Open Daylight Controller 指南

最新版: yeasy@github

更新历史:
V0.3: 2013-08-08
    补充内容，如何编译代码。
V0.2: 2013-07-14
    完成用户安装运行和开发环境部署。
V0.1: 2013-07-10
    完成框架。

第1章 安装运行

官方网站为 http://www.Open Daylight.org。

1.1 环境配置

Open Daylight Controller 是 java 程序，理论上可以运行在任何支持 java 的环境中。推荐环境为比较新的 Linux 环境和 Java 虚拟机 1.7+版本。

例如，在 Ubuntu 12.04 系统中，需要安装 java1.7 版本，命令为：

```
#sudo add-apt-repository ppa:webupd8team/java
#sudo apt-get update && sudo apt-get install oracle-jdk7-installer
```

之后，配置 JAVA_HOME 环境变量，添加下面的行到/etc/environment 文件。

```
JAVA_HOME=/usr/lib/jvm/java-7-oracle
```

1.2 安装和使用

1.2.1 获取源码

源码可以从 https://jenkins.Open Daylight.org/controller/job/controller-nightly/lastSuccessfulBuild/artifact/Open Daylight/distribution/Open Daylight/target/下载，或者利用 git 下载最新的版本

对于匿名用户：

```
git clone https://git.Open Daylight.org/gerrit/p/controller.git
```

对于项目注册用户：

```
git clone ssh://<username>@git.Open Daylight.org:29418/controller.git
```

1.2.2 编译运行

进入 Open Daylight/distribution/Open Daylight/src/main/resources 目录，下面一般有若干文件和目录，包括：

run.bat：Windows 系统下的运行脚本。

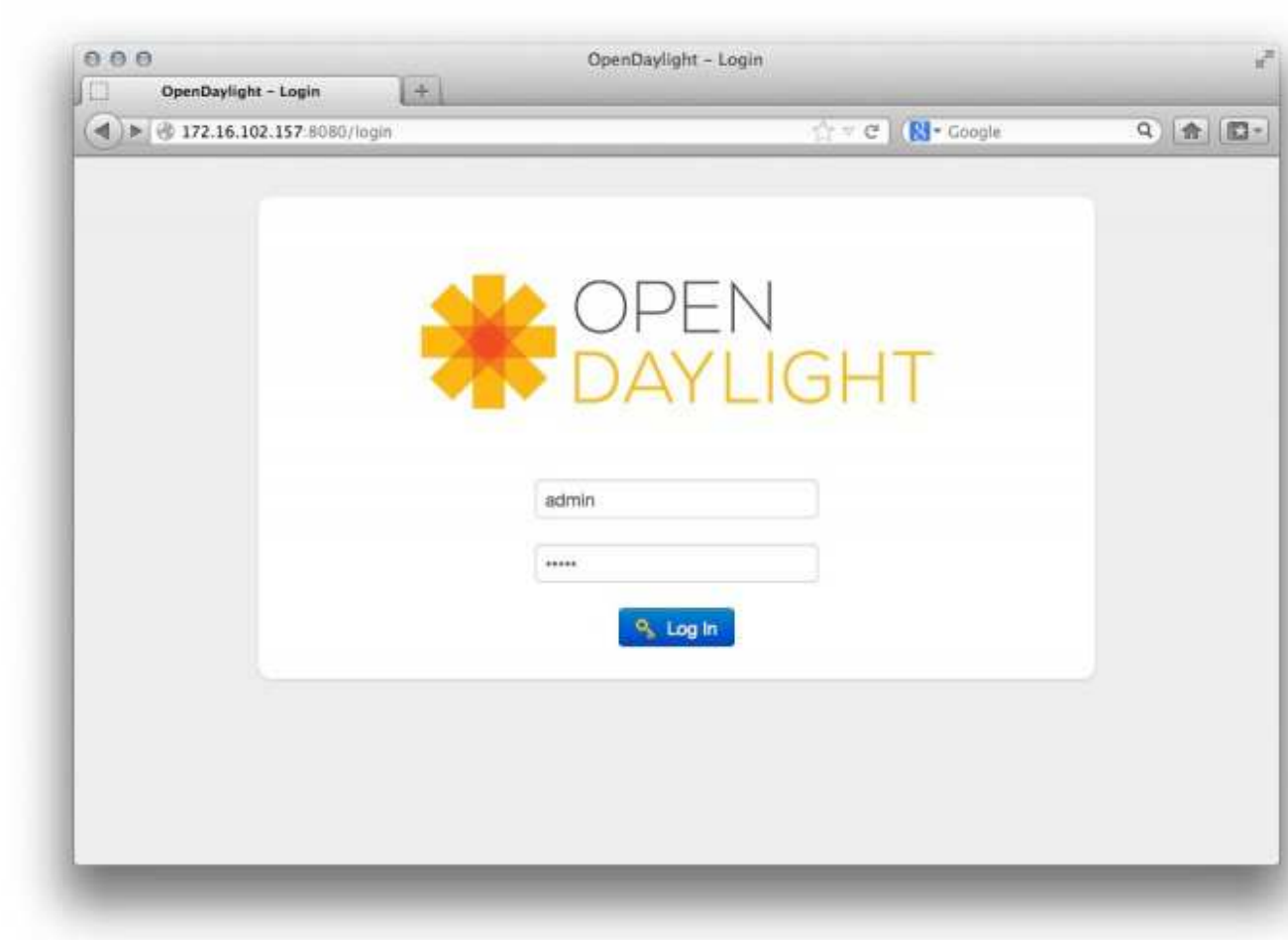
run.sh：Linux 系统下的运行脚本。

version.properties：编译的版本信息。

configuration：基本的配置信息。

在 Linux 平台上，需要用 root 权限运行./run.sh。

运行成功后可以打开浏览器访问本地地址，并登陆控制器 web UI，默认用户名和密码都是 admin。



图表 1 控制器 Web 登陆界面

1.2.3 常见运行问题

如果运行时报错：java.lang.OutOfMemoryError: PermGen space。可以通过修改 maven 配置为

```
export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
```

1.2.4 使用 Mininet 测试

Mininet 可以用来模拟大规模的虚拟网络。可以从 mininet.github.org 下载代码。

Open Daylight Controller 可以跟 Mininet 很好的联动。

在已经安装 Mininet 环境的机器（如果跟 controller 在同一个机器，可以用 127.0.0.1）中，启动 Mininet

```
mininet@mininet-vm:~$ sudo mn --controller=remote,ip=127.0.0.1 --topo tree
,3
*** Creating network
*** Adding controller
```

