目录

架构

Floodlight 不仅仅是一个支持 OpenFLow 协议的控制器（FloodlightCOntroller），也是一个基于 Floodlight 控制器的应用集。
当用户在 OpenFLow 网络上运行各种应用程序的时候，Floodlight 控制器实现了对 OpenFLow 网络的监控和查询功能。图 0.0 显示了 Floodlight 不同模块之间的关系，这些应用程序构建成 java 模块，和 Floodlight 一起编译。同时这些应用程序都是基于 REST API 的。

开始

安装指导

概述

基于 Java 的 Floodlight 可以用标准 jak 工具或 ant 编译运行，当然也可以有选择性的在 Eclipse 上运行。

准备工作

Linux：

- Ubuntu 10.04（Natty）及以上版本（运行 Ant1.8.1 及以下版本）
- 安装 JDK，Ant。（可在 eclipse 上安装）
  $sudo apt-get install build-essential default ant python-dev eclipse

Mac

- Mac 系统 x10.6 及以上版本（低版本未测试）
- Cxode4.1 或 Xcode4.0.2
- JDK:只需要在终端输入命令：‘javac’便可安装
- Eclipse（非必须 ）

下载编译

从 Github 下载并比编译 Floodlight

```
$git clone git://github.com/floodlight/floodlight.git
$cd floodlight
$ant
```

运行 Floodlight

如果 java 运行环境已经安装成功，就可以直接运行：

```
$java -jar target/floodlight.jar
```

Floodlight 就会开始运行，并在控制台打印 debug 信息

Eclipse 设置

通过 Eclipse 运行、开发、配置 Floodlight：

$ant eclipse

上述命令将创建多个文件：Floodlight.launch,Floodlight_junit.launch,classpath 和.project。通过这些设置 eclipse 工程

- 打开 eclipse 创建一个新的工程
- 文件->导入->常规->现有项目到工程中->下一步
- 点击“选择根目录”，点击“浏览”。选择之前放置 Floodlight 的父路径
- 点击 Floodlight
- 点击“完成”

现在就产生了一个 Floodlight 的 Eclipse 工程。由于我们是使用静态模块加载系统运行 Floodlight，我们必须配置 eclipse 来正确的运行 Floodlight。

创建 Floodlight 目标文件:

- 点击运行->运行配置
- 右击 java 应用->新建
- “Name”使用“FloodlightLaunch”
- “Project”使用“Floodlight”
- “Main”使用“net.floodlightcontroller.core.Main”
- 点击“应用”

虚拟一个网络

启动了 Floodlight 之后，就需要链接到一个 OpenFlow 的网络。Mininet 是最好的网络虚拟工具之一。

- 下载 Floodlight-vm 机自启动并内嵌 Mininet 工具。
- 使用 Vmware 或者 virtualbox 打开 Floodlight-vm，在启动之前，点击网络选项不要从导入虚拟机后的安装目录中运行脚本启动
- 登录（用户名是：floodlight 没有密码）
- 既可以在使用 Floodlight-vm 开机自启动的 Floodlight 控制器也可以通过命令指定到远程的控制器，输入：

```
$sudo mn --controller=remote,ip=<controller ip>,port=<controller port>
```

可以通过 ssh 远程登陆到 floodlight-vm 虚拟机运行 wireshark，监听 eth0 端口并用“of”协议过滤器过滤。

```
$ssh -X floodlight@<vm-ip>
$sudo wireshark
```

下一步

阅读完 getstart 文档之后，可以参阅 floodlight 开发文档，里面有很多实例和代码。

可兼容交换机

下面列出了可以和 Floodlight 控制器兼容的交换机

虚拟交换机

- Open vSwitch（OVS）

硬件交换机

- Arista 7050
- Brocade MLXe

- Brocade CER
- Brocade CES
- Dell S4810
- Dell Z9000
- Extreme Summit x440, x460, x670
- HP 3500, 3500yl, 5400zl, 6200yl, 6600, and 8200zl (the old-style L3 hardware match platform)
- HP V2 line cards in the 5400zl and 8200zl (the newer L2 hardware match platform)
- Huawei openflow-capable router platforms
- IBM 8264
- Juniper (MX, EX)
- NEC IP8800
- NEC PF5240
- NEC PF5820
- NetGear 7328SO
- NetGear 7352SO
- Pronto (3290, 3295, 3780) - runs the shipping pica8 software

可支持的拓扑结构

Floodlight 现在在支持梁中锋不同的包转发应用，这两种应用具有不同的行为，并且向下图的拓扑结构一样运行：

- 在转发方面：在网络中的任意两个终端设备之间进行端到端的数据转发

  内含 OpenFLow 网络孤岛（OpenFlow island）：到同一 OpenFlow 网络孤岛中的任意一设备 A 发送数据包到设备 B，转发模块会计算出 A 到 B 之间的最短路径。

  内含非 OpenFlow 网络孤岛（non-OpenFlow island）的 OpenFLow 网络孤岛：每一个 OpenFlow 网络孤岛都可能有一个连接到非 OpenFlow 孤岛的链接。另外，如图 3 所示，OpenFlow 和非 OpenFlow 网络孤岛之间不能构成环网

图 1：Floodlight 接入 OpenFlow 网络的例网拓扑

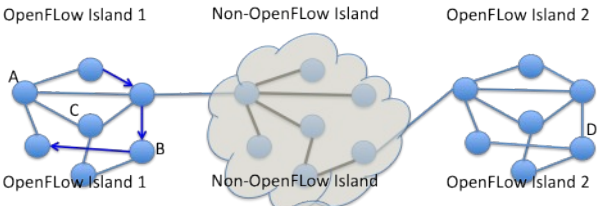


图 2：Floodlight 未接入 OpenFlow 网络的例网拓扑，OpenFlow 网络孤岛 1 有两个到非 OpenFlow 网络孤岛的链接

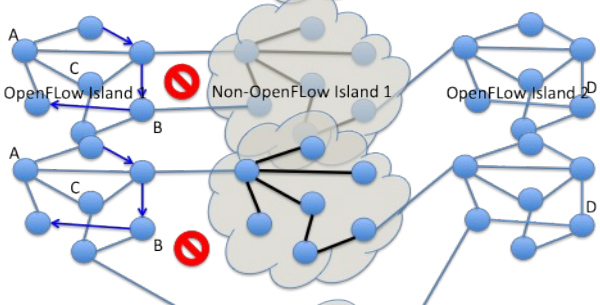
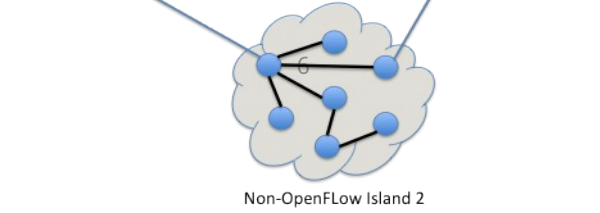


图 3：Floodlight 未接入



OpenFlow 网络的例网拓扑，OpenFlow 孤岛和非 OpenFlow 孤岛形成了一个环网，甚至每一个 OpenFlow 孤岛都有唯一一个连接到非 OpenFlow 孤岛的链接

- 当转发路径中超过指定的时间间隔（默认 5 秒），通过转发超时安装路径。
- 自我学习的交换机：一个简单的二层自我学习交换机：
  1 在任何数量的 OpenFlow 网络孤岛中使用，甚至在非 OpenFlow 的 2 层网络孤岛中。
  2 不能在环路网路中的网络孤岛中工作，也不能在孤岛形成的环网中工作
  3 数据转发效率远高于其他方法
- 另外，Floodlight 也提供一个 Static Flow Entry Pusher 应用和一个 Circuit Pusher 应用，允许用户主动安装转发路径（proactively install forwarding paths）的行为

1 Static Flow Entry Pusher 允许修改交换机流表项，从而创造由用户根据交换机端口明确选择的转发路径

2 CircuitPusher 是基于 Static Flow Entry Pusher，Device Manager，Routing services 的 RestAPI，在单个 OpenFlow 孤岛中建立一个最短路径流

*术语“孤岛”和“集群”是可以互换使用的。一个 OpenFlow 孤岛/集群就是 OpenFlow 交换机连接到其中任何设备的集合。类似的，非 OpenFlow 的孤岛/集群就是任何连接到非 OpenFlow 交换机的设备。

发布版

发行说明

Floodlight v0.9 发行说明

发布日期：2012 年 10 月

概述

Floodlighr v0.9 包含了控制器新的 RestAPI，新的应用，漏洞修复，新框架测试等

新添加的东西

REST APIs
显示如何进行外部连接，通过 BDDP 发现多跳链路而不是 LLDP

由于处理错误的 API 并不在发布包中，但是可以通过纯净版的 Floodlight 下载页面或者在 github 上的 Floodlight-master 下载获得。

- /wm/topology/external-links/json

显示在 LLDP 数据包中发现的直连（DIRECT）和隧道链接（TUNNEL）

- /wm/topology/links/json

给 OpenStack/quantum 插件的虚拟网络过滤器添加新的 API，以显示所有的创建的虚拟网络名，Guid，网关和主机

- /quantum/v1.0/tenants/<tenant>/networks

应用

- Circuit Pusher，一个基于 Python 的 REST 应用接口，使用 RESTAPI 来设置两台 IP 主机的流。包括新的 REST APIs
- Firewall，一个 java 应用模块，提供基于控制器的无状态 ACL 支持

用户文档

控制器

Floodlight 不仅仅是一个支持 OpenFLow 协议的控制器（FloodlightCOntroller），也是一个基于 Floodlight 控制器的应用集。
当用户在 OpenFLow 网络上运行各种应用程序的时候，Floodlight 控制器实现了对 OpenFLow 网络的监控和查询功能。图 0.0 显示了 Floodlight 不同模块之间的关系，这些应用程序构建成 java 模块，和 Floodlight 一起编译。同时这些应用程序都是基于 REST API 的。

当运行 floodlight 时，控制器和一组 java 应用模块（这些会在 floodlight 属性文件中载入）开始运行。REST API 通过 REST 端口（默认 8080）对所有的运行中的模块开放。

Configuration HOWTO

选择加载模块

Floodlight 可以配置载入不同的模块以适应不停地应用。配置不同的载入模块之后必须重启生效。目前对于 Floodlight 模块的配置都是通过需在启动时加载的一个配置文件的修改实现的。

简单的说，用户可以通过以下步骤，找到或控制 Floodlight 当前的配置:

- 打开 src/main/resources/META-INF/services/net.floodlightcontroller.core.module.IFloodlightModule 文件，就可以查看所有在 floodlight.jar 二进制编译中的模块编译。
- 打开 src/main/resources/floodlightdefault.properties 文件或者其他自定义属性文件，查看文件中选择加载/运行模块。这些文件时是配置某些启动时参数的地方，例如：REST API 服务和 WEB UI 端口（8080）、交换机连接 OpenFlow 端口（6633）、默认的超时值等。
  - 如果在 Eclipse 或命令行（java -jar floodlight.jar）运行 Floodlight，默认情况下加载默认属性文件。详细情况可以查看程序参数包括 -cf some_properties_file 文件。
  - 如果在 Floodlight-vm 上运行的 Floodlight，floodlight.jar 是按照 /opt/floodlight/floodlight/configuration/floodlight.properties instead of floodlightdefault.properties 文件，作为服务载入
  - 如果需要修改默认值，停止 Floodlight，更新以上所说的属性文件并重启。
- 如果自定义新的模块更新上述的两个文件并重启生效。

虽然大多数的应用都是按照默认属性配置运行，但是下面的应用程序需要一组特定的模块，因此需要一个特定的配置文件（由于某些模块之间不兼容）来运行。

- OpenStack Quantum plugin：需要和 src/main/resources/quantum.properties 文件一起运行
- Forwarding 和 StaticFlowEntryPusher：这两个模块都是默认加载的

，但有时你只需要加载其中的一个来实现应用程序的功能。例如，你想要一个完全自动配置的网络并且不会有转发反应，因此只需要 StaticFlowEntryPusher 模块而禁止 Forwarding 模块。

控制日志级别

在控制台显示的调试信息有时是很有帮助的，但有时又会显得十分繁杂。 Floodlight 使用 org.slf4j.Logger 模块，将日志信息划分不同的等级。同时，日志等级是可控的。在默认情况下，Floodlight 显示了所有的日志等级。为了控制日志等级，可以向 JVM 传递以下参数：

```
java -Dlogback.configurationFile=logback.xml -jar floodlight.jar
```

如果实在 Eclipse 下运行的 Floodlight，点击运行->运行/调试配置->参数->VM 参数，在这里添加 -Dlogback.configurationFile=logback.xml

Xml 文件已经包含在 Floodlight 根目录中。

```
<configuration scan="true">
  <appender name="STDOUT">
    <encoder>
      <pattern>%level [%logger:%thread] %msg%n</pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
  <logger name="org" level="WARN"/>
  <logger name="LogService" level="WARN"/> <!-- Restlet access logging -->
  <logger name="net.floodlightcontroller" level="INFO"/>
  <logger name="net.floodlightcontroller.logging" level="WARN"/>
</configuration>
```

在这个例子中，net,floodlightcontroller 包含了所有的 floodlight 模块，并且具有日志记录级别的信息，因此调试信息并不会出现在控制台中。你可以在 logxml 文件中指定 INFO，WARN 和 DEBUG 的级别。

监听地址和端口配置

为了改变主机地址或监听特定服务的端口号，可以在属性配置文件中使用以下的配置参数：

```
net.floodlightcontroller.restserver.RestApiServer.host
net.floodlightcontroller.restserver.RestApiServer.port
net.floodlightcontroller.core.internal.FloodlightProvider.openflowhost
net.floodlightcontroller.core.internal.FloodlightProvider.openflowport
net.floodlightcontroller.jython.JythonDebugInterface.host
net.floodlightcontroller.jython.JythonDebugInterface.port
```

默认的属性配置文件（例如：floodlightdefault.properties）

Floodlight REST API

虚拟网络过滤器的 REST API

URI	方法	URI 参数	数据	数据作用域	描述

/networkService/v1.1/tenants/{tenant}/networks/{network}	PUT/POST/DELETE	Tenant: 暂时忽略 Network: 网络ID（非网络名）	{"network": { "gateway": "<IP>", "name": "<Name>" }}\	IP:点分十进制的网关 IP, 可以为空 Name: 字符串形式的网络名	创建一个新的虚拟网络，ID 和 name 是必须的但是网关是可选的。
/networkService/v1.1/tenants/{tenant}/networks/{network}/ports/{port}/attachment	PUT/DELETE	Tenant: Currently ignored Network: The ID (not name) of the network Port: Logical port name	{"attachment": {"id": "<Network ID>", "mac": "<MAC>"}}	Network ID: 字符串形式的刚刚创建的网络的 ID MAC: 点分十进制的 MAC 地址	给虚拟网络添加主机
/networkService/v1.1/tenants/{tenant}/networks	GET	Tenant: Currently ignored	None	None	以 json 形式显示所有的网络的网络名、ID、网关，所有主机的 MAC 地址

