|  |
| --- |
| 密级：受控 |
|  |
|  |



MD-SAL应用开发指南

|  |  |
| --- | --- |
| **版本** | V0.1 |

修订记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期 | 修订版本 | 修改者 | 操作 | 修改描述 |
| 2016/5/18 | V0.1 | 黄东远 | 创建 | 创建并编写文档 |

目录

[1 开发环境 5](#_Toc451604078)

[2 开发指南 5](#_Toc451604079)

[2.1 开发流程 5](#_Toc451604080)

[2.2 代码骨架 5](#_Toc451604081)

[2.3 Yang建模 9](#_Toc451604082)

[2.3.1 模型声明 9](#_Toc451604083)

[2.3.2 数据建模 10](#_Toc451604084)

[2.3.3 Rpc 11](#_Toc451604085)

[2.3.4 Notification 13](#_Toc451604086)

[2.3.5 Yang建模规范 13](#_Toc451604087)

[2.4 Java API 14](#_Toc451604088)

[2.5 Rest API 15](#_Toc451604089)

[2.5.1 data 15](#_Toc451604090)

[2.5.2 rpc 16](#_Toc451604091)

[2.6 功能实现 16](#_Toc451604092)

[2.6.1 初始化 16](#_Toc451604093)

[2.6.2 MD-SAL服务获取 17](#_Toc451604094)

[2.6.3 消息订阅与发布 20](#_Toc451604095)

[2.6.4 Rpc 20](#_Toc451604096)

[2.6.5 数据存储实现 22](#_Toc451604097)

[2.6.6 集群实现 23](#_Toc451604098)

[2.7 feature声明 23](#_Toc451604099)

[2.8 构建控制器版本 23](#_Toc451604100)

[3 应用开发DEMO 24](#_Toc451604101)

[3.1 生成代码骨架 24](#_Toc451604102)

[3.2 Yang建模 24](#_Toc451604103)

[3.2.1 device-manager建模 24](#_Toc451604104)

[3.2.2 ping- discovery建模 26](#_Toc451604105)

[3.3 生成API 27](#_Toc451604106)

[3.4 功能实现 30](#_Toc451604107)

[3.4.1 初始化 31](#_Toc451604108)

[3.4.2 DeviceManagerProvider实现 32](#_Toc451604109)

[3.4.3 PingProvider实现 36](#_Toc451604110)

[3.5 feature声明 41](#_Toc451604111)

[3.6 构建发布版本 41](#_Toc451604112)

[3.7 控制器运行 41](#_Toc451604113)

[3.8 功能测试 42](#_Toc451604114)

[3.9 Rest API 48](#_Toc451604115)

[3.9.1 获取设备配置 48](#_Toc451604116)

[3.9.2 获取设备状态 48](#_Toc451604117)

[3.9.3 创建设备 49](#_Toc451604118)

[3.9.4 删除设备 49](#_Toc451604119)

[3.9.5 获取ping会话信息 49](#_Toc451604120)

# 开发环境

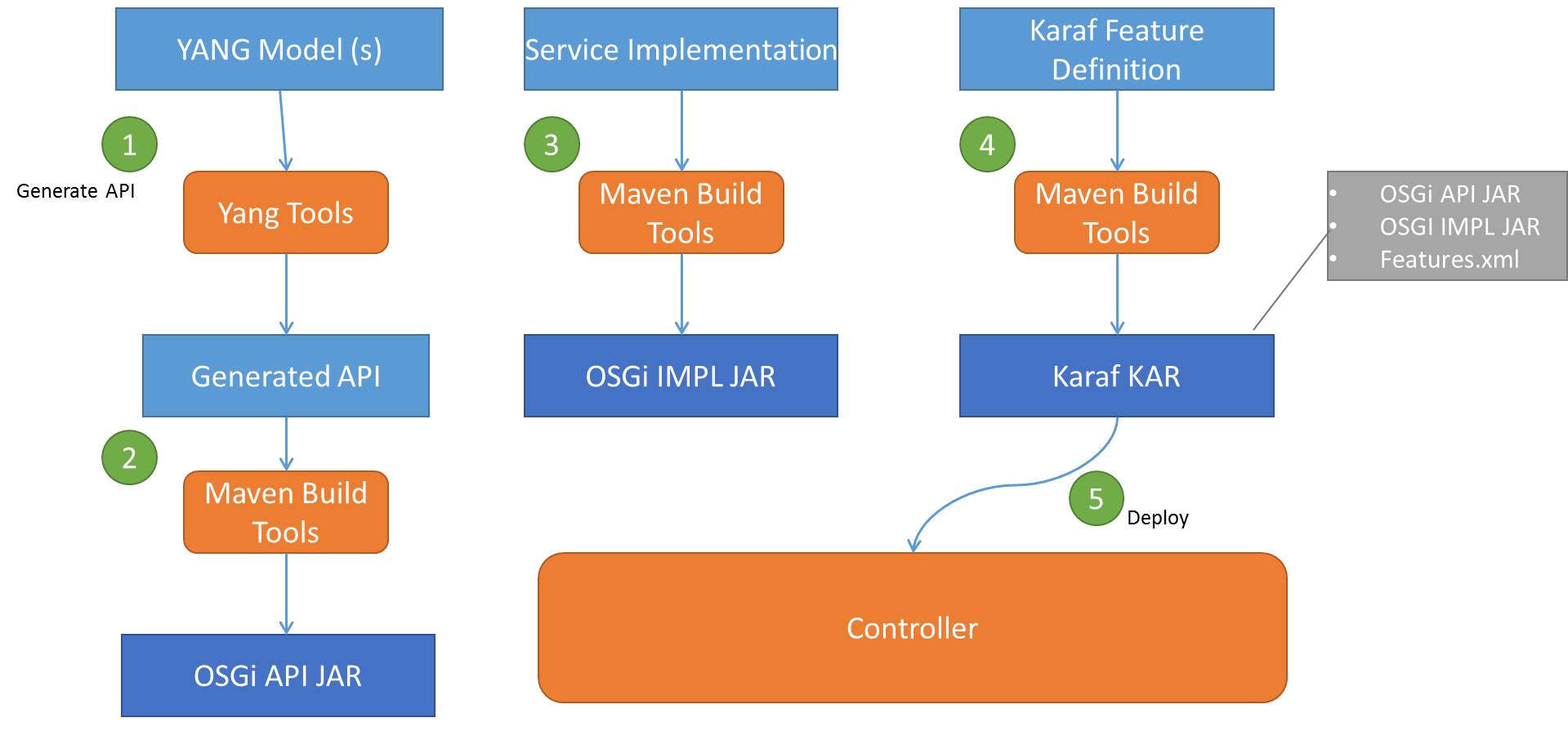
* Maven >= 3.2.3
* Eclipse >= 4.5.1
* JDK >= 1.7

关于开发环境搭建，请参考《RG-ONC环境搭建及开发指南.doc》，

本文后面的描述，要求已经完成jdk/maven/eclipse等的开发环境的搭建。

# 开发指南

## 开发流程



MD-SAL的应用开发，包括以下几个步骤：

1. Yang建模：对应用涉及的功能、消息、数据进行建模
2. 生成API：利用Yangtools的maven插件，生成Java API
3. API实现：应用的核心功能实现
4. 打包：将bundle封装成karaf的feature
5. 部署：应用部署及运行

## 代码骨架

为简化应用开发和统一规范，RG-ONC提供代码骨架生成工具，利用Maven的archetype实现。

在命令行窗口执行命令：

mvn archetype:generate -DarchetypeGroupId=cn.com.ruijie.rgonc -DarchetypeArtifactId=rgonc-startup-archetype -DarchetypeVersion=2.0.0 -DarchetypeRepository=http://admin:admin123@192.168.5.209:8081/nexus/content/repositories/releases/ -DarchetypeCatalog=http://192.168.5.209:8081/nexus/content/repositories/releases/archetype-catalog.xml

根据提示输入：groupId/artifactId/version/package/repoName/classPrefix/copyright等信息

Define value for property 'groupId': : cn.com.ruijie.rgonc.ping

Define value for property 'artifactId': : ping

Define value for property 'version': 1.0-SNAPSHOT: :

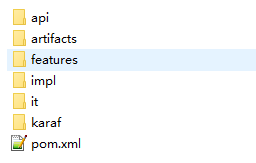
Define value for property 'package': cn.com.ruijie.rgonc.ping: :

Define value for property 'classPrefix': Ping: :

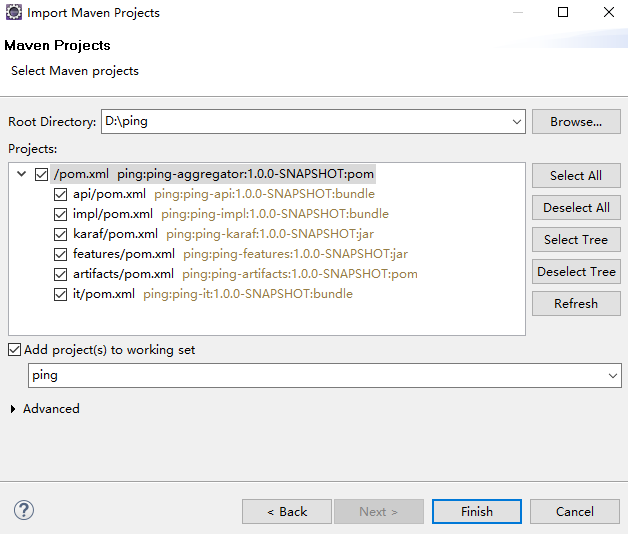
Define value for property 'copyright': : Copyright (c) 2016 Ruijie Networks, Inc.

