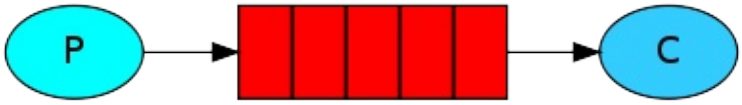
目錄

RabbitMQ 教程

本教程涵盖了利用RabbitMQ创建消息传递应用程序的基础知识。你需要安装RabbitMQ服务器来完成本教程，请阅读安装向导。本教程的实例代码是开源的，和RabbitMQ网站一样。

目录

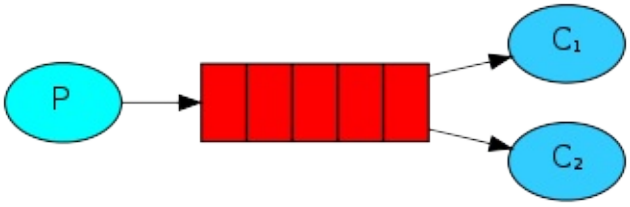
1.Hello World!

做最简单的事情 

- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C
- Swift

2.Work queues

任务分发(竞争消费者模式)

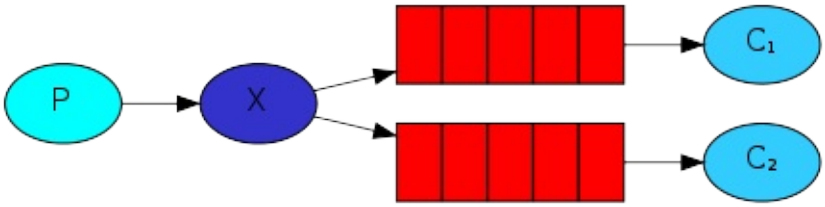


- Python
- Java
- Ruby
- PHP
- C#
- Javascript

- Go
- Elixir
- Objective-C
- Swift

3.Publish/Subscribe

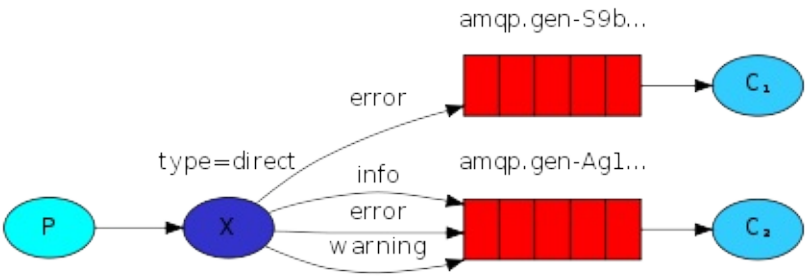
向多个消费者发送消息



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C
- Swift

4.Routing

选择性的接收消息

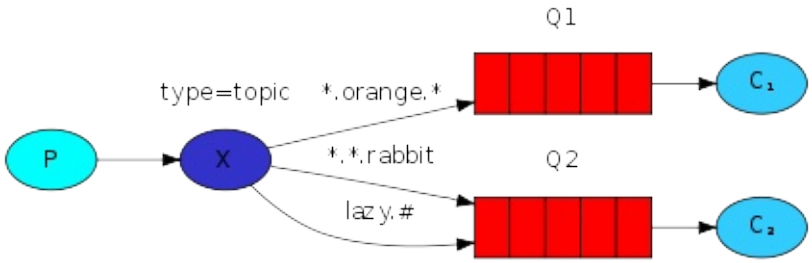


- Python
- Java
- Ruby
- PHP
- C#

- Javascript
- Go
- Elixir
- Objective-C
- Swift

5.Topics

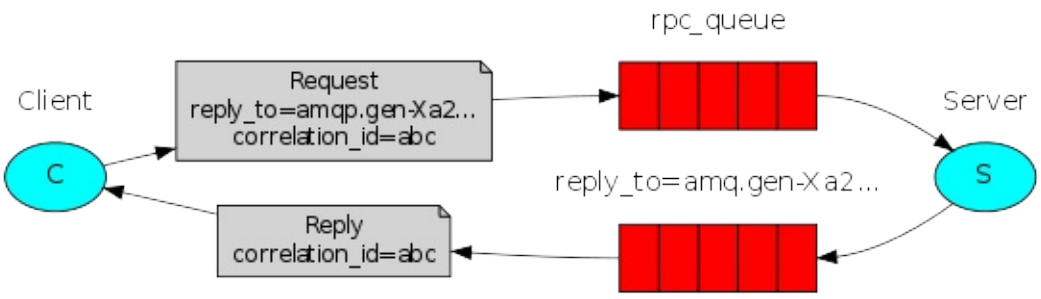
基于模式(主题)来接收消息



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C
- Swift

6.RPC

请求/回复(RPC) 模式实例



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C
- Swift

获取帮助

如果你对RabbitMQ有任何问题/意见，你可以随时在邮件列表中询问。

AMQP 0-9-1概述和快速参考

一旦你已经完成了教程（或如果你想跳过），你可能希望阅读RabbitMQ概念的介绍，并浏览我们的AMQP 0-9-1快速参考指南。

Hello World

先决条件

本教程假定RabbitMQ在localhost默认端口（5672）上安装和运行。如果您使用其他主机，端口或凭据，连接设置将需要调整。

在哪里获得帮助

如果您在阅读本教程时遇到问题，可以通过邮件列表与我们联系。

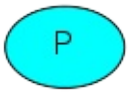
简介

RabbitMQ是一个消息代理。主要想法很简单：它接受和转发消息。你可以把它当成一个邮局：当你发送邮件到邮政信箱，你很确定邮递员最终将邮件发送给你的收件人。使用这个隐喻RabbitMQ是一个邮政信箱，邮局和邮递员。

RabbitMQ和邮局之间的主要区别是它不处理纸张，而是接受，存储和转发二进制数据块的消息。

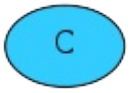
在使用RabbitMQ前，先让我们了解一些基本术语：

- Producing：发送消息，发送消息的程序是生产者(producer)，我们用P来表示它：



- Queue: queue 是邮箱的名称，它们存在于RabbitMQ中，虽然消息流经RabbitMQ和您的应用程序，但它们只能存储在队列中。队列不受任何限制，它可以存储尽可能多的消息，你喜欢 - 它本质上是一个无限缓冲区。许多生产者可以发送去往一个队列的消息，许多消费者可以尝试从一个队列接收数据。

- Consuming: 消费与接收具有相似的含义。消费者(consumer)是大多数等待接收消息的程序。