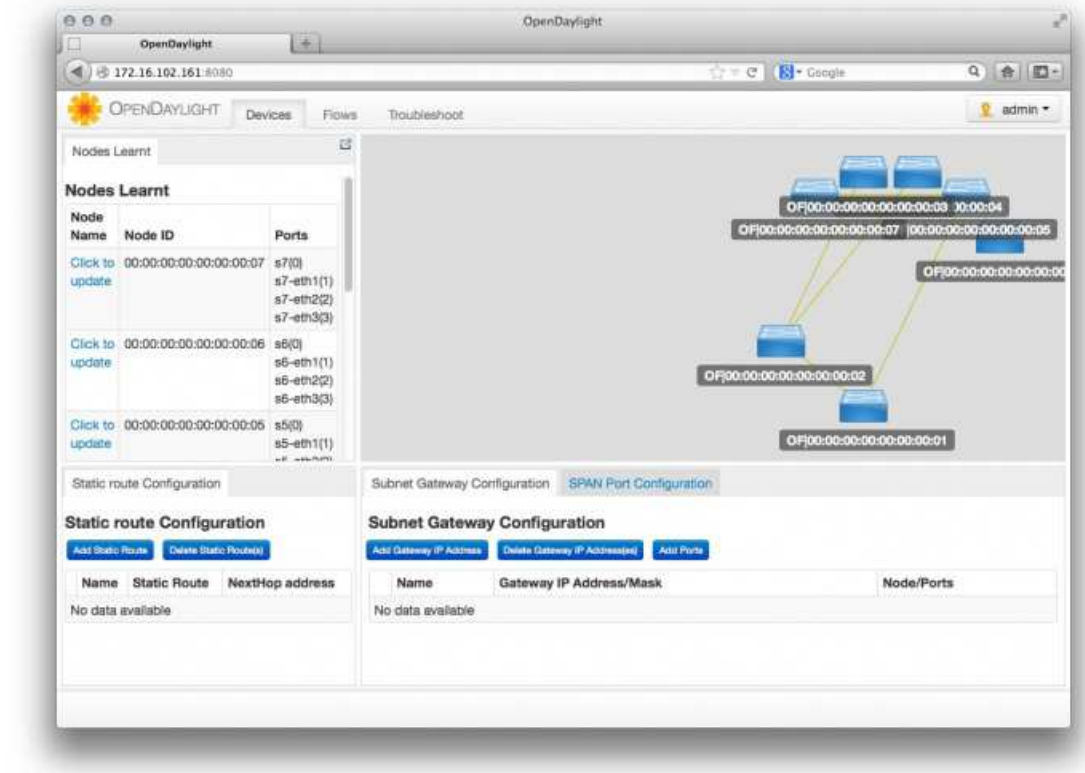
```
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
(h1, s3) (h2, s3) (h3, s4) (h4, s4) (h5, s6) (h6, s6) (h7, s7) (h8, s7) (s1, s2) (s1, s5) (s
2, s3) (s2, s4) (s5, s6) (s5, s7)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7
*** Starting CLI:
mininet>
```

连接成功后可以测试控制器的功能。

1.2.4.1 Simple Forwarding Application

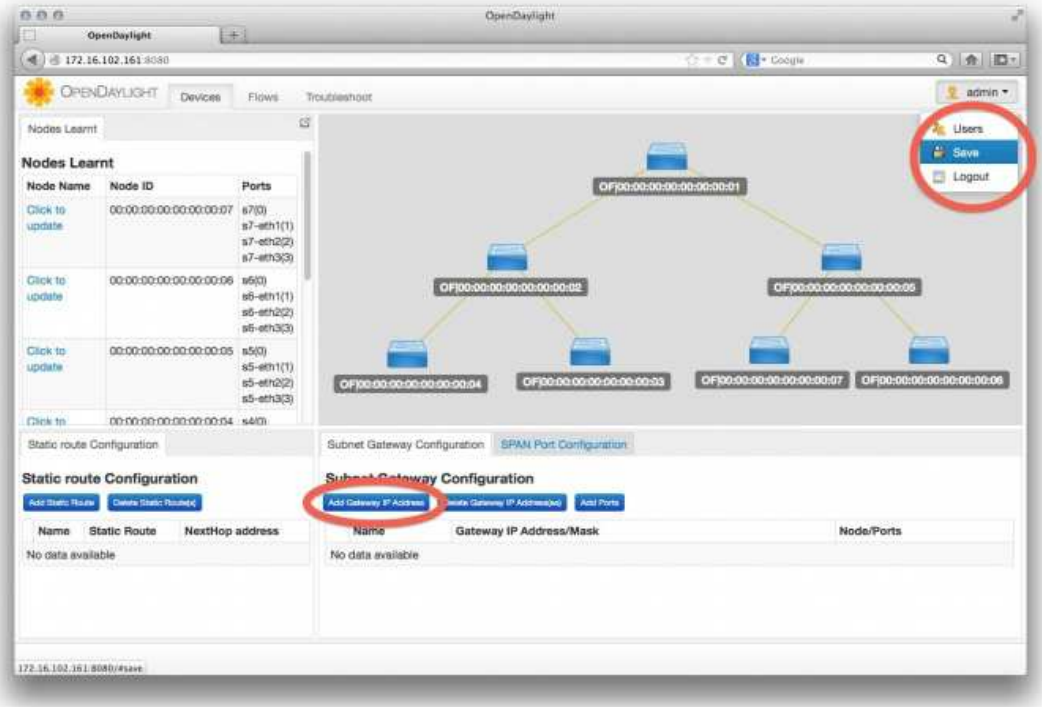
Simple Forwarding 是 Open Daylight Controller 上的一个应用，它通过 arp 包来探测连接到网络的每个主机，并给交换机安装规则（该规则指定到某个主机地址的网包路径），让网包能顺利转到各个主机。

