



Opendaylight 学习文档

qq 群#北邮-天依

目录

1. 概述	3
1.1 Opendaylight 简介	3
1.2 本文档组织结构	7
2. 感受 Opendaylight	7
2.1 环境搭建	7
2.2 获取代码	9
2.3 安装 mininet	11
2.4 controller 使用及功能介绍	11
2.5 Openflowplugin 功能及使用方法	14
2.6 Hydrogen	16
3 Maven 和 OSGI 基础	16
3.1 Maven	16
3.2 OSGI	20
4 使用 IDE	30
4.1 使用 Eclipse	31
4.1.1 导入 controller 项目	31
4.2 使用 IntelliJ idea	38
5 Controller 代码分析	39
5.1 代码目录	40
5.2 收发包过程简介 (packet service)	41
6 Opendaylight 重要技术及文档	44

1. 概述

1.1 Opendaylight 简介

Opendaylight ([Opendaylight 官网](#)) 是 Linux 基金会的一个合作项目。目前，包括十二个项目，每一个项目都有自己的代码库([Opendaylight 项目列表](#))。这些项目中与 openflow 相关的项目的有 controller、openflowjava 和 openflowplugin，目前，controller 仅支持 openflow 1.0， openflowplugin 是一个单独的项目，将来它的 core 部分要集成到 controller 中，使 controller 支持 openflow 1.3 及以上的版本。Opendaylight 的厂商成员分为铂金成员，金牌成员和银牌成员。



图 1 Opendaylight 阵营

Opendaylight controller 使用 java 编写，运行在 JVM 上，理论上来说可以部署到任何支持 JAVA 的平台上，但是其[官网文档](#)推荐的最佳运行环境为最新的 Linux (Ubuntu 12.04+) 及 JVM 1.7+。OpenDaylight 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），同时南向支持多种包括 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。

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

自 **opendaylight** 官方文档):

- 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服务的安装、删除和更新。
- 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。**MD-SAL (Model Driven Service Abstraction Layer)** 是 **opendaylight** 的一个主要 feature。
- 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**)。

Opendaylight controller 的架构框架:

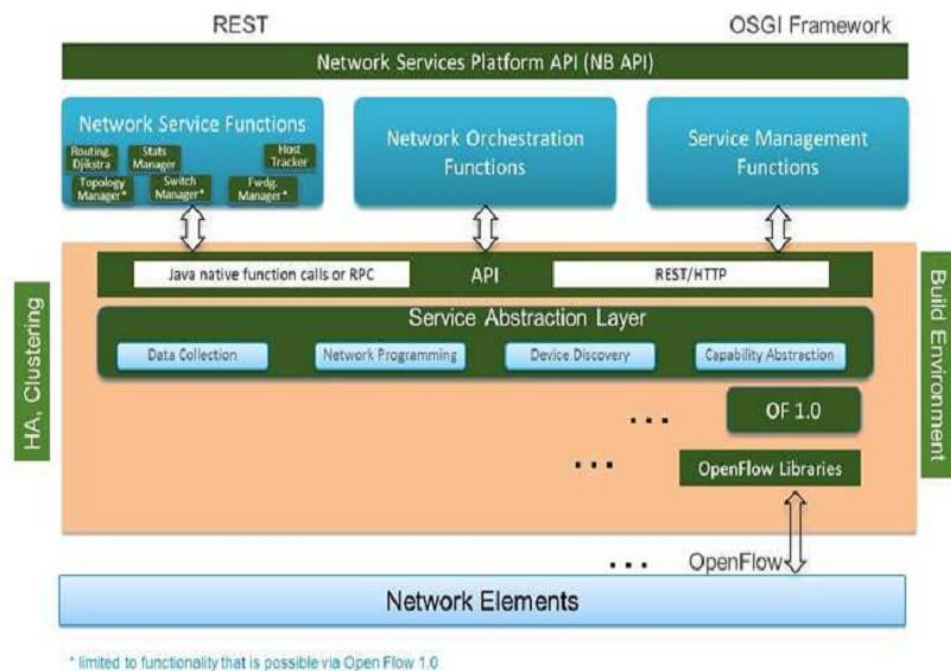


图 2 OpenDaylight controller 架构

如图 2 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。 控制器自带了 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。

OpenDaylight 的 openflowplugin 和 openflowjava 项目的目标是支持 openflow 1.3 及以上的协议，由于 openflow 协议设计时并没有考虑后向兼容性（例如 openflow 1.0 和 openflow 1.3 的连接建立发生了很大的变化），Openflow 1.3 plugin 采用和 Openflow 1.0 完全不同的设计。集成到 controller 的 openflow 1.0 plugin 是由 openflow plugin 和 openflowj 两部分实现，其中 openflowj 是 openflow 1.0 消息的静态库，它被 openflow 1.0 plugin 依赖。Openflow 1.3 plugin 中的 openflowjava 没有了 Openflow 1.3 的消息库（使用了 YANG，支持 Openflow 1.3 以上的版本？？？）还包括了连接建立和协议编解码部分，OpenflowPlugin 依赖 Openflowjava 实现消息处理等功能。Openflowjava 和 OpenflowPlugin 设计框图如下：

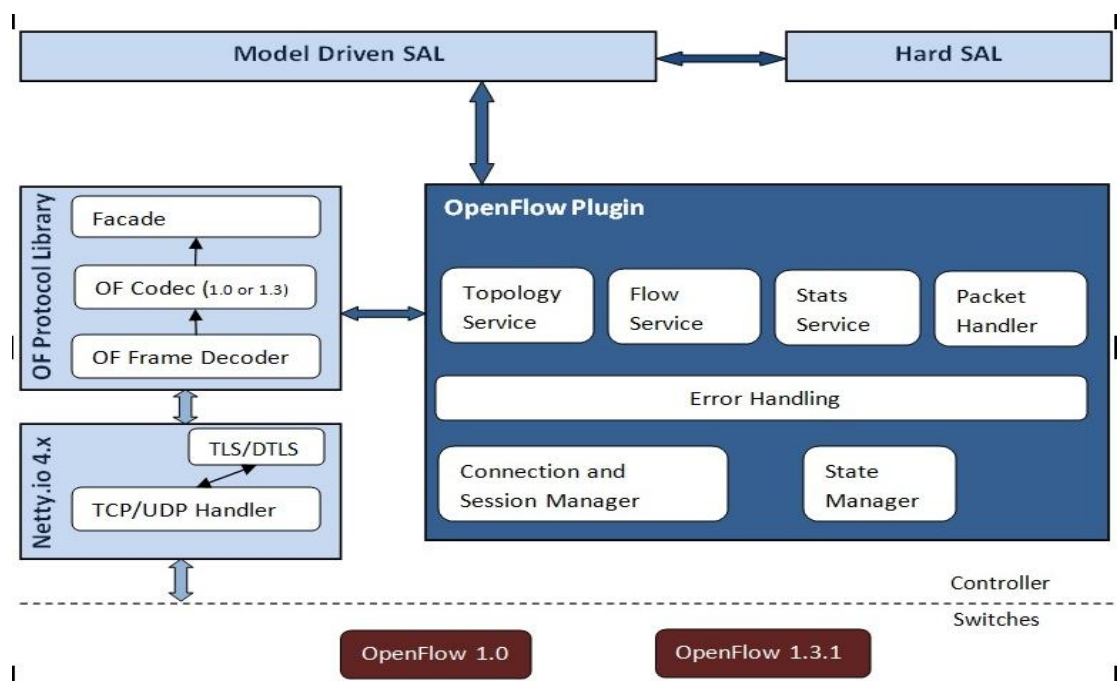


图 3 Openflowplugin 和 Openflowjava 设计框图

如图 3 所示，左侧为 openflowjava 的功能框图，右侧为 Openflowplugin 的框图。

1.2 本文档组织结构

本文档关注与 openflow 相关的 controller, openflowjava 和 openflowplugin 三个项目，重点分析 controller。本文本着由浅入深的原则，首先介绍如何获取相关项目代码，如何编译，运行；其次介绍 OSGI 框架和 maven 的一些基础概念，与此同时，分析 opendaylight controller 的代码结构，接着介绍如何将 controller 导入到 Eclipse 和 IntelliJ idea 中（之所以把这步放在 OSGI 和 maven 介绍之后，是因为导入工程到 IDE 的过程需要对 OSGI 和 maven 有一些了解），分析下这两种 IDE 在编译调试中各自的优缺点；然后是代码的分析；最后列举 opendaylight 的重要技术及文档。

Opendaylight 有很多重要的技术需要了解,每种技术都需要花一段时间研究,例如 OSGI 框架的使用, OSGI 服务的注册和调度, maven 工程中 bundle 的编写方法, sal, YANG, config, 等。由于时间精力的限制, 本文在最后简单分析了 controller 收发包过程中 openflow plugin, SAL 和上层应用间的关系。文章中对不太明确和有疑问的地方采用红色字体加问号来标识。

2. 感受 Opendaylight

这里的 Opendaylight 指的是 controller, openflowjava 和 openflowplugin, 它们使用不同的代码库, 但编译运行环境相同。

2.1 环境搭建

需要的环境如下:

- 1) Ubuntu 12.04 32bit
- 2) JVM 1.7+

3) Maven 3.04+

对于第 2) 和 3) 需说明的是, 最好先配置 java 环境再安装 maven, 因为我这边的情况是我本来先安装 maven, 挺费时间的 (如果网速慢), 后来删除 jdk-6, maven 也被删掉了

