在openstack各个项目中，我们通常会用到如下几种RPC请求

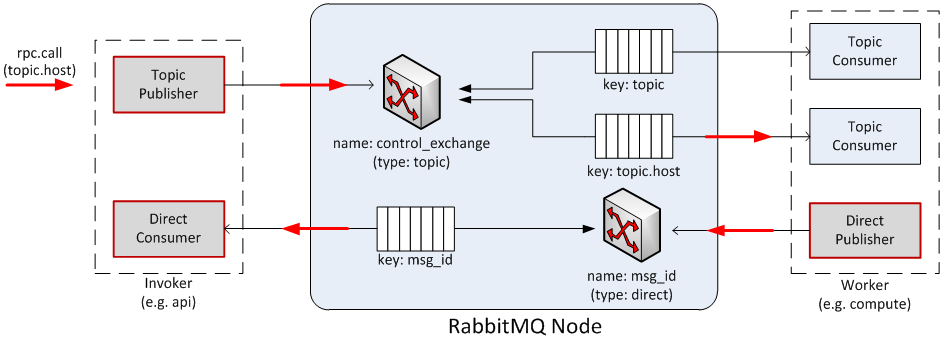
* RPC.call：发送请求到消息队列，等待返回最终结果。
* RPC.cast：发送请求到消息队列，不需要等待最终返回的结果。
* RPC.Notifier：发送各类操作消息到队列，不需要等待最终的返回结果。

RPC.call、RPC.cast一般用于同一个项目下的服务之间进行的“内部“请求；RPC.Notifier发送的操作消息，目前被ceilometer notification服务所接收。

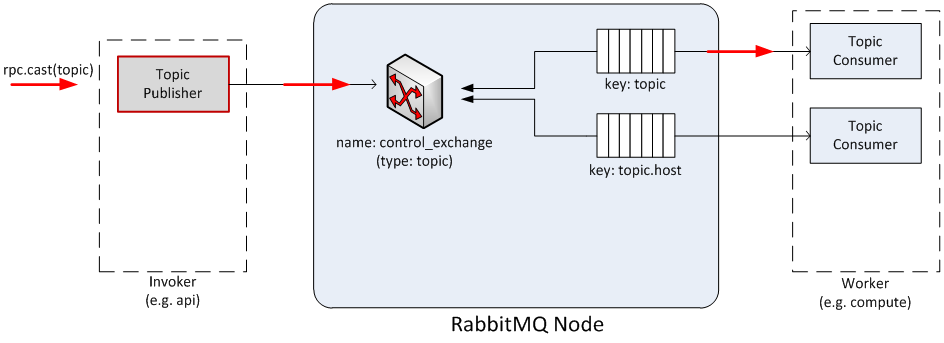
**RPC.call & RPC.cast**

一般情况下，openstack各个项目之间通过RestAPI接口进行相互访问，而项目内部服务之间则通过RPC请求的方式进行通信。

### RPC.call 请求

对于RPC.call请求，借助官方一张经典的图来描述：   
  
以nova-compute服务调用nova-network服务分配网络为例：   
1. nova-compute服务向消息队列服务的compute.node队列发送RPC请求，并等待请求的最终回复。   
2. nova-network服务通过nova exchange（topic exchange）从compute.node队列中获取消息并作出相应的处理。   
3. nova-network服务消息处理完了之后，向reply\_XXX队列发送一条回复消息   
4. nova-compute服务通过reply\_XXX exchange（direct exchange）接受从nova-network发送的RPC消息。

### RPC.cast请求

对于RPC.cast请求，同样借助官方一张经典的图来描述：   
  
以nova-conductor服务调用nova-compute服务build\_and\_run\_instance为例：   
1. nova-conductor服务向消息队列服务的compute队列发送RPC请求，请求结束，不需要等待请求的最终回复。   
2. nova-compute服务通过nova exchange（topic exchange）从compute队列中获取消息并作出相应的处理。

在openstack项目中，一般情况下，RPC server端发送一个请求到消息队列，一般只有一个消费者（及时有多个消费者）接受并处理这条消息，还有一种类型的RPC.cast请求，也称为fanout\_cast请求，fanout\_cast发送的是广播请求，所有对应的consumer都能接收到。

**RPC.Notifier**

openstack中对于资源的操作，如创建虚拟机，会向向消息队列的notifications.info，notifications.warn，notifications.error等队列发送对应的RPC消息，而ceilometer的notification服务会充当RPC 消费者端，获取对应的操作消息进行处理。   
RPC.Notifier提供的notifier消息有如下几种：

