

Ryu 的 REST API

Ryu 的 REST API

1. 课程说明
2. 学习方法
3. 本节内容简介
4. 推荐阅读
5. Ryu 的 APP 使用
 - 5.1 在外部启动 ryu，通过指定 ip 地址来连接
 - 5.2 直接指定 ryu 控制器
 - 5.3 在 xterm 中连接控制器
 - 5.4 openflow 1.3 交换机实例
6. Ryu 的 API 使用
7. 总结
8. 作业

1. 课程说明

本课程为动手实验教程，为了能说清楚实验中的一些操作会加入理论内容，也会精选最值得读的文章推荐给你，在动手实践的同时扎实理论基础。

2. 学习方法

学习方法是多实践，多提问。启动实验后按照实验步骤逐步操作，同时理解每一步的详细内容。

如果实验开始部分有推荐阅读的材料，请务必先阅读后再继续实验，理论知识是实践必要的基础。

3. 本节内容简介

本实验中我们初步接触 ryu 以及一些简单的 open vswitch 的控制命令。需要依次完成下面几项任务：

- Ryu 的 APP 使用
- Ryu 的 API 使用

4. 推荐阅读

本节实验推荐阅读下述内容：

- [ryu 官方手册](#)

- openflow 的流表项
- open vswitch 的常用命令

5. Ryu 的 APP 使用

在充分的了解底层的工作者，我们明白SDN 中交换机只是一个执行者，只是通过其数据库中的各种表格来进行操作，而操控这些底层工作者的是控制器，由控制器来下发流表等操作来真正的控制数据流的走向。

在之前我们说过我们将使用Ryu 控制器来作为我们学习的工具，首先我们来回顾一下我们使用Ryu 控制器的三种方式：

- 在外部启动，通过指定ip 地址、端口来连接
- 直接指定 ryu 控制器
- 通过 xterm 来制定控制器

5.1 在外部启动 ryu，通过指定 ip 地址来连接

首先我们启动 ryu-manager:

```
ryu-manager
```

```
shiyancelou:~/ $ ryu-manager
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
```



然后我们再打开一个终端，在该终端中使用mininet 远程连接：

```
sudo mn --controller=remote,ip=127.0.0.1
```

此时我们再打开一个终端，我们可以通过这样的命令来判断我们的mininet 是否连接上了 ryu 控制器：

```
sudo ovs-vsctl show
```

在上章节中的结构图中我们了解到 `ovs-vsctl` 命令是控制 ovssdb 的工具，通过该工具我们可以修改查看 ovssdb 中的一些配置信息，如使用 show 参数我们便可查看当前的网络：

```

shiyanolou:app/ (master*) $ sudo ovs-vsctl show
11f95a22-c377-4a79-8e3c-5bb3b4d7193e
    Bridge "s1"
        Controller "ntcp:6654"
        Controller "tcp:127.0.0.1:6653"
        is_connected: true
    fail_mode: secure
    Port "s1-eth2"
        Interface "s1-eth2"
    Port "s1"
        Interface "s1"
        type: internal
    Port "s1-eth1"
        Interface "s1-eth1"
    ovs_version: "2.0.2"
shiyanolou:app/ (master*) $

```



从显示的信息中我们可以看到 `is_connected` 值为 `true`，说明 mininet 中的 switch 已经成功的与我们的 ryu 控制器相连接，因为我们是首先启动的ryu，会占用 6653 端口，并且 controller 指定为 remote，所以这并不是默认所指定的pox 控制器。

这便是第一种方式启动、连接ryu 的方法

5.2 直接指定 ryu 控制器

在 2.2.2 版本中，mininet 修复了 ryu 控制器的支持，所以我们可以直接通过controller 来指定 ryu：

```
sudo mn --controller=ryu
```

```

shiyanolou:app/ (master*) $ sudo mn --controller=ryu
*** Creating network
*** Adding controller
warning: no Ryu modules specified; running simple_switch only
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>