首先，登录到 Controller 界面。



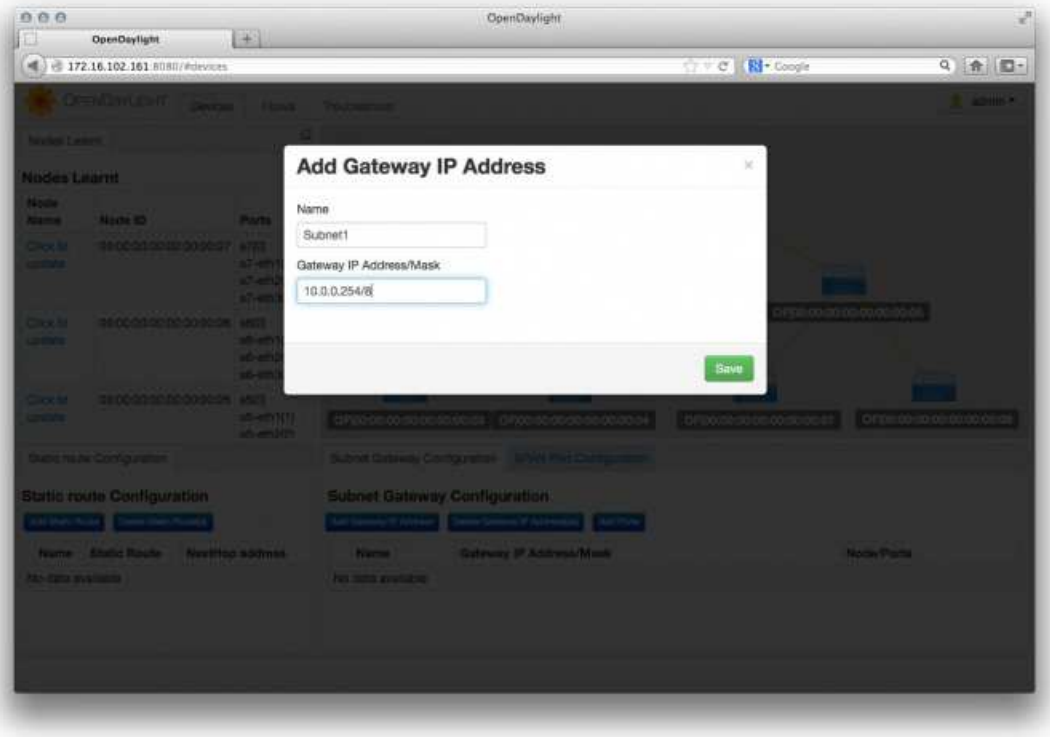
图表 2 Simple Forwarding 界面

通过拖动设备，形成逻辑拓扑，并保存配置。



图表 3 Simple Forwarding 拖动成逻辑拓扑

点击 Add Gateway IP Address 按钮，添加 IP 和 10.0.0.254/8 子网。

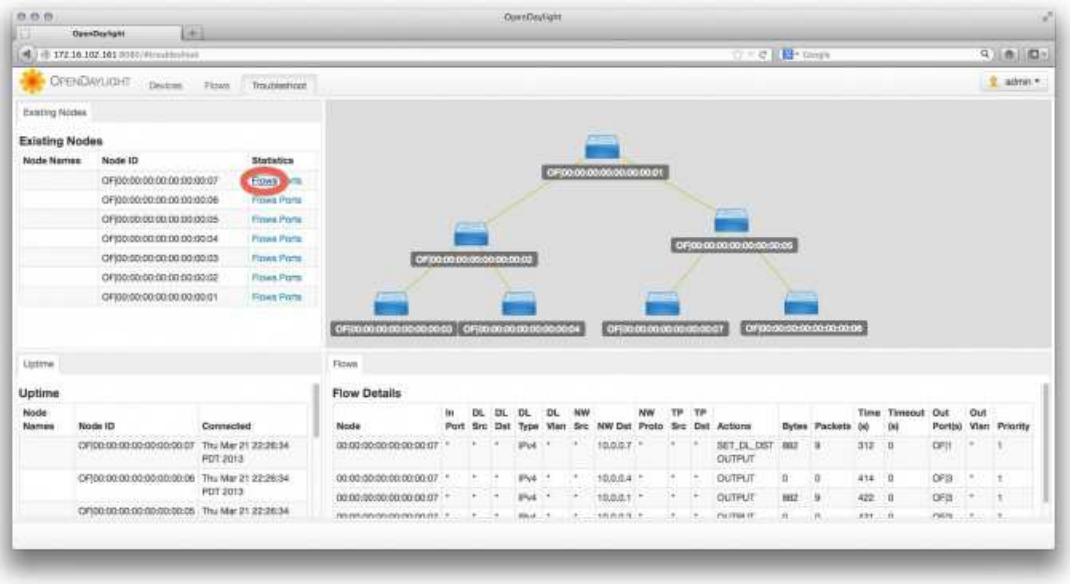


图表 4 添加网关 IP

在 Mininet 中确认主机可以相互连通。

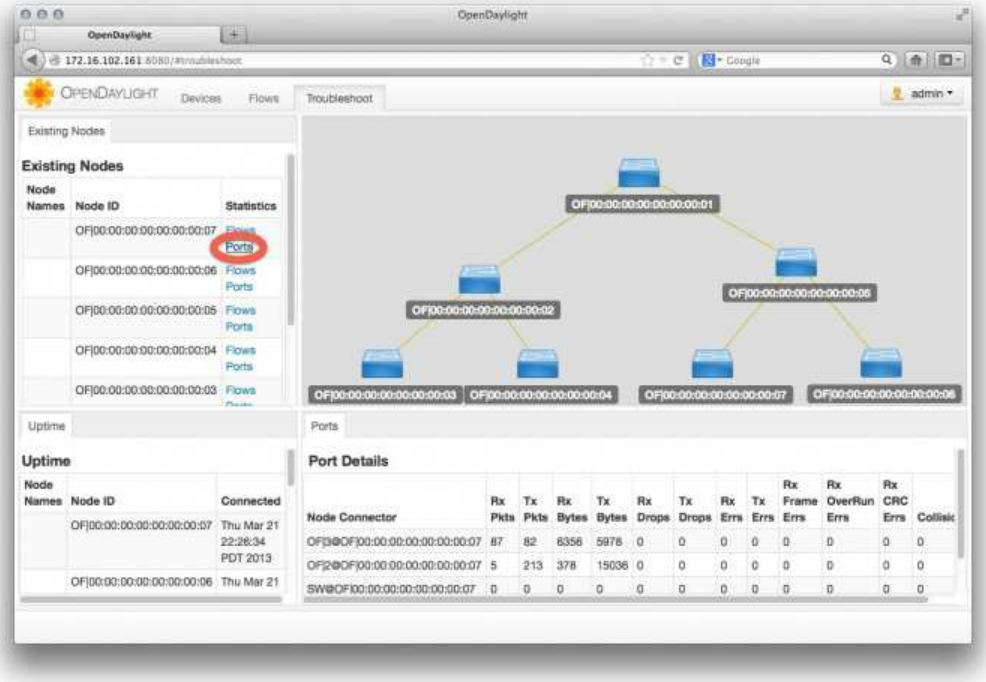
```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_req=1 ttl=64 time=1.52 ms
64 bytes from 10.0.0.7: icmp_req=2 ttl=64 time=0.054 ms
64 bytes from 10.0.0.7: icmp_req=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.7: icmp_req=4 ttl=64 time=0.052 ms
--- 10.0.0.7 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.052/0.422/1.523/0.635 ms
mininet>
```

点击 Troubleshooting 标签页，查看交换机的流表细节。



图表 5 查看交换机流表细节

查看端口细节。



图表 6 查看端口细节

在 osgi 的控制台，输入 ss simple，可以看到 Simple Forwarding 应用被激活。

```
osgi> ss simple
"Framework is launched."
```

```
id    State       Bundle
45    ACTIVE      org.Open Daylight.controller.samples.simpleforwarding_0.4.0.S
NAPSHOT
```

第2章 开发环境

2.1 概述

Open Daylight Controller 提供了一个模块化的开放 SDN 控制器。

它提供了开放的北向 API（开放给应用的接口），同时南向支持包括 OpenFlow 在内的多种 SDN 协议。底层支持混合模式的交换机和经典的 OpenFlow 交换机。

运行系统推荐为 Linux 系列，并安装了 Java1.7+环境。

自带的 Web UI 也是利用了北向的 API 进行交互。

2.2 架构原则

Open Daylight Controller 在设计的时候遵循了六个基本的架构原则：

运行时模块化和扩展化（Runtime Modularity and Extensibility）：支持在控制器运行时进行安装、删除和服务的更新。

多协议的南向支持（Multiprotocol Southbound）：南向支持多种协议。

服务抽象层（Service Abstraction Layer）：南向多种协议对上提供统一的北向服务接口。

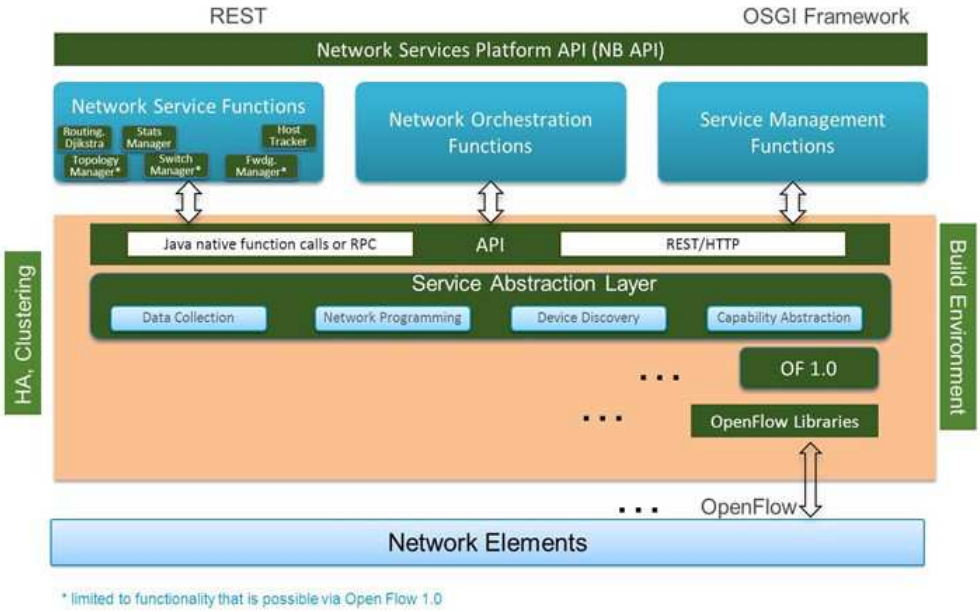
开放的可扩展北向 API（Open Extensible Northbound API）：提供可扩展的应用 API，通过 REST 或者函数调用方式。两者提供的功能要一致。