Curl 使用样例

创建一个名字是“VirtualNetwork1”的虚拟网络，ID 是“Networkid1”，网关是“10.0.0.7”，tenant 是“默认”（目前是忽略的）：

```
curl -X PUT -d '{ "network": { "gateway": "10.0.0.7", "name": "virtualNetwork1" } }' http://localhost:8080/networkService/v1.1/tenants/default/networks/NetworkId1
```

添加一个主机到 VirtualNetwork1，MAC 地址为“00:00:00:00:00:08”端口为“port1”

```
curl -X PUT -d '{"attachment": {"id": "NetworkId1", "mac": "00:00:00:00:00:08"}}' http://localhost:8080/networkService/v1.1/tenants/default/networks/NetworkId1/ports/port1/attachment
```

StaticFlow Pusher API（新）

什么是 Static Flow Pusher？

Static Flow Pusher 是 Floodlight 的一个模块，通过 REST API 形式向外曝露，这个接口允许用户手动向 OpenFlow 网络中插入流表。

主动和被动流插入

OpenFlow 支持两种流插入方式：主动式和被动式。当数据包到达 OpenFlow 交换机但未成功匹配流表时发生被动式流表插入。这个数据包将会被发送到控制器，控制器对数据包进行分析评估，添加相应的流表并允许交换机继续该数据包的转发。另外，也可以在数据包到达交换机之前，控制器可以主动地插入相应流表。

Floodlight 支持这两种的流表插入方式。Static Flow Pusher 对于主动插入流表的方式很有帮助。注意，在默认情况下，Floodlight 载入的转发模块是被动插入流表模式的。如果只使用静态流表，就必须将 Forwarding 模块从 floodlight.properties 文件中删除。

使用方法

API 总结

URI	Description	Arguments
/wm/staticflowentrypusher/json	Add/Delete static flow	HTTP POST data (add flow), HTTP DELETE (for deletion)
/wm/staticflowentrypusher/list/<switch>/json	List static flows for a switch or all switches	switch: Valid Switch DPID (XX:XX:XX:XX:XX:XX:XX:XX) or "all"
/wm/staticflowentrypusher/clear/<switch>/json	Clear static flows for a switch or all switches	switch: Valid Switch DPID (XX:XX:XX:XX:XX:XX:XX:XX) or "all"

添加一个静态流表

Static Flow Pusher 是通过 REST API 方式接入，所以有很多访问方式。例如：要在 switch1 中插入一个流表，使得 port1 进入的数据包从 port2 转发出去。这一操作可以通过使用简单的 curl 命令实现。第二个命令将显示流表设置。

```
curl -d '{"switch": "00:00:00:00:00:00:00:01", "name":"flow-mod-1", "priority":"32768", "ingress-port":"1","active":"true","actions":"output=2"}'
http://<controller_ip>:8080/wm/staticflowentrypusher/json
curl http://<controller_ip>:8080/wm/core/switch/1/flow/json;
```

删除静态流表

通过发送包含流表名称的 HTTP DELETE 来删除静态流表

```
curl -X DELETE -d '{"name":"flow-mod-1"}'
```

http://<controller_ip>:8080/wm/staticflowentrypusher/json

流表项属性

Key	Value	Notes
switch	<switch ID>	ID of the switch (data path) that this rule should be added to xx:xx:xx:xx:xx:xx:xx:xx
name	<string>	Name of the flow entry, this is the primary key, it MUST be unique
actions	<key>=<value>	See table of actions below Specify multiple actions using a comma-separated list Specifying no actions will cause the packets to be dropped
priority	<number>	D 默认值为 32767 最大值为 32767
active	<boolean>	
wildcards		
ingress-port	<number>	switch port on which the packet is received Can be hexadecimal (with leading 0x) or decimal
src-mac	<mac address>	xx:xx:xx:xx:xx:xx
dst-mac	<mac address>	xx:xx:xx:xx:xx:xx
vlan-id	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制
vlan-priority	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制
ether-type	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制
tos-bits	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制
protocol	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制
src-ip	<ip address>	xx.xx.xx.xx
dst-ip	<ip address>	xx.xx.xx.xx
src-port	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制

dst-port	<number>	可以使 16 进制（以 0X 开头）或者是 10 进制

操作域的操作选项

Key	Value	Notes
output	<number> all controller local ingress-port normal flood	没有丢弃选项 (但可以不指定操作从而丢弃数据包)
enqueue	<number>:<number>	第一个数字是端口号，第二个是队列 ID 可以是 16 进制（0x 开头）或者 10 进制
strip-vlan		
set-vlan-id	<number>	可以是 16 进制（0x 开头）或者 10 进制
set-vlan-priority	<number>	可以是 16 进制（0x 开头）或者 10 进制
set-src-mac	<mac address>	xx:xx:xx:xx:xx:xx
set-dst-mac	<mac address>	xx:xx:xx:xx:xx:xx
set-tos-bits	<number>	
set-src-ip	<ip address>	xx.xx.xx.xx
set-dst-ip	<ip address>	xx.xx.xx.xx
set-src-port	<number>	可以是 16 进制（0x 开头）或者 10 进制
set-dst-port	<number>	可以是 16 进制（0x 开头）或者 10 进制

在主动插入方式中使用 Static Flow Pusher

Static Flow Pusher 可以通过编写简单的 python 代码脚本来控制。例如，在运行 floodlight 之后配置 mininet 虚拟机。默认的拓扑结构式一个交换机（s1）和两个连接到交换机的主机（h2，h3）。

```
sudo mn --controller=remote  --ip=<controller ip> --port=6633
```

以下代码是插入从 h2 发送到 h3 和 h3 发送到 h2 的流表

```
import httplib
import json

class StaticFlowPusher(object):

    def __init__(self, server):
        self.server = server

    def get(self, data):
        ret = self.rest_call({}, 'GET')
        return json.loads(ret[2])
```

```
    def set(self, data):
        ret = self.rest_call(data, 'POST')
        return ret[0] == 200

    def remove(self, objtype, data):
        ret = self.rest_call(data, 'DELETE')
        return ret[0] == 200

    def rest_call(self, data, action):
        path = '/wm/staticflowentrypusher/json'
        headers = {
            'Content-type': 'application/json',
            'Accept': 'application/json',
            }
        body = json.dumps(data)
        conn = httplib.HTTPConnection(self.server, 8080)
        conn.request(action, path, body, headers)
        response = conn.getresponse()
        ret = (response.status, response.reason, response.read())
        print ret
        conn.close()
        return ret

pusher = StaticFlowPusher('<insert_controller_ip')

flow1 = {
    'switch':"00:00:00:00:00:00:00:01",
    "name":"flow-mod-1",
    "cookie":"0",
    "priority":"32768",
    "ingress-port":"1",
    "active":"true",
    "actions":"output=flood"
    }

flow2 = {
    'switch':"00:00:00:00:00:00:00:01",
    "name":"flow-mod-2",
    "cookie":"0",
    "priority":"32768",
    "ingress-port":"2",
    "active":"true",
```

```
  "actions":"output=flood"
  }

pusher.set(flow1)
pusher.set(flow2)
```

为了测试这个例子，可以再 mininet 虚拟机上运行 pingall（注意：必须禁用交换机的学习功能和其他路由代码以确保交换机按照静态流表工作）

```
Mininet> h2 ping h3
```

Firewall REST API

Firewall REST 接口

防火墙模块提供 REST 接口服务，该接口实现了采用 REST API 服务形式的 RestletRoutable 接口。以下是 REST 方法的列表：

URI	Method	URI Arguments	Data	Data Fields	Description
/wm/firewall/module/<op>/json	GET	选项：status,enable,disable 存储规则，子网掩码	None	None	查询防火墙状态，启用停止防火墙
/wm/firewall/rules/json	GET	None	None	None	以 json 格式理出所有的规则
	POST	None	{"<field 1>":"<value 1>", "<field 2>":"<value 2>", ...}	"field":"value" 所有的序列组合："switchid":"<xx:xx:xx:xx:xx:xx:xx:xx>", "src-inport":"<short>", "src-mac": "<xx:xx:xx:xx:xx:xx>", "dst-mac": "<xx:xx:xx:xx:xx:xx>", "dl-type": "<ARP or	创建新的防火墙规则

				IPv4>", "src-ip": "<A.B.C.D/M>", "dst-ip": "<A.B.C.D/M>", "nw-proto": "<TCP or UDP or ICMP>", "tp-src": "<short>", "tp-dst": "<short>", "priority": "<int>", "action": "<ALLOW or DENY>" Note: specifying src-ip/dst-ip without specifying dl-type as ARP, or specifying any IP-based nw-proto will automatically set dl-type to match IPv4.	
	DELETE	None	{"<ruleid>": "<int>"}	"ruleid": "<int>" 注意：rule id 是成功创建规则是以 json 形式生成并返回的随机数	根据 rule id 删除规则

Curl 使用样例

假设控制器在本机上运行，显示防火墙运行还是禁用

```
curl http://localhost:8080/wm/firewall/module/status/json
```

启用防火墙。默认情况下防火墙禁用所有的流量除非创建新的明确允许的规则

```
curl http://localhost:8080/wm/firewall/module/enable/json
```

添加了允许所有流通过交换机 00:00:00:00:00:00:00:01 的规则

```
curl -X POST -d '{"switchid": "00:00:00:00:00:00:00:01"}'
http://localhost:8080/wm/firewall/rules/json
```

添加允许所有 IP 为 10.0.0.3 的主机到 IP 为 10.0.0.5 的主机的流的规则。不指定动作就是允许的规则

```
curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32"}'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32"}'
http://localhost:8080/wm/firewall/rules/json
```

添加允许所有 MAC 地址为 00:00:00:00:00:00:00:0b 的主机到 MAC 地址为 00:00:00:00:00:00:00:0c 的主机的流的规则

```
curl -X POST -d '{"src-mac": "00:00:00:00:00:0a", "dst-mac": "00:00:00:00:00:0a"}'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-mac": "00:00:00:00:00:0b", "dst-mac": "00:00:00:00:00:0b"}'
http://localhost:8080/wm/firewall/rules/json
```

添加允许 IP 为 10.0.0.3 的主机到 IP 为 10.0.0.5 的主机 ping 测试的规则。

```
curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32", "nw-proto":"ICMP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"dst-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32", "nw-proto":"ICMP" }'
http://localhost:8080/wm/firewall/rules/json
```

添加允许 IP 为难 10.0.0.4 到 IP 为 10.0.0.10 主机的 UDP 转发（如 iperf）规则，并禁止 5010 端口。

```
curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"dst-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "nw-proto":"UDP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "nw-proto":"UDP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "nw-proto":"UDP", "tp-
src":"5010", "action":"DENY" }' http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "nw-proto":"UDP", "tp-
src":"5010", "actio
```

应用

REST 应用

Circuit Pusher

Circuit Pusher 采用 floodlight REST API 在所有交换机上创建基于 IP 地址与指定的优先级两个设备之间路由的一个双向电路即永久性的流表项。

注意

1. Circuit Pusher 现在只能创建两个 IP 主机之间的，虽然只是简单的扩展以创建基于 CIDR 格式的 IP 前缀（例如：192.168.0.0/16）电路，但是支持 Static Flow Pusher。
2. 在向 Circuit Pusher 发送 restAPIRequest 之前,控制器必须已经检测到终端设备的存在（即终端设备已经在网络中发送过数据，最简单的办法就是用网络中任意两台主机 ping 一下），只有这样控制器才知道这些网络终端设备的接入点从而计算他们的路由。
3.当前支持的命令格式语法为：

a) circuitpusher.py --controller={IP}:{rest port} --type ip --src {IP} --dst {IP} --add --name {circuit-name}

在目前 IP 链路允许的情况之下在目的和源设备之间新建一个链路。ARP 自动支持。
目前，由工作目录中的一个文本文件 circuits.json 提供一个简单的链路记录存储。
这个文件没有设置任何保护，并且在控制器重启时也不会重置。这个文件需要正确的操作。
用户也应该确保当 floodlight 控制器重启时删除该文件。

b) circuitpusher.py --controller={IP}:{rest port} --delete --name {circuit-name}

通过之前创建链路时定义的链接名删除已创建的链路（即 3circuits.json 文件中的一条记录）

应用模块

Firewall

简介

Firewall 应用已经作为 floodlight 模块实现，防火墙模块在 OpenFlow 网络之中强制执行 ACL 规则（接入控制列表）以确保在网络中的交换机使用的是在 packet-in 监控行为之下的流。
ACL 规则就是一组交换机入口 permit，allow 或者 deny 数据流的条件。
每个数据流的第一个数据包与交换机已存在的防火墙规则匹配从而决定是否允许通过交换机。
防火墙规则会按照已分配的优先级排序并且匹配 OFMatch（OpenFlow 标准 1.0）中定义的 Packet-in 数据包包头域
匹配防火墙规则的高优先权决定了对流执行什么操作（允许/拒绝）。
在 OPMatch 定义中可以使用通配符。

防火墙策略

防火墙工作在被动模式中。
防火墙规则在创建时按照优先级排序（通过 REST API）。
每个进入交换机的 packet-in 数据包都会从列表中的最高优先级开始比较，直到匹配成功或者将列表中的全部比较完毕。
如果匹配成功，规则中的操作（允许/拒绝）将会被存储在 IPoutingDecision 对象中，然后传递到其他 packet-in 数据包处理管道之中。
这一结果将会最终到达 Forwarding 或者其他被选择的包转发应用（如 LearingSwitch ）中。
如果结果是允许，Forwarding 模块将会发送一个常规的转发流表项；如果结果是拒绝，Forwarding 将会丢弃流表项。
在这两种情况之下，被发送到交换机的流表项都必须完全映射匹配防火墙规则的匹配属性（包括通配符）。
防火墙实现允许规则会产生部分的重叠留空间，这通过判断优先级决定。在下面的简单的例子中，所有送达 192.168.1.0/24 子网的流除了进入的 HTTP（TCP 端口 80）都被禁止。

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

**数字越小优先级越高*

需要特别注意的是在有通配符的情况。如果一个流表不与第一个流表（最高优先级）但和第二个流表（较低优先级）匹配成功，这个流表将通过 Forwawrding 转发到交换机，但这个流表不会通配目的端口；然而，在这个流中的特定端口会被指定到流表项中从而使得通过 80 端口的数据包将不会被交换机丢弃也不会像控制器发送 packet-in 数据包。

REST API

防火墙模块通过 REST 接口实现，采用 REST API 服务形式的 RestletRoutable 接口。（REST 方法见 P14-P15 防火墙 REST 方法列表）

Curl 使用样例

（详情请见 P15 防火墙 curl 样例）

问题和限制

防火墙模块的 DELETE REST API 的调用并不会删除交换机上的流表项。规则只会删除控制器中的存储而交换机内的流表项则会按照标准超时行为自动丢弃。这意味着删除规则之后的一定时间之内，被删除的规则仍然有效。

