Python 中使用 RabbitMQ 消息队列

一、RabbitMQ使用场景

RabbitMQ他是一个消息中间件，说道消息中间件【最主要的作用：信息的缓冲区】还是的从应用场景来看下:

1、系统集成与分布式系统的设计

各种子系统通过消息来对接，这种解决方案也逐步发展成一种架构风格，即“通过消息传递的架构”。

举个例子：现在医院有两个科“看病科”和“住院科”在之前他们之间是没有任何关系的，如果你在“看病课”看完病后注册的信息和资料，到住院科后还得重新注册一遍？那现在要改革，你看完病后可以直接去住院科那两个系统之间需要打通怎么办？这里就可以使用我们的消息中间件了。

2、异步任务处理结果回调的设计

举个例子：记录日志，假如需要记录系统中所有的用户行为日志，如果通过同步的方式记录日志势必会影响系统的响应速度，当我们将日志消息发送到消息队列，记录日志的子系统就会通过异步的方式去消费日志消息。这样不需要同步的写入日志了NICE

3、并发请求的压力高可用性设计

举个例子：比如电商的秒杀场景。当某一时刻应用服务器或数据库服务器收到大量请求，将会出现系统宕机。如果能够将请求转发到消息队列，再由服务器去消费这些消息将会使得请求变得平稳，提高系统的可用性。

二、RabbitMQ的介绍

	ActiveMQ	RabbitMQ	Kafka
开发语言	Java	Erlang	Java
支持的协议	XMPP、REST、AMQP、STOMP等	AMQP	仿AMQP
事务	支持	不支持	
集群	支持	支持	支持
负载均衡	支持	支持	支持
动态扩容	不支持	支持	支持

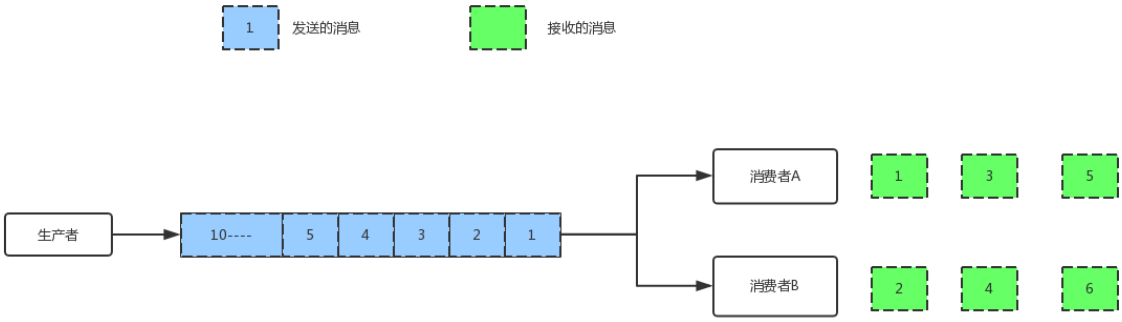
从上面的目前常用的消息中间件来说，感觉Kafka更好些，没错Kafka是大数据时代诞生的消息中间件，但对于目前来说使用最广的还是RabbitMQ。

高级消息队列协议 The Advanced Message Queuing Protocol (AMQP)

是一个标准开放的应用层的消息中间件（Message Oriented Middleware）协议。AMQP定义了通过网络发送的字节流的数据格式。因此兼容性非常好，任何实现AMQP协议的程序都可以和与AMQP协议兼容的其他程序交互，可以很容易做到跨语言，跨平台。

