RabbitMQ

RabbitMQ 是由 LShift 提供的一个 Advanced Message Queuing Protocol (AMQP) 的开源实现，由以高性能、健壮以及可伸缩性出名的 Erlang 写成，因此也是继承了这些优点。
很成熟，久经考验，应用广泛
文档详细，客户端丰富，几乎常用语言都有 RabbitMQ 的开发库

安装

http://www.rabbitmq.com/install-rpm.html
选择 RPM 包下载，选择对应平台，我安装在 CentOS 6.5

Installing on RPM-based Linux (RHEL, CentOS, Fedora, openSUSE)

Download the Server

Description	Download	
RPM for RHEL 7.x, CentOS 7.x, Fedora 19+ (supports systemd, from Bintray)	rabbitmq-server-3.7.0-1.el7.noarch.rpm	(Signature)
RPM for RHEL Linux 6.x, CentOS 6.x, Fedora prior to 19 (from Bintray)	rabbitmq-server-3.7.0-1.el6.noarch.rpm	(Signature)
RPM for RHEL Linux 7.x, CentOS 7.x, Fedora 19+ (supports systemd, from GitHub)	rabbitmq-server-3.7.0-1.el7.noarch.rpm	(Signature)
RPM for RHEL Linux 6.x, CentOS 6.x, Fedora prior to 19 (from GitHub)	rabbitmq-server-3.7.0-1.el6.noarch.rpm	(Signature)
RPM for openSUSE Linux (from Bintray)	rabbitmq-server-3.7.0-1.suse.noarch.rpm	(Signature)
RPM for SLES 11.x (from Bintray)	rabbitmq-server-3.7.0-1.sles11.noarch.rpm	(Signature)

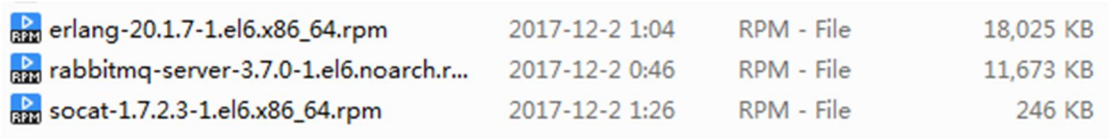
由于使用了 erlang 语言开发，所以需要 erlang 的包，该下载页给出了链接

Install zero-dependency Erlang from RabbitMQ

1. Download and install the zero dependency Erlang RPM package for running RabbitMQ. As the name suggests, the package strips off some Erlang modules and dependencies that are not essential for running RabbitMQ.

RabbitMQ 版本为 rabbitmq-server-3.7.0-1.el6.noarch.rpm，安装的时候提示 socat 缺失。

http://mirrors.yun-idc.com/epel/6/x86_64/socat-1.7.2.3-1.el6.x86_64.rpm



yum install socat-1.7.2.3-1.el6.x86_64.rpm erlang-20.1.7-1.el6.x86_64.rpm rabbitmq-server-3.7.0-1.el6.noarch.rpm
安装成功

```
[root@nodex ~]# rpm -ql rabbitmq-server
/etc/logrotate.d/rabbitmq-server
/etc/profile.d/rabbitmqctl-autocomplete.sh
/etc/rabbitmq
/etc/rc.d/init.d/rabbitmq-server
```

```
/usr/share/doc/rabbitmq-server-3.7.0/README
/usr/share/doc/rabbitmq-server-3.7.0/rabbitmq.config.example
/usr/share/doc/rabbitmq-server-3.7.0/set_rabbitmq_policy.sh.example
/usr/share/man/man5/rabbitmq-env.conf.5.gz
/usr/share/man/man8/rabbitmq-echopid.8.gz
/usr/share/man/man8/rabbitmq-plugins.8.gz
/usr/share/man/man8/rabbitmq-server.8.gz
/usr/share/man/man8/rabbitmq-service.8.gz
/usr/share/man/man8/rabbitmqctl.8.gz
/usr/share/zsh/vendor-functions/_enable_rabbitmqctl_completion
/var/lib/rabbitmq
/var/lib/rabbitmq/mnesia
/var/log/rabbitmq
```

配置

http://www.rabbitmq.com/configure.html#config-location

环境变量
使用系统环境变量，如果没有使用 rabbitmq-env.conf 中定义环境变量，否则使用缺省值
RABBITMQ_NODE_IP_ADDRESS the empty string, meaning that it should bind to all network interfaces.
RABBITMQ_NODE_PORT 5672
RABBITMQ_DIST_PORT RABBITMQ_NODE_PORT + 20000 内部节点和客户端工具通信用
RABBITMQ_CONFIG_FILE 配置文件路径默认为/etc/rabbitmq/rabbitmq
环境变量文件，可以不配置

工作特性配置文件

rabbitmq.config 配置文件

3.7 支持新旧两种配置文件格式

1、erlang 配置文件格式，为了兼容继续采用

```
[
  {rabbit, [{ssl_options, [{cacertfile,           "/path/to/testca/cacert.pem"},
                           {certfile,             "/path/to/server/cert.pem"},
                           {keyfile,              "/path/to/server/key.pem"},
                           {verify,               verify_peer},
                           {fail_if_no_peer_cert, true}]}]}
].
```