在最初的提案中，TCP/UDP 的所有端口都在防火墙规则支持范围内。然而，由于 OpenFlow 的流匹配机制不允许指定端口范围，这个功能并没有实现。

负载均衡

一个简单的 ping，tcp，udp 流的负载均衡。这个模块是通过 REST API 访问的，类似于 OpenStatck Quantum LBbaas（Load-baloance-as-a-Service）v1.0 版 API 方案。详情见 http://wiki.openstack.org/Quantum/LBaaS。由于该提案尚未完善，所有兼容新并未得到明确的证明。

代码并不完整但支持基本的为 icmp,tcp,udp 服务创建和使用负载均衡

局限性：

客户端不会再使用之后清除静态流表记录，长时间之后会造成交换机流表用尽；

基于连接的服务器轮叫策略，而不是基于流量；

状态检测功能尚未实现。

欢迎从 Floodlight 社区获得帮助来改善运行情况

尝试以下基本特征：

1、下载 2012/12/12 发布的 floodlihgt-master

确认 net.floodlightcontroller.loadbalancer.LoadBanancer 在 floodlight.defaultpeoperties 中

启动 floodlight

启动 mininet，创建至少 8 台主机的网络。如：

$sudo mn --controller=rmote --ip=<controller_ip> --mac --topo=tree,3

在 mininet 中执行 pingall 命令

在任意 linux 终端中设置负载均衡 vips,pools 和 members。使用以下脚本：

```
#!/bin/sh

curl -X POST -d
'{"id":"1","name":"vip1","protocol":"icmp","address":"10.0.0.100","port":"8"}'
http://localhost:8080/quantum/v1.0/vips/

curl -X POST -d '{"id":"1","name":"pool1","protocol":"icmp","vip_id":"1"}'
http://localhost:8080/quantum/v1.0/pools/
```

```
curl -X POST -d '{"id":"1","address":"10.0.0.3","port":"8","pool_id":"1"}'
http://localhost:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"2","address":"10.0.0.4","port":"8","pool_id":"1"}'
http://localhost:8080/quantum/v1.0/members/



curl -X POST -d
'{"id":"2","name":"vip2","protocol":"tcp","address":"10.0.0.200","port":"100"}'
http://localhost:8080/quantum/v1.0/vips/

curl -X POST -d '{"id":"2","name":"pool2","protocol":"tcp","vip_id":"2"}'
http://localhost:8080/quantum/v1.0/pools/

curl -X POST -d '{"id":"3","address":"10.0.0.5","port":"100","pool_id":"2"}'
http://localhost:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"4","address":"10.0.0.6","port":"100","pool_id":"2"}'
http://localhost:8080/quantum/v1.0/members/



curl -X POST -d
'{"id":"3","name":"vip3","protocol":"udp","address":"10.0.0.150","port":"200"}'
http://localhost:8080/quantum/v1.0/vips/

curl -X POST -d '{"id":"3","name":"pool3","protocol":"udp","vip_id":"3"}'
http://localhost:8080/quantum/v1.0/pools/

curl -X POST -d '{"id":"5","address":"10.0.0.7","port":"200","pool_id":"3"}'
http://localhost:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"6","address":"10.0.0.8","port":"200","pool_id":"3"}'
http://localhost:8080/quantum/v1.0/members/
```

7、在 mininet 中，执行'h1 ping -c1 10.0.0.100',然后执行'h2 ping -c1 10.0.0.100'。这两次的 ping 都会成功的轮换调用两台不同的真实主机执行。

OpenStack

安装 Floodlight 和 OpenStatic

概述

以下介绍的是在 ubuntu 虚拟机上使用 BigSwitch 开发的 decstack 脚本安装（last build）和 OpenStack（Grizzy）。

条件

- Ubuntu12.04.1 服务器版即以上
- 至少 2GB RAM（扩展应用需要更多）
- 至少 30GB 存储空间

安装 floodlight

需要运行一个 flooflight 控制器来支持 OpenStack Neutron 网络。Floodlight 控制器可以运行在一个特定的 floodlight 虚拟机中（floodlight 官网上下载 floodlight-vm 虚拟机镜像文件）.或者你可以下载 floodlight.zip 源代码压缩文件解压后进行编译运行。只需要在你的 ubuntu 虚拟机中进行以下几步简单的操作：

```
确保你有正常的网络连接
$ sudo apt-get update
$ sudo apt-get install zip default-jdk ant
$ wget --no-check-certificate https://github.com/floodlight/floodlight/archive/master.zip
$ unzip master.zip
$ cd floodlight-master; ant
$ java -jar target/floodlight.jar -cf src/main/resources/neutron.properties
```

你可以通过以下步骤确保你的 VirtualNetworkFilter 成功激活：

```
$ curl 127.0.0.1:8080/networkService/v1.1
{"status":"ok"}
```

通过 RestProxy Neutron Plugin 使用 Devstatck 安装 OpenStatic

一旦 Floodlight 控制器运行之后，我们就准备使用安装脚本安装 OpenStack。下面的步骤 1 在虚拟机上配置 OVS 监听 Floodlight，步骤 2 在虚拟机上安装 OpenStatic 和 BigSwitch REST procy 插件。

OpenStack Grizzly 版

```
$ wget https://github.com/openstack-dev/devstack/archive/stable/grizzly.zip
$ unzip grizzly.zip
$ cd devstack-stable-grizzly
```

OpenStack Folsom 版

```
$ wget https://github.com/bigswitch/devstack/archive/floodlight/folsom.zip
$ unzip folsom.zip
$ cd devstack-floodlight-folsom
```

使用编辑器创建一个”localrc”文件并且填在下面。记住，用你选择的密码替换下面的<passward>并更新'BS_FL_CONTROLLERS_PORT=<floodlight IP address>:8080'。如果在同一台虚拟机上启动的 floodlight，可以使用 127.0.0.1 替换控制器的 IP 地址；否则使用远程控制器所在主机的正确 IP 地址。

```
disable_service n-net
enable_service q-svc
enable_service q-dhcp
enable_service neutron
enable_service bigswitch_floodlight
Q_PLUGIN=bigswitch_floodlight
Q_USE_NAMESPACE=False
NOVA_USE_NEUTRON_API=v2
SCHEDULER=nova.scheduler.simple.SimpleScheduler
MYSQL_PASSWORD=<password>
RABBIT_PASSWORD=<password>
ADMIN_PASSWORD=<password>
SERVICE_PASSWORD=<password>
SERVICE_TOKEN=tokentoken
DEST=/opt/stack
SCREEN_LOGDIR=$DEST/logs/screen
SYSLOG=True
#IP:Port for the BSN controller
#if more than one, separate with commas
BS_FL_CONTROLLERS_PORT=<ip_address:port>
BS_FL_CONTROLLER_TIMEOUT=10
```

然后：

```
$ ./stack.sh
```

需要注意的是安装 OpenStack 时间比较长并且不能中断。任何中断和网络连接失败都会导致错误并无法恢复。建议你在安装之前使用 VirtualBox 的“快照功能”进行保存。这样就可以很方便的保存后中断安装也不会影响到以后继续安装。

安装完成

如果安装成功完成就会显示：

```
Horizon is now available at http://10.10.2.15/
Keystone is serving at http://10.10.2.15:5000/v2.0/
Examples on using novaclient command line is in exercise.sh
The default users are: admin and demo
The password: nova
This is your host ip: 10.10.2.15
stack.sh completed in 102 seconds.
```

验证 OpenStack 和 Floodlight 安装

下面显示的是一个会话的安装 devstack 后的快照：

```
~/quantum-restproxy$ source openrc demo demo
~/quantum-restproxy$ quantum net-list
~/quantum-restproxy$ quantum net-create net1
```

创建一个新的网络：

Field	Value
admin_state_up	True
id	9c1cca24-3b7c-456d-afdd-55bc178b1c83
name	net1
shared	False
status	ACTIVE
subnets	
tenant_id	4fafea2aac994c13a5f3034a35e583f4

```
~/quantum-restproxy$ quantum subnet-create 9c1cca24-3b7c-456d-afdd-55bc178b1c83 10.2.2.0/24
```

创建一个新的子网：

Field	Value
allocation_pools	{"start": "10.2.2.2", "end": "10.2.2.254"}
cidr	10.2.2.0/24
dns_nameservers	
enable_dhcp	True
gateway_ip	10.2.2.1
host_routes	
id	7f03b14a-c15e-4d7b-81a0-8e9e6c6bbd88
ip_version	4
name	
network_id	9c1cca24-3b7c-456d-afdd-55bc178b1c83

tenant_id	4fafea2aac994c13a5f3034a35e583f4

```
~/devstack$ IMG_ID=`nova image-list | grep cirros | grep -v kernel | grep -v ram | awk -F "|" '{print $2}'`
~/devstack$ nova boot --image $IMG_ID --flavor 1 --nic net-id=9c1cca24-3b7c-456d-afdd-55bc178b1c83 vm1
```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-STS:power_state	0
OS-EXT-STS:task_state	scheduling
OS-EXT-STS:vm_state	building
accessIPv4	
accessIPv6	
adminPass	ZTYnQ7PB3Z7M
config_drive	
created	2012-11-01T06:23:32Z
flavor	m1.tiny
hostId	
id	fe362118-77e3-441d-bb68-608394256259
image	cirros-0.3.0-x86_64-uec
key_name	None
metadata	{}
name	vm1
progress	0
security_groups	[{u'name': u'default'}]
status	BUILD
tenant_id	4fafea2aac994c13a5f3034a35e583f4
updated	2012-11-01T06:23:32Z

user_id	69c5935b290543278fb3c4037b44fe8e

~/devstack$ nova list

ID	Name	Status	Networks
fe362118-77e3-441d-bb68-608394256259	vm1	ACTIVE	net1=10.2.2.3

~/quantum-restproxy$ ping 10.2.2.3

PING 10.2.2.3 (10.2.2.3) 56(84) bytes of data.
64 bytes from 10.2.2.3: icmp_req=1 ttl=64 time=15.9 ms
64 bytes from 10.2.2.3: icmp_req=2 ttl=64 time=0.684 ms
64 bytes from 10.2.2.3: icmp_req=3 ttl=64 time=0.433 ms
^C
~/quantum-restproxy$ ssh cirros@10.2.2.3
The authenticity of host '10.2.2.3 (10.2.2.3)' can't be established.
RSA key fingerprint is cf:b0:bb:0f:a6:00:0c:87:00:fd:c5:ac:1d:41:03:77.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.2.2.3' (RSA) to the list of known hosts.
cirros@10.2.2.3's password:
$ ifconfig
eth0 Link encap:Ethernet HWaddr FA:16:3E:51:2F:27
inet addr:10.2.2.3 Bcast:10.2.2.255 Mask:255.255.255.0
inet6 addr: fe80::f816:3eff:fe51:2f27/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:353 errors:0 dropped:95 overruns:0 frame:0
TX packets:236 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:34830 (34.0 KiB) TX bytes:28414 (27.7 KiB)
Interrupt:11

另一个例子（基于 Essex 版本，一些命令格式发生改变）

创建网络、租户、虚拟机

下载我们拥有了一个正在工作的 OpenStack 服务器，我们可以在其上启动多个虚拟机并通过 Quantum API 关联到不同的虚拟网络

odd_even_essex.sh 创建两个网络每个网络中有两个虚拟机。如果节点创建成功，将会有许多表格被打印，但是确保节点的 WAIT PATIENTALY 完成了了引导过程。四个虚拟机嵌套在一个虚拟机中，执行都会变得缓慢。

开发者文档

模块描述及 javadoc

控制器模块

简介：
控制器模块实现了大多数应用程序常用的一些功能，例如：

a).发现和揭露网络状态和事件
b)实现控制器和网络交换机的通讯
c)控制 Floodlight 模块以及资源共享，如存储、线程、测试
d)实现一个 web UI 以及调试服务器

FloodLight 目前已经实现的控制器模块

FloodlightProvider

描述：

FloodlightProvider 提供了两个主要部分的功能，它处理交换机之间的连接并且将 OpenFlow 的消息转化成其他模块可以监听的事件。第二个主要的功能，它决定某些特定的 OpenFlow 消息（即 PacketIn，FlowRemoved，PortStatus 等）被分派到该侦听消息的模块的顺序。然后模块可以决定允许该消息进入下一个监听对象或停止处理消息。

提供的服务：

- IFloodlightProviderService

服务依赖性：

- IStorageSourceService
- IPktinProcessingTimeService
- IRestApiService
- ICounterStoreService
- IThreadPoolService

Java 实现位置:

net.floodlightcontroller.core.FloodlightProvider.

如何工作：

FloodlightProvider 使用 Netty 库来处理到交换机的线程和连接，每个 OpenFlow 消息将通过一个 Netty 的线程进行处理，并执行与所有模块的消息相关联的所有逻辑。其他模块也可以注册类似交换机连接或断开和端口状态通知特定事件。FloodlightProvider 将把这些线协议通知转换成基于 Java 的消息，以便其它模块可以处理。为了使模块注册为基于 OpenFlow 消息的，他们必须实现 IOFMessageListener 接口。

局限性：

None

配置：

该模块是默认启用的，要想加载此模块，不需要改变任何配置

配置选项：

Name	Type	Default	Description
openflowport	Int	6633	TCP 端口用于监听来自支持 OpenFlow 的设备的连接。
workerthreads	Int	0 (2 * # of CPUs)	产生的 Netty 线程数，如果这个数字为 0，则线程数默认为机器 CPU 数量的两倍。
controllerid	String	localhost	控制器的 ID
role	String	master	值可以是： master, slave, equal。从（slave）控制器将不接受来自交换机连接，留作备用。

REST API：

URI	Description	Arguments	
/wm/core/switch/all/<statType>/json	获取所有交换机的状态	statType: port, queue, flow, aggregate, desc, table, features, host	
/	获取单个交换机	switchId: Valid Switch DPID	

wm/core/switch/<switchId>/<statType>/json	的状态	(XX:XX:XX:XX:XX:XX:XX:XX) statType: port, queue, flow, aggregate, desc, table, features, host	
/wm/core/controller/switches/json	列出所有连接到控制器的交换机DPID	none	
/wm/core/role/json	获得当前的控制器角色	None.	
/wm/core/counter/<counterTitle>/json	控制器中的所有流量计数列表	counterTitle: "all" or something of the form DPID_Port#OFEventL3/4_Type. See CounterStore.java for details.	
/wm/core/counter/<switchId>/<counterName>/json	某个交换机的流量计数列表	switchId: Valid Switch DPID CounterTitle: see above	
/wm/core/memory/json	当前控制器的内存使用情况	none	
/wm/core/module/{all}/json	模块以及模块依赖性的返回信息	all: "all" or "loaded".	

DeviceManagerImpl

描述:、

DeviceManagerImpl 追踪网络周围的设备，并为目标设备定义一个新的流。

提供的服务：

IDeviceService

服务依赖性：

IStorageSourceService
IRestApiService
ICounterStoreService
IThreadPoolService
IFlowReconcileService
IFloodlightProviderService

Java 实现位置:

net.floodlightcontroller.devicemanager.internal.DeviceManagerImpl.

