

ODL碳版本模块开发及流程梳理

[...](#) • 17-09-13 • 3710 人围观

作者简介：徐军，北京邮电大学网络技术研究院，研二在读，sdn应用开发。

文章主要基于ODL碳版本，进行简单插件的构建、安装、部署，以一个插件开发为例，介绍ODL新版本开发过程中的一些具体问题。



一、碳版本简易开发流程

1.1 开发环境搭建

1. 安装java1.8以上环境，安装maven。

2. 配置maven settings.xml 。

首先在odl的git中访问odl-parent项目，进入项目可以看到settings.xml, 把这个项目拷贝到自己maven的.m2文件夹下。（一定要注意自己拷贝的版本和要开发的版本必须保持一致）

在使用maven-archetype-plugin:3.0.1插件创建项目时与之前版本略有不同，需要在setting.xml文件中添加odlarchtype标签。具体的配置信息如下。

Shell

```
<profile>
  <id>odlarchtype</id>
  <repositories>
    <repository>
      <id>archetype</id>
      <url>http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/</url>
      <releases>
        <enabled>true</enabled>
        <checksumPolicy>fail</checksumPolicy>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <checksumPolicy>warn</checksumPolicy>
      </snapshots>
    </repository>
  </repositories>
</profile>
```

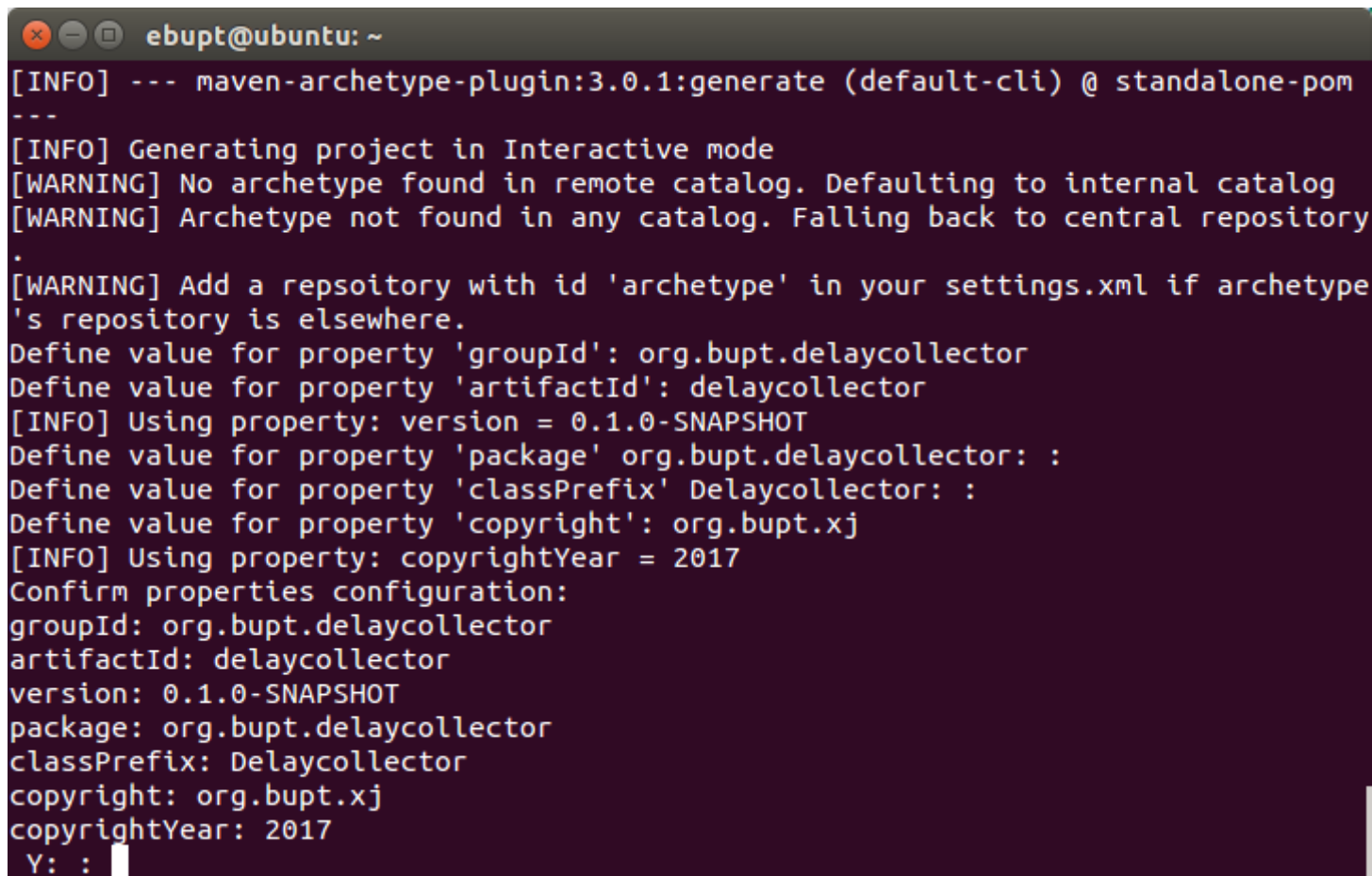
到此基本的开发环境配置完毕，由于odl的maven仓库在国外，可能需要自己在maven的settings.xml文件中自行配置代理。

