链滴

基于 go 语言的 RabbitMQ 教程

作者：Gakkiyomi2019

原文链接：https://ld246.com/article/1603180979798

来源网站：链滴

概述

不管在开发还是生活中，遇到不同的问题都要用不同的方法来解决。例如我们进行应用程序开发时常需要面对的三个问题：

- 解耦：将一个大的业务拆成多个模块，当主业务完成后，发送多个mq消息给其他模块，消费完成即可实现业务，降低了业务的耦合性。
- 异步： 主业务执行结束发送消息通知从属业务通过MQ异步执行，明显降低响应时间，提高用户体。
- 削峰： 高并发情况下，只允许一部分请求进入消息队列进行业务消费，将绝大部分的请求拦截在业外面避免系统业务瘫痪。

这三个问题我们通通都可以使用消息队列来进行解决。

顾名思义，消息队列是一种队列(Go实现队列)，不过里面存的是用来交互的消息。

JMS VS AMQP

java程序员可能都知道activeMQ,这是基于JMS实现的一个消息队列。那什么是JMS呢？通常来说JM（Java MessageService）实际上是指JMS API。是Sun公司早期提出的一个消息标准，目的是为jav应用提供统一的消息操作，包括create、send、receive。

但我们今天不会深入了解JMS，我们只需要知道JMS和AMQP的不同即可。

	JMS	AMQP
定义	Java api	Wire-protocol
跨语言	否	是
跨平台	否	是

Model 提供两种消息模型：
（1）、Peer-2-Peer
（）、Pub/sub 提供了五种消息模型：
（1）、direct exchange
（2）、fanout exchange
（3）、topic change
（4）、headers exchange
（5）、system exchange本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由制上做了更详细的划分；

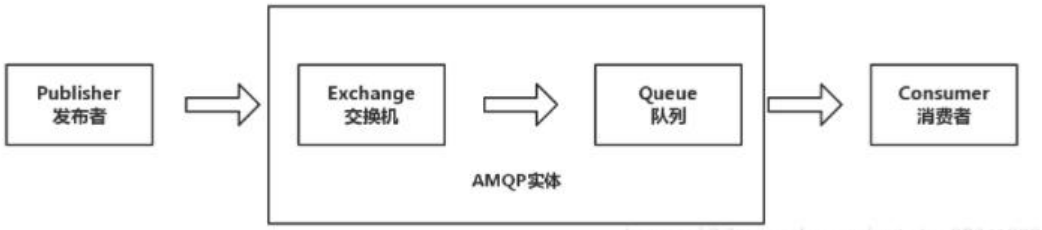
支持消息类型 多种消息类型：TextMessage
MapMessage
BytesMessage
StreamMessage
ObjectMessageMessage （只有消息头和属性）yte[]当实际应用时，有复杂的消息，
可以将消息序列化后发送。

综合评价 JMS 定义了JAVA API层面的标准；在java体系中，
个client均可以通过JMS进行交互，不需要应用修改代码，
但是其对跨平台的支持较差；MQP定义了wire-level层的协议标准；
天然具有跨平台、跨语言特性。

AMQP

AMQP 是一种基于TCP的应用层协议，更准确的说是一种binary wire-level protocol（链接协议）这是其和JMS的本质差别，AMQP不从API层进行限定，而是直接定义网络交换的数据格式。这使得现了AMQP的provider天然性就是跨平台的。而其中 **RabbitMQ** 就是最知名的实现AMQP协议的消队列之一。

工作流程：



1．发布者（publisher）发布消息（Message），经过交换机（Exchange）。

2．交换机根据路由规则将收到的消息分发给与该交换机绑定的队列（Queue）。

3．最后 AMQp 代理会将消息投递给订阅了此队列的消费者，或者消费者按照需求自行获取。

注意：

1.发布者、交换机、队列、消费者都可以有多个．同时因为 AMQp 是一个网络协议，所以这个过程的发布者，消费者，消息代理可以分别存在于不同的设备上。

2．发布者发布消息可以给消息指定各种消息属性（Message Meta -data）。有些属性有可能会被息代理（Brokers) 使用，然而其他的属性则是完全不透明的，它们只能被接收消息的应用所使用。

3．从安全角度考虑，网络是不可靠的，又或是消费者在处理消息的过程中意外的挂掉，这样没有处成功的消息就会丢失。基于此原因， AMQP 模块包含了一个消息确认（Message Acknowledgements）机制：当一个消息从队列中投递给消费者后，不会立即从队列中删除，直到它收到来自消费者的认回执（Acknowledgment）后，才完全从队列中删除。

