# n官方文档 <https://www.rabbitmq.com/admin-guide.html>

# 安装

<https://packagecloud.io/rabbitmq/erlang/packages/el/7/erlang-22.0.4-1.el7.x86_64.rpm>

在这里下载erlang-22.0.4-1.el7.x86\_64.rpm el7是针对centos7的版本

<https://packagecloud.io/rabbitmq/rabbitmq-server/packages/el/7/rabbitmq-server-3.7.15-1.el7.noarch.rpm> 在这里下载rabbitmq-server-3.7.15-1.el7.noarch.rpm

下载的erlang要是兼容rabbitmq的版本

使用rpm-ivh顺序安装上面两个

## 错误排解：

libcrypto.so.10(OPENSSL\_1.0.2)(64bit) 被 erlang-22.0.4-1.el7.x86\_64 需要---------

在这里下载[openssl-libs-1.0.2k-16.el7.x86\_64.rpm](http://rpmfind.net/linux/centos/7.6.1810/os/x86_64/Packages/openssl-libs-1.0.2k-16.el7.x86_64.rpm)  
<http://rpmfind.net/linux/rpm2html/search.php?query=libcrypto.so.10%28OPENSSL_1.0.2%29%2864bit%29&submit=Search+...&system=&arch=>

然后强制安装rpm -ivh openssl-libs-1.0.2k-16.el7.x86\_64.rpm --force

再rpm -ivh erlang-22.0.4-1.el7.x86\_64.rpm

socat 被 rabbitmq-server-3.7.15-1.el7.noarch 需要----------------------------

在这里下载**[socat-1.7.3.2-2.el7.x86\_64.rpm](http://mirror.centos.org/centos/7/os/x86_64/Packages/socat-1.7.3.2-2.el7.x86_64.rpm" \t "_blank)**

<https://centos.pkgs.org/7/centos-x86_64/socat-1.7.3.2-2.el7.x86_64.rpm.html>

然后rpm -ivh socat-1.7.3.2-2.el7.x86\_64.rpm

再rpm -ivh rabbitmq-server-3.7.15-1.el7.noarch.rpm

# 启动服务

rabbitmq-server在当前窗口启动

rabbitmq-server -detached后台运行

rabbitmqctl stop停止服务

systemctl enable rabbitmq-server.service开机启动

# 添加用户并配置权限

Setting up RabbitMQ

要使用celery,我们需要创建一个RabBMQ用户,一个虚拟主机,并允许用户访问该虚拟主机

rabbitmqctl add\_user myuser mypassword

rabbitmqctl add\_vhost myvhost

rabbitmqctl set\_user\_tags myuser mytag

rabbitmqctl set\_permissions -p myvhost myuser ".\*" ".\*" ".\*"

Substitute in appropriate values for**"myuser"**,**"mypassword"**,and**"myvhost"**above.

**"mytag"是用来标记管理权限的，为administrator即为管理员**

rabbitmqctl status 查看状态

Status of node rabbit@myhost ...

[{running\_applications,[{rabbit,"RabbitMQ","1.7.1"},

{mnesia,"MNESIA CXC 138 12","4.4.12"},

{os\_mon,"CPO CXC 138 46","2.2.4"},

{sasl,"SASL CXC 138 11","2.1.8"},

{stdlib,"ERTS CXC 138 10","1.16.4"},

{kernel,"ERTS CXC 138 10","2.13.4"}]},

{nodes,[rabbit@myhost]},

{running\_nodes,[rabbit@myhost]}]

...done.

# 开启web管理界面（可选）

rabbitmq-plugins enable rabbitmq\_management 启动web监控台

http://192.168.181.128:15672/ 访问web ui地址

只有设置tag为administrator的才可以登陆界面进行管理

client端通信端口 5672

web ui访问端口15672

server间通信端口25672

erlang发现端口4369

至此安装完成，可以开始使用--------------------------------------------

1. 1、RabbitMQ介绍
2. 1.1、什么是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。

1. 1.2、什么是AMQP?

  AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。它从生产者接收消息并递送给消费者，在这个过程中，根据规则进行路由，缓存与持久化。

  AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。

  RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：[Python](http://lib.csdn.net/base/11)、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

  而在AMQP中主要有两个组件：Exchange 和 Queue （在 AMQP 1.0 里还会有变动），如下图所示，绿色的 X 就是 Exchange ，红色的是 Queue ，这两者都在 Server 端，又称作 Broker ，这部分是 RabbitMQ 实现的，而蓝色的则是客户端，通常有 Producer 和 Consumer 两种类型：

1. 1.3、RabbitMQ的基础概念

* Broker：简单来说就是消息队列服务器实体
* Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列
* Queue：消息队列载体，每个消息都会被投入到一个或多个队列
* Binding：绑定，它的作用就是把exchange和queue按照路由规则绑定起来
* Routing Key：路由关键字，exchange根据这个关键字进行消息投递
* vhost：虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离
* producer：消息生产者，就是投递消息的程序
* consumer：消息消费者，就是接受消息的程序
* channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务

1. 1.4、RabbitMQ的特性

* 可靠性：包括消息持久化，消费者和生产者的消息确认
* 灵活路由：遵循AMQP协议，支持多种Exchange类型实现不同路由策略
* 分布式：集群的支持，包括本地网络与远程网络
* 高可用性：支持主从备份与镜像队列
* 多语言支持：支持多语言的客户端
* WEB界面管理：可以管理用户权限，exhange，queue，binding，与实时监控
* 访问控制：基于vhosts实现访问控制
* 调试追踪：支持tracing，方便调试

1. 2、RabbitMQ的官网在哪里？

<http://www.rabbitmq.com/>

1. 3、RabbitMQ在哪里下载?

<http://www.rabbitmq.com/download.html>

1. 4、如何安装RabbitMQ

## 4.1、通过安装RabbitMQ的源来安装

  在Ubuntu上安装RabbitMQ非常简单

lion@ubuntu1404:~$ sudo echo 'deb http://www.rabbitmq.com/debian/ testing main' | sudo tee /etc/apt/sources.list.d/rabbitmq.list

lion@ubuntu1404:~$ wget -O- https://www.rabbitmq.com/rabbitmq-release-signing-key.asc | sudo apt-key add -

lion@ubuntu1404:~$ sudo apt-get update

lion@ubuntu1404:~$ sudo apt-get install rabbitmq-server

  其他系统安装方法:<http://www.rabbitmq.com/download.html>

1. 4.2、通过源码安装

  本文中的实例，主要通过源码安装来演示。

4.2.1、安装Erlang

  相关安装文档：http://erlang.org/erldoc

lion@node1:~$ sudo apt-get install -y erlang-nox erlang-dev erlang-src

4.2.2、Rabbitmq 3.6.3安装

  相关安装文档：<http://www.rabbitmq.com/install-generic-unix.html>。

  我们先下载源码并解压

lion@node1:~$ mkdir -p \_app

lion@node1:~/\_app$ wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.3/rabbitmq-server-generic-unix-3.6.3.tar.xz

lion@node1:~/\_app$ xz -d rabbitmq-server-generic-unix-3.6.3.tar.xz

lion@node1:~/\_app$ tar -xvf rabbitmq-server-generic-unix-3.6.3.tar

lion@node1:~/\_app$ cd rabbitmq\_server-3.6.3

  设置环境变量$RABBITMQ\_HOME

lion@node1:~$ vi .bashrc

  在.bashrc中添加以下内容

export RABBITMQ\_HOME="/home/lion/\_app/rabbitmq\_server-3.6.3"

export PATH="$RABBITMQ\_HOME/sbin:$PATH"

  让环境变量生效

lion@node1:~$ source .bashrc

  启动Rabbitmq

lion@node1:~$ rabbitmq-server

  安装以后可以通过下面的命令，停止、启动：

lion@node1:~$ rabbitmqctl stop

lion@node1:~$ rabbitmqctl start

1. 4.3、开启web管理插件

  创建一个用户lion，并设置密码123456：

lion@node1:~$ rabbitmqctl add\_user lion 123456

  可以通过下面的命令，查看现有的用户更表

lion@node1:~$ rabbitmqctl list\_users

Listing users ...

guest [administrator]

lion []

  这个时候lion用户是不能访问web管理插件的，需要配置用户角色，用户角色可分为五类，超级管理员, 监控者, 策略制定者, 普通管理者以及其他。

* 超级管理员(administrator)

  可登陆管理控制台(启用management plugin的情况下)，可查看所有的信息，并且可以对用户，策略(policy)进行操作。

* 监控者(monitoring)

  可登陆管理控制台(启用management plugin的情况下)，同时可以查看rabbitmq节点的相关信息(进程数，内存使用情况，磁盘使用情况等)

* 策略制定者(policymaker)

  可登陆管理控制台(启用management plugin的情况下), 同时可以对policy进行管理。但无法查看节点的相关信息。

* 普通管理者(management)

  仅可登陆管理控制台(启用management plugin的情况下)，无法看到节点信息，也无法对策略进行管理。

* 其他

  无法登陆管理控制台，通常就是普通的生产者和消费者。

  通过下面的命令，可以将lion添加到administrator用户组：

lion@node1:~$ rabbitmqctl set\_user\_tags lion administrator

  然后可以用下面的命令来启用/信上管理插件：

lion@node1:~$ rabbitmq-plugins enable rabbitmq\_management (启用插件)

lion@node1:~$ rabbitmq-plugins disable rabbitmq\_management (禁用插件)

  通过浏览访问 http://127.0.0.1:15672/  
  输入用户名lion,密码123456就可以看到后台了。

*rabbitmqctl的更多命令参考：http://www.rabbitmq.com/man/rabbitmqctl.1.man.html*

1. 4.4、RabbitMQ 的配置文件介绍

  RabbitMQ的配置文件目录默认是$RABBITMQ\_HOME/etc/rabbitmq/rabbitmq-env.conf，如果文件不存在，可以自己创建。

  配置文件全部说明地址：http://www.rabbitmq.com/configure.html#configuration-file

%% -\*- mode: erlang -\*-

%% ----------------------------------------------------------------------------

%% RabbitMQ Sample Configuration File.

