，可以独立模拟CPU和其他一些基本外设，除此之外，QEMU还可以为KVM、Xen等流行的虚拟化技术提供设备模拟功能。使用KVM时，在其用户态的工具中，目前使用最广泛也是最流行的就是QEMU。在QEMU的1.3.0版本之前，KVM社区专门维护了一个qemu-kvm.git代码仓库，那时KVM虚拟化的一小部分功能还没有融入QEMU社区最普通的qemu.git代码仓库中，所以使用KVM时最好专门使用qemu-kvm，而不是普通的QEMU。不过，从2012年12月发布的QEMU1.3.0版本开始，KVM相关的全部代码都已经合并到QEMU社

区的qemu.git主干中[3]，所以使用KVM时只需普通的QEMU即可，而不需要使用专门定制的qemu-kvm。从qemu-kvm.git仓库（http://git.kernel.org/cgit/virt/kvm/qemu-kvm.git）可以看到，qemu-kvm最后一次发布的版本是2012年9月的qemu-kvm-1.2.0，此后qemu-kvm就没有单独的版本发布了。值得注意的是，在使用普通QEMU配合KVM使用时，在QEMU命令行启动客户机时要加上"-enable-kvm"参数来使用KVM加速，而较旧的qemu-kvm命令行默认开启使用KVM加速的选项。在Xen虚拟化方面，从Xen 4.3版本开始，也支持直接使用普通的QEMU作为设备模型[4]。

QEMU社区的代码仓库在http://git.qemu.org/，其中qemu.git就是QEMU upstream的主干代码树。可以使用GIT工具，通过两个URL（git://git.qemu.org/qemu.git和http://git.qemu.org/git/qemu.git）中的任意一个下载QEMU的最新源代码。

从QEMU源代码中的"MAINTAINERS"文件可知，QEMU的维护者是来自IBM的Anthony Liguori，QEMU中与KVM相关部分代码的维护者就是KVM的维护者Gleb Natapov和Paolo Bonzini，QEMU中与Xen相关部分代码的维护者是来自Citrix的Stefano Stabellini。

目前，QEMU/KVM社区的开发方式是：与KVM相关的新功能代码或者修复bug的代码需要发到KVM邮件组中，经过Gleb Natapov和Marcelo

Tosatti审核后，加入qemu-kvm.git代码仓库的uq/master分支中，然后再定期地由Gleb或Paolo将这些KVM的相关代码推进到Anthony维护的qemu.git主干代码仓库中，才正式成为QEMU主干的一部分。

9.1.4 其他开源社区

几乎每一个知名的开源软件都有一个对应的开源社区，而且其中部分开源软件项目还有对应的官方网站。除了前面介绍的Linux、KVM、QEMU等开源社区，本节再列举一些知名的开源软件社区及其官方网站，希望对读者了解更多的优秀开源软件和参与开源社区起到抛砖引玉的作用。

1）Libvirt，一个虚拟化API项目。

网址：libvirt.org

2）OpenStack，一个扩展性良好的开源云操作系统。

网址：openstack.org

3）CloudStack，一个提供高可用性、高扩展性的开源云计算平台。

网址：cloudstack.apache.org/

4）Xen，一个功能强大的虚拟化软件。

网址：xen.org

5）Ubuntu，一个开源的、非常流行的Linux操作系统。

网址：www.ubuntu.com

6）Fedora一个开源的Linux操作系统。

网址：fedoraproject.org

7）CentOS，一个基于RHEL开源代码进行重新编译构建的开源Linux操作系统。

网址：www.centos.org/

8）openSUSE，一个开源社区驱动的开放式开发的Linux操作系统。

网址：www.opensuse.org

9）Apache httpd，一个非常流行的开源HTTP服务器软件。

网址：httpd.apache.org/

10）Nginx，一个开源的高性能HTTP服务器和反向代理软件。

网址：nginx.org

11）Hadoop，一个开源的、可靠的、可扩展的分布式计算和数据存储框架。

网址：hadoop.apache.org

12）Python，一种非常流行的完全面向对象的编程语言。

网址：www.python.org

9.2 代码结构简介

9.2.1 KVM代码

KVM作为Linux内核中的一个模块而存在，KVM的代码已经在Linux 2.6.20版本时即完全加入到Linux内核主干代码仓库中了。由于Linux内核过于庞大，因此KVM开发者的代码一般是先进入KVM社区的kvm.git代码仓库，再由KVM维护者定期地将代码提供给Linux内核的维护者（Linus Torvalds）并由其添加到Linux内核代码仓库中。可以分别查看如下两个网页来了解最新的KVM开发和Linux内核的代码仓库的状态。

❑https://git.kernel.org/cgit/virt/kvm/kvm.git/

❑https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/

KVM内核部分的代码主要有三部分组成：KVM框架的核心代码、与硬件架构相关的代码和KVM相关的头文件。在Linux 3.10.0版本中，KVM相关的代码行数大约是11万多行，下面就以基于Linux 3.10.0版本的kvm.git为例来对KVM代码结构进行简单的介绍。

1.KVM框架的核心代码

是与具体硬件架构无关的代码，位于virt/kvm/目录中，有7309行代码，这部分的代码目录结构如下：

```
[root@jay-linux kvm.git]# ls virt/kvm/
assigned-dev.c  async_pf.h        coalesced_mmio.h  ioapic.c
iodev.h  irqchip.c   Kconfig
async_pf.c      coalesced_mmio.c  eventfd.c         ioapic.h
iommu.c  irq_comm.c  kvm_main.c
```

其中，kvm_main.c文件是KVM代码中最核心的文件之一，kvm_main.c中的kvm_init函数是与硬件架构无关的KVM初始化入口。

2.与硬件架构相关的代码

不同的处理器架构有对应的KVM代码，位于arch/$ARCH/kvm/目录中。KVM目前支持的架构包括：x86、ARM、PowerPC、S390、IA64、MIPS等。与处理器硬件架构相关的KVM代码目录如下：

```
[root@jay-linux kvm.git]# ls -d arch/*/kvm
arch/arm/kvm  arch/ia64/kvm  arch/mips/kvm  arch/powerpc/kvm
arch/s390/kvm  arch/tile/kvm  arch/x86/kvm
```

在KVM支持的架构中，Intel和AMD的x86架构的功能是最完善和成熟的。x86架构相关的代码位于arch/x86/kvm/目录中，有37076行代码，其代码结构如下：

```
[root@jay-linux kvm.git]# ls arch/x86/kvm/
cpuid.c     i8254.c  irq.c     kvm_cache_regs.h  Makefile
mmu.h           pmu.c     tss.h  x86.h
cpuid.h     i8254.h  irq.h     lapic.c           mmu_audit.c
mmutrace.h      svm.c     vmx.c
emulate.c   i8259.c  Kconfig   lapic.h           mmu.c
paging_tmpl.h   trace.h   x86.c
```

其中，vmx.c和svm.c分别是Intel和AMD CPU的架构相关模块kvm-intel和kvm-amd主要代码。以vmx.c为例，其中vmx_init()函数是kvm-intel模块加载时执行的初始化函数，而vmx_exit()函数kvm-intel模块卸载时会执行的函数。

3.KVM相关的头文件

在前面提及的代码中，一般都会引用到一些相关的头文件，包括与各个处理器架构相关的头文件（位于arch/*/include/asm/kvm*）和其他的头文件。KVM相关的头文件的结构如下：

```
[root@jay-linux kvm.git]# ls arch/*/include/asm/kvm*
arch/x86/include/asm/kvm_emulate.h
arch/x86/include/asm/kvm_host.h
arch/x86/include/asm/kvm_guest.h
arch/x86/include/asm/kvm_para.h
<!-- 省略其他输出 -->

[root@jay-linux kvm.git]# find include/ -name "kvm*"
include/uapi/linux/kvm_para.h
include/uapi/linux/kvm.h
include/uapi/asm-generic/kvm_para.h
include/linux/kvm_types.h
include/linux/kvm_para.h
include/linux/kvm_host.h
include/asm-generic/kvm_para.h
include/trace/events/kvm.h
```

另外，在Linux内核代码仓库中还有一些关于KVM的文档，主要位于Documentation/virtual/kvm/*和Documentation/*/kvm*.txt，如下：

```
[root@jay-linux kvm.git]# ls Documentation/virtual/kvm/*
Documentation/virtual/kvm/api.txt
Documentation/virtual/kvm/msr.txt
Documentation/virtual/kvm/cpuid.txt
Documentation/virtual/kvm/nested-vmx.txt
Documentation/virtual/kvm/hypercalls.txt
Documentation/virtual/kvm/ppc-pv.txt
Documentation/virtual/kvm/locking.txt
Documentation/virtual/kvm/review-checklist.txt
Documentation/virtual/kvm/mmu.txt
Documentation/virtual/kvm/timekeeping.txt

Documentation/virtual/kvm/devices:
mpic.txt  README  xics.txt
[root@jay-linux kvm.git]# ls Documentation/*/kvm*.txt
Documentation/ia64/kvm.txt  Documentation/powerpc/kvm_440.txt
Documentation/s390/kvm.txt
```

