FloodLight源码解读（所有模块）

整理—中南民族大学#Rajesh-

（sina@学霸无敌gogogo）

Mail:strongmaple@163.com

参考原文博客地址 http://lamoop.com/

首发布:www.sdnap.com

目录

前言

非常感谢AncerHaides所做的工作，他在2012年十二月左右，阅读了FloodLight所有的源码，同时自己也实现了若干控制器。我们应该向他踏实的态度学习，FloodLight使用Java开发，比较简洁，容易上手，所以在此希望所有参加SDN比赛的同学能够静下心来，认真做自己的事情，不要被外界干扰。

@AncerHaides心路历程

前一段时间忙死了，把openflow文档看了一编，全英文了，英语水平又提高不少；又把pox的代码基本上看了一遍，然后写几个openflow控制器，然后把floodlight的文档又看了一遍，不过还没用floodlight写控制器，然后又用oftest把实验室用netfpga做的openflow交换机测试了一下。基本上算是告一段了啦，年前再用flowscale做个负载均衡就差不多不干活了。

前段时间看floodlight的时候一时兴起，就边看边翻译，这样好处挺多的，把自己做的东西给记录了，以后说不定还能用上，还能锻炼一下自己的英语水平，翻译完发出来还可以供大家参考，呵呵。这次就是把那些翻译的文档给发布了，我英语水平有限，翻译的时候也是觉得自己能看懂就行，因此里边很多句子翻译的非常不通顺，要是你不太理解里边的内容还是看英文原版比较好。另外，鉴于现在好多人都没有读完一篇800字文章的能力，甚至有些人只能一口气读完140个字，我每篇文章只写一个模块。

一、Floodlight快速入门 添加module，并启动

a.新建一个类继承IOFMessageListerner和IFloodlightModule。

b.声明变量

```
protected IFloodlightProviderService floodlightProvider;
protected Set macAddresses;
protected static Logger logger;
```

c.实现getModuleDependencies方法，把依赖关系告诉模块加载系统

```
@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
    Collection<Class<? extends IFloodlightService>> l =
        new ArrayList<Class<? extends IFloodlightService>>();
    l.add(IFloodlightProviderService.class);
    return l;
}
```

d.实现init方法，这个方法会在controller启动时调用，以加载依赖和数据结构

```
@Override
public void init(FloodlightModuleContext context) throws FloodlightModuleException {
    floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
    macAddresses = new ConcurrentSkipListSet<Long>();
    logger = LoggerFactory.getLogger(MACTracker.class);
}
```

e. 实现startUp方法，为PACKET_IN消息绑定事件处理委托，在这之前我们必须保证所有依赖的模块已经初始化+

```
@Override
public void startUp(FloodlightModuleContext context) {
    floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this);
}
```

f.实现getName方法为OFMessage监听器添加ID

```
@Override
public String getName() {
    return "MACTracker";
}
```

g.为PACKET_IN事件处理程序添加实现代码，该方法返回Command.CONTINUE以便其它事件处理程序继续处理。

```
@Override
public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
 Ethernet eth
=IFloodlightProviderService.bcStore.get(cntx,IFloodlightProviderService.CONTEXT_PI_PAYLOAD);
Long sourceMACHash = Ethernet.toLong(eth.getSourceMACAddress());
 if (!macAddresses.contains(sourceMACHash))
{
  macAddresses.add(sourceMACHash);
logger.info("MAC Address: {} seen on switch:{}",HexString.toHexString(sourceMACHash),
sw.getId());
  }
 return Command.CONTINUE;
    }
```

h. 编写完事件处理程序还需要向Floodlight注册模块，这样Floodlight启动的时候才能加载，通过向src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule文件中添加以下语句告诉模块加载器该模块的存在

```
net.floodlightcontroller.mactracker.MACTracker
```

i.为了能让Floodlight启动时加载模块，我们还要向Floodlight模块配置文件中添加MACTracker，默认的配置文件是src/main/resources/floodlightdefault.properties，该文件中的键是floodlight，键值是以逗号分隔的各个模块名称

```
floodlight.modules = <leave the default list of modules in place>,
net.floodlightcontroller.mactracker.MACTracker
```

j.启动Main.java

k 验证成功

```
54:24.244 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 connected.
54:24.328 INFO [g.n.f.m.MACTracker:New I/O server worker #2-2] MAC Address:00:00:c6:f0:78:6b:a7:dd seen on switch :1
54:24.931 INFO [g.n.f.m.MACTracker:New I/O server worker #2-2] MAC Address:00:00:1a:9c:ef:9b:ab:43 seen on switch :1
54:25.183 INFO [g.n.f.m.MACTracker:New I/O server worker #2-2] MAC Address:00:00:36:9b:6b:4e:2b:66 seen on switch :1
```

二、floodlight架构模块分析

前一段时间忙死了，把openflow文档看了一编，全英文了，英语水平又提高不少；又把pox的代码基本上看了一遍，然后写几个openflow控制器，然后把floodlight的文档又看了一遍，不过还没用floodlight写控制器，然后又用oftest把实验室用netfpga做的openflow交换机测试了一下。基本上算是告一段了啦，年前再用flowscale做个负载均衡就差不多不干活了。

前段时间看floodlight的时候一时兴起，就边看边翻译，这样好处挺多的，把自己做的东西给记录了，以后说不定还能用上，还能锻炼一下自己的英语水平，翻译完发出来还可以供大家参考，呵呵。这次就是把那些翻译的文档给发布了，我英语水平有限，翻译的时候也是觉得自己能看懂就行，因此里边很多句子翻译的非常不通顺，要是你不太理解里边的内容还是看英文原版比较好。另外，鉴于现在好多人都没有读完一篇800字文章的能力，甚至有些人只能一口气读完140个字，我每篇文章只写一个模块。

嗯，废话说了一大堆，下边进入正题。

Floodlight使用模块框架实现控制器特性和一些应用，模块加载系统介绍了这个框架实现的IFloodlight接口。在功能上，Floodlight由控制器模块和应用模块组成，控制器模块实现了核心的网络服务并为应用程序提供接口，应用模块根据不同的目的实现不同的方案。

1. 控制器模块
2. 发现并且展示网络的状态和事件（拓扑，设备，流）
3. 控制器和交换机之间的通信
4. 管理Floodlight模块并共享资源（存储、线程、测试）
5. 一个web界面和调试服务器

目前以下列表实现了控制器模块：

1. FloodlightProvider（Dev）
2. DeviceManagerImpl（Dev）
3. LinkDiscoveryManager（Dev）
4. TopologyService（Dev）
5. RestApiServer（Dev）
6. ThreadPool（Dev）
7. MemoryStorageSource（Dev）
8. Flow Cache（API only）
9. Packet Streamer

和以下应用模块：

1. VirtualNetworkFilter
2. Forwarding
3. Firewall
4. Port Down Reconciliation

文章是从word里粘贴过来的，里边有些格式似乎乱了，不过不太影响阅读。

三、floodlight DeviceManagerImpl (Dev)

使用MAC地址和VLAN表示一个设备

1)简介：当设备在一个网络中移动的时候跟踪设备，并且为新流定义目的设备。

2)服务提供者：IDeviceService

3)服务依赖：IStorageSourceService、IRestApiService、ICounterStoreService、IThreadPoolService、IFlowReconcileService、IFloodlightProviderService

4)java文件：net.floodlightcontroller.devicemanger.internal.DeviceManageImpl

5)工作原理：设备管理器从PacketIn请求中得知设备，并从PackeetIn报文中获取设备信息，通过实体分类器将设备进行分类。默认情况下实体分类器使用MAC地址和VLAN表示一个设备，这两个属性可以唯一的标识一个设备，设备管理器也可以获取到其它的属性，例如IP地址。另外一个重要的信息是设备的安装点，在一个openflow区域中，一个设备只能有一个安装点，在这里openflow区域指的是和同一个Floodlight实例相连的多个交换机的集合。设备管理器也为IP地址、安装点、设备设置了过期时间，最后一次时间戳作为判断它们是否过期的依据。

6)局限性：设备是不可变的（immutable），这就意味着你不能持有一个设备的引用，它们只能通过IDeviceService APIs查询。

7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块。

8)配置参数：暂无

9)REST API：

四、floodlight LinkDiscoveryManager

LinkDiscoveryManager（Dev）

1)简介：发现和维护openflow网络中的链路状态。

2)服务提供者：ILinkDiscoveryService

3)服务依赖：IstorageSourceService、IThreadPoolService、IFloodlightProviderService

4)java文件：net.flooldlightcontroller.linkdiscovery.internal.LinkDiscoverManager

5)工作原理：链路发现服务使用LLDP和广播包（aka BDDPs）发现链路，LLDP的目的MAC地址是01:80:C2:00:00:0E，BDDP的目的地址是FF:FF:FF:FF:FF:FF，LLDP的以太网帧类型是0X88CC，BDDP的以太网帧类型是0X8999，链路发现服务能够正确地认知拓扑图是建立在两个假设上的：①所有的交换机都会销毁link-local packet（LLDP）②Honors layer 2 broadcasts。