1. audit：审计类消息，如compute.instance.exist
2. info：正常操作消息，如compute.instance.create.end
3. warn：告警类操作消息，暂无
4. error：错误类操作消息，如scheduler.run\_instance
5. critical：严重错误，暂无
6. sample：sample消息，ceilometer中使用

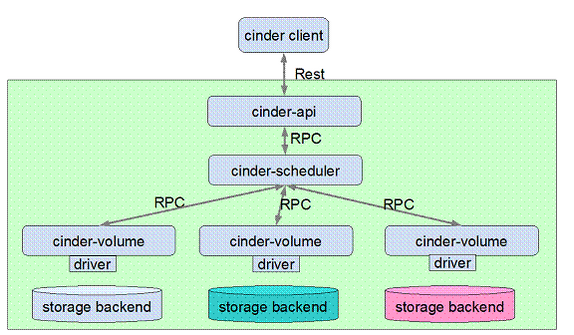
实际上openstack RPC notifier消息就是topic类型的消息，以虚拟机开始创建为例：   
1. nova-compute服务向消息队列服务的notifications.info队列发送RPC消息。   
2. ceilometer-agent-notification服务通过nova exchange（topic exchange）从notifications.info队列中获取消息并作出相应的处理。

========================

openstack rpc流程

Openstack 的主要组件有 Nova、Cinder、Neutron、Glance 等，分别负责云平台的计算、存储、网络资源管理。OpenStack 各组件之间是通过 REST 接口进行相互通信，而各组件内部则采用了基于 AMQP 模型的 RPC 通信

Cinder 主要由 3 个组件组成，cinder-api, cinder-scheduler 和 cinder-volume



Cinder-api 是 cinder 服务的 endpoint，提供 rest 接口，负责处理 client 请求，并将 RPC 请求发送至 cinder-scheduler 组件。

Cinder-scheduler 负责 cinder 请求调度，其核心部分就是 scheduler\_driver, 作为 scheduler manager 的 driver，负责 cinder-volume 具体的调度处理，发送 cinder RPC 请求到选择的 cinder-volume。

Cinder-volume 负责具体的 volume 请求处理，由不同后端存储提供 volume 存储空间。目前各大存储厂商已经积极地将存储产品的 driver 贡献到 cinder 社区

（一个cinder-volume服务对应一种存储类型，多个存储类型就有多个cinder-volume服务）

## OpenStack RPC 通信

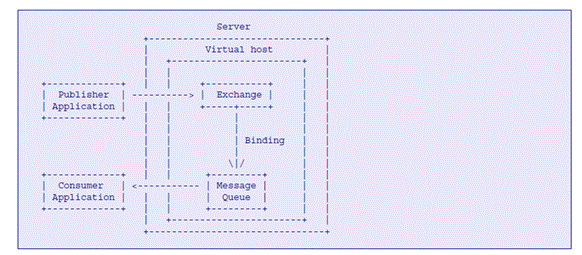
Openstack 组件内部的 RPC（Remote Producer Call）机制的实现是基于 AMQP(Advanced Message Queuing Protocol)作为通讯模型，从而满足组件内部的松耦合性。AMQP 是用于异步消息通讯的消息中间件协议，AMQP 模型有四个重要的角色:

* Exchange：根据 Routing key 转发消息到对应的 Message Queue 中
* Routing key：用于 Exchange 判断哪些消息需要发送对应的 Message Queue
* Publisher：消息发送者，将消息发送的 Exchange 并指明 Routing Key，以便 Message Queue 可以正确的收到消息
* Consumer：消息接受者，从 Message Queue 获取消息

消息发布者 Publisher 将 Message 发送给 Exchange 并且说明 Routing Key。Exchange 负责根据 Message 的 Routing Key 进行路由，将 Message 正确地转发给相应的 Message Queue。监听在 Message Queue 上的 Consumer 将会从 Queue 中读取消息。

Routing Key 是 Exchange 转发信息的依据，因此每个消息都有一个 Routing Key 表明可以接受消息的目的地址，而每个 Message Queue 都可以通过将自己想要接收的 Routing Key 告诉 Exchange 进行 binding，这样 Exchange 就可以将消息正确地转发给相应的 Message Queue。图 2 就是 AMQP 消息模型。