9.2.2 QEMU代码

QEMU 1.3.0版本已将qemu-kvm中的代码全部合并到纯QEMU中，此后使用KVM只需普通QEMU即可，不需要专门为KVM而改动的qemu-kvm。原来的qemu-kvm代码仓库和普通的QEMU代码仓库分别如下：

❑qemu-kvm:http://git.kernel.org/cgit/virt/kvm/qemu-kvm.git

❑QEMU：http://git.qemu.org/?p=qemu.git;a=summary

QEMU代码实现了对PC客户机的完全模拟，可以独立使用，也可以与KVM、Xen等Hypervisor一起使用来实现完整的虚拟化功能。QEMU中的代码比较复杂，实现的功能也非常多，本节只是简单提及一下QEMU中与KVM相关部分代码，以及KVM客户机创建过程的关键函数调用。

在QEMU（本次使用的版本是1.5.0）源码中，最重要的一个文件是vl.c，其中的main()函数就是QEMU工具的主函数，它主要处理QEMU的各个命令行参数，然后启动客户机并让vCPU运行起来。在QEMU代码中，对KVM提供相关支持的函数定义在kvm-all.c文件中。

QEMU通过IOCTL函数使用/dev/kvm设备来调用KVM内核模块提供的API，从而与KVM内核模块进行交互。提供的API中包括：创建客户机（KVM_CREATE_VM）、为客户机创建vCPU（KVM_CREATE_VCPU）、运行vCPU（KVM_RUN）、获取KVM的版本信息（KVM_GET_API_VERSION）、查询KVM的特性支持（KVM_CHECK_EXTENSION）等。QEMU中将这一些列IOCTL函数调用分为了3个类别：kvm_ioctl()、kvm_vm_ioctl()和kvm_vcpu_ioctl()，这3个函数都是在kvm-all.c文件中定义的，分别用于表示对KVM内核模块本身、对虚拟客户机和对客户机的vCPU来进行交互。

QEMU配合KVM来启动一个客户机，首先打开/dev/kvm设备，通过名为KVM_CREATE_VM的IOCTL调用来创建一个虚拟机对象，然后通过

KVM_CREATE_VCPU为虚拟机创建vCPU对象，最后通过KVM_RUN让vCPU运行起来。这个抽象后的简化过程的函数调用关系如下：

```
main(): vl.c
   |
configure_accelerator():vl.c ->kvm_init(): kvm_all.c
   |                            |
machine->init():vl.c      qemu_open("/dev/kvm"): kvm_all.c
   |                              |
pc_init_pci():pc_piix.c   kvm_ioctl(KVM_CREATE_VM): kvm_all.c
   |
pc_init1():hw/i386/pc_piix.c
   |
pc_cpus_init():hw/i386/pc.c
   |
pc_new_cpu(): hw/i386/pc.c
   |
cpu_x86_create():hw/i386/pc.c
     | （此处省略多次调用关系）
x86_cpu_common_class_init(): target-i386/cpu.c
   |
x86_cpu_realizefn():target-i386/cpu.c
   |
qemu_init_vcpu(): cpus.c
     |


qemu_kvm_start_vcpu(): cpus.c
   |
qemu_thread_create(): util/qemu-thread-posix.c
   |
pthread_create():  #create vCPU thread
   |
qemu_kvm_cpu_thread_fn():  cpus.c
   |
kvm_init_vcpu: kvm-all.c
   |
kvm_vm_ioctl(KVM_CREATE_VCPU)
```

当然，这里为了简化起见仅仅提及CPU相关的重要部分，整个初始化过程还做了很多其他的事情，比如中断芯片的模拟、内存的模拟、寄存器的设置等。

9.2.3 KVM单元测试代码

KVM相关的代码除了KVM和QEMU自身的代码之外，还包括本节介绍的KVM单元测试代码和下一节将介绍的KVM Autotest代码。KVM单元测试代码是用于测试一些细粒度的系统底层的特性（如：客户机CPU中的MSR的值），在一些重要的特性被加入KVM之时，KVM维护者也会要求代码作者在KVM单元测试代码中添加对应的测试用例。KVM单元测试的代码仓库也是存放在Linux内核社区的官方站点的，参考网页链接为https://git.kernel.org/cgit/virt/kvm/kvm-unit-tests.git/。

KVM单元测试的代码目录结构如下：

```
[root@jay-linux kvm-unit-tests.git]# ls
api                 config.mak                  config-x86_64.mak
formats         kvmtrace.c          Makefile     README          x86-run
apic.out            config-powerpc-440.mak  config-x86-common.mak
invpcid.out     kvmtrace_format     msr.out      run_tests.sh
config-i386.mak     config-powerpc.mak          COPYRIGHT
iotable.c       lib                 output.log   testdev.txt
config-ia64.mak     configure                   flat.lds
iotable.h       main-ppc.c          powerpc      x86
```

其中，lib目录下是一些公共的库文件，包括系统参数、打印、系统崩溃的处理，以及lib/x86/下x86架构相关的一些基本功能函数的文件（如apic.c、msr.h、io.c等）；x86目录下是关于x86架构下的一些具体测试代码（如msr.c、apic.c、vmexit.c等）。

KVM单元测试的基本工作原理是：将编译好的轻量级测试内核镜像（*.flat文件）作为支持多重启动的QEMU的客户机内核镜像来启动，测试使用一个通过客户机BIOS来调用的基础结构，该基础结构将会主要初始化客户机系统（包括CPU等），然后切换到长模式（x86_64 CPU架构的一种运行模式）并调用各个具体测试用例的主函数从而执行测试，在测试完成后QEMU进程自动退出。

编译KVM单元测试代码是很容易的，直接运行make命令即可；在编译完成后，在x86目录下会生成很多具体测试用例需要的内核镜像文件（*.flat）。

执行测试时，可以用QEMU命令行来执行某个具体的测试用例。对于QEMU 1.2及之前的版本，运行msr这个测试的命令行示例如下：

```
qemu-kvm -device testdev,chardev=testlog -chardev
file,id=testlog,path=msr.out -serial stdio
-kernel ./x86/msr.flat
```

对于QEMU 1.3及之后的版本（本节中代码示例为QEMU 1.5.0版本），运行msr这个测试的命令行示例如下：

```
qemu-system-x86_64 -enable-kvm -device pc-testdev -serial stdio
-deviceisa-debug-exit,iobase=0xf4,iosize=0x4
-kernel ./x86/msr.flat
```

其中，-kernel选项后的./x86/msr.flat文件即为被测试的内核镜像。测试结果会默认打印在当前执行测试的终端上。

KVM单元测试代码中还提供了一些脚本以便让单元测试的执行更容易，如x86-run脚本可以方便地执行一个具体的测试。执行msr测试的示例如下：

```
[root@jay-linux kvm-unit-tests.git]# ./x86-run ./x86/msr.flat
qemu-system-x86_64 -enable-kvm -device pc-testdev -device
isa-debug-exit,iobase=0xf4,iosize=0x4 -serial stdio -device
pci-testdev -kernel ./x86/msr.flat
VNC server running on '::1:5901'
enabling apic
MSR_IA32_APICBASE: PASS
MSR_IA32_APICBASE: PASS
IA32_SYSENTER_CS: PASS
<!-- 此处省略部分输出信息 -->
MSR_SYSCALL_MASK: PASS
MSR_*STAR eager loading: PASS
15 tests, 0 failures
Return value from qemu: 1
```

run_tests.sh脚本可以默认运行x86/unittests.cfg文件中配置的所有测试：

```
[root@jay-linux kvm-unit-tests.git]# ./run_tests.sh
PASS apic
PASS smptest
PASS smptest3
<!-- 此处省略部分输出信息 -->
skip taskswitch2 (i386 only)
PASS kvmclock_test
PASS pcid
```

9.2.4 KVM Autotest代码

Linux中有一个Autotest测试框架，主要为了完全自动化地执行Linux上的测试（主要是Linux内核测试，目前也有其他用途的测试）。作为Linux上的一个著名项目，KVM也借鉴并从Linux的Autotest框架中派生出专门针对KVM虚拟化测试的框架，这就是KVM Autotest框架。目前，Red Hat和IBM公司的KVM测试都在使用该套测试框架，KVM Autotest的代码和文档都放在Github上，详情见链接https://github.com/autotest/virt-test/。