> 生产者（producer），消费者(consumer)和代理(broker)不必驻留在同一台机器上;事实上在大多数应用中他们没有。

Hello World!

使用**php-amqplib**

在这一部分，我们将在php中编写两个应用程序：发送单个消息的生产者、以及接收消息并将其打印出来的消费者，我们将讨论php-amqplib API中的一些细节，集中讨论这个非常简单的事情。这是一个“Hello World”的消息。

在下图中，“P”是我们的生产者，“C”是我们的消费者。中间的框是一个队列--RabbitMQ代表消费者保存的消息缓冲区。

php-amqplib 客户端库

RabbitMQ支持多种协议，本教程包含AMQP 0-9-1，它是一个用于消息传递的开放式协议。RabbitMQ有许多不同语言的客户端。我们将在本教程中使用php-amqplib，并使用Composer进行依赖关系管理。

在你的项目根目录中添加 *composer.json*文件，并加入下列内容：

```
{
    "require": {
        "php-amqplib/php-amqplib": "2.5.*"
    }
}
```

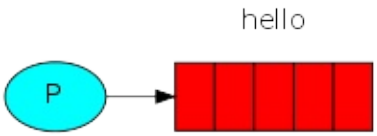
接下来在终端中执行下列命令来安装php-amqplib:

```
$ composer install
```

如果你没有安装composer，请自行安装。

现在我们已经安装了php-amqplib库，我们可以编写一些代码。

Sending



我们将发送消息的程序命名为 *send.php*，和接收消息的程序命名为 *receive.php*，发送方链接到RabbitMQ，发送单条消息，然后退出。

在send.php中，我们需要包含库并使用必要的类

```
require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;
```

接下来，我们创建一个到服务器的链接：

```
$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();
```

$connection 连接socket链接，并为我们处理协议版本和认证等等。在这个案例中，我们连接到本地的broker，所以这里是localhost。如果我们想连接到其它机器上的broker，只需要提供它的名字或者IP地址。

然后我们创建一个通道(channel)，这里有大多数的API来完成我们的任务。

要发送消息，我们必须先声明一个队列(queue)，然后才可以向队列发送消息：

```
$channel->queue_declare('hello',false,false,false,false);

$msg = new AMQPMessage('Hello World!');
$channel->basic_publish($msg,'','hello');

echo " [x] Sent 'Hello World!' \n";
```

声明队列是幂等的，它只有在它不存在时才会被创建。

消息内容是一个字节数组，所以你可以把任何想发送的内容编码后发送过来。

最后我们要关闭$channel 和$connection来释放资源：

```
$channel->close();
$connection->close();
```

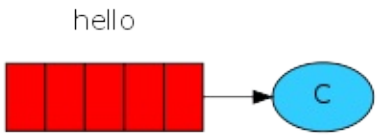
上面就是整个 *send.php*的代码了。

发送程序无法运行？

> 如果你第一次使用RabbitMQ，并且没有看到程序最后输出的Sent来告知你成功了，你可能会挠着头来思考到底是哪出错了？可能是broker没有足够的可用空间(默认情况下，它至少需要1GB的空间来运行)，因此拒绝接受消息。你可以检查下broker的日志文件来确定问题的原因，并在必要时减少该限制，配置文档里面有教你如何配置disk_free_limit.

