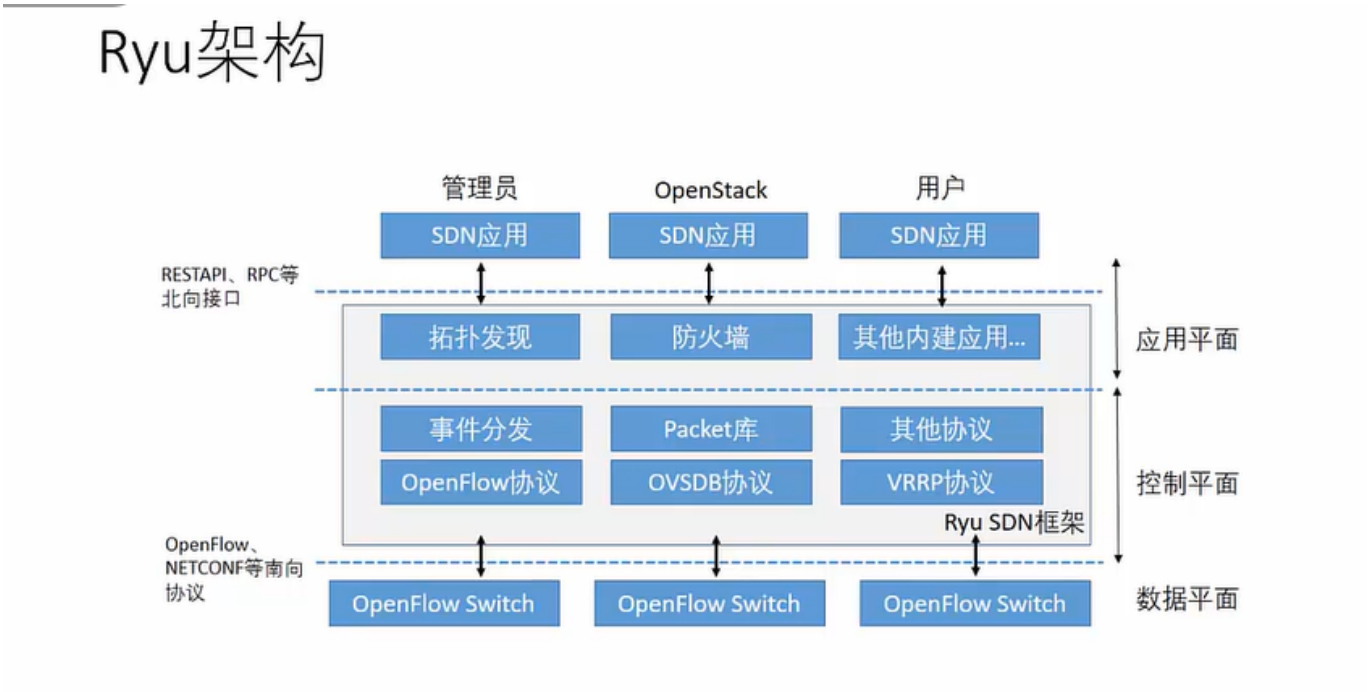
RYU

Ryu是一种基于Python语言的软件定义网络（SDN）控制器，它提供了一个开放的应用程序接口（API），使网络管理员和开发人员能够轻松地编写新的网络控制应用程序来进行网络流量的控制和管理。Ryu通过OpenFlow协议与网络交换机通信，可以对网络设备进行配置、监控和管理。Ryu是一个开源项目，由日本NTT实验室开发和维护，被广泛应用于SDN应用程序的开发和部署。

1.使用RYU做mininet的控制器

Ryu的架构



1.1 RYU介绍与安装

RYU是一款基于python的控制器，你可用Ryu实现各种想要实现的网络功能，它可以下发或接收流表进行各种路由运算。对流表的控制无外呼就是增删改查。

1.2 RYU的特点

Ryu在SDN控制器领域有以下几个特点：

- **1.简单易用**：Ryu采用Python语言编写，易于学习和使用，提供了丰富的API和文档，使开发人员能够快速地构建自己的SDN应用程序。
- **2.高度可扩展**：Ryu提供了一个基于插件的架构，使开发人员可以方便地扩展和定制SDN应用程序的功能。