```



由此我们便可直接使用ryu 控制器，十分的便捷，我们用dump 命令查看其中的详情，确定使用的是ryu 控制器：

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=5145>
<Host h2: h2-eth0:10.0.0.2 pid=5149>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=5154>
<Ryu c0: 127.0.0.1:6653 pid=5138>
mininet> █
```

这便是直接指定 ryu 的方式。

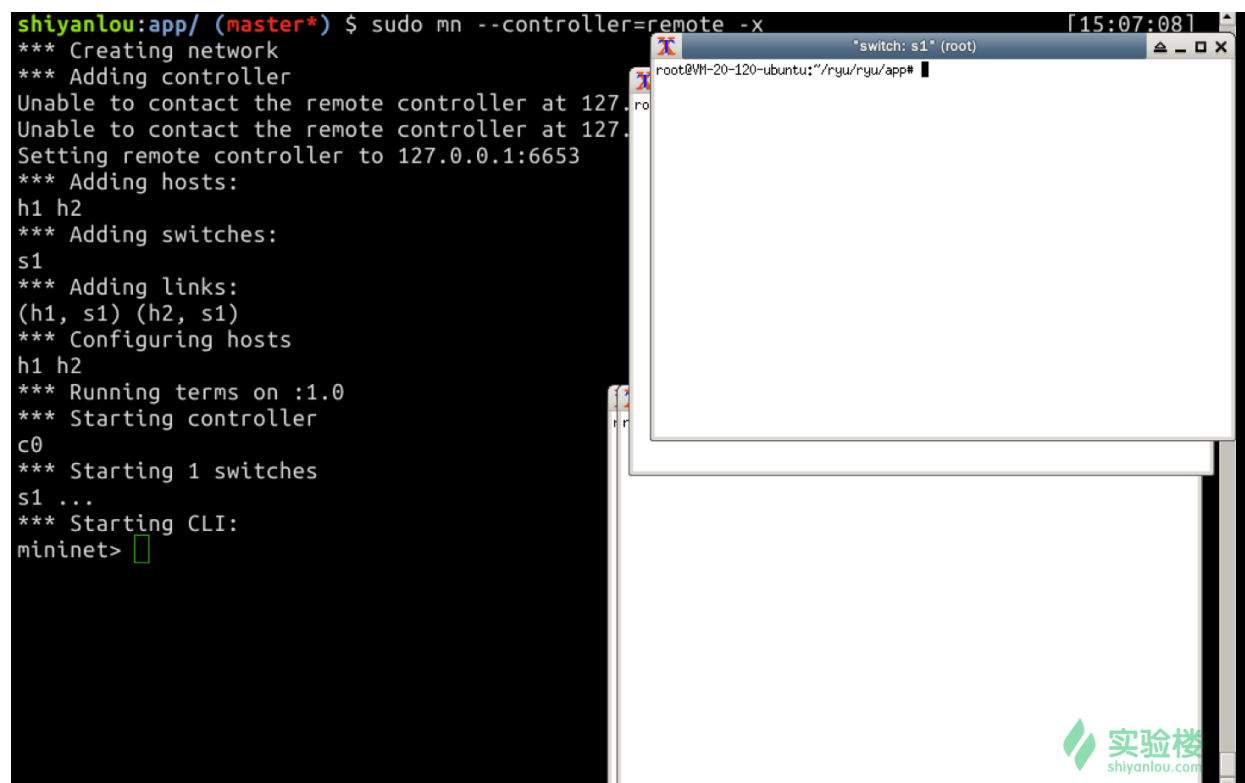
5.3 在 xterm 中连接控制器

还有一种方式是在 controller 的 xterm 中启动要使用的控制器。

首先启动 mininet:

```
sudo mn --controller=remote -x
```

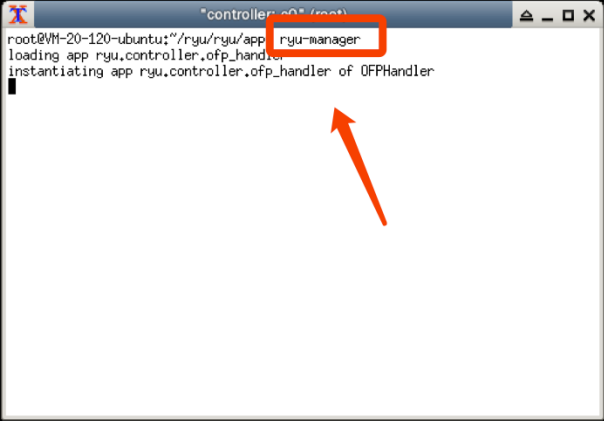
x 参数可以在启动的同时帮我们开启每个节点的xterm:



然后我们找到 c0 的 xterm 窗口，也就是 controller 所对应的终端窗口，在这里面我们启动 ryu:

```
ryu-manager
```

```
shiyanolou:app/ (master*) $ sudo mn --controller=remote -x [15:07:08]
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Running terms on :1.0
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> 
```



同样我们可以使用上述的方式来检测我们的controller 是否连接成功。

5.4 openflow 1.3 交换机实例

这是 ryu 的启动，我们使用 `cd /home/shiyanolou/ryu/ryu/app` 切换至 ryu 中，我们可以看见其中有很多的实例，如不同版本的支持，有API 的支持，有 stp 的功能支持，有路由、Qos、防火墙等功能的支持。

我们只需要在启动 ryu 的同时指定运行想要的功能的程序即可。例如simple_switch_13.py 便是一个实现对 openflow 1.3 支持的普通交换机，如何来运行它呢？

我们使用 xterm 的方式来实验一次如何使得我们的网络使用openflow1.3 的协议。

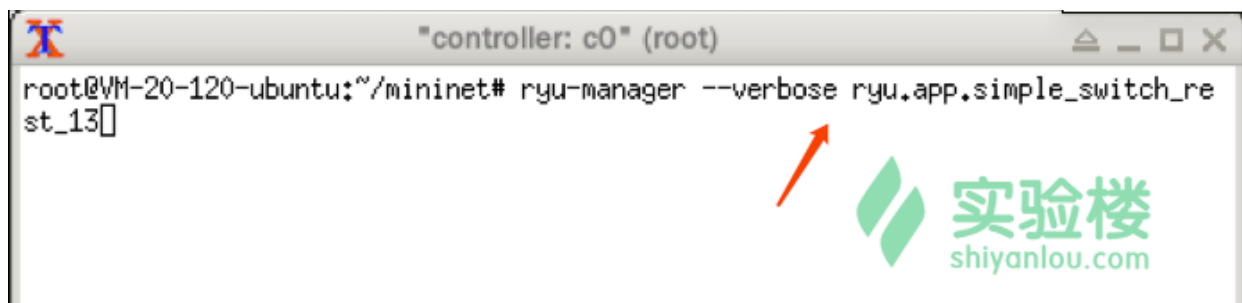
首先我们启动 mininet:

```
sudo mn --controller=remote -x
```

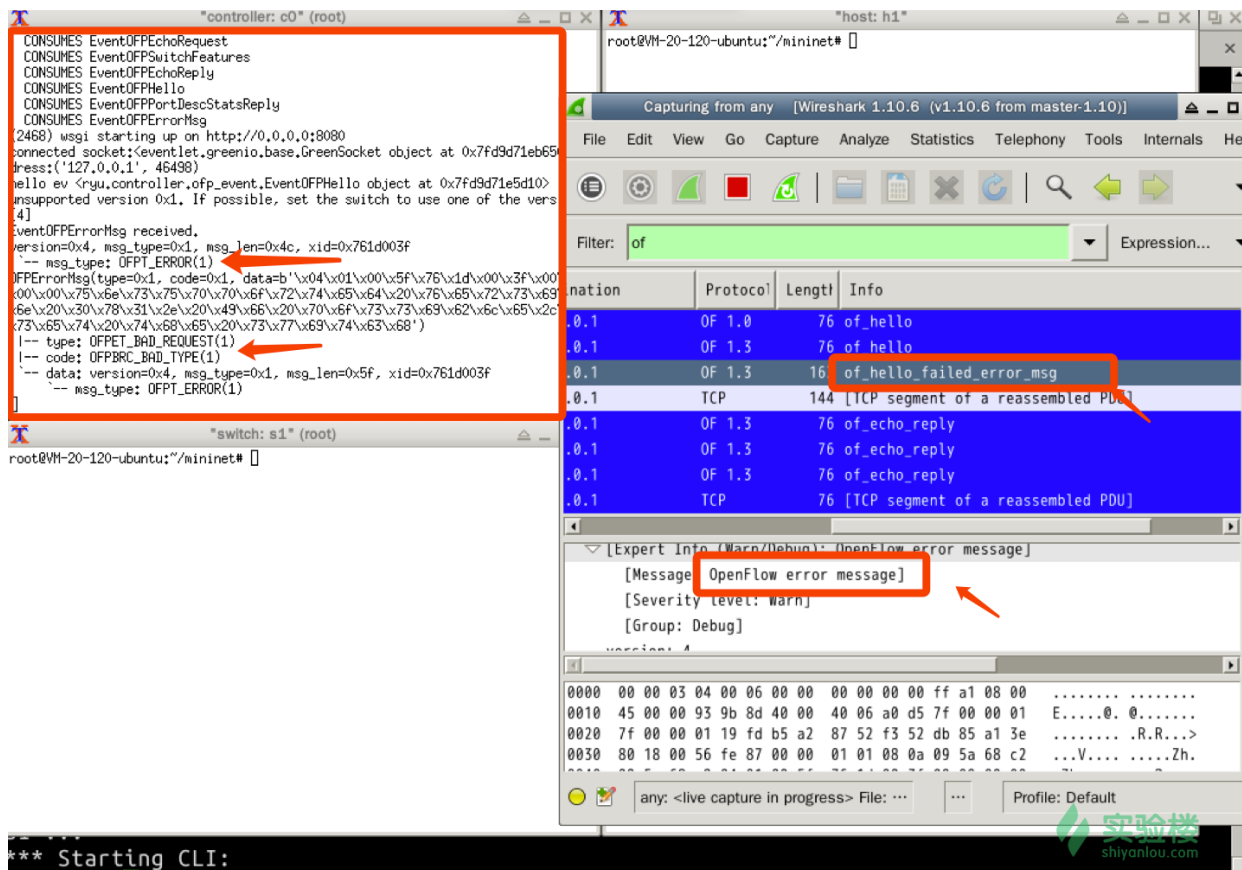
从上一章的实验中我们可以发现默认情况下使用的协议是openflow1.0，所以首先我们需要在 switch 的 xterm 中为其指定使用 openflow 协议使用的版本，否则在hello 包协商时会出错。

首先我们启动 simple_switch_13，让 controller 端使用 openflow1.3 版本，在此之前别忘记启动 wireshark 开启数据包的抓取：

```
ryu-manager --verbose ryu.app.simple_switch_13
```

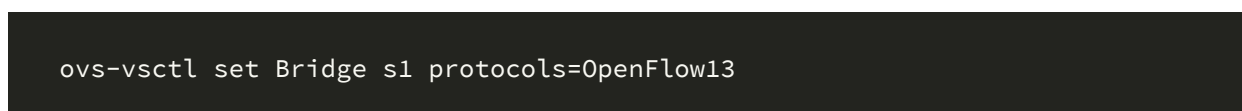


此时我们在终端，在 wireshark 都可以看到因协商失败而产生的报错与错误数据包的信息：

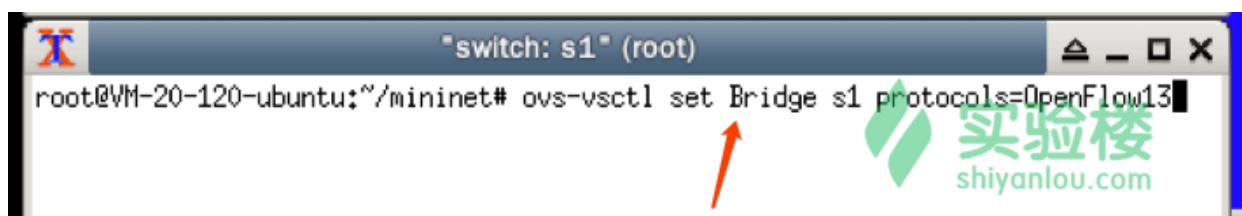


hello 数据包的阶段就是为了协商 controller 与 switch 之间相互通信的 openflow 协议的版本，毕竟低版本的 openflow 并不能支持高版本 openflow，所以此时协商失败便会导致连接不成功。

在这个时候我们需要使用 ovs-vsctl 来设置此时 switch 使用的 openflow 的版本：



我们在上一章说过其实 datapath 的作用就是转发数据包，本质就像是一个网桥，而mininet的实现就是使用 openvswitch 创建网桥，如这样的命令 `ovs-vsctl add-br s1`，所以此处我们是修改 switch 支持协议的版本所以使用的参数是 `set Bridge`，紧接着我们通过 `protocols` 参数来指定支持的协议版本。由此便可使得 s1 设备支持 openflow1.3 的版本。



既然 switch 可以通过命令的方式来创建，是否host 也能通过这样的方式来启动？

host 我们探究过是使用不同的namespace 来隔离实现虚拟化，所以若是使用命令来实现便是 `ip netns add h1`，而主机间的连接或者与 switch 之间的连接可以通过 `ip link add` 命令来连接，所以其实这是第四种自定义的方式，但是这样的方式有一定的门槛，需要对openvswitch 与 linux 一些不是特别常用的命令很熟悉，这也是为什么我们在第一章中选择了mininet 而没有选择 openvswitch 的方式来模拟网络，mininet 抽象出了一些简单易用的接口，隐藏了这些繁复的操作，减小了我们的学习成本，让我们更容易上手，同样让我们在模拟网络的时候也便捷了不少。

此时我们可以看到 c0 窗口中的提示立即发生了变化：