2、sysctl 格式，如果不需要兼容，RabbitMQ 鼓励使用。

```
ssl_options.cacertfile           = /path/to/testca/cacert.pem
ssl_options.certfile             = /path/to/server/cert.pem
ssl_options.keyfile              = /path/to/server/key.pem
ssl_options.verify               = verify_peer
ssl_options.fail_if_no_peer_cert = true
```

这个文件也可以不配置

插件管理

rabbitmq-plugins list

列出所有可用插件

启动 WEB 管理插件

rabbitmq-plugins enable rabbitmq_management

```
[root@nodex rabbitmq]# rabbitmq-plugins enable rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@nodex...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

started 3 plugins.
```

```
[root@nodex rabbitmq]# ss -tanl | grep 5672
LISTEN     0      128                          *:15672
LISTEN     0      128                         :::5672
LISTEN     0      128                          *:25672
```

开始登录 WEB 界面，http://ip:15672

用户管理

添加用户:

```
rabbitmqctl add_user username password
```

删除用户:

```
rabbitmqctl delete_user username
```

更改密码:

```
rabbitmqctl change_password username newpassword
```

设置权限 Tags，其实就是分配组

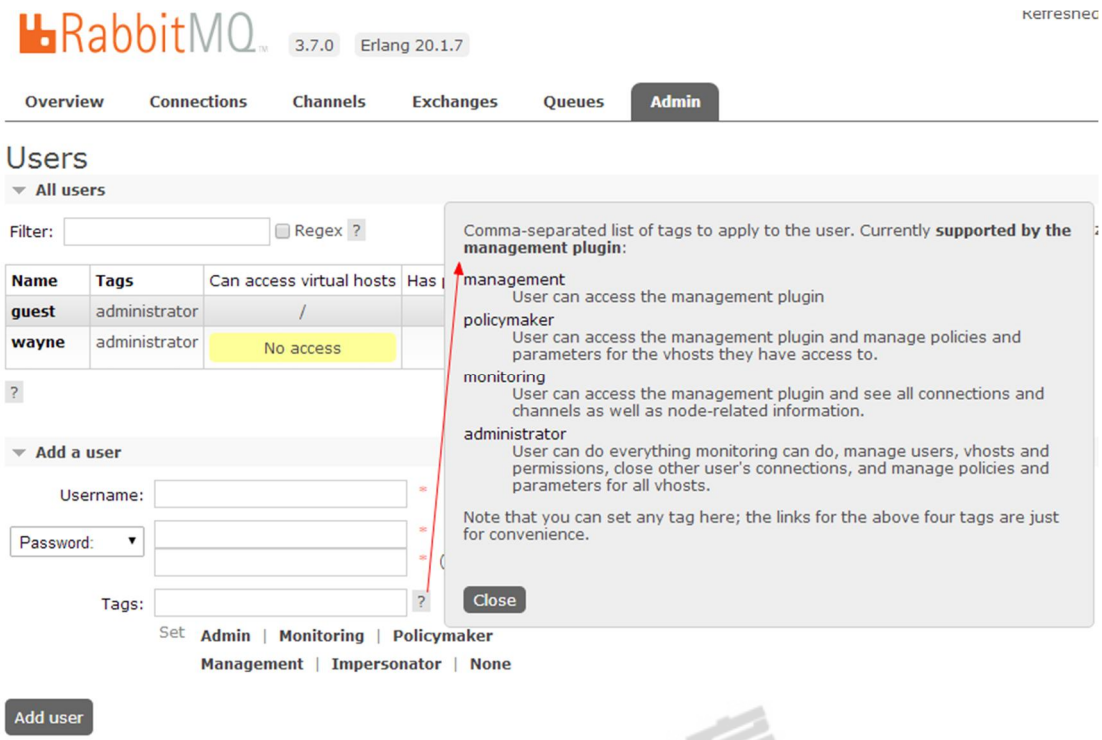
```
rabbitmqctl set_user_tags username tag
```

设置 wayne 用户

```
# rabbitmqctl add_user wayne wayne
# rabbitmqctl set_user_tags wayne administrator
```

