

# 山东大学

## 毕业论文(设计)

论文（设计）题目：

**SDN 环境集中控制下的分布式路由协议设计与实现**

姓 名 张振国  
学 号 201400301174  
学 院 山东大学软件学院  
专 业 软件工程  
年 级 2014 级  
指导教师 朱方金

2018 年 6 月 5 日

## 山东大学毕业设计（论文）成绩评定表

学院： 软件学院

专业： 软件工程

年级： 2014

学号	201400301174	姓名	张振国	设计（论文）成绩	
设计（论文）题目		SDN 环境集中控制下分布式路由协议设计与实现			
指导教师评语					
	评定成绩：		签名：		年 月 日
评阅人评语					
	评定成绩：		签名：		年 月 日
答辩小组评语					
	答辩成绩：		组长签名：		年 月 日

注：设计（论文）成绩=指导教师评定成绩（30%）+评阅人评定成绩（30%）+答辩成绩（40%）

## 目 录

摘 要.....	1
ABSTRACT.....	2
第 1 章.....	绪论 3
1.1 sdn 集中控制下的分布式网络开发背景.....	3
1.2 国内外研究现状 .....	4
1.3 本文解决的主要问题 .....	4
1.4 本文的主要工作 .....	6
1.5 论文的组织结构 .....	6
第 2 章 SDN 技术 .....	7
2.1 SDN 技术简介.....	7
2.1 SDN 技术遇到的几个问题.....	8
第 3 章 SDN 集中控制下的分布式路由协议介绍 .....	10
3.1 集中式网络与分布式网络简介 .....	10
3.2 Fibbing——集中控制下的分布式路由协议 .....	10
3.2 Fibbing——控制集中下的分布式路由协议解决的问题 .....	11
3.3 Fibbing 技术的应用环境 .....	12
3.3.1 应用环境整体介绍 .....	12
3.3.2 应用环境各个部分介绍.....	12
第 4 章 ODL+Mininet+OVS 试验环境设计及搭建 .....	14
4.1 测试环境分析及改良优化 .....	14
4.1.1 测试环境分析.....	14
4.1.2 整体测试环境简介.....	15
4.2 ODL+Mininet+OVS 工具基本介绍.....	15
4.2.1 Opendaylight 基本介绍 .....	15
4.2.2 Mininet 基本介绍 .....	16
4.2.3 OpenVswitch 基本介绍 .....	17
4.3 试验环境各工具的安装测试 .....	18

4.3.1 Opendaylight 的安装 .....	18
4.3.2 Mininet 的安装 .....	24
4.4 试验环境各部分工具间的连接以及测试 .....	26
4.4.1 试验环境各部分之间的连接及测试 .....	26
4.4.2 连接时可能会碰到的问题 .....	33
第 5 章 OpenDaylight 插件的设计及开发 .....	34
5.1 《论文》相关代码描述 .....	35
5.1.1 代码整体架构介绍 .....	35
5.1.2 代码具体部分介绍 .....	36
5.2 ODL 插件开发 .....	38
5.2.1 相关技术概念 .....	38
5.2.2 插件开发流程 .....	41
第 6 章 使用 Java 开发接入 ODL 插件与 Python 核心代码 .....	51
6.1 使用 Java 开发接入 ODL 接口 .....	51
6.1.1 接口连接参数分析 .....	51
6.1.2 接口数据格式分析 .....	57
6.1.3 代码具体实现 .....	60
6.2 使用 Java 进行 Python 调用 .....	61
第 7 章 结论 .....	63
致谢 .....	64
参考文献 .....	65
附录 1 英文原文 .....	66
附录 2 译文 .....	71

# SDN 环境集中控制下分布式路由协议设计与实现

## 摘 要

SDN[1], Software Defined Network, 软件定义网络, 作为一个新型网络创新架构, 承载着颠覆传统网络架构, 将网络架构升级为全新分布式下集中的未来架构的伟大理想, 在对未来网络研究的领域中扮演着越来越重要的角色。作为一个新兴的网络架构思想, 也许它目前的设计还不太成熟, 但是有能力颠覆当前已经疲惫不堪的传统分布式架构, 给人以未来网络的曙光。

本文是关于 SDN 中出现的一个具体问题展开具体的分析。当前 SDN 网络的发展遇到了很大的瓶颈, 其中之一就是由于可扩展性问题, 无法在当前传统网络架构[2]上进行部署和发展, 只能在一些小型网络中搭建使用, 因此 SDN 技术在大型网络的发展方向上遇到了障碍。

而本文学习了国外普林斯顿大学教授 Jennifer Rexford 提到的新思想----Fibbing 欺骗技术[3], 基于 SDN 控制集中环境下进行分布式管理, 对其可行性与尝试在自己搭建的网络上进行部署, 验证了该论文所阐述思想的可行性及优越性, 并对此阐述一些自己的想法, 以及在部署该论文思想到试验环境中的步骤和所遇到的问题进行探讨。

**关键字:** SDN; 控制集中的分布式路由协议;

## **ABSTRACT**

Software Defined Network, as a new technology in future network, it gives us the sunshine to the future network structure. Although it is borned not so long, it bring us network developers a new idea, and new way to change existing traditional network structure, as it no longer affords the heavy and enormous internet flows.

This article is a detailed analysis of a specific issue that arises in SDN. Currently, the development of SDN networks encounters a big bottleneck. One of them is due to the scalability problem. It cannot be deployed and developed on the current traditional network architecture and can only be used in some small networks. Therefore, SDN technology is used in large networks. The direction of development encountered obstacles.

This article is based on the new ideas mentioned in the paper “Central Control Over Distributed Routing” by Professor Jennifer Rexford of Princeton University abroad. Based on the SDN control centralized environment for distributed management, the feasibility and attempt to carry out their own network. The deployment verifies the feasibility and superiority of the ideas described in this paper, and elaborates some of their own ideas, and discusses the steps in deploying the paper's ideas into the experimental environment and the problems encountered.

## 第1章 绪论

随着 SDN 技术的不断兴起, SDN 技术对未来网络产生了越来越大的影响, 是未来网络架构发展的一个方向和趋势。本章将介绍 SDN 集中控制下的分布式网络的产生背景, 当前网络界对分布集中的网络架构的研究状况, 以及本文对该问题的探讨和主要的工作及解决问题。

### 1.1 sdn 集中控制下的分布式网络开发背景

SDN 技术从开始逐渐兴起到发展至今, 成长了很多, 也经历了很多波折。到现在为止 SDN 技术依然面临很多问题, 经历着各方面的考验。传统的网络架构为分布式, 网络中的每个节点都参与网络流量的转发, 并且自己独立计算转发路径, 是一个完全分布式的架构。虽然这种传统架构方便搭建, 只需要同一种协议即可支撑整个网络的运行, 并且单独节点单独管理, 当某个节点压力较大可以单独增加配置, 但是存在很多缺点, 每个节点都运行相同的协议, 耦合度太高, 每当要为 1 个节点升级的时候需要对网络中每个节点升级或者增加配置以使其能够支持该升级功能; 因此可扩展性也很差, 不能实现灵活的单节点部署。

不同于传统的全分分布式的网络架构, SDN 技术主张网络中的节点进行路由和转发进行分离, 节点只保留转发功能, 路由计算功能则由统一的控制器集中控制计算。这样一来便可在节点保留基本转发功能, 使网络基本转发功能得以保证的基础上, 添加控制器来实现对整个网络中节点的集中控制。传统网络架构需要每个节点单独计算网络路由转发路径, 而 SDN 中只需控制器来计算, 省下了很大一部分的硬件需求。并且由于有了控制器, 从而便于网络管理员对整个网络节点的管理。只需登录控制器便能监控控制器控制的网络范围, 无需对单个节点进行管理配置。

但这样也存在一个问题, 就是当 SDN 控制器集中控制的时候, 所有基础节点会向控制器进行交互, 从而在控制器附近的网路上产生大量的流量, 可能会造成拥塞等问题。再者, 由于当前网络节点设备的生产制造是由不同的厂家进行设计生产的, 对单个厂家路由器的升级会设计其产业利益的问题, 因此很难进行全部网络节点设备的升级, 以使其能够支持 SDN 相关协议。所以这也很大程度上限制了 SDN 技术在近期在大范围网络架构上的延伸和拓展。

## 1.2 国内外研究现状

SDN 最开始发展的时候起源于“可编程网络”的概念，后来由斯坦福大学领导了一个关于网络安全的项目，在此期间对网络流安全控制策略的学习研究中收到启发，产生 OpenFlow 协议的概念，而 2008 年发表的一篇 OpenFlow 概念的论文，标志着 SDN 技术的正式到来，SDN 时代由此开启。

由于 SDN 技术划时代的到来，可能会掀起未来网络架构更新的一番热潮，因此国内外很多企业和厂商都非常关注 SDN 技术的发展，并且也积极投入到 SDN 技术的研究和研发中来。

现如今，SDN 网络开放式网络基金会 (Open Networking Foundation, 简称 ONF)，定制和推广了包括 OpenFlow 标准、OpenFlow 配置协议和 SDN 白皮书等等 SDN 相关标准，这使 SDN 更加规范化，具体化，使 SDN 技术逐渐为世界研究者爱好者们所了解和学习，使 SDN 逐渐成为全球开放网络架构和网络虚拟化领域的研究热点。

在国内，清华大学带头，联合中科院计算所、北邮、东南大学、北京大学等单位参与开展了类 SDN 思想的“未来网络体系结构和创新环境” 863 项目研究。国外，美国 GENI、Internet2、欧洲 OFELIA 和日本的 JGN2plus 先后展开对 SDN 的研究和部署。

在产业界，研究机构以及 SDN 新型技术更是层出不穷。谷歌公司对 SDN 技术进行研究，推出 P4 协议技术，中国华为也不甘示弱，推出 POF 南向协议技术。还有其他的一些比如 OpenDayLight, FloodLight 等等控制器也是如雨后春笋般出现。SDN 思想变得更加丰富，实现变得更加广泛，在一些小型网络架构中 SDN 技术也逐渐部署发展开来。目前，SDN 技术在校园网，数据中心互联等方向取得了一些好的发展，不远的未来，SDN 的发展会向云数据中心网络，运营商网络，未来互联网等方向发展，可谓前途无限。

## 1.3 本文解决的主要问题

SDN 现在发展遇到了很多问题。比如控制器控制节点的规模，这是限制 SDN 部署在大型网络上的主要瓶颈；再比如当前网络设备不能很好的支持 SDN 相关



协议，对网络设备进行升级涉及到了每个网络设备生产制造商的利益问题，因此也限制了新型网络设备的发展及对 SDN 协议的支持。因此 SDN 的发展也产生了 2 种不同的思想观点。一种认为 SDN 协议应该完全颠覆传统网络，从底层做起，从底层实现对整体 SDN 协议的支持。另一种观点则认为 SDN 技术应该在现有网络架构的基础上进行改良，使其在不改变原来传统分布式架构的基础上，实现对 SDN 相关协议的支持，从而逐步完成从传统网络架构到基于 SDN 协议的传统分布式网络架构，再到支持 SDN 协议的新型互联网络架构的发展。

因此，为了能够使传统分布式网络在不改变原有架构的基础上支持 SDN 思想，并且可以使用 SDN 在传统网分布式网络上实现负载均衡，Jennifer 教授在《Central Control Over Distributed Routing》文章中提到的 Fibbing 思想，原文中的 Fibbing 是在安装 Quagga 软件将物理机模拟为节点路由器从而进行 Fibbing 的运行和测试。Jennifer 教授所发表的论文《Central Control Over Distributed Routing》，便是实现了后者的思想。她提出的思想是在现有的传统网络架构的基础上，增添控制器对网络节点的网络路由路径进行干涉，从而在原有协议的基础上实现对网络路径计算的控制，实现了 SDN 环境集中控制下的分布式路由架构，可以使用 SDN 的思想很好的解决传统分布式网络负载不均衡的问题。

本文要解决的主要问题是实现在运行 OSPF 协议的传统分布式网络架构的基础上，添加节点控制器实现对网络节点路由计算的修改，以此种方法实现控制器对网络节点的控制。

在路由协议设计方面，本文采用了 Jennifer 教授发表的论文《Central Control Over Distributed Routing》中所提供的方法。

在本地试验环境搭建方面，本文采用 OpenDaylight 控制器+Mininet 轻量级虚拟网络测试平台+OpenVswitch 节点交换机组成的试验环境，使用系统为 Windows10+Vmware12 安装 Ubuntu 16-kylin 系统。

在实现方面，本文使用 eclipse-java 对 Opendaylight 控制器进行插件开发接入。

。

## 1.4 本文的主要工作

本文的主要任务就是对集中控制下的分布式路由协议进行探讨，理解其主要的解决问题的思路及方法，并分析测试环境的优缺点，给出自己的观点和看法；简要叙述当前比较火的其中一种集中控制的分布式网络技术 Fibbing 所用的测试环境，介绍自己搭建的试验环境的流程，对试验环境进行测试，以及将 Fibbing 技术所提供的主要思想的核心实现代码嵌入到自己搭建的试验环境中，使用自己搭建的网络测试环境，进行与 Fibbing 技术的集成，实现在自己搭建的网络测试环境中对 Fibbing 技术进行运行和测试，对运行结果进行分析和整理。

## 1.5 论文的组织结构

第一章绪论，主要描述SDN集中控制下的分布式网络的发展过程，研究现状以及所遇到的问题，然后给出本篇文章要解决的主要问题以及主要任务。

第二章SDN技术，主要讲述SDN技术的发展，以及当前SDN技术遇到的几个瓶颈问题，随后我们介绍的技术是解决瓶颈中的一个或几个问题。

第三章将对集中控制下的分布式路由协议记性探讨，主要介绍当前比较热门的一种Fibbing技术，对Fibbing技术给出详细的解读，主要解决问题和对应的解决方法，并对文章中所提到的试验案例进行分析。

第四章ODL+Mininet+OVS试验环境设计及搭建，主要介绍本地试验环境的搭建，包括OpenDaylight控制器，Mininet轻量级虚拟网络测试平台，OpenVswitch节点交换机的基本介绍，安装流程介绍，以及安装完成后各工具连接的介绍。

第五章是对OpenDaylight控制器插件设计及开发，主要是对Opendaylight各个插件的介绍，以及在自己对插件开发时的流程和所遇到的问题的介绍和整理。

第六章将介绍如何使用Java技术对OpenDaylight控制器进行接口接入，以及使用Java技术调用Python核心代码，从而完成整个项目的集成。

第七章结论，是对正片论文所做工作进行整理和总结，阐述论文工作遇到的问题及对SDN未来发展方向上的展望。

## 第 2 章 SDN 技术

SDN, Software Defined Network, 全称为软件定义网络, 是Emulex网络下的一种新型的网络创新框架, SDN是一种对未来网络的新思想和新构思, 是网络虚拟化的一种实现方式。本章将会对SDN技术进行简单的介绍, 以及介绍当前SDN技术遇到的几个问题。这几个问题中的一个或多个将会是本章所介绍的Fibbing技术的解决问题。

### 2.1 SDN 技术简介

SDN是一种新型网络架构的思想, 其核心思想就是进行数据分离。不同于传统的分布式网络, 单个节点路由器既实现了路由表的计算, 由提供数据包的转发功能, 集路由计算及数据转发于一身。SDN则是想将路由计算和数据转发功能分开, 分为2个层面。转发功能留在节点交换机上, 实现最基本的数据转发功能。而将路由计算集中起来, 集中放置到节点控制器中, 由节点控制器实现路由的计算和路由表的下发功能。

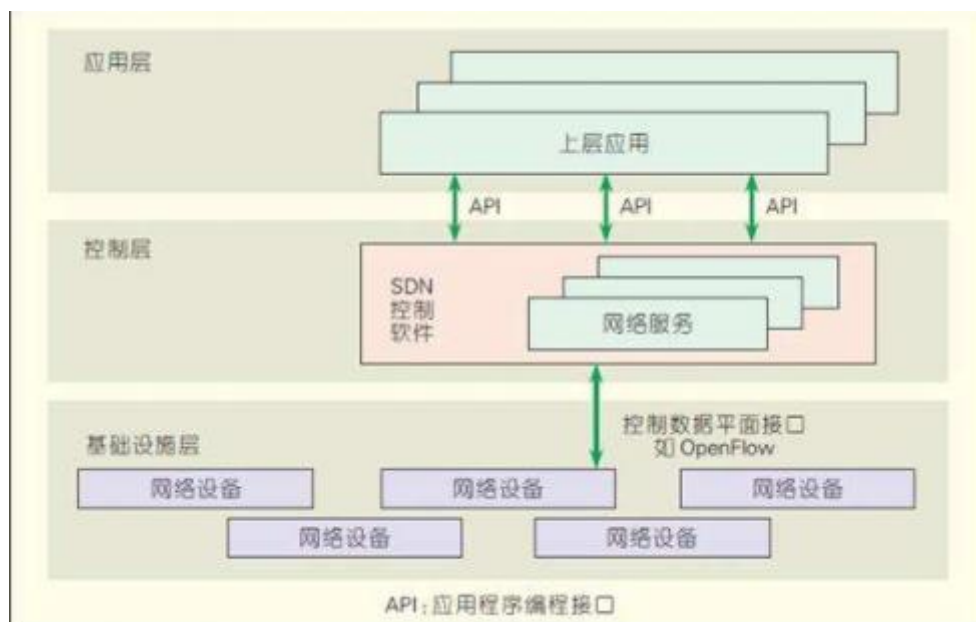


图2.1 SDN架构

如图2.1所示, SDN架构分为3层, 由上到下分为应用层, 控制层和基础设

施层，基础设施层即最基本的节点路由器，只提供数据包转发功能。控制层为节点控制器，负责链路路由的计算及转发，和基础设施层进行交互。应用层面向用户或者开发者，在应用层可以运行开发者们根据控制层提供的北向接口api开发的应用程序，使用户无需了解底层网络设备的运行，即可通过应用程序对底层设备进行管理和配置。控制层实现了应用层和基础设施层的过渡，与应用层的接口称为北向接口，与技术设施层的接口称为南向接口。

## 2.1 SDN 技术遇到的几个问题

SDN在不断发展的过程中，遇到了很多的挫折和阻碍。其中一个问题就是对大型网络的支持和对传统分布式网络的支持。由于SDN的思想是将路由功能和交换功能分离，分别在控制层和数据层实现。节点交换机或节点路由器处于数据层，只负责通过存储的转发表进行数据包的转发工作，而路由表的计算及转发表的生成则是由处于控制层的节点控制器进行整体的计算与下发服务。但是这样一来，节点控制器控制的网络的规模成为整个SDN网络的瓶颈，会产生鲁棒性问题。即随着网络规模的扩大，和控制器节点进行数据交互的流量越来越多，会导致网络控制器节点附近的网络压力增大，发生掉包甚至拥塞。

传统分布式网络也难于管理，每个节点运行相同的协议，如果要对每个节点进行升级则必须对全部的节点进行升级。并且如果要改变一个节点的配置则可能会不经意间影响其他节点的正常运行。而使用基于分布式下的控制集中的SDN思想则可以很有效的解决这个问题。

传统分布式网络没办法完全支持OpenFlow协议，也没有办法支持SDN的新思想以及产生的SDN的新技术，传统分布式网络的交换机硬件亟待升级和更新。但是由于网络中的交换机硬件是由不同厂商制造，和全部厂商进行沟通交流达成协议使其生产的交换机能够支持OpenFlow协议是不现实的，这牵扯到各个交换机生产厂家的利益问题。

并且在传统的分布式网络中，大都使用迪杰斯特拉算法进行路由的计算，计算的结果为最短路径。这样会使得在网络中权值较小的一部分链路会被很多流量使用，这条链路上的交换机的工作量会非常大，然而可能在这条链路周围的某

些链路虽然也能实现数据包的正确转发,但是却由于权值过大等其他原因使得链路空闲,没有流量经过。因此传统的分布式网络会存在负载不均衡的问题。但是SDN可以很好的解决这个问题。由于SDN是由一个控制器进行链路的计算和下发,所以可以在控制器中使用相关算法来解决负载不均衡的问题。

## 第3章 SDN 集中控制下的分布式路由协议介绍

本章将对 SDN 集中控制下的分布式路由协议进行简单的介绍。基于 SDN 集中控制的思想，会使传统的分布式网络拥有更加出色的性能，更易于管理等优点。除此之外，本章还将主要介绍一种 Fibbing 技术，它可以在传统的分布式网络的基础上，实现 SDN 集中控制的思想，从而很好的解决了 SDN 对传统网络的支持，能够实现传统网络向 SDN 新型网络架构的过渡，也解决了 SDN 面临的集中控制的瓶颈问题。

