

目 录

目 录.....	I
第 1 章 概述.....	1
第 1.1 节 关于 OpenFlow	1
第 1.2 节 关于本文.....	2
第 2 章 Openflow	3
第 2.1 节 概述.....	3
第 2.2 节 交换机组成.....	3
第 2.3 节 流表.....	3
2.3.1 包头域.....	4
2.3.2 计数器（counter）	5
2.3.3 行动（action）	6
2.3.4 匹配.....	8
第 2.4 节 安全通道.....	9
2.4.1 of 协议	9
2.4.2 连接建立.....	10
2.4.3 连接中断.....	10
2.4.4 加密	10
2.4.5 生成树.....	10
2.4.6 流表修改.....	11
2.4.7 流超时.....	12
第 2.5 节 of 协议	12
2.5.1 of 协议头	12
2.5.2 常用数据结构.....	14
2.5.3 Controller-to-Switch 消息	23
2.5.4 Asynchronous 消息	35
2.5.5 Symmetric 消息	40
第 2.6 节 规范 1.1 更新内容.....	Error! Bookmark not defined.
第 3 章 OpenvSwitch.....	42
第 3.1 节 概述.....	42
第 3.2 节 特性.....	42
第 3.3 节 代码.....	42
第 3.4 节 命令.....	43
第 4 章 NOX	44
第 4.1 节 网络操作系统.....	44

第 4.2 节	模型.....	44
第 4.3 节	架构.....	45
4.3.1	组件.....	45
4.3.2	操作.....	45
4.3.3	多粒度处理.....	46
4.3.4	开发实现.....	46
第 4.4 节	安装.....	47
4.4.1	步骤.....	47
4.4.2	依赖.....	48
4.4.3	选项.....	48
4.4.4	校验.....	49
第 4.5 节	应用.....	49
4.5.1	框架.....	49
4.5.2	运行与接口.....	50
4.5.3	例程.....	50
第 4.6 节	开发.....	51
4.6.1	组件.....	51
4.6.2	事件.....	54
4.6.3	开发例程.....	57
第 4.7 节	GUI.....	58
4.7.1	运行 GUI.....	58
4.7.2	扩展 GUI.....	59
第 4.8 节	相关工作.....	61
第 5 章 Mininet.....		62
第 5.1 节	概述.....	62
第 5.2 节	主要特性.....	62
第 5.3 节	镜像获取和使用.....	62
5.3.1	获取镜像.....	62
5.3.2	使用镜像.....	63
5.3.3	更新.....	63
第 5.4 节	简单测试.....	63
5.4.1	创建网络.....	63
5.4.2	查看信息.....	64
5.4.3	对节点进行单独操作.....	64
第 5.5 节	常用操作.....	65
5.5.1	快捷测试.....	65
5.5.2	自定义拓扑.....	65
5.5.3	使用友好的 mac 编号.....	66
5.5.4	使用 XTerm.....	66
5.5.5	链路操作.....	67

5.5.6	指定交换机跟控制器类型	67
5.5.7	名字空间	67
5.5.8	启动参数总结	68
5.5.9	常用命令总结	68
5.5.10	其他操作	69
第 5.6 节	高级功能	69
5.6.1	dpctl	69
5.6.2	控制器	70
5.6.3	交换机与控制器交互	70
5.6.4	使用 NOX	71
5.6.5	多条配置命令	72
第 5.7 节	代码分析	72
5.7.1	bin 子目录	72
5.7.2	mininet 子目录	73
5.7.3	custom 子目录	74
5.7.4	examples 子目录	74
5.7.5	其他文件	75
第 6 章 相关项目		77
第 6.1 节	Openflow	77
第 6.2 节	OpenvSwitch	77
第 6.3 节	NOX	77
第 6.4 节	Mininet	77

第1章 概述

坚持创新，我们的研究才有意思，才有意义。本段的目的是试图告诉大家，OpenFlow 为我们提供了更多更好的创新机会。

第1.1节 关于 OpenFlow

最初，OpenFlow 以交换机的形式出现在 Stanford 一伙人的实验室里，后来又部署到了他们计算机系的 Gates Building 里。OpenFlow 使传统的二层和三层交换机具备了细粒度流转发能力，即传统的基于 MAC 的网包转发，基于 IP 的路由转发，被拓展到了基于多域网包包头描述的流转发。同时，传统的控制层面从转发设备中剥离出来，所有转发行为的决策从交换机自身“迁移”到了某个集中控制器上。

随着 OpenFlow 的部署和应用，支持 OF 的交换设备的性能瓶颈相继出现。于是有了基于 NetFPGA 的实现，提供多千兆的 OF 交换。工业界如 HP，Juniper，NEC 等也相继提供了支持 OF 协议的交换机设备。值得注意的是，在交换机吞吐不是问题的情况下，新建连接速度（受 controller 限制）和流表大小（受 TCAM 限制）始终困扰 OpenFlow 发展。

稍后，流表大小的问题通过两个思路解决，一是使用 multi-table（OF 1.1 支持），通过 pipeline 的流表查找来解决指数及增长的流表项数；二是使用 Proactive+Reactive 的双重流表建立方法，通过分布查找+动态加载减少流表项数（DIFANE）。虽然这两种方案一定程度上减轻了流表大小的问题，但离问题最终解决还早。OpenFlow 是否支持大规模网络，需要进一步研究。

与此同时，集中控制端的网络操作系统的发展也在不断推进。最早的 Controller 仅用于校园内大楼里的，之后的 Ethane/Nox 拓展到了企业网和数据中心的范畴。近期提出的 Onix 完成了一套 Internet-scale 的 OpenFlow 部署方案。Onix 的管控粒度在不同 scale 下面有所不同：在数据中心汇聚层以及 Internet 核心层的 OF 功能越来越接近现有路由设备，而在设备接入层（服务器，用户）和网络接入（网关）层则保留基于 OF 的细粒度控制。

从 OF 的演进可以看出，创新与妥协的 trade-off 贯穿始终。一方面，OF 创新性地将路由和网关设备的数据平面推向网络，并将管理平面迁移整合到集中控制器，从而以分布式处理和集中式控制简化网络管理的难度，增强网络的可用性。另一方面，由于受到软硬件技术的约束和产业模式的不成熟，OF 的发展始终对现有网络进行各种各样的妥协。随着学界和业界越来越多的接受 OF 理念，这种 trade-off 越来越可能得到收敛，进而推动产业的成熟。

事实上，在 Cisco 和 HP 大张旗鼓的争斗 VN-TAG 和 VEPA 之际，基于 OpenFlow 的虚拟交换产品 OvS 已于斜刺里杀出。这种 Software-Defined Network，无论功能和成本均有自己独特的优势。至于是否能对传统网络技术产生巨大的冲击则需要包括我们在内的所有人，继续创新。

第1.2节 关于本文

斯坦福大学 OpenFlow 团队近十年来一贯坚持的开放和创新的态度，使得基于 OF 的开源项目不断增加，一个个精彩的 demo 不断呈现，最终促成了学术界和工业界的集体参与热情。

由于 OpenFlow 创新的目标是简化网络管理，即 OpenFlow 自身的创新实际是在驱动网络业务的创新。因此，从事 OpenFlow 的相关研究，不应仅停留在 OF 网络的部署上，更重要的是如何利用 OF 网络去进行业务创新。我们相信只有合理的产业化方向，出色的业务和应用，才能最终推动 OpenFlow 的发展，最终让软件掌控网络。

撰写本文的目的，一方面是收集、整理、分享 OpenFlow 的相关技术资料，另一方面则是希望籍此推进国内的 OpenFlow 研究发展。本文的技术内容均来自公开发表的学术文章和相关论坛、网站等，附带了本团队的一些理解和体会。我们期望通过自己的一点点努力，促进国内团队的 OpenFlow 研究与开发，以及广泛的和开放的交流与创新。

第2章 Openflow

第2.1节 概述

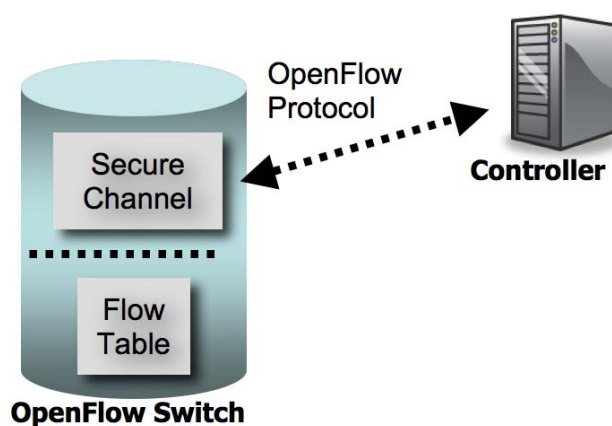
官方网站: <http://OpenFlowSwitch.org>。

本部分内容按照 Openflow 规范 1.0 版本撰写。1.0 之前版本都是草案，从 1.0 版本开始是正式版本，生产商们理论上应该都参照这个版本。1.0 版本的下载地址为 <http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf>。

目前最新规范版本为 1.1。在本章最后一节将说明规范 1.1 的修改内容。

第2.2节 交换机组成

每个 of 交换机（switch）都有一张流表，进行包查找和转发。交换机可以通过 of 协议经一个安全通道连接到外部控制器（controller），对流表进行查询和管理。图表 2-1 展示了这一过程。



图表 2-1 of 交换机通过安全通道连接到控制器

流表包括包头域（header fields，匹配包头多个域）、活动计数器（counters）、0 个或多个执行行动（actions）。对每一个包进行查找，如果匹配则执行相关策略，否则通过安全通道将包转发到控制器，控制器来决策相关行为。流表项可以将包转发到一个或者多个接口。

第2.3节 流表

流表是交换机进行转发策略控制的核心数据结构。交换芯片通过查找流表表项来决策

对进入交换机的网络流量采取合适的行为。

每个表项包括三个域，包头域（header field），计数器（counters），行动（actions）。如表格 2-1 所示。

表格 2-1 流表项结构

Head Fileds	Counter	Actions
-------------	---------	---------

2.3.1 包头域

包头域包括 12 个域，如表格 2-2 所示，包括：进入接口，Ethernet 源地址、目标地址、类型，vlan id，vlan 优先级，IP 源地址、目标地址、协议、IP ToS 位，TCP/UDP 目标端口、源端口。每一个域包括一个确定值或者所有值（any），更准确的匹配可以通过掩码实现。

表格 2-2 流表项的包头域

Ingress Port	Ether Source	Ether Dst	Ether Type	Vlan id	Vlan Priority	IP src	IP dst	IP proto	IP ToS bits	TCP/UDP Src Port	TCP/UDP Dst Port
--------------	--------------	-----------	------------	---------	---------------	--------	--------	----------	-------------	------------------	------------------

更具体的各个域的解释参见表格 2-3。

表格 2-3 包头域详细含义

Field	Bits	When applicable	Notes
Ingress Port	(Implementation dependent)	All packets	Numerical representation of incoming port, starting at 1.
Ethernet source address	48	All packets on enabled ports	
Ethernet destination address	48	All packets on enabled ports	
Ethernet type	16	All packets on enabled ports	An OpenFlow switch is required to match the type in both standard Ethernet and 802.2 with a SNAP header and OUI of 0x000000. The special value of 0x05FF is used to match all 802.3 packets without SNAP headers.
VLAN id	12	All packets of Ethernet type 0x8100	
VLAN priority	3	All packets of Ethernet type 0x8100	VLAN PCP field
IP source address	32	All IP and ARP packets	Can be subnetmasked
IP destination address	32	All IP and ARP packets	Can be subnetmasked
IP protocol	8	All IP and IP over Ethernet, ARP packets	Only the lower 8 bits of the ARP opcode are used

IP ToS bits	6	All IP packets	Specify as 8-bit value and place ToS in upper 6 bits.
Transport source port / ICMP Type	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Type
Transport destination port / ICMP Code	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Code

2.3.2 计数器（counter）

计数器可以针对每张表、每个流、每个端口、每个队列来维护。用来统计流量的一些信息，例如活动表项、查找次数、发送包数等。统计信息所需要的计数器在表格 2-4 中给出。

表格 2-4 统计信息需要的计数器

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

2.3.3 行动 (action)

每个表项对应到 0 个或者多个行动，如果没有转发行动，则默认丢弃。多个行动的执行需要依照优先级顺序依次进行。但对包的发送不保证顺序。另外交换机可以对不支持的行动返回错误 (unsupported flow error)。

行动可以分为两种类型：必备行动 (Required Actions) 和可选行动 (Optional Actions)，必备行动是默认支持的，交换机需要通知控制器它支持的可选行动。

2.3.3.1 必备行动

必备行动-转发 (Forward)

- ALL 转发到所有出口 (不包括入口)
- CONTROLLER 封装并转发给控制器
- LOCAL 转发给本地网络栈
- TABLE 对要发出的包执行流表中的行动
- IN_PORT 从入口发出

必备行动-丢弃 (Drop)

没有明确指明处理行动的表项，所匹配的所有网包默认丢弃。

2.3.3.2 可选行动

可选行动-转发

- NORMAL 按照传统交换机的 2 层或 3 层进行转发处理。
- FLOOD 通过最小生成树从出口泛洪发出，注意不包括入口。

可选行动-入队 (Enqueue)

将包转发到绑定到某个端口的队列中。

可选行动-修改域 (Modify-field)

修改包头内容。具体的行为见表格 2-5。

表格 2-5 修改域行为

Action	Associated Data	Description
Set VLAN ID	12 bits	If no VLAN is present, a new header is added with the specified VLAN ID and priority of zero. If a VLAN header already exists, the VLAN ID is re-placed with the specified value.
Set VLAN priority	3 bits	If no VLAN is present, a new header is added with the specified priority and a VLAN ID of zero. If a VLAN header already exists, the priority field is replaced with the specified value.
Strip VLAN header	-	Strip VLAN header if present.

Modify Ethernet source MAC address	48 bits: Value with which to replace existing source MAC address	Replace the existing Ethernet source MAC address with the new value
Modify Ethernet destination MAC address	48 bits: Value with which to replace existing destination MAC address	Replace the existing Ethernet destination MAC address with the new value.
Modify IPv4 source address	32 bits: Value with which to replace existing IPv4 source address	Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applicable to IPv4 packets.
Modify IPv4 destination address	32 bits: Value with which to replace existing IPv4 destination address	Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applied to IPv4 packets.
Modify IPv4 ToS bits	6 bits: Value with which to replace existing IPv4 ToS field	Replace the existing IP ToS field. This action is only applied to IPv4 packets.
Modify transport source port	16 bits: Value with which to replace existing TCP or UDP source port	Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum. This action is only applicable to TCP and UDP packets.
Modify transport destination port	16 bits: Value with which to replace existing TCP or UDP destination port	Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum This action is only applied to TCP and UDP packets.

