Mininet 代码分析

最新版: yeasy@github

更新历史:

V0.6: 2013-12-16
    完善对 topo 模块、net 模块等的分析。
V0.5: 2013-12-11
    补充对部分 example 文件的分析；
    增加对 clean 模块的分析。
V0.4: 2013-11-18
    补充对 link 和 node 库的核心代码分析；
    补充运行文件分析。
V0.3: 2013-11-16
    完成运行文件分析。
V0.2: 2013-11-15
    完成库文件分析。
V0.1: 2013-10-11
    完成代码结构分析。

1. 源代码结构

1.1. 运行相关

bin/mn

主运行文件，安装后执行 mn 即调用的本程序，是 python 程序。

mnexec.c

执行一些快速命令，比如关闭文件描述符等，是 C 程序，编译后生成二进制文件 mnexec 被 python 库调用。

1.2. Install 相关

INSTALL：安装说明

setup.py：安装 python 包时候的配置文件，被 Makefile 中调用。

debian/：生成 deb 安装包时的配置文件。

1.3. 核心代码

核心代码基本都在 mininet/子目录下。

注：最新的 2.1.0 版本，核心 python 代码仅为 4675 行。

```
find mininet  -name "*.py" | xargs cat | wc -l
```

1.4. 说明文件等

CONTRIBUTORS：作者信息

README.md：主说明文件

doc/doxygen.cfg：执行 doxygen 生成文档时的配置文件。

1.5. 其他文件

LICENSE

custom/ 目录下可以放一些用户自定义的 python 文件，比如自定义的拓扑类等。

test/目录下是一些测试的例子。

1.5.1. util/

目录下是一些辅助文件，包括安装脚本、文档辅助生成等，重要的文件包括：

1.5.1.1. m

bash 脚本提供用户直接在 host 执行命令的接口。例如

```
m host cmd args...
```

m 通过调用 mnexec 来实现对 Mininet 中的元素执行相应的命令。

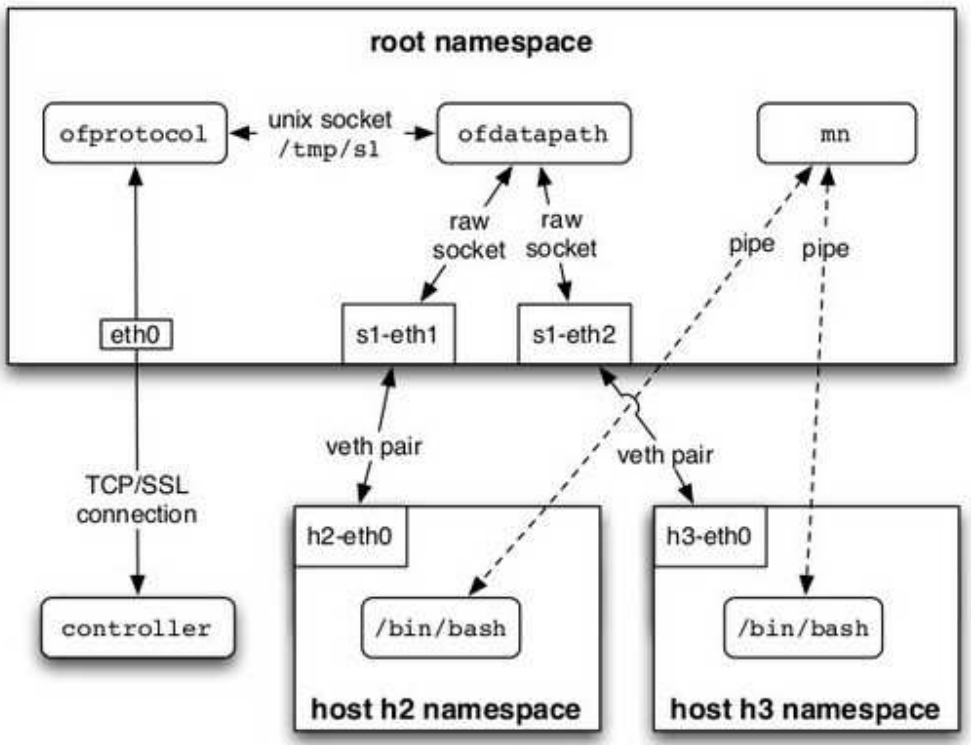
1.5.1.2. mnexec

C 程序，通过参数绑定到某个名字空间，并执行给定的命令。

1.6. 整体功能逻辑

整体上来看，mininet 作为一个基于 python 的网络模拟工具，可以分为两大部分：python 库和运行文件。前者提供对网络中元素进行抽象和实现，例如定义主机类来表示网络中的一台主机。后者则基于这些库来完成各种自定义的模拟过程。