支持多租户、切片（Support for Multitenancy/Slicing）：允许网络在逻辑上（或物理上）划分成不同的切片或租户。控制器的部分功能和模块可以管理指定切片。控制器根据所管理的分片来呈现不同的控制观测面。

一致性聚合（Consistent Clustering）：提供细粒度复制的聚合和确保网络一致性的横向扩展（scale-out）。

2.3 架构框架

2.3.1 框架概述



图表 7 架构框架

如图表 7 所示，南向通过 plugin 的方式来支持多种协议，包括 OpenFlow1.0、1.3，BGP-LS 等。这些模块被动态挂载到服务抽象层（SAL），SAL 为上层提供服务，将来自上层的调用封装为适合底层网络设备的协议格式。控制器需要获取底层设备功能、可达性等方面的信息，这些信息被存放在拓扑管理器（Topology Manager）中。其他的组件，包括 ARP handler、Host Tracker、Device Manager 和 Switch Manager，则为 Topology Manager 生成拓扑数据。

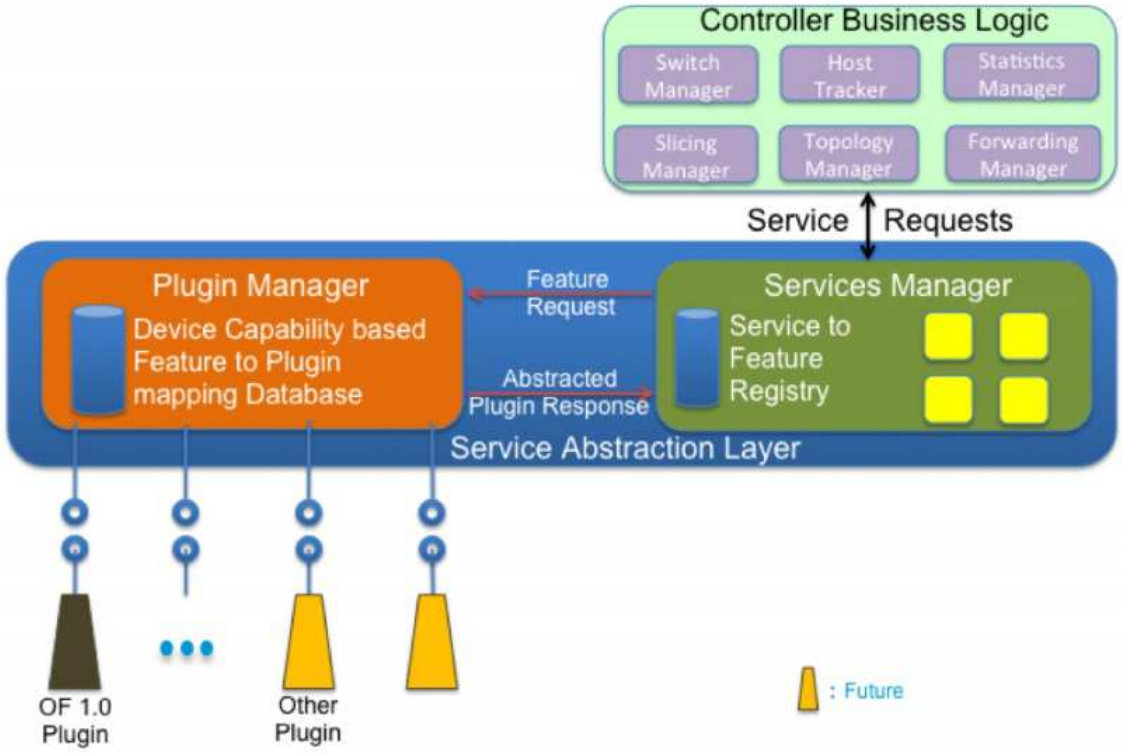
控制器为应用（App）提供开放的北向 API。支持 OSGi 框架和双向的 REST 接口。OSGi 框架提供给与控制器运行在同一地址空间的应用，而 REST API 则提供给运行在不同地址空间的应用。所有的逻辑和算法都运行在应用中。

控制自带了 GUI，这个 GUI 使用了跟应用同样的北向 API，这些北向 API 也可以被其他的应用调用。

2.3.2 功能概述

2.3.2.1 服务抽象层

服务抽象层（SAL）是整个控制器模块化设计的核心，支持多种南向协议，并为模块和应用支持一致性的服务。



图表 8 抽象服务层

OSGi 框架支持动态链接插件，来支持更多的南向协议。SAL 提供了基本的服务，比如设备探测（用于拓扑管理器）。服务基于插件提供的特性来构建。服务的请求被 SAL 映射到合适的插件上，采用合适的南向协议跟底层设备进行交互。各个插件相互之间独立并且跟 SAL 松耦合。SAL 框架是 Open Daylight Controller 项目的贡献之一。

拓扑服务（Topology service）是一系列服务，允许传输拓扑信息，包括探测新加入的节点和链路等；

数据包服务（Data Packet service）将来自代理的数据包发送给应用；

流编程服务（Flow Programming service）为不同的代理编程流规则提供必要的逻辑。

统计服务（Statistics service）搜集统计信息。

流

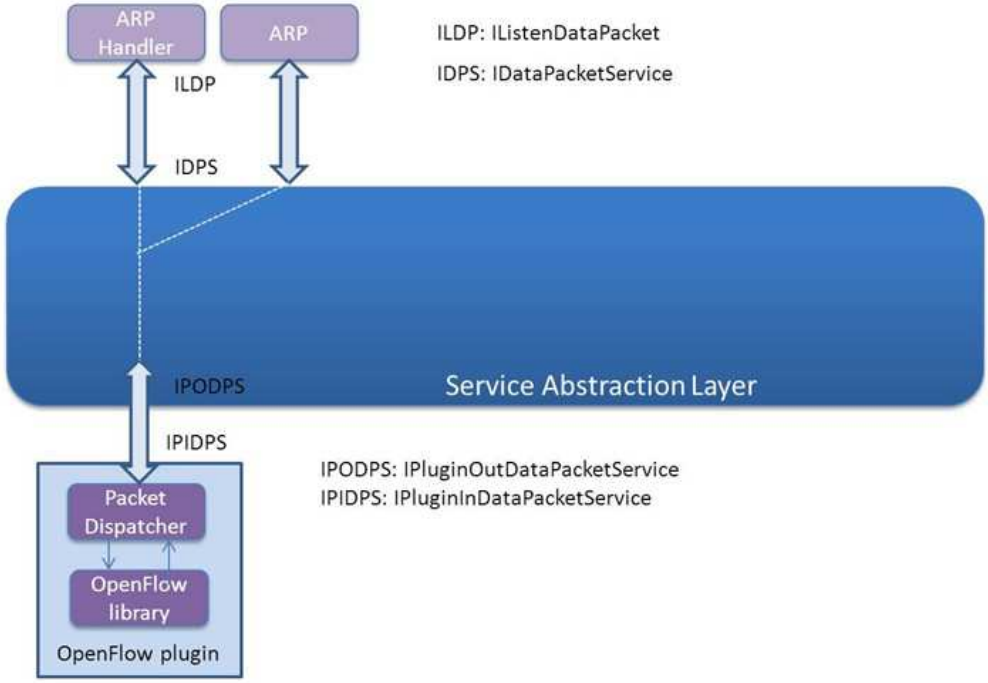
节点连接（Node Connector）或者端口。

队列。

清单服务（Inventory service），返回节点和端口的存储信息。

资源服务（Resource service）查询资源的状态。

2.3.2.2 SAL 服务：数据包服务



图表 9 数据包服务

图表 9 展示了一个实现 SAL 服务的例子，基于 OpenFlow1.0 来实现数据包服务。

- IListenDataPacket 是一个被上层模块或应用（例如 ARP Handler）实现的服务，用于接收数据包。
- IDataPacketService 接口被 SAL 实现，提供从代理发送和接收数据包的服务。这个服务在 OSGi 中注册，以便其他程序获取调用。
- IPluginOutDataPacketService 接口面向 SAL，用于当一个协议插件希望发送一个网包到应用层。
- IPluginInDataPacketService 接口面向协议插件，用于通过 SAL 向代理发送网包。