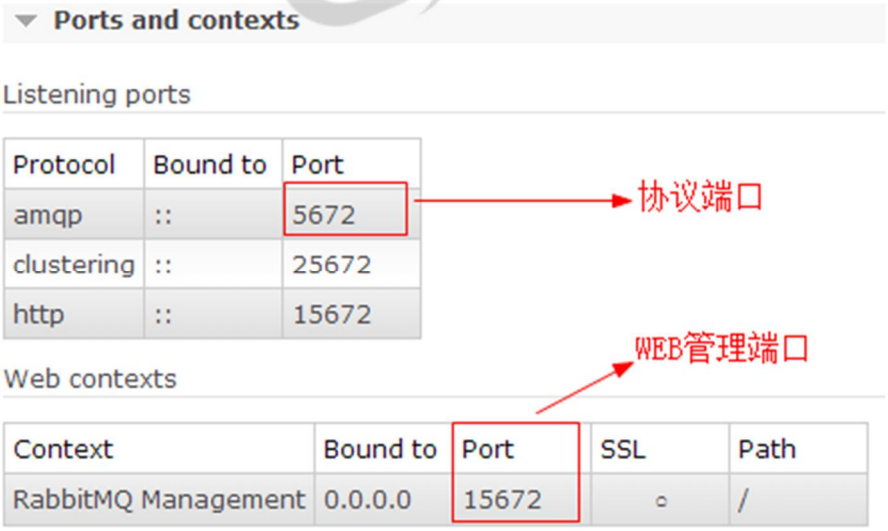
```
[root@nodex rabbitmq]# rabbitmqctl add_user wayne wayne
Adding user "wayne" ...
[root@nodex rabbitmq]# rabbitmqctl set_user_tags wayne administrator
Setting tags for user "wayne" to [administrator] ...
```

