

mininet 基础

mininet 基础

1. 课程说明
2. 学习方法
3. 本节内容简介
4. 推荐阅读
5. mininet 原理
 - 5.1 mininet 是什么
 - 5.2 mininet实现
6. mininet指令
 - 6.1 mininet 启动
 - 6.1.1 指令
 - 6.1.2 原理
 - 6.2 dump 命令
 - 6.2.1 指令
 - 6.2.2 原理
 - 6.3 nodes 命令
 - 6.3.1 指令
 - 6.3.2 原理
 - 6.4 net 命令
 - 6.5 节点执行命令
 - 6.6 xterm 命令
 - 6.7 pingall 命令
 - 6.8 link 命令
 - 6.9 ipref 命令
 - 6.10 dpctl 命令
 - 6.11 执行外部命令
 - 6.12 help 命令
7. 总结
8. 作业
9. 参考文章

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

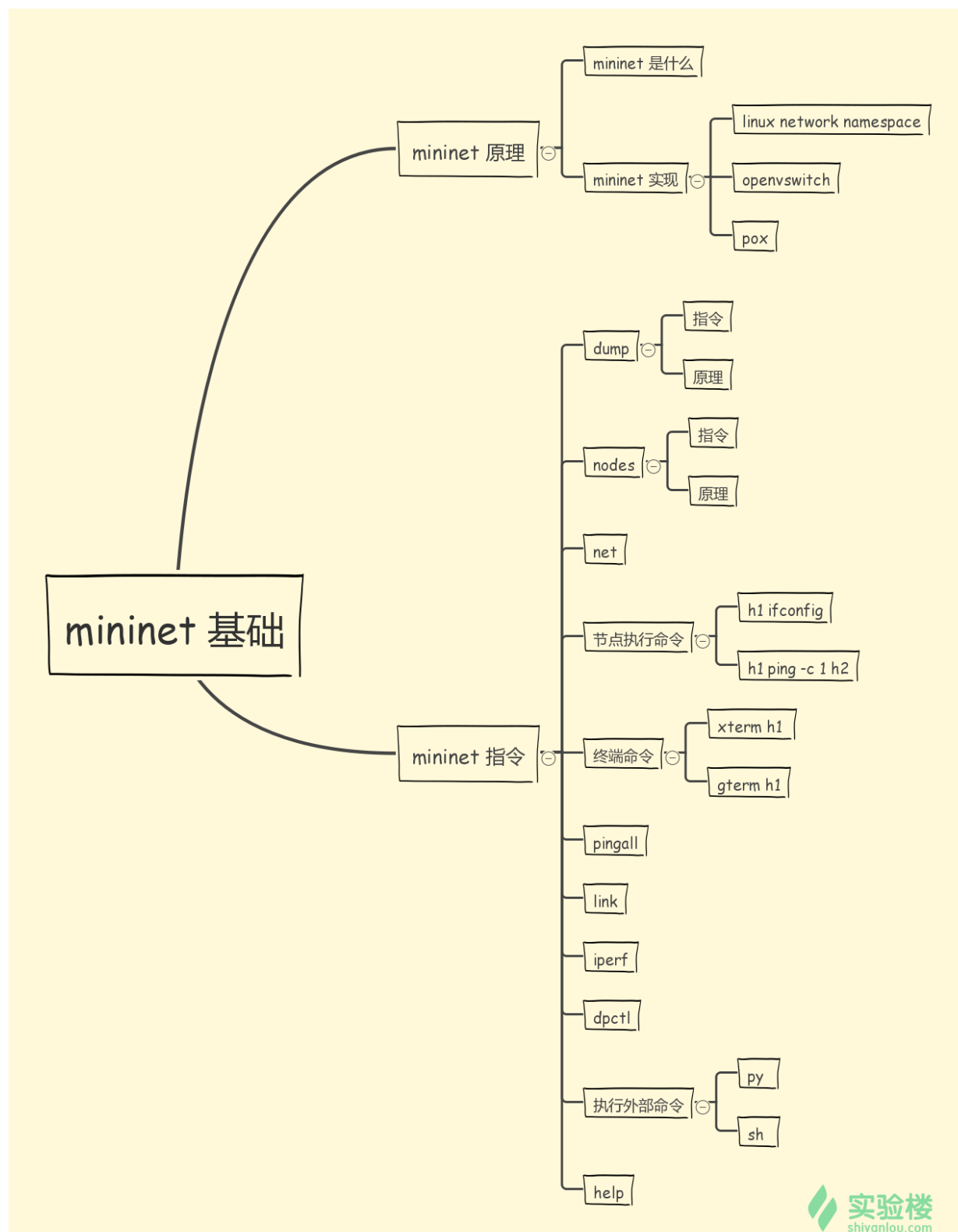
学习方法是多实践，多提问。启动实验后按照实验步骤逐步操作，同时理解每一步的详细内容。