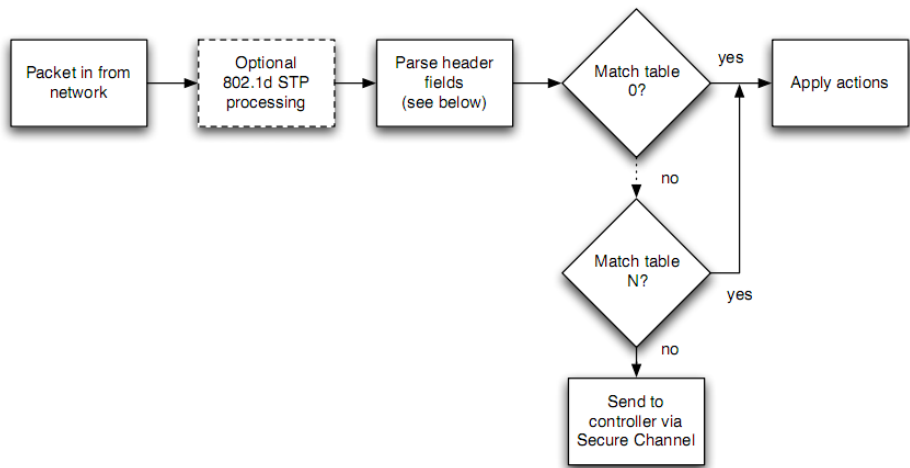
2.3.3.3 交换机类型

通过支持的行为类型不同，兼容 of 的交换机分为两类，一类是“纯 of 交换机”(of-only)，一类是“支持 of 交换机”(of-enable)。前者仅需要支持必备行动，后者还可以支持 NORMAL 行动，同时，双方都可已支持泛洪行动（Flood Action）。

表格 2-6 各种类型 of 交换机的支持行动

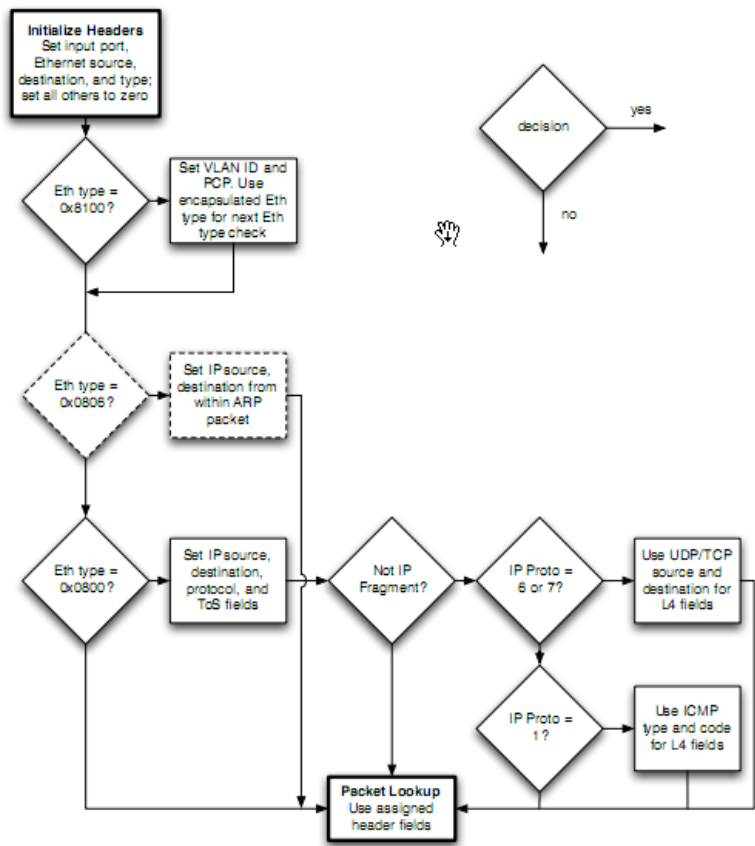
ACTION	of-only	of-enable
Required Actions	YES	YES
NORMAL	NO	CAN
FLOOD	CAN	CAN

2.3.4 匹配



图表 2-2 整体匹配流程

每个包按照优先级依次去匹配流表中表项，匹配包的优先级最高的表项即为匹配结果。一旦匹配成功，对应的计数器将更新；如果没能找到匹配的表项，则转发给控制器。整体流程参见图表 2-2，具体包头解析匹配过程见图表 2-3。



图表 2-3 包头解析的匹配流程

第2.4节 安全通道

安全通道用来连接交换机和控制器，所有安全通道必须遵守 of 协议。控制器可以配置、管理交换机、接收交换机的事件信息，并通过交换机发出网包等。

2.4.1 of 协议

of 协议支持三种消息类型：controller-to-switch，asynchronous（异步）和 symmetric（对称），每一类消息又有多个子消息类型。controller-to-switch 消息由控制器发起，用来管理或获取 switch 状态；asynchronous 消息由 switch 发起，用来将网络事件或交换机状态变化更新到控制器；symmetric 消息可由交换机或控制器发起。

2.4.1.1 controller-to-switch 消息

由控制器（controller）发起，可能需要或不需要来自交换机的应答消息。包括 Features、Configuration、Modify-state、Read-state、Send-packet、Barrier 等。

Features

在建立传输层安全会话（Transport Layer Security Session）的时候，控制器发送 feature 请求消息给交换机，交换机需要应答自身支持的功能。

Configuration

控制器设置或查询交换机上的配置信息。交换机仅需要应答查询消息。

Modify-state

控制器管理交换机流表项和端口状态等。

Read-state

控制器向交换机请求一些诸如流、网包等统计信息。

Send-packet

控制器通过交换机指定端口发出网包。

Barrier

控制器确保消息依赖满足，或接收完成操作的通知

2.4.1.2 asynchronous 消息

asynchronous 不需要控制器请求发起，主要用于交换机向控制器通知状态变化等事件信息。主要消息包括 Packet-in、Flow-removed、Port-status、Error 等。

Packet-in

交换机收到一个网包，在流表中没有匹配项，则发送 Packet-in 消息给控制器。如果交换机缓存足够多，网包被临时放在缓存中，网包的部分内容（默认 128 字节）和在交换机缓存中的序号也一同发给控制器；如果交换机缓存不足以存储网包，则将整个网包作为消息的附带内容发给控制器。

Flow-removed

交换机中的流表项因为超时或修改等原因被删除掉，会触发 Flow-removed 消息。

Port-status

交换机端口状态发生变化时（例如 down 掉），触发 Port-status 消息。

Error

交换机通过 Error 消息来通知控制器发生的问题。

2.4.1.3 symmetric 消息

symmetric 消息也不必通过请求建立，包括 Hello、Echo、Vendor 等。

Hello

交换机和控制器用来建立连接。

Echo

交换机和控制器均可以向对方发出 Echo 消息，接收者则需要回复 Echo reply。该消息用来测量延迟、是否连接保持等。

Vendor

交换机提供额外的附加信息功能。为未来版本预留。

2.4.2 连接建立

通过安全通道建立连接，所有流量都不经过交换机流表检查。因此交换机必须将安全通道认为是本地链接。今后版本中将介绍动态发现控制器的协议。

当 of 连接建立起来后，两边必须先发送 OFPT_HELLO 消息给对方，该消息携带支持的最高协议版本号，接受方将采用双方都支持的最低协议版本进行通信。一旦发现两者拥有共同支持的协议版本，则连接建立，否则发送 OFPT_ERROR 消息（类型为 OFPET_HELLO_FAILED，代码为 OFPHFC_COMPATIBLE），描述失败原因，并终止连接。

2.4.3 连接中断

当连接发生异常时，交换机应尝试连接备份的控制器。当多次尝试均失败后，交换机将进入紧急模式，并重置所有的 TCP 连接。此时，所有包将匹配指定的紧急模式表项，其他所有正常表项将从流表中删除。此外，当交换机刚启动时，默认进入紧急模式。

2.4.4 加密

安全通道采用 TLS（Transport Layer Security）连接加密。当交换机启动时，尝试连接到控制器的 6633 TCP 端口。双方通过交换证书进行认证。因此，每个交换机至少需配置两个证书，一个是用来认证控制器，一个用来向控制器发出认证。

2.4.5 生成树

交换机可以选择支持 802.1D 生成树协议。如果支持，所有相关包在查找流表之前应该先在本地进行传统处理。支持生成树协议的交换机在 OFPT_FEATURES_REPLY 消息的

compabilities 域需要设置 OFPC_STP 位，并且需要在所有的物理端口均支持生成树协议，但无需在虚拟端口支持。

生成树协议会设置端口状态，来限制发到 OFP_FLOOD 的网包仅被转发到生成树指定的端口。需要注意指定出口的转发或 OFP_ALL 的网包会忽略生成树指定的端口状态，按照规则设置端口转发。

如果交换机不支持 802.1D 生成树协议，则必须允许控制器指定泛洪时的端口状态。

2.4.6 流表修改

流表修改消息可以有以下类型：

```
enum ofp_flow_mod_command {
    OFPFC_ADD, /* New flow. */
    OFPFC_MODIFY, /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT, /* Modify entry strictly matching wildcards */
    OFPFC_DELETE, /* Delete all matching flows. */
    OFPFC_DELETE_STRICT /* Strictly match wildcards and priority. */
};
```

2.4.6.1 ADD

对于带有 OFPFF_CHECK_OVERLAP 标志的添加（ADD）消息，交换机将先检查新表项是否跟现有表项冲突（包头范围 overlap，且有相同的优先级），如果发现冲突，将拒绝添加，并返回 ofp_error_msg，并且指明 OFPET_FLOW_MOD_FAILED 类型和 OFPFMFC_OVERLAP 代码。

对于合法无冲突的添加，或不带 OFPFF_CHECK_OVERLAP 标志的添加，新表项将被添加到最低编号表中，优先级在匹配过程中获取。如果任何表中已经存在与新表项相同头部域和优先级的旧表项，该项将被新表项替代，同时计数器清零。如果交换机无法找到要添加的表，则返回 ofp_error_msg，并且指明 OFPET_FLOW_MOD_FAILED 类型和 OFPFMFC_ALL_TABLES_FULL 代码。

如果添加表项使用了交换机不合法的端口，则交换机返回 ofp_error_msg 消息，同时带有 OFPET_BAD_ACTION 类型和 OFPBAC_BAD_OUT_PORT 代码。

2.4.6.2 MODIFY

对于修改，如果所有已有表中没有与要修改表项同样头部域的表项，则等同于 ADD 消息，计数器置 0；否则更新现有表项的行为域，同时保持计数器、空闲时间均不变。

2.4.6.3 DELETE

对于删除，如果没有找到要删除表项，不发出任何消息；如果存在，则进行删除操作。如果被删除的表项带有 OFPFF_SEND_FLOW_REM 标志，则触发一条流删除的消息。删除紧急表项不触发消息。

此外，修改和删除还存在另一个 `_STRICT` 版本。对于非 `_STRICT` 版本，通配流表项是激活的，因此，所有匹配消息描述的流表项均受影响（包括包头范围被包含在消息表项中的流表项）。例如，一条所有域都是通配符的非 `_STRICT` 版本删除消息会清空流表，因为所有表项均包含在该表项中。

在 `_STRICT` 版本情况下，表项头跟优先级等都必须严格匹配才执行，即只有同一条表项会受影响。例如，一条所有域都是通配符的 `DELETE_STRICT` 消息仅删除指定优先级的某条规则。

此外删除消息还支持指定额外的 `out_port` 域。

如果交换机不能处理流操作消息指定的行为，则返回 `OFPET_FLOW_MOD_FAILED : OFPFMFC_UNSUPPORTED`，并拒绝该表项。

2.4.7 流超时

每个表项均有一个 `idle_timeout` 和一个 `hard_timeout`，前者计算没有流量匹配的时间（单位都是秒），后者计算被插入表中的时间。一旦到达时间期限，则交换机自动删除该表项，同时发出一个流删除的消息。

第2.5节 of 协议

包括 of 协议相关消息的数据结构等。

2.5.1 of 协议头

数据结构如下。

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version; /* OFP_VERSION. */
    uint8_t type; /* One of the OFPT_constants. */
    uint16_t length; /* Length including this ofp_header. */
    uint32_t xid;
    /* Transaction id associated with this packet.
    Replies use the same id as was in the request
    to facilitate pairing. */
};
OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

`Version` 用来标明 of 协议版本。当前 of 协议中，`version` 最重要的位用来标明实验版本，其他位标明修订版本。目前的版本是 `0x01`，最终的 `Type 0` 交换机应该是 `0x00`。

`Length` 用来标明消息长度。

`Type` 用来标明消息类型。可能的消息类型包括

```
enum ofp_type {
```

```

/* Immutable messages. */
OFPT_HELLO, /* Symmetric message */
OFPT_ERROR, /* Symmetric message */
OFPT_ECHO_REQUEST, /* Symmetric message */
OFPT_ECHO_REPLY, /* Symmetric message */
OFPT_VENDOR, /* Symmetric message */

/* Switch configuration messages. */ OFPT_FEATURES_REQUEST, /*
Controller/switch message */ OFPT_FEATURES_REPLY, /* Controller/switch message
*/ OFPT_GET_CONFIG_REQUEST, /* Controller/switch message */
OFPT_GET_CONFIG_REPLY, /* Controller/switch message */ OFPT_SET_CONFIG,
/* Controller/switch message */

/* Asynchronous messages. */
OFPT_PACKET_IN, /* Async message */
OFPT_FLOW_REMOVED, /* Async message */
OFPT_PORT_STATUS, /* Async message */

/* Controller command messages. */
OFPT_PACKET_OUT, /* Controller/switch message */
OFPT_FLOW_MOD, /* Controller/switch message */
OFPT_PORT_MOD, /* Controller/switch message */

/* Statistics messages. */
OFPT_STATS_REQUEST, /* Controller/switch message */
OFPT_STATS_REPLY, /* Controller/switch message */

/* Barrier messages. */
OFPT_BARRIER_REQUEST, /* Controller/switch message */
OFPT_BARRIER_REPLY, /* Controller/switch message */

/* Queue Configuration messages. */ OFPT_QUEUE_GET_CONFIG_REQUEST, /*
Controller/switch message */ OFPT_QUEUE_GET_CONFIG_REPLY /* Controller/switch
message */
};

```

2.5.2 常用数据结构

包括端口、队列、匹配、行动等。

2.5.2.1 端口

2.5.2.1.1 端口结构

```
/* Description of a physical port */
struct ofp_phy_port {
    uint16_t port_no;
    uint8_t hw_addr[OFP_ETH_ALEN];
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

    uint32_t config; /* Bitmap of OFPPC_* flags. */
    uint32_t state; /* Bitmap of OFPPS_* flags. */

    /* Bitmaps of OFPPF_* that describe features. All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr; /* Current features. */
    uint32_t advertised; /* Features being advertised by the port. */
    uint32_t supported; /* Features supported by the port. */
    uint32_t peer; /* Features advertised by peer. */
};
OFP_ASSERT(sizeof(struct ofp_phy_port) == 48);
```

port_no 标明绑定到物理接口的 datapath 值。

hw_addr 是该物理接口的 mac 地址。

OFP_MAX_ETH_ALEN 值为 6。

name 是该接口的名称字符串，以 null 结尾。

OFP_MAX_PORT_NAME_LEN 为 16。

2.5.2.1.2 Config

config 描述了生成树和管理设置，数据结构为

```
/* Flags to indicate behavior of the physical port. These flags are
 * used in ofp_phy_port to describe the current configuration. They are
 * used in the ofp_port_mod message to configure the port's behavior.
 */
enum ofp_port_config {
```

```

OFPPC_PORT_DOWN      = 1 << 0, /* Port is administratively down. */

OFPPC_NO_STP = 1 << 1, /* Disable 802.1D spanning tree on port. */
OFPPC_NO_RECV      = 1 << 2, /* Drop all packets except 802.1D spanning
tree packets. */
OFPPC_NO_RECV_STP   = 1 << 3, /* Drop received 802.1D STP packets. */
OFPPC_NO_FLOOD      = 1 << 4, /* Do not include this port when flooding. */
OFPPC_NO_FWD        = 1 << 5, /* Drop packets forwarded to port. */
OFPPC_NO_PACKET_IN  = 1 << 6 /* Do not send packet-in msgs for port. */
};

```

2.5.2.1.3 State

state 描述生成树状态和某个物理接口是否存在，数据结构为

```

/* Current state of the physical port. These are not configurable from
 * the controller.
 */
enum ofp_port_state {
OFPPS_LINK_DOWN = 1 << 0, /* No physical link present. */

/* The OFPPS_STP_* bits have no effect on switch operation. The
 * controller must adjust OFPPC_NO_RECV, OFPPC_NO_FWD, and
 * OFPPC_NO_PACKET_IN appropriately to fully implement an 802.1D spanning
 * tree. */
OFPPS_STP_LISTEN = 0 << 8, /* Not learning or relaying frames. */
OFPPS_STP_LEARN = 1 << 8, /* Learning but not relaying frames. */
OFPPS_STP_FORWARD = 2 << 8, /* Learning and relaying frames. */
OFPPS_STP_BLOCK = 3 << 8, /* Not part of spanning tree. */
OFPPS_STP_MASK = 3 << 8 /* Bit mask for OFPPS_STP_* values. */
};

```

2.5.2.1.4 端口号

端口号采用如下的数据结构。

```

/* Port numbering. Physical ports are numbered starting from 1. */
enum ofp_port {
/* Maximum number of physical switch ports. */
OFPP_MAX = 0xff00,

/* Fake output "ports". */
};

```

```

OFPP_IN_PORT = 0xffff8, /* Send the packet out the input port. This virtual
port must be explicitly used in order to send back out of the input port. */

OFPP_TABLE = 0xffff9, /* Perform actions in flow table.
NB: This can only be the destination
port for packet-out messages. */
OFPP_NORMAL = 0xffffa, /* Process with normal L2/L3 switching. */
OFPP_FLOOD = 0xffffb, /* All physical ports except input port and
those disabled by STP. */
OFPP_ALL = 0xffffc, /* All physical ports except input port. */
OFPP_CONTROLLER = 0xffffd, /* Send to controller. */
OFPP_LOCAL = 0xffffe, /* Local openflow "port". */
OFPP_NONE = 0xfffff /* Not associated with a physical port. */
};