%%

%% See http://www.rabbitmq.com/configure.html for details.

%% ----------------------------------------------------------------------------

[

{rabbit,

[%%

%% Network Connectivity

%% ====================

%%

%% By default, RabbitMQ will listen on all interfaces, using

%% the standard (reserved) AMQP port.

%% 默认的监听端口

%% {tcp\_listeners, [5672]},

%% To listen on a specific interface, provide a tuple of {IpAddress, Port}.

%% For example, to listen only on localhost for both IPv4 and IPv6:

%% 也可以使用下面的格式进行指定IP和端口的监听

%% {tcp\_listeners, [{"127.0.0.1", 5672},

%% {"::1", 5672}]},

%% SSL listeners are configured in the same fashion as TCP listeners,

%% including the option to control the choice of interface.

%% SSL连接端口配置

%% {ssl\_listeners, [5671]},

%% Number of Erlang processes that will accept connections for the TCP

%% and SSL listeners.

%% TCP连接的进程数

%% {num\_tcp\_acceptors, 10},

%% {num\_ssl\_acceptors, 1},

%% Maximum time for AMQP 0-8/0-9/0-9-1 handshake (after socket connection

%% and SSL handshake), in milliseconds.

%% 超时时间，单位毫秒

%% {handshake\_timeout, 10000},

%% Log levels (currently just used for connection logging).

%% One of 'debug', 'info', 'warning', 'error' or 'none', in decreasing

%% order of verbosity. Defaults to 'info'.

%% 日志的级别，默认是info

%% {log\_levels, [{connection, info}, {channel, info}]},

%% Set to 'true' to perform reverse DNS lookups when accepting a

%% connection. Hostnames will then be shown instead of IP addresses

%% in rabbitmqctl and the management plugin.

%%

%% {reverse\_dns\_lookups, true},

%%

%% Security / AAA

%% ==============

%% 安全配置

%% The default "guest" user is only permitted to access the server

%% via a loopback interface (e.g. localhost).

%% {loopback\_users, [<<"guest">>]},

%%

%% Uncomment the following line if you want to allow access to the

%% guest user from anywhere on the network.

%% {loopback\_users, []},

%% Configuring SSL.

%% See http://www.rabbitmq.com/ssl.html for full documentation.

%%

%% {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, false}]},

%% Choose the available SASL mechanism(s) to expose.

%% The two default (built in) mechanisms are 'PLAIN' and

%% 'AMQPLAIN'. Additional mechanisms can be added via

%% plugins.

%%

%% See http://www.rabbitmq.com/authentication.html for more details.

%%

%% {auth\_mechanisms, ['PLAIN', 'AMQPLAIN']},

%% Select an authentication database to use. RabbitMQ comes bundled

%% with a built-in auth-database, based on mnesia.

%%

%% {auth\_backends, [rabbit\_auth\_backend\_internal]},

%% Configurations supporting the rabbitmq\_auth\_mechanism\_ssl and

%% rabbitmq\_auth\_backend\_ldap plugins.

%%

%% NB: These options require that the relevant plugin is enabled.

%% See http://www.rabbitmq.com/plugins.html for further details.

%% The RabbitMQ-auth-mechanism-ssl plugin makes it possible to

%% authenticate a user based on the client's SSL certificate.

%%

%% To use auth-mechanism-ssl, add to or replace the auth\_mechanisms

%% list with the entry 'EXTERNAL'.

%%

%% {auth\_mechanisms, ['EXTERNAL']},

%% The rabbitmq\_auth\_backend\_ldap plugin allows the broker to

%% perform authentication and authorisation by deferring to an

%% external LDAP server.

%%

%% For more information about configuring the LDAP backend, see

%% http://www.rabbitmq.com/ldap.html.

%%

%% Enable the LDAP auth backend by adding to or replacing the

%% auth\_backends entry:

%%

%% {auth\_backends, [rabbit\_auth\_backend\_ldap]},

%% This pertains to both the rabbitmq\_auth\_mechanism\_ssl plugin and

%% STOMP ssl\_cert\_login configurations. See the rabbitmq\_stomp

%% configuration section later in this file and the README in

%% https://github.com/rabbitmq/rabbitmq-auth-mechanism-ssl for further

%% details.

%%

%% To use the SSL cert's CN instead of its DN as the username

%%

%% {ssl\_cert\_login\_from, common\_name},

%% SSL handshake timeout, in milliseconds.

%%

%% {ssl\_handshake\_timeout, 5000},

%% Password hashing implementation. Will only affect newly

%% created users. To recalculate hash for an existing user

%% it's necessary to update her password.

%%

%% {password\_hashing\_module, rabbit\_password\_hashing\_sha256},

%%

%% Default User / VHost

%% ====================

%% 用户访问设置

%% On first start RabbitMQ will create a vhost and a user. These

%% config items control what gets created. See

%% http://www.rabbitmq.com/access-control.html for further

%% information about vhosts and access control.

%%

%% {default\_vhost, <<"/">>},

%% {default\_user, <<"guest">>},

%% {default\_pass, <<"guest">>},

%% {default\_permissions, [<<".\*">>, <<".\*">>, <<".\*">>]},

%% Tags for default user

%%

%% For more details about tags, see the documentation for the

%% Management Plugin at http://www.rabbitmq.com/management.html.

%%

%% {default\_user\_tags, [administrator]},

%%

%% Additional network and protocol related configuration

%% =====================================================

%%

%% Set the default AMQP heartbeat delay (in seconds).

%% 设置默认AMQP心跳延迟（秒）

%% {heartbeat, 600},

%% Set the max permissible size of an AMQP frame (in bytes).

%%

%% {frame\_max, 131072},

%% Set the max frame size the server will accept before connection

%% tuning occurs

%%

%% {initial\_frame\_max, 4096},

%% Set the max permissible number of channels per connection.

%% 0 means "no limit".

%%

%% {channel\_max, 128},

%% Customising Socket Options.

%%

%% See (http://www.erlang.org/doc/man/inet.html#setopts-2) for

%% further documentation.

%%

%% {tcp\_listen\_options, [{backlog, 128},

%% {nodelay, true},

%% {exit\_on\_close, false}]},

%%

%% Resource Limits & Flow Control

%% ==============================

%%

%% See http://www.rabbitmq.com/memory.html for full details.

%% Memory-based Flow Control threshold.

%%

%% {vm\_memory\_high\_watermark, 0.4},

%% Alternatively, we can set a limit (in bytes) of RAM used by the node.

%%

%% {vm\_memory\_high\_watermark, {absolute, 1073741824}},

%%

%% Or you can set absolute value using memory units.

%%

%% {vm\_memory\_high\_watermark, {absolute, "1024M"}},

%%

%% Supported units suffixes:

%%

%% k, kiB: kibibytes (2^10 bytes)

%% M, MiB: mebibytes (2^20)

%% G, GiB: gibibytes (2^30)

%% kB: kilobytes (10^3)

%% MB: megabytes (10^6)

%% GB: gigabytes (10^9)

%% Fraction of the high watermark limit at which queues start to

%% page message out to disc in order to free up memory.

%%

%% Values greater than 0.9 can be dangerous and should be used carefully.

%% 内存最大使用比例

%% {vm\_memory\_high\_watermark\_paging\_ratio, 0.5},

%% Interval (in milliseconds) at which we perform the check of the memory

%% levels against the watermarks.

%% 检查内存的间隔(毫秒)

%% {memory\_monitor\_interval, 2500},

%% Set disk free limit (in bytes). Once free disk space reaches this

%% lower bound, a disk alarm will be set - see the documentation

%% listed above for more details.

%%

%% {disk\_free\_limit, 50000000},

%%

%% Or you can set it using memory units (same as in vm\_memory\_high\_watermark)

%% {disk\_free\_limit, "50MB"},

%% {disk\_free\_limit, "50000kB"},

%% {disk\_free\_limit, "2GB"},

%% Alternatively, we can set a limit relative to total available RAM.

%%

%% Values lower than 1.0 can be dangerous and should be used carefully.

%% {disk\_free\_limit, {mem\_relative, 2.0}},

%%

%% Misc/Advanced Options

%% =====================

%%

%% NB: Change these only if you understand what you are doing!

%%

%% To announce custom properties to clients on connection:

%%

%% {server\_properties, []},

%% How to respond to cluster partitions.

%% See http://www.rabbitmq.com/partitions.html for further details.

%%

%% {cluster\_partition\_handling, ignore},

%% Make clustering happen \*automatically\* at startup - only applied

%% to nodes that have just been reset or started for the first time.