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 mininet 的相关操作。需要依次完成下面几项任务：

- mininet 原理
- mininet 指令



4. 推荐阅读

本节实验推荐阅读下述内容：

- [openflow 初学](#)
- [mininet 使用](#)
- [mininet 源码分析](#)

5. mininet 原理

在上一个实验中我们大概明白了SDN 作用，对其有了一个大概的了解，并且我们选择了mininet 与 RYU 作为我们接下来学习的工具。

在使用 mininet 之前我们需要对其做一个更深入的了解，以便我们后续的学习。我们将围绕这样的两个话题来学习 mininet：

- mininet 是什么？
- mininet 是如何实现的？（只分析大概的框架结构）

5.1 mininet 是什么

mininet 是一个由 Nick McKeown 研究小组使用 python 开发一个轻量级的网络仿真工具（类似于 GNS3），所谓的仿真就是创建一个几乎与真实环境相同的网络，从拓扑结构到数据通信。

mininet 的便利之处是：

- 其代码迁移至硬件环境中几乎无差错，便于学习、测试
- 非常便捷、快速的创建大规模的网络环境

5.2 mininet实现

从上文我们了解到 SDN 主要包含这样的一些层次，且每个层次对应的组件：

层次	组件
业务层	SDN Application
控制层	controller
数据层	switch

既然 mininet 能够做到仿真模拟必然能够尽数包含在内：

- 在数据层方面有多种选择，即可使用openvSwitches 也可以使用 openflow switches 并且还可以使用自己的 switch；
- 在控制层方面 mininet 提供、使用 ovsc、pox、nox 等控制器（在 install.sh 脚本中可以看到还有其他控制器的安装），当然也可以使用远程的controller，也就是其他的 controller；

- 在业务层方面我们可以通过mininet 的命令行发送一些指令，还可以运行一些python 或者 shell 的脚本。

这些组件都存在于同一个架构中，又是如何让它们共存于同一台机器上呢？

mininet 是基于 Linux Container 开发的，而 Linux network namespace 机制可以说是实现这一切的功臣。

Linux Container (简称 LXC) 是一种操作系统层虚拟化 ((Operating system-level virtualization)) 技术，为Linux内核容器功能的一个用户空间接口。它将应用软件系统打包成一个软件容器 (Container) ，内含应用软件本身的代码，以及所需要的操作系统核心和库。通过统一的名字空间和共用API 来分配不同软件容器的可用硬件资源，创造出应用程序的独立沙箱运行环境，使得Linux 用户可以容易的创建和管理系统或应用容器。

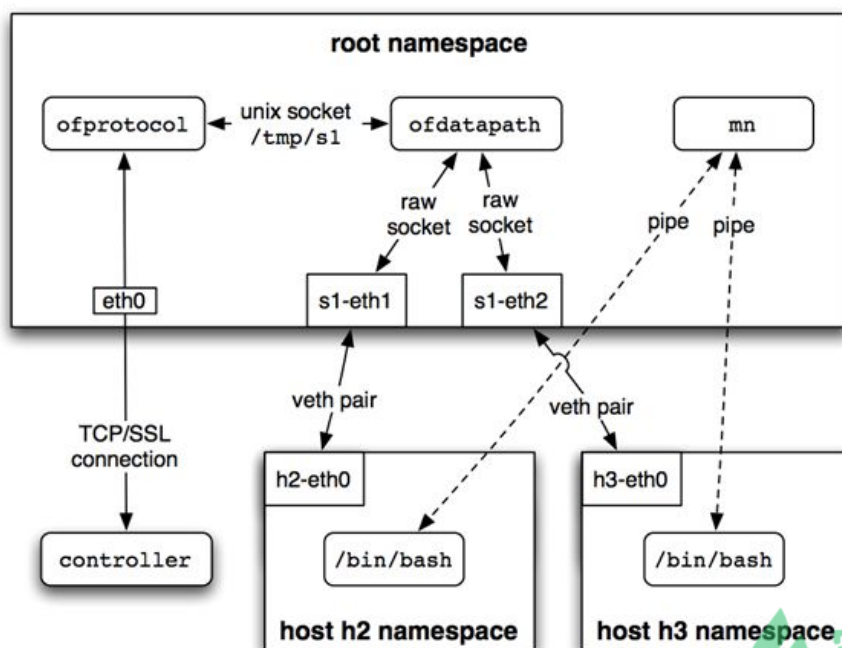
在 Linux 内核中，提供了 cgroups 功能，来达成资源的区隔化。它同时也提供了名称空间区隔化的功能，使应用程序看到的操作系统环境被区隔成独立区间，包括进程树，网络，用户 id，以及挂载的文件系统。但是 cgroups 并不一定需要引导任何虚拟机。

