HZ BOOKS
华章科技

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

實戰



KVM: Principles and Practices

KVM虚拟化技术
实战与原理解析

任永杰 单海涛 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

KVM虚拟化技术：实战与原理解析 / 任永杰，单海涛著. —北京：机械工业出版社，2013.10

ISBN 978-7-111-43900-4

I. K… II. ①任… ②单… III. 虚拟处理机 IV. TP338

中国版本图书馆CIP数据核字（2013）第209229号

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

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国内 KVM 技术的先驱者撰写，权威性毋庸置疑。在具体内容上，本书不仅系统介绍了 KVM 虚拟机的功能、特性和使用方法，而且还深入地剖析了 KVM 虚拟机的核心技术和工作原理，对 KVM 做了全面而透彻的讲解。

全书一共 9 章：第 1 章介绍了云计算和虚拟化的概念，并对 KVM 等几种流行的虚拟化技术做了比较；第 2 章介绍了 KVM 的基本架构、QEMU 的作用以及 Intel 的硬件虚拟化技术；第 3 章详细讲解了 KVM 依赖的硬件环境配置，编译和安装 KVM 和 QEMU 的步骤与方法，以及 KVM 客户机的启动；第 4 章深入讲解了 KVM 的基础功能，包括虚拟的 CPU、内存、存储、网络、图形显示等的配置和基本原理，以及 CPU 和内存的过载使用；第 5 章讲解了 KVM 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；第 6 章介绍了管理 KVM 虚拟化的上层软件；第 7 章介绍了 RHEL 等所有流行的 Linux 发行版中的 KVM 虚拟化功能的使用；第 8 章首先介绍了虚拟化性能测试，然后详细介绍了对 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、工具和步骤；第 9 章介绍了 Linux、KVM、QEMU 等开源社区的情况，KVM、QEMU 和 KVM 单元测试代码的基本结构，以及如何向 QEMU/KVM 开源社区贡献自己的代码和如何提交 KVM 相关的 bug。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姜 影

北京市荣盛彩色印刷有限公司印刷

2013年10月第1版第1次印刷

186mm × 240mm · 26.5印张

标准书号：ISBN 978-7-111-43900-4

定 价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

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

目　录

前言

第 1 章　虚拟化与云计算 …… 1

1.1　云计算概念 …… 1
1.1.1　发展历史和现状 …… 1
1.1.2　概念 …… 2
1.1.3　云计算模式 …… 3
1.2　云计算技术 …… 4
1.2.1　Map/Reduce …… 4
1.2.2　资源管理平台 …… 5
1.2.3　虚拟化 …… 5
1.3　虚拟化技术 …… 5
1.3.1　软件虚拟化和硬件虚拟化 …… 6
1.3.2　准虚拟化与全虚拟化 …… 8
1.4　KVM简介 …… 9
1.4.1　KVM的历史 …… 9
1.4.2　KVM功能概览 …… 9
1.4.3　KVM的前景 …… 11
1.5　Xen简介 …… 11
1.5.1　Xen的历史 …… 11
1.5.2　Xen功能概览 …… 12
1.5.3　Xen的前景 …… 13
1.6　其他虚拟化方案简介 …… 13
1.6.1　VMware …… 14
1.6.2　VirtualBox …… 14
1.6.3　Hyper-V …… 15

1.7 本章小结 …… 15
1.8 本章注释 …… 15

第 2 章 KVM 原理简介 …… 17

2.1 Linux操作系统简介 …… 17
2.2 虚拟化模型 …… 18
2.3 KVM架构 …… 19
2.4 KVM模块 …… 21
2.5 QEMU设备模型 …… 22
2.6 Intel虚拟化技术 …… 22
2.7 本章小结 …… 23

第 3 章 构建 KVM 环境 …… 24

3.1 硬件系统的配置 …… 24
3.2 安装宿主机Linux系统 …… 26
3.3 编译和安装KVM …… 28
3.3.1 下载KVM源代码 …… 28
3.3.2 配置KVM …… 30
3.3.3 编译KVM …… 33
3.3.4 安装KVM …… 35
3.4 编译和安装qemu-kvm …… 37
3.4.1 下载qemu-kvm源代码 …… 37
3.4.2 配置和编译qemu-kvm …… 38
3.4.3 安装qemu-kvm …… 39
3.5 安装客户机 …… 40
3.6 启动第一个KVM客户机 …… 42
3.7 本章小结 …… 43
3.8 本章注释和参考阅读 …… 44

第 4 章 KVM 核心基础功能 …… 45

4.1 硬件平台和软件版本说明 …… 45
4.2 CPU配置 …… 48

4.2.1 vCPU的概念 …… 48
4.2.2 SMP的支持 …… 49
4.2.3 CPU过载使用 …… 54
4.2.4 CPU模型 …… 55
4.2.5 进程的处理器亲和性和vCPU的绑定 …… 57
4.3 内存配置 …… 61
4.3.1 内存设置基本参数 …… 61
4.3.2 EPT和VPID简介 …… 64
4.3.3 大页（Huge Page） …… 66
4.3.4 内存过载使用 …… 68
4.4 存储配置 …… 70
4.4.1 存储配置和启动顺序 …… 70
4.4.2 qemu-img命令 …… 75
4.4.3 QEMU支持的镜像文件格式 …… 78
4.4.4 客户机存储方式 …… 80
4.5 网络配置 …… 82
4.5.1 QEMU支持的网络模式 …… 83
4.5.2 使用网桥模式 …… 85
4.5.3 使用NAT模式 …… 90
4.5.4 QEMU内部的用户模式网络 …… 98
4.5.5 其他网络选项 …… 102
4.6 图形显示 …… 103
4.6.1 SDL的使用 …… 103
4.6.2 VNC的使用 …… 105
4.6.3 VNC显示中的鼠标偏移 …… 112
4.6.4 非图形模式 …… 113
4.6.5 显示相关的其他选项 …… 115
4.7 本章小结 …… 116
4.8 本章注释和参考阅读 …… 116

第5章 KVM高级功能详解 …… 120

5.1 半虚拟化驱动 …… 120

5.1.1 virtio概述 …… 120
5.1.2 安装virtio驱动 …… 122
5.1.3 使用virtio_balloon …… 128
5.1.4 使用virtio_net …… 133
5.1.5 使用virtio_blk …… 137
5.1.6 kvm_clock配置 …… 138
5.2 设备直接分配（VT-d） …… 140
5.2.1 VT-d概述 …… 140
5.2.2 VT-d环境配置 …… 141
5.2.3 VT-d操作示例 …… 150
5.2.4 SR-IOV技术 …… 160
5.3 热插拔 …… 170
5.3.1 PCI设备热插拔 …… 170
5.3.2 PCI设备热插拔示例 …… 171
5.3.3 CPU和内存的热插拔 …… 176
5.4 动态迁移 …… 177
5.4.1 动态迁移的概念 …… 177
5.4.2 动态迁移的效率和应用场景 …… 178
5.4.3 KVM动态迁移原理和实践 …… 180
5.4.4 VT-d/SR-IOV的动态迁移 …… 187
5.5 嵌套虚拟化 …… 188
5.5.1 嵌套虚拟化的基本概念 …… 188
5.5.2 KVM嵌套KVM …… 188
5.6 KSM技术 …… 191
5.6.1 KSM基本原理 …… 191
5.6.2 KSM操作实践 …… 193
5.7 KVM其他特性简介 …… 197
5.7.1 1GB大页 …… 197
5.7.2 透明大页 …… 200
5.7.3 AVX和XSAVE …… 203
5.7.4 AES新指令 …… 205
5.7.5 完全暴露宿主机CPU特性 …… 210
5.8 KVM安全 …… 212

5.8.1 SMEP …… 212
5.8.2 控制客户机的资源使用——cgroups …… 213
5.8.3 SELinux和sVirt …… 220
5.8.4 可信任启动——Tboot …… 228
5.8.5 其他安全策略 …… 237
5.9 QEMU监控器 …… 241
5.9.1 QEMU monitor的切换和配置 …… 241
5.9.2 常用命令介绍 …… 242
5.10 qemu-kvm命令行参数 …… 249
5.10.1 回顾已用过的参数 …… 249
5.10.2 其他常用参数 …… 253
5.11 迁移到KVM虚拟化环境 …… 261
5.11.1 virt-v2v工具介绍 …… 261
5.11.2 从Xen迁移到KVM …… 262
5.11.3 从VMware迁移到KVM …… 265
5.11.4 从VirtualBox迁移到KVM …… 266
5.11.5 从物理机迁移到KVM虚拟化环境（P2V） …… 266
5.12 本章小结 …… 267
5.13 注释和参考阅读 …… 268

第 6 章 KVM 管理工具 …… 272

6.1 libvirt …… 272
6.1.1 libvirt简介 …… 272
6.1.2 libvirt的编译、安装和配置 …… 275
6.1.3 libvirt和libvirtd的配置 …… 281
6.1.4 libvirt域的XML配置文件 …… 285
6.1.5 libvirt API简介 …… 297
6.1.6 建立到Hypervisor的连接 …… 299
6.1.7 libvirt API使用示例 …… 303
6.2 virsh …… 308
6.2.1 virsh简介 …… 308
6.2.2 virsh常用命令 …… 310

6.3 virt-manager ····· 316
6.3.1 virt-manager简介 ····· 316
6.3.2 virt-manager编译和安装 ····· 317
6.3.3 virt-manager使用 ····· 317
6.4 virt-viewer、virt-install和virt-top ····· 324
6.4.1 virt-viewer ····· 324
6.4.2 virt-install ····· 325
6.4.3 virt-top ····· 327
6.5 OpenStack云计算平台 ····· 327
6.5.1 OpenStack简介 ····· 327
6.5.2 使用DevStack安装和配置OpenStack开发环境 ····· 330
6.5.3 在OpenStack中使用KVM ····· 332
6.6 本章小结 ····· 334
6.7 本章注释和参考阅读 ····· 334

第7章 Linux发行版中的KVM ····· 336

7.1 RHEL和Fedora中的KVM ····· 336
7.1.1 Red Hat、RHEL、Fedora和CentOS简介 ····· 336
7.1.2 RHEL中的KVM ····· 337
7.1.3 Fedora中的KVM ····· 338
7.2 SLES和openSUSE中的KVM ····· 339
7.2.1 SLES中的KVM ····· 339
7.2.2 openSUSE中的KVM ····· 341
7.3 Ubuntu中的KVM ····· 341
7.4 本章小结 ····· 343
7.5 本章注释和参考阅读 ····· 344

第8章 KVM性能测试及参考数据 ····· 345

8.1 虚拟化性能测试简介 ····· 345
8.2 CPU性能测试 ····· 347
8.2.1 CPU性能测试工具 ····· 347
8.2.2 测试环境配置 ····· 349

8.2.3 性能测试方法 …… 350
8.2.4 性能测试数据 …… 352
8.3 内存性能测试 …… 354
8.3.1 内存性能测试工具 …… 355
8.3.2 测试环境配置 …… 356
8.3.3 性能测试方法 …… 356
8.3.4 性能测试数据 …… 357
8.4 网络性能测试 …… 360
8.4.1 网络性能测试工具 …… 360
8.4.2 测试环境配置 …… 361
8.4.3 性能测试方法 …… 362
8.4.4 性能测试数据 …… 364
8.5 磁盘I/O性能测试 …… 367
8.5.1 磁盘I/O性能测试工具 …… 367
8.5.2 测试环境配置 …… 368
8.5.3 性能测试方法 …… 369
8.5.4 性能测试数据 …… 370
8.6 本章小结 …… 373
8.7 本章注释和参考阅读 …… 373

第9章 参与KVM开源社区 …… 375

9.1 开源社区介绍 …… 375
9.1.1 Linux开源社区 …… 375
9.1.2 KVM开源社区 …… 377
9.1.3 QEMU开源社区 …… 377
9.1.4 其他开源社区 …… 378
9.2 代码结构简介 …… 379
9.2.1 KVM代码 …… 379
9.2.2 QEMU代码 …… 381
9.2.3 KVM单元测试代码 …… 383
9.2.4 KVM Autotest代码 …… 385
9.3 向开源社区贡献代码 …… 386

9.3.1 开发者邮件列表 …… 386
9.3.2 代码风格 …… 387
9.3.3 生成patch …… 391
9.3.4 检查patch …… 394
9.3.5 提交patch …… 396
9.4 提交KVM相关的bug …… 399
9.4.1 通过邮件列表提交bug …… 399
9.4.2 使用bug管理系统提交bug …… 401
9.4.3 使用二分法定位bug …… 401
9.5 本章小结 …… 404
9.6 本章注释和参考阅读 …… 404