如何工作：

设备管理器通过 PacketIn 请求了解设备，通过 PacketIn 消息获取信息，根据实体如何建立进行分类。默认情况下，entity classifies 使用了 MAC 地址和 VLAN 来识别设备。这两个属性定义一个独一无二的设备，设备管理器将了解其他属性，如 IP 地址。信息中一个重要的部分是设备的连接点，如果一个交换机接收到一个 PacketIn 消息，则交换机将会创建一个连接点， A device can have as many as one attachment point per OpenFlow island, where an island is defined as a strongly connected set of OpenFlow switches talking to the same Floodlight controller. 设备管理器也将根据时间清空连接点，IP 地址，以及设备本身。最近看到的时间戳是用来保持清空过程的控制。

限制：

设备是不可变的，这意味着你不能持有设备的引用，这些引用必须通过 IDeviceService APIs 获取。

配置：

该模块是默认启用的，加载模块不需要改变任何配置。

配置参数：

None

REST API :

URI	Description	Arguments	
/wm/device/	控制器追踪设备的列表，包括 MACs,IPs 以及接入点	Passed as GET parameters: mac (colon-separated hex-encoded), ipv4 (dotted decimal),vlan, dpid attachment point DPID (colon-separated hex-encoded) and port the attachment point port.	

通过 curl 调用简单的 REST

获取所有设备:

```
curl -s http://localhost:8080/wm/device/
```

获取 IP 地址为 1.1.1.1 的设备:

```
curl -s http://localhost:8080/wm/device/?ipv4=1.1.1.1
```

LinkDiscoveryManager (Dev)

描述：

链接发现服务负责发现和维护 OpenFlow 网络中的网络链接的状态。

提供的服务：

ILinkDiscoveryService

服务依赖性：

IStorageSourceService
IThreadPoolService
IFloodlightProviderService

Java 实现位置：

net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.

如何工作：

链路发现服务使用 LLDPs 和广播包检测链路。LLDP 的目的 MAC 为 01:80：C2：00:00:0e 和 BDDP 目的 MAC 是 FF：FF：FF：FF：FF：FF（广播地址），LLDP 和 BDDP 的以太类型为 0x88cc 和 0x8999。为了使拓扑结构被正确的学习，还有两个假设：

A).任何交换机(包括 OpenFlow 交换机)将包括一个本地链路包(LLDP)

B).Honors layer 2 broadcasts

链路可以是直接的或广播的，如果一个 LLDP 从一个端口被送出并被另一个端口接受，则建立直接的链路，这意味着端口是直连的。如果一个 BDDP 从一个端口发出，被另一个端口接受，这意味着还有另外一个二层交换机没有这在两个端口之间的控制器的控制下。

限制：

None

配置：

该模块是默认启用的，加载模块不需要改变任何配置。

配置参数：

None

REST API

URI	Description	Arguments	
/wm/topology/links/json	列出控制器检测出的所有链路	None.	

通过 curl 调用简单的 REST

获得所有设备：curl -s http://localhost:8080/wm/topology/links/json

TopologyService

描述：

TopologyService 为控制器维护拓扑信息，以及在网路中寻找路由。

在网络中提供的服务：

ITopologyService

IRoutingService

服务依赖性：

ILinkDiscoveryService

IThreadPoolService

IFloodlightProviderService

IRestApiService

Java 位置：

net.floodlightcontroller.topology.TopologyManager.

如何工作：

拓扑服务基于从 lLinkDiscoveryService 学习到的链路信息进行计算拓扑，该 TopologyService 保持的一个重要概念是 OpenFlow 的“island”的想法。一个 island 被定义为一组同一 floodlight 实例下强连接 d 的 OpenFlow 交换机。isLand 可以使用在同一个 2 层域中非 OpenFlow 的交换机进行互连，例如：

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

两个 island 将由拓扑服务来形成，isLand1 包含 switch1 和 switch2，而 island2 只包含 switch3。

当前的拓扑信息将被存储在称为拓扑实例的不可变的数据结构中，在拓扑结构中如果有任何变化,一个新的实例将被创建并且拓扑变化通知消息将被调用。如果其他模块想监听拓扑结构的变化，它们需要实现 ITopologyListener 接口。

限制：

尽管你可以在一个 OpenFlow isLand 中有冗余链路，但不可以有从非 OpenFlow 交换机到 OpenFlow island 的冗余链路。

配置：

此模块被默认加载，不需要其它配置。

配置参数：

None

REST API

URI	Description	Arguments
/wm/topology/switchclusters/json	列出由控制器计算出的交换机集群	None.

简单应用：

获取所有设备：

Curl -s http://localhost:8080/wm/topology/switchclust

RestApiServer

描述：

REST API 服务器允许模块通过 HTTP 暴露 REST API。

提供的服务：

IRestApiService

服务依赖性：

None

Java 位置：

net.floodlightcontroller.restserver.RestApiServer.

如何工作：

REST API 服务使用 Restlets library。通过 REST 服务器作为一个依赖的其他模块通过添加一个实现 RestletRoutable 的类进行公开 API。每个 RestletRoutable 包含附加一个 Restlet 资源的路由器（最常见的是 ServerResource）。用户会附上自己的类扩展 Restlet 的资源，以处理特定 URL 的请求。里面的资源注释，如@ GET，@ PUT 等，都是选择哪个方法将被用于 HTTP 请求。序列化通过包含在 Restlet 库中的 Jackson library 实现。Jackson 可以通过两种方式进行序列化对象，第一，它会自动使用可用的 getter 对对象进行序列化这些字段，否则，自定义序列化可以创建和注释在类的顶部。

限制：

基本的路径不能重叠，并且是唯一的。

Restlets 只能通过服务接口访问模块的数据。如果一个模块需要通过 REST 服务器来公开数据，则它必须通过公开接口来得到这个数据。

配置：

此模块被默认加载，不需要其它配置。

配置参数：

Name	Type	Default	Description
port	Int	8080	The TCP port to listen on for HTTP connections.

REST API

None.

ThreadPool

描述：

ThreadPool 是一个 floodlight 模块被封装为一个 Java 的 ScheduledExecutorService，它可用于使线程在特定的时间或周期性地运行。

提供的服务：

IThreadPoolService

服务依赖性：

None

文件位置：

net.floodlightcontroller.threadpool.ThreadPool.

限制：

None

配置：

此模块被默认加载，不需要其它配置。

配置参数：

None

REST API：

None

MemoryStorageSource

描述:

该 MemoryStorageSource 是在内存中的 NoSQL 风格的存储源。也支持更改通知数据库

提供服务：

IStorageSourceService

服务依赖性：

ICounterStoreService

IRestApiService

Java 位置：

net.floodlightcontroller.storage.memory.MemoryStorageSource.

如何工作：

其它依赖于 IStorageSourceService 接口的 FloodLight 模块可以 create/delete/modify 内存资源中的数据，所有的数据是共享的，且没有强制执行。模块还可以注册在制定表和制定行的数据中进行修改，其它任何想实现这样功能的模块需要实现 IStorageSourceListener 接口。

限制：

一旦数据存储在内存中，当 FloodLight 关闭时，所有的状态将会丢失。

没有隔离强制执行的数据，即使一个模块创建了一张表，另一个模块可以覆写那部分数据。

配置：

此模块被默认加载，不需要其它配置。

配置参数：

None

REST API：

None

Flow Cache

flowCache API 由于记住在网络中一系列不同类型的事件的需要而被定义，而事件的处理，以及如何处理往往构建于不同的 Floodlight SDN 应用程序。例如，处理交换机/链路故障事件流是大多数应用程序最典型的需要。

Floodlight 定义了一个流缓存 API 和一组框架方法作为通用框架为应用程序开发人员实现适合他们的应用需求的解决方案。

我们正在努力的编写 API，晚些时候会提交到 floodlight 的官方网站上，与此同时，API 调用简洁的说明也可以在 flow cache 源中找到。

交换机/链路故障事件的例子:

对于流缓存目的的一个更高层次的解释，我们可以通过 switch/link 中断事件的生命周期来了解各种被调用的模块。

1.目前，当 LinkDiscoveryManager 检测到链路或端口出现故障，该事件由在 TopologyManager 中的一个“NewInstanceWorker”线程处理。请注意，在线程结束时，它调用 informListeners，这是一个标记用于告知此事件于其它对处理此事件有兴趣的模块。

2.所以，你会从实现一个实现了 ITologyListener 接口和 topologyChanged()方法的模块开始，并且通过调用 TopologyManager.addListener 将此模块添加到侦听器列表中。

3.在这个模块中你可以通过 Topology.getLastLinkUpdates()方法获取所有的之前发现的拓扑变化，并对事件进行排序去查看你感兴趣的事件。一个交换机出现故障导致相邻交换机链路断开，所以你应该寻找 ILinkDiscoery.UpdateOperation.LINK_REMOVED 事件（每个受影响的交换机有一个事件），找到的条目将告诉你所涉及的交换机端口。

4.接下来是要查询每一个受影响的交换机中所有的流表和受影响的端口进行匹配。该查询应该是一个 OFStatisticsRequest 消息，该消息将通过 sw.sendStatsQuery()被送到交换机。

5.一旦查询被发送出去，稍后你会收到响应，为了能够接收到 OF 包的响应，你的模块必须实现 IOFMessageListener 接口以及 OFType.STATS_REPLY 消息。一旦你接收到响应，你会看到它所有的流表项。现在你可以决定是否想要创建删除流表修改消息来清理流表项。

这似乎已经解决了这个问题，但我们还没有用到过流缓存以及与其相关的服务接口呢。

流缓存的概念是为了使控制器记录所有有效流，并且当事件被一个控制器的其它模块观察或者实时查询交换机时，此流缓存记录会更新。这种方式整合了不同模块更新和检索流记录。

流高速缓冲存储器的数据结构是留给实现者来决定的，而查询和响应（反流高速缓冲存储器）格式显示在 API 中。每个查询也可以指定其处理程序。

Flow reconcile 类是为了清理缓存以及交换机中的流表项，你可以由多个模块来处理不同的事件，每个模块将实现 IFlowReconcileListene 接口和 reconcileFlows 方法。这种方法既可以立即产生操作，你也可以通过 OFMatchReconcile 对象将决定传递给另一个模块。也有一些接口被定义为保持挂起的查询。

Packet Streamer

描述：

Packetstreamer 是包流服务，可以选择性的在交换机和控制器之间流式传输 OpenFlow 包。它包含了两个功能性接口：

1).基于 REST 的接口来定义自己感兴趣的 OpenFlow 消息的特性，被称为过滤器

2)一个基于 Thrift 接口的流过滤的数据包

REST API

过滤器定义的一个 Post 请求："http://<controller>:8080/wm/core/packettrace/json". 输入的数据是定义了我们感兴趣的 OpenFlow 消息特点的参数。FloodLight 配备了一个基于 Mac 地址的过滤器，例如，下面是一个过滤器的格式：

{'mac':<hostMac>, 'direction':<direction>, 'period':<period>, 'sessionId':<sessionid>}

Name	Value	Description
mac	<hostMac>	The OFMessage with matching hostMac (in the Ethernet frame in its payload) will be streamed.
direction	in	OFPacketIn
	out	OFPacketOut and FlowMod
	both	in and out
period	<period>	Defines the duration of the streaming session in seconds.
	-1	terminate the given session
sessionid	<sessionid>	The session to be terminated when period = -1. Otherwise, it is ignored.

对 REST API 返回的 sessionId，它可以被用来从流过滤服务器接收数据，数据是以 Json 格式返回的。

{'sessionId':<sessionid>}

下面是创建一个 1000 秒流会话的 Python 例子以及一个用来终止会话的函数。

在开始之前，确定你已经启动了 PacketStreamerServer

```
url = 'http://%s:8080/wm/core/packettrace/json' % controller

filter = {'mac':host, 'direction':'both', 'period':1000}

post_data = json.dumps(filter)

request = urllib2.Request(url, post_data, {'Content-Type':'application/json'})
```

```
response_text = None


try:
    response = urllib2.urlopen(request)
    response_text = response.read()
except Exception, e:
    # Floodlight may not be running, but we don't want that to be a fatal
    # error, so we just ignore the exception in that case.
    print "Exception:", e
    exit


if not response_text:
    print "Failed to start a packet trace session"
    sys.exit()


response_text = json.loads(response_text)


sessionId = None
if "sessionId" in response_text:
    sessionId = response_text["sessionId"]


def terminateTrace(sid):
    global controller


    filter = {SESSIONID:sid, 'period':-1}
    post_data = json.dumps(filter)
    url = 'http://%s:8080/wm/core/packettrace/json' % controller
    request = urllib2.Request(url, post_data, {'Content-Type':'application/json'})
    try:
```

```
        response = urllib2.urlopen(request)
        response_text = response.read()
    except Exception, e:
        # Floodlight may not be running, but we don't want that to be a fatal
        # error, so we just ignore the exception in that case.
        print "Exception:", e
```

基于过滤器的流服务：

包流服务是由一个基于 Thrift 流服务器代理。

该 Thrift 接口如下表所示，完整的 Thrift 接口见：src/main/thrift/packetstreamer.thrift

```
service 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)
}
```

floodlight 创建脚本为 Thrift 服务创建 java 和 python 库，其它语言的支持可以很容易的通过向创建脚本中添加语言选项：setup.sh

REST API 描述了流回话的创建。一旦 sessionId 被创建，过滤接口 getPackets(sessionId)，可以被用于给特定的部分接收 OF 包，terminateSession(sessionId)可以被用于终止实时会话。

下面是一个 python 的例子：

```
try:
  # Make socket
  transport = TSocket.TSocket('localhost', 9090)
  # Buffering is critical. Raw sockets are very slow
  transport = TTransport.TFramedTransport(transport)
  # Wrap in a protocol
  protocol = TBinaryProtocol.TBinaryProtocol(transport)
  # Create a client to use the protocol encoder
  client = PacketStreamer.Client(protocol)
  # Connect!
  transport.open()

  while 1:
    packets = client.getPackets(sessionId)
    for packet in packets:
      print "Packet: %s"% packet
      if "FilterTimeout" in packet:
        sys.exit()

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

客户端例子：

Floodlight 当前的版本带有基于 Mac 地址的数据包流的例子。客户端的一些代码都列在了前面的章节。

一个完整的 python 客户端例子，即描述了 REST API 和 Thrift 客户端的使用情况，可以再 floodlight 的源代码 net.floodlightcontroller.packetstreamer 中找到。

请确保在客户端机器中已经安装了 thrift 并且为 packetstreamer 的 gen-gy 和 thrift python 的目录给于了正确的路径

应用模块

虚拟网络过滤器（Quantum 插件）

简述：
虚拟网络过滤器模块是基于虚拟化网络的数据链路层。它允许你在独立的数据链路层上创建多个逻辑链路。这个模块可用于 OpenStack 的部署或者单例。

服务提供：
- IVirtualNetworkService

服务依赖：
- IDeviceService
- IFloodlightProviderService
- IRestApiService

Java 文件：
此模块实现在 net.floodlightcontroller.virtualnetwork.VirtualNetworkFilter。

如何工作：
在 Floodlight 启动时，没有虚拟网络创建，这时主机之间不能相互通信。一旦用户创建虚拟网络，则主机就能够被添加。在 PacketIn 消息转发实现前，模块将启动。一旦，一条 PacketIn 消息被接受，模块将查看源 MAC 地址和目的 MAC 地址。如果 2 个 MAC 地址是在同一个虚拟网络，模块将返回 Command.CONTINUE 消息，并且继续处理流。如果 MAC 地址不在同一虚拟网络则返回 Command.STOP 消息，并且丢弃包。

限制：
- 必须在同一个物理数据链路层中。
- 每个虚拟网络只能拥有一个网关（）（一个网关可被多个虚拟网络共享）。
- 多播和广播没有被隔离。
- 允许所有的 DHCP 路径。

配置：
模块不是默认启用的。它必须被加入配置文件，加入后，为了成功加载，重启 Floodlight。这个模块叫做“VirtualNetworkFilter”。包含此模块的默认配置文件位置：
src/main/resources/quantum.properties