一个典型的场景如下图所示。



图表 1 Mininet 模拟场景

2. 库文件分析

2.1. mininet.link 模块

描述链路相关的接口和连接。包括 Intf 类、Link 类、TCIntf 类和 TCLink 类。

2.1.1. mininet.link.Intf

表示基本的网络接口，比如 h1-eth0 表示 host 1 上的 eth0 接口。

属性包括所在的节点，名称，所接的 link，mac/ip 信息等。

构造的时候会传入节点、端口等属性，并绑定接口到对应的节点的端口上。

```
def __init__( self, name, node=None, port=None, link=None, **params ):
    """name: interface name (e.g. h1-eth0)
      node: owning node (where this intf most likely lives)
      link: parent link if we're part of a link
      other arguments are passed to config()"""
    self.node = node
    self.name = name
    self.link = link
    self.mac, self.ip, self.prefixLen = None, None, None
    # Add to node (and move ourselves if necessary )
    node.addIntf( self, port=port )
    # Save params for future reference
    self.params = params
    self.config( **params )
```

所支持的方法包括配置 mac/ip 等配置方法，大都是通过 ifconfig 命令在对应节点上调用 cmd 方法来实现。

此外，还提供了 config()方法来一次性配置所有的属性。

2.1.2. mininet.link.Link

表示基本的一条链路，最基本的链路在 mininet 中其实就是一对 veth 接口对。

```
def __init__( self, node1, node2, port1=None, port2=None,
              intfName1=None, intfName2=None,
              intf=Intf, cls1=None, cls2=None, params1=None,
              params2=None ):
    """Create veth link to another node, making two new interfaces.
      node1: first node
      node2: second node
      port1: node1 port number (optional)
      port2: node2 port number (optional)
      intf: default interface class/constructor
      cls1, cls2: optional interface-specific constructors
      intfName1: node1 interface name (optional)
      intfName2: node2  interface name (optional)
      params1: parameters for interface 1
      params2: parameters for interface 2"""
    # This is a bit awkward; it seems that having everything in
    # params would be more orthogonal, but being able to specify
    # in-line arguments is more convenient!
    if port1 is None:
       port1 = node1.newPort()
    if port2 is None:
       port2 = node2.newPort()
    if not intfName1:
       intfName1 = self.intfName( node1, port1 )
    if not intfName2:
       intfName2 = self.intfName( node2, port2 )

    self.makeIntfPair( intfName1, intfName2 )

    if not cls1:
       cls1 = intf
    if not cls2:
       cls2 = intf
```

```
        if not params1:
            params1 = {}
        if not params2:
            params2 = {}

        intf1 = cls1( name=intfName1, node=node1, port=port1,
                      link=self, **params1  )
        intf2 = cls2( name=intfName2, node=node2, port=port2,
                      link=self, **params2 )

        # All we are is dust in the wind, and our two interfaces
        self.intf1, self.intf2 = intf1, intf2
```

创建链路时，需要在两个节点上分别生成两个端口，利用节点和端口，获取对应的两个网络接口的名称，例如 s1-eth0 和 h1-eth0，然后调用 makeIntfPair()方法，最终调用 util.py 中的 makeIntfPair()方法，调用系统中的 ip link 命令来创造一堆 veth pair。

```
def makeIntfPair( intf1, intf2 ):
    """Make a veth pair connecting intf1 and intf2.
      intf1: string, interface
      intf2: string, interface
      returns: success boolean"""
    # Delete any old interfaces with the same names
    quietRun( 'ip link del ' + intf1 )
    quietRun( 'ip link del ' + intf2 )
    # Create new pair
    cmd = 'ip link add name ' + intf1 + ' type veth peer name ' + intf2
    return quietRun( cmd )
```