```

curr, advertised, supported 和 peer 域 标明 链路模式 (10M 到 10G, 全双工、半双工), 链路类型 (铜线/光线) 和链路特性 (自动协商和暂停)。链路特性 (Port features) 数据结构如下, 多个标识可以同时设置。

```

/* Features of physical ports available in a datapath. */
enum ofp_port_features {
OFPPF_10MB_HD = 1 << 0, /* 10 Mb half-duplex rate support. */
OFPPF_10MB_FD = 1 << 1, /* 10 Mb full-duplex rate support. */
OFPPF_100MB_HD = 1 << 2, /* 100 Mb half-duplex rate support. */
OFPPF_100MB_FD = 1 << 3, /* 100 Mb full-duplex rate support. */
OFPPF_1GB_HD = 1 << 4, /* 1 Gb half-duplex rate support. */
OFPPF_1GB_FD = 1 << 5, /* 1 Gb full-duplex rate support. */
OFPPF_10GB_FD = 1 << 6, /* 10 Gb full-duplex rate support. */
OFPPF_COPPER = 1 << 7, /* Copper medium. */
OFPPF_FIBER = 1 << 8, /* Fiber medium. */
OFPPF_AUTONEG = 1 << 9, /* Auto-negotiation. */
OFPPF_PAUSE = 1 << 10, /* Pause. */
OFPPF_PAUSE_ASYM = 1 << 11 /* Asymmetric pause. */
};

```

2.5.2.2 队列

2.5.2.2.1 队列结构

队列被绑定到某个端口, 实现有限的流量控制操作。
数据结构为

```

/* Full description for a queue. */
struct ofp_packet_queue {
    uint32_t queue_id; /* id for the specific queue. */
    uint16_t len; /* Length in bytes of this queue desc. */
    uint8_t pad[2]; /* 64-bit alignment. */
    struct ofp_queue_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_packet_queue) == 8);

```

2.5.2.2.2 队列特性

每一个队列都带有一些特性，数据结构为

```

enum ofp_queue_properties {
    OFPQT_NONE = 0, /* No property defined for queue (default). */
    OFPQT_MIN_RATE, /* Minimum datarate guaranteed. */
    /* Other types should be added here
    * (i.e. max rate, precedence, etc). */
};

```

特性头的数据结构为

```

/* Common description for a queue. */
struct ofp_queue_prop_header {
    uint16_t property; /* One of OFPQT_. */
    uint16_t len; /* Length of property, including this header. */
    uint8_t pad[4]; /* 64-bit alignemnt. */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_header) == 8);

```

目前，实现了最小速率队列，数据结构为

```

/* Min-Rate queue property description. */
struct ofp_queue_prop_min_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MIN, len: 16. */
    /*
    uint16_t rate; /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[6]; /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_min_rate) == 16);

```

2.5.2.3 流匹配

2.5.2.3.1 流表项结构

描述一个流表项的数据结构为

```
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards; /* Wildcard fields. */
    uint16_t in_port; /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan; /* Input VLAN id. */
    uint8_t dl_vlan_pcp; /* Input VLAN priority. */
    uint8_t pad1[1]; /* Align to 64-bits */
    uint16_t dl_type; /* Ethernet frame type. */
    uint8_t nw_tos; /* IP ToS (actually DSCP field, 6 bits). */
    uint8_t nw_proto; /* IP protocol or lower 8 bits of
    * ARP opcode. */
    uint8_t pad2[2]; /* Align to 64-bits */
    uint32_t nw_src; /* IP source address. */
    uint32_t nw_dst; /* IP destination address. */
    uint16_t tp_src; /* TCP/UDP source port. */
    uint16_t tp_dst; /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

2.5.2.3.2 wildcards

wildcards 可以设置为如下的一些标志。

```
/* Flow wildcards. */
enum ofp_flow_wildcards {
    OFPFW_IN_PORT = 1 << 0, /* Switch input port. */ OFPFW_DL_VLAN =
1 << 1, /* VLAN id. */
    OFPFW_DL_SRC = 1 << 2, /* Ethernet source address. */ OFPFW_DL_DST = 1
<< 3, /* Ethernet destination address. */ OFPFW_DL_TYPE = 1 << 4, /*
Ethernet frame type. */ OFPFW_NW_PROTO = 1 << 5, /* IP protocol. */
    OFPFW_TP_SRC = 1 << 6, /* TCP/UDP source port. */ OFPFW_TP_DST = 1
<< 7, /* TCP/UDP destination port. */

    /* IP source address wildcard bit count. 0 is exact match, 1 ignores the
```

```

* LSB, 2 ignores the 2 least-significant bits, ..., 32 and higher wildcard
* the entire field. This is the *opposite* of the usual convention where
* e.g. /24 indicates that 8 bits (not 24 bits) are wildcarded. */
OFPFW_NW_SRC_SHIFT = 8,
OFPFW_NW_SRC_BITS = 6,
OFPFW_NW_SRC_MASK = ((1 << OFPFW_NW_SRC_BITS) - 1) <<
OFPFW_NW_SRC_SHIFT,
OFPFW_NW_SRC_ALL = 32 << OFPFW_NW_SRC_SHIFT,

/* IP destination address wildcard bit count. Same format as source. */
OFPFW_NW_DST_SHIFT = 14,
OFPFW_NW_DST_BITS = 6,
OFPFW_NW_DST_MASK = ((1 << OFPFW_NW_DST_BITS) - 1) <<
OFPFW_NW_DST_SHIFT,
OFPFW_NW_DST_ALL = 32 << OFPFW_NW_DST_SHIFT,

OFPFW_DL_VLAN_PCP = 1 << 20, /* VLAN priority. */ OFPFW_NW_TOS = 1
<< 21, /* IP ToS (DSCP field, 6 bits). */

/* Wildcard all fields. */ OFPFW_ALL = ((1 << 22) - 1)
};

```

如果 wildcards 没有设置，则 ofp_match 精确的标明流表项。注意源和目的的掩码在 wildcards 中用 6 位来表示。其含义与 CIDR 相反，表示低位的多少位被掩掉。例如在 CIDR 中掩码 24 意味着 255.255.255.0，而在 of 中，24 意味着 255.0.0.0。

2.5.2.4 行为

2.5.2.4.1 行为类型

目前，行为类型包括

```

enum ofp_action_type {
OFPAT_OUTPUT, /* Output to switch port. */
OFPAT_SET_VLAN_VID, /* Set the 802.1q VLAN id. */
OFPAT_SET_VLAN_PCP, /* Set the 802.1q priority. */
OFPAT_STRIP_VLAN, /* Strip the 802.1q header. */
OFPAT_SET_DL_SRC, /* Ethernet source address. */

```

```

OFPAT_SET_DL_DST, /* Ethernet destination address. */ OFPAT_SET_NW_SRC,
/* IP source address. */ OFPAT_SET_NW_DST, /* IP destination address.
*/ OFPAT_SET_NW_TOS, /* IP ToS (DSCP field, 6 bits). */ OFPAT_SET_TP_SRC,
/* TCP/UDP source port. */ OFPAT_SET_TP_DST, /* TCP/UDP destination
port. */ OFPAT_ENQUEUE, /* Output to queue. */
OFPAT_VENDOR = 0xffff
};

```

2.5.2.4.2 行为结构

一个行为应该包括类型、长度和相应的数据。

```

/* Action header that is common to all actions. The length includes the
* header and any padding used to make the action 64-bit aligned.
* NB: The length of an action *must* always be a multiple of eight. */
struct ofp_action_header {
uint16_t type; /* One of OFPAT_*. */
uint16_t len; /* Length of action, including this
header. This is the length of action,
including any padding to make it
64-bit aligned. */
uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_action_header) == 8);

```

2.5.2.4.3 行为输出

包括如下数据结构

```

/* Action structure for OFPAT_OUTPUT, which sends packets out 'port' .
* When the 'port' is the OFPP_CONTROLLER, 'max_len' indicates the max
* number of bytes to send. A 'max_len' of zero means no bytes of the
* packet should be sent.*/
struct ofp_action_output {
uint16_t type; /* OFPAT_OUTPUT. */
uint16_t len; /* Length is 8. */
uint16_t port; /* Output port. */
uint16_t max_len; /* Max length to send to controller. */
};
OFP_ASSERT(sizeof(struct ofp_action_output) == 8);

```

其中，max_len 指明当端口为 OFPP_CONTROLLER 时，发送到控制器的包内容大

小。如果为 0，则发送一个长度为 0 的 packet_in 消息。port 指明发出的物理端口。

2.5.2.4.4 入队操作

入队(Enqueue)操作将流绑定到某个已配置好的队列中，而不管 TOS 和 VLAN PCP 位。包在入队操作后不应该被修改。如有必要，原始信息可以先备份出来，在网包发出的时候还原信息。如果交换机需要通过 TOS 和 VLAN PCP 位来进行队列操作，则用户需要指定这些位来让流绑定到合适的队列上。

入队操作包括如下的域

```
/* OFPAT_ENQUEUE action struct: send packets to given queue on port. */
struct ofp_action_enqueue {
    uint16_t type; /* OFPAT_ENQUEUE. */
    uint16_t len; /* Len is 16. */
    uint16_t port; /* Port that queue belongs. Should
refer to a valid physical port
(i.e. < OFPP_MAX) or OFPP_IN_PORT. */
    uint8_t pad[6]; /* Pad for 64-bit alignment. */
    uint32_t queue_id; /* Where to enqueue the packets. */
};
OFP_ASSERT(sizeof(struct ofp_action_enqueue) == 16);
```

action_vlan_vid 包括如下的域

```
/* Action structure for OFPAT_SET_VLAN_VID. */
struct ofp_action_vlan_vid {
    uint16_t type; /* OFPAT_SET_VLAN_VID. */
    uint16_t len; /* Length is 8. */
    uint16_t vlan_vid; /* VLAN id. */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_vid) == 8);
```

vlan_vid field 有 16 位，而实际的 VLAN id 仅有 12 位。0xffff 用来标明没有设置 VLAN id。

An action_vlan_pcp has the following fields:

```
/* Action structure for OFPAT_SET_VLAN_PCP. */
struct ofp_action_vlan_pcp {
    uint16_t type; /* OFPAT_SET_VLAN_PCP. */
    uint16_t len; /* Length is 8. */
    uint8_t vlan_pcp; /* VLAN priority. */
    uint8_t pad[3];
```

```
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_pcp) == 8);
```

vlan_pcp field 有 8 位长，但仅有低 3 位有意义。

action_strip_vlan 没有参数，仅包括一个通用的 ofp_action_header。该行为会剥掉 VLAN tag（如果存在）。

action_dl_addr 包括如下的域

```
/* Action structure for OFPAT_SET_DL_SRC/DST. */
struct ofp_action_dl_addr {
    uint16_t type; /* OFPAT_SET_DL_SRC/DST. */
    uint16_t len; /* Length is 16. */
    uint8_t dl_addr[OF_ETH_ALEN]; /* Ethernet address. */
    uint8_t pad[6];
};
OFP_ASSERT(sizeof(struct ofp_action_dl_addr) == 16);
```

其中，dl_addr field 是要设置的 mac 地址。

action_nw_addr 包括如下的域

```
/* Action structure for OFPAT_SET_NW_SRC/DST. */
struct ofp_action_nw_addr {
    uint16_t type; /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len; /* Length is 8. */
    uint32_t nw_addr; /* IP address. */
};
OFP_ASSERT(sizeof(struct ofp_action_nw_addr) == 8);
```

其中，nw_addr field 是要设置的 IP 地址。

action_nw_tos 包括如下的域

```
/* Action structure for OFPAT_SET_NW_TOS. */
struct ofp_action_nw_tos {
    uint16_t type; /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len; /* Length is 8. */
    uint8_t nw_tos; /* IP ToS (DSCP field, 6 bits). */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_tos) == 8);
```

nw_tos 域是要设置的 ToS 的高 6 位。

action_tp_port 包括如下的域

```
/* Action structure for OFPAT_SET_TP_SRC/DST. */
```

```

struct ofp_action_tp_port {
    uint16_t type; /* OFPAT_SET_TP_SRC/DST. */
    uint16_t len; /* Length is 8. */
    uint16_t tp_port; /* TCP/UDP port. */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_tp_port) == 8);

```

tp_port 域是要设置的 TCP/UDP/其它端口。

```

An action_vendor has the following fields:
/* Action header for OFPAT_VENDOR. The rest of the body is vendor-defined. */
struct ofp_action_vendor_header {
    uint16_t type; /* OFPAT_VENDOR. */
    uint16_t len; /* Length is a multiple of 8. */
    uint32_t vendor; /* Vendor ID, which takes the same form
as in "struct ofp_vendor_header". */
};
OFP_ASSERT(sizeof(struct ofp_action_vendor_header) == 8);

```

vendor 域是 Vendor 的 ID，跟结构 ofp_vendor 中一致。

2.5.3 Controller-to-Switch 消息

包括握手、配置交换机、修改状态、配置队列、读取状态、发包、保障等。

2.5.3.1 握手

TLS 连接建立之初，控制器发送一个仅有消息头的 OFPT_FEATURES_REQUEST 消息，交换机返回一个 OFPT_FEATURES_REPLY 消息。

```

/* Switch features. */
struct ofp_switch_features {
    struct ofp_header header;
    uint64_t datapath_id; /* Datapath unique ID. The lower 48-bits are for
a MAC address, while the upper 16-bits are
implementer-defined. */
    uint32_t n_buffers; /* Max packets buffered at once. */

    uint8_t n_tables; /* Number of tables supported by datapath. */
    uint8_t pad[3]; /* Align to 64-bits. */

    /* Features. */

```

```

uint32_t capabilities; /* Bitmap of support "ofp_capabilities". */
uint32_t actions; /* Bitmap of supported "ofp_action_type"s. */

/* Port info.*/
struct ofp_phy_port ports[0]; /* Port definitions. The number of ports
is inferred from the length field in
the header. */
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 32);

```

其中，datapath_id 是 datapath 的唯一标识，其中低 48 位设计为交换机 mac 地址，高 16 位由实现者来定义。

n_tables 域指明交换机支持的流表个数。

capabilities 域使用下面的标志。

```

/* Capabilities supported by the datapath. */
enum ofp_capabilities {
OFP_FLOW_STATS = 1 << 0, /* Flow statistics. */
OFP_TABLE_STATS = 1 << 1, /* Table statistics. */
OFP_PORT_STATS = 1 << 2, /* Port statistics. */
OFP_STP = 1 << 3, /* 802.1d spanning tree. */
OFP_RESERVED = 1 << 4, /* Reserved, must be zero. */
OFP_IP_REASM = 1 << 5, /* Can reassemble IP fragments. */
OFP_QUEUE_STATS = 1 << 6, /* Queue statistics. */
OFP_ARP_MATCH_IP = 1 << 7 /* Match IP addresses in ARP pkts. */
};

```

actions 是用来标志支持行动的 bit 串。

ports 是一个 ofp_phy_port 结构数组，来描绘所有支持 of 的交换机端口。

2.5.3.2 配置交换机

控制器通过向交换机发送 OFPT_SET_CONFIG 和 OFPT_GET_CONFIG_REQUEST 消息（该消息仅有头部）来配置和查询交换机配置。对于查询消息，交换机必须回复 OFPT_GET_CONFIG_REPLY 消息。

配置和查询消息结构如下

```

/* Switch configuration. */
struct ofp_switch_config {
struct ofp_header header;
uint16_t flags; /* OFPC_* flags. */
uint16_t miss_send_len; /* Max bytes of new flow that datapath should
send to the controller. */
};

```

```
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

配置标志包括

```
enum ofp_config_flags {
/* Handling of IP fragments. */
OFP_CFG_FRAG_NORMAL = 0, /* No special handling for fragments. */
OFP_CFG_FRAG_DROP = 1, /* Drop fragments. */
OFP_CFG_FRAG_REASM = 2, /* Reassemble (only if OFP_CFG_IP_REASM set). */
OFP_CFG_FRAG_MASK = 3
}
```

OFP_CFG_FRAG_*标志用来指示该如何处理 IP 碎片：正常、丢弃还是重组。

miss_send_len 用来指示当网包在交换机中不匹配或者匹配结果是发到控制器的时
候，该发送多少数据给控制器。如果为 0，则发送长度为 0 的 packet_in 消息到控制器。

