

RabbitMQ

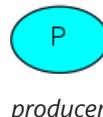
1. Hello World

The simplest thing that does something

RabbitMQ相当于message经纪人，它负责接收和转发信息，可以看作是邮局。当你将邮件投递到信箱的时候，你可以确信最终收信人可以收到邮件。

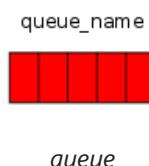
1. 生产者

生产只意味着发送，发送消息的程序是生产者。



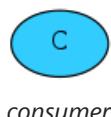
2. 队列

队列在RabbitMQ中相当于邮箱。尽管信息流经过RabbitMQ和你的程序，但是它们只能存储在队列中。队列大小只受主机的内存和硬盘限制，它本质上是一个大的信息缓存区。可以有很多生产者发送信息到同一个队列，也可以有很多消费者从一个队列中接收数据，下面是我们表示队列的符号。



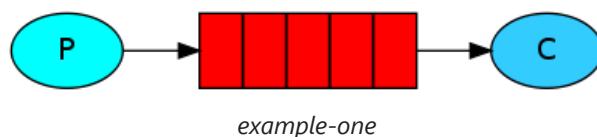
3. 消费者

消费相当于接收，消费者就是一个等待接收信息的程序。

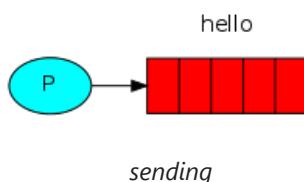


请注意，生产者、消费者，和broker不一定都位于同一台主机上。实际上，大多数应用程序都不会在同一台主机上。有时候，应用程序既可以是生产者，也可以是消费者。

下面我们将会使用C#编写两段代码，一段是生产者发送信息，另一段是消费者接收信息并打印出来。如下图所示，“P”是生产者，“C”是消费者，中间的是队列-消息缓冲区。



Sending

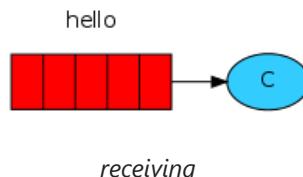


```
1 using System;
2 using RabbitMQ.Client;
3 using System.Text;
4
```

```

5  class Send
6  {
7      public static void Main()
8      {
9          var factory = new ConnectionFactory() { HostName = "localhost" };
10         using(var connection = factory.CreateConnection())
11         using(var channel = connection.CreateModel())
12         {
13             channel.QueueDeclare(queue: "hello",
14                                 durable: false,
15                                 exclusive: false,
16                                 autoDelete: false,
17                                 arguments: null);
18
19             string message = "Hello World!";
20             var body = Encoding.UTF8.GetBytes(message);
21
22             channel.BasicPublish(exchange: "",
23                                 routingKey: "hello",
24                                 basicProperties: null,
25                                 body: body);
26             Console.WriteLine(" [x] Sent {0}", message);
27         }
28
29         Console.WriteLine(" Press [enter] to exit.");
30         Console.ReadLine();
31     }
32 }
```

Receiving



```

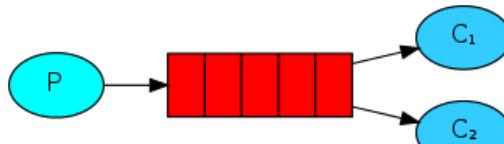
1  using RabbitMQ.Client;
2  using RabbitMQ.Client.Events;
3  using System;
4  using System.Text;
5
6  class Receive
7  {
8      public static void Main()
9      {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using(var connection = factory.CreateConnection())
12         using(var channel = connection.CreateModel())
13         {
14             channel.QueueDeclare(queue: "hello",
15                                 durable: false,
16                                 exclusive: false,
17                                 autoDelete: false,
```

```

18                         arguments: null);
19
20             var consumer = new EventingBasicConsumer(channel);
21             consumer.Received += (model, ea) =>
22             {
23                 var body = ea.Body.ToArray();
24                 var message = Encoding.UTF8.GetString(body);
25                 Console.WriteLine(" [x] Received {0}", message);
26             };
27             channel.BasicConsume(queue: "hello",
28                                 autoAck: true,
29                                 consumer: consumer);
30
31             Console.WriteLine(" Press [enter] to exit.");
32             Console.ReadLine();
33         }
34     }
35 }
```

2. Work queues

Distributing tasks among workers



example-two

在前面例子，我们可以使用程序在队列中发送和接收信息。现在我们将使用Work Queue 将耗时的任务分给多个消费者。这个主要思想是避免执行资源密集型任务，而且要等待任务完成。相反，我们在任务安排在后面完成。我们将任务封装为消息并将其发送到队列。后台运行的工作进程将弹出任务并最终执行作业。当你运行多个工作进程时任务将在它们之间共享。

这个概念在web应用程序中特别有用，因为在短的HTTP请求窗口中不可能处理复杂的任务

Round-robin dispatching 循环调度

使用任务队列的优点之一是能够轻松地并行处理工作。如果我们正在建立一个工作积压，我们可以增加更多地工人来扩大规模。

- 首先我们同时运行两个Worker实例，它们都从队列中获取信息。
- 你需要打开三个工作台，其中两个运行Worker程序，这就是我们的消费者C1和C2。
- 默认情况下，RabbitMQ将按顺序将每条消息发给下一个使用者。平均而言，每个消费者都会收到相同数量的消息。这种分发消息的方式称为循环。

消息确认

- 完成一项任务可能需要几秒钟，你可以想象如果一个消费者开始了一个很长的任务当只完成一部分就死掉了。在我们当前的代码中，一旦RabbitMQ向用户发送了一条信息，它就会立即将其标记为删除。在这种情况下，如果你杀死一个工人，我们将失去它正在处理的信息。我们还将丢失所有发送给这个特定工作者但尚未处理的信息。
- 但我们不想失去任何任务。如果一个worker死了，我们希望任务被交付给另外一个工人。
- 为了确保信息不会丢失，RabbitMQ支持消息确认。一个ack(nowledgement)被消费者发送回来，告诉RabbitMQ已经接收到一个特定消息，并被处理，RabbitMQ可以自由删除它。
- 如果一个消费者死了(它的通道被关闭，连接被关闭，或者TCP连接丢失)而没有发送一个ack，RabbitMQ将会理解一个消息没有被完全处理，并将它重新排队。如果同时有其他消费者在线，它将迅速将其重新发送给另一个消费者。这样你就可以确保没有信息丢失，即使workers偶尔死亡。
- 在消费者交付确认时强制执行超时(默认为30分钟)，这有助于检测从不确认交付的有bug(卡住)的消费者。你可以根据交付确认超时时间中描述来增加此超时时间。

- 默认情况下，手动消息确认是打开的。在前面的例子中，我们通过将autoAck("自动确认模式")参数设置为true显示地关闭它们。一旦我们完成了一个任务，现在是时候删除这个标志并手动从worker发送一个适当的确认了。

容易发生错误：

错失BasicAck是一个常见的错误。这是一个很容易犯的错误，但后果是严重的。当你的客户端退出时，消息会被重新传递(看起来像是随机的重新传递)，但是RabbitMQ会消耗越来越多的内存，因为它不能释放任何未被释放的消息。

为了调试这种错误，你可以使用rabbitmqctl 打印messages_unacknowledge字段：

```
sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

在Windows,去掉sudo,

```
rabbitmqctl.bat list_queues name messages_ready messages_unacknowledged
```

Message durability 消息持久化

- 我们已经学会了如何确保即使消费者死亡，任务也不会丢失。但是如果RabbitMQ服务器停止，我们的任务仍然会丢失。
- 当RabbitMQ退出或崩溃时，它会忘记队列和消息，除非你告诉它不要这样做。要确保消息不丢失，需要做两件事情：我们需要将队列和消息都标记为持久的。
- 首先，我们需要确保队列在RabbitMQ节点重启后能够存活。为了做到这一点，我们需要声明它时耐用的：