1.2 项目框架建立

打开终端，输入指令：

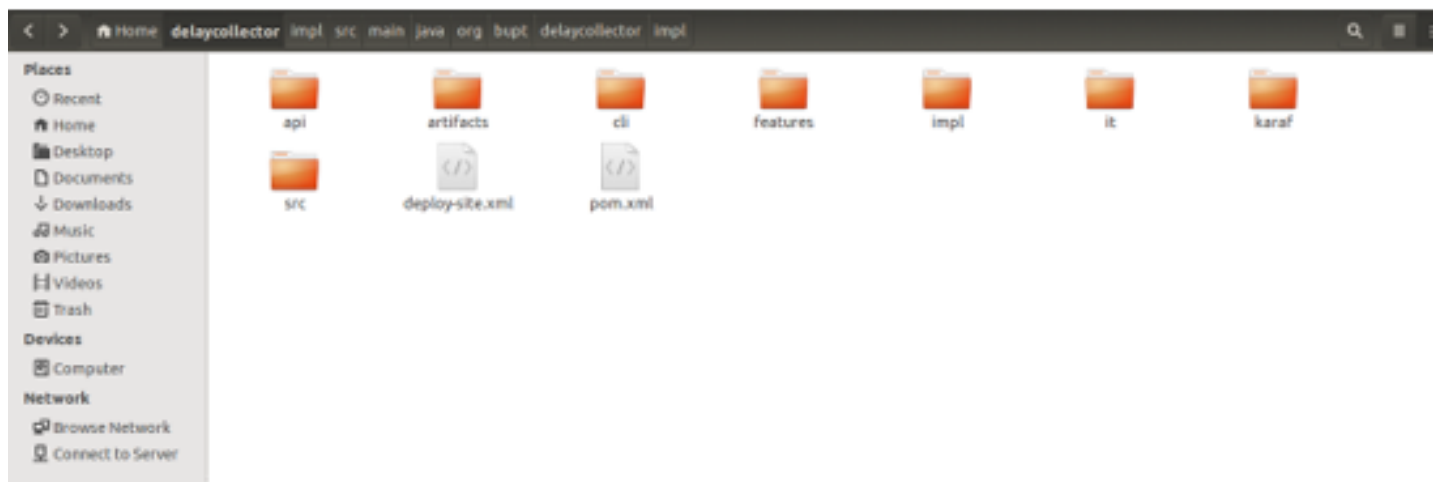
Shell

注：1.3.0-Carbon表示项目的版本号

A terminal window titled 'ebupt@ubuntu: ~' showing the execution of the Maven command 'maven-archetype-plugin:3.0.1:generate (default-cli) @ standalone-pom'. The output includes several informational and warning messages, followed by a series of prompts to define project properties. The user has entered the following values: groupId: org.bupt.delaycollector, artifactId: delaycollector, version: 0.1.0-SNAPSHOT, package: org.bupt.delaycollector, classPrefix: Delaycollector, copyright: org.bupt.xj, and copyrightYear: 2017. The terminal ends with a confirmation prompt 'Y: : ' followed by a cursor.

```
ebupt@ubuntu: ~  
[INFO] --- maven-archetype-plugin:3.0.1:generate (default-cli) @ standalone-pom  
---  
[INFO] Generating project in Interactive mode  
[WARNING] No archetype found in remote catalog. Defaulting to internal catalog  
[WARNING] Archetype not found in any catalog. Falling back to central repository  
.  
[WARNING] Add a repository with id 'archetype' in your settings.xml if archetype  
's repository is elsewhere.  
Define value for property 'groupId': org.bupt.delaycollector  
Define value for property 'artifactId': delaycollector  
[INFO] Using property: version = 0.1.0-SNAPSHOT  
Define value for property 'package' org.bupt.delaycollector: :  
Define value for property 'classPrefix' Delaycollector: :  
Define value for property 'copyright': org.bupt.xj  
[INFO] Using property: copyrightYear = 2017  
Confirm properties configuration:  
groupId: org.bupt.delaycollector  
artifactId: delaycollector  
version: 0.1.0-SNAPSHOT  
package: org.bupt.delaycollector  
classPrefix: Delaycollector  
copyright: org.bupt.xj  
copyrightYear: 2017  
Y: : 
```

输入建立项目的基本信息，生成如下项目骨架：



我们需要关注的主要是api和impl两个文件夹，其中api中存放我们项目要对外提供的webapi，impl中是项目的具体实现。打开impl，可以看到里边已经生成了DelaycollectorProvider.java文件，该文件是整个项目的入口。

在代码的impl/src/main/resources文件夹下生成了impl-blueprint.xml文件，用于为项目提供依赖注入。

（具体内容可参考https://wiki.opendaylight.org/view/Using_Blueprint）

我们可以结合上边DelaycollectorProvider.java文件的代码大致的分析一下该文件的具体含义，其中标签后边的内容正好与上边的Provider代码相互对应，指明了函数的入口。其中init-method对应初始化方法，也就是插件install之后会首先运行这个方法。而destory-method方法是项目卸载时运行的方法，可以把一些资源的清理放在该方法中。

标签与下面的标签对应，为插件的构造方法提供参数注入。

在odl的开发过程中，以上两个文件尤为重要。

到目前为止，虽然我们一行代码都没写，但它已经是一个完整的插件了。在项目目录下运行如下命令对项目进行编译：

Shell

1

```
mvn clean install -DskipTests
```

```
ebupt@ubuntu: ~/delaycollector
tor-aggregator ---
[INFO] Attaching 'src/site/site.xml' site descriptor with classifier 'site'.
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-api SUCCESS [ 24.083 s]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-impl SUCCESS [ 5.811 s]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-cli SUCCESS [ 4.326 s]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-features SUCCESS [ 10.07
9 s]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-karaf SUCCESS [01:03 min
]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-artifacts SUCCESS [ 1.4
37 s]
[INFO] ODL :: org.bupt.delaycollector :: delaycollector-it SUCCESS [ 30.228 s]
[INFO] delaycollector ..... SUCCESS [ 16.467 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:40 min
[INFO] Finished at: 2017-08-28T05:10:15-07:00
[INFO] Final Memory: 219M/671M
[INFO] -----
ebupt@ubuntu:~/delaycollector$
```

进入/delaycollector/karaf/target/assembly/bin\$目录下，运行./karaf，但是并没有发生什么，因为我们的项目现在只是一个空壳，没有编写任何代码。Init方法中只是在日志中打印了一句话。打开/delaycollector/karaf/target/assembly/data/log\$文件夹下的log文件。

```

karaf.log (~/.delaycollector/karaf/target/assembly/data/log) - gedit
2017-08-28 05:13:29,143 | INFO | ntAdminThread #7 | BlueprintBundleTracker | 173 - org.opendaylight.controller.blueprint - 0.6.0.Carbon |
Creating blueprint container for bundle org.opendaylight.aaa.encrypt-service_0.5.0.Carbon [244] with paths [bundleentry://244.fwk1164371389/org/
opendaylight/blueprint/encryptservice.xml]
2017-08-28 05:13:29,156 | INFO | Event Dispatcher | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.encrypt-service/0.5.0.Carbon is waiting for dependencies [(type=default)(!(type=*))
(objectClass=org.opendaylight.md.sal.binding.dom.codec.api.BindingNormalizedNodeSerializer))]
2017-08-28 05:13:29,167 | INFO | Event Dispatcher | BlueprintBundleTracker | 173 - org.opendaylight.controller.blueprint - 0.6.0.Carbon |
Creating blueprint container for bundle org.opendaylight.aaa.cert_0.5.0.Carbon [247] with paths [bundleentry://247.fwk1164371389/org/opendaylight/
blueprint/aaaCert.xml]
2017-08-28 05:13:29,176 | INFO | Event Dispatcher | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.cert/0.5.0.Carbon is waiting for dependencies [(objectClass=org.opendaylight.controller.md.sal.binding.api.DataBroker),
(objectClass=org.opendaylight.controller.sal.binding.api.RpcProviderRegistry), (objectClass=org.opendaylight.aaa.encrypt.AAAEncryptionService), (&(|
(type=default)(!(type=*)))(objectClass=org.opendaylight.md.sal.binding.dom.codec.api.BindingNormalizedNodeSerializer))]
2017-08-28 05:13:29,238 | INFO | Event Dispatcher | BlueprintBundleTracker | 173 - org.opendaylight.controller.blueprint - 0.6.0.Carbon |
Creating blueprint container for bundle org.opendaylight.aaa.shiro_0.5.0.Carbon [258] with paths [bundleentry://258.fwk1164371389/org/opendaylight/
blueprint/impl-blueprint.xml]
2017-08-28 05:13:29,241 | INFO | Event Dispatcher | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.shiro/0.5.0.Carbon is waiting for dependencies [(objectClass=org.opendaylight.controller.md.sal.binding.api.DataBroker),
(objectClass=org.opendaylight.aaa.cert.api.ICertificateManager)]
2017-08-28 05:13:29,321 | INFO | config-pusher | HydrogenDataBrokerAdapter | 166 - org.opendaylight.controller.sal-binding-broker-impl -
1.5.0.Carbon | ForwardedBackwardsCompatibleBroker started.
2017-08-28 05:13:29,340 | INFO | rint Extender: 2 | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.cert/0.5.0.Carbon is waiting for dependencies [(objectClass=org.opendaylight.controller.sal.binding.api.RpcProviderRegistry),
(objectClass=org.opendaylight.aaa.encrypt.AAAEncryptionService), (&(|(type=default)(!(type=*))
(objectClass=org.opendaylight.md.sal.binding.dom.codec.api.BindingNormalizedNodeSerializer))]
2017-08-28 05:13:29,346 | INFO | config-pusher | RootBindingAwareBroker | 166 - org.opendaylight.controller.sal-binding-broker-impl -
1.5.0.Carbon | Starting Binding Aware Broker: binding-broker-impl
2017-08-28 05:13:29,358 | INFO | rint Extender: 3 | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.shiro/0.5.0.Carbon is waiting for dependencies [(objectClass=org.opendaylight.aaa.cert.api.ICertificateManager)]
2017-08-28 05:13:29,378 | INFO | rint Extender: 2 | DelaycollectorProvider | 208 - org.bupt.delaycollector.impl - 0.1.0.SNAPSHOT |
DelaycollectorProvider Session Initiated
2017-08-28 05:13:29,401 | INFO | ntAdminThread #8 | BlueprintBundleTracker | 173 - org.opendaylight.controller.blueprint - 0.6.0.Carbon |
Blueprint container for bundle org.bupt.delaycollector.impl_0.1.0.SNAPSHOT [208] was successfully created
2017-08-28 05:13:29,409 | INFO | config-pusher | ConfigPusherImpl | 131 - org.opendaylight.controller.config-persister-impl -
0.6.0.Carbon | Successfully pushed configuration snapshot 01-md-sal.xml(odl-restconf-noauth,odl-restconf-noauth)
2017-08-28 05:13:29,409 | INFO | config-pusher | ConfigPusherImpl | 131 - org.opendaylight.controller.config-persister-impl -
0.6.0.Carbon | Pushing configuration snapshot 05-clustering.xml(odl-restconf-noauth,odl-restconf-noauth)
2017-08-28 05:13:29,477 | INFO | rint Extender: 2 | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.controller.sal-cluster-admin-impl/1.5.0.Carbon is waiting for dependencies
[(objectClass=org.opendaylight.controller.sal.binding.api.RpcProviderRegistry)]
2017-08-28 05:13:29,610 | INFO | rint Extender: 3 | BlueprintContainerImpl | 15 - org.apache.aries.blueprint.core - 1.6.1 | Bundle
org.opendaylight.aaa.cert/0.5.0.Carbon is waiting for dependencies [(objectClass=org.opendaylight.aaa.encrypt.AAAEncryptionService), Initial app
Plain Text ▾ Tab Width: 8 ▾ Ln 440, Col 1 INS

