# 搭建Nexus私服

由于OpenDayLight使用Maven 管理项目，每次编译缺少依赖项都需要前往国外站点进行下载，速度非常慢（只有30KB/s左右），严重影响工作效率。另外公司的环境也不允许使用外网环境。故需要搭建一个私服，用于提供项目所需的文件。

## 1.1下载安装Nexus

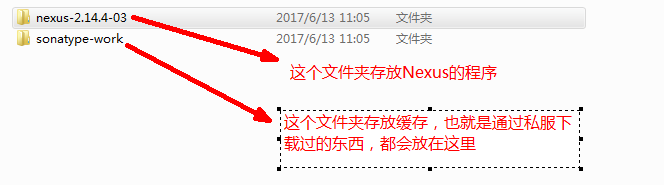
前往<https://www.sonatype.com/download-oss-sonatype>

下载nexus-2.14.4-03注意是版本2。

另外，可以上共享172.16.82. /share 获取nexus-2.14.4-03-bundle.zip。

下载完后，解压到指定目录，如E:\nexus。注意本文介绍的是在Windows环境下搭建，在Linux环境下搭建类似。

解压后如图所示：

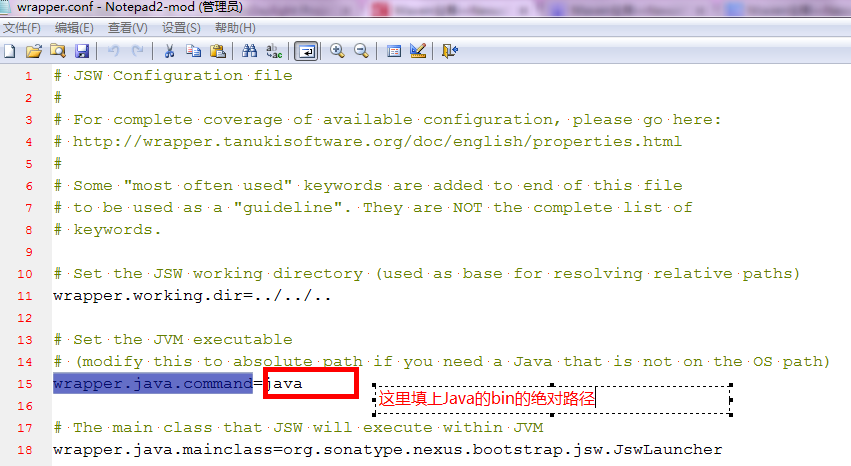


## 1.2配置Nexus

1、配置Java环境

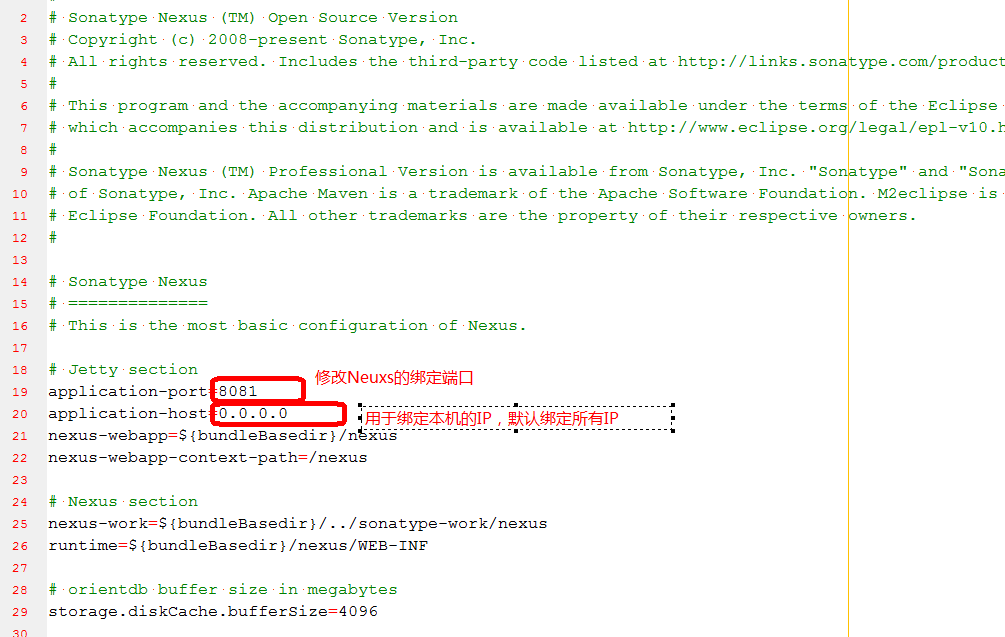
如果系统环境已经有Java环境了(有Java环境变量)，就不需要配置这一步(关于Java的配置，已在《SDN开发环境的搭建》中详细说明了)。这里的配置把，Nexus的Java命令指向Java的bin文件夹。

打开 ...\nexus-2.14.4-03\bin\jsw\conf 目录下的wrapper.conf，找到 wrapper.java.command 配置项，在 “=”，后面加上Java的bin文件夹的绝对路径。



2、配置绑定IP、端口

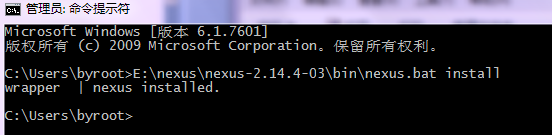
打卡 ...\nexus-2.14.4-03\conf 目录下的 nexus.properties 配置文件，按下图的方法修改:



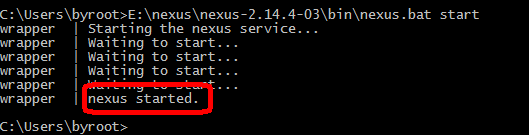
## 1.3启动Nexus

打开cmd 并把目录切换到nexus-2.14.4-03\bin目录下。

1、输入 nexus.bat install 安装Nexus服务



2、输入nexus.bat start启动Nexus服务。

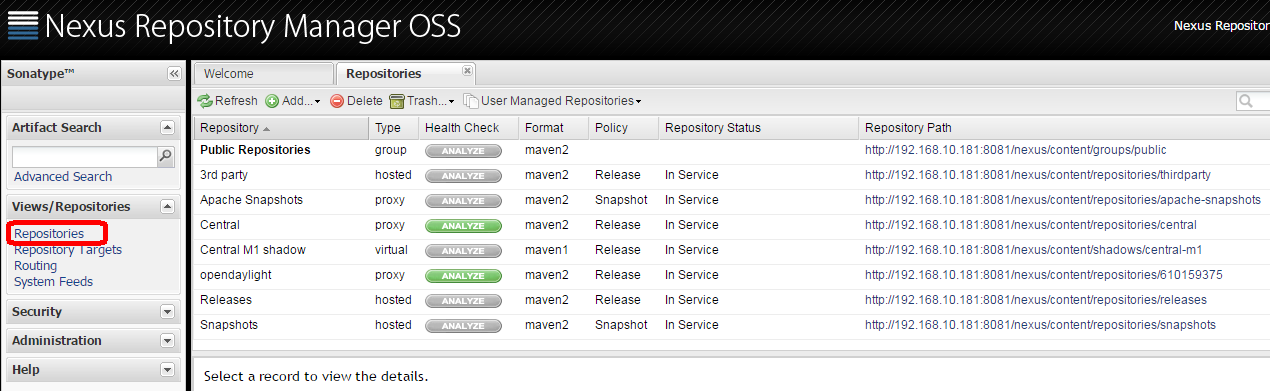


3、打开浏览器，输入 192.168.10.96:8181/nexux



点击右上角的log in，输入账号 admin 密码 admin123

点击Repositories进入仓库:



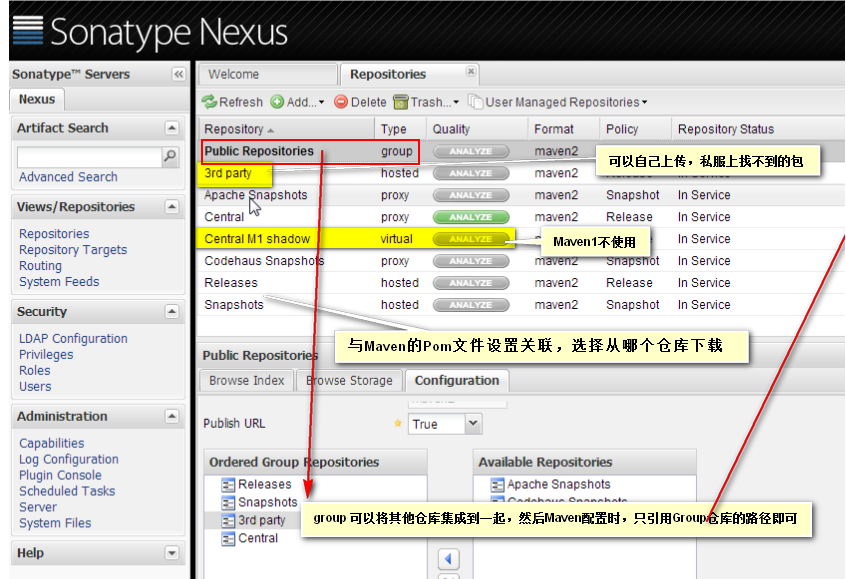
## 1.4Nexus仓库介绍

1、简单介绍下几种repository的类型：

hosted，本地仓库，通常我们会部署自己的构件到这一类型的仓库。比如公司的第二方库。

proxy，代理仓库，它们被用来代理远程的公共仓库，如maven中央仓库。

group，仓库组，用来合并多个hosted/proxy仓库，当你的项目希望在多个repository使用资源时就不需要多次引用了，只需要引用一个group即可。



2、管理本地仓库

　我们前面讲到类型为hosted的为本地仓库，Nexus预定义了3个本地仓库，分别是Releases, Snapshots, 3rd Party. 分别讲一下这三个预置的仓库都是做什么用的:

Releases:

这里存放我们自己项目中发布的构建, 通常是Release版本的, 比如我们自己做了一个FTP Server的项目, 生成的构件为ftpserver.war, 我们就可以把这个构建发布到Nexus的Releases本地仓库. 关于符合发布后面会有介绍.

Snapshots:

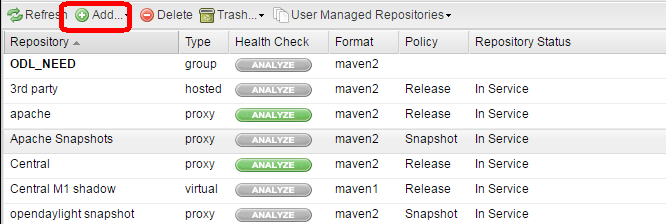
这个仓库非常的有用, 它的目的是让我们可以发布那些非release版本, 非稳定版本, 比如我们在trunk下开发一个项目,在正式release之前你可能需要临时发布一个版本给你的同伴使用, 因为你的同伴正在依赖你的模块开发, 那么这个时候我们就可以发布Snapshot版本到这个仓库, 你的同伴就可以通过简单的命令来获取和使用这个临时版本.

3rd Party:

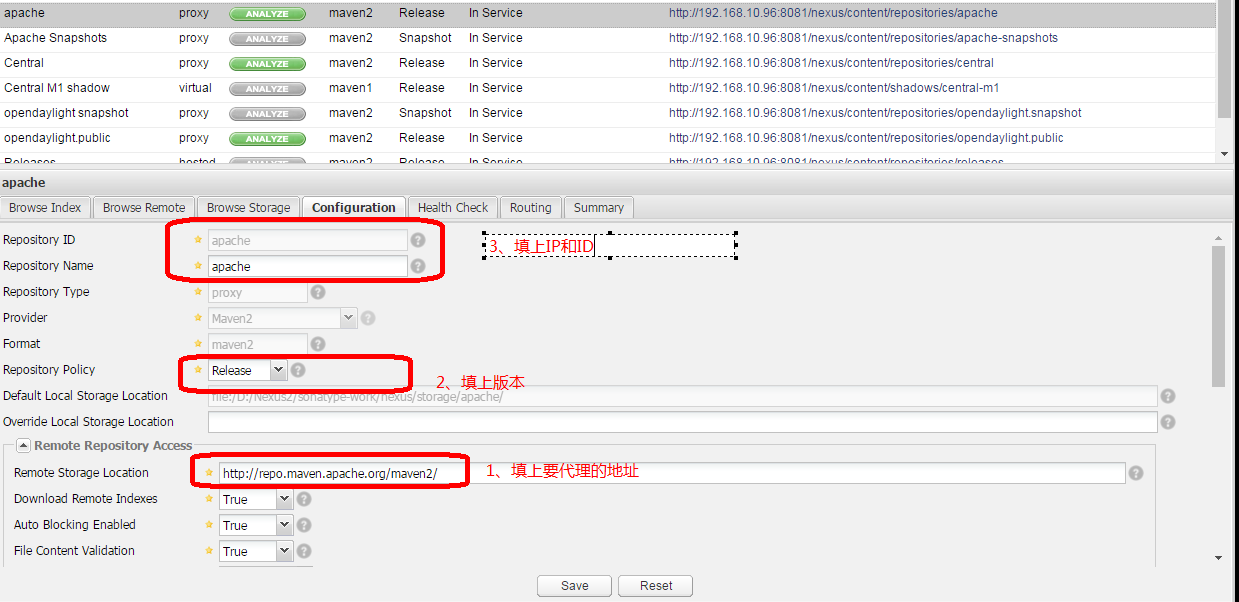
顾名思义, 第三方库, 你可能会问不是有中央仓库来管理第三方库嘛,没错, 这里的是指可以让你添加自己的第三方库, 比如有些构件在中央仓库是不存在的. 比如你在中央仓库找不到Oracle 的JDBC驱动, 这个时候我们就需要自己添加到3rdparty仓库。