2.1.3. mininet.link.TCIntf

被 TC（linux 下的 traffic control 的工具）自定义的接口，可以配置包括带宽、延迟、丢包率、最大队列长度等参数。

2.1.4. mininet.link.TCLink

表示一对对称的 TC 接口连接到一起。

2.2. mininet.node 模块

节点模块表示网络中的基本元素（包括主机、交换机和控制器），十分关键。

其中，每个主机默认在一个单独的名字空间中，交换机和控制器都在 root 名字空间中。

2.2.1. 基类

2.2.1.1. mininet.node.Node

表示一个基本的虚拟网络节点，是所有的网络节点的父类。

实现上其实就是在网络名字空间中的一个 shell 进程，可以通过各种管道进行通信。该类是模块中其他类的根本，其它类都是直接或间接继承。

节点包括名称、是否在网络名字空间、接口、端口等可能的属性。

__init__

```
params: Node parameters (see config() for details)"""

    # Make sure class actually works
    self.checkSetup()

    self.name = name
    self.inNamespace = inNamespace

    # Stash configuration parameters for future reference
    self.params = params

    self.intfs = {}  # dict of port numbers to interfaces
    self.ports = {}  # dict of interfaces to port numbers
                # replace with Port objects, eventually ?
    self.nameToIntf = {}  # dict of interface names to Intfs

    # Make pylint happy
    ( self.shell, self.execed, self.pid, self.stdin, self.stdout,
      self.lastPid, self.lastCmd, self.pollOut ) = (
          None, None, None, None, None, None, None, None )
    self.waiting = False
    self.readbuf = ''

    # Start command interpreter shell
    self.startShell()
```

初始化函数主要进行参数的初始化，之后通过调用 startShell()启动一个 shell 进程（该进程默认关闭描述符，并从 tty 上分离开来），等待接受传入的命令。句柄被发送给 self.shell 上。

addIntf

```
def addIntf( self, intf, port=None ):
    """Add an interface.
       intf: interface
       port: port number (optional, typically OpenFlow port number)"""
    if port is None:
        port = self.newPort()
    self.intfs[ port ] = intf
    self.ports[ intf ] = port
    self.nameToIntf[ intf.name ] = intf
    debug( '\n' )
    debug( 'added intf %s:%d to node %s\n' % ( intf, port, self.name ) )
    if self.inNamespace:
        debug( 'moving', intf, 'into namespace for', self.name, '\n' )
        moveIntf( intf.name, self )
```

添加一个接口（比如<Intf h1-eth0>）到节点上，如果给定了 port（比如 0），则建立端口到接口的映射关系。这个映射关系通过 self.intfs 和 self.ports 两个字典来分别维护。

cmd

该函数能在节点所在的进程 shell 上执行输入的命令。

```
def cmd( self, *args, **kwargs ):
    """Send a command, wait for output, and return it.
       cmd: string"""
    verbose = kwargs.get( 'verbose', False )
    log = info if verbose else debug
    log( '*** %s : %s\n' % ( self.name, args ) )
    self.sendCmd( *args, **kwargs )
    return self.waitOutput( verbose )
```

config

配置 mac，ip 或 defaultRoute 信息。

```
def config( self, mac=None, ip=None,
            defaultRoute=None, lo='up', **_params ):
    """Configure Node according to (optional) parameters:
      mac: MAC address for default interface
      ip: IP address for default interface
      ifconfig: arbitrary interface configuration
      Subclasses should override this method and call
      the parent class's config(**params)"""
    # If we were overriding this method, we would call
    # the superclass config method here as follows:
    # r = Parent.config( **_params )
    r = {}
    self.setParam( r, 'setMAC', mac=mac )
    self.setParam( r, 'setIP', ip=ip )
    self.setParam( r, 'setDefaultRoute', defaultRoute=defaultRoute )
    # This should be examined
    self.cmd( 'ifconfig lo ' + lo )
    return r
```

connectionsTo

返回所有从自身连接到给定节点的接口，即[intf1, intf2...]。

```
def connectionsTo( self, node):
    "Return [ intf1, intf2... ] for all intfs that connect self to node."
    # We could optimize this if it is important
    connections = []
    for intf in self.intfList():
        link = intf.link
        if link:
            node1, node2 = link.intf1.node, link.intf2.node
            if node1 == self and node2 == node:
                connections += [ ( intf, link.intf2 ) ]
            elif node1 == node and node2 == self:
                connections += [ ( intf, link.intf1 ) ]
    return connections
```