最后，输入Y，进行确认

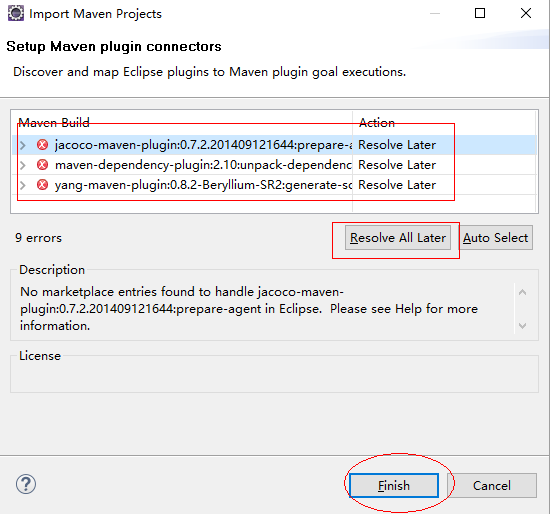
生成如下目录结构：



导入到Eclipse：



导入过程可能会弹出下面信息，直接点击“Resolve All Later”然后Finish即可。

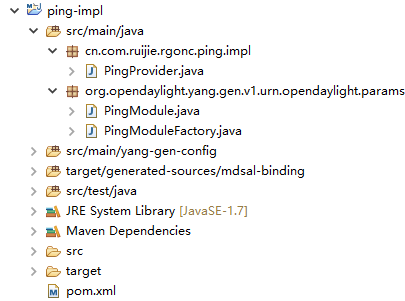


导入完成，看到工程视图：



各工程说明如下：

* ping-api：yang/工具类/其它接口
  + src/main/yang/： yang文件存放路径
  + target/generated-sources/mdsal-binding：yang生成的java api
* ping-impl：主应用工程，实现应用核心逻辑



关注一下现面几个重点类：

* + PingModule：createInstance方法是应用的初始化入口
  + PingProvider：实现应用功能逻辑，start方法用来完成初始化
* ping-features：生成karaf feature的工程，声明应用提供的feature和依赖
* ping-artifacts：包含应用提供的所有工程依赖，用来保证版本依赖一致性
* ping-karaf：一个karaf打包工程，生成控制器的运行环境，包含当前APP和依赖的feature集
* ping-it：生成集成测试工程

## Yang建模

MD-SAL使用Yang语言进行建模，Yang建模主要有如下几种：

* typedef：数据类型定义，通过typedef实现数据类型的扩展
* container/list/leaf/leaf-list：数据结构定义
* rpc：功能定义
* notification：消息定义
* augment：扩展数据结构

有关Yang的详细语法说明，请参考文档《YANG入门介绍.docx》，这里只对部分进行说明。

### 模型声明

一个完整的模型module声明包括yang-version, namespace，revision，prefix，其它可选信息有organization，contact，description

module opendaylight-inventory {

namespace "urn:opendaylight:inventory";

prefix inv;

organization "Netconf Central";

contact

"Andy Bierman <andy@netconfcentral.org>";

description

"YANG version of the TOASTER-MIB.";

revision "2013-08-19" {

description "Initial revision of Inventory model";

}

其中，namespace和revision在生成Java API时，会转化成包名；revision在应用开发时，要根据自己的设计时间进行修改。

### 数据建模

MD-SAL的数据库有两种类型，

* config：表示可配置数据，保存在配置数据库后，会被持久化，重启不会丢失；
* operational：表示运行时数据，保存到运行库，重启会丢失。

一个YANG数据定义如下：

module opendaylight-inventory

namespace "urn:opendaylight:inventory";

prefix inv;

container nodes {

description "The root container of all nodes.";

list node {

key "id";

description "A list of nodes (as defined by the 'grouping node').";

...

}

}

其中，

module标记一个Yang模型单元；

container定义一个数据节点，包含多个的node节点；

key表示list下节点的ID

config子句用来指定是配置数据还是运行时数据，config为true表示配置数据，为false表示运行时数据。

**上面的Yang定义了一个以nodes为根的数据节点树，在MD-SAL中，使用树形结构对数据进行存储管理。在读写数据时，需要指定数据节点的路径位置，这个路径我们使用InstanceIdentifier表示。InstanceIdentifier有多种形式：**

* XML/JSON-形式：

/inv:nodes/inv:node[inv:id='foo']

* Binding-形式(Java)

import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.Nodes;

import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.nodes.Node;

import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.nodes.NodeKey;

import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;

InstanceIdentifier<Node> identifier = InstanceIdentifier.builder(Nodes.class).child(Node.class,new NodeKey("foo")).build();

* REST API路径

http://localhost:8080/restconf/operational/opendaylight-inventory:nodes/node/foo

### Rpc

rpc可以用来表示应用提供的功能，如下Yang语句，

module sal-flow {

...

rpc add-flow {

input {

}

output {

...

}

}

}

定义了一个添加流表项的Rpc，在生成代码阶段会映射成一个Java方法

Future<RpcResult<AddFlowOutput >> addFlow(AddFlowInput input)

有两种类型的rpc

* Global：全局只有一个服务实例，上面的Yang定义的就是全局rpc
* Routed：全局有多个服务实例，多个实例通过不同的上下文实例进行标记

声明Routed Rpc包含两个步骤：

* + 声明Rpc上下文实体

identity node-context {

description "A node-context is a classifier for node elements which allows an RPC to provide a service on behalf of a particular element in the data tree.";

}

* + 声明Rpc上下文实例

声明Rpc上下文实例，即对应到数据树上的某个路径。

声明需要导入yang-ext，

**import yang-ext {prefix ext; revision-date "2013-07-09";}**

按下面语法进行声明

**ext:context-instance *identity\_name***

其中，identity\_name是rpc上下文实体的名称，所下所示：

container nodes {

description "The root container of all nodes.";

list node {

key "id";

description "A list of nodes (as defined by the 'grouping node').";

ext:context-instance "node-context";

...

}

}

* + 声明Rpc为Routed Rpc

同上，也需要导入yang-ext，使用下面语法进行声明

**ext:context-reference *identity\_name***

其中，identity\_name是rpc上下文实体的名称，所下所示：

typedef node-ref {

type instance-identifier;

description "A reference that points to an opendaylight-light:nodes/node in the data tree.";

}

grouping node-context-ref {

description

"A helper grouping which contains a reference to a node classified with a node-context. This allows RPCs in other yang files to refine their input to a particular node instance.";

leaf node {

ext:context-reference "node-context";

type node-ref;

description "A reference to a particular node.";

}

}

rpc add-flow {

description "Adding flow to openflow device.";

input {

uses node-context-ref;

...

}

output {

...

}

}

ext:context-reference需要使用在某个leaf节点，调用rpc时，通过该节点传递上下文实例参数。

### Notification

Notification用来对事件或消息进行建模，使用**notification** 语句：

module sal-flow {

notification flow-removed {

...

}

}

在生成Java API后，上述定义会生成消息侦听接口SalFlowListener

public interface SalFlowListener extends NotificationListener {

v

### Yang建模规范

* 统一命名规范：
  + 大小写：

除定义枚举的常量使用大写外，其它一律使用小写

* + 分隔符

rpc/notification/数据节点的名称，多个单词之间使用减号进行分隔，

例如：node-connector/add-flow/flow-removed等

* 命名空间：

***urn:rgonc:model:app:classifier***

app – 应用名称

classifier – 应用分类名称，可以只有一级

例如：

namespace "urn:rgonc:model:flow:statistics";

* rpc命名：动词+名词的方式

rpc add-device {

}

* notification命名：形容词或者名词+动词的过去式

notification device-connected {

}

* 多使用description

使用description语句增加注释，提高可读性

* 层次分明

避免一个语句单元占用过多行，难以理解。多使用grouping子句进行分层。

* 避免重名：

对于相同内容的数据，要求使用grouping语句进行复用，避免生成重名的接口/类

* 利用已有的建模：

对一些常见的模型，例如ip/mac等类型，首先查询ietf标准模型是否有定义，优先引入已知的模型。

## Java API

Yang建模完成后，借助Yangtools提供的yang-maven-plugin可生成Java API，关于Yang与Java API的映射关系请参考《YangTools使用指南.docx》。

这里，重点说明一下几个关键接口。

每个module在映射生Java API时，会有两个关键接口：

***ModuleNameService***和***ModuleNameListener***

例如：下面的Yang文件，

module sal-flow {

notification flow-removed {

...

}

rpc add-flow {

input {

}

output {

...