Receiving

以上就是我们的sender了，我们的接收器是推送来自RabbitMQ的消息，因此与发送单个消息的sender是不一样的。我们继续运行监听消息并将其打印出来。



receive.php 中同样需要include类库和use必要的类：

```
require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
```

设置与sender 相同，我们打开一个连接(connection)和一个通道(channel)，并且声明一个我们将要使用的队列，这里要注意到队列必须和sender发送的队列是同一个。

```
$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

$channel->queue_declare('hello',false,false,false,false);

echo ' [*] Waiting for messages. To exit press CTRL+C',"\n";
```

> 注意：我们在这里声明队列，是因为我们可能在发送方之前启动接收方，所以我们希望在尝试使用消息前确保队列存在。并且因为这里声明是幂等的，所以在发送方声明时，则可能不会创建新的队列，而是打开现有的队列

我们告诉服务器使用队列来传递消息。我们定义一个php回调(PHP callable)，它来接收服务器发送的消息。

```
$callback = function($msg) {
    echo " [x] Received ",$msg->body,"\n";
}

$channel->basic_consume('hello','',false,true,false,false,$callback);

while(count($channel->callbacks)) {
    $channel->wait();
}
```

上述代码会阻塞住程序结束，而 $channel有一个回调函数，每当我们收到一个程序时，我们的$callback函数将被传递接收道德消息。

以上就是**receive.php**的全部代码了

将它们放在一起

现在我们可以运行这两个脚本了，在终端里，运行发送程序：

```
$ php send.php
```

然后运行接收程序

```
$ php receive.php
```

接收方将通过RabbitMQ接收到的消息打印出来，接收者将继续运行，等待消息（使用Ctrl-C停止它），所以尝试从另一个终端运行发送者。

如果你想检查队列，你可以运行在RabbitMQ目录下的sbin目录中的rabbitmqctl:

```
$ rabbitmqctl list_queues.
```

你可以将rabbitmqctl加入到你的环境变量中，那样你就可以直接运行rabbitmqctl,而不用进入RabbitMQ目录中。

> 输出：Hello World!

先决条件

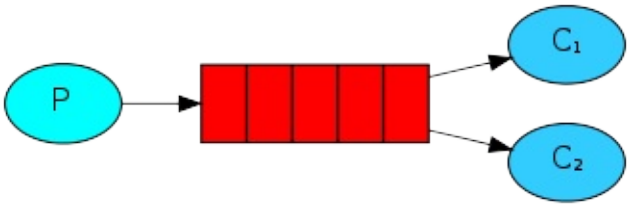
本教程假定RabbitMQ在localhost默认端口（5672）上安装和运行。如果您使用其他主机，端口或凭据，连接设置将需要调整。

在哪里获得帮助

如果您在阅读本教程时遇到问题，可以通过邮件列表与我们联系。

Work Queues

使用**php-amqplib**



在第一篇教程中，我们编写程序以从命名队列中发送和接收消息。在这一节中，我们将创建一个工作队列(work queues)，用于在多个工作者之间分配耗时的任务。

工作队列(也叫任务队列)背后主要的想法是避免立刻执行资源密集型任务，并不得不等待它完成。想法，我们安排任务以后再执行。我们将任务封装为消息，并将其发送到队列。在后台运行的worker进程将弹出任务并最终执行作业。当你运行了多个workershi时，任务将在它们之间共享。

此概念在Web应用程序中尤其有用，因为在短期HTTP请求窗口中无法处理复杂任务。

准备

在上一节教程中，我们发送了一个内容为"Hello World!"的消息，现在我们要发送代表复杂任务的字符串。我们没有一个现实的任务，例如调整图像大小或者呈现PDF文档，所以让我们使用sleep()函数来假装这个任务是一个耗时较长的复杂任务。我们将字符串中的"."作为其复杂性，每个"."耗时1秒。例如"Hello..." 这个任务需要三秒。

我们稍微修改上个例子中的send.php的代码，以允许从命令行发送任意消息。这个程序将任务调度到我们的工作队列，让我们给它命名为 *new_task.php*

```
$data = implode(' ',array_slice($grav,1));

if(empty($data)) $data = "Hello World!";
$msg = new AMQPMessage($data,array('delivery_mode' => 2); #make message persistent

$channel->basic_publish($msg,'','task_queue');
echo " [x] Sent",$data,"\n";
```

receive.php脚本同样需要修改：需要为消息中的每个点伪造一秒的工作，它会从消息队列中弹出消息并执行任务，我们给它命名为：

```
$callback = function($msg){
    echo " [x] Received ",$msg->body,"\n";
    sleep(substr_count($msg->body,'.'));
    echo " [x] Done","\n";
}

$channel->basic_qos(null,1,null);
$channel->basic_consume('task_queue','',false,true,false,false,$callback);
```

接下来让我们运行这两个脚本：

```
$ php new_task.php "A very hard task witch takes two seconds.."
$ php worker.php
```

