RabbitMQ在分布式系统中的应用

Python程序员 5月19日

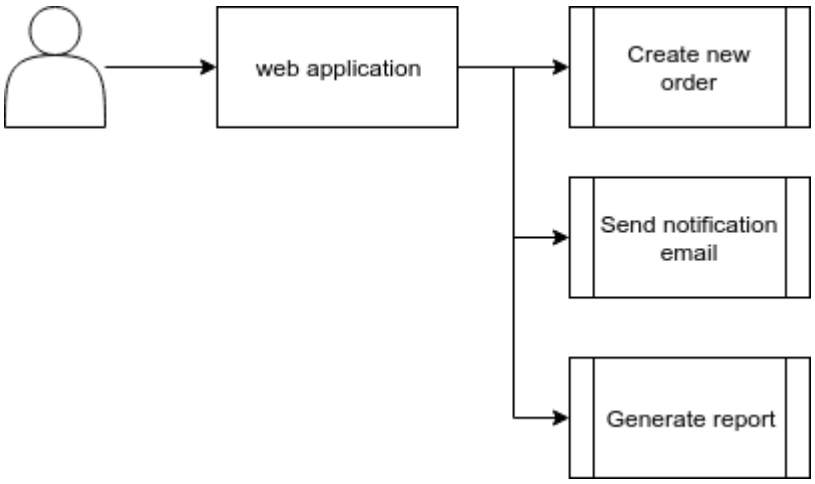
在本文中，我们首先要来探讨一下使用分布式系统的好处，以及如何借助RabbitMQ来迁移至分布式系统。然后我们也会学习到一些RabbitMQ的基本知识，最后会结合理论知识，学习一下如何用Python编程语言跟它进行交互。

分布式系统

我们先假设一下，设想我们正在做一个电商网站。有用户下订单了，除了要在数据库中新建一条记录之外，很显然我们还需要给用户发送一封邮件通知他们订单的详情以及一份报表，以便将来某些时候用户可能会用到。

如果画流程图的话，可能会是这样：

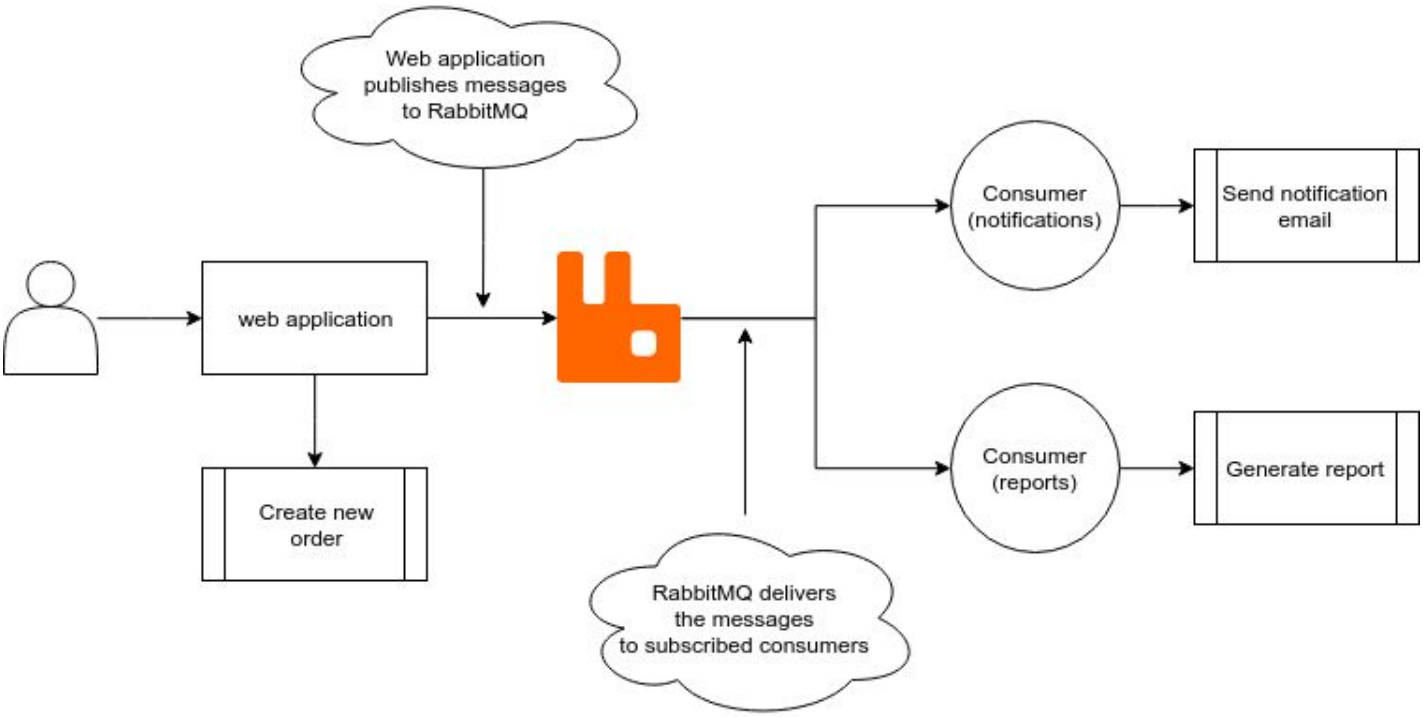
1.