%% See http://www.rabbitmq.com/clustering.html#auto-config for

%% further details.

%% 设置集群启动的节点

%% {cluster\_nodes, {['rabbit@my.host.com'], disc}},

%% Interval (in milliseconds) at which we send keepalive messages

%% to other cluster members. Note that this is not the same thing

%% as net\_ticktime; missed keepalive messages will not cause nodes

%% to be considered down.

%% 集群消息同步的时间(毫秒)

%% {cluster\_keepalive\_interval, 10000},

%% Set (internal) statistics collection granularity.

%%

%% {collect\_statistics, none},

%% Statistics collection interval (in milliseconds).

%%

%% {collect\_statistics\_interval, 5000},

%% Explicitly enable/disable hipe compilation.

%%

%% {hipe\_compile, true},

%% Timeout used when waiting for Mnesia tables in a cluster to

%% become available.

%%

%% {mnesia\_table\_loading\_timeout, 30000},

%% Size in bytes below which to embed messages in the queue index. See

%% http://www.rabbitmq.com/persistence-conf.html

%%

%% {queue\_index\_embed\_msgs\_below, 4096}

]},

%% ----------------------------------------------------------------------------

%% Advanced Erlang Networking/Clustering Options.

%%

%% See http://www.rabbitmq.com/clustering.html for details

%% ----------------------------------------------------------------------------

{kernel,

[%% Sets the net\_kernel tick time.

%% Please see http://erlang.org/doc/man/kernel\_app.html and

%% http://www.rabbitmq.com/nettick.html for further details.

%%

%% {net\_ticktime, 60}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ Management Plugin

%%

%% See http://www.rabbitmq.com/management.html for details

%% ----------------------------------------------------------------------------

{rabbitmq\_management,

[%% Pre-Load schema definitions from the following JSON file. See

%% http://www.rabbitmq.com/management.html#load-definitions

%%

%% {load\_definitions, "/path/to/schema.json"},

%% Log all requests to the management HTTP API to a file.

%% 所有请求的HTTP API文件日志的路径。

%% {http\_log\_dir, "/path/to/access.log"},

%% Change the port on which the HTTP listener listens,

%% specifying an interface for the web server to bind to.

%% Also set the listener to use SSL and provide SSL options.

%% Web管理的地址和端口

%% {listener, [{port, 12345},

%% {ip, "127.0.0.1"},

%% {ssl, true},

%% {ssl\_opts, [{cacertfile, "/path/to/cacert.pem"},

%% {certfile, "/path/to/cert.pem"},

%% {keyfile, "/path/to/key.pem"}]}]},

%% One of 'basic', 'detailed' or 'none'. See

%% http://www.rabbitmq.com/management.html#fine-stats for more details.

%% {rates\_mode, basic},

%% Configure how long aggregated data (such as message rates and queue

%% lengths) is retained. Please read the plugin's documentation in

%% http://www.rabbitmq.com/management.html#configuration for more

%% details.

%%

%% {sample\_retention\_policies,

%% [{global, [{60, 5}, {3600, 60}, {86400, 1200}]},

%% {basic, [{60, 5}, {3600, 60}]},

%% {detailed, [{10, 5}]}]}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ Shovel Plugin

%%

%% See http://www.rabbitmq.com/shovel.html for details

%% ----------------------------------------------------------------------------

{rabbitmq\_shovel,

[{shovels,

[%% A named shovel worker.

%% {my\_first\_shovel,

%% [

%% List the source broker(s) from which to consume.

%%

%% {sources,

%% [%% URI(s) and pre-declarations for all source broker(s).

%% {brokers, ["amqp://user:password@host.domain/my\_vhost"]},

%% {declarations, []}

%% ]},

%% List the destination broker(s) to publish to.

%% {destinations,

%% [%% A singular version of the 'brokers' element.

%% {broker, "amqp://"},

%% {declarations, []}

%% ]},

%% Name of the queue to shovel messages from.

%%

%% {queue, <<"your-queue-name-goes-here">>},

%% Optional prefetch count.

%%

%% {prefetch\_count, 10},

%% when to acknowledge messages:

%% - no\_ack: never (auto)

%% - on\_publish: after each message is republished

%% - on\_confirm: when the destination broker confirms receipt

%%

%% {ack\_mode, on\_confirm},

%% Overwrite fields of the outbound basic.publish.

%%

%% {publish\_fields, [{exchange, <<"my\_exchange">>},

%% {routing\_key, <<"from\_shovel">>}]},

%% Static list of basic.properties to set on re-publication.

%%

%% {publish\_properties, [{delivery\_mode, 2}]},

%% The number of seconds to wait before attempting to

%% reconnect in the event of a connection failure.

%%

%% {reconnect\_delay, 2.5}

%% ]} %% End of my\_first\_shovel

]}

%% Rather than specifying some values per-shovel, you can specify

%% them for all shovels here.

%%

%% {defaults, [{prefetch\_count, 0},

%% {ack\_mode, on\_confirm},

%% {publish\_fields, []},

%% {publish\_properties, [{delivery\_mode, 2}]},

%% {reconnect\_delay, 2.5}]}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ Stomp Adapter

%%

%% See http://www.rabbitmq.com/stomp.html for details

%% ----------------------------------------------------------------------------

{rabbitmq\_stomp,

[%% Network Configuration - the format is generally the same as for the broker

%% Listen only on localhost (ipv4 & ipv6) on a specific port.

%% {tcp\_listeners, [{"127.0.0.1", 61613},

%% {"::1", 61613}]},

%% Listen for SSL connections on a specific port.

%% {ssl\_listeners, [61614]},

%% Number of Erlang processes that will accept connections for the TCP

%% and SSL listeners.

%%

%% {num\_tcp\_acceptors, 10},

%% {num\_ssl\_acceptors, 1},

%% Additional SSL options

%% Extract a name from the client's certificate when using SSL.

%%

%% {ssl\_cert\_login, true},

%% Set a default user name and password. This is used as the default login

%% whenever a CONNECT frame omits the login and passcode headers.

%%

%% Please note that setting this will allow clients to connect without

%% authenticating!

%%

%% {default\_user, [{login, "guest"},

%% {passcode, "guest"}]},

%% If a default user is configured, or you have configured use SSL client

%% certificate based authentication, you can choose to allow clients to

%% omit the CONNECT frame entirely. If set to true, the client is

%% automatically connected as the default user or user supplied in the

%% SSL certificate whenever the first frame sent on a session is not a

%% CONNECT frame.

%%

%% {implicit\_connect, true}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ MQTT Adapter

%%

%% See https://github.com/rabbitmq/rabbitmq-mqtt/blob/stable/README.md

%% for details

%% ----------------------------------------------------------------------------

{rabbitmq\_mqtt,

[%% Set the default user name and password. Will be used as the default login

%% if a connecting client provides no other login details.

%%

%% Please note that setting this will allow clients to connect without

%% authenticating!

%%

%% {default\_user, <<"guest">>},

%% {default\_pass, <<"guest">>},

%% Enable anonymous access. If this is set to false, clients MUST provide

%% login information in order to connect. See the default\_user/default\_pass

%% configuration elements for managing logins without authentication.

%%

%% {allow\_anonymous, true},

%% If you have multiple chosts, specify the one to which the

%% adapter connects.

%%

%% {vhost, <<"/">>},

%% Specify the exchange to which messages from MQTT clients are published.

%%

%% {exchange, <<"amq.topic">>},

%% Specify TTL (time to live) to control the lifetime of non-clean sessions.

%%

%% {subscription\_ttl, 1800000},

%% Set the prefetch count (governing the maximum number of unacknowledged

%% messages that will be delivered).

%%

%% {prefetch, 10},

%% TCP/SSL Configuration (as per the broker configuration).

%%

%% {tcp\_listeners, [1883]},

%% {ssl\_listeners, []},

%% Number of Erlang processes that will accept connections for the TCP

%% and SSL listeners.

%%

%% {num\_tcp\_acceptors, 10},

%% {num\_ssl\_acceptors, 1},

%% TCP/Socket options (as per the broker configuration).

%%

%% {tcp\_listen\_options, [{backlog, 128},

%% {nodelay, true}]}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ AMQP 1.0 Support

%%

%% See https://github.com/rabbitmq/rabbitmq-amqp1.0/blob/stable/README.md

%% for details

%% ----------------------------------------------------------------------------

{rabbitmq\_amqp1\_0,

[%% Connections that are not authenticated with SASL will connect as this

%% account. See the README for more information.

%%

%% Please note that setting this will allow clients to connect without

%% authenticating!

%%

%% {default\_user, "guest"},

%% Enable protocol strict mode. See the README for more information.

%%

%% {protocol\_strict\_mode, false}

]},

%% ----------------------------------------------------------------------------

%% RabbitMQ LDAP Plugin

%%

%% See http://www.rabbitmq.com/ldap.html for details.

%%

%% ----------------------------------------------------------------------------

{rabbitmq\_auth\_backend\_ldap,

[%%

%% Connecting to the LDAP server(s)

%% ================================

%%

%% Specify servers to bind to. You \*must\* set this in order for the plugin

%% to work properly.