### 3.1 集中式网络与分布式网络简介

集中式网络，顾名思义，就是在网络上能够实现集中的管理。网络中的节点分为控制节点和功能节点。功能性节点的只是起到数据转发的基本功能，而控制节点则可以对网络中的节点进行控制，能够对每个节点控制集中到一个节点上，便于管理与控制的优点。缺点是集中式网络中控制节点附近的网络路径会很繁忙，有拥塞的风险，控制节点附近的流量大小成为整个网络的瓶颈。控制节点一旦失效，将会影响到整个网络的运行。

分布式网络，即网络中的每个网络节点地位都是平等的，不存在谁控制谁的说法，每个网络节点共同维护整个网络的正常运行。分布式网络有稳定的优点，即其中的节点有一个不能使用了，但是不影响整个网络的正常使用，有灵活和可伸缩的优点。

而集中控制下的分布式网络，则是兼具了集中式网络与分布式网络的有点，优势互补，解决了二者部分的缺点，使得整个网络即具有分布式网络的稳定性，又能通过控制节点进行集中控制，解决了传统网络上管理配置复杂的缺点，实现了统一的配置管理。

### 3.2 Fibbing——集中控制下的分布式路由协议

集中控制的路由优点是提升了灵活性，但是以牺牲分布式路由协议的健壮性为代价[4]。我们采用 Fibbing 技术[5]来实现在传统分布式网络上实现控制

集中。

Fibbing 可以创建一个虚拟的节点来欺骗节点路由器或节点交换机,通过适当的控制虚拟节点两边链路的权值,使节点交换机在进行网络路径计算的时候将虚拟节点加入到分发路径中,从而实现了对分布式网络的干预。当节点路由器要使用 OSPF 协议进行路由路径计算时,给节点路由器散布这些虚拟路由节点的虚假消息,从而影响节点路由器生成的路由表,使节点路由器根据路由表转发数据包时改变原来的转发路径,从而达到控制节点路由器的目的。

Fibbing 通过欺骗路由器,发布虚拟节点的消息,让路由器以为链路中有一些不存在的节点和联络,然后再基于传统网络中 OSPF 协议或者 IS-IS 协议这些分布式链路状态协议,进行转发路径的计算。所以, Fibbing 是容易表达的, Fibbing 只需要通过模仿节点路由器,分发一些数据包给网络中的节点,让节点以为 Fibbing 是个真实存在的路由器,那么 Fibbing 就可以成功设置虚拟节点。Fibbing 容错率高,即使设置的虚拟节点出现问题了,基于分布式链路状态协议的网络环境仍然可以依靠原有的路由算法进行路径选择与数据分发,而不是出错。Fibbing 适应性强, Fibbing 是工作在传统的支持链路状态协议的网络拓扑上的,因此实现之后可以直接使用。

### 3.2 Fibbing——控制集中下的分布式路由协议解决的问题

传统分布式网络不便于网络管理员对网络进行管理,因为没有有一个集中的控制节点,每个节点都需要单独进行配置,大大加大了传统的分布式网络管理的复杂度。而 SDN 集中控制的思想可以很好的解决这个问题。使用 SDN 思想,加入控制器来进行统一管理,可以很好的改善传统网络的这个缺点。

并且由于传统交换机等节点设备无法很好的支持 SDN 的 openflow 等相关协议,因此使得 SDN 思想无法很快的并且广泛的应用到当前网络中来,因此, Jennifer 教授提出这么一个观点,可以在原有的传统分布式网络的基础上,添加 SDN 思想,加入控制器,在不改变现有交换机和支持的 OSPF 等协议的基础上,使用控制器进行干预,从而在传统分布式网络环境下实现控制集中。Fibbing 思想应运而生。该篇论文就是以 Fibbing 的新思想新技术,来解决传统分布式

路由难于管理的问题。

### 3.3 Fibbing 技术的应用环境

#### 3.3.1 应用环境整体介绍

作为集中控制下的分布式路由协议，Fibbing 技术实现在使用 Quagga 搭建的 Linux 操作系统上，Fibbing 核心代码为 Python 编写，使用 C 语言进行和节点交换机中运行的 OSPF 协议进行交互。

#### 3.3.2 应用环境各个部分介绍

##### 1. Quagga 软件。

Quagga 是一款开源路由软件，功能强大，简单易用，可以通过简单的编译安装就可以让一台 Linux 系统的电脑成为一台路由器，并且提供命令行配置模型，也有图形用户界面，可以让用户方便的配置相关信息，可以动态配置或者静态配置路由功能。目前 Quagga 能够支持的协议有 ip,ospf-v2,ospf-v3,ripng, bgp 等协议。

Quagga主要结构是以zebra守护进程作为核心，其他动态路由模块RIP/OSPF/BGP这类的程序做为client;

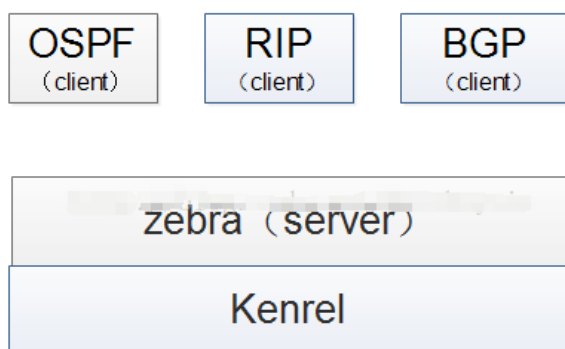


图 3.1 Quagga 的大体架构

##### 2. OSPF 协议

OSPF 协议，（Open Shortest Path First），开放最短路径优先协议，属于内部网关协议（Interior Gateway Protocol，简称 IGP）的一种，应用在单一的自



治系统中，用于节点交换机进行路由路径的计算，区别于 BGP（边界网关协议，Border Gateway Protocol）。该协议是链路状态路由协议的一种，使用 Dijkstra 算法（迪杰斯特拉算法）进行路由计算，其计算的结果为最短路径。每个节点通过迪杰斯特拉算法分别进行计算，不保留全部路径，只保留下一跳的目的地。该协议目前有 2 个版本，OSPF-v2 和 OSPF-v3，其中前者 OSPF-v2 应用在 IPv4 网络中，而后者 OSPF-v3 协议应用到 IPv6 网络中。

## 第 4 章 ODL+Mininet+OVS 试验环境设计及搭建

相比较而言, OpenDaylight+Mininet+Ovs 的测试环境具有易于搭建和测试的优点。使用 Quagga 软件将主机虚拟化为网络路由器, 虽然高度仿真, 测试结果精确, 并且有很好的可移植性, 但是对个人而言, 需要多台物理机的支持, 并且每台物理机上都需要手动进行配置, 操作繁琐。因此, 使用 odl+mininet+ovs 的试验环境可以很好的解决这个问题。该试验环境只需要在单个电脑上进行配置即可完成, mininet 提供的虚拟网络测试平台可以对真实网络进行模拟仿真, 配置简单有效。

### 4.1 测试环境分析及改良优化

#### 4.1.1 测试环境分析

上一部分说到 Jennifer 教授使用 Quagga 物理机形成的网络进行测试。该测试环境注重仿真, 通过 Quagga 软件将运行 Linux 系统的电脑虚拟成节点交换机, 使用多台电脑虚拟为交换机形成所需的拓扑网络结构, 从而在有真实物理电脑组成的真实网络中进行测试, 得出的数据更真实。

但该测试网络环境搭建比较复杂, 需要大量的物理机才能实现测试, 并且需要配置每一台物理机, 安装测试环境太过于复杂和昂贵, 以及可能不利于形成虚拟网络, 有网路上其他加入的节点的流量干扰。

基于以上几种原因, 本片论文准备采用 Opendaylight+Mininet 的测试环境, 使用运行在 Jdk 上的.opendaylight 作为控制器, 支持 SDN 和 openflow 协议。使用 Mininet 作为虚拟网络拓扑的搭建测试环境, Mininet 易于管理, 易于操作, 提供通用的命令行进行操作, 并且也可以使用图形用户界面进行拖拽, 实现网络拓扑的搭建, 并且 Mininet 上可以虚拟出 ovsk 节点, 支持 OpenFlow13 协议。

该测试环境简单操作, 易于管理, 能够很好的实现 Jennifer 教授《论文》中提到的 Fibbing 的测试, 因此本片论文采用此环境进行 Fibbing 的可行性和有效性的测试。

### 4.1.2 整体测试环境简介

本文章实现的测试环境整体为 opendaylight+mininet+OpenVswitch 的测试环境。其中 opendaylight 控制器运行在 windows 主机上，mininet 虚拟网络测试平台运行在 vmware 虚拟机上运行的 ubuntu-kylin 操作系统上。Openvswitch 交换机运行在 mininet 虚拟网络平台上，ovs 交换机只是 mininet 提供的一种支持 openflow 协议的交换机，并没有实体物理机和实体应用安装。

Vmware 虚拟机采用桥接的方式接入到网络中，因此 ubuntu 和 windows2 个系统可以看做是连接在同一个路由器下的 2 台不同的电脑系统，路由器使用 DHCP 协议自动分发局域网 ip。Mininet 和 opendaylight 之间的连接是通过远程调用的方式，通过路由器完成。

测试的时候，先由 Windows 主机启动 OpenDaylight 控制器，OpenDaylight 控制器自动实现对 6633 端口的监听。随后通过在 Mininet 虚拟网络测试平台上进行虚拟链路以及虚拟节点的搭建，搭建完成后启动 Mininet，在启动 mininet 的参数中设置 OpenDaylight 控制器的网络地址，实现远程连接。连接成功后，便可在 OpenDaylight 的控制台页面上看到 Mininet 的网络拓扑，并且可以使用 OpenDaylight 提供的相关控制功能对 Mininet 网络拓扑进行控制。

## 4.2 ODL+Mininet+OVS 工具基本介绍

### 4.2.1 Opendaylight 基本介绍

OpenDaylight 是由 Linux 基金会推出的一个开源项目，进行这个项目的目的就是要通过开源的方式，创建一个供应商支持框架，该框架需要所有供应商共同使用，而不依赖于任何一个供应商，让每个人都可以贡献自己的一份力量，一起推动 SDN 的发展和创新。

ODL 控制器是基于 Java 语言进行的开发，好处就是可移植性很强，只要有 Java 虚拟机的支持，便可以运行在任何环境上，目前最新版本支持的 Java 版本为 Java1.8。

ODL 控制器采用了 OSGi 框架，OSGI 框架是面向 Java 的，它优雅的实现了一个动态的组建模型，应用程序可以通过组件的形式进行安装和卸载，而对

其他功能不会产生任何影响。应用程序在 OpenDayLight 中以 Bundle 的形式存在，每一个 Bundle 就是一个应用程序，每个应用程序之间相互隔离，保证了程序功能可扩展性的问题，也可以通过访问接口的形式进行交互，方便各个应用程序之间的协同工作。

ODL 控制器还采用了 SAL (Service Abstracted Layer)，服务抽象层，SAL 对北向连接用户编写的应用程序，即功能模块，以插件的形式操作底层设备；南向连接多种协议，只要协议开发遵守 SAL 的规则，便可进行集成。SAL 为上层应用屏蔽了下层协议的差异，使得上层应用和下层协议分离开来，互不干扰，使北向应用和南向协议的开发者们可以专注于自己业务应用的开发，而不需要管具体的底层实现方式。

更多关于 ODL 的介绍，可以在 [opendaylight](https://www.opendaylight.org/) 官网 (<https://www.opendaylight.org/>) 中获取。

#### 4.2.2 Mininet 基本介绍

Mininet 是一款强大的网络仿真工具，通过使用 Mininet，我们可以很方便的在 Mininet 上实现对真实环境的仿真模拟，并且可以很方便的对方针网络环境进行操作和架构。

在 SDN 发展如日中天，如火如荼的时候，对开发者们而言，单纯建立一个实体物理机网络进行 SDN 相关开发的测试已经太过繁琐和昂贵，新的支持 SDN 的仿真测试工具一直被期盼着，等待着。而 Mininet 就在此时应运而生。

Mininet 可以虚拟出一个虚拟仿真测试网络，由一些终端节点，交换机，路由器组成。Mininet 轻量级的构造使得它自己可以很容易的构建出大型的网络拓扑，并且可以和真实的网络环境相媲美。

Mininet 可以很方便的创建一个支持 SDN 的虚拟拓扑，在这个网络拓扑中，每个终端节点可以和真实电脑一样，可以互相 ping，可以向外发送数据包，而此时产生的流量能够通过虚拟的网络拓扑上的交换机和路由器进行接收、处理和转发。Mininet 能够缩短测试周期，支持 Openflow，能够很好的将开发结果移植到真实的网络中；简单易用，在本地电脑上就可以安装使用，并且提供了传统的命令行操作和个性化的图形用户界面操作；支持 Python 编程，可以很好

的支持较为复杂的网络拓扑和测试环境的构建。

更多的 Mininet 介绍可去 Mininet 官网 (<http://mininet.org/>) 上阅读查看。

### 4.2.3 OpenVswitch 基本介绍

OpenVswitch 交换机, 简称 OVS 交换机, 是一种开源的虚拟交换软件, 是在开源的 Apache2.0 许可下的产品级质量的多层虚拟交换标准。OpenVswitch 交换机设计的目标是想通过编程来对交换机功能进行扩展, 以使大型网络中的节点交换机实现网络自动换, 可以自动进行配置, 管理和维护。并且 OpenVswitch 交换机支持很多标准的管理接口和协议, 比如 sFlow, SPAN, CLI, NetFlow, LACP 等等。

整个 OpenVswitch 交换机全部由 C 语言编写, 目前支持以下功能:

1. 支持有主框架和运行节点的标准 802.1Q 模型。
2. 在 upstream 交换机上 有或者没有 LACP 的 NIC 绑定。
3. 为 NetFlow, sFlow 和 mirroring 增加了可见性。
4. Qos (Quality of Service, 服务质量) 的配置, 附加策略等。
5. 支持基于 LISP 隧道, VXLAN, IPSEC 的 GRE。
6. 支持 802.1ag 的联通错误管理。
7. 支持 OpenFlow1.0 ++ 的大量插件。
8. 支持 C 和 Python 绑定的业务配置数据库。
9. 支持使用 Linux 内核模块进行高性能转发。

Openflow 可以作为一个在管理程序上运行的软开源, 也可以作为开关控制堆栈。OpenFlow 整个由 C 语言编写, 有很好的可移植性。并且 OpenVswitch 作为一种开源软件, 专门用于管理多租赁公共云计算环境, 能够比较方便的给网络管理员提供虚拟 Vm 之间和之内的流量的可见性和可控性。

更多关于 OpenVswitch 的介绍, 详细请访问 OpenVswitch 官方网站 <http://www.openvswitch.org/>。

## 4.3 试验环境各工具的安装测试

### 4.3.1 Opendaylight 的安装

#### 4.3.1.1 安装步骤

安装基本环境：

操作系统：windows10、

软件环境：JDK1.8，maven，opendaylight-carbon

安装软件获取：

JDK1.8 和 maven 自行安装，opendaylight 请访问官网地址获取安装：

<http://docs.opendaylight.org/en/latest/downloads.html>

本文中对 opendaylight 的操作由于不只是需要运行，还需要对 opendaylight 进行编码，因此需要使用源码进行安装，所以需要下载 opendaylight 源码，然后使用 maven 对源码进行编译及项目构建。

工程构建：

1). 由于 opendaylight 使用 maven 进行构建，而 opendaylight 所需资源不在默认仓库内，因此需要单独为 opendaylight 添加 maven 仓库。

2). Windows 下请找到 maven 安装目录，然后在 conf 文件夹下找到 settings.xml 文件，为了方便，可以直接使用以下链接文件替换 xml。

<https://raw.githubusercontent.com/opendaylight/odlparent/stable/carbon/settings.xml>

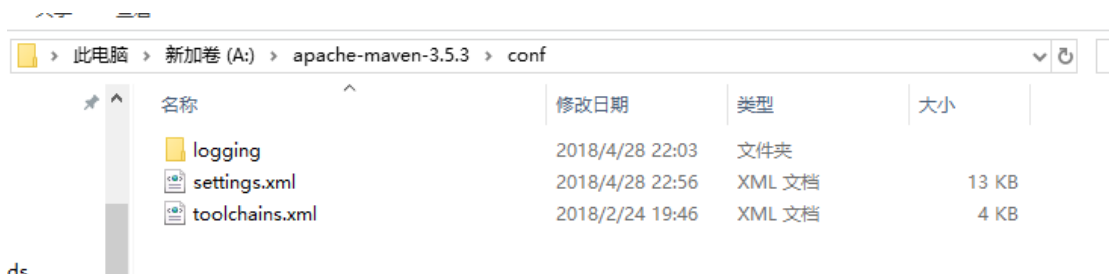


图 4.1 maven 安装目录

3). 根 opendaylight 官网上的官方文档提供的 Maven 进行项目骨架的生成:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
-DarchetypeGroupId=org.opendaylight.controller
-DarchetypeArtifactId=opendaylight-startup-archetype
-DarchetypeRepository=https://nexus.opendaylight.org/content/repositories/public
-DarchetypeVersion=1.3.2-Carbon
```

此处根据自己安装 opendaylight 版本的不同需要更换不同的参数配置, 各个参数说明如下:

DarchetypeGroupId: archetype 原型的 Group Id, 一般为网站名倒序

DarchetypeArtifactId: archetype 原型的 Artifact Id, 即项目 ID

DarchetypeRepository: archetype 原型的 maven 仓库链接

DarchetypeVersion: archetype 原型制定的 opendaylight 版本。

将此命令运行之后, 会出现让你输入你要搭建工程的基本信息, 如下:

```
Defined value for property 'groupId': org.opendaylight.topology
Defined value for property 'artifactId': topology
Defined value for property 'package': org.opendaylight.topology::
Defined value for property 'classPrefix': topology::
Defined value for property 'copyright': ZrbraDecoder
```

groupId 为网站链接的倒序, artifactId 为所要搭建的项目名称, 其他的默认即可, 到最后会提示你是否确认, 输入 Y 回车即可完成。

4). 第一次进行构建需要下载大量的依赖文件及项目架构文件。稍等一会, 即可显示构建成功, 成功之后在当前文件夹下查看当前文件夹下的文件列表, 即可发现一个叫 topology 的文件夹被生成了, 这就是你项目的整体。项目整体架构如图 4.2 所示。

新加卷 (Hi) > odl-carbon > topology >					▼	🔄	搜
名称	修改日期	类型	大小				
.settings	2018/5/4 22:28	文件夹					
api	2018/5/4 23:05	文件夹					
artifacts	2018/5/4 23:08	文件夹					
cli	2018/5/4 23:06	文件夹					
features	2018/5/4 23:06	文件夹					
impl	2018/5/4 23:06	文件夹					
it	2018/5/4 23:08	文件夹					
karaf	2018/5/4 23:07	文件夹					
src	2018/5/4 16:49	文件夹					
target	2018/5/4 23:08	文件夹					
.gitignore	2018/5/4 16:49	文本文档	1 KB				
.gitreview	2018/5/4 16:49	GITREVIEW 文件	1 KB				
.project	2018/5/4 22:28	PROJECT 文件	1 KB				
deploy-site.xml	2018/5/4 16:49	XML 文档	2 KB				
pom.xml	2018/5/4 16:49	XML 文档	4 KB				

图 4.2 编译完成后的项目目录

各个文件夹的作用分别为：

- api/ : YANG 模型目录
- artifacts/: 项目组件坐标管理器。
- Cli/: karaf 的 cli 命令代码开发目录。
- Features/: feature 组件的管理目录。
- Impl/: 业务逻辑代码的实现目录。
- It/: 集成测试目录。
- Karaf/: karaf 打包目录。
- Src/: 使用 eclipse 后提供的存放源文件的文件夹。
- Target/: 存放目标代码的文件夹。
- Deploy-site.xml: 部署的配置文件。
- Poe.xml: 项目的根 pom 文件，描述 maven 项目的基本信息。

5) . 进入已经搭建完成的项目架构中，cd topology，进入项目的根目录下，执行 mvn 命令：

```
Mvn clean install
```

来进行项目的整体编译。



需要注意的是，第一次编译会花费大量的时间，其中还包含了执行测试用例和 checktype 检查，因此要想缩短编译时间，可以使用如下 mvn 命令：