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。

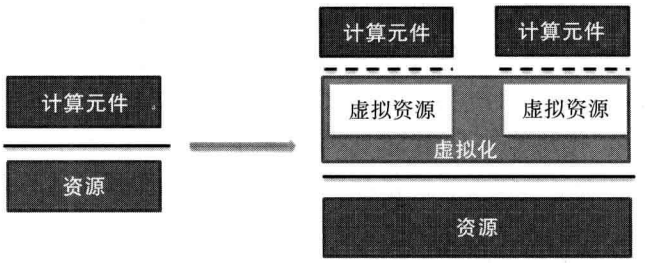


图 1-1 虚拟化

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。

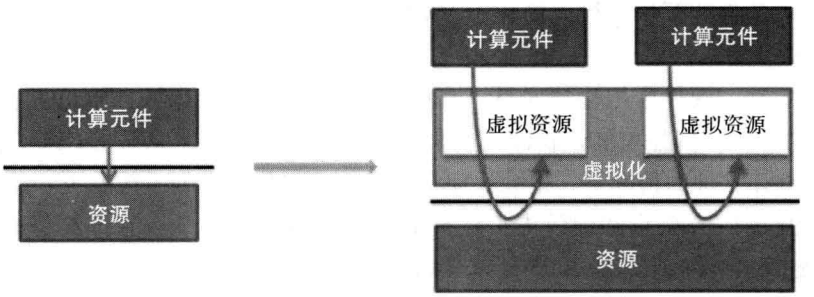


图 1-2 虚拟化的可实现性

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。

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。

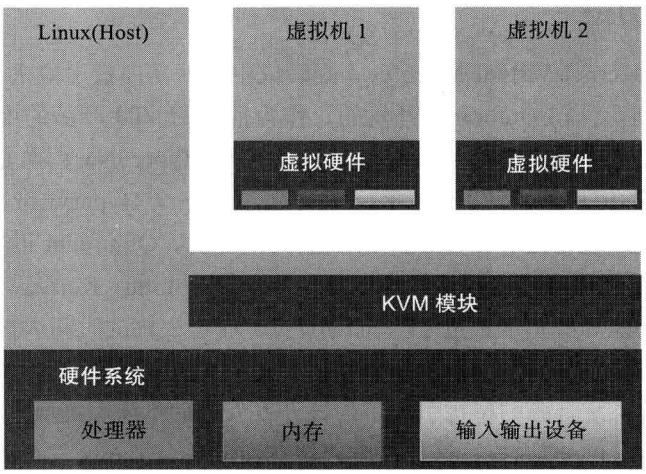


图 1-3 KVM 架构

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。

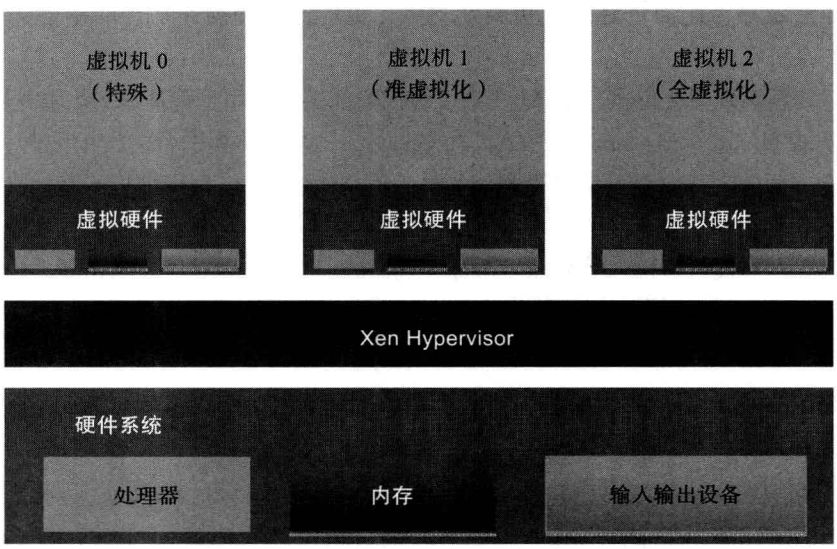


图 1-4 Xen 架构

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。

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。

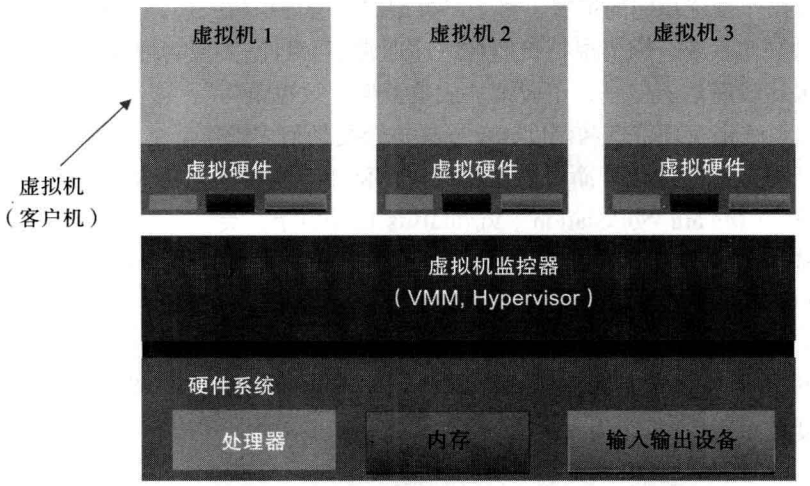


图 2-1 虚拟化模型

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。

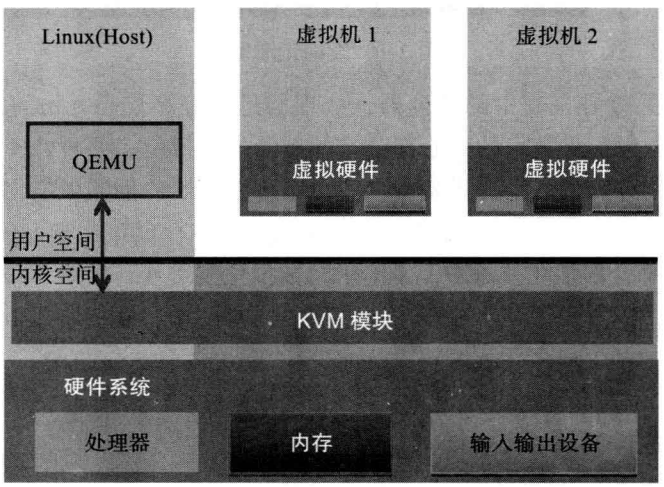


图 2-2 KVM 架构

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）。

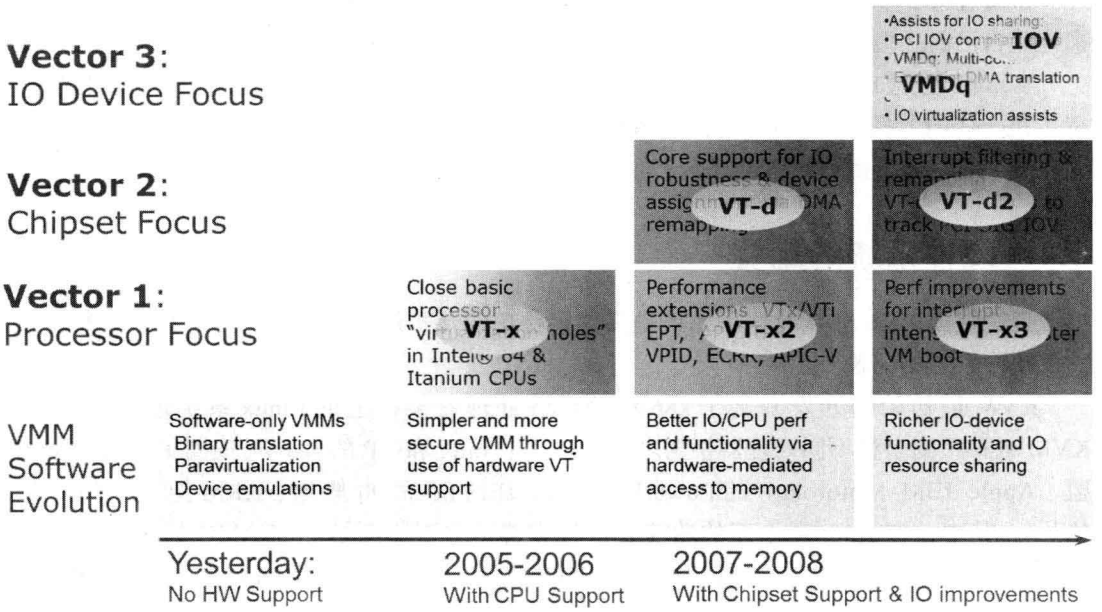


图 2-3 Intel 虚拟化技术进化蓝图

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。

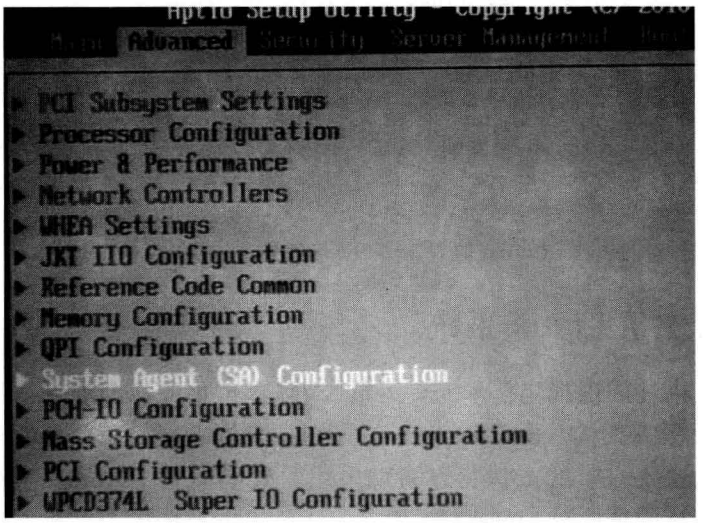


图 3-1 BIOS 中的 Advanced 选项

2）BIOS 中的 VT 和 VT-d 选项，如图 3-2 所示。

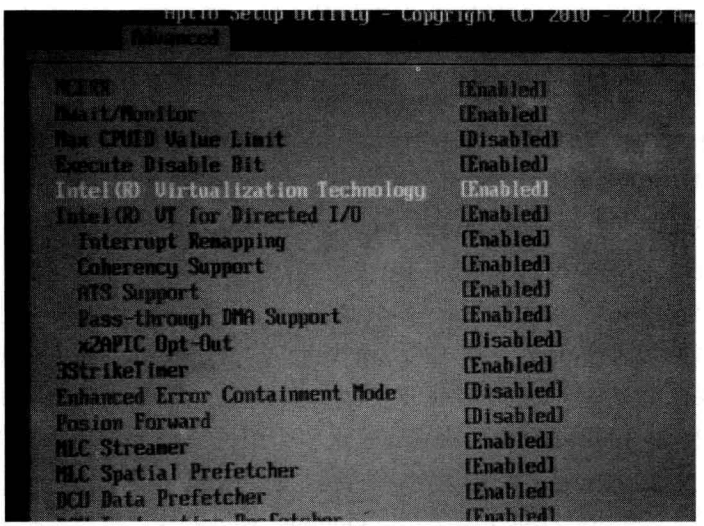


图 3-2 BIOS 中 Enabled 的 VT 与 VT-d 选项

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 pdpelgb 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。