2.5.3.3 修改状态

控制器修改流表需要通过 OFPT_FLOW_MOD 消息。

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
struct ofp_header header;
struct ofp_match match; /* Fields to match */
uint64_t cookie; /* Opaque controller-issued identifier. */

/* Flow actions. */
uint16_t command; /* One of OFPFC_*. */
uint16_t idle_timeout; /* Idle time before discarding (seconds). */
uint16_t hard_timeout; /* Max time before discarding (seconds). */
uint16_t priority; /* Priority level of flow entry. */
uint32_t buffer_id; /* Buffered packet to apply to (or -1).
Not meaningful for OFPFC_DELETE*. */
uint16_t out_port; /* For OFPFC_DELETE* commands, require
matching entries to include this as an
output port. A value of OFPP_NONE
indicates no restriction. */
uint16_t flags; /* One of OFPFF_*. */
struct ofp_action_header actions[0]; /* The action length is inferred
from the length field in the
header. */
};
```

```
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 72);
```

command 域必须为下面的类型之一。

```
enum ofp_flow_mod_command {

    OFPFC_ADD, /* New flow. */
    OFPFC_MODIFY, /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT, /* Modify entry strictly matching wildcards */
    OFPFC_DELETE, /* Delete all matching flows. */
    OFPFC_DELETE_STRICT /* Strictly match wildcards and priority. */
};
```

priority 仅跟设置了通配符的域相关。priority 域指明了流表的优先级，数字越大，则优先级越高。因此，高优先级的带通配符流表项必须要被放到低编号的流表中。交换机负责正确的顺序，避免高优先级规则覆盖掉低优先级的。

buffer_id 标志被 OFPT_PACKET_IN 消息发出的网包在 buffer 中的 id。

out_port 可选的用于进行删除操作时的匹配。

flags 域可能包括如下的标志。

```
enum ofp_flow_mod_flags {
    OFPFF_SEND_FLOW_REM = 1 << 0, /* Send flow removed message when
flow
* expires or is deleted. */
    OFPFF_CHECK_OVERLAP = 1 << 1, /* Check for overlapping entries first. */
    OFPFF_EMERG = 1 << 2 /* Remark this is for emergency. */
};
```

当 OFPFF_SEND_FLOW_REM 被设置的时候，表项超时删除会触发一条表项删除的信息。

当 OFPFF_CHECK_OVERLAP 被设置的时候，交换机必须检查同优先级的表项之间是否有匹配范围的冲突。

当 OFPFF_EMERG 被设置的时候，交换机将表项当作紧急表项，只有当与控制器连接断开的时候才启用。

控制器使用 OFPT_PORT_MOD 消息来修改交换机物理端口的行为。

```
/* Modify behavior of the physical port */
struct ofp_port_mod {
    struct ofp_header header;
    uint16_t port_no;
    uint8_t hw_addr[OF_ETH_ALEN]; /* The hardware address is not
configurable. This is used to
sanity-check the request, so it must
be the same as returned in an
```

```

ofp_phy_port struct. */

uint32_t config; /* Bitmap of OFPPC_* flags. */
uint32_t mask; /* Bitmap of OFPPC_* flags to be changed. */

uint32_t advertise; /* Bitmap of "ofp_port_features"s. Zero all bits to
prevent any action taking place. */
uint8_t pad[4]; /* Pad to 64-bits. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 32);

```

mask 用来选择 config 中被修改的域。advertise 域没有掩码，所有的特性一起修改。

2.5.3.4 配置队列

队列的配置不在 of 协议考虑内。可以通过命令行或者其他协议来配置。控制器可以利用下面的结构来查询某个端口已经配置好的队列信息。

```

/* Query for port queue configuration. */
struct ofp_queue_get_config_request {
    struct ofp_header header;
    uint16_t port; /* Port to be queried. Should refer
to a valid physical port (i.e. < OFPP_MAX) */
    uint8_t pad[2]; /* 32-bit alignment. */
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_request) == 12);

```

交换机回复一个 ofp_queue_get_config_reply 命令，其中包括配置好队列的一个列表。

```

/* Queue configuration for a given port. */
struct ofp_queue_get_config_reply {
    struct ofp_header header;
    uint16_t port;
    uint8_t pad[6];
    struct ofp_packet_queue queues[0]; /* List of configured queues. */
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_reply) == 16);

```

2.5.3.5 获取状态

2.5.3.5.1 基本结构

系统运行时，datapath 可以通过 OFPT_STATS_REQUEST 消息来查询当前状态。

```
struct ofp_stats_request {
    struct ofp_header header;
    uint16_t type; /* One of the OFPST_* constants. */
    uint16_t flags; /* OFPSF_REQ_* flags (none yet defined). */
    uint8_t body[0]; /* Body of the request. */
};
OFP_ASSERT(sizeof(struct ofp_stats_request) == 12);
```

交换机则回复一条或多条 OFPT_STATS_REPLY 消息。

```
struct ofp_stats_reply {
    struct ofp_header header;
    uint16_t type; /* One of the OFPST_* constants. */
    uint16_t flags; /* OFPSF_REPLY_* flags. */
    uint8_t body[0]; /* Body of the reply. */
};
OFP_ASSERT(sizeof(struct ofp_stats_reply) == 12);
```

其中，flags 用来标明是否有多条附加的回复，还是仅有一条。type 则定义信息的类型，以决定 body 中信息该如何解析。

```
enum ofp_stats_types {
    /* Description of this OpenFlow switch.
     * The request body is empty.
     * The reply body is struct ofp_desc_stats. */
    OFPST_DESC,

    /* Individual flow statistics.
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_stats. */
    OFPST_FLOW,

    /* Aggregate flow statistics.
     * The request body is struct ofp_aggregate_stats_request.
     * The reply body is struct ofp_aggregate_stats_reply. */
    OFPST_AGGREGATE,
```



```

/* Flow table statistics.
 * The request body is empty.
 * The reply body is an array of struct ofp_table_stats. */
OFPST_TABLE,

/* Physical port statistics.
 * The request body is struct ofp_port_stats_request.
 * The reply body is an array of struct ofp_port_stats. */
OFPST_PORT,

/* Queue statistics for a port
 * The request body defines the port
 * The reply body is an array of struct ofp_queue_stats */
OFPST_QUEUE,

/* Vendor extension.
 * The request and reply bodies begin with a 32-bit vendor ID, which takes
 * the same form as in "struct ofp_vendor_header". The request and reply
 * bodies are otherwise vendor-defined. */
OFPST_VENDOR = 0xffff
};

```

2.5.3.5.2 整体信息

OFPST_DESC 请求类型提供了制造商、软件、硬件版本、序列号等整体信息。

```

/* Body of reply to OFPST_DESC request. Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc_stats {
char mfr_desc[DESC_STR_LEN]; /* Manufacturer description. */
char hw_desc[DESC_STR_LEN]; /* Hardware description. */
char sw_desc[DESC_STR_LEN]; /* Software description. */
char serial_num[SERIAL_NUM_LEN]; /* Serial number. */
char dp_desc[DESC_STR_LEN]; /* Human readable description of datapath.
 */
};
OFP_ASSERT(sizeof(struct ofp_desc_stats) == 1056);

```

每一项都是 ASCII 格式，以 null 结尾。DESC_STR_LEN 是 256，而 SERIAL_NUM_LEN 是 32。dp_desc 作为调试目的，可以自定义一些交换机标识信息。

```

/* Body for ofp_stats_request of type OFPST_FLOW. */

```

```

struct ofp_flow_stats_request {
    struct ofp_match match; /* Fields to match. */
    uint8_t table_id; /* ID of table to read (from ofp_table_stats),
    0xff for all tables or 0xfe for emergency. */
    uint8_t pad; /* Align to 32 bits. */
    uint16_t out_port; /* Require matching entries to include this
    as an output port. A value of OFPP_NONE
    indicates no restriction. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 44);

```

2.5.3.5.3 单流请求信息

OFPPST_FLOW 请求类型则可以获取针对某个流的单独信息。

```

/* Body for ofp_stats_request of type OFPPST_FLOW. */
struct ofp_flow_stats_request {
    struct ofp_match match; /* Fields to match. */
    uint8_t table_id; /* ID of table to read (from ofp_table_stats), 0xff for all
    tables or 0xfe for emergency. */
    uint8_t pad; /* Align to 32 bits. */
    uint16_t out_port; /* Require matching entries to include this
    as an output port. A value of OFPP_NONE
    indicates no restriction. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 44);

```

match 域包括对于匹配流的描述。

table_id 描述要读取的流表的 id, 0xff 则表示所有的流表。

out_port 是可选的对出口进行过滤的域。如果不是 OFPP_NONE, 那么可以进行匹配过滤。注意如果不需要进行出口过滤, 则需要被设置为 OFPP_NONE。

2.5.3.5.4 单流回复消息

回复信息包括下列的任意数组。

```

/* Body of reply to OFPPST_FLOW request. */
struct ofp_flow_stats {
    uint16_t length; /* Length of this entry. */
    uint8_t table_id; /* ID of table flow came from. */
    uint8_t pad;
    struct ofp_match match; /* Description of fields. */
};

```

```

uint32_t duration_sec;    /* Time flow has been alive in seconds. */
uint32_t duration_nsec;  /* Time flow has been alive in nanoseconds beyond
duration_sec. */
uint16_t priority;       /* Priority of the entry. Only meaningful
when this is not an exact-match entry. */
uint16_t idle_timeout;   /* Number of seconds idle before expiration. */
uint16_t hard_timeout;   /* Number of seconds before expiration. */
uint8_t pad2[6];         /* Align to 64-bits. */
uint64_t cookie;         /* Opaque controller-issued identifier. */
uint64_t packet_count;   /* Number of packets in flow. */
uint64_t byte_count;     /* Number of bytes in flow. */
struct ofp_action_header actions[0]; /* Actions. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 88);

```

这些域包括 flow_mod 消息中提供的项、要插入的流表、包计数、流量计数等。duration_sec 和 duration_nsec 包括了流表项自创建起的时间。单位分别是秒和纳秒。

2.5.3.5.5 多流请求信息

OFPST_AGGREGATE 请求类型可以获取多流信息。

```

/* Body for ofp_stats_request of type OFPST_AGGREGATE. */
struct ofp_aggregate_stats_request {
    struct ofp_match match; /* Fields to match. */
    uint8_t table_id;       /* ID of table to read (from ofp_table_stats)
0xff for all tables or 0xfe for emergency. */
    uint8_t pad;            /* Align to 32 bits. */
    uint16_t out_port;      /* Require matching entries to include this
as an output port. A value of OFPP_NONE
indicates no restriction. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 44);

```

match、table_id 跟 out_port 作用跟单流信息类似。

2.5.3.5.6 多流回复信息

多流回复信息体位如下结构。

```

/* Body of reply to OFPST_AGGREGATE request. */
struct ofp_aggregate_stats_reply {
    uint64_t packet_count; /* Number of packets in flows. */
};

```

```
uint64_t byte_count; /* Number of bytes in flows. */
uint32_t flow_count; /* Number of flows. */
uint8_t pad[4]; /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_reply) == 24);
```

2.5.3.5.7 表统计信息

表统计信息请求是通过 OFPST_TABLE 请求类型，该请求消息仅有消息头，不包括消息体。

表统计回复信息包括下面结构的一个数组。

```
/* Body of reply to OFPST_TABLE request. */
struct ofp_table_stats {
    uint8_t table_id; /* Identifier of table. Lower numbered tables
are consulted first. */
    uint8_t pad[3]; /* Align to 32-bits. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint32_t wildcards; /* Bitmap of OFPFW_* wildcards that are supported by the
table. */
    uint32_t max_entries; /* Max number of entries supported. */
    uint32_t active_count; /* Number of active entries. */
    uint64_t lookup_count; /* Number of packets looked up in table. */
    uint64_t matched_count; /* Number of packets that hit table. */
};
OFP_ASSERT(sizeof(struct ofp_table_stats) == 64);
```

body 域中包括通配符，来指明表中支持通配符的域。例如，直接 hash 表该域为 0，而顺序匹配表则为 OFPFW_ALL。各项以包经过表的顺序排序。

OFP_MAX_TABLE_NAME_LEN 是 32。

2.5.3.5.8 端口统计请求信息

端口状态的请求消息类型为 OFPST_PORT。

```
/* Body for ofp_stats_request of type OFPST_PORT. */
struct ofp_port_stats_request {
    uint16_t port_no; /* OFPST_PORT message must request statistics
* either for a single port (specified in
* port_no) or for all ports (if port_no ==
* OFPP_NONE). */
    uint8_t pad[6];
```

```
};
OFP_ASSERT(sizeof(struct ofp_port_stats_request) == 8);
```

port_no 用来过滤获取信息的端口，如果获取所有端口信息，则设置为 OFPP_NONE。

2.5.3.5.9 端口统计回复信息

回复消息体中包括下面结构的一个数组。

```
/* Body of reply to OFPST_PORT request. If a counter is unsupported, set
 * the field to all ones. */
struct ofp_port_stats {
    uint16_t port_no;
    uint8_t pad[6]; /* Align to 64-bits. */
    uint64_t rx_packets; /* Number of received packets. */
    uint64_t tx_packets; /* Number of transmitted packets. */
    uint64_t rx_bytes; /* Number of received bytes. */
    uint64_t tx_bytes; /* Number of transmitted bytes. */
    uint64_t rx_dropped; /* Number of packets dropped by RX. */
    uint64_t tx_dropped; /* Number of packets dropped by TX. */
    uint64_t rx_errors; /* Number of receive errors. This is a super-set
    of more specific receive errors and should be
    greater than or equal to the sum of all
    rx_*_err values. */
    uint64_t tx_errors; /* Number of transmit errors. This is a super-set
    of more specific transmit errors and should be
    greater than or equal to the sum of all
    tx_*_err values (none currently defined.) */
    uint64_t rx_frame_err; /* Number of frame alignment errors. */

    uint64_t rx_over_err; /* Number of packets with RX overrun. */
    uint64_t rx_crc_err; /* Number of CRC errors. */
    uint64_t collisions; /* Number of collisions. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 104);
```

对于不可用的计数器，交换机将返回-1。

2.5.3.5.10 队列信息

请求消息类型为 OFPST_QUEUE。消息体中 port_no 和 queue_id 分别指明要查询的端口和对列。如果查询全部，则分别置为 OFPP_ALL 和 OFPQ_ALL。

```

struct ofp_queue_stats_request {
    uint16_t port_no; /* All ports if OFPT_ALL. */
    uint8_t pad[2]; /* Align to 32-bits. */
    uint32_t queue_id; /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_request) == 8);

```

回复消息体中包括下面结构的一个数组。

```

struct ofp_queue_stats {
    uint16_t port_no;
    uint8_t pad[2]; /* Align to 32-bits. */
    uint32_t queue_id; /* Queue i.d */
    uint64_t tx_bytes; /* Number of transmitted bytes. */
    uint64_t tx_packets; /* Number of transmitted packets. */
    uint64_t tx_errors; /* Number of packets dropped due to overrun. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 32);

```

2.5.3.5.11 生产商信息

生产商信息通过 OFPST_VENDOR 消息类型来请求获取，消息首 4 个字节为生产商的标志号。

vendor 域为 32 位，每个生产商是唯一的。如果首字节为 0，则剩下 3 个字节为生产商对应的 IEEE OUI。

2.5.3.6 发包

如果控制器希望通过交换机发包，需要利用 OFPT_PACKET_OUT 消息。

```

/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id; /* ID assigned by datapath (-1 if none). */
    uint16_t in_port; /* Packet's input port (OFPP_NONE if none). */
    uint16_t actions_len; /* Size of action array in bytes. */

    struct ofp_action_header actions[0]; /* Actions. */
    /* uint8_t data[0]; */ /* Packet data. The length is inferred
    from the length field in the header.
    (Only meaningful if buffer_id == -1.) */
};

```

```
OFPP_ASSERT(sizeof(struct ofp_packet_out) == 16);
```

buffer_id 跟 ofp_packet_in 中给出的一致。如果 buffer_id 是-1，则网包内容被包括在 data 域中。如果 OFPP_TABLE 被指定为发出行为的出口，则 packet_out 中的 in_port 将被用来进行流表查询。

2.5.3.7 保障消息

控制器如果仅仅想确保消息依赖完整或指令完成，可以使用 OFPT_BARRIER_REQUEST 消息。该消息类型没有消息体。

交换机一旦收到该消息，则需要先执行完该消息前到达的所有指令，然后再执行其后的。之前指令处理完成后，交换机要回复 OFPT_BARRIER_REPLY 消息，且携带有原请求信息的 xid 信息。

2.5.4 Asynchronous 消息

包括包进入（Packet_In）、流删除（Flow Removed）、端口状态（Port Status）、错误（Error）等。

2.5.4.1 包进入

当网包被 datapath 获取并发给控制器的时候，使用 OFPT_PACKET_IN 消息。