3、创建自己的Nexus仓库

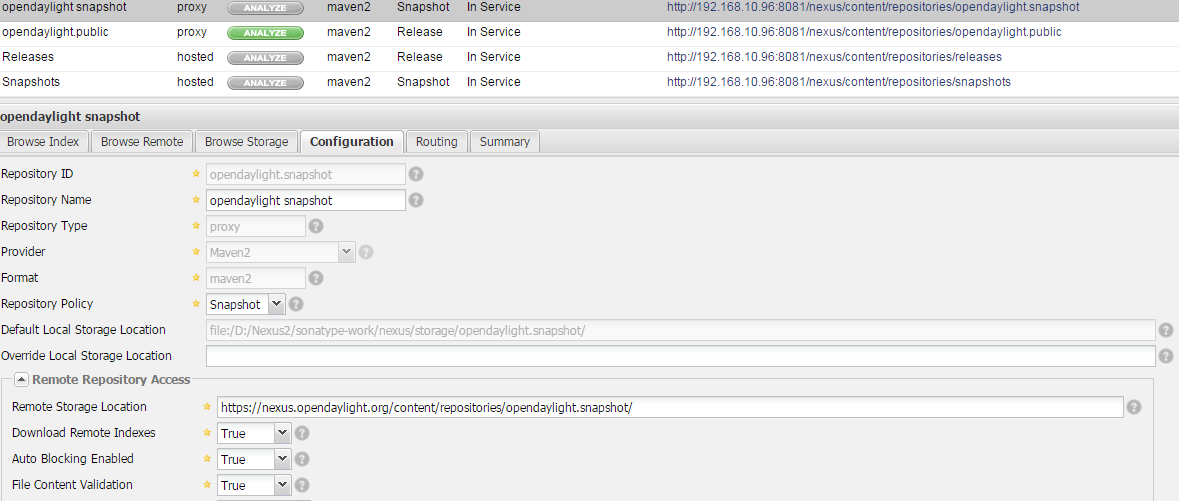
点击 add



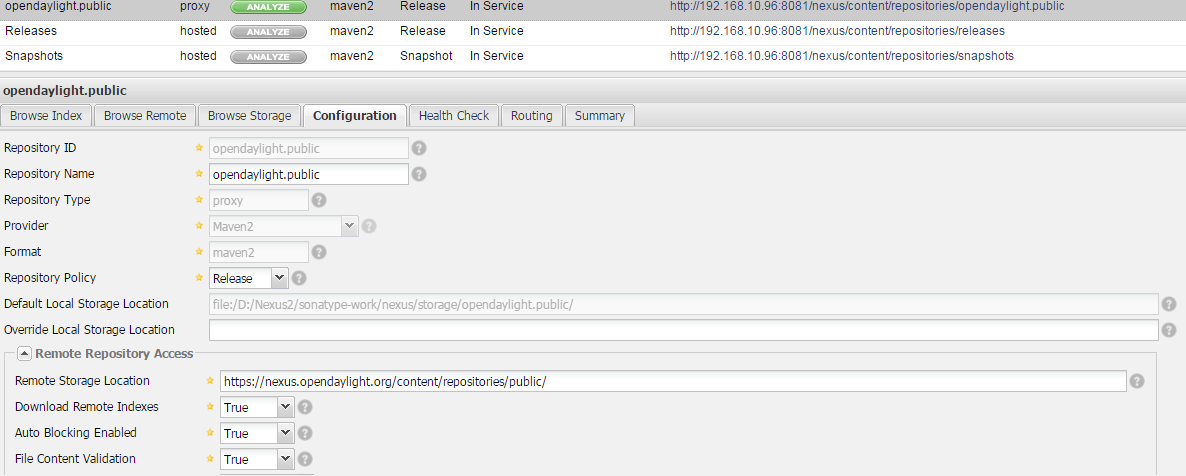
创建apache仓库



创建opendaylight snapshot仓库



创建opendaylight public

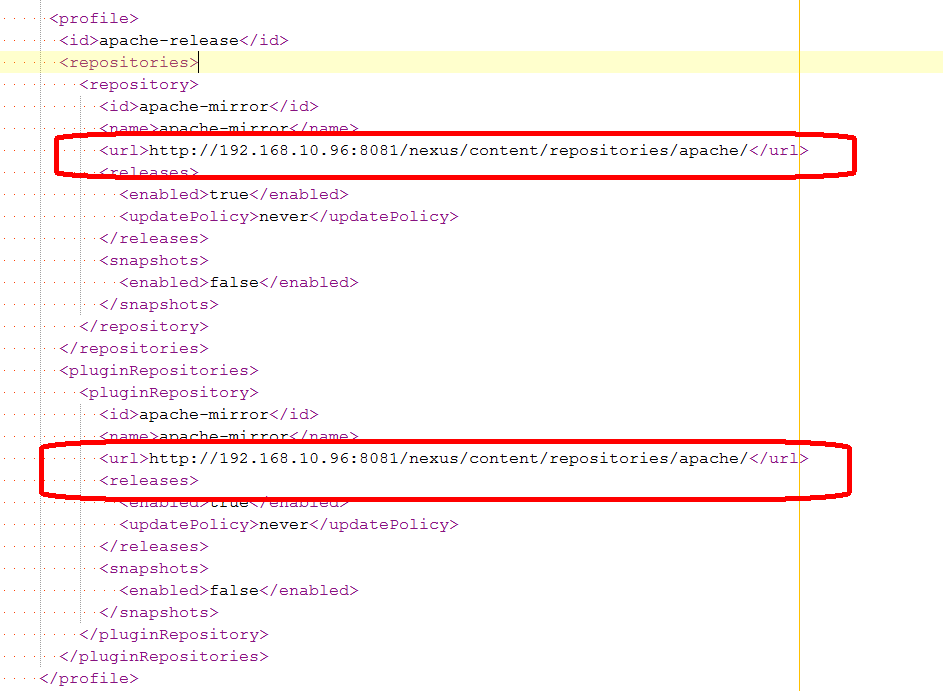


## 1.5Maven使用Nexus私服

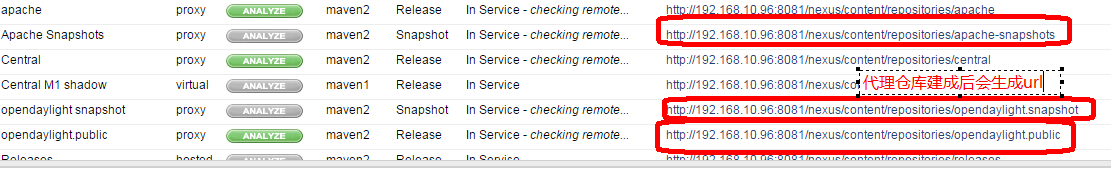
把如下文件(settings.xml)复制到 /root/.m2/ 目录下



修改settings.xml



把文件下所有的url都指向私服的url:



使用mvn clean install -Pq 测试，可发现下载速度相当快。

# 生成项目骨架

## 2.1骨架配置文件



把archetype-catalog.xml拷贝到 /root/.m2/ 目录下

## 2.2生成项目骨架

1、运行命令

mvn archetype:generate -DarchetypeGroupId=org.opendaylight.controller -DarchetypeArtifactId=opendaylight-startup-archetype \

-DarchetypeRepository=http://nexus.opendaylight.org/content/repositories/opendaylight.release/ \

-DarchetypeCatalog=local \

-DarchetypeVersion=1.2.2-Boron-SR2

* 注意:
* -DarchetypeCatalog=local 表示使用本地的archetype-catalog.xml文件，不能使用网上的，会报错。
* -DarchetypeVersion=1.2.2-Boron-SR2，表示版本号，要和使用的发行版的版本号要相同，若不相同，到时候把模块集合到发行版，要修改配置文件才能正常安装，但这样也不确定会不会有隐患。各版本对应的值:
* For the current Master (Carbon) use Snapshot-Type=**opendaylight.snapshot** Archetype-Version=**1.3.0-SNAPSHOT**
* For the Carbon snapshot use Snapshot-Type=**opendaylight.release** Archetype-Version=**1.3.0-Carbon**
* For Boron "SR0" use Snapshot-Type=**opendaylight.release** Archetype-Version=**1.2.0-Boron**
* For Boron SR1 use Snapshot-Type=**opendaylight.release** Archetype-Version=**1.2.1-Boron-SR1**
* For Boron SR2 use Snapshot-Type=**opendaylight.release** Archetype-Version=**1.2.2-Boron-SR2**
* For the Boron snapshot use Snapshot-Type=**opendaylight.snapshot** Archetype-Version=**1.2.2-SNAPSHOT**

2、填写模块信息

Define value for property 'groupId': : org.opendaylight.hello

Define value for property 'artifactId': : hello

Define value for property 'version': 1.0-SNAPSHOT

Define value for property 'package': org.opendaylight.hello: :

Define value for property 'classPrefix': hello

Define value for property 'copyright': : Yoyodyne, Inc.

填完后按回车就可以了。

'groupId'以org.opendaylight.X 这样的形式填写

'artifactId'填写X。

'package'这里不需要填，按回车即可。

'classPrefix'这里填

${artifactId.substring(0,1).toUpperCase()}${artifactId.substring(1)}

'copyright'填Yoyodyne, Inc.