```
mvn clean install -DskipTests -Dmaven.javadoc.skip=true
-Dcheckstyle.skip=true
```

命令分析：

-DskipTests 表示跳过执行测试用例，

-Dmaven.javadoc.skip=true 表示跳过 javadoc

-Dcheckstyle.skip=true 表示跳过检查 checkstyle

在等待约 30 分钟左右之后即可编译完成，如果没有报错显示的话即表示项目成功构建完成。

#### 6) opendaylight 启动

在根目录下使用此命令运行 opendaylight 控制器：

Windows: karaf\target\assembly\bin\karaf.bat

Linux: sudo ./karaf/target/assembly/bin/karaf

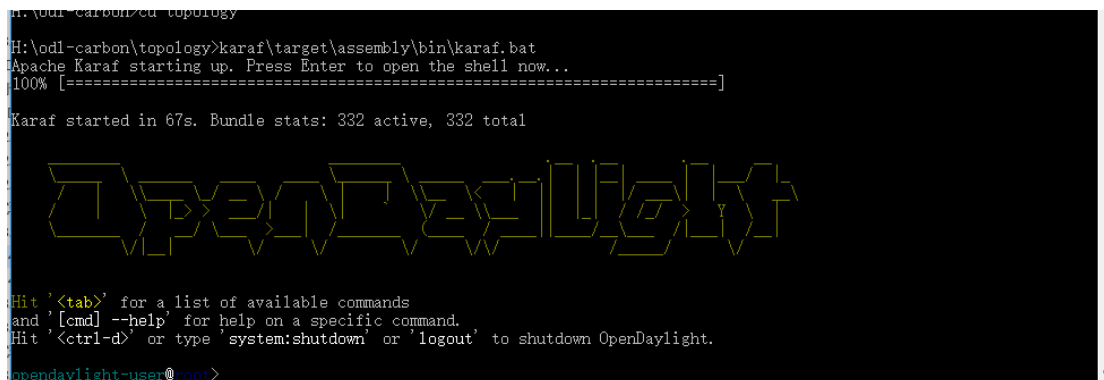


图 4.3 opendaylight 启动完成后界面

#### 7) 安装相关 bundle。

Bundle 简单来说即运行在 opendaylight 上的 java 应用程序，他们以插件的形式运行，每当需要一个功能时便自己手动安装该插件。

安装 REST API 插件。REST API 为 odl 支持北向的一个通用接口。

```
feature:install odl-restconf
```

安装 L2 交换机：

```
feature:install odl-l2switch-all
```

安装 OpenFlow 的支持:

```
feature:install odl-OpenFlowplugin-all
```

安装 DLUX 相关功能:

```
feature:install odl-dlux-all  
feature:install odl-mdsal-all
```

安装基于 Karaf 控制台的 ad-sal 功能, 其中包括 Connection Manager, Flows, NetWork, Container, 是支持南向接口的功能:

```
Feature:install odl-adsal-northbound
```

8) 安装情况测试及查看:

打开浏览器, 手动输入以下网址打开 opendaylight 网页控制台:

```
http://127.0.0.1:8181/index.html
```

打开之后默认的用户名密码为 admin/admin, 输入之后即可进入 opendaylight 控制台, 如图 4.4 所示。

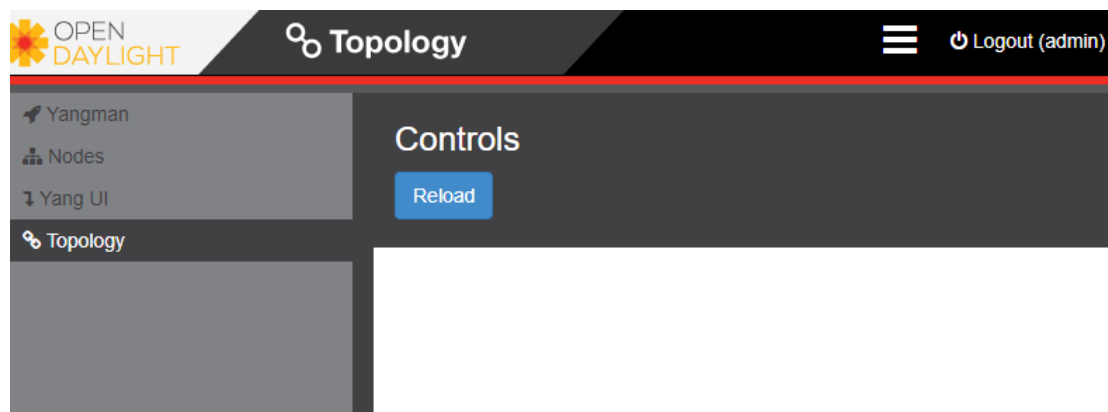


图 4.4 opendaylight 控制台页面

出现此页面即说明 opendaylight 安装成功。

#### 4.3.1.2 安装注意及可能出现的问题

1. 当运行相关命令行代码的时候, 在 windows 下使用 tab 寻找下一级路径, 此时路径间隔符应该使用 “\” 而不是 “/”。

2. 运行 karaf 失败，发现提示 need root，在启动 opendaylight 的命令前加上 sudo 即可。或者使用命令 su 输入 root 用户密码后进入 root 账户，再执行运行命令，此时无需再加 sudo。

3. 在安装 bundle 的时候可能会出现安装出错的情况，如图 4.5 所示。

```
opendaylight-user@root>feature:install odl-openflowplugin-all
Error executing command: Can't install feature odl-openflowplugin-all/0.0.0:
null
opendaylight-user@root>feature:list | grep openflowplugin
```

图 4.5 安装 bundle 出现找不到插件。

此时有 3 种情况，一个是已经安装该插件，另一个是插件库中没有该插件，再一个是插件名输入错误或者已经改名。

使用以下命令查看已经安装的插件：

```
Feature:list -i
```

查看是否有和自己所需安装的 bundle 类似的插件，有的话则说明已经安装该插件，则直接跳过即可。

如果没有则使用以下命令安装的时候按 2 下 tab 键，它会显示所能安装的所有插件的列表：

```
Feature:install
```

在所有未安装插件列表中查找是否有和自己所需安装的插件类似的插件，然后进行安装。

如果以上 2 种方法都不能解决问题，则可能是插件并没有在当前插件库中注册，需要修改 feature 目录下的配置文件，进行插件的添加。此方法会在后文中对 opendaylight 插件进行开发并注册时提及。

4. 在安装 opendaylight 的时候提示找不到 java，如图 4.6 所示。

```
cang@cang-virtual-machine:~/下载/distribution-karaf-0.3.3-Lithium-SR3/bin$ java
-v
Unrecognized option: -v
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
cang@cang-virtual-machine:~/下载/distribution-karaf-0.3.3-Lithium-SR3/bin$ java
version
```

图 4.6 安装 odl 提示找不到 java

则进行 java 的安装，安装完成之后使用以下命令行对 java 是否安装成

功进行测试：

```
Java -version
Javac -version
```

5. 在使用 maven 编译 opendaylight 的时候编译失败，提示 maven 要求版本 3.3.x 以上，我使用的是 3.0.5，重装更高阶版本后解决问题。

6. 在 windows 下尝试编译出错，原因是无法识别 windows LR 的换行符。此时说明在进行文件拷贝的时候出现了问题，需要重新下载源码进行编译。重新下载编译之后解决问题。

### 4.3.2 Mininet 的安装

#### 4.3.2.1 安装步骤

安装基本环境：

操作系统：ubuntu-kylin、

软件环境：mininet, git

安装过程：

Mininet 作为一个虚拟网络测试平台，ubuntu 对其有很好的仓库安装资源支持，只需要在 ubuntu 里输入一下命令行即可完成安装。

```
Sudo apt update
Sudo apt upgrade
Sudo apt install mininet
```

前 2 个命令行为更新安装源和已经安装的依赖，最后一个命令行为安装 mininet。

安装完成后进行测试，在命令提示符里输入一下命令即可查看 mininet 的版本，如果有响应则说明 mininet 安装成功：

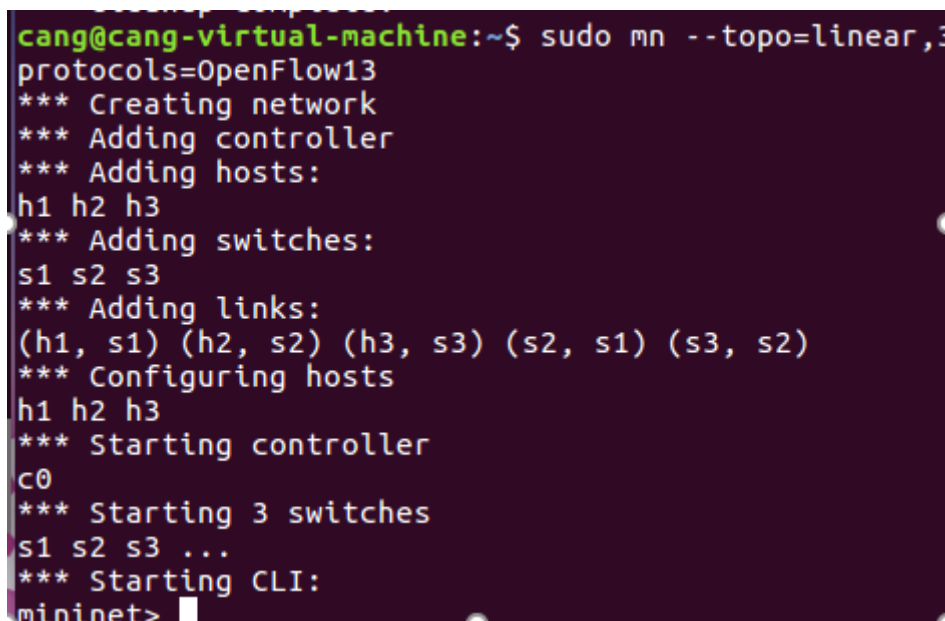
```
Sudo mn -v
```

可以使用下列方法使用 MIninet 创建简单的网络拓扑：

```
Sudo mn --mac --topo=linear,3 --switch=ovsk,protocols=OpenFlow13
```

```
--controller=remote,ip=192.168.1.101,port=6633
```

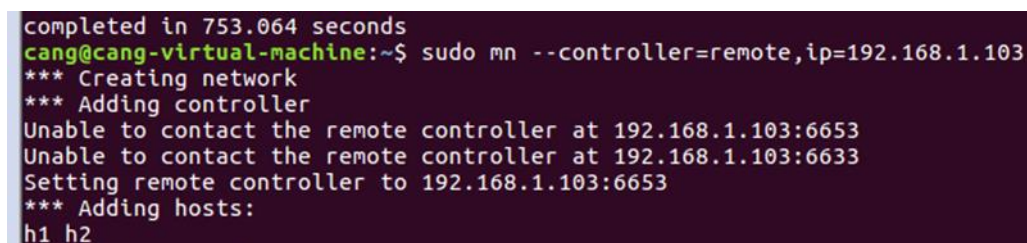
创建完成后的界面显示如图 4.7。



```
cang@cang-virtual-machine:~$ sudo mn --topo=linear,3
protocols=OpenFlow13
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet>
```

图 4.7 mininet 成功启动界面

此处截图为实现成功连接 OpenDaylight 控制器之后的截图，若此时 OpenDaylight 控制器没有打开，或者连接失败，则会如图 4.8 显示：



```
completed in 753.064 seconds
cang@cang-virtual-machine:~$ sudo mn --controller=remote,ip=192.168.1.103
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.1.103:6653
Unable to contact the remote controller at 192.168.1.103:6653
Setting remote controller to 192.168.1.103:6653
*** Adding hosts:
h1 h2
mininet>
```

图 4.8 mininet 启动未连接上控制器界面

命令行参数解析：

--mac：每次 mininet 创建新的拓扑的时候 mac 地址都会随机变化，mac 作用就是保证 mac 地址简单易懂，一般都会选择从很小的地址开始增加。

--topo：该参数后面可以选择 mininet 要创建的拓扑，mininet 提供了 5 中类型的拓扑，分别为：linear、single、tree、reversed 和 minimal，其中 minimal 为最小的网络拓扑，也是默认创建的网络拓扑。

--switch：该参数后面跟交换机的类型，此处为 ovs 交换机。

Protocol: 该参数跟在 switch 参数后面, 表示 switch 中规定类型的交换机应该支持的协议是什么。

--controller: Controller 即控制器, 此处为选择 mininet 链接的控制器是远程的还是本地的, 并且在此参数后面跟 ip 和 port。此处的意思为远程控制器的 ip 地址和连接端口。

#### 4.3.2.2 安装注意及遇到的问题

1. 当 mininet 搭建完毕, 开始使用 mininet 对某个网卡进行监听的时候, 需要用到以下命令:

```
Tcpdump -i eth1 src 127.0.0.1 and ! port 22
```

该命令是从各个论坛上摘的, 但是运行起来会出错, 原因就是并不是每个 Linux 系统的第一个网卡都叫 eth1, 在使用 ifconfig 命令进行网卡及网络信息的查询, 你会发现自己所用的那个网卡的名称, 本机上的连接名称叫 ens33, 因此只要把 eth1 改为本机连接名即可。

### 4.4 试验环境各部分工具间的连接以及测试

#### 4.4.1 试验环境各部分之间的连接及测试

本节将讲述一下 OpenDaylight 与 Mininet 的连接步骤[8]。

1. 开启 Opendaylight 控制器。

在 windows 系统上运行 OpenDaylight 控制器, 找到 OpenDaylight 所在文件夹根目录, 运行 opendaylight:

```
karaf\target\assembly\bin\karaf.bat
```

Opendaylight 启动完成后, 如图 4.9 所示。

```
H:\odl-carbon\topology>karaf\target\assembly\bin\karaf.bat
Apache Karaf starting up. Press Enter to open the shell now...
100% [=====]

Karaf started in 61s. Bundle stats: 332 active, 332 total

      OpenDaylight

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

图 4.9 odl 成功启动界面

然后打开浏览器，输入 Opendaylight 控制器的网页控制台地址，使用 admin/admin 默认用户名密码进入控制台查看。



图 4.10 odl 网页控制台界面

2. 打开 Vmware 虚拟机，为虚拟机设置桥接连接网络。

打开 Vmware，右键点击虚拟机→ 设置，找到网络适配器，选择桥接模式，启动时自动连接。如图 4.11 所示：

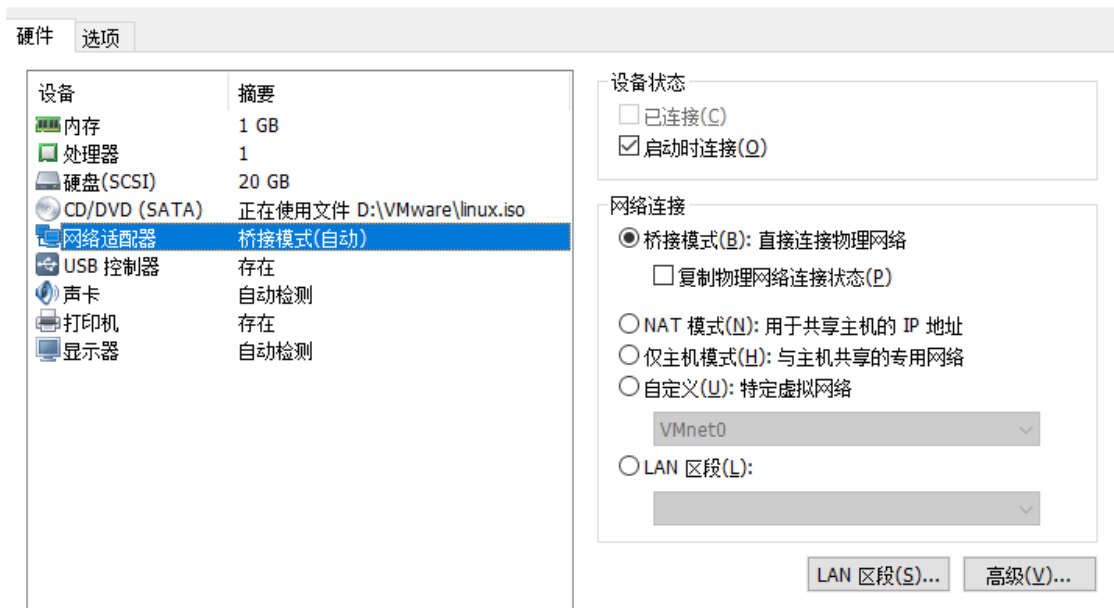


图 4.11 vmware 网络适配器界面

打开 Ubuntu-kylin 虚拟机，进行虚拟机设置。右键编辑→ 网络虚拟编辑器，查看是否已有一个默认配置，如图 4.12。





图 4.12 vmware 网络虚拟编辑器界面

更改默认的配置信息，选择桥接模式，并且在桥接到里选择物理网卡，如图 4.13 所示：

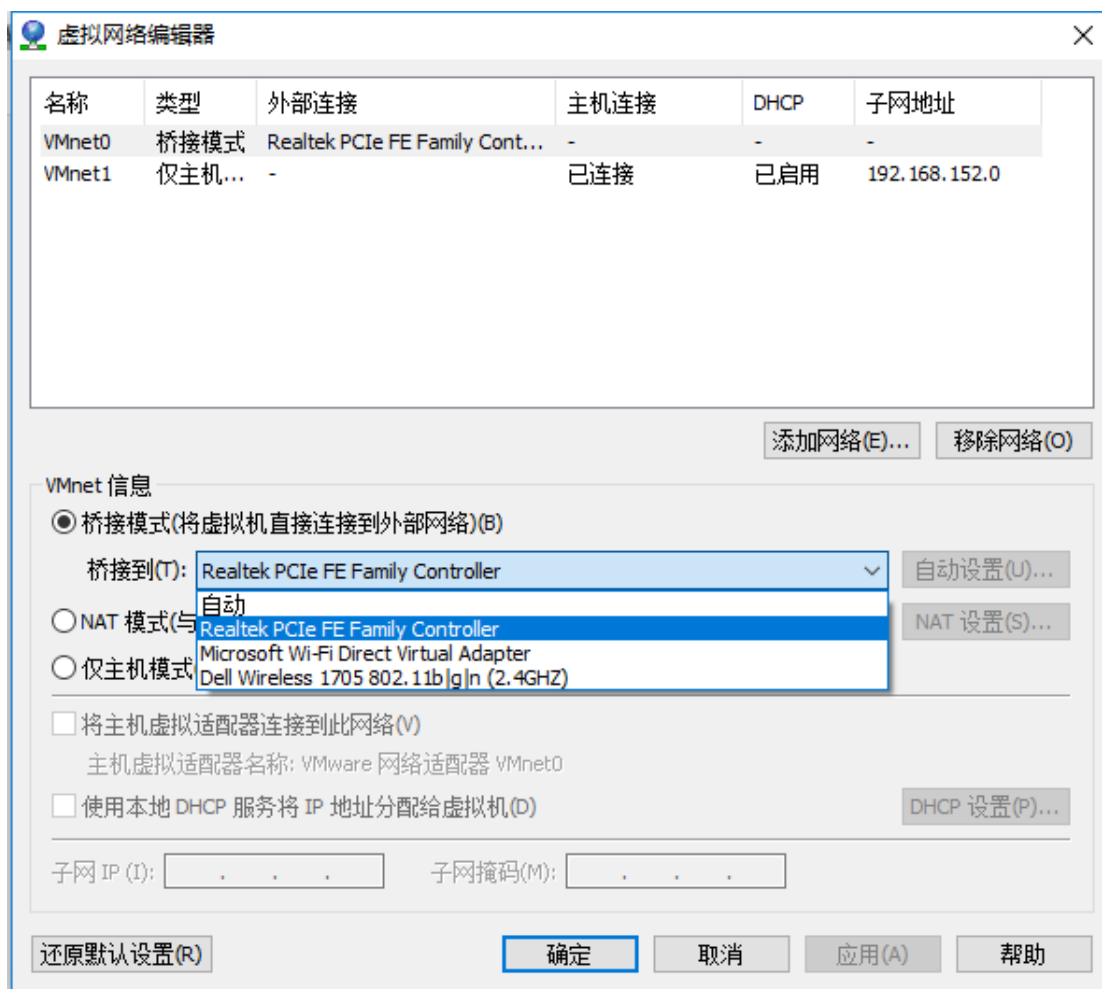


图 4.13 vmware 网络虚拟编辑器配置

此时，重启 Vmware 虚拟机，即可实现桥接联网。但有可能会发生 DHCP 没有分发的的问题，此时需要关闭 DHCP，手动编写网络配置，进行桥接联网。

3. 打开 ubuntu-kylin 虚拟机，使用如下命令行建立网络拓扑初始化 mininet 并连接 Opendaylight 控制器。

```
sudo mn --mac --controller=remote,ip=192.168.1.101
--topo=linear,3 --switch=ovsk,protocols=OpenFlow13
```