```
1 channel.QueueDeclare(queue: "hello",
2                      durable: true,
3                      exclusive: false,
4                      autoDelete: false,
5                      arguments: null);
```

- 尽管这个命令本身是正确的，但它在我们当中的设置无法工作。这是因为我们已经定义了一个名为hello的队列，它不是持久的。RabbitMQ不允许你用不同的参数重新定义一个现有的队列，并且会向任何尝试这样做的程序返回一个错误。但是有一个快速的解决方法—让我们用不同的名字来声明一个队列，例如task_queue:

```
1 channel.QueueDeclare(queue: "task_queue",
2                      durable: true,
3                      exclusive: false,
4                      autoDelete: false,
5                      arguments: null);
```

- 这个QueueDeclare更改需要同时应用于生产者和消费者代码。你还需要更改BasicConsume和BasicPublish的名称。
- 此时，我们可以确定task_queue队列不会丢失，即使RabbitMQ重新启动。现在我们需要将消息标记为持久消息。
- 在已有的GetBytes之后，设置IBasicProperties。持久为真：

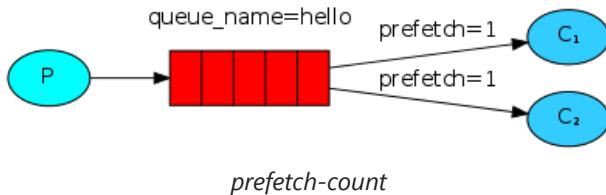
```
1 var body = Encoding.UTF8.GetBytes(message);
2
3 var properties = channel.CreateBasicProperties();
4 properties.Persistent = true;
```

需要注意：

将消息设置为持久性并不能完全保证消息不会消失。尽管它告诉RabbitMQ将消息保存在磁盘上，但当RabbitMQ接受了消息但还没有保存它时，仍然有一个很短的时间窗口。此外，RabbitMQ并不会对每条消息执行fsync(2)操作—它可能只是被保存到缓存中，而不是真正写入磁盘。持久性保证并不强，但对于简单的任务队列来说已经足够了。如果你需要更有力的保证，你可以使用发行商确认。

Fair Dispatch 公平调度

- 你可能已经注意到，调度仍然不能完全按照我们想要的方式工作。例如，在有两个工作人员的情况下，当所有奇数消息都很重而偶数消息很轻时，一个工作人员将一直很忙，而另一个几乎不做任何工作。好吧，RabbitMQ对此一无所知，仍然会均匀地分布消息。
- 这是因为RabbitMQ只是在消息进入队列时分派消息。它不查看用户未确认消息的数量。它只是盲目地将第n条消息发送给第n个消费者。



- 为了改变这种行为，我们可以使用带有prefetchCount=1设置的BasicQos方法。这告诉RabbitMQ一次不要给一个worker多个消息。或者，换句话说，在一个worker处理并确认了前一条消息之前，不要向它分派一条新消息。相反，它将把它分派给下一个不忙碌的worker。
- 在Work.cs中已有的QueueDeclare之后，添加对BasicQos的调用：

```

1 channel.QueueDeclare(queue: "task_queue",
2                     durable: true,
3                     exclusive: false,
4                     autoDelete: false,
5                     arguments: null);
6
7 channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);

```

注意队列大小：

如果所有的工人都很忙，你的队伍就会排满。你会想要密切关注这一点，可能会增加更多的workers，或者采取其他策略

首先运行消费者，这样拓扑(主要是队列)就位：

```

1
2 using System;
3 using RabbitMQ.Client;
4 using RabbitMQ.Client.Events;
5 using System.Text;
6 using System.Threading;
7
8 class Worker
9 {
10     public static void Main()
11     {
12         var factory = new ConnectionFactory() { HostName = "localhost" };
13         using(var connection = factory.CreateConnection())
14             using(var channel = connection.CreateModel())
15             {
16                 channel.QueueDeclare(queue: "task_queue",
17                     durable: true,
18                     exclusive: false,
19                     autoDelete: false,
20                     arguments: null);
21
22                 channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);
23
24                 Console.WriteLine(" [*] Waiting for messages.");
25
26                 var consumer = new EventingBasicConsumer(channel);
27                 consumer.Received += (sender, ea) =>
28                 {
29                     var body = ea.Body.ToArray();
30                     var message = Encoding.UTF8.GetString(body);

```

```

31     Console.WriteLine(" [x] Received {0}", message);
32
33     int dots = message.Split('.').Length - 1;
34     Thread.Sleep(dots * 1000);
35
36     Console.WriteLine(" [x] Done");
37
38     // Note: it is possible to access the channel via
39     //        ((EventingBasicConsumer)sender).Model here
40     channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);
41 };
42 channel.BasicConsume(queue: "task_queue",
43                      autoAck: false,
44                      consumer: consumer);
45
46     Console.WriteLine(" Press [enter] to exit.");
47     Console.ReadLine();
48 }
49 }
50 }
```

然后运行任务的发布器：

```

1
2     using System;
3     using RabbitMQ.Client;
4     using System.Text;
5
6     class NewTask
7     {
8         public static void Main(string[] args)
9         {
10             var factory = new ConnectionFactory() { HostName = "localhost" };
11             using(var connection = factory.CreateConnection())
12             using(var channel = connection.CreateModel())
13             {
14                 channel.QueueDeclare(queue: "task_queue",
15                                     durable: true,
16                                     exclusive: false,
17                                     autoDelete: false,
18                                     arguments: null);
19
20                 var message = GetMessage(args);
21                 var body = Encoding.UTF8.GetBytes(message);
22
23                 var properties = channel.CreateBasicProperties();
24                 properties.Persistent = true;
25
26                 channel.BasicPublish(exchange: "",
27                                     routingKey: "task_queue",
28                                     basicProperties: properties,
29                                     body: body);
30                 Console.WriteLine(" [x] Sent {0}", message);
31             }
32 }
```

```

32
33     Console.WriteLine(" Press [enter] to exit.");
34     Console.ReadLine();
35 }
36
37 private static string GetMessage(string[] args)
38 {
39     return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");
40 }
41 }
```

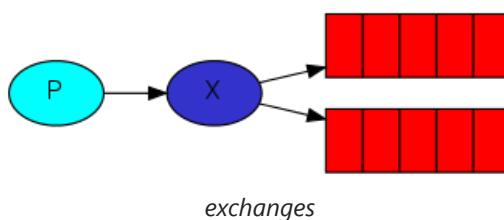
3. Public/Subscribe

Sending messages to many consumers at once

- 在上一篇中，我们创建了一个工作队列。工作队列背后的假设是，每个任务恰好交付给一个工作人员。在这一部分中，我们将做一些完全不同的事情，我们将向多个消费者传递消息。这种模式成为“发布/订阅”。
- 为了说明该模式，我们将构建一个简单的日志系统。它将由两个程序组成—第一个程序将发出日志信息，第二个程序将接收和打印它们。
- 在为我们的日志系统中，接收程序的每个运行副本都将获得消息。这样我们就可以运行一个接收器并将日志定向到磁盘上；与此同时，我们可以运行另一个接收器，在屏幕上看到日志。实际上，发布的日志消息将被广播到所有的接收器。

Exchanges 交换机

- 在前面所介绍的是向队列发送和从队列接收信息。现在是时候在Rabbit中介绍完整的消息传递模型了。
- 先来回顾一下之前所介绍的内容：
 - 生产者是发送消息的用户应用程序。
 - 队列是存储消息的缓存区。
 - 消费者是接收消息的用户应用程序
- RabbitMQ中消息传递模型的核心思想是生产者从不直接向队列发送任何消息。实际上，生产者通常甚至不知道消息是否会被传递到任何队列。
- 相反，生产者只向交换机发送消息。交换机是一件非常简单的事情。一边它从生产者接收信息，另一边它将消息推入队列。exchange必须确切地知道如何处理它收到的信息。是否应该将它附加到特定的队列中？它应该被附加到许多队列吗？或者它应该被丢弃吗？这些规则由交换机类型来定义。



- 这里有几种可用的exchange类型：direct, topic, headers和fanout。我们将关注最后一个—fanout。让我们创建一个这种类型的exchange，并名为“logs”。