LXC 利用 cgroups 与名称空间的功能，提供应用软件一个独立的操作系统环境。LXC 不需要 Hypervisor 这个软件层，软件容器 (Container) 本身极为轻量化，提升了创建虚拟机的速度。(此段来自于 wikipedia)

简单来说：LXC 通过 cgroups 子系统利用 namespace 来实现系统资源使用上的隔离控制，内核支持这样 6 个命名空间：ipc (进程间通信资源命名空间)，uts (系统变量命名空间)，mount (文件系统挂载命名空间)，pid (进程 ID 号命名空间)，network (网络命名空间) and user (用户和组的命名空间)。

通过 namespace 的隔离，使得创建出来的每个 controller，switch 都在一个单独的 namespace 中，每个 namespace 中都可以模拟出网卡，并通过创建 veth pair 来连接不同的 namespace，使不同的 namespace 之间相互通信。就像每个都在一个单独的虚拟机中一般，从而便可实现大规模网络的模拟。

所有的组件在系统中类似于这样存在：



(此图来自于 [slideplayer](#))

6. mininet指令

6.1 mininet 启动

6.1.1 指令

在明白了 mininet 是什么，主体的实现方式之后我们便来尝试一下mininet 的使用。

通过这样的命令启动 mininet:


```
sudo mn
```

我们可以看到命令行的开头变成了 **mininet>** 说明我们启动了 mininet 并进入了其命令行，同时我们看到上面显示了一大串信息：

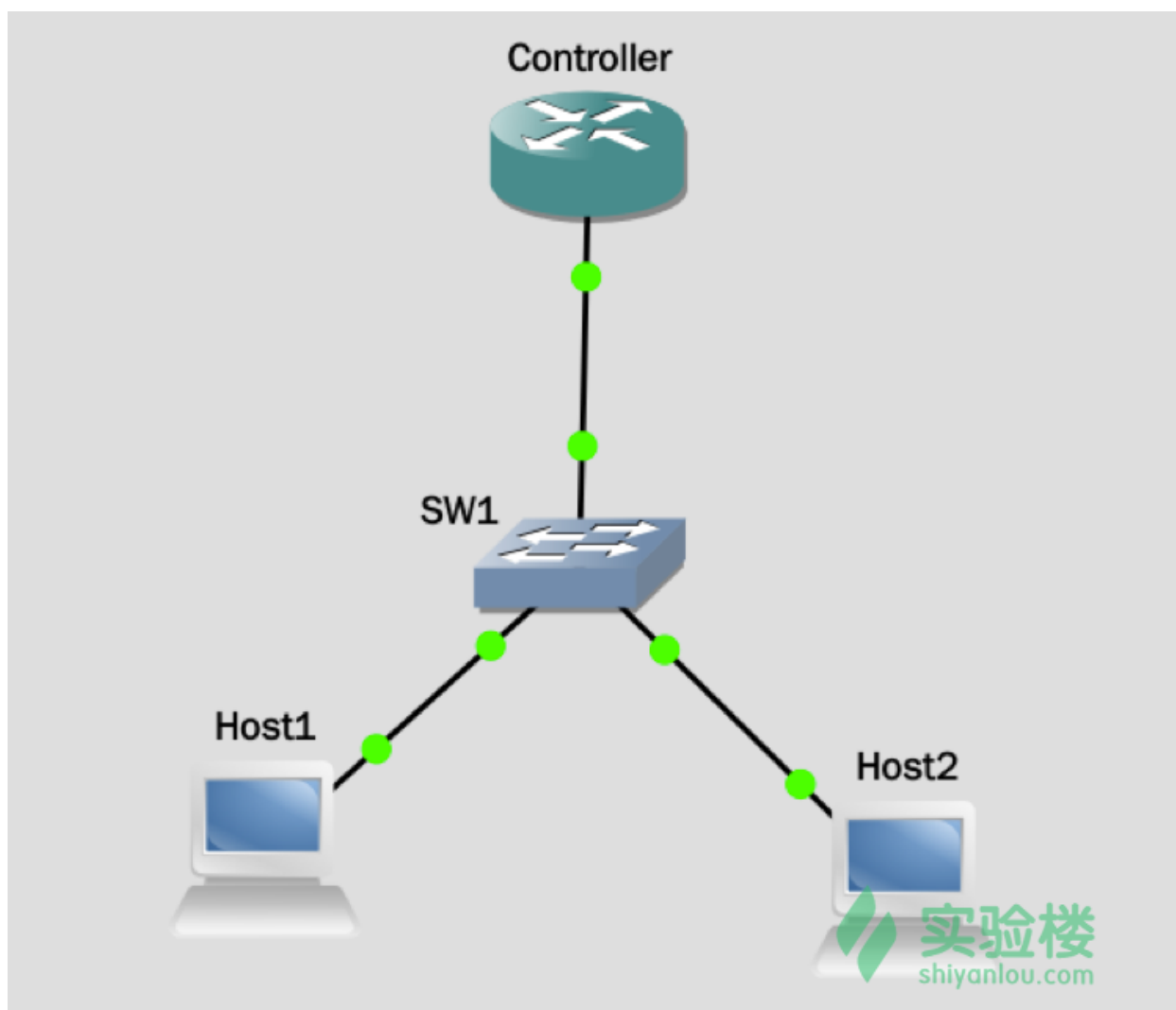
```
shiyanolou:app/ $ sudo mn
*** Error setting resource limits. Mininet's performance may be affected.
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>
mininet> 
```