```

可以看到确实在日志中打印了字符串。如果觉得效果还不明显，那么我们可以在eclipse中对代码进行远程调试。

1.3 代码调试

首先将项目导入eclipse，然后重新启动控制器，运行：

```

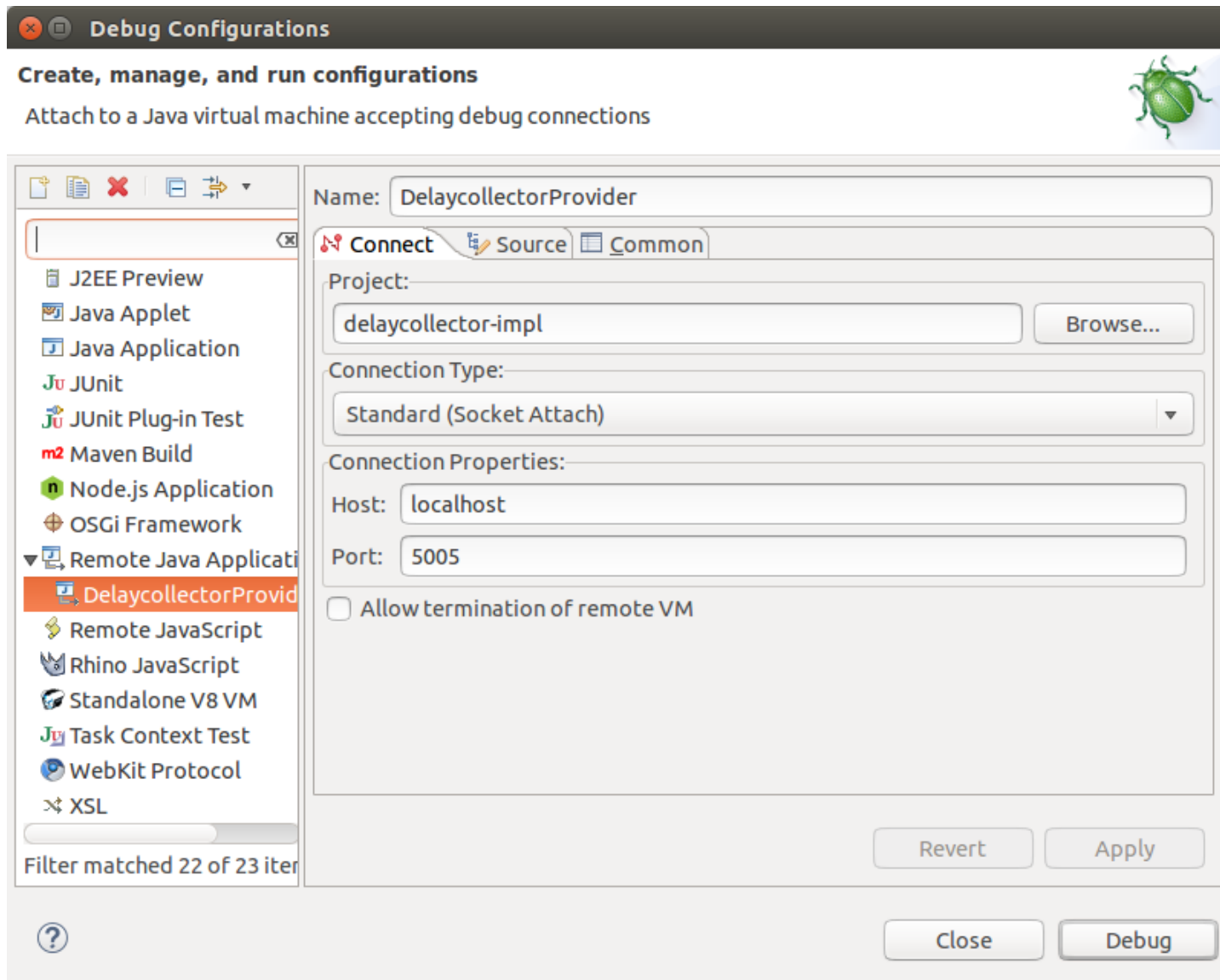
Shell
1
./karaf debug