## 2.3项目骨架介绍

等待项目骨架生成完成后，切到该目录下，就可以看到如下文件夹:



api/

artifacts/

cli/

deploy-site.xml

features/

impl/

karaf/

pom.xml

src/

target/

其中api/这里存放 \*.yang文件;

impl/ 自己实现接口;

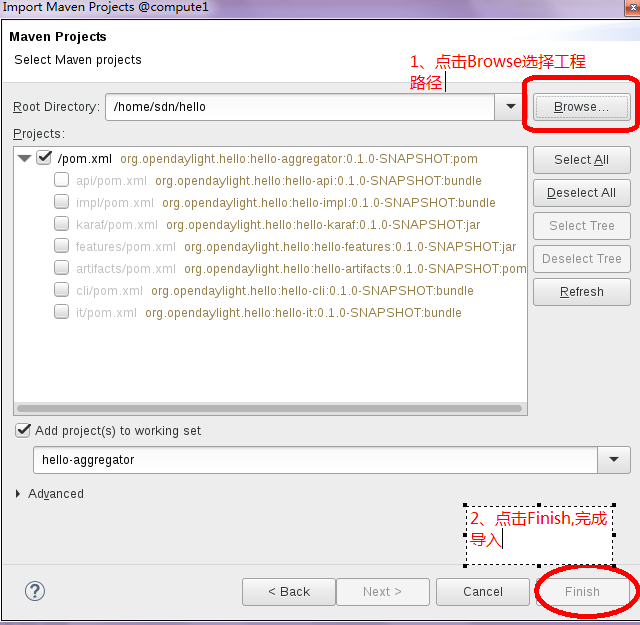
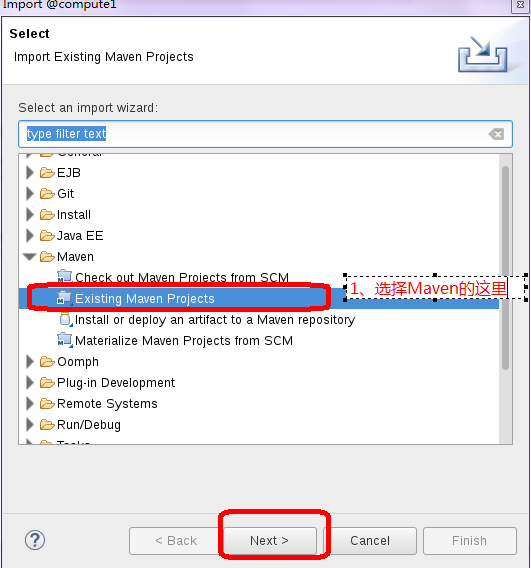
代码的编写主要在 api/和impl/下完成。

# 模块逻辑代码编写

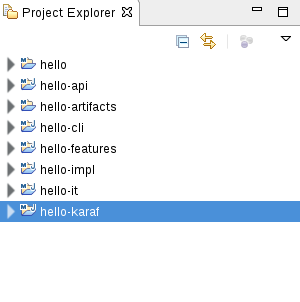
## 3.1导入模块工程

打开Eclipse(版本采用Eclipse Jee Neon，如何安装Eclipse已经在《SDN开发环境搭建》中详细说明)，点击file -> Import

弹出窗口后按如下的操作:



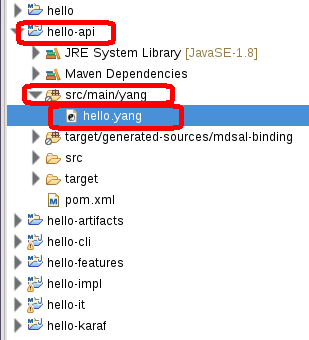
导入完成后，在右侧可以看到整个项目:



## 3.2在Yang文件添加RPC API

### 3.2.1代码添加

打开hello-api->src/main/yang->hello.yang



在这个Yang模型中可以添加数据、RPC和异步通知。

按如下代码添加进去:

module hello {

yang-version 1;

namespace "urn:opendaylight:params:xml:ns:yang:hello";

prefix "hello";

revision "2015-01-05" {

description "Initial revision of hello model";

}

rpc hello-world {

input {

leaf name {

type string;

}

}

output {

leaf greeting {

type string;

}

}

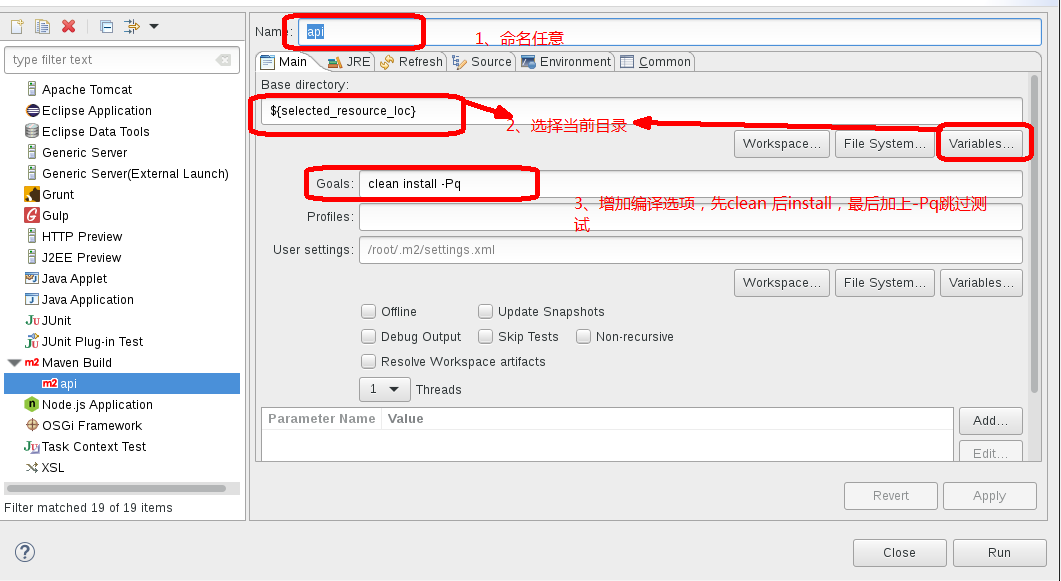
}

}

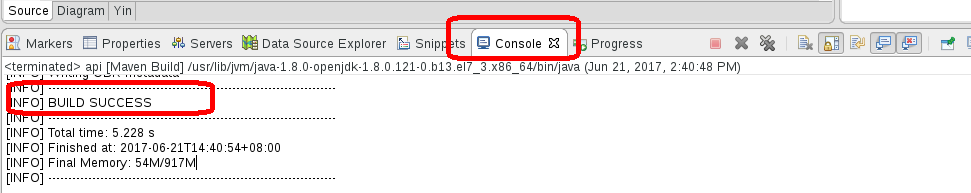
### 3.2.2编译小模块

右键点击hello-api，run as->Maven build

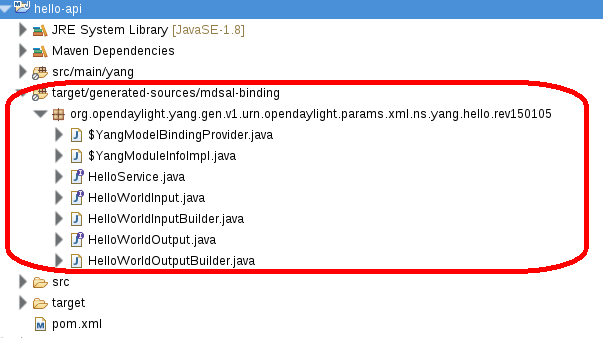
如果缺少编译命令，就按如下方法添加:



编译成功后，在下方的Console可以看到编译的详细情况:



在右边可以看到编译生成的target:



这里注意：有多少个Yang文件，它就能生成多少个包，即可以定义多个模型等。

RPC对应 \*Service.java。

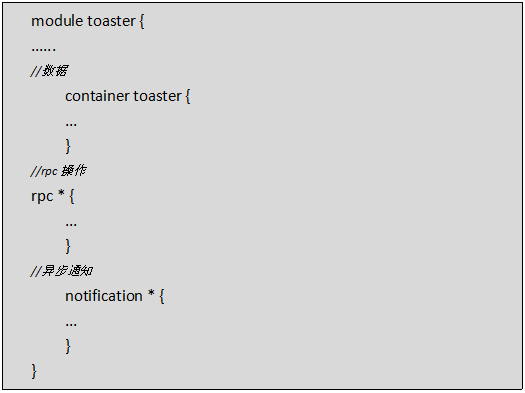
Yang文件定义好后，可以使用Yang Tools进行Yang语法解析、语义模型以及Java文件的映射，这样做可以极大地提高了编程效率。映射关系在3.2.3中详细说明。