KVM Autotest框架包括了许多的虚拟化功能测试和性能测试，非常强大，也比较复杂。它的基本架构如图9-2所示。

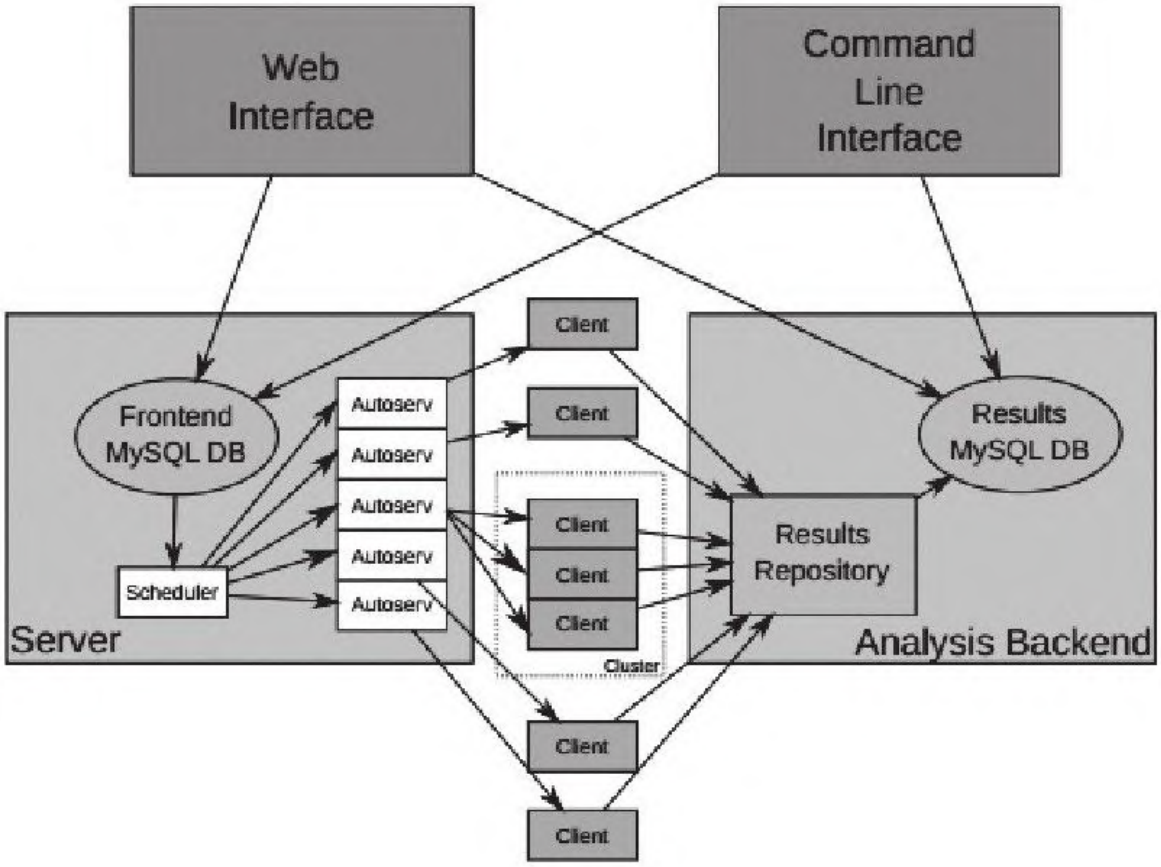


图　9-2　KVM Autotest的基本架构

其中，Web界面是基于Django框架的，主要用于创建和管理测试任务、管理执行客户机等；数据库使用的是MySQL，主要用于保存测试任务、测试结果、测试机器等信息；命令行提供了autotest命令行工具来调用需要执行的测试任务。

KVM Autotest代码的目录结构如下：

```
[root@jay-linux virt-test.git]# ls
__init__.py  libvirt  MAINTAINERS  qemu        run     tests
v2v      virttestlibguestfs   LICENSE  openvswitch
README.rst  shared  tools  virt.py
```

其中，在qemu、libvirt、openvswitch、v2v等目录下分别对相应程序进行测试，比如在qemu/tests目录下执行QEMU命令行来执行KVM测试。

KVM Autotest是基于GPL协议的一个开源测试框架，可以通过邮件列表和IRC参与到Autotest社区的代码开发和问题讨论。邮件列表订阅地址为http://www.redhat.com/mailman/listinfo/virt-test-devel；IRC频道为irc.oftc.net#virt-test。

9.3　向开源社区贡献代码

开源软件有着自由和开放的特性，任何开发者或者用户都可以参与到开源社区中参与代码的开发和讨论实际使用中遇到的问题。在使用Linux、KVM、QEMU等开源软件获得它们的价值的同时，如果能将一些修复bug的代码或者增加某个新功能的代码贡献给开源社区，就可以形成一些正反馈，如果越来越多的人向开源社区做出贡献，那么开源社区就会越来越活跃，开源软件的质量会逐渐提高，软件的功能也会逐渐增多。

也许有不少读者会问，为开源社区做贡献并且贡献代码应该很困难吧？我的回答是，也难也不难。如果要新增一个比较大的功能特性到开源软件（特别是Linux等大型系统软件），这是比较困难的，因为这需要开发者对该软件的代码非常熟悉并有很强的开发能力，而且要在开源社区中进行多次讨论，要试图说服一些维护者接受你的想法和代码。当然，也可以通过相对来说比较容易做的一些事情来参与开源社区，比如，参与社区的一些讨论，帮助解决一些初级使用者遇到的小问题；可以对软件做一些测试工作并将发现的bug反馈到社区；为缺乏文档的开源软件添加说明文档并加入到它的代码库中；可以为某些复杂的函数添加几句注释；修改其中的一些错别字（哪怕是printf输

出的文字）；还可以为开源软件编写测试代码以便丰富测试用例的覆盖，等等。

9.3.1 开发者邮件列表

开源社区的沟通交流方式有很多种，如邮件列表、IRC[5]、wiki、博客、论坛等。一般来说，开发者的交流使用邮件列表比较多，普通用户则使用邮件列表、论坛、IRC等多种方式。关于KVM和QEMU开发相关的讨论，主要都依赖于邮件列表。

电子邮件列表，是互联网上最早的社区形式之一，其起源可以追溯到20世纪70年代，它是Internet上的一种重要工具，用于各种群体之间的信息交流和信息发布。邮件列表有两种基本形式：

❑公告型（邮件列表），通常由一个管理者向小组中的所有成员发送信息，如电子杂志、新闻邮件等；

❑讨论型（讨论组），所有的成员都可以向组内的其他成员发送信息，其操作过程简单来说就是发一个邮件到小组的公共电子邮件，通过系统处理后，将这封邮件分发给组内所有成员。

❑KVM和QEMU等开发社区使用的邮件列表是属于讨论型的邮件列表，任何人都可以向该列表中的成员发送电子邮件。

尽管非订阅用户也可以向KVM、QEMU等开发者邮件列表发送邮件，如果需要查收到邮件列表中的邮件，还是需要对相应的邮件列表进行订阅。下面分别介绍KVM和QEMU的邮件列表及其订阅方式。

KVM开发者邮件列表是kvm@vger.kernel.org，KVM内核部分以及QEMU中与KVM相关部分的讨论都会在该邮件列表中进行。订阅方法是向majordomo@vger.kernel.org邮箱发送一封以"subscribe kvm"为正文的邮件（可能需要邮箱验证确认）。在Outlook中查看KVM邮件列表中的邮件，如图9-3所示。

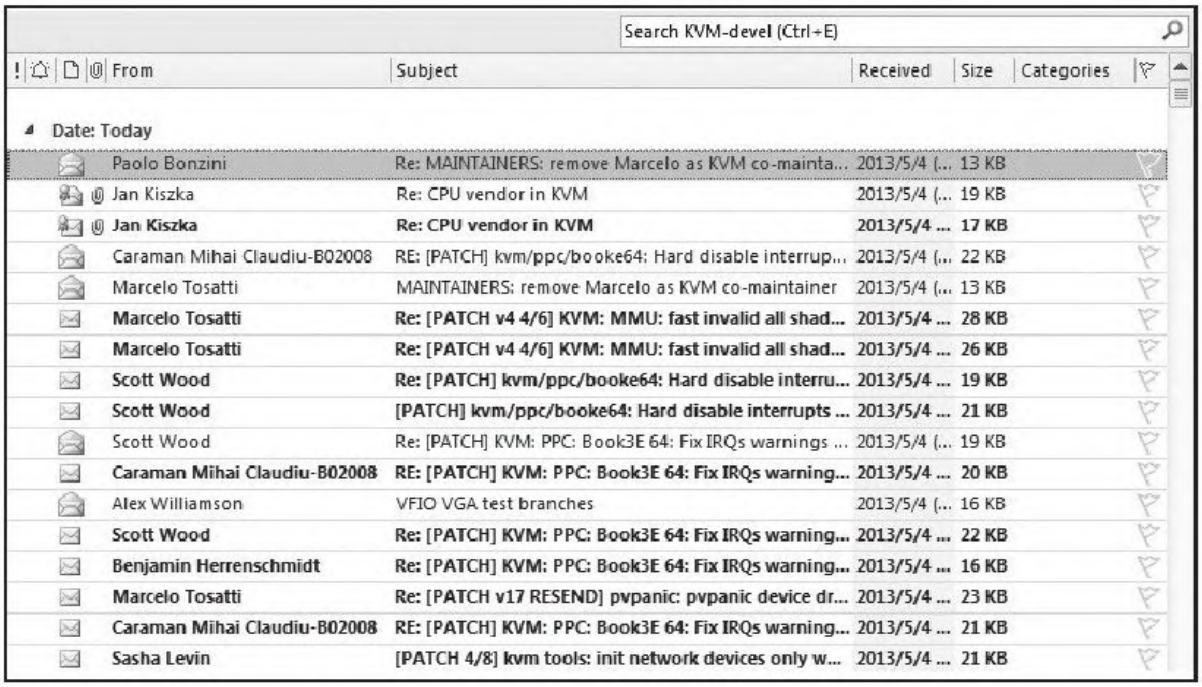


图　9-3　在Outlook中查看KVM邮件

KVM邮件列表和IRC等交流方式的说明，可参考http://www.linux-kvm.org/page/Lists%2C_IRC。

QEMU开发者邮件列表是qemu-devel@nongnu.org，QEMU普通用户讨论的邮件列表是qemu-discuss@nongnu.org。可以根据一个网页（http://lists.nongnu.org/mailman/listinfo/qemu-devel）中的指导信息来订阅QEMU开发者邮件列表。QEMU邮件列表的说明，可参考http://wiki.qemu.org/MailingLists.