```
channel.ExchangeDeclare("logs", ExchangeType.Fanout);
```
 - fanout exchange 非常简单。正如你可能从名称中猜到的那样，它只是将它收到的所有消息广播到它所知道的所有队列。这正是我们所需要的。
 - 命令行列出exchanges


```
sudo rabbitmqctl list_exchanges
```
 - 在这个列表中会有一些amq.* 和默认的(未命名的)exchange。它们是默认创建的，但目前暂时不使用它们。
- The default exchange**
- 之前，我们对交换机一无所知，但仍然能够向队列发送消息。这是可能的，因为我们使用的默认exchange，通过空字符串("")进行标识。回想一下我们之前是这样发布消息的：

```

1 var message = GetMessage(args);
2 var body = Encoding.UTF8.GetBytes(message);
```

```

3     channel.BasicPublish(exchange: "",
4                         routingKey: "hello",
5                         basicProperties: null,
6                         body: body);

```

第一个参数是exchange的名称。空字符串表示默认的或无名的exchange：消息被路由到具有由routingKey指定的名称的队列(如果它存在的话)。

现在我们可以发布我们的命名的exchange：

```

1 var message = GetMessage(args);
2 var body = Encoding.UTF8.GetBytes(message);
3 channel.BasicPublish(exchange: "logs",
4                      routingKey: "",
5                      basicProperties: null,
6                      body: body);

```

Temporary queues 临时队列

你可能还记得，前面我们使用了具有特定名称的队列(hello和task_queue)。能够命名一个队列对我们来说非常重要—我们需要将workers指向同一个队列。当你希望在生产者和消费者之间共享队列时，为队列指定一个名称非常重要。

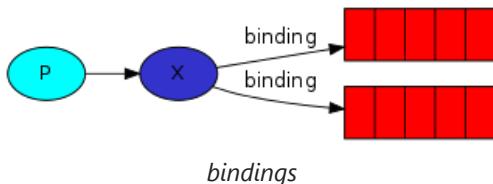
但我们的记录器不是这样，我们想要了解所有的日志消息，而不仅仅是其中的一个子集。我们也只对当前流动的消息感兴趣，而对旧的消息不感兴趣。要解决这个问题，我们需要两个条件：

1. 首先，每当我们连接到Rabbit时，我们需要一个新的空队列。为此，我们客户可以创建一个具有随机名称的队列，或者更好的方法是让服务器为我们选择一个随机的队列名称。
 2. 其次，一旦断开消费者的连接，队列就会被自动删除。
- 在.NET客户端中，当我们不向QueueDeclare()提供参数时，我们创建了一个非持久的，独占的，带有生成名称的，可自动删除的队列。

```
var queueName = channel.QueueDeclare().QueueName;
```

此时，queueName包含一个随机队列名。例如，它可能看起来像amq.gen-JzTY20BRgKO-HjmUj0wLg。

Bindings 绑定



- 我们已经创建了fanout exchange和队列。现在我们需要告诉exchange将消息发送到我们的队列。exchange和队列的这种关系我们成为绑定。

```

1 channel.QueueBind(queue: queueName,
2                    exchange: "logs",
3                    routingKey: "");

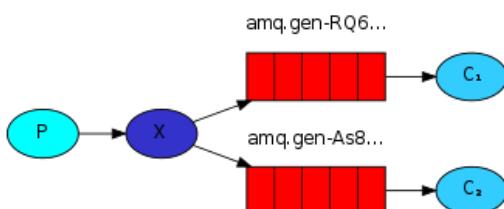
```

使用上面代码，我们就能将logs的exchange向我们的队列中添加消息。

查询指令：

```
rabbitmqctl list_bindings
```

Putting it all together 综合起来



example-three-overall

发出日志信息的生产者和之前的没有太大区别。最重要的变化是，我们现在希望将消息发布到logs交换中，而不是匿名exchanges。我们需要在发送时提供**routingKey**，但是对于fanout，它的值会被忽略。

发送者：

1. 在建立连接后，我们声明了exchange，这一步是必要的，因为禁止向不存在的exchange发布。
2. 如果还没有队列绑定到交换器，则消息将丢失，但对我们来说没有问题。如果没有消费者在listening，我们可以安全地丢弃消息。

```
1  using System;
2  using RabbitMQ.Client;
3  using System.Text;
4
5  class EmitLog
6  {
7      public static void Main(string[] args)
8      {
9          var factory = new ConnectionFactory() { HostName = "localhost" };
10         using(var connection = factory.CreateConnection())
11         using(var channel = connection.CreateModel())
12         {
13             channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);
14
15             var message = GetMessage(args);
16             var body = Encoding.UTF8.GetBytes(message);
17             channel.BasicPublish(exchange: "logs",
18                                 routingKey: "",
19                                 basicProperties: null,
20                                 body: body);
21             Console.WriteLine(" [x] Sent {0}", message);
22         }
23
24         Console.WriteLine(" Press [enter] to exit.");
25         Console.ReadLine();
26     }
27
28     private static string GetMessage(string[] args)
29     {
30         return ((args.Length > 0)
31                 ? string.Join(" ", args)
32                 : "info: Hello World!");
33     }
34 }
```

接收者：

```
1  using System;
2  using RabbitMQ.Client;
3  using RabbitMQ.Client.Events;
4  using System.Text;
5
6  class ReceiveLogs
7  {
```

```

8  public static void Main()
9  {
10     var factory = new ConnectionFactory() { HostName = "localhost" };
11     using(var connection = factory.CreateConnection())
12     using(var channel = connection.CreateModel())
13     {
14         channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);
15
16         var queueName = channel.QueueDeclare().QueueName;
17         channel.QueueBind(queue: queueName,
18                           exchange: "logs",
19                           routingKey: "");
20
21         Console.WriteLine(" [*] Waiting for logs.");
22
23         var consumer = new EventingBasicConsumer(channel);
24         consumer.Received += (model, ea) =>
25         {
26             var body = ea.Body.ToArray();
27             var message = Encoding.UTF8.GetString(body);
28             Console.WriteLine(" [x] {0}", message);
29         };
30         channel.BasicConsume(queue: queueName,
31                               autoAck: true,
32                               consumer: consumer);
33
34         Console.WriteLine(" Press [enter] to exit.");
35         Console.ReadLine();
36     }
37 }
38 }
```

4.Routing

Receiving messages selectively

- 在上一节，我们构建了一个简单的日志系统，我们能够向许多消费者广播日志信息。
- 在本节，我们将向它添加一个特性—我们将使它能够只订阅一个子集。例如，我们将能够只将关键的错误信息定向到日志文件(以节省磁盘空间)，同时仍然能够在控制台打印所有的日志消息。

Bindings 绑定

- 在上一节，我们已经创建了绑定，你可能会想起这样的代码：

```

1  channel.QueueBind(queue: queueName,
2                     exchange: "logs",
3                     routingKey: "");
```

- 绑定是exchange和队列之间的一种关系。这可以简单地理解为：队列对来自此exchange的消息刚兴趣。
- 绑定可以接受一个额外的routingKey参数。为了避免与BasicPublish参数混淆，我们将其称为**binding key**—“routingKey”

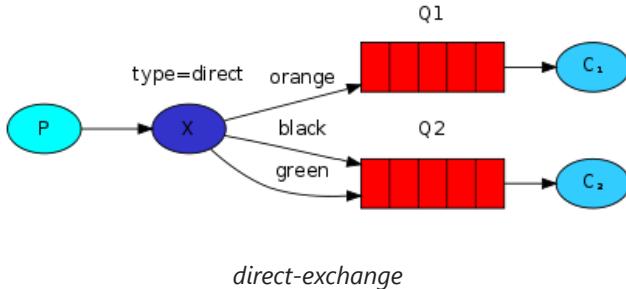
```