### 3.2.3Yang的详细介绍

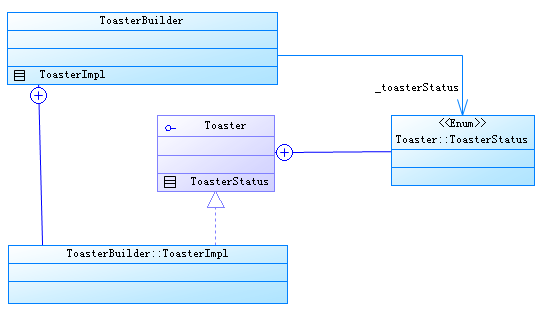
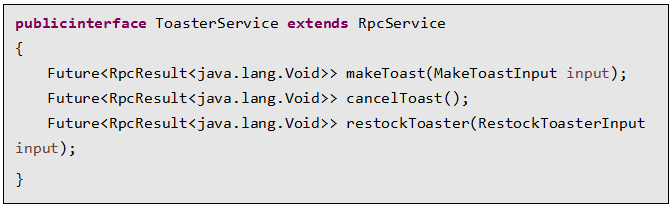
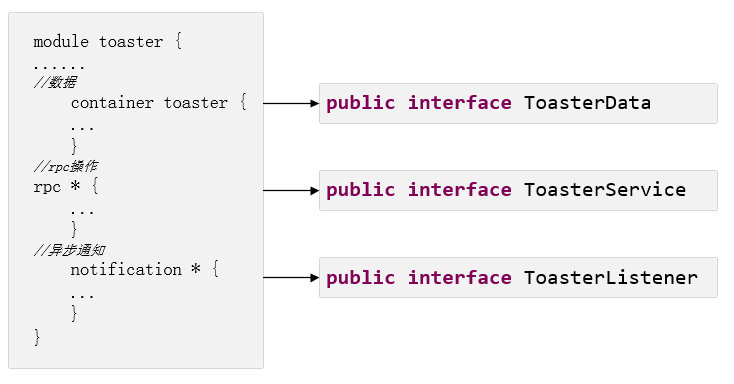
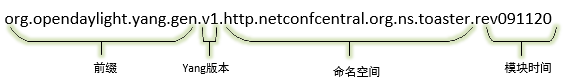
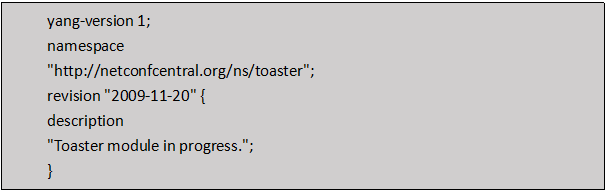
OpenDaylight中的YANG Tools，为使用YANG提供了一组支撑工具（库）。主要功能包括将YANG模型转译为Java代码，含将YANG模型转译为标准化节点的方式（类DOM树模型）和Java Binding方式。并提供了YANG数据模型与XML、JSON，以及java Binding与标准节点格式间的序列化和反序列化支持。其包括的YANG Maven Plugin所实现的Maven插件，可以在Maven构建生命周期中，基于YANG数据模型为其它组件生成Java代码（框架）。

由于Hello的模型材料比较少，以下使用toaster为例进行说明:

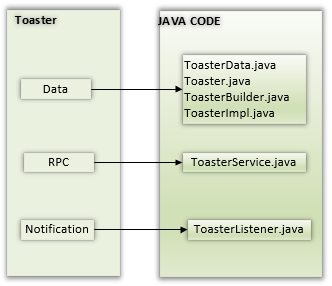
Yang的内容框架如下所示:



来看下包路径的映射关系，toaster.yang文件中定义信息如下：  
  
所以，从定义来看，包路径的映射关系如下：  
  
其次，用来描述数据的Container标识映射到ToasterData.java；用来定义RPC方法的RPC标识映射为ToasterService.java；用来描述异步通知方法的notification标识映射为ToasterListener.java。它们的映射关系如下图所示：  
  
RPC操作和Notification异步通知各自对应接口文件，文件内包含对应相应的方法，如ToasterService文件定义如下，比较简单。  
  
再次，我们来详细说明如何来构造烤面包机的数据？Yang Tools根据Container生成类或接口文件ToasterData、Toaster和ToasterBuilder，其中ToasterData、Toaster是接口类，而ToasterBuilder为具体实例的建造者。其类关系图如下所示：  
  
通俗一点来解释，客户小A想要一台烤面包机(小熊牌、型号R7等)，便找ToasterBuilder，让其帮忙构建一台烤面包机，此时ToasterBuilder为难了，因为其只能构建一些零件，不掌握整装技术，怎么办呢？找整装设计师吧，于是ToasterImpl产生了，告诉ToasterBuilder如何一步一步去完成，ToasterBuilder和ToasterImpl相互配合就构建出了小A的烤面包机。



最后，通过上面的分析，我们可以更详细地得到Yang Tools将Yang文件映射为Java文件的关系图，如下所示：



## 3.3.实现HelloWorld RPC API

在3.2中定义了一些接口，在这里节中，就是实现这些接口。代码的实现在impl/ 目录下。

在impl里面新增HelloWorldImpl.java文件。



添加如下代码:

package org.opendaylight.hello.impl;

import java.util.concurrent.Future;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.hello.rev150105.HelloService;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.hello.rev150105.HelloWorldInput;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.hello.rev150105.HelloWorldOutput;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.hello.rev150105.HelloWorldOutputBuilder;

import org.opendaylight.yangtools.yang.common.RpcResult;

import org.opendaylight.yangtools.yang.common.RpcResultBuilder;

public class HelloWorldImpl implements HelloService {

@Override

public Future<RpcResult<HelloWorldOutput>> helloWorld(HelloWorldInput input) {

HelloWorldOutputBuilder helloBuilder = new HelloWorldOutputBuilder();

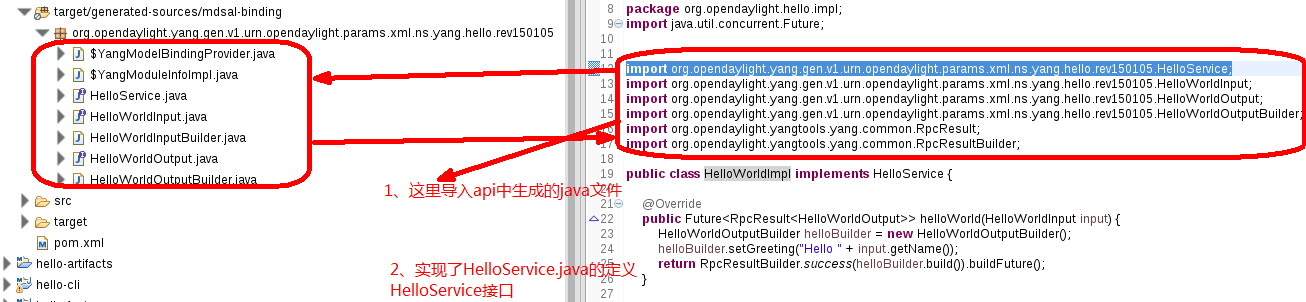
helloBuilder.setGreeting("Hello " + input.getName());

return RpcResultBuilder.success(helloBuilder.build()).buildFuture();

}

}

代码说明:



## 3.4注册RPC 到 MD-SAL

3.3中，实现了HelloWorld的RPC接口，但RPC接口还没有在ODL中注册，这一节介绍如何注册RPC到MD-SAL中。

### 3.4.1添加引用到impl-blueprint.xml

为了在MD-SAL中注册我们的RPC，我们首先需要创建对RPC注册表的引用。 这是通过几个步骤完成的。 首先，我们需要在src / main / resources / org / opendaylight / blueprint中的impl-blueprint.xml文件中添加RPC注册表引用，如下所示：

增加一行来定义引用为RPC注册接口的实现，我们称这个引用为rpcRegisty

将对rpc注册表接口的实现的引用添加到“HelloProvider”构造函数的参数。

<?xml version="1.0" encoding="UTF-8"?>

<!-- vi: set et smarttab sw=4 tabstop=4: -->

<!--

Copyright © 2016 Cisco Systems and others. All rights reserved.