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。

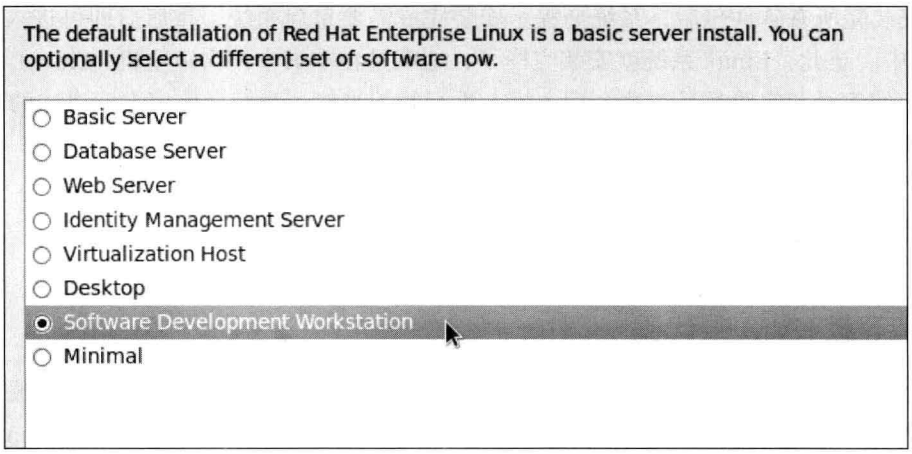


图 3-3 在 RHEL 6.3 安装中选择服务器类型

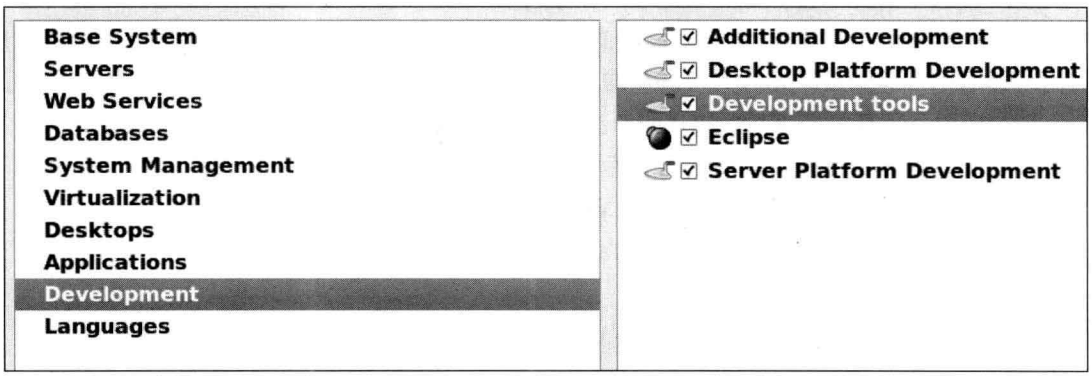


图 3-4 Software Development Workstation 默认选择的具体组件

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。

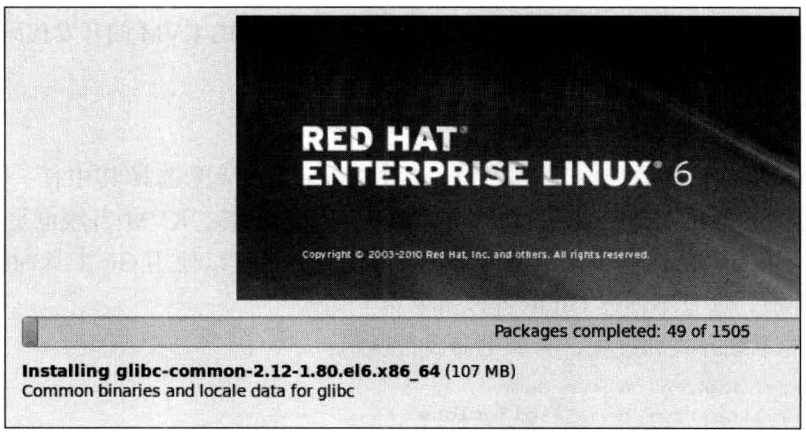


图 3-5 RHEL 6.3 安装过程快照

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。

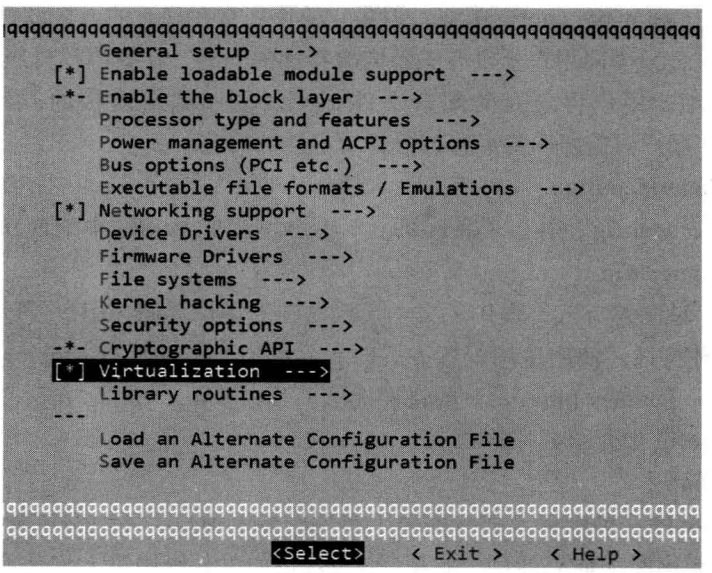


图 3-6 make menuconfig 命令的选择界面

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。

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
```

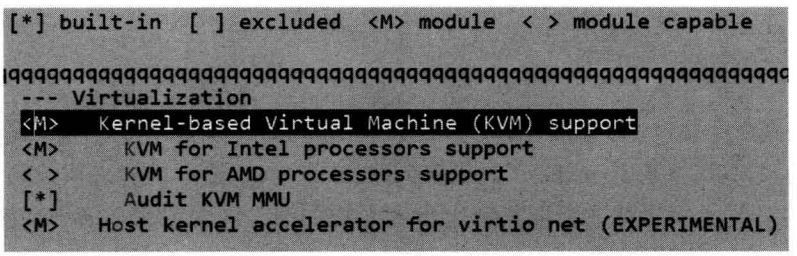


图 3-7 Virtualization 中的配置选项

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。

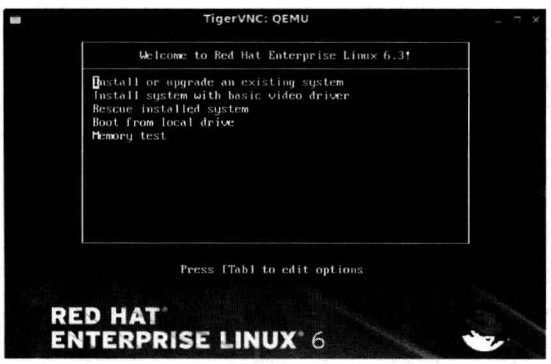


图 3-8 客户机安装的选择界面

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。

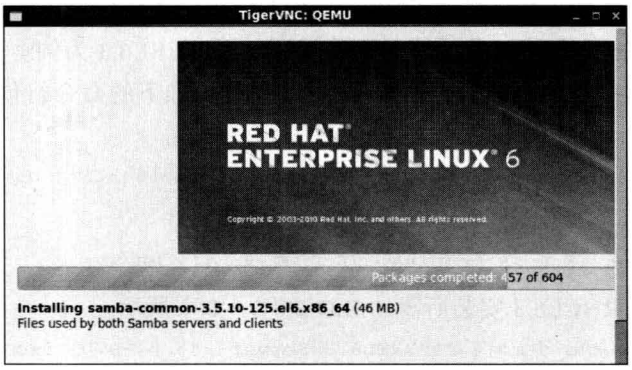


图 3-9 客户机安装过程的快照

在系统安装完成后，客户机中安装程序提示，如图 3-10 所示。

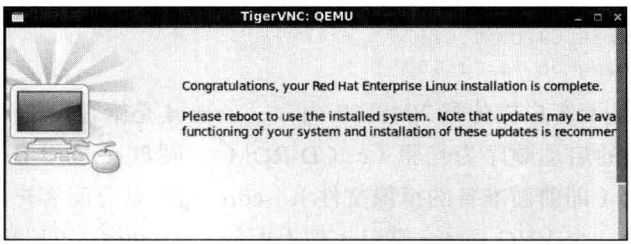


图 3-10 客户机安装完成后的提示信息

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。

客户机启动完成后的登录界面如图 3-12 所示。

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）。

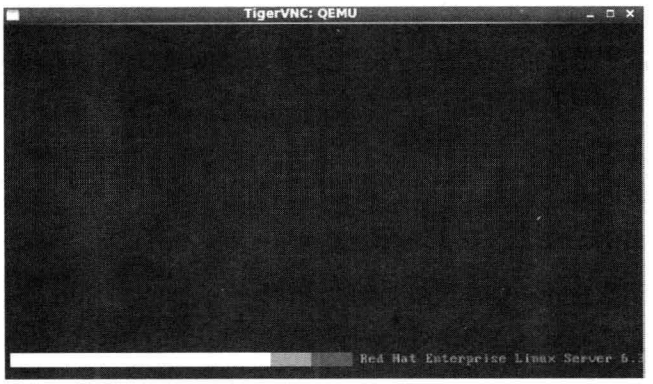


图 3-11 客户机启动中状态

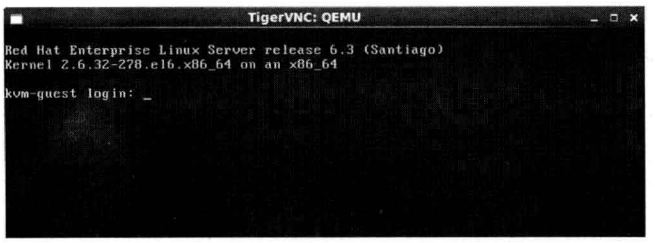


图 3-12 客户机启动后的登录界面

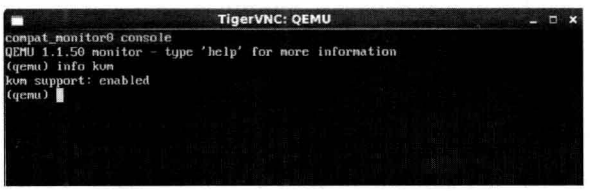


图 3-13 QEMU Monitor 中“info kvm”命令

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。

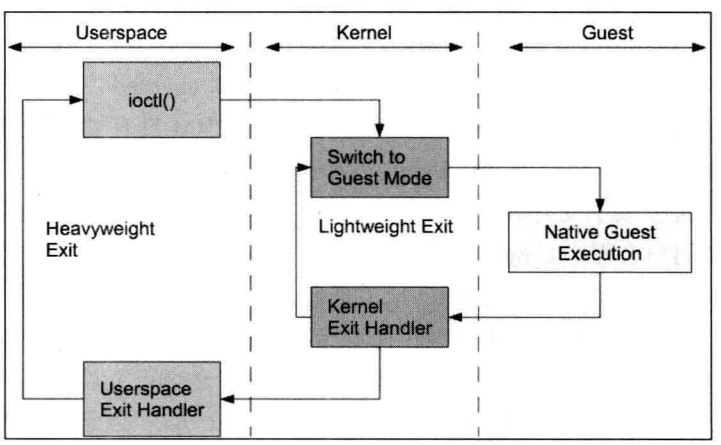


图 4-1 vCPU 在 KVM 中的三种执行模式

在 KVM 环境中，整个系统的基本架构如图 4-2 所示。

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。

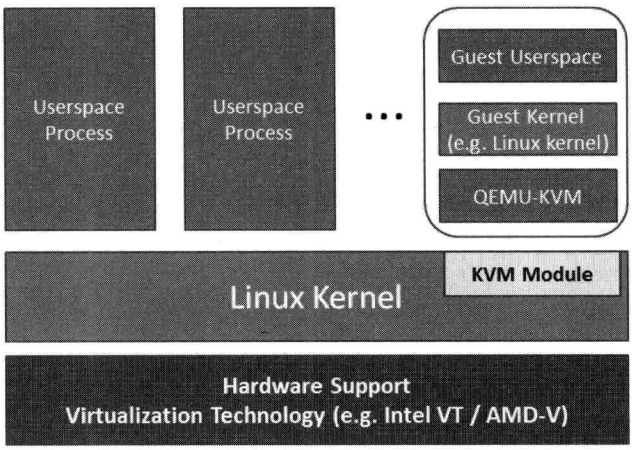


图 4-2 KVM 系统的分层架构

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）、Core

2 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）。

总的来说，在 KVM 环境中，一般并不推荐手动设置 QEMU 进程的处理器亲和性来绑定 vCPU，但是，在非常了解系统硬件架构的基础上，根据实际应用的需求，可以将其绑定到特定的 CPU 上，从而提高客户机中的 CPU 执行效率或实现 CPU 资源独享的隔离性。

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。

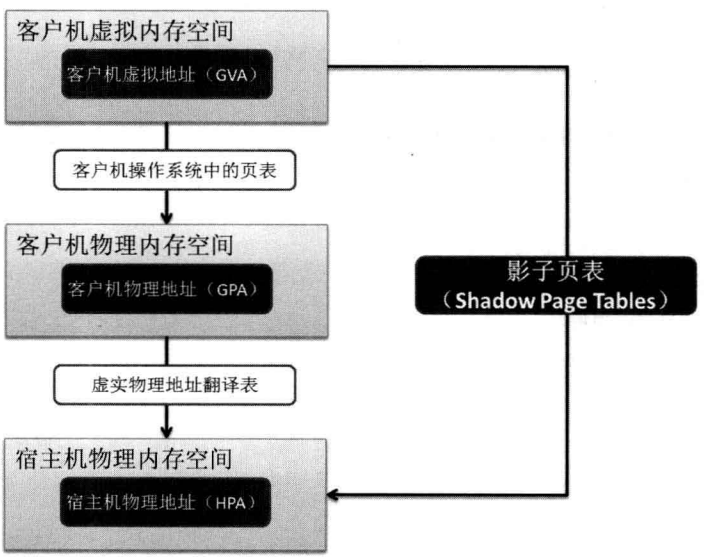


图 4-3 影子页表的作用

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。

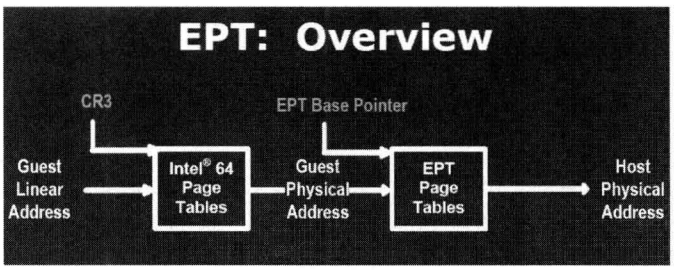


图 4-4 EPT 基本原理

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 f*ile*

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。

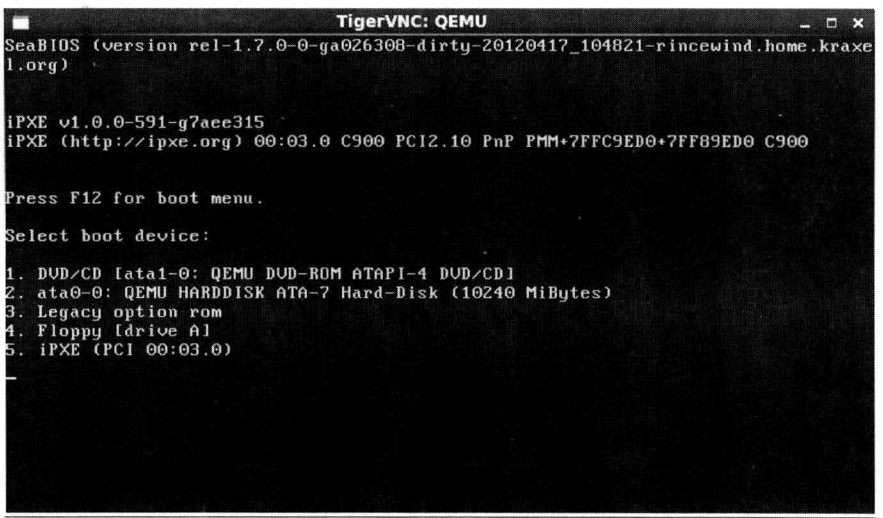


图 4-5 客户机中的启动菜单选项

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*] *filename* [*filename*2 [...]] *output_filename*