tag 的意义如下：

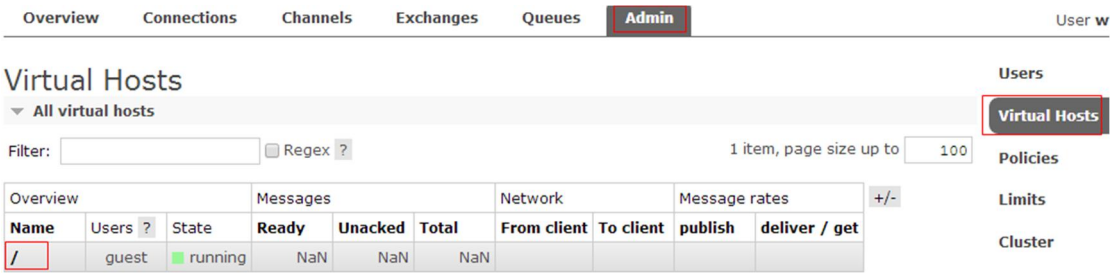


administrator 可以管理用户、权限、虚拟主机

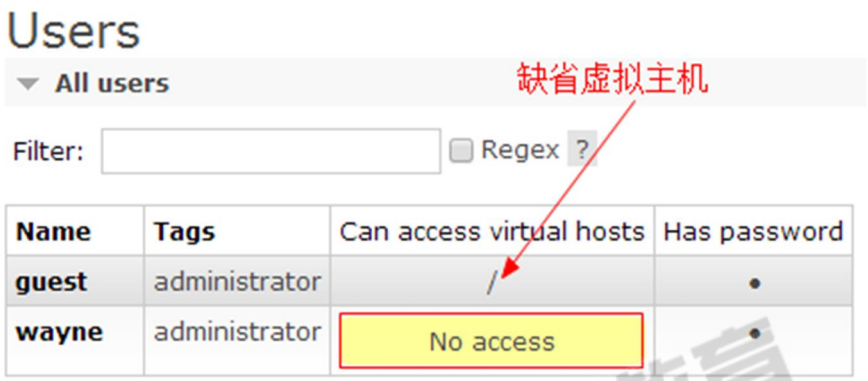
基本信息



虚拟主机



/为确实虚拟主机



缺省虚拟主机，默认只能是 guest 用户在本机连接。上图新建的用户 wayne 默认无法访问任何虚拟主机。

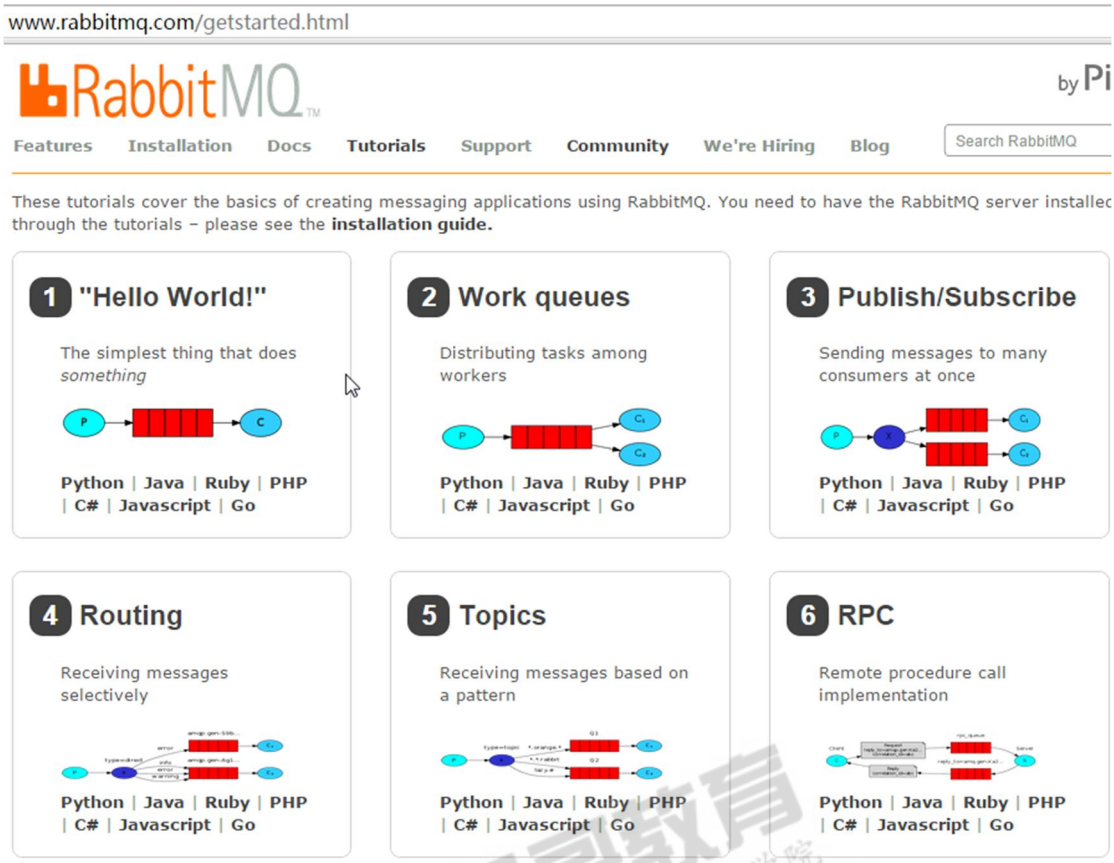
Python 库

Pika 是纯 Python 实现的支持 AMQP 协议的库

```
pip install pika
```

原理及应用

https://www.rabbitmq.com/getstarted.html



上图，列出了 RabbitMQ 的使用模式，学习上面的模式，对理解所有消息队列都很重要。

名词解释

Server：服务器，接收客户端连接，实现消息队列及路由功能的进程（服务），也称为 消息代理。注意，客户端包括生产者和消费者

Connection：网络物理连接

Channel：一个连接允许多个客户端连接

Exchange：交换器，接收生产者发来的消息，决定如何路由给服务器中的队列。常用的类型有：direct (point-to-point)、topic (publish-subscribe)、fanout (multicast)。

Message：消息

Message Queue：消息队列，数据的存储载体

Bind：绑定，建立消息队列和交换器之间的关系，也就是说交换器拿到数据，把什么样的数据送给哪个队列

Virtual Host：虚拟主机，一批交换机、消息队列和相关对象的集合。为了多用户互不干扰，使用虚拟主机分组交换机、消息队列

Topic：主题，

Broke：可等价为 Server

队列

这种模式就是最简单的 生产者消费者模型，消息队列就是一个 FIFO 的队列

生产者 send.py，消费者 receive.py
官方例子 https://www.rabbitmq.com/tutorials/tutorial-one-python.html
参照官方例子，写一个程序

send.py

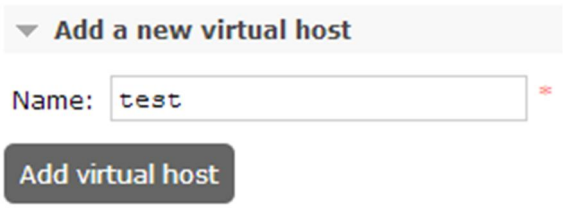
```
import pika


# 建立连接
params = pika.ConnectionParameters('192.168.142.135')
connection = pika.BlockingConnection(params)
channel = connection.channel()


# queue 命名为 hello
channel.queue_declare(queue='hello')


channel.basic_publish(exchange='', # 缺省 exchange，routing_key 必须指定
                      routing_key='hello',
                      body='Hello World')
print(" [x] Sent 'Hello World!'")


connection.close()
```

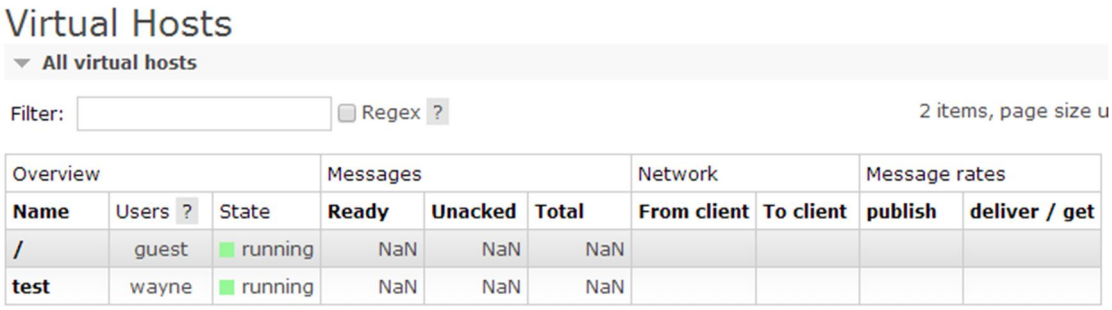
运行结果如下

```
pika.exceptions.ProbableAuthenticationError: (403, 'ACCESS_REFUSED - Login was refused using authentication mechanism PLAIN. For details see the broker logfile.')
```