2.2.2. 主机类

2.2.2.1. mininet.node.Host

表示一个主机节点，目前跟 Node 类定义相同。

在主机类上执行命令可以通过 Cmd()或者 sendCmd()方法，前者会等待命令的输出结果，后者会直接返回，并允许使用后续的 monitor()来进行监视跟踪。

2.2.2.2. mininet.node.CPULimitedHost

继承自 Host 类，通过 cgroup 工具来对 cpu 进行限制。

__init__

```
def __init__( self, name, sched='cfs', **kwargs ):
    Host.__init__( self, name, **kwargs )
    # Initialize class if necessary
    if not CPULimitedHost.inited:
       CPULimitedHost.init()
    # Create a cgroup and move shell into it
    self.cgroup = 'cpu,cpuacct,cpuset:/' + self.name
    errFail( 'cgcreate -g ' + self.cgroup )
    # We don't add ourselves to a cpuset because you must
    # specify the cpu and memory placement first
    errFail( 'cgclassify -g cpu,cpuacct:/%s %s' % ( self.name, self.pid ) )
    # BL: Setting the correct period/quota is tricky, particularly
    # for RT. RT allows very small quotas, but the overhead
    # seems to be high. CFS has a mininimum quota of 1 ms, but
    # still does better with larger period values.
    self.period_us = kwargs.get( 'period_us', 100000 )
    self.sched = sched
    self.rtprio = 20
```

2.2.3. 控制器类

2.2.3.1. mininet.node.Controller

控制器基类。默认的控制器是一个参考的实现，controller。

表示一个控制器节点。包括 ip 地址、端口等。主要方法包括启动和停止一个控制器。

__init__

```
def __init__( self, name, inNamespace=False, command='controller',
              cargs='-v ptcp:%d', cdir=None, ip="127.0.0.1",
              port=6633, **params ):
    self.command = command
    self.cargs = cargs
    self.cdir = cdir
    self.ip = ip
    self.port = port
    Node.__init__( self, name, inNamespace=inNamespace,
                   ip=ip, **params  )
    self.cmd( 'ifconfig lo up' )  # Shouldn't be necessary
    self.checkListening()
```

checkListening

确保系统中安装了 telnet，并且在监听端口上并没有其他控制器在监听。

start

```
def start( self ):
    """Start <controller> <args> on controller.
      Log to /tmp/cN.log"""
    pathCheck( self.command )
    cout = '/tmp/' + self.name + '.log'
    if self.cdir is not None:
       self.cmd( 'cd ' + self.cdir )
    self.cmd( self.command + ' ' + self.cargs % self.port +
          ' 1>' + cout + ' 2>' + cout + '&' )
    self.execed = False
```

该方法检查控制器 程序存在，并根据传入的参数来启动它。

stop

```
def stop( self ):
    "Stop controller."
    self.cmd( 'kill %' + self.command )
    self.terminate()
```

该方法杀死控制器进程和节点的 shell 进程，并执行清理工作。

2.2.3.2. mininet.node.NOX

表示一个 NOX 控制器（需要系统中事先安装了 NOX）。

2.2.3.3. mininet.node.OVSController

表示一个 ovs-controller（需要系统中实现安装了 ovs-controller）。

2.2.3.4. mininet.node.RemoteController

表示一个在 mininet 控制外的控制器，即用户自己额外运行了控制器，此处需要维护连接的相关信息。

2.2.4. 交换机类

2.2.4.1. mininet.node.Switch

表示一个交换机的基类。

运行在 root 名字空间。主要包括 dpid、listenport 等属性。

2.2.4.2. mininet.node.IVSSwitch

表示一台 indigo 交换机（需要系统中已存在）。

2.2.4.3. mininet.node.OVSLegacyKernelSwitch

传统的 openvswitch 交换机，基于 ovs-openflowd。不推荐。

2.2.4.4. mininet.node.OVSSwitch

表示一台 openvswitch 交换机（需要系统中已经安装并配置好 openvswitch），基于 ovs-vsctl 进行操作。目前所谓的 OVSKernelSwitch 实际上就是 OVSSwitch