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。

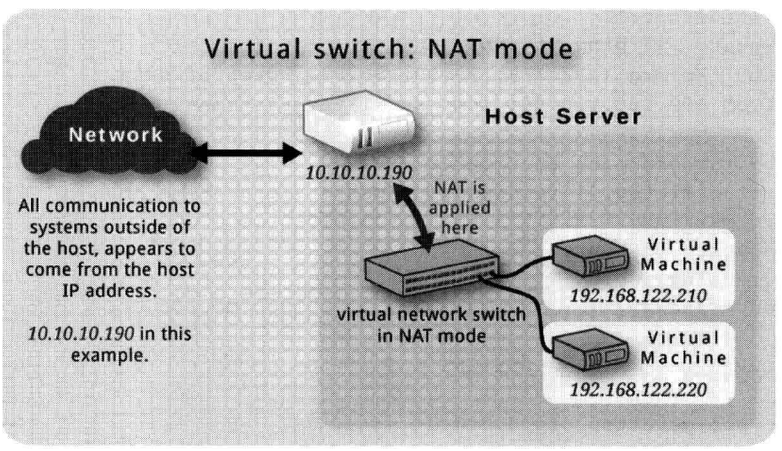


图 4-6 KVM 中的 NAT 模式网络

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。

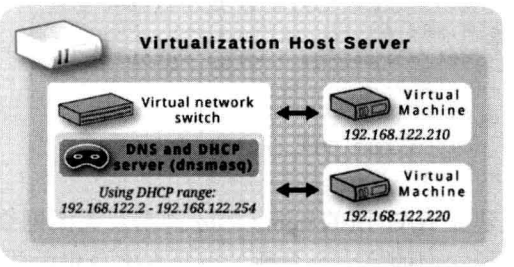


图4-7 宿主机中的dnsmasq为客户机提供DHCP服务

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.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 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

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。

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。

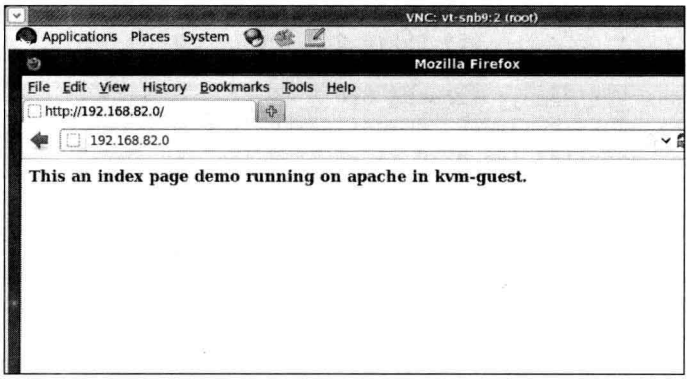


图 4-8 外部网络主机访问宿主机而被映射到客户机中相应的端口

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=*file*，让 *file* 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。

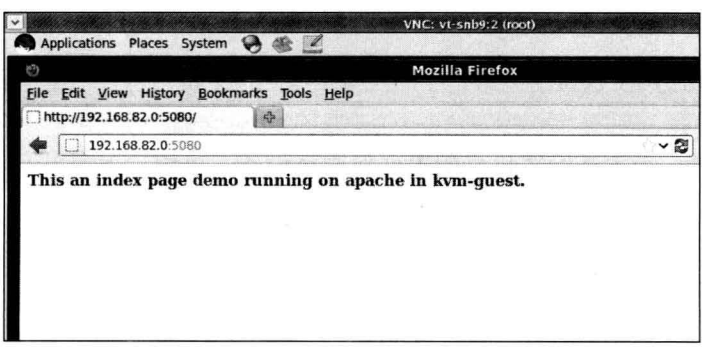


图 4-9 在宿主机转发功能作用下，外部网络得以访问用户模式网络中的客户机

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。

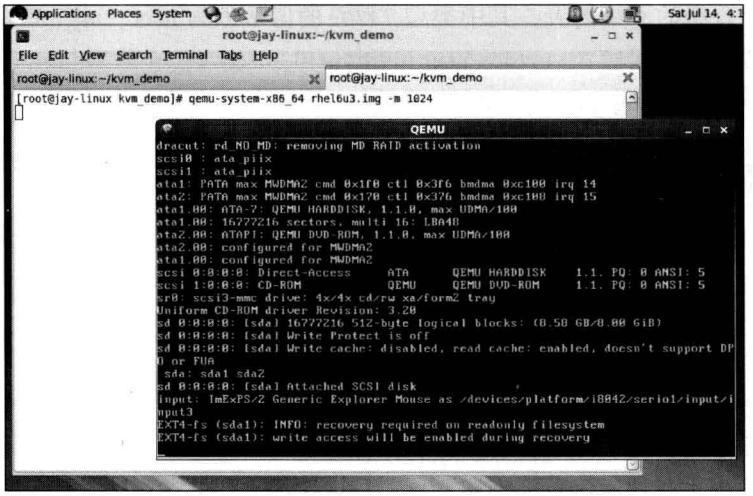


图 4-10 启动客户机时使用 SDL，自动弹出客户机的显示窗口

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。

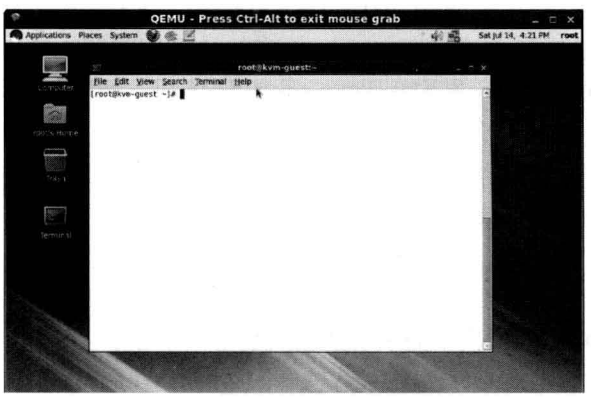


图 4-11 使用 SDL 时客户机完全占用鼠标

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。

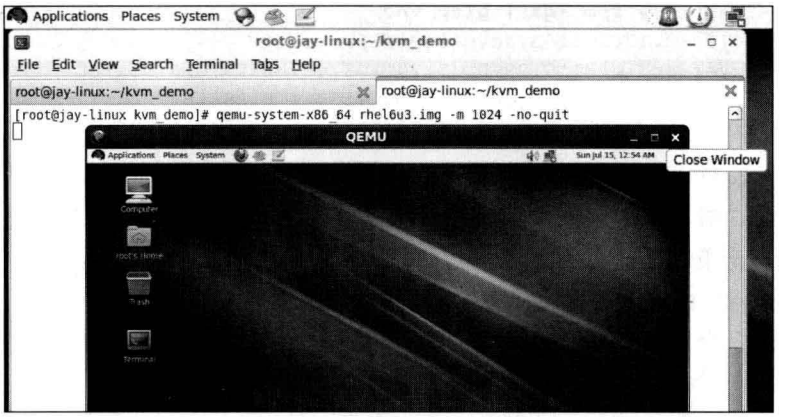


图 4-12 添加“-no-quit”参数后，SDL 窗口中的关闭功能失效

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。

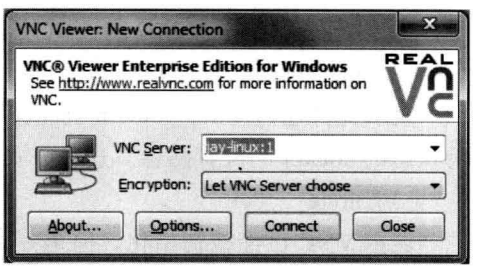


图4-14 在Windows 7上的VNC Viewer的远程连接界面

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。

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。

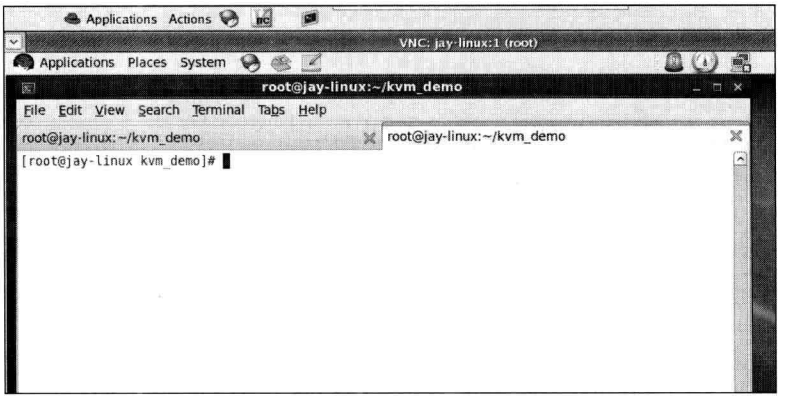


图 4-15 远程连接到宿主机的 1 号 VNC 桌面

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[18]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。

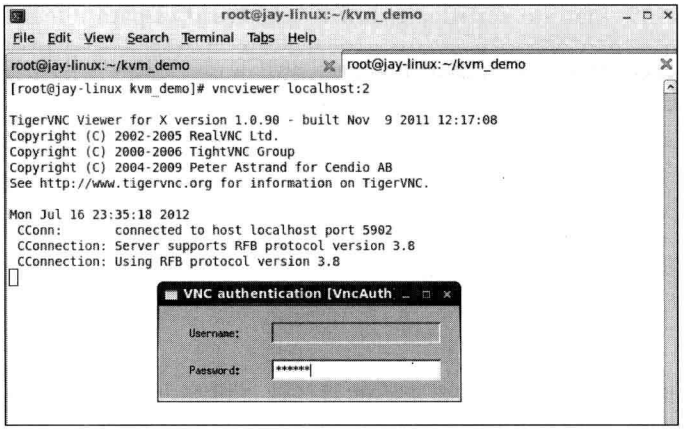


图 4-16 宿主机连接本机的 VNC，需要输入密码

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。

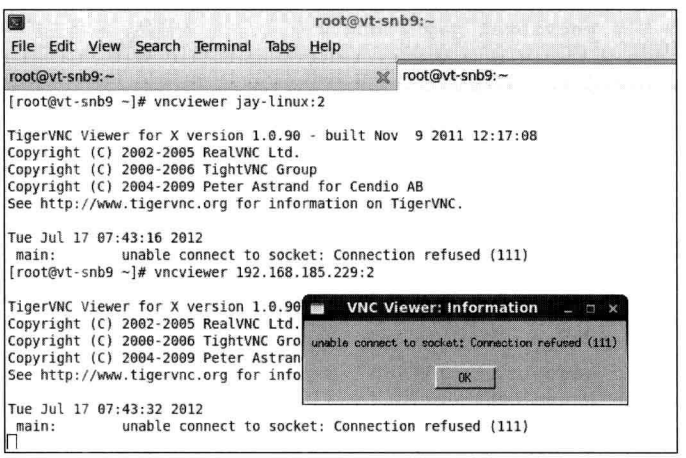


图 4-17 B 系统连接 A 系统中 QEMU 的 VNC 服务，连接被拒绝

（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。

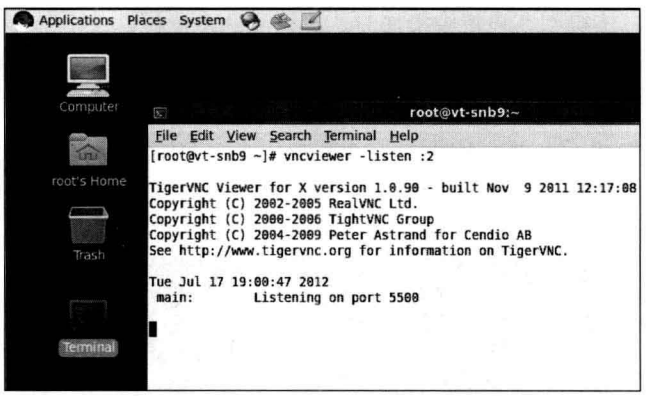


图 4-18 B 在系统中启动一个 VNC 客户端处于监听状态

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。

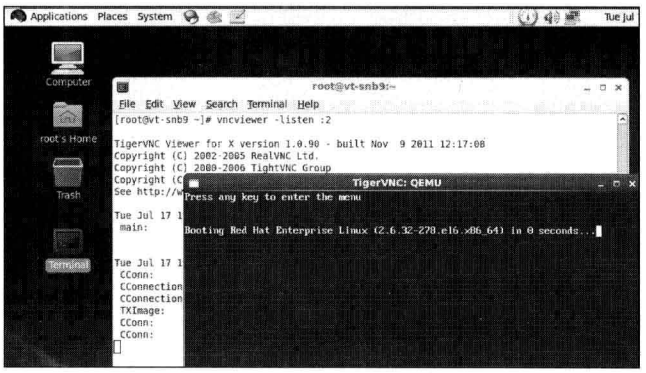


图 4-19 B 系统中监听着的 VNC 客户端已经连接到了 A 系统中的客户机

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。

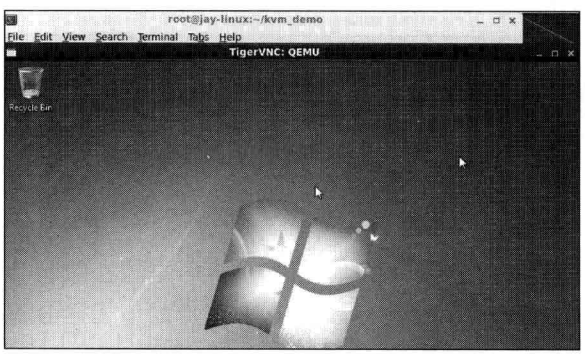


图 4-20 VNC 中的 Windows 7 客户机的鼠标偏移现象

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。

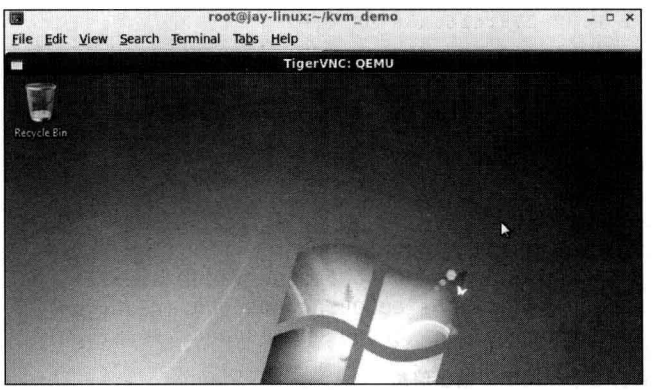


图 4-21 VNC 连接到 Windows 7 客户机中，无鼠标偏移现象