在向KVM、QEMU等邮件列表发送邮件之时，有几点需要注意：

❑邮件内容都使用英文（包括个人签名和问候语）。

❑邮件使用纯文本格式（如Outlook中写邮件时选择为"plain text"格式）而尽量不要使用HTML格式（当然有时为了描述清楚问题也可能会附加图片或外部URL链接）。

❑回复别人邮件时尽量在他人提问题的地方进行相应的回复而不是直接将内容写在邮件的开头部分。

除了邮件列表之外，KVM的IRC讨论方式为在irc.freenode.net服务器上的"#kvm"频道，QEMU的IRC讨论方式为在irc.oftc.net服务器上的"#qemu"频道。

9.3.2 代码风格

代码风格是指程序开发人员编写程序源代码时的书写风格。良好的代码风格可以让代码易于阅读、也让源代码易于维护。代码风格就像人的衣着打扮，漂亮的一致的代码风格会给人以清爽、干净的良好形象。很多软件项目都有自己统一的代码风格，在写代码时要严格遵守这一规范，如果每个人写的代码风格千变万化，会为阅读和维护代码带来很大的困难，从而可能影响到项目的开发质量和开发进度。尽管代码风格并没有一个放之四海而皆准的规范，但随着时间的推移和项目的实践，也有一些较好的代码风格规范，如网上比较有名的有Linux内核代码风格、Google代码风格[6] 、华为编程规范等。KVM和QEMU都分别有各自的代码风格，本节对其进行简单介绍。

1.KVM内核部分的代码风格

由于KVM是Linux内核的一个模块，所有KVM内核代码都会最终被合并到Linux内核中，故KVM的代码风格就是完全依照Linux内核规定的代码风格。关于Linux内核的代码风格，在Linux内核代码仓库中的Documentation/CodingStyle文件中有详细的说明，另外也可以查看其网页版本[7] 。这里简单列举其中的几项代码风格。

（1）缩进

使用制表符来表示缩进而不是使用空格，一个制表符长度是8个字符，不要试图使用4个（甚至2个）字符长度来表示缩进。不过，在注

释、文档和Kconfig中可以使用空格来表示缩进。8个字符长度的制表符的缩进可以让代码更容易阅读，还有一个好处是当函数嵌套太深的时候可以给出警告。不要把多个语句放在一行里，除非你有什么东西要隐藏；也不要在一行里放多个赋值语句。

在switch语句中消除多级缩进的首选方式是让"switch"和从属于它的"case"标签对齐于同一列，而不要“两次缩进”"case"标签。正确示例如下：

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
        mem <<= 10;
        /* fall through */
default:
        break;
}
```

（2）把长的行和字符串打散

每一行长度的限制是80列，强烈建议遵守这个惯例。长于80列的语句要打散成有意义的片段。每个片段要明显短于原来的语句，而且放置的位置也明显地靠右。同样的规则也适用于有很长参数列表的函数头。长字符串也要打散成较短的字符串。例如：

```
void fun(int a, int b, int c)
{
    if (condition)
            printk(KERN_WARNING "this is a long printk with"
                                "3 parameters a: %u b: %u "
                                   "c: %u \n", a, b, c);
    else
            next_statement;
}
```

（3）大括号的位置

在C语言风格中，另外一个常见问题是大括号的放置。和缩进大小不同，选择或弃用某种放置策略并没有多少技术上的原因，不过首选的方式，就像Kernighan和Ritchie[8] 提出的，是把起始大括号放在行尾，而把结束大括号放在行首，这适用于所有的非函数语句块（if、switch、for、while、do）。示例如下：

```
if (x is true) {
        we do y
}
```

不过，有一个例外，那就是函数：函数的起始大括号放置于下一行的开头，示例如下：

```
int function(int x)
{
        body of function
}
```

结束大括号独自占据一行，除非它后面跟着同一个语句的剩余部分，也就是do语句中的"while"或者if语句中的"else"，示例如下：

```
do {
        body of do-loop
} while (condition);
```

当只有一个单独的语句的时候，不用加大括号，示例如下：

```
if (condition)
        action();
```

这点不适用于本身为某个条件语句的一个分支的单独语句。这时需要在两个分支里都使用大括号。示例如下：

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

（4）空格的使用

在Linux内核中，空格使用方式（主要）取决于它是用于函数还是关键字。（大多数）关键字后要加一个空格。值得注意的例外是sizeof、typeof、alignof和__attribute__，这些关键字在某些程度上看起来更像函数（它们在Linux中也常常伴随小括号使用，尽管在C语言里这样的小括号不是必需的，就像"struct fileinfo info"声明后的"sizeof info"）。所以在if、switch、case、for、do、while等关键字之后放一个空格，但是不要在sizeof、typeof、alignof或者__attribute__等关键字之后放空格。另外，也不要在小括号内的表达式两侧加空格。正确示例如下：

```
s = sizeof(struct file);
```

当声明指针类型或者返回指针类型的函数时，“*”的首选使用方式是使之靠近变量名或者函数名，而不是靠近类型名。示例如下：

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

在大多数二元和三元操作符两侧使用一个空格，例如下面所有这些操作符：

```
=  +  -  <>  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

但是一元操作符后不要加空格：

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

后缀自加和自减一元操作符（++和--）前不加空格，前缀自加和自减一元操作符后不加空格。“.”和“→”结构体成员操作符前后不加空格。最后，也不要在行尾留空白。

（5）命名

C是一个简朴的语言，在命名时也应该保持这样的风格。和Modula-2和Pascal程序员不同，C程序员不使用类似ThisVariableIsATemporaryCounter这样华丽的名字。C程序员会称那个变量为"tmp"，这样写起来会更容易，而且至少不会令其难于理解。

不过，虽然混用大小写的名字是不提倡使用的，但是全局变量还是需要一个具描述性的名字。称一个全局函数为"foo"是一个难以饶恕的错误。全局变量（只有真正需要它们的时候才用它）需要有一个具

描述性的名字，就像全局函数。如果有一个可以计算活动用户数量的函数，应该叫它"count_active_users()"或类似的名字，不应该叫它"cntuser()"。

Linux内核是不会像“匈牙利命名法”[9] 那样在函数名中包含函数类型的。

本地变量名应该简短，而且能够表达相关的含义。如果有一些随机的整数型的循环计数器，应该称它为"i"。叫它"loop_counter"并无益处，如果它没有被误解的可能。类似的，"tmp"可以用来称呼任意类型的临时变量。

2.QEMU的代码风格

作为KVM的用户态工具的QEMU的代码风格在QEMU代码仓库中的名为CODING_STYLE的文件中有详细的讲述。QEMU代码风格与Linux内核（包括KVM）的不完全相同，主要包含以下几点规范：

（1）缩进

使用4个空格符作为QEMU中的代码缩进，一般不会用到制表符。只有在Makefile中才使用制表符，因为Makefile的规范要求在某些情况下一定要用到制表符。注意QEMU的缩进和Linux内核中的规定是不一样

的，KVM开发者在开发KVM内核程序时使用内核的代码风格，开发qemu-kvm（或普通QEMU）时使用QEMU的代码风格。

（2）每行的长度

每行最多是80个字符，如需更多字符时需要换行书写，这与Linux内核代码风格的要求是一致的。

（3）命名规范

变量名是由小写字母和下划线组成的，如lower_case_with_uderscores。结构体和枚举类型是由首字母大写的多个词组成，如CamelCase。标量类型是由小写字母、下划线最后加一个_t后缀组成，如target_phys_addr_t。当包装标准库函数时，一般使用"qemu_"前缀以便让读者知道这是一个QEMU包装过的函数版本，如qemu_gettimeofday()。

（4）语句块结构

每一个缩进的语句块都应该用大括号括起来，即使语句块只包含一个语句，示例如下：

```
if (a == 5) {
    printf("a was 5.\n");
} else if (a == 6) {
    printf("a was 6.\n");
} else {
    printf("a was something else entirely.\n");
}
```

注意，这个规范与Linux内核的不太一样（内核代码中只有一条语句的语句块一般不要使用大括号括起来，只需要有缩进来标志即可）。

函数中大括号使用位置的示例如下（与Linux内核的风格相同）：

```
void a_function(void)
{
    do_something();
}
```

9.3.3 生成patch

patch（补丁），在计算机软件开发中是一个普遍使用的术语，在使用微软的Windows操作系统时，经常能收到为了修复某个安全漏洞而向用户提供的补丁。在开源软件的开发社区中，patch是指用于添加新功能或者修复某个bug的代码修改，一般来说，它是两个代码版本的差异信息。patch是GNU diff工具生成的输出内容，这种输出格式能够被patch工具正确读取并添加到相应的代码仓库中。当然，patch文件还