链路可以是直连的，也可以是广播。如果LLDP包从一个端口发出去，另外一个端口收到了相同的LLDP包，说明这两个端口是直连的，就会建立一个直连链路；如果一个BDDP包从一个端口发送，在其它的端口接收，说明在两个交换机之间有控制器无法控制的二层交换机，就会建立一个广播链路。

6)局限性：暂无

7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块。

8)配置参数：暂无

9)REST API：

五、floodlightTopologyService

TopologyService（dev）

1)简介：拓扑服务为控制器维护拓扑信息，也会发现网络中的路由。

2)服务提供者：ItopologyService、IRoutingService

3)服务依赖：ILinkDiscoveryService、IThreadPoolService、IFloodlightProviderService、IRestApiService

4)java文件：net.floodlightcontroller.topology.TopologyManager

5)工作原理：该服务从ILinkDiscoveryService中获取链路信息并计算出网络拓扑，在拓扑服务中有一个很重要的概念叫做openflow区域，openflow区域指的是和同一个Floodlight实例相连的一组交换机，openflow区域之间可以用二层的非openflow交换机相连，例如：

[OF switch 1] -- [OF switch 2] -- [traditional L2 switch] -- [OF switch 3]

上述连接将会被看成两个openflow区域，第一个由s1和s2组成，第三个由s3组成。

当前所有的拓扑信息都被存储在一个叫做拓扑实例的不可变数据结构中，拓扑中有任何信息的改变都会建立一个新的拓扑实例，并且发出拓扑改变的消息，如果其它模块需要监听拓扑改变消息，该模块就要实现ITopologyListener接口。

6)局限性：可以在openflow区域内建立一个冗余链路，但是不能建立一个从非openflow交换机到openflow区域的冗余链路。

7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块。

8)配置参数：暂无

9)REST API：

六、floodlight RestApiServer

查看加载端口文件（一步一步跟踪源码看）

/src/main/resources/rentron.properties 配置rest 启动监听端口

Floodlightdefault.properties 启动服务

Services/net.floodlightcontroller.core.module.IFloodLightModule模块

RestApiServer（Dev）

1)简介：RestApiServer通过HTTP协议提供REST API。
2)服务提供者：IRestApiService
3)服务依赖：暂无
4)java文件：net.floodlightcontroller.restserver.RestApiServer
5)工作原理：Rest
API服务使用Restlets库，其它基于REST服务提供API的模块需要添加一个实现RestletRoutable接口的类，每一个RestletRoutable都会包含一个与Restlet资源（通常是ServerResource）相连的路由。用户需要添加一个自定义的类用来处理特定URL的请求，该类需要继承Restlet。使用@GET、@POST注解来选择用于HTTP请求的方法。序列化通过Restlet中包含的Jackson库来完成，Jackson有两种方法序列化对象，一种是自动的使用getter方法序列化那些字段，一种事通过添加注解自定义序列化。
6)局限性：
①基本路径必须唯一，不能重复
②Restlets只能通过服务接口访问模块，如果一个模块需要通过REST服务提供一数据，那么它就必须提供一个获取该数据的接口
7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块
8)配置参数：
port：HTTP协议监听的TCP端口，类型：int，默认值：8080
9)REST API：

七 floodlight ThreadPool（Dev）

ThreadPool（Dev）
1)简介：该模块包装了java中的ScheduledExecutorService，它可以在指定的时间运行一个线程，也可以用来周期性的执行一个线程。
2)服务提供者：IThreadPoolService
3)服务依赖：暂无
4)java文件：net.floodlightcontroller.threadpool.ThreadPool
5)工作原理：参考java文档中的ScheduledExecutorService。
6)局限性：暂无
7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块
8)配置参数：暂无
9)REST API：

八、数据库内存h2,someMemoryStorageSource

MemoryStorageSource（Dev）
1)简介：MemoryStorageSource是一个在内存中的NoSQL存储，提供了数据改变时的通知。
2)服务提供者：IStorageSourceService
3)服务依赖：ICounterStoreService、IRestApiService
4)java文件：net.floodlightcontroller.storage.memory.MemoryStroageSource

5)工作原理：基于这个模块的其它模块可以创建、删除、修改内存中存储的数据资源，所有的数据都是共享的，任何实现了IStorageSourceListener接口的模块可以都监听某个表或者某行数据的改变。

6)局限性：

①因为数据存放在内存里，Floodlight关闭后数据就会丢失。

②数据之间没有隔离，某个模块建立了一个数据表，其它的模块可以改写里边的数据。

7)配置：这个模块默认是启用的，不需要任何的配置就加载使用这个模块

8)配置参数：暂无

9)REST API：暂无

九、Flow Cache（API only）

Flow Cache（API only）

考虑到在网络中需要处理某个范围内不同的事件，就定义了流缓存API，哪些事件需要处理、怎么处理取决于基于floodlight的SDN程序，例如对交换机/链路故障中流的处理是大多数应用程序都需要的。

floodlight定义了流缓存API和一系列骨架方法，应用程序开发人员可以使用这个通用框架针对他们应用的需求来实现解决方案

我们正在努力整理出该API的书面记录，稍后就会发布到Floodlight网站上，同时你可以在流缓存源码中找到调用API的简洁说明。

交换机/链路关闭事件的例子：

为了对流缓存的目的做一个宽泛的说明，我们跟随交换机/链路关闭事件处理的过程来看看各个模块的调用：

①当LinkDiscoveryManager发现一个线路或者端口关闭时，TopologyManager中的“NewInstanceWorker”线程将会处理这个事件，在这个线程的结尾会调用informListeners，informListeners是一个留在这个事件中的钩子，以便让所有对此事件刚兴趣的模块处理该事件。

②这是你的模块就会启动，这个模块必须实现ITopologyListener接口的topologyChanged方法，并且通过TopologyManager.addListener方法将这个模块添加到监听者列表中。

③在你的模块中，可以通过Topology.getLastLinkUpdates方法获取以前所有拓扑的变化，并且可以通过排序找出所需要的事件，可以在相邻交换机中链路的关闭中找出故障交换机，因此你需要要找出ILinkDiscover.updateOperation.LINK_REMOVED事件（每个相关的交换机都有一个事件），根据找到的条目你就可以得到相关的交换机端口。

④下一步是所有受影响的交换机中和相关端口有关系的流，查询是一个OFStatisticsRequest消息，可以用过sw.sendStatsQuery方法将此消息发送给交换机sw。

⑤查询发出去后不久你就会收到一个响应，为了能够接受这个openflow数据包的响应，你的模块就要实现IOFMessageListener接口，并且处理OFType.STATS_REPLY消息。获取到响应后你就可以看到里边所有的流，现在你就可以决定是否创建一个flowmods的删除消息清理到那些流。

看起来问题已经解决了，但是我们还没有使用流缓存以及相关的服务接口。

流缓存的理念是让控制器持有所有有效流的记录。当事件被一个控制器的不同模块监听或者随时查询交换机时，流缓存就会更新，这就让不同模块对流的更新和检索整合到了一个地方。

流缓存的数据结构留给实现者决定，在API中已经展示了查询和响应的格式（对于流缓存的格式），也可以为每一个查询指定自己的处理器。

流协商（reconcile）类用来清理流缓存和交换机中的流，你可以使用多个模块监听不同的事件，它们都要实现IFlowReconcileListener接口和reconcileFlows方法。该方法可以立即产生结果，也可以通过OFMatchReconcile对象把事件传递给其它模块，同时还定义了一些接口用于跟踪正在等待的查询。

十Packet Streamer

Packet Streamer

1)简介：该模块是一个数据包流服务，使用此项服务可以让任何交换机、控制器和它的观察者之间有选择的交换数据。它由两个接口组成：①一个基于REST的接口，该接口定义了它所感兴趣的openflow消息的特征，我们称之为过滤器；②一个基于Thrift的数据包过滤器

2)REST API：

3)基于Thrift的流服务：数据包流服务是基于Thrift的流服务器的代理，下边列出了Thrift接口，完成的Thrift接口可以在/src/main/thrift/packetstreamer.thrift中找到。

Floodlight构建脚本生成了Thrift服务的java和python库，通过在构建脚本setup.sh中添加简单的语言选项就可以支持其它语言。

REST

API中描述了流会话的创建，会话ID创建以后，可以使用Thrift接口中的getPackets（sessionId）方法接收指定会话的数据包，terminateSession（sessionId）方法用来终止一个现有的会话。下边有一个python的例子。

```
rvice PacketStreamer {
 Synchronous method to get packets for a given sessionid
 list<binary> getPackets(1:string sessionid),
 // Synchronous method to publish a packet.
 //It ensure the order that the packets are pushed
 i32 pushMessageSync(1:Message packet),
//Asynchronous method to publish a packet.
 //Order is not guaranteed.
 oneway void pushMessageAsync(1:Message packet)
/Terminate a session
 void terminateSession(1:string sessionid)

y:
Make socket
  transport = TSocket.TSocket('localhost', 9090)
  # Buffering is critical. Raw sockets are very slow
ansport = TTransport.TFramedTransport(transport)
Wrap in a protocol
  protocol = TBinaryProtocol.TBinaryProtocol(transport)
  # Create a client to use the protocol encoder
  client = PacketStreamer.Client(protocol)
  # Connect!
ansport.open()
  while 1:
      packets = client.getPackets(sessionId)
      for packet in packets:
          print "Packet: %s" % packet
          if "FilterTimeout" in packet:
              sys.exit()
```

```
cept Thrift.TException, e:
  print '%s' % (e.message)
  terminateTrace(sessionId)
```

