

RabbitMQ 使用参考

邹业盛 2016-01-10 16:55 更新



1. 安装

从网站 <http://www.rabbitmq.com/install-generic-unix.html> 下载到二进制源码, 进入 `sbin` 目录, 直接运行 `server` 即可.

默认服务监听在 5672 端口上(带上 SSL 默认在 5671 上).

2. 基本概念

RabbitMQ, 是一个使用 **erlang** 编写的 **AMQP** (高级消息队列协议) 的服务实现. 简单来说, 就是一个功能强大的消息队列服务.

通常我们谈到队列服务, 会有三个概念, **发消息者**, **队列**, **收消息者**. (**消息** 本来也应该算是一个独立的概念, 但是简单处理之下, 它可能并没有太多的内涵)

流程上是, **发消息者** 把消息放到 **队列** 中去, 然后 **收消息者** 从 **队列** 中取出消息.

RabbitMQ 在这个基本概念之上, 多做了一层抽象, 在 **发消息者** 和 **队列** 之间, 加入了 **交换器 (Exchange)**. 这样 **发消息者** 和 **队列** 就没有直接联系, 转而变成 **发消息者** 把消息给 **交换器**, **交换器** 根据调度策略再把消息再给 **队列**.

当然, 多一层抽象会增加复杂度, 但是同时, 功能上也更灵活. 事实上, 很多时候面对具体场景时, 在这种"四段式"的结构下, 你可选择的方案不止一种的. 不过也不必过于担心, 在一些自我规定的"原则"之下, "正确"的方案也不会那么纠结.

总结一下 **4 + 1** 个概念, 或者说, 五种角色:

Producing, 生产者, 产生消息的角色.

Exchange, 交换器, 在得到生产者的消息后, 把消息扔到队列的角色.

Queue, 队列, 消息暂时呆的地方.

Consuming, 消费者, 把消息从队列中取出的角色.

消息 Message , RabbitMQ 中的消息有自己的一系列属性, 某些属性对信息流有直接影响.

在使用过程中, 我们通常还会关注如下的机制:

持久化 , 服务重启时, 是否能恢复队列中的数据.

调度策略 , 交换器如何把消息给到哪些队列, 是每个队列给一条, 或者把一条消息给多个队列.

分配策略 , 队列面对消费者时, 如何把消息吐出去, 来一个消费者就把消息全给它, 还是只给一条.

状态反馈 , 当消息从某一个队列中被提出后, 这个消息的生命周期就此结束, 还是说需要一个具体的信号以明确标识消息已被正确处理.

上面这些内容, 初看之下好像情况有些复杂了, 不过在具体使用过程中, 这些东西都是很自然地需要考虑的. 当一套服务跑起来之后, 这些细枝末节自然消失在无形之中.

3. 基本形式

当服务启在 5672 端口之后, 我们就可以开始使用 **RabbitMQ** 了.

根据前面的内容, 我们需要站在两个角度(消息的提供方, 和消息的使用方), 去分别考虑五种角色的情况. 当然, 在使用时其实只是两个角度, 每边四种角色的情况. 因为消息的提供方不关心使用方, 反之, 消息的使用方也不关心消息的提供方. 这种关系上的无依赖本身是"队列服务"的一个最大使用意义所在, 用于业务间的分离(不管是分了好, 还是必须分).

我们先看如何产生消息, 即把消息放到队列当中, 等待下一步的处理.

(之后的代码, 使用 Python , 相应的 AMQP 协议实现的模块是 **pika**)

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.exchange_declare(exchange='first', type='fanout')  
channel.queue_declare(queue='hello')  
channel.queue_bind(exchange='first', queue='hello')  
channel.basic_publish(exchange='first', routing_key='', body='Hello World!')  
  
-----
```

上面代码的细节先不用管它, 但是直观看到做的事有:

- 获取连接.

- 从连接上获取一个 `channel` , 类似于数据库访问在连接上获取一个 `cursor` .
- 声明一个 `exchange` . (只会创建一次)
- 声明一个 `queue` . (只会创建一次)
- 把 `queue` 绑定到 `exchange` 上.
- 向指定的 `exchange` 发送一条消息.

消息发出之后, 可以使用 `rabbitmqctl` 这个工具查看服务的一些当前状态, 比如队列情况:

```
-----  
$ ./rabbitmqctl list_queues  
Listing queues ...  
hello      3  
...done.  
-----
```

然后是另一边, 从队列取出消息:

```
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.queue_declare(queue='hello')  
  
def callback(ch, method, properties, body):  
    print body  
  
channel.basic_consume(callback, queue='hello', no_ack=True)  
channel.start_consuming()  
  