提示信息

mininet 的命令行界面



这是因为当我们不添加任何参数的时候，默认会创建一个这样的拓扑结构：



通过显示的信息我们也可以看出其添加了1 个 controller，2 个 hosts，一个 switch。

注意：若是不想了解源码实现的同学可以跳过这一段

6.1.2 原理

我们可以通过查看源代码明白其中的缘由，进入mininet 源码的目录：

```
cd ~/mininet
```

查看 mn 的源文件：

```
less bin/mn
```

```
41 from functools import partial
42
43 # Experimental! cluster edition prototype
44 from mininet.examples.cluster import ( MininetCluster, RemoteHost,
45                                       RemoteOVSSwitch, RemoteLink,
46                                       SwitchBinPlacer, RandomPlacer,
47                                       ClusterCleanup )
48 from mininet.examples.clustercli import ClusterCLI
49
50 默认值 PLACEMENT = { 'block': SwitchBinPlacer, 'random': RandomPlacer }
51
52 # built in topologies, created only when run
53 TOPODEF = 'minimal'
54 TOPOS = { 'minimal': MinimalTopo,
55          'linear': LinearTopo,
56          'reversed': SingleSwitchReversedTopo,
57          'single': SingleSwitchTopo,
58          'tree': TreeTopo,
59          'torus': TorusTopo }
60
61 SWITCHDEF = 'default'
62 SWITCHES = { 'user': UserSwitch,
63             'ovs': OVSSwitch,
64             'ovsbr': OVSBridge,
65             # Keep ovsk for compatibility with 2.0
66             'ovsk': OVSSwitch,
67             'ivs': IVSSwitch,
```



查看 52 行左右，或者通过搜索 `TOPOS` 我们可以看见，当我们不指定 topo 参数的值时，默认为 `minimal`，而 `minimal` 对应的值为 `MinimalTopo`，我们退出该文件，查看 `MinimalTopo` 定义处：

```
less mininet/topo.py
```

```
320
321 class MinimalTopo( SingleSwitchTopo ):
322     Minimal topology with two hosts and one switch"
323     def build( self ):
324         return SingleSwitchTopo.build( self, k=2 )
325
```

查看 321 行左右，或者通过搜索 `MinimalTopo` 我们可以看见，在使用 `MinimalTopo` 时会直接调用 `SingleSwitchTopo` 的 `build` 方法，在本文中我们在搜索 `SingleSwitchTopo`，我们会在 293 行左右看到：

```
293 class SingleSwitchTopo( Topo ):
294     "Single switch connected to k hosts."
295
296     def build( self, k=2, **_opts ):
297         "k: number of hosts"
298         self.k = k
299         switch = self.addSwitch( 's1' )
300         for h in irange( 1, k ):
301             host = self.addHost( 'h%s' % h )
302             self.addLink( host, switch )
303
```





通过 `addSwitch()` 添加了一个名为 `s1` 的 switch，通过一个循环，循环中使用 `addHost()` 添加 host，循环的 `k` 值为 2，所以只会产生两个 host，并通过 `addLink()` 将 switch 与 host 连接起来。这便是为何默认创建一个 switch 与两个 host 的原因

同样我们在之前查看 `TOPOS` 的下方，我们可以看到 `CONTROLLERS` 的默认值为 `DefaultController`。

我们在终端中 `grep -R DefaultController` 我们可以看到：

```
shiyanolou:mininet/ (master) $ grep -R DefaultController [16:05:52]
bin/mn:             DefaultController, NullController,
bin/mn:             'default': DefaultController, # Note: overridden below
mininet/net.py:from mininet.node import ( Node, Host, OVSKernelSwitch, DefaultController,
mininet/net.py:             controller=DefaultController, link=Link, intf=Intf,
mininet/node.py:DefaultControllers = ( Controller, OVSController )
mininet/node.py:der findController( controllers=DefaultControllers ):
mininet/node.py:def DefaultController( name, controllers=DefaultControllers, **kwargs ):
shiyanolou:mininet/ (master) $ [16:06:21]
shiyanolou:mininet/ (master) $ [16:06:22]
```