4)客户端示例：当前版本的Floodlight中有一个基于MAC地址的数据包流例子，其中部分的客户端代码已经在前面展示出来了。可以在例子部分找到一个介绍REST API和Thrift客户端使用方法的完成python客户端代码。要确保客户机上安装了Thrift，并且给出示例中gen-py和Thrift python目录的正确路径。

5)如何扩展服务：可以在基于MAC的例子上扩展数据包流服务，net/floodlightcontroller/core/web/PacketTraceResource.java定义了REST接口，可以在FilterParameters类中添加字段来扩展匹配字段，可以在net/floodlightcontroller/core/OFMessageFilterManager.java中实现匹配逻辑，也可以简单的在getMatchedFilters方法中添加匹配逻辑来扩展服务。如果有更多的用例出现，可以使用一个插件框架取代当前的实现。

6)数据包格式：在基于MAC地址的流示例中有一个执行数据包格式化的控制器，数据包的格式化是在net/floodlightcontroller/core/OFMessageFilterManager.java中完成的，在某些应用中，可能使用原始的数据包更好，数据包格式可以作为会话属性的一部分添加到数据包跟踪过滤器中，系统会根据会话属性调用响应的格式化程序。

十一、VirtualNetworkFilter（Quantum Plugin）（Dev）

应用模块包含了VirtualNetworkFilter、Forwarding、Firewall、Port Down Reconciliation模块。

VirtualNetworkFilter（Quantum Plugin）（Dev）

1)简介：VirtualNetworkFilter模块是二层上一个简单的虚拟化网络。你可以使用它在一个二层的域中建立多个逻辑的二层网络，该模块可以独立使用也可以在OpenStack上使用。

2)服务提供者：IVirtualNetworkService

3)服务依赖：IDeviceService、IFloodlightProviderService、IRestApiService

4)java文件：net.floodlightcontroller.virtualnetwork.VirtualNetWorkFilter

5)工作原理：Floodlight启动的时候并不会创建虚拟网络，这就意味着主机之间不能相互通信，虚拟网络创建以后就可以把主机添加到虚拟网络中，该模块会把自己插入到PacketIn消息处理链的前面，一旦受到PacketIn消息，它就会查看消息的源MAC和目的MAC，如果源MAC和目的MAC在同一个虚拟网络中，它就会返回Command.CONTINUE，这个流就会被其它模块继续处理，否则它就返回Command.STOP，这个包就会被丢弃。

6)局限性：

①物理网络必须是二层的

②每个虚拟网络只能有一个网关（多个虚拟网络可以使用同一个网关）

③多播和广播流量不是孤立的

④所有的DHCP流量都可以通过

7)配置：这个模块在默认情况下是不启用的，需要在配置文件中写上配置，然后重启Floodlight才能加载它，下边提供了一个简单的配置，这个模块的名称是VirtualNetworkFilter，默认的配置文件是：src/main/resources/quantum.properties。

```
The default configuration for openstack
oodlight.modules = net.floodlightcontroller.storage.memory.MemoryStorageSource,\
t.floodlightcontroller.staticflowentry.StaticFlowEntryPusher,\
t.floodlightcontroller.forwarding.Forwarding,\
t.floodlightcontroller.jython.JythonDebugInterface,\
t.floodlightcontroller.counter.CounterStore,\
t.floodlightcontroller.perfmon.PktInProcessingTime,\
t.floodlightcontroller.ui.web.StaticWebRoutable,\
t.floodlightcontroller.virtualnetwork.VirtualNetworkFilter
t.floodlightcontroller.restserver.RestApiServer.port = 8080
t.floodlightcontroller.core.FloodlightProvider.openflowport = 6633
t.floodlightcontroller.jython.JythonDebugInterface.port = 6655
```

8)配置参数：暂无

9)REST API：

十二、Forwarding（Dev）

Forwarding（Dev）

1、简介：Forwarding在两个设备之间转发数据包，IDeviceService会把发送设备和接收设备分类。

2、服务提供者：暂无

3、服务依赖：IDeviceService、IFloodlightProviderService、IRestApiService、IRoutingService、ITopologyService、ICounterStoreService

4、java文件：net.floodlightcontroller.forwarding.Forwarding

5、工作原理：由于Floodlight被设计成可以工作在同时包含openflow交换机和非openflow交换机的网络中，Forwarding也就考虑了这种情况，算法会为发送设备和接收设备找出所有包含设备安装点的openflow区域，FlowMods将会沿着最短路径安装流，如果在一个openflow区域上收到一个PacketIn消息，而这个区域内没有那么设备的安装点，这个数据包将会被泛洪。

6、局限性：

①目前不提供路由功能

②没有VLAN的封装和解封

7、配置：该模块默认是启用的，不需要任何配置就能加载

8、配置参数：暂无

9、REST API：暂无

十三、Firewall（Dev）

Firewall（Dev）

1、简介：防火墙应用程序实现了一个floodlight模块，该模块通过检测PacketIN行为使用流在openflow enable交换机上强制执行ACL。ACL是一系列的条件，这些条件允许或者拒绝接入交换机上的流量。

每个数据流中和防火墙规则匹配的第一个数据包都会触发一个packet-in。防火墙规则按照优先级排序，并按照Openflow1.0标准中规定的OFMatch和PacketIn中的头字段进行匹配。优先级最高的匹配将决定处理流的动作（允许/拒绝），也可以使用OFMatch中定义的通配符。

2、防火墙策略：防火墙以反应的方式工作，防火墙规则按照创建时的优先级进行排序（通过它的REST API），每一个到来的PacketIn数据包都会从最高的优先级和列表匹配，直到找出一个匹配或者耗尽列表。如果找到一个匹配，该规则存储在IRoutingDecision对象中的操作就会传递到PacketIn处理管道中的其余部分，这一操作会到达Forwarding或者其它的包转发程序（例如学习型交换机），如果那个操作是ALLOW，那么Forwarding将会执行一个普通的转发，如果是DENY，就会丢弃数据包。无论哪种情况，发送给交换机的流条目都必须明确的反应防火墙的匹配规则。

防火墙允许规则上有部分的重叠，它们将会按照优先级进行裁决，下边是一个简单的例子，所有到达192.168.1.0/24子网的流量都会被拒绝，除了HTTP流量（TCP 80端口）。

protocol	destination IP	destination port	action	priority*
TCP	192.168.1.0/24	80	ALLOW	1
TCP	192.168.1.0/24	wildcard	DENY	2

注意这种情况下通配符的处理，如果一个数据包不能匹配优先第一个级较高的流，而匹配了第二个优先级较低的流，被Forwarding转发到交换机的流条目不会将目的端口作为匹配项；相反，如果数据包的目的端口是流指定的端口，在不向控制器发送PacketIn消息的情况下，以后目的端口是80的数据包将不会被丢弃。

3、REST Interface：

4、Examples using curl：

5、测试方法：测试包含了使用EasyMock创建的自动化单元测试，FirewallTest类可以使用JUnit和Eclipse执行的测试用例。在大部分测试用例中，PacketIn事件都是模拟的，防火墙的决定是按照规则定义来验证的。下边是不同测试用例的列表：

①testNoRules：在没有任何规则的情况下发送一个PacketIn事件，防火墙应该拒绝这些流量，这是一个边界测试用例。

②testRuleInsertionIntoStorage：向防火墙添加一个规则，通过检查存储来验证它，这是一个积极测试用例。

③testRuleDeletion：从防火墙中删除一条规则，通过检查存储器来验证它，这也是一个积极测试用例。

④testFirewallDisabled：在防火墙禁用并且插入一个规则的情况下发送一个PacketIn事件，防火墙应该拒绝这些流量，这是一个消极测试用例。

⑤testSimpleAllowRule：添加一调规则，允许一个IP到另一个IP之间的TCP通信，并发送一个packetIn事件，在校验防火墙的决定之后再发送一个应该被丢弃的数据包，然后校验这个数据包的丢弃，这个测试覆盖了普通规则（不是边界测试，因为没有畸形包和广播包）。

⑥testOverlappingRules：添加一些重叠的规则（拒绝所有的TCP流量除了80端口），这个测试覆盖了多个规则的复杂情况（多个allow-deny的重叠规则）。

⑦testARP：测试ARP的广播请求和单播响应，因为没有允许ARP响应包的规则，所有只有请求的广播包可以通过。

⑧testIPBroadcast：在没有任何规则的情况下发送一个三层的IP广播包Packetin事件，防火墙应该允许这些流量，这是一个积极测试方法，覆盖了指定的IP广播方案（二层+三层的广播）。

⑨testMalformedIPBroadcast：在没有任何规则的情况下发送可以畸形的IP广播packetIn事件，防火墙应该拒绝这些流量，因为这个包的二层是广播但是三层是单播，这是一个边界测试。