-----
```

前面部分和之前的一样, 获取连接, 拿到 `channel` . 这里声明 `queue` 是在做重复的事 (之前 `Producing` 的代码已经做过声明了). 但是 `Producing` 和 `Consuming` 的代码你并不知道哪一个会先执行, 所以为了确保需要的 `queue` 是存在的, 使用时总先声明一下是好的方式.

接下来就是定义了一个异步回调, 标明在获取到消息之后要执行的处理函数.

最后, 开始接收服务器的消息.

和前面一样, 看一下代码做的事:

- 获取连接.

- 从连接上拿到 `channel` .
- 声明需要的 `queue` .
- 定义一个从指定 `queue` 获取消息的回调处理.
- 开始接收消息.

两边的代码都完成了, 可以先把取出消息的代码跑起来, 然后再重复运行产生消息的代码, 就能看到效果.

这里我们可以先反思一下我们的思维. 从流程上来说, 之前我们是先考虑如何产生消息, 然后是如何获取消息. 我们按这个顺序来编写了两段代码. 但是我们在使用时, 顺序反过来是一种更直观的方式. 即先是有服务跑起来, 守着等消息. 然后才是不定时有消息产生出来. 这一前一后在思维上是有一些微妙的不同的. 如果从 `C/S` 结构上来看, `Consuming` 的角色更像是 `Server`, 而 `Producing` 的角色更像是 `Client` .

为什么在这里讲这个呢, 是因为稍后会依次介绍整个流程中的细节, 比如 `exchange` 的调度策略, 多个 `Producing`, 多个 `Consuming`, 多个 `Queue` 的情况下, 我们如何去实现期望的行为. 当系统中的元素与角色有些多的时候, 我们需要一个比较明确的思维方式来保持自己的清醒.

基本的流程就是前面的两段代码. 接下来会依次介绍提到过的, 我们关心的几个机制.

- 持久化
- 调度策略
- 分配策略
- 状态反馈

然后, 会有几个实例分析, 用以演示一些典型的使用模式.

4. 持久化

考虑这样的场景, 当消息被暂存到队列后, 在没有被提取的情况下, `RabbitMQ` 服务停掉了怎么办.

```
-----  
# -*- coding: utf-8 -*-
```

```
import pika
```

```
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.exchange_declare(exchange='first', type='fanout')  
channel.queue_declare(queue='hello')
```

```
channel.queue_bind(exchange='first', queue='hello')
channel.basic_publish(exchange='first', routing_key='', body='Hello World!')
```

上面的代码，我们创建了一条内容为 `Hello World!` 的消息，通过命令行工具：

```
$ ./rabbitmqctl list_queues
Listing queues ...
hello      1
...done.

$ ./rabbitmqctl list_exchanges
Listing exchanges ...
    direct
amq.direct  direct
amq.fanout  fanout
amq.headers headers
amq.match   headers
amq.rabbitmq.log    topic
amq.rabbitmq.trace  topic
amq.topic    topic
first       fanout
...done.
```

可以查到，在名为 `hello` 的队列中，有 1 条消息。有一个类型为 `fanout`，名为 `first` 的交换器。

此时通过 `Ctrl-C` 或 `./rabbitmqctl stop` 把 RabbitMQ 服务停掉，再重启。交换器，队列，消息都是不会恢复的。

所以，默认情况下，**消息**，**队列**，**交换器** 都不具有持久化的性质。如果我们需要持久化功能，那么在声明的时候就需要配置好。

交换器和队列的持久化性质，在声明时通过一个 `durable` 参数即可实现：

```
channel.exchange_declare(exchange='first', type='fanout', durable=True)
channel.queue_declare(queue='hello', durable=True)
```

这样，在服务重启之后，`first` 和 `hello` 都会恢复。但是 `hello` 中的消息不会，还需要额外配置。这是 **消息** 的属性的相关内容：

```
channel.basic_publish(exchange='first',
                      routing_key='',
                      body='Hello World!',
                      properties=pika.BasicProperties(
```

```
        delivery_mode = 2,
    ))
```

通过 `properties`，把此条消息，仅仅是此条消息配置成需要持久化的。这样，在服务重启之后，队列中的这种消息就可以恢复。

这里注意一下，消息的持久化并不是一个很强的约束，涉及数据落地的时机，及系统层面的 `fsync` 等问题，不要认为消息完全不会丢。如果要尽可能高地提高消息的持久化的有效性，还需要配置其它的一些机制，比如后面会谈到的 `状态反馈` 中的 `confirm mode`。

`交换器`，`队列`，`消息` 这三者的持久化问题都介绍过了。前两者是一经声明，则其性质无法再被更改，即你不能先声明一个非持久化的队列，再声明一个持久化的同名队列，企图修改它，这是不行的。你重复声明时，相关参数需要一致。当然，你可以删除它们再重新声明：

```
channel.queue_delete(queue='hello')
channel.exchange_delete(exchange='first')
```

5. 调度策略

我们考虑交换器 `Exchange` 和队列 `Queue` 的关系。`Exchange` 在得到消息后会依据规则把消息投到一个或多个队列当中。

在调度策略方面，有两个需要了解的地方，一是交换器的类型(前面我们用的是 `fanout`)，二是交换器和队列的绑定关系。在绑定了的的前提下，我们再谈不同类型的交换器的规则。绑定动作本身也会影响交换器的行为。

交换器的类型，内置的有四种，分别是：

- `fanout`
- `direct`
- `topic`
- `headers`

下面一一介绍。

5.1. fanout

故名思义, **fanout** 类型的交换器, 其行为是把消息转发给所有绑定的队列上, 就是一个"广播"行为.

```
# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='first', type='fanout')
channel.queue_declare(queue='A')
channel.queue_declare(queue='B')
channel.queue_declare(queue='C')

channel.queue_bind(exchange='first', queue='A')
channel.queue_bind(exchange='first', queue='B')

channel.basic_publish(exchange='first',
                      routing_key='',
                      body='Hello World!')
```

运行 N 次, 通过 `rabbitmqctl` 可以看到 A 和 B 中就有 N 条消息, 而 C 中没有消息. 因为只有 A 和 B 是绑定到了 `first` 上的.

5.2. direct

direct 类型的行为是"先匹配, 再投送". 即在绑定时设定一个 **routing_key**, 消息的 **routing_key** 匹配时, 才会被交换器投送到绑定的队列中去.