以上命令行首先要进行 Opendaylight 控制器所在主机的 ip 查询，查询

到 ip 地址后再使用该命令进行连接。

连接成功时如图 4.14 所示：

```

cang@cang-virtual-machine:~$ sudo mn --mac --controller=remote,ip=192.168.1.105
--topo=linear,3 --switch=ovsk,protocols=OpenFlow13
*** Creating network
*** Adding controller
Connecting to remote controller at 192.168.1.105:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet>
    
```

图 4.14 mininet 启动成功连接控制器

连接失败时如图 4.15 所示：

```

cang@cang-virtual-machine:~$ sudo mn --mac --controller=remote,ip=192.168.1.101
--topo=linear,3 --switch=ovsk,protocols=OpenFlow13
[sudo] cang 的密码:
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.1.101:6653
Unable to contact the remote controller at 192.168.1.101:6633
Setting remote controller to 192.168.1.101:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> quit
    
```

图 4.15 mininet 启动连接控制器失败

这里把 ip 换成不存在的控制器。Mininet 会默认尝试连接该 ip 下的 6653 端口以及 6633 端口，当 2 者连接都失败的时候也会自动创建网络拓扑，但是这个时候只存在拓扑，拓扑里的节点间没有办法相互连通。使用 pingall 对每个节点间的连接状态进行测试：

```

Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
mininet>

```

图 4.16 mininet 使用 pingall 测试

发现每个节点之间都无法连通，因为缺少控制器，节点没有路由表，无法进行网络拓扑的计算及数据包的转发。但是当连接成功之后，使用 pingall 对节点间是否连通进行测试：

```

Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
mininet>

```

图 4.17 mininet 使用 pingall 测试

发现所有节点之间都能连通，掉包率为 0%。这是由于连接了 Opendaylight 控制器，控制器通过获取每个节点的状态信息计算拓扑，每个节点已经自动被下发流表，所以能够 ping 通。

4. 使用 MIninet 连接 Opendaylight 控制器，连接成功后回到 Opendaylight 控制台，点击 Topology→reload，即可出现刚刚搭建的 mininet 网络拓扑，如图 4.18 所示：

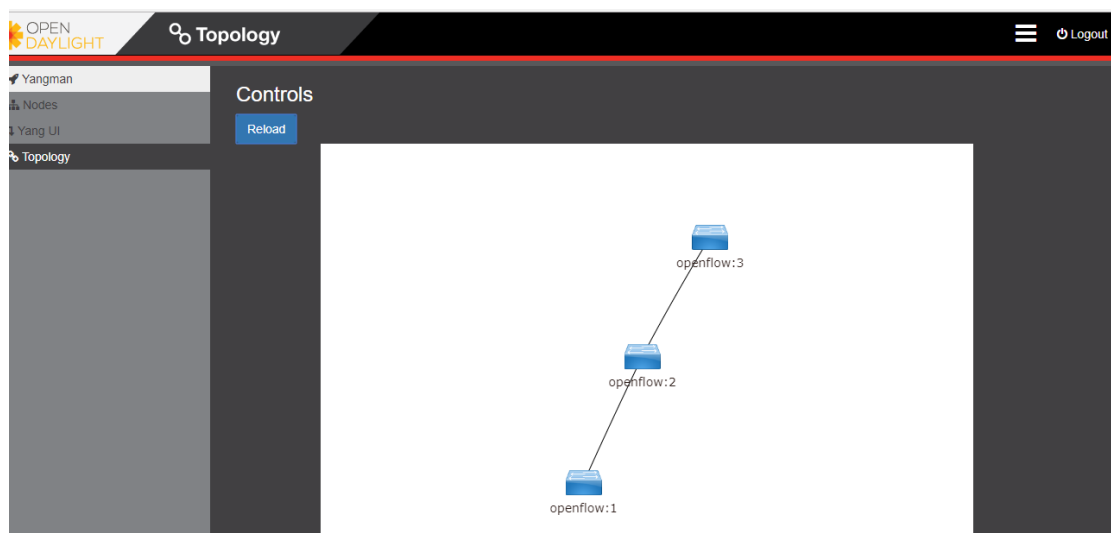


图 4.18 odl 控制台显示 mininet 网络拓扑

此时即可说明 Mininet 和 OPendaylight 控制器连接成功，否则 reload 后仍然是空白则说明连接失败。

然后使用 Mininet 进行 pingall 测试，全部能够 ping 通，此时在 reload，发现拓扑发生变化，如图 4.19 所示：

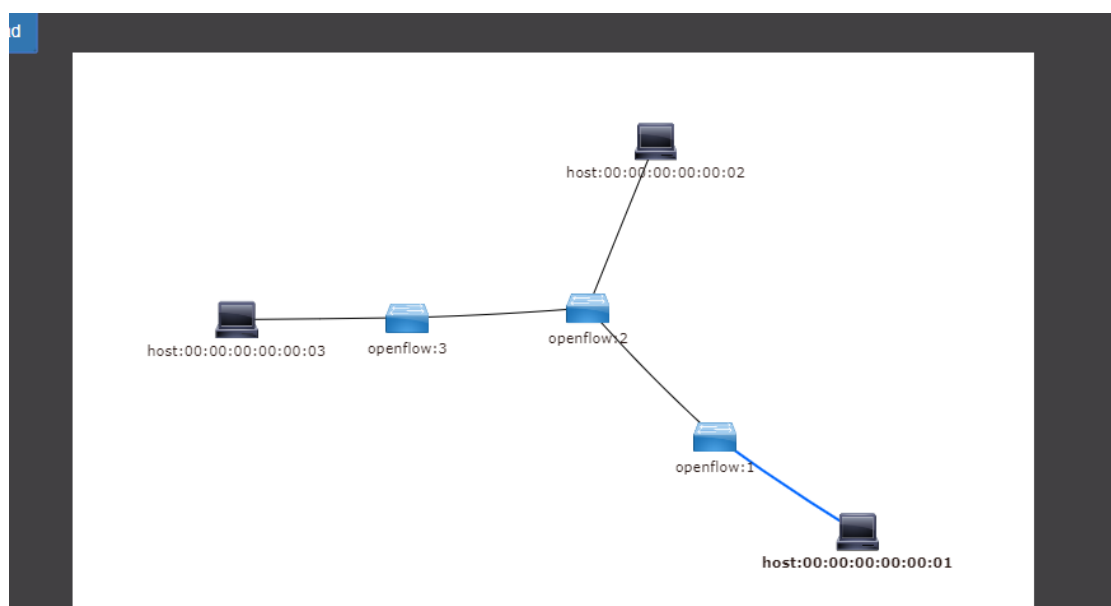


图 4.19 mininet 使用 pingall 后的 odl 控制台

这是因为每个 OpenFlow 节点有自动学习功能，在进行 pingall 测试之后，每个 OpenFlow 节点知道了与自己相邻的主机和交换机的信息，因此从每个 OpenFlow 节点中存储的节点信息整合出主机的位置及其信

息。

#### 4.4.2 连接时可能会碰到的问题

1. Mininet 默认连接的接口为 6653 和 6633，在 mininet 进行 OpenDaylight 控制器连接的时候，如果不对 port 参数进行具体的设置，则 mininet 默认会对 6653 和 6633 两个接口分别进行测试。而 OpenDaylight 控制器则是默认监听 6633 接口的，该监听接口的配置文件可以在 OpenDaylight 安装目录下的 etc 目录下的 custom.properties 文件中编写，你可以在此设置 OpenDaylight 需要监听的端口，注意在修改完成后需要重启 OpenDaylight。如下图 4.20 所示：

```
# Open Flow related system parameters
# TCP port on which the controller is listening (default 6633)
of.listenPort=6633
# IP address of the controller (default: wild card)
# of.address = 127.0.0.1
# The time (in milliseconds) the controller will wait for a response after se
```

图 4.20 odl 的 custom.properties 文件中对端口的配置

2. 可能会存在 OpenDaylight 控制器和 mininet 怎么都连接不上的问题，最后发现是 OpenDaylight 控制器所在主机打开了防火墙，需要使用命令行 `sudo iptables -F` 关闭防火墙，然后再进行重新连接，发现问题解决。

## 第 5 章 OpenDaylight 插件的设计及开发

当搭建完成测试的基本环境，实现 opendaylight 控制器+Mininet 虚拟网络测试平台+OpenVswitch 交换机的交互之后，我们开始着手实现 Opendaylight 控制器的插件开发，开发文档使用官网文档[9]。

由于 Opendaylight 是由应用程序提供功能，而应用程序则是以 bundle 插件的形式加载到 Opendaylight 控制器中的，我们的目标是将 Fibbing 技术移植到我们搭建好的测试平台中，因此本章将为介绍内容最丰富最详细的一章。

在开始进行应用程序开发解析之前，我想先描述一下实现我们的目标的大体流程以及具体思路。我们的环境是基于 Opendaylight 控制器+Mininet 的，OpenDaylight 控制器的应用程序编写则是使用的 Java 语言，因此我们需要将 Jennifer 教授在 Github 上提供的 python 代码链接到我们的项目中，需要使用 Java 的 jni 技术实现。于是，我确定了对 OpenDaylight 的研究步骤。首先，我会从网上及 OpenDaylight 管网进行相关资料的查询，查询之后我发现需要进行 bundle 的开发，在使用小例子测试完 bundle 应用程序已经成功嵌入 OpenDaylight 控制器之后，我会对 OpenDaylight 提供的 REST API 接口进行分析，找到我们所需要的接口，即获取每个 OVS 交换机节点信息，获取每个 ovs 交换机节点的流表，为每个 ovs 交换机节点下发流表。以上功能需要使用 Java 程序来实现，以使得当插件开发完成之后，可以将此 java 程序段代码接入到我们所编写的 OpenDaylight 控制器插件中。

在本章，我们会对 Jennifer 教授提供的关于 Fibbing 的核心代码进行解读，分析每段代码的大体作用，由于本人对 python 语言不是很熟悉，因此只能介绍个大概。然后我会介绍 OpenDaylight 控制器的插件的开发流程，实现在 OpenDaylight 控制器中对所开发的应用程序插件进行注册并运行。接着，我会介绍 Java 实现 OpenDaylight 控制器提供的 api 的使用，及每个接口的数据格式。然后，我会介绍 Java 技术的使用，并撰写一个小 demo 来演示。

由于时间问题，以及其他因素的制约，本人只能完成到 JNI 的学习。由于本人不擅长 Python 语言的编程学习，所以对 Jennifer 教授编写的 Fibbing 协议核心代码的解读上做的不是很好，因此没有办法实现 java 对 python 核心代

码的接入，所以这里点到即止，说了一下大体的思路，如果有人能够有幸阅读到这篇文章，有一定的能力和精力，我希望他可以继续尝试一下。

## 5.1 Fibbing 相关代码描述

### 5.1.1 代码整体架构介绍

关于 Fibbing 技术的有详细介绍的在其官方网站[5]可以找到，我们可以在这里阅读 Fibbing 技术实现的功能，以及找到 Fibbing 在 Quagga 软件上进行测试的 Demo 和视频。我们可以在 Github 上找到《Central Control Over Distributed Routing》相对应的项目的代码，网址为：<https://github.com/fibbing/>，如图 5.1 所示。

#### FibbingNode

The Fibbing controller code

● Python ★ 21 🍴 7 📄 GPL-2.0 Updated on 22 Mar

#### fibbing.github.io

The website gathering information on Fibbing

● HTML 🍴 2 Updated on 4 Oct 2017

#### Quagga

A custom version of Quagga allowing to inject arbitrary Type-5 LSAs

● C ★ 1 🍴 4 Updated on 2 Dec 2016

#### virtual-machine

VirtualBox VM to experiment with Fibbing (with or without mininet)

● Shell ★ 1 Updated on 4 Jun 2016

#### labs

Simple labs to show how fibbing works

● Python 🍴 2 Updated on 14 Jan 2016

图 5.1 Fibbing 项目代码架构

Fibbing 项目由 5 部分构成，Fibbing Node, fibbing.github.io, Quagga, virtual-machine, labs. 各个项目的作用分别为：

FibbingNode: 是 Fibbing 控制器的代码，负责实现 Fibbing 的主要控制逻辑。

Fibbing.github.io: 搭建的 Fibbing 网站的代码，该网站用于 Fibbing 上的信息。

Quagga: 是一个自定义版本的 Quagga 代码，该自定义版本支持注入任意一个第五种链路状态类型的协议。

Virtual-machine: VirtualBox 虚拟机，用于体验 Fibbing, Mininet 可以有也可以没有。

Labs: 是一些简单的例子，用于展示 Fibbing 是怎么工作的。

### 5.1.2 代码具体部分介绍

本节只对 FibbingNode 项目进行介绍，主要是介绍一下 FibbingNode 项目的分布，各个包的功能及作用，不会对实现进行过多介绍。

将 FibbingNode 从 Github 上下载下来之后，导入 PyCharm，项目的整体分布如图 5.2 所示：



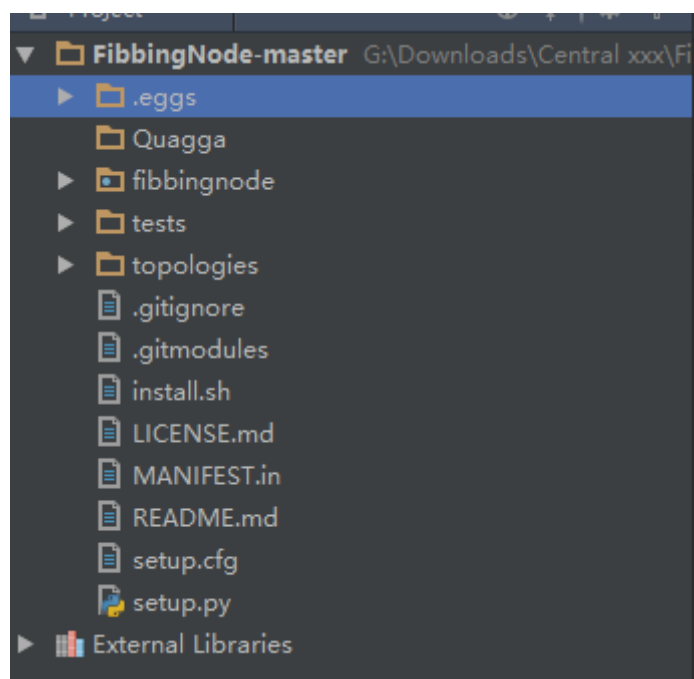


图 5.2 Fibbing-master 项目代码架构

其中每个包的作用如下：

1. .eggs 文件夹下存放一些文件，这些文件由安装工具下载，用于搭建和测试项目，也用于运行插件。这个文件夹下的内容作为构建项目等操作的缓存文件夹，可以多次使用，防止每次都要进行重复下载。但是删除该文件夹是不会对项目搭建造成影响的，只是可能会在搭建过程中重新生成并缓存相关文件。
2. Fibbingnode 文件夹为整个项目的核心包，所有的核心代码都在这里编写实现。Fibbing 中的代码组成如图所示：

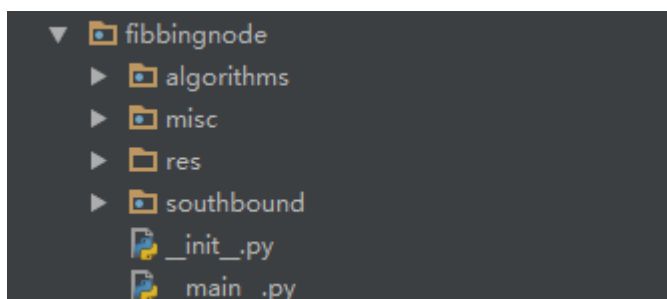


图 5.3 Fibbing 项目代码架构

Algorithms 包下为实现 Fibbing 的核心算法。

southbound 为南向接口的编写算法。

Res 内为支持的 OFPS 协议和 zebra 的配置文件。

Misc 下为对 mininet 以及 Quagga 的支持。

3. Tests 文件夹为对每一个功能模块进行测试的代码。代码结构如下图所示：

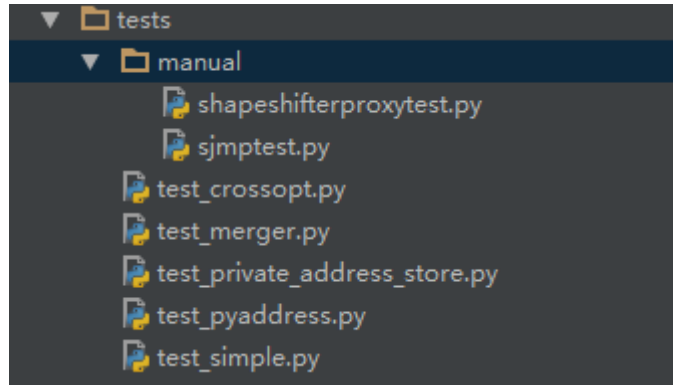


图 5.3 Fibbing-master 中 Tests 目录代码架构

4. Topologies 内为对拓扑和路径权重等的配置。

## 5.2 ODL 插件开发

该部分介绍如何对 Opendaylight 控制器进行插件开发[6][10][11]。开发 Opendaylight 插件需要学习以 Java 为基础，学习 Opendaylight 控制器接入的接口及相应机制。本文会先介绍与 Opendaylight 控制器相关的部分所需要用到的技术及知识点，再介绍 Opendaylight 控制器插件开发的流程及步骤。

### 5.2.1 相关技术概念

1. RPC。 Opendaylight 控制器使用的是 RPC 通信方式，RPC 全称 Remote Procedure Call，远程过程调用，是一种进程间进行通信的一种方式。这种方式有可以让被调用这透明化的优点。即被调用者只需要提供一个接口，规定好接口的功能及传过来的参数的格式，调用者不需要了解被调用者的内部细节，只需要传入相应的参数，即可拿到正确的结果。RPC 技术可以实现跨平台调用，即只有接口与接口之间的通信，不涉及到不管是调用者还是被调用者的具体实

现，可以使被调用者提供的功能更广泛的被调用者使用，提高了广泛性和通用性。

2. YANG 语言。YANG 语言是一种数据建模语言，其主要的目的和作用就是对 NETCONF 协议所操作的数据进行基础的建模。YANG 语言类似于 JSON 格式，但又不同于 JSON。YANG 语言也通过层级嵌套的结构以及其中夹杂着各个属性以及其属性值的形式进行定义，每一个 YANG 语言模块为一棵树形结构，对某一个节点或者部分进行数据结构的定义。

在 OpenDaylight 中，YANG 语言主要用于对远程过程调用 RPC、资源和通知进行建模。对于远程过程调用 RPC 来说，YANG 语言对 RPC 提供接口的调用者的编码架构进行建模和定义；对数据来说，YANG 语言对需要操作的资源进行建模，比如对网络中的节点设备的配置文件等进行建模，使得配置文件可以统一格式，方便更改与管理；对通知进行建模，即对网络上的节点设备发出的请求或者事件或者通知的格式进行管理，统一通知格式，方便接受与显示。YANG 模型也是 NETCONF 客户端以及服务器端所有数据进行交互的数据格式的完整表述。

3. RESTCONF。OpenDaylight 控制器的北向接口的功能就是使用 RESTCONF 协议的方式提供给外界使用。RESTCONF 是一种管理协议，它基于 REST 模式进行设计，用于网络的配置和管理。在 OpenDaylight 中，RESTCONF 为 OpenDaylight 提供一个以 HTTP 协议传输数据为基础的配置数据，状态数据，和通知时间的标准的架构机制。一个 RESTCONF 的操作是由一个 HTTP 连接和被请求资源的 URI 路径组成。

4. OSGI 框架。OSGI 为一种通用的框架协议规范，它提供了一个开发标准，每个插件只要都遵从 OSGi 框架标准，开发人员可以动态化，模块化的进行应用程序的开发。每个应用程序都作为一个模块，彼此互不影响，相互隔离，各个模块化的相关应用便可以像搭积木一样完成编写与搭建。

在 OpenDaylight 中，bundle 即为每个互不相干的模块。每一个 Bundle 就是一个应用程序，其真实形式就是 Java 开发中的 jar 包。与 jar 包一样，一个 Bundle 也是通过 META-INF 文件夹下边的 MANIFEST.MF 进行定义。OSGi 框架管理着 Bundle 的状态，主要有如下几个状态：已安装 installed，已解析 resolved，