循环调度

使用任务队列的优点之一是能够更轻松的并行工作，如果我们积压了工作，我们可以简单的只增加更多的worker，这样可以方便架构的扩展。

首先让我们同时运行两个worker.php脚本，它们都会从队列中获取消息，让我们看看执行后的输出：

```
shell 1:
$ php worker.php
[*] Waiting for messages. To exit press CTRL+C

-------

shell 2:
$ php worker.php
[*] Waiting for messages. To exit press CTRL+C
```

启动接收器后，我们发送以下消息给队列：

```
$ php new_task.php First message.
$ php new_task.php Second message..
$ php new_task.php Third message...
$ php new_task.php Fourth message....
$ php new_task.php Fifth message.....
```

让我们看看我们的worker执行结果：

```
shell 1:
$php worker.php
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'First message.'
[x] Received 'Third message...'
[x] Received 'Fifth message.....'

-------
shell 2:
$ php worker.php
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Second message..'
[x] Received 'Fourth message....'
```

默认情况下，RabbitMQ按顺序将每个消息发送给下一个消费者。平均每个消费者将获得相同数量的消息。这种分发消息的方式称为循环。尝试这个使用三个worker或者更多。

消息确认

执行任务可能需要几秒钟。你可能会想知道，如果一个消费者开始一个漫长的任务，并且只有部分完成，会发生什么。我们当前的代码，一旦消息发送给消费者，就会将消息从内存中删除。在这种情况下，如果某个worker一旦被kill掉，我们将丢失它正在处理的消息。并且还将丢失所有发送给该worker但尚未处理的消息。

但是我们不想丢失任何任务，如果一个worker结束，我们希望将任务交给另外一个worker.

为了确保消息不会丢失，RabbitMQ支持消息确认。从消费者返回ack（nowledgement）以告诉RabbitMQ已经接收到特定消息，处理并且RabbitMQ可以自由删除它。

如果一个消费者关闭(通道关闭、连接关闭、或者TCP丢失)而不发送确认，RabbitMQ则会认为它没有被处理完成，并将它重新排队。如果同一时间内有其它消费者在线的话，它会迅速发送给其它消费者。这样你可以确保你的消息不会再丢失了，即使突然有个worker异常结束了。

没有任何消息超时，当有消费者死亡时，RabbitMQ将重新发送消息。即使一个任务需要很长的时间，也会很好的处理。

默认情况下，消息确认是关闭的。现在是时候打开消息确认了，将basic_comsume的第四个参数设置为false(true的意思是关闭消息确认)，现在一旦我们完成一个工作，就从worker中发送一个正确的确认。

```
$callback = function($msg) {
    echo " [x] Received ",$msg->body,"\n";
    sleep(substr_count($msg->body,'.'));
    $msg->delivery_info['channel']->basic_ack($msg->delivery_info['delivery_tag']);
};

$channel->basic_consume('task_queue','',false,false,false,false,$callback);
```

使用这段代码，我们可以确定当我们使用CTRL + C 来kill掉worker，也不会有信息丢失，在worker结束后的不久，所有未确认消息也将重新发送。

忘记确认消息

> 错过basic_ack是一个常见的错误。这是一个很容易的错误，但是造成的后果却很严重。当您的客户端退出时，消息将会重新提交，但是RabbitMQ将会消耗非常多的内存，因为它将无法释放任何未得到ask的消息。
>
> 为了调试这种错误，你可以使用rabbitmqctl打印messages_unacknowledged字段：``` $ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged

Listing queues ... hello 0 0 ...done.

```
## 消息持久性

我们已经学会了如何确保即使消费者程序意外终止，任务也不会丢失。但是如果RabbitMQ服务终止了，我们的任务同样是会丢失。

当RabbitMQ服务退出或者崩溃时，它还是会忘记队列和消息，除非你告诉它记住这些。需要做两件事来确保消息不会丢失：我们需要将队列和消息标记为持久化。

首先，我们需要确保RabbitMQ永远不会丢失我们的队列。为了这样做，我们需要声明它是持久的。为此，我们将queue_declare第三个参数设置为true:
```php
$channel->queue_declare('hello',false,true,false,false);
```

虽然这个命令本身是正确的，但它在我们当前的设置中不起作用。这是因为我们已经定义了一个名为hello的队列，它并不是持久的。RabbitMQ不允许您使用不同的参数重新定义一个已经存在的队列并且会向尝试执行该操作的任何程序返回错误。

这里有一个快速的变通方法：让我们定义一个不同名字的队列例如：task_queue:
```

```
$channel->queue_declare('task_queue',false,true,false,false);
```

此标志设置为true，需要应用于生产者和消费者代码。

此刻我们确定名为task_queue的队列即使RabbitMQ重启也不会再丢失消息。现在我们需要将消息标记为持久的 - 通过在初始化AMQPMessage时，提供一个内容为delivery_mode = 2的属性数组来实现。

```
$msg = new AMQPMessage($data,
    array('delivery' => 2)  #将消息标志位持久化的
);
```