%%

%% {servers, ["your-server-name-goes-here"]},

%% Connect to the LDAP server using SSL

%%

%% {use\_ssl, false},

%% Specify the LDAP port to connect to

%%

%% {port, 389},

%% LDAP connection timeout, in milliseconds or 'infinity'

%%

%% {timeout, infinity},

%% Enable logging of LDAP queries.

%% One of

%% - false (no logging is performed)

%% - true (verbose logging of the logic used by the plugin)

%% - network (as true, but additionally logs LDAP network traffic)

%%

%% Defaults to false.

%%

%% {log, false},

%%

%% Authentication

%% ==============

%%

%% Pattern to convert the username given through AMQP to a DN before

%% binding

%%

%% {user\_dn\_pattern, "cn=${username},ou=People,dc=example,dc=com"},

%% Alternatively, you can convert a username to a Distinguished

%% Name via an LDAP lookup after binding. See the documentation for

%% full details.

%% When converting a username to a dn via a lookup, set these to

%% the name of the attribute that represents the user name, and the

%% base DN for the lookup query.

%%

%% {dn\_lookup\_attribute, "userPrincipalName"},

%% {dn\_lookup\_base, "DC=gopivotal,DC=com"},

%% Controls how to bind for authorisation queries and also to

%% retrieve the details of users logging in without presenting a

%% password (e.g., SASL EXTERNAL).

%% One of

%% - as\_user (to bind as the authenticated user - requires a password)

%% - anon (to bind anonymously)

%% - {UserDN, Password} (to bind with a specified user name and password)

%%

%% Defaults to 'as\_user'.

%%

%% {other\_bind, as\_user},

%%

%% Authorisation

%% =============

%%

%% The LDAP plugin can perform a variety of queries against your

%% LDAP server to determine questions of authorisation. See

%% http://www.rabbitmq.com/ldap.html#authorisation for more

%% information.

%% Set the query to use when determining vhost access

%%

%% {vhost\_access\_query, {in\_group,

%% "ou=${vhost}-users,ou=vhosts,dc=example,dc=com"}},

%% Set the query to use when determining resource (e.g., queue) access

%%

%% {resource\_access\_query, {constant, true}},

%% Set queries to determine which tags a user has

%%

%% {tag\_queries, []}

]}

].

1. 5、Golang调用RabbitMQ的案例

  下载Golgang运行amqp协议的包，在Rabbitmq官网上有提供现在的golang包来使用amqp协议与Rabbitmq交互 。

  我们先将包下载到本地,然后就可以直接使用了：

lion@node1:~$ go get github.com/streadway/amqp

## 5.1、使用Golang来发送第一个hello idoall.org

  在第一个教程中，我们写程序从一个命名的队列(test-idoall-queues)中发送和接收消息。

  producer\_hello.go(消息生产者):

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues"

//Body of message

bodyMsg string = "hello idoall.org"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用发布消息函数

publish(uri, exchangeName, queueName, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, queue string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

// Producer只能发送到exchange，它是不能直接发送到queue的。

// 现在我们使用默认的exchange（名字是空字符）。这个默认的exchange允许我们发送给指定的queue。

// routing\_key就是指定的queue名字。

err = channel.Publish(

exchange, // exchange

q.Name, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_hello(消息消费者).go

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange nam

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, queueName)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func consumer(amqpURI string, exchange string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

true, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

  Console1(运行producer)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_hello.go

2016/07/23 02:29:51 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 02:29:51 got Connection, getting Channel

2016/07/23 02:29:51 got queue, declaring "test-idoall-queues"

2016/07/23 02:29:51 declared queue, publishing 16B body ("hello idoall.org")

2016/07/23 02:29:51 published 16B OK

  然后运行以下命令，可以看到我们刚才创建的queues在列表中

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_queues

Listing queues ...

test-idoall-queues 1

  Console2(运行consumer)打印消息到屏幕，可以看到刚才我们通过producer发送的消息hello idoall.org

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_hello.go

2016/07/23 03:33:14 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 03:33:14 got Connection, getting Channel

2016/07/23 03:33:14 got queue, declaring "test-idoall-queues"

2016/07/23 03:33:14 Queue bound to Exchange, starting Consume

2016/07/23 03:33:14 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 03:33:14 Received a message: hello idoall.org

1. 5.2、Rabbitmq的任务分发机制

  在5.1章节中，我们写程序从一个命名的队列中发送和接收消息。在这个章节中，我们将创建一个工作队列，将用于分配在多个工人之间的耗时的任务。

  RabbitMQ的分发机制非常适合扩展，而且它是专门为并发程序设计的。如果任务队伍过多，那么只需要创建更多的Consumer来进行任务处理即可。

  producer\_task.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-task"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, queueName, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, queue string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

// Producer只能发送到exchange，它是不能直接发送到queue的。

// 现在我们使用默认的exchange（名字是空字符）。这个默认的exchange允许我们发送给指定的queue。

// routing\_key就是指定的queue名字。

err = channel.Publish(

exchange, // exchange

q.Name, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_task（消息消费者）.go

package main

import (

"fmt"

"log"

"bytes"

"time"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange nam

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-task"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, queueName)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func consumer(amqpURI string, exchange string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

dot\_count := bytes.Count(d.Body, []byte("."))

t := time.Duration(dot\_count)

time.Sleep(t \* time.Second)

log.Printf("Done")

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  Console1(consumer)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_task.go

2016/07/23 10:11:40 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:11:40 got Connection, getting Channel

2016/07/23 10:11:40 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:11:40 Queue bound to Exchange, starting Consume

2016/07/23 10:11:40 [\*] Waiting for messages. To exit press CTRL+C

  Console2(consumer)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_task.go

2016/07/23 10:11:40 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:11:40 got Connection, getting Channel

2016/07/23 10:11:40 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:11:40 Queue bound to Exchange, starting Consume

2016/07/23 10:11:40 [\*] Waiting for messages. To exit press CTRL+C

  这个时候我们使用Producer 来 Publish Message：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_task.go First message. && go run producer\_task.go Second message.. && go run producer\_task.go Third message... && go run producer\_task.go Fourth message.... && go run producer\_task.go Fifth message.....

2016/07/23 10:17:13 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:17:13 got Connection, getting Channel

2016/07/23 10:17:13 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:17:13 declared queue, publishing 14B body ("First message.")

2016/07/23 10:17:13 published 14B OK

2016/07/23 10:17:14 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:17:14 got Connection, getting Channel

2016/07/23 10:17:14 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:17:14 declared queue, publishing 16B body ("Second message..")

2016/07/23 10:17:14 published 16B OK

2016/07/23 10:17:15 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:17:15 got Connection, getting Channel

2016/07/23 10:17:15 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:17:15 declared queue, publishing 16B body ("Third message...")

2016/07/23 10:17:15 published 16B OK

2016/07/23 10:17:16 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:17:16 got Connection, getting Channel

2016/07/23 10:17:16 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:17:16 declared queue, publishing 18B body ("Fourth message....")

2016/07/23 10:17:16 published 18B OK

2016/07/23 10:17:16 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:17:16 got Connection, getting Channel

2016/07/23 10:17:16 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:17:16 declared queue, publishing 18B body ("Fifth message.....")

2016/07/23 10:17:16 published 18B OK

  这时我们再看刚才打开的两个Consumer的结果：  
  Console1(consumer)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_task.go

2016/07/23 10:11:21 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:11:21 got Connection, getting Channel

2016/07/23 10:11:21 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:11:21 Queue bound to Exchange, starting Consume

2016/07/23 10:11:21 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 10:17:13 Received a message: First message.

2016/07/23 10:17:14 Done

2016/07/23 10:17:15 Received a message: Third message...

2016/07/23 10:17:18 Done

2016/07/23 10:17:18 Received a message: Fifth message.....

2016/07/23 10:17:23 Done

  Console2(consumer)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_task.go

2016/07/23 10:11:40 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 10:11:40 got Connection, getting Channel

2016/07/23 10:11:40 got queue, declaring "test-idoall-queues-task"

2016/07/23 10:11:40 Queue bound to Exchange, starting Consume

2016/07/23 10:11:40 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 10:17:14 Received a message: Second message..

2016/07/23 10:17:16 Done

2016/07/23 10:17:16 Received a message: Fourth message....