查看了日志，也是一样，还是说 guest 用户只能访问 localhost 上的/虚拟主机

WEB 管理中的虚拟主机
缺省虚拟主机，默认只能在本机访问，不要修改为远程访问，是安全的考虑。
因此，新建一个虚拟主机





修改代码

```
import pika


# 建立连接
credentials = pika.PlainCredentials('wayne', 'wayne')
params = pika.ConnectionParameters(
    '192.168.142.135',
    5672,
    'test',
    credentials # 用户名、密码
)
connection = pika.BlockingConnection(params)
channel = connection.channel()


# queue 命名为 hello
channel.queue_declare(queue='hello')



channel.basic_publish(exchange='', # 缺省 exchange，routing_key 必须指定
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
```

```
connection.close()
```
测试通过

URLParameters
使用 URL 创建参数
```
amqp://username:password@host:port/<virtual_host>[?query-string]
parameters = pika.URLParameters('amqp://guest:guest@rabbit-server1:5672/%2F')
```
%2F 指代/，就是缺省虚拟主机

send.py
```
import pika

# 阻塞连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()

# queue 命名为 hello
channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', # 缺省 exchange，routing_key 必须指定
                      routing_key='hello',
                      body='welcom magedu')
print(" [x] Sent 'Hello World!'")

connection.close()
```

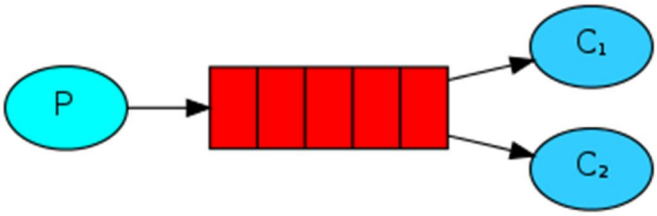
queue_declare 声明一个 queue，有必要的话，创建它。
basic_publish exchange 为空就使用缺省 exchange，如果找不到指定的 exchange，抛异常
使用缺省 exchange，就必须指定 routing_key，使用它找到 queue

生产者代码做一些改动
send.py 生产者
```
import pika
import time
```

```
# 阻塞连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()


# queue 命名为 hello
channel.queue_declare(queue='hello')


for i in range(40):
    channel.basic_publish(exchange='', # 缺省 exchange，routing_key 必须指定
                          routing_key='hello',
                          body="data{:02}".format(i))
    time.sleep(2)


print(" [x] Sent 'Hello World!'")


connection.close()


receive.py 消费者代码
import pika


# 阻塞连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()


# queue 命名为 hello，必须。没有就创建
channel.queue_declare(queue='hello')


# 回调函数，接收到数据如何处理
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
```

```
#
channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)
# 循环取队列数据
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming() # Ctl + C 退出
```

工作队列



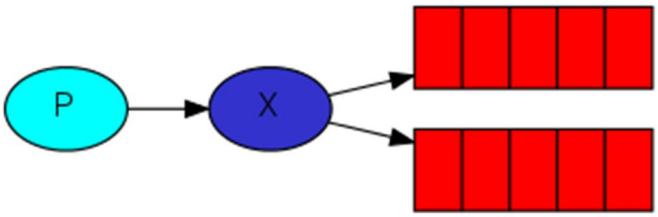
把 recieve.py 复制一份，再启动一个消费者。观察结果

这种工作模式是一种竞争工作方式，对某一个消息来说，只能有一个消费者拿走它。
但是从结果知道，其实使用的是轮询方式拿走数据的。
尝试再增加一个消费者，试试看是不是轮询的。

注意：虽然上面的图中没有画出 exchange，但是用到缺省 exchange。

发布、订阅模式 Publish/Subscribe

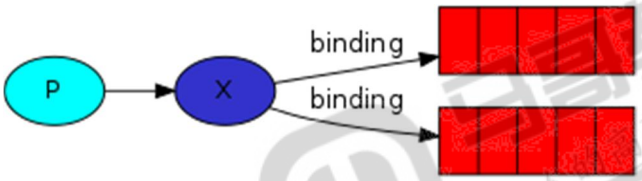
想象一下订阅报纸，所有订阅者（消费者）订阅这个报纸（消息），都应该拿到一份同样内容的报纸。
订阅者和消费者之间还有一个 exchange，可以想象成邮局，消费者去邮局订阅报纸，报社发报纸到邮局，邮局决定如何投递到消费者手中。



当前模式的 exchange 的 type 是 fanout，就是一对多，即广播模式。

```
channel.exchange_declare(exchange='logs',
                         exchange_type='fanout')
```

```
result = channel.queue_declare() # 生成一个随机名称的 queue
result = channel.queue_declare(exclusive=True) # 生成一个随机名称的 queue，并在断开连接时删除 queue
可以通过 result.method.queue 查看随机名称
```