}

}

}

会生成SalFlowService和SalFlowListener。

public interface SalFlowListener extends NotificationListener {

void onFlowRemoved(FlowRemoved notification);

}

public interface SalFlowService extends RpcService {

Future<RpcResult<AddFlowOutput>> addFlow(AddFlowInput input);

}

## Rest API

在Yang建模后，同时就得到了相应的Rest API，有两部分data和rpc。

### data

在Yang建模完成数据树定义后，RestConf可自动生成对数据树的增、删、改、查API，每个操作分别对应HTTP的POST、DELETE、PUT、GET方法，细化到每一层节点，同时由于支持config库和operational库(只能查询)。

例如：一个nodes/node/的数据树，会有以下的相应的API：

* config

POST /config/

POST /config/opendaylight-inventory:nodes/

POST /config/opendaylight-inventory:nodes/node/{id}/

DELETE /config/opendaylight-inventory:nodes/

DELETE /config/opendaylight-inventory:nodes/node/{id}/

PUT /config/opendaylight-inventory:nodes/

PUT /config/opendaylight-inventory:nodes/node/{id}/

GET /config/opendaylight-inventory:nodes/

GET /config/opendaylight-inventory:nodes/node/{id}/

* operational

GET /operational/opendaylight-inventory:nodes/

GET /operational/opendaylight-inventory:nodes/node/{id}/

其中，URL前缀都是**http://*ip*:*port***

### rpc

rpc定义后，同样对应有相应的REST API，格式如下

* 请求方法：POST
* 请求路径：

**http://*ip*:*port*/restconf/operations/*module\_name*:*rpc\_name***

其中，module\_name指的是声明rpc的module名称；rpc\_name是rpc的名称，例如，

http://localhost:8080/restconf/operations/sal-flow:add-flow

* 请求内容：

XML或JSON，以JSON为例，形如

{

"input" : {

...

}

}

需要注意的是，Routed Rpc需要在input参数，包含上下文实例，例如：

{

"input" : {

"node" : "/inv:nodes/inv:node[inv:id='foo']",

...

}

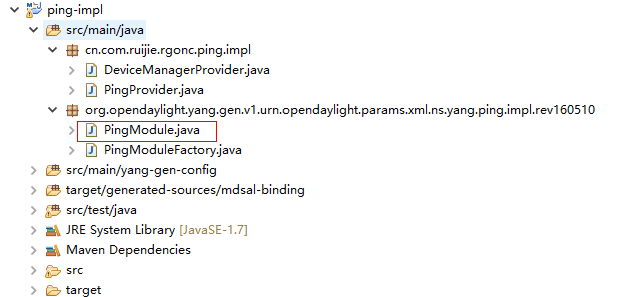
}

## 功能实现

在这一步，开发者通过生成的JAVA API实现应用的功能。在这一节，我们对开发中普遍的问题点进行描述，主要描述应用同MD-SAL之间的联系。

### 初始化

应用初始化，由Config Subsystem实现，在代码骨架中，已实现初始化的入口，



编辑PingModule，实现createInstance方法，完成应用初始化。

初始化时以下几个步骤：

* 创建应用功能实现类：

PingProvider pingProvider = new PingProvider();

* 设置依赖的服务：

依赖的服务通过getXXXXDependency()获取

pingProvider.setDataBroker(getDataBrokerDependency()); pingProvider.setRpcProviderRegistry(getRpcRegistryDependency());

...

* 应用功能实现类初始化

pingProvider.start();

这个初始化，主要包括：

* + 注册Rpc
  + 获取其它应用的Rpc
  + 注册消息侦听器
  + 注册DataChange侦听器
* 返回AutoCloseable对象

这一步要求返回一个实现AutoCloseable接口的对象，用来销毁相关资源。

### MD-SAL服务获取

MD-SAL的服务接口，依赖sal-binding-api工程，使用时，需要添加相关maven依赖

<dependency>

<groupId>org.opendaylight.controller</groupId>

<artifactId>sal-binding-api</artifactId>

<version> 1.3.2-Beryllium-SR2</version>

</dependency>

MD-SAL提供服务，这些服务有(部分服务接口不建议使用未给出)：

* 消息订阅服务：

org.opendaylight.controller.md.sal.binding.api.NotificationService

当需要侦听消息时，使用该服务。

**需要注意的是，还有另外一个同名的标记为@deprecated的NotificationService接口位于包org.opendaylight.controller.sal.binding.api下，这是氦版的保留接口，不建议使用，引入时注意不要出错。**

* 消息发布服服务：

org.opendaylight.controller.md.sal.binding.api.NotificationPublishService

包含三个方法：

|  |
| --- |
| **putNotification** |
| void putNotification(Notification notification)  throws InterruptedException |
| 将消息异步发送给订阅者，当消息队列满时，会阻塞等待 |
| **参数：** |
| notification – 消息 |
| **返回值：** |
| 无 |
| **offerNotification** |
| ListenableFuture<? extends Object>  offerNotification(Notification notification) |
| 将消息异步发送给订阅者，立即返回Future，保证不阻塞 |
| **参数：** |
| listener – 包含多个消息的侦听器 |
| **返回值：** |
| ListenableFuture – Future对象，可调用get方法进行同步等待，也可以异步。 |
| **offerNotification** |
| ListenableFuture<? extends Object>  offerNotification(  Notification notification ,  int timeout, TimeUnit unit)  throws InterruptedException |
| 将消息发送给订阅者，保证在指定的时间内不阻塞，如在指定时间还未完成，直接返回  Future对象 |
| **参数：** |
| notification – 消息  timeout – 超时时间  unit – 时间单位 |
| **返回值：** |
| ListenableFuture – Future对象，可调用get方法进行同步等待，也可以异步。 |

* Rpc注册服务：

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

当需要注册Rpc或者调用其它应用Rpc时，使用该服务

* 数据管理服务：

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

当需要存储数据、查询其它应用的数据、侦听数据的变化时，使用该服务。

**有两种方式，去获取MD-SAL服务**：

* 从BindingAwareBroker获取：

应用首先从Config Subsystem获取BindingAwareBroker服务，同时自己实现一个BindingAwareProvider接口，然后注册给BindingAwareBroker，然后在回调方法里获取MD-SAL服务，如下代码片段：

public class HelloProvider implements BindingAwareProvider {

@Override

public void onSessionInitiated(ProviderContext session) {

DataBroker dataBroker = session.getSALService(DataBroker.class);

}

* 从Config Subsystem获取：

所有MD-SAL提供的服务都被Config Subsystem托管，因此，可以不经过BindingAwareBroker，直接从Config Subsystem获取服务，具体参考《Config应用建模》文档。下面是ConfigSubsystem程序入口，获取服务的代码片段：

public java.lang.AutoCloseable createInstance() {

PingProvider provider = new PingProvider();

provider.setDataBroker(getDataBrokerDependency());

provider.setRpcProviderRegistry(getRpcRegistryDependency());

provider.setNotificationPublishService(getNotificationPublishServiceDependency());

provider.setNotificationService(getNotificationServiceDependency());

provider.start();

return provider;

}

这里推荐使用后一种，因为第一种无法得到全部的服务(不包含集群服务)。

### 消息订阅与发布

#### 消息订阅

实现消息订阅，包含以下几个步骤：

1. 实现消息侦听器接口

如下所示：

public class SalFlowListenerImpl implements SalFlowListener {

public void onFlowRemoved(FlowRemoved notification) {

...

}

}

1. 获取消息订阅服务NotificationService

获取方式见上一节描述。

1. 注册消息侦听器

SalFlowListenerImpl listener = new SalFlowListenerImpl();

notificationService.registerListener(listener);

#### 消息发布

消息发布比较简单，主要是包含以下几个步骤：

1. 获取NotificationPublishService
2. 构造消息并发布

FlowRemovedBuilder frb = new FlowRemovedBuilder();

...

notificationPublishService.putNotification(frb.build());

### Rpc

#### Rpc实现与注册

* Rpc实现：

SalFlowServiceImpl.java:

public class SalFlowServiceImpl implements SalFlowService {

@Override

public Future<RpcResult<addFlowOutput>> addFlow(AddFlowInput input) {

/\* construct addFlowOutput \*/

return RpcResultBuilder

.success(addFlowOutput)

.buildFuture();

// eg:Rpc实现：

addFlowOutputBuilder addBuilder = new addFlowOutputBuilder();

addBuilder.setGreeting(“ ”+input.getName());

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

}

}

* Global Rpc注册：

SalFlowServiceProvider.java:

eg:

private RpcRegistration<SalFlowService> reg;

**@Override**

public void onSessionInitiated(ProviderContext session) {

LOG.info("SalFlowServiceProvider Session Initiated");

SalFlowServiceImpl ss = new SalFlowServiceImpl();

reg = session.addRpcImplementation(SalFlowService.class,ss);

}

HelloWorldImpl impl = new HelloWorldImpl();

RpcRegistration<?> reg = rpcProviderRegistry.addRpcImplementation(HelloService.class, impl);

* Routed Rpc注册：

Routed Rpc注册稍微复杂一些，还需要注册Rpc上下文实例，如下所示：

HelloWorldImpl impl = new HelloWorldImpl();

RoutedRpcRegistration<?> reg rpcProviderRegistry.addRoutedRpcImplementation(HelloService.class, impl);

// struct path

InstanceIdentifier path = ...

reg.registerPath(NodeContext.class, path);

#### Rpc获取与调用

* Rpc获取：

SalFlowService salFlowService = rpcProviderRegistry.getRpcService(SalFlowService.class);