Step1: 配置 java 环境

- 执行 `java -version` 查看使用的 java 版本

这个版本的 ubuntu 装的应该是 jdk 1.6

- 执行 `apt-get remove openjdk-6-jre-lib openjdk-6-jre-headless` 删除 jdk 1.6 相关的东西

注意: 执行这个命令的时候, 需要观察下 `remove` 过程, 会发现它会自动安装 java jre 1.7 的东西, 但是不要认为就不需要装 jdk 1.7 了, 因为 ubuntu 默认安装的东西是不全的, 至少我安装的过程中遇到了问题。确认是否 java jdk 是否安装全面的方法是, 查看 jdk 的安装目录是否完全

```
flight@flight-virtual-machine:~$ ls /usr/lib/jvm/java-7-openjdk-i386/lib/
```

```
ct.sym dt.jar ir.idl  jconsole.jar  jexec orb.idl          sa-jdi.jar    tools.jar
```

以上这个“tools.jar”是必不可少的, 因为 maven 编译的时候对它有依赖。如果发现 java-7-openjdk-i386 下没有 lib 文件夹, 那么继续执行下一步

- 执行 `sudo apt-get -y install openjdk-7-jdk`

注意: 依然要看一下安装过程, 如果发现似乎什么都装不上, 那么请执行

`apt-get remove openjdk-7-jre-lib openjdk-7-jre-headless` 删除 ubuntu 给你默认安

装的东西，再执行 `sudo apt-get -y install openjdk-7-jdk`，之后再按上步所述查看 `jdk` 是否安装完全，一般没有问题了。

- 修改 `java` 环境变量

`Vim /etc/profile` 在其末尾添加如下，并保存：

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-i386
```

```
export JRE_HOME=${JAVA_HOME}/jre
```

```
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
```

```
export PATH=${JAVA_HOME}/bin:${JRE_HOME}/bin:$PATH
```

完成以下修改后，需重启虚拟机或使用命令 `source /etc/profile` 使配置生效

Step2: 安装 `maven`：

- `sudo apt-get install maven`

备注：使用 `mvn -v` 可以查看当前系统中 `maven` 的版本，可以安装前和安装后看一下

- 编译时，为避免出现 “Out of memory error - java.lang.OutOfMemoryError: PermGen space:” 错误，可加入如下环境变量

`Home` 目录下执行：`vim .bashrc` 在其结尾添加如下一行：

```
export MAVEN_OPTS="-Xmx1024m -XX:MaxPermSize=256m"
```

然后 `logout ubuntu` 再 `login` 使配置生效

2.2 获取代码

`Controller`, `openflowplugin` 和 `openflowjava` 的代码都可以匿名 `git clone` 到本地。

下载 controller: git clone <https://git.opendaylight.org/gerrit/p/controller.git>

下载 openflowplugin: git clone <https://git.opendaylight.org/gerrit/p/openflowplugin.git>

下载 openflowjava: git clone <https://git.opendaylight.org/gerrit/p/openflowjava.git>

下载完毕后,会在当前目录下生成 controller、openflowplugin 和 openflowjava 三个目录。

因为 openflowplugin 依赖 openflowjava,在编译 openflowplugin 时会同时编译 openflowjava 的相关 bundle,因此 openflowjava 是不需要单独编译的,下载下它的目的是为了后续分析它的代码结构。

需说明的是,openflowplugin 目前是一个单独的项目,它包含基本的 controller 代码,所以它是可以单独运行的,只是没有 controller 那样可以直观的从浏览器中看到和使用它的功能。Openflowplugin 如何运行使用将在后面介绍。

编译运行

编译:

Controller:

- cd controller/.opendaylight/distribution/.opendaylight
- mvn clean install

Openflowplugin:

- cd.opendaylight/openflowplugin/distribution/base
- mvn clean install

为避免编译过程中 test 的编译错误,可使用 mvn clean install -DskipTests 跳

过测试

执行:

Controller:

- cd
controller/.opendaylight/distribution/.opendaylight/target/distribution.opendaylight
-OSGipackage/.opendaylight
- ./run.sh

Openflowplugin:

- cdopenflowplugin/distribution/base/target/distributions-openflowplugin-base-0.0.
1-SNAPSHOT-OSGipackage/.opendaylight
- ./run.sh

2.3 安装 mininet

我们使用 mininet 与.opendaylight controller 连接,我们使用的 mininet 的版本为 mininet-of-1.3, 既支持 openflow1.0 也支持 openflow1.3 协议, 下载地址为 [mininet-of-1.3](#)。

启动 mininet, 使用 ifconfig 查看网络配置, 配置 mininet 的网络环境使之可以与.opendaylight controller 之间的连通, 配置完成后, 从 controller 和 mininet 两边分别 ping 一下。

2.4 controller 使用及功能介绍

本节将通过.opendaylight controller 与 mininet 连接,演示 controller 的使用。
Controller 支持的是 openflow 1.0 因此我们看到 controller 与 mininet 协商的版本时

1.0 版本。

2.4.1 连接 controller 和 mininet

Step1: 启动 controller

执行 `./run.sh`，等待几分钟，打开浏览器，输入：<http://localhost:8080>，进入 opendaylight 的登陆页面，用户名和密码都是 admin

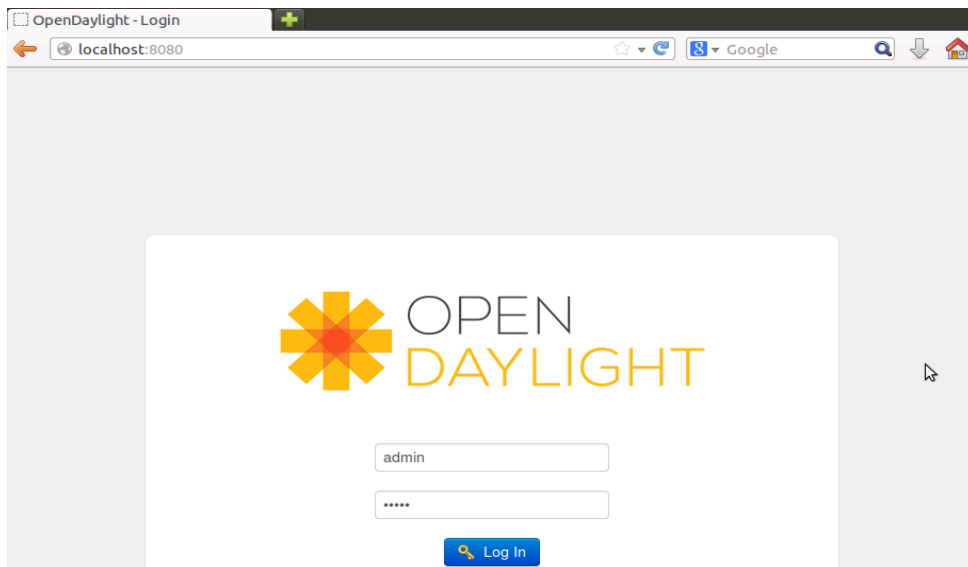


图 4 opendaylight controller 登陆页面

Step2: 启动 mininet 连接 controller

Mininet 连接 controller 的命令为 `mn`，启动 mininet，执行 `which mn`，可以看到 `mn` 已经被安装到 `/usr/local/bin` 下。使用 `mn -help` 可以查看 `mn` 命令的帮助，mininet 命令的详见 <http://mininet.org/walkthrough/>。

本例中,mininet连接controller的命令为(使用 tree 类型拓扑, remote controller):