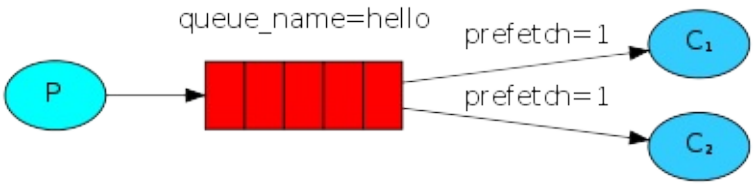
有关于消息持久化的注意事项

> 将消息标志位持久性的，并不能保证消息完全不会丢失。虽然它告诉RabbitMQ将消息保存在磁盘中，但是当RabbitMQ已经接受消息但是尚未保存时，这中间仍然有一个很短的时间差。此外，RabbitMQ不对每一个消息执行fync(2) - 它可能只是保存到缓存中，而不是真正的写入磁盘。持久性的保证还是不够强，但是对于我们简单任务队列来说已经足够了。如果您需要强大的持久性保证，那么您可以使用发布商确认。

公平派遣(Fair dispatch)

您可能注意到，调度仍然不是按照我们想要的方式进行工作。例如在有两个workers的情况下，当所有奇怪的消息都很重甚至消息都很轻的时候，一个worker将不断忙碌，另外一个worker几乎不做任何工作。那么，RabbitMQ不知道任何情况，仍然会均匀发送消息。

这是因为RabbitMQ只是在消息进入队列时分派消息。它并不查看消费者还有多少未确认的消息。它只是盲目的将第n个消息发送给第n个消费者。



为了解决这个问题，我们可以使用basic_qos方法并将prefetch_count设置为1(prefetch_count=1)。这里告诉RabbitMQ不要一次给一个worker一个以上的消息。或者换句话说，不要向一个worker分派新的任务，知道它处理完成并已经确认了上一个消息。相反，它会将其分派给下一个仍然不忙的worker。

```
$channel->basic_qos(null,1,null);
```

有关队列大小的注意事项

> 当所有的worker都忙碌时，你的队列可能会被填满，你会想关注这一点，并可能添加更多的worker，或者一些其它策略。

将上面代码组成一个完整的程序

new_task.php 文件的最终代码：

```
<?php

require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

//将第三个参数设置为true,标记队列是持久化的
$channel->queue_declare('task_queue',false,true,false,false);

//array_slice从数组的第二个元素开始去除后面所有数据，并用implode组成由空格分割的字符串
$data = implode(' ',array_slice($argv,1));

if(empty($data)) $data = "Hello World!";
$msg = new AMQPMessage($data,
    array('delivery_mode' => 2)      # 标记消息是持久的
);

$channel->basic_publish($msg,'','task_queue');

echo " [x] Sent ", $data, "\n";

//释放资源
$channel->close();
$connection->close();
```

下面是worker.php的全部代码：

```php
<?php

require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

$channel->queue_declare('task_queue',false,true,false,false);

echo ' [*] Waiting for messages. To exit press CTRL+C', "\n";

$callback = function($msg){
    echo " [x] Received ", $msg->body, "\n";
    sleep(substr_count($msg->body,'.'));
    echo " [x] Done", "\n";
    $msg->delivery_info['channel']->basic_ask($msg->delivery_info['delivery_tag']);
};

//任务RabbitMQ，不要一次给一个worker发送一个以上的消息，只有当前消息确认处理完成后，才发送新的消息。
$channel->basic_qos(null,1,null);

//将第四个参数设置为false,来向RabbitMQ确认消息
$channel->basic_consume('task_queue','',false,false,false,false,$callback);

while(count($channel->callbacks)){
    $channel->wait();
}

$channel->close();
$connection->close();
```

使用消息确认和预取可以设置工作队列。即使RabbitMQ重新启动，持久性选项仍可以使任务继续存在。

先决条件

本教程假定RabbitMQ在localhost默认端口（5672）上安装和运行。如果您使用其他主机，端口或凭据，连接设置将需要调整。

在哪里获得帮助

如果您在阅读本教程时遇到问题，可以通过邮件列表与我们联系。

发布/订阅

在上一个教程中，我们创建了一个work队列。工作队列背后的假定是每个任务被交付给正好一个worker.在本节中，我们将做一些完全不一样的事情 - 我们将向多个消费者发送消息。此模式称为“发布/订阅”。