默认使用的是 `OVSController` 控制器。

就这样我们便创建了这样的一个简单的 topo 结构。


6.2 dump 命令

6.2.1 指令

在 mininet 的命令行中，提供了这样的一些命令来辅助我们使用。

我们可以通过 `dump` 来查看所有节点的相关信息：

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=12246>
<Host h2: h2-eth0:10.0.0.2 pid=12247>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=12252>
<OVSController c0: 127.0.0.1:6633 pid=12238>
mininet>
```



通过 `dump` 我们可以看到节点的类别、名字、IP 信息、pid。通过 `dump` 我们也可以看到我们使用的默认 controller 是 `OVSController`。


6.2.2 原理

之前我们说过其实每个节点都在一个单独的namespace 里来隔离资源，从而做到虚拟化，实现模拟大规模网络的目的，有一定LXC 基础的同学会尝试使用 `ip netns list`，查看一下当前 network namespace 的情况，但是结果回令人失望，没有任何信息的返回。

这是因为 `ip netns` 只会显示在 `/var/run/netns` 中有名字的、有挂载的 namespace，而 mininet 中新的 namespace 是通过 PID 来与主机相关联，使用 `unshare()` 方式来创建出新的 namespace，所以是没有名字的 namespace，以至于我们通过 `ip netns list` 是无法查看到节点的 namespace。

在 mininet 的 net.py 中 Mininet 类中我们发现 `addHost()` 方法的定义：

```
206     def addHost( self, name, cls=None, **params ):
207         """Add host.
208             name: name of host to add
209             cls: custom host class/constructor (optional)
210             params: parameters for host
211             returns: added host"""
212         # Default IP and MAC addresses
213         defaults = { 'ip': ipAdd( self.nextIP,
214                                 ipBaseNum=self.ipBaseNum,
215                                 prefixLen=self.prefixLen ) +
216                     '/%s' % self.prefixLen }
217         if self.autoSetMacs:
218             defaults[ 'mac' ] = macColonHex( self.nextIP )
219         if self.autoPinCpus:
220             defaults[ 'cores' ] = self.nextCore
221             self.nextCore = ( self.nextCore + 1 ) % self.numCores
222         self.nextIP += 1
223         defaults.update( params )
224         if not cls:
225             cls = self.host
226         h = cls( name, **defaults )
227         self.hosts.append( h )
228         self.nameToNode[ name ] = h
229         return h
230
```



我们可以看到当 `cls` 为空时，也就是当没有制定的 host 时便使用 mininet 的 host，而 `self.host` 中的 host 是由 Host 类创建的对象。

```

class Mininet( object ):
    "Network emulation with hosts spawned in network namespaces."

    def __init__( self, topo=None, switch=OVSKernelSwitch, host=Host,
                  controller=DefaultController, link=Link, intf=Intf,
                  build=True, xterms=False, cleanup=False, ipBase='10.0.0.0/8',
                  inNamespace=False,
                  autoSetMacs=False, autoStaticArp=False, autoPinCpus=False,
                  listenPort=None, waitConnected=False ):
        """Create Mininet object.
        topo: Topo (topology) object or None
        switch: default Switch class
        host: default Host class/constructor
        controller: default Controller class/constructor
        link: default Link class/constructor
        intf: default Intf class/constructor
        ipBase: base IP address for hosts,
        build: build now from topo?
        xterms: if build now, spawn xterms?
        cleanup: if build now, cleanup before creating?
        inNamespace: spawn switches and controller in net namespaces?
        autoSetMacs: set MAC addrs automatically like IP addresses?
        autoStaticArp: set all-pairs static MAC addrs?
        autoPinCpus: pin hosts to (real) cores (requires CPULimitedHost)?
        listenPort: base listening port to open; will be incremented for
                    each additional switch in the net if inNamespace=False"""
        self.topo = topo
        self.switch = switch

```

通过 `grep` 我们查看到在 `node.py` 中 `Host` 类继承于 `Node`:

```

646 class Host( Node ):
647     "A host is simply a Node"
648     pass
649

```

在 `Node` 类中, `inNamespace` 默认值为 `true`, 在经过属性的初始化之后会调用 `startShell()` 方法, 下方的 `startShell()` 中我们看到, 当 `inNamespace` 为 `true` 会在稍后执行命令参数变量中加 `n`, 而执行的命令便是调用 `mnexec`, 而 `mnexec` 是位于根目录中一个用 C 写的程序, 查看该程序, 我们会发现这样的语句:

```

int main(int argc, char *argv[])
{
    int c;
    int fd;
    char path[PATH_MAX];
    int nsid;
    int pid;
    char *cwd = get_current_dir_name();

    static struct sched_param sp;
    while ((c = getopt(argc, argv, "+cdnpa:g:r:vh")) != -1)
        switch(c) {
            case 'c':
                /* close file descriptors except stdin/out/error */
                for (fd = getdtablesize(); fd > 2; fd--)
                    close(fd);
                break;
            case 'd':
                /* detach from tty */
                if (getpgrp() == getpid()) {
                    switch(fork()) {
                        case -1:
                            perror("fork");
                            return 1;
                        case 0: /* child */
                            break;
                        default: /* parent */
                            return 0;
                    }
                }
                setsid();
                break;
            case 'n':
                /* run in network and mount namespaces */
                if (unshare(CLONE_NEWNET|CLONE_NEWNS) == -1) {
                    perror("unshare");
                    return 1;
                }