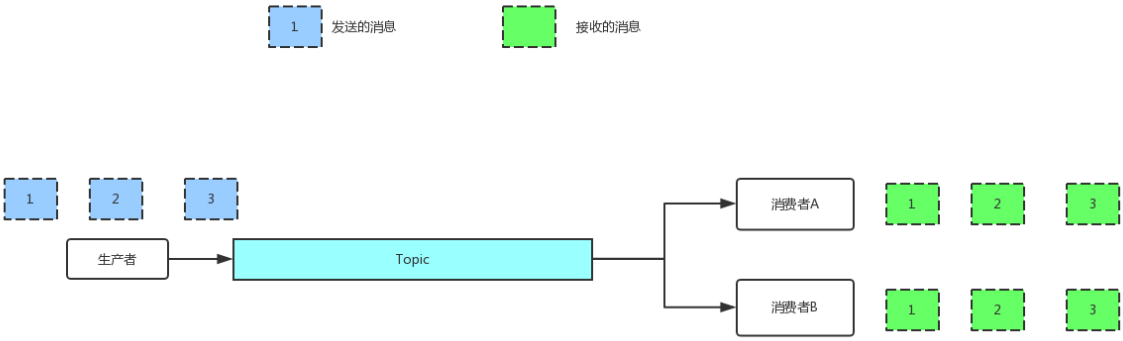
三、RabbitMQ和一般的消息传输模式：队列模式&主题模式区别

1、队列模式



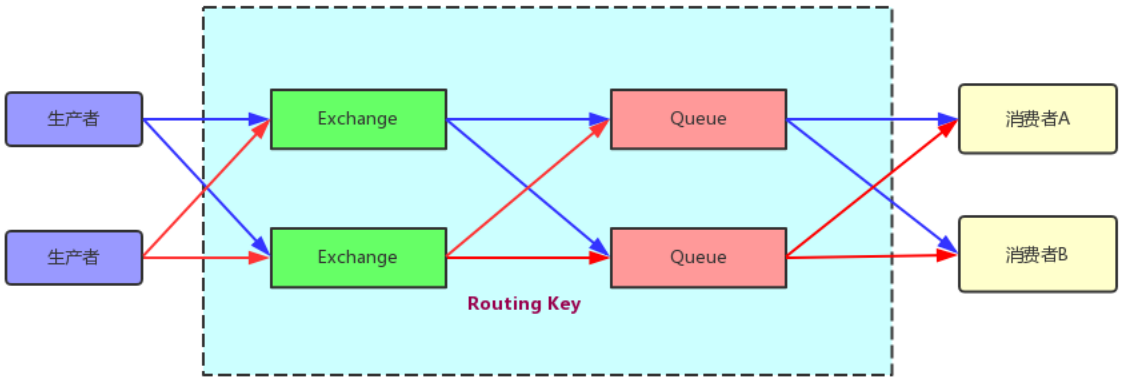
一个发布者发布消息，下面的接收者按队列顺序接收，比如发布了10个消息，两个接收者A,B那就是A,B总共会收到10条消息，不重复。

2、主题模式



对于Topic模式，一个发布者发布消息，有两个接收者A,B来订阅，那么发布了10条消息，A,B各收到10条消息。

3、RabbitMQ的模式



生产者生产消息后不直接直接发到队列中，而是发到一个交换空间：Exchange，Exchange会根据Exchange类型和Routing Key来决定发到哪个队列中，这个讲到发布订阅在详细来看

RabbitMQ安装配置

一、环境安装

```
#安装epel源[EPEL (Extra Packages for Enterprise Linux，企业版Linux的额外软件包) 是Fedora小组维护的一个软件仓库项目为RHEL/CentOS提供他们默认不提供的软件.]
rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
# 扩展erlang源安装最新的erlang
rpm -Uvh http://packages.erlang-solutions.com/erlang-solutions-1.0-1.noarch.rpm
yum -y install erlang
# 安装RabbitMQ
wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.1/rabbitmq-server-3.6.1-1.noarch.rpm
rpm --import https://www.rabbitmq.com/rabbitmq-signing-key-public.asc
yum install rabbitmq-server-3.6.1-1.noarch.rpm
```

各位看官如果想通过源码安装"**搜狗一下就知道**"这里不在过多的复述

二、用户配置

```
# 启动服务
/etc/init.d/rabbitmq-server start
# 添加用户
rabbitmqctl add_user admin admin
# 添加管理员权限
rabbitmqctl set_user_tags admin administrator
# 修改密码
abbitmqctl add_user admin youpassword
# 设置权限
rabbitmqctl  set_permissions  -p  '/'  admin '.' '.' '.'
```

三、启用WEB管理

```
# 启动Web插件
rabbitmq-plugins enable rabbitmq_management
# 删除guest用户
rabbitmqctl delete_user guest
# 添加Web访问权限
"""注意：rabbitmq从3.3.0开始禁止使用guest/guest权限通过除localhost外的访问。如果想使用guest/guest通过远程机器访问，需要在rabbitmq配置文件中(/etc/rabbitmq/rabbitmq.config)中设置loopback_users为[],配置文件不存在创建即可。"""
# 添加配置
[{rabbit, [{loopback_users, ["admin"]}]}].
```

RabbitMQ的消息玩法-Python

1、最简单的玩法-生产者消费者

send.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
生产者/发送消息方
"""
```

```
import pika

# 远程主机的RabbitMQ Server设置的用户名密码
credentials = pika.PlainCredentials('admin', 'admin')
connection = pika.BlockingConnection(pika.ConnectionParameters('192.168.31.123',
5672, '/', credentials))
"""
A virtual host holds a bundle of exchanges, queues and bindings. Why would you
want multiple virtual hosts?
Easy. A username in RabbitMQ grants you access to a virtual host…in its
entirety.
So the only way to keep group A from accessing group B's
exchanges/queues/bindings/etc.
is to create a virtual host for A and one for B. Every RabbitMQ server has a
default virtual host named "/".
If that's all you need, you're ready to roll.

virtualHost is used as a namespace
for AMQP resources (default is \"/\"),so different applications could use
multiple virtual hosts on the same AMQP server

[root@localhost ~]# rabbitmqctl list_permissions
Listing permissions in vhost "/" ...
admin    .    .    .
guest    .*    .*    .*
...done.