⑩testLayer2Rule：有一条规则允许两个指定MAC地址之间的通信，还有一个优先级较低的规则拒绝所有的TCP流量，模拟一个TCP PacketIn事件，防火墙应该允许这些流量，这是一个消极测试，覆盖了只包含二层规则的一个子集。

6、问题和局限性：

①调用防火墙模块DELETE REST API不能删除交换机中的流，规则只能在交换机流条目到它的过期时间时才能删除，这就意味着删除一个规则只能在一段时间之后才能起效，主要有连续的流量通过交换机现有的流就会一直存在。

②在最初的提议中，防火墙规则支持TCP/UDP端口范围，但是openflow流匹配机制不允许指定端口范围，所有这个特性没有实现。

十四、Port Down Reconciliation（Dev）

这一节翻译的超烂，建议直接看原文

Port Down Reconciliation（Dev）

1、简介：PortDownReconciliation模块的实现是为了在端口关闭的时候处理网络中的流。在PORT_DOWN链路发现更新消息之后，这个模块将会找出并且删除直接指向这个端口的流。所有无效的流都被删除之后，floodlight应该重新评估链路，流量也要采用新的拓扑。如果没有这个模块，流量会持续的进入到这个坏掉的端口，因为流的过期时间还没有到。

2、工作原理：想象一下我们有一个拓扑，包含了交换机s1，s1连接了一个主机h1,和两个交换机s2、s3，现在流量从s2、s3进入到s1，目标地址是h1，但是到h1的链路关闭了，s1将会给controller发送一个PORT_DOWN通知。

在收到PORT_DOWN链路更新之后，该模块就会找到那个关闭的端口，然后向s1查询所有目的端口是链路发现更新描述的端口的流，它会分析这些流，并且为进入端口和把流路由到已关闭端口的OFMatch创建索引。

索引可能看起来像这样：

Short ingressPort	List<OFMatch> (Invalid match objects of flows directing traffic to down port)
1	[match1:dataLayerDestination: "7a:59:cd:34:a7:9b", dataLayerSource: "c6:cb:ad:80:49:70"]
2	[match2:dataLayerDestination: "7a:59:cd:34:a7:9b", dataLayerSource: "c6:cb:ad:80:49:69"]

跟随s1的索引，该模块向拓扑服务询问网络中所有交换机的连接，以此追溯链路。找出所有的交换机，这些交换机包含了目标交换机对应s1，目标端口对应索引中描述的无效进入端口，如果找到这样的匹配，那么这个交换机就会被添加到相邻交换机索引中，此时指向源端口的连接和无效的OFMatch。

相邻交换机索引：

IOFSwitch sw	Short outPort (outPort to link connected to the base switch)	List<OFMatch> (Invalid match objects of flows directing traffic towards the base switch)
sw2	3	["match1:"dataLayerDestination: "7a:59:cd:34:a7:9b", dataLayerSource: "c6:cb:ad:80:49:70"]
sw3	3	["match2:"dataLayerDestination: "7a:59:cd:34:a7:9b", dataLayerSource: "c6:cb:ad:80:49:69"]

现在s1就不需要这些信息了，所有目标端口是故障端口的流都将会从s1中删除，然后该模块遍历和s1相邻交换机的索引，并在上边执行相同的操作，这个过程会逐跳的递归进行，直到网络中没有无效的流。

3、问题和局限性：如果在一个源地址和目的地址的路由中有重叠的交换机，那些重叠的交换机将会因不同的流被统计多次，这就花费了额外的时间，但是对于维护网络中流的完整性，这也是必要的。

十五、Module Loading System（一）

一、简介：Floodlight使用土生土长的模块系统来决定哪些模块会运行，这个系统的设计目标是：
1）通过修改配置文件决定哪些模块会被加载
2）实现一个模块不需要修改他所依赖的模块
3）创建一个定义良好的平台和API以扩展Floodlight
4）对代码进行强制的模块化

二、主要部件：模块系统包含几个主要的部件：模块加载器、模块、服务、配置文件和一个在jar文件中包含了了可用模块列表的文件。

1）模块：模块被定以为一个实现了IFloodlightModule接口的类。IFloodlightModule接口的定义如批 注所示。

```
*
 Defines an interface for loadable Floodlight modules.

 At a high level, these functions are called in the following order:
 <ol>
 <li> getServices() : what services does this module provide
 <li> getDependencies() : list the dependencies
 <li> init() : internal initializations (don't touch other modules)
 <li> startUp() : external initializations (<em>do</em> touch other modules)
 </ol>

 @author alexreimers
/
blic interface IFloodlightModule {


  /**
   * Return the list of interfaces that this module implements.
   * All interfaces must inherit IFloodlightService
   * @return
   */


  public Collection<Class<? extends IFloodlightService>> getModuleServices();


  /**
   * Instantiate (as needed) and return objects that implement each
   * of the services exported by this module.  The map returned maps
   * the implemented service to the object.  The object could be the
   * same object or different objects for different exported services.
   * @return The map from service interface class to service implementation
   */
  public Map<Class<? extends IFloodlightService>,
             IFloodlightService> getServiceImpls();


  /**
   * Get a list of Modules that this module depends on.  The module system
   * will ensure that each these dependencies is resolved before the
   * subsequent calls to init().
   * @return The Collection of IFloodlightServices that this module depnds
```

```
     *          on.
     */


    public Collection<Class<? extends IFloodlightService>> getModuleDependencies();


    /**
     * This is a hook for each module to do its <em>internal</em> initialization,
     * e.g., call setService(context.getService("Service"))
     *
     * All module dependencies are resolved when this is called, but not every module
     * is initialized.
     *
     * @param context
     * @throws FloodlightModuleException
     */


    void init(FloodlightModuleContext context) throws FloodlightModuleException;


    /**
     * This is a hook for each module to do its <em>external</em> initializations,
     * e.g., register for callbacks or query for state in other modules
     *
     * It is expected that this function will not block and that modules that want
     * non-event driven CPU will spawn their own threads.
     *
     * @param context
     */


    void startUp(FloodlightModuleContext context);
```

2）服务：一个模块可能包含一个或多个服务，服务被定义为一个继承IFloodlightService接口的接口。

```
*
 This is the base interface for any IFloodlightModule package that provides
 a service.
 @author alexreimers
/
blic abstract interface IFloodlightService {
  // This space is intentionally left blank....don't touch it
```

现在这是一个空接口，在我们的加载系统中它是用来强制保证类型安全的。
3）配置文件：配置文件明确的规定了哪些模块可以加载，它的格式是标准的java属性，使用键值对，在模块列表中键是floodlight.modules，值是以逗号分隔的模块列表，可以在一行，或者使用\
断行，下边是Floodlight默认的配置文件：

```
oodlight.modules = net.floodlightcontroller.staticflowentry.StaticFlowEntryPusher,\
t.floodlightcontroller.forwarding.Forwarding,\
t.floodlightcontroller.jython.JythonDebugInterface
```

有许多没有写在这个列表里的模块也被加载了，这是因为模块系统自动加载了依赖，如果一个模块没有提供任何服务，那么就必须在这里明确的定义。
4）我们使用java的ServiceLoader找到类路径中的模块，这就要求我们列出文件中所有的类，这个文件的格式是在每一行都有一个完整的类名，这个文件在src/main/resource/MEAT-INFO/service/net.floodlightcontroller.module.IFloodModule。你使用的每个jar文件（如果使用多个jar文件就接着看）都要有它自己的META-INFO/services/net.floodlightcontroller.module.IFloodlightModule文件，列出实现了IFloodlightModule接口的类。下边是一个示例文件：

```
t.floodlightcontroller.core.CoreModule
t.floodlightcontroller.storage.memory.MemoryStorageSource
t.floodlightcontroller.devicemanager.internal.DeviceManagerImpl
t.floodlightcontroller.topology.internal.TopologyImpl
t.floodlightcontroller.routing.dijkstra.RoutingImpl
t.floodlightcontroller.forwarding.Forwarding
t.floodlightcontroller.core.OFMessageFilterManager
t.floodlightcontroller.staticflowentry.StaticFlowEntryPusher
t.floodlightcontroller.perfmon.PktInProcessingTime
t.floodlightcontroller.restserver.RestApiServer
t.floodlightcontroller.learningswitch.LearningSwitch
t.floodlightcontroller.hub.Hub
t.floodlightcontroller.jython.JythonDebugInterface
```

十六、Module Loading System（二）

三、启动顺序：
1）模块发现：所有在类路径中的模块（实现IFloodlightModule的类）都会被找到，并且建立三个映射（map）：
①服务映射：建立服务和提供该服务的模块之间的映射
②模块服务映射：建立模块和他提供的所有服务之间的映射
③模块名称映射：建立模块类和模块类名之间的映射
2）找出需要加载的最小集合：使用深度优先遍历算法找出需要加载模块的最小集合，所有在配置文件中定义的模块都会添加到队列中，每个模块出对后都会被添加到模块启动列表中，如果一个模块的依赖还没有添加到模块启动列表中，将会在该模块上调用getModuleDependenceies方法。在这里有两种情况可能引起FloodlightModuleException异常，第一种情况找不到在配置文件中定义的模块或者模块的依赖，第二种情况是两个模块提供了相同的服务，却没有指明使用哪个。
3）初始化模块：集合中的模块会迭代的加载，并且在上边调用init方法，现在模块会做两件事情