2016/07/23 10:17:20 Done

  默认情况下，RabbitMQ 会顺序的分发每个Message。当每个收到ack后，会将该Message删除，然后将下一个Message分发到下一个Consumer。这种分发方式叫做round-robin，也叫消息轮询

1. 5.3、Message acknowledgment 消息确认

  每个Consumer可能需要一段时间才能处理完收到的数据。如果在这个过程中，Consumer出错了，异常退出了，而数据还没有处理完成，那么非常不幸，这段数据就丢失了。因为我们的代码，一旦RabbitMQ Server发送给Consumer消息后，会立即把这个Message标记为完成，然后从queue中删除。我们将无法再操作这个尚未处理完成的消息。

  实际场景中，如果一个Consumer异常退出了，我们希望它处理的数据能够被另外的Consumer处理，这样数据在这种情况下（通道关闭、连接关闭、TCP连接丢失等情况）就不会丢失了。

  为了保证数据不被丢失，RabbitMQ支持消息确认机制，ack(nowledgments)是从Consumer消费后发送到一个特定的消息告诉RabbitMQ已经收到、处理结束，RabbitMQ可以去安全的删除它了。

  如果Consumer退出了但是没有发送ack，那么RabbitMQ就会把这个Message重新排进队列，发送到下一个Consumer。这样就保证了在Consumer异常退出的情况下数据也不会丢失。

  这里并没有用到超时机制。RabbitMQ仅仅通过Consumer的连接中断来确认该Message并没有被正确处理。也就是说，RabbitMQ给了Consumer足够长的时间来做数据处理。

  消息确认默认是关闭的，我们需要通过，d.ACK(false)来告诉RabbitMQ我们已经完成任务。

  producer\_acknowledgments(消息生产者).go:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-acknowledgments"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, queueName, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, queue string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

// Producer只能发送到exchange，它是不能直接发送到queue的。

// 现在我们使用默认的exchange（名字是空字符）。这个默认的exchange允许我们发送给指定的queue。

// routing\_key就是指定的queue名字。

err = channel.Publish(

exchange, // exchange

q.Name, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_acknowledgments（消息消费者）.go

package main

import (

"fmt"

"log"

"bytes"

"time"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange nam

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-acknowledgments"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, queueName)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func consumer(amqpURI string, exchange string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

dot\_count := bytes.Count(d.Body, []byte("."))

t := time.Duration(dot\_count)

time.Sleep(t \* time.Second)

log.Printf("Done")

d.Ack(false)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  我们先使用Producer来发送一列消息：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_acknowledgments.go First message. && go run producer\_acknowledgments.go Second message.. && go run producer\_acknowledgments.go Third message... && go run producer\_acknowledgments.go Fourth message.... && go run producer\_acknowledgments.go Fifth message.....

2016/07/23 21:41:40 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:41:40 got Connection, getting Channel

2016/07/23 21:41:40 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:41:40 declared queue, publishing 14B body ("First message.")

2016/07/23 21:41:40 published 14B OK

2016/07/23 21:41:41 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:41:41 got Connection, getting Channel

2016/07/23 21:41:41 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:41:41 declared queue, publishing 16B body ("Second message..")

2016/07/23 21:41:41 published 16B OK

2016/07/23 21:41:41 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:41:41 got Connection, getting Channel

2016/07/23 21:41:41 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:41:41 declared queue, publishing 16B body ("Third message...")

2016/07/23 21:41:41 published 16B OK

2016/07/23 21:41:42 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:41:42 got Connection, getting Channel

2016/07/23 21:41:42 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:41:42 declared queue, publishing 18B body ("Fourth message....")

2016/07/23 21:41:42 published 18B OK

2016/07/23 21:41:43 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:41:43 got Connection, getting Channel

2016/07/23 21:41:43 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:41:43 declared queue, publishing 18B body ("Fifth message.....")

2016/07/23 21:41:43 published 18B OK

  通过rabbitmqctl命令，来看下messages\_unacknowledged的情况：

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

Listing queues ...

test-idoall-queues-task 0 0

test-idoall-queues 0 0

test-idoall-queues-acknowledgments 5 0

  使用Consumer来订阅消息操作到第三条的时候，我们按CTRL+C退出，这个时候相当于消息已经被读取，但是未发送d.ACK(false)：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_acknowledgments.go

2016/07/23 21:56:35 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 21:56:35 got Connection, getting Channel

2016/07/23 21:56:35 got queue, declaring "test-idoall-queues-acknowledgments"

2016/07/23 21:56:35 Queue bound to Exchange, starting Consume

2016/07/23 21:56:35 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 21:56:35 Received a message: First message.

2016/07/23 21:56:36 Done

2016/07/23 21:56:36 Received a message: Second message..

2016/07/23 21:56:38 Done

2016/07/23 21:56:38 Received a message: Third message...

^Csignal: interrupt

  再通过rabbitmqctl命令可以看到，还是有3条消息未处理

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

Listing queues ...

test-idoall-queues-task 0 0

test-idoall-queues 0 0

test-idoall-queues-acknowledgments 3 0

1. 5.4、Message durability消息持久化

  如果服务器死机或程序 crash了，数据仍然会丢失。为了确保消息不会丢失，我们需要将queue和Message做持久化操作。

  将durable设置为true可以做持久化处理（生产者和消息者的代码里都要设置），如果是已经存在的一个queue 没有设置过持久化，再重新设置是不起作用的，我们需要重新为queue设置一个名字。

  最后在Producer发布消息的时候，我们需要设置DeliveryMode为amqp.Persistent，持久化的工作就做完了，下面我们来看代码

  producer\_durability.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-durability"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, queueName, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, queue string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

true, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

// Producer只能发送到exchange，它是不能直接发送到queue的。

// 现在我们使用默认的exchange（名字是空字符）。这个默认的exchange允许我们发送给指定的queue。

// routing\_key就是指定的queue名字。

err = channel.Publish(

exchange, // exchange

q.Name, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

DeliveryMode: amqp.Persistent,

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_durability.go（消息接收者）：

package main

import (

"fmt"

"log"

"bytes"

"time"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange nam

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-durability"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, queueName)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func consumer(amqpURI string, exchange string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

true, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

dot\_count := bytes.Count(d.Body, []byte("."))

t := time.Duration(dot\_count)

time.Sleep(t \* time.Second)

log.Printf("Done")

d.Ack(false)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  我们先使用Producer来发送一列消息：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_durability.go First message. && go run producer\_durability.go Second message.. && go run producer\_durability.go Third message... && go run producer\_durability.go Fourth message.... && go run producer\_durability.go Fifth message.....

2016/07/23 22:35:03 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 22:35:03 got Connection, getting Channel

2016/07/23 22:35:03 got queue, declaring "test-idoall-queues-durability"

2016/07/23 22:35:04 declared queue, publishing 14B body ("First message.")

2016/07/23 22:35:04 published 14B OK

2016/07/23 22:35:04 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 22:35:04 got Connection, getting Channel

2016/07/23 22:35:04 got queue, declaring "test-idoall-queues-durability"

2016/07/23 22:35:04 declared queue, publishing 16B body ("Second message..")

2016/07/23 22:35:04 published 16B OK

2016/07/23 22:35:05 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 22:35:05 got Connection, getting Channel

2016/07/23 22:35:05 got queue, declaring "test-idoall-queues-durability"

2016/07/23 22:35:05 declared queue, publishing 16B body ("Third message...")

2016/07/23 22:35:05 published 16B OK

2016/07/23 22:35:06 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 22:35:06 got Connection, getting Channel

2016/07/23 22:35:06 got queue, declaring "test-idoall-queues-durability"

2016/07/23 22:35:06 declared queue, publishing 18B body ("Fourth message....")

2016/07/23 22:35:06 published 18B OK

2016/07/23 22:35:06 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 22:35:06 got Connection, getting Channel

2016/07/23 22:35:06 got queue, declaring "test-idoall-queues-durability"

2016/07/23 22:35:06 declared queue, publishing 18B body ("Fifth message.....")

2016/07/23 22:35:06 published 18B OK

  通过rabbitmqctl list\_queues命令，来看下messages\_unacknowledged的情况：

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_queues

Listing queues ...

test-idoall-queues-task 0

test-idoall-queues 0

test-idoall-queues-durability 5

test-idoall-queues-acknowledgments 0

  重启RabbitMQ-Server

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl stop

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmq-server

RabbitMQ 3.6.3. Copyright (C) 2007-2016 Pivotal Software, Inc.

## ## Licensed under the MPL. See http://www.rabbitmq.com/

## ##

########## Logs: /home/lion/\_app/rabbitmq\_server-3.6.3/var/log/rabbitmq/rabbit@node1.log

###### ## /home/lion/\_app/rabbitmq\_server-3.6.3/var/log/rabbitmq/rabbit@node1-sasl.log

##########

Starting broker...

completed with 6 plugins.

  再次通过rabbitmqctl list\_queues命令查看，可以看到消息是存在的，说明我们的持久化是成功的

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_queues

Listing queues ...

test-idoall-queues-durability 5

1. 5.5、Fair dispatch 公平分发

  上面的，分发机制不是那么优雅。默认状态下，RabbitMQ将第n个Message分发给第n个Consumer。当然n是取余后的。它不管Consumer是否还有unacked Message，只是按照这个默认机制进行分发。

  那么如果有个Consumer工作比较重，那么就会导致有的Consumer基本没事可做，有的Consumer却是毫无休息的机会。

  通过 ch.Qos 方法设置预读取消息prefetch count=1 。这样RabbitMQ就会使得每个Consumer在同一个时间点最多处理一个Message。换句话说，在接收到该Consumer的ack前，他它不会将新的Message分发给它。

  producer\_fair\_dispatch.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-fair\_dispatch"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, queueName, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, queue string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

true, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

// Producer只能发送到exchange，它是不能直接发送到queue的。

// 现在我们使用默认的exchange（名字是空字符）。这个默认的exchange允许我们发送给指定的queue。

// routing\_key就是指定的queue名字。

err = channel.Publish(

exchange, // exchange

q.Name, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

DeliveryMode: amqp.Persistent,

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_fair\_dispatch.go（消息消费者）：

package main

import (

"fmt"

"log"

"bytes"

"time"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange nam

exchangeName = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-fair\_dispatch"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, queueName)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func consumer(amqpURI string, exchange string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

log.Printf("got queue, declaring %q", queue)

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

true, // durable

false, // delete when unused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

//每次只取一条消息

err = channel.Qos(

1, // prefetch count

0, // prefetch size

false, // global

)

failOnError(err, "Failed to set QoS")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

dot\_count := bytes.Count(d.Body, []byte("."))

t := time.Duration(dot\_count)

time.Sleep(t \* time.Second)

log.Printf("Done")

d.Ack(false)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  我们先使用Producer来发送一列消息：

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_fair\_dispatch.go First message. && go run producer\_fair\_dispatch.go Second message.. && go run producer\_fair\_dispatch.go Third message... && go run producer\_fair\_dispatch.go Fourth message.... && go run producer\_fair\_dispatch.go Fifth message.....