```
# The default configuration for openstack
floodlight.modules = net.floodlightcontroller.storage.memory.MemoryStorageSource,\
net.floodlightcontroller.staticflowentry.StaticFlowEntryPusher,\
net.floodlightcontroller.forwarding.Forwarding,\
```

```
net.floodlightcontroller.jython.JythonDebugInterface,\
net.floodlightcontroller.counter.CounterStore,\
net.floodlightcontroller.perfmon.PktInProcessingTime,\
net.floodlightcontroller.ui.web.StaticWebRoutable,\
net.floodlightcontroller.virtualnetwork.VirtualNetworkFilter
net.floodlightcontroller.restserver.RestApiServer.port = 8080
net.floodlightcontroller.core.FloodlightProvider.openflowport = 6633
net.floodlightcontroller.jython.JythonDebugInterface.port = 6655
```

如果你正在使用 Floodlight 虚拟机，机子中已经有配置文件，简单的执行一下命令来启动它。

```
floodlight@localhost:~$ touch /opt/floodlight/floodlight/feature/quantum

floodlight@localhost:~$ sudo service floodlight stop

floodlight@localhost:~$ sudo service floodlight start
```

REAT API

URL	Method	URL Arguments	Data	Data fields	description
/ networkService/v1.1/tenants/ {tenant}/networks/ {network}	PUT/POST/ DELETE	Tenant:忽略 网络：网络的 id	{"network": { "gateway": "<IP>", "name": "<Name>" }}\	IP:网关 IP 以“1.1.1.1”的格式，可为空 名字：网络以字符串命名。	创建新的虚拟网络，名字和 ID 是必须的，网关可选

/networkService/v1.1/tenants/{tenant}/networks/{network}/ports/{port}/attachment	PUT/DELETE	网络：网络 ID 端口：逻辑端口名	{"attachment": {"id": "<Network ID>", "mac": "<MAC>"}}	Network ID: Network ID 自己创建的字符串 MAC: MAC 地址以 "00:00:00:00:00:09" 格式	将主机与虚拟网络连接
/networkService/v1.1/tenants/{tenant}/networks	GET				以 json 格式将所有网络的网关，ID 以及主机的 MAC 地址显示出来

实例：

创建一个名为“VirtualNetwork1”的虚拟网，ID 设为“NetworkId1”，网关为“10.0.0.7”，租户是默认的（当前是忽略的）。

```
curl -X PUT -d '{ "network": { "gateway": "10.0.0.7", "name": "virtualNetwork1" } }'
http://localhost:8080/networkService/v1.1/tenants/default/networks/NetworkId1
```

向虚拟网络中添加一个 MAC 地址为“00:00:00:00:00:08”，端口为“port1”的主机。

```
curl -X PUT -d '{"attachment": {"id": "NetworkId1", "mac": "00:00:00:00:00:08"}}'
http://localhost:8080/networkService/v1.1/tenants/default/networks/NetworkId1/ports
/port1/attachment
```

转发

简介：
转发将在 2 个设备之见转发包。源设备和目的设备通过 IDeviceService 区分。

服务提供：

- 没有

服务依赖：

- IDeviceService
- IFloodlightProviderService
- IRestApiService
- IRoutingService
- ITopologyService
- ICounterStoreService

Java 文件：
此模块实现在 net.floodlightcontroller.forwarding.Forwarding。

如何工作：
交换机需要考虑到，控制器可能需要工作在一个包含 Openflow 交换机和非 Openflow 交换机的网络中。模块将发现所有的 OF 岛。FlowMod 将被安装到最短路径上。如果一条 PacketIn 被接收到一个 OF 岛，而该岛没有挂载点，则这个网包将被洪泛。

限制：
不提供路由功能。
没有 VLAN 的加减包头。

配置：
该模块默认启动，在加载模块时，配置无需更改。

防火墙

简介：
防火墙已被作为一个模块实现，它通过 ACL 规则实现流量过滤。每个被首包触发的 PacketIn 消息

将跟规则集进行匹配，按照最高权值匹配规则行为进
行处理。防火墙匹配的最高优先级决定流的操作。通配符在 OFMatc 中用作定义。

防火墙策略：

防火墙被动运行。防火墙规则在他们被创建时（通过 REST API）通过优先级排序。每个 packet_in 将从列表中的最高级开始匹配，直到列表结束。如果找到匹配，操作指令（允许或拒绝）储存到一个 IRoutingDecision 对象中，并发送其余的 pack_in 处理管道。指令最后将到达转发模块或者其他数据包转发的模块(例如 LearningSwitch)。如果，指令允许操作，转发模块将推送一个常规的转发流表，否则，推送一个丢弃流表。不管哪种，被推送给交换机的流表都必须准确的反应出防火墙规则的匹配属性(包括通配符).

因此实现的防火墙,根据不同的优先级,允许拥有部分重叠的流空间。下面是个简单的例子，192.168.1.0/24 段内的子网的流量都是被拒绝的，除了入站 HTTP(TCP 端口 80)流量.

协议	目的 IP	目的端口	指令	优先级
TCP	192.168.1.0/24	80	ALLOW	1
TCP	192.168.1.0/24	Wildcard	DENY	2

优先级数字越低，优先级别越高。

这里要特别处理通配符。如果流没有匹配最高级，而匹配了次高级，那么由转发模块发送给交换的流表将不会通配目的端口，

而是在流表中指定端口，所以 80 端口的包将不会被丢弃。

REST 接口

防火墙模块实现了 REST 接口，该接口实现了采用 REST API 服务的 RestletRoutable 的接口。下面是 REST 方法的列表。

URI	Method	URI Arguments	Data	Data Fields	Description
/wm/firewall/module/<op>/json	GET	op: status, enable, disable, storageRules, subnet-mask	None	None	查询防火墙的启动专业。

/wm/firewall/rules/json	GET	None	None	None	以 json 的格式列出规则集
	POST	None	{"<field 1>":"<value 1>", "<field 2>":"<value 2>", ...}	"**field**":"**value**" pairs below in any order and combination: "**switchid**":"<xx:xx:xx:xx:xx:xx:xx:xx>", "**src-inport**":"<short>", "**src-mac**": "<xx:xx:xx:xx:xx:xx>", "**dst-mac**": "<xx:xx:xx:xx:xx:xx>", "**dl-type**": "<ARP *or* IPv4>", "**src-ip**": "<A.B.C.D/M>", "**dst-ip**": "<A.B.C.D/M>", "**nw-proto**": "<TCP *or* UDP *or* ICMP>", "**tp-src**": "<short>", "**tp-dst**": "<short>", "**priority**": "<int>", "**action**": "<ALLOW *or* DENY>" Note: specifying src-ip/dst-ip without specifying dl-type as ARP, or specifying any IP-based nw-proto will automatically set dl-type to match IPv4.	创建新的防火墙规则
	DELETE	None	{"<ruleid>":"<int>"}	"**ruleid**": "<int>" Note: ruleid 是 rules 创建成功时返回的一个随机数	通过 ruleid 删除 rule

实例：

假定控制器运行在本机。显示出防火墙是否被启动。

```
curl http://localhost:8080/wm/firewall/module/status/json
```

启动防火墙。默认的，防火墙拒绝所有流，除非一个显式的 ALLOW rule 被创建。

```
curl http://localhost:8080/wm/firewall/module/enable/json
```

添加一个 ALLOW rules 为所有的流，来能够通过交换机 00:00:00:00:00:00:00:01。

```
curl -X POST -d '{"switchid": "00:00:00:00:00:00:00:01"}'
http://localhost:8080/wm/firewall/rules/json
```

为 ip 为 10.0.0.3 和 10.0.0.7 的主机的所有的流添加一个 ALLOW rules。Action 意味着 ALLOW rules

```
curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32"}'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32"}'
http://localhost:8080/wm/firewall/rules/json
```

为 mac 地址为 00:00:00:00:00:0a 和 00:00:00:00:00:0b 的主机的所有流添加一个 ALLOW rules。

```
curl -X POST -d '{"src-mac": "00:00:00:00:00:0a", "dst-mac": "00:00:00:00:00:0b"}'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"dst-mac": "00:00:00:00:00:0b", "dst-mac": "00:00:00:00:00:0a"}'
http://localhost:8080/wm/firewall/rules/json
```

添加一个 ALLOW rules 使 ip 为 10.0.0.3 和 10.0.0.7 的主机能够 ping 通

```
curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.3/32", "dst-ip": "10.0.0.7/32", "nw-proto":"ICMP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.7/32", "dst-ip": "10.0.0.3/32", "nw-proto":"ICMP" }'
http://localhost:8080/wm/firewall/rules/json
```

添加一个 ALLOW rules,ip 为 10.0.0.4 和 10.0.0.10 主机之间能够 UDP 通信，同时阻塞 5010、端口

```
curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "dl-type":"ARP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "nw-proto":"UDP" }'
http://localhost:8080/wm/firewall/rules/json
```

```
curl -X POST -d '{"src-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "nw-proto":"UDP" }'
http://localhost:8080/wm/firewall/rules/json

curl -X POST -d '{"src-ip": "10.0.0.4/32", "dst-ip": "10.0.0.10/32", "nw-proto":"UDP", "tp-
src":"5010", "action":"DENY" }' http://localhost:8080/wm/firewall/rules/json
curl -X POST -d '{"src-ip": "10.0.0.10/32", "dst-ip": "10.0.0.4/32", "nw-proto":"UDP", "tp-
dst":"5010", "action":"DENY" }' http://localhost:8080/wm/firewall/rules/json
```

测试方法：

测试包括自动化单元，自动化单元通过 EasyMock 创建。”Firewalltest”类包括可由 JUnit 和 Eclipse 执行的测试案例。在大多数的测试案例中，packer_in 事件是模拟的，由此产生的防火墙行为能够被验证，它们是基于被定义的规则集的。以下是各种测试案例。

testNoRules

1 Description: 没有任何的 rules，发送一个 packet_in 事件，.防火墙将会拒绝所有流。这是一个边界测试案例

testRuleInsertionIntoStorage

2 Description: 添加一个 rules，通过检查存储来验证. 这是一个简单 positive 测试案例.

testRuleDeletion

3 Description: 删除一个 rules，通过检查存储来验证. Again, 这是一个简单 positive 测试案例.

testFirewallDisabled

4 Description: 在防火墙没有启动时，插入一个 rules，并发送一个 packet_in 事件. 防火墙将会拒绝所有的流.这是一个简单的 negative 的测试案例.

testSimpleAllowRule

5 Description:添加一个简单的 rules，使两个 2 不同的 ip 之间能够 tcp 通信，并且发送一个 packet_in 事件。在验证防火墙行为之后，能够发送另一个应该被丢弃的包 。This test case covers normal rules (non-boundary case – i.e. no malformed packets or broadcasts)这个测试案例包含一个简单的 rules（无边界案例——例如 一个完好的包或者广播）

testOverlappingRules

6 Description:添加 overlapping rules (拒绝所有的 TCP 流 除非目的端口是 80). 这个测试案例包含复杂情况下的多个规则(multiple allow-deny overlapping rules).

testARP

7 Description:测试一个 ARP 广播请求包和单播 ARP 回应。没有允许 ARP 回应的防火墙 rules，所以只有广播请求包能够通过.

testIPBroadcast

8 Description: 够在防火墙没有任何 rules 情况下，发送一个 ip 广播（L3）packt_in 事件。这是一个包含 IP 广播的 positive 测试案例（L3+L2 广播）.

testMalformedIPBroadcast

9 Description:在没有任何 rules 情况下,发送一个坏的 IP 广播 packet_in 事件. 防火墙将会拒绝这个流, 因为这个包一个 L2 广播 L3 单播. 这是一个边界案例.

testLayer2Rule

10 Description:一个规则允许指定的 MAC 通信，另一个规则拒绝所有的 TCP 通信. 防火墙将接受这个流. 这是一个 negative 测试案例，这里规则包含 L2.

问题和局限:

1.防火墙 模块 DELETE REST API 功能的调用没有删除交换机上的流表。Rules 将会被从控制器存储中删除，当交换机上的流表处理时间超过标准。这意味着在一段时间后，删除规则是有效的。流可以持续存在,只要它在交换机中持续通信.

2.最初的，TCP/UDP 端口范围是通过防火墙 rules 来支持的。但是，作为 OpenFlow 流匹配机制不允许指定端口范围,此功能没有实现。

Port Down Reconciliation

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。

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 flo towards the base switch)

sw2	3	[match1: dataLayerDestination: "7a:59:cd:3 "c6:cb:ad:80:49:70"]
sw3	3	[match2: dataLayerDestination: "7a:59:cd:3 "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。
问题和局限:
1.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。
2.这个模块依赖于转发模块实现。

模块加载系统

简介:
Floodlight 使用自己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
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。
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
 */
```

```
public 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);
}
```

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
 *
 */
public 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。

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：

```
floodlight.modules = net.floodlightcontroller.staticflowentry.StaticFlowEntryPusher,\
net.floodlightcontroller.forwarding.Forwarding,\
net.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。

模块文件:

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：

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

启动序列：

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）

服务映射：建立服务和提供该服务的模块之间的映射

模块服务映射：建立模块和他提供的所有服务之间的映射

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。

通过命令行使用控制器:

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

```
$java -jar floodlight.jar
```

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。
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。

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。
java -cp floodlight.jar:/path/to/other.jar net.floodlightcontroller.core.Main -cf path/to/config/file.properties
-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）。

每个模块的配置选项:
Properties 文件能够指定每个模块的配置选项。格式是<规范的模块名>.<配置选项名>=<内容>。我们使用规范的模块名，这样任何的模块都可以创建配置选项来实现自身.
例如,如果我们想指定 REST API 端口,就向 property 文件中加入。
net.floodlightcontroller.restserver.RestApiServer.port = 8080

我们来分析一下 RestApiServer 的 init 方法.