```
/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id; /* ID assigned by datapath. */
    uint16_t total_len; /* Full length of frame. */
    uint16_t in_port; /* Port on which frame was received. */
    uint8_t reason; /* Reason packet is being sent (one of OFPR_*) */
    uint8_t pad;
    uint8_t data[0]; /* Ethernet frame, halfway through 32-bit word,
so the IP header is 32-bit aligned. The
amount of data is inferred from the length
field in the header. Because of padding,
offsetof(struct ofp_packet_in, data) ==
sizeof(struct ofp_packet_in) - 2. */
};
OFPP_ASSERT(sizeof(struct ofp_packet_in) == 20);
```

其中 buffer_id 是 datapath 用来标志一个在缓存中的网包。当包进入消息发给控制

器的时候，部分网包信息也会被一并发给控制器。

如果网包是因为流表项行为指定（发给控制器）而发给控制器，则有 `action_output` 中的 `max_len` 字节数据发给控制器；如果是因为查不到匹配表项而发给控制器，则至少 `OFPT_SET_CONFIG` 消息中的 `miss_send_len` 字节发给控制器。默认 `miss_send_len` 是 128（字节）。如果网包没有放入缓存，则整个网包内容都需要发给控制器，此时 `buffer_id` 被设置为-1。

交换机需要负责在控制器接手前保护相关的 `buffer` 信息不被复用。

`reason` 域可以为如下类型。

```
/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_NO_MATCH, /* No matching flow. */

    OFPR_ACTION /* Action explicitly output to controller. */
};
```

2.5.4.2 流删除

如果控制器希望在流表项超时删除的时候得到通知，则 `datapath` 通过 `OFPT_FLOW_REMOVED` 消息类型来通知控制器。

```
/* Flow removed (datapath -> controller). */
struct ofp_flow_removed {
    struct ofp_header header;
    struct ofp_match match; /* Description of fields. */
    uint64_t cookie; /* Opaque controller-issued identifier. */

    uint16_t priority; /* Priority level of flow entry. */
    uint8_t reason; /* One of OFPRR_*. */
    uint8_t pad[1]; /* Align to 32-bits. */

    uint32_t duration_sec; /* Time flow was alive in seconds. */
    uint32_t duration_nsec; /* Time flow was alive in nanoseconds beyond
    duration_sec. */
    uint16_t idle_timeout; /* Idle timeout from original flow mod. */
    uint8_t pad2[2]; /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 88);
```

其中，`match`、`cookie`、`priority` 等域的定义跟建立流表项时候一致。

`reason` 域可以为如下类型之一。


```

/* Why was this flow removed? */
enum ofp_flow_removed_reason {
    OFPPR_IDLE_TIMEOUT, /* Flow idle time exceeded idle_timeout. */
    OFPPR_HARD_TIMEOUT, /* Time exceeded hard_timeout. */
    OFPPR_DELETE /* Evicted by a DELETE flow mod. */
};

```

idle_timeout 则为添加表项时候指定。

packet_count 和 byte_count 分别用来计数与该流表项相关的包和流量信息。

2.5.4.3 端口状态

当物理端口被添加、删除或修改的时候，datapath 需要使用 OFPT_PORT_STATUS 消息来通知控制器。

```

/* A physical port has changed in the datapath */
struct ofp_port_status {
    struct ofp_header header;
    uint8_t reason; /* One of OFPPR_*. */
    uint8_t pad[7]; /* Align to 64-bits. */
    struct ofp_phy_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 64);

```

status 可以为如下的值之一。

```

/* What changed about the physical port */
enum ofp_port_reason {
    OFPPR_ADD, /* The port was added. */
    OFPPR_DELETE, /* The port was removed. */
    OFPPR_MODIFY /* Some attribute of the port has changed. */
};

```

2.5.4.4 错误

2.5.4.4.1 基本结构

当交换机需要通知控制器发生问题时，可以使用 OFPT_ERROR_MSG 消息。

```

/* OFPT_ERROR: Error message (datapath -> controller). */
struct ofp_error_msg {
    struct ofp_header header;
};

```

```
uint16_t type;
uint16_t code;
uint8_t data[0]; /* Variable-length data.  Interpreted based
on the type and code. */
};
OFP_ASSERT(sizeof(struct ofp_error_msg) == 12);
```

`type` 用来标明高层的错误类型, `code` 为对应类型的错误代码。`data` 则为错误相关的其他信息。

以 `_EPM` 结尾的 `code` 对应为权限问题。

目前错误类型有

```
/* Values for 'type' in ofp_error_message.  These values are immutable: they
* will not change in future versions of the protocol (although new values may
* be added). */
enum ofp_error_type {

    OFPET_HELLO_FAILED, /* Hello protocol failed. */
    OFPET_BAD_REQUEST, /* Request was not understood. */
    OFPET_BAD_ACTION, /* Error in action description. */
    OFPET_FLOW_MOD_FAILED, /* Problem modifying flow entry. */
    OFPET_PORT_MOD_FAILED, /* Port mod request failed. */
    OFPET_QUEUE_OP_FAILED /* Queue operation failed. */

};
```

2.5.4.4.2 错误类型对应代码

对 `OFPET_HELLO_FAILED` 错误类型, 错误代码有

```
/* ofp_error_msg 'code' values for OFPET_HELLO_FAILED.  'data' contains
an
* ASCII text string that may give failure details. */
enum ofp_hello_failed_code {
    OFPHFC_INCOMPATIBLE, /* No compatible version. */
    OFPHFC_EPERM /* Permissions error. */
};
```

`data` 域包括 ASCII 字符串来描述错误的一些细节。

对于 `OFPET_BAD_REQUEST` 错误类型, 错误代码有

```
/* ofp_error_msg 'code' values for OFPET_BAD_REQUEST.  'data' contains at
least
```

```

* the first 64 bytes of the failed request. */
enum ofp_bad_request_code {
    OFPBRC_BAD_VERSION, /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE, /* ofp_header.type not supported. */
    OFPBRC_BAD_STAT, /* ofp_stats_request.type not supported. */
    OFPBRC_BAD_VENDOR, /* Vendor not supported (in ofp_vendor_header
* or ofp_stats_request or ofp_stats_reply). */

    OFPBRC_BAD_SUBTYPE, /* Vendor subtype not supported. */
    OFPBRC_EPERM, /* Permissions error. */
    OFPBRC_BAD_LEN, /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY, /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN /* Specified buffer does not exist. */
};

```

data 域至少包括 64 字节的失败请求。

对于 OFPET_BAD_ACTION 错误类型，错误代码有

```

/* ofp_error_msg 'code' values for OFPET_BAD_ACTION. 'data' contains at least
* the first 64 bytes of the failed request. */
enum ofp_bad_action_code {

    OFPBAC_BAD_TYPE, /* Unknown action type. */
    OFPBAC_BAD_LEN, /* Length problem in actions. */
    OFPBAC_BAD_VENDOR, /* Unknown vendor id specified. */
    OFPBAC_BAD_VENDOR_TYPE, /* Unknown action type for vendor id. */
    OFPBAC_BAD_OUT_PORT, /* Problem validating output action. */
    OFPBAC_BAD_ARGUMENT, /* Bad action argument. */
    OFPBAC_EPERM, /* Permissions error. */
    OFPBAC_TOO_MANY, /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE /* Problem validating output queue. */
};

```

data 域至少包括 64 字节的失败请求。

对于 OFPET_FLOW_MOD_FAILED 错误类型，错误代码有

```

/* ofp_error_msg 'code' values for OFPET_FLOW_MOD_FAILED. 'data'
contains
* at least the first 64 bytes of the failed request. */
enum ofp_flow_mod_failed_code {
    OFPFMFC_ALL_TABLES_FULL, /* Flow not added because of full tables. */
    OFPFMFC_OVERLAP, /* Attempted to add overlapping flow with
* CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM, /* Permissions error. */
};

```

```

OFPFMFC_BAD_EMERG_TIMEOUT, /* Flow not added because of non-zero
idle/hard
* timeout. */
OFPFMFC_BAD_COMMAND, /* Unknown command. */
OFPFMFC_UNSUPPORTED /* Unsupported action list - cannot process in
* the order specified. */
};

```

data 域至少包括 64 字节的失败请求。

对于 OFPET_PORT_MOD_FAILED 错误类型，错误代码有

```

/* ofp_error_msg 'code' values for OFPET_PORT_MOD_FAILED. 'data'
contains
* at least the first 64 bytes of the failed request. */
enum ofp_port_mod_failed_code {
OFPFMFC_BAD_PORT, /* Specified port does not exist. */
OFPFMFC_BAD_HW_ADDR, /* Specified hardware address is wrong. */
};

```

data 域至少包括 64 字节的失败请求。

对于 OFPET_QUEUE_OP_FAILED 错误类型，错误代码有

```

/* ofp_error_msg 'code' values for OFPET_QUEUE_OP_FAILED. 'data'
contains
* at least the first 64 bytes of the failed request */
enum ofp_queue_op_failed_code {
OFPQOFC_BAD_PORT, /* Invalid port (or port does not exist). */

OFPQOFC_BAD_QUEUE, /* Queue does not exist. */ OFPQOFC_EPERM /*
Permissions error. */
};

```

data 域至少包括 64 字节的失败请求。

如果错误消息是回复给控制器的指定请求，如 OFPET_BAD_REQUEST，OFPET_BAD_ACTION 或 OFPET_FLOW_MOD_FAILED，则消息头中的 xid 需要跟对应请求消息一致。

2.5.5 Symmetric 消息

包括 Hello、响应请求、回复、生产商信息等。

2.5.5.1 Hello

OFPT_HELLO 消息没有消息体，仅有 of 消息头。扩展版本应该能解释带有消息体的 hello 消息，并将其消息体内容忽略。

2.5.5.2 响应请求

响应请求消息由一个 of 消息头加上任意的消息体组成，用来协助测量延迟、带宽、控制器跟交换机之间是否保持连接等信息。

2.5.5.3 响应回复

响应回复消息由一个 of 消息头加上对应请求的无修改消息体组成，用来协助测量延迟、带宽、控制器跟交换机之间是否保持连接等信息。

2.5.5.4 生产商信息

生产商消息结构如下

```
/* Vendor extension. */
struct ofp_vendor_header {
    struct ofp_header header;    /* Type OFPT_VENDOR */
    uint32_t vendor;
    /* Vendor ID:
     * - MSB 0: low-order bytes are IEEE OUI.
     * - MSB != 0: defined by OpenFlow consortium. */
    /* Vendor-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_vendor_header) == 12);
```

vendor 域为 32 位长，如果首字节为 0，则其他 3 个字节定义为 IEEE OUI。

如果交换机无法理解一条生产商消息，它需要发送 OFPT_ERROR 消息带有 OFPBRC_BAD_VENDOR 错误代码和 OFPET_BAD_REQUEST 错误代码。

第3章 OpenvSwitch

第3.1节 概述

官方首页为 <http://openvswitch.org/>,

第3.2节 特性

OVS 官方的定位是要做一个产品级质量的多层虚拟交换机, 通过支持可编程扩展来实现大规模的网络自动化。设计目标是方便管理和配置虚拟机网络, 检测多物理主机在动态虚拟环境中的流量情况。针对这一目标, OVS 具备很强的灵活性。可以在管理程序中作为软件 switch 运行, 也可以直接部署到硬件设备上作为控制层。同时在 Linux 上支持内核态 (性能高)、用户态 (灵活)。此外 OVS 还支持多种标准的管理接口, 如 Netflow、sFlow、RSPAN、ERSPAN、CLI。对于其他的虚拟交换机设备如 VMware 的 vNetwork 分布式交换机跟思科 Nexus 1000V 虚拟交换机等它也提供了较好的支持。

目前 OVS 的官方版本为 1.1.0pre2, 主要特性包括

- 虚拟机间互联的可视性;
- 支持 trunking 的标准 802.1Q VLAN 模块;
- 细粒度的 QoS; 每虚拟机端口的流量策略;
- 负载均衡支持 OpenFlow ;
- 远程配置兼容 Linux 桥接模块代码

第3.3节 代码

由于是开源项目, 代码获取十分简单, 最新代码可以利用 git 从官方网站下载。此外官方网站还提供了比较清晰的文档资料和应用例程, 其部署十分轻松。当前最新代码包主要包括以下模块和特性:

ovs-vswitchd 主要模块, 实现 switch 的 daemon, 包括一个支持流交换的 Linux 内核模块;

ovsdb-server 轻量级数据库服务器, 提供 ovs-vswitchd 获取配置信息;

ovs-brcomptd 让 ovs-vswitch 替换 Linux bridge, 包括获取 bridge ioctls 的 Linux 内核模块;

ovs-dpctl 用来配置 switch 内核模块;

一些 Scripts and specs 辅助 OVS 安装在 Citrix XenServer 上, 作为默认 switch;

ovs-vsctl 查询和更新 ovs-vswitchd 的配置;

ovs-appctl 发送命令消息, 运行相关 daemon; ovsdbmonitor GUI 工具, 可以远程获取

OVS 数据库和 OpenFlow 的流表。

此外，OVS 也提供了支持 OpenFlow 的特性实现，包括

ovs-openflowd 一个简单的 OpenFlow 交换机；

ovs-controller 一个简单的 OpenFlow 控制器；

ovs-ofctl 查询和控制 OpenFlow 交换机和控制器；

ovs-pki 为 OpenFlow 交换机创建和管理公钥框架；tcpdump 的补丁，解析 OpenFlow 的消息；

第3.4节 命令

第4章 NOX

现代大规模的网络环境十分复杂，给管理带来较大的难度。特别对于企业网络来说，管控需求繁多，应用、资源多样化，安全性、扩展性要求都特别高。因此，网络管理始终是研究的热点问题。对于传统网络来说，交换机等设备提供的可观测性和控制性都十分有限。一方面管理员难以实时获取足够的网络统计信息，另一方面控制手段十分单一，依赖于静态的 policy 部署。而基于软件定义网络，这两方面的问题几乎迎刃而解。

第4.1节 网络操作系统

早期的计算机程序开发者直接用机器语言编程。因为没有各种抽象的接口来管理底层的物理资源（内存、磁盘、通信），使得程序的开发、移植、调试等费时费力。而现代的操作系统提供更高的抽象层来管理底层的各种资源，极大的改善了软件程序开发的效率。

同样的情况出现在现代的网络管理中，管理者的各种操作需要跟底层的物理资源直接打交道。例如通过 ACL 规则来管理用户，需要获取用户的实际 IP 地址。更复杂的管理操作甚至需要管理者事先获取网络拓扑结构、用户实际位置等。随着网络规模的增加和需求的提高，管理任务实际上变成巨大的挑战。

而 NOX（开源项目，目前由 nicira 公司维护）则试图从建立网络操作系统的层面来改变这一困境。网络操作系统（Network Operating System）这个术语早已经被不少厂家提出，例如 Cisco 的 IOS、Novell 的 NetWare 等。这些操作系统实际上提供的是用户跟某些部件（例如交换机、路由器）的交互，因此称为交换机/路由器操作系统可能更贴切。而从整个网络的角度来看，网络操作系统应该是抽象网络中的各种资源，为网络管理提供易用的接口。

第4.2节 模型

从面向层面的不同，NOX 主要功能包括两部分，一是针对下层的 SDN 交换机（例如 of 交换机）作为控制器，解析交换机行为和进行管理操作；一是针对上层网络管理 app 开发者，作为一个抽象层（操作系统）提供易用的开发接口。

针对这两部分功能，NOX 的开发模型主要包括两个部分。

一是集中的编程模型。开发者不需要关心网络的实际架构，部署在单一节点上的 NOX 管理网络中的所有交换设备，解析后呈现给开发者。在开发者看来整个网络就好像一台单独的机器一样，有统一的资源分配和接口管理。

二是抽象的开发模型。应用程序开发需要面向的是 NOX 提供的高层接口，而不是底层。例如，应用面向的是用户、机器名，但不面向 IP 地址、MAC 地址等。

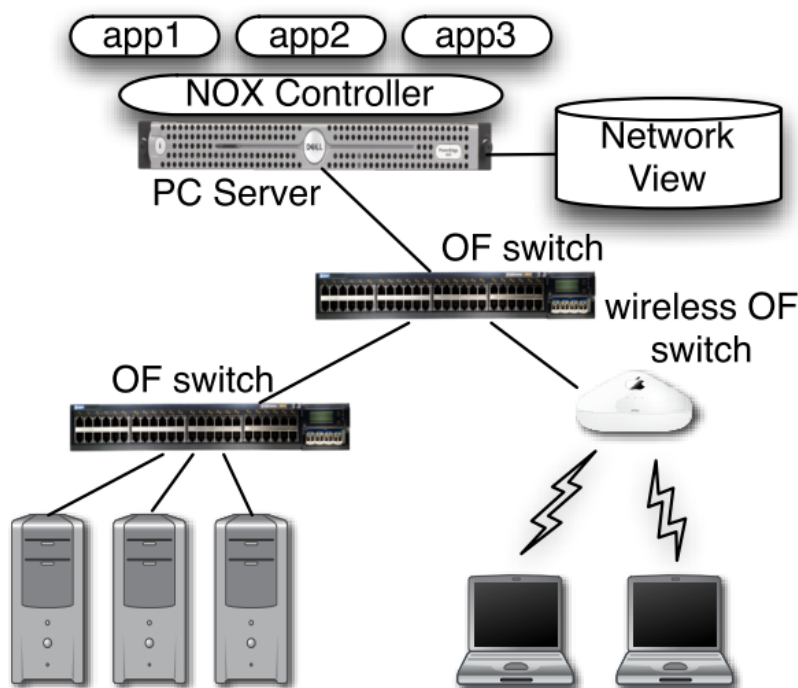
正如计算机操作系统本身并不实现复杂的各种软件功能，NOX 本身并不完成对网络管理任务，而是通过在其上运行的各种“应用”（Application）来实现具体的管理任务。管

理者和开发者可以利用高级语言包括 C++ 和 Python 来专注到这些应用的开发上，而无需花费时间在对底层细节的分析上。为了实现这一目的，NOX 需要提供尽可能通用（General）的接口，来满足各种不同的管理需求。

第4.3节 架构

4.3.1 组件

下图给出了使用 NOX 管理网络环境的主要组件。包括交换机和控制（服务）器（其上运行 NOX 和相应的多个管理应用，以及 1 个 Network View），其中 Network View 提供了对网络物理资源的不同观测和抽象解析。注意到 NOX 通过对交换机操作来管理流量，因此，交换机需要支持相应的管理功能。此处采用支持 OpenFlow 的交换机。



图表 4-1 NOX 管理网络的主要部件

4.3.2 操作

NOX 针对的对象是流（Flow），对于每一个新建立的流，第一个包被发送到 NOX，由相应的 APP 来处理。

对于标准的 of 交换机，当流量经过交换机时，如果发现没有对应的匹配表项，则转发到运行 NOX 的控制器，NOX 上的应用通过流量信息来建立 Network View 和决策流量的行为。同样的，NOX 也可以控制哪些流量需要转发给控制器。

4.3.3 多粒度处理

对网络中不同粒度的事件提供不同的处理。

网包

对于 10 Gbps 链路，网包到达速度可以达到每秒百万级别。因此，大部分网包应该通过交换机来处理。

网流

一般情况下，网流的更新速度要比网包的低至少数量级。网络需要通过管理应用（APP on NOX）来处理。

network view

网络自身的更新，频度更低。

4.3.4 开发实现

NOX 上的开发支持 Python、C++ 语言，NOX 核心架构跟关键部分都是使用 C++ 实现以保证性能。代码可以从 <http://www.noxrepo.org> 获取，并遵循 GPL 许可。

4.3.4.1 接口

事件

事件包括流到达、结束、用户进入、离开、交换机进入、离开等等。

Network View 和名称空间

通过检测 DNS 来管理用户、主机验证和名称映射，提供上层拓扑无关的编程接口。只有在改变的时候才需要。

控制

NOX 对流量的控制是通过管理交换机来实现的。网络功能诸如 DPI 通过重定向网络流量到中间件来实现。

系统库

提供基本的高效系统库，包括路由、包分类、标准的网络服务（DHCP、DNS）、协议过滤器等。

第4.4节 安装

4.4.1 步骤

4.4.1.1 从源代码编译

在运行 Linux 且满足相关依赖的主机上，可以通过下面的操作来安装最新的 NOX。