```
version=0x4, msg_type=0x1, msg_len=0x4c, xid=0x761d003f
-- msg_type: OFPT_ERROR(1)
OFPTErrorMsg(type=0x1, code=0x1, data=b'\x04\x01\x00\x5f\x76\x1d\x00\x3f\x00\x00\x00\x00\x75\x6e\x73\x75\x70\x70\x6f\x72\x74\x65\x64\x20\x76\x65\x72\x73\x69\x6f\x6e\x20\x30\x78\x31\x2e\x20\x49\x66\x20\x70\x6f\x73\x73\x69\x62\x6c\x65\x2c\x20\x73\x65\x74\x20\x74\x68\x65\x20\x73\x77\x69\x74\x63\x68')
|-- type: OFPET_BAD_REQUEST(1)
|-- code: OFPERR_BAD_TYPE(1)
-- data: version=0x4, msg_type=0x1, msg_len=0x5f, xid=0x761d003f
-- msg_type: OFPT_ERROR(1)
connected socket: <eventlet.greenio.base.GreenSocket object at 0x7fd9d71eb150> a
ddress: ('127.0.0.1', 46657)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x7fd9d71e5bd0>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version=0x4, msg_type=0x6, msg_len=0x20, xid=0xe2d81e89, OFPSwitc
hFeatures(auxiliary_id=0, capabilities=71, datapath_id=1, n_buffers=256, n_tables=25
4)
move onto main mode
```

我们可以看到 controller 立即从 error 的状态变成了连接成功，从 hello 转变成了 config 的模式与 main 模式



与此同时我们也可以看到 controller 结束了暗无天日的 of_echo_reply 的数据包发送，进入了正常的流程中：

33692	2243.806203000	127.0.0.1	127.0.0.1	TCP	76 [TCP segment of a reassembled PDU]
33693	2243.806925000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_reply
33770	2248.806937000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_reply
33796	2253.806754000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_reply
33824	2254.884936000	127.0.0.1	127.0.0.1	OF 1.3	84 of_hello
33828	2254.888078000	127.0.0.1	127.0.0.1	OF 1.3	76 of_hello
33830	2254.888121000	127.0.0.1	127.0.0.1	OF 1.3	76 of_features_request
33832	2254.888217000	127.0.0.1	127.0.0.1	OF 1.3	100 of_features_reply
33833	2254.889999000	127.0.0.1	127.0.0.1	OF 1.3	84 of_port_desc_stats_request
33834	2254.890095000	127.0.0.1	127.0.0.1	OF 1.3	148 of_flow_add
33835	2254.890104000	127.0.0.1	127.0.0.1	OF 1.3	276 of_port_desc_stats_reply
33860	2259.806254000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_request
33862	2259.806934000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_reply
33899	2264.806180000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_request
33900	2264.806950000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_reply
34105	2269.806155000	127.0.0.1	127.0.0.1	OF 1.3	76 of_echo_request

从数据包的 ID 可以判断这是修改后的 of_hello 数据包

正确的交互流程

```
> Frame 33824: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface 0
> Linux cooked capture
> Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
> Transmission Control Protocol, Src Port: 46657 (46657), Dst Port: openflow (6653), Seq: 1, Ack: 1, Len: 16
> OpenFlow (LOXI)
  version: 4
  type: OFPT_HELLO (0)
  length: 16
  xid: 119
```



我们可以通过这样的方式来查看此时是否能够正常的通信：

pingall


```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```



所有的数据包都通过，没有被丢的数据包，于此同时我们可以查看控制器中的信息：

```
[-- code: OFPBRC_BAD_TYPE(1)
-- data: version=0x4, msg_type=0x1, msg_len=0x5f, xid=0x761d003f
-- msg_type: OFPT_ERROR(1)
connected socket: <eventlet.greenio.base.GreenSocket object at 0x7fd9d71eb150> a
ddress: ('127.0.0.1', 46657)
hello ev <ryu.controller.ofp_event.EventOFHello object at 0x7fd9d71e5bd0>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version=0x4, msg_type=0x6, msg_len=0x20, xid=0xe2d81e89, OFPSwitc
hFeatures(auxiliary_id=0, capabilities=71, datapath_id=1, n_buffers=256, n_tables=25
4)
move onto main mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 e6:30:60:84:d0:f5 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 2a:88:4c:2a:9c:d3 e6:30:60:84:d0:f5 2
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 e6:30:60:84:d0:f5 2a:88:4c:2a:9c:d3 1
```