```
// read our config options

Map<String, String> configOptions = context.getConfigParams(this);

String port = configOptions.get("port");

if (port != null) {

      restPort = Integer.parseInt(port);

}
```

注意 null 检查是必须的.如果配置选项没有提供，. FloodlightModuleLoader 模块将不会将其添加到文本中。

通过命令行,选项可以指定为 Java 属性. 这些可以重写 Floodlight 配置文件中任何指定的东西.

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

有两点要注意,第一 ,Java 属性在运行 floodlight.jar 之前 First 应被指定.之后,所有的被作为可执行的命令行文本传递给 Java. The second is that there are no spaces with the -D option.

注意事项

- 为了处理循环依赖关系,init()方法和 startup()方法的调用顺序是不确定的,因此,你不能假设任何的 init()和 startUp 方法的调用。.
- 你的配置文件不能调用 FLoodlight，properties 文件，这是 jar 包中的默认配置文件。.
- 每个模块必须有一个 0 参数(最好是空的)构造函数.做什么应该在构造函数中实现,而不是调用 init()。.
- .模块之间可能没有服务重叠,但是存在功能重叠,例如, LearningSwitch 模块 Forwarding 都有转发包的方法 Since they do not provide a common service w do not detect and overlap.

Javadoc entry

综述:

Overview 页面提供了所有的包的摘要,也包含了包集合的说明

包

每个包都有一个列出它的接口和类的页面,并都对每个类和接口有个简介.这个页面可能包含的:

- Interfaces (italic)
- Classes
- Enums
- Exceptions
- Errors
- Annotation Types

Class/Interface 类和接口

每个类,接口以及嵌套类和嵌套接口有自己的单独的页面.这些页面有三个部分,包括 a class/interface description, summary tables, and detailed member descriptions:

- Class inheritance diagram
- Direct Subclasses
- All Known Subinterfaces
- All Known Implementing Classes
- Class/interface declaration
- Class/interface description
- Nested Class Summary
- Field Summary
- Constructor Summary

- Method Summary
- Field Detail
- Constructor Detail
- Method Detail

每个摘要通过第一段来详细描述的内容.摘要条目按照字母顺序排列,而详细描述以出现在源代码的顺序. 这个保存由程序员建立的逻辑组.

Annotation Type 注释种类

每种注释有自己的页面,并有一下几个部分:

- Annotation Type declaration
- Annotation Type description
- Required Element Summary
- Optional Element Summary
- Element Detail

Enum 枚举

每个枚举有自己单独的页面,并有以下部分

- Enum declaration
- Enum description
- Enum Constant Summary
- Enum Constant Detail

Use

每个文件包,类和接口有自己的 Use 页面。这个页面描述，包，类，构造器和字段用的了哪些包和类。

Tree (Class Hierarchy)树（类的层次结构）

这儿有所有包的层次结构,也有每个包的层次结构.每个页面就是类和接口的列表.注意接口不是从 java.lang.Object 继承的

当查看概览页面,点击“树”显示所有包的层次结构.
当浏览一个特定的包,类或接口页面,点击“树”显示,包的层次结构.

不赞成使用的 API

Deprecated API 页面列出了所有不赞同使用的 API,之所以,不赞同使用,是为了优化,或者一个代替的 API 已经给出。

Index 索引

索引是一个字母列表，其中包含了所有类、接口、构造函数、方法和字段。

Prev/Next 上/下页

These links take you to the next or previous class, interface, package, or related page.这些链接指向下一页，其中包括相关的类,接口,包,或相关页面。

Frames/No Frames

These links show and hide the HTML frames. All pages are available with or without frames.

序列化格式

每个序列化类或者外部类都有一个描述它的字段和方法。这是 re-implementors 兴趣的，而不是使用 API 的开发者。虽然没有链接在导航栏中,你可以得到这个信息,通过定位任何序列化的类并单击“Serialized Form”。

固定字段值

固定字段值页面列出了静态的 final 字段和他们的值

This help file applies to API documentation generated using the standard doclet.

添加模块

创建一个监听模块

在 Eclipse 中添加类

- 在 floodlight 中找出“src/main/java”。
- 在“src/main/java” 目录下选择“New/Class”.
- 在 packet 中输入 “net.floodlightcontroller.mactracker”
- 在 name 中输入“MACTracker”
- 在“Interfaces”中, 单击“Add...” , “choose interface”增加“IOFMessageListener” and the “IFloodlightModule”, 单击“OK”.
- 单击“Finish”

产生的对应程序如下：

```
package net.floodlightcontroller.mactracker;

import java.util.Collection;
import java.util.Map;

import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFType;

import net.floodlightcontroller.core.FloodlightContext;
import net.floodlightcontroller.core.IOFMessageListener;
import net.floodlightcontroller.core.IOFSwitch;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;

public class MACTracker implements IOFMessageListener, IFloodlightModule {

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean isCallbackOrderingPrereq(OFType type, String name) {
        // TODO Auto-generated method stub
        return false;
    }
```

```
@Override
public boolean isCallbackOrderingPostreq(OFType type, String name) {
    // TODO Auto-generated method stub
    return false;
}

@Override
public Collection<Class<? extends IFloodlightService>> getModuleServices() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService> getServiceImpls() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public void init(FloodlightModuleContext context)
        throws FloodlightModuleException {
    // TODO Auto-generated method stub
```

```
    }

    @Override
    public void startUp(FloodlightModuleContext context) {
        // TODO Auto-generated method stub
    }

    @Override
    public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

设置模块化关系并初始化

开始前需处理一系列代码依赖关系。Eclipse 中可根据代码需要在编译过程中自动添加依赖包描述。没有相关工具，就需要手动添加代码如下：

```
import net.floodlightcontroller.core.IFloodlightProviderService;
import java.util.ArrayList;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.Set;
import net.floodlightcontroller.packet.Ethernet;
import org.openflow.util.HexString;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

至此，代码基本框架完成，进而要实现必要功能使模块能正确被加载。首先，注册 Java 类中需要的成员变量。由于要监听 openflow 消息，所以要向 FloodlightProvider 注册。同时需要一个集合变量 macAddresses 来存放控制器发现的 MAC 地址。最终，需要一个记录变量 logger 来输出发现过程中的记录信息。

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

编写模块加载代码。 通过完善 getModuleDependencies() 告知加载器在 floodlight 启动时将自己加载。

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

创建 Init 方法，Init（ ）将在控制器启动初期被调到，其主要功能室加载依赖关系并初始化数据结构。

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

处理 Packet-In 消息

在实现基本监听功能时，packet-in 消息需在 startup 方法中被记录和注册，同时确认新增模块需要依赖的其他模块已被正常初始化。

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

```
}
```

还需要为 OFMessage 监听者提供一个 ID，可通过调用 getName()实现。

```
@Override
public String getName() {
    return MACTracker.class.getSimpleName();
}
```

至此，与 packet-in 消息相关的操作完成。另外还需注意，要返回 Command.CONTINUE 以保证这个消息能够继续被 packet-in 消息处理。

```
@Override
  public net.floodlightcontroller.core.IListener.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;
   }
```

注册模块

如果要在 floodlight 启动时加载新增模块，需向加载器告知新增模块的存在，在 src/main/resources/META-INF/services/net.floodlight.core.module.IFloodlightModule 文件上增加一个符合规则的模块名，即打开该文件并在最后加上如下代码。

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

然后，修改 floodlight 的配置文件将 MACTracker 相关信息添加在文件最后。Floodlight 的缺省配置文件是 src/main/resources/floodlightdefault.properties。其中，floodlight.module 选项的各个模块用逗号隔开，相关信息如下：

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

最终，即可运行控制器并观察新增模块的功能

Mininet 虚拟网络连接 floodlight

如果在某主机的虚拟机中运行 mininet，同时 floodlight 也运行在该主机上，必须确保主机 IP 地址和 mininet 对应，下例中都设置为网关（192.168.110.2）

```
mininet@mininet:~$ sudo route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.110.0   0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.110.2   0.0.0.0         UG    0      0        0 eth0
```

```
mininet@mininet:~$ sudo mn --mac --controller=remote --ip=192.168.110.2 --port=6633
*** Loading ofdatapath
*** Adding controller
*** Creating network
*** Adding hosts:
h2 h3
*** Adding switches:
s1
*** Adding edges:
(s1, h2) (s1, h3)
*** Configuring hosts
h2 h3
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>pingall
```

Pingall 命令生成的 debug 信息都将从 MACTracker 发送到控制台

添加模块服务

简介

控制器由一个负责监听 openflow socket 并派发时间的核心模块，以及一些向核心模块注册用于处理响应事件的二级模块构成。当控制器启动时，可启用 debug log，进而看的这些二级模块的注册过程，示例如下：

```
17:29:23.231 [main] DEBUG n.f.core.internal.Controller - OFListeners for PACKET_IN:
devicemanager,

17:29:23.231 [main] DEBUG n.f.core.internal.Controller - OFListeners for PORT_STATUS:
devicemanager,

17:29:23.237 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.restserver.RestApiServer

17:29:23.237 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.forwarding.Forwarding

17:29:23.237 [main] DEBUG n.f.forwarding.Forwarding - Starting
net.floodlightcontroller.forwarding.Forwarding

17:29:23.237 [main] DEBUG n.f.core.internal.Controller - OFListeners for PACKET_IN:
devicemanager,forwarding,

17:29:23.237 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.storage.memory.MemoryStorageSource

17:29:23.240 [main] DEBUG n.f.restserver.RestApiServer - Adding REST API routable
net.floodlightcontroller.storage.web.StorageWebRoutable

17:29:23.242 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.core.OFMessageFilterManager

17:29:23.242 [main] DEBUG n.f.core.internal.Controller - OFListeners for PACKET_IN:
devicemanager,forwarding,messageFilterManager,

17:29:23.242 [main] DEBUG n.f.core.internal.Controller - OFListeners for PACKET_OUT:
messageFilterManager,

17:29:23.242 [main] DEBUG n.f.core.internal.Controller - OFListeners for FLOW_MOD:
messageFilterManager,

17:29:23.242 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.routing.dijkstra.RoutingImpl

17:29:23.247 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.core.CoreModule
```

```
17:29:23.248 [main] DEBUG n.f.core.internal.Controller - Doing controller internal setup

17:29:23.251 [main] INFO  n.f.core.internal.Controller - Connected to storage source

17:29:23.252 [main] DEBUG n.f.restserver.RestApiServer - Adding REST API routable
net.floodlightcontroller.core.web.CoreWebRoutable

17:29:23.252 [main] DEBUG n.f.c.module.FloodlightModuleLoader - Starting
net.floodlightcontroller.topology.internal.TopologyImpl

17:29:23.254 [main] DEBUG n.f.core.internal.Controller - OFListeners for PACKET_IN:
topology,devicemanager,forwarding,messageFilterManager,

17:29:23.254 [main] DEBUG n.f.core.internal.Controller - OFListeners for PORT_STATUS:
devicemanager,topology,
```

针对不同事件，对应不同类型的 openflow 消息生成，这些动作大部分与 packetin 有关，packet-in 是交换机没流表项能与数据包想匹配时，由交换机发给控制器的 openflow 消息，控制器进而出来数据包，用一组 flowmod 消息在交换机上部署流表项，下文示例增加一个新 packet-in 监听器用于存放 packet-in 消息，进而允许 rest API 获得这些消息。

创建类

在 Eclipse 中添加类

1 在 floodlight 项目中找到"src/main/java" 文件
2 在 "src/main/java"文件下选择 "New/Class."
3 在 packet 中输入"net.floodlightcontroller.pktinhistory"
4 在 name 中输入 "PktInHistory"。
5 在"Interfaces"中选择 choose "Add..."，单击“choose interface”增加"IFloodlightListener"和"IFloodlightModule"，然后单击“OK”
6 最后单击“finish”。

得到如下程序：

```
package net.floodlightcontroller.pktinhistory;


import java.util.Collection;

import java.util.Map;


import org.openflow.protocol.OFMessage;

import org.openflow.protocol.OFType;
```

```
import net.floodlightcontroller.core.FloodlightContext;
import net.floodlightcontroller.core.IOFMessageListener;
import net.floodlightcontroller.core.IOFSwitch;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;

public class PktInHistory implements IFloodlightModule, IOFMessageListener {

   @Override
   public String getName() {
      // TODO Auto-generated method stub
      return null;
   }

   @Override
   public boolean isCallbackOrderingPrereq(OFType type, String name) {
      // TODO Auto-generated method stub
      return false;
   }

   @Override
   public boolean isCallbackOrderingPostreq(OFType type, String name) {
      // TODO Auto-generated method stub
      return false;
   }

   @Override
```

```
public net.floodlightcontroller.core.IListener.Command receive(
        IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Collection<Class<? extends IFloodlightService>> getModuleServices() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService> getServiceImpls() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public void init(FloodlightModuleContext context)
        throws FloodlightModuleException {
    // TODO Auto-generated method stub
}
```

```
    @Override
    public void startUp(FloodlightModuleContext context) {
        // TODO Auto-generated method stub
    }
}
```

设置模块依赖关系

模块需要监听 openflow 消息，因此需要向 FloodlightProviderprotected 注册，需要增加依赖关系，创建成员变量如下：

```
IFloodlightProviderService floodlightProvider;
```

然后将新增模块与模块加载相关联，通过完善 getModuleDependencies() 告知模块加载器在 floodlight 启动时自己加载。

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

初始化内部变量

```
@Override
public void init(FloodlightModuleContext context) throws FloodlightModuleException {
    floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
}
```

处理 OpenFlow 消息

本部分实现对of packet_in消息的处理，利用一个buffer来存储近期收到的of消息，以备查询。

在 startup()中注册监听器，告诉 provider 我们希望处理 OF 的 PacketIn 消息。

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

为 OFMessage 监听器提供 id 信息，需调用 getName（）。

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

对 CallbackOrderingPrereq() 和 isCallbackOrderingPostreq()的调用，只需让它们返回 false，packetin 消息处理链的执行顺序并不重要。

作为类内部变量，创建 circular buffer（import 相关包），存储 packet in 消息。

```
protected ConcurrentCircularBuffer<SwitchMessagePair> buffer;
```

在初始化过程中初始化该变量。

```
@Override
public void init(FloodlightModuleContext context) throws FloodlightModuleException {
    floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
    buffer = new ConcurrentCircularBuffer<SwitchMessagePair>(SwitchMessagePair.class, 100);
}
```

最后实现模块接收到 packetin 消息时的处理动作。

```
@Override
public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
    switch(msg.getType()) {
        case PACKET_IN:
            buffer.add(new SwitchMessagePair(sw, msg));
```