```

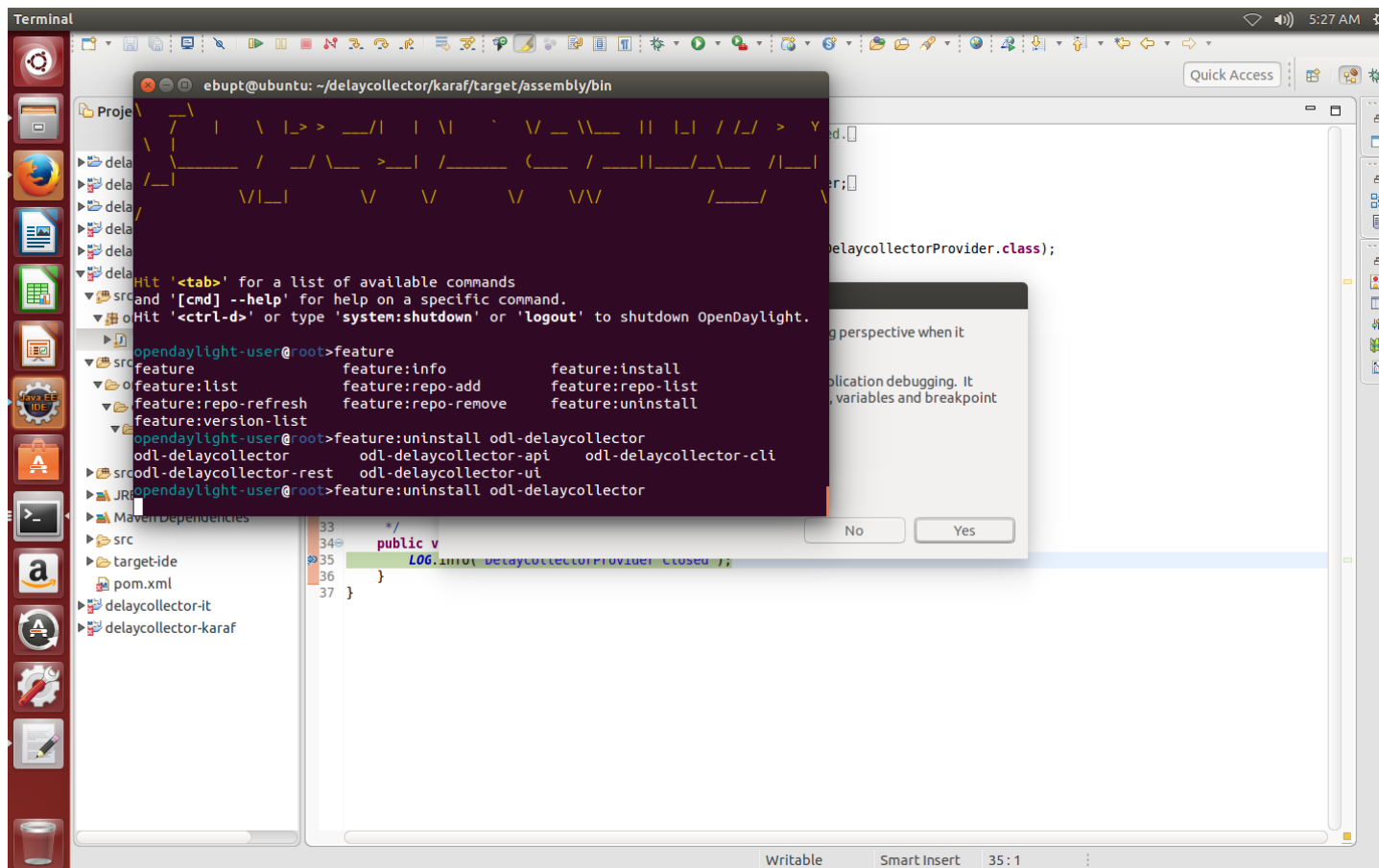
```
ebupt@ubuntu: ~/delaycollector/karaf/target/assembly/bin
spring-jdbc          spring-jms
spring-orm           spring-oxm
spring-security      spring-struts
spring-test          spring-tx
spring-web           spring-web-portlet
spring-websocket     transaction
webconsole           weld
wrapper
opendaylight-user@root>feature:install odl-delaycollector-cli
opendaylight-user@root>feature
feature              feature:info          feature:install
feature:list         feature:repo-add      feature:repo-list
feature:repo-refresh feature:repo-remove   feature:uninstall
feature:version-list
opendaylight-user@root>logout

ebupt@ubuntu:~/delaycollector/karaf/target/assembly/bin$ ./karaf debug^C
ebupt@ubuntu:~/delaycollector/karaf/target/assembly/bin$ ./karaf ]
Error: Could not find or load main class ]
ebupt@ubuntu:~/delaycollector/karaf/target/assembly/bin$ ^C
ebupt@ubuntu:~/delaycollector/karaf/target/assembly/bin$ ./karaf debug
Listening for transport dt_socket at address: 5005
Apache Karaf starting up. Press Enter to open the shell now...
86% [=====>
```

可以看到控制器已经处于debug模式下并且在监听5005端口。打开eclipse，配置eclipse远程调试：



点击debug就可以开启调试了。（如果调试显示无法连接，请尝试更换eclipse版本）由于插件在odl启动的过程中已经被安装了，我们可以先把它卸载，同时在close方法内打一个断点。在odl中运行feature:uninstall odl-delaycollector，可以看到eclipse弹出了进入调试窗口的提示。

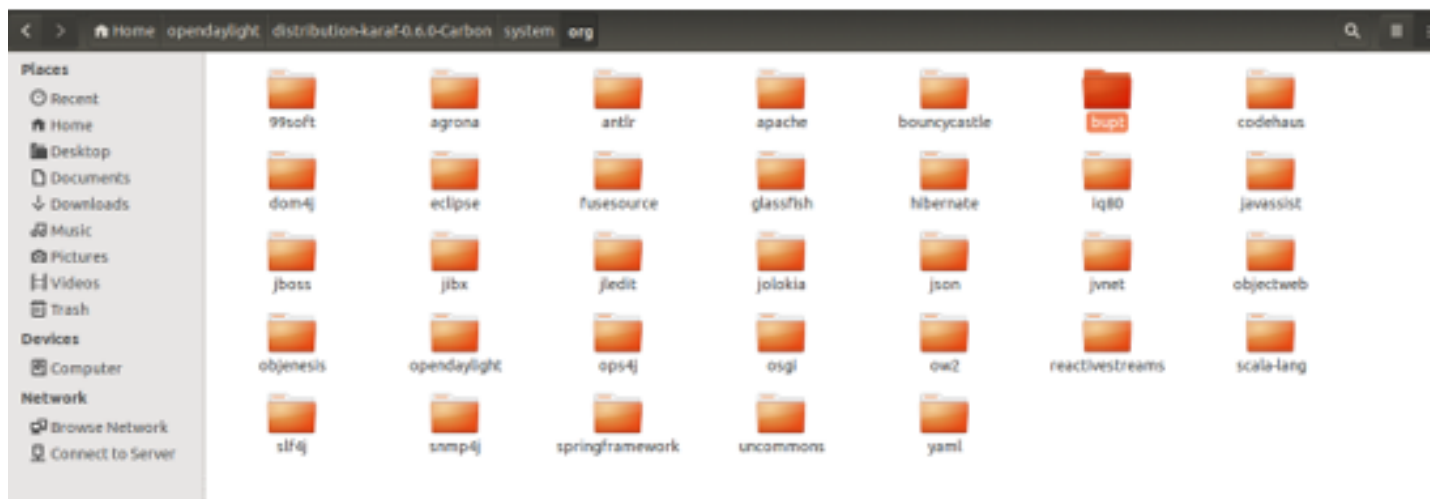


代码确实停在了我们打断点的地方。点击上边的绿色调过按钮，插件被成功卸载。之后可以再次运行feature:install odl-delaycollector安装插件，同时观察init方法中的断点。

1.4 项目集成

现在我们已经可以运行自己的项目了，那么我们要怎样把我们的插件集成到发行版控制器中呢？

打开/delaycollector/karaf/target/assembly/system\$目录，进入org.bupt目录，可以看到我们的插件所生成的jar包。可以直接把该目录拷贝到对应版本的odl发行版的对应目录下。



然后启动发行版odl控制器。运行`feature repo-add: mvn:org.bupt.delaycollector/delaycollector-features/0.1.0-SNAPSHOT/xml/features`(这条指令的后半部分可以在我们的插件的`delaycollector/karaf/target/assembly/etc$`目录下找到)。这条指令的意思就是告诉odl我要把自己的插件添加进odl, karaf会读取到插件的信息。然后安装插件`feature:install odl-delaycollector`。

进行到这一步, 我们已经完成了一个最简单插件的构建, 安装, 部署, 接下来会以之前写过的一个插件为例, 介绍odl新版本开发过程中的一些具体问题。

二、模块插件功能开发

SDN集中控制的思想使得控制器可以获取全网的链路信息, 交换机的传输时延就是其中之一。该插件的目标之一就是测量网路的链路时延。传统的sdn网络时延测量方法主要是采用的“三角”测量法, 该方法实现较为简单, 但是缺点就是测量的准确性一般。今天介绍另一种方法, 该方法需要修改一部分ovs交换机的代码。(这部分代码是学长做的, 我不太了解具体内幕) 总而言之就是让交换机在某种特定协议的包上打上时间戳, 当一个数据包从交换机出发, 经过两个ovs交换机之后, 它的身上就携带了两个时间戳, 只要我们在控制器中把时间戳取出并相减即可得到这条链路的时延大小。这种方法可以直接得出链路时延, 无需像三角法一样减掉两边, 因此准确度可以得到大大提高(大概在微妙级)。