```
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。

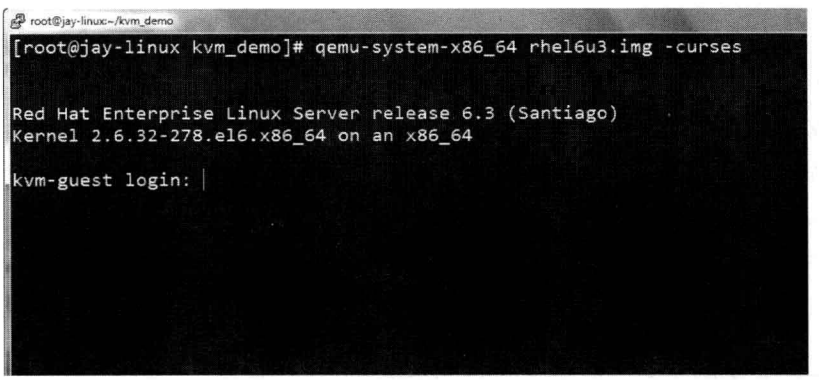


图 4-22 通过“ncurses”显示的文本模式下的客户机启动命令行和登录界面

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#vnc 005fsecurity

[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。

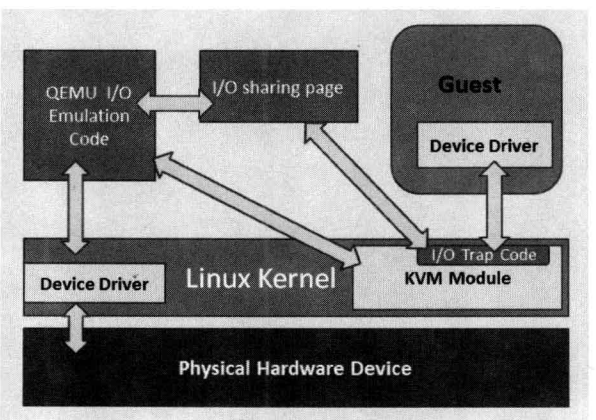


图 5-1　QEMU 模拟 I/O 设备

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]。

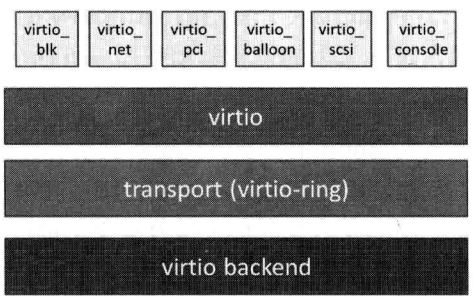


图 5-2 KVM 中 virtio 基本架构

在 QEMU/KVM 中，virtio 的基本结构框架如图 5-2 所示。

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。

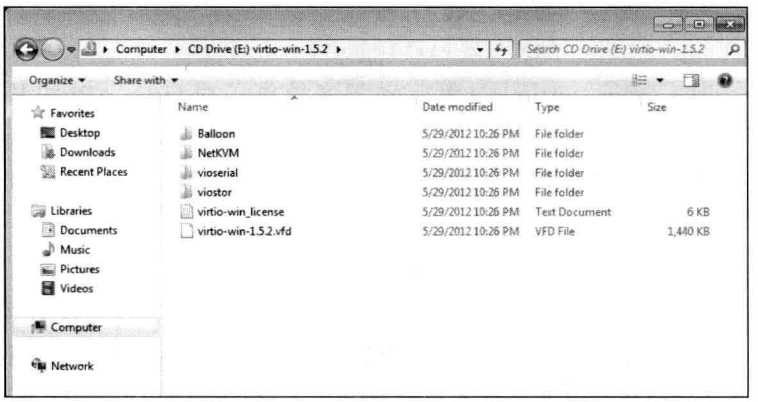


图 5-3 virtio-win 中的目录结构

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。

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。

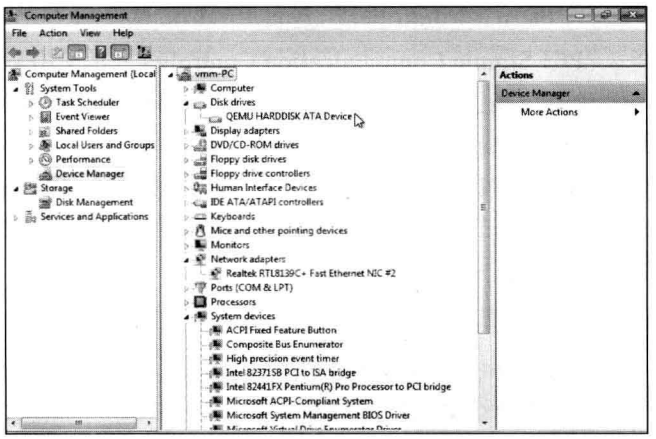


图 5-4 在安装 virtio 前，Windows 客户机中的设备管理器

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。

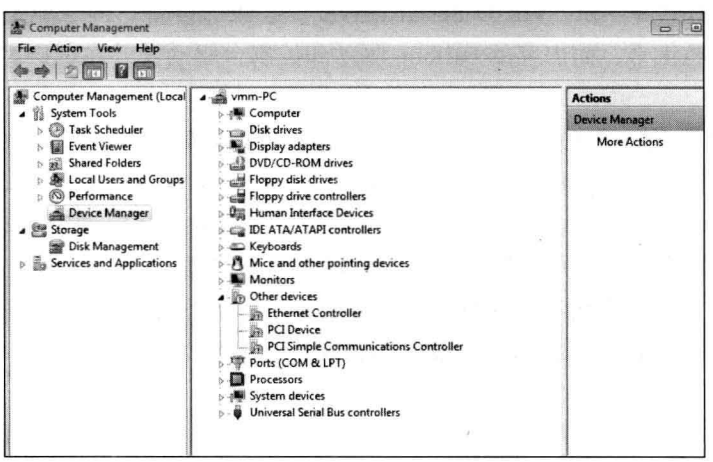


图 5-5 Windows 客户机中未安装 virtio 的驱动程序

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。

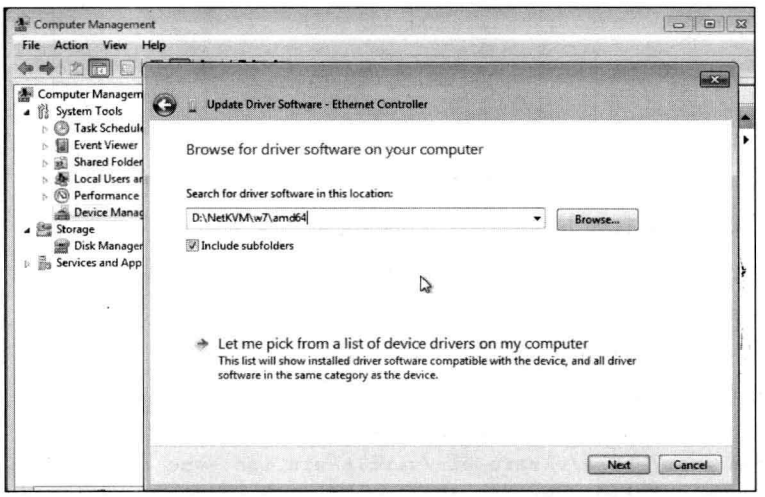


图5-6 选择网卡相关的virtio驱动程序目录

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。

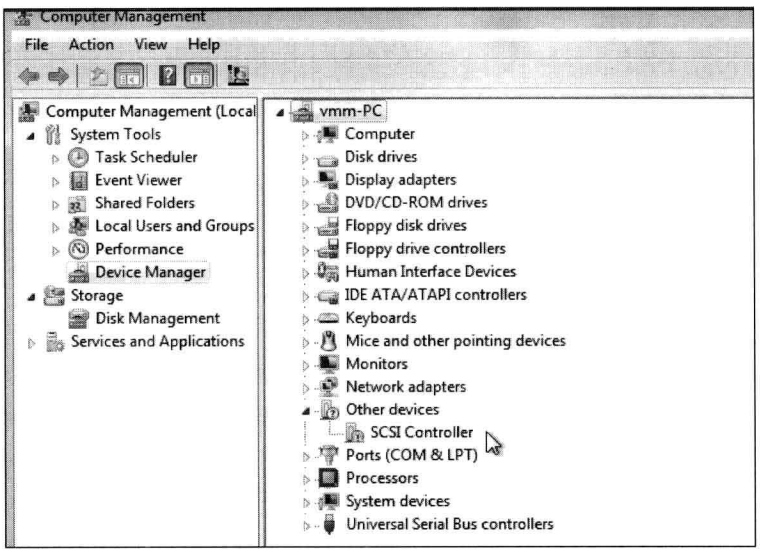


图 5-7 Windows 客户机在安装 virtio_scsi 驱动之前

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。

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。

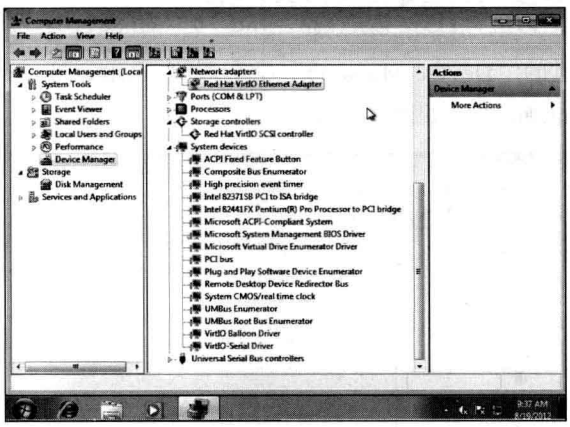


图 5-8 在 Windows 的“设备管理器”中查看 virtio 驱动

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。

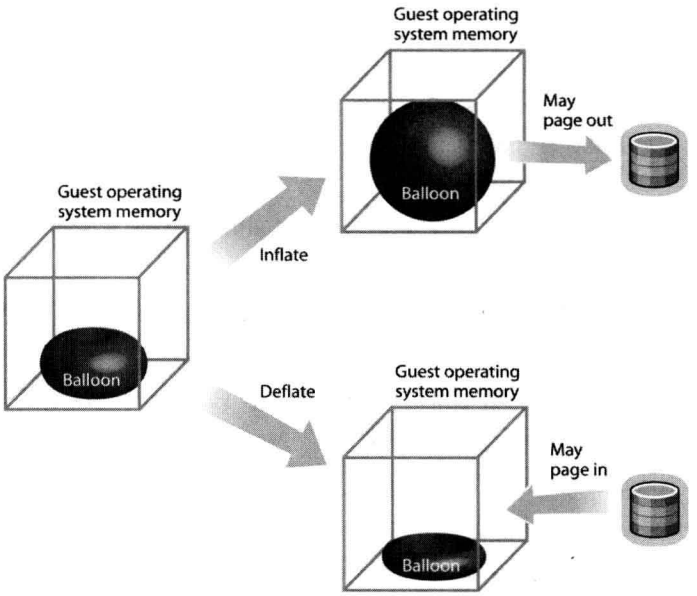


图 5-9 内存 balloon 的概念

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。

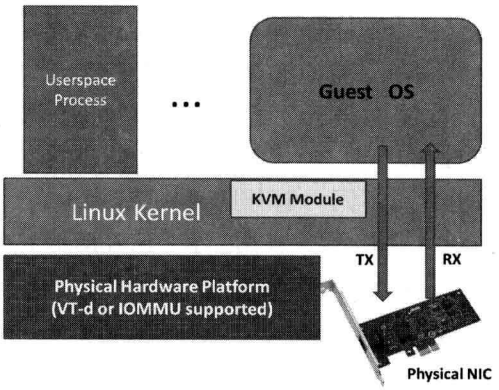


图 5-10 KVM 客户机直接分配 PCI-E 设备（以网卡为例）

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“-balloon

virtio”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。

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。

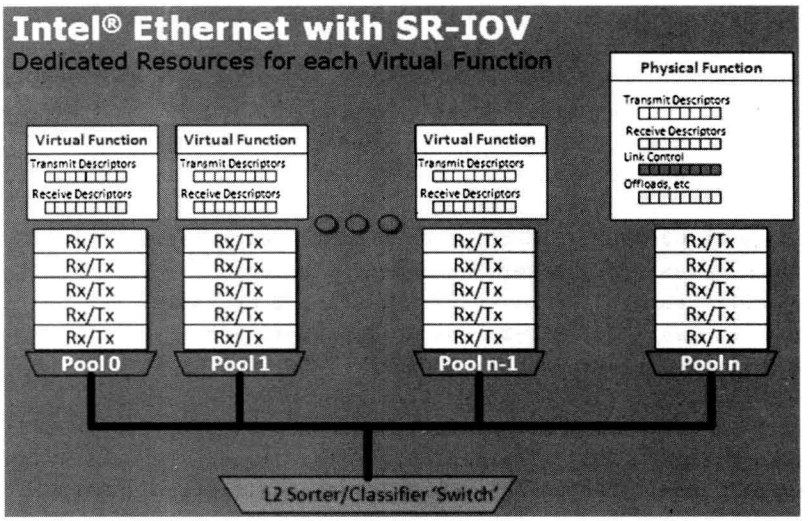


图 5-11 Intel 以太网卡的 SR-IOV 示例

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。

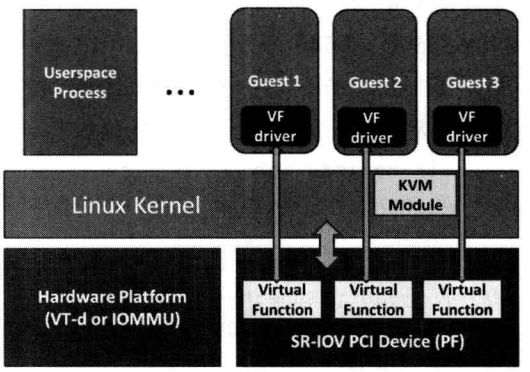


图 5-12 在 KVM 中使用 SR-IOV 功能

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。

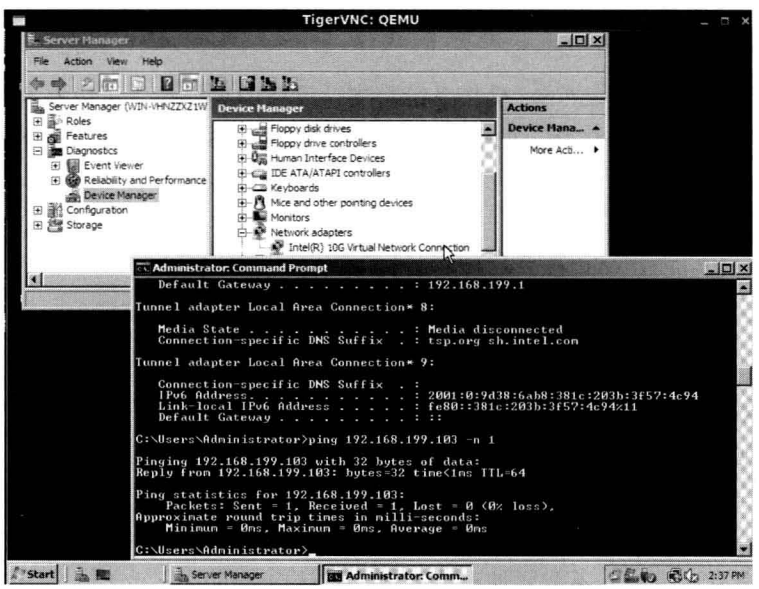


图 5-13 32 位 Windows 2008 Server 客户机中 82599 VF 的网络状态

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。

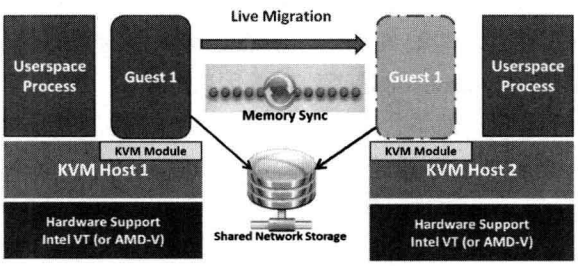


图 5-14 基于共享存储的 KVM 动态迁移

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。

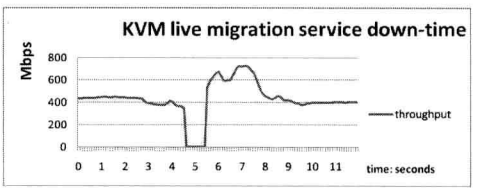


图 5-15 KVM 动态迁移中服务停机时间的某次测量结果

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。

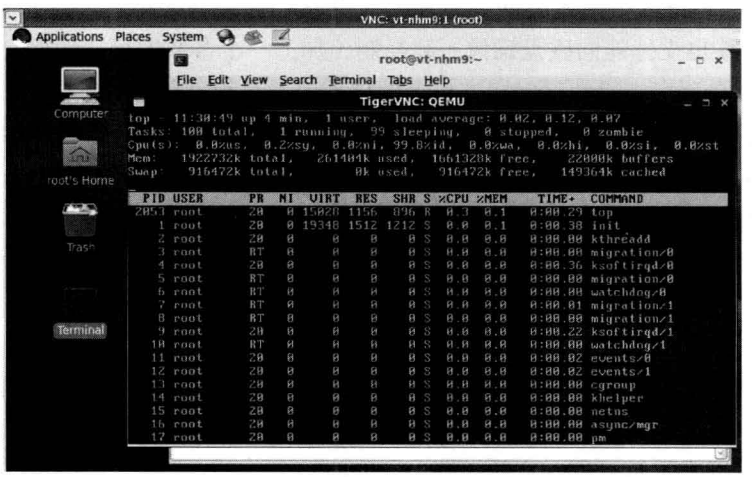


图 5-16 动态迁移前，客户机中运行着“top”命令

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。

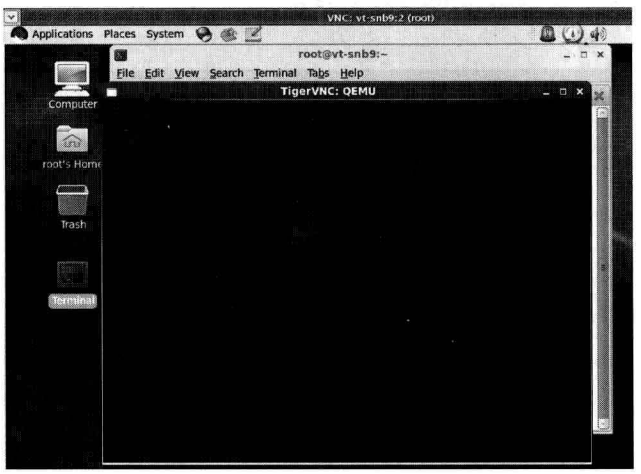


图 5-17 目的宿主机上带有“-incoming”参数的客户机

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。

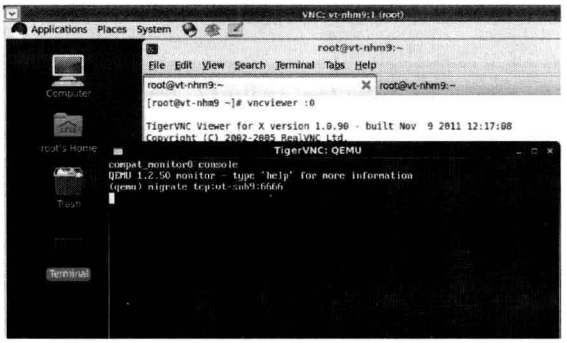


图 5-18 在 QEMU monitor 中使用“migrate”命令进行动态迁移

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。

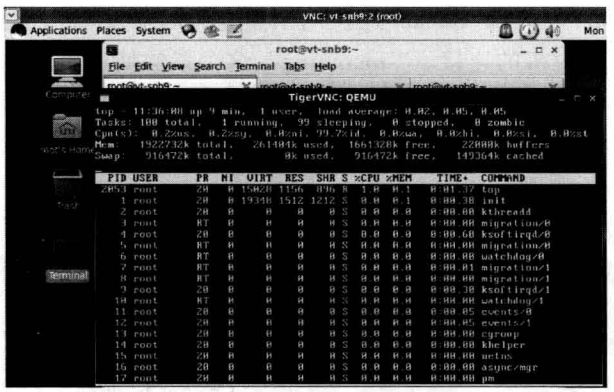


图 5-19 动态迁移后，客户机中的“top”命令依然在继续运行

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。

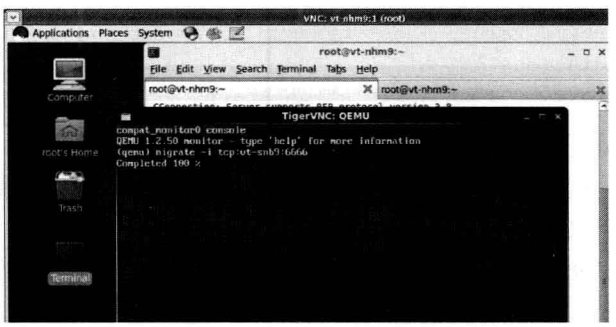


图 5-20 动态迁移“-i”参数显示传输增量部分的客户机镜像

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。

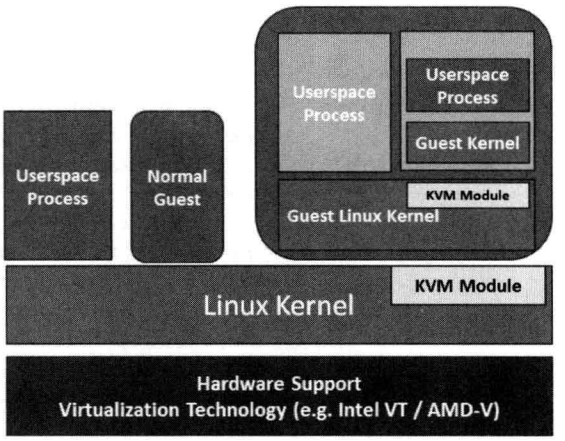


图 5-21 KVM 嵌套 KVM 的基本架构图

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“-cpu

SandyBridge,+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。

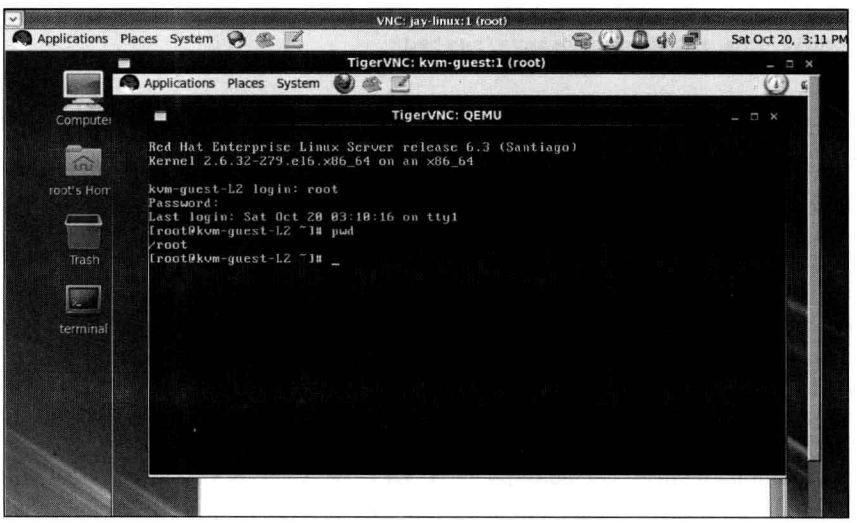


图5-22 KVM嵌套KVM的L0、L1、L2的示例

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。

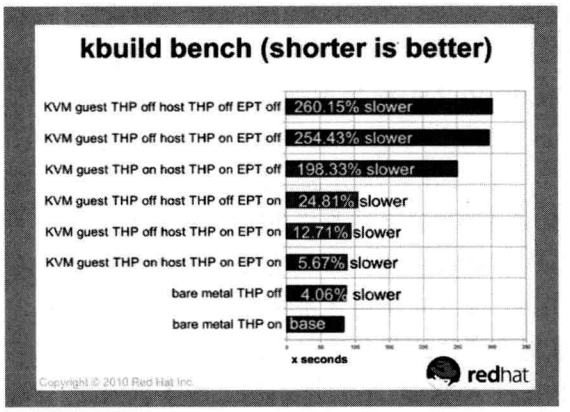


图 5-23　打开 EPT、THP 对内核编译效率的提高

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”。Sandy

Bridge 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。

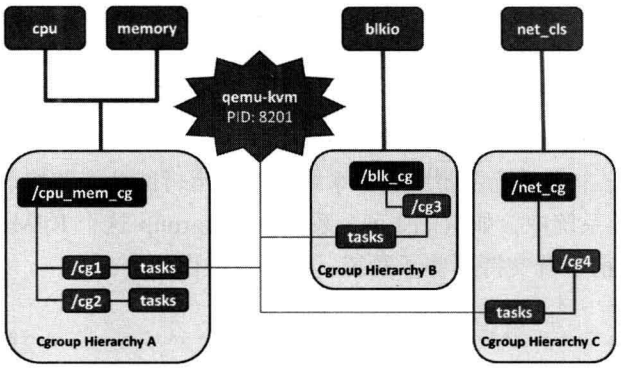


图 5-24 一个 cgroups 模型的示例

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。

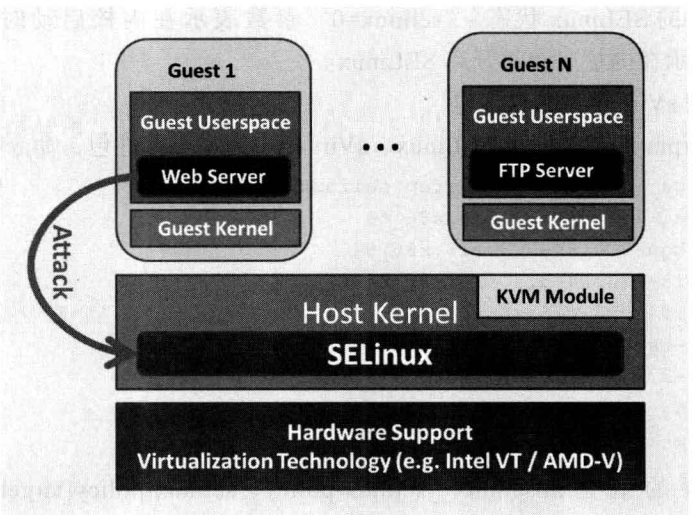


图5-25 宿主机中的sVirt（SELinux）阻止来自客户机的攻击

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。

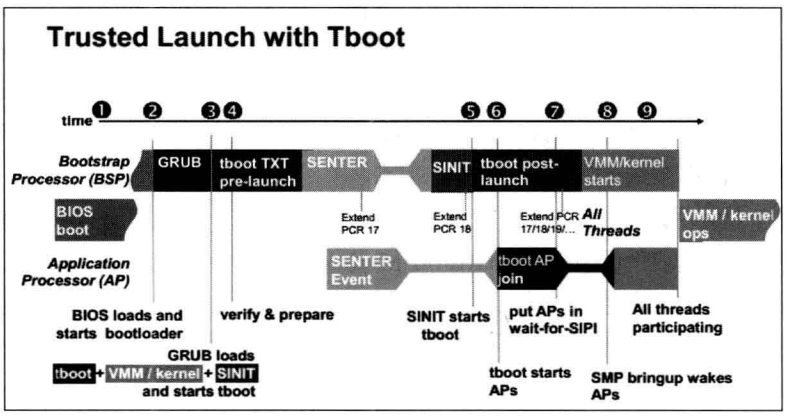


图 5-26 在 Tboot 支持下的可信任启动过程

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 /fmt *addr* 转存（dump）出从 *addr* 开始的虚拟内存地址。
- xp /fmt *addr* 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-config*”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-config*”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-mgrtvm1/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。

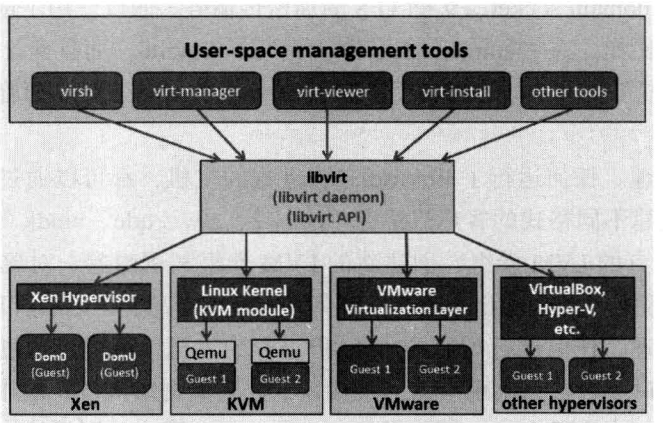


图 6-1　虚拟机管理工具通过 libvirt 管理各种类型的虚拟机

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。

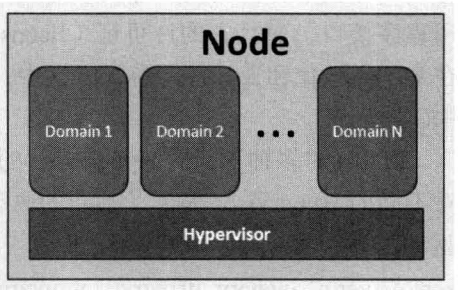


图 6-2 节点、Hypervisor 和域之间的关系

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(virConnectPtr conn, int id)函数是根据域的id值到conn这个连接上去查找相应的域。类似地，virDomainLookupByName、virDomainLookupByUUID等函数分别是根据域的名称和UUID去查找相应的域。在得到某个域的对象后，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。