ConnectionParameters 中的参数:virtual_host 注:
    相当于在rabbitmq层面又加了一层域名空间的限制,每个域名空间是独立的有自己的Echange/queues
等
    举个好玩的例子Redis中的db0/1/2类似

"""
# 创建通道
channel = connection.channel()
# 声明队列hello,RabbitMQ的消息队列机制如果队列不存在那么数据将会被丢掉,下面我们声明一个队列如
果不存在创建
channel.queue_declare(queue='hello')
# 给队列中添加消息
channel.publish(exchange="",
                routing_key="hello",
                body="Hello World")
print("向队列hello添加数据结束")
# 缓冲区已经flush而且消息已经确认发送到了RabbitMQ中，关闭通道
channel.close()
```

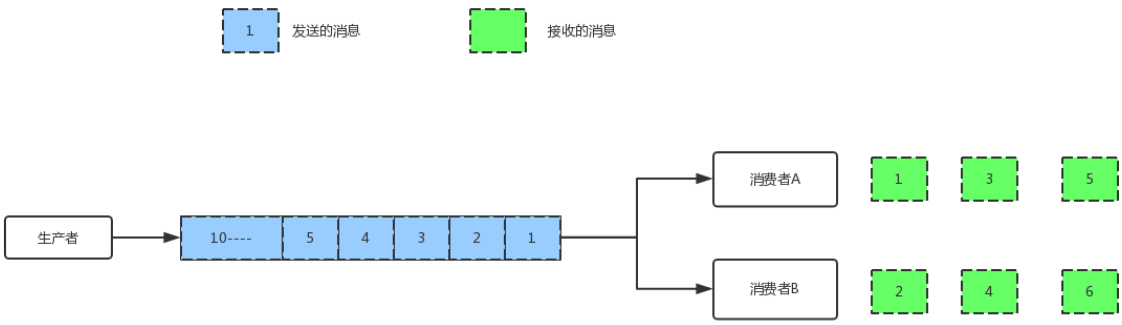
receive.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
消费者/接收消息方
"""
import pika

# 远程主机的RabbitMQ Server设置的用户名密码
```

```
credentials = pika.PlainCredentials('admin', 'admin')
connection = pika.BlockingConnection(pika.ConnectionParameters('192.168.31.123',
5672, '/', credentials))

# 创建通道
channel = connection.channel()
# 声明队列
channel.queue_declare(queue='hello')
"""
你可能会问为什么我们还要声明队列呢? 我们在之前代码里就有了,但是前提是我们已经知道了我们已经声明了代码,但是我们可能不太确定
谁先启动~ So如果你们100%确定也可以不用声明,但是在很多情况下生产者和消费者都是分离的.所以声明没有坏处
"""


# 订阅的回调函数这个订阅回调函数是由pika库来调用的
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

# 定义通道消费者参数
channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
# 开始接收信息，并进入阻塞状态，队列里有信息才会调用callback进行处理。按ctrl+c退出。
channel.start_consuming()
```

2、升级点玩法-工作队列

为了好理解还是把咱们的玩的东西应用到实际生活场景中比如：写日志同步慢、或者请求量写日志机器扛不住怎么办？异步、并且分担日志记录压力到多台服务器上。



类似上面的图的效果：

Work Queue背后的主要思想是避免立即执行资源密集型任务的时，需要等待其他任务完成。所以我们把任务安排的晚一些，我们封装一个任务到消息中并把它发送到队列，一个进程运行在后端发送并最终执行这个工作,当你运行多个消费者的时候这个任务将在他们之间共享。

send.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
```

```
"""
生产者/发送方
"""
import sys
import pika

# 远程主机的RabbitMQ Server设置的用户名密码
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters('192.168.31.123',
5672, '/', credentials))

# 创建通道
channel = connection.channel()

# 声明队列task_queue,RabbitMQ的消息队列机制如果队列不存在那么数据将会被丢掉,下面我们声明一个
队列如果不存在创建
channel.queue_declare(queue='task_queue')

# 在队列中添加消息
for i in range(100):
    message = '%s Meassage '% i or "Hello World!"
    # 发送消息
    channel.basic_publish(exchange='',
                          routing_key='task_queue',
                          body=message,
                          )

    # 发送消息结束,并关闭通道
    print(" [x] Sent %r" % message)

channel.close()
```

receive1.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
消费者/接收方
"""
import time
import pika

# 认证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))
# 建立通道
channel = connection.channel()
# 创建队列
channel.queue_declare("task_queue")


# 订阅的回调函数这个订阅回调函数是由pika库来调用的
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    print(body.count(b'.'))
```

```
    time.sleep(body.count(b'.'))
    print(" [x] Done")

# 定义通道消费者参数
channel.basic_consume(callback,
                      queue="task_queue",
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
# 开始接收信息，并进入阻塞状态，队列里有信息才会调用callback进行处理。按ctrl+c退出。
channel.start_consuming()
```

receive2.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
消费者/接收方
"""
import time
import pika

# 认证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))
# 建立通道
channel = connection.channel()
# 创建队列
channel.queue_declare("task_queue")