```
sudo mn --controller=remote,ip=192.168.1.29 --topo tree,3
```

2.4.2 controller 功能介绍

Mininet 连接上 controller 后，会在首页形成拓扑图，需要注意的是拓扑图开始只会显示交换机不会显示主机，这是由链路发现协议(LLDP)来决定的，当主机发起流量时，相关主机才会在拓扑图中显示。

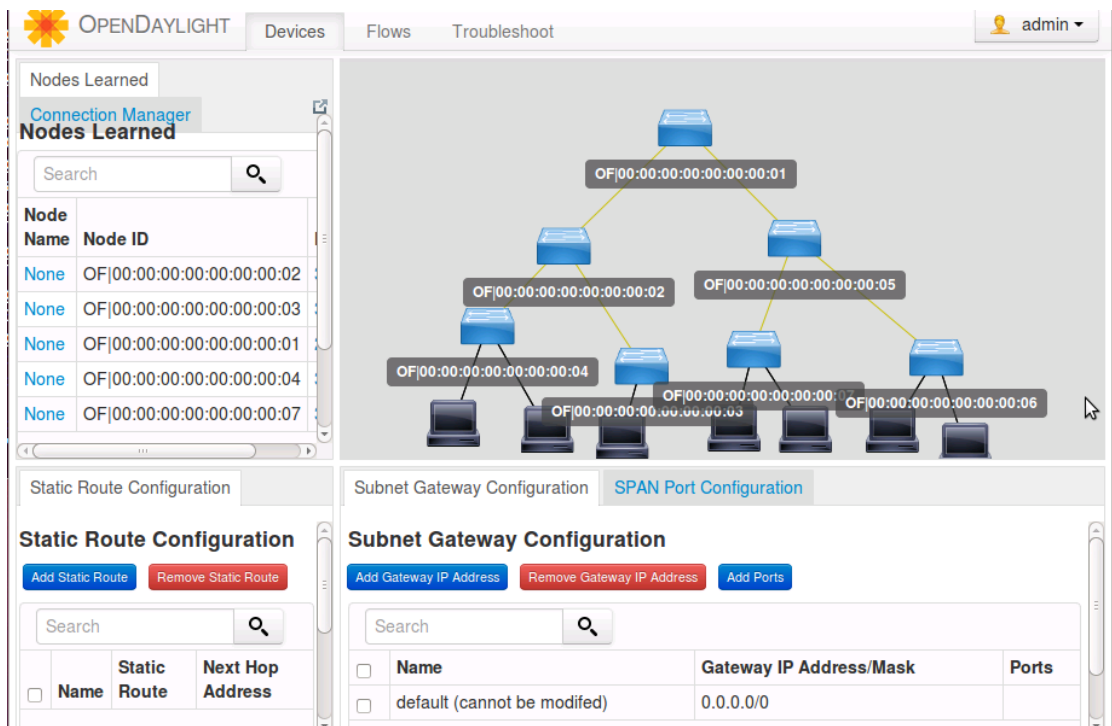


图 5 mininet 连接上 controller

如上图控制器主要包含的功能有：设备、静态流表及统计。

- **Devices** 功能，可以实现网络节点的学习（Nodes learned），连接管理（connection Manager），网络拓扑图、静态路由配置（Static Route Configuration）、子网网关配置（Sub Gateway Configu）及 SPAN Port Configuration。

其中 Nodes learned 功能，可以发现网络设备，得到对应的网络节点（网络设备，交换机）的 Id（Node ID）及对应交换机的端口信息；Connection

Manager 可以对这些网络设备进行管理, 可以实现网络设备的添加和删除; **Static Route Configuration** 可以手动添加或删除静态路由, 进行路由配置管理; **Subnet Gateway Configuration** 可以手动实现子网网关的添加和删除配置, 并可以实现只针对交换机具体端口的网关的配置, **controller** 的三层转发功能必须配置网关。

SPAN Port Configuration, 即交换机分析器的端口配置。**SPAN**, 全称为 **Switched Port Analyzer**, 直译为交换端口分析器。是一种交换机的端口镜像技术。作用主要是为了给某种网络分析器提供网络数据流, **SPAN** 并不会影响源端口的数据交换, 它只是将源端口发送或接收的数据包副本发送到监控端口。

2.5 Openflowplugin 功能及使用方法

如 2.3 所述, **openflowplugin** 也可以单独连 **Mininet**, 但不能像 **controller** 那样通过浏览器直接查看和使用其提供的功能。**Openflowplugin** 获取当前交换机的连接状态以及下发流表都需要手动发送 **http** 的 **post/get** 请求。使用的工具是 **google** 的 **chromium** 浏览器(**windows** 上的 **chrome** 浏览器)所提供的“**postman rest client**”插件。在 **ubuntu** 中通过包管理中心可以下载 **chromium Web Browser**, 如下图所示:

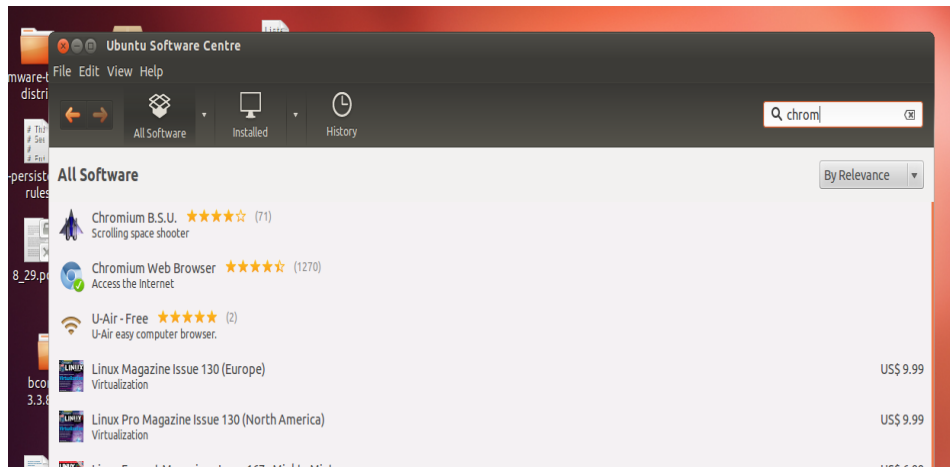


图 6 下载 chromium

下载安装完 chromium 后,使用 Tools->Extensions 进入 chromium 的扩展下载页面,搜索 postman,选择 postman rest api,点击“下载”,下载前需要注册 Google 账号。下载完成后,可在如下界面看到 postman 已经安装好。

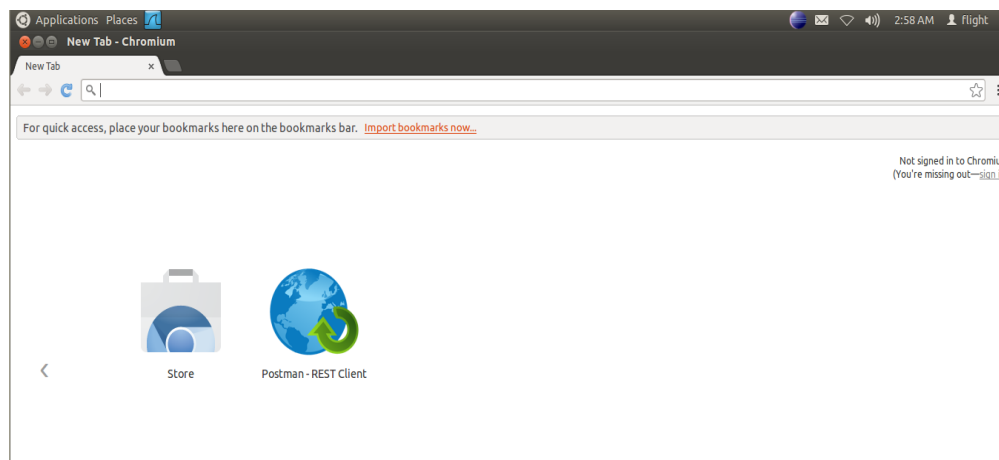


图 7 安装好 postman

安装完成后,即可按照 [User Guide#Tutorial .2F How-To](#) 中的 End to End xxx (xxx 指 Inventory、Flow、Meter、Group) 进行相关的测试了,需要说明的是:

- 下发流表/Group 表/Meter 表的 xml 例子在 test-script/目录下,以 f 开头的是有关流表的,以 g 开头的是 Group 表的,以 M 开头是 Meter 表相关的
- 自行下载 Openflowplugin 代码并编译时,还可以在 OSGI 的命令行下使用诸如 addMDFlow 之类的命令下发上述表项,详见

[OpenDaylight_OpenFlow_Plugin:Test_Provider](#), Hydrogen 并不支持这些命令,

因为 Hydrogen 没有 Test Provider bundle

- 使用 postman 时, 要注意 url 一行的最左边一定不要有空格否则会出错; 下发流表时一定要使用 Basic Auth 而非 Normal, 如下图:



图 8

2.6 Hydrogen

Hydrogen 我用的是 base 版, 具体如何下载, 群里有文档, 我就不赘述了, 需说明的是 Hydrogen 既支持 openflow 1.0 又支持 openflow1.3, 若要使用 openflow 1.0 直接执行 ./run.sh, 若使用 openflow 1.3, 使用 ./run.sh -of13.

3 Maven 和 OSGI 基础

前面介绍编译 opendaylight 时使用的是 maven, 概述中提到 opendaylight controller 使用的是 OSGI 框架, 在分析代码前, 我们需要弄清楚 Maven 和 OSGI 的一些基础知识, 否则 controller 那么多的代码将无从下手, 在介绍这些基础知识时, 我会结合 controller 项目代码, 给出自己的理解。

3.1 Maven

本部分有很多都摘自一篇 [maven](#) 的书。

Maven 是一个优秀的构建工具, 能够帮我们自动化构建过程, 从清理、编译、测试到生成报告, 再到打包和部署。Maven 是跨平台的, 这意味着无论是在 Windows 上, 还是在 Linux 或者 Mac 上, 都可以使用同样的命令。

在不了解 `maven` 时，你可以想象它完成的是像 `make` 或 `ant` 那样的功能。就像 `Make` 的 `makefile`, `Ant` 的 `build.xml` 一样, `Maven` 项目的核心是 `pom.xml`。
`POM` (`Project Object Model`, 项目对象模型) 定义了项目的基本信息, 用于描述项目如何构建, 声明项目依赖, 等等。打开 `opendaylight controller` 代码的任意一个目录, 例如 `controller/arphandler` 目录, 就可以看到 `pom.xml`, 稍后, 简单分析一下 `pom.xml`, 网上也有很多相关的文章。

使用 `Ubuntu` 的 `apt-get install` 安装 `maven` 时, `maven` 的安装目录一般在 `/usr/share/maven`, `Maven` 的安装目录被称之为 `M2_HOME`(我在试验时, 环境变量没有设置这一项, 所以不确定若不设置这一环境变量有什么后果)。以下是 `maven` 安装目录的分析:

- **Bin**: 该目录包含了 `mvn` 运行的脚本, 这些脚本用来配置 `Java` 命令, 准备好 `classpath` 和相关的 `Java` 系统属性, 然后执行 `Java` 命令。其中 `mvn` 是基于 `UNIX` 平台的 `shell` 脚本, `mvn.bat` 是基于 `Windows` 平台的 `bat` 脚本。在命令行输入任何一条 `mvn` 命令时, 实际上就是在调用这些脚本。该目录还包含了 `mvnDebug` 和 `mvnDebug.bat` 两个文件, 同样, 前者是 `UNIX` 平台的 `shell` 脚本, 后者是 `windows` 的 `bat` 脚本。那么 `mvn` 和 `mvnDebug` 有什么区别和关系呢? 打开文件我们就可以看到, 两者基本是一样的, 只是 `mvnDebug` 多了一条 `MAVEN_DEBUG_OPTS` 配置, 作用就是在运行 `Maven` 时开启 `debug`, 以便调试 `Maven` 本身。此外, 该目录还包含 `m2.conf` 文件。
- **Boot 目录**: 据说普通用户不需要关心这个目录。

- **Conf**: 该目录包含了一个非常重要的文件 `settings.xml`。直接修改该文件，就能在机器上全局地定制 **Maven** 的行为。一般情况下，我们更偏向于复制该文件至 `home` 目录下的 `.m2/` 目录下（这里 `~` 表示用户目录），然后修改该文件，在用户范围定制 **Maven** 的行为。`~/.m2` 是默认的 **maven** 本地仓库，我们可以通过加入环境变量 (`export MAVEN_OPTS="-Dmaven.repo.local=/path/to/repository"`) 来修改本地仓库的路径。`~/.m2` 的作用在下面详细介绍。关于 `setting.xml` 对于 **maven** 应该是一个很重要的文件，导入工程到 **Eclipse** 和 **Intellij idea** 两个 IDE 中时都需要将 `setting.xml` 的位置设置正确。

- **Lib**: 该目录包含了所有 **Maven** 运行时需要的 **Java** 类库。

`~/.m2/repository` 目录为 **maven** 的本地仓库，观察 `mvn clean install` 执行时，会发现有很多 `download` 的过程，`download` 下的内容即保存到了 `repository` 下的对应目录下了。从目前的经验来看，安装到这个仓库的内容包括执行 `mvn` 命令时需要的或 `pom.xml` 所依赖的各种插件。按照 **maven** 的一些资料所述，一个 **maven** 构件若想能被其他 **maven** 项目使用，则需要将此构建安装到本地仓库中，但是我还不确定这一点在 `opendaylight controller` 上是怎么表现的，因为我查看 `repository` 中没有 `controller` 各个 `bundle` 的 `jar`，也许跟使用的 **OSGI** 框架有关。

maven 的主要命令包括：`mvn clean compile`、`mvn clean test`、`mvn clean package`、`mvn clean install`。`Clean` 是清空，`compile` 是编译，`test` 是编译 `java` 的 `test` 代码，`package` 是打包，打包默认存放路径是 `target/` 目录，`install` 是将打包出的 `jar` 安装

到 maven 的本地仓库中，默认为 home 目录下的 .m2/repository。根据我们使用 mvn clean install 的经验，install 之前，会先执行 clean，compile 和 package，执行这些命令时实际上我们是在执行相应的 maven 插件。

下面以 controller/arphandler/为例来介绍下 pom.xml 的含义。

pom.xml 的开头的最重要的是 “groupId”， “artifactId” 和 “version” 三行。这三个元素定义了一个项目基本的坐标，在 Maven 的世界，任何的 jar、pom 或者 war 都是以基于这些基本的坐标进行区分的。GroupId 定义了项目属于哪个组，这个组往往和项目所在的组织或公司存在关联，譬如你在 googlecode 上建立了一个名为 myapp 的项目，那么 groupId 就应该是 com.googlecode.myapp，如果你的公司是 mycom，有一个项目为 myapp，那么 groupId 就应该是 com.mycom.myapp。ArtifactId 定义了当前 Maven 项目在组中唯一的 ID。顾名思义，version 指定了项目当前的版本随着项目的发展，version 会不断更新。

“packaging”指的是打包方式，bundle 指打包成 bundle，pom 指不打包。“build”是跟构建相关的设置。“plugin”中的 maven-bundle-plugin 指 arphandler 要通过 maven 的 maven-bundle-plugin 插件编译成 OSGI 的 bundle。“plugin”里的内容将在介绍 OSGI 时再介绍 “dependencies”即依赖关系，但我没研究过依赖的机制(如何确定依赖)。