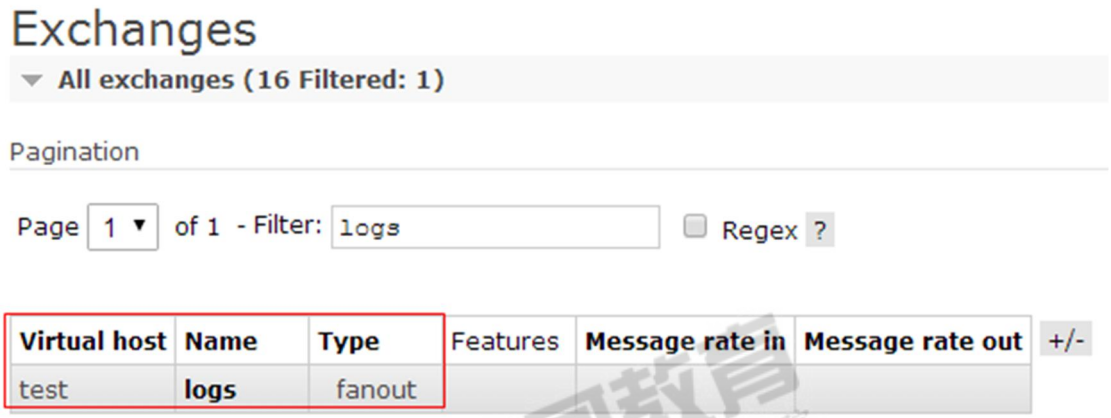
绑定 Bingding，建立 exchange 和 queue 之间的联系

```
send.py
import pika
import time


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange='logs', # 不是用默认 exchange
                         exchange_type='fanout') # 广播


for i in range(40):
    msg = "pc-data{:02}".format(i)
    channel.basic_publish(exchange='logs', # 发送到指定 exchange
                          routing_key='', # fanout 不指定
```

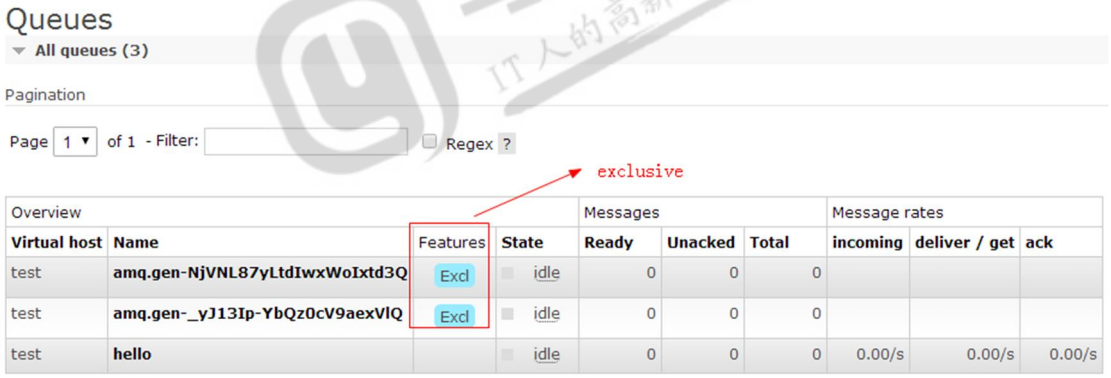
```
                            body=msg)
    print(msg)
    time.sleep(2)


connection.close()


receive.py
import pika


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange='logs',
                         exchange_type='fanout')


# 生成 queue
q1 = channel.queue_declare(exclusive=True)
q2 = channel.queue_declare(exclusive=True)
q1name = q1.method.queue
q2name = q2.method.queue
print(q1name, q2name)


# 绑定
channel.queue_bind(exchange='logs', queue=q1name)
channel.queue_bind(exchange='logs', queue=q2name)


# 回调函数，接收到数据如何处理
def callback(ch, method, properties, body):
    print(ch, method, properties, body)
    print(" [x] Received %r" % body)


#
channel.basic_consume(callback, queue=q1name, no_ack=True)
channel.basic_consume(callback, queue=q2name, no_ack=True)
```

```
# 循环取队列数据
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming() # Ctl + C 退出
```

先启动消费者可以看到已经创建了 exchange



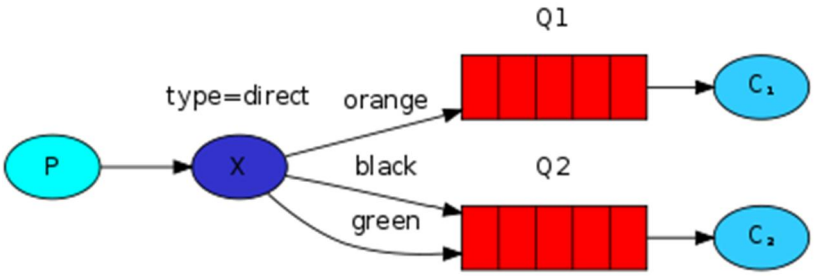
如果 exchange 是 fanout，也就是广播了，routing_key 就无所谓是什么了。




```
q1 = channel.queue_declare(exclusive=True)
q2 = channel.queue_declare(exclusive=True)
```