```
            break;
        default:
            break;
    }
    return Command.CONTINUE;
}
```

每次 packetin 发生，其相应消息都会增加相关的交换机消息。该方法返回 Command.CONTINUE 以告知 IFloodlightProvider 能将 packetin 发给下一模块，若返回 Command.STOP，则指消息就停在该模块不继续被处理。

添加 rest API

在实现了一个完整的模块之后，我们可以实现一个rest api，来获取该模块的相关信息。需要完成两件事情：利用创建的模块导出一个服务，并把该服务绑到REST API模块。
具体说来，注册一个新的Restlet，包括
1. 在net.floodlightcontroller.controller.internal.Controller中注册一个restlet。

2. 实现一个*WebRoutable类。该类实现了RestletRoutable，并提供了getRestlet()和basePath()函数。

3. 实现一个*Resource类，该类扩展了ServerResource()，并实现了@Get或@Put函数。

下面具体来看该如何实现。
创建并绑定接口*IPktInHistoryService*
首先在pktinhistory包中创建一个从IFloodlightService扩展出来的接口IPktInHistoryService（IPktInHistoryService.java），该服务拥有一个方法getBuffer()，来读取circular buffer中的信息。

```
package net.floodlightcontroller.pktinhistory;
import net.floodlightcontroller.core.module.IFloodlightService;
import net.floodlightcontroller.core.types.SwitchMessagePair;
public interface IPktinHistoryService extends IFloodlightService {
public ConcurrentCircularBuffer<SwitchMessagePair> getBuffer();
}
```

现在回到原先创建的PktInHistory.java。相应类定义修订如下，让它具体实现IpktInHistoryService接口，

```
public class PktInHistory implements IFloodlightModule, IPktinHistoryService, IOFMessageListener {
```

并实现服务的getBuffer()方法。

```
@Override
public ConcurrentCircularBuffer<SwitchMessagePair> getBuffer() {
return buffer;
}
```

通过修改PktInHistory模块中getModuleServices()和getServiceImpls()方法通知模块系统，我们提供了IPktInHistoryService。

```
@Override
public Collection<Class<? extends IFloodlightService>> getModuleServices() {
        Collection<Class<? extends IFloodlightService>> l = new ArrayList<Class<? extends IFloodlightService>>();
        l.add(IPktinHistoryService.class);
        return l;
}
@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService> getServiceImpls() {
Map<Class<? extends IFloodlightService>, IFloodlightService> m = new HashMap<Class<? extends IFloodlightService>, IFloodlightService>();
m.put(IPktinHistoryService.class, this);
return m;
}
```

getServiceImpls()会告诉模块系统，本类（PktInHistory）是提供服务的类。
添加变量引用REST API服务
之后，需要添加REST API服务的引用（需要import相关包）。

```
protected IRestApiService restApi;
```

并添加IRestApiService作为依赖，这需要修改init()和getModuleDependencies()。

```
@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
Collection<Class<? extends IFloodlightService>> l = new ArrayList<Class<? extends IFloodlightService>>();
l.add(IFloodlightProviderService.class);
l.add(IRestApiService.class);
return l;
}
@Override
public void init(FloodlightModuleContext context) throws FloodlightModuleException {
floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
restApi = context.getServiceImpl(IRestApiService.class);
```

```
buffer = new ConcurrentCircularBuffer<SwitchMessagePair>(SwitchMessagePair.class, 100);
}
```

创建REST API相关的类PktInHistoryResource和PktInHistoryWebRoutable
现在创建用在REST API中的类，包括两部分，创建处理url call的类和注册到REST API的类。
首先创建处理REST API请求的类PktInHistoryResource（PktInHistoryResource.java）。当请求到达时，该类将返回circular buffer中的内容。

```
package net.floodlightcontroller.pktinhistory;

import java.util.ArrayList;
import java.util.List;
import net.floodlightcontroller.core.types.SwitchMessagePair;
import org.restlet.resource.Get;
import org.restlet.resource.ServerResource;

public class PktInHistoryResource extends ServerResource {
@Get("json")
public List<SwitchMessagePair> retrieve() {
IPktinHistoryService pihr =
(IPktinHistoryService)getContext().getAttributes().get(IPktinHistoryService.class.getCanonicalName());
List<SwitchMessagePair> l = new ArrayList<SwitchMessagePair>();
l.addAll(java.util.Arrays.asList(pihr.getBuffer().snapshot()));
return l;
}
}
```

现在创建PktInHistoryWebRoutable类（PktInHistoryWebRoutable.java），负责告诉REST API我们注册了API并将它的URL绑定到指定的资源上。

```
package net.floodlightcontroller.pktinhistory;
import org.restlet.Context;
import org.restlet.Restlet;
import org.restlet.routing.Router;
import net.floodlightcontroller.restserver.RestletRoutable;
public class PktInHistoryWebRoutable implements RestletRoutable {
@Override
public Restlet getRestlet(Context context) {
Router router = new Router(context);
router.attach("/history/json", PktInHistoryResource.class);
return router;
}
@Override
public String basePath() {
```

```
return "/wm/pktinhistory";
}
}
```

并将Restlet PktInHistoryWebRoutable注册到REST API服务，这通过修改PktInHistory类中的startUp()方法来完成。

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

自定义序列化类

数据会被Jackson序列化为REST格式。如果需要指定部分序列化，需要自己实现序列化类OFSwitchImplJSONSerializer（OFSwitchImplJSONSerializer.java，位于net.floodlightcontroller.web.serialzers包中），并添加到net.floodlightcontroller.web.serialzers包。

```
package net.floodlightcontroller.core.web.serializers;

import java.io.IOException;
import net.floodlightcontroller.core.internal.OFSwitchImpl;
import org.codehaus.jackson.JsonGenerator;
import org.codehaus.jackson.JsonProcessingException;
import org.codehaus.jackson.map.JsonSerializer;
import org.codehaus.jackson.map.SerializerProvider;
import org.openflow.util.HexString;

public class OFSwitchImplJSONSerializer extends JsonSerializer<OFSwitchImpl> {
/**
* Handles serialization for OFSwitchImpl
*/
@Override
public void serialize(OFSwitchImpl switchImpl, JsonGenerator jGen,
SerializerProvider arg2) throws IOException,
JsonProcessingException {
jGen.writeStartObject();
jGen.writeStringField("dpid", HexString.toHexString(switchImpl.getId()));
jGen.writeEndObject();
}
/**
* Tells SimpleModule that we are the serializer for OFSwitchImpl
*/
@Override
public Class<OFSwitchImpl> handledType() {
return OFSwitchImpl.class;
}
}
```

现在需要告诉Jackson使用我们的序列器。打开OFSwitchImpl.java（位于net.floodlightcontroller.core.internal包），修改如下（需要import 我们创建的OFSwitchImplJSONSerializer包）

```
@JsonSerialize(using=OFSwitchImplJSONSerializer.class)
public class OFSwitchImpl implements IOFSwitch {
```

至此，新建模块基本完成，还需告诉loader我们的模块存在，添加模块名字到src/main/resources/META-INF/services/net.floodlight.core.module.IfloodlightModule。

```
net.floodlightcontroller.pktinhistory.PktInHistory
```

然后告知模块需要被加载。修改模块配置文件src/main/resources/floodlightdefault.properties中的floodlight.modules变量。

```
floodlight.modules = net.floodlightcontroller.staticflowentry.StaticFlowEntryPusher,\
```

```
net.floodlightcontroller.forwarding.Forwarding,\
net.floodlightcontroller.pktinhistory.PktInHistory
```

启动mininet。

```
mn --controller=remote --ip=[Your IP Address] --mac --topo=tree,2
*** Adding controller
*** Creating network
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s5 s6 s7
*** Adding links:
(h1, s6) (h2, s6) (h3, s7) (h4, s7) (s5, s6) (s5, s7)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
*** Starting 3 switches
s5 s6 s7
*** Starting CLI:
启动后，运行
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (0/12 lost)
```

利用 REST URL 拿到结果

```
$ curl -s http://localhost:8080/wm/pktinhistory/history/json | python -mjson.tool

[

  {

    "message": {

      "bufferId": 256,

      "inPort": 2,

      "length": 96,

      "lengthU": 96,
```

```
        "packetData":
"MzP/Uk+PLoqIUk+Pht1gAAAAABg6/wAAAAAAAAAAAAAAAAAAAD/AgAAAAAAAAAAAH
/Uk+PhwAo2gAAAD+gAAAAAAACyKiP/+Uk+P",
        "reason": "NO_MATCH",
        "totalLength": 78,
        "type": "PACKET_IN",
        "version": 1,
        "xid": 0
      },
      "switch": {
        "dpid": "00:00:00:00:00:00:00:06"
      }
    },
    {
      "message": {
        "bufferId": 260,
        "inPort": 1,
        "length": 96,
        "lengthU": 96,
        "packetData":
"MzP/Uk+PLoqIUk+Pht1gAAAAABg6/wAAAAAAAAAAAAAAAAAAAD/AgAAAAAAAAAAAH
/Uk+PhwAo2gAAAD+gAAAAAAACyKiP/+Uk+P",
        "reason": "NO_MATCH",
        "totalLength": 78,
        "type": "PACKET_IN",
        "version": 1,
        "xid": 0
      },
      "switch": {
        "dpid": "00:00:00:00:00:00:00:05"
```

```
    }
  },
etc....etc....
```

Floodlight rest API 开发

利用 rest 接口编写应用不限于编程语言，基本开发步骤如下：

1、确定应用所需的网络服务信息
2、从 Floodlight REST API 列表中选择满足服务需求的 REST API
    1 若发现合适的接口，根据 rest API 语法，输入参数和必要的选型信息。
    2 若没有现成的接口，相关服务和信息已能够有 floodlight 提供，但未封装为接口，则需自主开发相关接口。
    3 若 floodlight 当前也不能实现相关服务，则需要基于 Java 自主开发控制器模块或应用模块
3、利用所有可以用的 rest API 调用，设计，实现，测试应用

在 floodlight/apps 目录下，有一个用 Python 编写的 Circuit Pusher 应用实例，用于在同一 openflow 集群中的两个具有 IP 地址 A 和 B 的主机之间创建一个静态单路径链路。其开发过程按上述步骤执行。

1、确定应用所需的网络服务信息

    4 主机 A 和 B 的接触点，即可表示主机物理位置的数据信息
    5 主机 A 和 B 对应接触点之间的路由
    6 用在 A 到 B 路由上所有交换机上部署流量信息服务

2、选择能够满足服务需求的 REST API
    7 /wm/device/ 发现每个设备的接触点信息
    8 /wm/topology/route/<switchIdA>/<portA>/<switchIdB>/<portB>/json （参数为 get ）发现 A 和 B 之间的路由
    9 /wm/staticflowentrypusher/json （参数为 post ）在每个交换机上部署流表项
3、应用设计: (source)
    10 选择 python 作为编程语言
    11 使用 os.popen 发送 curl 命令用于调用 REST API
    12 使用/wm/device 语法，并解析主机 A 和 B 接触点的交换机端口信息
    13 使用/vm/topology/route 返回交换机端口对用于形成流表项
    14 使用/wm/staticflowentrypusher/json，针对每个交换机端口对，在交换机 X 上部署流表项:
        1 ether-type '0x0800', port M → N
        2 ether-type '0x0806', port M → N

3 ether-type '0x0800', port N → M
4 ether-type '0x0806', port N → M
15 如果应用工作正常（如 A 与 B 能 ping），则完成应用实现和测试
16 为了应用更具真实性，可增加两个附属功能：
1 删除流，因为示例采用的是"静态"设置，因此流表项不会超时失效
2 记录已被推送的流，可通过普通文本文件保存流信息

Floodlight-Test

Floodlight-Test 是一个测试执行框架，与 floodlight 共同发布 ，供开发者实行 floodlight 一体化测试，以及各种扩展开发。

Floodlight-Test 允许如下开发:

- 实例化一个或多个有 mininet 的虚拟机
- 在开发者主机（如在 Eclipse 上）或虚拟机上运行 floodlight
- 运行一组提供的一体化基础测试
- 为所有新扩展的部分添加新的一体化测试

Floodlight-Test 用于保证 floodlight 和其所有扩展部分的高质量，Flood-Test 支持并帮助开发者在他们设计的过程中遵守正确的测试原则，此外，它还是社区贡献给 floodlight 资源库的质量标准。

起初，Floodlight 作为一个开源控制器，去建立 openflow 应用和/或控制特征。后来，在开源社区的贡献下成长为一个平稳的控制平台。Openbench 将会提供测试工具和过程保证 floodlight 的健全成长。

系统需求

7 VirtualBox v4.1.14 或更新(较新的版本可能有效但是未测试过)
8 开始安装时网络连通性
9 Floodlight vmdk

安装步骤

1.在主机中，下载 floodlight-vm.zip， http://www.projectfloodlight.org/download/;解压到你指定的工作目录 ， say ~/work

2. 在主机中，获取 VM 安装脚本：

```
1 git clone https://github.com/floodlight/floodlight-test
2 scripts are under floodlight-test/scripts
```

3.两种方案： a: 重命名 VM 压缩文件，改为 onetime-create-vm.sh 中给定 的默认名(如 floodlightcontroller-test)； b：编译 onetime-create-vm.sh 中的文件名，与 VM 文件相匹配(如., floodlightcontroller-[release date]).

4. 在主机中, 运行 onetime-create-vm.sh; 在 VirtualBox 图形界面上, 点击 "Network" 和 "OK",然后点击启动虚拟机， 登录 (username: floodlight, no password), 运行 'ifconfig' 确认并记录 eth0 的 IP 地址。

5. 在主机中, 用 VM IP 编译 onetime-setup-vm.sh 和 setup-bench.sh ; 运行 onetime-setup-vm.sh ，将会进入 VM (consolve-vm) 并且安装 Floodlight-Test 。

运行测试

每次你想进行测试，你需要打开所有虚拟机并且做如下步骤：

1. 如果需要则更新 floodlight.jar (以及 floodlight.properties) :

3 倘若你还未改变 floodlight 代码 (i.e., floodlight.jar is up-to-date on your test VMs),你可以简单的打开这三个 虚拟机(一个控制机，两个测试机)

4 如果你需要更新 floodlight.jar, 提供一个简便的方法，在 update-floodlight.sh 中更新路径为 floodlight 源文件根目录；更新 VM IP ；运行 update-floodlight.sh 。

2. 在 "console" VM, 'cd floodlight-test' 然后 ‘source setup-python-env.sh’

3. 在 "console" VM, 'bm clean' ，清空之前运行时旧的 VM 状态.

4. Edit build/Makefile.workspace to confirm/edit VM IP addresses under make target 'register-vms-floodlight'

5. 在 "console" VM, 'bm register-vms-floodlight'

6. 在 "console" VM, 'bm check-vms-floodlight'; see failed-check-vms-floodlight file for failed tests, if any

7. 在"console" VM, 'bm check-tests-floodlight'; see failed-check-tests file for failed tests, if any

8.在 "console" VM, 'bigtest/[test-dir]/[test.py]' to run individual failed tests directly to diagnose cause of failure

有效建议:

1.一开始安装就为 "tester VMs" 拍快照 ，点击一个 VM, 点击右上端的 Snapshots 然后点击“add ”添加快照。例如，运行完 check-tests-floodlight 后你想恢复 画面到默认模式。

2.使用 ssh 客户端可以查看更多历史

3. 很多安装错误都是由于网络错误，如以下典型错误：

- 在网桥模式下配置安装脚本: ifconfig 确保拥有有效地址，若没有，可以在 DHCP 服务器中 设置 IP，也可以 点击 VirtualBox VM's GUI Network tab，若都不行，分配静态 IP，'ifconfig eth0 xx.xx.xx.xx 255.255.255.0'
- 安装后，VirtualBox menu bar > VirtualBox > Preferences > Network > Add a host-only Network，如果没有 (vboxnet0). 点击 VM's Network，设置为 host-only Adapter/vboxnet0.

合并 floodlight 扩展部分的要求

Floodlight 严格执行质保联系，floodlight 中所有模块既要单独测试又要整体测试.。

1. JUnit unit tests. Code coverage threshold, eclipse, bm check
2. OpenBench integration tests
3. Floodlight committer tests and code review

添加新的一体化测试

用 Python 添加一个一体化测试的过程很明确，怎样创建测试环境，怎样快速添加自己的测试命令。

思考下面的例子:

1. bigtest/firewall/FloodlightFirewallTest.py