- 3.**支持多种OpenFlow协议版本**：Ryu支持多种OpenFlow协议版本，包括1.0、1.3和1.4版本，使其可以兼容不同厂家的网络交换机。
- 4.**高性能**：Ryu采用异步I/O模型和事件驱动的设计，提供了高性能的网络流量控制和管理能力。
- 5.**开放源代码**：Ryu是一个开源项目，可以自由获取代码并进行修改和定制，有助于推动SDN技术的发展和普及。

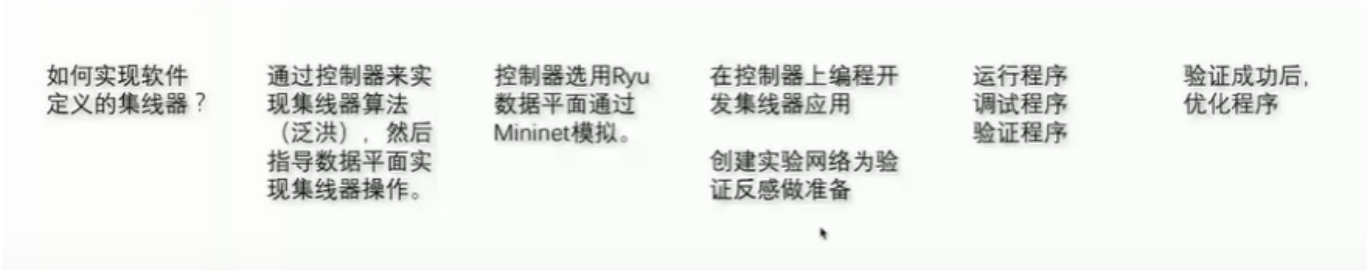
1.3 RYU功能

- 1.**网络拓扑发现和管理**：RYU可以发现和管理SDN网络中的拓扑结构，包括交换机、主机、链路和路径等信息。
- 2.**网络流量控制和管理**：RYU可以控制网络流量的转发和策略，支持流表、QoS、ACL等功能，可以实现网络流量的精确控制和管理。
- 3.**网络安全和监控**：RYU可以对网络流量进行监控和分析，能够识别和处理恶意流量和攻击行为，提高网络安全性。
- 4.**网络服务质量（QoS）保障**：RYU可以实现基于流量的服务质量保障，包括带宽限制、拥塞控制、流量分类等功能。
- 5.**网络编程和应用开发**：RYU提供了丰富的API和SDK，支持Python编程语言，可以方便地开发和部署SDN应用程序，如基于SDN的网络监控、负载均衡、流量控制和优化等。