因为刚刚启动的控制器与 switch，所以此时的流表是空的，当 datapath 接受 ping 所再来的数据包是并不知道该如何处理，所以只有向上汇报请示 controller，所以 controller 会接受到 packet in 的处理，通过 `h1 ifconfig` 我们可以得知 `e6:30:60:84:d0:f5` 这个 Mac 地址是属于 h1 的，而 `ff:ff:ff:ff:ff:ff` 是一个广播泛洪的地址，这是因为 h1 向 h2 发送 ping 数据包，但是并不知道 h2 的 ip 地址与 Mac 地址，此时就会泛洪，而当 h1 回复该数据包时，switch 就会将这两个 Mac 地址记录下来，这样 h1 与 h2 之间就可以相互通信了。

当然我们在此使用 `pingall` 命令，便会发现控制器中没有再接收到 packet in 的事件，这便是 controller 在 packet out 的同时下发流表，更新了 switch 中的流表，后续 datapath 接受到数据包与流表项匹配，匹配成功便执行其对应的操作，所以这样便不会有 packet in 的事件了。

我们可以通过这样的命令来查看此时的流表项：

```
sudo ovs-ofctl dump-flows -O openflow13 s1
```

```
shiyanlou:app/ (master*) $ sudo ovs-ofctl dump-flows -O openflow13 s1 [16:38:23]
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=335.051s, table=0, n_packets=7, n_bytes=518, priority=1,in_port=2,dl_
dst=e6:30:60:84:d0:f5 actions=output:1
  cookie=0x0, duration=335.049s, table=0, n_packets=6, n_bytes=420, priority=1,in_port=1,dl_
dst=2a:88:4c:2a:9c:d3 actions=output:2
  cookie=0x0, duration=1633.775s, table=0, n_packets=3, n_bytes=182, priority=0 actions=CONT
ROLLER:65535
```

可以看到此时多了三个流表项，分别是：

- 发送给 h1 的处理方式
- 发送给 h2 的处理方式
- 以及没有匹配到则发送给控制器

此时添加了 `-O` 参数是我们使用了 openflow13 来通信，若是不添加该参数指明使用的协议版本默认使用的 openflow1.0，如此会有这样的报错：

```
shiyanlou:app/ (master*) $ sudo ovs-ofctl dump-flows s1 [16:55:23]
2017-04-07T09:01:06Z|00001|vconn|WARN|unix:/var/run/openvswitch/s1.mgmt: version negotiatio
n failed (we support version 0x01, peer supports version 0x04)
ovs-ofctl: s1: failed to connect to socket (Broken pipe)
```



这就是使用自定义应用的方式，我们只需要在启动的时候在参数中指定即可，而这样的应用可以让我们实现各种各样的功能，这就是SDN 的灵魂所在，数据层与控制层的分离，功能的自由添加，实现。

6. Ryu 的 API 使用

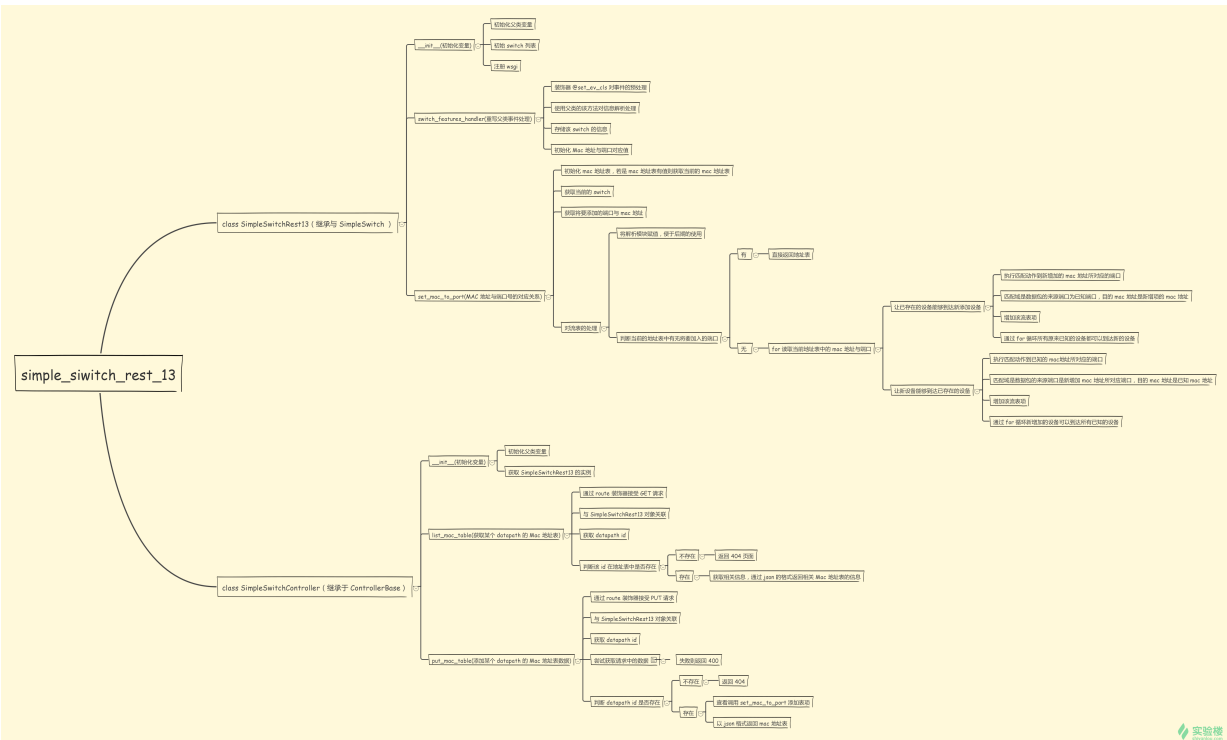
不仅如此，我们还可以使用REST API 来控制 controller 的一些行为，如下发表表、删除流表项等等，这样使得我们在一些临时的变动时更易修改，若是传统的机制我们还需到物理设备前，亦或者是远程登录至设备，一台台配置，在此凸显了SDN 的灵活。