但是，上述这个解决方案的问题在于发送邮件和生成报表都是非常耗时的任务。如果我们在一个请求/响应周期当中，使用同一个进程来处理这2个耗时任务，那么用户将会从服务器端等待比较长的时间。甚

至，你的应用服务将会变得很难扩展，因为越多用户向服务器发起请求，就要花越多的时间去处理这些请求。而且，一旦在处理请求上花费很多时间，那么就会给服务器增加负担，更坏的情况下，如果服务器处理的时间很长，那么服务器甚至会向用户返回一个请求超时的错误。

解决办法是让Web应用解耦。Web应用可以首先将消息发送给消息代理(message broker)，然后由消息代理将这些消息分发给能执行这些任务的消息消费者，这样一来，Web应用就不用亲自去执行这些任务了。



基本上，消费者是相互之间能独立分开工作的程序，并且一般情况下消费者程序来自web应用本身。而那些用来服务消费者的服务器，可以坐落在不同的地方。

除了能减轻服务器的压力之外，分布式系统的另外一个优势是即使其中一个应用挂了，整套系统仍然还是可以工作的。假如其中一个消费者无法给用户发送通知邮件了，那么我们可以把它停掉。即使我们的消费者挂了，我们的web应用仍然可以继续处理用户的请求并且给代理发送消息。一旦消费者恢复了，它马上就能接收到之前web应用发来的消息。

现在我们来看一下RabbitMQ ,它是一个在生产者(Web应用)和消费者之间的中间人。

RabbitMQ 要点知识

RabbitMQ 是一个消息代理。它实现了不同的协议，但是最重要的是，它实现了AMQP(高级消息队列协议)，AMQP是一个用来在多个系统之间通过生产者，代理以及消费者交换消息的协议。