```
# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='first', type='direct')
channel.queue_declare(queue='A')
channel.queue_declare(queue='B')

channel.queue_bind(exchange='first', queue='A', routing_key='a')
channel.queue_bind(exchange='first', queue='B', routing_key='b')

channel.basic_publish(exchange='first',
                      routing_key='a',
                      body='Hello World!')
```

A 和 B 虽然都绑定在了类型为 `direct` 的 `first` 上, 但是绑定时的 `routing_key` 不同.

当一个 `routing_key` 为 `a` 的消息出来时, 只会被 `first` 投送到 A 里.

5.3. topic

`topic` 和 `direct` 类似, 只是匹配上支持了"模式", 在"点分"的 `routing_key` 形式中, 可以使用两个通配符:

- `*` 表示一个词.
- `#` 表示零个或多个词.

代码:

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
  
channel.exchange_declare(exchange='first', type='topic')  
channel.queue_declare(queue='A')  
channel.queue_declare(queue='B')  
  
channel.queue_bind(exchange='first', queue='A', routing_key='a.*')  
channel.queue_bind(exchange='first', queue='B', routing_key='a.#')  
  
channel.basic_publish(exchange='first',  
                      routing_key='a',  
                      body='Hello World!')  
  
channel.basic_publish(exchange='first',  
                      routing_key='a.b.c',  
                      body='Hello World!')
```

在发出的两条消息当中, `a` 只会被 `a.#` 匹配到. 而 `a.b.c` 会被两个都匹配到.

所以, 最终的结果会是 A 中有一条消息, B 中有两条消息.

5.4. headers

`headers` 也是根据规则匹配, 相较于 `direct` 和 `topic` 固定地使用 `routing_key`, `headers` 则是一个自定义匹配规则的类型.

在队列与交换器绑定时，会设定一组键值对规则，消息中也包括一组键值对(`headers` 属性)，当这些键值对有一对，或全部匹配时，消息被投送到对应队列。

```
-----

# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='first', type='headers')
channel.queue_declare(queue='A')
channel.queue_declare(queue='B')

channel.queue_bind(exchange='first', queue='A', arguments={'a': '1'})
channel.queue_bind(exchange='first', queue='B', arguments={'b': '2', 'c': 3, 'x-match': 'all'})

channel.basic_publish(exchange='first',
                      routing_key='',
                      properties=pika.BasicProperties(
                          headers = {'a': '2'},
                      ),
                      body='Hello World!')

channel.basic_publish(exchange='first',
                      routing_key='',
                      properties=pika.BasicProperties(
                          headers = {'a': '1', 'b': '2'},
                      ),
                      body='Hello World!')
```

绑定时，通过 `arguments` 参数设定匹配规则，`x-match` 是一个特殊的规则，表示需要全部匹配上，还是只匹配一条：

- `all` ，全部匹配。
- `any` ，只匹配一个。

消息的 `headers` 属性会用于规则的匹配。

上面的代码中，第一条消息不会匹配任何规则。第二条消息，匹配到 `A` ，但是不会匹配到 `B` (虽然有一条 `b:2`)。

最终的结果是，`A` 中有一条消息，`B` 中没有消息。

6. 分配策略

调度策略是影响 `Exchange` 是不是要把消息给 `Queue` ，而分配策略影响队列如何把消息给 `Consuming` 。

考虑这样的场景：队列中有多条消息，每一个消费者取出消息后，都要花 10 秒来处理它，处理完一条消息之后才可能再取出一条继续处理。刚开始只有一个消费者，过了 2 秒后来了第二个消费者，此时，这两个消费者获取消息的行为是一个什么状态？

我们的需求可能是，当一个消费者来时，只给它一条消息，等它再“请求”时，再给。或者也可能是，当有消费者时，就把目前有的消息全给它（因为不知道是否还有其它的消费者，所以既然来了一个就让它尽量多处理一些消息）。

先产生一些等待处理的消息：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
  
channel.exchange_declare(exchange='first', type='fanout')  
channel.queue_declare(queue='A')  
  
channel.queue_bind(exchange='first', queue='A')  
  
for i in range(10):  
    channel.basic_publish(exchange='first',  
                          routing_key='',  
                          body=str(i))  
  
-----
```

然后是消费者的实现：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.queue_declare(queue='A')  
  
def callback(ch, method, properties, body):  
    import time  
    time.sleep(10)  
    print body  
  
channel.basic_consume(callback, queue='A', no_ack=True)  
channel.start_consuming()  
  
-----
```

上面的代码，是假设处理一条消息需要 10 秒的时间。但是事实上，你只要一执行代码，马上再使用 `rabbitmqctl` 查看队列状态时，会发现队列已经空了。因为在关闭 `ack` 的情

况下, **Queue** 的行为是, 一旦有消费者请求, 那么当前队列中的消息它都会一次性吐很多出去.

ack 机制在后面 **状态反馈** 会介绍到, 简单来说是一种确认消息被正确处理的机制.

如果我们想一次只吐一条消息, 当其它消费者连上来时, 还可以并行处理, 简单地把 **ack** 打开就可以了(默认就是打开的).