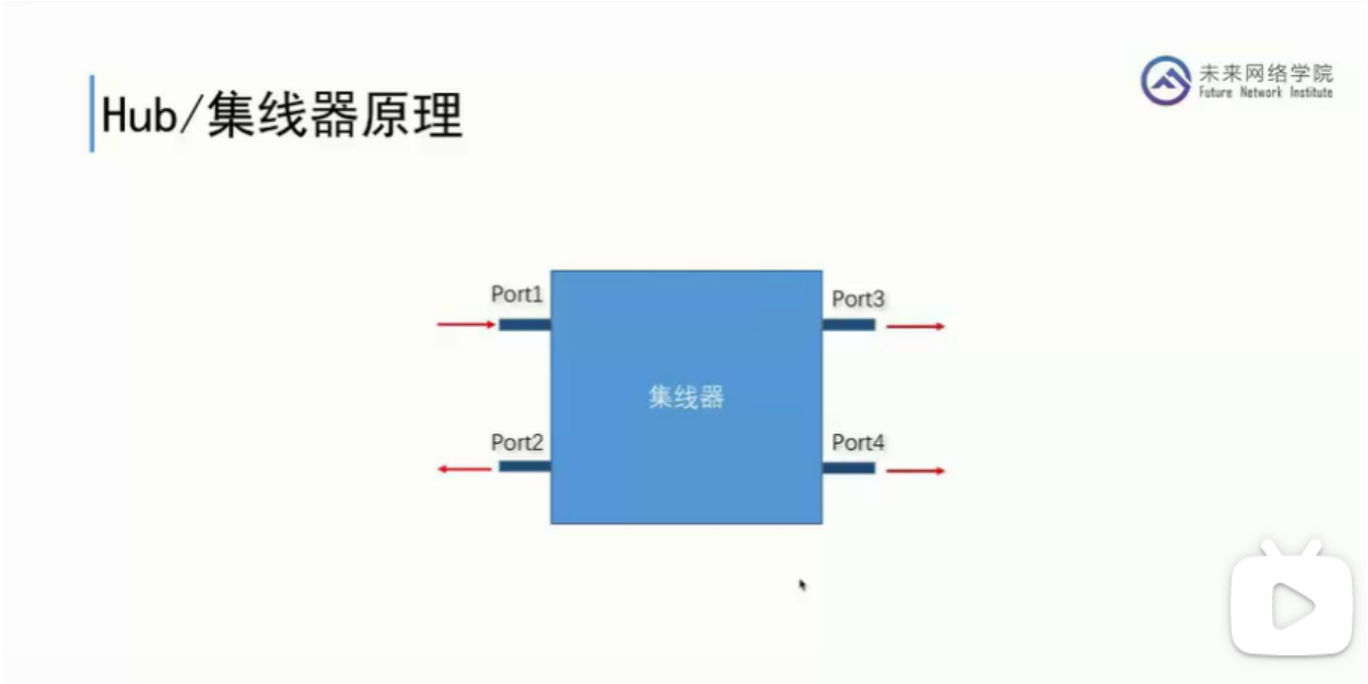
总之，RYU可以实现SDN网络的灵活和高效管理和控制，提高网络性能和安全性，同时也为SDN应用程序的开发和部署提供了便利。