可以由一些源代码版本控制工具来生成，如SVN、CSV、Mercurial、Git等。

在Linux内核社区中，所有的开发代码都以patch的形式发送到开发者邮件列表中，然后经过讨论、审核、测试等之后才会被项目（或子项目）维护者加入到代码仓库中去。QEMU/KVM的功能开发和修复bug的代码，也都是以patch的形式发送出去的。

在目前的KVM社区中，生成KVM内核部分的patch，是要基于kvm.git代码仓库的next分支来进行开发的，而QEMU部分的patch，是要基于qemu-kvm.git代码仓库的uq/master分支来开发的（目前也有部分人直接基于qemu.git来开发patch）。在动手修改代码做自己的patch之前，需要先将自己的工作目录切换到对应的开发分支并更新最新的代码中去，示例如下：

```
[root@jay-linux kvm.git]# git checkout next
Branch next set up to track remote branch next from origin.
Switched to a new branch 'next'
[root@jay-linux kvm.git]# git branch
  master
* next
[root@jay-linux kvm.git]# git pull

[root@jay-linux qemu-kvm.git]# git checkout uq/master
Branch uq/master set up to track remote branch uq/master from origin.
Switched to a new branch 'uq/master'
[root@jay-linux qemu-kvm.git]# git branch
  master
* uq/master
[root@jay-linux qemu-kvm.git]# git pull
```

下面以kvm.git代码仓库为例，分别介绍使用diff工具和使用Git工具来生成patch的方法。生成QEMU的patch的方法和KVM内核中是完全一样的，只是修改各自的代码时注意遵循各自的并不完全相同的代码编写风格。

1.使用diff工具生成patch

最简单的生成patch的方法是准备两个代码仓库，其中一个是未经任何修改的代码仓库，另一个就是经过自己修改后的代码仓库。假设kvm-my.git是经过修改后的代码仓库，kvm.git是修改前的原生代码仓库，可以使用如下的命令来生成patch：

```
[root@jay-linux kvm_demo]# diff -urN kvm.git/ kvm-my.git/ > my.patch
```

其中，-u标志表示生成的patch使用统一后的diff格式，以便于读懂；-r标志表示让diff工具循环遍历所有的目录，-N标志表示让diff工具将新增加的文件也添加到diff生成的patch中。

假设只是对virt/kvm/kvm_main.c文件了一点修改，生成的my.patch文件中的内容示例如下：

```
diff -urN kvm.git/virt/kvm/kvm_main.c
kvm-my.git/virt/kvm/kvm_main.c
--- kvm.git/virt/kvm/kvm_main.c 2013-05-26 16:39:37.837048437 +0800
+++ kvm-my.git/virt/kvm/kvm_main.c      2013-05-26 16:49:47.836021500 +0800
@@ -3105,6 +3105,8 @@
        int r;
        int cpu;
```

```
+       printk(KERN_INFO "Hey, KVM is initializing.\n");
+
        r = kvm_arch_init(opaque);
        if (r)
                goto out_fail;
```

当然，如果只是想对某个修改的文件或目录生成patch，则可以使用如下的命令：

```
[root@jay-linux kvm_demo]# diff -u kvm.git/virt/kvm/kvm_main.c
kvm-my.git/virt/kvm/kvm_main.c > my.patch
[root@jay-linux kvm_demo]# diff -urN kvm.git/virt/kvm/
kvm-my.git/virt/kvm/ > my.patch
```

对于刚才这样生成的patch，在使用它时可以用如下命令：

```
[root@jay-linux kvm.git]# patch -p1 < ../my.patch
patching file virt/kvm/kvm_main.c
```

其中，-p1表示跳过patch中的第1层目录（即忽略patch中的kvm-my.git和kvm.git这样的1级目录），因为patch使用者当前的目录可能与patch生成者使用的目录命名不是一样。加上-p1参数后，patch使用者就不用考虑与patch生成者的目录一致性了。

diffstat这个工具可以生成patch引起的变化的统计信息，包括：修改的文件位置、被改动的文件个数、增加或删除的行数。在往邮件列表发送patch时，最好能能够将diffstat工具统计的信息附加在patch之前。由于patch工具使用patch时，会忽略"diff"关键字之前的

所有行中的信息，所以统计信息不会影响到patch的使用。stat使用示例如下：

```
[root@jay-linux kvm_demo]# diffstat -p1 my.patch
 virt/kvm/kvm_main.c |     2 ++
 1 file changed, 2 insertions(+)
```

2.使用Git工具生成patch

由于kvm.git和qemu-kvm.git（或qemu.git）代码仓库都是使用Git进行源代码版本管理的，所以在KVM的开发中也通常使用Git工具来生成patch。在Git代码仓库中，用Git工具生成patch只需要简单的两个步骤：第一步是使用"git commit"命令在本地仓库中提交修改内容；第二步是使用"git format-patch"命令生成所需的patch文件。如下的命令演示了使用git生成了一个patch并继续修改再生成另外一个patch的过程。

```
# 假设前面已经对 virt/kvm/kvm_main.c 进行了修改
# 将修改的文件添加到 git 管理的索引中
[root@jay-linux kvm.git]# git add virt/kvm/kvm_main.c
# 提交修改到本地仓库
[root@jay-linux kvm.git]# git commit -m "just a demo"
[next 86abe87] just a demo
 1 files changed, 2 insertions(+), 0 deletions(-)
# 根据前面的修改，生成对应的 patch
[root@jay-linux kvm.git]# git format-patch -1
0001-just-a-demo.patch
```

```
# 做了另一些其他的修改之后，提交本次修改内容
[root@jay-linux kvm.git]# git commit -m "just another demo"
[next 486236a] just another demo
 1 files changed, 1 insertions(+), 0 deletions(-)
# 分别生成两次提交的patch
[root@jay-linux kvm.git]# git format-patch -2
0001-just-a-demo.patch
0002-just-another-demo.patch
```

其中，git commit命令的-m参数表示添加对本次提交的描述信息；git format-patch-N命令表示从本地最新的提交开始往前根据最新的N次提交信息生成对应的patch；git format-patch origin命令可以生成所有的本地提交对应的patch文件（在原来的代码仓库中已存在的信息则不会生成patch）。

生成的两个patch中的第一个为0001-just-a-demo.patch，内容如下：

```
From 86abe871f15004faa9a950f445dd710ec70f97bc Mon Sep 17 00:00:00 2001
From: Jay<smile665@gmail.com>
Date: Sun, 26 May 2013 17:56:49 +0800
Subject: [PATCH 1/2] just a demo

---
 virt/kvm/kvm_main.c |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

diff --git a/virt/kvm/kvm_main.c b/virt/kvm/kvm_main.c
index 302681c..f944735 100644
--- a/virt/kvm/kvm_main.c
+++ b/virt/kvm/kvm_main.c
@@ -3105,6 +3105,8 @@ int kvm_init(void *opaque, unsigned
vcpu_size, unsigned vcpu_align,
        int r;
        int cpu;

+       printk(KERN_INFO "Hey, KVM is initializing.\n");
+
        r = kvm_arch_init(opaque);
        if (r)
                goto out_fail;
--
1.7.1
```

9.3.4 检查patch

在代码风格中介绍了KVM内核和QEMU的代码规范，而且开源社区对代码规范的执行也比较严格，如果发送的patch不符合代码风格，维护者是不会接受的，还会让别人觉得你太不专业而鄙视你。所以，在将patch正式发送出去之前，非常有必要进行检查，至少用它们项目源代码仓库提供的自动检查脚本进行patch检查。使用脚本自动检查patch可以发现大多数的代码风格的问题，对于脚本检查发现的问题（包括

错误和警告），原则上都应该全部解决掉（尽管偶尔也有可能遇到实际并不不需要改正的警告信息）。

Linux内核与QEMU都分别提供了检查patch的脚本，它们的位置分别如下：

```
[root@jay-linux kvm.git]# ls scripts/checkpatch.pl
scripts/checkpatch.pl

[root@jay-linux qemu-kvm.git]# ls scripts/checkpatch.pl
scripts/checkpatch.pl
```

检查9.3.3节生成的patch，示例如下：

```
[root@jay-linux kvm.git]# scripts/checkpatch.pl
0001-just-a-demo.patch
WARNING: Prefer netdev_info(netdev, ... then dev_info(dev, ...
then pr_info(...  to printk(KERN_INFO ...
#18: FILE: virt/kvm/kvm_main.c:3108:
+       printk(KERN_INFO "Hey, KVM is initializing.\n");

ERROR: Missing Signed-off-by: line(s)

total: 1 errors, 1 warnings, 8 lines checked

0001-just-a-demo.patch has style problems, please review.

If any of these errors are false positives, please report
them to the maintainer, see CHECKPATCH in MAINTAINERS.
```

发现了一个错误和一个警告，错误是缺少"Signed-off-by:"这样的行，警告是printk(KERN_INFO...)这样的写法是不推荐的（最好用pr_info()函数来代替）。所以需要在编辑patch，在其中添