AMQP是如何工作的

现在，我们有一个生产者和一个消费者。生产者产生消息，消费者消费消息。在它们二者之间我们还有一个代理，代理从生产者那里接收消息然后发送给消费者。

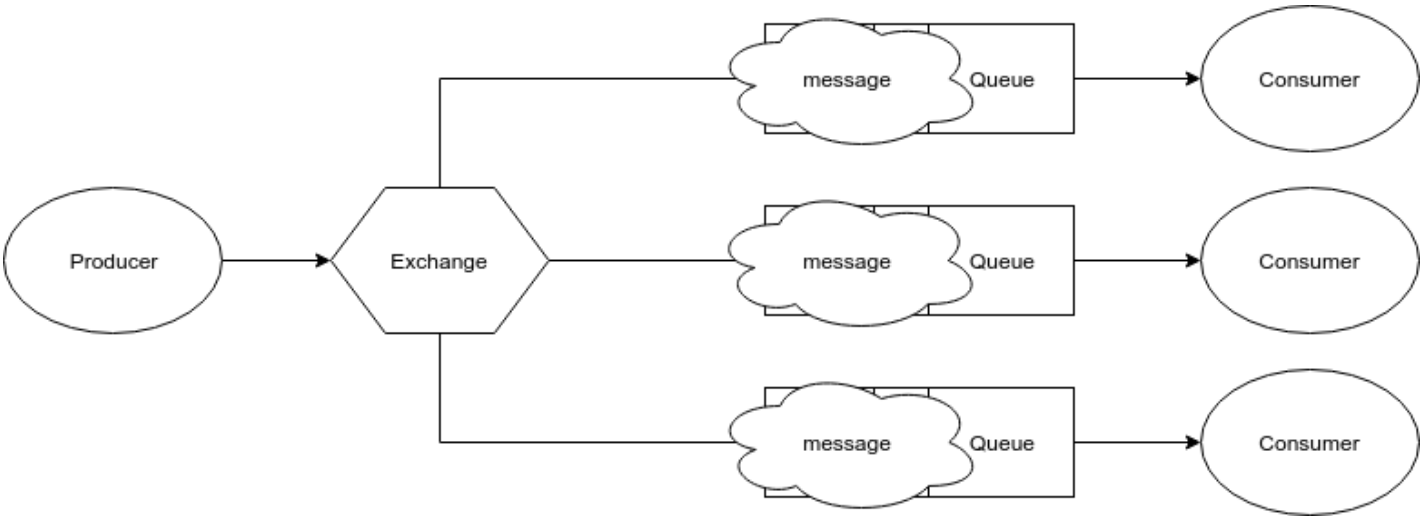
如果我们仔细研究下代理的工作原理，可能会有些难理解。代理由如下3个组件组成：

1. Exchange – 接受生产者发送的消息，并将消息路由给Queue。
2. Queue – 消息队列，一种磁盘或者内存中用来存储消息的数据结构；
3. Binding – 连接Exchange和Queue，它告诉Exchange消息应该被传送到哪个Queue。

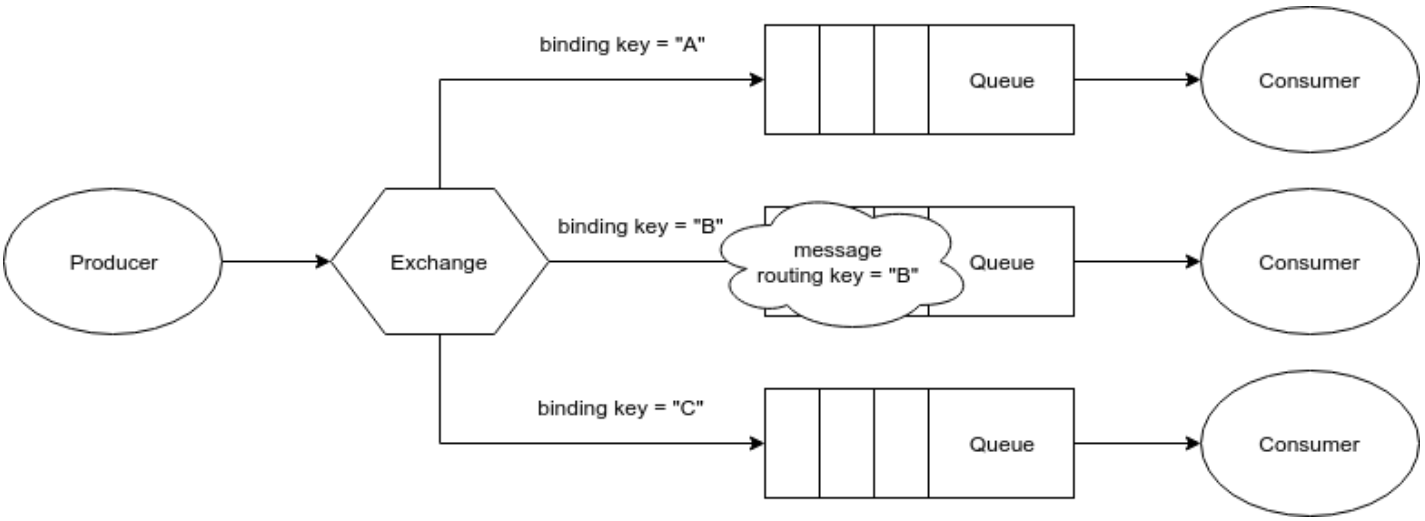
当创建Exchange时，我们会指定一个exchange类型。当创建一个binding用来连接一个exchange和一个队列时，我们会指定一个Binding key。 当发布一条消息时，我们会指定一个exchange和一个routing Key。 哪条消息会被发送给哪个queue，取决于下面这4个标准：