在 ryu 中提供了 wsgi 的 web server 端，所以只需要我们实现相关的访问接口来相结合便可实现。

在 ryu 的源码中同样有前辈为我们贡献出了实现简单REST API 的相关事例以供我们参考。

```
less /home/shiyanlou/ryu/ryu/app/simple_switch_rest_13.py
```

查看代码，首先注意到的便是引入的外部文件，与之前我们所看到simple_switch_13最大的不同便是引入了 wsgi 的相关工具，随之我们看到这份简单的代码结构很简单：



代码的简洁是我们非常容易读懂，主要是理清楚其处理的流程。

在理清楚其如何实现之后我们便来尝试一下我们该如何使用相关的API。

1.启动模拟网络

第一步当然是启动网络拓扑结构，设置交换机使用的协议。

```
sudo mn --controller=remote -x --mac
```

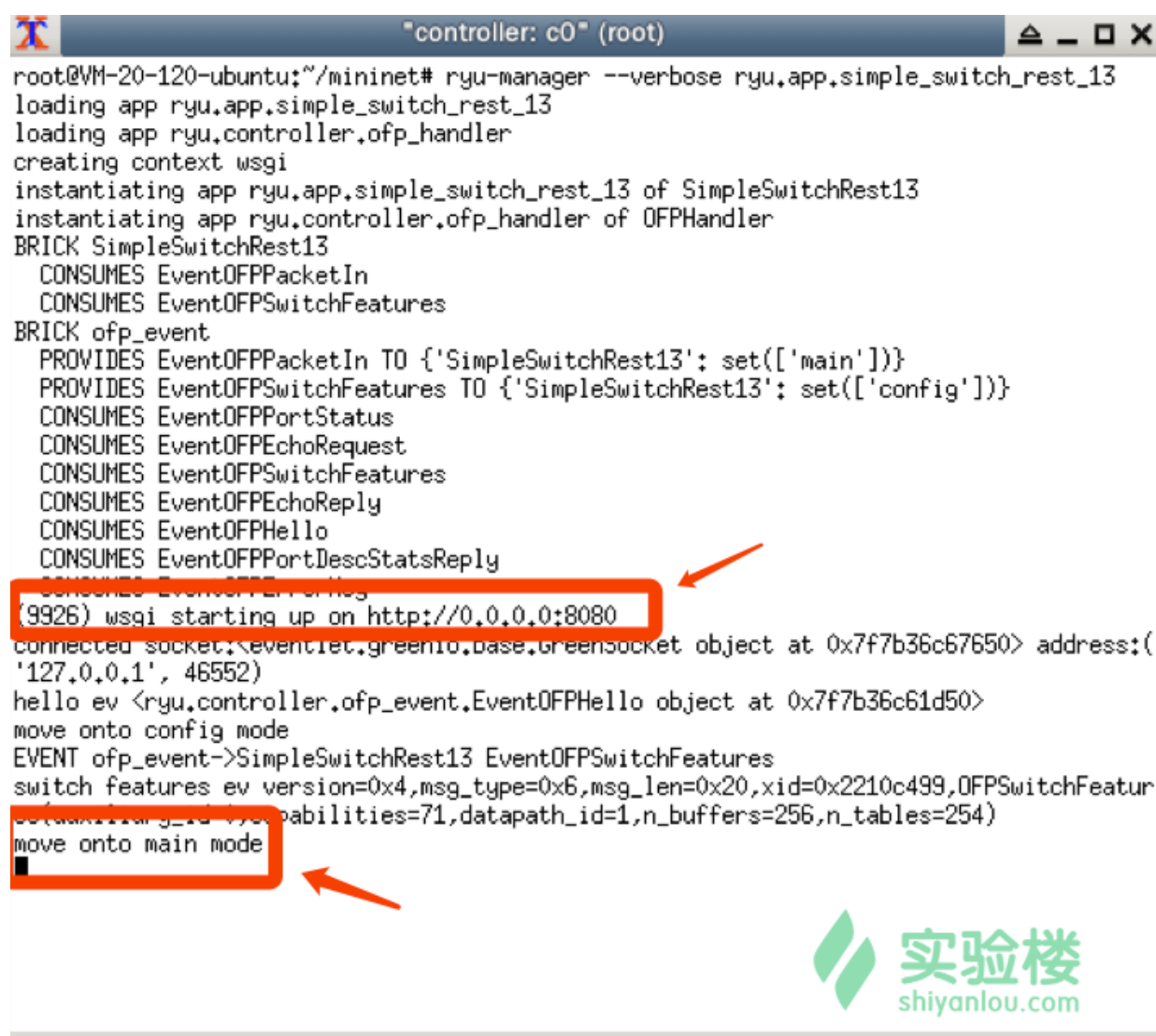
通过上述命令启动需要模拟的拓扑结构，其中-mac 参数是为了让 mac 地址的值人性化显示，而不像以前那般都是随机值，紧接着设置交换机使用的协议版本：

```
#在 s1 窗口中执行
ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

2.启动 ryu 控制器，同时运行我们的 simple_switch_rest_13

```
#在 c0 窗口中运行
ryu-manager --verbose ryu.app.simple_switch_rest_13
```

成功启动之后我们会看到这样的信息：