再考虑一下细节. 当有多个消费者连上时, 它是从队列一次取一条消息, 还是一次取多条消息(这样至少可以改善性能). 这可以通过配置 **channel** 的 **qos** 相关参数实现:

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.queue_declare(queue='A')  
channel.basic_qos(prefetch_count=2)  
  
def callback(ch, method, properties, body):  
    import time  
    time.sleep(10)  
    print body  
    ch.basic_ack(delivery_tag = method.delivery_tag)  
  
channel.basic_consume(callback, queue='A', no_ack=False)  
channel.start_consuming()  
  
-----
```

通过配置 **prefetch_count** 参数, 来设置一次从队列中取多少条消息. 要看到效果, 至少需要启 2 个消费者.

之前是 10 个数字按顺序入了队列, **channel** 的配置是一次取 2 个, 那么启 2 个消费者的话, 过 10 秒, 在两个消费者的输出中分别能看到 0, 2. 这时把两个消费者都 **Ctrl-C**, 通过 **rabbitmqctl** 能看到 **A** 队列中还有 8 条消息.

7. 状态反馈

状态反馈 的功能目的是为了确认行为的结果. 比如, 当你向 **Exchange** 提交一个消息时, 这个消息是否提交成功, 是否送达到了队列中. 当你从队列中提取消息之后, **RabbitMQ** 的 **Server** 如何处理, 因为在提取消息之后, **Consuming** 可能判断消息有问题, 可能在处理的过程中出现了异常.

在一些关键的节点上, 要保证消息的正确处理, 安全处理, 是需要很多细节上的控制的. **AMQP** 协议本身也为此作了相关设计, 甚至是事务机制. 事实上在 **AMQP** 中要确保消

信息的业务可靠性只能使用事务，不过在 **RabbitMQ** 中有一些相应的简便的扩展机制来达到同样目的。

7.1. 信息发布的确认

回看一下之前的一段代码：

```
channel.basic_publish(exchange='first', routing_key='', body='Hello World!')
```

这段代码要做的事，是把一条消息发给名为 **first** 的交换器。这个过程中可能出现意外：

- **exchange** 的名字写错了。
- **exchange** 得到消息后，发现没有对应的 **queue** 可以投送。
- 投送到 **queue** 后当前没有消费者来提取它。

上面的三种情况，第一种，会直接引发一个调用错误。第三种，通常不是问题，反正消息会在 **queue** 中暂存。但是第二种情况很多时候是需要避免的，否则消息就丢失了，更严重的是 **Producing** 对此浑然不知。

在这个地方，我们就需要确认消息发出之后，是否成功地被投送到 **queue** 中去了(或者知道它不能被投送到任何 **queue** 中去)。

要确认这些状态信息，首先需要把 **channel** 设置到 **confirm mode**，也称之为 **Publisher Acknowledgements** 机制(和消息的 **ack** 机制区分开)。它的目的就是为了确认 **Producing** 发出的信息的状态。

打开 **confirm mode** 的方法是：

```
channel.confirm_delivery()
```

之后的 **publish** 行为就可以收到服务器的反馈。比如在 **basic_publish** 函数中，通过 **mandatory=True** 参数来确认发出的消息是否有 **queue** 接收，并且所有 **queue** 都成功接收。

```
import pika
```

```
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()
```

```

channel.exchange_declare(exchange='first', type='fanout')
channel.queue_declare(queue='A')

#channel.queue_bind(exchange='first', queue='A')

channel.confirm_delivery()

r = channel.basic_publish(exchange='first',
                          routing_key='',
                          body='Hello',
                          mandatory=True,
                          )

print r

```

上面的代码中，因为名为 `first` 的 `Exchange` 没有绑定任何的 `queue`，在 `mandatory` 参数的作用下，`basic_publish` 会返回 `False`。

对于持久化性质，`confirm mode` 的确认结果是表示，一条 `persisting` 的消息，投送给一个 `durable` 的队列成功，并且数据已经成功写到磁盘。当然，因为系统缓存的问题，为确保数据成功落地，得到确认信息有时可能需要长达几百毫秒的时间，应用对此应该有所准备，而不至于在性能上受此影响。

7.2. 消息提取的确认

在未关闭消息的 `ack` 机制的情况下，当消息被 `Consuming` 从队列中提取后，在未明确获取确认信息之前，队列中的消息是不会被删除的。这样，流程上就变成，当消息被提取之后，队列中的这条消息处于“等待确认”的状态。如果 `Consuming` 反馈“成功”给队列，则消息可以安全地被删除了。如果反馈“拒绝”给队列，则消息可能还需要再次被其它 `Consuming` 提取。

看下面的例子，我们先创建顺序的 10 个数字为内容的 10 条消息：

```

# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='first', type='fanout')
channel.queue_declare(queue='A')

channel.queue_bind(exchange='first', queue='A')

for i in range(10):
    channel.basic_publish(exchange='first', routing_key='', body=str(i))

```

提取消息的逻辑：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
r = channel.basic_get(queue='A', no_ack=False) #0  
print r[-1], r[0].delivery_tag  
  
-----
```

上面的代码会提取第一条消息，但是并没有向 **Queue** 反馈此消息是否被正确处理，所以这条消息在队列中仍然存在，直到 **Connection** 被释放后，被提取过但是未被确认的消息的状态被重置，它就可以被重新提取。

要确认消息，或者拒绝消息，使用对应的 **basic_ack** 和 **basic_reject** 方法：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
r = channel.basic_get(queue='A', no_ack=False) #0  
print r[-1], r[0].delivery_tag  
#channel.basic_ack(delivery_tag=r[0].delivery_tag)  
channel.basic_reject(delivery_tag=r[0].delivery_tag)  
  