# 订阅的回调函数这个订阅回调函数是由pika库来调用的
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    print(body.count(b'.'))
    time.sleep(body.count(b'.'))
    print(" [x] Done")

# 定义通道消费者参数
channel.basic_consume(callback,
                      queue="task_queue",
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
# 开始接收信息，并进入阻塞状态，队列里有信息才会调用callback进行处理。按ctrl+c退出。
channel.start_consuming()
```

默认RabbitMQ按照顺序发送每一个消息，每个消费者会获得相同的数量消息，这种分发消息的方式称之为循环。

3、持久化和公平分发

1、消息持久化

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。 **这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑...千面是因为我们在消费者端标记了ACK=True关闭了它们，如果你没有增加ACK=True或者没有回执就会出现这个问题**

生产者需要在发送消息的时候标注属性为持久化

```
# 在队列中添加消息
for i in range(100):
    message = '%s Meassage '% i or "Hello World!"
    # 发送消息
    channel.basic_publish(exchange='',
                          routing_key='task_queue',
                          body=message,
                          properties=pika.BasicProperties(delivery_mode=2, ))  #
标记属性消息为持久化消息需要客户端应答

    # 发送消息结束,并关闭通道
    print(" [x] Sent %r" % message)
```

消费者需要发送消息回执

```
# 订阅回调函数,这个订阅回调函数是由pika库来调用
def callback(ch, method, properties, body):
    """
    :param ch: 通道对象
    :param method: 消息方法
    :param properties:
    :param body: 消息内容
    :return: None
    """
    print(" [x] Received %r" % (body,))
    time.sleep(2)
    print(" [x] Done")
    # 发送消息确认,确认交易标识符
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

我们可以通过命令查看那些消费者没有回复ack确认

```
# Linux
rabbitmqctl list_queues name messages_ready messages_unacknowledged
# Windows
rabbitmqctl.bat list_queues name messages_ready messages_unacknowledged
```

2、队列持久化

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。 这里我们需要修改下生产者和消费者设置RabbitMQ消息的持久化***[生产者/消费者]都需要配置***

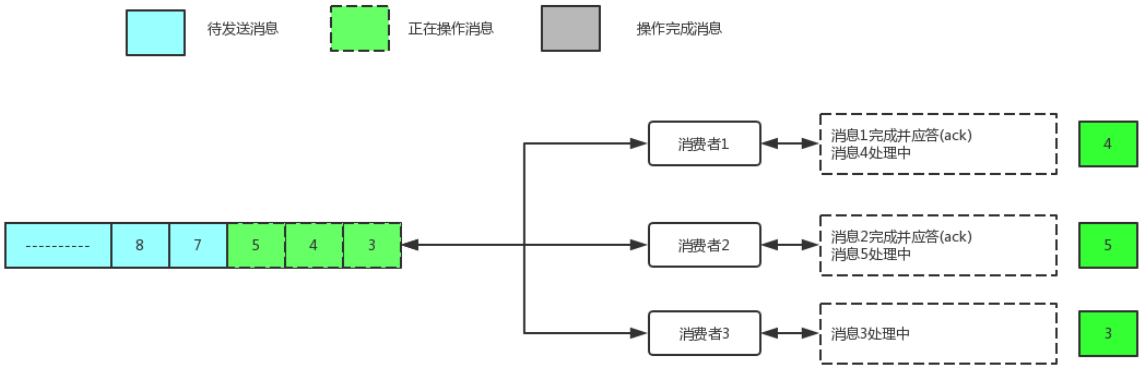
```
channel.queue_declare(queue='task_queue', durable=True)  # 队列持久化
```

3、公平分发

默认情况下RabitMQ会把队列里面的消息立即发送到消费者，无论该消费者有多少消息没有应答，也就是说即使发现消费者来不及处理，新的消费者加入进来也没有办法处理已经堆积的消息，因为那些消息已经被发送给老消费者了。类似下面的

在消费者中增加:`channel.basic_qos(prefetch_count=1)`

prefetchCount：会告诉RabbitMQ不要同时给一个消费者推送多于N个消息，即一旦有N个消息还没有ack，则该consumer将block掉，直到有消息ack。 这样做的好处是，如果系统处于高峰期，消费者来不及处理，消息会堆积在队列中，新启动的消费者可以马上从队列中取到消息开始工作。



工作过程如下:

1. 消费者1接收到消息后处理完毕发送了ack并接收新的消息并处理
2. 消费者2接收到消息后处理完毕发送了ack并接收新的消息并处理
3. 消费者3接收到消息后一直处于消息中并没有发送ack不在接收消息一直等到消费者3处理完毕后发送ACK后再接收新消息

4、发布与订阅-高级玩法

发布与订阅这里同样是拿两个好玩的实际例子：我们来写一个日志系统

在前面我们学了work Queue它主要是把每个任务分给一个worker[工作者]接下来我们要玩些不同的,把消息发多个消费者(不同的队列中). 这个模式称之为"发布订阅"