                /* Mark our whole hierarchy recursively as private, so that our
                 * mounts do not propagate to other processes.
                 */
        }
}

```

我们会发现当传入的参数中有n时，就会通过 `unshare(CLONE_NEWNET | CLONE_NEWNS)` 来创建新的网络 namespace 并挂载，而 `unshare()` 系统调用主要的作用便是在不启动一个新的进程情况下，也就是在当前的进程中执行，将当前进程分离出所在的 namespace 并加入新创建的 namespace，由此我们使用 `ip netns list` 便看不到该 namespace 的存在。

就类似于我们使用这样的命令：

```
sudo unshare --net --mount /bin/bash
```

我们在新的 namespace 的 shell 中执行 `ifconfig`，我们会看到没有任何信息，这就是因为我们在新的 namespace 中，并没有创建任何的虚拟网络接口，所以没有任何信息的输出。

```

shiyancelou:~$ sudo unshare --net --mount /bin/bash
root@VM-20-120-ubuntu:~/mininet# ifconfig
root@VM-20-120-ubuntu:~/mininet#

```

若还有兴趣验证的同学通过 `/proc/$$/ns` 查看两个 shell 的 namespace inode 值，可以发现新创建的 namespace 的 net 与 mnt 的 inode 值与之前的不同。

```
这是 unshare() 后的 bash
root@VM-20-120-ubuntu:~/mininet# ls -lah /proc/$$/ns
总用量 0
dr-x--x--x 2 root root 0 3月 30 09:59 .
dr-xr-xr-x 9 root root 0 3月 30 06:13 ..
lrwxrwxrwx 1 root root 0 3月 30 09:59 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 3月 30 09:59 mnt -> mnt:[4026532327]
lrwxrwxrwx 1 root root 0 3月 30 09:59 net -> net:[4026532329]
lrwxrwxrwx 1 root root 0 3月 30 09:59 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 3月 30 09:59 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 3月 30 09:59 uts -> uts:[4026531838]
root@VM-20-120-ubuntu:~/mininet#

这是原来的 zsh
shiyanolou:custom/ (master) $ sudo ls -lah /proc/$$/ns
总用量 0
dr-x--x--x 2 shiyanolou shiyanolou 0 3月 30 09:56 .
dr-xr-xr-x 9 shiyanolou shiyanolou 0 3月 29 14:21 ..
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 net -> net:[4026531956]
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 pid -> pid:[4026531836]
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 user -> user:[4026531837]
lrwxrwxrwx 1 shiyanolou shiyanolou 0 3月 30 09:58 uts -> uts:[4026531838]
shiyanolou:custom/ (master) $
```

这是不同的

6.3 nodes 命令

6.3.1 指令

通过 `dump` 我们可以查看所有节点的信息，但是我们若是只是想知道有哪些节点，我们就可以只使用 `nodes` 命令了：

```
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet>
```

6.3.2 原理


所有的命令行指令都在 `cli.py` 中，在该文件中我们可以找到这个函数：

```
def do_nodes( self, _line ):
    "List all nodes."
    nodes = ' '.join( sorted( self.mn ) )
    output( 'available nodes are: \n%s\n' % nodes )
```

6.4 net 命令

通过 `nodes` 命令我们可以查看到所有的节点，但是只是知道网络的节点还不够，我们还需要知道网络的连接，这种情况我们可以使用 `net` 命令：

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



6.5 节点执行命令

知道了网络的框架结构，若是我们需要知道某一个节点的具体配置，亦或者是需要让某个节点去做联通测试又该如何呢？


我们可以通过这样的命令来查看h1 主机的网络配置：

```
h1 ifconfig
```

```
mininet>
mininet> h1 ifconfig
h1-eth0  Link encap:以太网  硬件地址 7e:e6:a4:c0:73:75
         inet 地址:10.0.0.1  广播:10.255.255.255  掩码:255.0.0.0
         inet6 地址: fe80::7ce6:a4ff:fec0:7375/64  Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
         接收数据包:7 错误:0 丢弃:0 过载:0 帧数:0
         发送数据包:8 错误:0 丢弃:0 过载:0 载波:0
         碰撞:0 发送队列长度:1000
         接收字节:558 (558.0 B)  发送字节:648 (648.0 B)

lo       Link encap:本地环回
         inet 地址:127.0.0.1  掩码:255.0.0.0
         inet6 地址: ::1/128  Scope:Host
         UP LOOPBACK RUNNING  MTU:65536  跃点数:1
         接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
         发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
         碰撞:0 发送队列长度:0
         接收字节:0 (0.0 B)  发送字节:0 (0.0 B)

mininet>
```



与之前 `dump` 查看到的信息相同，命令便是 `节点名 ifconfig`，若我们想做连通性测试，了解h1能否与 h2 通信，我们可以这样：