启动中 starting, 激活 active, 停止中 stopping, 已卸载 uninstalled。其状态转换表如下图 5.4 所示:

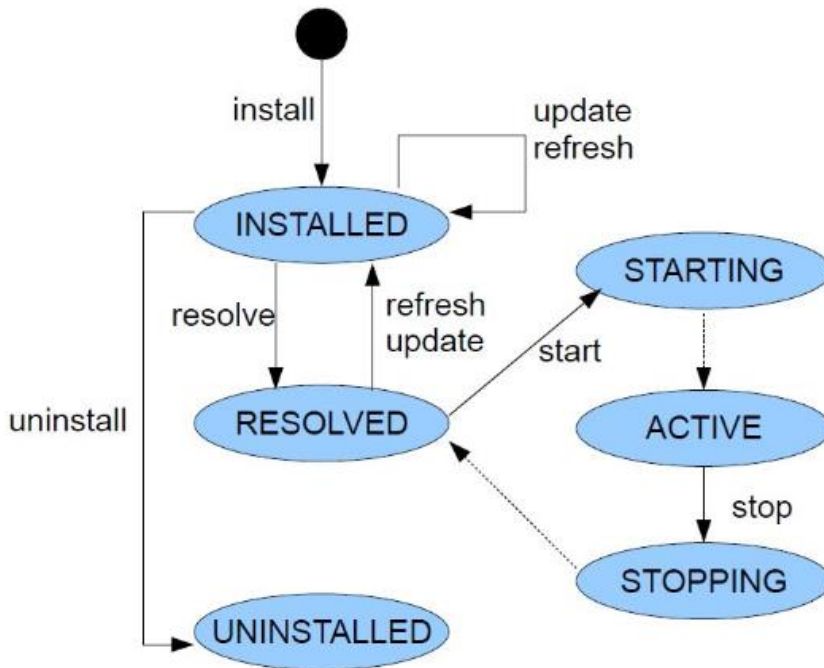


图 5.4 opendaylight 插件状态转换表

在 OpenDaylight 中, 还存在 OSGi Service, 即 OSGi 服务, 其表现形式就是一个 Java 对象, 它通过在 OSGi 框架中进行注册成为一个 OSGi 服务。在 OSGi 框架中注册了之后, 其他的 bundle 便可以对该服务进行调用, 使用该对象提供的服务, 从而实现了在 OSGi 框架中每个互不干涉的模块进行合作与通信。

5. MD-SAL。Model-Driven Service Abstraction Layer, 模型驱动的服务抽象层。OpenDaylight 使用了 MD-SAL 用来实现南向接口相关开发和北向接口相关开发的解耦。即模型驱动的服务抽象层为一个南北向接口的适配层, 它向南支持南向接口的开发, 使得南向接口开发可以接入 MD-SAL, 面向北向又通过北向的接口来提供南向接口所提供的这些功能, 北向接口不需要知道南向接口的具体实现过程与细节, 只需要调用 MD-SAL 的北向接口, 即可得到所需要的功能。因此, MD-SAL 实现了南北向开发的解耦, 使得南北向接口独立开发, 独立发展, 互不干扰, 互不影响, 加快了南北向接口的开发进度。

### 5.2.2 插件开发流程

本节将介绍 OpenDaylight 控制器插件的开发流程，本章介绍的流程是基于 OpenDaylight 使用代码进行编译安装之后的后续流程，具体的 OpenDaylight 使用 mvn 编译安装流程请详见第四章相关章节。

本章节将实现编写一个插件，实现获取网络拓扑的功能，并且能够在 OpenDaylight 控制器中注册使用。编程环境为 Eclipse 作为开发工具，Java 版本 Java1.8。

- 1、将 OpenDaylight 控制器代码导入 eclipse[15]。

打开 eclipse，在项目列表空白处右键导入，然后在列表里选择 Maven project，如下图 5.5 所示，

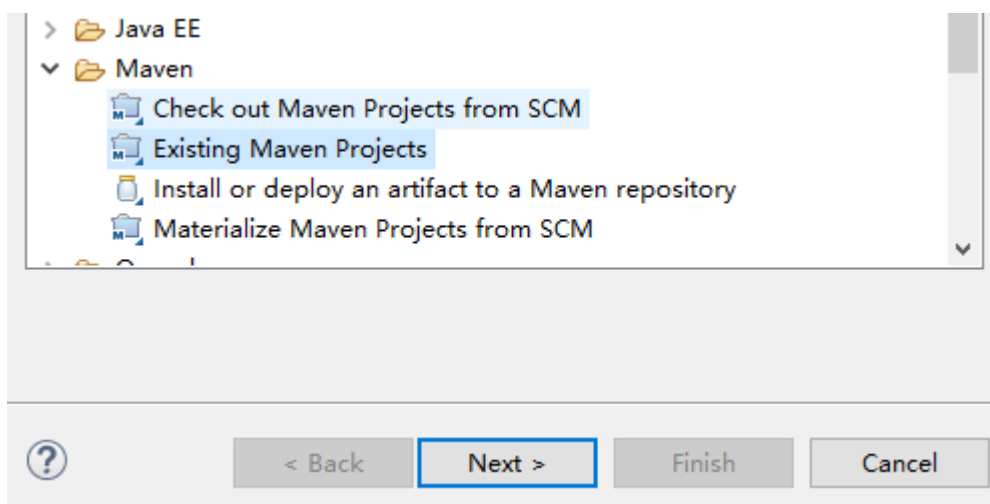


图 5.5 eclipse 导入 maven 项目

然后便可以在 eclipse 中查看并编写相关代码。此时 eclipse 会提示项目报错，这是项目引用的问题，不影响项目的正常运行，选择跳过即可。

- 2、进入 api/src/main/yang 目录下的 topology.yang 文件中，进行 YANG 模型语言的编写。部分 YANG 文件代码如图 5.6 所示：

```

1. module topology {
2.
3.   yang-version 1;
4.
5.   namespace "urn:opendaylight:params:xml:ns:yang:topology";
6.
7.   prefix "topology";
8.
9.   import ietf-inet-types { prefix inet; revision-date 2013-07-15; }
10.
11.  description "get links and ports information";
12.
13.  revision "2018-03-07" {
14.
15.    description "Initial revision of topology model";
16.
17.  }
18.

```

图 5.6 部分 YANG 文件代码

现在解读一下上图中的几个属性定义：

Module: module 为一个模块，每个模块拥有不同的数据类型。

yang-version: 表明 YANG 模型的版本。

Namespace: 命名空间，使用此命名空间以避免同名冲突。

Prefix: 简称。

Import: 导入相关其他的 YANG 模块，从而可以在本模块中进行其他模块的相关引用，类似于 Java 中的对象调用，提高了灵活性。

Description: 对每个 module 模块的描述。

Revision: 叙述编写的该模块的版本信息。

更多的关于 YANG 语法的相关内容可以去 RFC 官方文档上进行查阅和学习：  
<https://tools.ietf.org/html/rfc6020>。

由于此处有 import 引用了其他的模块，因此需要在 api 目录下的 pom.xml 中添加相关模块的依赖。加入之后在 api 模块进行重新编译的时候才会正确运行，否则会提示找不到相关依赖而出错。添加部分如图 5.7 所示：

```
<modelVersion>4.0.0</modelVersion>
<dependencies>
  <dependency>
    <groupId>org.opendaylight.md.sal.model</groupId>
    <artifactId>ietf-inet-types-2013-07-15</artifactId>
    <version>1.2.2-Carbon</version>
  </dependency>
</dependencies>
```

图 5.7 pom.xml 添加代码部分

编写完成 YANG 文件之后，后续对项目的编译 OpenDaylight 会自动根据该 YANG 文件产生对应的 Java 文件模块，从而进行自己所需功能的开发。

3. 编辑好 YANG 文件之后，要重新编译以生成相关文件，但此时无需对整个项目进行编译，只需编译 api 目录，因此只需到 api 目录下，运行以下命令即可。

```
mvn      clean      install      -DskipTests      -Dmaven.javadoc.skip=true
-Dcheckstyle.skip=true
```

稍等一会，等待编译完成之后，你会发现 api 目录下多了很多 java 工程相关文件，如下图所示。这是因为 OpenDaylight 控制器中的相关插件 YANG-Tools 会根据 YANG 文件自动生成相关的 Java 文件，关于具体的生成原理在此不多做赘述。

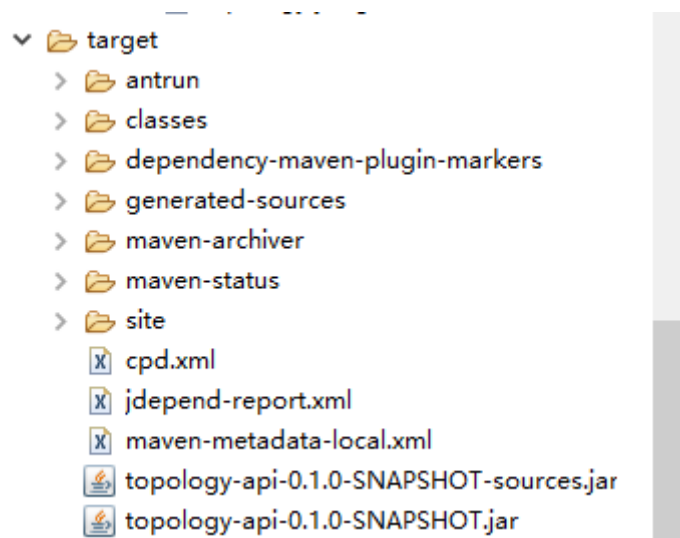


图 5.8 odl-api 目录下代码架构

4、在编译完成 api 目录之后，便可以进行 rpc 的编写了。RPC 即远程过程

调用，即此部分实现的会是一个 RPC 接口，可以通过 http 的 get 和 post 的形式进行访问和提供服务。

由于本部分的模块名称叫做 topology，则相关代码也会在该目录下进行实现。在该目录下打开文件 impl/src/main/java/org/opendaylight/topology/impl/TopologyProvider.java 进行相关代码的编写与实现，由于代码可能有点多，这里只贴出部分代码。

这 2 个方法是在插件进行注册和注销的时候使用的，当触发的时候会执行图 5.9 中显示的 2 个方法。

```
public void init() {
    LOG.info("TopologyProvider Session Initiated");
}

public void close() {
    LOG.info("TopologyProvider Closed");
}
```

图 5.9 TopologyProvider.java 文件中的部分代码

这个方法为核心代码的实现，主要功能为列出所有节点的信息。

```
public Future<RpcResult<ListLinksInfoOutput>> listLinksInfo() {

    final SettableFuture<RpcResult<ListLinksInfoOutput>> futureResult = SettableFuture.create();
    ListLinksInfoOutputBuilder outputBuilder = new ListLinksInfoOutputBuilder();

    final InstanceIdentifier.InstanceIdentifierBuilder<Topology> topologyId = InstanceIdentifier
        .child(Topology.class, new TopologyKey(new TopologyId(new Uri(FlowId)))));
    InstanceIdentifier<Topology> topologyIID = topologyId.build();
    Topology topology = read(LogicalDatastoreType.OPERATIONAL, topologyIID);

    if (topology == null || topology.getLink() == null || topology.getLink().size() < 1) {
        futureResult.set(RpcResultBuilder.success(outputBuilder.build()).build());
        return futureResult;
    }
    List<LinksInfo> linkInfos = new ArrayList<>();
    topology.getLink().forEach(temp -> {
        LinksInfoBuilder lib = new LinksInfoBuilder();

        lib.setLinkId(temp.getLinkId())
            .setSrcDevice(temp.getSource().getSourceNode())
            .setDstPort(getPort(temp.getSource().getSourceTp().getValue()))
            .setDstDevice(temp.getDestination().getDestNode())
            .setDstPort(getPort(temp.getDestination().getDestTp().getValue()));
        linkInfos.add(lib.build());
    });

    outputBuilder.setLinksInfo(linkInfos);
    futureResult.set(RpcResultBuilder.success(outputBuilder.build()).build());
    return futureResult;
}
```

图 5.10 TopologyProvider.java 文件中的主要方法代码

其实现的大体流程为：



先去 OpenDaylight 的 net-work 的 operational 数据库中进行获取 id 为 openflow:1 的节点的信息，然后使用自己编写的过滤器进行节点信息的过滤，找出我们所需的相关信息，然后在将我们得到的所需的信息封装成为 Future 对象格式进行返回，这也是最终插件会返回的数据。

由于我们在代码中使用了 model-flow-service 、 ietf-topology 、 model-inventory 相关模块，因此我们需要在 pom.xml 中进行该依赖的添加。添加代码如图 5.11 所示：

```
<dependency>
  <groupId>org.opendaylight.mdsal.model</groupId>
  <artifactId>ietf-topology</artifactId>
  <version>2013.10.21.10.2-Carbon</version>
</dependency>
<dependency>
  <groupId>org.opendaylight.controller.model</groupId>
  <artifactId>model-inventory</artifactId>
  <version>1.5.2-Carbon</version>
</dependency>
<dependency>
  <groupId>org.opendaylight.openflowplugin.model</groupId>
  <artifactId>model-flow-service</artifactId>
  <version>0.4.2-Carbon</version>
</dependency>
```

图 5.11 pom.xml 添加依赖代码

由于在此使用了 openflowplugin 依赖，但是在 maven 仓库中没有该源的支持，因此需要自行配置源。需要在 feature 目录下的 src/main/features/features.xml 文件中进行相关仓库源引用的添加。如图 5.12 所示：

```
<repository>mvn:org.opendaylight.dluxapps/features-dluxapps/{VtKSLUW}}/xml/features</repository>
<repository>mvn:org.opendaylight.openflowplugin/features-openflowplugin/${openflowplugin.version}/xml/features</repository>
<repository>mvn:org.opendaylight.l2switch/features-l2switch/${l2switch.version}/xml/features</repository>
<feature name='odl-topology-api' version='${project.version}'
```

图 5.12 仓库源引用代码

同时也需要在该文件中名为 odl-topology-api 的 feature 标签中添加对 openflowplugin 插件的依赖，为了能让我们所写的 topology 插件能够有权对 openflowplugin 插件进行引用。添加代码如图 5.13 所示：

```

<feature name='odl-topology-api' version='${project.version}'
    description='OpenDaylight :: topology :: api'
    <feature version='${mdsal.model.version}'>odl-mdsal-models</feature>
    <feature version='${openflowplugin.version}'>odl-openflowplugin-flow-services</feature>
    <bundle>mvn:org.opendaylight.topology/topology-api/{{VERSION}}</bundle>
</feature>
    
```

图 5.13 feature 源标签中引用代码

添加依赖之后需要修改 feature 目录下的 pom.xml，添加相关定义：

```

<properties>
    <mdsal.model.version>0.10.2-Carbon</mdsal.model.version>
    <mdsal.version>1.5.2-Carbon</mdsal.version>
    <restconf.version>1.5.2-Carbon</restconf.version>
    <yangtools.version>1.1.2-Carbon</yangtools.version>
    <dluxapps.version>0.5.2-Carbon</dluxapps.version>
    <configfile.directory>etc/opendaylight/karaf</configfile.directory>
    <openflowplugin.version>0.4.2-Carbon</openflowplugin.version>
    <l2switch.version>0.5.2-Carbon</l2switch.version>
</properties>
    
```

图 5.14 feature pom.xml 中引用代码

到此为止，我们项目所需要使用的获取拓扑的插件源便已经完成了。接下来进行 RPC 插件的注册。

5. OpenDaylight 的插件注册有很多种形式，这里采用的是在 blueprint.xml 文件中进行自定义插件的添加。Blueprint 类似于 Spring 的依赖注入功能，只要在这里注册之后，其他插件便可以在这里进行调用。对这些服务进行调用时，如果服务没有注册或者不可用，则会阻塞。在 impl/src/main/resources/org/opendaylight/blueprint/impl-blueprint.xml 中对我们的插件进行注册添加，添加内容如下图所示：

```

<bean id="provider"
    class="org.opendaylight.topology.impl.TopologyProvider"
    init-method="init" destroy-method="close">
    <argument ref="dataBroker" />
</bean>
<odl:rpc-implementation ref="provider" />
</blueprint>
    
```

图 5.15 blue-print.xml 中引用代码

需要注意的是，这 2 个 id 需要是相同的，否则会报错。

6. 解决 RPC 需要依赖的其他插件。由于使用我们编写的 RPC 插件需要 OpenDaylight 控制台和 l2switch 相关组件的支持，因此我们需要在 feature 目录下添加这两个插件的引用仓库，即让 OpenDaylight 安装这 2 个插件，使得我们的插件可以正常运行。

在 features 目录下的/src/main/features/文件夹下的 feature.xml 文件中添加对 l2switch 的依赖，如图 5.16 所示：

```
<repository>mvn:org.opendaylight.l2switch/features-l2switch/${l2switch.version}/xml/features</repository>
<feature name='odl-topology-api' version='${project.version}'
```

图 5.16 feature.xml 中引用代码

L2switch 使用的版本定义需要去 features 目录下的 pom.xml 文件中添加，添加内容如图 5.17 所示：

```
<openflowplugin.version>0.4.2-Carbon</openflowplugin.version>
<l2switch.version>0.5.2-Carbon</l2switch.version>
</properties>
```

图 5.17 feature pom.xml 中引用代码

7. 依赖插件添加完成之后，此时可以对项目进行整体的编译了，执行以下命令进行编译：

```
mvn clean install -DskipTests -Dmaven.javadoc.skip=true
-Dcheckstyle.skip=true
```

编译没有出错，使用以下命令运行 OpenDaylight：

```
./karaf/target/assembly/bin/karaf
```

启动完成之后，我们可以查看当前安装的插件列表，能发下我们安装的 topology 插件已经在运行中了。查看命令与查看结果如图 5.18 所示：

```
opendaylight-user@root>feature:list |grep topology
odl-dluxapps-topology 0.5.2-Carbon x odl-dlux-0.5.2-Carbon Enable nodes in OpenDaylight dlux
odl-openflowplugin-app-topology 0.4.2-Carbon x openflowplugin-0.4.2-Carbon OpenDaylight :: Openflow Plugin :: Application - t
odl-topology-api 0.1.0-SNAPSHOT x odl-topology-0.1.0-SNAPSHOT OpenDaylight :: topology :: api
odl-topology 0.1.0-SNAPSHOT x odl-topology-0.1.0-SNAPSHOT OpenDaylight :: topology
odl-topology-rest 0.1.0-SNAPSHOT x odl-topology-0.1.0-SNAPSHOT OpenDaylight :: topology :: REST
odl-topology-ui 0.1.0-SNAPSHOT x odl-topology-0.1.0-SNAPSHOT OpenDaylight :: topology :: UI
odl-topology-cli 0.1.0-SNAPSHOT x odl-topology-0.1.0-SNAPSHOT OpenDaylight :: topology :: CLI
```

图 5.18 odl 命令行中显示已安装

黄色标注即为正在运行的 topology 插件，我们也可以看到 openflowplugin 和 dlux 插件对 topology 插件的支持，以及我们所定义的版本。

接下来，我们需要继续安装我们需要使用的插件，因为我们在 features 中添加了仓库，但是 OpenDaylight 不会自行安装，还是需要我们手动安装。因此，我们使用如下命令进行相关插件的安装：

```
feature:install          odl-dluxapps-nodes          odl-dluxapps-yanguis
odl-dluxapps-topology          odl-openflowjava-all
odl-openflowplugin-flow-services-ui odl-l2switch-all
```

登录我们的 OpenDaylight 控制台，依次点击 Yangman，下拉到最后，你会发现我们所自定义的 topology 插件，如图 5.19 所示：

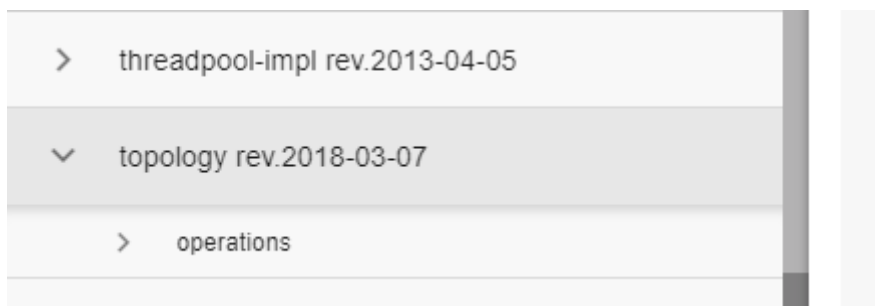


图 5.19 odl 控制台显示已安装插件

然后点击进入，点击 operations，继续点击 list-links-info，这个名字即为我们编写的方法名。然后在右侧直接点击 send，我们会发现下方返回的数据为空，如图 5.20 所示：