1.4 Hub+Learning Switch应用开发

1.4.1 编程



(1) 集线器的原理：



（2）集线器实施部署--ryu控制器api的学习和使用

```python
#实现集线器功能
from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER,CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls



#定义一个类hub，继承app_manager，位于ryu下的base内，
#版本选择openflow1.3，然后初始化操作。
class Hub(app_manager.RyuApp):
    "集线器"

    OFP_VERSIONS=[ofproto_v1_3.OFP_VERSION]#openflow版本

    def __init__(self,*args,**kwargs):
        super(Hub,self).__init__(*args,**kwargs)


#接下来，需要定义packet in函数，用来处理交换机和控制器的流表交互，
#在执行之前需要先对packetin事件进行监听。
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,CONFIG_DISPATCHER)
    #注册事件
    def switch_features_handler(self,ev):#处理交换机连接

    # 这几句是对数据结构进行解析，是一种固定的形式，可以记住就行
        datapath=ev.msg.datapath#数据平面的通道,网桥
        ofproto = datapath.ofproto#版本
        ofp_parser = datapath.ofproto_parser

        #install the table flow entry
        match=ofp_parser.OFPMatch()#匹配域
        actions=[ofp_parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                            ofproto.OFPCML_NO_BUFFER)]
                                            #发送入端口，buffer_id

        self.add_flow(datapath,0,match,actions)
        #默认流表项，将消息发送到控制器



#ryu的数据平面是由若干网元（Network Element）组成，每个网元包含一个或
#多个SDN数据路径（SDN Datapath）.SDN Datapath是逻辑上的网络设备，负责转发
#和处理数据无控制能力，一个SDN DataPath包含控制数据平面接口
#（Control Data Plane Interface,CDPI）、代理、转发引擎（Forwarding Engine）表
#和处理功能（Processing Function）SDN数据面（转发面）的关键技术:
#对数据面进行抽象建模。
#对于添加流表，可以将其单独拿出了写一个函数。
    def add_flow(self,datapath,priority,match,actions):
```

```
        #add a flow entry and install it into datapath
        ofproto =datapath.ofproto
        ofp_parser =datapath.ofproto_parser

        #construct a flow_mod msg and send it
        inst = [ofp_parser.OFPInstructionActions(
                                            ofproto.OFPIT_APPLY_ACTIONS,
                                            actions)]
        mod = ofp_parser.OFPFlowMod(datapath=datapath,priority=priority,
                                    match=match,instructions=inst)
        datapath.send_msg(mod)#发送消息

    @set_ev_cls(ofp_event.EventOFPPacketIn,MAIN_DISPATCHER)#使用装饰器，监听
    def packet_in_handler(self,ev):#packet_in数据包进入，handler解决
        msg=ev.msg
        datapath =msg.datapath
        ofproto=datapath.ofproto
        ofp_parser = datapath.ofproto_parser #协议项的数据结构
        in_port = msg.match['in_port']#匹配域

        #instruct a flow entry
        match = ofp_parser.OFPMatch()
        actions =[ ofp_parser.OFPActionOutput(ofproto.OFPP_FLOOD) ]
        #把数据包发送到全部端口,泛洪（除了正在使用的端口）

        #install a flow mod to avoid packet_in next time
        self.add_flow(datapath,1,match,actions)


        #处理当下数据包
        out = ofp_parser.OFPPacketOut(datapath=datapath,
                                      buffer_id=msg.buffer_id,
                                    in_port=in_port,actions=actions)
                                      #buffer缓冲区，数据包存储

        datapath.send_msg(out)


rt=in_port,actions=actions)#buffer缓冲区，数据包存储

        datapath.send_msg(out)
```

（3）ryu/ryu目录下的主要目录内容。

base

base中有一个非常重要的文件：app_manager.py，其作用是RYU应用的管理中心。用于加载RYU应用程序，接受从APP发送过来的信息，同时也完成消息的路由。

其主要的函数有app注册、注销、查找、并定义了RYUAPP基类，定义了RYUAPP的基本属性。包含name, threads, events, event_handlers和observers等成员，以及对应的许多基本函数。如：start(), stop()等。

这个文件中还定义了AppManager基类，用于管理APP。定义了加载APP等函数。不过如果仅仅是开发APP的话，这个类可以不必关心。

controller

controller文件夹中许多非常重要的文件，如events.py, ofp_handler.py, controller.py等。其中controller.py中定义了OpenFlowController基类。用于定义OpenFlow的控制器，用于处理交换机和控制器的连接等事件，同时还可以产生事件和路由事件。其事件系统的定义，可以查看events.py和ofp_events.py。

在ofp_handler.py中定义了基本的handler（应该怎么称呼呢？句柄？处理函数？），完成了基本的如：握手，错误信息处理和keep alive 等功能。更多的如packet_in_handler应该在app中定义。

在dpset.py文件中，定义了交换机端的一些消息，如端口状态信息等，用于描述和操作交换机。如添加端口，删除端口等操作。

lib

lib中定义了我们需要使用到的基本的数据结构，如dpid, mac和ip等数据结构。在lib/packet目录下，还定义了许多网络协议，如ICMP, DHCP, MPLS和IGMP等协议内容。而每一个数据包的类中都有parser和serialize两个函数。用于解析和序列化数据包。

ofproto

在这个目录下，基本分为两类文件，一类是协议的数据结构定义，另一类是协议解析，也即数据包处理函数文件。如ofproto_v1_0.py是1.0版本的OpenFlow协议数据结构的定义，而ofproto_v1_0_parser.py则定义了1.0版本的协议编码和解码。具体内容不赘述，实现功能与协议相同。