```
h1 ping -c 4 h2
```

```
mininet> h1 ping -c 4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.22 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.052 ms

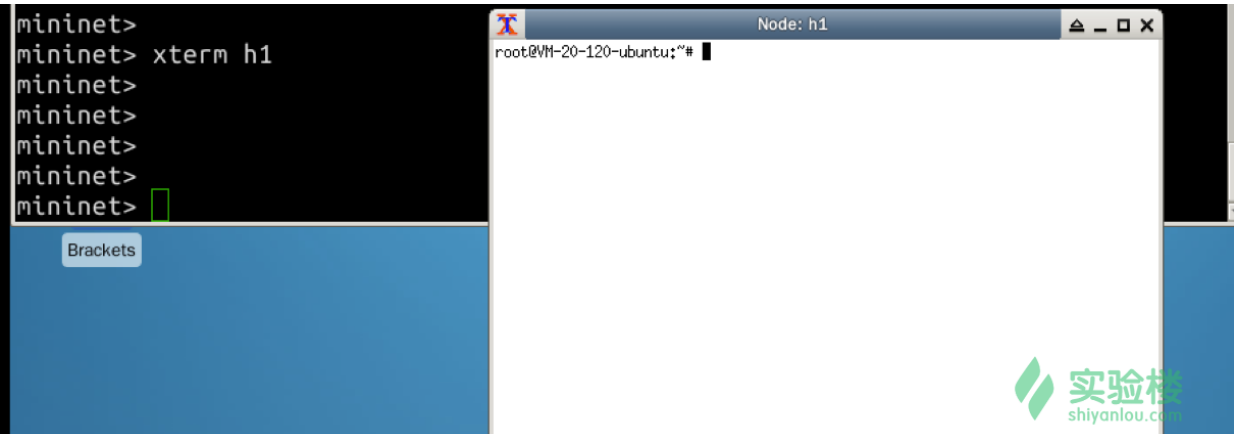
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.052/0.640/2.226/0.917 ms
mininet>
```

通过这样两个例子，其实我们很容易明白其实命令结构是这样的 **节点名 命令**，指定某个节点执行 Linux 中的某个命令。

6.6 xterm 命令

若是我们觉得通过指定节点名指定命令来执行很麻烦，我们可以直接为该节点开启一个终端：

```
xterm h1
```



当需要启动多个的时候可以是在后方加节点名称，用空格隔开。例如 **xterm h1 h2**。

目前 mininet 只支持启动两种终端：

- xterm
- gnome-terminal

若是想使用 gnome-terminal 的话，则使用 **gterm h1** 命令。

注意：当你的机器中没有这两种终端是输入响应的命令是没有任何反应的，因为系统没有办法调用到，例如需要使用 xterm 的时候，需要提前安装 xterm，在 ubuntu 中使用 `sudo apt-get install xterm`，若是使用的 ubuntu 的 desktop 版本，则可以直接使用 gnome-terminal。

6.7 pingall 命令

上述的方式都是一个节点去ping 另外一个节点，在小规模的网络是没有问题的，但是当在测试一个大规模的网络的时候我们会发现这样的方式非常的低效，这样的方式太过麻烦，所以有了这个偷懒的命令 `pingall`，自动的帮助我们测试所有节点的连通性：

```
pingall
```

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

与之类似的还有 `pingallfull`，`pingpair`、`pingpairfull`。其中 `pingpair` 的作用是两个节点的互 ping。

6.8 link 命令

当我们在一些场景中我们需要做一些局部的测试，我们可能需要关闭一些连接来排除一些因素，已确定我们的一些功能是否成功或者排查问题的出处，这个时候我们便需要使用 `link` 命令，例如：

```
link s1 h1 down
```

通过这样的方式我们便关闭了switch 与 h1 之间的网络连接，所以此时我们再用h1 ping h2 是不可达的：

```
mininet> link s1 h1 down
mininet> h1 ping -c 3 h2
connect: Network is unreachable
mininet> h2 ping -c 3 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2015ms
mininet>
```

当我们测试完毕，我们需要重新开启该链接的时候我们只需要这样的命令即可：

```
link s1 h1 up
```




```

mininet> link s1 h1 up
mininet> h1 ping -c 3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.19 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.248 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.063 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.063/0.836/2.198/0.966 ms
mininet> h2 ping -c 3 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.74 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.238 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.061 ms