This program and the accompanying materials are made available under the

terms of the Eclipse Public License v1.0 which accompanies this distribution,

and is available at http://www.eclipse.org/legal/epl-v10.html

-->

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"

xmlns:odl="http://opendaylight.org/xmlns/blueprint/v1.0.0"

odl:use-default-for-reference-types="true">

<reference id="dataBroker"

interface="org.opendaylight.controller.md.sal.binding.api.DataBroker"

odl:type="default" />

<reference id="rpcRegistry"

interface="org.opendaylight.controller.sal.binding.api.RpcProviderRegistry"/>

<bean id="provider"

class="org.opendaylight.spark.impl.HelloProvider"

init-method="init" destroy-method="close">

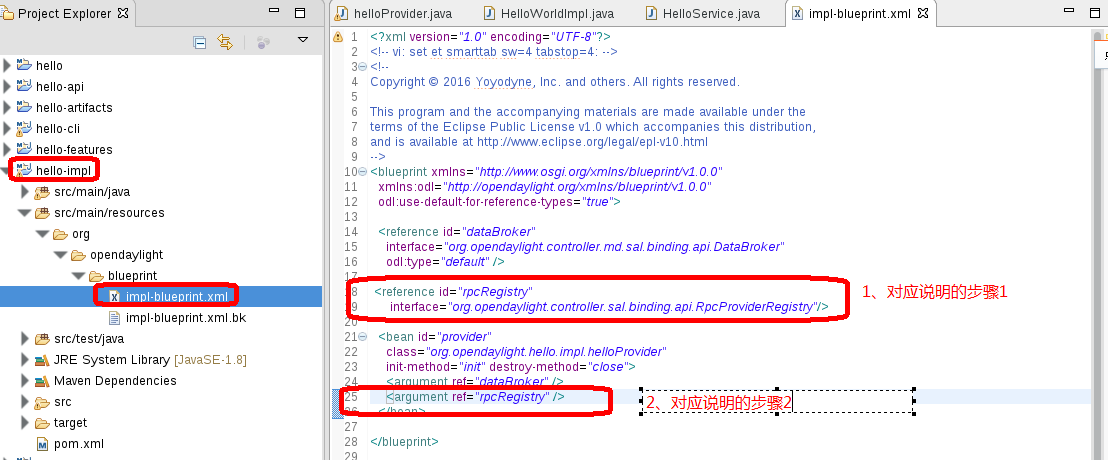
<argument ref="dataBroker" />

<argument ref="rpcRegistry" />

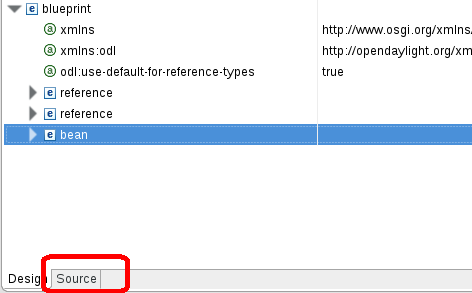
</bean>

</blueprint>

添加的地方如下图所示:



这里有个小技巧,Eclipse默认用design的方法打开xml，导致编译起来有些困难，注意点Source切换到平时看到的状态:



### 3.4.2注入Provider

接下来，我们将RPC注册表的引用添加到HelloProvider类，并允许通过HelloProvider的构造函数注入到类中：

/\*

\* Copyright © 2016 Cisco Systems and others. All rights reserved.

\*

\* This program and the accompanying materials are made available under the

\* terms of the Eclipse Public License v1.0 which accompanies this distribution,

\* and is available at http://www.eclipse.org/legal/epl-v10.html

\*/

package org.opendaylight.hello.impl;

import org.opendaylight.controller.md.sal.binding.api.DataBroker;

import org.opendaylight.controller.sal.binding.api.RpcProviderRegistry;

import org.opendaylight.controller.sal.binding.api.BindingAwareBroker.RpcRegistration;

import org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.hello.rev150105.HelloService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class HelloProvider {

private static final Logger LOG = LoggerFactory.getLogger(HelloProvider.class);

private final DataBroker dataBroker;

private final RpcProviderRegistry rpcProviderRegistry;

public HelloProvider(final DataBroker dataBroker, RpcProviderRegistry rpcProviderRegistry) {

this.dataBroker = dataBroker;

this.rpcProviderRegistry = rpcProviderRegistry;

}

/\*\*

\* Method called when the blueprint container is created.

\*/

public void init() {

LOG.info("HelloProvider Session Initiated");

}

/\*\*

\* Method called when the blueprint container is destroyed.

\*/

public void close() {

LOG.info("HelloProvider Closed");

}

}

最后，在HelloProvider的init（）函数中（当创建blueprint容器时调用），我们使用MD-SAL注册HelloService RPC，如下所示：

....

private RpcRegistration<HelloService> serviceRegistration;

...

/\*\*

\* Method called when the blueprint container is created.

\*/

public void init() {

serviceRegistration = rpcProviderRegistry.addRpcImplementation(HelloService.class, new HelloWorldImpl());

LOG.info("HelloProvider Session Initiated");

}

/\*\*

\* Method called when the blueprint container is destroyed.

\*/

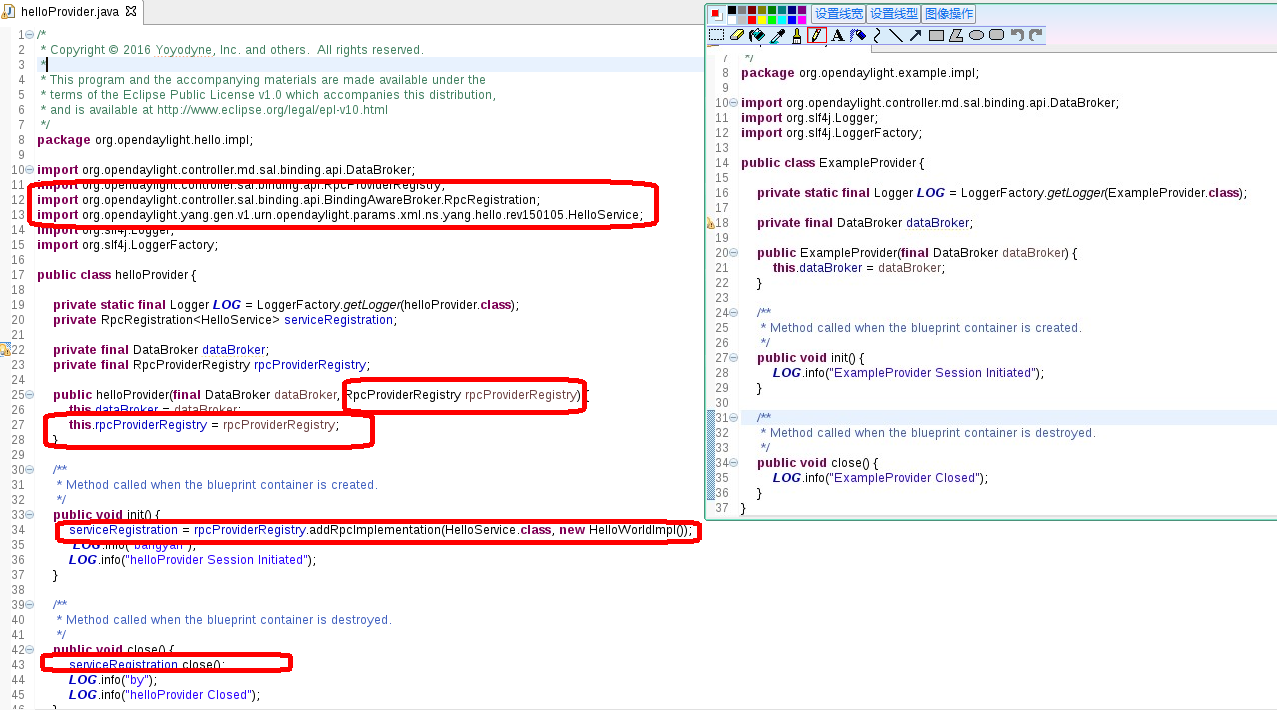
public void close() {

serviceRegistration.close();

LOG.info("HelloProvider Closed");

}

添加部分如下图所示:



### 3.4.3MD-SAL框架内部实现

HelloWorldimpl.java将从restful API（北向接口发送）发送过来(实际上经过HelloService)的信息进行处理，然后再返回给北向接口。