* Rpc调用：

// contruct addFlowInput

Future<RpcResult<AddFlowOutput>> future;

// 异步调用

future = salFlowService.addFlow(addFlowInput)

// 同步调用

future = salFlowService.addFlow(addFlowInput);

AddFlowOutput ouput = future.get().getResult();

### 数据存储实现

MD-SAL提供了统一的DataBroker进行数据存储。在开始一次读写前都要先创建一个事务。

* 读数据

ReadOnlyTransaction transaction = dataBroker.newReadWriteTransaction();

Optional<Node> nodeOptional;

InstanceIdentifier<Node> nodePath = ...; // contruct path

nodeOptional = transaction.read(

LogicalDataStore.OPERATIONAL,

nodePath).checkedGet();

* 写数据

ReadWriteTransaction transaction = dataBroker.newReadWriteTransaction();

InstanceIdentifier<Node> nodePath = ...; // construct path

NodeBuilder nb = new NodeBuilder();

... // contruct node

transaction.put(

LogicalDataStore.CONFIGURATION,

nodePath,

nb.build());

transaction.delete(

LogicalDataStore.CONFIGURATION,

nodePath);

CheckedFuture<Void,TransactionCommitFailedException> future;

future = transaction.submit();

* 注册数据变化侦听

需要指定侦听的数据库(配置库还是运行库)，侦听的数据路径，以及数据变化的范围。如下所示：

DataChangeListener listenerInstance = new DataChangeListener();

ListenerRegistration<DataChangeListener> dataChangeReg =

dataBroker.registerDataChangeListener(

LogicalDatastoreType.CONFIGURATION, devicePath, listenerInstance, DataChangeScope.BASE);

其中，数据变化范围有三个取值：

BASE – 只关心订阅路径上的节点的变更

DIRECT – 只关心订阅路径上的节点和下一级子节点的更

SUBSTREE – 关心所有子节点、子孙节点的变更

### 集群实现

待补充

## feature声明

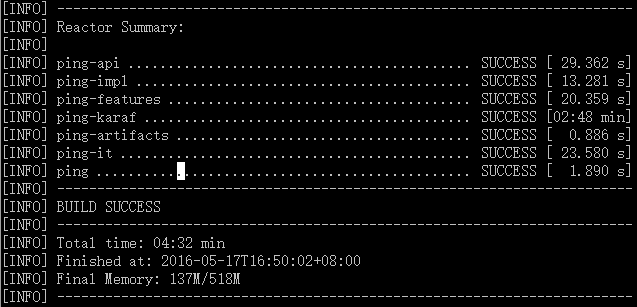
关于feature的定义，请具体参考karaf相关文档，开发中，我们只需拷贝一份feature.xml进行修改，遵循相应的使用规范即可。

## 构建控制器版本

按照代码生成框架，会生成一个karaf工程，用来构建控制器版本。只要在在根目录，执行一次构建：

mvn clean install –DskipTests –Dcheckstyle.skip

待构建成功后，



从下面路径找到版本包：

karaf/target/ping-karaf-1.0.0-SNAPSHOT.tar.gz

## 控制器部署及运行

* 运行：

目前只完整支持Linux系统，其它系统未充分测试

将发布版本上传至Linux服务器，使用

tar -zxvf *ping-karaf-1.0.0-SNAPSHOT.*tar.gz

进入目录：

cd ping-karaf-1.0.0-SNAPSHOT

./bin/start [debug]

debug可以支持远程调试

* 进入控制台

./bin/client –u karaf –h *ip*

退出karaf控制台，执行logout

* 关闭：

karaf控制台执行shutdown -f

或者通过ps –ef|grep java找到java进程，使用kill -9 pid杀掉进程。

* 控制器重启：

按上面方式，即先关闭，再启动控制器即可

* JAR包增量部署

在开发中，经常碰到需要修改代码，重新构建Bundle的情况，但每次都完整构建一个控制器版本效率非常低下。这里，我们借用karaf的热部署机制进行增量部署，即只构建有修改的工程，在工程的target目录下，找到构建后的JAR包，上传到控制器的运行目录下的deploy文件夹(ping-karaf-1.0.0-SNAPSHOT/deploy)。

由于ODL无法支持在线热部署，我们需要重启控制器，才能完成增量部署。

# 应用开发DEMO

在本节中，以一个实际用例Ping，一步步演示如何开发一个MD-SAL应用。在这个例子中，我们实现一个简单的设备管理，设备通过ping是否可达进行发现。

功能设计如下：

* device-manager：
  + 支持创建、删除设备
  + 接收ping可达的消息，更新设备状态为在线；
  + 接收ping不可达的消息，更新设备状态为离线
* ping-discovery：
  + 侦听设备的创建消息，触发一次ping发现，保存到ping会话
  + ping保活：隔30秒进行一次ping保活，当ping会话状态发生变更时，进行通知
  + 在设备删除后，删除相应的ping会话信息
* **ping延时统计：**
  + **保存最近10次的ping延时数据**
  + **获取设备的ping平均延时**

**（未完成功能，分组完成，相互评审）**

## 生成代码骨架

见2.1。

## Yang建模

本例中，有两个相对独立的功能，我们使用两个Yang文件进行建模。

### device-manager建模

在ping-api：src/main/yang新建rgonc-device.yang，添加如下内容：

module rgonc-device {

yang-version 1;

namespace "urn:rgonc:model:device";

prefix "pinv";

import ietf-inet-types {prefix inet; revision-date "2010-09-24";}

revision "2016-05-10" {

description "Initial revision of rgonc-device model";

}

typedef online-status {

type enumeration {

enum ON\_LINE {

value 1;

}

enum OFF\_LINE {

value 2;

}

}

}

container devices {

config true;

description "Top-level container for all devices.";

list device {

key "ip";

leaf ip {

type inet:ipv4-address;

}

leaf online-status {

config false;

type online-status;

}

}

}

rpc add-device {

input {

leaf ip {

type inet:ipv4-address;

}

leaf name {

type string;

}

}

output {

}

}

rpc remove-device {

input {

leaf ip {

type inet:ipv4-address;

}

}

output {

}

}

}

这里，因为使用了ip地址，引入标准的yang模型ietf-inet-types，需要增加相应的maven依赖。

打开ping-api的工程pom.xml，添加如下依赖：

<dependencies>

<dependency>

<groupId>org.opendaylight.mdsal.model</groupId>

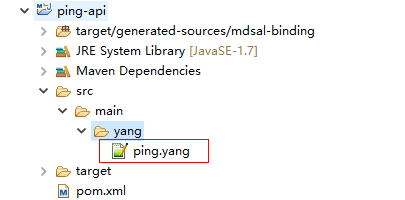
<artifactId>ietf-inet-types</artifactId>

</dependency>

</dependencies>

### ping-discovery建模

编辑ping.yang，



复制下面的内容，

module ping {

yang-version 1;

namespace "urn:rgonc:model:ping";

prefix "ping";

import ietf-inet-types {prefix inet; revision-date "2010-09-24";}

revision "2016-05-10" {

description "Initial revision of ping model";

}

container sessions {

config true;

description "Top-level container for all ping sessions.";

list session {

key "ip";

leaf ip {

type inet:ipv4-address;

}

leaf reachable {

type boolean;

}

}

}

notification ip-ping-reachable {

leaf ip-address {

type inet:ipv4-address;

}

}

notification ip-ping-unreachable {

leaf ip-address {

type inet:ipv4-address;