```
controller: c0 (root)
root@VM-20-120-ubuntu:~/mininet# ryu-manager --verbose ryu.app.simple_switch_rest_13
loading app ryu.app.simple_switch_rest_13
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app ryu.app.simple_switch_rest_13 of SimpleSwitchRest13
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK SimpleSwitchRest13
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {'SimpleSwitchRest13': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitchRest13': set(['config'])}
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPEchoReply
  CONSUMES EventOFPHello
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPErrMsg
(9926) wsgi starting up on http://0.0.0.0:8080
connected socket (<eventlet.greenio.base.GreenSocket object at 0x7f7b36c67650> address:('127.0.0.1', 46552))
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x7f7b36c61d50>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version=0x4,msg_type=0x6,msg_len=0x20,xid=0x2210c499,OFPSwitchFeatures{capabilities=71,datapath_id=1,n_buffers=256,n_tables=254}
move onto main mode
```

从 `move onto main mode` 的结束语我们便可得知我们启动中应该是没有遇到什么问题，我们往上查看信息，我们会看到有 `creating context wsgi` 以及 `(9926) wsgi starting up on http://0.0.0.0:8080` 这样的提示信息。

说明我们成功的启动了 wsgi，并且 wsgi 的相关端口运行在 8080，也就是说若是在这之前有应用占用了 8080 端口，此处启动的时候会报错，若是报错了请用 `sudo lsof -i:8080` 查看相关端口运行的信息，停掉相关应用在尝试启动 (tomcat 就是典型会占用 8080 端口的应用)。

3.使用相关 API

还记得源码中在引入文件之后定义了两个全局变量：

- simple_switch_instance_name
- url

第一个变量用于 wsgi 注册时与 controller 对 SimpleSwitchRest13 实例的绑定，而第二变量便是我们将要使用接口的地址，仔细观察每个 route 装置器便可得知。

所以若是此时我们想获取 s1 交换机的 Mac 地址表，我们只需要在新的终端中运行这样的命令：

```
curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/(交换机对应的datapath id)
```

若是不清楚我们即将访问的 switch 的 datapath id 值，我们可以使用这样的方式获取：

```
ovs-ofctl -O openflow13 show s1
```

```
root@VM-20-120-ubuntu:~/mininet# ovs-ofctl -O openflow13 show s1
OFPT_FEATURES_REPLY (OF1.3) (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS
OFPT_GET_CONFIG_REPLY (OF1.3) (xid=0x4): frags=normal miss_send_len=0
root@VM-20-120-ubuntu:~/mininet#
```



再在我们的终端中执行这样的语句：

```
curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001
```

其实我们得到了这样的结果：

```
shiyancelou:app/ (master*) $ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001
{}%
```

这是因为刚刚启动的 switch，其中的 Mac 地址当然是空的，我们在 mininet 中执行：

```
pingall
```

然后我们在尝试同样的命令：

```
shiyancelou:app/ (master*) $ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001
{}%
shiyancelou:app/ (master*) $ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}%
shiyancelou:app/ (master*) $
```



我们可以看到有两条记录，分别是h1 对应一号端口，h2 对应二号端口：

```
root@VMH-20-120-ubuntu:~/mininet#
h1-eth0 Link encap:以太网 硬件地址 00:00:00:00:00:01
inet 地址:10.0.0.1 掩:255.0.0.0 网:10.0.0.0
inet6 地址: fe80::200:ff:fe00:1/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 点数:1
接收数据:4 :0 弃:0 :0 数:0
发送数据:13 :0 弃:0 :0 波:0
接收:0 发送:1000
接收字 :280 (280.0 B) 送字 :1018 (1.0 KB)

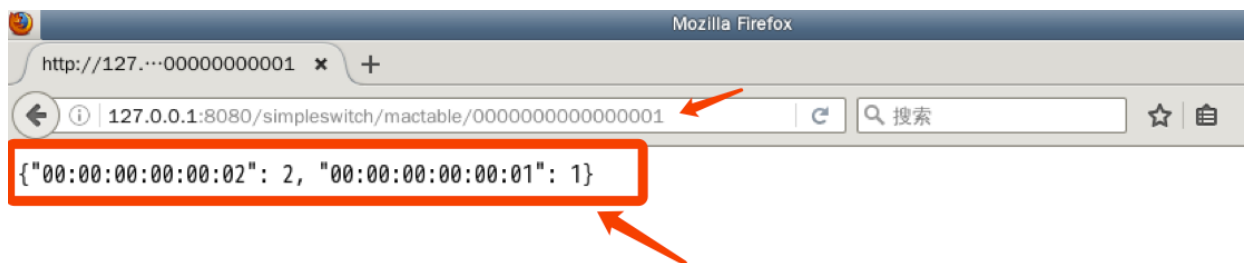
lo Link encap:本地 回
inet 地址:127.0.0.1 掩:255.0.0.0
inet6 地址: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 点数:1
接收数据:0 :0 弃:0 :0 数:0
发送数据:0 :0 弃:0 :0 波:0
接收:0 发送:0
接收字 :0 (0.0 B) 送字 :0 (0.0 B)

root@VMH-20-120-ubuntu:~/mininet#

mininet>
mininet>
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo: s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet>

shiyanyou:app/ (master*) $ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/00000000
00000001
{}%
shiyanyou:app/ (master*) $ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable/00000000
00000001
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}%
shiyanyou:app/ (master*) $
```

因为这是使用的 GET 方法，若是不喜欢使用命令行，以curl 来获取信息，你也可以在浏览器中访问该地址：



而实现这样功能的就是 `list_mac_table()`。

我们还有另外一个方法 `put_mac_table()` 可以用来插入新的记录，以此更新Mac 地址表，我们只需执行这样的命令：

```
curl -v -X PUT -d '{"mac" : "00:00:00:00:00:01", "port" : 1}' http://127.0.0.1:8080/simpleswitch/mactable/000000000000000001
```

我们会从返回的状态码中发现我们添加成功，因为返回的是200:

```
Terminal 终端 - shiyanlou@VM-20-120-ubuntu: ~/ryu/ryu/app
simple_switch_rest_13.py (~/.ryu/ryu/app) - VIM x shiyanlou@VM-20-120-ubuntu: ~/ryu/ryu/app x shiyanlou@VM-20-120-ubuntu: ~/ryu/ryu/app x
shiyanlou:app/ (master*) $ curl -v -X PUT -d '{"mac" : "00:00:00:00:00:01", "port" : 1}' http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> PUT /simpleswitch/mactable/0000000000000001 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1:8080
> Accept: */*
> Content-Length: 41
> Content-Type: application/x-www-form-urlencoded
>
upload completely sent off: 41 out of 41 bytes
HTTP/1.1 200 OK
< Content-type: application/json
< Content-Length: 48
< Date: Tue, 11 Apr 2017 14:43:13 GMT
* Connection #0 to host 127.0.0.1 left intact
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
shiyanlou:app/ (master*) $
```

```
*controller: c0* (root)
-- data: version=0x4, msg_type=0x1, msg_len=0x5f, xid=0xec5e440c
-- msg_type: OFPT_ERROR(1)
connected socket: <eventlet.greenio.base.GreenSocket object at 0x7fde84b79250> address: ('127.0.0.1', 34085)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x7fde84b73f50>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version=0x4, msg_type=0x6, msg_len=0x20, xid=0x684b8903, OFPSwitchFeatures(auxiliary_id=0, capabilities=71, datapath_id=1, n_buffers=256, n_tables=254)
move onto main mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
(21506) accepted ('127.0.0.1', 39433)
127.0.0.1 - - [11/Apr/2017 22:43:02] "PUT /simpleswitch/mactable/0000000000000000
1 HTTP/1.1" 200 156 0,001827
(21506) accepted ('127.0.0.1', 39433)
127.0.0.1 - - [11/Apr/2017 22:43:13] "PUT /simpleswitch/mactable/0000000000000000
1 HTTP/1.1" 200 156 0,000899
```

但是有的同学会认为，Mac 地址表中本来就拥有该表项，这并不能说明什么，我们可以关掉控制器，再启动，重新启动的控制数据便是空的，此时我们在put 一次，然后在 get 一次，就能证明确实是添加成功的了。

这便是 Ryu 运行 app 的方法，以及对 app 的 API 使用的方法，其实往简单的来说就是对datapath 数据的一个接受与处理，若 API 接收到请求的处理，最重要的就是我们希望操作的流程我们是否清楚，若是清楚实现起来变简单不少。

这样的代码只是一个实例代码，并不是非常的完美，例如当我们刚刚启动应用时便put 表项，但是 put 时我们一不小心将 mac 地址写错了，多加了一对0，成了 `curl -v -X PUT -d '{"mac" : "00:00:00:00:00:00:00:01", "port" : 1}' http://127.0.0.1:8080/simpleswitch/mactable/0000000000000001` 这样，我们依然会插入成功，我们 get 也能够看到相应的数据，但是当我们在插入数据就会失败，会返回一个500 的错误，这是因为尝试更新 mac 地址表的时候出错了。

当我们想尝试修改这个 app 的代码时，亦或者书写我们自己的 app 代码，直接修改此处的代码在运行时没有用的，因为真正运行读取的是 `/usr/local/lib/python2.7/dist-packages/ryu/ryu/app` 目录中编译好的 python 文件，所以我们需要这样来运行我们修改好的代码：

```
ryu-manager /home/shiyanlou/ryu/ryu/app/simple_switch_rest_13.py
```

若是自己编写的 app，便将此处的路径换成代码所在的路径，将执行文件名换成自己写的python 文件名。

7. 总结

本节实验中我们学习了以下内容：

- Ryu 的 APP 使用
- Ryu 的 API 使用

看着本章节只有两大板块APP的使用与API的使用，但其实本章节的涵盖知识面很多，如一些 open vswitch 的一些命令，python 代码的理解，通过这些实例代码的参照，我们可以写出属于我们自己的 app。

请务必保证自己能够动手完成整个实验，只看文字很简单，真正操作的时候会遇到各种各样的问题，解决问题的过程才是收获的过程。

8. 作业

1.通过 APP 使用方式启动 Ryu 所带的 GUI。（提示位于 app 中的 gui_topology 中）

成功的效果图应该是这样的：



2.有兴趣的朋友可以探究在 put 一次错误数据之后，在此 put 相同数据没有影响，但是 put 其他无论正确与否的信息都会报错。（不适合初学者，提示关键点在于 OFPMATCH() 上，调试过程较为麻烦，但是通过这样的过程可以进一步理解 openflow 数据解析的过程）

给大家分享一下，这是我 debug 出来的结果：


```
(Pdb) l
21     def __init__(self, addr, strat, **kwargs):
22         self._addr = addr
23         self._strat = strat
24         self._addr_kwargs = kwargs
25
26 ->     def text_to_bin(self, text):
27         return self._addr(text, **self._addr_kwargs).packed
28
29         def bin_to_text(self, bin):
30             return str(self._addr(self._strat.packed_to_int(bin),
31                                 **self._addr_kwargs))
(Pdb) n
> /usr/local/lib/python2.7/dist-packages/ryu/lib/addrconv.py(27)text_to_bin()
-> return self._addr(text, **self._addr_kwargs).packed
(Pdb) n
AddrFormatError: AddrForm...EUIv48",)
> /usr/local/lib/python2.7/dist-packages/ryu/lib/addrconv.py(27)text_to_bin()
-> return self._addr(text, **self._addr_kwargs).packed
(Pdb) █
```