1  channel.QueueBind(queue: queueName,
2                     exchange: "direct_logs",
3                     routingKey: "black");
```

- binding key的含义取决于交换类型。之前所用到的fanout忽略了它的价值。

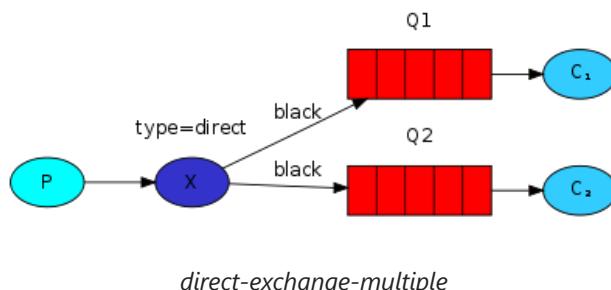
Direct exchange

- 上一节，我们将日志系统的所有消息广播给所有使用者。我们希望对其进行扩展，以允许根据消息的重要程度过滤信息。例如，我们可能希望写入磁盘的日志消息只接收严重错误，而不将磁盘空间浪费在警告和信息日志上。
- 我们使用的fanout不能给我们带来太多的灵活性—它只是进行无脑广播。
- 我们将使用direct exchange代替，direct exchange 背后的路由算法很简单—消息进入binding key 和消息routing key完全匹配的队列。
- 为了说明这一点，请看下图：



- 在这个设置中，我们可以看到 direct exchange 绑定了两个队列。第一个队列用orange—binding key绑定，第二个队列有两个binding key绑定， black 和 green。
- 在这样的设置中，发布到exchange的消息将被路由到Q1。带有black 和green的消息将被路由到Q2。所有其他信息将会被丢弃。

Multiple bindings 多重绑定



- 使用相同的binding key绑定多个队列完全是合法的。在我们的实例中，我们在X和Q1之间添加绑定， binding key 为 black。在这种情况下，这个direct exchange的行为类似于 fanout，将消息广播到匹配的队列。一个routing key带有black的消息将同时发送到Q1和Q2。

Emitting logs 发送日志

- 我们将在日志系统中使用这个模型。我们将通过direct exchange 发送信息，而不是fanout。我们将提供日志严重性作为routing key。这样，接收脚本将能够选择它想接收的严重性。让我们先关注发送日志。
- 一如既往，我们需要创建一个exchange:
channel.ExchangeDeclare(exchange: "direct_logs", type: "direct");
- 然后我们准备传递一个信息：

```
1 var body = Encoding.UTF8.GetBytes(message);
2 channel.BasicPublish(exchange: "direct_logs",
3                     routingKey: severity,
4                     basicProperties: null,
5                     body: body);
```

- 为了简化，我们假设“严重性”可以是“info”，“warning”， “error”中的一个。

Subscribing 订阅

- 接收消息的工作方式与上一节类似，只有一个例外—我们将为我们感兴趣的每个严重性创建一个绑定。

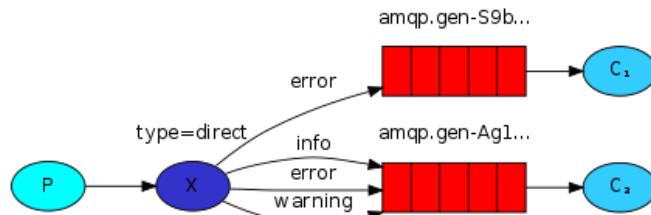
```
1 foreach(var severity in args)
2 {
3     channel.QueueBind(queue: queueName,
```

```

4         exchange: "direct_logs",
5         routingKey: severity);
6     }

```

Putting it all together 综合起来



example-four

```

1  using System;
2  using System.Linq;
3  using RabbitMQ.Client;
4  using System.Text;
5
6  class EmitLogDirect
7  {
8      public static void Main(string[] args)
9      {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using(var connection = factory.CreateConnection())
12             using(var channel = connection.CreateModel())
13             {
14                 channel.ExchangeDeclare(exchange: "direct_logs",
15                                         type: "direct");
16
17                 var severity = (args.Length > 0) ? args[0] : "info";
18                 var message = (args.Length > 1)
19                             ? string.Join(" ", args.Skip( 1 ).ToArray())
20                             : "Hello World!";
21                 var body = Encoding.UTF8.GetBytes(message);
22                 channel.BasicPublish(exchange: "direct_logs",
23                                       routingKey: severity,
24                                       basicProperties: null,
25                                       body: body);
26                 Console.WriteLine($" [x] Sent '{severity}'::{message}");
27             }
28
29             Console.WriteLine(" Press [enter] to exit.");
30             Console.ReadLine();
31     }
32 }

```

```

1  using System;
2  using RabbitMQ.Client;
3  using RabbitMQ.Client.Events;
4  using System.Text;

```

```

5
6 class ReceiveLogsDirect
7 {
8     public static void Main(string[] args)
9     {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using(var connection = factory.CreateConnection())
12             using(var channel = connection.CreateModel())
13             {
14                 channel.ExchangeDeclare(exchange: "direct_logs",
15                                         type: "direct");
16                 var queueName = channel.QueueDeclare().QueueName;
17
18                 if(args.Length < 1)
19                 {
20                     Console.Error.WriteLine("Usage: {0} [info] [warning] [error]",
21                                         Environment.GetCommandLineArgs()[0]);
22                     Console.WriteLine(" Press [enter] to exit.");
23                     Console.ReadLine();
24                     Environment.ExitCode = 1;
25                     return;
26                 }
27
28                 foreach(var severity in args)
29                 {
30                     channel.QueueBind(queue: queueName,
31                                     exchange: "direct_logs",
32                                     routingKey: severity);
33                 }
34
35                 Console.WriteLine(" [*] Waiting for messages.");
36
37                 var consumer = new EventingBasicConsumer(channel);
38                 consumer.Received += (model, ea) =>
39                 {
40                     var body = ea.Body.ToArray();
41                     var message = Encoding.UTF8.GetString(body);
42                     var routingKey = ea.RoutingKey;
43                     Console.WriteLine(" [x] Received '{0}':'{1}'",
44                                     routingKey, message);
45                 };
46                 channel.BasicConsume(queue: queueName,
47                                       autoAck: true,
48                                       consumer: consumer);
49
50                 Console.WriteLine(" Press [enter] to exit.");
51                 Console.ReadLine();
52             }
53         }
54     }

```

5.Topic

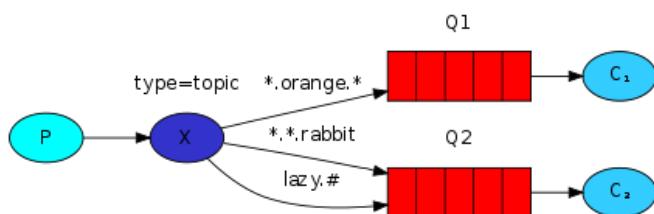
Receiving messages based on a pattern(topic)

- 上一节，我们改进了日志系统。我们没有使用fanout exchange，而是直接使用了direct，从而获得选择性接收日志的可能性。
- 虽然使用direct exchange改进了我们的系统，但它仍然有局限性—它不能基于多个标准进行路由。
- 在我们的日志记录系统中，我们可能不仅希望订阅基于严重性日志，还希望订阅基于发出日志的源的日志。你可能从syslog unix工具中了解到这个概念，该工具基于(info/warn/critical...)和功能(auth/cron/kern...)路由日志。
- 这将给我们很大的灵活性—我们可能想要只侦听'cron'的关键错误，但也要侦听来自'kern'的所有日志。
- 要在我们的日志系统中实现它，我们需要了解更复杂的主题交换。

Topic exchange

- 发送都topic exchange的信息不能有任何的routing key。它必须是一个由点分隔的单词列表。这些词可以是任何东西，但通常它们指定与信息相关的一些特征。一些有效的routing key示例：“stock.usd.nyse”, “nyse.vmw”, “quick.orange.rabbit”。routing key中可以有许多单词，最多255字节的限制。
- binding key 也必须是相同的形式。topic exchange 背后的逻辑与direct 类似。使用特定的routing key 发送信息到所有匹配 binding key 的队列。然后，binding key有两种特殊情况：
 1. *(星号)只能代替一个单词。
 2. #(井号)可以代替零个或多个单词。

用一个例子来简单解释：



example-five

- 在这个例子中，我们要发送的消息都是描述动物的。消息将与一个由三个字（两个点）组成的routing key一起发送。routing key 中的第一个单词将描述速度，第二个是颜色，第三个是物种：<speed>.<colour>.<species>
- 我们创建了三个绑定：Q1使用`*.orange.*`,Q2使用`*.*.rabbit`和`lazy.#`。
- 这些绑定可以总结为：
 1. Q1对所有橙色的动物感兴趣。
 2. Q2想知道关于兔子和懒惰动物的一切。
- routing key 设置为“quick.orange.rabbit”的信息将会发送到两个队列。“lazy.orange.elephant”的信息也会发送到两个队列。另外，“quick.roange.fox”的信息会去到第一个队列，“lazy.brown.fox”的信息会去到第二个队列。“lazy.pink.rabbit”的信息只会发送到第二个队列一次，尽管它匹配两个绑定。“quick.brown.fox”不匹配任务绑定，因此它将被丢弃。
- 如果我们毁约了，然后发送一条1个或者4个关键字的信息，例如“orange”或者“quick.orange.male.rabbit”,会怎么样？好吧，**这些信息不会匹配任何绑定，会丢失。**
- 另一方面，“lazy.orange.male.rabbit”即使有4个字，也会匹配最后一个绑定，然后会被送到第二个队列。

Topic exchange

- Topic exchange功能强大，可以像其他exchanges.当队列用“#”绑定时，它会接收到所有的信息，不管routing key s是什么，就是fanout exchange.
- 当“*”和“#”没有在绑定中使用，这个topic exchange就是direct exchange.

Put it all together 综合起来

-我们将日在组织系统中使用topic exchange。我们将从一个可行的假设开始，即日志的routing key有两个词，<facility>.<severity>