为了说明这个模式，我们将建立一个简单的日志系统。它包含两个程序 - 第一个程序将发出日志消息，第二个程序接收和打印它们。

在我们的日志系统中，接受程序的每个运行副本将获得消息。这样我们就可以运行一个接收器并将日志定向到磁盘；并且同时我们可以运行另外一个接收器并在屏幕上看到日志。

本质上，发布的日志消息将被广播到所有接受者。

Exchanges

在教程的前面部分，我们向队列发送和从队列接收消息。现在是时候在Rabbit中引入完整的消息模型。

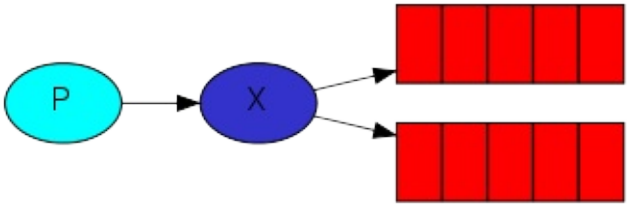
让我们快速了解在前面的教程中介绍的内容：

- 生产者是发送消息的用户应用程序。
- 队列是存储消息的缓冲区。
- 消费者是接收消息的用户应用程序。

RabbitMQ中的消息传递模型的核心思想是，生产者从不会将任何消息直接发送到队列。实际上，生产者通常甚至不知道消息是否被传递到任何队列。

相反，生产者只能向Exchange发送消息。exchange是一个很简单的事情。一方面，它从生产者接收消息，另一方面将它推送到队列。Exchange必须确切的知道对接收到的消息做什么。是否将其添加到特定队列？是否将其添加到多个队列中？或者是否将其丢弃。这里面的规则

由Exchange 类型来定义。



> Exchange是接受生产者消息并且将消息路由到消息队列的关键组件。ExchangeType和Binding决定了消息的路由规则。

这里有几种类型的Exchange可用：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，并且将其称为logs:

```
$channel->exchange_declare('logs','fanout',false,false,false);
```

fanout exchange非常简单。正如你从它的名字上所猜测的，它仅仅只是将它接收到的消息广播到它所知道的所有队列。这正是我们的日志记录器所需要的。

exchanges 清单

要想列出服务器上的exchanges，你可以运行rabbitmqctl:

```
$ sudo 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.trace     topic
amq.topic       topic
logs            fanout
...done.
```

在这个列表中有一些amq.* exchanges 和一些默认(未命名)的exchange。这些是默认创建的，但是现在不太可能需要使用它们。

未命名的*exchange* 在教程前面部分，我们不知道关于exchanges的内容，但是任然能够发送消息到队列。这个是可行的，因为我们使用了默认的exchange，它由空字符串("")标识。

回想一下我们之前发布的消息：

```
$channel->basic_publish($msg,'','hello');
```

这里我们使用了默认的或者无名的exchange：消息将路由到具体由routing_key指定名称的队列中(如果存在)。routing_key是由basic_publish的第二个参数来指定。

现在我们可以发布到我们命名的exchange中：

```
$channel->exchange_declare('logs','fanout',false,false,false);
$channel->basic_publish($msg,'logs');
```

临时队列

你可能记得，我们之前使用具有指定名称的队列(例如hello和task_queue)。能够命名的队列对我们至关重要 - 我们需要将worker执行同一队列。当您想在生产者和消费者之间共享队列时，给队列一个名称是很重要的。

但是这并不适用与我们的日志记录器。我们想听到所有日志消息，而不仅仅是它们的子集。我们也只对当前流动的消息感兴趣，而不是旧的消息。要解决我们需要的两件事情。

其次，一但我们断开消费者，队列应该被自动删除。

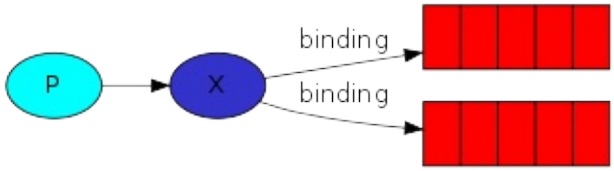
在pp-amqplib客户端中，当我们将队列名称设置为空字符串时，我们使用生成的名称创建非持久队列。

```
list($queue_name, ,) = $channel->queue_declare("");
```

当方法返回时，$queue_name变量包含由RabbitMQ生成的随机队列名称。例如，它可能看起来像amq.gen-JzTY20BRgKO-HjmUJj0wLg。

当声明它的连接关闭时，队列将被删除，因为它被声明为exclusive。

绑定



我们已经创建了一个 fanout exchange和一个队列，现在我们需要告诉exchange发送消息到我们的队列中。exchange 和队列之间的联系我们称之为绑定(Binding)。

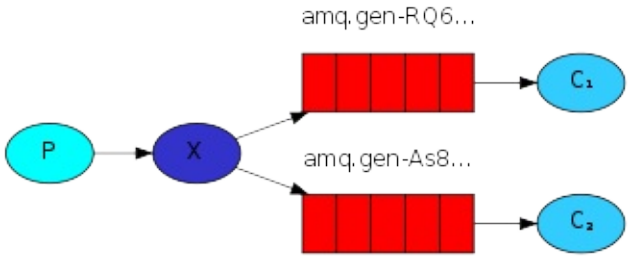
```
$channel->queue_bind($queue_name,'logs');
```