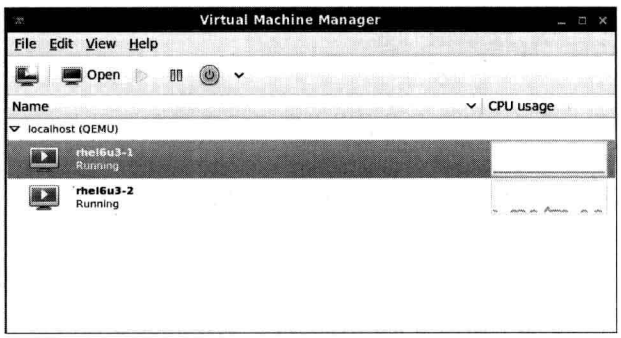


图 6-3 RHEL6.3 中的 virt-manager 界面

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。

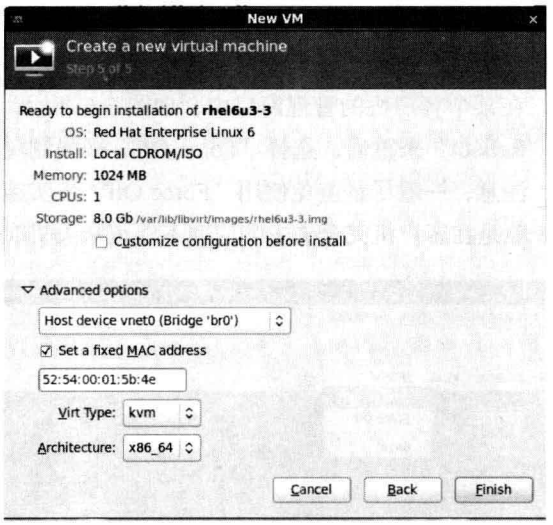


图 6-4 virt-manager 中创建客户机的最后一个步骤

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。

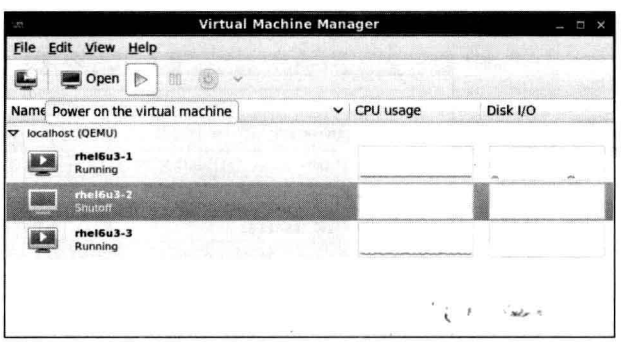


图 6-5 virt-manager 中的开机按钮

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。

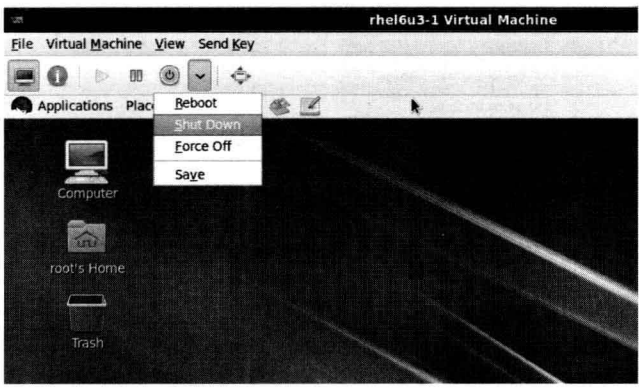


图 6-6 virt-manager 中的关机选项

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。

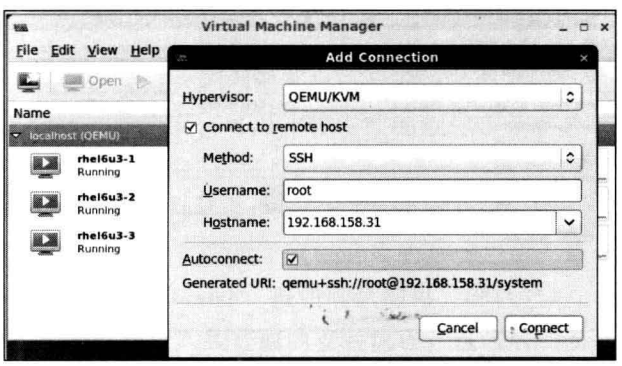


图 6-7 在 virt-manager 中建立到远程主机的连接

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。

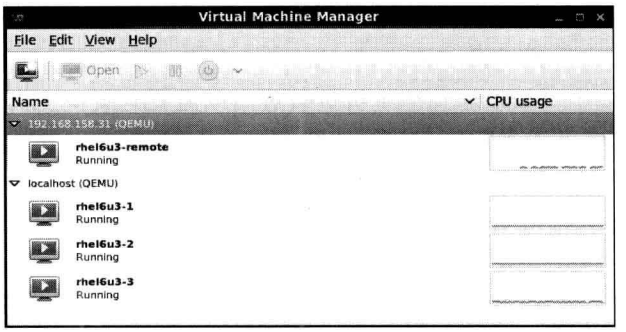


图 6-8 在 virt-manager 中管理本地和远程主机

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。

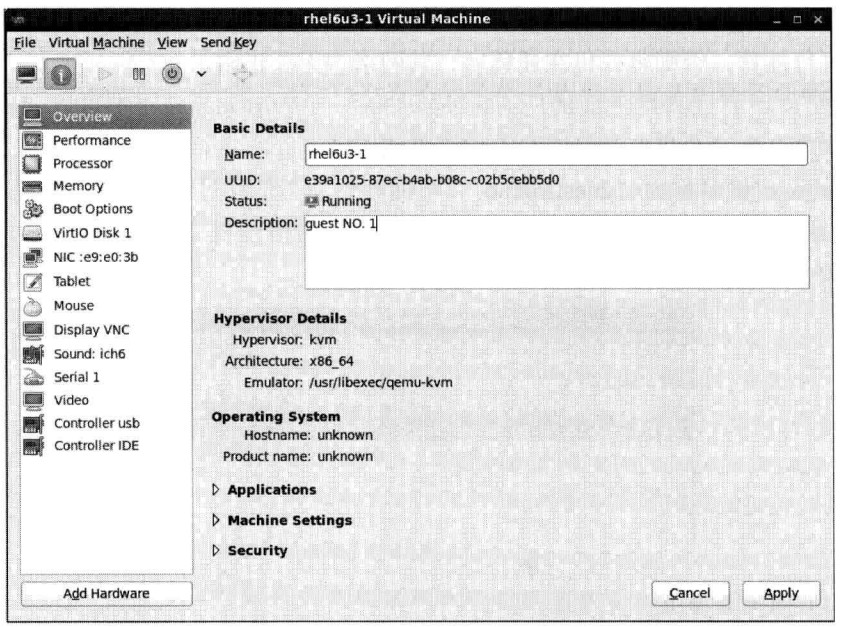


图 6-9 virt-manager 中客户机的详细配置

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。

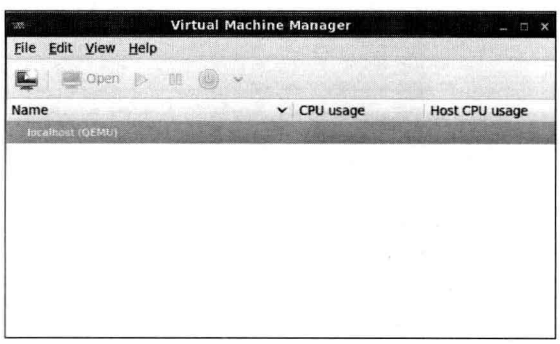


图 6-10 在动态迁移前，目的宿主机上没有运行客户机

（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。

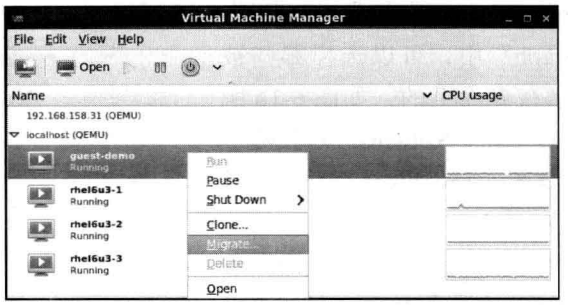


图 6-11　选择“Migrate”选项

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。

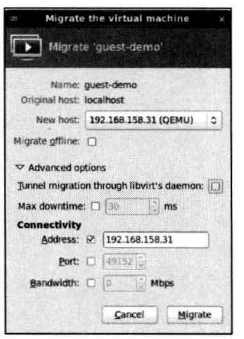


图 6-12　virt-manager 中动态迁移选项的配置

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。

（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。

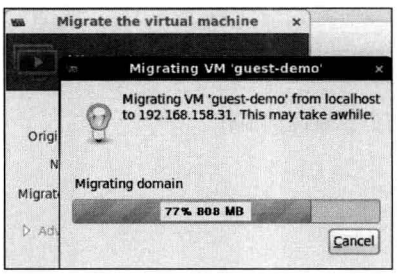


图 6-13　virt-manager 中动态迁移的进度展示

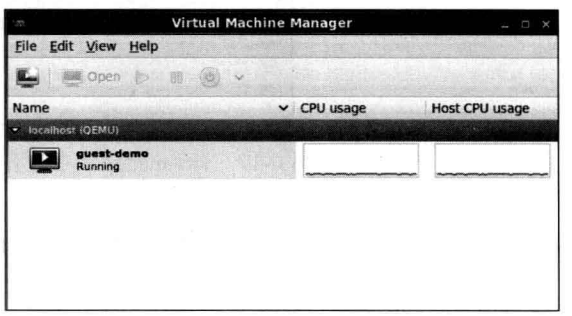


图 6-14　动态迁移完成后，在目的宿主机上查看客户机

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。

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。

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。

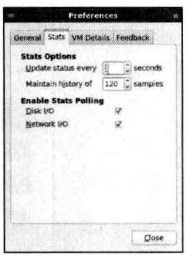


图6-15 virt-manager中的监控参数的配置

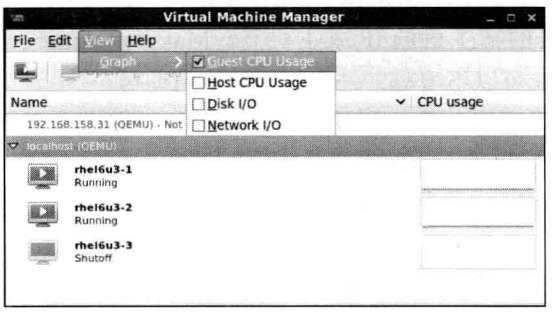


图6-16 选择需要显示到资源监控图

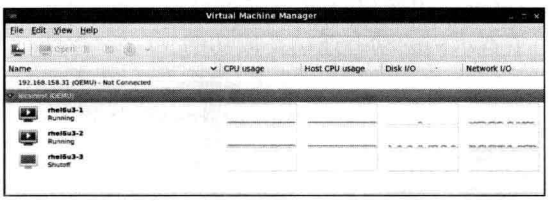


图6-17 virt-manager中对客户机和宿主机资源的监控

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。

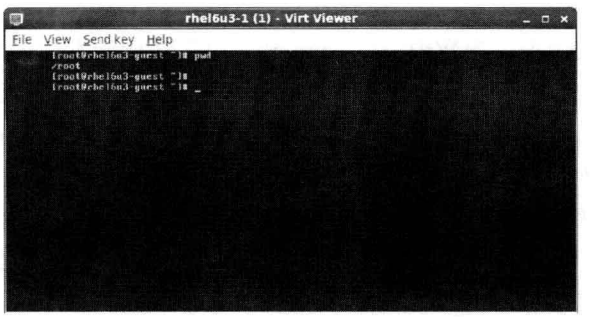


图 6-18 virt-viewer 连接到本地的一个客户机

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。

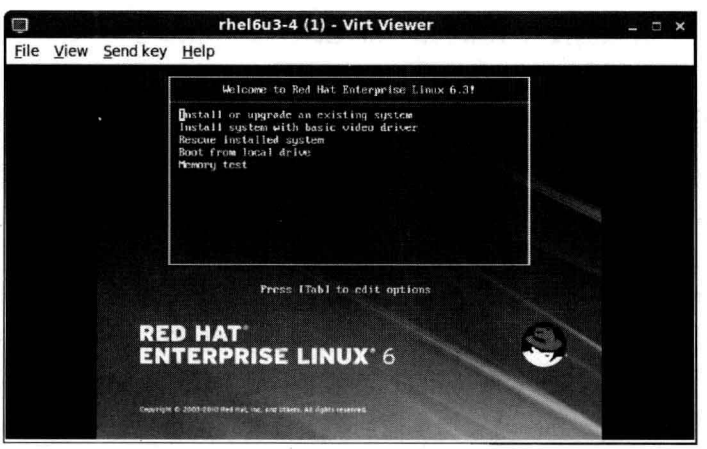


图 6-19 virt-install 在安装一个 RHEL6.3 客户机

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。

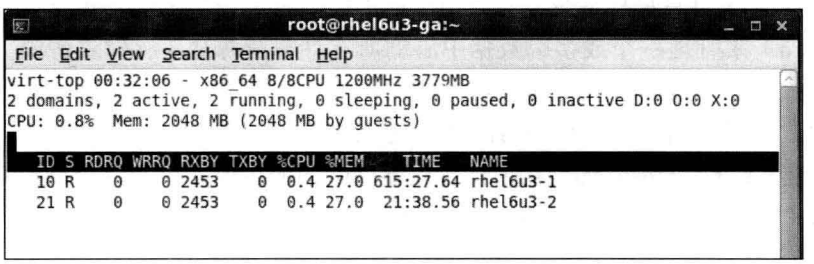


图 6-20 运行“virt-top”命令查看客户机的状态和资源利用率

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）。

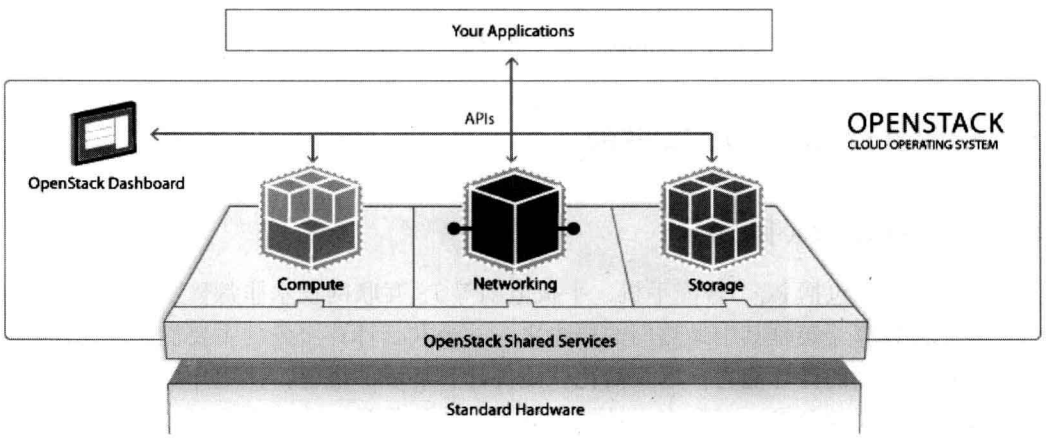


图 6-21 OpenStack 在云计算架构中的位置

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]）。