尝试先启动生产者，再启动消费者试试看。
部分数据丢失，因为，exchange 收到了数据，没有 queue 接收，所以，exchange 丢弃了这些数据。

路由 Routing



路由其实就是数据经过 exchange 的时候，通过匹配规则，决定数据的去向。

send.py

```
import pika
import time
import random


exchangename = 'color'
colors = ('orange', 'black', 'green')


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange=exchangename, # 不是用默认 exchange
                         exchange_type='direct') # 路由


for i in range(40):
    rk = colors[random.randint(0,2)]
    msg = "pc-data{:02}".format(i)
    channel.basic_publish(exchange=exchangename, # 发送到指定 exchange
                          routing_key=rk, # 指定 routing_key
                          body=msg)
    print(rk, msg)
    time.sleep(2)


connection.close()
```

receive.py

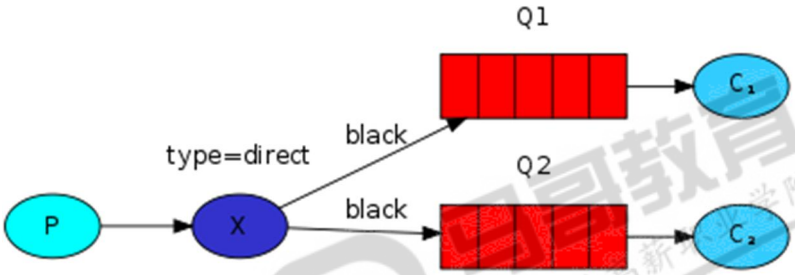
```
import pika


exchangename = 'color'
colors = ('orange', 'black', 'green')


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange=exchangename,
                         exchange_type='direct')


# 生成 queue
q1 = channel.queue_declare(exclusive=True)
q2 = channel.queue_declare(exclusive=True)
q1name = q1.method.queue
q2name = q2.method.queue
print(q1name, q2name)


# 绑定，一定要指定 routing_key
channel.queue_bind(exchange=exchangename, queue=q1name, routing_key=colors[0])
channel.queue_bind(exchange=exchangename, queue=q2name, routing_key=colors[1])
channel.queue_bind(exchange=exchangename, queue=q2name, routing_key=colors[2])


# 回调函数，接收到数据如何处理
def callback(ch, method, properties, body):
    print(" [x] %r:%r" % (method.routing_key, body))


# 消费
channel.basic_consume(callback, queue=q1name, no_ack=True)
channel.basic_consume(callback, queue=q2name, no_ack=True)


# 循环取队列数据
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming() # Ctl + C 退出
```

绑定结果如下

To	Routing key	Arguments	
amq.gen-7r6yIi3zU5640REm-VZqYw	black		Unbind
amq.gen-7r6yIi3zU5640REm-VZqYw	green		Unbind
amq.gen-x_zH0Y4z3xvjtT_dlkuKGw	orange		Unbind

思考

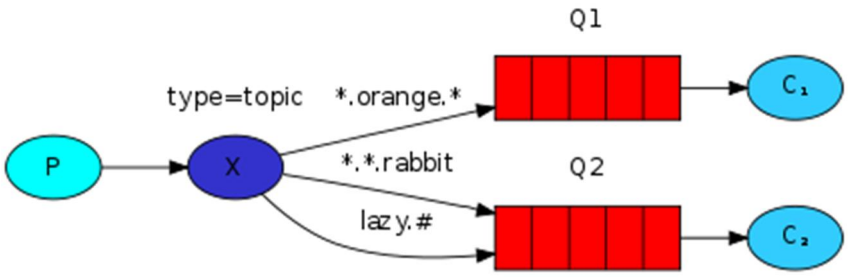
如果 routing_key 设置的都一样，会怎么样？



绑定的时候指定的 routing_key='black'，如上图，和 fanout 就类似了，但是不同。

因为 fanout 时，exchange 不做数据过滤的。

Topic 话题



Topic 的 routing_key 必须使用. 点号分割的单词组成

支持使用通配符：

* 表示严格的一个单词

#表示 0 个或者多个单词

如果 queue 绑定的 routing_key 只是一个#，这个 queue 其实可以接收所有的消息，类似于 fanout。
如果没有使用任何通配符，效果类似于 direct。

send.py

```
import pika
import time
import random


exchangename = 'products'
topics = ('phone.*', '*.red')
producttype = ('phone', 'pc', 'tv')
colors = ('red', 'green', 'blue')


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange=exchangename, # 不是用默认 exchange
                         exchange_type='topic') # Topic


for i in range(40):
    rk =
"{}.{}".format(producttype[random.randint(0,2)],colors[random.randint(0,2)])
    msg = "{} {:02}".format(rk, i)
    channel.basic_publish(exchange=exchangename, # 发送到指定 exchange
                          routing_key=rk, # 指定 routing_key
                          body=msg)
    print(msg)
    time.sleep(2)


connection.close()
```