从现在起，logs exchange将消息添加到我们的队列中。

绑定列表

> 你可以使用rabbitmqctl list_bindings来列出现有的绑定

(把他们放在一块) Putting it all together



生产者程序发出日志消息，开起来和以前的教程不太相同。最重要的变化是，我们现在想要发布消息到我们的logs exchange 而不是未命名的队列中。这里是emit_log.php脚本的代码：

```
<?php

require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

$channel->exchange_declare('logs','fanout',false,false,false);

$data = implode(' ',array_slice($argv,1));
if(empty($data)) $data = "info: Hello World!";
$msg = new AMAPMessage($data);

$channel->basic_publish($msg,'logs');

echo " [x] Sent ", $data, "\n";

$channel->close();
$connection->close();
```

如你所见，建立连接后，我们声明了fanout类型的exchange。这个步骤是必要的，因为不允许向不存在的exchange发布消息。

如果没有绑定队列到exchange，消息将会丢失，但是对我们而言，并不重要。如果没有消费者正在监听队列，我们可以安全的将消息丢弃。

receive_logs.php的代码：

```
<?php

require_once __DIR__ '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');

$channel->exchange_declare('logs','fanout',false,false,false);

list($queue_name, ,) = $change->queue_declare("",false,false,true,false);

$channel->queue_bind($queue_name,'logs');

echo ' [*] Waiting for logs. To exit press CTRL+C', "\n";

$callback = function($msg) {
    echo " [x] ", $msg-body, "\n";
}

$channel->basic_consume($queue_name,'',false,true,false,false,$callback);

while(count($channel->callbacks)) {
    $channel->wait();
}

$channel->close();
$connection->close();
```

如果你想将日志信息保存到文件中，只需要在终端中执行下面命令：

```
$ php receive_logs.php > logs_from_rabbit.log
```

如果你想在屏幕上看到日志，只要在终端中执行下面命令：

```
$ php receive_logs.php
```

当然，你要执行日志生产程序：

```
$ php emit_log.php
```

使用rabbitmqctl list_bindings 你可以验证代码实际上绑定和我们想要的队列。有两个receive_logs.php，你应该看到类似的内容：

```
$ sudo rabbitmqctl list_bindings
Listing bindings ...
logs    exchange        amq.gen-JzTY20BRgKO-HjmUJj0wLg  queue           []
logs    exchange        amq.gen-vso0PVvyiRIL2WoV3i48Yg  queue           []
...done.
```

结果是很直接的：来自exchange logs的数据到具有服务器分配的名称的两个队列，这正是我们的意图。

先决条件

本教程假定RabbitMQ在localhost默认端口（5672）上安装和运行。如果您使用其他主机，端口或凭据，连接设置将需要调整。

在哪里获得帮助

如果您在阅读本教程时遇到问题，可以通过邮件列表与我们联系。

路由

在上一节教程中，我们创建了一个简单的日志系统。我们可以向多个接受者广播日志消息。

在本教程中，我们将向其添加一个功能 - 我们将只允许订阅一个消息的子集。例如，我们能够仅将关键错误消息定向到日志文件(以节省磁盘空间)，同时仍能够在控制台打印所有日志消息。

绑定

在前面的例子中，我们已经创建了绑定。你可以调用以下代码：

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

绑定将exchange和队列关联起来。这可以简单的理解为：队列对来自这个exchange的消息感兴趣。

bindings 可以接受一个额外的 routing_key参数。为了避免与basic_publish参数混淆，我们将它称为绑定键。下面是使用绑定建的示例：

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

绑定键的含义取决于exchange的类型，我们以前使用的fanout类型，忽略了它的价值。

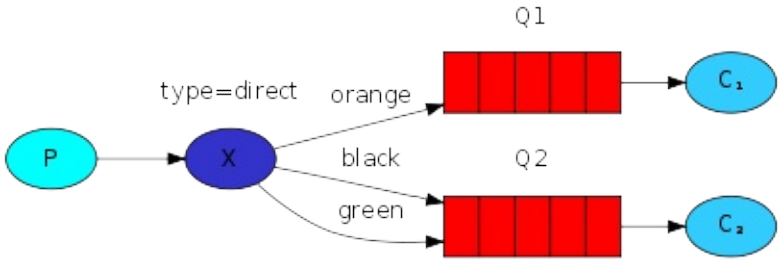
Direct exchange

上一个教程中，我们的日志系统向所有消费者广播所有消息。我们希望为其扩展通过消息严重性来过滤消息的功能。例如，我们希望仅将关键错误消息写入磁盘，而不在警告或者信息日志上浪费磁盘空间。

上个案例中，我们使用了fanout类型的exchange，它没有太多的灵活性，只能将消息广播给所有接受者。

本节中，我们将使用direct类型来替代fanout。 direct exchange的路由算法非常简单 - 消息只会发往与其绑定的key和路由秘钥完全匹配的队列。

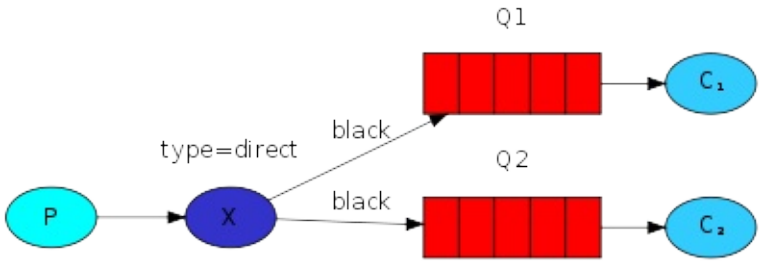
为了解释这一点，请看看以下插图：



这上图中，我们可以看到一个direct类型的exchange绑定到两个队列中。第一个队列绑定了一个名为orange的key，第二个队列绑定了两个key，一个是名为black，还有一个叫green.