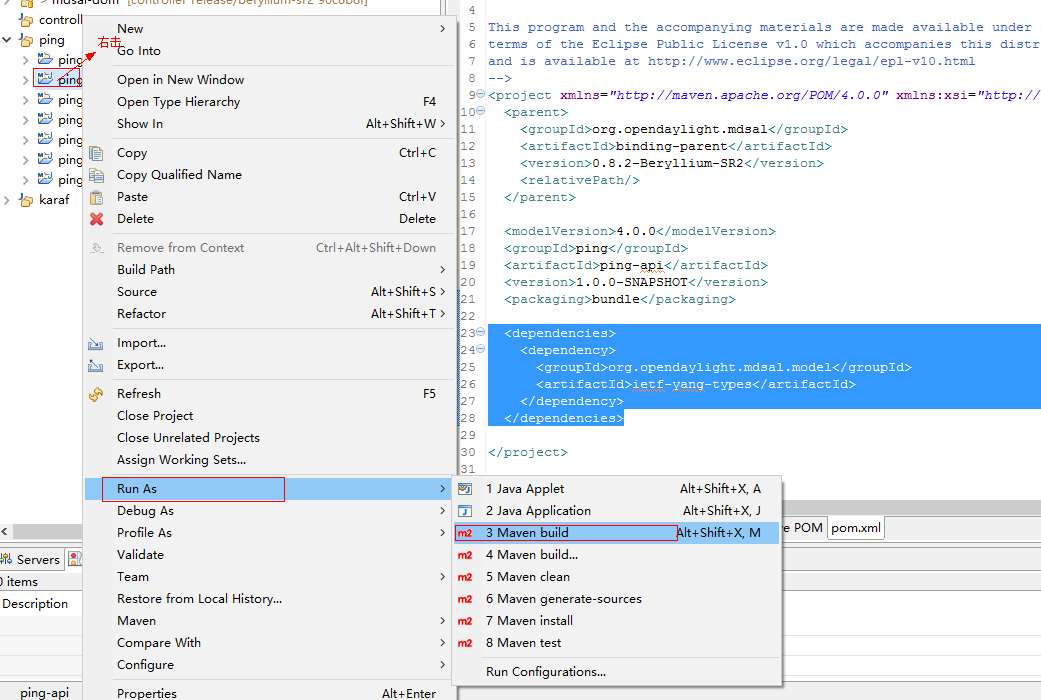
}

}

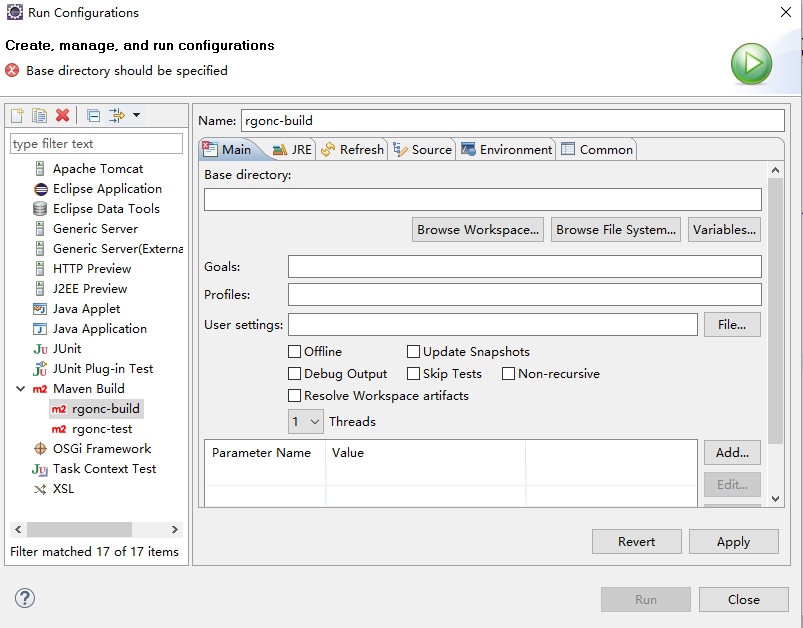
}

## 生成API

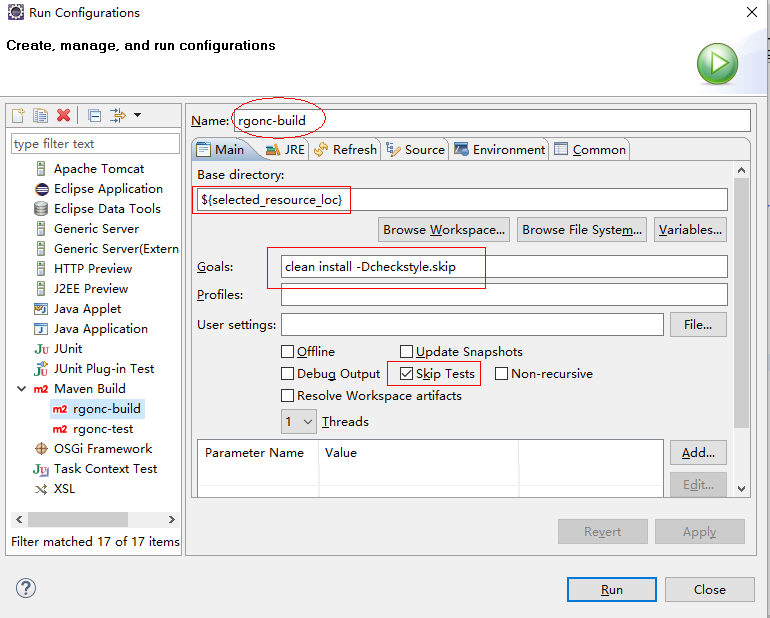
编译ping-api工程，在eclipse右击ping-api工程，依次选择Run As >= Maven build



如果弹出，如下界面，表示没有添加相应的构建目标，

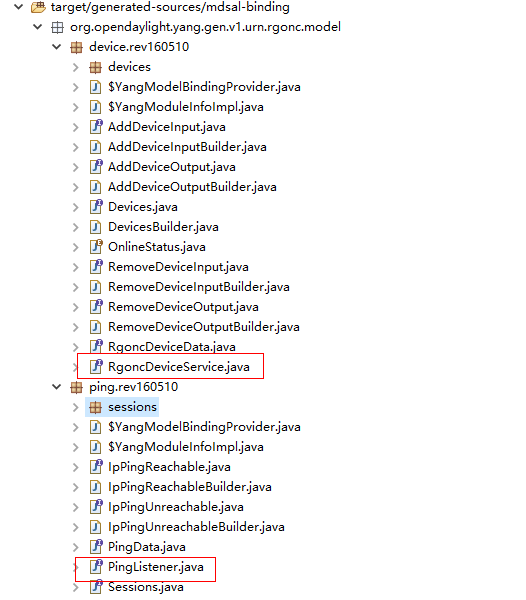


按下图方式填写：



点击Apply，然后Run

编译完成后，编译成功后，可以在target/generated-sources/mdsal-binding目录看到生成的代码。



其中，

RgoncDeviceService – 表示设备管理核心功能接口，包含创建、删除设备的逻辑

PingListener – 表示ping可达/不可达消息侦听接口

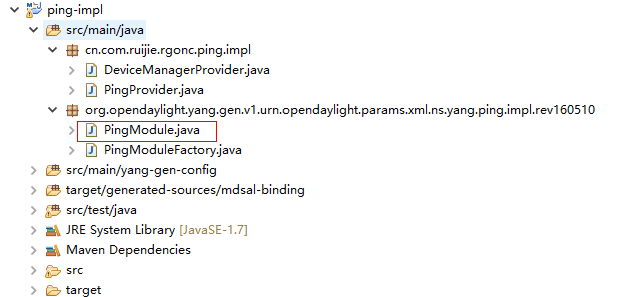
## 功能实现

按功能设计分成两个实现类：

* DeviceManagerProvider：
  + 实现RgoncDeviceService接口和PingListener接口
  + 设备配置保存在配置库
  + 设备状态保存到运行库
* PingProvider：
  + 侦听设备树devices/device变更
  + ping-discover功能实现
  + ping会话保活

### 初始化

打开PingModule.java，



编辑PingModule，实现createInstance方法，完成应用初始化。

默认已经提供了PingProvider，在本例中，还需要一个DeviceManagerProvider，最后需要对PingProvider和DeviceManagerProvider进行关闭。

public java.lang.AutoCloseable createInstance() {

// DeviceManager start

final DeviceManagerProvider deviceProvider = new DeviceManagerProvider();

deviceProvider.setDataBroker(getDataBrokerDependency());

deviceProvider.setRpcProviderRegistry(getRpcRegistryDependency());

deviceProvider

.setNotificationService(getNotificationServiceDependency());

deviceProvider.start();

// Ping-Discover start

final PingProvider pingProvider = new PingProvider();

pingProvider.setDataBroker(getDataBrokerDependency());

pingProvider.setRpcProviderRegistry(getRpcRegistryDependency());

pingProvider.setNotificationPublishService(

getNotificationPublishServiceDependency());

pingProvider.setNotificationService(getNotificationServiceDependency());

pingProvider.setEntityOwnershipService(

getEntityOwnershipServiceDependency());

pingProvider.start();

return new AutoCloseable() {

@Override

public void close() throws Exception {

pingProvider.close();

deviceProvider.close();

}

};

}

### DeviceManagerProvider实现

1. 类声明

DeviceManagerProvider需要实现RgoncDeviceService、PingListener、AutoCloseable三个接口：

public class DeviceManagerProvider

implements RgoncDeviceService, PingListener, AutoCloseable {

1. **设置依赖的服务**

**依赖的MD-SAL服务都作为成员，提供设置的方法。**

private DataBroker dataBroker;

private RpcProviderRegistry rpcRegistry;

private NotificationService notificationService;

public void setRpcProviderRegistry(RpcProviderRegistry rpcRegistry) {

this.rpcRegistry = rpcRegistry;

}

public void setDataBroker(DataBroker dataBroker) {

this.dataBroker = dataBroker;

}

public void setNotificationService(NotificationService notificationService) {

this.notificationService = notificationService;

}

1. start

start方法用来完成初始化：这里需要注册Rpc、同时注册PingListener侦听

private RpcRegistration<?> rpcReg;

private ListenerRegistration<?> listenerReg;

public void start() {

this.rpcReg = this.rpcRegistry

.addRpcImplementation(RgoncDeviceService.class, this);

listenerReg = this.notificationService

.registerNotificationListener(this);

LOG.info("RG-ONC-DeviceManager has started");

}

1. close

close用来清理资源，包括关闭注册以及其它线程资源等等。

public void close() {

if (rpcReg != null) {

rpcReg.close();

}

if (listenerReg != null) {

listenerReg.close();

}

}

1. createDevicePath

创建设备路径

public static InstanceIdentifier<Device> createDevicePath(Ipv4Address ip) {

return InstanceIdentifier.builder(Devices.class)