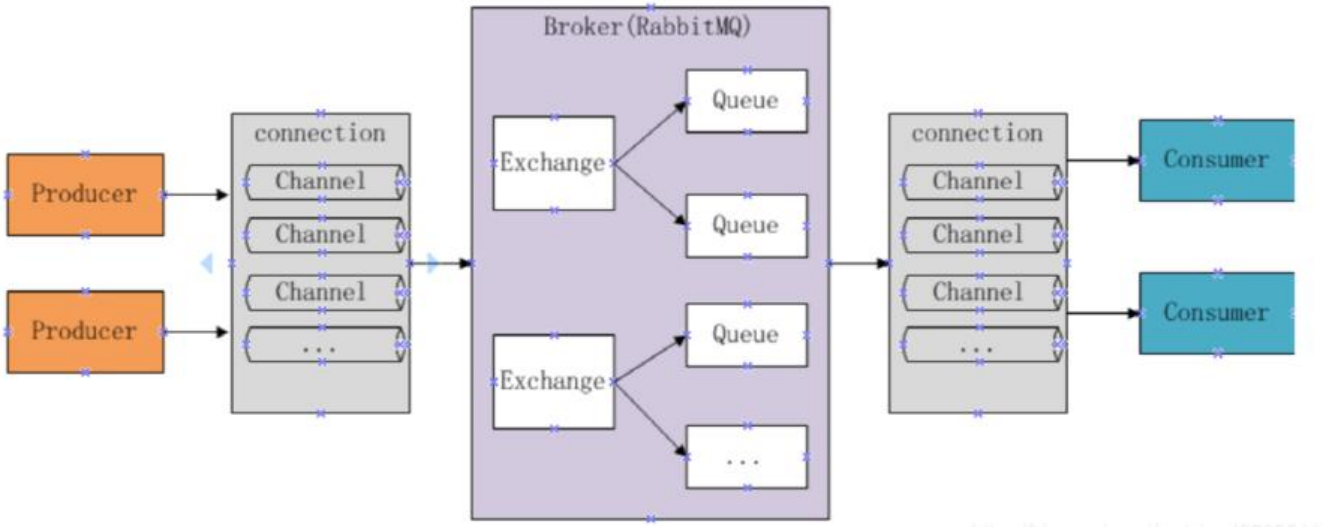
4．在某些情况下，例如当一个消息无法被成功路由时（无去从交换机分发到队列），消息或许会被回给发布者并被丢弃。或者，如果消息代理执行了延期操作，消息会被放入一个所谓的死信队列中．时，消息发布者可选择某些参数来处理这些特殊情况。

RabbitMQ

RabbitMQ是一个由Rabbit开发的基于AMQP的开源消息队列，能够实现异步消息处理，使用erlang言开发。

消息发送过程

ConnectionFactory、Connection、Channel这三个时RabbitMQ对外封装的重要对象。Connection装了socket协议相关逻辑。ConnectionFactory是Connection的制造工厂，Channel类似于Java NIO的Channel 做选择 TCP 连接复用，不仅可以减少性能开销，同时也便于管理。



组件

队列（Queue）

队列是用来存储消息的，RabbitMQ的消息只能存在队列里面，生产者产生消息发送至交换机再然后过一系列的路由规则最终投递到队列，消费者可以从队列中获取消息。

为了处理消费者没有处理完消息就宕机的情况，我们可以要求消费者消费完消息之后需要发送一个回给rabbitmq,rabbitmq收到回执之后才会将这条消息从队列里面移除。

如果希望rabbitmq宕机时不会丢失消息，我们可以将queue和message都设置为可持久化(durable)。

声明

- name：队列名称
- durable：消息是否持久化
- auto-deleted：队列接受完消息是否自动删除
- exclusive：是否为独占队列 (独占队列只能由声明他们的连接访问，并在连接关闭时删除)
- no-wait：如果为true,队列默认认为已存在交换机Exchange,连接不上会报错
- argument: 队列的其他选项

```
ch.QueueDeclare(
```

原文链接：基于 go 语言的 RabbitMQ 教程

```
    r.QueueName,
    false,  //是否持久化
    false,  //是否为自动删除
    false,  //是否具有排他性
    false,  //是否阻塞
    nil,  //额外属性
)
```

消费

- name: 队列名称
- consumer: 消费者名称
- autoAck: 自动应答
- exclusive: 排他性
- noLocal: 不允许本地消费(使用同一个connection)
- nowait: 是否阻塞
- args: 其他参数

```
ch.Consume(
    q.Name,
    "",   //用来区分多个消费者
    true, //是否自动应答,告诉我已经消费完了
    false,
    false, //若设置为true,则表示为不能将同一个connection中发送的消息传递给这个connection中消费者.
    false, //消费队列是否设计阻塞
    nil,
)
```

交换机(Exchange)