```

1  using System;
2  using System.Linq;
3  using RabbitMQ.Client;
4  using System.Text;
5
6  class EmitLogTopic
  
```

```

7  {
8      public static void Main(string[] args)
9      {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using(var connection = factory.CreateConnection())
12         using(var channel = connection.CreateModel())
13         {
14             channel.ExchangeDeclare(exchange: "topic_logs",
15                                     type: "topic");
16
17             var routingKey = (args.Length > 0) ? args[0] : "anonymous.info";
18             var message = (args.Length > 1)
19                         ? string.Join(" ", args.Skip( 1 ).ToArray())
20                         : "Hello World!";
21             var body = Encoding.UTF8.GetBytes(message);
22             channel.BasicPublish(exchange: "topic_logs",
23                                   routingKey: routingKey,
24                                   basicProperties: null,
25                                   body: body);
26             Console.WriteLine(" [x] Sent '{0}':'{1}'", routingKey, message);
27         }
28     }
29 }
```

```

1  using System;
2  using RabbitMQ.Client;
3  using RabbitMQ.Client.Events;
4  using System.Text;
5
6  class ReceiveLogsTopic
7  {
8      public static void Main(string[] args)
9      {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using(var connection = factory.CreateConnection())
12         using(var channel = connection.CreateModel())
13         {
14             channel.ExchangeDeclare(exchange: "topic_logs", type: "topic");
15             var queueName = channel.QueueDeclare().QueueName;
16
17             if(args.Length < 1)
18             {
19                 Console.Error.WriteLine("Usage: {0} [binding_key...]",
20                                       Environment.GetCommandLineArgs()[0]);
21                 Console.WriteLine(" Press [enter] to exit.");
22                 Console.ReadLine();
23                 Environment.ExitCode = 1;
24                 return;
25             }
26
27             foreach(var bindingKey in args)
28             {
29                 channel.QueueBind(queue: queueName,
```

```

30                     exchange: "topic_logs",
31                     routingKey: bindingKey);
32     }
33
34     Console.WriteLine(" [*] Waiting for messages. To exit press CTRL+C");
35
36     var consumer = new EventingBasicConsumer(channel);
37     consumer.Received += (model, ea) =>
38     {
39         var body = ea.Body.ToArray();
40         var message = Encoding.UTF8.GetString(body);
41         var routingKey = ea.RoutingKey;
42         Console.WriteLine(" [x] Received '{0}':'{1}'",
43                           routingKey,
44                           message);
45     };
46     channel.BasicConsume(queue: queueName,
47                           autoAck: true,
48                           consumer: consumer);
49
50     Console.WriteLine(" Press [enter] to exit.");
51     Console.ReadLine();
52 }
53 }
54 }
```

6.RPC

Request/reply pattern example

Remote procedure call

- 在第2节，我们学习了如何使用工作队列在多个工作者之间分配耗时的任务。
- 但是，如果我们需要在远程计算机上运行函数并等待结果，该怎么办？那就另当别论了。这种模式通常被称为远程过程调用（Remote Procedure）或RPC。
- 在本节中，我们将使用RabbitMQ构建一个RPC系统：一个客户端和一个可拓展的RPC服务器。由于我们没有任务值得分配的耗时任务，我们将创建一个返回斐波那契的虚拟RPC服务器。

Client interface 客户端接口

- 为了说明如何使用RPC服务器，我们将创建一个简单的客户机类。它将公开一个名为Call的方法，它发送一个RPC请求并阻塞，知道接收到答案。

```

1 var rpcClient = new RPCCClient();
2
3 Console.WriteLine(" [x] Requesting fib(30)");
4 var response = rpcClient.Call("30");
5 Console.WriteLine(" [.] Got '{0}'", response);
6
7 rpcClient.Close();
```

A note on RPC

- 虽然RPC是计算机中非常常见的模式，但它经常受到批评。当程序员不知道函数调用是本地调用还是缓慢的RPC时，就会出现问题。这样的混淆会导致不可预测的系统，并为调试增加不必要的复杂性。误用RPC不但不能简化软件，反而会导致不可维护的套管程序(spaghetti code)。
- 记住这一点，考虑以下建议：

- 确保知道哪些函数调用是本地的，哪些是远程的。
 - 管理好文件系统，保证组件之间没有依赖关系。
 - 处理错误情况，当RPC服务器长时间停机时，客户端应该如何反应。
- 如果有疑问，请避免使用RPC。如果可以的话，你应该使用异步管道，而不是类似于RPC阻塞，结果会异步的推到下一个计算阶段。

Callback queue

- 一般来说，在RabbitMQ上进行RPC很容易。客户端发送请求信息，服务器响应消息进行回复。为了接收响应，我们需要发送一个'callback'队列地址进行请求。

```

1 var props = channel.CreateBasicProperties();
2 props.ReplyTo = replyQueueName;
3
4 var messageBytes = Encoding.UTF8.GetBytes(message);
5 channel.BasicPublish(exchange: "",
6                     routingKey: "rpc_queue",
7                     basicProperties: props,
8                     body: messageBytes);
9
10 // ... then code to read a response message from the callback_queue ...

```

Message properties

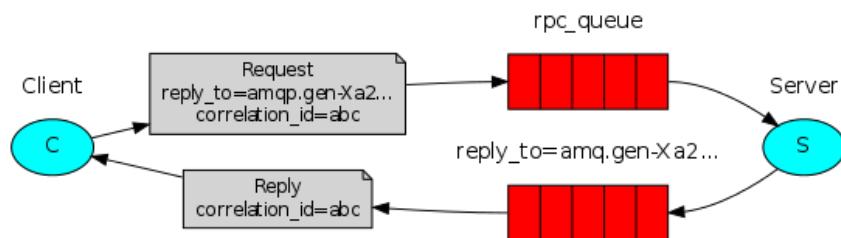
- AMQ 0-9-1 协议预先定义了消息的14个属性。大多数属性很少被使用，除了以下属性：

 - Persistent**: 将消息标记为Persistent(值为true)或transient(任何其他值)。
 - DeliveryMode**: 熟悉协议的人可能会使用这个属性而不是**Persistent**。它们控制的是同一件事。
 - ContentType**: 用于描述编码的mime-type。例如经常使用的JSON编码，最好将属性设置为：**application/json**。
 - ReplyTo**: 通常用于命名回调队列。
 - CorrelationId**: 用于将RPC响应与请求关联起来。

Correlation Id

- 在上述方法中，我们建议为每个RPC请求创建一个回调队列。这是相当低效的，但幸运的是有一个更好的方法—让我们为每个客户端创建一个回调队列。
- 这就产生了一个新问题，在队列中接收到响应后，不清楚该响应属于哪个请求。这就是要使用**CorrelationId**属性的时候。我们将为每个请求设置一个唯一值。稍后，当我们在回调队列中收到消息时，我们将查看这个属性。基于这个属性，我们将能够将响应和请求相匹配。如果我们看到一个未知的"CorrelationId"值，我们可以安全地丢弃消息—它不属于我们的请求。
- 你可能会问，为什么我们要忽略回调队列中未知消息，而不是错误识别？这是由于服务器可能存在竞态条件。尽管不太可能，但RPC服务器有可能在向我们发送应答之后就死掉。但在这之前没有为请求发送了确认信息。如果发生这种情况，重新启动的RPC服务器将再次处理请求。这就是为什么在客户端我们必须优雅地处理重复的响应，并且RPC在理想情况下应该是幂等的。

Summary



example-six

RPC工作流程如下：

- 当客户机启动时，它创建一个匿名独占回调队列。
- 对于RPC请求，客户机发送两个属性的消息：**Reply To**和**CorrelationId**，前者被设置为回调队列，而后者被设置为每个请求的唯一值。

3. 请求被发送到`rpc_queue`。
4. RPC worker(又名：服务器)正在等待队列上的请求。当一个请求出现时，它执行任务，并使用Reply To属性中的队列将一个带有结果的消息发送回Client。
5. 客户端等待回调队列中的数据。当消息出现时，它检查CorrelationId属性。如果它匹配来自请求的值，它将向应用程序返回响应。

Putting it all together

- 斐波那契任务：