2.1 插件配置

在插件开发过程中我们常常添加一些用户配置, 比如本插件中我们想控制发包器的发包速率。那么我们如何定义这些配置文件呢。

在carbon版本中, 插件的配置通过yang文件来定义。在`impl`文件下新建yang文件夹, 之后新建`delaycollector-config.yang`文件。在文件中声明我们所需的参数。

```
Delay_Sender_Ta  *IPv4.java  Ipv4Option.java  Delaycollecto
1 module delaycollector-config {
2
3     yang-version 1;
4     namespace "urn:opendaylight:packet:delaycollector-config";
5     prefix "delaycollector-config";
6
7     description
8         "This module contains the base configuration for
9         implementation.";
10
11     revision 2014-05-28 {
12         description "Initial revision.";
13     }
14
15     container delaycollector-config {
16         leaf is-active{
17             type boolean;
18             default false;
19         }
20         leaf query-delay {
21             type uint8;
22             default 1;
23         }
24     }
25 }
26 }
27
```

其中is-active为boolean类型，用于控制发包器的开关。query-delay为uint8，用于控制发包的频率，这里设置默认值为1，每秒发送一次。之后再次编译impl插

件。编译成功后在delaycollector/impl/target/generated-sources/mdsal-

binding/org.opendaylight/yang/gen/v1/urn/opendaylight/packet/delaycollector/config/rev140528目录下可以看到yang-tools生成的config类。

2.2 模块开发

介绍完大体思路后我们就可以进行模块的开发，这个插件中控制器其实主要就做了两部分。

1. 发送特定协议的数据包

2. 接收指定协议数据包并解析数据包内的时间戳。

整理好思路之后我们就可以写代码了。首先写发包模块，既然要发包，那么我们肯定要用到ODL提供给我们的发包服务，那么如何在我们的项目中引入这个服务呢？

上篇文章中我们提到了关键的blueprint.xml文件用于依赖注入，引入服务的话也要在这个文件中进行声明。

关键代码按如图所示：

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:odl="http://opendaylight.org/xmlns/blueprint/v1.0.0"
  odl:use-default-for-reference-types="true">

  <reference id="dataBroker"
    interface="org.opendaylight.controller.md.sal.binding.api.DataBroker"
    odl:type="default" />
  <reference id="notificationProviderService" interface="org.opendaylight.controller.sal.binding.api.NotificationProviderService" />
  <odl:clustered-app-config id="delaycollectorConfig"
    binding-class="org.opendaylight.yang.gen.v1.urn.opendaylight.packet.delaycollector.config.rev140528.DelaycollectorConfig">
    </odl:clustered-app-config>

  <odl:rpc-service id="mypacketHandlerService"
    interface="org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketProcessingService" />

  <reference id="rpcRegistry"
    interface="org.opendaylight.controller.sal.binding.api.RpcProviderRegistry" />

  <bean id="provider"
    class="delaycollector.impl.DelaycollectorProvider"
    init-method="init" destroy-method="close">
```

其中我们之前定义的配置文件也要在blueprint文件中进行注入。

```

<odl:rpc-service id="mypacketHandlerService"
    interface="org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketProcessingService" />

<reference id="rpcRegistry"
    Amazon:face="org.opendaylight.controller.sal.binding.api.RpcProviderRegistry" />

<bean id="provider"
    class="delaycollector.impl.DelaycollectorProvider"
    init-method="init" destroy-method="close">
    <argument ref="dataBroker" />
    <argument ref="delaycollectorConfig" />
    <argument ref="notificationProviderService" />
    <argument ref="mypacketHandlerService" />
    <argument ref="rpcRegistry" />
</bean>

</blueprint>

```

这里具体的服务名称建议大家去ODL中的官方插件的xml文件中去找。（ODL初学者建议先去阅读ODL 12-switch项目源码，因为这个插件涉及的协议较为简单，基本都是二层和arp协议，其中的代码相对容易理解）

```

Copyright (c) 2016 Huawei and others. All rights reserved.
package delaycollector.impl;
import java.net.InetAddress;
public class DelaySenderTask implements Runnable{
    private final DelaycollectorConfig delaycollectorConfig;
    private final PacketProcessingService packetProcessingService;
    private final DataBroker dataService;
    public DelaySenderTask(DelaycollectorConfig delayCollectorConfig,PacketProcessingService packetProcessingService,DataBroker dataService){
        this.delaycollectorConfig=delayCollectorConfig;
        this.packetProcessingService=packetProcessingService;
        this.dataService=dataService;
    }
    @Override

```

其中packetprocessingservice类就是我们要用的发包服务，它提供了一系列发送数据包的方法。

DelaycollectorConfig类是我们之前定义的配置类，其中包含我们所需的配置参数。

首先发送数据包需要一直运行，因此肯定是一个多线程任务，继承Runnable接口，之后再run函数中写我们的具体执行逻辑。

发包先要生成一个包，那么如何生成一个可被发包服务发送的数据包呢？

我这里用到了ipv4数据包，具体代码在ipv4.java中，该类需要继承org.opendaylight.controller.liblldp.Packet基类。（这个类如果大家看过ODL链路发现插件的代码应该会熟悉，在构造lldp数据包时，就用到了相关的方法。具体代码大家可以自行阅读体会，总之就是它给我们提供了一个大致的模板，只要我们在自己想构造的数据包中添加好相应的字段，基类提供的方法将会自动帮我们完成数据包的序列化）

有了IPv4.java，我们就可以用它来构建一个ip数据包了。在这个插件中，我们用来打标签的数据包是协议号为253的实验用数据包，这样可以尽量避免对其他网络协议的影响。

```
//generate a ipv4 packet
IPv4 ipv4=new IPv4();
ipv4.setTtl((byte)1).setProtocol((byte)KnownIpProtocols.Experimentation1.getIntValue());
try {
    ipv4.setSourceAddress(InetAddress.getByName("0.0.0.1")).setDestinationAddress(InetAddress.getByName("0.0.0.2"));
}
```

New一个对象然后一路set，三层包就造好了。当然到这还未结束，要正常发送数据包我们还需要一个二层包头，这里ODL给我们提供了现成的类，org.opendaylight.controller.liblldp.Ethernet，直接调用即可。这块代码就不具体介绍了，比较好理解，总之也是写好包头信息即可。

之后会用到两个ODL-l2-switch中arphandler中的两个类。PacketDispatcher，InventoryReader。其中PacketDispatcher用于处理数据包，提供了例如广播，单播数据包等操作，InventoryReader用于获取网络的拓扑信息，例如获取所有交换机，获取某个交换机的所有端口等等，这两个类的源码也相对简单，强烈建议大家去阅读相关代码。

目前为止我们已经构造好了相应数据包并且有了发包的方法，之后就可以以一定的频率把这种数据包在交换机的所有端口flood出去。