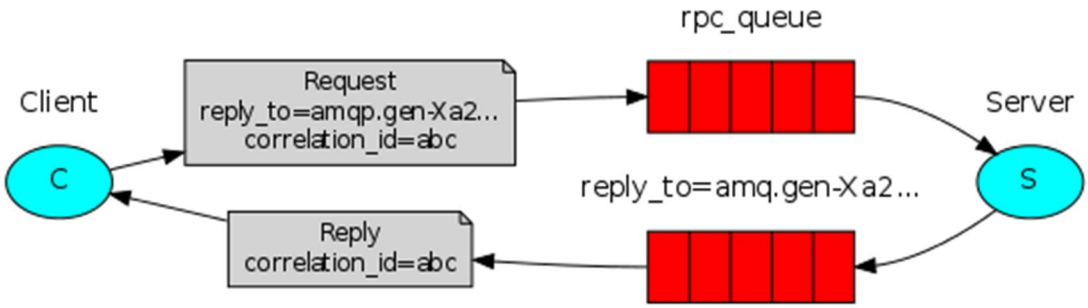
receive.py

```
import pika


exchangename = 'products'
```

```
topics = ('phone.*', '*.red')
producttype = ('phone', 'pc', 'tv')
colors = ('red', 'green', 'blue')


# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange=exchangename, # 不是用默认 exchange
                         exchange_type='topic') # Topic


# 生成 queue
q1 = channel.queue_declare(exclusive=True)
q2 = channel.queue_declare(exclusive=True)
q1name = q1.method.queue
q2name = q2.method.queue
print(q1name, q2name)


# 绑定，一定要指定 routing_key
channel.queue_bind(exchange=exchangename, queue=q1name, routing_key=topics[0])
channel.queue_bind(exchange=exchangename, queue=q2name, routing_key=topics[1])


# 回调函数，接收到数据如何处理
def callback(ch, method, properties, body):
    print(" [x] %r:%r" % (method.routing_key, body))


# 消费
channel.basic_consume(callback, queue=q1name, no_ack=True)
channel.basic_consume(callback, queue=q2name, no_ack=True)


# 循环取队列数据
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming() # Ctl + C 退出
```

Topic 其实就是更加高级的路由，支持模式匹配而已。

RPC 远程过程调用



Client 发送数据到 rpc_queue，Server 消费。
Server 把响应数据发送到 reply_to 关联的 queue，Client 消费。

server.py

```
import pika



# 创建连接
params = pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
connection = pika.BlockingConnection(params)
channel = connection.channel()
# rpc 使用缺省 exchange


# 生成 queue
q = channel.queue_declare(queue='rpc_queue')


# 业务处理，累加
def add(data):
    sum = 0
    for x in data:
        try:
            sum += int(x)
        except:
            pass
    return sum
```

```
# 回调函数，接收到数据如何处理
def on_request(ch, method, props, body):
    res = add(body.split()) # 业务处理
    print(" [x] %r" % body)


    # 响应，即写入另一个队列
    ch.basic_publish(exchange='',
                     routing_key=props.reply_to,
                     properties=pika.BasicProperties(
                         correlation_id=props.correlation_id),
                     body=str(res))
    ch.basic_ack(delivery_tag=method.delivery_tag)


# 消费
channel.basic_consume(on_request, queue='rpc_queue')




# 循环取队列数据
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming() # Ctl + C 退出


client.py PRC 客户端
import pika
import uuid


class RpcClient:
    def __init__(self):


        # 创建连接
        params =
pika.URLParameters('amqp://wayne:wayne@192.168.142.135:5672/test')
        self.connection = pika.BlockingConnection(params)
        self.channel = self.connection.channel()
        # rpc 使用缺省 exchange


        # 响应队列
```

```
        q = self.channel.queue_declare(exclusive=True)
        self.response_queue = q.method.queue
        self.channel.basic_consume(self.on_response, no_ack=True,
queue=self.response_queue)


        self.corr_id = None
        self.response = None


    # 回调函数，处理 Server 的响应数据
    def on_response(self, ch, method, props, body):
        if self.corr_id == props.correlation_id:
            print(" [x] reponse = %r" % body)
            self.response = body


    # send 发送数据
    def send(self, *data):
        msg = " ".join(map(str, data))
        self.response = None
        self.corr_id = uuid.uuid4().hex # 关联 ID
        self.channel.basic_publish(exchange='', # 缺省 exchange
                        routing_key='rpc_queue', # queue 名称
                        properties=pika.BasicProperties(
                            reply_to=self.response_queue, # 响应队列
                            correlation_id=self.corr_id
                        ),
                        body=msg)
        while self.response is None:
            self.connection.process_data_events()
        return self.response


client = RpcClient()
res = client.send(1,3,5)
res = client.send(2,4,6)
```

RPC 的应用场景较少，因为有更好的 RPC 通信框架。

消息队列的作用

1、系统间解耦

2、解决生产者、消费者速度匹配

由于稍微上规模的项目都会分层、分模块开发，模块间或系统间尽量不要直接耦合，需要开放公共接口供别的模块或系统调用，而调用可能触发并发问题，为了缓冲和解耦，往往采用中间件技术。

RabbitMQ 只是消息中间件中的一种应用程序。