-----
```

AMQP 协议中，只提供了 **reject** 方法，它只能处理一条消息。因为 **Consuming** 是可以一次性提取多条消息的，所以 **RabbitMQ** 为此做了扩展，提供了 **basic_nack** 方法，它和 **basic_reject** 的唯一区别就是支持一次性拒绝多条消息。

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
r = channel.basic_get(queue='A', no_ack=False) #0  
r = channel.basic_get(queue='A', no_ack=False) #1  
r = channel.basic_get(queue='A', no_ack=False) #2  
channel.basic_nack(delivery_tag=r[0].delivery_tag, multiple=True)  
  
-----
```

`delivery_tag` 是在 `channel` 中的一个消息计数，每次消息提取行为都对应一个数字。`ack` 的 `multiple` 机制会自动把不大于指定 `delivery_tag` 的消息提取都 `reject` 掉。

在 `reject` 和 `ack` 中还有一个 `requeue` 参数，表示被拒绝的消息是否可以被重新分配。默认是 `True`。如果消息被 `reject` 之后，不希望再被其它的 `Consuming` 得到，可以把 `requeue` 参数设置成 `False`：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
r = channel.basic_get(queue='A', no_ack=False) #0  
channel.basic_ack(delivery_tag=r[0].delivery_tag, multiple=False, requeue=False)  
  
-----
```

`basic_consume` 和 `basic_get` 都是从指定 `queue` 中提取消息，前者是一个更高层的方法，还支持 `qos` 等。

8. 示例：多消费者，并行处理

这可能是最常遇到的一种场景了。消息产生之后堆到队列里，有多个消费者的 `worker` 来共同处理这些消息，以并行的方式提高处理效率。

这种场景在 `Exchange` 的类似选择上，不管是 `fanout` 或者是 `direct` 都可以实现。稍有不同在于，`fanout` 类型的话，你在一个 `exchange` 上就不要乱绑定队列。`direct` 类型的话，则是需要每条消息自己处理好 `routing_key`。

这里以 `fanout` 类型先创建一些消息到 `A` 这个队列中：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
  
channel.exchange_declare(exchange='first', type='fanout')  
channel.queue_declare(queue='A')  
  
channel.queue_bind(exchange='first', queue='A')  
  
for i in range(10):  
    channel.basic_publish(exchange='first', routing_key='', body=str(i))  
  
-----
```

消费者的实现没什么特别的:

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.queue_declare(queue='A')  
channel.basic_qos(prefetch_count=1)  
  
def callback(ch, method, properties, body):  
    print body  
    import time  
    time.sleep(4)  
    ch.basic_ack(delivery_tag = method.delivery_tag)  
  
channel.basic_consume(callback, queue='A')  
channel.start_consuming()  
  
-----
```

多个消费者直接跑就行了. `A` 中的消息会被多个 `Consuming` 提取处理.

9. 示例: 一条消息多种处理, 临时队列

`fanout` 典型的广播模式就是我们这里考虑的场景, 其它的还有像"发布/订阅"的模式也是这种. 就是一条消息, 最终会有多个消费者得到它(前面说的多消费者并行处理的场景, 是一条消息, 只会给到一个消费者).

实现上, 自然可以是当一个消费者被创建之后, 同时也创建一个自己的 `queue`, 然后绑定到指定的 `exchange` 上. 每个 `Consuming` 有自己的 `queue`, 那么于其自己做一套命名方法, 不如就忽略 `queue` 的名字, 让系统处理, 这就是 **临时队列**.

在声明队列时, 不指定名字:

```
-----  
  
r = channel.queue_declare()  
  
-----
```

系统会创建一个队列, 并且随机给一个类似于 `amq.gen-a_rJcuQ1mJigV9xp5G_uZQ` 这样的名字. 从返回的 `r` 中可以得到这个信息. 接下来, 就可以把这个队列绑定到 `exchange`.

但是还有一个问题, `Consuming` 自己创建了一个 `queue`, 那么在 `Consuming` 断开连接之后, 这个 `queue` 也是应该被销毁的. 自己在 `on_close` 之类的事件回调中处理

不是不过以，不过 **RabbitMQ** 有提供现成的机制，声明 **queue** 时使用 **exclusive=True** 即可：

```
-----  
  
r = channel.queue_declare(exclusive=True)
```

这样当连接断掉后，声明的 **queue** 会被自动删除(相应的 **bind** 关系也会取消)。

一个即插即用的 **Consuming** 就是：

```
-----  
  
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
channel.exchange_declare(exchange='first', type='fanout')  
r = channel.queue_declare(exclusive=True)  
channel.queue_bind(exchange='first', queue=r.method.queue)  
channel.basic_qos(prefetch_count=1)  
  
def callback(ch, method, properties, body):  
    print body  
    import time  
    time.sleep(4)  
    ch.basic_ack(delivery_tag = method.delivery_tag)  
  
channel.basic_consume(callback, queue=r.method.queue)  
channel.start_consuming()  
  
-----
```

特别之处只是动态创建 **queue** 。

这种情况下的 **Producing**，就只关注 **Exchange**，不关心 **queue** 了：