举个例子我们将创建一个简单的日志系统,包含两个程序第一个是用来发送日志,第二个是用来接收日志，接收日志的程序每一个副本都将收到消息,这样我们可以一个接收器用来写入磁盘，一个接收器用来输入到日志~ cool~

Exchanges可用的类型很少：**direct, topic, headers, fanout**---4种,我们先看最后一种

1、fanout模式

模式特点：

- 可以理解他是一个广播模式
- 不需要routing key它的消息发送时通过Exchange binding进行路由的~~在这个模式下routing key失去作用
- 这种模式需要提前将Exchange与Queue进行绑定，一个Exchange可以绑定多个Queue，一个Queue可以同多个Exchange进行绑定
- 如果接收到消息的Exchange没有与任何Queue绑定，则消息会被抛弃。

send.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
import sys
import time
import pika

# 远程主机的RabbitMQ Server设置的用户名密码
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))

# 创建通道
channel = connection.channel()

# 声明Exchanges
channel.exchange_declare(exchange="logs",
                         exchange_type="fanout")
"""
这里可以看到我们建立连接后,就声明了Exchange,因为把消息发送到一个不存在的Exchange是不允许的,
如果没有消费者绑定这个,Exchange消息将会被丢弃这是可以的因为没有消费者
"""
# 添加消息
message = ' '.join(sys.argv[1:]) or "info: Hello World!"
channel.basic_publish(exchange="logs",  # 将消息发送至Exchange为logs绑定的队列中
                      routing_key="",
                      body=message,)

print(" [x] Sent %r" % message)
# 关闭通道
connection.close()
```

receive.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
import pika

# 连接RabbitMQ验证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))

# 建立通道
channel = connection.channel()
```

```
# Bindings Exchanges 接收消息
channel.exchange_declare(exchange='logs',
                         exchange_type="fanout")
# 使用随机队列/并标注消费者断开连接后删除队列
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

# Queue 与 Exchanges绑定
channel.queue_bind(exchange='logs',
                   queue=queue_name)

print(' [*] Waiting for logs. To exit press CTRL+C')


# 定义回调函数这个回调函数将被pika库调用
def callback(ch, method, properties, body):
    print(" [x] %r" % body)

# 定义接收消息属性
channel.basic_consume(callback,
                      queue=queue_name,
                      no_ack=True)

# 开始接收消息
channel.start_consuming()
```

测试：那现在我就可以把信息分开来记录了

现在如果你想把日志存入文件

```
python fanout_receive1.py > fanout_receive_log.txt
```

并且想再屏幕也输出,在打开一个窗口

```
python fanout_receive1.py
```

概念解释：binding

当我们创建了Exchanges和(QUEUE)队列后,我们需要告诉Exchange发送到们的Queue队列中,所需要需要把Exchange和队列(Queue)进行绑定,

```
channel.queue_bind(exchange='logs',
                   queue=result.method.queue)
```

2、Direct 模式

任何发送到Direct Exchange的消息都会被转发到routing_key中指定的Queue

\1. 一般情况可以使用rabbitMQ自带的Exchange：”” (该Exchange的名字为空字符串)， 也可以自定义Exchange

\2. 这种模式下不需要将Exchange进行任何绑定(bind)操作。当然也可以进行绑定。可以将不同的routing_key与不同的queue进行绑定，不同的queue与不同exchange进行绑定

\3. 消息传递时需要一个“routing_key”

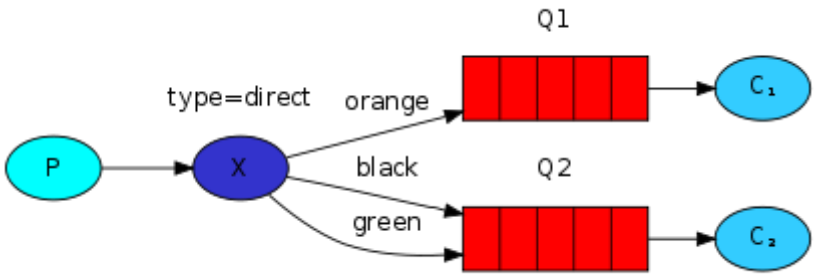
\4. 如果消息中不存在routing_key中绑定的队列名，则该消息会被抛弃。

如果一个exchange 声明为direct，并且bind中指定了routing_key,那么发送消息时需要同时指明该exchange和routing_key.