①在FloodlightModuleContext上调用getServiceImpl方法把它的依赖写到一起。

②对自己内部的数据结构执行初始化。

init方法调用的顺序是不确定的。

4）启动模块：在每个模块上调用init方法后，就会调用startUp方法，在这个方法中模块将会调用它所依赖的模块，例如：使用IStorageSourceService模块在数据库中建立一个表、或者用IFloodlightProviderService的executor服务建立一个线程。

四、通过命令行使用控制器：

1）只使用floodlight.jar：如果你只是想使用默认配置运行floodlight，最简单的方法就是运行这个命令

```
1   java -jar floodlight.jar
```

2）使用多个jar文件：也可以使用多个jar文件运行openflow，如果你想用另外的包分发，这将会非常有用，只是命令有些不同。

```
1   java -cp floodlight.jar:/path/to/other.jar net.floodlightcontroller.core.Main
-
```

cp参数告诉java使用类路径中的那些jar文件，main方法坐在的类由net.floodlightconntroller.core.Main指定。如果你添加的jar文件包含了实现IFloodlightModule接口的类，你就要确保创建了MAIN-INF/services/net.floodlightcontroller.core.module.IFloodlightModule。

3）指定其它的配置文件：使用这两种方法 你可以指定一个其它的配置文件，这需要用到-cf选项。

```
1   java -cp floodlight.jar:/path/to/other.jar net.floodlightcontroller.core.Main -cf path/to/
-
```

cf参数必须放到所有选项的后边，这就让参数传递到了java程序而不是java虚拟机。使用哪个配置文件的顺序是：使用-cf选项指定的配置文件、config/floodlight.properties文件（如果存在的话）、在jar文件中的floodlightdefault.properties文件（在src/main/resources中）。

十七、Module Loading System（三）

五、每个模块的配置选项：配置文件也可以指定每个模块的配置选项，参数的格式是：<fully qualified module name>.<config option name> = <value>，我们使用完整的类名，这样就可以为每个类指定它自己的配置。

例如我们想要为REST API指定一个端口，可以在配置文件中加入以下配置：

```
1   net.floodlightcontroller.restserver.RestApiServer.port = 8080
```

在RestApiServer.java的init方法中解析这个选项：

```
1   // read our config options
2   Map<String, String> configOptions = context.getConfigParams(this);
3   String port = configOptions.get("port");
4   if (port != null) {
5           restPort = Integer.parseInt(port);
6   }
```

这不需要任何检查，如果没有提供配置选项，FloodlightModuleLoader不会把它加入到环境中。

也可以使用java属性通过命令行的方式指定选项。这将会覆盖任何floodlight配置文件中指定的选项

```
1    java -Dnet.floodlightcontroller.restserver.RestApiServer.port=8080 -jar floodlight.jar
```

有两点要注意：第一点是所有的java属性必须在-jar floodlight.jar之前指定，在它后边的所有选项都会作为命令行参数传递给java程序。第二点是在-D选项中没有空格。

六、说明：

①为了处理循环的依赖，init方法和startUp方法的调用顺序是不确定的

②你的配置文件不能被floodlightdefault.properties调用，它默认的包含在jar文件中。

③每个模块都要有一个无参构造函数（最好是空的），需要在构造函数中完成的任务应该放到init方法中。

④多个模块不应该有服务的重叠，但是可以有功能的重叠。例如LearningSwitch和Forwarding都是转发PacketIn数据包的响应，由于他们并没有提供相同的服务，我们并没有检测到重叠。

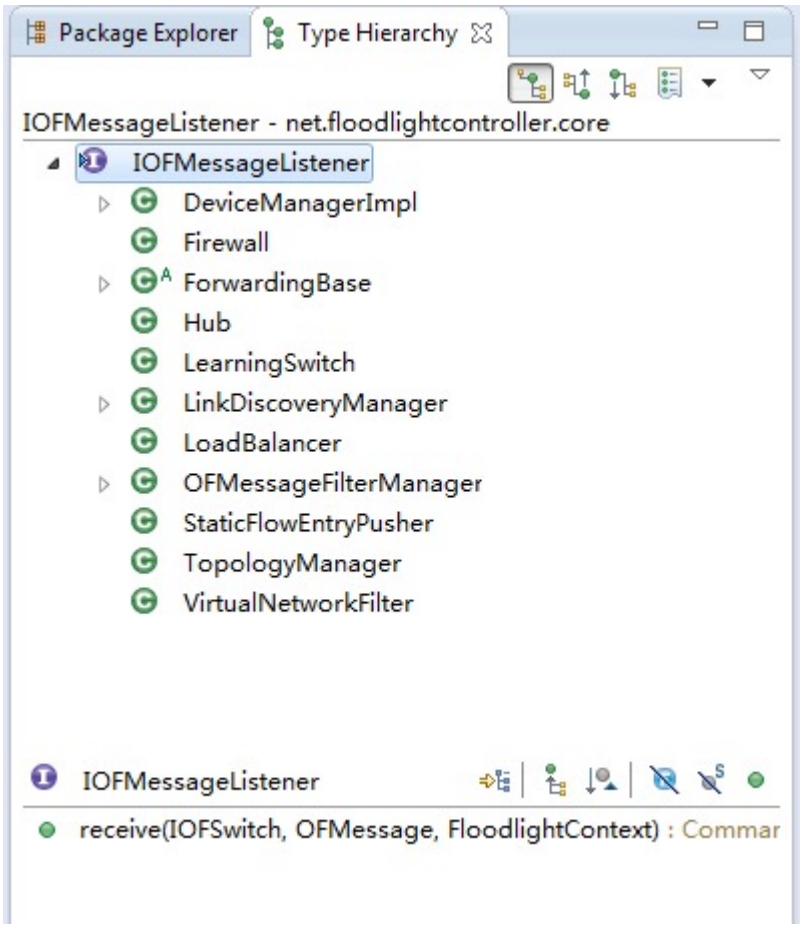
到此关于floodlight的内容就算完了，关于测试、REST API、QoS等其它的东西就不发了。

这里是作者2013.12月所写，时隔一年，

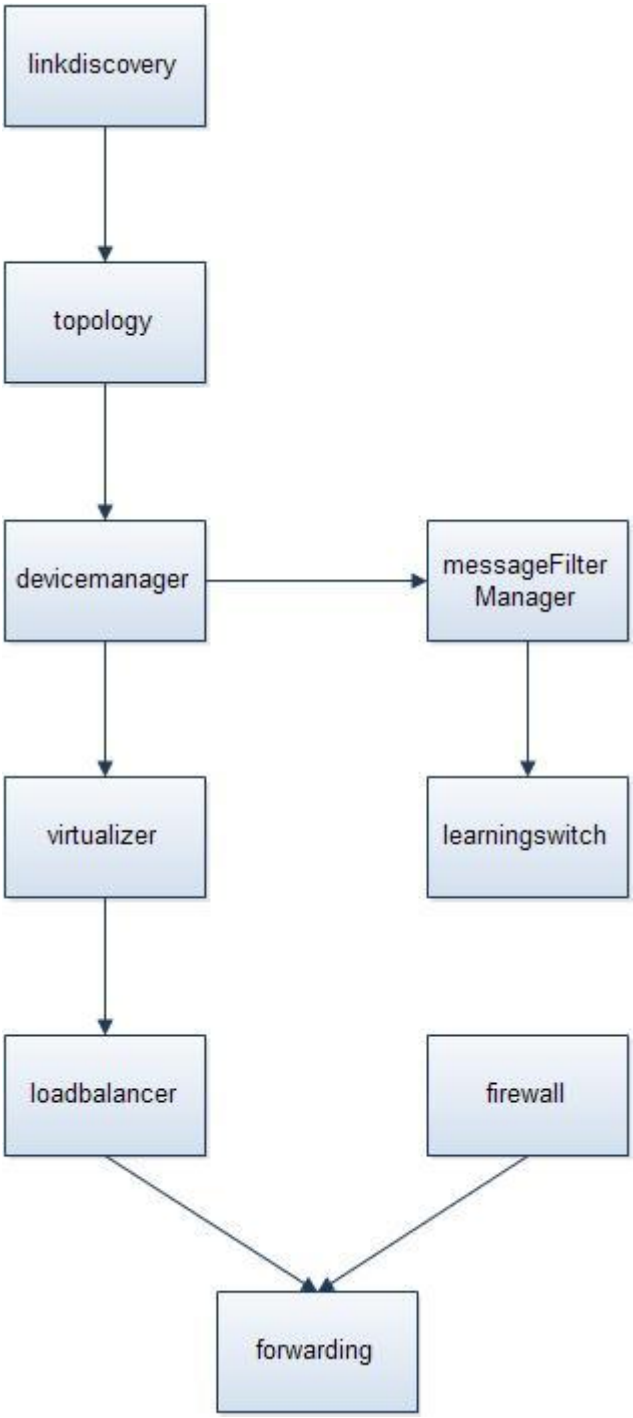
十八、Floodlight各模块处理PacketIn消息的顺序

在OpenFlow网络中PacketIn消息是交换机向控制器发送最多的消息，也是控制器最需要关注的消息，没有之一。在Floodlight有很多模块都会处理PacketIn消息，他们之间是以什么顺序执行的是一个很重要的问题。