```
git clone git://noxrepo.org/nox
cd nox
./boot.sh
mkdir build/
cd build/
../configure --with-python=yes
make
make check
```

如果是从源代码 tarfile 安装，则无需运行 boot.sh。

如果确保无需 C++ STL 的 debugging 检查，可以使用下面的命令来关闭该检查以获取更快的安装速度。

```
./configure --with-python=`which python2.5` --enable-ndebug
```

编译完成后，在本地 src 目录下会生成 nox_core 程序。需要注意 nox_core 只能从 src 目录下进行执行，且编译目录的相关路径要确保不变。

4.4.1.2 Deb 包安装

通过下面命令来获取依赖包信息

```
wget http://openflowswitch.org/downloads/debian/binary/nox-dependencies.deb
dpkg --info nox-dependencies.deb
```

通过下面的命令来使用 apt-get 安装

```
cd /etc/apt/sources.list.d
sudo wget http://openflowswitch.org/downloads/debian/nox.list
sudo apt-get update
```

```
sudo apt-get install nox-dependencies
```

4.4.2 依赖

理论上 NOX 可以运行在大部分的 Linux 发行版上，官方开发是基于 Debian 系列，因此推荐采用 Debian Lenny 或者 Ubuntu。

NOX 依赖以下的安装包。

- g++ 4.2 or greater
- Boost C++ libraries, v1.34.1 or greater (<http://www.boost.org>)
- Xerces C++ parser, v2.7.0 or greater (<http://xerces.apache.org/xerces-c>)

为了支持 Python，还需要

- SWIG v1.3.0 or greater (<http://www.swig.org>)
- Python2.5 or greater (<http://python.org>)
- Twisted Python (<http://twistedmatrix.com>)

用户界面还需要

- Mako Templates (<http://www.makotemplates.org/>)
- Simple JSON (<http://www.undefined.org/python/>)

以在 Debian 系统为例，编译核心功能具体需要安装的包有

```
apt-get install autoconf automake g++ libtool python python-twisted swig
libboost1.35-dev libxerces-c2-dev libssl-dev make
```

编译全部功能还需要执行

```
apt-get install libsqlite3-dev python-simplejson
```

生成文档需要执行

```
apt-get install python-sphinx
```

4.4.3 选项

仅编译核心功能。

```
./boot --apps-core
```

配置 python

```
./configure --with-python=[yes|no|path/to/python]
```

关闭测试检查

```
./configure --enable-ndebug
针对某个指定版本的 of 协议进行编译
./configure --with-openflow=/path/to/openflow
```

4.4.4 校验

可以通过下面的命令来校验 NOX 是否编译正确。

```
make check
```

同时，在 src 目录下，可以对生成的 nox_core 程序进行单元测试。

```
cd src
./nox_core tests
```

一个简单的例子是采用如下命令来打印 10 个相同的网包描述。

```
cd src/
./nox_core -v -i pgen:10 packetdump
```

第4.5节 应用

4.5.1 框架

在 nox 的应用环境中，网络中存在多台 of 的交换设备（交换机），nox 监听指定的 tcp 端口，一旦有 of 交换机通过监听端口与 nox 建立连接，nox 将转发收到的流量到运行部件，并将运行部件的命令发送给交换机。nox 可以同时运行多个部件，部件所在目录包括 src/nox/coreapps/, src/nox/netapps/, src/nox/webapps/。一般 nox 需要通过命令行依照下面的格式来执行

```
./nox_core [OPTIONS] [APP[=ARG[,ARG]...]] [APP[=ARG[,ARG]...]]...
```

例如下面的例子让 nox 监听 6633 端口（of 协议默认端口），且让 packetdump 程序来处理所收到的网包。

```
./nox_core -i ptcp:6633 packetdump
```

4.5.2 运行与接口

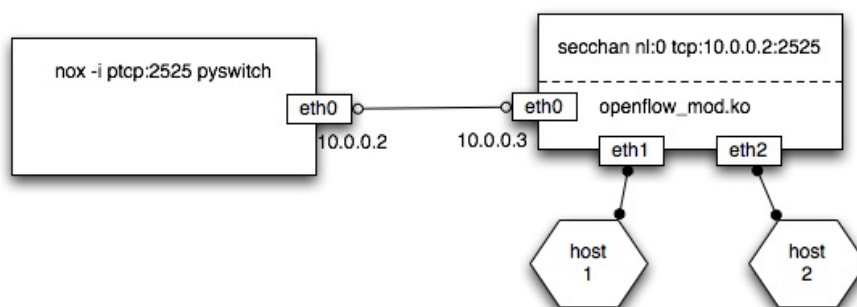
运行 nox 需要进入 src 目录下，执行 nox_core。使用信息可以使用 nox_core -help 来获取。

nox 支持的接口有如下 6 种。

- Passive TCP (-i ptcp:<portno>) 接受连入的 TCP 连接
- Active TCP (-i tcp:ip:<portno>) 主动发出的 TCP 连接
- Passive SSL (-i pssl:<portno>) 接受连入的 SSL 连接
- Active SSL (-i ssl:ip:<portno>) 主动发出的 SSL 连接
- pcap (-i pcap:/path/to/pcapfile[:/path/to/outfile]) 从指定 pcap 文件中读取所有的流量，同时可以将输出的网包写到指定文件
- Generated packets (-i pgen:<# to generate>) 生成一些相同的 flood 包（用于测试）

4.5.3 例程

例程拓扑如图表 4-14-2 所示。nox 跟 of 交换机都运行在普通的 linux 主机上，of 交换机有两个网口，其中 eth1 跟 eth2 用于普通的交换端口，eth0 作为与 nox 连接的控制口。



图表 4-3 拓扑结构

完成整个网络部署主要需要三步内容。

第一步，交换机用一个 datapath 来连接 eth1 和 eth2。相关命令为

```
insmod datapath/linux-2.6/openflow_mod.ko
utilities/dpctl adddp nl:0
utilities/dpctl addif nl:0 eth1
utilities/dpctl addif nl:0 eth2
```

第二步，启用 nox 程序，监听 2525 端口，并运行 python 的 l2 交换组件。

```
cd src/
./nox_core -i ptcp:2525 pyswitch
```

最后一步是运行交换机上的安全通道，来连接 datapath 和 nox。命令为

```
secchan/secchan nl:0 tcp:10.0.0.2:2525
```

此时，of 交换机已经可以通过 pyswitch 发挥交换作用，两台主机已经可以通过 of 交换机进行通信。

另外，nox 上可以同时执行多个组件，例如同时运行二层交换和打印收到包的组件。

```
nox_core -i ptcp:2525 switch packetdump
```

第4.6节 开发

NOX 的 app 开发需要了解两个基本的概念：组件（Component）跟事件（Event）。

4.6.1 组件

4.6.1.1 概念

组件是一些功能、声名或动态加载的集合。例如路由模块。

组件可以用 c++或 python（甚至同时使用）来编写。前者提供更好的性能，而后者的 api 更为友好。组件的代码一般在 src/nox/apps/目录，一般建议创建独立的子目录来存放相关的代码。

4.6.1.2 组件列表

核心 app

位于 src/nox/coreapps/目录下。包括

- messenger: 提供 TCP/SSL 服务器端的 sockets 来连接其他设备
- snmp: 使用 python 脚本来处理 snmptrap 作为 NetSNMP 处理器

网络 app

位于 src/nox/netapps/目录下。包括

- [Discovery](#) 跟踪所控制的交换机之间的链路状态
- [Topology](#) 保存网络中目前激活的链路结构
- [Authenticator](#) 跟踪网络中 hosts 和 switches 的位置
- [Routing](#) 计算路径

webapp

位于 src/nox/webapps/目录下。包括

- [Webservice](#) 为 nox 的 app 提供 web 服务接口

- [Webserver](#) 负责控制接口
- [Webserviceclient](#)

第三方 app

部分包括

- [OVN](#) 基于 NOX/Openflow 的网络可视化框架
- [Basic Spanning Tree](#) 为 of 网络创建一棵生成树
- [Mobile VMs](#) 迁移虚拟机的 demo, sigcomm2008 的最佳 demo
- [RipCord](#) 数据中心网络的组件化平台

4.6.1.3 创建组件

4.6.1.3.1 C++例程

以要实现一个简单的 Hub 为例。

Hub 类负责将收到的网包复制并从所有其他的口（不包括入口）转发出去。对于 nox 来说，需要转发收到的包，并在 of 交换机中添加流表项，来转发同一流的后续网包到其他所有端口。

代码如下

```
#include "component.hh"

class Hub
    : public Component
{
public:
    Hub(const Context* c,
        const xercesc::DOMNode*)
        : Component(c) {}

    void configure(const Configuration*) {}

    void install()
    {}
};

REGISTER_COMPONENT(container::Simple_component_factory<Hub>, Hub);
```

从这个例程可以看到，一个新创建的组件类必须从 Component 类继承得来，其中，

REGISTER_COMPONENT 宏用来帮助动态加载器。configure 和 install 方法在加载的时候被调用，用来注册事件和事件处理器。创建组件的同一目录下必须有一个 meta.xml 文件。

nox 启动时，会搜索目录树下的 meta.xml，分析依赖关系来决定可用的系统部件。对于 c++ 部件来说，<library>值必须跟共享库的名称一致。简单的部件一般在 install 方法进行事件注册，并执行相关的处理函数。对于本例来说，对于每一个收到的网包需要注册如下处理。

```
void install()
{
    register_handler<Packet_in_event>(boost::bind(&Hub::handler, this, _1));
}
```

组件之间可以通过事件或直接进行通信，如果要直接访问组件，需要先利用 resolve 函数（从 Component 类继承而来）来获得相应的句柄。例如要获取 topology 组件的句柄，需要执行

```
Topology* topology;
resolve(topology);
```

4.6.1.3.2 Python 例程

Python 创建组件要更为简单，一般的，组件代码需要有如下的结构。

```
from nox.lib.core import *

class foo(Component):

    def __init__(self, ctxt):
        Component.__init__(self, ctxt)

    def install(self):
        # register for event here
        pass

    def getInterface(self):
        return str(foo)

def getFactory():
    class Factory:
        def instance(self, ctxt):
            return foo(ctxt)
```

```
return Factory()
```

除了这些方法外，还可以添加 `configure` 方法，会在 `install` 方法之前执行。

下面的步骤创建一个纯粹的 python 组件。

- 添加 `.py` 文件到 `src/nox/apps/examples/` 目录
- 从 `src/nox/apps/examples/pyloop` 中复制代码(复制除了 `install` 方法部分的所有代码)
- 将创建的 python 文件名称添加到 `src/nox/apps/examples/Makefile.am` 中的 `NOX_RUNTIMEFILES` 中。
- 更新 `src/nox/apps/examples/meta.xml` 来包括新的 app。确保 python runtime 在依赖中。

句柄

核心的 python api 在 `nox/lib/core.py` 和 `nox/lib/util.py`

获取其他组件的句柄，使用 `Component.resolve(..)` 方法。例如