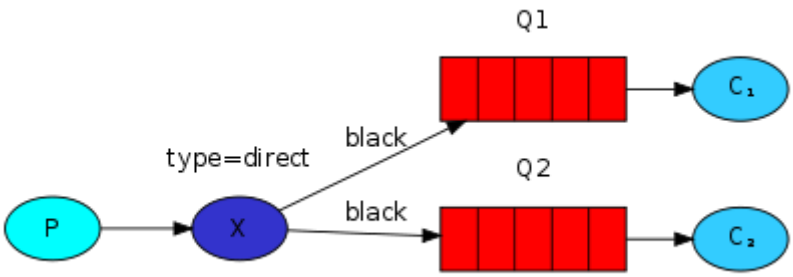
简而言之就是：生产者生成消息发送给Exchange, Exchange根据Exchange类型和basic_publish中的routing_key进行消息发送 消费者：订阅Exchange并根据Exchange类型和binding key(bindings 中的routing key) ,如果生产者和订阅者的routing_key相同，Exchange就会路由到那个队列。

老规矩还是通过实例来说：

在上面的文档中我们创建了一个简单的日志系统，我们把消息发给所有的订阅者 在下面的内容中将把特定的消息发给特定的订阅者，举个例子来说,把error级别的报警写如文件,并把所有的报警打印到屏幕中,进行了路由的规则类似下面的架构



这里也要注意一个routing key 是可以绑定多个队列的



在上面我们已经创建过bindings了类似下面

```
channel.queue_bind(exchange=exchange_name,
                   queue=queue_name)
```

消费者端：Bindings可以增加routing_key 这里不要和basic_publish中的参数弄混了，我们给它称之为***binding key***

```
channel.queue_bind(exchange=exchange_name,
                   queue=queue_name,
                   routing_key='black')
```

binding key的含义依赖于Exchange类型,fanout exchanges类型只是忽略了它

emit_log_direct.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
```

```
生产者/发送方
"""
import pika
import sys

# 添加RabbitMQ Server端认证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))

# 创建通道
channel = connection.channel()
# 声明Exchange 且类型为 direct
channel.exchange_declare(exchange='direct_logs',
                         exchange_type='direct')

# 从调用参数中或去类型
severity = sys.argv[1] if len(sys.argv) > 2 else 'info'
message = ' '.join(sys.argv[2:]) or 'Hello World!'
# 发送消息指定Exchange为direct_logs,routing_key 为调用参数获取的值
channel.basic_publish(exchange='direct_logs',
                      routing_key=severity,
                      body=message)
# 打印信息
print(" [x] Sent %r:%r" % (severity, message))
# 关闭通道
connection.close()
```

receive_logs_direct.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
消费者/接收方
"""
import pika
import sys

# 添加RabbitMQ Server端认证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))

# 创建通道
channel = connection.channel()
# binding Exchange 为direct_logs, 且Exchange类型为direct
channel.exchange_declare(exchange='direct_logs',
                         exchange_type='direct')

# 生成随机接收队列
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

# 接收监听参数
severities = sys.argv[1:]
if not severities:
```

```
    sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
    sys.exit(1)

# 从循环参数中获取routing_key,并绑定
for severity in severities:
    channel.queue_bind(exchange='direct_logs',
                       queue=queue_name,
                       routing_key=severity)

print(' [*] Waiting for logs. To exit press CTRL+C')


# 定义回调参数
def callback(ch, method, properties, body):
    print(" [x] %r:%r" % (method.routing_key, body))

# 定义通道参数
channel.basic_consume(callback,
                      queue=queue_name,
                      no_ack=True)
# 开始接收消息
channel.start_consuming()
```

测试接收error 写入到文件

```
python receive_logs_direct.py error > logs_from_rabbit.log
```

测试接收所有级别的报警输出值公屏

```
python receive_logs_direct.py info warning error
```

测试发送消息

```
python emit_log_direct.py error "Run. Run. Or it will explode."
```

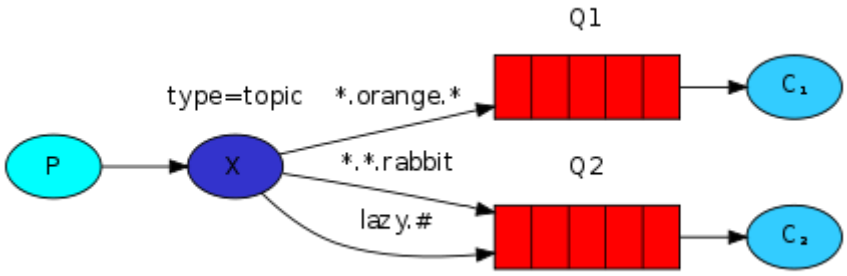
3、topic类型

前面讲到direct类型的Exchange路由规则是完全匹配binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。

topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，

它约定：

- routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
- binding key与routing key一样也是句点号“. ”分隔的字符串
- binding key中可以存在两种特殊字符“”*与“#”，用于做模糊匹配，其中“*”*用于匹配一个单词，“#”用于匹配多个单词（可以是零个）