在Floodlight中所有监听OpenFlow消息的模块都需要实现IOFMessageListener接口，在Eclipse里打开IOFMessageListener，Type Hierarchy窗口可以看到所有监听OpenFlow消息的类，如下：



它们之间执行的先后顺序如下图：



到这里已经把文章标题所提出的问题解决了，如果你只是想知道Floodlight各模块之间处理PacketIn的先后顺序就可以点右上角的叉号了。如果想知道的多点，请继续往下看。

如果PacketIn消息并没有按照上图的顺序进行处理，交换他们之间的处理顺序会发生什么情况呢，那就假设forwarding模块在virtualizer之前执行对PacketIn消息的处理，当某个交换机向控制器发送了一个PacketIn消息之后，Floodlight将按照以下过程处理该消息：

1） PacketIn消息先交给forwarding，forwarding模块会决定如何转发该数据包，并向OpenFlow网络中的各个交换机添加响应的流，以保证数据包可以准确的到达目的地。

2） PacketIn消息又交给virtualizer，virtualizer模块会看看源mac和目的mac是否在同一个virtual network，然后决定时候转发该数据包。但是已经没有任何意义了，因为就算virtualizer发现源mac和目的mac不在同一个virtual network，不应该转发该数据包，可是forwarding向交换机下发过转发命令了。

由此可见，各个模块之间的顺序并不能随意打乱，所以Floodlight必须以某种方式保证每个模块的执行顺序。现在就开始介绍Floodlight是如何决定模块执行顺序的。

我们已经知道每个处理OpenFlow消息的模块都要实现IOFMessageListener接口，而IOFMessageListener接口又继承了IListener接口，所以每个监听OpenFlow消息的模块都必须实现以下四个方法：

```
public String getName();

public boolean isCallbackOrderingPrereq(T type, String name);

public boolean isCallbackOrderingPostreq(T type, String name);

public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx);
```

getName方法返回该模块的名字，在决定模块执行顺序的时候，会用到模块名字。

receive方法是用来处理OpenFlow消息的，一个模块注册监听OpenFlow消息后，如果有OpenFlow消息到来，Floodlight就会调用该模块的receive方法。

isCallbackOrderingPrereq方法和isCallbackOrderingPostreq方法是用来控制模块调用顺序的，这两个方法的参数都是T type和String name。其中type是泛型参数，在IOFMessageListener中已被声明为OFType，也就是使用getName方法得到的字符串，用来标示一个消息监听模块。如果将模块A的名称传给模块B的isCallbackOrderingPrereq方法，且该方法返回True，那就表示模块A要在模块B之前执行，如果返回False那就表示模块A不在乎自己是否在模块B之前执行；同样，如果将模块A的名称传给模块B的isCallbackOrderingPostreq方法，且该方法返回Tru

e，那就表示模块A要在模块B之后执行，如果返回False那就表示模块A不在乎自己是否在模块B之后执行。在有些模块中这两个方法都返回False，那就表示这个模块不在乎自己在PacketIn处理序列中的顺序，比如linkdiscovery模块，但这并不能表明该模块可以处于PacketIn处理序列中的任何位置，例如在topology模块中就明确指明linkdiscovery要在topology模块之前执行。所以必须遍历所有实现了IOFMessageListener的类才能确定各个模块的执行顺序。

遍历完所有实现了IOFMessageListener的类之后，Floodlight需要两个步骤来确定所有模块的执行顺序：

1）寻找Terminal module（不知道如何翻译才合适）

Terminal module是指没有任何模块在需要在其后面执行，此类模块并不执行顺序的限制，需要注意的是Terminal module不止一个。伪代码如下：

```
for(int i=0; i<modules.size; i++){
  isTerminal = true;
  for(int j=0;j<modules.size;j++){
    if( modules[j] is after modules[i]){
      isTerminal = false;
      break;
    }
  }
  if(isTerminal)
  terminalQueue.add(modules[i]);
}
```

只要任何一个模块需要在modules[i]之后执行，modules[i]就不是最后执行的模块。

2）在每个最后执行的模块上执行DFS算法以建立执行顺序，伪代码如下：

```
function dfs(){
  for(int i=0; i<terminalQueue.size(); i++){
    visit(terminalQueue[i]);
  }
}


function visit(listener){
  if(!visted.contain(listener){
    visted.add(listener)
```

```
11      for(int i=0;i<modules.size();i++){
12        if( modules[i] is before listener)
13          visit(modules[i])
14      }
15      orderingQueue.add(listener);
16    }
  }
```

到这里就讲完了，如果要为Floodlight添加模块，就应该先想清楚自己的模块应该在什么时候运行，然后再实现isCallbackOrderingPrereq方法和isCallbackOrderingPostreq即可。

以下为一些本文的一些说明

1）你可能注意到这个图中没有Hub和StaticFlowEntryPusher模块；原因是Hub并不关心自己在PacketIn处理序列中的顺序，而且Hub默认没有启动，默认没有启动的模块还有learningswitch。没有StaticFlowEntryPusher的原因是该模块不但不关心自己的执行顺序，而且它也并不处理PacketIn消息。

2）模块名称和类名对照表

模块名称	对应的类 net.floodlightcontroller.
linkdiscovery	linkdiscovery.internal.LinkDiscoveryManager
topology	topology.TopologyManager
devicemanager	devicemanager.internal.DeviceManagerImpl
virtualizer	virtualnetwork.VirtualNetworkFilter
loadbalancer	loadbalancer.LoadBalancer
forwarding	routing.ForwardingBase
messageFilterManager	core.OFMessageFilterManager
learningswitch	learningswitch.LearningSwitch
firewall	firewall.Firewall
staticflowentry	staticflowentry.StaticFlowEntryPusher
net.floodlightcontroller.hub	hub.Hub

3）Floodlight所指定的模块执行顺序表

上边的图是省略了某些不需要的箭头，而简化后的结果，Floodlight的具体实现可在下表中看到（忽略了消息类型参数）。

linkdiscovery	无	无
topology	linkdiscovery	无
devicemanager	topology	无
virtualizer	linkdiscovery devicemanager	forwarding
loadbalancer	topology devicemanager virtualizer	forwarding
forwarding	topology devicemanager	无
messageFilterManager	devicemanager	learningswitch
LearningSwitch	无	无
firewall	无	forwarding
staticflowentry	无	无
net.floodlightcontroller.hub	无	无

十九、Floodlight的forwarding模块流程简单分析

(SDNAP播报)

主模块在初始化并启动各个二级模块后，通过Netty网络应用框架监听6633端口，当有packet-in进来的时候，会调用各个二级模块的receive函数（java中似乎喜欢称之为方法，那下面就都用”方法”来表述吧），此处暂不详表。

对于Forward模块很轻松可以定位到Forwarding.java文件，但是这个文件中或者说Forwarding类中并没有定义receive方法，其实这里可以通过两个途径找到receive函数：1、Forwarding类中首先定义了一个processPacketInMessage方法，见名知义，这个方法是负责处理packet-in消息的，那么通过eclipse中的”open call hierarchy”直接定位到调用该方法的方法，发现是ForwardingBase.java文件中的receive方法；2、Farwording类在声明时继承了ForwardingBase类，可以直接即继承了ForwardingBase类的receive方法。

为了方便描述，以下分析只分析我们关心的流程：Forwarding模块如何向各个交换机添加flow。
通过以上描述，我们找到了Forwarding模块处理packet-in的入口receive方法，该方法中调用了processPacketInMessage方法，该方法又调用doForwardFlow方法。

下面描述doForwardFlow方法的实现细节：

首先声明一个OFMatch对象，并将packet-in中的相关信息加载到该对象中：

```
OFMatch match = new OFMatch(); match.loadFromPacket(pi.getPacketData(), pi.getInPort());
```

然后通过packet-in消息获取目标和源设备（idevice），即以下代码：

```
IDevice dstDevice =IDeviceService.fcStore.get(cntx, IDeviceService.CONTEXT_DST_DEVICE);
if (dstDevice != null) {
IDevice srcDevice =
IDeviceService.fcStore.
get(cntx, IDeviceService.CONTEXT_SRC_DEVICE);
```

再然后定位发生packet-in消息的switch所属于的island的标志

```
Long srcIsland = topology.getL2DomainId(sw.getId());
```

接下来，判断目标和源设备是否处于同一个island，是则继续，否则执行doFlood方法（类似广播），代码如下：

```
 boolean on_same_island = false;
 boolean on_same_if = false;
 for (SwitchPort dstDap : dstDevice.getAttachmentPoints()) {
 long dstSwDpid = dstDap.getSwitchDPID();
 Long dstIsland = topology.getL2DomainId(dstSwDpid);
 if ((dstIsland != null) && dstIsland.equals(srcIsland)) {
 on_same_island = true;
 if ((sw.getId() == dstSwDpid) &&
 (pi.getInPort() == dstDap.getPort())) {
 on_same_if = true;
 }
 break;
 }
 }
 if (!on_same_island) {
 // Flood since we don’t know the dst device
 if (log.isTraceEnabled()) {
 log.trace(“No first hop island found for destination ” +
 “device {}, Action = flooding”, dstDevice);
 }
 doFlood(sw, pi, cntx);
 return;
 }
```