```
while(delaycollectorConfig.isIsActive()){
    //generate a ipv4 packet
    IPv4 ipv4=new IPv4();
    ipv4.setTtl((byte)1).setProtocol((byte)KnownIpProtocols.Experimentation1.getIntValue());
    try {
        ipv4.setSourceAddress(InetAddress.getByName("0.0.0.1")).setDestinationAddress(InetAddress.getByName("0.0.0.2"));
    }
    //generate a ethernet packet
    Ethernet ethernet=new Ethernet();
    EthernetAddress srcMac=new EthernetAddress(new byte[] {(byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0xee});
    EthernetAddress destMac=new EthernetAddress(new byte[] {(byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0xef});
    ethernet.setSourceMACAddress(srcMac.getValue()).setDestinationMACAddress(destMac.getValue());
    ethernet.setEtherType(EtherTypes.Ipv4.shortValue());
    ethernet.setPayload(ipv4);
    TransmitPacketInputBuilder transmitPacketInputBuilder=new TransmitPacketInputBuilder();
    packetDispatcher.setInventoryReader(inventoryReader);
    HashMap<String, NodeConnectorRef> nodeConnectorMap=inventoryReader.getControllerSwitchConnectors();
    for(String nodeid:nodeConnectorMap.keySet()){
        packetDispatcher.floodPacket(nodeid, ethernet.serialize(), nodeConnectorMap.get(nodeid), null,true);
    }
    catch (ConstructionException | UnknownHostException | PacketException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
try {
    Thread.sleep(delaycollectorConfig.getQueryDelay()*1000);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

发包器的工作基本就是这些了。

有了发包器我们还需要一个收包器接收发送的数据包。在ODL中如果我们想接收特定类型的数据包只需要使用相应的Listener接口。以下是ODL中提供的各种Listener，我们只需在插件中implements相关接口并使用NotificationProviderService对其进行注册即可，这样当符合条件的包进入控制器中就会触发我们的回调函数。


```
package delaycollector.impl;

import java.util.Map;

public class DelayFromIpv4 implements Ipv4PacketListener {
    private InfluxDB influxDB ;
    private DelaycollectorConfig config;
    private Map<String, Long> delayMap;
    public DelayFromIpv4(DelaycollectorConfig config, Map<String, Long> delayMap) {
        // TODO Auto-generated constructor stub
        this.config=config;
        this.delayMap=delayMap;
        // if(config.getInfluxdbAddress()!=null)
        // influxDB=InfluxDBFactory.connect(config.getInfluxdbAddress());
    }
    @Override
    public void onIpv4PacketReceived(Ipv4PacketReceived packetReceived) {
        // TODO Auto-generated method stub
        if (packetReceived == null || packetReceived.getPacketChain() == null) {
            return;
        }

        RawPacket rawPacket = null;
        EthernetPacket ethernetPacket = null;
        Ipv4Packet ipv4Packet = null;
        for (PacketChain packetChain : packetReceived.getPacketChain()) {
            if (packetChain.getPacket() instanceof RawPacket) {
                rawPacket = (RawPacket) packetChain.getPacket();
            } else if (packetChain.getPacket() instanceof EthernetPacket) {
                ethernetPacket = (EthernetPacket) packetChain.getPacket();
            } else if (packetChain.getPacket() instanceof Ipv4Packet) {
                ipv4Packet = (Ipv4Packet) packetChain.getPacket();
            }
        }
        if (rawPacket == null || ethernetPacket == null || ipv4Packet == null) {
            return;
        }
        if(ipv4Packet.getProtocol()!=KnownIpProtocols.Experimentation1){
            return;
        }
        Ipv4Option ipv4option=new Ipv4Option();
        try {
            ipv4option=(Ipv4Option)ipv4option.deserialize(ipv4Packet.getIpv4Options(),0,0);
            String ncId = rawPacket.getIngress().getValue().firstIdentifierOf(NodeConnector.class).firstKeyOf(NodeConnector.class);

            long delay=cacluateDelay(ipv4option);
            delayMap.put(ncId, delay);
            // Save(System.currentTimeMillis(),ncId,delay);
            // System.out.println(ncId+"<><><><>"<div class="text">+delay);
        } catch (PacketException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

其中的关键代码在onIpv4PacketReceived方法中，首先对收到的packet进行简单判断，包括包的类型和包的协议号，如果不符合条件，方法直接返回。之后从数据包

的ipv4options中读出两个时间戳计算时延数据和包的来源即可。用一个Map对其进行缓存。

2.3 RPC的定义与实现

时延数据我们已经得到了，那么我们怎样从控制器中把它取出来呢？答案是通过rpc（远程服务调用）。之前我们提到过在ODL开发中有两个文件夹特别重要，一个是impl，一个是api，我们的rpc定义就是放在api文件夹中。进入delaycollector/api/src/main/yang文件夹，可以看到里边已经有一个项目自动生成的yang文件，但是其中并没有定义实际内容，我们直接在这个文件中添加我们的rpc定义。

```

1 module delaycollector {
2     yang-version 1;
3     namespace "urn:opendaylight:params:xml:ns:yang:delaycollector";
4     prefix "delaycollector";
5
6     revision "2015-01-05" {
7         description "Initial revision of delaycollector model";
8     }
9
10    rpc getDelay {
11        input {
12            leaf NodeConnector{
13                type string;
14            }
15        }
16        output {
17            leaf delay {
18                type uint32;
19            }
20        }
21    }
22    rpc getGlobalDelay
23    {
24        output
25        {
26            list delay-list{
27                description "latency info";
28                config false;
29                key "nodeConnector";
30                leaf nodeConnector{
31                    description "srcnode";
32                    type string;
33                }
34            }
35        }
36    }
37    leaf delay{
38        description "delay";
39        type uint32;
40    }
41
42 }
43 }
44
45 }
46

```

这里我们定义了两个rpc，其中一个用于返回特定节点的时延，另一个返回所有节点的时延。之后再次编译api项目，在target的对应目录下将生成相应的java文件。

要实现rpc功能，需要在我们的实现类中implements DelaycollectorService接口。

```

8 package delaycollector.impl;
9
9* import java.math.BigInteger;[]
7
8
9
0 public class DelayServiceImpl implements DelaycollectorService {
1     private Map<String, Long> delayMap;
2* public DelayServiceImpl(Map<String, Long> delayMap) {
3     // TODO Auto-generated constructor stub
4     this.delayMap=delayMap;
5 }
6
7* @Override
8 public Future<RpcResult<GetDelayOutput>> getDelay(GetDelayInput input) {
9     // TODO Auto-generated method stub
0     String nodeconnector=input.getNodeConnector();
1     GetDelayOutputBuilder getDelayOutputBuilder=new GetDelayOutputBuilder();
2     getDelayOutputBuilder.setDelay(delayMap.get(nodeconnector));
3     return RpcResultBuilder.success(getDelayOutputBuilder.build()).buildFuture();
4 }
5
6* @Override
7 public Future<RpcResult<GetGlobalDelayOutput>> getGlobalDelay() {
8     // TODO Auto-generated method stub
9     GetGlobalDelayOutputBuilder getGlobalDelayOutputBuilder=new GetGlobalDelayOutputBuilder();
0     List<DelayList> delayLists=new ArrayList<DelayList>();
1     for(String ncid:delayMap.keySet()){
2         DelayListBuilder delayListBuilder=new DelayListBuilder();
3         delayListBuilder.setKey(new DelayListKey(ncid));
4         delayListBuilder.setDelay(delayMap.get(ncid));
5         delayLists.add(delayListBuilder.build());
6     }
7     getGlobalDelayOutputBuilder.setDelayList(delayLists);
8     return RpcResultBuilder.success(getGlobalDelayOutputBuilder.build()).buildFuture();
9 }
0 }
1 }
2

```

这部分代码也很简单，从input中获取输入数据，通过outputBuilder构造输出。

2.4 项目整合

插件的各个模块我们已经创建完毕，之后要做的就是各个模块有序的运行起来。

回到DelayCollector.Java文件，在init方法中进行插件的初始化工作。