一共有4种类型的exchange:

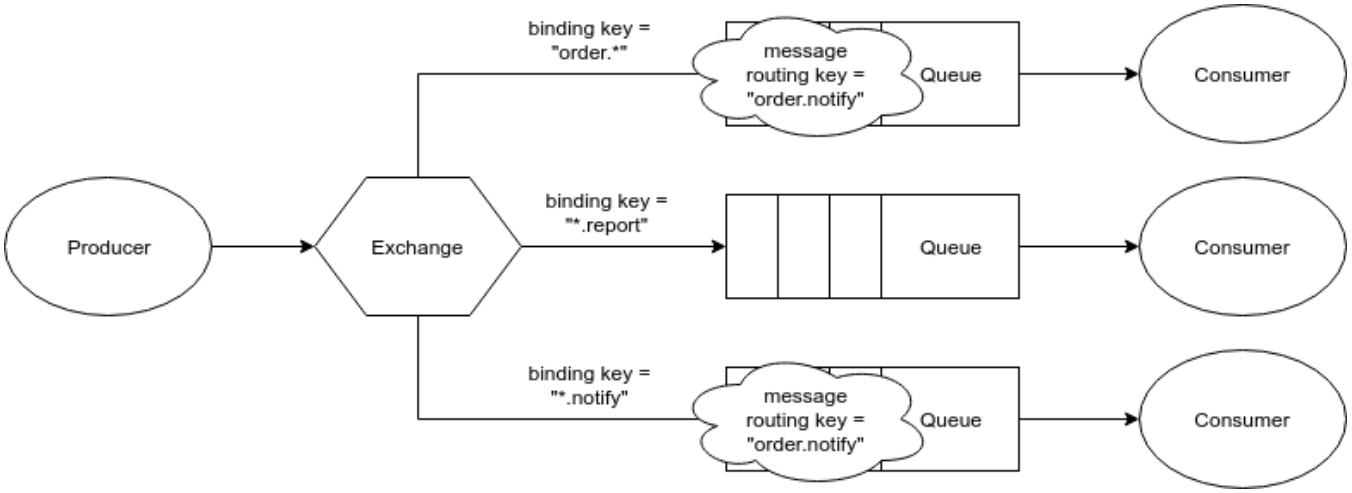
- Fanout. 这种类型的exchange只是简单地发送消息给它知道的所有队列。



- Direct. 这种类型的exchange会发送消息给符合routing key = binding key条件的队列。



- Topic. 这种类型的exchange发送消息给符合routing key能部分匹配binding key的队列。



- Header. 这种类型的exchange允许你跟你根据header的值来路由消息，而不是根据routing key。

最后，说点题外话，默认情况下RabbitMQ其实是有一个匿名的exchange。这个exchange会用队列的名字跟routing key做匹配，而不是binding key。所以，如果你发布一个routing key = “order”的消息到这个exchange，exchange将会路由这个消息到名为“order”的队列。

Python操作RabbitMQ

下面我来演示一下如何创建一个简单的Python程序。它可以帮助我们更好地理解生产者/代理/消费者这套流程。

我们会用到RabbitMQ客户端Python库Pika:

```
$ pip install pika
```

我们来声明一个类型为direct的exchange，然后发布几条消息到这个exchange中：

```python
# publish.py
import pika
import json
import uuid
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
channel.exchange_declare(
    exchange='order',
    exchange_type='direct'
)
order = {
    'id': str(uuid.uuid4()),
    'user_email': 'john.doe@example.com',
    'product': 'Leather Jacket',
    'quantity': 1
}
channel.basic_publish(
    exchange='order',
    routing_key='order.notify',
    body=json.dumps({'user_email': order['user_email']})
)
print(' [x] Sent notify message')
channel.basic_publish(
    exchange='order',
    routing_key='order.report',
    body=json.dumps(order)
)
print(' [x] Sent report message')
connection.close()
```