2016/07/23 23:09:24 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:09:24 got Connection, getting Channel

2016/07/23 23:09:24 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:09:24 declared queue, publishing 14B body ("First message.")

2016/07/23 23:09:24 published 14B OK

2016/07/23 23:09:24 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:09:24 got Connection, getting Channel

2016/07/23 23:09:24 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:09:24 declared queue, publishing 16B body ("Second message..")

2016/07/23 23:09:24 published 16B OK

2016/07/23 23:09:25 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:09:25 got Connection, getting Channel

2016/07/23 23:09:25 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:09:25 declared queue, publishing 16B body ("Third message...")

2016/07/23 23:09:25 published 16B OK

2016/07/23 23:09:26 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:09:26 got Connection, getting Channel

2016/07/23 23:09:26 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:09:26 declared queue, publishing 18B body ("Fourth message....")

2016/07/23 23:09:26 published 18B OK

2016/07/23 23:09:27 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:09:27 got Connection, getting Channel

2016/07/23 23:09:27 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:09:27 declared queue, publishing 18B body ("Fifth message.....")

2016/07/23 23:09:27 published 18B OK

  再依次在两个Console中依次执行下面的命令，可以看到消息被正常的分发了

  Console1（consumer）:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_fair\_dispatch.go

2016/07/23 23:10:47 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:10:47 got Connection, getting Channel

2016/07/23 23:10:47 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:10:47 Queue bound to Exchange, starting Consume

2016/07/23 23:10:47 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 23:10:47 Received a message: First message.

2016/07/23 23:10:48 Done

2016/07/23 23:10:48 Received a message: Second message..

2016/07/23 23:10:50 Done

2016/07/23 23:10:50 Received a message: Fourth message....

2016/07/23 23:10:54 Done

  Console2（consumer）:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_fair\_dispatch.go

2016/07/23 23:10:49 dialing "amqp://guest:guest@localhost:5672/"

2016/07/23 23:10:49 got Connection, getting Channel

2016/07/23 23:10:49 got queue, declaring "test-idoall-queues-fair\_dispatch"

2016/07/23 23:10:49 Queue bound to Exchange, starting Consume

2016/07/23 23:10:49 [\*] Waiting for messages. To exit press CTRL+C

2016/07/23 23:10:49 Received a message: Third message...

2016/07/23 23:10:52 Done

2016/07/23 23:10:52 Received a message: Fifth message.....

2016/07/23 23:10:57 Done

​ 基于AMQP的更多通道和消息属性，可以浏览[AMQP API](http://godoc.org/github.com/streadway/amqp)参考

1. 5.6、Exchanges & Bindings

  RabbitMQ 的Messaging Model就是Producer并不会直接发送Message到queue。实际上，Producer并不知道它发送的Message是否已经到达queue。

  Producer发送的Message实际上是发到了Exchange中。它的功能也很简单：从Producer接收Message，然后投递到queue中。Exchange需要知道如何处理Message，是把它放到一个queue中，还是放到多个queue中？这个rule是通过Exchange 的类型定义的。

  我们知道有三种类型的Exchange：direct,，topic，headers 和fanout。fanout就是广播模式，会将所有的Message都放到它所知道的queue中。

  现在我们已经创建了fanout类型的exchange和没有名字的queue（实际上是RabbitMQ帮我们取了名字）。那exchange怎么样知道它的Message发送到哪个queue呢？答案就是通过bindings

  通过rabbitmqctl可以列出当前所有的Exchange：

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_exchanges

Listing exchanges ...

amq.direct direct

amq.fanout fanout

amq.match headers

amq.headers headers

direct

amq.rabbitmq.trace topic

amq.topic topic

amq.rabbitmq.log topic

*注意：amq.\* 是RabbitMQ默认创建的。*

​ 我们假设做一个日志系统，其中一个运行的接收程序Consumer发到消息后写入到磁盘中，同时， 另一个Consumer将收到的日志输出到屏幕上。

  producer\_exchange\_logs.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "fanout"

//AMQP routing key

routingKey = ""

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, exchangeType, routingKey, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@routingKey, routingKey的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, exchangeType string, routingKey string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个queue

log.Printf("got Channel, declaring %q Exchange (%q)", exchangeType, exchange)

err = channel.ExchangeDeclare(

exchange, // name

exchangeType, // type

true, // durable

false, // auto-deleted

false, // internal

false, // noWait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

// 发布消息

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

err = channel.Publish(

exchange, // exchange

routingKey, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_exchange\_logs.go（消息消费者）：

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "fanout"

//AMQP binding key

bindingKey = ""

//Durable AMQP queue name

queueName = ""

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, exchangeType, queueName, bindingKey)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@queue, queue的名称

//@key , 绑定的key名称

func consumer(amqpURI string, exchange string, exchangeType string, queue string, key string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个exchange

log.Printf("got Channel, declaring Exchange (%q)", exchange)

err = channel.ExchangeDeclare(

exchange, // name of the exchange

exchangeType, // type

true, // durable

false, // delete when complete

false, // internal

false, // noWait

nil, // arguments

);

failOnError(err, "Exchange Declare:")

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

true, // exclusive 当Consumer关闭连接时，这个queue要被deleted

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

//绑定到exchange

err = channel.QueueBind(

q.Name, // name of the queue

key, // bindingKey

exchange, // sourceExchange

false, // noWait

nil, // arguments

);

failOnError(err, "Failed to bind a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf(" [x] %s", d.Body)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

  在AMQP客户端 ，当routing key为空的时候， 自动创建一个随机的queue，同时设置exclusive为true时，当这个Consumer关闭链接 时，会删除这个queue。

  当使用fanout类型的exchange和没有名字的queue，Cusomer并不知道消息发送到了哪个queue,这个时候我们就需要用到QueueBind方法，来绑定到exchange。

*过程中可以使用rabbitmqctl list\_bindings命令来查看绑定的列表*

**查看结果**

  Console1(Consumer)，输出到文件:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_logs.go &> consumer\_exchange\_logs.log

  Console2(Consumer)，打印到控制台:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_logs.go

  使用Producer来发送消息:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_exchange\_logs.go

2016/07/24 02:21:49 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 02:21:49 got Connection, getting Channel

2016/07/24 02:21:49 got Channel, declaring "fanout" Exchange ("test-idoall-exchange-logs")

2016/07/24 02:21:49 declared queue, publishing 16B body ("hello idoall.org")

2016/07/24 02:21:49 published 16B OK

  这时可以使用rabbitmqctl list\_bindings来查看我们的绑定信息，可以看到queueu的名字是随机的

lion@node1:~/\_code/\_rabbitmq/\_golang$ rabbitmqctl list\_bindings

Listing bindings ...

exchange amq.gen-D2AnzGsLUMhJCPk7YxgUUw queue amq.gen-D2AnzGsLUMhJCPk7YxgUUw []

exchange amq.gen-GC4VDS3mxsAOTEqii\_WsWw queue amq.gen-GC4VDS3mxsAOTEqii\_WsWw []

test-idoall-exchange-logs exchange amq.gen-D2AnzGsLUMhJCPk7YxgUUw queue []

test-idoall-exchange-logs exchange amq.gen-GC4VDS3mxsAOTEqii\_WsWw queue []

  使用cat命令，查看consumer\_exchange\_logs.log文件，可以看到内容被输入到文件中

lion@node1:~/\_code/\_rabbitmq/\_golang$ cat consumer\_exchange\_logs.log

2016/07/24 02:25:17 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 02:25:17 got Connection, getting Channel

2016/07/24 02:25:17 got Channel, declaring Exchange ("test-idoall-exchange-logs")

2016/07/24 02:25:17 Queue bound to Exchange, starting Consume

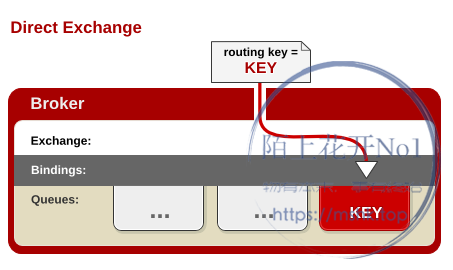
2016/07/24 02:25:17 [\*] Waiting for messages. To exit press CTRL+C

signal: interrupt

1. 5.7、Direct exchange

  RabbitMQ支持同一个binding key绑定到多个queue中。Direct exchange的算法就是通过binding key来做匹配的。

*对于fanout的exchange来说，routing\_key这个参数是被忽略的。*



  producer\_exchange\_direct\_logs.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-direct-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "direct"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, exchangeType, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 3) || os.Args[2] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[2:], " ")

}

return s

}

func severityFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "info"