```
~/  
public void init() {  
    LOG.info("DelaycollectorProvider Session Initiated");  
    Delay_Sender_Task delay_Sender_Task=new Delay_Sender_Task(config, packetProcessingService, dataBroker);  
    thread=new Thread(delay_Sender_Task, "ipv4packetSpeaker");  
    thread.start();  
    DelayFromIpv4 delayFromIpv4=new DelayFromIpv4(config, delayMap);  
    registration=notificationProviderService.registerNotificationListener(delayFromIpv4);  
    DelayServiceImpl delayServiceImpl=new DelayServiceImpl(delayMap);  
    rpcRegistration=rpcProviderRegistry.addRpcImplementation(DelaycollectorService.class, delayServiceImpl);  
}  
/**
```

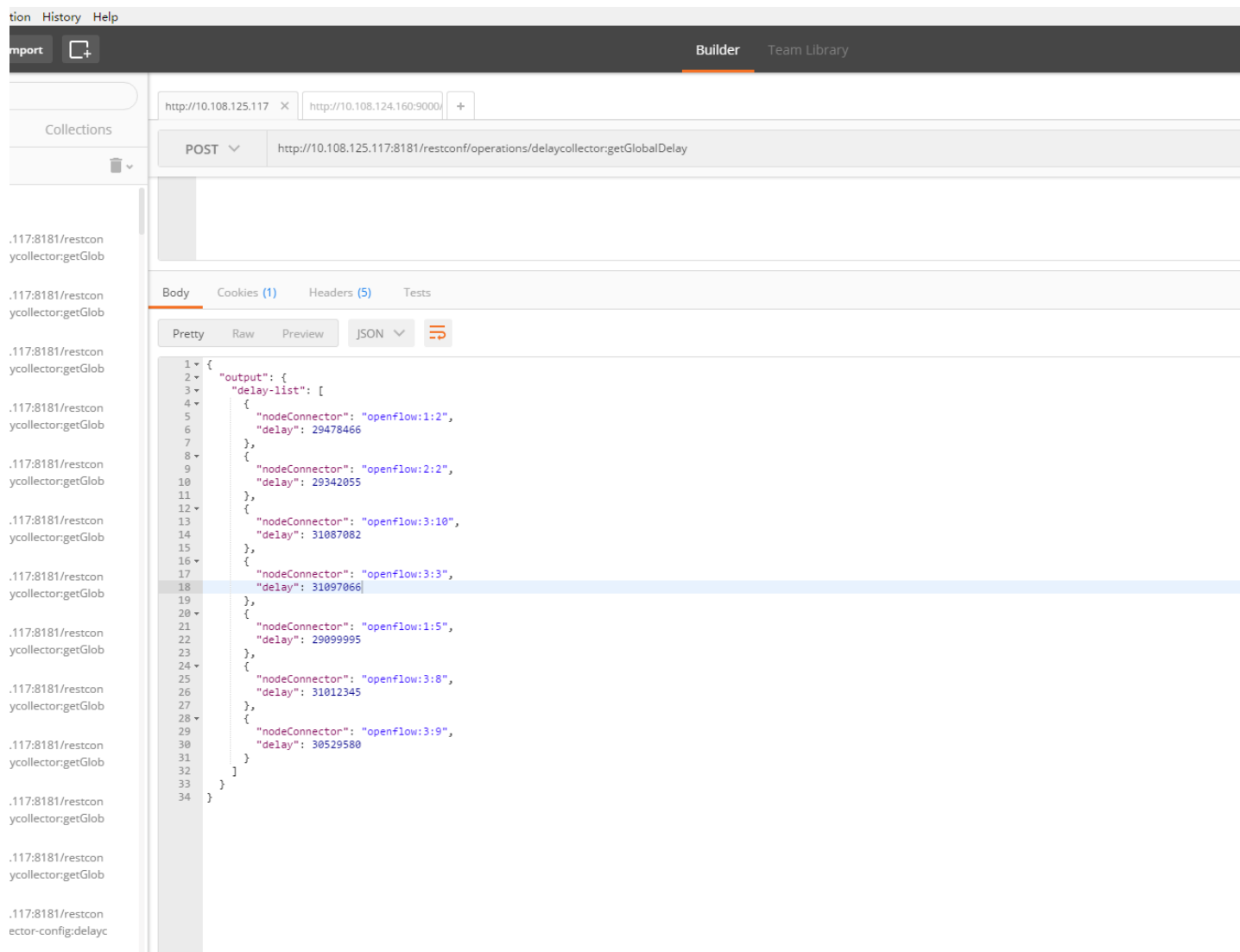
其中需要注意的是对rpc服务和ipv4包监听器进行注册。

2.5 项目验证

对整个插件进行最后一次编译，并把它集成到发行版ODL中，启动ODL，安装插件。进入YangUI，可以看到插件已经有了相关的接口。

打开PostMan软件，输入相应链接，getGlobalDelay接口不需要输入参数，直接Post，即可得到返回参数。

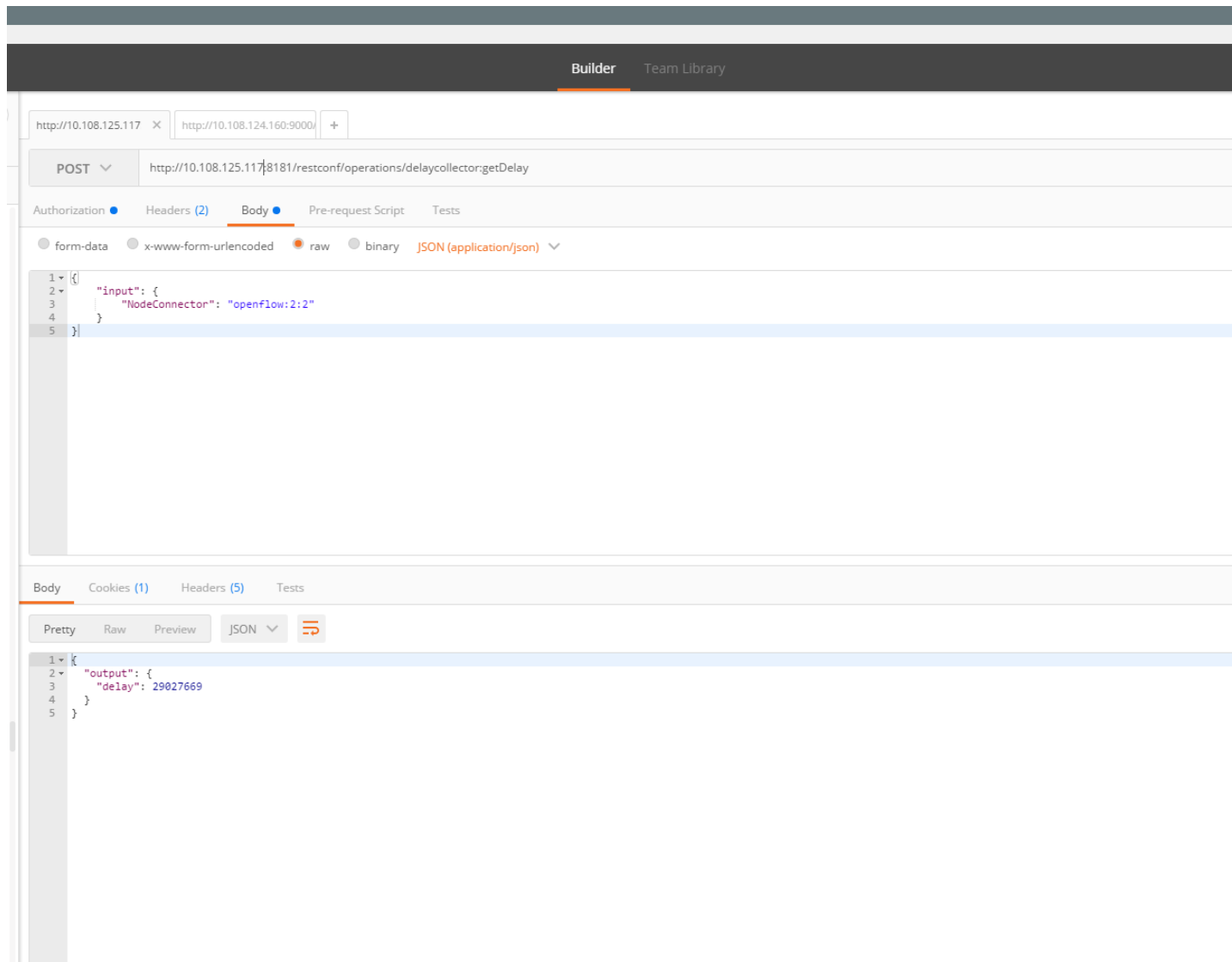
The screenshot displays the OpenDaylight YangUI interface. The top navigation bar includes the OpenDaylight logo, the 'YangUI' title, and a 'Logout (admin)' button. The left sidebar shows a 'Topology' tab and a 'Yang UI' tab. The main content area has four tabs: 'API', 'HISTORY', 'COLLECTION', and 'PARAMETERS'. The 'API' tab is active, showing a tree view of YANG models under the 'ROOT' node. The tree includes nodes like 'aaa-encrypt-service-config', 'address-tracker-config', 'arp-handler-config', 'cluster-admin', 'config', 'delaycollector', 'entity-owners', 'flow-capable-transaction', 'flow-topology-discovery', and 'forwarding-rules-manager-config'. The 'delaycollector' node is expanded, revealing its 'operations' section with 'getDelay' and 'getGlobalDelay' methods. A 'Custom API request' button is visible below the tree. A green notification bar at the bottom states 'Loading completed successfully'.



The screenshot shows a REST client interface with a dark theme. The top bar includes 'tion History Help' and 'Builder Team Library'. The main area displays a POST request to `http://10.108.125.117:8181/restconf/operations/delaycollector:getGlobalDelay`. The response body is shown in JSON format, displaying a list of delay collector configurations.