```
from nox.app.examples.pyswitch import pyswitch
self.resolve(pyswitch)
```

4.6.2 事件

4.6.2.1 概念

事件来推动所有的执行命令。一般常常与 `nox` 组件相关。例如一些内置的组件用来处理 of 消息。包括 `datapath` 加入事件、包进入事件、`datapath` 离开事件、流超时事件等。事件是组件之间相互联系的主要方式。

抽象的说，组件由一系列事件的处理器组成。事件是 `nox` 执行的驱动所在。

4.6.2.2 事件列表

核心事件

`nox` 包括一系列的内建事件，用来处理 of 消息，包括

- [Datapath join event](#) (`src/include/datapath-join.hh`) 处理新添加交换机
- [Datapath leave event](#) (`src/include/datapath-leave.hh`) 处理删除交换机
- [Packet in event](#) (`src/include/packet-in.hh`) `nox` 收到交换机发来的新的网包，包括交换机 ID, 入口和包 buffer 等。
- [Flow mod event](#) `nox` 添加或修改流表项

- [Flow removed event](#) 流超时或被删除
- [Port status event](#) (src/include/port-status.hh) 端口状态改变。当前端口状态，包括是否启用，速度和端口名称等。
- [Port stats in](#) 受控的交换机响应 Port_stats_request 消息发出的 Port_stats 消息被控制器收到时触发，包括指定端口的当前计数器值（包括 rx,tx, and errors）。
- Port_stats (src/include/port-stats.hh) 响应 Port_stats_request 消息

应用事件

另外，组件自身也可以定义或触发高层的可能被其他事件处理的事件，例如

- [Host event](#) (src/nox/apps/authenticator/host-event.hh) 新的主机加入网络或者由主机离开网络（generally due to timeout）
- [Flow in event](#) 当网络中的 [Packet in event](#) 事件被收到时候，由 [Authenticator](#) 触发，[Flow in event](#) 被路由程序处理。
- [Link event](#) (src/nox/apps/discovery/link-event.hh) 网络中发现 link 时候发送事件。由 [Discovery](#) 触发，可以用于自动维护网络拓扑。

4.6.2.3 注册事件

使用 c++ 语言注册事件，需要使用 Component::register_handler 方法（src/builtin/component.cc）。nox 中的 api 依赖于 boost 库的相关函数，可以参考 <http://www.boost.org/>。

所有句柄必须（handle）返回一个 Disposition 类型，可以是 CONTINUE 或 STOP。
下面的程序给出了注册一个网包进入事件的例子。

```
Disposition handler(const Event& e)
{
    return CONTINUE;
}

void install()
{
    register_handler<Packet_in_event>(boost::bind(handler, this, _1));
}
```

用 python 来注册事件方法类似，采用 src/nox/lib/core.py 中的 register_handler 接口，例如

```
def handler(self):
    return CONTINUE

def install(self):
```

```
self.register_handler (Packet_in_event.static_get_name(), handler)
```

4.6.2.4 发布事件

所有的应用采用如下的方法来创建和发布事件。

```
void post(Event*) const;
```

例如

```
post(new Flow_in_event(flow, *src, *dst, src_dl_authed, src_nw_authed,
dst_dl_authed, dst_nw_authed, pi));
```

使用 python 程序是类似的，例如

```
e = Link_event(create_datapathid_from_host(linktuple[0]),
create_datapathid_from_host(linktuple[2]), linktuple[1], linktuple[3], action)
self.post(e)
```

4.6.2.5 发布计时器

在 nox app 中，运行是事件驱动的，nox 提供了计时器的方法来让事件在指定时间后执行。这一功能同样是通过发布（post）来实现的，格式如下

```
Timer post(const Timer_Callback&, const timeval& duration) const;
```

例如，让某个事件每一秒钟都运行一次，可以由如下 c++ 代码实现。

```
void timer(){
    using namespace std;
    cout << "One second has passed " << endl;
    timevale tv={1,0}
    post(boost::bind(&Example::timer, this), tv);
}

timevale tv={1,0}
post(boost::bind(&Example::timer, this), tv);
```

相应的 python 代码为

```
def timer():
    print 'one second has passed'
    self.post_callback(1, timer)

post_callback(1, timer)
```

添加新的功能后，需要注意重新编译和安装 nox。

4.6.3 开发例程

- 检查环境。

确保 NOX 已经正确安装并配置好相关环境。

```
git clone git://noxrepo.org/nox noxtutorial
cd noxtutorial
./boot.sh
mkdir build
cd build
../configure
make -j
```

编译完成后测试是否编译成功。

```
cd src
./nox_core -v -i ptcp: switch
```

- 创建新的 C/C++ 组件。
- 1) 进入想要放置组件子目录的目录，例如 src/nox/netapps。利用 nox-new-c-app.py（位于 src/script/，默认在系统路径中）创建组件子目录，并添加进 configure.ac.in。本步骤也可以手动执行。

```
nox-new-c-app.py myApp
```

- 2) 在 src 目录下重新编译 NOX 平台。

```
./boot.sh
cd build
../configure
make -j
```

3) 编译通过后，新的组件已经被添加了，可以通过下面的命令使用。

```
./nox_core -v -i ptcp: myApp
```

第4.7节 GUI

nox 提供了一套 gui 来进行网络的可视化和监控，也可以作为用户交互的接口。

4.7.1 运行 GUI

4.7.1.1 启动 GUI

首先需要安装 QT 依赖，

```
apt-get install python-qt4 python-simplejson
```

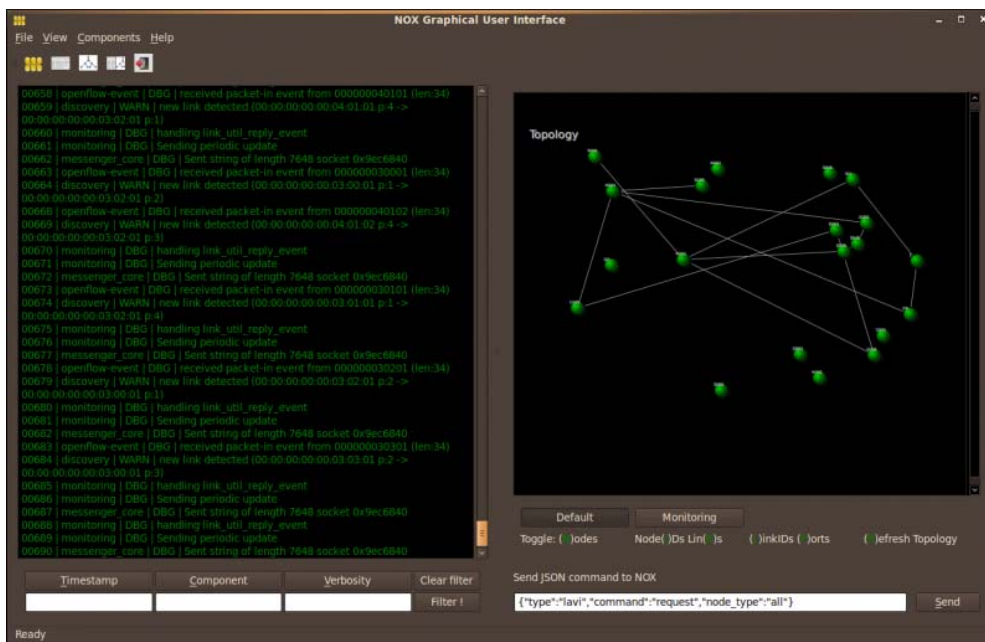
进入 src 目录，通过下面的命令启动 nox gui

```
./nox-gui.py
```

会自动尝试连接到 nox。如果 nox 运行在远程主机上，则需要执行

```
./nox-gui.py <NOX IP>
```

启动成功后，打开如图表 4-4 所示界面，其中左侧是调试信息，右侧是拓扑信息。



图表 4-4 nox gui 启动界面

4.7.1.2 界面

界面主要包括日志（log）信息和拓扑信息。

日志界面打印 nox 相关的输出信息，可以通过自定义过滤来分析信息。

拓扑界面显示通过 messenger 组件获取来的拓扑信息，用户可以进行交互。在节点上右键可以获取一些交换机信息。

4.7.2 扩展 GUI

创建新的视图

首先需要定义视图名称，例如

```
self.name = "STP"
```

添加按钮和相关的响应事件。

```
# Add custom view buttons
infoBtn = QtGui.QPushButton('What is STP?')
self.connect(infoBtn, QtCore.SIGNAL('clicked()'), self.showInfo)
self.buttons.append(infoBtn)
```

在 topology.py 文件最顶部，添加

```
from views.sample_routing import Sample_Routing_View
```

在 TopoWidget 类的 constructor 中，添加

```
self.stp_view = STP_View(self)
self.views[self.stp_view.name] = self.stp_view
```

在 backend 部分顶部，添加

```
from nox.coreapps.messenger.pyjsonmsgevent import JSONMsg_event
```

在 install()中，添加如下部分，让组件接收来自 gui 的消息。

```
# Register for json messages from the gui
self.register_handler( JSONMsg_event.static_get_name(), \
                        lambda event: self.handle_jsonmsg_event(event))

# Subscribers for json messages
#(eg. self.subscribers["stp_ports"] = [guistream]
self.subscribers = {}
```

在 frontend，在 constructor 中注册处理来自 backend 的消息。

```
# Subscribe for stp_ports
msg = {}
msg ["type"] = "spanning_tree"
msg ["command"] = "subscribe"
msg ["msg_type"] = "stp_ports"
self.topologyInterface.send( msg )
```

并将其绑定到句柄。

```
# Connect signal raised when msg arrives from backend to handler
self.topologyInterface.spanning_tree_received_signal.connect( \
    self.got_json_msg )
```

最后在 communication.py 文件中，定义新的信号，例如

```
# Signal used to notify STP view of new msg
spanning_tree_received_signal = QtCore.pyqtSignal(str)
```

定义监听器，处理消息类型

```
# Signal used to notify STP view of new msg
spanning_tree_received_signal = QtCore.pyqtSignal(str)
```

第4.8节 相关工作

NOX 项目主页在 <http://noxrepo.org>。

类似的项目包括 SANE、Ethane、Maestro、onix、difane 等。其中 onix 是面向产品的分布式 NOX 的一个实现，同样由 nicira 公司维护。

第5章 Mininet

第5.1节 概述

Stanford 大学 Nick McKeown 的研究小组基于 Linux Container 架构，开发出了这套进程虚拟化的平台。在 mininet 的帮助下，你可以轻易的在自己的笔记本上测试一个软件定义网络（software-defined Networks），对基于 Openflow、Open vSwitch 的各种协议等进行开发验证，或者验证自己的想法。最令人振奋的是，所有的代码几乎可以无缝迁移到真实的硬件环境中。在实验室里，一行命令就可以创建一个支持 SDN 的任意拓扑的网络结构，并可以灵活的进行相关测试，验证了设计的正确后，又可以轻松部署到真实的硬件环境中。目前 Mininet 已经作为官方的演示平台对各个版本的 Openflow 协议进行演示和测试。

Mininet 项目的首页在 <http://www.openflowswitch.org/foswiki/bin/view/OpenFlow/Mininet>，目前主要维护者为 Bob Lantz 跟 Brandon Heller。讨论组地址为 mininet-discuss@lists.stanford.edu。

第5.2节 主要特性

- mininet 作为一个轻量级软定义网络研发和测试平台，其主要特性包括
- 支持 Openflow、OpenvSwitch 等软定义网络部件
- 方便多人协同开发
- 支持系统级的还原测试支持复杂拓扑、自定义拓扑
- 提供 python API
- 很好的硬件移植性（Linux 兼容），结果有更好的说服力
- 高扩展性，支持超过 4096 台主机的网络结构

第5.3节 镜像获取和使用

5.3.1 获取镜像

官方网站已经提供了配置好相关环境的基于 Debian Lenny 的虚拟机镜像，下载地址为 <http://openflowswitch.org/downloads/OpenFlowTutorial-081910.vmware.zip>，压缩包大小为 700M 左右，解压后大小为 2.1G 左右。虚拟机镜像格式为 vmware 的 vmdk，可以直接使用 vmware workstation 或者 virtualbox 等软件打开。如果使用 QEMU 和 KVM 则需要先进行格式转换。后面我们就以这个虚拟 os 环境为例，介绍 mininet 的相关功能。

如果使用 virtualbox 进行加载，需要注意

尽量使用最新版本，host 操作系统需要支持 paef，并在 virtualbox 中打开 paef 支持。

5.3.2 使用镜像

默认用户名密码均为 `openflow`，建议通过本地利用 `ssh` 登录到虚拟机上使用（可以设置自动登录并将 `X` 重定向到本地），比较方便操作。

注意事项：

建议将 `guest` 主机采用 `bridge` 方式联网，以获取 `host` 可见的独立 IP；也可采用为 `guest` 配置两块网卡方式，一块采用 `NAT`，一块采用 `host-only`，但 `host-only` 的网卡可能无法自动 `dhcp` 到地址，需要手动配置（`ifconfig eth1 ip/mask`）将 `host` 机 `.ssh` 目录下 `id_rsa.pub` 复制到 `guest` 机的 `.ssh` 目录下，并写入 `authorized_keys`，实现自动认证。

5.3.3 更新

进入 `mininet/mininet` 目录，执行

```
git pull
```

获取最新版本代码，默认是 `master` 分支，然后返回上层目录，执行

```
make install
```

第5.4节 简单测试

5.4.1 创建网络

`mininet` 的操作十分简单，启动一个小型测试网络只需要下面几个步骤。

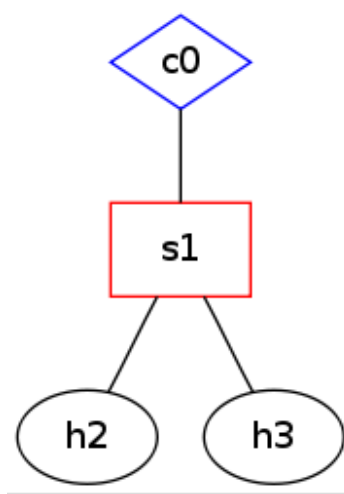
登录到虚拟机命令行界面，打开 `wireshark`，使其后台运行，命令为

```
sudo wireshark &
```

启动 `mininet`，命令为 `sudo mn`，则默认创建如下图所示的网络拓扑；

经过短暂的等待即可进入以 `mininet>` 引导的命令行界面。

好了，从现在开始，我们就拥有了一个 1 台控制节点(`controller`)、一台交换(`switch`)、两台主机(`host`)的网络，并且用 `wireshark` 进行观测。下面进行几项简单的测试。拓扑结构参见图表 5-1。



图表 5-1 简单测试拓扑

注意：使用 wireshark 检测 lo 网卡，并通过过滤 of 协议可以看到 OpenFlow 的网包。

5.4.2 查看信息

查看全部节点：

```
mininet> nodes
available nodes are:
c0 h2 h3 s1
```

查看链路信息：

```
mininet> net
s1 <-> h2-eth0 h3-eth0
```

输出各节点的信息：

```
mininet> dump
c0: IP=127.0.0.1 intfs= pid=1679
s1: IP=None intfs=s1-eth1,s1-eth2 pid=1682
h2: IP=10.0.0.2 intfs=h2-eth0 pid=1680
h3: IP=10.0.0.3 intfs=h3-eth0 pid=1681
```

5.4.3 对节点进行单独操作

如果想要对某个节点的虚拟机单独进行命令操作，也十分简单，格式为 `node cmd`。例

如查看交换机 s1 上的网络信息,我们只需要在执行的 `ifconfig` 命令前加上 s1 主机标志即可,即 `s1 ifconfig`, 同样, 如果我们想用 `ping` 3 个包的方法来测试 h2 跟 h3 之间连通情况, 只需要执行 `h2 ping -c 3 h3` 即可。得到的结果为

```
mininet> h2 ping -c 3 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=7.19 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.239 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.136 ms
— 10.0.0.3 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 0.136/2.523/7.194/3.303 ms
```

在本操作执行后, 可以通过 `wireshark` 记录查看到创建新的流表项的过程, 这也是造成第一个 `ping` 得到的结果偏大的原因。更简单的全网络互 `ping` 测试命令是 `pingall`, 会自动所有主机节点逐对进行 `ping` 连通测试。

第5.5节 常用操作

5.5.1 快捷测试

除了 `cli` 的交互方式之外, `mininet` 还提供了更方便的自动执行的快捷测试方式, 其格式为 `sudo mn --test cmd`, 即可自动启动并执行 `cmd` 操作, 完成后自动退出。

例如 `sudo mn --test pingpair`, 可以直接对主机连通性进行测试, `sudo mn --test iperf` 启动后直接进行性能测试。用这种方式很方便直接得到实验结果。

5.5.2 自定义拓扑

`mininet` 提供了 `python` api, 可以用来方便的自定义拓扑结构, 在 `mininet/custom` 目录下给出了几个例子。例如在 `topo-2sw-2host.py` 文件中定义了一个 `mytopo`, 则可以通过 `--topo` 选项来指定使用这一拓扑, 命令为

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

同样的, 我们可以通过下面的 `python` 脚本来完成对一个 2 层 `tree` 拓扑网络的测试