2.2.4.5. mininet.node.UserSwitch

用户态的 openflow 参考交换机，即 ofdatapath。不推荐。

2.3. mininet.net 模块

主要包括 Mininet 和 MininetWithControlNet 两个类。

2.3.1. mininet.net.Mininet

模拟一个 mininet 中的网络，包括拓扑、交换机、主机、控制器、链路、接口等。

其中最主要的部分是 build()函数，依次执行：根据拓扑创建网络，配置网络名字空间，配置主机的 ip、mac 等信息，检查是否启动 xterm，是否配置自动静态 arp 等。

2.3.1.1. __init__

```
        inNamespace=False,
             autoSetMacs=False, autoStaticArp=False, autoPinCpus=False,
             listenPort=None ):
        """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
        self.host = host
        self.controller = controller
        self.link = link
        self.intf = intf
        self.ipBase = ipBase
        self.ipBaseNum, self.prefixLen = netParse( self.ipBase )
        self.nextIP = 1  # start for address allocation
        self.inNamespace = inNamespace
        self.xterms = xterms
        self.cleanup = cleanup
```

```
        self.autoSetMacs = autoSetMacs
        self.autoStaticArp = autoStaticArp
        self.autoPinCpus = autoPinCpus
        self.numCores = numCores()
        self.nextCore = 0  # next core for pinning hosts to CPUs
        self.listenPort = listenPort

        self.hosts = []
        self.switches = []
        self.controllers = []

        self.nameToNode = {}  # name to Node (Host/Switch) objects

        self.terms = []  # list of spawned xterm processes

        Mininet.init()  # Initialize Mininet if necessary

        self.built = False
        if topo and build:
            self.build()
```

该方法主要根据传入参数配置相关的数据结构。如果还通过参数执行了拓扑信息，则执行 build 方法，调用 buildFromTopo 方法来根据拓扑信息创建节点和相关的连接等。

2.3.1.2. buildFromTopo

```
def buildFromTopo( self, topo=None ):
    """Build mininet from a topology object
       At the end of this function, everything should be connected
       and up."""

    # Possibly we should clean up here and/or validate
    # the topo
    if self.cleanup:
        pass

    info( '*** Creating network\n' )

    if not self.controllers and self.controller:
        # Add a default controller
        info( '*** Adding controller\n' )
        classes = self.controller
        if type( classes ) is not list:
            classes = [ classes ]
        for i, cls in enumerate( classes ):
            self.addController( 'c%d' % i, cls )

    info( '*** Adding hosts:\n' )
    for hostName in topo.hosts():
        self.addHost( hostName, **topo.nodeInfo( hostName ) )
        info( hostName + ' ' )

    info( '\n*** Adding switches:\n' )
    for switchName in topo.switches():
        self.addSwitch( switchName, **topo.nodeInfo( switchName) )
        info( switchName + ' ' )

    info( '\n*** Adding links:\n' )
```

```
    for srcName, dstName in topo.links(sort=True):
        src, dst = self.nameToNode[ srcName ], self.nameToNode[ dstName ]
        params = topo.linkInfo( srcName, dstName )
        srcPort, dstPort = topo.port( srcName, dstName )
        self.addLink( src, dst, srcPort, dstPort, **params )
        info( '(%s, %s) ' % ( src.name, dst.name ) )

    info( '\n' )
```

2.3.2. mininet.net.MininetWithControlNet

继承自 Mininet 类，主要用于在使用用户态 datapath 的时候模拟一个控制器网络，即连接用户态的交换机和用户态的控制器。

2.4. mininet.cli 模块

主要包括 CLI 类，该类继承自 python 库的 Cmd 类。

提供对 CLI 的支持，创建 mininet 的 bash，接受通过 bash 传输的 mininet 命令，形成可以进行交互的 mininet 命令行环境。

主要方法包括初始化之后提供一个界面，通过 python 库的 Cmd 类的 cmdloop()方法不断执行输入的命令。这些命令可以是指定对某个节点进行的操作或者是对 mininet 对象本身。