最后要提的是 maven 的一个重要概念，archetype。一般 maven 项目的根目录下会创建 pom.xml，src/main/java 中为项目主代码，src/test/java 为执行单元测试的代码，有些工程还会有 resource 目录，以上目录都统称为项目骨架。每件一个 maven 项目都要建立那些目录，为了尽量减少重复工作，maven 提供了 mvn

archetype:generate 命令 (maven 3.0 可以直接执行此命令), 其中 “archetype” 指我们使用的是 maven-archetype-plugin 插件, “generate” 指我们的目标是产生项目骨架。执行 mvn archetype:generate 命令后, 我们可以自己选择希望用的 archetype, 输入我们要创建项目的 groupId、artifactId、version、以及包名 package 并确认, Archetype 插件将根据我们提供的信息创建项目骨架, 因此 archetype 类似于模板的概念。在 opendaylight controller 中使用的 maven archetype 为 [odl maven archetypes](#), Controller 代码中有一个对应的 archetypes 目录, 其官网对 archetype 的介绍为 “A maven archetype to create a yang model project that will generate java code from .yang files”, 我认为研究这个 archetype 关系到我们将来如何在 opendaylight 写应用, 因此很重要。

在下一节, 我们还要结合 opendaylight controller 的代码结构, 介绍 maven 的两个插件, maven-bundle-plugin 和 maven-assembly-plugin。它们与 OSGI bundle 的编译, 打包部署相关。

3.2 OSGI

OSGI 框架的实现有多种, Opendaylight 使用的是 equinox。Equinox 是 Eclipse 所使用的 OSGI 框架, Eclipse 强大的插件体系就是构建在 OSGI bundles 的基础之上。

本节我们将介绍 OSGI 框架的运行方法 (CLI 下) 以及 OSGI bundle 创建及部署。由于 OSGI 框架的运行离不开 bundle 的概念, 所以我们首先展示 bundle 是什么, 如何开发 bundle, 然后介绍 OSGI 框架的搭建和运行, 最后结合 opendaylight controller 代码介绍 bundle 在 OSGI 框架的打包和部署

3.2.1 基于 OSGI 开发的一个例子

什么是 **bundle**? 我现在的理解和表述也许有差池, 但是从网上的例子及 **opendaylight controller** 代码来看, **bundle** 就是一个基于 OSGI 框架的 **plug-in** 工程编译打包后生成的 **jar**。

Eclipse 可以创建基于 OSGI 框架的 **plug-in** 工程, 网上有很多相关的文章, 请自行学习下。在下一章, 我们将介绍如何将 **opendaylight controller** 项目导入到 Eclipse 工程中, 每一个 **bundle** 都会有一个 **plug-in** 工程的视图, 与网上的 **helloworld** 视图基本类似 (除了多了一个 **pom.xml**)。

每个 **plug-in** 工程都有一个 **activator.java**, 这是 **bundle** 启动时首先调用的程序入口, 相当于 **Java** 模块中的 **main** 函数。不同的是, **main** 需要通过命令行调用, 而 OSGI 的 **Activator** 是被动的接受 OSGI 框架的调用, 收到消息后才开始启动。最佳实践: 不要在 **Activator** 中写太多的启动代码, 否则会影响 **bundle** 启动速度, 相关的服务启动可以放到服务的监听器中。

Plug-in 工程中的另一个重要文件是 **MANIFEST.MF**。**MANIFEST.MF** 用来存储插件的配置信息, 包括这个插件的依赖, **import-package**, **export-package** 等, 具体在下面介绍。

3.2.2 Opendaylight 的 bundle 开发

opendaylight 是基于 OSGI 的 **Maven** 工程。对于每一个 **odl bundle**, 我的理解包括:

1. **opendaylight controller** 有自己的 **maven archetype** 来生成 **bundle** 代码框架。

2. pom.xml 配置 maven-bundle-plugin 插件可以实现 maven 根据用户配置的 instructions 来自动生成 plug-in 中非常重要的 MANIFEST.MF 文件的内容
3. 配置 pom.xml 能指定 maven 将一个工程打包成 bundle, 指定<packaging> 标签值为 bundle, maven-assembly-plugin 完成 bundle 的打包和部署 (3.2.3 节详细介绍)。

本节通过例子简单介绍下使用 maven-bundle-plugin 插件自动生成 MANIFEST.MF 时 pom.xml 的编写规则。

controller/opendaylight/arphandler 下的 pom.xml

下面是 controller/opendaylight/arphandler 的 pom.xml 中<plugin>标签的内容,

...

<plugin>

<groupId>org.apache.felix</groupId>

<artifactId>maven-bundle-plugin</artifactId>

<version>\${bundle.plugin.version}</version>

<extensions>>true</extensions>

<configuration>

<instructions>

<Import-Package>

org.opendaylight.controller.sal.packet.address,

org.opendaylight.controller.connectionmanager,

org.opendaylight.controller.sal.connection,

org.opendaylight.controller.sal.core,
org.opendaylight.controller.sal.utils,
org.opendaylight.controller.sal.packet,
org.opendaylight.controller.sal.routing,
org.opendaylight.controller.switchmanager,
org.opendaylight.controller.topologymanager,
org.opendaylight.controller.clustering.services,
org.opendaylight.controller.hosttracker,
org.opendaylight.controller.hosttracker.hostAware,
org.apache.felix.dm,
org.OSGI.service.component,
org.slf4j

</Import-Package>

<Export-Package>

org.opendaylight.controller.arphandler

</Export-Package>

<Bundle-Activator>

org.opendaylight.controller.arphandler.internal.Activator

</Bundle-Activator>

</instructions>

<manifestLocation>\${project.basedir}/META-INF</manifestLocation>

</configuration>

</plugin>

其中，比较重要的几个标签有 <Import-Package>、<Export-Package>、<Bundle-Activator>以及<manifestLocation>，这几个属性都是plug-in工程里非常重要的文件 MANIFEST.MF 的属性，使用 maven-bundle-plugin 插件就不需要手工设置 MANIFEST.MF，你只需在pom.xml里指定相应的值，maven-bundle-plugin将会为做剩下的事情。下面简单介绍下，上述几个属性的含义：

- <Export-Package>是导出包的列表。这些包被复制到打成的结果 bundle JAR 中。导出包可以使用通配符*来匹配一些类包例如 org.apache.felix.*。这表示 org.apache.felix 下面所有的子包。此外，排除导入包可以是!*。例如：org.foo.*,!org.foo.impl 第二个模式中的种写法不起作用。因为 org.foo 包已经被第一个模式所选择了。正确的写法：!org.foo.impl,org.foo.*。从 org.foo 包中排除 org.foo.impl 包。包模式可以包括以下标准的 OSGI R4 的语法，指令和属性，这将被复制到适当的清单中。

- <Private-Package>指令和<Export-Package>指令是相类似的。除了这些包不会被导出。如果一个包在<Private-Package>和<Export-Package>都写了。<Export-Package>是优先的。

- <Import-Package>指令是导入 bundle 所需要的软件包列表。默认值是*，这意味着导入所有的包。然而，在某些情况下，会有不必要的包被导入。例如你想导入所有的包，但是要排除 org.foo.impl。需要这样写：“!org.foo.impl,*”。

Opendaylight controller 代码中<Import-Package>为*的很少，可能是因为项

目太大吧。关于 maven-bundle-plugin 的更多文档见：

<http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>; (官网)

<http://wso2.com/library/tutorials/develop-OSGI-bundles-using-maven-bundle-plugin/>;

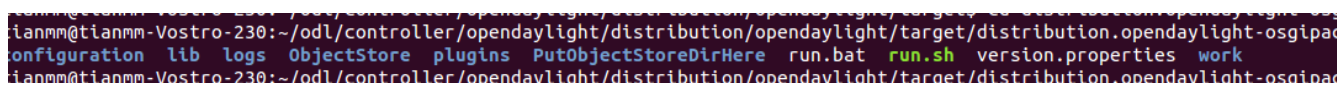
<http://eureka.ykyuen.info/2010/03/12/maven-create-a-OSGI-bundle-using-maven-bundle-plugin/>;

<http://marsvaadin.iteye.com/blog/1460281>。

综上，目前我认为 opendaylight 开发一个新的 bundle 必须要了解的包括 opendaylight 的 maven archetype 和 maven-bundle-plugin 的用法，maven pom.xml 的编写方法。

3.2.3 opendaylight bundle 的打包和部署

我们先看一下 opendaylight 编译完目录是如何组织的，target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGIpackage/opendaylight/即为 bundles 发布完成后的目录，如下：