```
from mininet.net import Mininet from mininet.topolib import TreeTopo tree4 =
TreeTopo(depth=2,fanout=2) net = Mininet(topo=tree4) net.start() h1, h4 =
net.hosts[0], net.hosts[3] print h1.cmd('ping -c1 %s' % h4.IP()) net.stop()
```

5.5.3 使用友好的 mac 编号

默认情况下，主机跟交换机启动后分配的 MAC 地址是随机的，这在某些情况下不方便查找问题。可以使用--mac 选项，这样主机跟交换机分配到的 MAC 地址跟他们的 ID 是一致的，容易通过 MAC 地址较快找到对应的节点。

5.5.4 使用 XTerm

为了能够正确使用 xterm，我们需要做些准备工作。在这里推荐利用远程方式登录到 OpenflowVM。

5.5.4.1 客户端

对于自带 X 的 linux 主机，无需配置 X。

如果客户端是 windows 主机，需要先在 windows 机器上安装 Xserver (Xming) 下载地址：<http://sourceforge.net/projects/xming/files/Xming/6.9.0.31/Xming-6-9-0-31-setup.exe>。

如果你使用 secureCRT 远程登录到 OpenflowVM，需要在“会话选项->SSH2->密钥交换”下，取消“diffie-hellman-group14”和“diffie-hellman”选择；同时在“远程/X11”下，选择转发“X11 数据包 (F)”；点击确定，开启 Xming，使用如下命令远程到 Openflow VM 即可。

```
ssh -X openflow@[Guest IP here]
```

如果你使用 puTTY 远程登录到 OpenflowVM:

点击“puTTY->Connection->SSH->X11”，选择“X11 forwarding->Enable X11 forwarding”；开启 Xming，点击 windows 开始按钮，在运行栏输入“cmd”，打开终端，输入“cd <dir>”切换到保存 puTTY 的目录下；使用下面的命令远程登录到 openflow VM。

```
putty.exe -X openflow@[Guest IP here]
```

5.5.4.2 主机端

通过使用-x 参数，mn 在启动后会在每个节点上自动打开一个 XTerm，方便某些情况下的对多个节点分别进行操作。命令为

```
sudo mn -x
```

在进入 mn cli 之后，也可以使用 xterm node 命令指定启动某些节点上的 xterm，例如

分别启用 s1 跟 h2 上的 xterm，可以用

```
xterm s1 h2
```

5.5.5 链路操作

在 mn cli 中，使用 link 命令，禁用或启用某条链路，格式为 link node1 node2 up/down，例如临时禁用 s1 跟 h2 之间的链路，可以用

```
link s1 h2 down
```

5.5.6 指定交换机跟控制器类型

通过--switch 选项跟--controller 选项可以分别指定采用哪种类型的交换机跟控制器，例如使用用户态的交换

```
sudo mn --switch user
```

使用 OpenvSwitch

```
sudo mn --switch ovsk
```

使用 NOX pyswitch

1) 首先确保 nox 运行

```
cd $NOX_CORE_DIR  
./nox_core -v -i ptcp:
```

然后 ctrl-c 杀死 nox 进程

2) 然后指定 nox 交换机

```
sudo -E mn --controller nox_pysw
```

注意：通过-E 选项来保持预定义的环境变量（此处为 NOX_CORE_DIR）。

5.5.7 名字空间

默认情况下，主机节点有用独立的名字空间（namespace），而控制节点跟交换节点都在根名字空间（root namespace）中。如果想要让所有节点拥有各自的名字空间，需要添加

--innamespace 参数，即启动方式为 `sudo mn --innamespace`

注意：为了方便测试，在默认情况下，所有节点使用同一进程空间，因此，在 h2 跟 h3 或者 s1 上使用 ps 查看进程得到的结果是一致的，都是根名字空间中的进程信息。

5.5.8 启动参数总结

```
-h, --help          show this help message and exit
--switch=SWITCH      [kernel user ovsk]
--host=HOST          [process]
--controller=CONTROLLER [nox_dump none ref remote nox_pysw]
--topo=TOPO          [tree reversed single linear minimal],arg1,arg2,...argN
-c, --clean          clean and exit
--custom=CUSTOM      read custom topo and node params from .py file
--test=TEST          [cli build pingall pingpair iperf all iperfudp none]
-x, --xterms         spawn xterms for each node
--mac                set MACs equal to DPIDs
--arp                set all-pairs ARP entries
-v VERBOSITY, --verbosity=VERBOSITY [info warning critical error debug output]
--ip=IP              [ip address as a dotted decimal string for aremote controller]
--port=PORT          [port integer for a listening remote controller]
--innamespace        sw and ctrl in namespace?
--listenport=LISTENPORT [base port for passive switch listening controller]
--nolistenport       don't use passive listening port
--pre=PRE            [CLI script to run before tests]
--post=POST          [CLI script to run after tests]
```

5.5.9 常用命令总结

```
help  默认列出所有命令文档，后面加命令名将介绍该命令用法
dump  打印节点信息
gterm  给定节点上开启 gnome-terminal。注：可能导致 mn 崩溃
xterm  给定节点上开启 xterm
intfs  列出所有的网络接口
iperf  两个节点之间进行简单的 iperf TCP 测试
iperfudp  两个节点之间用制定带宽 udp 进行测试
net  显示网络链接情况
noecho 运行交互式窗口，关闭回应（echoing）
pingpair 在前两个主机之间互 ping 测试
source 从外部文件中读入命令
```

```
dpctl  在所有交换机上用 dptcl 执行相关命令，本地为 tcp 127.0.0.1:6634
link   禁用或启用两个节点之间的链路
nodes  列出所有的节点信息
pingall 所有 host 节点之间互 ping
py     执行 python 表达式
sh     运行外部 shell 命令
quit/exit 退出
```

5.5.10 其他操作

执行 `sudo mn -c` 会进行清理配置操作，适合故障后恢复。

执行 `exit` 会退出 `mininet` 的 `cli`，同时给出运行时间统计。

`py cmd` 使用 `python` 来执行 `cmd`。

测试 `mininet` 启动后立刻关闭的时间可以用 `sudo mn --test none`

第5.6节 高级功能

下面我们通过一个具体管理 `of switch` 的例子来介绍一些比较高级的命令。

首先，启动 `vm`，然后执行

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

生成一个小的网络，三台主机连到一台交换机上，交换机为 `ovs` 交换机，指定 `remote` 控制器（默认为本地）。

5.6.1 dpctl

执行

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

可以查看到交换机的端口等基本情况，其中 `tcp` 端口 `6634` 是默认的交换机监听端口。
执行

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

可以看到更详细的流表信息。

此时，流表为空，执行 `h2 ping h3` 无法得到响应。因此我们需要通过 `dpctl` 手动添加流表项，实现转发。

命令为

```
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
```

此时查看流表可以看到新的转发信息，同时可以在 h2 和 h3 之间 ping 通。

5.6.2 控制器

通过执行

5.6.3 交换机与控制器交互

我们可以启动一个简单的控制器，默认没有任何流表项，仅仅作为一台带学习功能的交换机。控制器默认监听端口是 6633。以下控制器与交换机之间的消息交互过程，可以通过 wireshark，配置 of 过滤器观察到交换机跟控制器之间的交互消息。参见表格 5-1。

表格 5-1 交换机和控制器之间的交互消息

Message	Type	Description
Hello	Controller->Switch	following the TCP handshake, the controller sends its version number to the switch.
Hello	Switch->Controller	the switch replies with its supported version number.
Features Request	Controller->Switch	the controller asks to see which ports are available.
Set Config	Controller->Switch	in this case, the controller asks the switch to send flow expirations.
Features Reply	Switch->Controller	the switch replies with a list of ports, port speeds, and supported tables and actions.
Port Status	Switch->Controller	enables the switch to inform that controller of changes to port speeds or connectivity. Ignore this one, it appears to be a bug.

同样，我们可以用 wireshark 观察到当第一次有 ping 包从 h2 发到 h3 时，控制器如何自动添加相应的表项到交换机。wireshark 相应的过滤器为

```
of && (of.type != 3) && (of.type != 2)
```

相关的消息过程参考表格 5-2。

表格 5-2 添加流表项的消息过程

Message	Type	Description
Packet-In	Switch->Controller	a packet was received and it didn't match any entry in the switch's flow table, causing the packet to be sent to the controller.
Packet-Out	Controller->Switch	controller send a packet out one or more switch ports.
Flow-Mod	Controller->Switch	instructs a switch to add a particular flow to its flow table.
Flow-Expired	Switch->Controller	a flow timed out after a period of inactivity.

5.6.4 使用 NOX

首先确定没有其他控制器在运行（占据 6633 端口）

```
sudo killall controller
```

同样的，启动 mininet

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

然后启动 nox，默认路径为~/noxcore/build/src，重新打开一个 ssh 终端执行

```
./nox_core -v -i ptcp: pytutorial
```

会自动打开运行 tutorial 应用的 nox，打印出详细的调试信息，并监听 6633 端口。直到打印出类似如下信息，说明交换机已经成功连接到 nox。

```
00039|nox|DBG:Registering switch with DPID = 1
```

通过互 ping 测试，各个主机连通，此时 switch 等同于一个 hub。

然后通过修改~/noxcore/src/nox/coreapps/tutorial/pytutorial.py 中代码，让 nox 工作成一个带学习功能的交换机。相关命令参考 ofinclude 代码，以及 nox 对各个包的解析代码目录：

~/noxcore/src/nox/lib/packet/。通过编写 nox 程序，我们可以让交换机的行为更加智能化、复杂化。为了测试我们编写的 nox 程序，我们可以使用 cbench 来进行测试。

5.6.5 多条配置命令

可以写到一个文件中，用 mn 直接调用。例如脚本文件名为 my_cli_script 则可以执行

```
mininet> source my_cli_script
```

或者

```
# mn --pre my_cli_script
```

第5.7节 代码分析

进入 mininet 目录下，主要包括 7 个子目录(bin、build、custom、dist、example、mininet、util)和 8 个文件(doxyen.cfg、INSTALL、LICENSE、Makefile、mnexec、mnexec.c、README、setup.py)。

5.7.1 bin 子目录

包括 mn 和 mnexec 两个可执行文件。

● mn 文件

py 代码文件，是程序执行的解释器，解析参数，例如-h 参数或--custom 参数等定义 MininetRunner 类，为整个测试创建基础平台。

主要执行步骤有

parseArgs() //解析参数

setup() //调用 mininet.net.init()来校验运行环境配置

begin()

//执行给定参数，包括创建拓扑、地址分配等

//调用 mininet.net.Mininet()创建网络平台 mn

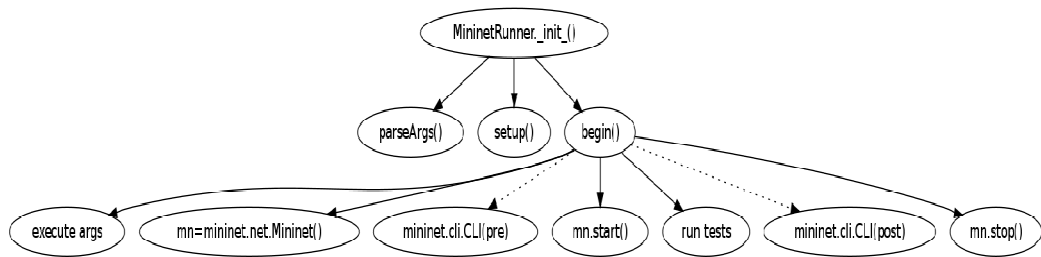
//调用 mininet.cli.CLI()创建 CLI 对象

//调用 mn.start()启动网络

//执行指定的测试命令，默认为 cli，即调用 CLI(mn)进入交互环境

//执行结束后调用 mn.stop()退出

该文件代码的主要结构如图表 5-2 所示。



图表 5-2 MininetRunner 类结构示意图

- mnexec 文件

mininet 执行文件。完成一些 python 代码执行起来比较慢或者难以实现的功能，包括关闭文件描述、使用 setsid 从控制 tty 剥离、在网络空间运行、打印 pid 等。

5.7.2 mininet 子目录

mininet 相关实现的主代码目录，包括若干.py 源文件。

- __init__.py 文件

python 代码导入控制文件。

- clean.py

提供两个函数

sh(cmd) 调用 shell 来执行 cmd;

cleanup() 清理实验可能留下的残余进程或临时文件。

- cli.py

定义 CLI 类，在 mn 运行后提供简单的命令行接口，解析用户键入的各项命令。例如

```
mininet> h1 ping h2
```

- log.py

实现日志记录等功能，定义三个类 MininetLogger、Singleton 和 StreamHandlerNoNewLine。

- moduledeps.py

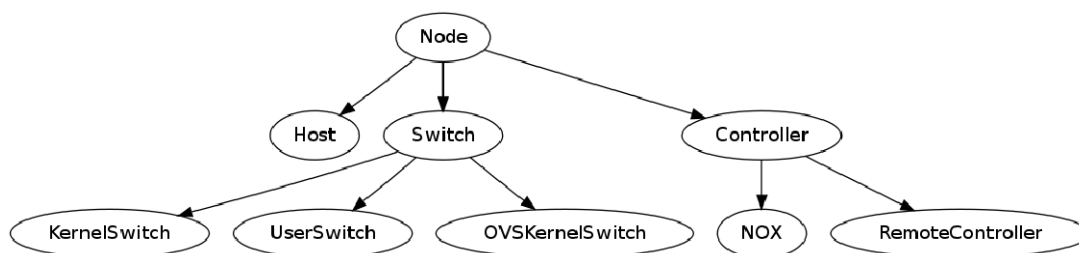
模块依赖相关处理。定义 5 个函数：lsmod()、rmmod(mod)、modprobe(mod)、moduleDeps(subtract=None, add=None)、pathCheck(*args,**kwargs)。

- net.py

mn 网络平台的主类。完成包括节点管理、基本测试等功能。

- node.py

实现网络节点功能的几个类。包括主机、交换机、控制器等，各个类的集成关系如图表 5-3 所示。



图表 5-3 node.py 类继承关系

各个类功能为

Host: 等同于 Node。

KernelSwitch: 内核空间的交换机, 仅能在 root 名字空间中执行。

UserSwitch: 用户空间的交换机。

OVSKernelSwitch: Open vSwitch 的内核空间交换机, 仅能在 root 名字空间中执行。

NOX: NOX 控制器。

RemoteController: mininet 外的控制, 通过指定 IP 地址等进行连接。

- term.py

实现在每个节点上运行 terminal 的功能。

- toplib.py

实现可能需要的拓扑结构。

包括一个 TreeTopo 类和 TreeNet 函数。前者提供给定深度和 fanout 的树状结构, 后者返回一个树状结构的 Mininet 网络。

- topo.py

创建网络拓扑的主文件。包括一些常见拓扑, 例如线型、星型等。

- util.py

mininet 的一些辅助功能, 包括调用 shell 执行命令, 网络地址格式的解析等。

5.7.3 custom 子目录

存放一些自定义的拓扑文件, mininet 可以通过 --custom mytopo 命令进行调用。

5.7.4 examples 子目录

包括一些利用 mininet 进行测试的用例。

- baresshd.py

测试运行 sshd。

- consoles.py

生成多个控制台。

- controller.py

创建一个多个控制器的网络。

- emptynet.py

创建无拓扑网络并添加节点。

- `hwintf.py`

添加新的硬件接口。

- `linearbandwidth.py`

创建自定义拓扑，并进行测试。

- `miniedit.py`

通过图形界面创建网络。

- `multitest.py`

创建拓扑并进行多个测试。

- `scratchnet.py`

- `scratchnetuser.py`

使用底层的 `api` 创建网络。

- `sshd.py`

在每个创建的 `host` 上运行 `sshd`。

- `tree1024.py`

创建 1024 个节点的树状结构。

- `treeping64.py`

创建 64 个节点的树状结构并测试。

- `util` 子目录

包括 `debian` 的安装脚本，`of` 的补丁等。

5.7.5 其他文件

- `doxygen.cfg`

用 `doxygen` 生成代码文档时的配置文件。

- `INSTALL`

说明版权信息。

- `LICENSE`

说明安装步骤。

- `README`

简要介绍功能等。

- `Makefile`

编译、安装配置文件。

- `mnexec`

同 `bin` 目录下的 `mnexec` 文件。

- `mnexec.c`

`mnexec` 的 C 源代码文件。

- `setup.py`

安装脚本。

第6章 相关项目

第6.1节 Openflow

项目主页

<http://openflowswitch.org/>

of 使用例程

http://www.openflow.org/wk/index.php/OpenFlow_Tutorial

第6.2节 OpenvSwitch

项目主页

<http://openvswitch.org/>

第6.3节 NOX

项目主页

<http://noxrepo.org>

项目 wiki

<http://noxrepo.org/noxwiki/>

第6.4节 Mininet

Mininet 使用例程

<http://www.openflow.org/foswiki/bin/view/OpenFlow/MininetWalkthrough>