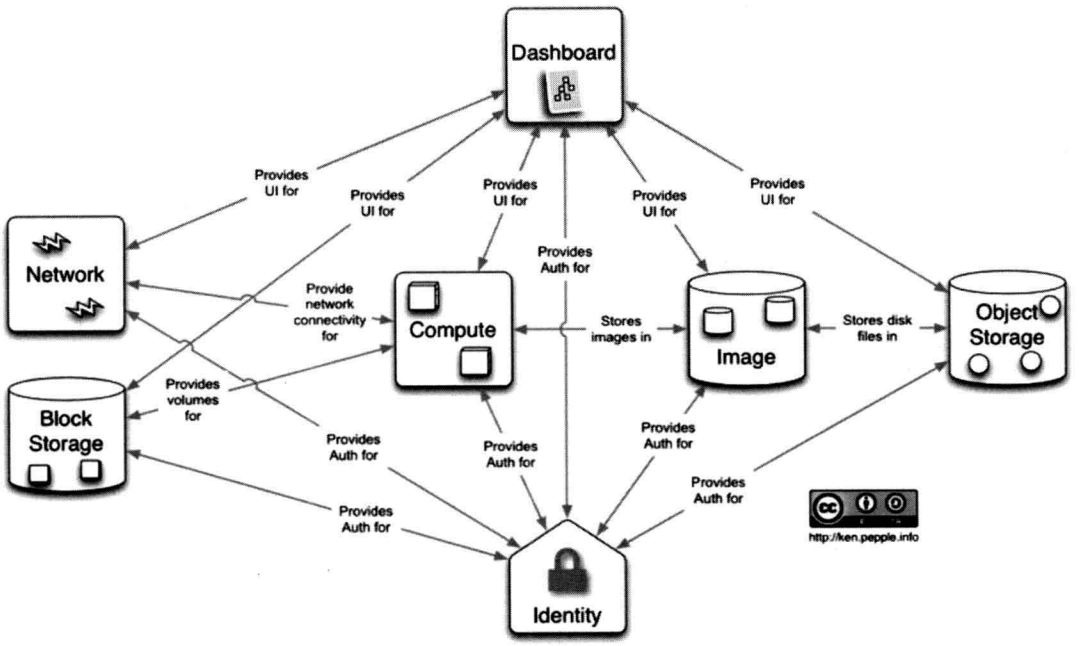


图 6-22 OpenStack 中 7 个核心组件的逻辑架构

（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）。

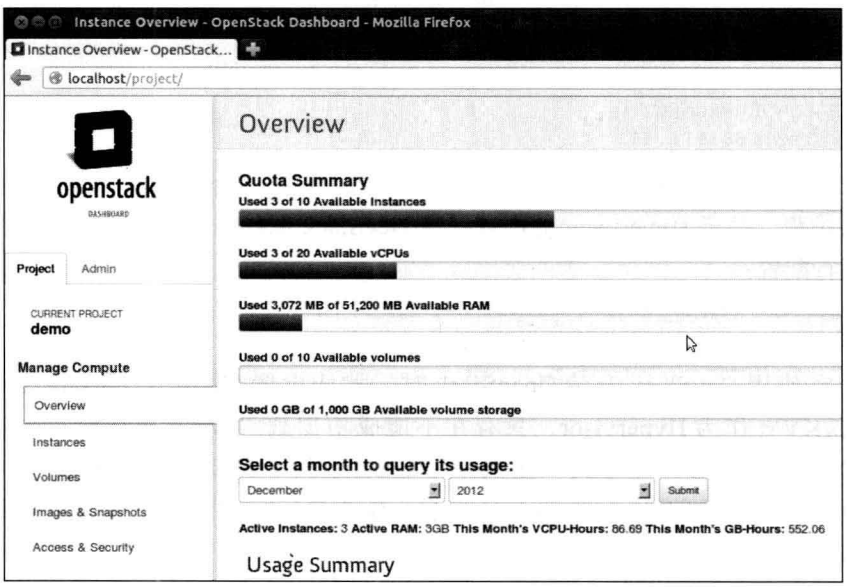


图 6-23 在 OpenStack 控制面板中查看 Demo 项目

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。

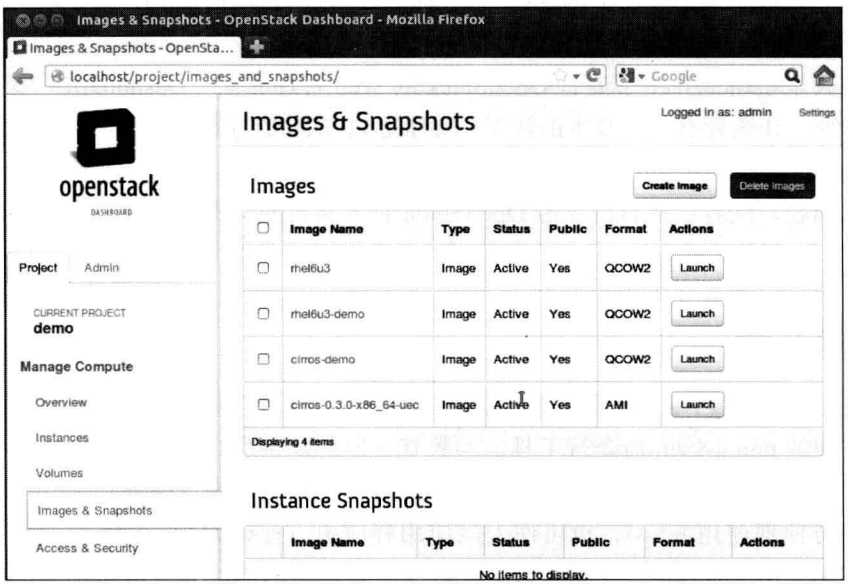


图 6-24　OpenStack 中管理镜像和快照

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

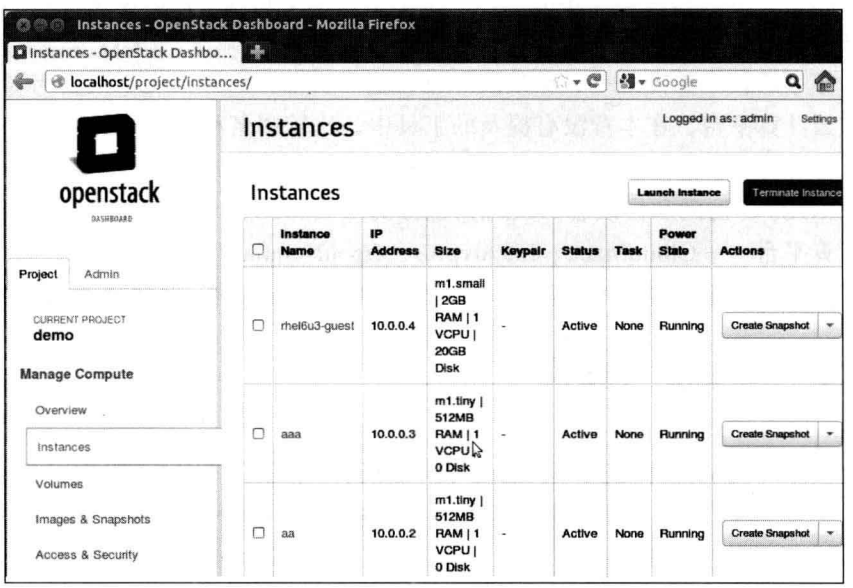


图 6-25　OpenStack 中的实例管理

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。

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。

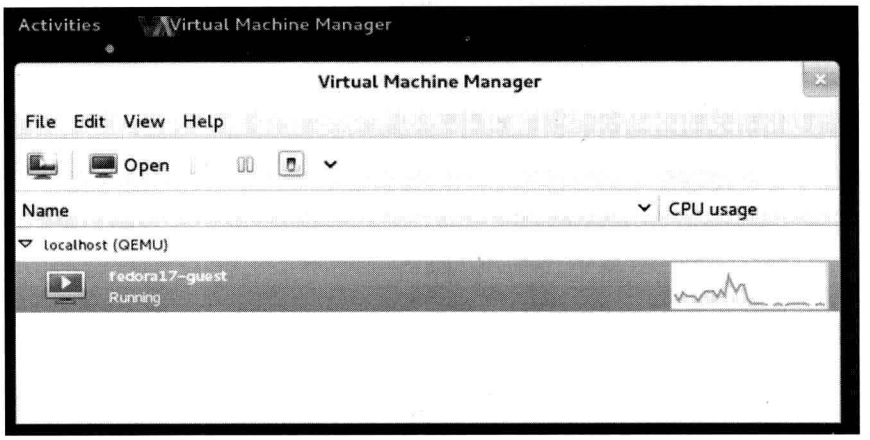


图 7-1 Fedora 17 系统的 virt-manager 工具中运行着 KVM 客户机

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。

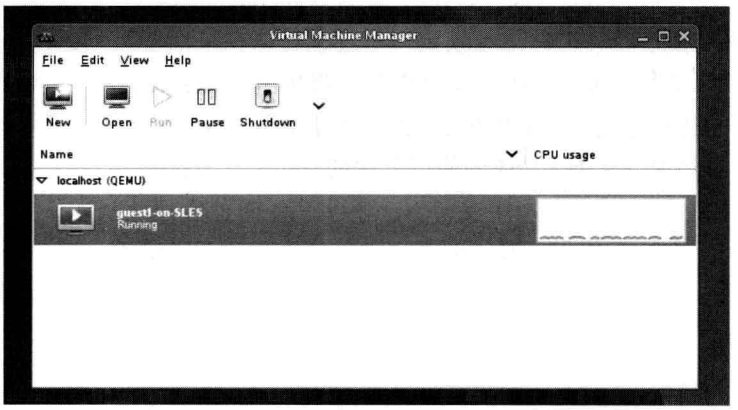


图 7-3 SLES11 SP2 中的 virt-manager 运行了一个 KVM 客户机

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。

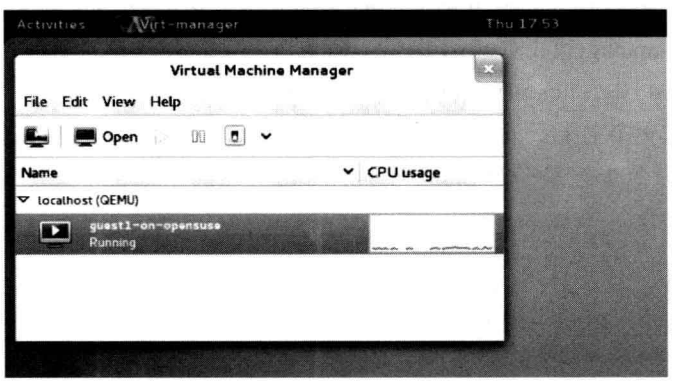


图 7-4 openSUSE 12.2 中的 virt-manager

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。

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。

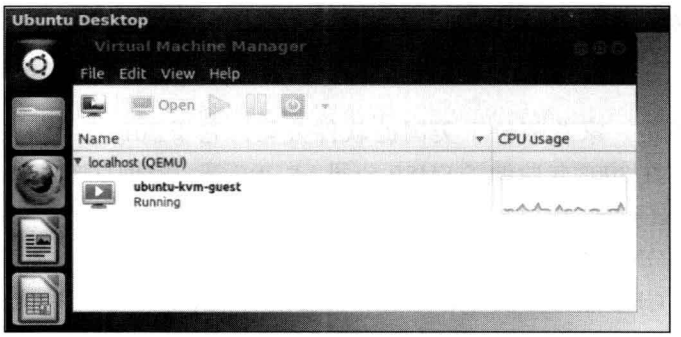


图 7-5 运行着一个 KVM 客户机的 virt-manager



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%。

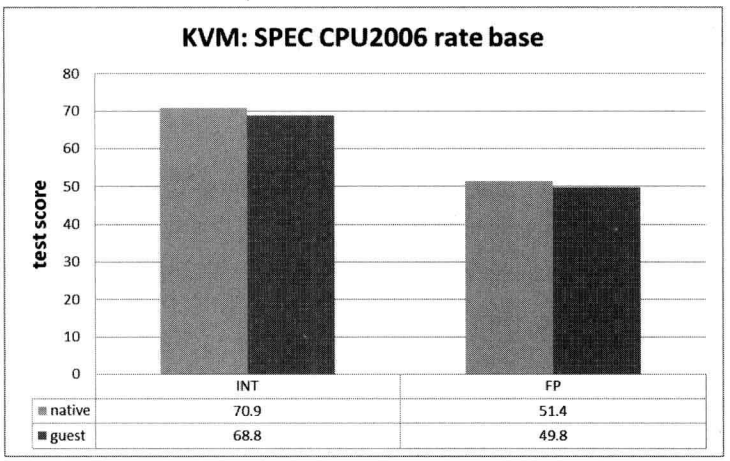


图 8-1 SPEC CPU2006 整型和浮点型测试的总体结果对比 [5]

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。

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%）。

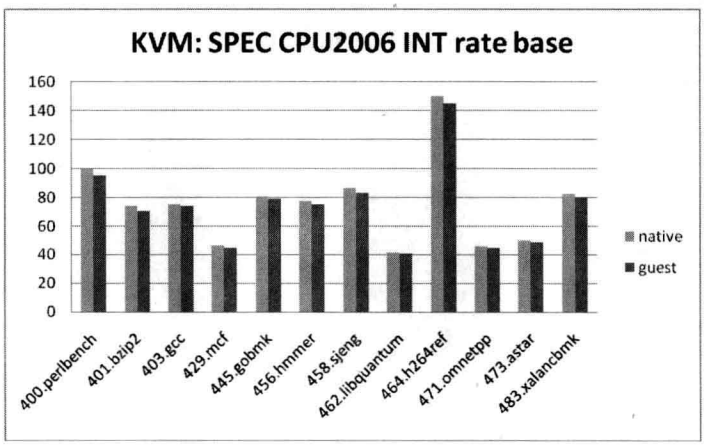


图 8-2 SPEC CPU2006 整型测试数据的对比

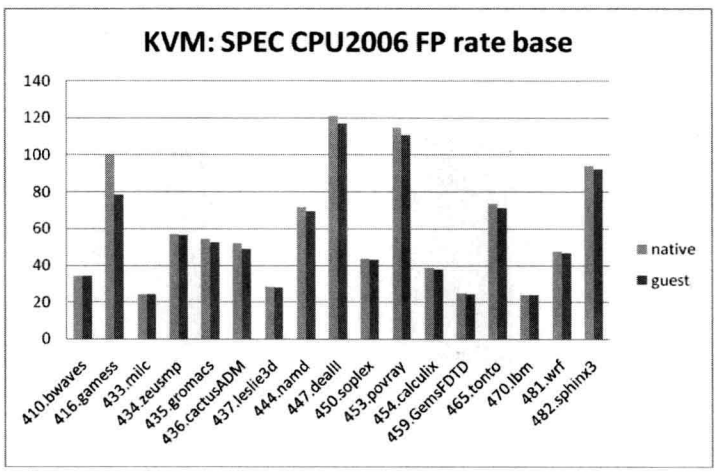


图 8-3 SPEC CPU2006 浮点型测试数据的对比

（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。

（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%。

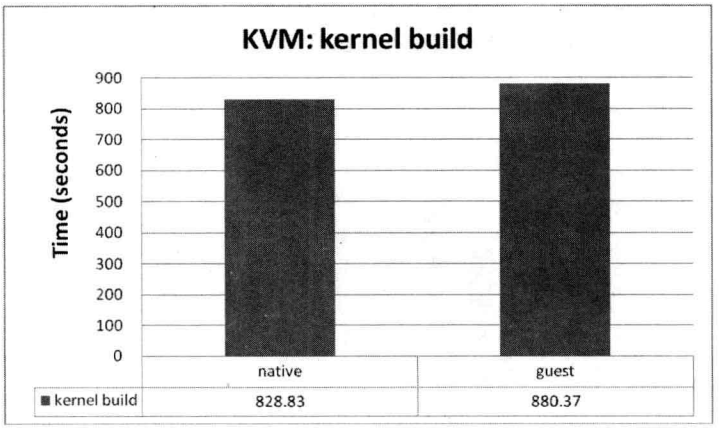


图 8-4 内核编译所需时间的对比

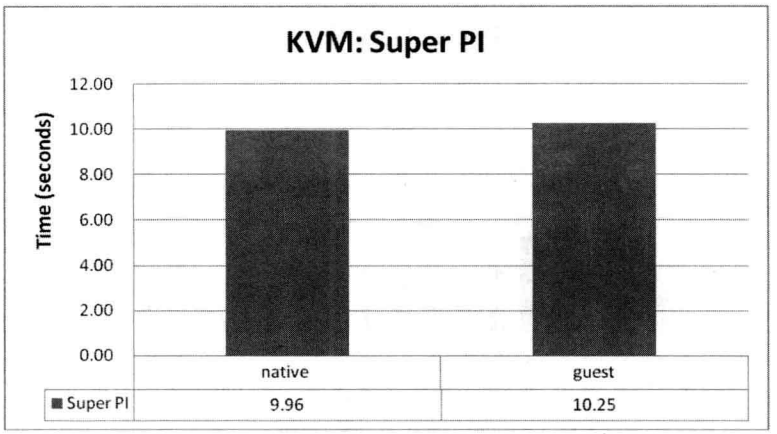


图 8-5 Super PI 基准测试数据对比

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。

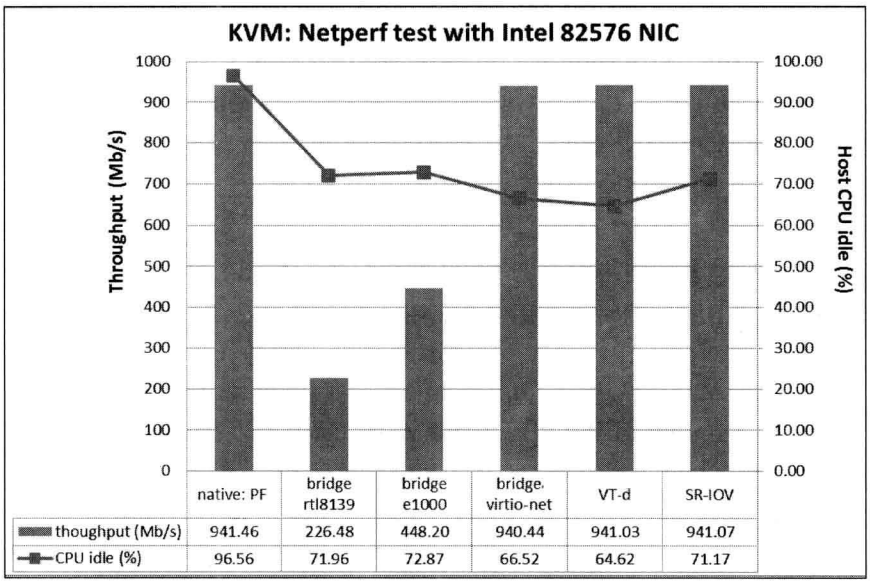


图 8-6 Netperf 使用 Intel 82576 网卡的测试数据对比

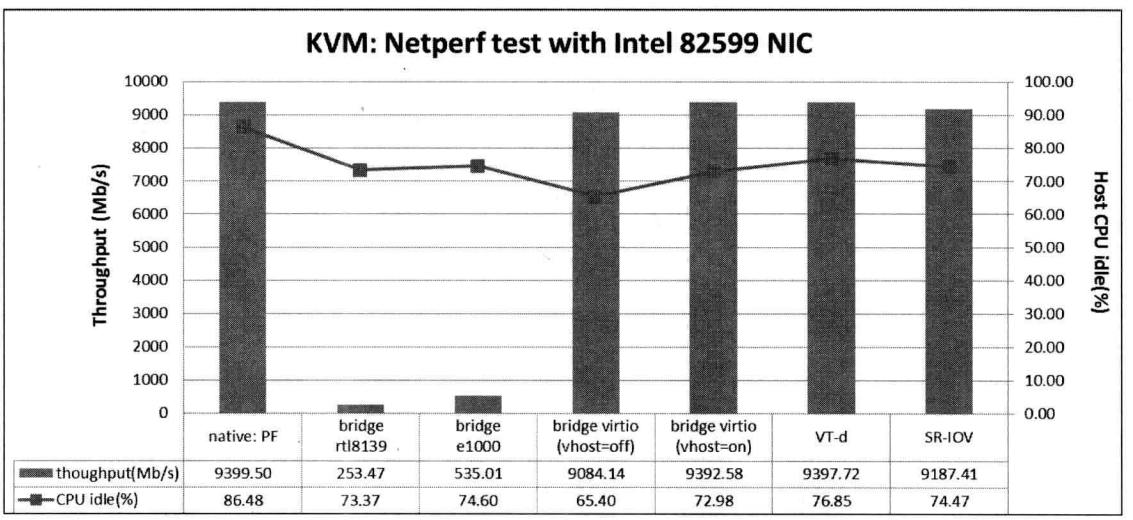


图 8-7 Netperf 使用 Intel 82599 网卡的测试数据对比

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。

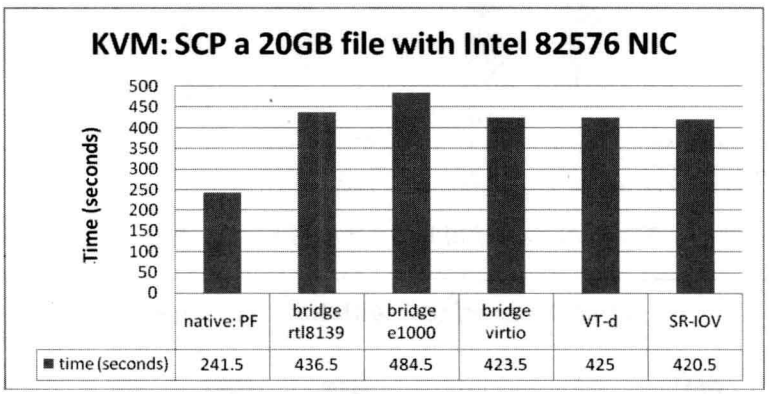


图 8-8 SCP 使用 Intel 82576 网卡的测试数据对比

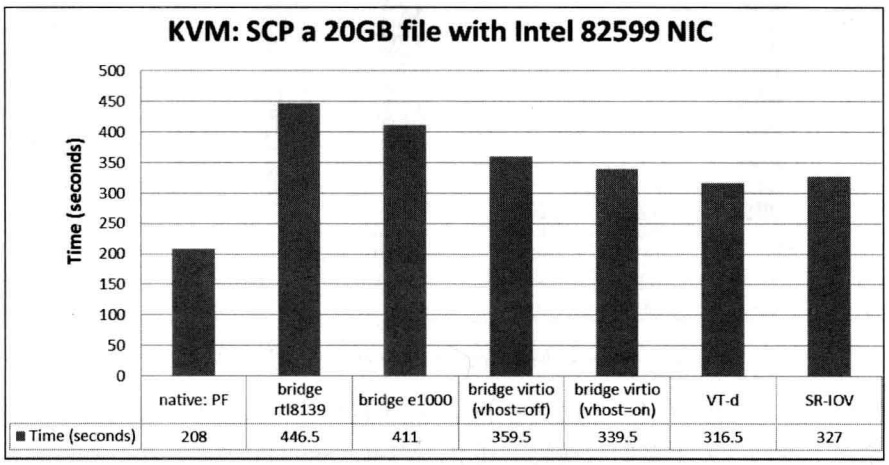


图 8-9 SCP 使用 Intel 82599 网卡的测试数据对比

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

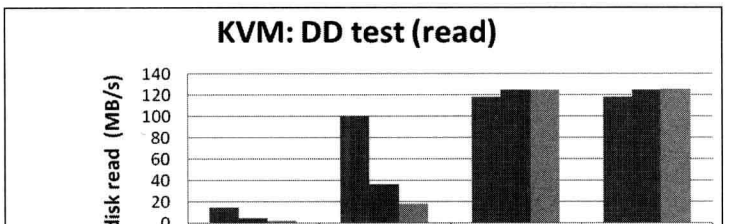
分别用 DD、IOzone、Bonnie++ 这 3 个工具在原生系统和 KVM 客户机中进行测试，然后对比其测试结果数据。为了尽量减小误差，每个测试项目都收集了 3 次测试数据，下面提