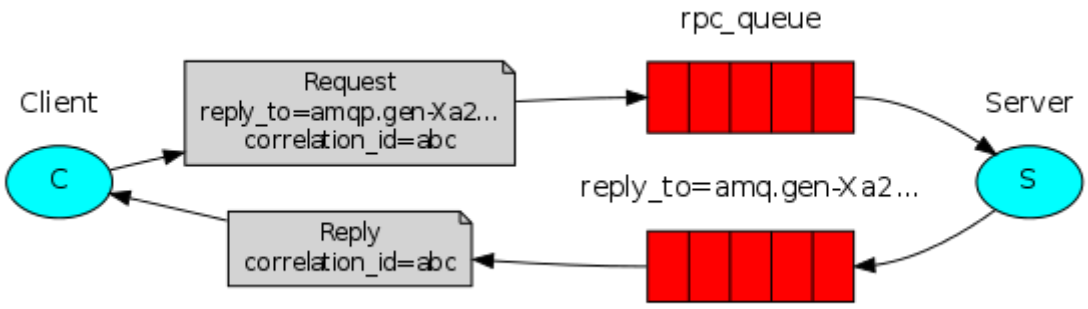


以上图中的配置为例，routingKey=”quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=”lazy.orange.fox”的消息会路由到Q1，routingKey=”lazy.brown.fox”的消息会路由到Q2，routingKey=”lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；routingKey=”quick.brown.fox”、routingKey=”orange”、routingKey=”quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

RPC

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。

但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。



1、RabbitMQ实现RPC机制是

- 客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了14种properties，这些属性会随着消息一起发送）中设置两个值replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）
- 服务器端收到消息并处理
- 服务器端处理完消息后，将生成一条应答消息到replyTo指定的Queue，同时带上correlationId属性
- 客户端之前已订阅replyTo指定的Queue，从中收到服务器的应答消息后，根据其中的correlationId属性分析哪条请求被执行了，根据执行结果进行后续业务处理

虽然RPC在计算中是一个很常见的模式，但它经常受到批评。当程序员不知道函数调用是本地的还是慢RPC时，问题就出现了。这样的混淆导致了不可预测的系统，并增加了调试的不必要的复杂性。

与简化软件不同，误用的RPC可能导致无法维护的面代码。 >

记住上面几点问题,考虑下面几个建议

- - 确保调用的函数是本地的还是远程的

- - 记录您的系统。明确组件之间的依赖关系
- - 处理错误案例。当RPC服务器关闭很长时间时，客户端应该如何反应?

如果对RPC有很多疑问，如果可以的话最好使用异步管道 一般来说在RabbitMQ执行RPC是很简单的,客户端发送请求服务端响应消息，为了接收响应，客户端需要发送一个“回调”队列地址。

```
result = channel.queue_declare(exclusive=True)
callback_queue = result.method.queue

channel.basic_publish(exchange='',
                      routing_key='rpc_queue',
                      properties=pika.BasicProperties(
                            reply_to = callback_queue,
                            ),
                      body=request)
```

AMQP 0-9-1 协议预先定义了14种属性,除了以下几种之外常用其他的不经常用

- - delivery_mode: 标记消息持久化,值为2的时候为持久化其他任何值都是瞬态的
- - content_type: 用来描述mime-type编码,举个例子来说经常使用JSON编码的话将此属性设置为: application/json.
- - reply_to:通常用于命名回调队列
- - correlation_id:将RCP请求和响应进行关联的id

Correlation id

上面的描述中我们建议为每个RPC创建一个队列但是这个很低效，幸运的是我们有更好的办法：为每个客户端创建一个队列 但是这就会触发一个新的问题，我们不确定这个消息是哪个返回的这里就用到了Correlation id 我们给每个请求设置一个唯一值，最后当我们收到消息在这个callback Queue中，我们查看这个属性和请求的属性(Correlation_id)进行匹配如果没有匹配上，我们将拒绝这个消息它不是我们的~

您可能会问，为什么我们应该忽略回调队列中的未知消息，而不是错误地失败呢?这是由于服务器端可能出现竞态条件。虽然不太可能，但是RPC服务器可能在发送了答案后才会死亡，但在发送请求消息之前。

如果发生这种情况，重新启动的RPC服务器将再次处理此请求。这就是为什么在客户端我们必须优雅地处理重复的响应，而RPC应该是幂等（）的。

RPC幂等性:

- f(x)=f(f(x))
- 如果消息具有操作幂等性，也就是一个消息被应用多次与应用一次产生的效果是一样的

2、工作机制

- 当客户端启动会创建一个匿名回调队列

- 在RCP请求Client发送两个属性:reply_to 标记callback队列,correlation_id 每个请求的唯一值，这个请求是被发送到rpc_queue 中

- 这个RPC worker (aka: server)等待请求在这个rpc_queue中,当请求出现它执行任务并将结果返回给客户端，发送到那个队列呢？就是我们reply_to标记的队列，客户端等待数据返回在这个callback队列，当消息出现 检查correlation_id 属性是否是和请求的如果匹配进行相应

rpc_server.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
```

```
# __author__ == 'LuoTianShuai'
"""
RPC/Server端
"""
import pika
# 添加认证信息
credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123",
5672, "/", credentials))