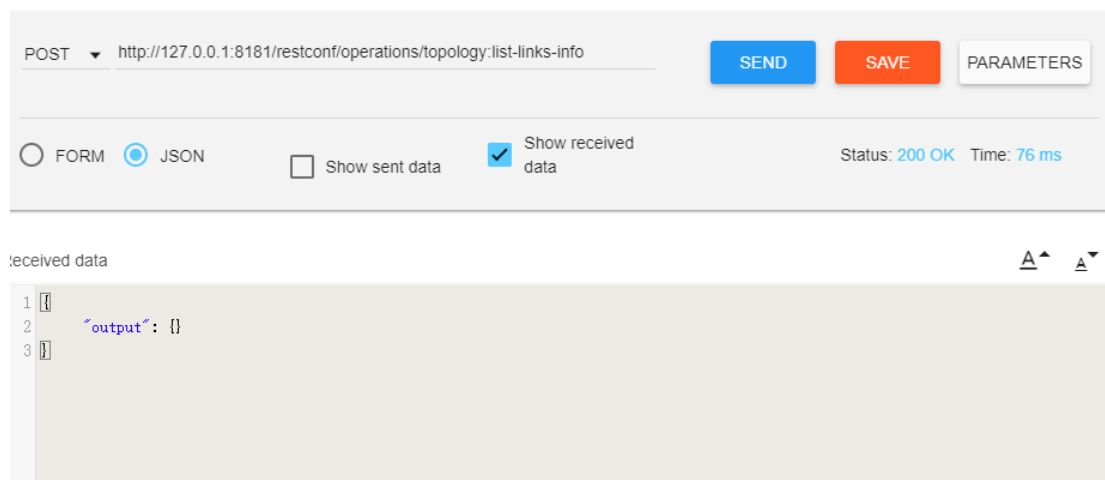


图 5.20 odl 控制台已安装插件页面

这是因为我们还没有连接 mininet。

首先查看自己 OpenDaylight 控制器所在主机的 ip 地址：

启用 DHCP	是
IPv4 地址	192.168.1.105
IPv4 子网掩码	255.255.255.0
获得租约的时间	2018年5月10日 11:17:42
租约过期的时间	2018年5月10日 16:17:40
IPv4 默认网关	192.168.1.1
IPv4 DHCP 服务器	192.168.1.1
IPv4 DNS 服务器	192.168.254.141
	192.168.254.245

图 5.21 查看主机 ip

打开 mininet，使用以下命令进行连接：

```
sudo mn --mac --topo=linear,3 --switch=ovsk,protocols=OpenFlow13
--controller=remote,ip=192.168.1.105
```

Mininet 提示连接成功：

```
cang@cang-virtual-machine:~$ sudo mn --mac --topo=linear,3 --switch=ovsk,protocols=OpenFlow13 --controller=remote,ip=192.168.1.105
[sudo] cang 的密码:
*** Creating network
*** Adding controller
Connecting to remote controller at 192.168.1.105:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet>
```

图 5.22 mininet 成功连接控制器

此时我们在 OpenDaylight 控制台上重新 SEND，发现返回信息为如图 5.23 所示：

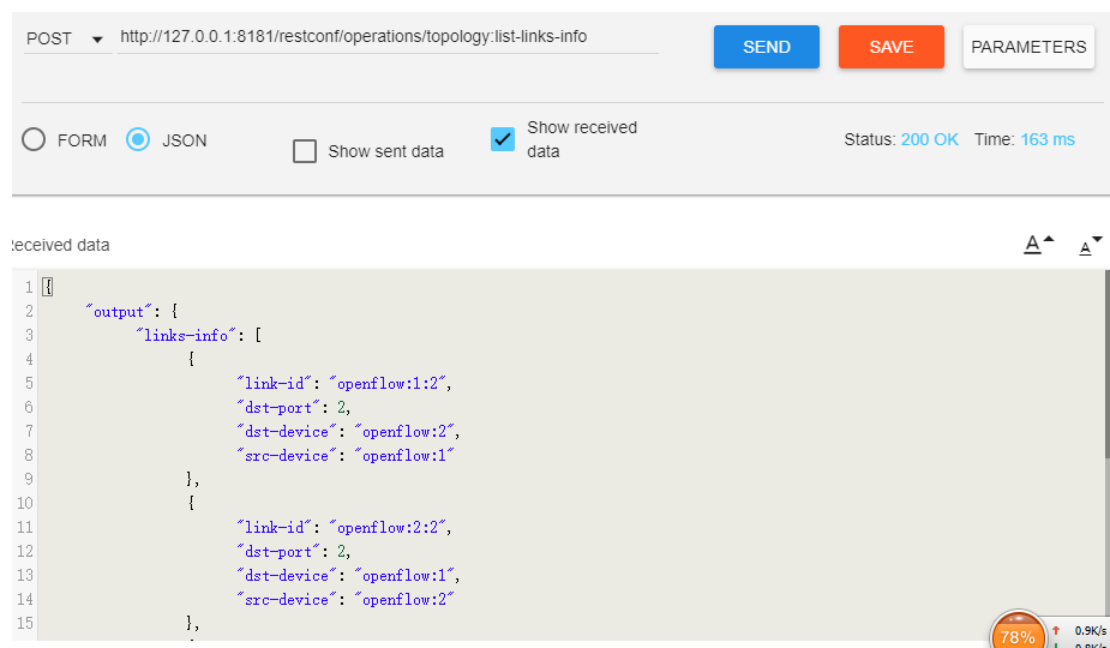


图 5.23 odl 控制台返回的 mininet 网络拓扑中的节点信息

发现已经有关于链路连接的信息了，此时说明我们的插件开发已经成功。

## 第 6 章 使用 Java 开发接入 ODL 插件与 Python 核心代码

上一章节中我们介绍了如何为 OpenDaylight 控制器编写相关插件，为我们使用自定义插件获取相关信息奠定了基础。在本章节，我们将介绍如何使用 Java 语言对 OpenDaylight 控制器进行接入，即使用 Java 语言进行编程调用 OpenDaylight 的 REST API 接口，以达到实现我们可以使用编程自行进行我们所需的相关信息获取与相关流表信息的下发功能，从而实现 Java 开发的接入。在另一个方面，我也会在本章介绍 Java jni 技术的使用。由于本人能力有限，对 Jennifer 教授提供的 Fibbing 技术的核心代码理解不是很充分，因此在此不会使用 Fibbing 技术的核心代码进行接入，而是使用一个小例子对此方法进行介绍，证明本思路的可行性。

### 6.1 使用 Java 开发接入 ODL 接口

#### 6.1.1 接口连接参数分析

OpenDaylight 插件所提供的功能都是使用 REST API 的形式提供给外界使用的，即使是 OpenDaylight 控制器自己的控制台也是如此。因此我们需要知道自己需要获取哪些信息，而要获取这些信息所需要访问的接口是那些。我通过从网络上查找相关个人博客及网站，加上从 OpenDaylight 官网上提供的官方文档，找到了我们需要的接口，接下来我会对这些接口进行分析。

1. 阅读官方文档。我在 OpenDaylight 官网文档中找到所需的信息如图 6.1 所示：

## Restconf for Inventory

The REST url for listing all the nodes is:

```
http://localhost:8181/restconf/operational/.opendaylight-inventory:nodes/
```

You will need to set the Accept header:

```
Accept: application/xml
```

You will also need to use HTTP Basic Auth with username: admin password: admin.

Alternately, if you have a node's id you can address it as

```
http://localhost:8181/restconf/operational/.opendaylight-inventory:nodes/node/<id>
```

for example

```
http://localhost:8181/restconf/operational/.opendaylight-inventory:nodes/node/openflow:1
```

图 6.1.opendaylight 官方文档对接口的描述

官网上提供的监听所有节点信息，即获取所有节点信息的接口为：

```
http://127.0.0.1:8181/restconf/operational/.opendaylight-inventory:nodes/
```

具体的使用方法为：对该网址发起 HTTP GET 请求，设置 Accept 请求头为 Accept:application/xml，并且需要使用 HTTP Basic Auth 来进行用户名和密码的认证。当你要访问一个具体的节点的时候，可以在该网址后面继续添加要访问的节点路由器的 id，此时接口网址如下：

```
http://localhost:8181/restconf/operational/.opendaylight-inventory:nodes/node/<id>
```

Localhost 需要改为 OpenDaylight 控制器所在主机的 ip 地址，<id>需要修改为目标节点路由器的真实 id。

2. 官网上所给的建议是使用一款叫 PostMan 的谷歌浏览器插件进行接口的访问[12][13]。

### How to hit RestConf with Postman

Install Postman for Chrome

In the chrome browser bar enter

```
chrome://apps/
```

And click on Postman.

Enter the URL. Click on the Headers button on the far right. Enter the Accept: header. Click on the Basic Auth Tab at the top and setup the username and password. Send.

图 6.2.opendaylight 官方文档对插件的描述



我们打开谷歌浏览器，输入 chrome://apps/ 进入谷歌应用中心，搜索 postman 插件并下载。下载完成之后，打开插件显如图 6.3 所示。

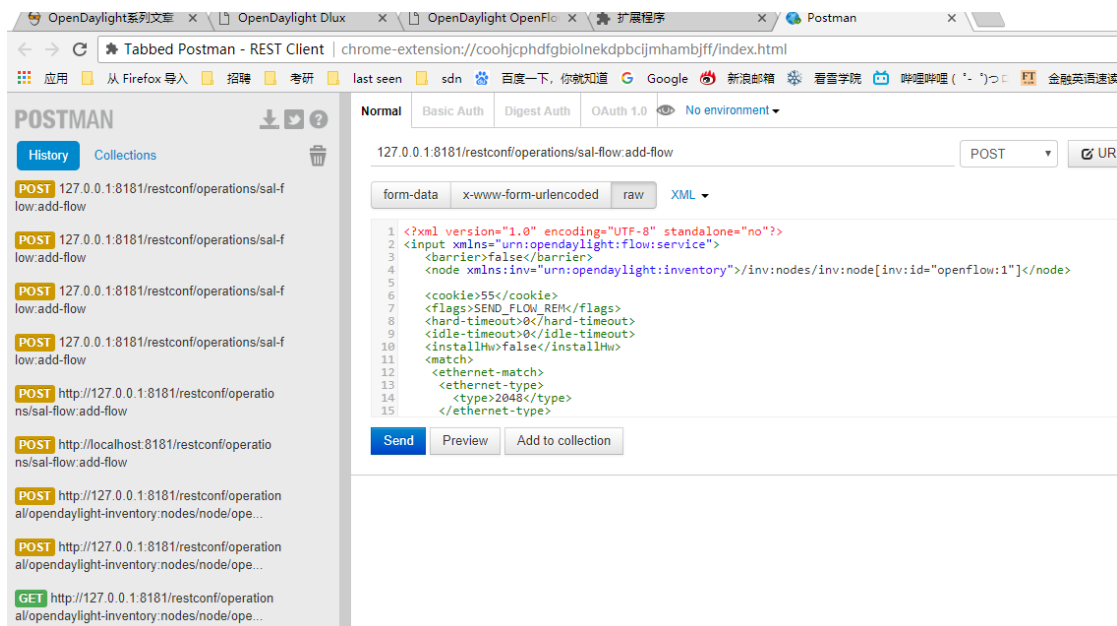


图 6.3 postman 插件页面

我们可以使用该插件进行对该接口的访问。当然，经过对该链接的分析，我发现也可以在 OpenDaylight 控制器中找到该访问接口。

打开 OpenDaylight 控制器的控制台页面，点击 YANGman，从 modules 列表中找到 OpenDaylight-inventory，如图 6.4 所示：

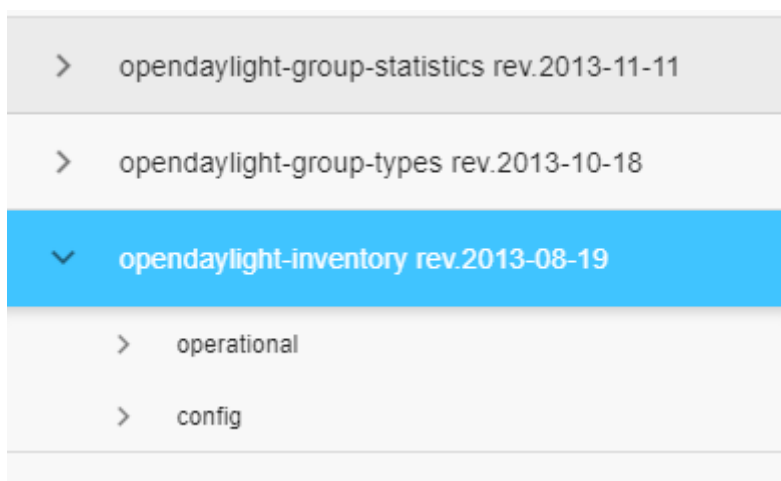


图 6.4 odl 控制台找到接口

点击进入，我们会发现和 postman 有一样的界面，并且所显示的接口链接也是和我们所要使用的接口链接是相同的，如图 6.5 所示：

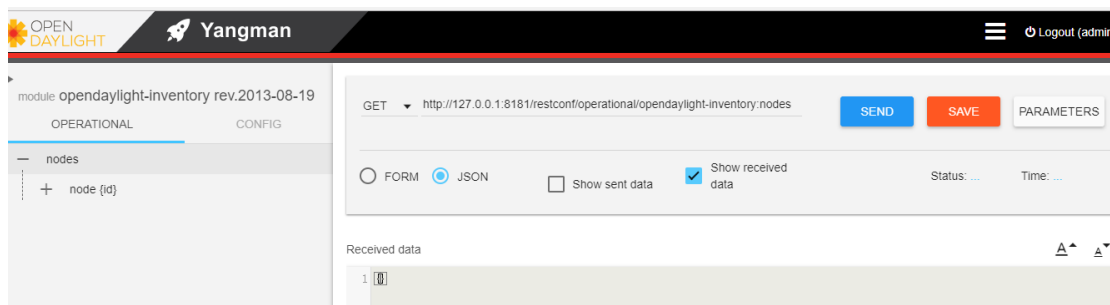


图 6.5 odl 控制台接口界面

3. 点击 F12 打开控制台，找到 network 界面，我们准备进行页面发送请求的抓包分析。再次点击 SEND，抓取到的结果如图 6.6 所示：

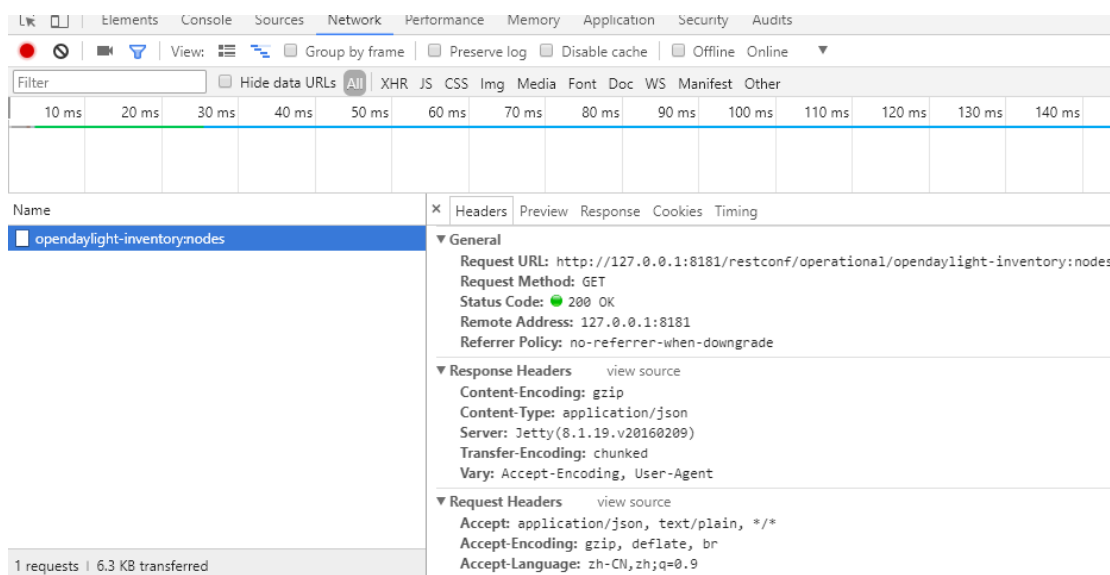


图 6.6 对 odl 控制台抓包，显示包的信息

通过对该抓取的包进行分析，我们得到了这么几组数据：

请求的网络地址为：

`http://127.0.0.1:8181/restconf/operational/.opendaylight-inventory:nodes`

请求的方法为：GET

请求头信息为：

`Accept: application/json, text/plain, */*`

`Accept-Encoding: gzip, deflate, br`

```
Accept-Language: zh-CN, zh;q=0.9
Authorization: Basic YWRtaW46YWRtaW4=
Connection: keep-alive
Cookie: JSESSIONID=122ddxkdbjbws1plle8d0zzt4v
DNT: 1
Host: 127.0.0.1:8181
Referer: http://127.0.0.1:8181/index.html
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/66.0.3359.139 Safari/537.36
```

对此抓包结果进行分析，我们在使用 Java 语言进行模拟提交表单的时候，要设置请求方法为 GET，并且要在请求头中设置 Accept 头，Authorization 属性，User-Agent 属性，Connection 属性，附加 Accept-Language 属性和 Accept-Encoding 属性。

但是这里有一个 Authorization，其属性值我们无法得知是如何生成的，但根据官方文档介绍，这个属性是使用用户名密码进行计算的，是必须的。因此需要继续对这个属性进行分析。

4. 随后，我们对 postman 源码进行测试，在 basic auth 界面输入用户名密码，点击 refresh 后，出现了 Basic Authorization，如图 6.7 所示：

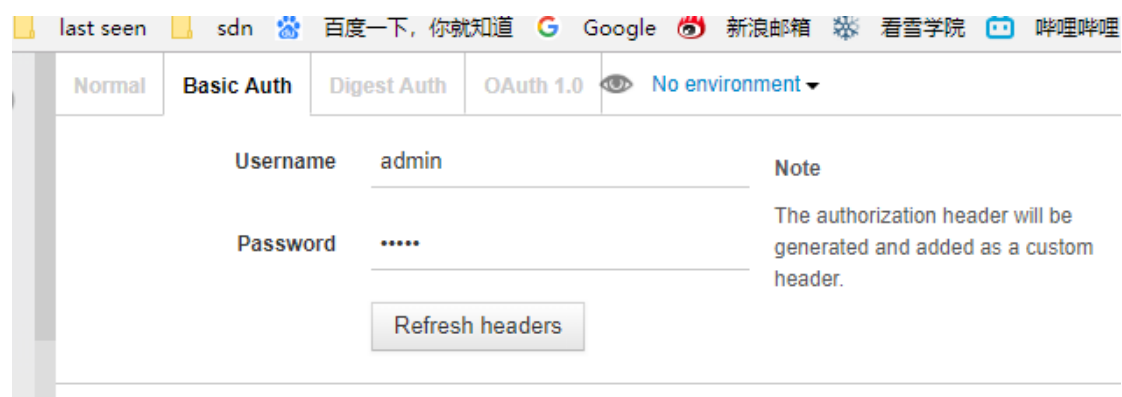


图 6.7 postman 插件中对用户名密码的处理截图

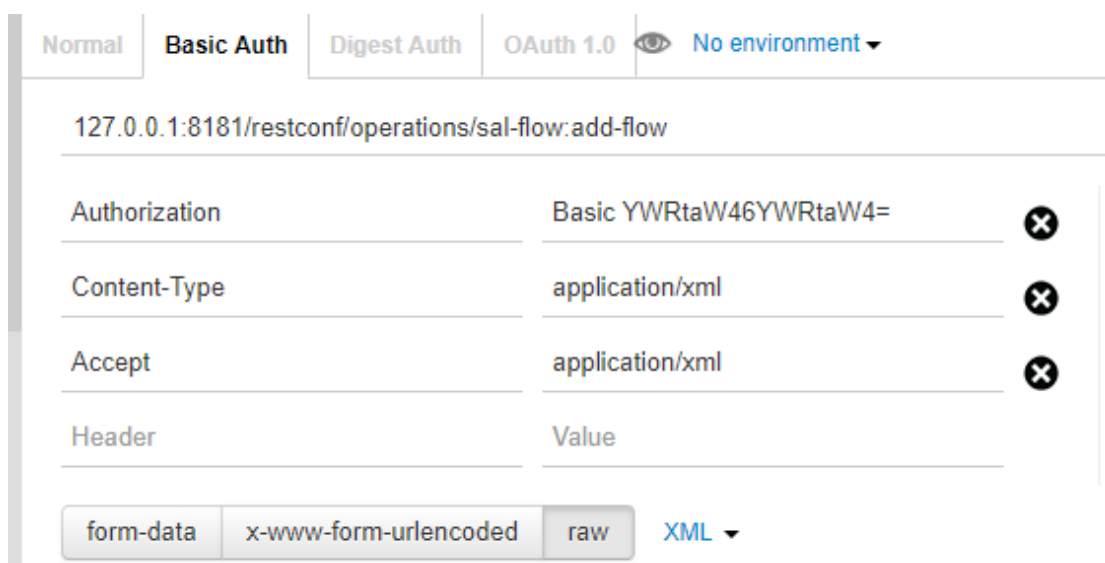


图 6.8 postman 插件中设置的请求头信息

我们发现这个 Authorization 和控制台里抓包获取的是一样的，因此我估计是用的同一种算法，所以我对点击该按钮所调用的方法进行了查找，发现如图 6.9 所示的方法。

```
basic: {
  process: function () {
    var headers = pm.request.headers;
    var authHeaderKey = "Authorization";
    var pos = findPosition(headers, "key", authHeaderKey);

    var username = $('#request-helper-basicAuth-username').val();
    var password = $('#request-helper-basicAuth-password').val();

    username = pm.envManager.convertString(username);
    password = pm.envManager.convertString(password);

    var rawString = username + ":" + password;
    var encodedString = "Basic " + btoa(rawString);

    if (pos >= 0) {
      headers[pos] = {
        key: authHeaderKey,
        name: authHeaderKey,
        value: encodedString
      };
    }
  }
}
```