```
#!/usr/bin/env python

## Creates a tree,4 topology to test different firewall rules

## with ping and iperf (TCP/UDP, differed ports)
## @author KC Wang


# import a number of basic bigtest libraries

import bigtest.controller

import bigtest


# import a number of useful python utilities.

# This particular example does REST API based testing, hence urllib is useful for sending REST commands and

# json is used for parsing responses

import json

import urllib

import time

from util import *

import httplib


# bigtest function to connect to two active tester VMs

# make sure you already started the VM and have done bm register-vms-floodlight

# (with the correct two nodes indicated in build/Makefile.workspace)

env = bigtest.controller.TwoNodeTest()
```

```
log = bigtest.log.info


# use the first tester VM's floodlight controller
# since its a linux node, we use its bash mode as command line interface
controllerNode = env.node1()
controllerCli = controllerNode.cli()
controllerIp = controllerNode.ipAddress()
controllerCli.gotoBashMode()
controllerCli.runCmd("uptime")


# use the second tester VM to run mininet
mininetNode = env.node2()
mininetCli = mininetNode.cli()
# this starts mininet from linux console and enters mininet's command line interface
mininetCli.gotoMininetMode("--controller=remote --ip=%s --mac --topo=tree,4" %
controllerIp)


# this function uses REST interface to keep on querying floodlight until the specified
switches are all
# connected to the controller correctly and seeing each other in the same connected
cluster
switches = ["00:00:00:00:00:00:00:1%c" % x for x in ['1', '2', '3', '4', '5', '6', '7', '8', '9', 'a',
'b', 'c', 'd', 'e', 'f']]
controllerNode.waitForSwitchCluster(switches)
```

现在你已经添加了一些用于不同情况的测试命令，确保 floodlight 正常工作。

```
....
# issuing a mininet command
# pingall should succeed since firewall disabled
x = mininetCli.runCmd("pingall")
```

```
# return is stored in x and the bigtest.Assert method can check for a specific string in the response
bigtest.Assert("Results: 0%" in x)

# you can use python's sleep to time out previous flows in switches
time.sleep(5)

# Sending a REST API command
command = "http://%s:8080/wm/firewall/module/enable/json" % controllerIp
x = urllib.urlopen(command).read()
bigtest.Assert("running" in x)

...

# clean up all rules - testing delete rule
# first, retrieve all rule ids from GET rules
command = "http://%s:8080/wm/firewall/rules/json" % controllerIp
x = urllib.urlopen(command).read()
parsedResult = json.loads(x)

for i in range(len(parsedResult)):
    # example sending a REST DELETE command.  Post can be used as well.
    params = "{\"ruleid\":\"%s\"}" % parsedResult[i]['ruleid']
    command = "/wm/firewall/rules/json"
    url = "%s:8080" % controllerIp
    connection =  httplib.HTTPConnection(url)
    connection.request("DELETE", command, params)
```

```
    x = connection.getresponse().read()
    bigtest.Assert("Rule deleted" in x)


...


# iperf TCP works, UDP doesn't
mininetCli.runCmd("h3 iperf -s &")
x = mininetCli.runCmd("h7 iperf -c h3 -t 2")
# bigtest.Assert can also test for a "not" case
bigtest.Assert(not "connect failed" in x)
```

2. bigtest/forwarding/IslandTest1.py

这个例子展示怎样定义随机的拓扑结构，主机互联，交换机可按选择被不同控制器侦听。这在 OF 岛与 non-OF 岛互联的拓扑中有用，因为控制器 B 控制的岛在控制器 A 看来是 non-OF 的。

```
import bigtest
from mininet.net import Mininet
from mininet.node import UserSwitch, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel
import bigtest.controller
from bigtest.util.context import NetContext, EnvContext


def addHost(net, N):
    name= 'h%d' % N
    ip = '10.0.0.%d' % N
    return net.addHost(name, ip=ip)
```

```
def MultiControllerNet(c1ip, c2ip):
    "Create a network with multiple controllers."


    net = Mininet(controller=RemoteController, switch=UserSwitch)


    print "Creating controllers"
    c1 = net.addController(name = 'RemoteFloodlight1', controller = RemoteController,
defaultIP=c1ip)
    c2 = net.addController(name = 'RemoteFloodlight2', controller = RemoteController,
defaultIP=c2ip)


    print "*** Creating switches"
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )
    s4 = net.addSwitch( 's4' )


    print "*** Creating hosts"
    hosts1 = [ addHost( net, n ) for n in 3, 4 ]
    hosts2 = [ addHost( net, n ) for n in 5, 6 ]
    hosts3 = [ addHost( net, n ) for n in 7, 8 ]
    hosts4 = [ addHost( net, n ) for n in 9, 10 ]


    print "*** Creating links"
    for h in hosts1:
        s1.linkTo( h )
    for h in hosts2:
        s2.linkTo( h )
    for h in hosts3:
        s3.linkTo( h )
```

```
    for h in hosts4:
        s4.linkTo( h )

    s1.linkTo( s2 )
    s2.linkTo( s3 )
    s4.linkTo( s2 )

    print "*** Building network"
    net.build()

    # In theory this doesn't do anything
    c1.start()
    c2.start()

    #print "*** Starting Switches"
    s1.start( [c1] )
    s2.start( [c2] )
    s3.start( [c1] )
    s4.start( [c1] )

    return net


with EnvContext(bigtest.controller.TwoNodeTest()) as env:
  log = bigtest.log.info

  controller1 = env.node1()
  cli1 = controller1.cli()
```

```
  controller2 = env.node2()
  cli2 = controller2.cli()


  print "ip1:%s ip2:%s" % (controller1.ipAddress(), controller2.ipAddress())


  with NetContext(MultiControllerNet(controller1.ipAddress(), controller2.ipAddress()))
as net:
    sleep(20)
    ## net.pingAll() returns percentage drop so the bigtest.Assert(is to make sure 0%
dropped)
    o = net.pingAll()
      bigtest.Assert(o == 0)
```

Unit 测试

简介

Floodlight 采用 Junit 框架和 EasyMock 进行单元测试。你可以运行所有 Junit 测试并且用 ant 检查单元测试的范围，命令如下：

```
# runs the unit tests with coverage
ant coverage


browse unit test reports at floodlight/target/coverage/index.html
```

开发新代码就得添加相应的单元测试，如果代码中涉及已存在的类，即使没有改变类，需要为这个类扩大单元测试范围。

例子

仔细阅读已有的单元测试，可以从 net.floodlightcontroller.forwarding/ForwardingTest.java 或者 net.floodlightcontroller.devicemanager.internal/DeviceManagerImplTest.java 开始

写一个 EasyMock 测试:

1.清楚要 "mocking" 什么，要测试什么；

记住你要测试的仅仅是代码，代码可能调用其他类的方法，但这是是那个类需要“mocked”的。在 ForwardingTest.java,在 src/test/java/*.test packages 中可以找到很多“mock class”，当你需要模拟典型的 floodlight 服务时需要依靠这些类。同时会多次调用 createMock(*.class)，这是模拟类示例的基本组成。如果不知道选择哪种方法就参考已有的。

2. 使用模拟接口方法就得进行声明，浏览代码记录下什么时候调用什么方法，打算用什么数测试，算出应该返回的正确结果。这些值应该包括正常情况以及边界情况，并且可以进行多个分开的单元测试，每次测试某一组值。

3. createMock(), reset(), expect(), replay(), verify()

总而言之，用先用 createMock（）创建模拟对象，再用 reset()清空对模拟对象的期望，之后用 replay（）进入准备状态，准备运行要被测试的代码，最后用 verify（）验证调用的方法使用情况是否与预期结果一致。任何一点不符都会出错。

4.测试范围

‘ant coverage’分析了部分代码，即使是测试结果是 100%也不代表代码完全正确。正确性取决于测试案例的范围，用常见值与边界值测试代码各种情况。

关于 Junit 和 EasyMock 的阅读

Junit Tutorial (1)
Junit Tutorial (2)
Unit testing with JUnit and EasyMock
Mock controls with EasyMock
Using captures with EasyMock
EasyMock README

控制器基准配置

基准配置

更新 floodlight properties 文件

编译 src/main/resources/floodlightdefault.properties 如下：

```
floodlight.modules =
```

```
net.floodlightcontroller.learningswitch.LearningSwitch,net.floodlightcontroller.counter.
NullCounterStore,net.floodlightcontroller.perfmon.NullPktInProcessingTime
```

创建 Floodlight

Floodlight propertie 文件缓存在 Floodlight jar.中，可以解压 Floodlight 活着在 floodlight.sh 根目录中添加以下命令 "-cf floodlightdefault.properties"

```
.$ ant
```

运行 Floodlight

运行 floodlight.sh 。

注意

根据 MAC 地址的数量进一步调整内存性能。

Cbench (New)

用来测试 openflow 控制器。Cbench 可以仿真一组连接控制器的交换机，发送 packet-in 消息，可用于测试细微变化产生的影响。

安装 cbench

参考 http://www.openflow.org/wk/images/3/3e/Manual.pdf Chapter 2 中详细的安装指导。

Under debian/ubuntu Linux:

```
$ sudo apt-get install autoconf automake libtool libsnmp-dev libpcap-dev

$ git clone git://gitosis.stanford.edu/oflops.git

$ cd oflops; git submodule init && git submodule update

$ git clone git://gitosis.stanford.edu/openflow.git

$ cd openflow; git checkout -b release/1.0.0 remotes/origin/release/1.0.0

$ wget http://hyperrealm.com/libconfig/libconfig-1.4.9.tar.gz

$ tar -xvzf libconfig-1.4.9.tar.gz

$ cd libconfig-1.4.9

$ ./configure
```

```
$ sudo make && sudo make install

$ cd ../../netfpga-packet-generator-c-library/

$ sudo ./autogen.sh && sudo ./configure && sudo make

$ cd ..

$ sh ./boot.sh ; ./configure --with-openflow-src-dir=<absolute path to openflow branch>; make

$ sudo make install

$ cd cbench
```

此时可运行 cbench

运行 cbench

Cbench 有一系列参数. 我们所需的有:

M	number of MACs / hosts to emulate per switch
s	number of switches to emulate
t	throughput (vs. latency mode)

例子:

```
./cbench -c localhost -p 6633 -m 10000 -l 10 -s 16 -M 1000 -t
```

怎样用 floodlight 满足服务质量

简介

- Openflow1.0 协议中有设置网络服务类型的方法，就像匹配流的包在某个端口进入某个队列。给使用者提供简单的方法将 Qos 状态压入交换机。协议 1.3 将进一步改进， 依旧支持 DSCP 或 ToS 位，并构建深层次的 Qos 框架，OFconfig 还是有重要作用，这样的协议对 Qos 队列的建立和拆除很有用。
- 以下示例就是将限速 Qos 状态压入 OVswitches 中

.https://groups.google.com/a/openflowhub.org/forum/#!msg/floodlight-dev/y5yJRTcfS48/418QH9zLMKoJ|https://groups.google.com/a/openflowhub.org/forum/#!msg/floodlight-dev/y5yJRTcfS48/418QH9zLMKoJ

QoS 应用

REST API	GET	POST	Description of use

/wm/qos/tool/<op>/json	Yes	N/A	用 "status", "enable" or "disable" status - 模块状态，有效为 true,否则为 false enable -服务质量模块有效 disable - 服务质量模块无效
/wm/qos/service/json	Yes	Yes	接受加到控制器的服务，该服务可应用于某一级别的网络服务（交换机必须支持该操作） 返回一组控制器熟识的服务
/wm/qos/policy/json	Yes	Yes	接受一项规则，该规则可应用于一个或多个交换机，可以设置“none”使控制器保存规则，但不可以将规则压入交换机，使用 Qospath 应用时需用到该技术 返回一组控制器熟识的规则

服务

Field	Description
id	唯一的身份 (not needed in POST)
name	唯一的，便于理解的，如 "Best Effort"
tos	代表 service 或 DSCP 的类型位(Differentiated Services Code Point)

规则

Field	Description
id	唯一的身份 (not needed in POST)
name	唯一的，便于理解的，如 "Enqueue 1:1 on switch 1"
sw	10 有以下几种取值. "none" ：交换机没有什么规则，但控制器可保存，在 QoSPath 中使用 11 "all"：，所有交换机，主要用于 Qos 服务类型 12 "<Switch-ID>"：将规则压入某个交换机 dpid 或一组 dpids
service	参考 Tos 服务级别，tos=0x00.将会设置规则为 _set-nw-tos. _Default 行动

	服务与入队必须使用，一般只需一个即可
queue	在入队方法中使用，队列不能没有入队
enqueue-port	入队行动并且队列连接该端口 服务与入队必须使用，一般只需一个即可
priority	规则的优先级 (flow) default is 32767 极限值是 32767
Policy Match Fields	same from the staticflowpusher
protocol	可以是十六进制（以 0x 开头）或十进制
eth-type	可以是十六进制（以 0x 开头）或十进制
ingress-port	交换机接收包的端口 可以是十六进制（以 0x 开头）或十进制
ip-src	xx.xx.xx.xx
ip-dst	xx.xx.xx.xx
tos	可以是十六进制（以 0x 开头）或十进制
vlan-id	可以是十六进制（以 0x 开头）或十进制
eth-src	可以是十六进制（以 0x 开头）或十进制
eth-dst	可以是十六进制（以 0x 开头）或十进制
src-port	可以是十六进制（以 0x 开头）或十进制
dst-port	可以是十六进制（以 0x 开头）或十进制
*"wildcards" and "active" keys are unsupported right now.	

REST 应用

QoSPusher.py Python 应用用来管理 QoS

QoSPath.py QoSPath 是一个 python 应用，用 cirtcuitpusher.py 将 QoS 状态压入某网络的环路。

例子

Network

Mininet Topo Used

#sudo mn --topo linear,4 --switch ovsk --controller=remote,ip= --ipbase=10.0.0.0/8