} else {

s = os.Args[1]

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@body, 主体内容

func publish(amqpURI string, exchange string, exchangeType string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个queue

log.Printf("got Channel, declaring %q Exchange (%q)", exchangeType, exchange)

err = channel.ExchangeDeclare(

exchange, // name

exchangeType, // type

true, // durable

false, // auto-deleted

false, // internal

false, // noWait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

// 发布消息

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

err = channel.Publish(

exchange, // exchange

severityFrom(os.Args), // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_exchange\_direct\_logs.go（消息消费者）：

package main

import (

"fmt"

"log"

"os"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-direct-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "direct"

//AMQP binding key

bindingKey = ""

//Durable AMQP queue name

queueName = ""

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, exchangeType, queueName, bindingKey)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@queue, queue的名称

//@key , 绑定的key名称

func consumer(amqpURI string, exchange string, exchangeType string, queue string, key string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个exchange

log.Printf("got Channel, declaring Exchange (%q)", exchange)

err = channel.ExchangeDeclare(

exchange, // name of the exchange

exchangeType, // type

true, // durable

false, // delete when complete

false, // internal

false, // noWait

nil, // arguments

);

failOnError(err, "Exchange Declare:")

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

true, // exclusive 当Consumer关闭连接时，这个queue要被deleted

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

if len(os.Args) < 2 {

log.Printf("Usage: %s [info] [warning] [error]", os.Args[0])

os.Exit(0)

}

for \_, s := range os.Args[1:] {

log.Printf("Binding queue %s to exchange %s with routing key %s",

q.Name, exchange, s)

//绑定到exchange

err = channel.QueueBind(

q.Name, // name of the queue

s, // bindingKey

exchange, // sourceExchange

false, // noWait

nil, // arguments

);

failOnError(err, "Failed to bind a queue")

}

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf(" [x] %s", d.Body)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  Console1(Consumer)，输出到文件:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_direct\_logs.go warning error &> consumer\_exchange\_direct\_logs.log

  Console2(Consumer)，打印到控制台:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_direct\_logs.go info warning error

2016/07/24 08:48:17 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 08:48:17 got Connection, getting Channel

2016/07/24 08:48:17 got Channel, declaring Exchange ("test-idoall-exchange-direct-logs")

2016/07/24 08:48:17 Binding queue amq.gen-vE-62-Lwt4VQYjlBbMLTjQ to exchange test-idoall-exchange-direct-logs with routing key info

2016/07/24 08:48:17 Binding queue amq.gen-vE-62-Lwt4VQYjlBbMLTjQ to exchange test-idoall-exchange-direct-logs with routing key warning

2016/07/24 08:48:17 Binding queue amq.gen-vE-62-Lwt4VQYjlBbMLTjQ to exchange test-idoall-exchange-direct-logs with routing key error

2016/07/24 08:48:17 Queue bound to Exchange, starting Consume

2016/07/24 08:48:17 [\*] Waiting for messages. To exit press CTRL+C

  使用Producer来发送消息:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_exchange\_direct\_logs.go error "Error. Error" && go run producer\_exchange\_direct\_logs.go info "Info. Info" && go run producer\_exchange\_direct\_logs.go warning "warning. warning"

  我们可以看到，在Console2控制台上能够看到error、info、waring的所有消息，而在文件中只能看到和error相关的消息。

1. 5.7、Topic exchange

  对于Topic的exchange中Message的routing\_key是有限制的，不能太随意。格式是以点号“.”分割的字符表。比如：”stock.usd.nyse”, “nyse.vmw”, “quick.orange.rabbit”。你可以放任意的key在routing\_key中，不过长度不能超过255 bytes。

  对于routing\_key，有两个特殊字符（在正则表达式里叫元字符）

* \* (星号) 代表任意 一个单词
* # (hash哈希) 0个或者多个单词

  Topic exchange和其他exchange的区别,由于有”\*”和”#”， Topic exchange 非常强大并且可以转化为其他的exchange:

* 如果binding\_key 是 “#” – 它会接收所有的Message，不管routing\_key是什么，就像是fanout exchange。
* 如果 “\*”和”#”没有被使用，那么topic exchange就变成了direct exchange。

  下面的代码中，我们将演示Topic的exchange使用”#”和”\*”来匹配binding key。

  producer\_exchange\_topic\_logs.go（消息生产者）:

package main

import (

"fmt"

"log"

"os"

"strings"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-direct-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "fanout"

//AMQP routing key

routingKey = ""

//Durable AMQP queue name

queueName = "test-idoall-queues-direct"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

bodyMsg := bodyFrom(os.Args)

//调用发布消息函数

publish(uri, exchangeName, exchangeType, routingKey, bodyMsg)

log.Printf("published %dB OK", len(bodyMsg))

}

func bodyFrom(args []string) string {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "hello idoall.org"

} else {

s = strings.Join(args[1:], " ")

}

return s

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@routingKey, routingKey的名称

//@body, 主体内容

func publish(amqpURI string, exchange string, exchangeType string, routingKey string, body string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个queue

log.Printf("got Channel, declaring %q Exchange (%q)", exchangeType, exchange)

err = channel.ExchangeDeclare(

exchange, // name

exchangeType, // type

true, // durable

false, // auto-deleted

false, // internal

false, // noWait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

// 发布消息

log.Printf("declared queue, publishing %dB body (%q)", len(body), body)

err = channel.Publish(

exchange, // exchange

routingKey, // routing key

false, // mandatory

false, // immediate

amqp.Publishing {

Headers: amqp.Table{},

ContentType: "text/plain",

ContentEncoding: "",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

}

  consumer\_exchange\_topic\_logs.go（消息消费者）：

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = "test-idoall-exchange-topic-logs"

//Exchange type - direct|fanout|topic|x-custom

exchangeType = "topic"

//AMQP binding key

bindingKey = ""

//Durable AMQP queue name

queueName = ""

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用消息接收者

consumer(uri, exchangeName, exchangeType, queueName, bindingKey)

}

//接收者方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@exchangeType, exchangeType的类型direct|fanout|topic

//@queue, queue的名称

//@key , 绑定的key名称

func consumer(amqpURI string, exchange string, exchangeType string, queue string, key string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个exchange

log.Printf("got Channel, declaring Exchange (%q)", exchange)

err = channel.ExchangeDeclare(

exchange, // name of the exchange

exchangeType, // type

true, // durable

false, // delete when complete

false, // internal

false, // noWait

nil, // arguments

);

failOnError(err, "Exchange Declare:")

//创建一个queue

q, err := channel.QueueDeclare(

queueName, // name

false, // durable

false, // delete when unused

true, // exclusive 当Consumer关闭连接时，这个queue要被deleted

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

//绑定到exchange

err = channel.QueueBind(

q.Name, // name of the queue

key, // bindingKey

exchange, // sourceExchange

false, // noWait

nil, // arguments

);

failOnError(err, "Failed to bind a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

//创建一个channel

forever := make(chan bool)

//调用gorountine

go func() {

for d := range msgs {

log.Printf(" [x] %s", d.Body)

}

}()

log.Printf(" [\*] Waiting for messages. To exit press CTRL+C")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

**查看结果**

  Console1(Consumer)，接收所有的日志:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_topic\_logs.go "#"

2016/07/24 09:28:29 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 09:28:29 got Connection, getting Channel

2016/07/24 09:28:29 got Channel, declaring Exchange ("test-idoall-exchange-topic-logs")

2016/07/24 09:28:29 Binding queue amq.gen-jW2-PIBg4izXpt96CynyFw to exchange test-idoall-exchange-topic-logs with routing key #

2016/07/24 09:28:29 Queue bound to Exchange, starting Consume

2016/07/24 09:28:29 [\*] Waiting for messages. To exit press CTRL+C

  Console2(Consumer)，接收以”kern”开头的日志:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_topic\_logs.go "kern.\*"

2016/07/24 09:34:00 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 09:34:00 got Connection, getting Channel

2016/07/24 09:34:00 got Channel, declaring Exchange ("test-idoall-exchange-topic-logs")

2016/07/24 09:34:00 Binding queue amq.gen-8zYBz2uXYbWXcItJMZ3AQA to exchange test-idoall-exchange-topic-logs with routing key kern.\*

2016/07/24 09:34:00 Queue bound to Exchange, starting Consume

2016/07/24 09:34:00 [\*] Waiting for messages. To exit press CTRL+C

  Console3(Consumer)，接收第二个单词以”critical”结尾的日志:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_topic\_logs.go "\*.critical"

2016/07/24 09:37:21 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 09:37:21 got Connection, getting Channel

2016/07/24 09:37:21 got Channel, declaring Exchange ("test-idoall-exchange-topic-logs")

2016/07/24 09:37:21 Binding queue amq.gen-tq9QsD1i1mCps-jrqDtTTA to exchange test-idoall-exchange-topic-logs with routing key \*.critical

2016/07/24 09:37:21 Queue bound to Exchange, starting Consume

2016/07/24 09:37:21 [\*] Waiting for messages. To exit press CTRL+C

  Console4(Consumer)， 可以创建多个绑定关系:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run consumer\_exchange\_topic\_logs.go "kern.critical" "A critical kernel error"

2016/07/24 09:39:35 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 09:39:35 got Connection, getting Channel