# 添加一个通道
channel = connection.channel()
# 添加一个队列,这个队列在Server就是我们监听请求的队列
channel.queue_declare(queue='rpc_queue')


# 斐波那契计算
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)


#  应答函数,它是我们接受到消息后如处理的函数替代原来的callback
def on_request(ch, method, props, body):
    n = int(body)
    print(" [.] fib(%s)" % n)
    response = fib(n)

    ch.basic_publish(exchange='',
                     # 返回的队列,从属性的reply_to取出来
                     routing_key=props.reply_to,
                     # 添加correlation_id,和Client进行一致性匹配使用的

properties=pika.BasicProperties(correlation_id=props.correlation_id),
                     # 相应消息/我们的处理结果
                     body=str(response))
    # 增加消息回执
    ch.basic_ack(delivery_tag=method.delivery_tag)

# 每次预处理的请求个数/这个请求我没有处理完不要给我发了
channel.basic_qos(prefetch_count=1)
# 定义接收通道的属性/定义了callback方法,接收的队列,返回的队列在哪里? on_request 的
routing_key=props.reply_to
channel.basic_consume(on_request, queue='rpc_queue')

# 开始接收消息
print(" [x] Awaiting RPC requests")
channel.start_consuming()
```

这个Server端的代码是非常简单的

- 往常一样我们先创建连接并声明队列 - 我们声明我们的斐波那契额函数，它只假设有效的正整数输入。(不要期望这个能够为大数据工作，它可能是最慢的递归实现)。

- 我们声明了callback在basic_consume，它是RCP核心请求过来执行callback，它来工作并发送相应

- 我们可能希望运行多个服务器进程。为了在多个服务器上平均分配负载，我们需要设置prefetch_count设置

rpc_client.py

```
# !/usr/bin/env python3.5
# -*- coding:utf-8 -*-
# __author__ == 'LuoTianShuai'
"""
RPC/Client端
"""
import pika
import uuid


# 定义菲波那切数列RPC Client类调用RPC Server
class FibonacciRpcClient(object):
    def __init__(self):
        # 添加认证信息
        credentials = pika.PlainCredentials("admin", "admin")
        self.connection =
pika.BlockingConnection(pika.ConnectionParameters("192.168.31.123", 5672, "/",
credentials))
        # 添加一个通道
        self.channel = self.connection.channel()
        # 生成一个随机队列-定义callback回调队列
        result = self.channel.queue_declare(exclusive=True)
        self.callback_queue = result.method.queue
        # 定义回调的通道属性
        self.channel.basic_consume(self.on_response,  # 回调结果执行完执行的Client端
的callback方法
                                   no_ack=True,
                                   queue=self.callback_queue)
        # 这里注意我们并没有直接阻塞的开始接收消息了

    # Client端 Callback方法
    def on_response(self, ch, method, props, body):
        if self.corr_id == props.correlation_id:
            # 定义了self.response = body
            self.response = body

    def call(self, n):
        # 定义了一个普通字段
        self.response = None
        # 生成了一个uuid
        self.corr_id = str(uuid.uuid4())
        # 发送一个消息
        self.channel.basic_publish(exchange='',    # 使用默认的Exchange,根据发送的
routing_key来选择队列
                                   routing_key='rpc_queue',  # 消息发送到rpc_queue
队列中
                                   # 定义属性
                                   properties=pika.BasicProperties(
                                         reply_to=self.callback_queue,  # Client
端定义了回调消息的callback队列
```

```
                                        correlation_id=self.corr_id,  # 唯一值用来做什么的？request和callback 匹配用
                                        ),
                                    body=str(n))

        # 开始循环 我们刚才定义self.response=None当不为空的时候停止`
        while self.response is None:
            # 非阻塞的接受消息
            self.connection.process_data_events(time_limit=3)
        return int(self.response)

# 实例化对象
fibonacci_rpc = FibonacciRpcClient()

print(" [x] Requesting fib(30)")
# 发送请求计算菲波那切数列 30
response = fibonacci_rpc.call(30)
print(" [.] Got %r" % response)
```

- 我们建立一个连接，通道，并声明一个排他的“回调”队列作为回复

- 我们订阅这个callback队列，所以我们可以接收RPC Server消息

- 每个相应回来之后执行非常简单的on_response方法，每个相应来后检查下correlation_id是否是匹配的，保存相应并退出接收循环

- 接下来，我们定义主调用方法——它执行实际的RPC请求。

- 我们定义了uuid，并在消息发送时添加了两个属性:reply_to、correlation_id

概念汇总

ConnectionFactory、Connection、Channel

ConnectionFactory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。

ConnectionFactory为Connection的制造工厂。

Channel是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

exchange

实际上生产者不会直接把消息放到消息队列里，实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）

queue

Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示。

RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。

Message acknowledgment

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会

将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。

这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑...

Message durability

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。

Prefetch count

生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。 在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。 RabbitMQ为routing key设定的长度限制为255 bytes。

Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。 在绑定（Binding）Exchange与Queue的同时，一般会指定一个binding key；生产者将消息发送给Exchange时，一般会指定一个routing key；当binding key与routing key相匹配时，消息将会被路由到对应的Queue中。这个将在Exchange Types章节会列举实际的例子加以说明。

在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的binding key。 binding key并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视binding key，而是将消息路由到所有绑定到该Exchange的Queue。