.child(Device.class, new DeviceKey(ip)).build();

}

1. add-device

添加设备时，首先对参数检查，同时判断ip是否冲突，检查通过直接保存到DataBroker

@Override

public Future<RpcResult<AddDeviceOutput>> addDevice(AddDeviceInput input) {

Ipv4Address ip = input.getIp();

if (ip == null) {

return RpcResultBuilder.<AddDeviceOutput> failed().buildFuture();

}

InstanceIdentifier<Device> path = createDevicePath(ip);

Device device = new DeviceBuilder().setIp(ip).build();

ReadWriteTransaction tx = this.dataBroker.newReadWriteTransaction();

try {

Optional<Device> deviceOptional = tx

.read(LogicalDatastoreType.CONFIGURATION, path).get();

if (deviceOptional.isPresent()) {

return RpcResultBuilder.<AddDeviceOutput> failed()

.withError(ErrorType.APPLICATION, "ip aready exists")

.buildFuture();

}

tx.put(LogicalDatastoreType.CONFIGURATION, path, device, true);

tx.submit().checkedGet();

} catch (Exception e1) {

return RpcResultBuilder.<AddDeviceOutput> failed()

.withError(ErrorType.APPLICATION, "exception happen")

.buildFuture();

} finally {

tx.cancel();

}

return RpcResultBuilder.<AddDeviceOutput> success().buildFuture();

}

1. remove-device

删除设备前同时做下设备是否存在的检查，检查通过后直接删除数据，同时删除配置库和运行库。

public Future<RpcResult<RemoveDeviceOutput>> removeDevice(

RemoveDeviceInput input) {

Ipv4Address ip = input.getIp();

if (input.getIp() == null) {

return RpcResultBuilder.<RemoveDeviceOutput> failed()

.buildFuture();

}

InstanceIdentifier<Device> path = createDevicePath(ip);

ReadWriteTransaction tx = this.dataBroker.newReadWriteTransaction();

try {

Optional<Device> deviceOptional = tx

.read(LogicalDatastoreType.CONFIGURATION, path).get();

if (!deviceOptional.isPresent()) {

return RpcResultBuilder.<RemoveDeviceOutput> failed()

.withError(ErrorType.APPLICATION,

"device not found")

.buildFuture();

}

// delete operational data

Optional<Device> deviceStatOpt = tx

.read(LogicalDatastoreType.OPERATIONAL, path).get();

if (deviceStatOpt.isPresent()) {

tx.delete(LogicalDatastoreType.OPERATIONAL, path);

}

tx.delete(LogicalDatastoreType.CONFIGURATION, path);

tx.submit().checkedGet();

} catch (Exception e1) {

return RpcResultBuilder.<RemoveDeviceOutput> failed()

.withError(ErrorType.APPLICATION, "exception happen")

.buildFuture();

} finally {

tx.cancel();

}

return RpcResultBuilder.<RemoveDeviceOutput> success().buildFuture();

}

1. PingListener实现

@Override

public void onIpPingReachable(IpPingReachable notification) {

onIpPingEvent(notification.getIp(), true);

}

@Override

public void onIpPingUnreachable(IpPingUnreachable notification) {

onIpPingEvent(notification.getIp(), false);

}

public void onIpPingEvent(Ipv4Address ip, boolean online) {

OnlineStatus status = online ? OnlineStatus.ONLINE

: OnlineStatus.OFFLINE;

DeviceBuilder db = new DeviceBuilder();

db.setIp(ip);

db.setOnlineStatus(status);

InstanceIdentifier<Device> path = createDevicePath(ip);

WriteTransaction tx = this.dataBroker.newWriteOnlyTransaction();

try {

tx.put(LogicalDatastoreType.OPERATIONAL, path, db.build());

tx.submit().checkedGet();

} catch (Exception e) {

LOG.error("", e);

} finally {

tx.cancel();

}

}

### PingProvider实现

PingProvider需要实现DataChange、Runnable、AutoCloseable三个接口，其中Runnable用来实现ping保活线程调度。

1. 类声明

public class PingProvider

implements DataChangeListener, Runnable, AutoCloseable

1. 设置依赖的服务

依赖的MD-SAL服务都作为成员，提供设置的方法。

private DataBroker dataBroker;

private NotificationService notificationService;

private NotificationPublishService notificationPublishService;

private RpcProviderRegistry rpcProviderRegistry;

private EntityOwnershipService entityOwnershipService;

public void setRpcProviderRegistry(

RpcProviderRegistry rpcProviderRegistry) {

this.rpcProviderRegistry = rpcProviderRegistry;

}

public void setDataBroker(DataBroker dataBroker) {

this.dataBroker = dataBroker;

}

public void setNotificationService(

NotificationService notificationService) {

this.notificationService = notificationService;

}

public void setNotificationPublishService(

NotificationPublishService notificationPublishService) {

this.notificationPublishService = notificationPublishService;

}

public void setEntityOwnershipService(

EntityOwnershipService entityOwnershipService) {

this.entityOwnershipService = entityOwnershipService;

}

1. start

start方法完成初始化：包括注册devices/device的数据变化侦听 、ping保活定时器初始化

private ScheduledExecutorService scheduledExecutorService;

private ListenerRegistration<DataChangeListener> dataChangeReg;

public void start() {

InstanceIdentifier<Device> devicePath = InstanceIdentifier

.builder(Devices.class).child(Device.class).build();

dataChangeReg = this.dataBroker.registerDataChangeListener(

LogicalDatastoreType.CONFIGURATION, devicePath, this,

DataChangeScope.BASE);

scheduledExecutorService = Executors.newScheduledThreadPool(1);

scheduledExecutorService.scheduleAtFixedRate(this, 0,

PING\_TICKET\_INTERNAL, TimeUnit.MILLISECONDS);

LOG.info("PingProvider Stated");

}

1. close

close用来清理资源，包括关闭注册以及其它线程资源等等。

public void close() throws Exception {

if (dataChangeReg != null) {

dataChangeReg.close();

}

if (scheduledExecutorService != null) {

scheduledExecutorService.shutdown();

}

LOG.info("PingProvider Closed");

}

1. createSessionPath

创建sessions/session数据路径

public static InstanceIdentifier<Session> createSessionPath(

Ipv4Address ip) {

return InstanceIdentifier.builder(Sessions.class)

.child(Session.class, new SessionKey(ip)).build();

}

1. saveSession

保存ping会话信息，即将会话信息保存到DataBroker

public void saveSession(Ipv4Address ip, boolean reachable) {

InstanceIdentifier<Session> path = createSessionPath(ip);

SessionBuilder sb = new SessionBuilder();

sb.setIp(ip);

sb.setReachable(reachable);

WriteTransaction wtx = dataBroker.newWriteOnlyTransaction();

wtx.put(LogicalDatastoreType.OPERATIONAL, path, sb.build());

wtx.submit();

}

1. remove-session

删除ping会话，从数据库删除ping会话数据

public void removeSession(Ipv4Address ip) {

InstanceIdentifier<Session> path = createSessionPath(ip);

WriteTransaction wtx = dataBroker.newWriteOnlyTransaction();

wtx.delete(LogicalDatastoreType.OPERATIONAL, path);

wtx.submit();

}

1. pingDiscover

给定ip地址以及之前的旧会话状态，进行一次ping，如果可达状态发生改变，发布ping可达或不可达的消息，同时进行保存。

public void pingDiscover(Ipv4Address ip, Session oldSession) {

boolean reachable = isIpReachable(ip);

LOG.debug("ip={} reachable={}", ip, reachable);

if (oldSession == null || reachable != oldSession.isReachable()) {

LOG.info("ip={} reachable status change to {}", ip, reachable);

notifyIpPingResult(ip, reachable);

saveSession(ip, reachable);

}

}

private void notifyIpPingResult(Ipv4Address ip, boolean reachable) {

Notification notification;

if (reachable) {

notification = new IpPingReachableBuilder().setIp(ip).build();

} else {

notification = new IpPingUnreachableBuilder().setIp(ip).build();

}

try {

this.notificationPublishService.putNotification(notification);

} catch (InterruptedException e) {

LOG.error("", e);

}

}

1. onDataChange

当设备数据被添加或删除时，通知给用方法，

通过AsyncDataChangeEvent参数获取变更的设备信息。

设备新增时，触发一次ping发现

设备删除时，删除相应的ping会话信息

public void onDataChanged(

AsyncDataChangeEvent<InstanceIdentifier<?>, DataObject> change) {

// new device

for (Entry<InstanceIdentifier<?>, DataObject> entry : change

.getCreatedData().entrySet()) {

InstanceIdentifier<?> path = entry.getKey();

if (path.getTargetType().equals(Device.class)) {

Device device = (Device) entry.getValue();

pingDiscover(device.getIp());

}

}

// remove device

for (InstanceIdentifier<?> path : change.getRemovedPaths()) {

if (path.getTargetType().equals(Device.class)) {

DeviceKey key = path.firstKeyOf(Device.class);

removeSession(key.getIp());

}

}

}

1. run

定时器线程每隔30秒，调用一次run方法，进行所有会话的保活。

public void run() {

Sessions sessions = getSessions();

if (sessions == null) {

return;

}

for (Session session : sessions.getSession()) {

pingDiscover(session.getIp(), session);