```
tianmm@tianmm-Vostro-230:~/odl/controller/opendaylight/distribution/opendaylight/target/distribution.opendaylight-osgipad
configuration lib logs ObjectStore plugins PutObjectStoreDirHere run.bat run.sh version.properties work
tianmm@tianmm-Vostro-230:~/odl/controller/opendaylight/distribution/opendaylight/target/distribution.opendaylight-osgipad
```

图 9 target 目录

其中，

Configuration/: 存放 OSGI 框架配置的目录

Lib/: 存放 OSGI 相关 jar 文件的目录

Plugin/: Opendaylight 所有 bundle 的 jar 文件存放的位置

Run.bat: windows 下的启动脚本

Run.sh: linux 下的启动脚本

Work/: 未研究，run.sh 中 equinox 的启动参数中使用了此目录

那么，这些目录都是如何生成的，即 opendaylight controller 的 bundle 是如何发布的呢？答案是 maven-assembly-plugin 帮我们实现的。观察下 controller/opendaylight/distribution/opendaylight 目录：

```
flight@flight-virtual-machine:~/code_controller/opendaylight/controller/opendaylight/distribution/opendaylight$ ls
distribution.opendaylight.iml  opendaylight-assembly-noclean.launch  opendaylight-osgi-launcher-local.launch  runsanity.bat
opendaylight-application.launch  opendaylight-assembly-skiput.launch  opendaylight-sonar-fast.launch  runsanity.sh
opendaylight-assembly-fast.launch  opendaylight-assembly-sonar.launch  opendaylight-sonar.launch  src
opendaylight-assembly-full.launch  opendaylight-local.target  opendaylight.target  target
opendaylight-assembly.launch  opendaylight-osgi-launcher.launch  pom.xml
```

图 10 distribution 目录

观察此目录下的 pom.xml 中关于 maven-assembly-plugin 的配置，还是在 <plugin>标签下，

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<version>2.3</version>

<executions>

<execution>

<id>distro-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

```

<configuration>

  <descriptors>

    <descriptor>src/assemble/bin.xml</descriptor>

  </descriptors>

  <finalName>${project.artifactId}</finalName>

</configuration>

</execution>

</executions>

</plugin>

```

比较重要的是<descriptor>src/assemble/bin.xml</descriptor>，说明此插件的配置信息是当前目录的 src/assemble/bin.xml。继续查看 bin.xml

```

<outputDirectory>opendaylight/plugins</outputDirectory>

<excludes>

  <exclude>equinoxSDK381:org.eclipse.OSGI</exclude>

  <exclude>equinoxSDK381:org.eclipse.equinox.console</exclude>

  ...

</excludes>

```

这一段表明，除<excludes>里的 jar 文件外，其他的 jar 文件都保存到 target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGIpackage/下 opendaylight/plugins 目录下。

```

<outputDirectory>opendaylight/lib</outputDirectory>

<includes>

```

```
<include>equinoxSDK381:org.eclipse.OSGI</include>

<include>equinoxSDK381:org.eclipse.equinox.console</include>

<include>equinoxSDK381:org.eclipse.equinox.launcher</include>

...

</includes>
```

这一段表明，<includes>里的 OSGI 相关的 jar 文件都保存到 target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGIpackage/下 opendaylight/lib 目录下。

```
<fileSet>

  <directory>

    src/main/resources/

  </directory>

  <excludes>

    <exclude>version.properties</exclude>

  </excludes>

  <outputDirectory>

    opendaylight/

  </outputDirectory>

</fileSet>
```

这段表明，当前目录下的 src/main/resoures 里的文件输出到 target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGIpackage/下 opendaylight/目录下，查看下 src/main/resources 目录，原来 run.bat，run.sh，

version.properties 和 configuration 这几个文件在这里。(没搞明白那个<exclude>?, 编译完成后的 target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGipackage/下 opendaylight/下也有 version.properties) 。

有关 maven-assembly-plugin 的详细文档，见：

<http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html>;

<http://maven.apache.org/plugins/maven-assembly-plugin/usage.html>。

至此，OSGI 和 opendaylight controller 的代码框架基本介绍完了。

3.2.4 OSGI 框架的搭建和运行

OSGI 框架的启动方式及 equinox 的一些理论，网上有很多相关的文档，请自行查看。本节仅介绍 opendaylight 相关的。

我们执行./run.sh 启动 opendaylight 时，就是启动了 OSGI 框架。Run.sh 中启动 OSGI 框架的命令是：

```
$JAVA_HOME/bin/java  ${extraJVMOpts} \  
    -Djava.io.tmpdir=${datadir}/work/tmp \  
    -DOSGI.install.area=${basedir} \  
    -DOSGI.configuration.area=${datadir}/configuration \  
    -DOSGI.frameworkClassPath=${FWCLASSPATH} \  
  
-DOSGI.framework=file:${basedir}/lib/org.eclipse.OSGI-3.8.1.v20120830-144521.ja
```

r \

```
-Djava.awt.headless=true \
-classpath ${CLASSPATH} \
org.eclipse.equinox.launcher.Main \
-console \
-consoleLog
```

“\$JAVA_HOME/bin/java” 表示使用 java 的路径

“-DOSGI.configuration.area=\${datadir}/configuration \” 表示 equinox 启动配置 config.ini 的存放位置，使用 config.ini 可以配置框架启动时，启动的 bundles 及其启动级别，例如

target/distributions-openflowplugin-base-0.0.1-SNAPSHOT-OSGipackage/opendaylight/configuration/config.ini 开头：

```
OSGI.bundles=\
reference\;file\;../lib/org.apache.felix.fileinstall-3.1.6.jar@1:start,\
```

"@1"指明该组件的启动级别，":"后的"start"标明该组件在加载后启动

在终端里执行./run.sh，启动了 equinox 框架后可以进入 OSGI 控制台，OSGI 台提供了丰富的命令用来查看当前启动的 bundle 及相关信息等。

4 使用 IDE

由于 java 代码目录的结构特点，分析 opendaylight controller 代码必须使用一个 JAVA IDE。本章我们将介绍如何将 opendaylight controller 代码导入

到 Eclipse 和 IntelliJ idea 中, 并介绍如何使用它们编译, 运行, 调试 controller 项目。

4.1 使用 Eclipse

4.1.1 导入 controller 项目

Step1: 下载 Eclipse 的 java developer 版或 j2ee develop 版, 下载地址为 [eclipse download](#), 下载文件如 eclipse-java-kepler-SR1-linux-gtk.tar.gz 或 eclipse-jee-kepler-SR1-linux-gtk.tar.gz。

Step2: 在 ubuntu 中安装 eclipse。

.tar.gz 的安装包解压即可使用, 推荐使用命令:

```
sudo tar -xzf eclipse-java-kepler-SR1-linux-gtk.tar.gz -C /usr/lib
```

sudo 是为了获得 /usr/lib 的写权限, 执行完成后, eclipse 被解压到 /usr/lib/eclipse 目录。为了方便使用, 建议在 /usr/share/applications 目录下, 新建 eclipse.desktop 文件, 添加如下内容:

[Desktop Entry]

Name=Eclipse

Comment=Eclipse SDK

Encoding=UTF-8

Exec=/usr/lib /eclipse/eclipse

Icon=/usr/lib/eclipse/icon.xpm

Terminal=false

Type=Application

Categories=Application; Development;

注意：“Exec”和“Icon”的设置要与 eclipse 的安装目录相符。

这样,以后就可以从 Application/Programming/目录下找到 eclipse 的图标了(我使用的是 Gnome, 使用 Ubuntu Utility 的可能不太一样)。

Step3: 按照 [OpenDaylight Controller:Eclipse CLI Setup](#) 中的“Starting From Scratch”, 给 Eclipse 安装 m2e 和 Xtend 插件。需注意的是, 安装 m2e 时, 建议的 1.2.0 版本看不到, 只能看到最新的 1.4.0 版本, 安装此版本即可。

Step4: 输入你想设定的 workspace 名, 启动 Eclipse。点击“Window->Preferences->Maven->User Setting”确认下 maven 的设置, 如下图:

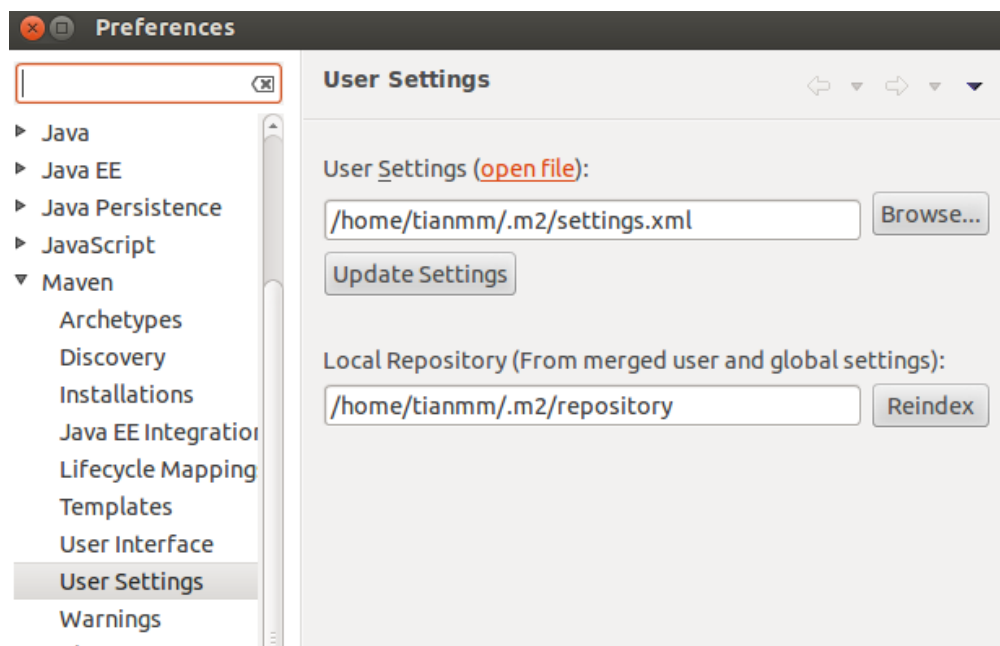


图 11 maven 设置

注意, 如上的默认设置需要你自已将 maven 安装目录的 settings.xml 拷贝到 .m2 目录下。将来当你导入的工程有很多 maven 的错误时, 可以点击“Update

Settings”来更新项目，有时能够解决那些错误。

Step5: 点击“file->import”选择“Maven->Existing Maven Project”选择要导入的项目，找到 controller/目录，这里是根 pom.xml 所在的位置，如下图：

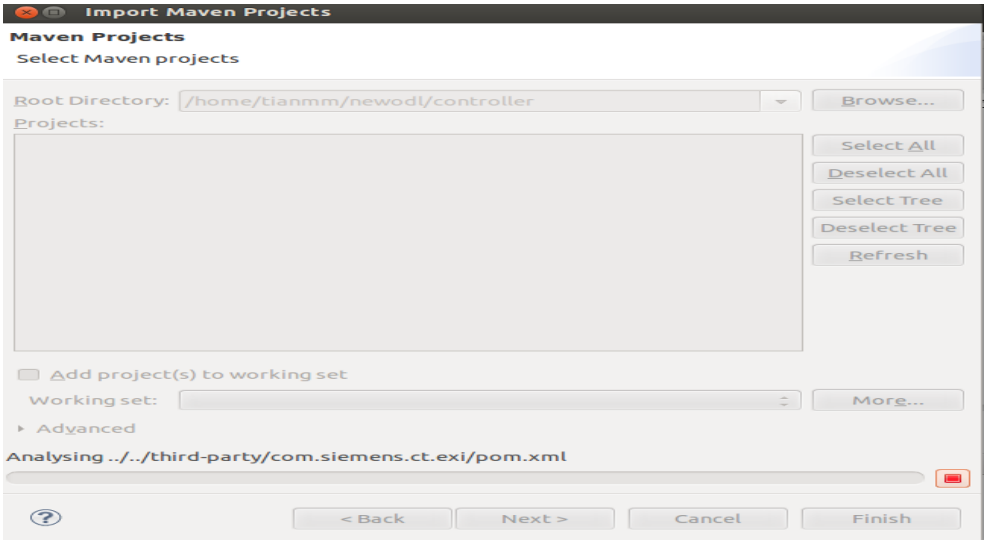


图 12

Step6: “Next”到如下页面时，会看到 yang-test 的两个错误，如下图：

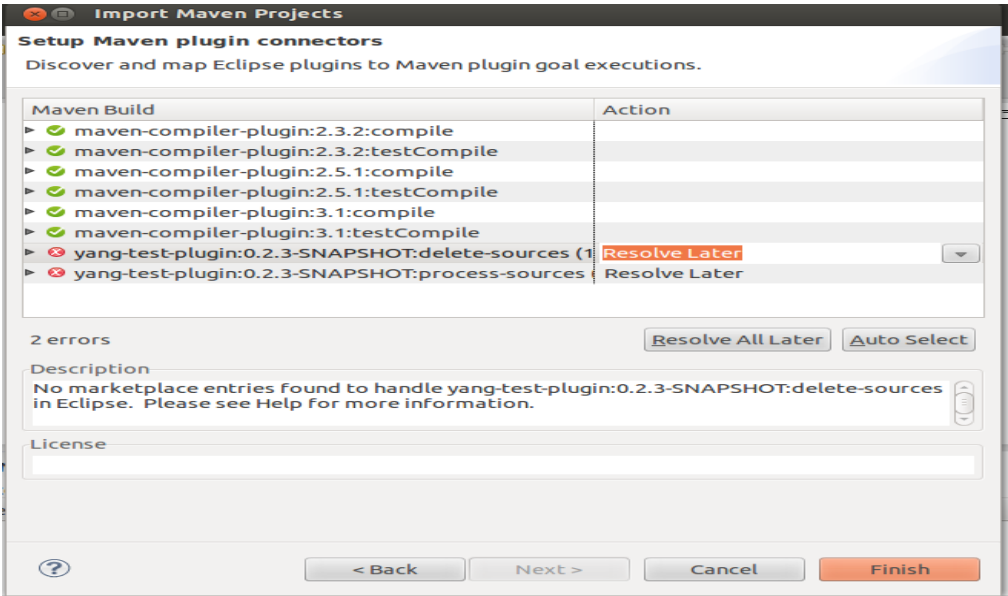


图 13

这里需要注意的是，除了“Resolve Later”，还提供了“Do Not Execute (add

to pom)” 和 “Do Not Execute (add to parent)”，我一般会选择 Do Not Execute (add to pom)”，然后 finish 项目导入完成了，（这块我没有研究）如下图：

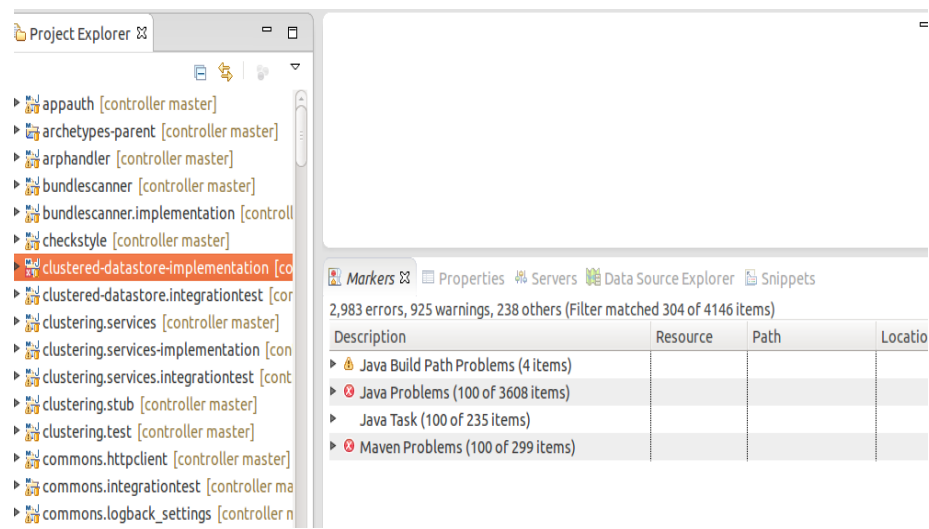


图 14 工程视图

如上，你可以看到很多错误，但是不用紧张，事实证明上说错误并不影响成功编译和运行工程，如果你选择 IntelliJ idea 做你的 IDE，就不会出现那些问题，因此我们可以暂时忽略这些问题。[OpenDaylight Controller:Eclipse CLI Setup](#) 对那些莫名其妙的错误也提出了一些解决方法，你可以尝试一下。我偶然能解决那些问题,但我没有可重用的方法,那些错误可能与环境的设置有关(Eclipse 和 Maven 插件的版本兼容等)。

4.1.2 编译，运行和调试

编译 controller 工程：

Project Explorer 视图中，进入 distribution.opendaylight 目录，可以看到一堆 .launch 文件，选择.opendaylight-assembleit.launch，右键 run as 选择.opendaylight-assembleit.launch，即可以编译 controller。

修改编译选项：

选择 run->run configurations->Maven Build->opendaylight-assembly 如下图，可以修改编译的选项。其中 main 中的 Goals 代表执行的编译命令，默认为 clean install；JRE 是你使用的 JAVA 环境，在 VM arguments 栏，可以设置你想要的 maven 选项，例如我想在编译时跳过测试那么，将添加 “-DskipTests=true”。

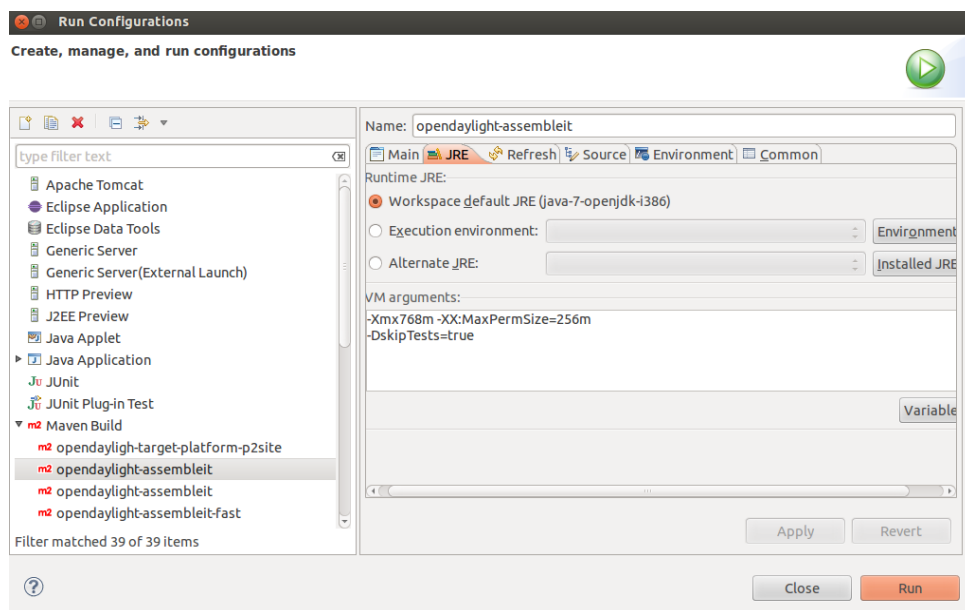


图 15

运行 controller:

Project Explorer 视图中，进入 distribution.opendaylight 目录，可以看到一堆 .launch 文件，选择 opendaylight-application.launch，右键 run as 选择 opendaylight-application.launch，即可以运行 controller。

修改运行选项：

选择 run->run configurations->Java Application->opendaylight-application 如下图，可以修改运行的选项。这个选项页的东西如何修改我还不太确定，基本上没

怎么配置过,但是我觉得默认的配置有问题,我想我目前环境的不稳定很可能跟这些选项有关,例如 JRE 和 Classpath 的默认值一个是 JavaSE1.7,一个是 Java1.6。我修改了这些参数后,运行就有问题,这个还需要研究下。

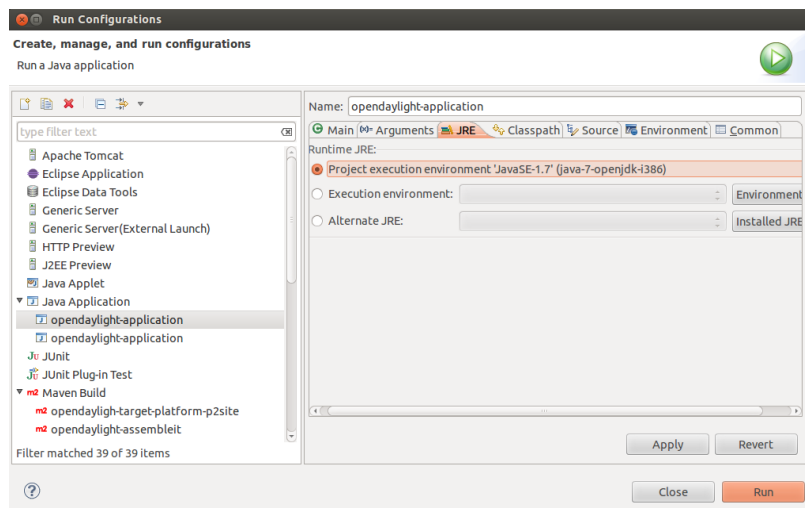


图 16