```
1 {
2   "output": {
3     "delay-list": [
4       {
5         "nodeConnector": "openflow:1:2",
6         "delay": 29478466
7       },
8       {
9         "nodeConnector": "openflow:2:2",
10        "delay": 29342055
11      },
12      {
13        "nodeConnector": "openflow:3:10",
14        "delay": 31087082
15      },
16      {
17        "nodeConnector": "openflow:3:3",
18        "delay": 31097066
19      },
20      {
21        "nodeConnector": "openflow:1:5",
22        "delay": 29099995
23      },
24      {
25        "nodeConnector": "openflow:3:8",
26        "delay": 31012345
27      },
28      {
29        "nodeConnector": "openflow:3:9",
30        "delay": 30529580
31      }
32    ]
33  }
34 }
```

再测试另一个接口，输入参数，直接post。

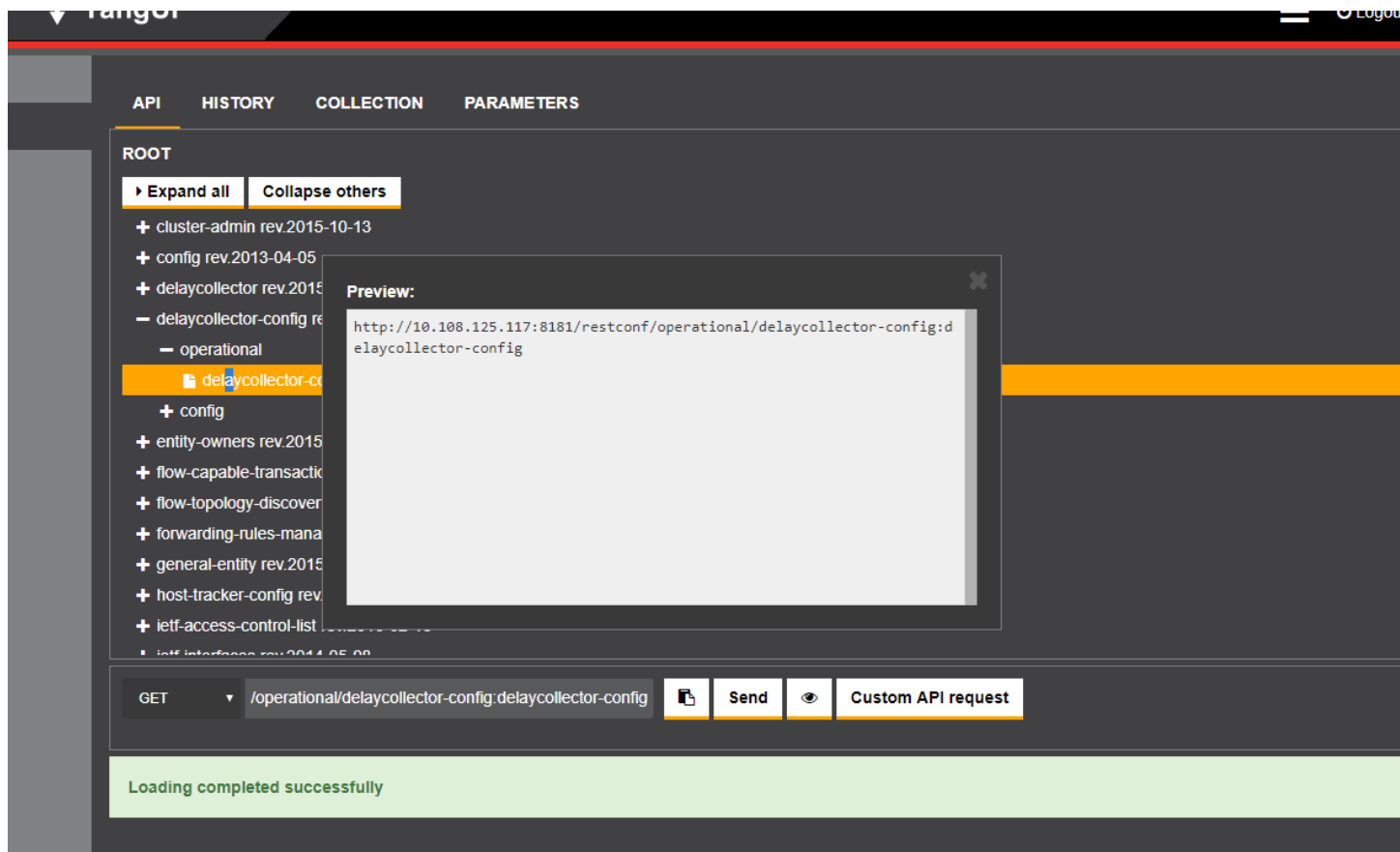


更改插件配置

之前在插件添加了相应的配置文件，那么如何更改插件的配置呢：

1. 建立xml文件。
2. 调用rpc

这里只介绍第二种。使用rpc更改插件配置。依然是使用postman，输入链接和配置参数（参数的名称参考自己定义的Yang文件）。



这里是调用put方法。成功更改配置后，插件会重新进行初始化过程，具体大家可以通过eclipse调试进行验证。

具体代码地址可以到Git地址获取：[git@github.com:xujunnnn/DelayCollector.git](https://github.com:xujunnnn/DelayCollector.git)

作者信息：

徐军 北京邮电大学网络技术研究院网络智能研究中心





- 本站原创文章仅代表作者观点，不代表SDNLAB立场。所有原创内容版权均属SDNLAB，欢迎大家转发分享。但未经授权，严禁任何媒体（平面媒体、网络媒体、自媒体等）以及微信公众号复制、转载、摘编或以其他方式进行使用，转载须注明来自 SDNLAB并附上本文链接。本站中所有编译类文章仅用于学习和交流目的，编译工作遵照 CC 协议，如果有侵犯到您权益的地方，请及时联系我们。
- 本文链接: <https://www.sdnlab.com/19857.html>