供的测试数据都是根据 3 次测试计算出的平均值。

（1）DD

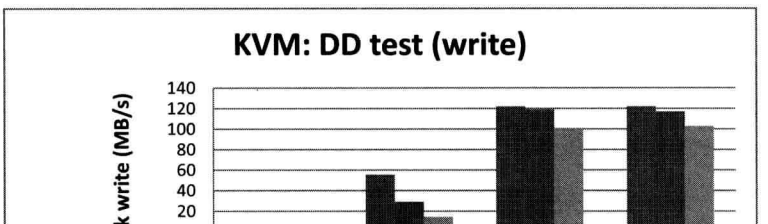
使用 DD 工具测试磁盘读写性能，将得到的测试数据进行对比，如图 8-10 和图 8-11 所示，读写速度越大，说明磁盘 I/O 性能越好。

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。



	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



	bs=1k	bs=8k	bs=1M	bs=8M
native STAT	6.2	55.5	122	122
guest virtio-blk	3.6	29.1	119	117
guest emulated IDE	1.8	14.6	101	103

图 8-11 DD 测试的磁盘写入速度对比

（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

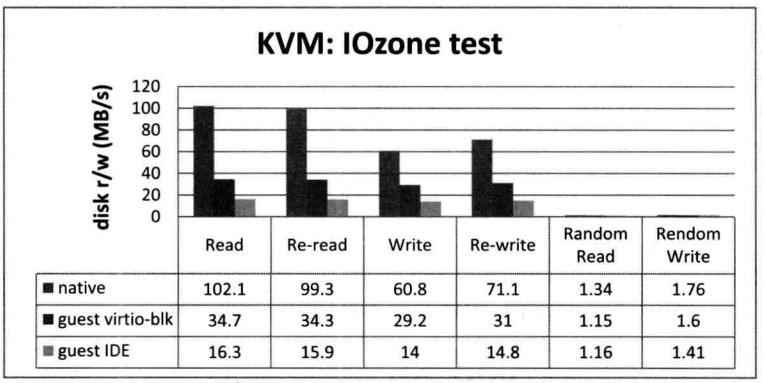
次 IOzone 测试的是一次读写 8 KB 大小的数据块（-r 8k 参数），本次测试结果与使用 DD 工具测试的图 8-10 和图 8-11 中 bs=8k 时的测试数据基本一致。

（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）。

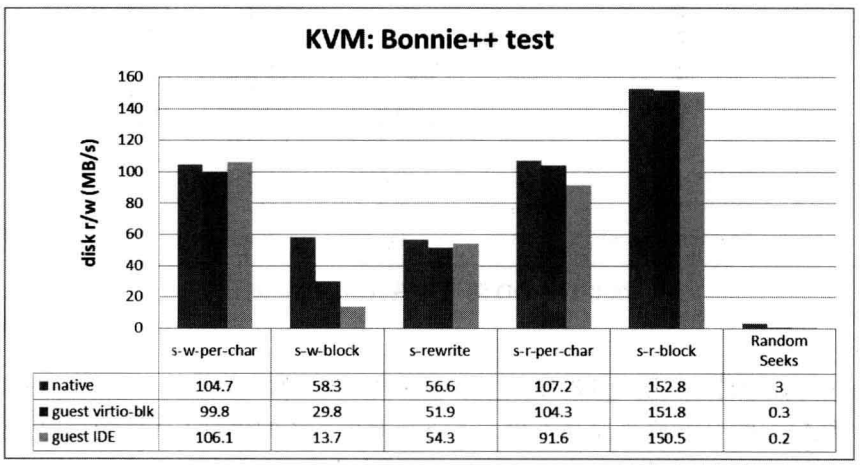
本次 Bonnie++ 测试没有指定一次读写数据块的大小，默认值是 8 KB，所以图 8-13 的测试结果与前面 DD、IOzone 工具测试每次读写 8 KB 数据时的测试结果是基本一致的。

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。



	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



	s-w-per-char	s-w-block	s-rewrite	s-r-per-char	s-r-block	Random Seeks
native	104.7	58.3	56.6	107.2	152.8	3
guest virtio-blk	99.8	29.8	51.9	104.3	151.8	0.3
guest IDE	106.1	13.7	54.3	91.6	150.5	0.2

图 8-13 Bonnie++ 测试的磁盘 I/O 速度对比

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。

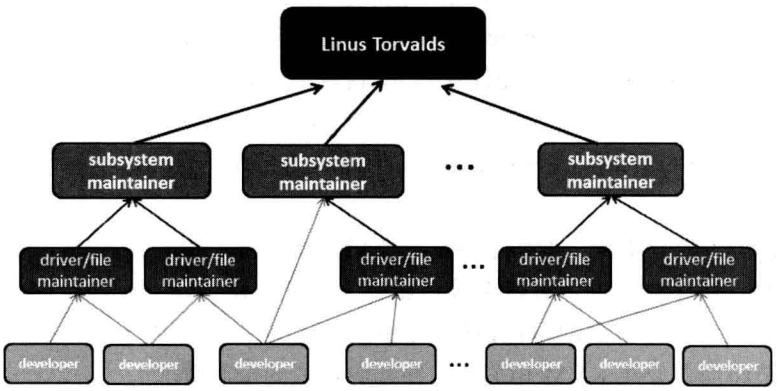


图 9-1 Linux 内核代码维护层次结构

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。

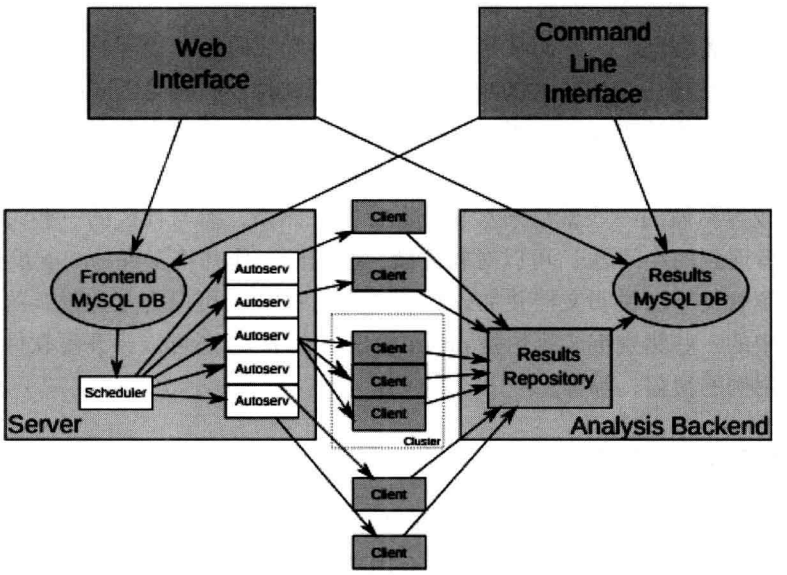


图 9-2 KVM Autotest 的基本架构

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。

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。

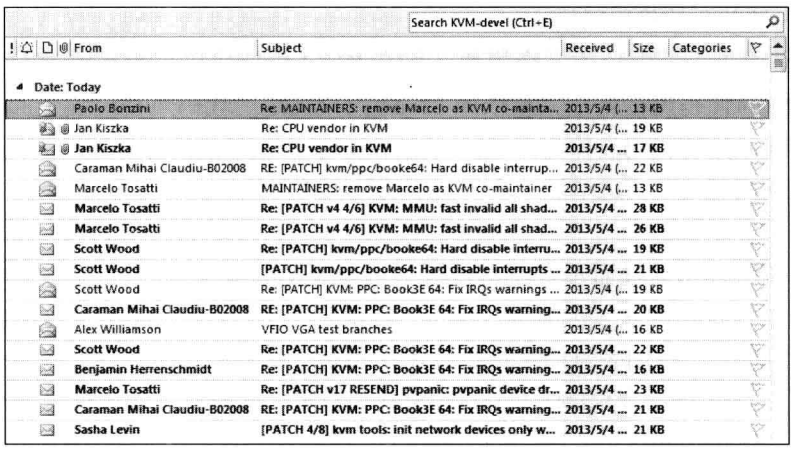


图 9-3 在 Outlook 中查看 KVM 邮件

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

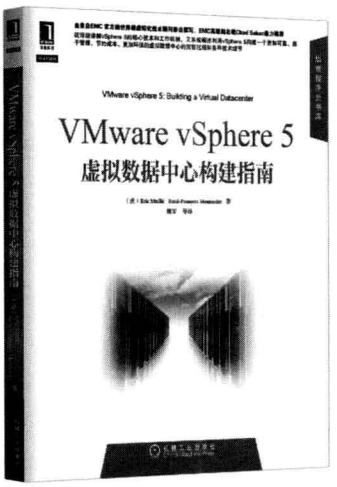
（1）Wikipedia 中关于 Linux 和 Linux 内核的词条（http://en.wikipedia.org/wiki/Linux 和 http://en.wikipedia.org/wiki/Linux_kernel）。

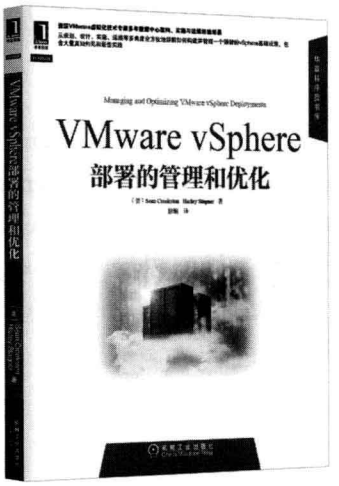
（2）Linux 内核档案网站（https://kernel.org/）。

（3）Linux 基金会（http://www.linuxfoundation.org/）。

（4）KVM Autotest 的文档（https://github.com/autotest/virt-test/wiki/）。

推荐阅读

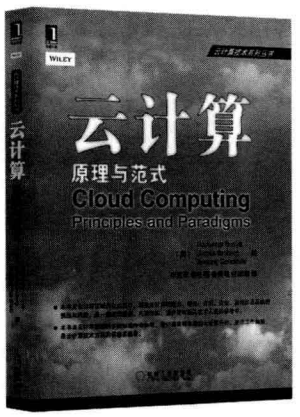

VMware vSphere 5: Building a Virtual Datacenter
VMware vSphere 5
虚拟数据中心构建指南

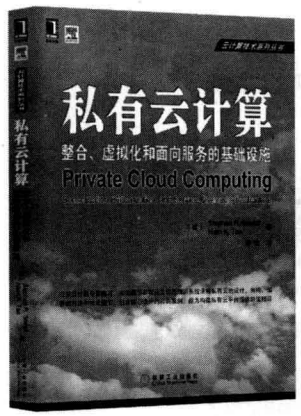

Managing and Optimizing VMware vSphere Deployments
VMware vSphere
部署的管理和优化

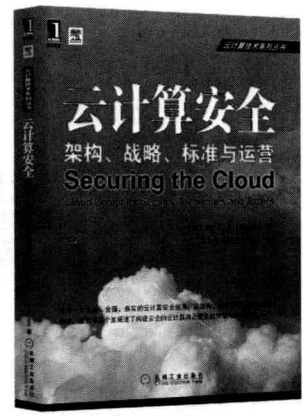

VMware、Citrix和Microsoft
虚拟化技术详解与应用实践


虚拟化技术丛书
虚拟化技术指南
irtualization: A Beginner's Guide

云计算系列丛书







云计算：原理与范式

作者：（澳）Rajkumar Buyya 等编 ISBN：978-7-111-41733-0 定价：99.00元

私有云计算：整合、虚拟化和面向服务的基础设施

作者：（美）Stephen R. Smoot 等 ISBN：978-7-111-40481-1 定价：69.00元

云计算安全：架构、战略、标准与运营

作者：（美）Vic (J. R.) Winkler ISBN：978-7-111-40139-1 定价：59.00元

云计算架构：解决方案设计手册

作者：（美）John Rhoton 等 ISBN：978-7-111-39056-5 定价：69.00元

云计算揭秘：企业实施云计算的核心问题

作者：（美）Jothy Rosenberg 等 ISBN：978-7-111-38494-6 定价：59.00元

云计算：企业实施手册

作者：John Rhoton ISBN：978-7-111-35177-1 定价：49.00元

云计算安全与隐私

作者：Tim Mather 等 ISBN：978-7-111-34525-1 定价：65.00元

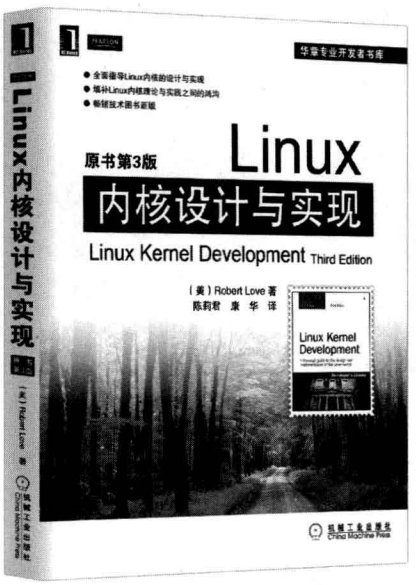
云计算实践指南

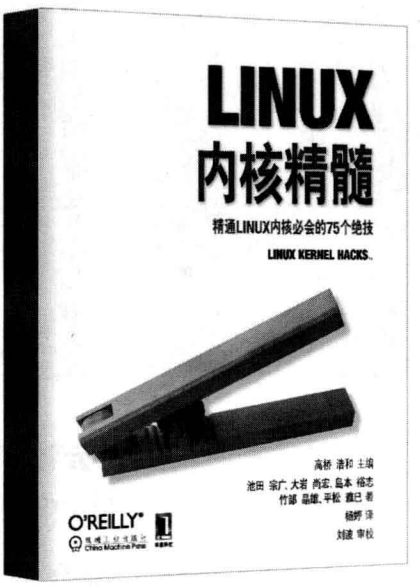
作者：Toby Velte 等 ISBN：978-7-111-30531-6 定价：45.00元

云计算：实现、管理与安全

作者：John Rittinghouse 等 ISBN：978-7-111-30481-4 定价：39.00元

推荐阅读





Linux内核设计与实现（原书第3版）

世界范围内公认的Linux内核经典著作，畅销全球多个国家

Linux内核精髓

畅销书，一线内核技术专家经验和智慧结晶，深刻解读Linux内核的资源管理、文件系统、网络、虚拟化、省电技术、调试、性能调优、分析与追踪等核心主题

實戰

51CTO.com
技术成就梦想 联合策划

经过几年的发展，云计算服务逐渐成为各行业实现信息化的基础性服务，而虚拟化则是云计算的基础技术。目前，商业化的虚拟化产品和技术几乎被VMware、Citrix和Microsoft三家公司垄断，开源的虚拟化技术则逐渐呈现出KVM一家独大的趋势。从2007年2月Linux 2.6.20内核加入KVM模块至今，KVM受到了以Intel、Redhat、IBM、Novell、Siemens、华为、Google为代表的来自全球的大量软件公司、硬件公司和应用型公司的大力支持，从而在软件、硬件和应用层面发展非常迅速，形成了一个成熟的生态链，成为企业进行系统虚拟化和云平台建设的重要选择。

Intel为KVM虚拟化技术的发展和普及做出了重要的贡献，本书的两位作者来自于Intel虚拟化技术小组，专门从事KVM等虚拟化技术的研究和实践，是国内KVM虚拟化技术领域的先驱者，对KVM的应用和原理都有深刻的认识。本书不仅系统讲解了KVM虚拟机的功能、特性和使用方法，还深入剖析KVM虚拟机的核心技术和原理。

本书主要内容如下：

- 云计算和虚拟化技术的概念性介绍，主流虚拟化方案的介绍和比较，以及KVM的功能介绍；
- KVM的基本原理剖析；
- KVM工作环境的搭建；
- KVM的核心基础功能的系统，包括CPU配置、内存配置、网络配置、图形显示等；
- KVM的高级功能，包括半虚拟化驱动、设备直接分配、热插拔、动态迁移、嵌套虚拟化、KSM技术、安全性、QEMU监控器等；
- KVM的管理工具，如libvirt、virsh、virt-manager、virt-viewer、virtinstall、virt-top等；
- 各个流行的Linux发行版中的KVM虚拟化功能的使用；
- KVM的性能测试及参考数据；
- KVM开源社区的情况，代码结构，以及如何向KVM贡献代码等。

上架指导：计算机/操作系统
ISBN 978-7-111-43900-4
9 787111 439004 >
定价：79.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604
数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com