如果在同一个island中，则通过getAttachmentPoints方法获取目标和源设备相连的交换机的端口信息

```
SwitchPort[] srcDaps = srcDevice.getAttachmentPoints();
Arrays.sort(srcDaps, clusterIdComparator);
SwitchPort[] dstDaps = dstDevice.getAttachmentPoints();
Arrays.sort(dstDaps, clusterIdComparator);
```

获取到的结果是两个排序的端口数组，数组的元素类似：

```
SwitchPort [switchDPID=7, port=2, errorStatus=null]
```

再接下来进入一个while循环，循环的终止条件是取完上一步得到的两个数组中的元素，目标是找到属于同一个island的分布在两个数组中的元素。循环内部先通过getL2DomainId方法判断，所选取的与目标和源相连接的交换机是否在同一个island，如果不在，则根据两个island标志的大小选择性的选取srcDaps或dstDaps中的其它元素，继续比较，之所以可以这么做，是因为两个数组是经过排序的，而且key就是各个交换机所属的island标志，代码为while循环最后的if-else语句：

```
} else if (srcVsDest < 0) {
iSrcDaps++;
} else {
iDstDaps++;
}
```

如果找到两个island相同的元素，会调用routingEngine的getRoute方法获取两个端口之间的路由，这才是我们真正关心的流程。这里并未继续跟踪getRoute是如何获取两个交换机端口之间的最短路径（官网提到获取的路由是最短路径），其获取的结果类似：

```
[[id=00:00:00:00:00:00:00:07, port=2], [id=00:00:00:00:00:00:00:07, port=3],
[id=00:00:00:00:00:00:00:05, port=2], [id=00:00:00:00:00:00:00:05, port=3],
[id=00:00:00:00:00:00:00:01, port=2], [id=00:00:00:00:00:00:00:01, port=1],
[id=00:00:00:00:00:00:00:02, port=3], [id=00:00:00:00:00:00:00:02, port=1],
[id=00:00:00:00:00:00:00:03, port=3], [id=00:00:00:00:00:00:00:03, port=1]]
```

接下来利用最初生成的OFMatch对象信息定义一个规则：

```
wildcard_hints = ((Integer) sw.getAttribute(IOFSwitch.PROP_FASTWILDCARDS))
.intValue()
& ~OFMatch.OFPFW_IN_PORT
& ~OFMatch.OFPFW_DL_VLAN
& ~OFMatch.OFPFW_DL_SRC
& ~OFMatch.OFPFW_DL_DST
& ~OFMatch.OFPFW_NW_SRC_MASK
& ~OFMatch.OFPFW_NW_DST_MASK;
```

最后调用pushRoute方法，下发路由策略，即flow信息。

```
pushRoute(route, match, wildcard_hints, pi, sw.getId(), cookie,
cntx, requestFlowRemovedNotifn, false,
OFFlowMod.OFPFC_ADD);
```

继续追踪pushRoute函数，其在ForwardingBase.java中实现，其实就是循环上面route获取到的最短路径，在每个交换机上添加一条flow，包含inport和outport，当然上面提到的wildcard_hints也会通过setMatch方法设置到没条flow中。

以上过程建立了从07:2（[id=00:00:00:00:00:00:00:07,
port=2]）到03:1的一条单向通路，从03:1到07:2的通路又是一个相同的过程。
再回到Forwarding.java文件，PushRoute结束之后，while循环会同时增加两个数组的元素下标，继续进行比较判断，这个地方我被迷惑掉了，既然已经找到一条通路，并下发了流表，那接下来的比较意义是什么？找到一个比

之前的路径有优的路径？但是流表已经下发了，找到了又会发生什么，而且前面的路径应该是没有记录的，怎么判断更优呢？很奇怪为什么在pushRoute之后没有break掉while循环，邮件开发论坛。

开发论坛回复：the purpose in incrementing those variables is to support forwarding in multi island toplogies. Each island in this case gets a separate route pushed.

无法想象多island的拓扑结构o(╯□╰)o，按照回复所说的，在doForwardFlow函数的第二步

```
1  IDevice dstDevice =
2  IDeviceService.fcStore.
3  get(cntx, IDeviceService.CONTEXT_DST_DEVICE);
4
5
6  if (dstDevice != null) {
7  IDevice srcDevice =
8  IDeviceService.fcStore.
   get(cntx, IDeviceService.CONTEXT_SRC_DEVICE);
```

获取到的结果

```
1  Device [deviceKey=5, entityClass=DefaultEntityClass, MAC=00:00:00:00:00:01, IPs=[10.0.0.1
```

其中APs应该是AttachmentPoints的缩写，如果是多island拓扑结构，这里APs应该是由多个SwitchPort构成的数组！但是想象不出一个设备（一块网卡）如何连接到两个交换机上（猜测桥接）。

无论如何，针对单island的情况上述流程分析是行的通的。

二十、使用open vswitch构建虚拟网络

一、open vswitch简介

Open vSwitch是一个高质量的、多层虚拟交换机，使用开源Apache2.0许可协议，由

Nicira
Networks开发，主要实现代码为可移植的C代码。它的目的是让大规模网络自动化可以通过编程扩展,同时仍然支持标准的管理接口和协议（例如NetFlow, sFlow, SPAN, RSPAN, CLI, LACP,
802.1ag）。此外,它被设计位支持跨越多个物理服务器的分布式环境，类似于VMware的vNetwork分布式vswitch或Cisco Nexus 1000 V。Open vSwitch支持多种linux 虚拟化技术，包括Xen/XenServer，
KVM和VirtualBox。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-brcompatd 让ovs-vswitch替换Linuxbridge，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 ：OpenFlow交换机创建和管理公钥框架；

ovs-tcpundump：tcpdump的补丁，解析OpenFlow的消息；

内核模块实现了多个“数据路径”（类似于网桥），每个都可以有多个“vports”（类似于桥内的端口）。每个数据路径也通过关联一下流表（flow
table）来设置操作，而这些流表中的流都是用户空间在报文头和元数据的基础上映射的关键信息，一般的操作都是将数据包转发到另一个vport。当一个数据包到达一个vport，内核模块所做的处理是提取其流的关键信息并在流表中查找这些关键信息。当有一个匹配的流时它执行对应的操作。如果没有匹配，它会将数据包送到用户空间的处理队列中（作为处理的一部分，用户空间可能会设置一个流用于以后碰到相同类型的数据包可以在内核中执行操作）。

二、open vswitch常用操作

以下操作都需要root权限运行，在所有命令中br0表示网桥名称，eth0为网卡名称。

添加网桥：

```
1    #ovs-vsctl add-br br0
```

列出open vswitch中的所有网桥：

```
1    #ovs-vsctl list-br
```

判断网桥是否存在

```
1    #ovs-vsctl br-exists br0
```

将物理网卡挂接到网桥：

```
1    #ovs-vsctl add-port br0 eth0
```

列出网桥中的所有端口：

```
1    #ovs-vsctl list-ports br0
```

列出所有挂接到网卡的网桥：

```
1    #ovs-vsctl port-to-br eth0
```

查看open vswitch的网络状态：

```
1    #ovs-vsctl show
```

删除网桥上已经挂接的网口：

```
#vs-vsctl del-port br0 eth0
```

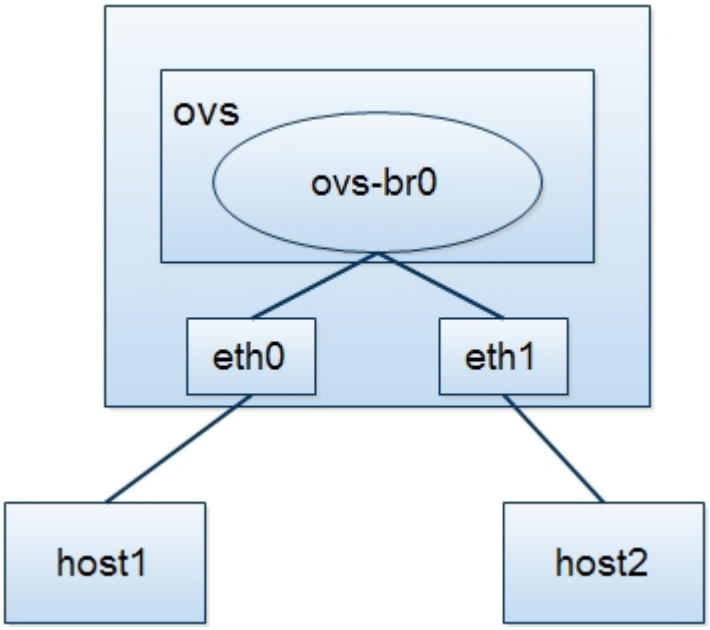
删除网桥：

```
#ovs-vsctl del-br br0
```

三、使用open vswitch构建虚拟网络

1、构建物理机和物理机相互连接的网络

在安装open vswitch的主机上有两块网卡，分别为eth0、eth1，把这两块网卡挂接到open vswitch的网桥上，然后有两台物理机host1、host2分别连接到eth0和eth1上，实现这两台物理机的通信。构建结果图如下：