}

}

public Sessions getSessions() {

InstanceIdentifier<Sessions> path = InstanceIdentifier

.builder(Sessions.class).build();

ReadTransaction rtx = dataBroker.newReadOnlyTransaction();

try {

Optional<Sessions> optional = rtx

.read(LogicalDatastoreType.OPERATIONAL, path).checkedGet();

if (optional.isPresent()) {

return optional.get();

}

} catch (ReadFailedException e) {

LOG.error("", e);

}

return null;

}

## feature声明

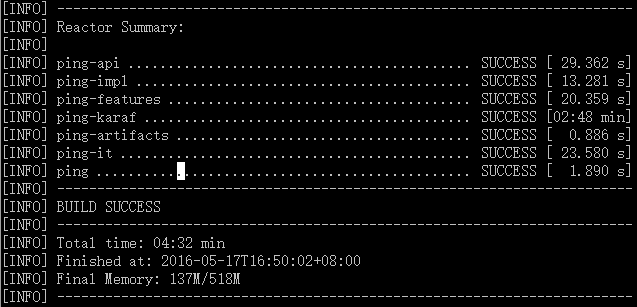
如果没有使用第三方的工具包或者应用，默认的feature声明就可以满足。关于feature使用参考karaf相关文档。

## 构建发布版本

在代码根目录，执行构建：

mvn clean install –DskipTests –Dcheckstyle.skip

待构建成功后，



从下面路径找到版本包：

karaf/target/ping-karaf-1.0.0-SNAPSHOT.tar.gz

## 控制器运行

见2.9

## 功能测试

1. 启动控制器，部署应用

控制器运行后，进入到控制台，执行feature:install rgonc-ping-ui部署应用。

执行log:tail cn.com.ruijie.rgonc.ping，直到打印下面日志，表示应用已完成初始化

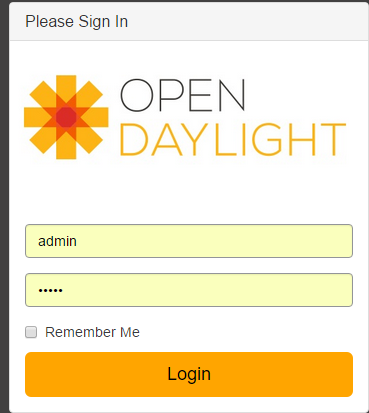
opendaylight-user@root>log:tail cn.com.ruijie.rgonc.ping

2016-05-21 08:53:03,574 | INFO | config-pusher | DeviceManagerProvider | 159 - cn.com.ruijie.rgonc.ping.impl - 1.0.0.SNAPSHOT | RG-ONC-DeviceManager has started

2016-05-21 08:53:03,670 | INFO | config-pusher | PingProvider | 159 - cn.com.ruijie.rgonc.ping.impl - 1.0.0.SNAPSHOT | PingProvider Stated