```
# -*- coding: utf-8 -*-  
  
import pika  
  
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))  
channel = connection.channel()  
  
channel.exchange_declare(exchange='first', type='fanout')  
  
for i in range(10):  
    channel.basic_publish(exchange='first', routing_key='', body=str(i))
```

10. 示例：发布订阅，多种形式的实现

"发布/订阅"的模式，在前面已经提过。简单地使用临时队列就可以实现。

不过在这里，我们多思考一点。"发布"的形式倒是单一，就是把消息提交到 `exchange`。但是"订阅"的行为，就可以有多种解释了。

最简单的创建一个临时队列，`bind` 到了 `exchange` 上，就算是"订阅了这个 `exchange`"。

其它，还可以针对 `direct` 或 `topic` 类型的 `exchange`，创建临时队列之后，`bind` 到 `exchange` 上时指定 `routing_key`，这可以说是"订阅了这个 `exchange` 中的某些消息"。`headers` 类型的 `exchange` 同理。

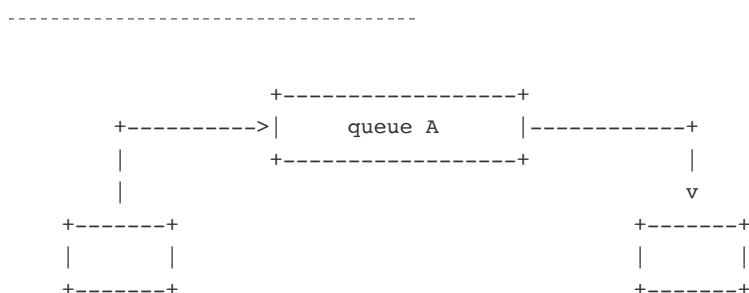
换个角度来看这个问题，是选择使用 `exchange` 来分割消息，还是使用 `routing_key` 来分割消息。前者在 `Producing` 阶段会比较麻烦，因为你需要往多个 `exchange` 提交消息。而后者在 `Producing` 和 `Consuming` 阶段都要多做一些事，`Producing` 阶段需要正确设定消息的 `routing_key`，在 `Consuming` 阶段 `bind` 时也需要正确设置 `routing_key`。更进一步说，我们在提交消息时，是愿意选择 `exchange`，还是更愿意考虑给一个合适的 `routing_key` 呢？

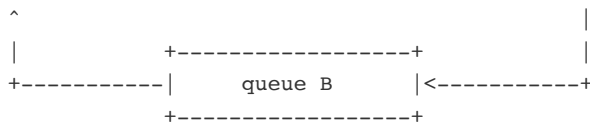
当然，`exchange` 和 `routing_key` 肯定不是矛盾的。它们是两个层面的抽象，彼此应该独立。在具体业务中使用时，也应该对应到合适的业务抽象层中去。就消息而言，如果是一个有多项目的大系统共用一个 `RabbitMQ` 服务，那在 `exchange` 这层可能就是"项目"的分割。而如果这种环境下你把 `exchange` 搞成"业务"的分割，情况就复杂了，我认为这是错误的设计。"面向数据而不是面向业务"的原则，在这里同样适用。

11. 示例：远程调用，信息流方向与角色转换

队列系统有一个很本质的东西，就是信息流的方向是单一的。信息总是被放进 `queue` 后，再被取出。

考虑远程调用的模型，"调用"本身是一个"请求/响应"的过程，这是两个方向的信息流。对应到队列中，两个方向，则至少需要两个队列。想明白了这点，我们要做的事就清楚了：





上图是我们要实现的信息流，左侧是调用方，要做的事是把参数写到 `queue A`，然后从 `queue B` 中取出结果。右侧是计算方，要做的事是从 `queue A` 中取出参数，运算后把结果写到 `queue B` 中。

计算方的代码：

```

# -*- coding: utf-8 -*-

import pika

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_r', type='fanout')
ch.queue_declare(queue='rpc_r')
ch.queue_declare(queue='rpc_p')
ch.queue_bind(exchange='rpc_r', queue='rpc_r')

def callback(channel, method, properties, body):
    print body
    s = sum(int(x) for x in body.split(','))
    channel.basic_publish(exchange='rpc_r', routing_key='', body=str(s))
    channel.basic_ack(delivery_tag = method.delivery_tag)

ch.basic_consume(callback, queue='rpc_p')
ch.start_consuming()

```

逻辑是从 `rpc_p` 中取出数据，计算后把结果写到 `rpc_r` 中。

调用方代码：

```

# -*- coding: utf-8 -*-

import pika

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_p', type='fanout')
ch.queue_declare(queue='rpc_p')
ch.queue_declare(queue='rpc_r')
ch.queue_bind(exchange='rpc_p', queue='rpc_p')

def callback(channel, method, properties, body):
    print body
    channel.basic_ack(delivery_tag = method.delivery_tag)

ch.basic_consume(callback, queue='rpc_r')

```

```
ch.basic_publish(exchange='rpc_p', routing_key='', body='1,2,3,4')
ch.start_consuming()
```

逻辑是把参数写到 `rpc_p` 中，然后等着从 `rpc_r` 中读出结果。

两段代码，实现了最简单的功能。但是这样有一个很明显的问题，调用和输出之间，是没有任何联系的，即函数调用的输入和输出之间无法对应起来。当有多个调用方时，就乱套了。所以我们需要改进一下，让输入和输出能一一对应上。

输入部分不用改，还是使用 `rpc_p` 保存参数。输出部分，我们把一个确定的 `queue` 换成每次生成的临时队列，并且把其对应的 `exchange` 改成 `headers` 类型，目的是通过 `headers` 参数实现，从 `rpc_p` 取出一组参数之后，往 `rpc_r` 写回的东西只路由到特定的一个临时队列去。

计算方代码：