##### 图 2. AMQP 消息模型



AMQP 定义了三种类型的 Exchange，不同类型 Exchange 实现不同的 routing 算法：

* Direct Exchange：Point-to-Point 消息模式，消息点对点的通信模式，Direct Exchange 根据 Routing Key 进行精确匹配，只有对应的 Message Queue 会接受到消息
* Topic Exchange：Publish-Subscribe(Pub-sub)消息模式，Topic Exchange 根据 Routing Key 进行模式匹配，只要符合模式匹配的 Message Queue 都会收到消息
* Fanout Exchange：广播消息模式，Fanout Exchange 将消息转发到所有绑定的 Message Queue

OpenStack 目前支持的基于 AMQP 模型的 RPC backend 有 RabbitMQ、QPid、ZeroMQ，对应的具体实现模块在 cinder 项目下 Openstack/common/RPC/目录下，impl\_\*.py 分别为对应的不同 backend 的实现

## 安装和配置 RPC 服务

想要使 cinder 通信机制正常工作必须要安装并启动 rabbitmq 服务

Cinder 需要配置 RPC backend，并且设置 rabbitmq server 信息,在配置文件cinder.conf文件中，RPC\_backend参数 是设置RPC backend driver 的

## OpenStack RPC 通信角色

在 RPC 通信中有两个角色：Client 和 Server。Client 发起 RPC 请求，Server 端接收 RPC 请求调用本地的程序执行，并将执行结果返回 Client。Openstack RPC 实现采用了 AMQP 机制，AMQP 模型中由两个重要角色 Publisher 和 Consumer，Openstack RPC 即利用了 Publisher 和 Consumer 实现 Client 和 Server 之间的 RPC 通信过程。

Publisher 就是用于发布消息，在 Openstack 内部有四种类型的 Publisher 负责不同类型的消息发布，分别是 Topic Publisher、Direct Publisher、Notification Publisher 和 Fanout Publisher。不同的 Publisher 分别对应 AMQP 模型中的三种类型 Exchange，其中 Nofication Publisher 其实就是 Topic Publisher 的一种。Qpid 模块中 Publisher 的 UML 如图 3 所示，从 UML 图中可以看出 RPC 通信的关键元素

Publisher 就是用于发布消息，在 Openstack 内部有四种类型的 Publisher 负责不同类型的消息发布，分别是 Topic Publisher、Direct Publisher、Notification Publisher 和 Fanout Publisher。不同的 Publisher 分别对应 AMQP 模型中的三种类型 Exchange，其中 Nofication Publisher 其实就是 Topic Publisher 的一种

四种不同的 Publisher 在 RPC 通信有着不同的消息处理方式也有着不同的作用。

1、Direct Publisher：用于点对点的消息通信，在 OpenStack RPC 通信中用于建立 RPC 消息应答通路。创建或声明 Direct Exchange，用于 RPC 的消息返回，Exchange 的名字以消息 id 命名。

2、Topic Publisher：用于 Publish-Subscribe（Pub-Sub）模式的通信，Topic Publisher 创建 Topic Exchange，用于 RPC 消息发送，并设置消息的 Routing Key 转发消息

3、Fanout Publisher：创建 Fanout Exchange，用于广播消息的发送，所有绑定到该 Exchange 的 Message Queue 都会接收到消息

4、Notify Publisher：同 Topic Publisher，在 Openstack 系统中用于发送 Notification 相关的消息。

Consumer 就是消息接受方并处理消息调用服务方法，在 OpenStack 中定义了三种类型的 Consumer，如图 4 所示，分别是 Topic Consumer、Direct Consumer 和 Fanout Consumer，不同类型的 Consumer 也对应处理不同类型的 Message。Consumer 实际的工作是创建 Message Queue，并且设置 Routing Key

Callback：回调对象，即 RPC 服务提供者，根据消息内容调用具体的处理方法

不同类型的 Consumer 创建不同的 Message Queue 并绑定到对应的 Exchange 用于接收不同类型的消息

1、Direct Consumer：接收 Direct Exchange 发送的消息，创建专属的 Message Queue，即 Message Queue 的消息只能由创建 Message Queue 的 Consumer 获取，

2、Topic Consumer：接收 topic Exchange 根据 Routing Key 转发的消息，创建的 Message Queue 是可共享的，这就意味着多个 Consumer 可以订阅同一个 Message Queue，并从其中读取消息

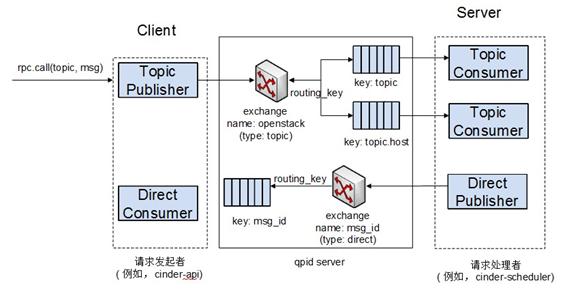
3、Fanout Consumer：接收 Fanout Exchange 广播的消息，并创建专属的 Message Queue