代码在 SAL 和各个服务、插件之间的执行过程：

OpenFlow 插件获取了 ARP 包，该包需要被 ARP Hanlder 应用进行处理；

OpenFlow 插件调用 IPluginOutDataPacketService 将包发送到 SAL；

ARP Handler 应用实现已经注册到了 IListenDataPacket 服务，SAL 收到网包后将把包发送到 ARP Handler 应用；

ARP Handler 应用将处理网包。

反向的处理过程（发送出去网包）是

应用创建包并调用 SAL 提供的 IDataPacketService 接口发出包。目标网络设备作为 API 参数的一部分；

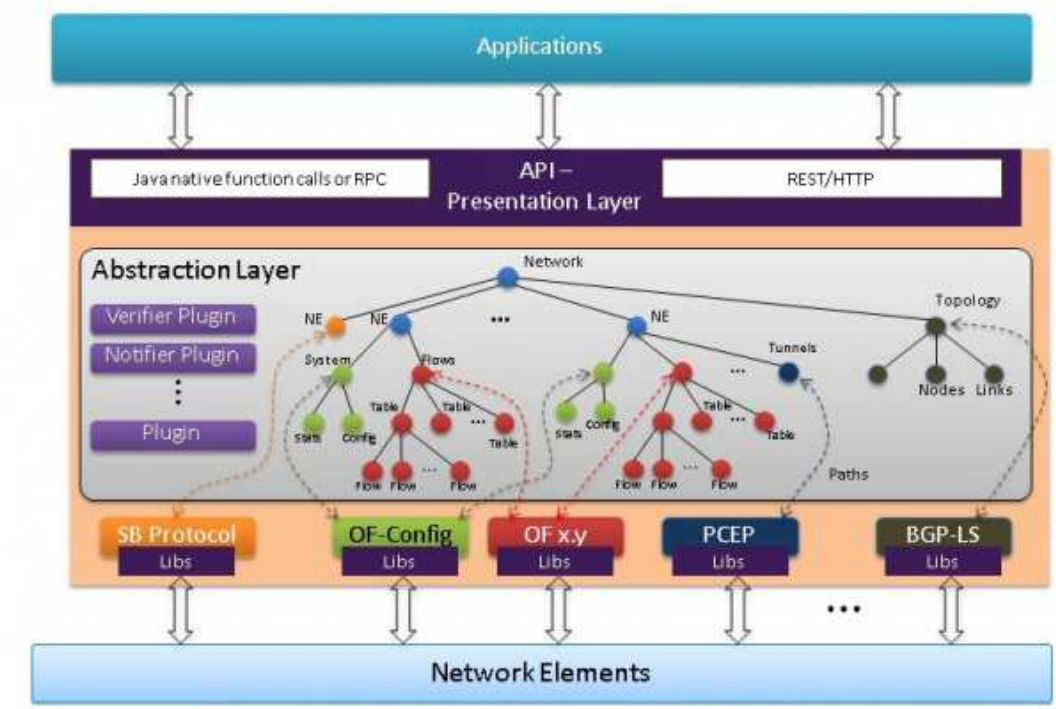
SAL 根据目标网络设备为对应的协议插件（例如 OpenFlow 插件）调用 IPluginInDataPacketService 接口。

协议插件将包发送给适当的网络元素。协议相关的处理都由插件完成。

2.3.2.3 控制器 SAL 的演进

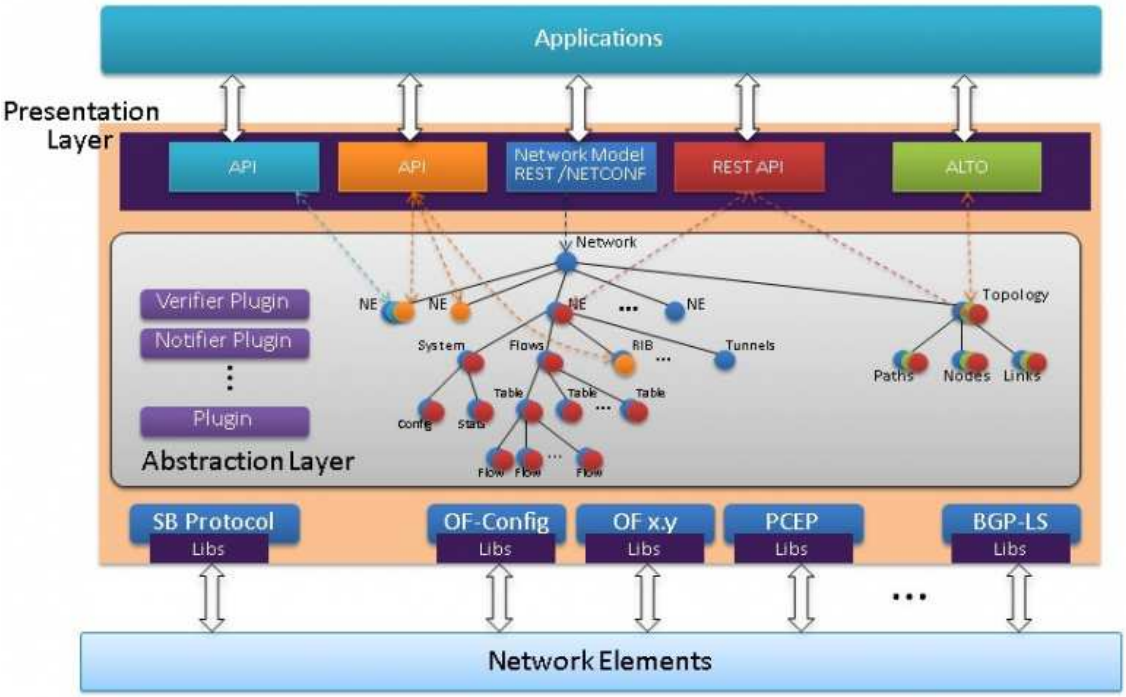
SAL 已经演进成为了一种基于模型的方法。框架被用于对网络（属性和设备）建模，并动态地在服务应用之间通过北向 API 和协议插件提供的南向 API 进行映射。图表 10 展示了南向

插件如何提供网络模型树的部分。



图表 10 南向插件提供网络模型树的部分

图表 11 展示了应用如何通过北向 API 来获取网络模型的信息。



图表 11 应用通过北向 API 来获取网络模型信息

2.3.2.4 交换机管理器

Switch Manager API 处理网络元素（一般为路由器、交换机）的细节。当网络元素被探测到的时候，它的属性（包括交换机和路由器类型、版本等）存放在 Switch Manager 的数据库中。

2.3.2.5 GUI

GUI 作为一个应用来实现，采用了北向的 REST API 来跟其他模块交互。因此，GUI 能实现的功能很容易被集成到其他的管理系统中。

2.3.2.6 高可靠性

Open Daylight 控制器支持一个基于集群的高可用性模型。多个 Open Daylight 控制器可以配合作为一个逻辑控制器。不仅支持了细粒度的冗余，并且支持线性的横向扩展。为了支持高可用性，需要在下面几个方面增加回弹性能：