对于大部分对 mininet 对象的操作命令 xxx，会调用 python 库的 Cmd 类的 onecmd()方法来执行，该方法会对应调用 do_xxx 方法。

各个 do_xxx 方法分别用于执行支持的命令。例如 do_dpctl()方法响应用户输入 dpctl 相关命令。目前包括 dpctl，dump，EOF，exit，gterm，help，intfs，iperf，iperfudp，link，net，nodes，noecho，pingall，pingallfull，pingpair，pingparifull，px，py，quit，sh，source，time，x，xterm 等。

2.5. mininet.clean 模块

提供对执行 mininet 后的清理工作，主要包括 cleanup()函数，该函数实际上调用了 sh()函数。

cleanup()函数主要包括清除僵尸进程，临时文件，X11 tunnel，额外的内核态 datapath，ovs datapath，ip link 等。

实现过程主要是通过调用 subprocess 模块（主要用于执行外部命令和程序）中的 Popen 类中方法来对进程发送指令。

```
def sh( cmd ):
   "Print a command and send it to the shell"
   info( cmd + '\n' )
   return Popen( [ '/bin/sh', '-c', cmd ], stdout=PIPE ).communicate()[ 0 ]
```

communicate()是 Popen 对象的一个方法，该方法会阻塞父进程，直到子进程完成。

通过指定 stdout=PIPE，可以通过 stdout 获取程序的返回值。通过列表传入要执行的命令和参数。

2.6. mininet.log 模块

利用 logging 包，主要提供进行 log 相关的功能，包括三个类：MininetLogger、Singleton、StreamHandlerNoNewline。

2.6.1. mininet.log.MininetLogger

自定义的 logger 类。

提供输出 log、配置 LogLevel 功能。

2.6.2. mininet.log.Singleton

软件设计模式，限定所创建的类只能有一个实例。供 MininetLogger 使用。

2.6.3. mininet.log.StreamHandlerNoNewline

自动添加换行和对流进行格式化，供 MininetLogger 使用。

2.7. mininet.topo 模块

维护网络拓扑相关的信息。

2.7.1. mininet.topo.MultiGraph

表示一个图结构。

类似于 networkx 中的图 G(V,E)的概念。主要维护节点、边信息。

在 MultiGraph 中，节点就是一个序号，边则通过节点和节点所对应的连接列表中元素来表示。节点和节点的连接列表的对应关系通过字典结构来维护。

__init__

```
def __init__( self ):
    self.data = {}
```

图结构最主要的功能就是维护一个字典。Key 是节点，value 是该节点所连接的所有的其他节点的列表。

add_node

```
def add_node( self, node ):
    "Add node to graph"
    self.data.setdefault( node, [] )
```

添加一个节点，实际上就是添加一个 key 到 data 字典中。

add_edge

```
def add_edge( self, src, dest ):
    "Add edge to graph"
    src, dest = sorted( ( src, dest ) )
    self.add_node( src )
    self.add_node( dest )
    self.data[ src ].append( dest )
```

添加一条边，实际上就是添加两个节点，然后将连接信息放到 data 字典中（需要注意的是一条边的信息仅被保存了一次，即放到序号较小的节点对应的 list 中）。

2.7.2. mininet.topo.Topo

拓扑基类，默认的拓扑图被 multigraph 类维护，此外还包括节点、连接信息等。主要的方法就是添加节点、连接等。

Topo 中一个 node 实际上就是图结构中的一个节点，一个 port 是全局维护增长的源和目的 node 所对应的序号，而连接

__init__

```
def __init__(self, hopts=None, sopts=None, lopts=None):
    """Topo object:
      hinfo: default host options
      sopts: default switch options
      lopts: default link options"""
    self.g = MultiGraph()
    self.node_info = {}
    self.link_info = {}  # (src, dst) tuples hash to EdgeInfo objects
    self.hopts = {} if hopts is None else hopts
    self.sopts = {} if sopts is None else sopts
    self.lopts = {} if lopts is None else lopts
    self.ports = {}  # ports[src][dst] is port on src that connects to dst
```

addNode

```
def addNode(self, name, **opts):
    """Add Node to graph.
      name: name
      opts: node options
      returns: node name"""
    self.g.add_node(name)
    self.node_info[name] = opts
    return name
```