### RPC 发送请求

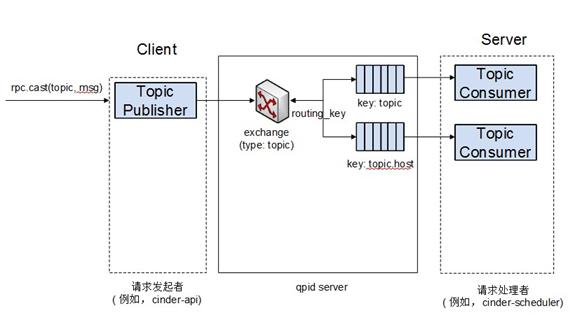
Client 端发送 RPC 请求由 publisher 发送消息并声明消息地址，consumer 接收消息并进行消息处理，如果需要消息应答则返回处理请求的结果消息。OpenStack RPC 模块提供了 rpc.call，rpc.cast, rpc.fanout\_cast 三种 RPC 调用方法，发送和接收 RPC 请求。

* rpc.call 发送 RPC 请求并返回请求处理结果，由 Topic Publisher 发送消息，Topic Exchange 根据消息地址进行消息转发至对应的 Message Queue 中，Topic Consumer 监听 Message Queue，发现需要处理的消息则进行消息处理，并由 Direct Publisher 将请求处理结果消息，请求发送方创建 Direct Consumer 监听消息的返回结果
* rpc.cast 发送 RPC 请求无返回，与 rpc.call 不同之处在于，不需要请求处理结果的返回，因此没有 Direct Publisher 和 Direct Consumer 处理。
* rpc.fanout\_cast 用于发送 RPC 广播信息无返回结果

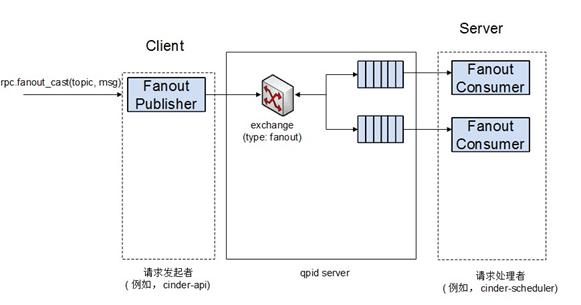
##### 图 5. RPC.call 消息流程



##### 图 6. RPC.cast 消息流程



##### 图 7. RPC.fanout 消息处理



### RPC 调用过程

Server 负责处理请求，首先需要创建 Consumer。cinder-scheduler server 创建了 cinder-scheduler 和 cinder-scheduler:host 的 Topic Consumer 和 cinder-scheduler Fanout Consumer，分别用于接收不同类型的消息。Consumer 创建时，就会在 Qpid server（或者rabbitmq server） 上创建对应的 Message Queue，并声明 Routing Key 绑定到 Exchange 上。Cinder-scheduler 服务启动时，将 cinder-scheduler manager 注册为了 RPC\_dispatcher(调度) 即 callback 对象即 Client 发送的请求最终会有 cinder-scheduler manager 对象调用执行。启动 consumer 线程，接受 Queue 上的消息，当线程接收到 Queue 上消息后，将消息传递给 RPC\_dispatcher 即 callback 对象，由 callback 对象根据消息内容，调用对应的处理函数，例如 create\_volume，并处理请求。

exchange类型

Openstack 负责 Topic Publisher 的请求根据具体的 route key 发送到对应的消息队列中；

Reply\_XXX 是由 Direct Publisher 创建用于消息处理返回结果通信

**OpenStack Notification**

OpenStack Notification 机制用于通知用户或者开发者当前请求执行状态

Notification priority 提供了 DEBUG, WARN, INFO, ERROR, CRITICAL 五个级别的消息，Cinder 的 notification 的 Exchange name 是 Openstack，topic 是 notifications.priority，例如 notifications.info

Cinder 项目默认的 Exchange name 是 OpenStack，由于 Exchange 负责信息转发，因此能够看到所有流入 Exchange 的消息，如果我们仅想要查看 notification 类型的消息，也可以自己创建 Message Queue，使用 Routing key 绑定到 Exchange 上

==========================

一个队列可以有多个receiver，队列里的一个消息只能发给一个receiver。至于发送给哪个receiver,由AMQP的负载算法决定，默认为在所有receiver中均匀发送(round robin).