2016/07/24 09:39:35 got Channel, declaring Exchange ("test-idoall-exchange-topic-logs")

2016/07/24 09:39:35 Binding queue amq.gen-vcaHyCor5bbB2NX7YQhmzA to exchange test-idoall-exchange-topic-logs with routing key kern.critical

2016/07/24 09:39:35 Binding queue amq.gen-vcaHyCor5bbB2NX7YQhmzA to exchange test-idoall-exchange-topic-logs with routing key A critical kernel error

2016/07/24 09:39:35 Queue bound to Exchange, starting Consume

2016/07/24 09:39:35 [\*] Waiting for messages. To exit press CTRL+C

  使用Producer来发送消息:

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run producer\_exchange\_topic\_logs.go "kern.critical" "A critical kernel error"

2016/07/24 09:56:33 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 09:56:33 got Connection, getting Channel

2016/07/24 09:56:33 got Channel, declaring "topic" Exchange ("test-idoall-exchange-topic-logs")

2016/07/24 09:56:33 declared queue, publishing 23B body ("A critical kernel error")

2016/07/24 09:56:33 [x] Sent A critical kernel error

2016/07/24 09:56:33 published 23B OK

1. 5.7、远程调用RPC

  之前的实例都是通过一个或多个Consumer来订阅消息，如果我们需要在远程机器上运行一个函数，来等待结果呢？这是一个不同的场景，例如做云计算。

  AMQP协议预定义了14个属性，大多数我们都很少用到，以下几个是比较常用的。

* persistent：消息持久性
* content\_type：用来描述编码的MIME类型
* reply\_to：回调queue的名字
* correlation\_id：将远程RPC请求，进行关联的唯一标识

***correlation\_id***

*如果为每个RPC的请求创建一个queue效率是非常低的，正常发送到queue的一个Message，它不知道是从哪里发过来的，而correlation\_id属性的存在就是为每个请求设置一个唯一值，在回调接收消息的时候，也会带回这个属性进行匹配，如果不匹配，这个消息就不会被处理。*

  接下来我们将使用RabbitMQ搭建一个RPC系统：一个客户端和一个可扩展的RPC服务器，RPC的工作流程如下：

* 客户端启动时，创建一个匿名的exclusive callback queue
* 客户端发送请求时，要带两个属性reply\_to（设置回调的queue）和correlation\_id(唯一标识)
* 将请求发送到一个RPC queue
* RPC的server端 ，一直在等待请求，当消息到达时会对过reply\_to回复到指定的queue
* 客户端在等queue从server的回调，检查 correlation\_id是否一致，如果和请求时发送的一致，则做其他响应。

  rpc\_server.go（服务端代码）:

package main

import (

"fmt"

"log"

"strconv"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP queue name

queueName = "rpc-queue"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func main(){

//调用发布消息函数

publish(uri, queueName)

}

//发布者的方法

//

//@amqpURI, amqp的地址

//@queue, queue的名称

func publish(amqpURI string, queue string){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个queue

log.Printf("got queue, declaring %q", queue)

q,err := channel.QueueDeclare(

queue, // name

false, // durable

false, // delete when usused

false, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

//均衡处理，每次处理一条消息

err = channel.Qos(

1, // prefetch count

0, // prefetch size

false, // global

)

failOnError(err, "Failed to set QoS")

//订阅一个消息

//log.Printf("Queue bound to Exchange, starting Consume")

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

false, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

forever := make(chan bool)

// 发布消息

go func() {

for d := range msgs {

n, err := strconv.Atoi(string(d.Body))

failOnError(err, "Failed to convert body to integer")

log.Printf(" [.] server端接收到的数据是 (%d)", n)

response := n\*2

err = channel.Publish(

"", // exchange

d.ReplyTo, // routing key

false, // mandatory

false, // immediate

amqp.Publishing{

ContentType: "text/plain",

CorrelationId: d.CorrelationId,

Body: []byte(strconv.Itoa(response)),

})

failOnError(err, "Failed to publish a message")

d.Ack(false)

}

}()

log.Printf(" [\*] Awaiting RPC requests")

//没有写入数据，一直等待读，阻塞当前线程，目的是让线程不退出

<-forever

}

  consumer\_exchange\_topic\_logs.go（消息消费者）：

package main

import (

"fmt"

"log"

"math/rand"

"os"

"strconv"

"strings"

"time"

"github.com/streadway/amqp"

)

const (

//AMQP URI

uri = "amqp://guest:guest@localhost:5672/"

//Durable AMQP exchange name

exchangeName = ""

//Exchange type - direct|fanout|topic|x-custom

queueName = "rpc-queue"

)

//如果存在错误，则输出

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

panic(fmt.Sprintf("%s: %s", msg, err))

}

}

func randomString(l int) string {

bytes := make([]byte, l)

for i := 0; i < l; i++ {

bytes[i] = byte(randInt(65, 90))

}

return string(bytes)

}

func randInt(min int, max int) int {

return min + rand.Intn(max-min)

}

func bodyFrom(args []string) int {

var s string

if (len(args) < 2) || os.Args[1] == "" {

s = "30"

} else {

s = strings.Join(args[1:], " ")

}

n, err := strconv.Atoi(s)

failOnError(err, "Failed to convert arg to integer")

return n

}

func main(){

rand.Seed(time.Now().UTC().UnixNano())

n := bodyFrom(os.Args)

log.Printf(" [x] 请求的数据是(%d)", n)

res, err := fibonacciRPC(n, uri, exchangeName, queueName)

failOnError(err, "Failed to handle RPC request")

log.Printf(" [.] 计算结果为 %d", res)

}

//RPC client调用方法

//

//@amqpURI, amqp的地址

//@exchange, exchange的名称

//@queue, queue的名称

func fibonacciRPC(n int, amqpURI string, exchange string, queue string) (res int, err error){

//建立连接

log.Printf("dialing %q", amqpURI)

connection, err := amqp.Dial(amqpURI)

failOnError(err, "Failed to connect to RabbitMQ")

defer connection.Close()

//创建一个Channel

log.Printf("got Connection, getting Channel")

channel, err := connection.Channel()

failOnError(err, "Failed to open a channel")

defer channel.Close()

//创建一个queue

log.Printf("got queue, declaring %q", queue)

q,err := channel.QueueDeclare(

"", // name

false, // durable

false, // delete when usused

true, // exclusive

false, // no-wait

nil, // arguments

)

failOnError(err, "Failed to declare a queue")

log.Printf("Queue bound to Exchange, starting Consume")

//订阅消息

msgs, err := channel.Consume(

q.Name, // queue

"", // consumer

true, // auto-ack

false, // exclusive

false, // no-local

false, // no-wait

nil, // args

)

failOnError(err, "Failed to register a consumer")

corrId := randomString(32)

err = channel.Publish(

"", // exchange

queue, // routing key

false, // mandatory

false, // immediate

amqp.Publishing{

ContentType: "text/plain",

CorrelationId: corrId,

ReplyTo: q.Name,

Body: []byte(strconv.Itoa(n)),

})

failOnError(err, "Failed to publish a message")

for d := range msgs {

if corrId == d.CorrelationId {

res, err = strconv.Atoi(string(d.Body))

failOnError(err, "Failed to convert body to integer")

break

}

}

return

}

**查看结果**

  Console1(rpc server):

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run rpc\_server.go

2016/07/24 11:20:32 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 11:20:32 got Connection, getting Channel

2016/07/24 11:20:32 got queue, declaring "rpc-queue"

2016/07/24 11:20:32 [\*] Awaiting RPC requests

  Console2(rpc client):

lion@node1:~/\_code/\_rabbitmq/\_golang$ go run rpc\_client.go 69

2016/07/24 11:24:37 [x] 请求的数据是(69)

2016/07/24 11:24:37 dialing "amqp://guest:guest@localhost:5672/"

2016/07/24 11:24:37 got Connection, getting Channel

2016/07/24 11:24:37 got queue, declaring "rpc-queue"

2016/07/24 11:24:37 Queue bound to Exchange, starting Consume

2016/07/24 11:24:37 [.] 计算结果为 138

  以上只是简单实现了RPC的功能，如果你有复杂的需求，需要根据需求对Server和Client做调整。

1. 6、写在后面

  业界对于消息传输有很多种方案，之前我们也介绍过[Kafka](https://mshk.top/2014/08/kafka/)，[Kafka](https://mshk.top/2014/08/kafka/)是Linkedin于2010年12月份开源的消息发布订阅系统，它主要用于处理活跃的流式数据，大数据量的数据处理上。RabbitMQ在吞吐量方面稍逊于kafka，他们的出发点不一样，RabbitMQ支持对消息的可靠的传递，支持事务，不支持批量的操作。

  RabbitMQ的消息应当尽可能的小，并且只用来处理实时且要高可靠性的消息。消费者和生产者的能力尽量对等，否则消息堆积会严重影响RabbitMQ的性能。

  本文实例代码在这里可以下载：[点击下载](https://img.mshk.top/_golang.zip)

1. 7、参考资料

<http://www.rabbitmq.com/getstarted.html>

<https://github.com/streadway/amqp>

1. 8、FAQ

## 安装Erlang过程中出现提示configure: error: No curses library functions found

因为缺少缺少ncurses安装包，执行以下命令，即可解决：

lion@node1:~/$ sudo apt-get install libncurses5-dev