图 6.9 postman 插件中设置 auth 属性的源代码

经过从网上搜索该 btoa 方法，我发现这个方法是 JavaScript 用于进行 base64 编码的函数。因此我尝试使用 Java 编写方法进行对用户名和密码的编

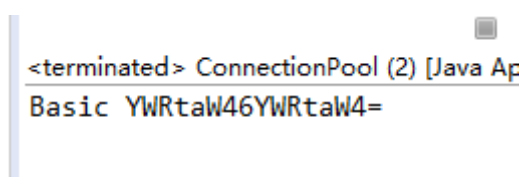
码，代码如图 6.10 所示：

```
private String createAuth(String name, String pwd) {
    String tmp = name + ":" + pwd;
    String ori = "Basic " + Base64.getEncoder().encodeToString(tmp.getBytes());
    return ori;
}

public static void main(String[] args) {
    new ConnectionPool().createAuth("admin", "admin");
}
```

图 6.10 java 编写编码 auth 属性的代码

输入用户名密码之后输出的结果如图 6.11 所示：



```
<terminated> ConnectionPool (2) [Java Ap
Basic YWRtaW46YWRtaW4=
```

图 6.11 java 编码 auth 的输出

输出结果与 Postman 和 OpenDaylight 控制台抓包获取的 auth 都一样，证明已经成功破解 auth 的生成方式。所有需要的属性都已具备，接下来要进行 Java 接入 OpenDaylight 控制器的开发。

### 6.1.2 接口数据格式分析

本小节主要介绍 3 个接口的数据格式，对其中返回数据的各个属性只会找几个重要的进行解释，全部的含义请参考官网上的官方文档[7][14]。

1. 得到所有节点的信息，接口 url 为：

`http://127.0.0.1:8181/restconf/operational/.opendaylight-inventory:nodes`

请求方法为 GET，返回数据为 json 格式，由于数据太长，此处只进行部分数据的截图。



```

{
  "nodes": {
    "node": [
      {
        "id": "openflow:1",
        "node-connector": [
          "flow-node-inventory:switch-features": {
            "flow-node-inventory:manufacturer": "Nicira, Inc.",
            "flow-node-inventory:ip-address": "192.168.1.107",
            "flow-node-inventory:software": "2.5.4",
            "flow-node-inventory:serial-number": "None",
            "flow-node-inventory:port-number": 50926,
            "flow-node-inventory:table": {
              "flow-node-inventory:hardware": "Open vSwitch",
              "flow-node-inventory:description": "s1",
              "flow-node-inventory:snapshot-gathering-status-start": {
                "flow-node-inventory:snapshot-gathering-status-end": {
                  "opendaylight-group-statistics:group-features": {
                    "id": "openflow:3",

```

图 6.12 odl 接口返回的 json 数据部分截图

如上图所示，使用该接口请求会返回一个相当庞大的 json 格式数据，其中包括了所有的节点信息。此处 node[] 里便保存了 3 个节点路由器信息。途中所示的是 id 为 openflow:1 的节点路由器。

Node-connector 属性里保存了和该节点相邻的节点的信息。

flow-node-inventory:xxx 等这些属性里保存了该节点的配置或者属性，比如，manufacturer 为节点路由器硬件制造商；ip-address 为节点路由器的 ip 地址，由于这里使用了默认设置所有所有的节点的 ip 全为虚拟机获得的 ip；hardware 为硬件是 OVS；description 为该节点路由器的描述，这里默认设置节点路由器名为 s1。

## 2. 得到某个节点的所有流表。接口 url 为：

```

http://127.0.0.1:8181/restconf/operational/opendaylight-inventory:nodes/node/
<id>/flow-node-inventory:table/<whichTable>
    
```

其中 id 为要得到的某个节点路由器的 id；whichTable 为你想要得到的节点路由器的哪个流表。

本测试用例使用了得到 id 为 openflow:1 且获得 0 级流表，获得返回数据如下图所示：

```

"flow-node-inventory:table": [
  {
    "id": 0,
    "opendaylight-flow-table-statistics:flow-table-statistics": {
      "flow": [
        {
          "id": "L2switch-1",
          "idle-timeout": 0,
          "cookie": 3098476543630901000,
          "flags": "",
          "hard-timeout": 0,
          "instructions": {
            "cookie_mask": 0,
            "opendaylight-flow-statistics:flow-statistics": {
              "priority": 2,
              "table_id": 0,
              "match": {
                "id": "L2switch-2",
                "idle-timeout": 0,

```

图 6.13 odl-table 接口返回的 json 数据部分截图

如图 6.13 所示，在 table 里存放了多条流表，每条流表里存放一个节点路由器的信息，图中显示为 id 为 L2switch-1 的路由器，priority 权重为 2，table-id 是 0；match 里为 in-port: 2, 即说明该路由器节点使用的进入的端口是 2 号端口。

### 3. 下发流表信息，接口为：

<http://127.0.0.1:8181/restconf/operations/sal-flow:add-flow>

测试发送的流表数据如图 6.14 所示：

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?><input xmlns="urn:opendaylight:flow:service"><barrier>false</barrier><node
xmlns:inv="urn:opendaylight:inventory">/inv:nodes/inv:node[inv:id="openflow:1"]</node><cookie>55</cookie><flags>SEND_FLOW_REM<,
timeout>0</hard-timeout><idle-timeout>0</idle-timeout><installHw>false</installHw><match><ethernet-match><ethernet-type><type
type></ethernet-match><ipv4-destination>10.0.10.2/32</ipv4-destination></match><instructions><instruction><order>0</order><apply
<output-action><output-node-connector>1</output-node-connector></output-action><order>0</order></action></apply-actions></in
<priority>0</priority><strict>false</strict><table_id>0</table_id></input>

```

图 6.14 测试发送的流表数据部分截图

主要看划红线的部分，这里想要修改下该节点的 ip 地址，执行之后没有任何返回信息，也没有报错，此时在使用查询命令查询流表信息，返回数据如图 6.15：

```

        "priority": 0,
        "table_id": 0,
        "match": {
            "ipv4-destination": "10.0.10.2/32",
            "ethernet-match": {
                "ethernet-type": {
                    "type": 2048
                }
            }
        }
    },
    {
        "id": "ETHERNET-2048"
    }
}

```

图 6.15 查询节点信息返回部分数据

由返回的数据可知，修改成功。

### 6.1.3 代码具体实现

代码实现使用了 Jsoup+Json 的数据格式，在对 POST 请求上使用的是 Java 原生的 url 包发送的请求，因为整个消息体不是属性:属性值的形式，而是整个消息体全部都是发送的数据格式。而 Jsoup 只能发送键值对形式的数据格式，因此无法使用 jsoup。

代码实现编写了一个 ConnectionPool 类，类中的每个方法为一个接口的请求方法。由于每个请求头都相同，因此为了减少编码长度，使用了一个统一的方法进行请求头的添加。如图 6.16 所示：

```

private void addHeaderInfo(Connection conn, Method method) {
    conn.header("Authorization", auth).header("Accept", "application/json, text/plain, */")
        .header("Accept-Encoding", "gzip, deflate, br").header("Accept-Language", "zh")
        .header("Connection", "keep-alive")
        // .header("Content-Type", "application/json; charset=utf-8")
        .ignoreContentType(true).header("DNT", "1")
        .header("User-Agent",
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
        .method(method);
}

```

图 6.16 java 实现添加请求头代码

关于 auth 的生成，我在创建 ConnectionPool 类的时候构造方法中进行设置，不输入 auth 用户名密码则会报错从而无法使用，图 6.17 为构造方法代码：



```

public ConnectionPool(String name, String pwd) {
    auth = createAuth(name, pwd);
}

public ConnectionPool() {}
if (auth == null) {
    throw new NullPointerException("No authorization!Please use 'new ConnectionPool
}
}

```

图 6.17 java 实现请求类的构造方法

计算 Basic Auth 的私有方法如图 6.18。

```

private String createAuth(String name, String pwd) {
    String tmp = name + ":" + pwd;
    String ori = "Basic " + Base64.getEncoder().encodeToString(tmp.getBytes());
    return ori;
}

```

图 6.18 java 实现计算 Basic Auth 的私有方法

整体添加完成之后，运行获取单个节点信息的结果如图 6.19 所示：



```

3*import java.io.IOException;
9
10 public class Try2Connect {
11
12     private static final Try2Connect dao = new Try2Connect();
13
14     public static void main(String[] args) {
15         String name = "admin";
16         String pwd = "admin";
17
18         ConnectionPool cp = new ConnectionPool(name, pwd);
19         cp.getNode("openflow:1");
20         // cp.getNodeTable("openflow:1", 0);
21         // cp.getNodesInfo();
22         // cp.addFlow_UseHttpPost("", "", "", "xml");
23     }
24

```

```

<terminated> Try2Connect [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (2018年5月10日 下午8:50:56)
{"node":{"id":"openflow:1","node-connector":[{"id":"openflow:1:1","opendaylight-port-statistics:flow-capable-node-connector-statistics

```

图 6.19 运行获取单个节点信息的结果

## 6.2 使用 Java 进行 Python 调用

该部分介绍如何使用 Java 进行 Python 的调用，由于本人不擅长 Python 编程，因此这里先介绍一个小 demo。

此处使用的 Java 第三方 jar 包为 jython，可以在 java 中对 python 进行调用。方法如下：

首先在一个文件夹下编写一个简单的 python 文件，如图 6.20 所示：

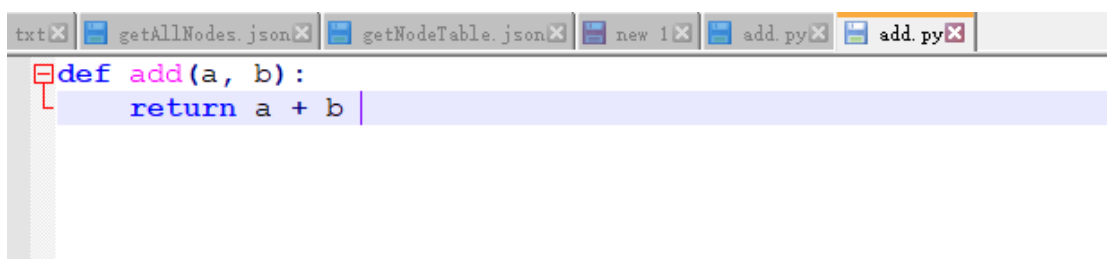


图 6.20 python 文件中的代码

其中实现了一个 add 方法。

然后编写一个简单的 Java 程序对其进行调用，如图 6.21 所示。

```
public static void main(String[] args) {
    PythonInterpreter interpreter = new PythonInterpreter();
    interpreter.execfile("E:\\add.py");
    PyFunction func = (PyFunction) interpreter.get("add", PyFunction.class);