- 控制器层：通过集群方式添加一个或多个控制器实例。
- 交换机支持 multi-homed，支持多控制器。
- 应用支持 multi-homed，支持多控制器。

交换机通过持续的点到点 TCP/IP 协议连接到多个控制器。控制器和应用之间通过 REST 接口连接，是基于 HTTP 的。因此所有的基于 HTTP 的回弹特性将可以被利用，包括：

- 为控制器集群提供一个虚拟 IP。
- 利用轮询 DNS 查询后，应用跟集群之间发送交互。
- 在应用和控制器集群之间部署一个 HTTP 的负载均衡，不仅提供可靠性，还可以根据请求 URL 来提供负载均衡。

在 OpenFlow1.2 规范中定义了交换机对 multi-home 的支持，包括两种操作模型：

- 对等交互：所有的控制器都能读写交换机，需要之间进行同步；
- 主从交互：存在一个主控制器和多个从控制器。

以上两种模型都可以采用。在主从模型情况下，会容易避免相互竞争造成的逻辑紊乱。在 OpenFlow1.0 中不支持 multi-home，因此可以作为扩展来实现。

对于多个控制器之间的相互交互，需要同步以下的信息：

- 内存数据库中的拓扑信息；
- 交换机和主机在数据库中的记录；
- 配置文件；
- 交换机的主控制器；
- 用户数据库。

一般假设在各个节点上的路径计算是各自独立的。如果需要保持一致性，那么路径信息也需要进行同步。

使用 REST API 的应用在应用和控制器之间采用了非持久性连接，因此，当控制器断线后，应用将重新发起新的连接。如果在传输中发生失败，则将产生 HTTP 错误并采取纠错行为。

对于使用 OSGi 框架的应用来说，应用在某个控制器上运行，如果控制器断线，那么应用也随之停止工作。应用需要自己负责在多个控制器存在时的可靠性和多实例之间的状态同步。

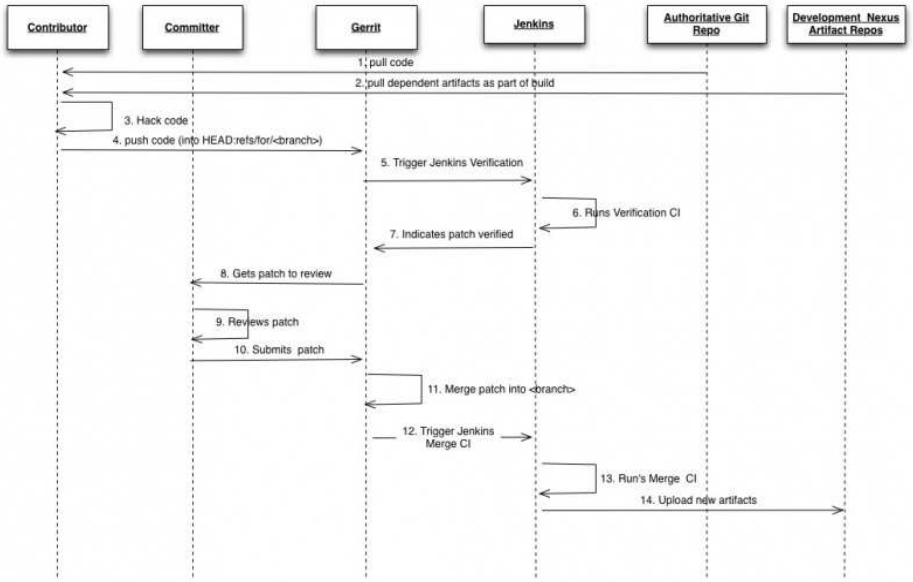
控制器提供集群服务，多个控制器之间可以同步状态和事件。同时提供了交互（transaction）API 来维护集群中节点之间的交互。

2.4 开发框架概述

包括如下几个部分。

版本管理：git：https://git.opendaylight.org/

代码 review：Gerrit：https://git.opendaylight.org/gerrit/
连续集成：Jenkins：https://jenkins.opendaylight.org/
人工 Repo：Nexus：https://nexus.opendaylight.org/
质量管理：Sonar：https://nexus.opendaylight.org/
Bug 跟踪：Bugzilla：http://bugs.opendaylight.org/
Wiki：MediaWiki：https://wiki.opendaylight.org/
整体的代码流如图表 12 所示。



图表 12 整体代码流

首先，用户通过认证的 git 方式来获取代码；
从开发 Nexus 的库中获取需要的依赖；
修改和编写代码；
提交代码；
触发 Jenkins 审查；
进行连续集成；
提交 patch；
审查 path；
提交 patch；
合并 patch 到 branch 中；
触发 Jenkins 的合并 CI；
检查需要的新的依赖；
提交新的依赖。

2.5 开发和更新代码

2.5.1 通过 CLI

用 git 从库中获取代码：

```
git clone ssh://<username>@git.opendaylight.org:29418/controller.git
```

匿名获取代码可以执行：

```
git clone https://git.opendaylight.org/gerrit/p/controller.git
```

配置 Gerrit Change-id 提交信息。

```
cd controller
scp -p -P 29418 <username>@git.opendaylight.org:hooks/commit-msg .git/hooks/
chmod 755 .git/hooks/commit-msg
```

编译代码

```
cd opendaylight/distribution/opendaylight/
mvn clean install [-DskipTests]
```

运行控制器

```
cd  target/distribution.opendaylight-0.1.0-SNAPSHOT-osgipackage/opendaylight/
./run.sh
```

提交代码

```
git commit --signoff
```

从代码库获取最新代码：

```
git pull ssh://<username>@git.opendaylight.org:29418/controller.git HEAD:refs/for/master
```

推送代码到代码库：

```
git push  ssh://<username>@git.opendaylight.org:29418/controller.git HEAD:refs/for/master
```

之后可以登录 Gerrit 查看提交信息。

2.5.2 通过 Eclipse

2.5.2.1 配置 eclipse

首先通过 git 下载代码。

打开 eclipse，安装 maven 插件。

Eclipse->help->install new software。

添加源 http://download.eclipse.org/technology/m2e/releases，搜索 m2e 和 m2e-slf4j，都安装 1.2.0 版本。然后完成安装。

重启 eclipse。

导入 git 下载的项目。

Eclipse->file->import->maven-> Existing Maven Projects，找到 opendaylight/distribution/opendaylight 目录，完成。如果询问是否安装 Tycho，选择是。

2.6 编译代码

2.6.1 通过 CLI

进入 opendaylight\distribution\opendaylight 目录下，执行如下命令来干净编译整个项目（需

要先安装 maven）。

```
mvn clean install
```

编译后生成的可执行包在 opendaylight\distribution\opendaylight\target 目录下。

2.6.2 通过 Eclipse

导入到 eclipse 之后在 run 配置中，执行 opendaylight-assembleit 是干净编译整个项目。几个 target 的含义如下。

opendaylight-application.launch => 运行控制器。

opendaylight-assembleit-fast.launch => 仅编译所选资源 (Project / Bundle)。

opendaylight-assembleit-noclean.launch =>编译所有 bundle，但不执行 clean。

opendaylight-assembleit-skiput.launch =>编译所有 bundle，但不进行 Unit Tests。

opendaylight-assembleit-sonar.launch =>编译所有 bundle 并运行 Sonar (Code-Coverage, Static-Analysis tool)。