```
# -*- coding: utf-8 -*-

import pika

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_r', type='headers')
ch.queue_declare(queue='rpc_p')

def callback(channel, method, properties, body):
    print body
    s = sum(int(x) for x in body.split(','))
    q = properties.headers['q']
    channel.basic_publish(exchange='rpc_r', routing_key='', body=str(s),
                          properties=pika.BasicProperties(
                              headers = {'q': q},
                          ))
    channel.basic_ack(delivery_tag = method.delivery_tag)

ch.basic_consume(callback, queue='rpc_p')
ch.start_consuming()
```

计算之后，往 `rpc_r` 中写回数据时，数据带上特殊的头，而特殊的头的值则是原始消息中自带的，指明了这组参数对应的临时队列。

调用方的代码改成：

```
# -*- coding: utf-8 -*-

import pika
```

```

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_p', type='fanout')
ch.exchange_declare(exchange='rpc_r', type='headers')
ch.queue_declare(queue='rpc_p')
ch.queue_bind(exchange='rpc_p', queue='rpc_p')

def callback(channel, method, properties, body):
    print body
    channel.basic_ack(delivery_tag = method.delivery_tag)

rq = ch.queue_declare(exclusive=True)
qname = rq.method.queue
ch.queue_bind(exchange='rpc_r', queue=qname, arguments={'q': qname})
ch.basic_consume(callback, queue=qname)
ch.basic_publish(exchange='rpc_p', routing_key='', body='1,2,3',
                  properties=pika.BasicProperties(
                      headers = {'q': qname},
                  ))
ch.start_consuming()

```

调用前生成一个用于保存结果的临时队列，把这个队列绑定到 `rpc_r` 这个交换器上，并且规定了一个路由规则。

然后把参数写到 `rpc_p` 时，同时也在数据中写入了结果队列的路由规则。

这是一个很常用的"先挖坑，再填坑"的方式。

同理，换成 `direct` 类型的 `exchange`，以 `routing_key` 保存路由规则也能实现类似的效果。

道理就是上面讲的，不过做同样的事，`RabbitMQ`，或者说 `AMQP` 中有一些事先定义参数可以直接拿来用。比如消息中的 `reply_to` 参数来标明此消息处理后的结果往哪个队列中送，`correlation_id` 来标明"请求"与"响应"的对应关系。

计算方实现：

```

# -*- coding: utf-8 -*-

import pika

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_r', type='direct')
ch.queue_declare(queue='rpc_p')

def callback(channel, method, properties, body):
    print body
    s = sum(int(x) for x in body.split(','))
    channel.basic_publish(exchange='rpc_r', routing_key=properties.reply_to, body=str(s),

```

```

        properties=pika.BasicProperties(correlation_id=properties.correlation_id))
    channel.basic_ack(delivery_tag = method.delivery_tag)

ch.basic_consume(callback, queue='rpc_p')
ch.start_consuming()

```

调用方实现：

```

# -*- coding: utf-8 -*-

import pika
import uuid

conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
ch = conn.channel()
ch.exchange_declare(exchange='rpc_p', type='fanout')
ch.exchange_declare(exchange='rpc_r', type='direct')
ch.queue_declare(queue='rpc_p')
ch.queue_bind(exchange='rpc_p', queue='rpc_p')

def callback(channel, method, properties, body):
    print body, properties.correlation_id
    channel.basic_ack(delivery_tag = method.delivery_tag)

rq = ch.queue_declare(exclusive=True)
qname = rq.method.queue
ch.queue_bind(exchange='rpc_r', queue=qname, routing_key=qname)
ch.basic_consume(callback, queue=qname)
ch.basic_publish(exchange='rpc_p', routing_key=qname, body='1,2,3',
                  properties=pika.BasicProperties(
                      reply_to = qname,
                      correlation_id = uuid.uuid4().hex
                  ))
ch.start_consuming()

```

12. 消息的BasicProperties

在 **AMQP** 协议中，为消息预定了 14 个属性，有些在前面我们已经用到过了，有些则本来就很少用到：

- **content_type** 标明消息的类型。
- **content_encoding** 标明消息的编码。
- **headers** 可扩展的信息对。
- **delivery_mode** 为 2 时表示该消息需要被持久化支持。
- **priority** 该消息的权重。
- **correlation_id** 用于"请求"与"响应"之间的匹配。
- **reply_to** "响应"的目标队列。

- `expiration` 有效期.
- `message_id` 消息的ID.
- `timestamp` 一个时间戳.
- `type` 消息的类型.
- `user_id` 用户的ID.
- `app_id` 应用的ID.
- `cluster_id` 服务集群ID.

13. pika在Tornado中的使用

前面的介绍中, 我们都使用的是 `BlockingConnection`, 是一个同步阻塞的交互模型. `pika` 的代码本身在组织时, 其实是以异步的方式来架构的, `BlockingConnection` 不过是一个特殊的封装而已.

`pika` 提供了多种异步调度机制的适配实现, 对应 Tornado 的是 `TornadoConnection`. 同时, 在 `channel` 的 API 上, 本身就有 `callback` 的实现, 所以, 把之前的代码改成异步的形式, 多是在获取 `Connection` 和 `channel` 这两步上的异步形式调整.

13.1. Producing

先看同步的代码:

```
# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='first', type='fanout')
channel.queue_declare(queue='A')
channel.queue_bind(exchange='first', queue='A')
channel.confirm_delivery()

r = channel.basic_publish(exchange='first', routing_key='', body='hello', mandatory=True)
print r
```

它做的事有:

- 获取连接.
- 从连接上获取 `channel`.

- 声明 `exchange` .
- 声明 `queue` .
- 绑定 `exchange` 和 `queue` .
- 打开 `confirm mode` .
- 发出消息.
- 确认结果.

除了前面两步："获取连接" 和 "从连接上获取 `channel`" , 其它的操作都是在 `channel` 上完成的. 对应到 Tornado 上来做, 就是:

```
-----
# -*- coding: utf-8 -*-

import pika
import tornado.gen
import tornado.ioloop

def on_confirm(method):
    print method

def on_publish((ch, method, prop, body)):
    print method.reply_code

@tornado.gen.engine
def on_channel(channel):
    method = yield tornado.gen.Task(channel.exchange_declare, exchange='first', type='fanout')
    method = yield tornado.gen.Task(channel.queue_declare, queue='A')
    #method = yield tornado.gen.Task(channel.queue_bind, exchange='first', queue='A')
    channel.confirm_delivery(callback=on_confirm, nowait=True)
    channel.add_on_return_callback(on_publish)
    channel.basic_publish(exchange='first', routing_key='', body='good', mandatory=True)
    #channel.basic_publish(exchange='first', routing_key='', body='good')
    #channel.close()

def on_connect(conn):
    conn.channel(on_open_callback=on_channel)

def publish():
    pika.TornadoConnection(pika.ConnectionParameters('localhost'),
                           on_open_callback=on_connect)

if __name__ == '__main__':
    publish()
    tornado.ioloop.IOLoop().current().start()
-----
```

代码好像会复杂一些, 不过继续封装一下用起来也可以很方便.

前面提到过, 好消息是 `channel` 的所有 API 都带有 `callback` 实现, 说的是 `channel` 有所有 API 几乎都有一个 `callback` 参数. 坏消息是, `callback` 参数全是第一个参数. 所以, 如果要直接使用 `tornado.gen.engine` 的话, 后面的参数需要全带上参数名以 `keyword` 形式传递.

从上面异步结构的代码中，更容易把 **RabbitMQ** 自己扩展实现的 **confirm mode** 说清楚了。

```
channel.confirm_delivery(callback=on_confirm, nowait=True)
```

是打开 **confirm mode** 功能，当向服务器 **publish** 一条消息，服务器确认消息之后，就会回调这里指定的函数。回调的内容要么是 **ACK**，要么是 **NACK**。按官方文档的说法，出现 **NACK** 的情况只可能是服务内部出现了错误。而正确的回调函数被执行时，意即服务器确认了消息内容，消息已经被所有对应的队列接收，如果是需要持久化支持的内容，则相关数据已经写到磁盘。

后面的：

```
channel.add_on_return_callback(on_publish)
```

是对应 **mandatory=True** 的回调的。即当 **publish** 出去的消息无法被投递到任何队列时，服务会回调这里的函数。

上面的 **confirm** 和 **on_return** 是两套东西。示例代码执行时，**mandatory=True** 生效的情况下，你会看到如下输入：

```
312
<METHOD(['channel_number=1', 'frame_type=1', "method=<Basic.Ack(['delivery_tag=1', 'multiple=False'])>"])>
```

先执行的是 **on_return** 的回调，再是 **confirm** 的回调。

还有一点，从：

```
channel.add_on_return_callback(on_publish)
```

这里也可以看出，**pika** 的 API 组织上，**channel** 中有很多的回调函数是单独定义的（可能有 **AMQP** 协议有关）。

```
add_callback(callback, replies, one_shot=True)
    在当前 channel 上注册指定类型的事件回调。
```

```
add_on_cancel_callback(callback)
    使用 basic_cancel 回调的函数.
```

```
add_on_close_callback(callback)
    channel 关闭时回调的函数.
```

```
add_on_return_callback(callback)
    basic_publish 中的消息被拒绝时的回调函数.
```

13.2. Consuming

Consuming 部分在异步环境上比 **Producing** 部分还简单一点. 最大的不同, 可能在于异步环境下需要自己控制对消息的提取(提取之后再次监听, 而不像同步环境下一个 **while**).

先看同步的代码:

```
-----

# -*- coding: utf-8 -*-

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='first', type='fanout')
r = channel.queue_declare(exclusive=True)
channel.queue_bind(exchange='first', queue=r.method.queue)
channel.basic_qos(prefetch_count=1)

def callback(ch, method, properties, body):
    print body
    import time
    time.sleep(4)
    ch.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_consume(callback, queue=r.method.queue)
channel.start_consuming()
```

异步代码:

```
-----

# -*- coding: utf-8 -*-

import pika
import tornado.gen
import tornado.ioloop
import functools

def on_msg(qname, ch, method, prop, body):
    print qname, body
    ch.basic_ack(delivery_tag = method.delivery_tag)
    ch.basic_consume(functools.partial(on_msg, qname), queue=qname)
```

```
@tornado.gen.engine
def on_channel(channel):
    method = yield tornado.gen.Task(channel.exchange_declare, exchange='first', type='fanout')
    method = yield tornado.gen.Task(channel.queue_declare)
    qname = method.method.queue
    method = yield tornado.gen.Task(channel.queue_bind, exchange='first', queue=qname)
    method = yield tornado.gen.Task(channel.basic_qos, prefetch_count=1)
    channel.basic_consume(functools.partial(on_msg, qname), queue=qname)

def on_connect(conn):
    conn.channel(on_open_callback=on_channel)

def consume():
    pika.TornadoConnection(pika.ConnectionParameters('localhost'),
                           on_open_callback=on_connect)

if __name__ == '__main__':
    consume()
    tornado.ioloop.IOLoop().current().start()
```

评论