加"Signed-off-by:Jay＜smile665@gmail.com＞"这样的作者信息行，当然有多个作者可以用多行"Signed-off-by:"。另外，有时还可以根据需要添加其他的多行信息，如下：

❑"Reviewed-by:"当有人做过审查代码时；（一般是社区中资深人士）。

❑"Acked-by:"当有人表示响应和同意时；（一般是社区中资深人士）。

❑"Reported-by:"当某人报告了一个问题，本patch就修复那个问题时。

❑"Tested-by:"当某人测试了本patch时。

另外，可以在检查脚本中加--no-signoff参数来忽略对"Signed-off-by:"的检查，示例命令为：

```
scripts/checkpatch.pl --no-signoff my.patch
```

当然，如果是用"git format-patch"命令来生成patch的，则可以在生成patch时就添加-s或--signoff参数，以便在生成patch文件时就添加"Signed-off-by:"的信息行。

对于那个警告，使用pr_info()来替换printk(KERN_INFO...)函数即可。最后，用"git format-patch-s origin"命令生成0001-just-a-demo.patch，示例如下：

```
From 2c2118137eaa86bdce3c85016819dff336ca61f7 Mon Sep 17 00:00:00 2001
From: Jay <smile665@gmail.com>
Date: Sun, 26 May 2013 22:19:19 +0800
Subject: [PATCH] just a demo


Signed-off-by: Jay<smile665@gmail.com>
---
 virt/kvm/kvm_main.c |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

diff --git a/virt/kvm/kvm_main.c b/virt/kvm/kvm_main.c
index 302681c..e0bbfe6 100644
--- a/virt/kvm/kvm_main.c
+++ b/virt/kvm/kvm_main.c
@@ -3105,6 +3105,8 @@ int kvm_init(void *opaque, unsigned
vcpu_size, unsigned vcpu_align,
        int r;
        int cpu;

+       pr_info("Hey, KVM is initializing.\n");
+
        r = kvm_arch_init(opaque);
        if (r)
                goto out_fail;
--
1.7.1
```

在修正错误和警告后，用checkpatch.pl脚本重新检查生成的patch，命令如下：

```
[root@jay-linux kvm.git]# scripts/checkpatch.pl
0001-just-a-demo.patch
total: 0 errors, 0 warnings, 8 lines checked

0001-just-a-demo.patch has no obvious style problems and is ready
for submission.
```

可见，本次检查没有发现任何错误和警告，没有明显的代码风格问题，可以向开源社区提交这个patch了。

9.3.5 提交patch

准备好patch了后，最后要做的事情当然是提交patch了。KVM和QEMU的patch提交都是通过发送到邮件列表来实现的，KVM开发者邮件列表是kvm@vger.kernel.org，QEMU开发者邮件列表是qemu-devel@nongnu.org。QEMU代码中针对KVM相关修改的patch，需要发送到KVM邮件列表，并且抄送QEMU邮件列表。

除了邮件列表之外，一般收件人或抄送人中还包含该项目或子模块的维护者，以便让维护者将patch添加到upstream中。如何才能找到patch相关的维护者呢？首先，可以根据Linux内核和QEMU项目源代码中的MAINTAINERS文件查看维护者的信息以及他们负责的模块，然后根据patch中的改动及其影响找到影响的维护者。其次，Linux内核和QEMU代码仓库中都提供了一个根据patch查找到维护者的脚本，其位置都在源代码的"scripts/get_maintainer.pl"中。根据patch获得维护者信息的脚本的示例如下：

```
[root@jay-linux kvm.git]# scripts/get_maintainer.pl 0001-just-a-demo.patch
Gleb Natapov <gleb@redhat.com> (supporter:KERNEL VIRTUAL MA...)
Paolo Bonzini <pbonzini@redhat.com> (supporter:KERNEL VIRTUAL MA...)
kvm@vger.kernel.org (open list:KERNEL VIRTUAL MA...)

[root@jay-linux qemu-kvm.git]# scripts/get_maintainer.pl 0001-hello.patch
Anthony Liguori <aliguori@us.ibm.com> (supporter:Main loop)
```

一般来说，发送patch邮件时需要将维护者放在“抄送人”这一栏。不过目前没有非常严格地将维护者放在抄送人一栏：有的人发patch时，将邮件列表作为收件人，将维护者作为抄送人；有的人将维护者作为收件人，将KVM或QEMU邮件列表作为抄送人；也有的人将维护者和邮件列表都作为收件人。另外，也可以将任何与这个patch相关的人员（如部门同事、经理、测试人员）加到你发送patch的收件人或抄送人列表中。

在向QEMU/KVM的邮件组发送patch时，有不少朋友在初次使用Linux开发相关邮件列表时都可能会犯一些邮件格式相关的小错误，笔者根据经验总结了如下几个注意点：

1）邮件内容请使用纯文本格式，不要使用HTML格式修饰得很复杂和漂亮（因为一些Linux下的开发者就是用纯文本模式的邮件收发工具来处理邮件）。发往KVM相关邮件列表的邮件都应该使用英文来作为相互交流的语言，因为开发者来自世界各地（不要使用中文等非英语语言，否则因不专业而受到批评）。

2）将patch的内容直接粘贴在邮件正文中（尽量不要使用附件，除非你的邮件客户端不方便编辑纯文本的邮件正文），因为一些维护者是直接使用邮件来生成patch的（这样就不需要一个复制和粘贴patch内容的过程）。

3）使用"[PATCH]"字符作为主题的开头来表明是patch，需要维护者关注。一般来说，还会在主题的开始部分表明该patch是属于QEMU/KVM中的哪个模块。

4）如果实现某一个功能的patch的代码量比较大，尽量将一个大patch拆成多个相对独立的小patch。一个大的patch信息量较大，如果逻辑不是很清晰，则很容易引入bug，别人也不是很容易看得懂，维护者当然也不会很快同意将此patch加入到upstream中，另外将大patch拆分的过程也可以让作者自己重新理一下思路，从而减少一些错误。将一个相关功能拆分成m个patch，发送其中第n个patch时，一般邮件主题标注为"[PATCH n/m]"表示当前patch在这一系列patch中的位置，一般还会首先发一个[PATCH 0/m]作为第一个patch，这个数字编号0的patch通常用于书写本系列patch的概况信息。

5）根据社区的意见修改patch后，发送后续版本版本时，需要加上当前patch的版本号，比如，通过主题中类似"[PATCH v6 05/12]"这样关键字。特别是当patch修改的是比较核心的功能或添加一个较为重要的特性时，社区中可能有不少人会和你讨论给出对patch的意见，这

就可能需要根据一些意见来修改自己的patch，然后发出更新后的版本。笔者就在KVM邮件列表中见到过patch发送了超过10个版本才最终被维护者接受，所以开发者不能一下就做出完美的patch，那需要有耐心去参与讨论，并发送较多的patch版本，直到社区中的维护者和其他大牛们都接受为止。

下面是一些patch邮件的主题和存档地址的范例，供大家参考。

1）一个普通的patch邮件。

主题：[PATCH]virtio_pci:fix macro exported in uapi

URL：http://lkml.org/lkml/2013/5/16/441

2）一系列patch中的一个patch的邮件。这里表示当前patch是第3版，本系列patch共有13个，这是13个patch中的第1个包含代码修改内容的patch。

主题：[PATCH v3 01/13]nEPT:Support LOAD_IA32_EFER entry/exit controls for L1

URL：http://www.spinics.net/lists/kvm/msg91009.html

3）一系列patch中的第1个patch邮件。一般来说，当patch个数较多时，一般用00/N做第一个概括性描述的patch，其中一般是关于各个

版本的改变历史和本次这一系列patch的概况（包括问题描述、解决方案和改动的代码汇总），0号patch中一般不包含实际的patch中的代码内容。

主题：[PATCH v7 00/11]KVM:MMU:fast zap all shadow pages

URL：http://www.spinics.net/lists/kvm/msg91763.html

4）RFC（a request forcomments）patch邮件，多用于提交一个不太成熟的patch，呼吁大家对该patch提出改进意见。

主题：[PATCH RFC]KVM:Fix race in apic->pending_events processing

URL：www.mail-archive.com/kvm@vger.kernel.org/msg91204.html

关于发送patch时的其他问题，可咨询有经验人士或者仔细阅读Linux内核代码仓库中的文档Documentation/SubmittingPatches。

一般来说，可以使用Outlook、Foxmail、Thunderbird等客户端，也可以使用在线Gmail等邮箱来发送patch和接受邮件列表的邮件。另外，如是对Git工具比较熟悉，还可以使用git-email安装包中的"git send-email"命令行工具来发送patch。

最后，在发送patch之后，就耐心地等待社区中开发者的回复。如果收到一些批评的意见，不要太受打击，一方面要检查自己是否的确可以将patch做得更完美，另一方面，如果不同意别人给的批评意见，你也可以直接与之进行技术讨论，必要时甚至可以请社区中一些“德高望重”的大牛们来评判。收到别人的回复始终比没有人理你要好，因为一般来说一个patch不会在没有任何人讨论或回复的情况下被加入upstream中的。偶尔也会在发送patch后几天都没有任何回复，这时你可以检查一下邮件格式、patch内容、收件人等方面是否有问题，确认自己没问题后可以发邮件提醒一下维护者或其他相关人员对你的patch给出评价。一般来说，当收到维护者发给你的带有"applied"字样的回复时，恭喜，你的patch可以顺利进入upstream了。你的patch可能进入到QEMU中，也可能进入KVM内核中，这样在下一个Linux内核发布版本中就很可能有你贡献的代码了。