一个消息可以被发送到一个队列中，也可以被发送到多个对列中。多队列情况下，一个消息可以被多个receiver收到并处理。Openstack RPC中这两种情况都会用到。

======================

**RPC的使用场景**

Openstack中RPC的主要使用场景：

**随机调用某server上的一个方法：**

Invoke Method on One of Multiple Servers

这个应该是Openstack中最常用的一种RPC调用，每个方法都会有多个server来提供，client调用时由底层机制选择一个server来处理这个调用请求。

像nova-scheduler, nova-conductor都可以以这种多部署方式提供服务。

这种场景通过AMQP的topic exchange实现。

所有server在binding中为binding key指定一个相同的topic(理解为routing key)， client在调用时使用这个topic既可实现。

**调用某特定server上的一个方法：**

Invoke Method on a Specific Server

一般Openstack中的各种scheduler会以这种方式调用。通常scheduler都会先选定一个节点，然后调用该节点上的服务。

这种场景通过AMQP的topic exchange实现。

每个server在binding中为其binding key指定一个自己特有的topic理解为routing key)， client在调用时使用这个topic既可实现。

**调用所有server上的一个方法：**

Invoke Method on all of Multiple Servers

这种其实就是一个广播系统。就像开会议，台上的人讲话，台下的人都能听到。

Openstack中有些rpcapi.py的某些方法带有fanout=True参数，这些都是让所有server处理某个请求的情况。

例子： neutron中所有plugin都会有一个AgentNotifierApi，这个rpc是用来调用安装在compute上的L2 agent。因为存在多个L2 agent(每个compute上都会有)，所以要用广播模式。

这种场景通过AMQP的fanout exchange实现。

每个server在binding中将其队列绑定到一个fanout exchange， client在调用时指定exchange类型为fanout即可。server和client使用同一个exchange。

**RCP的实现**

目前Openstack中有两种RPC实现，一种是在oslo messaging,一种是在openstack.common.rpc。

openstack.common.rpc是旧的实现，oslo messaging是对openstack.common.rpc的重构。openstack.common.rpc在每个项目中都存在一份拷贝，oslo messaging即将这些公共代码抽取出来，形成一个新的项目

以后的方向是各个项目都会使用oslo messaging的RPC功能，停止使用openstack.common.rpc。目前(icehouse release)nova, cinder都已经完成转变，neutron还在使用openstack.common.rpc。

rpc.call和rpc.cast的区别：

从实现代码上看，他们的区别很小，就是call调用时候会带有wait\_for\_reply=True参数，cast不带

**notification**

oslo messaging中除了RPC外，还有另外一种跨进程通信方式，即消息通知(notification)。notification和前面的第三种RPC场景(广播系统)非常类似，区别就是notification的消息(message)格式是有固定格式的，而RPC中的消息并无固定格式，取决于client/server之间的约定。

目前消息系统的主要receiver(消息收集者)为ceilometer系统，而publisher就是Openstack个项目的service。如nova-compute会针对虚拟机的生命周期发出各种通知：start/stop/create/destroy等。

notification的底层机制可以使用RPC，及driver类型为MessagingDriver。

=======================

openstack里面的基本概念

* Server: 为各个Client提供RPC接口,它是消息的最终处理者;
* Client: RPC接口的调用端, 我们常见的cast和call方法就是在这端调用;
* Exchange: 理解为一个消息交换机， 把消息分类，告诉何种路由到何种queue;
* Topic: 是一个RPC消息的唯一标识; servers监听这个topic的消息; client负责发出这个topic的消息;
* Namespace: servers可以在一个topic上，提供多种方法集合， 这些方法集合通过namespace来分开管理;
* Method: 这个慨念很简单， 就是函数, 即远程方法调用中的方法;
* API version: 也就是server上提供的RPC api接口集合的版本号，openstack中1.0起步, servers可以一次提供多种api version，client每次请求时只需描述它所需要的最低version就ok;
* Transport: 可以理解为传输载体，这个很好理解, 就是我们使用的消息队列中间件RabbitMQ, Qpid, ZeroMQ等等, 是负责整个消息处理的系统， 它负责消息传输直到提供给clients返回， 使用此系统者， 不用了解细节,  Openstack中实现的主要有这三种， AMQP标准下的rabbitMQ和Qpid， 和非AMQP的ZeroMQ， ZeroMQ更底层， 速度更快, 据说快10倍。
* Target这是个很重要的概念， 它描述了信息的处理方式， 该发哪里去(server属性)和消息处理端(server)监听什么信息(topic 属性)。以下是Target的属性