添加节点方法被添加主机 addHost、交换机 addSwitch、控制器 addController 等方法使用。

该方法在图上添加一个节点，然后添加对应的节点信息到拓扑的参数上。

addPort

```
def addPort(self, src, dst, sport=None, dport=None):
    '''Generate port mapping for new edge.
    @param src source switch name
    @param dst destination switch name
    '''
    self.ports.setdefault(src, {})
    self.ports.setdefault(dst, {})
    # New port: number of outlinks + base
    src_base = 1 if self.isSwitch(src) else 0
    dst_base = 1 if self.isSwitch(dst) else 0
    if sport is None:
        sport = len(self.ports[src]) + src_base
    if dport is None:
        dport = len(self.ports[dst]) + dst_base
    self.ports[src][dst] = sport
    self.ports[dst][src] = dport
```

addPort 会同时创建源和目标节点上的端口号信息。

拓扑中所有的 port 都被 self.ports 结构维护，其中 ports[src][dst]表示在 src 节点上的 port，该 port 所在 link 连接到 dst 节点。

添加一个 port 就是更新了这些信息。

addLink

```
def addLink(self, node1, node2, port1=None, port2=None,
            **opts):
    """node1, node2: nodes to link together
      port1, port2: ports (optional)
      opts: link options (optional)
      returns: link info key"""
    if not opts and self.lopts:
       opts = self.lopts
    self.addPort(node1, node2, port1, port2)
    key = tuple(self.sorted([node1, node2]))
    self.link_info[key] = opts
    self.g.add_edge(*key)
    return key
```