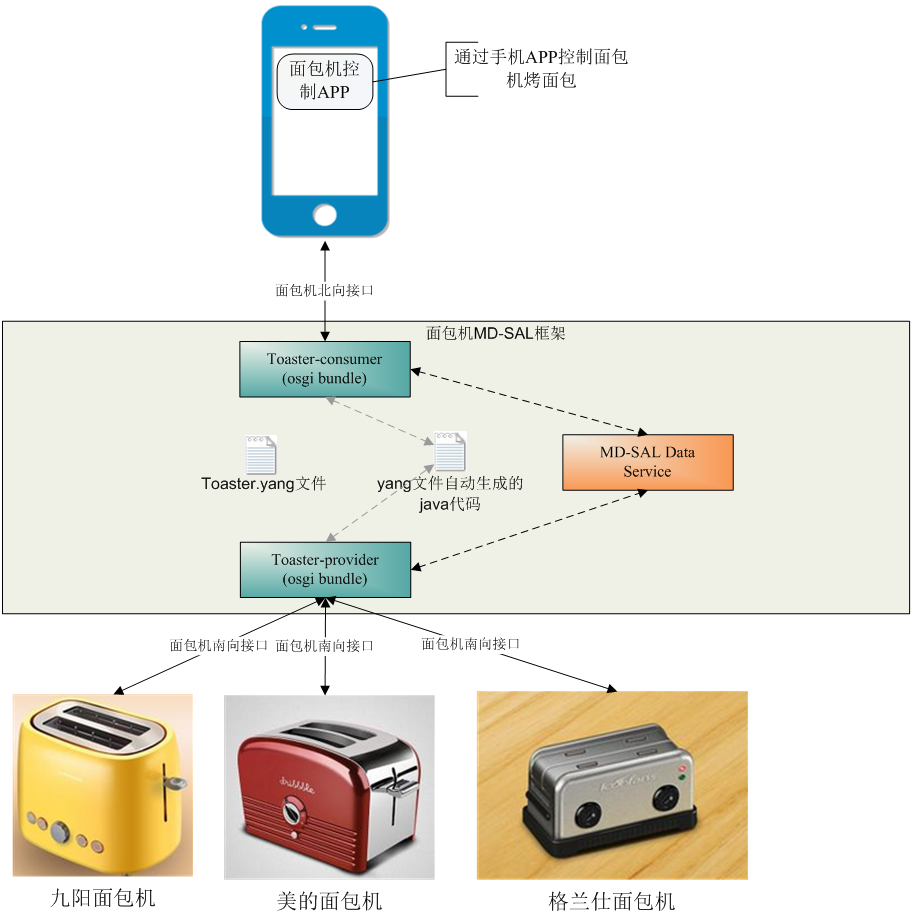
在

public Future<RpcResult<HelloWorldOutput>> helloWorld(HelloWorldInput input)

这个方法里面实现南向接口(官方的网站没实现，只是在你输入的字符里面简单添加“hello”)。

关于北向接口，不用我们自己实现，在3.4节中，注册RPC到MD-SAL 就是个实现过程。

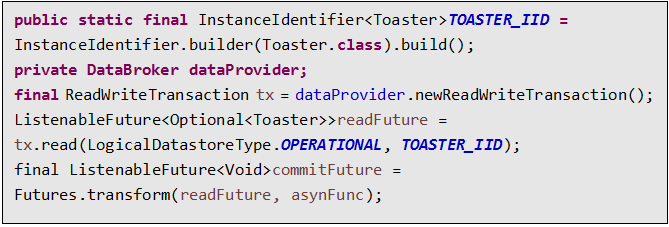
实现过程类似于：



其中，如果要使用到数据库，使用如下方法:

在OpenDaylight中，所有数据都保存在DataStore中，并且数据以树形结构存储，而DataBroker是访问MD-SAL数据存储的接口，它提供了3种访问方式只读、只写和读写，分别对应newReadOnlyTransaction、newWriteOnlyTransaction和newReadWriteTransaction，具体代码如下所示：

这节理解还是有点模糊，但先出一个版本，比较深刻理解，再出一版,2017.6.21



### 3.4.4编译模块

右键点击hello，run as->Maven build

等待编译完成即可。

# 模块功能测试

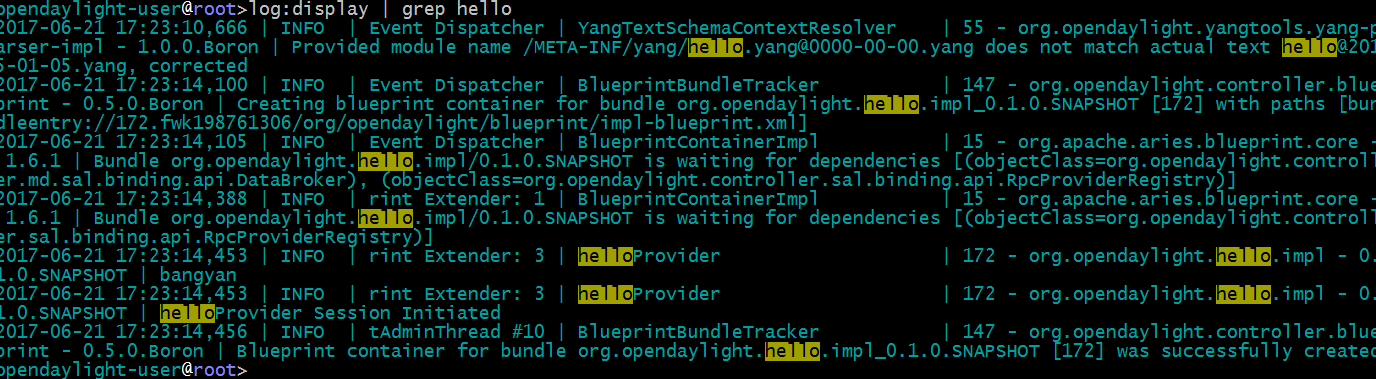
## 4.1系统打印检测

启动karaf容器

切换到hello/karaf/target/assembly/bin/

./karaf

输入 log:display | grep Hello



## 4.2restful API测试

使用浏览器登录

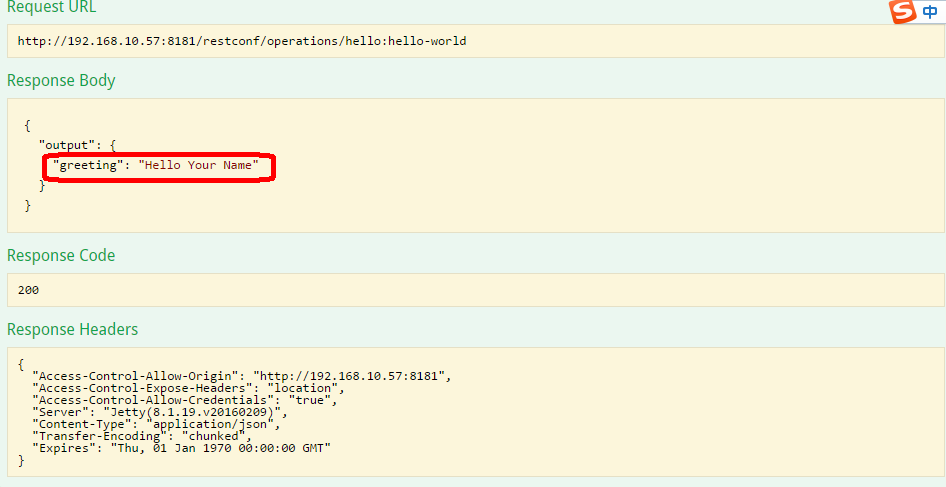
<http://192.168.10.57:8181/apidoc/explorer/index.html>

在下图的位置键入：

{"hello:input": { "name":"Your Name"}}



结果如图所示:



可以看出，新加进来的模块已经正常运行了。

# 模块加入发行版和测试

## 5.1拷贝模块到发行版

1、编译完成的模块都会在本地仓库生成jar源。

Linux中，在/root/.m2/repository/org/opendaylight/ 目录下。



2、发行版的存放模块的路径放在

/home/distribution-karaf-0.5.2-Boron-SR2/system/org/opendaylight/

3、拷贝模块到发行版

cp /root/.m2/repository/org/opendaylight/hello /home/distribution-karaf-0.5.2-Boron-SR2/system/org/opendaylight –rf

## 5.2发行版安装模块

1、启动OpenDayLight

cd .../bin/

./karaf

2、让新模块加入repo

feature:repo-add mvn:org.opendaylight.hello/hello-features/0.1.0-SNAPSHOT/xml/features

注意: 后面的xml/features照着写。

org.opendaylight.hello/hello-features/0.1.0-SNAPSHOT/

这个路径代表着

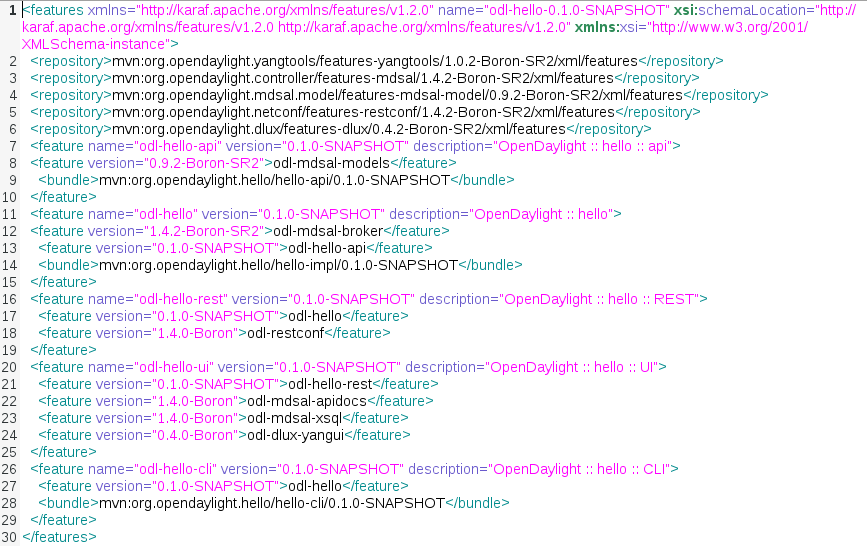
.../distribution-karaf-0.5.2-Boron-SR2/system/org/opendaylight/hello/hello-features/0.1.0-SNAPSHOT。

* 1. SNAPSHOT版本号要正常对应你的模块的版本号。

xml代表访问0.1.0-SNAPSHOT目录下的xml文件，如：

hello-features-0.1.0-SNAPSHOT-features.xml

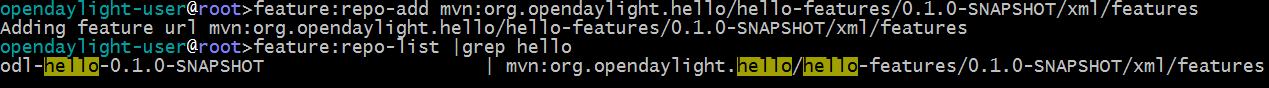
features指xml里面的项:



如果在第2章中指定错了版本，可以修改xml里面的项目，将它们的版本改为发行版存在版本即可，改完后，经测试，应用能正常运行，输入输出也正常，但不清楚会不会存在其它问题，此方案只可作备用方案，建议在第二章中，正常指定版本号。

3、验证repo

输入 feature:repo-list |grep hello ，发现表项存在：



## 5.3安装模块

feature:install odl-hello



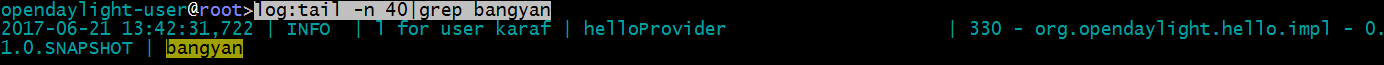
## 5.4模块测试

1、查看日志信息

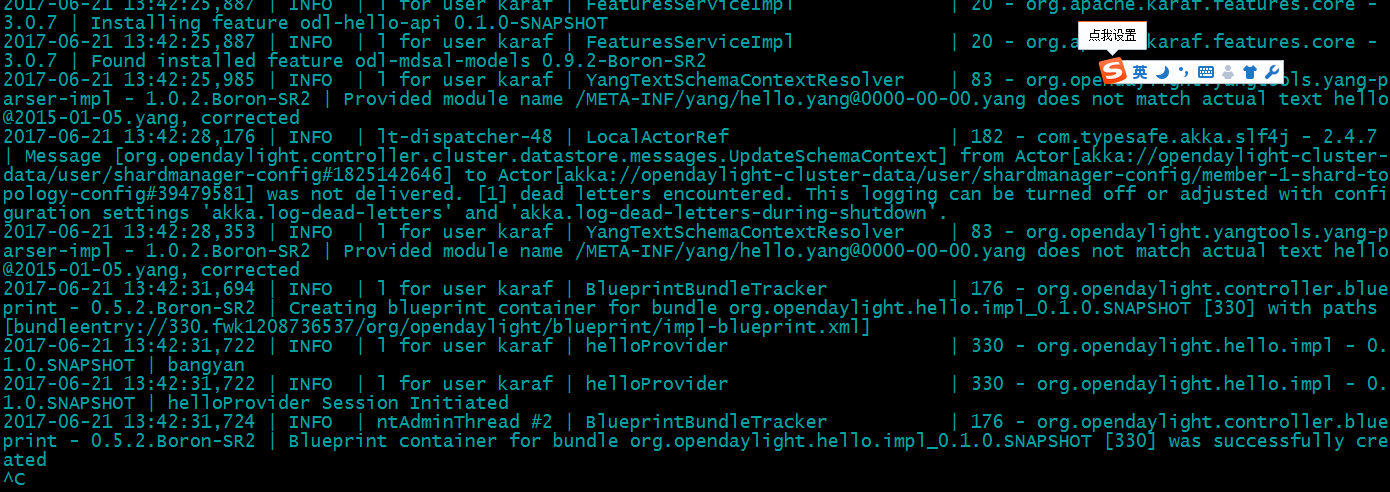
log:tail -n 40|grep hellp

或者

log:display|grep bangyan



log:display



可以看出模块已经成功加载。

2、restful API测试

使用浏览器登录

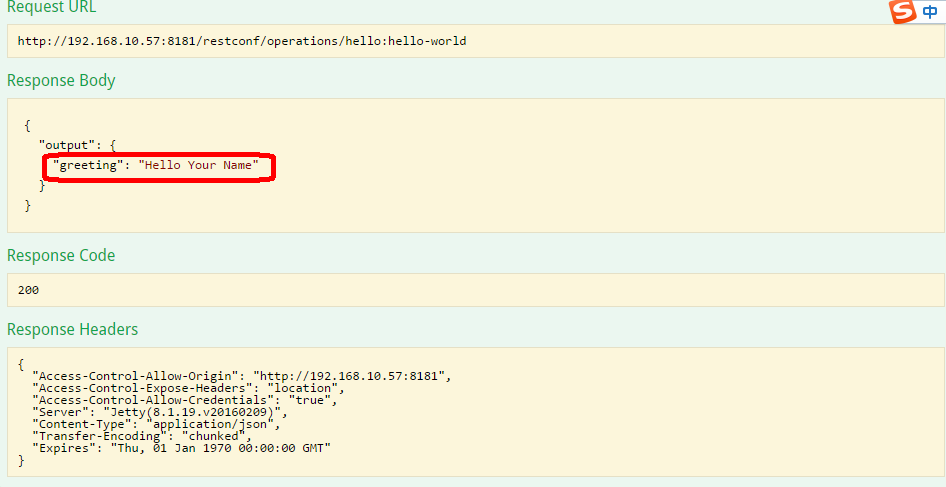
<http://192.168.10.57:8181/apidoc/explorer/index.html>

在下图的位置键入：

{"hello:input": { "name":"Your Name"}}



结果如图所示:



可以看出，新加进来的模块已经正常运行了。