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.061/0.681/1.745/0.755 ms
mininet>

```



6.9 iperf 命令

在完成网络拓扑的部署之后，我们可能还需要查看网络的性能如何，这个时候我们的第一反应可能会想到使用 `iperf`，mininet 中同样对 `iperf` 提供了支持：


Iperf:是美国伊利诺斯大学 (University of Illinois) 开发的一种网络性能测试工具。可以用来测试网络节点间TCP或UDP连接的性能，包括带宽、延时抖动 (jitter, 适用于UDP) 以及误码率 (适用于UDP) 等。若是有对 iperf 源码有兴趣的朋友推荐看[该博文](#)。

`iperf`

```

mininet>
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['28.4 Gbits/sec', '28.5 Gbits/sec']
mininet> iperf h1 h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['18.9 Gbits/sec', '18.9 Gbits/sec']
mininet>

```



`iperf` 可以完成两个节点之间简单的TCP测试，若需要UDP测试可以使用 `iperfudp` 命令。

6.10 dpctl 命令

以上我们看到了关于节点的命令，关于链路的相关命令。在mininet中还有关于 switch 的命令。

`dpctl` 是一个 OpenFlow 数据通路的检测和管理工具，它能够显示当前的状态数据通路，功能配置和表中的条目，可以用来对 datapaths 的添加，删除，修改和监控 (datapaths 将在后面为大家介绍)。

通过 `dpctl show` 可以查看交换机的一些基本信息：

```
mininet> dpctl show
*** s1 -----
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST SET_N
W_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST ENQUEUE
 1(s1-eth1): addr:2a:62:54:ab:a0:85
   config:      0
   state:       0
   current:     10GB-FD COPPER
   speed: 10000 Mbps now, 0 Mbps max
 2(s1-eth2): addr:e2:6a:26:02:b6:8b
   config:      0
   state:       0
   current:     10GB-FD COPPER
   speed: 10000 Mbps now, 0 Mbps max
LOCAL(s1): addr:ea:ca:77:fa:b3:4b
  config:      0
  state:       0
  speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
mininet>
```

还可通过 `dpctl dump-flows` 查看流表的具体信息，还有 `dpctl add-flow` 来添加一些表项等等。

命令有很多，我们可以在需要使用的时候通过 `dpctl --help` 来查看。

6.11 执行外部命令

当我们发现 mininet 这些内部命令还是不能满足我们的需求，我们还可以使用 `py` 与 `sh` 来执行一些 python 的表达式或者 shell 命令来完成我们想要的功能。

例如使用 `py locals()` 查看相关信息：

```
mininet> py locals()
{'h2': <Host h2: h2-eth0:10.0.0.2 pid=8778> , 'net': <mininet.net.Mininet object
at 0x7feb7495a350> , 'h1': <Host h1: h1-eth0:10.0.0.1 pid=8773> , 'c0': <Control
ler c0: 127.0.0.1:6653 pid=8766> , 's1': <OVSSwitch s1: lo:127.0.0.1,s1-eth1:Non
e,s1-eth2:None pid=8784> }
```

例如我们想单独查看 h1 的 IP 地址：

```
mininet> py h1.IP()
10.0.0.1
```

而若是想执行一些 shell 中的命令，只需使用 `sh 命令` 即可，例如：

```
mininet> sh ls
build-ovs-packages.sh  install.sh  openflow-patches  versioncheck.py
clustersetup.sh        kbuild      sch_htb-ofbuf      vm
colorfilters           m           sysctl_addon
doxify.py              nox-patches unpep8
mininet> sh pwd
/home/shiyanlou/mininet/util
```

6.12 help 命令

你可能会说你罗列了这么多的命令根本记不住，平时需要使用的时候忘记命令该怎么办？mininet 贴心的为你提供了 `help` 命令，查看到命令即可立即唤起你的记忆：

```
mininet> help

Documented commands (type help <topic>):
=====
EOF      gterm  iperfudp  nodes      pingpair    py      switch
dpctl    help   link      noecho     pingpairfull  quit    time
dump     intfs  links     pingall    ports       sh      x
exit     iperf  net       pingallfull px          source  xterm

You may also send a command to a node using:
  <node> command {args}
For example:
  mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
  mininet> h2 ping h3
should work.
```



并且在 mininet 中也同样拥有自动补全的功能，只记得半个命令，`tab` 键会帮你做剩下的事情。

这便是 mininet 命令行中的常用命令。

7. 总结

本节实验中我们学习了以下内容：

- mininet 原理
- mininet 指令

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. 在 mininet 中有很多命令都可以添加在启动参数中，例如 xterm, pingall。请通过 `mn --help` 查看相关指令，在启动的同时打开所有节点的 xterm 终端。
2. 查看推荐阅读的文章，能画出 mininet 构建一个 topo 的流程图。

9. 参考文章

- mininet 的 namespace 显示：<https://www.grotto-networking.com/BBSDNOverview.html>