topology

包含了switches.py等文件，基本定义了一套交换机的数据结构。event.py定义了交换上的事件。dumper.py定义了获取网络拓扑的内容。最后api.py向上提供了一套调用topology目录中定义函数的接口。

contrib

这个文件夹主要存放的是开源社区贡献者的代码

cmd

定义了RYU的命令系统

services

完成了BGP和vrrp的实现。

tests

tests目录下存放了单元测试以及整合测试的代码

（4）代码讲解

首先，我们从ryu.base import app_manager，在前面我们也提到过这个文件中定义了RyuApp基类。我们在开发APP的时候只需要继承这个基类，就获得你想要的一个APP的一切了。

```
from ryu.base import app_manager


class Hub(app_manager.RyuApp):

    def __init__(self, *args, **kwargs):

        super(L2Switch, self).__init__(*args, **kwargs)
```

- ev.msg：每一个事件类ev中都有msg成员，用于携带触发事件的数据包
- msg.datapath：已经格式化的msg其实就是一个packet_in报文，**msg.datapath直接可以获得packet_in报文的datapath结构。**datapath用于描述一个交换网桥。也是和控制器通信的实体单元。

  Datapath类在RYU中极为重要，每当一个datapath实体与控制器建立连接时，就会实例化一个Datapath的对象。 该类中不仅定义了许多的成员变量用于描述一个datapath，还管理控制器与该datapath通信的数据收发
- datapath.send_msg()函数用于发送数据到指定datapath。通过datapath.id可获得dpid数据，在后续的教程中会有使用。
- datapath.ofproto对象是一个OpenFlow协议数据结构的对象，成员包含OpenFlow协议的数据结构，如动作类型OFPP_FLOOD。
- datapath.ofp_parser则是一个按照OpenFlow解析的数据结构。
- actions是一个列表，用于存放action list，可在其中添加动作.

- 通过ofp_parser类，可以构造构造packet_out数据结构。括弧中填写对应字段的赋值即可。
- 如果datapath.send_msg()函数发送的是一个OpenFlow的数据结构，RYU将把这个数据发送到对应的datapath。

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls


class L2Switch(app_manager.RyuApp):
    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser

        actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]

        out = ofp_parser.OFPPacketOut(
            datapath=datapath, buffer_id=msg.buffer_id,
                    in_port=msg.in_port,actions=actions)

        datapath.send_msg(out)
```

其中ofp_event完成了事件的定义，从而我们可以在函数中注册handler，监听事件，并作出回应。

packet_in_handler方法用于处理packet_in事件。@set_ev_cls修饰符用于告知RYU，被修饰的函数应该被调用。（翻译得有点烂这句）

set_ev_cls第一个参数表示事件发生时应该调用的函数，第二个参数告诉交换机只有在交换机握手完成之后，才可以被调用。

1.3.2 事件管理（Event handler ）

对于 Ryu 来说，接受到任何一个 OpenFlow 讯息即会产生一个相对应的事件。而 Ryu 应用程序则是必须实作事件管理以处理相对应发生的事件。

事件管理（Event Handler ）是一个拥有事件对象（Event Object ）做为参数，并且使用``ryu.controller.handler.set_ev_cls``修饰（Decorator ）的函数。

set_ev_cls 则指定事件类别得以接受讯息和交换器状态作为参数。

事件类别名称的规则为 `ryu.controller.ofp_event.EventOFP` + <OpenFlow 讯息名称 >，例如：在 Packet-In 讯息的状态下的事件名称为 `EventOFPPacketIn` 。详细的内容请参考 Ryu 的文件API 参考数据。对于状态来说，请指定下面列表的其中一项。

名称	说明
ryu.controller.handler.HANDSHAKE_DISPATCHER	交换 HELLO 讯息
ryu.controller.handler.CONFIG_DISPATCHER	接收 SwitchFeatures 讯息
ryu.controller.handler.MAIN_DISPATCHER	一般状态
ryu.controller.handler.DEAD_DISPATCHER	联机中断