```

1  private static int fib(int n)
2  {
3      if (n == 0 || n == 1) return n;
4      return fib(n - 1) + fib(n - 2);
5  }

```

- 我们声明fibonacci函数。它假设只有有效的正整数输入。(不要期望这个方法适用于较大的数字，它可能是最慢的递归实现)。

```

1  using System;
2  using RabbitMQ.Client;
3  using RabbitMQ.Client.Events;
4  using System.Text;
5
6  class RPCServer
7  {
8      public static void Main()
9      {
10         var factory = new ConnectionFactory() { HostName = "localhost" };
11         using (var connection = factory.CreateConnection())
12         using (var channel = connection.CreateModel())
13         {
14             channel.QueueDeclare(queue: "rpc_queue", durable: false,
15                 exclusive: false, autoDelete: false, arguments: null);
16             channel.BasicQos(0, 1, false);
17             var consumer = new EventingBasicConsumer(channel);
18             channel.BasicConsume(queue: "rpc_queue",
19                 autoAck: false, consumer: consumer);
20             Console.WriteLine("[x] Awaiting RPC requests");
21
22             consumer.Received += (model, ea) =>
23             {
24                 string response = null;
25
26                 var body = ea.Body.ToArray();
27                 var props = ea.BasicProperties;
28                 var replyProps = channel.CreateBasicProperties();
29                 replyProps.CorrelationId = props.CorrelationId;
30
31                 try
32                 {
33                     var message = Encoding.UTF8.GetString(body);
34                     int n = int.Parse(message);
35                     Console.WriteLine("[.] fib({0})", message);
36                     response = fib(n).ToString();
37                 }

```

```

38         catch (Exception e)
39     {
40         Console.WriteLine(" [.] " + e.Message);
41         response = "";
42     }
43     finally
44     {
45         var responseBytes = Encoding.UTF8.GetBytes(response);
46         channel.BasicPublish(exchange: "", routingKey: props.ReplyTo,
47             basicProperties: replyProps, body: responseBytes);
48         channel.BasicAck(deliveryTag: ea.DeliveryTag,
49             multiple: false);
50     }
51 };
52
53     Console.WriteLine(" Press [enter] to exit.");
54     Console.ReadLine();
55 }
56 }
57 /**
58 */
59
60 /**
61 * Assumes only valid positive integer input.
62 * Don't expect this one to work for big numbers, and it's
63 * probably the slowest recursive implementation possible.
64 */
65
66 private static int fib(int n)
67 {
68     if (n == 0 || n == 1)
69     {
70         return n;
71     }
72
73     return fib(n - 1) + fib(n - 2);
74 }

```

上面服务器的代码非常简单：

- 像往常一样，我们直接建立连接、通道和声明队列。
- 我们可能需要运行多个服务器进程。为了在多个服务器上平均分配负载，我们需要在设置[prefetchCount](#)和[channel.BasicQos](#)。
- 我们使用[BasicConsume](#)来访问队列。然后注册一个传递处理程序，在该处理程序中完成工作并将响应发送出去。

```

1  using System;
2  using System.Collections.Concurrent;
3  using System.Text;
4  using RabbitMQ.Client;
5  using RabbitMQ.Client.Events;
6
7  public class RpcClient
8  {
9      private readonly IConnection connection;

```

```
10     private readonly IModel channel;
11     private readonly string replyQueueName;
12     private readonly EventingBasicConsumer consumer;
13     private readonly BlockingCollection<string> respQueue = new BlockingCollection<string>
14     ();
15     private readonly IBasicProperties props;
16
17     public RpcClient()
18     {
19         var factory = new ConnectionFactory() { HostName = "localhost" };
20
21         connection = factory.CreateConnection();
22         channel = connection.CreateModel();
23         replyQueueName = channel.QueueDeclare().QueueName;
24         consumer = new EventingBasicConsumer(channel);
25
26         props = channel.CreateBasicProperties();
27         var correlationId = Guid.NewGuid().ToString();
28         props.CorrelationId = correlationId;
29         props.ReplyTo = replyQueueName;
30
31         consumer.Received += (model, ea) =>
32         {
33             var body = ea.Body.ToArray();
34             var response = Encoding.UTF8.GetString(body);
35             if (ea.BasicProperties.CorrelationId == correlationId)
36             {
37                 respQueue.Add(response);
38             }
39         };
40
41         channel.BasicConsume(
42             consumer: consumer,
43             queue: replyQueueName,
44             autoAck: true);
45     }
46
47     public string Call(string message)
48     {
49         var messageBytes = Encoding.UTF8.GetBytes(message);
50         channel.BasicPublish(
51             exchange: "",
52             routingKey: "rpc_queue",
53             basicProperties: props,
54             body: messageBytes);
55
56         return respQueue.Take();
57     }
58
59     public void Close()
60     {
61         connection.Close();
62     }
63 }
```

```

64 public class Rpc
65 {
66     public static void Main()
67     {
68         var rpcClient = new RpcClient();
69
70         Console.WriteLine(" [x] Requesting fib(30)");
71         var response = rpcClient.Call("30");
72
73         Console.WriteLine(" [.] Got '{0}'", response);
74         rpcClient.Close();
75     }
76 }

```

客户端代码稍微复杂一些：

- 我们建立一个连接和通道，并为回复声明一个独占的“回调”队列。
- 我们订阅‘callback’队列，这样我们就可以接收RPC响应。
- 我们的Call方法发出实际的RPC请求。
- 在这里，我们首先生成一个唯一的“CorrelationId”编号，并保存它用于识别合适的响应。
- 接下来，我们发布带有两个属性的请求信息：Reply To和CorrelationId。
- 在这一点上，我们可以坐下来，等待适当的回应到来。
- 对于每个响应消息，客户端检查CorrelationId是否是我们正在寻找的那个。如果是，则保存响应。
- 最后，我们将响应返回给用户。

Making the Client request: 发送客户请求