在这样的设置中，使用orange为路由key发布到exchange的消息将被发送到队列Q1中。路由key为black或者green的消息将会发送到Q2队列上。其他的消息都讲被丢弃。

多重绑定



可以使用相同的key来绑定多个队列。在我们的例子中，我们将使用black为路由键来绑定X(exchange) 和 Q1(queue)。在这种情况下，direct exchange像fanout一样，我们将消息广播到使用相同key的队列上。使用black为路由键的消息将被发送到Q1和Q2上。

发送日志

我们将上面的模型用于我们的日志系统，我们使用direct类型来取代fanout。我们将使用日志的严重等级作为路由键。这样接受器可以选择它想要接受的日志等级。

和往常一样，首先我们需要创建一个exchange：

```
$channel->exchange_declare('direct_logs','direct',flase,false,false);
```

然后我们准备发送消息：

```
$channel->exchange_declare('direct_logs','direct',false,false,false);
$channel->basic_publish($msg, 'direct_logs', $severity);
```

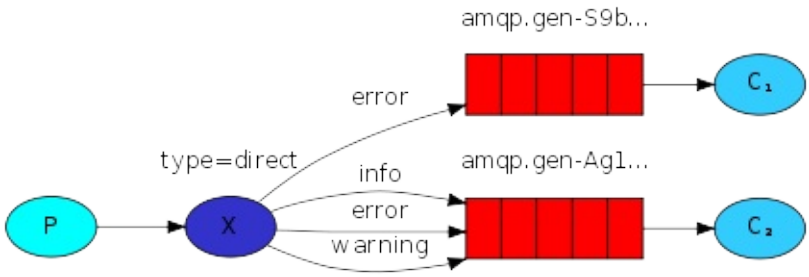
为了简化，我们假设severity可以是info、warning、error其中的一个。

订阅

接受程序还是和上一个教程一样，只有一个地方需要修改：我们将为每个我们感兴趣的严重等级创建一个新的绑定。

```
foreach($severities as $severity){
    $channel->queue_bind($queue_name,'direct_logs',$severity);
}
```

将代码组织在一起



emit_log_direct.php

```php
<?php

require_once __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

$channel->exchange_declare('direct_logs','direct',false,false,false);

$severity = isset($argv[1]) && !empty($argv[1]) ? $argv[1] : 'info';

$data = implode(' ',array_slice($argv,2));

if(empty($data)) $data = "Hello World!";

$msg = new AMQPMessage($data);

$channel->basic_publish($msg,'direct_logs',$severity);

echo " [x] Sent ", $severity, ':', $data, "\n";

$channel->close();
$connection->close();
```

receive_logs_direct.php:

```
<?php

require __DIR__ . '/vendor/autoload.php';
use PhpAmqpLib\Connection\AMQPStreamConnection;

$connection = new AMQPStreamConnection('localhost',5672,'guest','guest');
$channel = $connection->channel();

$channel->exchange_declare('direct_logs', 'direct', false, false, false);

list($queue_name, ,) = $channel->queue_declare("", false, false, true, false);

$severities = array_slice($argv,1);
if(empty($severities)) {
    file_put_contents('php://stderr', "Usage: $argv[0] [info] [warning] [error] \n");
    exit(1);
}

foreach($severities as $severity) {
    $channel->queue_bind($queue_name, 'direct_logs', $severity);
}

echo ' [*] Waiting for logs. To exit press CTRL+C', "\n";

$callback = function($msg) {
    echo " [x]", $msg->delivery_info['routing_key'], ":", $msg->body, "\n";
};

$channel->basic_consume($queue_name, '', false, true, false, false, $callback);

while(count($hannel->callbacks)) {
    $channel->wait();
}

$channel->close();
$connection->close();
```

如果你只想保存'warning' 和 'error'，而不是 'info'到文件，你只需要打开终端输入下面脚本就行：

```
$ php receive_logs_direct.php warning error > logs_from_rabbit.log
```

如果你想在屏幕上看到所有的日志信息，你可以在终端输入下列脚本：

```
$ php receive_logs_direct.php info warning error
```

例如只想发出一个错误日志，只需要键入：

```
$ php emit_log_direct.php error "Run. Run. Or it will explode."
  [x] Sent 'error' : 'Run. Run. Or it will explode.'
```