4.1.3 已知的问题

使用 Eclipse 可以比较方便的编译和调试,但是它的编译方式和在 CLI 里不同,目前还不确定是什么原因。现在总结已知的问题如下:

截止目前(2014/1/28)的 controller 代码在 Eclipse 第一次编译时,会报“ config-subsystem ” Failure 。 Console 会提示 Controller/opendaylight/config/yang-test/pom.xml 的某些行有 tab。失败原因是插件 maven-checkstyle-plugin 是 maven 用来静态检查代码的工具,它有一个配置文件来规定,哪些是非法的,这些非法检查会导致 build 失败。Opendaylight 进行检查的配置文件是 commons/checkstyle/src/main/resources/controller/checkstyle.xml。为了避免这个的编译错误,请修改: config/yang-test/pom.xml 去掉其中的 tab。

另外由于此插件不允许文件中有空格，因此若我们在文件中加 debug 信息时，使用了 tab（为了对齐，不可避免），工程就不能编译成功了。解决办法是，修改此插件的另一个配置文件 Controller/opendaylight/commons/opendaylight/pom.xml，排除对.java 文件的检查，如下所示：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>${checkstyle.version}</version>
      <dependencies>
        <dependency>
          <groupId>org.opendaylight.controller</groupId>
          <artifactId>checkstyle</artifactId>
          <version>0.0.2-SNAPSHOT</version>
        </dependency>
      </dependencies>
      <executions>
        <execution>
          <phase>process-sources</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <failsOnError>true</failsOnError>
        <configLocation>controller/checkstyle.xml</configLocation>
        <consoleOutput>true</consoleOutput>
        <includeTestSourceDirectory>true</includeTestSourceDirectory>
        <sourceDirectory>${project.basedir}</sourceDirectory>
        <!--<includes>**/*.java,**/*.xml,**/*.ini,**/*.sh,**/*.bat</includes>tianmm debug-->
        <includes>**/*.xml,**/*.ini,**/*.sh,**/*.bat</includes>
        <excludes>**\target\,**\bin\,**\target-ide\</excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

图 17

第二个问题，添加打印信息时，要使用 info 级别，debug 级别打印不出来。

第三个问题，尽量用新代码，controller 的代码再一直更新，我曾经遇到一个问题想了很长时间也不知道为什么，换了个新版本，发现新版本有对应的更新，而且新版本可以通过一些方法来规避问题。

还有一些问题是我现在也没有解决掉的，因为原因不详，在这就不介绍了。

4.2 使用 IntelliJ idea

Opendaylight wiki 上介绍了如何导入工程到 IntelliJ idea 中，文档位置[导入工程到 IntelliJ idea](#)。IntelliJ idea 内置了 maven 等，不需要单独安装，正因如此它不存在版本冲突问题，工程导入 IntelliJ idea 中不会出现像在 Eclipse 中那样的问题。

使用 IntelliJ idea 需要说明的几点是：

- 下载 idea 的 community 版本，community 是免费的
- 安装 idea 可以仿照 Eclipse 的安装方法，将 idea.sh 的路径加入 PATH 变量，这样就能在终端输入 idea.sh 中快捷的启动 idea
- 导入工程时，选择“External”中的 Maven 工程
- 编译工程前，需先创建一个像 Eclipse 里那样的 config，如下

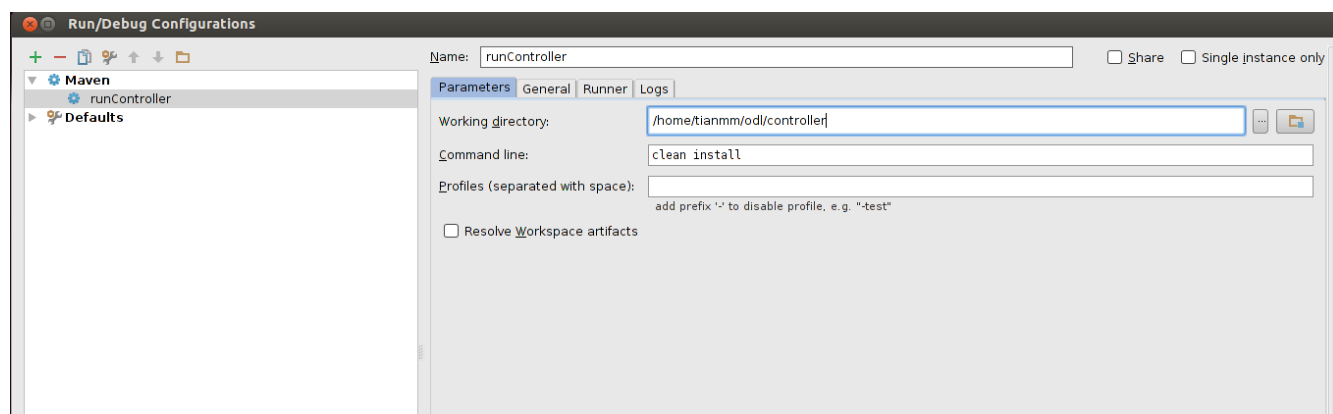


图 18

点击图上的+，可以新建 configuration。如图 18，“parameters”用来设定工程的位置（根 pom.xml 所在的位置）及编译命令。

如图 19，“General”为 maven 的设置。

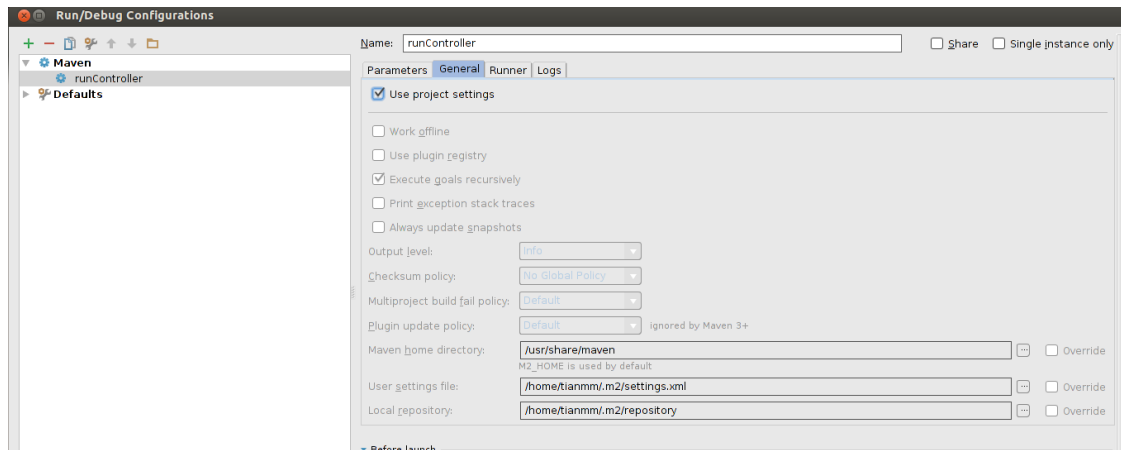


图 19

如图，配置 maven 选项，比如跳过测试

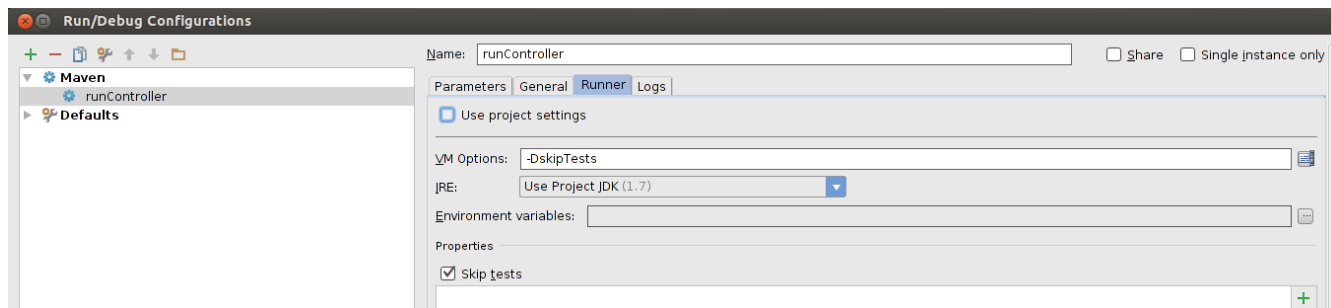


图 20

- **IntelliJ idea 只能编译不能运行！** 一般在 **idea** 中编译成功后，可以回到终端里执行 `./run.sh` 运行 **controller**。**Idea 13** 中有一个内置的 **terminal** 就跟在 **shell** 里一样，你也可以在里面执行 `run.sh`。我还不确定如何使用 **idea** 里 **debug**，比如说断点，单步等。