生产者会将消息发送到Exchange, 由Exchange将消息路由至一个或者多个队列(或者丢弃)。生产者将息发送给Exchange时会指定routing key 来指定该消息的路由规则。

声明

- name: 交换机名称
- type: 交换机类型
- durable: 是否持久化
- auto-deleted: 当关联的队列都删除之后自动删除
- internal: 是否为rabbitmq内部使用
- no-wait: 如果为false,则不期望Rabbitmq服务器有一个 Exchange.DeclareOk这样的响应
- argument: 其他选项

```
// demo
ch.ExchangeDeclare(
    "eventti.event",   // name
```

```
    "fanout", // type
    true,     // durable
    false,    // auto-deleted
    false,    // internal
    false,    // no-wait
    nil,      // arguments
);
```

发布

```
ch.Publish(
    r.ExChange,  // 交换机名称
    "",    // 路由键
    false,  // 消息发送成功确认(没有队列会异常)
    false,  // 消息发送失败回调(队列中没有消费者会异常)
    amqp.Publishing{        // 发送的消息
        ContentType: "text/plain",
        Body:        []byte(message),
})
```

绑定(Binding)

RabbitMQ中通过Binding将Exchange和queue关联起来，这样就可以正确的路由到对应的队列。

在绑定交换机和队列时通常会指定一个binding key 当binding key 和生产者指定的routing key 相配的时候，消息就会被路由到对应的队列中。

binding key 不是一定会生效，要看交换机的类型，比如类型时fanout，则会进行广播，将消息发送所有绑定的队列。

交换机类型(Exchange Types)

Exchange Type有fanout1、direct、topic、headers这四种。

fanout

fanout类型会把所有发送到fanout Exchange的消息都会被转发到与该Exchange 绑定(Binding)的所Queue上。就是广播。

direct

direct类型会把消息路由到那些binding key与routing key**完全匹配**的Queue中。

topic

topic类型在direct类型的匹配规则上有约束：

- routing key是一个句点号"."分隔的字符串
- binding key也是一个句点号"."分隔的字符串
- binding key中存在两种特殊字符*、#进行模糊匹配,其中*匹配一个单词，#匹配零个或者多个单词

实例：a.b.c 会被匹配到 *.b.*和*.*.c

headers

消息发布前,为消息定义一个或多个键值对的消息头,然后消费者接收消息同时需要定义类似的键值对求头:(如:x-mactch=all或者x_match=any)，只有请求头与消息头匹配,才能接收消息,忽略RoutingKe
。

RabbitMQ队列模式

简单模式

点对点,一个生产者产生消息发送至消息队列，一个消费者消费。

go实现：

1. 定义结构体

```
package RabbitMq

import (
    "fmt"
    "github.com/streadway/amqp"
)

// 这里主要是RabbitMQ的一些信息。包括其结构体和函数。

// 连接信息
const MQURL = "amqp://du:du@129.211.78.6:5672/dudevirtualhost"

// RabbitMQ结构体
type RabbitMQ struct {
    //连接
    conn    *amqp.Connection
    channel *amqp.Channel
    //队列
    QueueName string
    //交换机名称
    ExChange string
    //绑定的key名称
    Key string
    //连接的信息，上面已经定义好了
    MqUrl string
}

// 创建结构体实例，参数队列名称、交换机名称和bind的key（也就是几个大写的，除去定义好的常信息）
func NewRabbitMQ(queueName string, exChange string, key string) *RabbitMQ {
```

```
    return &RabbitMQ{QueueName: queueName, ExChange: exChange, Key: key, MqUrl: MQU
L}
}

// 关闭conn和chanel的方法
func (r *RabbitMQ) Destory() {
    r.channel.Close()
    r.conn.Close()
}

// 错误的函数处理
func (r *RabbitMQ) failOnErr(err error, message string) {
    if err != nil {
        fmt.Printf("err是:%s,小杜同学手写的信息是:%s", err, message)
    }
}
```