添加一条连接实际上就是添加对应的两个 port，并在图上添加上边。

2.7.3. mininet.topo.LinearTopo

表示一个线行拓扑，交换机连接成一条链，每个交换机上挂载相等个数的主机。

2.7.4. mininet.topo.SingleSwitchTopo

单个交换机上挂载若干主机，主机序号按照从小到大的顺序依次挂载到交换机的各个端口上。

2.7.5. mininet.topo.SingleSwitchReversedTopo

单个交换机上挂载若干主机，主机序号按照从大到小的顺序依次挂载到交换机的各个端口上。

2.8. mininet.topolib

提供用户自己创建复杂拓扑相关的库，目前仅包括一个 Tree 拓扑。

2.8.1. mininet.topolib.TreeTopo

树拓扑类，给定深度和广度可以自己生成相应的标准树拓扑。

2.9. mininet.moduledeps 模块

定义几个对 linux 系统中内核模块进行操作的函数，包括列出模块 lsmod，移除模块 rmmod，探测模块 modprobe 和处理模块的依赖等。

2.10. mininet.term 模块

支持 term 相关的命令，例如在主机上创建一个 xterm。实现依赖于 socat 和 xterm。

2.11. mininet.util 模块

一些辅助的方法。包括如下重要的方法。

2.11.1. errFail

利用 errRun（利用 popen 来在 shell 中执行命令）来执行一个命令，并且如果执行不成功则抛出异常。

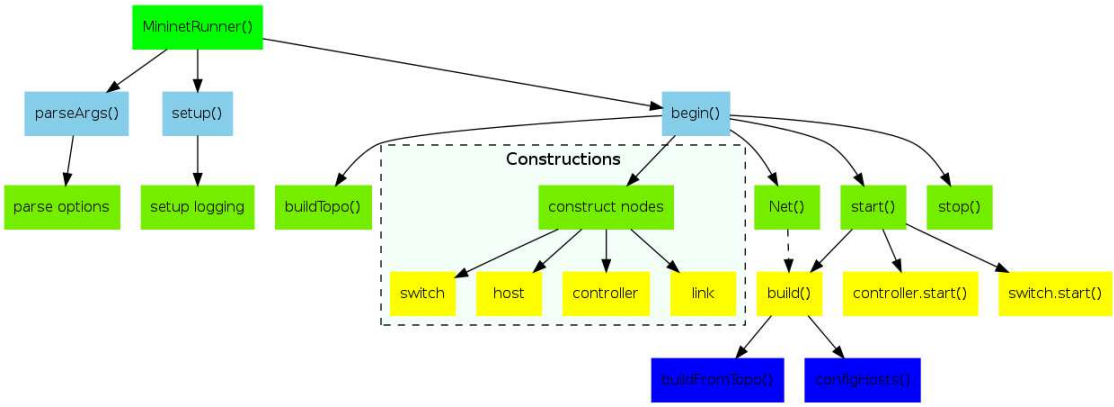
3. 运行文件分析

3.1. mn 脚本

该脚本定义了一个 MininetRunner 类，用来表示模拟网络的主程序。

主要过程是创建一个 MininetRunner()实例，依次解析传入参数，进行初始化后开启网络。

整体过程如下图所示。



图表 2

图表 3 mn 脚本主要过程

其中 Mininet 类的 start()方法是核心的启动过程，主要包括调用 build 方法来根据拓扑创建网络、控制器、交换机、主机和连接等。之后依次启动控制器和交换机进程。

在执行完 start()之后，通过 test 参数来判断 mininet 运行的模式。

```
if test == 'none':
        pass
    elif test == 'all':
        mn.start()
        mn.ping()
        mn.iperf()
    elif test == 'cli':
        CLI( mn )
    elif test != 'build':
        getattr( mn, test )()
```

默认情况下，参数为 cli，即进入到控制台模式，允许用户自己输入对 mininet 的操作命令。

最终执行 mininet.stop()进行删除资源的工作。

4. 示例程序

Mininet 代码中带有了大量的示例程序，供大家参考和理解代码。所有的示例程序都在 example 目录下，包括

4.1. baresshd.py:

使用 Mininet's 的中层 API 来在一个 namespace 中创建主机、链路，并在主机上启动 sshd 进程，让用户可以登录。并未使用 OpenFlow。

4.2. consoles.py:

为每一个节点都创建一些 console 窗口，并允许用户对这些节点进行操作和观测，支持图形界面。

4.3. controllers.py:

使用一个自定义的 Switch() 子类，创建一个带有多个控制器的网络。

4.4. controllers2.py:

创建一个拥有多个控制器的网络，通过创建空的网络，添加节点和手动启动交换机实现。

4.5. controlnet.py:

通过创建两个 mininet 对象来建模一个控制网络和数据网络。

4.6. cpu.py:

在不同的 CPU 限制下测试 iperf 的带宽性能。

4.7. emptynet.py:

演示创建一个空的网络，之后添加节点进去。

4.8. hwintf.py:

添加一个接口（例如一个物理接口）到一个网络中。

4.9. limit.py:

演示如何使用 link 和 CPU 限制。

4.10. linearbandwidth.py:

基于 Topo 创建一个拓扑子类，并进行简单的测试。

4.11. miniedit.py:

通过一个图形界面的编辑器来创建网络。

4.12. multiping.py:

使用 node.monitor()来检测多个主机的输出。

4.13. multipoll.py:

检测多个主机的输出文件。

4.14. multitest.py:

创建一个网络，并在其上进行多个测试。

4.15. nat.py:

将 Mininet 的网络通过 nat 连接到外部网络中。

4.16. popen.py:

使用 host.popen()和 pmonitor()来检测多个主机。

4.17. popenpoll.py:

使用 node.popen()和 pmonitor()检测多个主机的输出。

4.18. scratchnet.py, scratchnetuser.py:

使用底层的 Mininet 函数来创建网络。

4.19. simpleperf.py:

配置网络和 cpu 带宽限制等。

4.20. sshd.py:

在每个主机里面运行一个 sshd 进程，使得用户可以通过 ssh 来访问主机。这需要将 Mininet 的数据网络连接到 root 名字空间的一个接口上。一般的，控制网络已经在 root 名字空间了，所以默认已经被连接。

4.21. tree1024.py:

创建一个 1024 主机的网络，然后运行 CLI。根据系统资源情况，可能需要利用 sysctl 进行相关修改。

4.22. treeping64.py:

创建一个 64 主机的树状网络，利用 ping 来检查连通性。