如果你跑一下这个脚本，你将得到下面结果：

```
[x] Sent notify message
[x] Sent report message

```

你可以通过在终端执行以下命令，来检查exchange是否真的创建成功：

```
$ sudo rabbitmqctl list_exchanges
Listing exchanges
...
order    direct

```

这个脚本里我们发送了2条消息。第一条消息的routing key = “order.notify”，第二条消息的routing key = “order.report”。不过这2条消息现在还不知道要发送到哪里，因为我们在exchange中并没有将它们绑定到任何的队列(queue)。

下面让我们来创建一个消费者，以便可以消费这些notify类型的消息：

```python
# notify.py
import pika
import json
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
queue = channel.queue_declare('order_notify')
queue_name = queue.method.queue
channel.queue_bind(
    exchange='order',
    queue=queue_name,
    routing_key='order.notify'  # binding key
)
def callback(ch, method, properties, body):
    payload = json.loads(body)
    print(' [x] Notifying {}'.format(payload['user_email']))
    print(' [x] Done')
    ch.basic_ack(delivery_tag=method.delivery_tag)
channel.basic_consume(callback, queue=queue_name)
print(' [*] Waiting for notify messages. To exit press CTRL+C')
channel.start_consuming()

```

首先我们定义了一个队列名为“order_notify”的queue，接着我们使用binding key = “order.notify”将其绑定到exchange。可以了，这样当我们发布一条routing key = “order.notify”的消息时，该条消息就会被发送给这条queue，然后我们就可以在callback回调函数里消费这条消息了。

可能最后一行的回调函数写得有些疑惑：

```python
def callback(ch, method, properties, body):
    ...
    ch.basic_ack(delivery_tag=method.delivery_tag)
    
```

这行回调函数代码向RabbitMQ发送了一个回执来告诉它消息已经接收并且被处理了，你RabbitMQ可以放心地将这条消息从你那边删除了。所以，如果一个消费者接收到了消息然后挂了，那样的话消息也不会丢。

首先，执行notify.py启动消费者：

```
$ python notify.py
```

接着，使用之前创建的脚本publish.py发布消息：

```
$ python publish.py

```

你将会在终端看到如下输出：

```
[x] Notifying john.doe@example.com
[x] Done
```

从输出可以看出，消息已经被成功消费掉了。

使用同样的方式，我们可以创建一个report类型的消费者：

```python
# report.py
import pika
import json
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
queue = channel.queue_declare('order_report')
queue_name = queue.method.queue
channel.queue_bind(
    exchange='order',
    queue=queue_name,
    routing_key='order.report'  # binding key
)
def callback(ch, method, properties, body):
    payload = json.loads(body)
    print(' [x] Generating report')
    print(f"""
    ID: {payload.get('id')}
    User Email: {payload.get('user_email')}
    Product: {payload.get('product')}
    Quantity: {payload.get('quantity')}
    """)
    print(' [x] Done')
    ch.basic_ack(delivery_tag=method.delivery_tag)
channel.basic_consume(callback, queue=queue_name)
print(' [*] Waiting for report messages. To exit press CTRL+C')
channel.start_consuming()
```

总结

使用RabbitMQ作为消息代理是一个很不错的选择。这篇文章里，我们学到了RabbitMQ的基础知识以及如何使用Pika库来跟它交互。但是在实际使用中，我们可能更愿意使用诸如Celery这类的库而不是Pika。所以，如果你愿意掌握地更升入一些，你不妨可以去找来研究研究。

> 英文原文：https://apirobot.me/posts/distributed-systems-with-rabbitmq
> 译者：疯禽忘肿