5 Controller 代码分析

在第三章介绍 **Maven** 和 **OSGI** 时,我们对代码结构已经进行了详细的分析。本章首先介绍下 **controller** 代码下的几个目录，然后以一个简单的 **application-sal-openflowplugin** 的模型介绍下收发包的过程。

5.1 代码目录

Controlle/根目录

Third-party/openflowj: openflow 1.0 消息库

Controller/opendaylight/目录

Arphandler: 处理 arp 包

Archetyps: opendaylight 使用的 maven archetype

Forwarding: 静态路由

Forwardingrulesmanager: openflow 流表管理

Protocol_plugins/openflow: openflow1.0 南向插件

Samples/loadbalancer: 负载均衡应用

Samples/simpleforwarding: 转发应用, 支持二三层转发

Sal: sal 相关

protocol_plugins/openflow 目录

Org.opendaylight.controller.protocol_plugin.openflow:

此目录主要是接口, 提供的服务包括: 收发包, 统计, 链路发现, 拓扑管理等。

Org.opendaylight.controller.protocol_plugin.openflow.internal:

对上面目录接口的实现, 其中 DataPacketMuxDemx.java 实现收发包。

Org.opendaylight.controller.protocol_plugin.openflow.core:

此目录主要是接口，提供的服务包括消息监听，发送/读消息，交换机添加删除

Org.opendaylight.controller.protocol_plugin.openflow.core.internal:

实现上目录的服务，其中 controllerIO.java 通过 socket 接收来自交换机的连接，controller.java 是核心文件，实现 plugin 的 init，start，stop 以及 destroy 等，处理连接建立和拆除。

5.2 收发包过程简介 (packet service)

以下有些来自.opendaylight 的文档，有些自己查看代码的理解。

概述中，我们给出了.opendaylight controller 的设计框图，暂时忽略它的复杂，使用一个简单的模型，如下图，来分析下原来框图中的三个重要组成部分，即 application，sal 和 plugin。

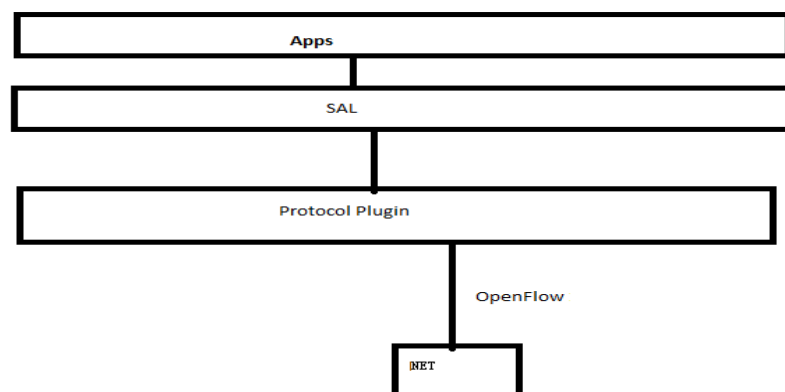


图 21

下面结合 controller 代码，以 packet service 为例介绍下它们是怎么工作的。

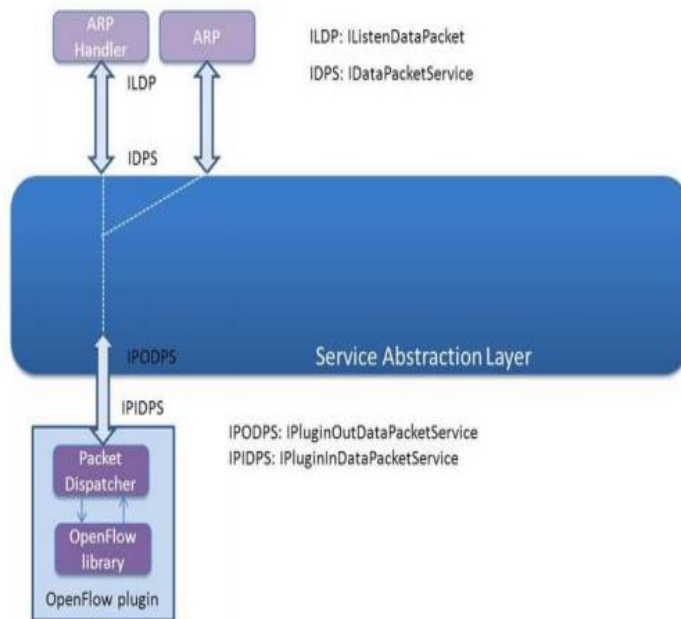


图 22 数据包服务

图 22 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。DataPacketMuxDemux.java 监听 PacketIn 消息，当数据包到来时，会触发 DataPacketMuxDemux.java 的 receive () 方法，然后调用 this.pluginOutDataPacketServices.receiveDataPacket() 来将数据包交给 pluginOutDataPacketServices 来处理。pluginOutDataPacketService 的类型是 IPluginOutDataPacketService，它由 sal.implementation.internal.DataPacketService.java 来实现。于是数据包到达了 SAL 层，由 DataPacketService.java 的 receiveDataPacket () 方法继续调用 dispatchpacket () 方法将数据包分发到实现 IListenDataPacket 服务的应用，这时数据包就到达了应用层，arphandler, simpleforwarding, loadbalance 都实现了 IListenDataPacket，数据包会发送到它们，然后分别处理。

下面是几个重要的服务（黑体的是我们刚才用到的）：

`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：

总结一下，接收数据包的流程是（以 `arp` 包为例）：

`OpenFlow` 插件获取了 `ARP` 包，该包需要被 `ARP Handler` 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。

6 Opendaylight 重要技术及文档

这段日子对.opendaylight 的很多文档都只是草草的看了一下，发现自己还不能对某个技术来做一下分析，下面列举下.opendaylight 几个关键技术的文档：

Opendaylight controller develop guid:

[GettingStarted:Developer_Main](#)

SAL 相关:

[MD-SAL:Architecture](#)

MD-SAL 下开发 plugin:

[MD-SAL:Southbound Plugin Development Guide](#)

Config subsystem:

[Config:Main](#)

[Config:Examples:Sample_Project](#)

Archetype:

[Maven Archetypes:odl-model-project](#)

另外，.opendaylight 各项目的邮件列表非常重要，因为它的很多文档都在逐步完善的过程中，遇到问题时，通过查看邮件列表，会很有效。