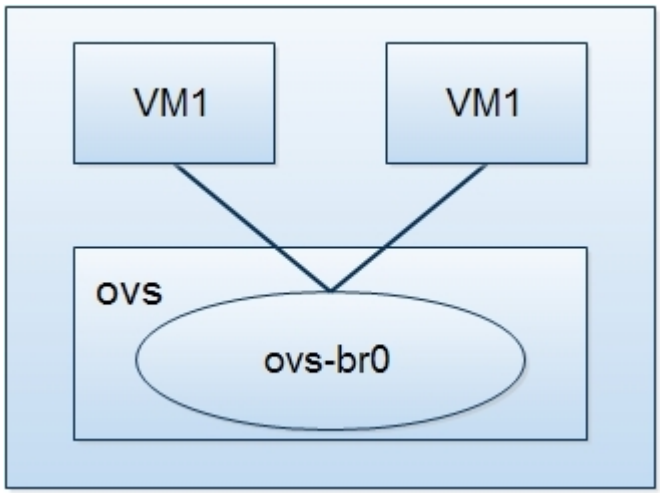


执行以下命令：

```
#ovs-vsctl add-br br0 //建立一个名为br0的open vswitch网桥
1
2
3   #ovs-vsctl add-port br0 eth0 //把eth0挂接到br0中
4
5
#ovs-vsctl add-port br0 eth1 //把eth1挂接到br0中
```

2、构建虚拟机与虚拟机相连的网络

在安装open vswitch的主机上安装两个虚拟机，把两个虚拟机的网卡都挂接在open vswitch的网桥上，实现两台虚拟机的通信，构建结果图如下：



执行以下命令：

ovs-vsctl add-br br0 //建立一个名为br0的open vswitch网桥

如果使用vbox或virt-manager把bridge设置为br0即可，如果使用cli kvm则先创建两个文件，用于虚拟网卡的添加于删除。假设这两个文件分别为/etc/ovs-ifup和/etc/ovs-ifdown，则向这两个文件中写入以下内容

/etc/ovs-ifup

```
#!/bin/sh


switch='br0'

/sbin/ifconfig $1 0.0.0.0 up

ovs-vsctl add-port ${switch} $1
```

/etc/ovs-ifdown

```
#!/bin/sh
```

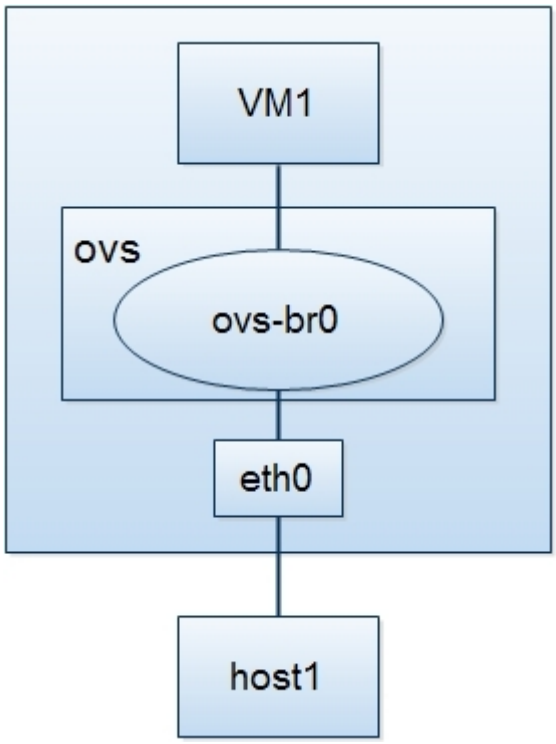
```
4
5
6
7
8   switch='br0'
9
    /sbin/ifconfig $1 0.0.0.0 down

    ovs-vsctl del-port ${switch} $1
```

使用以下命令建立虚拟机

```
    kvm -m 512 -net nic,macaddr=00:11:22:33:44:55-net \
1
2     tap,script=/etc/ovs-ifup,downscript=/etc/ovs-ifdown-drive \
3
4
5     file=/path/to/disk-image,boot=on
6
7
8     kvm -m 512 -net nic,macaddr=11:22:33:44:55:66-net \
9
10
11    tap,script=/etc/ovs-ifup,downscript=/etc/ovs-ifdown-drive \

      file=/path/to/disk-image,boot=on
```

3、构建虚拟机与物理机相连的网络

在装有open vswitch的主机上有一个物理网卡eth0，一台主机通过网线和eth0相连，在open vswitch的主机上还装有一台虚拟机，把此虚拟机和连接到eth0的主机挂接到同一个网桥上，实现两者之间的通信，构建结果图如下：

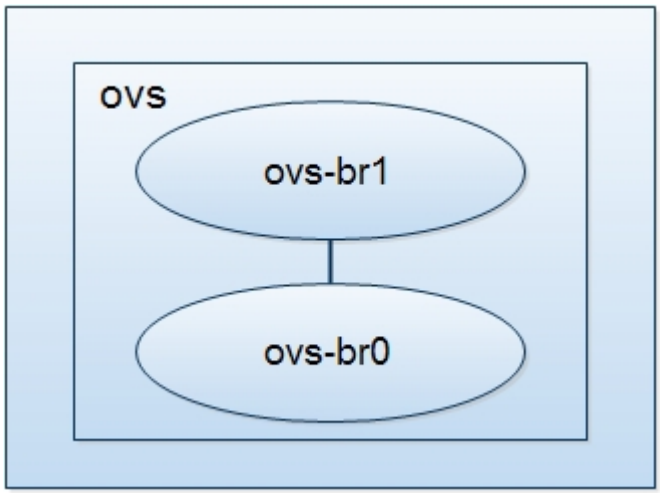


执行命令：

```
# ovs-vsctl add-br br0 //建立一个名为br0的open vswitch网桥

# ovs-vsctl add-port br0 eth0 //把eth0挂接到br0中

# kvm -m 512 -net nic,macaddr=00:11:22:33:44:55-net \

tap,script=/etc/ovs-ifup,downscript=/etc/ovs-ifdown-drive\

file=/path/to/disk-image,boot=on     //ovs-ifup和ovs-ifdown和上一节中相同
```

4、构建网桥和网桥相连的网络

以上操作都是将多个主机（物理机或虚拟机）连接到同一个网桥上，实现它们之间的通信，但是要构建复杂的网络，就需要多个网桥，在装有open vswitch的主机上建立两个网桥，实现它们之间的连接，构建结果如下：



执行命令：

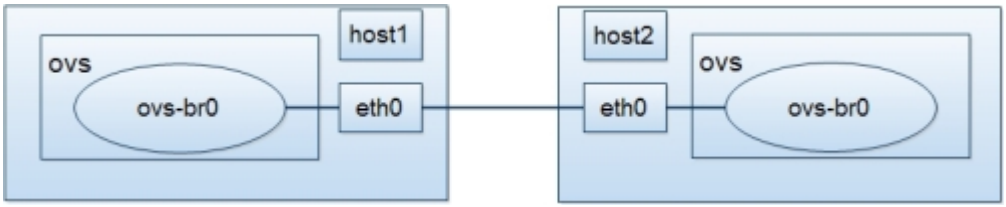
```
ovs-vsctl add-br br0    //添加一个名为br0的网桥

ovs-vsctl add-br br1    //添加一个名为br0的网桥
```

```
1
2
3
4
5   ovs-vsctl add-port br0 patch-to-br1    //为br0添加一个虚拟端口
6
7
8   ovs-vsctl set interface patch-to-br1type=patch      //把patch-to-br1的类型设置为patch
9
10
11  ovs-vsctl  set interface  patch-to-br1  options:peer=patch-to-br0      //把对端网桥和此网桥
12  br0
13
14
15
16
17
18
19  ovs-vsctl add-port br1 patch-to-br0    //为br0添加一个虚拟端口

    ovs-vsctl set interface patch-to-br0type=patch      //把patch-to-br0的类型设置为patch

    ovs-vsctl set interface patch-to-br0options:peer=patch-to-br1      //把对端网桥和此网桥连接
```

ovs-vsctl set interface patch-to-br0type=patch 和ovs-vsctl set interface patch-to-br0 options:peer=patch-to-br1是对ovs-database的操作，有有兴趣的同学可以参考ovs-vswitchd.conf.db.5

5、在不同的主机之间构建网桥之间的连接

在两台机器上分别安装上open vswitch并创建网桥，分别为两个网桥添加物理网卡，然后通过网线连接两个网桥，实现两个网桥之间的互通。构建结果图如下：



执行命令：

host1

```
#ovs-vsctl add-br br0              //添加名为br0的网桥

#ovs-vsctl add-port br0 eth0       //把eth0挂接到br0上
```

host2

```
#ovs-vsctl add-br br0              //添加名为br0的网桥

#ovs-vsctl add-port br0 eth0       //把eth0挂接到br0上
```

然后使用网线把host1的eth0和host2的eth0相连即可。

使用上边五种方法的组合就可以构建出各种复杂的网络，为各种实验提供网络的支持。