```

1 var rpcClient = new RPCCClient();
2
3 Console.WriteLine(" [x] Requesting fib(30)");
4 var response = rpcClient.Call("30");
5 Console.WriteLine(" [.] Got '{0}'", response);
6
7 rpcClient.Close();

```

- 这里提出的设计并不是RPC服务的唯一可能实现，但是它有一些重要的优点：
1. 如果RPC服务器太慢，你可以通过运行另一个服务器来进行扩展。尝试在新的控制台中运行第二个RPCServer。
 2. 在客户端，RPC只需要发送和接收一个消息。不需要像QueueDeclare这样的同步调用。因此，RPC客户端对于单个RPC请求只需要一次网络往返。

我们的代码仍然非常简单，没有尝试解决更复杂(但重要)的问题，如：

1. 如果没有服务器运行，客户端应该如何反应？
2. 客户端是否应该为RPC设置某种超时？
3. 如果服务器发生故障并引发异常，是否应该将其转发到客户端？
4. 在处理之前防止无效的传入消息？(检查bouds, type)

这里每次只展示一个新概念，并且可能故意简化一些东西而忽略其他东西。例如，为了简洁起见，连接管理、错误处理、连接恢复、并发性和度量收集等主题在很大程度上被省略了。这种简化的代码不应该被认为可以用于生产。

7. Publisher Confirms

Reliable publishing with publisher confirms

- Publisher Confirms是一个RabbitMQ扩展来实现可靠的发布。当在通道上启用发布者确认时，客户端发布的消息将由代理异步确认，这意味着它们在服务器端得到了处理。
- 在本节，我们将使用publisher confirm来确保发布的消息已安全到达代理。我们将介绍几种使用publisher confirmsd的策略，并解释它们的优缺点。

Enabling Publisher Confirms on a Channel

- Publisher confirm确认是RabbitMQ对AMQP 0.9.1协议的扩展。所以在默认情况下不启动。Publisher confirm 通过[ConfirmSelect](#)方法在通道lever中使用：

```
1 var channel = connection.CreateModel();
2 channel.ConfirmSelect();
```

- 必须在预期使用publisher confirms的每个通道上调用此方法。确认应该只启用一次，而不是对每条发布的信息都启动。

Strategy #1: Publishing Messages Individually

- 让我们从最简单的使用确认发布的方法开始，即发布消息并同步等待其确认：

```
1 while (ThereAreMessagesToPublish())
2 {
3     byte[] body = ...;
4     IBasicProperties properties = ...;
5     channel.BasicPublish(exchange, queue, properties, body);
6     // uses a 5 second timeout
7     channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
8 }
```

- 在前面的例子中，我们像往常一样发布消息，并使用[Channel#WaitForConfirmsOrDie\(TimeSpan\)](#)方法等待其确认。该方法在确认消息后立即返回。如果消息在超时时间内没有得到确认，或者消息被nack-ed（意味着代理由于某种原因无法处理它），则该方法将抛出异常。异常的处理通常包括记录错误消息和/或重新尝试发送该消息。
- 不同的客户端有不同的同步处理publisher confirm的方法，因此一定要仔细阅读正在使用的客户端的文档。
- 这种技术非常简单，但也有一个主要缺点：它大大减慢了发布速度，因为消息的确认会阻塞所有后续消息的发布。这种方法不会提供超过每秒几百条发布消息的吞吐量。尽管如此，这对于某些应用程序来说已经足够好了。

Are Publisher Confirms Asynchronous?

- 我们在开始提到，代理异步确认发布的消息，但在第1个示例中，代码同步等待，直到消息被确认。客户端实际上异步地接收确认，并相应地解除对[WaitForConfirmsOrDie](#)的调用。可以把[WaitForConfirmsOrDie](#)看作一个同步助手，它在底层依赖异步通知。

Strategy #2: Publishing Messages in Batches

- 为了改进前面的示例，我们可以发布一批消息并等待整个消息得到确认。下面的例子使用了100个批次：

```
1 var batchSize = 100;
2 var outstandingMessageCount = 0;
3 while (ThereAreMessagesToPublish())
4 {
5     byte[] body = ...;
6     IBasicProperties properties = ...;
7     channel.BasicPublish(exchange, queue, properties, body);
8     outstandingMessageCount++;
9     if (outstandingMessageCount == batchSize)
10    {
11        channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
12        outstandingMessageCount = 0;
13    }
14 }
15 if (outstandingMessageCount > 0)
16 {
17     channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
18 }
```

- 等待一批消息被确认比单个消息被确认大大提高了吞吐量(在远程RabbitMQ节点上高达20-30倍)。一个缺点是，在发生故障时，我们无法确切地直到哪里出了问题，因为我们可能不得不在内存中保留整个批处理，以便记录有意义的内容或重新发布消息。而且这个解决方案仍然是同步的，因此它会阻止消息的发布。

Strategy #3: Handling Publisher Confirms Asynchronously

- 代理异步确认发布的消息，只需要在客户端注册一个回调就可以得到这些确认的通知：

```

1 var channel = connection.CreateModel();
2 channel.ConfirmSelect();
3 channel.BasicAcks += (sender, ea) =>
4 {
5     // code when message is confirmed
6 };
7 channel.BasicNacks += (sender, ea) =>
8 {
9     //code when message is nack-ed
10};

```

- 有两个回调：一个用于确认消息，另一个用于nack-ed消息（可以被代理认为丢失的消息）。两个回调函数都有对应的EventArgs参数(ea)，其中包含：

1. delivery tag：传递标签，标识已确认或已识别的消息的序列号。我们将很快看到如何将它与发布的消息关联起来。
2. multiple：布尔值。如果为false，则只有一条消息被确认/nack-ed，如果为true，则所有序列号较低或相等的消息都被确认/nack-ed。

- 序列号可以在发布前获得：[Channel#NextPublishSeqNo](#)

```

1 var sequenceNumber = channel.NextPublishSeqNo;
2 channel.BasicPublish(exchange, queue, properties, body);

```

- 将消息与序列号关联的一种简单方法是使用字典。假设我们希望发布字符串，因为它们很容易转换为用于发布的字节数组。下面是一个使用字典将发布序列号与消息的字符串体关联起来的代码示例：

```

1 var outstandingConfirms = new ConcurrentDictionary<ulong, string>();
2 // ... code for confirm callbacks will come later
3 var body = "...";
4 outstandingConfirms.TryAdd(channel.NextPublishSeqNo, body);
5 channel.BasicPublish(exchange, queue, properties, Encoding.UTF8.GetBytes(body));

```

- 发布代码现在使用字典跟踪出站消息。当确认到达时，我们需要清理这个字典，并在消息被nack-ed时做一些事情，比如记录一个警告：

```

1 var outstandingConfirms = new ConcurrentDictionary<ulong, string>();
2
3 void cleanOutstandingConfirms(ulong sequenceNumber, bool multiple)
4 {
5     if (multiple)
6     {
7         var confirmed = outstandingConfirms.Where(k => k.Key <= sequenceNumber);
8         foreach (var entry in confirmed)
9         {
10             outstandingConfirms.TryRemove(entry.Key, out _);
11         }
12     }
13     else
14     {
15         outstandingConfirms.TryRemove(sequenceNumber, out _);
16     }
17 }

```