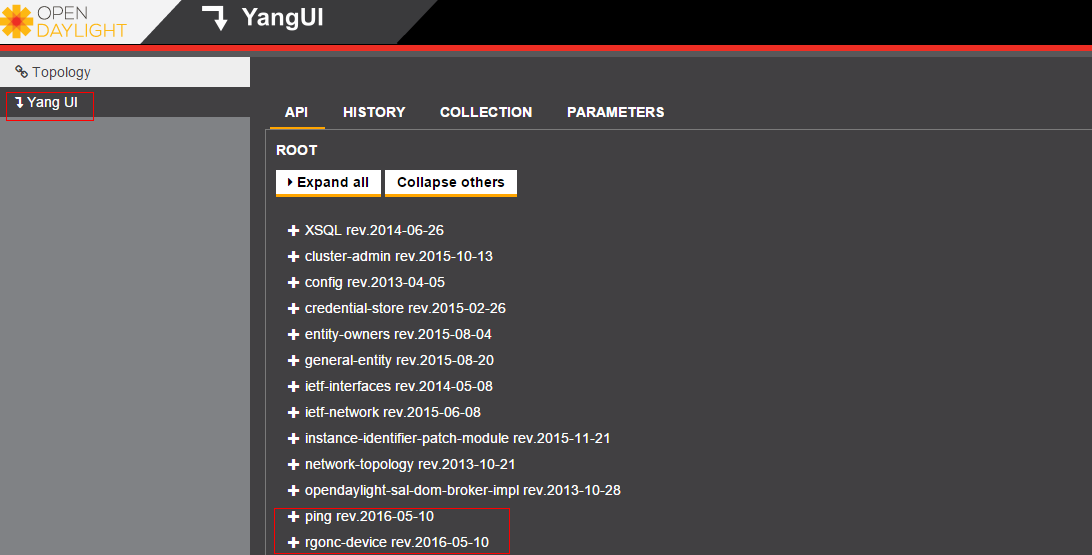
1. 访问YangUI

打开浏览器，访问地址：http://172.18.106.18:8/index.html



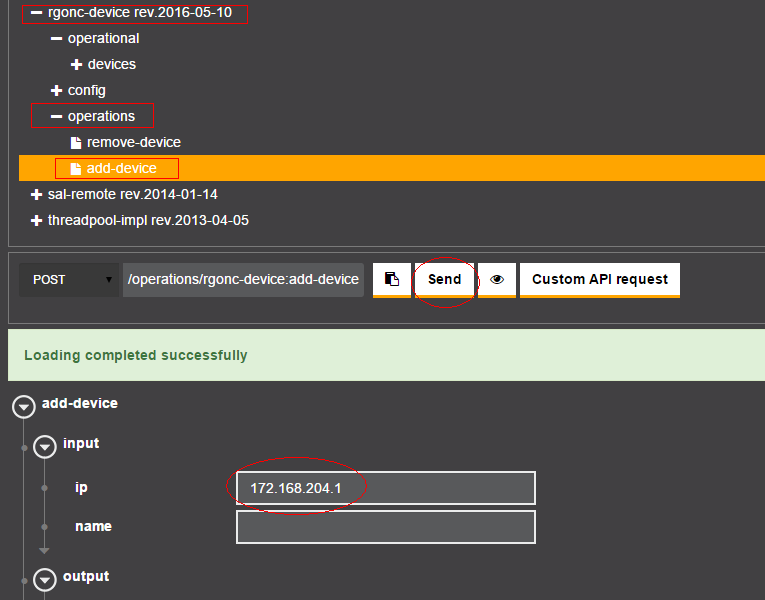
默认用户名密码：admin/admin

登录后，进入YangUI界面。

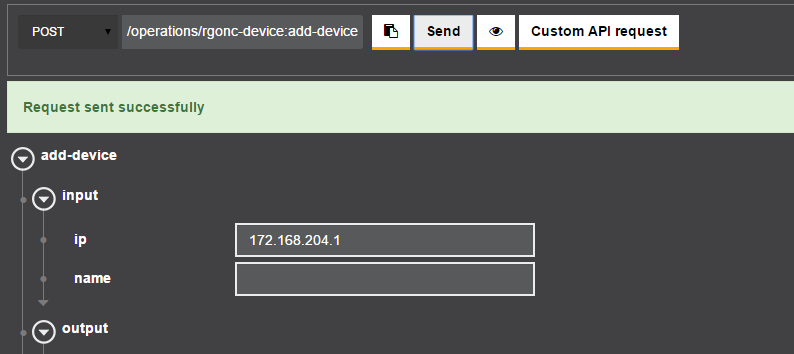


1. 创建设备172.168.204.1

在YangUI页面，点击rgonc-device，展开operations –> add-device



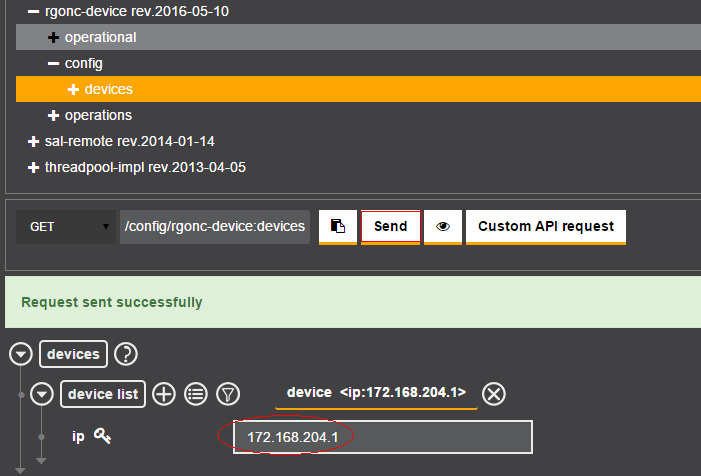
填写ip地址，点击Send



1. 查看设备配置

在YangUI页面，点击rgonc-device，展开config –> devices

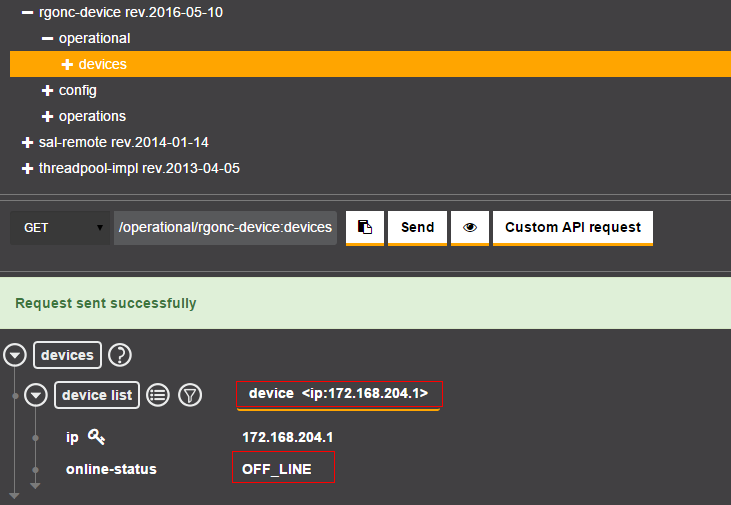
点击Send



1. 查看设备状态

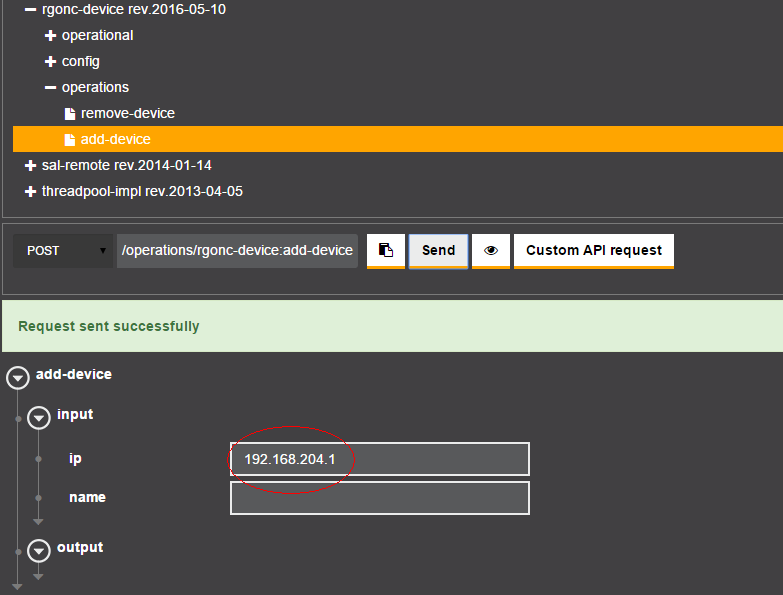
在YangUI页面，点击rgonc-device，展开operational –> devices

点击Send

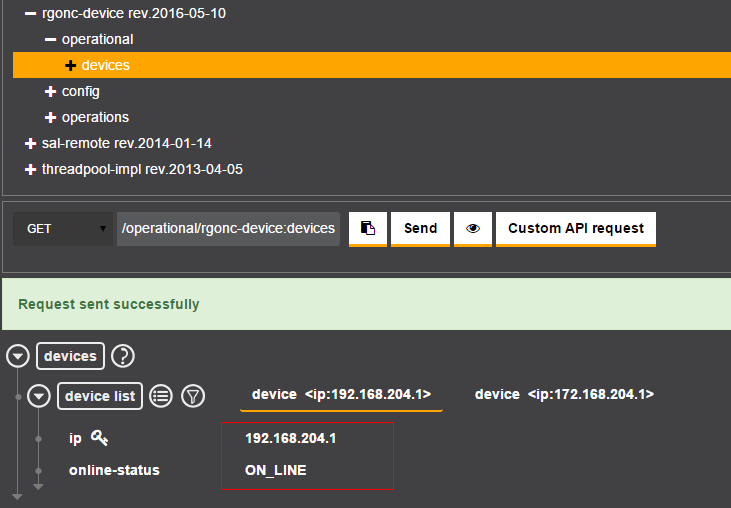


由于172.168.204.1 ping不可达，所有显示离线

1. 创建设备192.168.70.1



查看设备状态

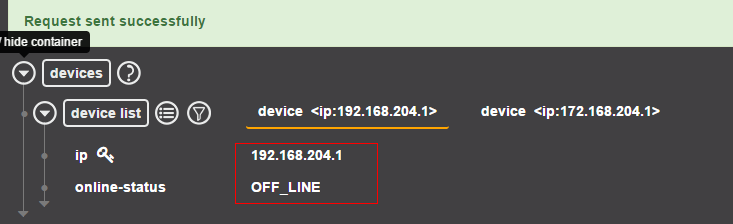


显示设备在线

修改控制器所有服务器的路由表，使192.168.204.1不可达

route add -net 192.168.204.0/24 gw 172.18.106.3

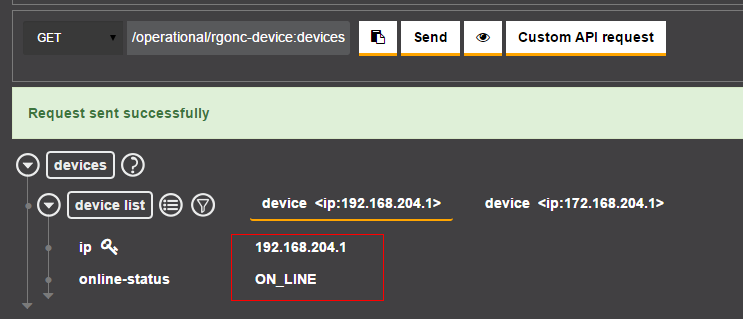
等待30s，再次看下设备状态



删除前面的路由配置，

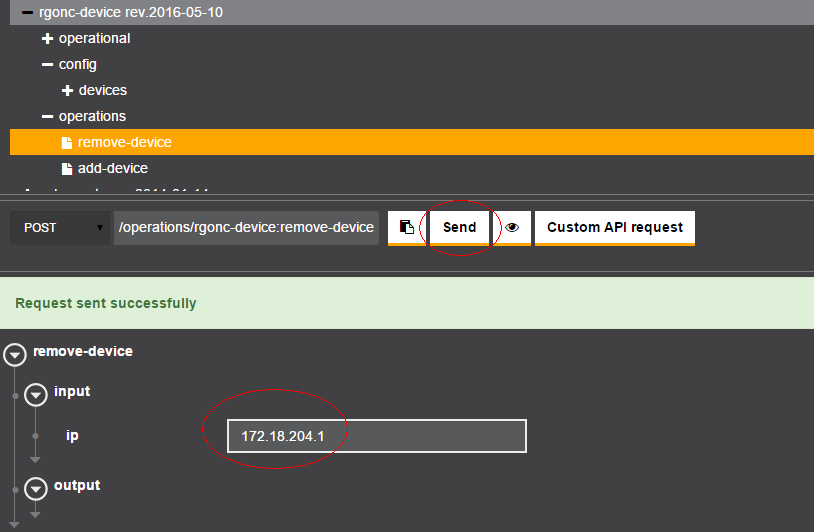
route del -net 192.168.204.0/24 gw 172.18.106.3

再等30s，看下设备状态又切换在线



1. 删除设备

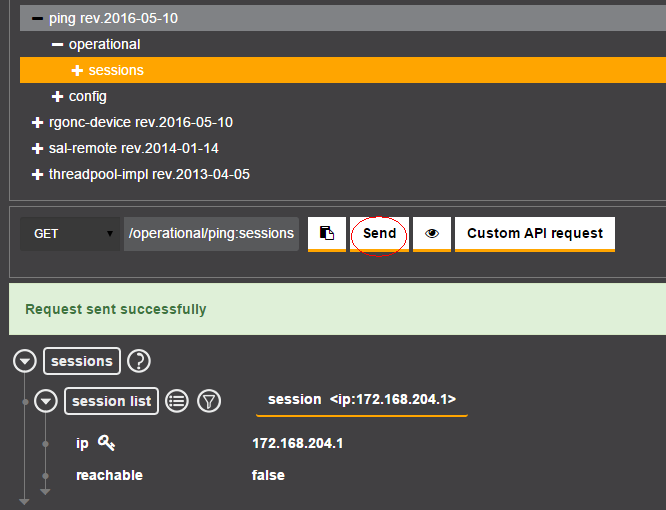
在YangUI页面，点击rgonc-device，展开operations –> remove-device



1. 查看ping会话

在YangUI页面，点击ping，展开operational –> sessions

点击Send



## Rest API

### 获取设备配置

HTTP Method: GET

HTTP URL: http://ip:8181/restconf/config/rgonc-device:devices

Header：Content-Type: application/json

### 获取设备状态

HTTP Method: GET

HTTP URL: http://ip:8181/restconf/o/rgonc-device:devices

Header：Content-Type: application/json

### 创建设备

HTTP Method: POST

HTTP URL: http://ip:8181/restconf/operations/rgonc-device:add-device

Header：Content-Type: application/json

Body：

{

"input": {

"ip" : "172.18.106.240"

}

}

### 删除设备

HTTP Method => POST

HTTP Method: POST

HTTP URL: http://ip:8181/restconf/operations/rgonc-device:remove-device

Header：Content-Type: application/json

Body：

{

"input": {

"ip" : "172.18.106.240"

}

}

### 获取ping会话信息

HTTP Method: GET

HTTP URL: http://ip:8181/restconf/config/ping:sessions

Header：Content-Type: application/json