    int a = 2010, b = 2;
    PyObject pyobj = func.__call__(new PyInteger(a), new PyInteger(b));
    System.out.println("a+b = " + pyobj.toString());
}
```

图 6.21 java 调用 python 数据代码

使用 PyFunction 在用 PythonInterpreter 类解析的 python 对象中获得所需要调用的方法，然后使用 call 方法传入参数，随即运行，得到运行结果如图 6.22 所示：

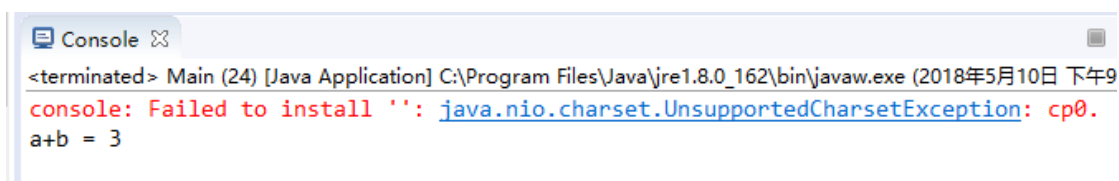


图 6.22 java 调用 python 执行结果

实现输出 1+2=3，表明程序调用成功。

## 第 7 章 结论

本文章主要讲述了对 Jennifer 教授关于实现 SDN 集中控制下的分布式网络的论文《Central Control Over Distributed Routing》的解读，讲了如何安装搭建一个 OpenDaylight 控制器+Mininet 虚拟网络测试平台+OpenVswitch 交换机的集成测试环境。并在此基础上，尝试对 OpenDaylight 控制器进行 Java 开发接入，并使用 Java 对 Python 语言程序进行调用。文章中分别采用了几个例子，并取得了良好的测试结果，说明了该思路的正确性以及可行性。

只不过由于时间的原因，本文所提及的任务只做了一部分，还剩下一小部分没有完成，比如对 Jennifer 教授提到的 Fibbing 技术的核心代码的解读，以及使用 Java 对该核心代码进行接入，继而运行在我们所搭建好的平台上。但我们已经论述了该思路的正确性以及可行性，我认为接下来只要有充足的时间，我们完全可以实现这个思路，实现在另一个测试平台上运行 Jennifer 教授的 Fibbing 技术，而不是只在 Quagga 软件搭建的网络平台上。

因此，在接下来的时间里，对 Fibbing 核心代码的解读是下一个主要需要完成的任务。与此同时，我们也应该注意到，我们对 OpenDaylight 控制器的接入也存在一定的问题，比如对各种插件的尝试。我们只编写了 RESTAPI 插件，然后对外提供接口，但我们仍没有找到一个更合理，更有效的方式让我们所要接入的协议自然的运行在 OpenDaylight 控制器上的方法。因此，对 OpenDaylight 控制器的深入学习在接下来的学习中也不可少。

## 致谢

在本篇文章即将完成之际，想到我本科阶段的学习生活，想到我在撰写论文的时候的经历，我想衷心感谢我的导师，是导师为我提供了解决问题的思路，是导师在我遇到问题的时候耐心听我的疑惑，耐心的解答我提出的问题，从而让我在学习的思路上有新的探索，新的思考，新的收获。是老师一丝不苟的教学态度，育人态度感染了我，让我能够冷静的、耐心的完成我的任务，完成本片论文。也感谢我的同学们和舍友们，在我遇到困难的时候给我以极大的鼓舞和勇气，让我能够继续下去。

在此，我要向老师们和同学们，向那些在我学习成长的道路上支持我的人，鼓舞我的人，发表真诚的敬意和衷心的感谢！

## 参考文献

- [1]. 黄韬、刘江、魏亮主编,《软件定义网络核心应原理与用实践 (第二版)》,人民邮电出版, 2016. 09
- [2]. 谢希仁主编,《计算机网络 (第 6 版)》, 电子工业出版社, 2012.06
- [3]. Jennifer Rexford,《Central Control Over Distributed Routing》, 2015
- [4]. <http://www.cnblogs.com/keepthebeats/p/4976574.html>, <<阅读笔记>>
- [5]. <http://fibbing.net/>, Fibbing 官方网站
- [6]. <https://www.sdnlab.com/community/article/974> , OpenDaylight 系列文章
- [7]. [https://wiki.opendaylight.org/view/OpenDaylight\\_OpenFlow\\_Plugin:End\\_to\\_End\\_Flows](https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:End_to_End_Flows), End to End Flows, 流表返回数据介绍,.opendaylight 维基百科
- [8]. <https://www.cnblogs.com/liu424/p/8186001.html> , 基于组表的简单负载均衡
- [9]. <http://docs.opendaylight.org/en/stable-oxygen/developer-guide/index.html> ,  
.opendaylight 官网文档
- [10]. <http://docs.opendaylight.org/en/stable-boron/developer-guide/developing-apps-on-the-odl-controller.html> , 如何在 odl-controller 上设计控制器,  
.opendaylight 官方文档
- [11]. <https://www.sdnlab.com/16902.html> , odl 新建 hello 工程
- [12]. <https://www.sdnlab.com/16824.html>, 如何用 postman 控制 ODL 查看和下发流表
- [13]. <https://www.sdnlab.com/15173.html> , odl 简单应用及流表操作指南
- [14]. <https://blog.csdn.net/jh19900712/article/details/24783959> , OpenFlow 交换机之流表
- [15]. <https://blog.csdn.net/hsj521li/article/details/46043733> , Ubuntu 下  
Openaylight 导入到 eclipse 教程

## 附录 1 英文原文

### Central Control Over Distributed Routing

#### 0. ABSTRACT

*Centralizing routing decisions offers tremendous flexibility, but sacrifices the robustness of distributed protocols. In this paper, we present Fibbing, an architecture that achieves both flexibility and robustness through central control over distributed routing. Fibbing introduces fake nodes and links into an underlying linkstate routing protocol, so that routers compute their own forwarding tables based on the augmented topology. Fibbing is expressive, and readily supports flexible load balancing, traffic engineering, and backup routes. Based on high-level forwarding requirements, the Fibbing controller computes a compact augmented topology and injects the fake components through standard routing-protocol messages. Fibbing works with any unmodified routers speaking OSPF. Our experiments also show that it can scale to large networks with many forwarding requirements, introduces minimal overhead, and quickly reacts to network and controller failures.*

#### 1. INTRODUCTION

Consider a large IP network with hundreds of devices, including the components shown in Fig. 1a. A set of IP addresses ( $D_1$ ) see a sudden surge of traffic, from multiple entry points ( $A$ ,  $D$ , and  $E$ ), that congests a part of the network. As a network operator, you suspect a denial-of-service attack (DoS), but cannot know for sure without inspecting the traffic as it could also be a

flash crowd. Your goal is therefore to: (i) isolate the flows destined to these IP addresses, (ii) direct them to a scrubber connected between  $B$  and  $C$ , in order to “clean” them if needed, and (iii) reduce congestion by load-balancing the traffic on unused links, like  $(B,E)$ .

Performing this routine task is very difficult in traditional networks. First, since the middlebox and the destinations are not adjacent to each other, the configuration of multiple devices needs to change. Also, since intra-domain routing is typically based on shortest path algorithms, modifying the routing configuration is likely to impact many other flows not involved in the attack. In Fig. 1a, any attempt to reroute flows to  $D_1$  would also reroute

flows to D2 since they home to the same router. Advertising D1 from the middlebox would attract the right traffic, but would not necessarily alleviate the congestion, because all D1 traffic would

traverse (and congest) path (A,D,E,B), leaving (A,B) unused. Well-known Traffic-Engineering (TE) protocols (e.g., MPLS RSVP-TE [1]) could help. Unfortunately, since D1 traffic enters the network from multiple points, many tunnels (three, on A, D, and E, in our tiny example) would need to be configured and signaled. This increases both control-plane and data-plane overhead. Software Defined Networking (SDN) could easily solve the problem as it enables centralized and direct control of the forwarding behavior. However, moving away from distributed routing protocols comes at a cost. Indeed, IGPs like OSPF and IS-IS are scalable (support networks with hundreds of nodes), robust, and quickly react to failures. Building a SDN controller with comparable scalability and reliability is challenging. It must compute and install forwarding rules for all the switches, and respond quickly to topology changes. Even the simple task of updating the switch rule tables can then become a major bottleneck for a central controller managing hundreds of thousands of rules in hundreds of switches. In contrast, distributed routing protocols naturally parallelize this work. For reliability and scalability, a SDN controller should also be replicated and geographically distributed, leading to additional challenges in managing controller state. Finally, the deployment of SDN as a whole is a major hurdle as many networks have a huge installed base of devices, management tools, and human operators that are not familiar with the technology. As a result, existing SDN deployments are limited in scope, e.g., new deployments of private backbones [8, 9] and software deployments at the network edge [10]. This paper introduces Fibbing, a technique that offers direct control over the routers' forwarding information base (FIB) by manipulating the input of a distributed routing protocol. Fibbing relies on traditional link-state protocols such as OSPF [5] and IS-IS [6], where routers compute shortest paths over a synchronized view of the topology. Fibbing controls routers by carefully lying to them, removing the need to configure them. It coaxes the routers into computing the target forwarding entries by presenting them with a carefully constructed augmented topology that includes fake nodes (providing fake announcements of destination address blocks) and fake links (with fake weights). In essence, Fibbing inverts the routing function: given the forwarding entries (i.e., the desired output) and the routing protocol (i.e., the function), Fibbing automatically computes the routing messages to send to the routers (i.e., the input). Fibbing can solve the problem in Fig. 1a adding two fake nodes (Fig. 1b), connected to A and E with the depicted weights. Both fake nodes advertise that they can reach D1 directly. Based on the augmented topology, D starts to use A to reach D1, as the new cost (3) is lower than the original one (6). A and E also select different paths. Since the fake nodes do not really exist, packets forwarded by A or E actually flow through B. Routers B and C do not change their forwarding decisions.

Table 1 gives an overview of how Fibbing improves flexibility and manageability by adopting a SDN-like approach while keeping the advantages of distributed protocols (e.g., robustness and fast FIB modifications).

Fibbing is expressive. Fibbing can steer flows along any set of per-destination loop-free paths. In other words, it can exert full control at a per-destination granularity. For this reason, Fibbing readily supports advanced forwarding applications such as: (a) traffic engineering,

(b) load balancing, (c) fast failover, and (d) traffic steering through middleboxes. By relying on destination-based routing protocols, Fibbing does not support finer-grained routing and forwarding policies such as matching on port numbers. Though, those policies can easily be supported via middleboxes.

Fibbing scales and is robust to failures. Lying to routers is powerful but challenging. Indeed, Fibbing must be fast in computing augmented topologies to avoid loops and blackholes upon network failures. At the same time, Fibbing must compute small augmented topologies since routers have limited resources. Finally, Fibbing must be reliable and gracefully handle controller failures. We address all three challenges.

Fibbing differs from previous approaches that rely on routing protocols to program routers. Prior approaches like the Routing Control Platform [11] rely on BGP as a “poor man’s” SDN protocol to install a forwarding rule for each destination prefix on each router. In contrast, Fibbing leverages the routing protocol implementation on the routers. Doing so, Fibbing can adapt the forwarding behavior of many routers at once, while allowing them to compute forwarding table entries and converge on their own. That is, while the controller computes the routing input centrally, the routing output is still computed in a distributed fashion.

Fibbing works on existing routers. We implemented a fully-functional Fibbing prototype and used it to path network per-destination augmented reduced  $\{\text{SEP}\}$  running reqs. topology forwarding DAGs topology topology network Figure 2: The four-staged Fibbing workflow.

gram real routers (both Cisco and Juniper). Based on an augmented topology, these routers can install hundreds of thousands of forwarding entries with an average installation time of less than 1ms per entry. This offers much greater scale and faster convergence than is possible with state-of-the-art SDN switches [12, 13], without requiring the deployment of new equipment and per-device actions from the controller. This also means that Fibbing can implement recent SDN proposals, like Google’s B4 [8] and Microsoft’s SWAN [9]—on top of existing networks.

Our earlier work showed that Fibbing can enforce any set of forwarding DAGs [14]. This paper goes further by describing the complete design, implementation, and evaluation of a Fibbing controller managing intradomain routing. Rather than focusing on specific use cases (like traffic engineering), we describe its support for different higher-level approaches (e.g., [8, 9]). We make the following contributions:

- Abstraction: We show how to express and realize high-level forwarding requirements by manipulating a distributed link-state routing protocol (§2).
- Algorithms: We propose new, efficient algorithms to compute compact augmented topologies (§3).
- Implementation: We describe a complete Fibbing implementation which works with unmodified Cisco and Juniper routers (§4).
- Evaluation: We show that our Fibbing controller quickly generates small augmented topologies, inducing minimal load on routers (§5).

## 2. FLEXIBLE FIBBING

Fibbing workflow proceeds in four consecutive stages based on two inputs: the desired forwarding graphs (one Directed Acyclic Graph, or DAG, per destination) and the IGP topology (Fig. 2). The forwarding DAGs can either be provided directly or expressed indirectly, at a



high-level, using a simple path-based language. In the latter case, the Compilation (§2) stage starts by compiling the requirements into concrete forwarding DAGs. Then, the Topology Augmentation (§3) stage computes an augmented topology that achieves these forwarding DAGs. While computing an augmented topology is fast, the resulting topology can be large. As such, the job of the next Topology Optimization (§3) stage is to reduce the augmented topology while preserving the forwarding paths. Finally, the Injection & Mon-

itoring (§4) stage turns fake information into actual “lies” that the controller injects into the network.

pol ::= (s1;...;sn) Fibbing Policy  
 s ::= p | b Requirement r ::= p1 and p2 | p1 or p2 | p Path Req. p ::= Path(n+) Path Expr. n ::= id | \* | n1 and n2 | n1 or n2 Node Expr. b ::= r as backupof ((id1,id2)+) Backup Req.

Figure 3: Syntax of Fibbing high-level language.

In this section, we present the high-level language and compilation process (§2.1), and show that Fibbing can express a wide range of forwarding behaviors (§2.2).

### 2.1 Fibbing high-level language

Fibbing language (Fig. 3) provides operators with a succinct and easy way to specify their forwarding requirements. A Fibbing policy is a collection of requirements, naturally expressed as regular expressions on paths. Each requirement is either a path requirement which specifies how traffic should flow to a given destination, or a backup requirement which specifies how traffic should flow if the IGP topology changes. Each path requirement is recursively defined as a composition of path requirements through logical AND and OR. Operators can express load-balancing requirements using a

conjunction of n requirements. Similarly, they can ensure that traffic to a specific destination will take one of n paths (e.g., containing a firewall) using disjunction. Path requirements are composed of a sequence of node requirements. A node requirement is either a node (router) identifier or the wildcard \*, representing any sequence of nodes. Like path requirements, node requirements can be combined using logical AND and OR. Whenever no path requirement is stated, the original IGP paths should be used. This way, operators do not have to express all the unmodified paths, only deviations from the IGP shortest paths.

The following example illustrates the main features of the language. It states that traffic between E and D1 should be load-balanced on two paths, traffic between A and D2 should cross B or C and that traffic from F to D3 should be rerouted via G if the link (F,H) fails.

((E,C,D1) and (E,G,D1) ;  
 ((A,\* ,B,\* ,D2) or (A,\* ,C,\* ,D2) ) ; (F,G,\* ,D3) as backupof ((F,H) ) ) ; )

Fibbing policies are compiled into per-destination forwarding DAGs by finding convenient network paths (if any). Compilation works in two consecutive stages. First, the compiler expands any requirement with wildcards into paths. This step can be computationally expensive as, in general, a network can have a number of paths exponential in the number of nodes. While this is unlikely, especially for networks designed according to best practices, we bounded the number of paths that can be expanded out of a single requirement. We only expand again if no solution is found with the current set of paths. Once all requirements are expanded, the compiler groups them by destination and computes the Disjunctive Normal Form (DNF) of each requirement. To finally produce a forwarding DAG, the compiler iterates over the disjunction of path requirements and checks whether the resulting graph is loop-free.

## 2.2 Fibbing expressiveness

Beyond steering traffic along a given path (§1), we now show that Fibbing can also: (i) balance load over multiple paths and; (ii) provision backup paths.

Fibbing can forward any flow on any set of paths. Fibbing can load-balance traffic over multiple paths to maximize throughput, minimize response time, or increase reliability. For example, consider the network in Fig. 4a where three sources S1, S2, and S3 send traffic to three corresponding destinations. Demands and link capacities are such that link (F,G) is congested. One way to alleviate congestion is to split traffic destined to D2 over the top (via (B,C)) and bottom (via (F,G)) paths. Load-balancing traffic coming from E on multiple paths is possible under conventional link-state routing (e.g., by re-weighting links (F,G) to 15). However, this would force the traffic from S2 and S3 to spread over both paths, creating congestion. More generally, it is impossible to route the traffic destined to D2 and D3 on different links under conventional link-state routing.

Figure 4: Fibbing supports multi-path forwarding. Here, it avoids the initial congestion (see (a)) by load-balancing traffic for D2 (see (b)).

This simple requirement can easily be expressed as:

((S2 ,E ,B ,C ,H ,D2) and (S2 ,E ,F ,G ,H ,D2) )

Fig. 4b shows the augmented topology which realizes this requirement. A fake node announcing D1 (with a weight of 6) is inserted between E and B. After introducing this node, E has two shortest paths (of cost 7) to reach D2 and, hence, splits D2 traffic over B and F. In this example, Fibbing enables maximum network efficiency as each link is used to its full capacity.

Fibbing can provision backup paths for any flow. Fibbing can provision backup paths to prevent congestion or increased delays after link and node failures. As an illustration, consider the network in Figure 5a. The failure of link (E,F) leads to congestion since traffic flows for D1 and D2 are both rerouted to the same path via link (A,B). To prevent congestion, traffic destined to D1 and D2 should be split over the two remaining disjoint paths but only upon failure on the path (A,D,E,F). This is another example of a requirement that is impossible to achieve with link-state routing, and would require significant control-plane overhead in MPLS. In contrast, it is easily done with Fibbing.

Figure 5: Fibbing can provision backup paths. Here, possible congestion upon a link failure (see (a)) is avoided by adding a fake node (see (b)).

Backup paths can be specified in our language as:

(A ,B ,\* ,D1) and (A ,G ,\* ,D2) as backupof((A ,D) or (D ,E) or (E ,F) )

Fig. 5b shows the corresponding augmented topology, which has a single fake node advertising D2. Weights are set to prevent A from using the fake node to reach D2 unless a failure occurs along the path (A,D,E,F). While successful for this example, Fibbing cannot satisfy all possible requirements for backup paths (§3.3).

## 附录 2 译文

### 0. ABSTRACT

中心路由选择虽然提升了灵活性，但是牺牲了分布式协议的健壮性。这篇文章提出了 Fibbing（谎言），一种架构，在分布式路由上实现中心控制。Fibbing 以一个潜在的链路状态路由协议欺骗结点和链路，使得路由器可以基于增长的拓扑计算他们的转发表。Fibbing 是有表达性的，并且支持灵活的分路平衡、转发工程、且支持路由。基于高级转发需求，谎言控制器计算出一个紧凑的增长的拓扑并且通过标准路由协议信息注入欺骗元素。谎言可以与任何不可修改的支持 OSPF 协议的路由器兼容。我们的实验也显示了它可以通过转发需求、引进最小限制、快速反应网络状况和控制器的错误来扩大网络规模。

### 1. Introduction

作为网络管理员，怀疑网络上发生了 denial-of-service attack (Dos) [1]，要检查网络，需要做：

1. 孤立目标为这些 ip 地址的流
2. 将它们直接交给洗涤器，如果需要就清洗他们。
3. 通过路径平衡的方法利用未被使用的链路来降低拥塞。

这些工作在传统网络中非常难做。因为，中间件和目的地离得很远，许多设备的配置需要修改，修正一个错误会改变规则使得其他地方又发生错误。并且，因为内部路由基本上是基于最短路径算法的，修改单个的路由配置可能会影响其他多个不在攻击范围之内的网络流。

软件定义网络可以很容易地解决这些问题，因为它可以中心化并直接控制转发行为。然而，移除分布式路由协议又会造成一些问题。INDEED，像 OSPF 和 IS-IS 这样的 IGP 协议通常是可扩展的，强健的，对故障反应很迅速。对于 SDN 控制器来说，想建立一个有相当的稳定性和可靠性的 SDN 控制器是很有挑战的，他必须未全部的交换机计算和注册转发规则，还需要对拓扑变化迅速做出反应，甚至对一个控制成百上千台交换机规则的控制器来说，只是简单地更新交换机的路由表也可能会成为一个主要的瓶颈。可扩展性和可靠性会下降，由于所有工作都要由控制器来做，而分布式路由协议可以将这些工作分摊开。为了可扩展性和可靠性，SDN 控制器必须地理上分布式重复放置，但这样又会产生管理控制其状态的挑战。而且现有网络难以大量向 SDN 过渡，许多网络中已经注册了大量的基本设备，管理工具，并且操作员也不熟悉这项新技术，故 SDN 目前只是小范围部署。

Fibbing 欺骗路由器，让路由器以为链路中有一些不存在的结点或者链路，基于像 OSPF 或 IS-IS 这样的链路状态协议，让路由器计算转发路径。Fibbing 通过精心的欺骗而非配置来控制路由器。它提供给交换机精心设计好的增广拓扑来欺骗它们，其中拓扑里包含假节点（提供虚假声明目的地节点）和虚假链路（带虚拟权重）。必要时，控制器还会转换路由功能：给予转发条目和路由协议后，Fibbing 自动计算路由信息然后发送给交换机。

Fibbing 易于表述的。Fibbing 可以控制流按照有预设目的地的无环路径流动。换句话说，它可以达到完全控制按目的地转发的粒度。因为这个原因，Fibbing 可以很容易的支持先进的基于转发的应用，例如 交通工程，负载均衡，快速故障恢复，控制器路由集中控制。通过依赖基于目的地的路由协议，Fibbing 不支持细粒度的路由转发协议，但是代理可以很容易的支持这种协议。MiddleBox？

Fibbing scales 并且对错误很健壮？容错率高

Fibbing 不同于以前依赖路由协议为路由器编程的那些方法。

Fibbing 可以工作在现存路由器上。

我们之前的工作展示了 Fibbing 可以加强任何一种转发 DAG 图。这篇论文更深入的描述了完整的设计，实现，和对 Fibbing 路由器配置域内路由的评估。除了专注用具体的使用例子，我们也描述了它对不同的高级方法（approach）的支持。

他们做了如下工作：

1. 抽象方面：展示了如何通过操作分布式链路状态路由协议去表达和实现高级转发需求。
2. 算法方面：提出了新的、高效的算法去计算紧密增长的拓扑
3. 实现方面：描述了一个完整的实现，在未被修改的 Cisco 和 Juniper 路由器上。
4. 评估：展示了欺骗控制器在路由器上快速生成小的拓扑和路径。

## 2. 灵活的 Fibbing

DAG: Directed Acyclic Graph 有向无环图

Fibbing 工作流有以下 4 个阶段，这 4 个阶段基于两个输入：

1. 一个希望做的事情（转发）的图，每个目标都有一个 DAG
2. 一个 IGP 拓扑。（interior Gateway Protocol, 内部网关协议）

4 个阶段如图：

1. 转发 DAG 的提供可以以直接的方式，也可以以不直接的方式，即以高级的方式，用一种简单的基于路径的语言。若采用后一种方式，在编辑阶段开始时会将转发需求编辑为具体的转发 DAGs。
2. 随后，在拓扑扩展阶段计算出扩展的拓扑去实现这些 DAGs。
3. 计算扩展的拓扑很快，产生的结果可能会很大。于是在拓扑优化阶段（在不影响转发路径的情况下）削减拓扑。
4. 注入及监视阶段将虚假信息转化为谎言由控制器注入网络。

这部分，我们呈现了高级的语言以及编辑的过程，并展示了 Fibbing 可以表达宽泛的转发行为。

### 2.1 Fibbing 高级语言

Fibbing 语言提供了简洁易懂的操作符，用来具体描述他们的转发需求。一个 Fibbing 转发策略就是一组需求，可以被自然地使用规则的路径表达式表达。每个

requirement: 表达了传说应该如何向目的地进行的需求，或者表达了当 IGP 拓扑发生变化时传输应该如何进行的后备需求。

每个路径需求都由路径需求的与、或递归定义。

操作者可以用 n 个需求联合起来表达一个均衡分路的需求；同样也可以用这 n 个需求中的单独的一个来确保一个特定的传输目标。

路径需求由许多节点需求的队列组成。

一个节点需求可以是针对某个特定点的路由，也可以是对任意点队列都适用。

与路径需求一样，点需求也可以用逻辑与、逻辑或结合起来定义。

当没有点需求时，将会使用最初的 IGP 路径。这时，操作者不用表述所有未修改的路径，只需偏离 IGP 最短路径。（这句原文：This way, operators do not have to express all the unmodified paths, only deviations from the IGP shortest path.）

例子：

E 到 D1，有两个分路平衡的路径；

A 到 D2，要经过 B 或者 C；

当(F, H)断了的时候，F 到 D3 要经过 G。

per-destination?

通过发现方便的网络路径（如果有的话），Fibbing 政策被编辑为针对每个目标的 DAGs。

编辑工作分为两个阶段:

1. 编辑器将任何的非特定节点的路径需求进行扩展。

这一步从计算上讲花费可能会很昂贵,一般来说,一个有大量节点的网络可能有指数级的路径数量。尽管这看起来很难实现,特别是对于想设计最好的训练网络来说。我们限制了可以从单一路径中扩展的路径的数量,我们只会在当前路径中没有找到解决办法时才扩展。一旦所有的需求全部扩展了,编译器会通过目的地来讲他们分组,并且计算每个需求的 DNF。

2. 编辑器通过这些需求的目标将其归类并计算其析取范式(Disjunctive Normal Form(DNF),用“或”“V”把几个公式连接构成的公式)。为了最终生成转发 DAG,编辑器分别迭代这些路径需求,检查生成的图是否无环。

## 2.2 Fibbing Expressiveness

除了根据一个现成的路径去传输数据,我们现在展示 Fibbing 还可以: 1. 在多条路径上平衡分路。2. 提供备用路径。

Fibbing 可以以任意路径转发任意的流。

Fibbing 在多条路径上平衡传输,以最大化生产量,减少响应时间或提升可靠性。

图 a, IGP 协议计算的最短路径为 (E, F, G, H), 所以蓝色黄色都走这段, (F, G) 的承载量是 1, 但  $0.75 + 0.5 = 1.25 > 1$ , 故拥塞了。

(有一种减弱拥塞的方法,是将黄色的流分两路, (B, C) 和 (F, G), 传统的链路状态路由可以将从 E 过来的数据平衡分路(如将 (F, G) 的权值改为 15)。但是这样会使从 S2 和 S3 过来的数据都分散到两条路,创造新的拥塞。更一般的来讲,传统链路状态路由不可能令去往 D2 和 D3 的数据流走两条不同的路。)

Fibbing 将拓扑改成图 b 的样子,加入虚结点,由于  $1 + 6 < 3 + 1 + 3 + x$  (x 为 h 到黄色目标节点的权值),故黄色的流走上面,这样就解除了拥塞。

为什么蓝色的流不走上面? 要走一起走啊。

这个例子中, Fibbing 可以在任何一条链路满负荷时最大化网络效率。

Fibbing 可以给任何流提供备用路径,以防止拥塞或由于链路或节点失效而上升的延迟。

图 a 中 (E, F) 坏了,故 S1 到 D1, S2 到 D2 的数据都经 (A, B) 于是拥塞 ( $0.5 + 0.5 > 0.75$ ), 我们要把 D1 和 D2 的流分为两个保持不解体的路径,但仅在 (A, D, E, F) 有故障时才这样做。这用链路状态路由无法实现,而且这需要用到 MPLS (Multi-Protocol Label Switching 多协议标签交换) 之上的意义重大的控制层。

相对地, Fibbing 可以很容易地实现,如图 b, 备用节点用我们的语言描述为 (A, B, \*, D1) AND (A, G, \*, D1) as backup of ((A, D) or (D, E) or (E, F))。加入虚结点后,虚结点的那条路总权值为 4,比 3 大,比 9 小。

虽然在这个例子中成功了,但是 Fibbing 并不能满足所有对备用路径的需求。