9.4 提交KVM相关的bug

有句话这样说的，没有任何软件没有bug。当然，QEMU/KVM也不例外，在使用它们的过程中也可能遇到一些bug，有一些严重的bug可能会导致客户机甚至是宿主机系统崩溃，有一些比较轻微的bug可能会导致在特殊的（通常是老旧的）硬件平台上使用某个版本客户机的某个小功能不可用。KVM和QEMU作为开源软件，有着强大的开源社区的支持，在遇到bug时，就可以提交出去大家一起讨论，对于不会修复bug的新手，很可能遇到热心的开发者帮一起解决bug。本节将给大家介绍如何在开源社区中提交QEMU/KVM相关的bug和用git bisect命令来定位bug。

9.4.1 通过邮件列表提交bug

在目前的QEMU/KVM开源社区中，比较简单直接和可以很快得到反馈的提交bug的方式是使用开发者邮件列表来反映问题。邮件列表的详情可参考9.3.1节中的介绍。

对于KVM内核部分的Bug，要发送邮件到kvm@vger.kernel.org邮件列表。对于QEMU部分的bug，要发送邮件到qemu-devel@nongnu.org（与KVM相关的，也同时发送到KVM的邮件列表）。

如果认为可能是某个人引入了该bug或者知道某个人是这方面的专家，也可以将邮件列表的bug邮件抄送相关的人员，请求他们协助解决。

在向邮件列表提交bug时，注意将问题尽可能描述清楚，这样做至少有两个原因：一是清晰的描述可以让其他开发者清楚所遇到的问题，从而快速地帮你解决问题，二是详细的描述是请教的真诚和技术的专业性的表现。如果只有“can’t boot a KVM guest（不能启动一个KVM客户机）”这样内容而没有任何其他详细描述的内容，会被认为很不专业也没有诚意，也许很多人就不理会这个邮件。

下面这几个方面是笔者根据经验总结的如何清晰地描述一个KVM相关bug。

（1）测试环境

通常包括遇到bug时的硬件环境和软件环境。硬件环境主要包括：CPU架构（Intel的x86/x86-64还是ARM等，有时还需要提供具体的型号）、磁盘类型（如IDE、SATA、SAS、SSD等）、网卡型号（如Intel的E1000E类型的网卡或82599等10G网卡）、显卡类型（Intel的核心集成显卡还是Nvidia的独立显卡）等，当然根据实际遇到的问题只需要选择性地提供其中的一部分硬件信息，不过一般最好把CPU架构提一下（如果别人不能从上文中看出来）。软件环境，首先包括KVM内核的体系架构（x86、x86-64、ARM、PowerPC等，当然和硬件CPU架构有关

联），然后是KVM内核的版本、QEMU的版本、客户机操作系统的版本（如RHEL 6.4、Ubuntu 13.04、Windows 7等，它们是32位系统还是64位系统），最后是其他可能相关的一些特定软件（如客户机中因运行某个软件而发现了虚拟机的bug）。如果是使用kvm.git和qemu.git代码仓库编译的KVM和QEMU时遇到的bug，需要提供准确的在Git工具的"git log"命令显示的commit ID。

（2）现象描述

当然，反映一个问题，需要将问题描述清楚。一般需要说明做了什么操作，得到什么结果，有时也加上所期望的结果是什么样的。另外，尽可能多做一点对比的实验，以便其他开发者可以方便快速定位问题，比如，使用某个i版本QEMU遇到了问题而j版本的QEMU是正常的，使用某个网卡遇到了问题但另一个网卡却正常工作，Windows客户机不能启动但Linux客户机能启动，等等。最后，如果是使用kvm.git和qemu.git时遇到的问题，而且刚好有一个可以正常工作和一个不正常工作的版本，可以考虑多做一些实验通过二分法找到引入bug的点（详见9.4.3节）。

（3）详细日志

对于很多bug，仅仅通过前面的现象描述很难将问题讲清楚，因此一般需要提供尽可能详细的日志信息来辅助描述bug。日志信息主要包

括：宿主机KVM内核的信息、QEMU命令行的错误、libvirt日志（如果使用的是libvirt）、客户机内核的信息、客户机中某个引发bug的应用软件的日志等。根据实际bug的情况不同，需要提供的日志信息差别也很大。如果bug导致宿主机或客户机内核都在打印一些错误了，那么它们的内核的日志是非常重要的，在Linux上可以通过dmesg命令来获取。如果bug导致宿主机或客户机系统直接崩溃，那么可以将它们的串口重定向到另外的地方以便获取其崩溃之时打印的函数调用、堆栈、寄存器的信息。如果某个PCI设备不可用，那么可能需要内核信息中关于PCI设备的部分，也需要这个PCI设备的一些详情，比如可以通过"lspci-vvv$BDF"命令来获取PCI设备的一些信息。总之，对于bug分析，可能有用的日志信息都应该尽可能被提交到bug中去，即使有一些信息是没用到的，社区开发者忽略便是。

（4）重现步骤

俗话说，如果能稳定重现一个bug，那么这个bug就算修复了一半。KVM虚拟化中的一些系统性的bug更是这样，一旦某个大牛能够稳定重现你的bug了，那么bug被修复也就指日可待了。所以，重现bug的操作步骤也是提交的bug中至关重要的信息。重现步骤要写得条理清晰且信息饱满，一般可分为1、2、3这样的步骤来书写，而且其中每个操作步骤涉及命令的也将完整的命令写上去（特别是QEMU命令行中启动客户机的命令是至关重要的）。可能有一些开发者会回复你说“不能

重现你报的bug”，这时你需要耐心地将你的测试环境和重现步骤更详细地告诉他们，让别人也能重现bug，从而使问题得到快速解决。

提交bug的内容看起来还是比较复杂，注意事项也比较多，不过也不要被复杂吓退了，一个简单的方法就是：订阅邮件列表，关注里面关于提交bug相关的邮件，自己提交bug时可以作为参考。

9.4.2 使用bug管理系统提交bug

有时，在邮件列表中提交的一个bug并不能在几天时间内一下解决掉，过了几个月基本上就没人会记得你报过的bug，更别提修复bug了，所以除了通过邮件提交bug之外，还可以通过KVM和QEMU各自的bug跟踪系统来提交bug。在bug跟踪系统中提交的bug更加便于长期的管理和今后的bug数据分析。KVM作为Linux内核的一部分，是使用Linux内核社区的Bugzilla系统来跟踪bug的。Bugzilla是一个非常流行的开源的基于Web的bug管理系统，包括Linux内核、Xen、GNOME、Mozilla、Apache、Red Hat、Novell等很多著名的项目或组织都在使用它。QEMU使用的bug跟踪系统是在Launchpad网站上的一个bug管理系统。下面分别介绍KVM和QEMU的bug管理系统及其使用的注意事项。

KVM的bug管理系统在https://bugzilla.kernel.org/，在提交bug时需要选择“虚拟化”（Virtualization），然后在bug的具体描述中选择KVM这个组件为对象来提交bug。在搜索KVM内核相关的bug时，也

需要在搜索条件中选择"Virtualization"作为产品，选择"KVM"作为组件。

QEMU的bug管理系统在https://bugs.launchpad.net/qemu，在这个网页可以查看或搜索到目前哪些bug正在修复或讨论中，在网页的右边有"Report a bug"（报告一个bug）的链接供提交bug之用。尽管QEMU不是与KVM一样使用Bugzilla系统，不过Launchpad上的QEMU bug系统也是非常易于使用的。

在bug跟踪系统中提交的bug的基本要素与前一节中提到的清晰描述一个bug的注意事项是完全一样的，所以这就不重复介绍了。笔者对KVM和QEMU两个项目都提交过不少的bug，下面分别提供两个bug的网络地址，供大家提交bug时参考。

❑KVM bug例子：https://bugzilla.kernel.org/show_bug.cgi?id=43328和https://bugzilla.kernel.org/show_bug.cgi?id=45931。

❑QEMU bug例子：https://bugs.launchpad.net/qemu/+bug/1013467和https://bugs.launchpad.net/qemu/+bug/1096814。

9.4.3 使用二分法定位bug

通常，一旦能够定位到bug是在某个准确的点引入的，那么修复这个bug一般都会比较容易了。如果定位到在Linux内核3.10.2版本中存在某个bug，而在3.10.1版本中不存在该bug，要修复这个bug就不难了：要么revert掉这两个版本之间的某些patch，要么精确地找到哪里的代码错误并进行修正。

然后并不是总能够很容易找到一个可以正常工作和一个存在bug的代码版本。一般来说，在遇到bug时，当然很清楚当前这个版本是有bug的版本。那么如何找到之前可以正常工作的版本呢？也没有特别好的方法，可以考虑找到几个月前的版本来试试，如果它依然是有这个bug的，那么只能往前再找（比如找一两年前的版本）。在一般情况下，除非遇到的bug确实很偏门或者非常新的特性（从来就没正常工作过），否则总可以找到一个版本会出现某个bug，也能找到老的可以没有某bug的版本。