opendaylight-assembleit.launch => 干净编译所有 bundle。

opendaylight-sonar-fast.launch =>仅对所选的资源运行 Sonar 任务。

opendaylight-sonar.launch => 执行所有的 Sonar 任务。

2.7 示例应用

2.7.1 Simple Forwarding Application

2.7.2 Statistics Application

2.7.2.1 JAXB 统计客户端

下面介绍一个基于 JAXB 的应用，使用了统计模块。

```
package org.opendaylight.controller.topology;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import org.apache.commons.codec.binary.Base64;
import org.opendaylight.controller.sal.reader.FlowOnNode;
import org.opendaylight.controller.statistics.northbound.AllFlowStatistics;
import org.opendaylight.controller.statistics.northbound.FlowStatistics;

public class JAXBStatisticsClient {
```

```
public static void main(String[] args) {

System.out.println("Starting Statistics JAXB client.");

String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

URL url;
try {
url = new java.net.URL(baseURL + "/" + containerName + "/flowstats");

String authString = user + ":" + password;
byte[] authEncBytes = Base64.encodeBase64(authString.getBytes());
String authStringEnc = new String(authEncBytes);
URLConnection connection = url.openConnection();
connection.setRequestProperty("Authorization", "Basic "
+ authStringEnc);

connection.setRequestProperty("Content-Type", "application/xml");
connection.setRequestProperty("Accept", "application/xml");

connection.connect();

JAXBContext context = JAXBContext.newInstance(AllFlowStatistics.class);
Unmarshaller unmarshaller = context.createUnmarshaller();

InputStream inputStream = connection.getInputStream();

AllFlowStatistics result = (AllFlowStatistics) unmarshaller.unmarshal(inputStrea
m);

System.out.println("We have these statistics:");
```

```
for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t" + flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}

} catch (Exception e) {
System.out.println(e.getLocalizedMessage());
}
}
}
```

下面具体分析代码的工作过程，首先是创建连接，练到统计模块上。

```
String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

URL url;
try {
url = new java.net.URL(baseURL + "/" + containerName + "/flowstats");

String authString = user + ":" + password;
byte[] authEncBytes = Base64.encodeBase64(authString.getBytes());
String authStringEnc = new String(authEncBytes);
URLConnection connection = url.openConnection();
connection.setRequestProperty("Authorization", "Basic "
+ authStringEnc);
```

```
connection.setRequestProperty("Content-Type", "application/xml");
connection.setRequestProperty("Accept", "application/xml");

connection.connect();
```

为了获取流统计信息，需要创建一个 JAXB 上下文，然后数据被传输到 AllFlowStatistics 对象。

```
for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t" + flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}
```

2.7.2.2 Jersey 统计客户端

跟基于 JAXB 的统计客户端类似，代码为

```
package org.opendaylight.controller.topology;

import org.opendaylight.controller.sal.reader.FlowOnNode;
import org.opendaylight.controller.statistics.northbound.AllFlowStatistics;
import org.opendaylight.controller.statistics.northbound.FlowStatistics;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;

public class JerseyStatisticsClient {

 public static void main(String[] args) {

 System.out.println("Starting Topology JAXB client.");
```

```
String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

try {

Client client = com.sun.jersey.api.client.Client.create();
client.addFilter(new HTTPBasicAuthFilter(user, password));

AllFlowStatistics result = client.resource(
baseURL + "/" + containerName + "/flowstats").get(
AllFlowStatistics.class);

System.out.println("We have these statistics:");

for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t"
+ flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}

} catch (Exception e) {
System.out.println(e.getLocalizedMessage());
}
}
```

```
}
```

下面分析代码过程，首先，也是需要连接到统计模块。

```
String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
 String containerName = "default";
 String user = "admin";
 String password = "admin";

 try {

 Client client = com.sun.jersey.api.client.Client.create();
 client.addFilter(new HTTPBasicAuthFilter(user, password));
```

为了获取流信息，需要创建一个 Jersey 客户端对象，数据被存放到 AllFlowStatistics 对象中。

```
AllFlowStatistics result = client.resource(
 baseURL + "/" + containerName + "/flowstats").get(
 AllFlowStatistics.class);
```

AllFlowStatistics 对象被转化为一个 FlowStatistics 对象，结果从 FlowStatistic 对象中获取，可以通过 get 方法来实现。

```
for (FlowStatistics statistics : result.getFlowStatistics()) {
 System.out.println(statistics.getNode().getNodeIDString());
 System.out.println(statistics.getNode().getType());
 for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
 System.out.println("\t" + flowOnNode.getByteCount());
 System.out.println("\t"
 + flowOnNode.getDurationNanoseconds());
 System.out.println("\t" + flowOnNode.getDurationSeconds());
 System.out.println("\t" + flowOnNode.getPacketCount());
 System.out.println("\t" + flowOnNode.getTableId());
 System.out.println("\t" + flowOnNode.getFlow());
 }
 }
```

2.7.3 Load Balancer Application

应用可以通过源地址和源端口将流量负载均衡到后端的服务器上。该服务被动的安装流表规则，将所有的特定来源地址和端口的数据包发送到合适的后端服务器上。该服务可以通过 REST API 来被外部程序配置。

要使用这个服务，首先所有客户端需要使用一个 VIPP 作为目标地址。VIP 包括虚拟 IP、

端口和协议。

一些前提假设：

- 同一个服务器池可以被分配一个或多个 VIP，但同一个池必须使用同样的均衡策略。
- 对于每一个 VIP，最多分配一个服务器池。
- 所有的流表项默认超时时间为 5 秒。
- 到达 VIP 的网包在离开 OpenFlow 机器的时候必须从进入的交换机离开。
- 删除 VIP、服务器池，或从池中删除一个服务器的时候，服务并不删除之前添加的流表项，这些流表项通过自动超时来删除。

2.7.3.1 REST API

表格 1 负载均衡服务的 REST API

Description	URI	Type	Request Body/Arguments	Response Codes
List details of all existing pools	/one/nb/v2/lb/{container-name*}/	GET		200 ("Operation successful") 404 ("The containerName is not found") 503 "Load balancer service is unavailable")
List details of all existing VIPs	/one/nb/v2/lb/{container-name}/vips	GET		200 ("Operation successful") 404 ("The containerName is not found") 503 ("Load balancer service is unavailable")

Create pool	/one/nb/v2/lb/{container-name}/create/pool	POST	{
			"name":"",
			"lbmethod":""
			}
				201 ("Pool created successfully")
				404 ("The containerName not found")
				503 ("Load balancer service is unavailable")
				409 ("Pool already exist")
				415 ("Invalid input data")

Delete pool	/one/nb/v2/lb/{container-name}/delete/pool/{pool-name}	DELETE		200 ("Pool deleted successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("Pool not found") 500 ("Failed to delete pool")
Create VIP	/one/nb/v2/lb/{container-name}/create/vip	POST	{ "name":"", "ip":"ip in (xxx.xxx.xxx.xxx) fo	201 ("VIP created successfully") 404 ("The containerName not found")

			r m a t " , "protocol":"TCP / U D P " , "port":"any valid port number " , "poolname":"" (optional) }	503 ("Load balancer service is unavailable") 409 ("VIP already exists ") 415 ("Invalid input data ")
Update VIP	/one/nb/v2/lb/{container-name}/update/vip	P U T	{ "name":"", "poolname":"" }	201 ("VIP updated successfully") 404 ("The containerName not found") 503 ("VIP not found") 404 ("Pool not found") 405 ("Pool already attached to the VIP") 415 ("Invalid input name")
Delete VIP	/one/nb/v2/lb/{container-name}/delete/vip/{vip-name}	D E LETE		200 ("VIP deleted successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("VIP not found") 500 ("Failed to delete VIP")
Create pool member	/one/nb/v2/lb/{container-name}/create/poolmember	P O ST	{ "name":"", "ip":"ip in (xxx.xxx.xxx.xxx) format ", "poolname":"existing pool name " }	201 ("Pool member created successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("Pool not found") 409 ("Pool member already exists") 415 ("Invalid input data ")
Delete pool member	/one/nb/v2/lb/{container-name}/delete/poolmember/{pool-member-name}/{pool-name}	D E LETE		200 ("Pool member deleted successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("Pool member not

				found") 404 ("Pool not found")