```

16     }
17 }
18
19 channel.BasicAcks += (sender, ea) => cleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);
20 channel.BasicNacks += (sender, ea) =>
21 {
22     outstandingConfirms.TryGetValue(ea.DeliveryTag, out string body);
23     Console.WriteLine($"Message with body {body} has been nack-ed. Sequence number: {ea.Del
24     iveryTag}, multiple: {ea.Multiple}");
25     cleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);
26 };
27 // ... publishing code

```

- 前面的示例包含一个回调函数，它在确认到达时清楚字典。注意，这个回调即处理单个确认，也处理多个确认。这个回调函数在确认到达时使用 ([Channel#BasicAcks](#))。nack-ed消息的回调将检索消息体并发出警告。然后，它重用前面的回调来清楚字典中未完成的确认（无论是确认的还是nack-ed的，都必须删除字典中相应的条目）。

How to Track Outstanding Confirms?

- 我们的示例使用[ConcurrentDictionary](#)来跟踪未完成的确认。这种数据结构很方便，原因有几个：它允许轻松地将序列号与消息（无论消息数据是什么）关联起来，并轻松地将条目清楚到给定地序列id(以处理多个确认/nacks)。最后，它支持并发访问，因为确认回调是在客户端拥有地线程中调用的，该线程应该与发布线程保持不同。
- 与复杂的字典实现相比，还有其他方法可以跟踪未完成的确认，比如使用简单的并发哈希表和一个变量来跟踪发布序列的下界，但这些方法通常更复杂，不属于教程。

综上所述，异步处理publisher confirms通常需要以下步骤：

- 提供一种将发布序列号与消息关联起来的方法。
- 在通道上注册confirm listeners以便在发布者ack/nacks达到时通知其执行适当的操作，如记录或重新发布nack-ed消息。在此步骤中，序列号到消息的关联机制可能还需要进一步处理。
- 在发布消息之前跟踪发布序列号。

Re-publishing nack-ed Messages?

-

从相应的回调重新发布nack-ed消息可能很诱人，但应该避免这样做，因为确认回调实在I/O线程中分派的，而通道不知道执行操作。更好的解决方案是将消息放入由发布线程轮询的内存队列中。像[ConcurrentQueue](#)这样的类，是在确认回调和发布线程之间传输消息的很好的候选类。

Summary

- 在某些应用程序中，确保发布的消息能够发送到代理是非常重要的。发布者确认RabbitMQ特性有助于满足这一需求。发布者确认本质上是异步的，但也可以同步处理它们。没有确认的方法来实现publisher confirm，这通常取决于应用程序和整个系统中的约束。典型的技术：
 - 单独发布消息，同步等待确认：很简单，但吞吐量非常有限。
 - 批处理发布消息，等待批处理的同步确认：简单、合理的吞吐量，但很难在出现错误时进行推断。
 - 异步处理：最佳的性能和资源使用，在错误的情况下良好的控制，但可以涉及到正确的实现。

Putting It All Together

```

1  using RabbitMQ.Client;
2  using System;
3  using System.Collections.Concurrent;
4  using System.Diagnostics;
5  using System.Text;
6  using System.Linq;
7  using System.Threading;
8

```

```
9  class PublisherConfirms
10 {
11     private const int MESSAGE_COUNT = 50_000;
12
13     public static void Main()
14     {
15         PublishMessagesIndividually();
16         PublishMessagesInBatch();
17         HandlePublishConfirmsAsynchronously();
18     }
19
20     private static IConnection CreateConnection()
21     {
22         var factory = new ConnectionFactory { HostName = "localhost" };
23         return factory.CreateConnection();
24     }
25
26     private static void PublishMessagesIndividually()
27     {
28         using (var connection = CreateConnection())
29         using (var channel = connection.CreateModel())
30         {
31             // declare a server-named queue
32             var queueName = channel.QueueDeclare(queue: "").QueueName;
33             channel.ConfirmSelect();
34
35             var timer = new Stopwatch();
36             timer.Start();
37             for (int i = 0; i < MESSAGE_COUNT; i++)
38             {
39                 var body = Encoding.UTF8.GetBytes(i.ToString());
40                 channel.BasicPublish(exchange: "", routingKey: queueName, basicProperties: null, body: body);
41                 channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
42             }
43             timer.Stop();
44             Console.WriteLine($"Published {MESSAGE_COUNT:N0} messages individually in {timer.ElapsedMilliseconds:N0} ms");
45         }
46     }
47
48     private static void PublishMessagesInBatch()
49     {
50         using (var connection = CreateConnection())
51         using (var channel = connection.CreateModel())
52         {
53             // declare a server-named queue
54             var queueName = channel.QueueDeclare(queue: "").QueueName;
55             channel.ConfirmSelect();
56
57             var batchSize = 100;
58             var outstandingMessageCount = 0;
59             var timer = new Stopwatch();
60             timer.Start();
61             for (int i = 0; i < MESSAGE_COUNT; i++)
```

```

62    {
63        var body = Encoding.UTF8.GetBytes(i.ToString());
64        channel.BasicPublish(exchange: "", routingKey: queueName, basicProperties: null, body: body);
65        outstandingMessageCount++;
66
67        if (outstandingMessageCount == batchSize)
68        {
69            channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
70            outstandingMessageCount = 0;
71        }
72    }
73
74    if (outstandingMessageCount > 0)
75        channel.WaitForConfirmsOrDie(new TimeSpan(0, 0, 5));
76
77    timer.Stop();
78    Console.WriteLine($"Published {MESSAGE_COUNT:N0} messages in batch in {time
r.ElapsedMilliseconds:N0} ms");
79}
80
81
82    private static void HandlePublishConfirmsAsynchronously()
83    {
84        using (var connection = CreateConnection())
85        using (var channel = connection.CreateModel())
86        {
87            // declare a server-named queue
88            var queueName = channel.QueueDeclare(queue: "").QueueName;
89            channel.ConfirmSelect();
90
91            var outstandingConfirms = new ConcurrentDictionary();
92
93            void cleanOutstandingConfirms(ulong sequenceNumber, bool multiple)
94            {
95                if (multiple)
96                {
97                    var confirmed = outstandingConfirms.Where(k => k.Key <= sequenceNum
ber);
98                    foreach (var entry in confirmed)
99                        outstandingConfirms.TryRemove(entry.Key, out _);
100                }
101                else
102                    outstandingConfirms.TryRemove(sequenceNumber, out _);
103            }
104
105            channel.BasicAcks += (sender, ea) => cleanOutstandingConfirms(ea.DeliveryTa
g, ea.Multiple);
106            channel.BasicNacks += (sender, ea) =>
107            {
108                outstandingConfirms.TryGetValue(ea.DeliveryTag, out string body);
109                Console.WriteLine($"Message with body {body} has been nack-ed. Sequence
number: {ea.DeliveryTag}, multiple: {ea.Multiple}");
110                cleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);
111            };

```

```

112
113     var timer = new Stopwatch();
114     timer.Start();
115     for (int i = 0; i < MESSAGE_COUNT; i++)
116     {
117         var body = i.ToString();
118         outstandingConfirms.TryAdd(channel.NextPublishSeqNo, i.ToString());
119         channel.BasicPublish(exchange: "", routingKey: queueName, basicProperties: null, body: Encoding.UTF8.GetBytes(body));
120     }
121
122     if (!WaitUntil(60, () => outstandingConfirms.IsEmpty))
123         throw new Exception("All messages could not be confirmed in 60 seconds");
124
125     timer.Stop();
126     Console.WriteLine($"Published {MESSAGE_COUNT} messages and handled confirms asynchronously {timer.ElapsedMilliseconds:N0} ms");
127 }
128 }
129
130 private static bool WaitUntil(int numberSeconds, Func<bool> condition)
131 {
132     int waited = 0;
133     while(!condition() && waited < numberSeconds * 1000)
134     {
135         Thread.Sleep(100);
136         waited += 100;
137     }
138
139     return condition();
140 }
141 }
```

•

三种效果比较:

```

1 Published 50,000 messages individually in 5,549 ms
2 Published 50,000 messages in batch in 2,331 ms
3 Published 50,000 messages and handled confirms asynchronously in 4,054 ms
```

- 如果客户机和服务器位于同一台机器上，那么这三种效果的输出看起来类似。单独发布消息的性能不像预期的那样好，但是与批发布相比，异步处理的结果有点令人失望。
- Publisher confirm非常依赖于网络，所以我们最好尝试使用远程节点，这更现实，因为客户端和服务器通常不在生产中的同一台机器上。上面的代码可以很容易地更改为使用非本地节点：

```

1 Published 50,000 messages individually in 5,549 ms
2 Published 50,000 messages in batch in 2,331 ms
3 Published 50,000 messages and handled confirms asynchronously in 4,054 ms
```

三种效果比较(非本地):

```

1 Published 50,000 messages individually in 231,541 ms
2 Published 50,000 messages in batch in 7,232 ms
```

- 我们看到现在独立发布的表现很糟糕。但是，由于客户端和服务器之间由网络，批处理和异步处理现在执行起来是类似的。对于发布器确认的异步处理有一点优势。
- 记住，批发布实现起来很简单，但是在发布者否定确认的情况下，不容易知道哪些消息无法到达代理。异步处理发布者确认需要更多的实现。但可以提供更好的粒度，更好地控制发布地消息被ack时执行地操作。