有时在找到的可以工作和有bug的版本之间发布的时间相隔较长、差异较大时，可以使用二分法来进一步定位引入bug的具体版本。假设已知3.10.8版本是有某个bug的版本而3.10.0版本是可以正常工作的，使用二分法来查找的示例方法为：先测试它们的中间版本3.10.4版本，如果可以正常工作的，则在3.10.4和3.10.8之间再次做二分法，应该继续测试3.10.6版本，如果3.10.6版本是有bug的，则查找3.10.4

和3.10.6之间的版本（自然就是3.10.5版本了），如果3.10.5是正常工作，就可以确认bug是在3.10.5和3.10.6版本之间的patch引入的。

对KVM和QEMU开源社区来说，它们的源代码仓库都是使用Git工具来做代码控制管理的，而Git工具提供了便捷的命令工具"git bisect"来支持通过二分法来查找引入bug的代码修改。当使用kvm.git和qemu.git这样的Git管理的代码仓库编译遇到问题时，使用"git bisect"工具定位bug是非常方便的。下面对其进行简单的介绍。

"git bisect"的基本原理是：开始执行二分法查找后，需要先告诉Git一个有bug的版本（标记为bad）和一个正常工作的版本（标记为good），然后Git会自动切换当前代码库到二者的一个中间版本，接着经过测试后告诉Git当前版本是正常的还是有bug的，Git根据它得到的信息再次切换到相应的一个中间版本，如此循环，直到Git发现一个版本是有bug的而它的前一个版本是没有bug的，这时Git就会报告出找到了第一个引入bug的版本。

下面的命令行示例是使用"git bisect"工具进行二分法查找引入某个bug的第一个版本的过程。

```
# 开始 Git 的二分查找
[root@jay-linux kvm.git]# git bisect start

# 标记当前版本为有 bug 的
[root@jay-linux kvm.git]# git bisect bad

# 标记以前的 8b19d450ad18 版本为正常工作的版本
[root@jay-linux kvm.git]# git bisect good 8b19d450ad18
Bisecting: 316 revisions left to test after this (roughly 8 steps)
[ff9129b06cfb05cb5920f1151c75506afe1586fe] Merge tag

 'devicetree-for-linus' of git://git.secretlab.ca/git/linux
 #Git 已经将当前版本切换到了中间的一个版本

 # 经过编译、测试后，告诉 Git 当前版本是一个正常工作的版本
 [root@jay-linux kvm.git]# git bisect good
 Bisecting: 153 revisions left to test after this (roughly 7 steps)
 [cbfd2cd7195cf4500d428a04c79509445aa3924e] Merge tag
 'mfd-fixes-3.10-1' of git://git.kernel.org/pub/scm/linux/kernel/git/sameo/mfd-fixes
 # 经过编译、测试后，告诉 Git 当前版本是有 bug 的
 [root@jay-linux kvm.git]# git bisect bad
 Bisecting: 82 revisions left to test after this (roughly 6 steps)
 [622f223488517f2b0a5a5e518b2a6c950cf0a2ee] Merge tag 'hwmon-for-linus'
 of git://git.kernel.org/pub/scm/linux/kernel/git/groeck/linux-staging
 # ……（经过多次的编译、测试，以及和 Git 交互后）
 # 在最后一次标记为 bad 时，Git 就查找到了第一次引入 bug 的 commit
 [root@jay-linux kvm.git]# git bisect bad
 29589f06d2430efb76c227b0117029ebd3101eec is the first bad commit
 commit 29589f06d2430efb76c227b0117029ebd3101eec
 Author: XXX<XX@xxx.com>
 Date:   Sun May 12 15:19:46 2013 +0200

 drivers/ata: don't check resource with devm_ioremap_resource

     Signed-off-by: XXX<XX@xxx.com>

 :040000 040000 2eca78d5482dea95549994ee56da9732d96752c0
 b1d60c470f6fc92bc866ff73ae9807388f99c081 M      drivers
```

在使用"git bisect"进行查找过程中，可能遇到切换到中间某个版本不能编译或有其他bug存在而不能对查找中的bug进行验证，那么可以使用"git bisect skip"命令跳过当前版本。"git bisect log"命

令可以显示出本次使用二分法的具体过程，包括哪些commit被标记为good，哪些被标记为bad。对于已知某个目录的代码引入了这个bug，在执行二分法查找时，可以指定仅对某个目录来做，如"git bisect start--arch/x86/kvm/"。

每次Git切换KVM的当前工作代码目录到某个版本后，都需要经过编译和测试，这时根据具体情况需要重启宿主机系统或者重新加载kvm、kvm_intel等模块，而对QEMU进行二分法查找时，由于QEMU是用户态工具，故不需要重启宿主机系统而只需要用新编译的QEMU工具重新创建客户机即可验证bug。

另外，有的时候，比如要向后移植某个patch来解决某个bug时，我们可能也需要找到修复这个bug的精确版本。这时，如果完全按照寻找引入bug的第一个版本的思路和操作来做，是不能达到目的的，因为"git bisect"命令设计的初衷是找到引入bug的具体版本，当标记为good的版本比标记为bad的版本要新时，Git就会报错。在这种情况下，可以稍微反向操作一下：将有bug的版本（较老的）标记为good，而将没有bug的版本（较新的）标记为bad，然后使用"git bisect"工具向前面示例那样操作即可。当Git告诉你第一次引入bug的版本时，那个版本即是你要寻找的修复某个bug的具体版本。

9.5 本章小结

本章首先介绍了Linux、KVM、QEMU等开源社区的基本情况，然后对KVM、QEMU和KVM测试的一些代码结构进行了简单的梳理，接着介绍了如何向KVM开源社区贡献代码，最后介绍了如何提交和定位KVM虚拟化的bug。本章旨在介绍KVM开源社区，进而鼓励大家参与到KVM社区中去贡献代码、提交bug，或者仅仅是学习一下。开源软件和开源社区是非常博大精深的，如果大家能够以开源精神参与到社区之中，一定会发现你学习和了解到的知识远比想象的要多，同时也对使用开源软件进行集成或开发有较大的正面促进作用。

9.6 本章注释和参考阅读

1.本章注释

[1]upstream，中文有时也被翻译为“上游”，是指在一些开源软件的开发过程中最主干的且一直向前发展的代码树。upstream中包含了最新的代码，一些新增的功能特性和修复bug的代码一般都先进入到upstream中，然后才会到某一个具体的发型版本中。对于Linux内核来说，Linux upstream就是指最新的Linux内核代码树，而一些Linux发行版（如RHEL、Fedora、Ubuntu等）的内核都是基于upstream中的某个版本再加上一部分patch来制作的。

[2]RC（Release Candidate）与Alpha、Beta等版本类似，RC也是一种正式产品发布前的一种测试版本。RC是发布前的候选版本，比Alpha/Beta更加成熟，一般来说RC版本是前面已经经过测试并修复了大部分bug的，一般比较稳定、接近正式发布的产品。

[3]请参考linux-kvm.com网站中关于qemu-kvm合并到QEMU 1.3的新闻（http://www.linux-kvm.com/content/qemu-13-released-qemu-kvm-merge-qemu-complete）。

[4]在Xen中使用QEMU upstream的信息，请参考xen.org的官方wiki上的文章（http://wiki.xen.org/wiki/QEMU_Upstream）。

[5]IRC（Internet Relay Chat，因特网中继聊天）是一种通过网络的即时聊天方式。其主要用于群体聊天，但同样也可以用于个人对个人的聊天。IRC是一个分布式的客户端/服务器结构。通过连接到一个IRC服务器，我们可以访问这个服务器及它所连接的其他服务器上的频道。要使用IRC，必须先登录到一个IRC服务器上，如一个常见的服务器为irc.freenode.net。

[6]Google对开源项目代码风格的指导文档（https://code.google.com/p/google-styleguide/）。[7]Linux内核代码风格的中英文版本见https://www.kernel.org/doc/Documentation/zh_CN/CodingStyle和https://www.kernel.org/doc/Documentation/CodingStyle。

[8]Brian Kernighan和Dennis Ritchie共同编写C语言中经典的书籍《C programming language》（也称为K＆R C），Linux内核中关于大括号的使用风格就是来自于K＆R C中的规范。他们二人是UNIX操作系统的最主要的开发者中的两位，Brian Kernighan是AWK编程语言的联合作者，Dennis Ritchie发明了C语言（是“C语言之父”）。

[9]匈牙利命名法是Microsoft公司推荐的命名方法，可以参考网页http://en.wikipedia.org/wiki/Hungarian_notation。

2.参考阅读

（1）Wikipedia中关于Linux和Linux内核的词条（http://en.wikipedia.org/wiki/Linux和http://en.wikipedia.org/wiki/Linux_kernel）。

（2）Linux内核档案网站（https://kernel.org/）。

（3）Linux基金会（http://www.linuxfoundation.org/）。

（4）KVM Autotest的文档（https://github.com/autotest/virt-test/wiki/）。