2.7.3.2 使用负载均衡

运行控制器，确保 samples.loadbalancer 和 samples.loadbalancer.northbound 模块都被加载。

运行 mininet，连接到控制器，例如创建包含 16 台主机（10.0.0.1~10.0.0.16/8）的树状拓扑。

```
mn --topo=tree,2,4 --controller=remote,ip=<Host IP where controller is running>,port=6633
```

在控制器的 web ui 中添加网关（同一网段的未占用的 ip），之后各个主机可以互通。

创建轮询策略的负载均衡服务。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://<Controller_IP>:8080/one/nb/v2/lb/default/create/pool -d '{"name":"PoolRR","lbmethod":"roundrobin"}'
```

创建 VIP 10.0.0.20。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://Controller_IP:8080/one/nb/v2/lb/default/create/vip -d '{"name":"VIP-RR","ip":"10.0.0.20","protocol":"TCP","port":"5550"}'
```

添加 10.0.0.2 和 10.0.0.3 到池中。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://Controller_IP:8080/one/nb/v2/lb/default/create/poolmember -d '{"name":"PM2","ip":"10.0.0.2","poolname":"PoolRR"}'

curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://Controller_IP:8080/one/nb/v2/lb/default/create/poolmember -d '{"name":"PM3","ip":"10.0.0.3","poolname":"PoolRR"}'
```

因为 VIP 在网络中实际并不存在，无法解析 ARP 请求，因此需要手动添加对象的表项。

在 h1 上启动 xterm，添加 VIP 的 mac 表项。

```
arp -s 10.0.0.20 00:00:10:00:00:20
```

在 h2 到 h4 上启动 xterm，开启 iperf 服务端，例如监听 5550 端口（iperf –s –p 5550）。

在 h1 上启动 iperf 发送请求到 VIP（iperf -c 10.0.0.20 -p 5550）。

此时请求被发送到了 h2，结束后再次发送请求，则被发送到了 h3。

可以通过下面的命令来删除池中的成员

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/poolmember/PM2/PoolRR'
```

删除 VIP。

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/vip/VIP-RR
```

删除资源池。

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/pool/PoolRR
```

2.8 库函数

2.8.1 C 客户端库

C 模块能生成 ANSI-C 兼容的 C 代码，可以跟 libxml2 结合，（解）序列化 XML 为 REST 资源。

生成的 C 代码依赖 XML Reader API 和 XML Writer API，以及<time.h>、<string.h>和<stdlib.h>等标准的 C 库。

REST XML 例子：

```
#include <full.c>
//...

xmlTextReaderPtr reader = ...; //set up the reader to the url.
full_ns0_edgeProps *response_element = ...;
response_element = xml_read_full_ns0_edgeProps(reader);

//handle the response as needed...

//free the full_ns0_edgeProps
free_full_ns0_edgeProps(response_element);
```

2.8.2 .NET 客户端库

.NET 客户端库定义了跟 XML 相互转化的类。例子：

```
//read a resource from a REST url
 Uri uri = new Uri(...);

 XmlSerializer s = new XmlSerializer(
   typeof( EdgeProps )
 );

   //Create the request object
 WebRequest req = WebRequest.Create(uri);
 WebResponse resp = req.GetResponse();
 Stream stream = resp.GetResponseStream();
 TextReader r = new StreamReader( stream );

 EdgeProps order = (EdgeProps) s.Deserialize( r );

 //handle the result as needed...
```

2.8.3 Java 客户端库

Java 客户端库用于访问应用的 Web 服务 API。

JAX-WS 客户端库用于提供可以利用 JAXB 跟 XML 之间相互转化的 java 对象。

Raw JAXB 的 REST 代码例子：

```
java.net.URL url = new java.net.URL(baseURL + "/{containerName}");
 JAXBContext context = JAXBContext.newInstance( EdgeProps.class );
 java.net.URLConnection connection = url.openConnection();
 connection.connect();

 Unmarshaller unmarshaller = context.createUnmarshaller();
 EdgeProps result = (EdgeProps) unmarshaller.unmarshal( connection.getInputS
tream() );
 //handle the result as needed...
```

Jersey 客户端的 REST 代码例子：

```
</nowiki>
com.sun.jersey.api.client.Client client = com.sun.jersey.api.client.Client.create();
```

```
EdgeProps result = client.resource(baseUrl + "/{containerName}")

 .get(EdgeProps.class);

//handle the result as needed...
</nowiki>
```

2.8.4 Java JSON 客户端库

提供了跟 jackson 之间转化的 java 对象。代码例子：

```
java.net.URL url = new java.net.URL(baseURL + "/{containerName}");
 ObjectMapper mapper = new ObjectMapper();
 java.net.URLConnection connection = url.openConnection();
 connection.connect();

 EdgeProps result = (EdgeProps) mapper.readValue( connection.getInputStrea
m(), EdgeProps.class );
 //handle the result as needed...
```

2.8.5 Objective C 客户端库

Objective C 模块必须生成能用于 libxml2 的 Objective C 类和序列化相关函数。

生成的 Objective C 源代码依赖于 XML Reader API 和 XML Writer API，和基本的 OpenStep 基础类。例子：

```
#import <full.h>
 //...

 FULLNS0EdgeProps *responseElement;
 NSData *responseData; //data holding the XML from the response.
 NSURL *baseURL = ...; //the base url including the host and subpath.
 NSURL *url = [NSURL URLWithString: @"/{containerName}" relativeToURL: bas
eURL];
 NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:
url];
 NSURLResponse *response = nil;
 NSError *error = NULL;
 [request setHTTPMethod: @"GET"];
```

```
//this example uses a synchronous request,
//but you'll probably want to use an asynchronous call
responseData = [NSURLConnection sendSynchronousRequest:request returnin
gResponse:&response  error:&error];
FULLNS0EdgeProps *responseElement = [FULLNS0EdgeProps readFromXML: r
esponseData];
[responseElement retain];

//handle the response as needed...
```

2.9 REST 调用和认证

REST API 包括多个模块。同时提供了 HTTP Digest 认证的 REST 认证。管理员用户可以通过 web 来管理用户。以后将支持 HTTPs，并把 REST API 迁移到 HTTP Basic。

2.9.1 Topology REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/topology/target/site/wsdocs/index.html。

2.9.2 Host Tracker REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/hosttracker/target/site/wsdocs/index.html。

2.9.3 Flow Programmer REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/flowprogrammer/target/site/wsdocs/index.html。

2.9.4 Static Routing REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/staticrouting/target/site/wsdocs/index.html。

2.9.5 Statistics REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/statistics/target/site/wsdocs/index.html。

2.9.6 Subnets REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/subnets/target/site/wsdocs/index.html。

2.9.7 Switch Manager REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/switchmanager/target/site/wsdocs/index.html。

2.10 Java API

参考自动生成的 javadoc：https://jenkins.opendaylight.org/controller/job/controller-merge/lastSuccessfulBuild/artifact/opendaylight/distribution/opendaylight/target/apidocs/index.html。

2.11 拓扑

Open Daylight Controller 提供了对物理网络的集中式的逻辑拓扑。并且可以更改转发的规则。控制器实现拓扑是基于 LLDP 消息。

Web UI 中可以看到连接到控制器的交换机信息以及主机的信息。

所有拓扑信息是由 Topology Manager 来维护的。

2.12 集成测试