2. 实现

```
package RabbitMq

import (
    "fmt"
    "github.com/streadway/amqp"
    "log"
)

//创建简单模式下的实例，只需要queueName这个参数，其中exchange是默认的，key则不需要。
func NewRabbitMQSimple(queueName string) *RabbitMQ {
    rabbitmq := NewRabbitMQ(queueName, "", "")
    var err error
    //获取参数connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.MqUrl)
    rabbitmq.failOnErr(err, "连接connection失败")
    //获取channel参数
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "获取channel参数失败")
    return rabbitmq
}

//直接模式,生产者.
func (r *RabbitMQ) PublishSimple(message string) {
    //第一步，申请队列，如不存在，则自动创建之，存在，则路过。
    _, err := r.channel.QueueDeclare(
        r.QueueName,
        "",
        false,
        false,
        false,
        false,
        nil,
    )
    if err != nil {
        fmt.Printf("创建连接队列失败：%s", err)
```

```
    }

    //第二步，发送消息到队列中
    r.channel.Publish(
        r.ExChange,
        r.QueueName,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//直接模式，消费者
func (r *RabbitMQ) ConsumeSimple() {
    //第一步,申请队列,如果队列不存在则自动创建,存在则跳过
    q, err := r.channel.QueueDeclare(
        r.QueueName,
        //是否持久化
        false,
        //是否自动删除
        false,
        //是否具有排他性
        false,
        //是否阻塞处理
        false,
        //额外的属性
        nil,
    )
    if err != nil {
        fmt.Println(err)
    }
    //第二步,接收消息
    msgs, err := r.channel.Consume(
        q.Name,
        "",   //用来区分多个消费者
        true, //是否自动应答,告诉我已经消费完了
        false,
        false, //若设置为true,则表示为不能将同一个connection中发送的消息传递给这个connection
的消费者.
        false, //消费队列是否设计阻塞
        nil,
    )
    if err != nil {
        fmt.Printf("消费者接收消息出现问题:%s", err)
    }

    forever := make(chan bool)
    //启用协程处理消息
    go func() {
        for d := range msgs {
            log.Printf("小杜同学写的Simple模式接收到了消息:%s\n", d.Body)
        }
```

```
    }()
    log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
    <-forever
}
```

3. 生产者发送消息

```
package main

import (
    "fmt"
    "rabbitmq20181121/RabbitMq"
)

func main() {
    rabbitmq := RabbitMq.NewRabbitMQSimple("duQueueName1912161843")
    rabbitmq.PublishSimple("他是客，你是心上人。 ---来自simple模式")
    fmt.Println("发送成功！ ")
}
```

4. 消费者消费消息

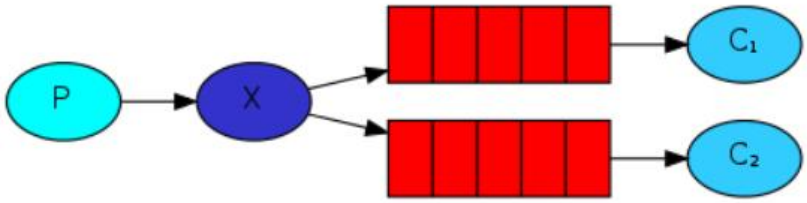
```
package main

import (
    "fmt"
    "rabbitmq20181121/RabbitMq"
)

func main() {
    rabbitmq := RabbitMq.NewRabbitMQSimple("duQueueName1912161843")
    rabbitmq.ConsumeSimple()
    fmt.Println("接收成功！ ")
}
```

发布/订阅模式

只有绑定了当前交换机的队列才能收到消息。



代码:

1. 实现

```
package RabbitMq

import (
    "fmt"
```

```
    "github.com/streadway/amqp"
)

//这里是订阅模式的相关代码。
//订阅模式需要用到exchange。

//获取订阅模式下的rabbitmq的实例
func NewRabbitMqSubscription(exchangeName string) *RabbitMQ {
    //创建rabbitmq实例
    rabbitmq := NewRabbitMQ("", exchangeName, "")
    var err error
    //获取connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.MqUrl)
    rabbitmq.failOnErr(err, "订阅模式连接rabbitmq失败。")
    //获取channel
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "订阅模式获取channel失败")
    return rabbitmq
}

//订阅模式发布消息
func (r *RabbitMQ) PublishSubscription(message string) {
    //第一步，尝试连接交换机
    err := r.channel.ExchangeDeclare(
        r.ExChange,
        "fanout", //这里一定要设计为"fanout"也就是广播类型。
        true,
        false,
        false,
        false,
        nil,
    )
    r.failOnErr(err, "订阅模式发布方法中尝试连接交换机失败。")

    //第二步，发送消息
    err = r.channel.Publish(
        r.ExChange,
        "",
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//订阅模式消费者
func (r *RabbitMQ) ConsumeSbuscription() {
    //第一步，试探性创建交换机exchange
    err := r.channel.ExchangeDeclare(
        r.ExChange,
        "fanout",
        true,
        false,
```

```
        false,
        false,
        nil,
    )
    r.failOnErr(err, "订阅模式消费方法中创建交换机失败。")

    //第二步，试探性创建队列queue
    q, err := r.channel.QueueDeclare(
        "", //随机生产队列名称
        false,
        false,
        true,
        false,
        nil,
    )
    r.failOnErr(err, "订阅模式消费方法中创建创建队列失败。")

    //第三步，绑定队列到交换机中
    err = r.channel.QueueBind(
        q.Name,
        "", //在pub/sub模式下key要为空
        r.ExChange,
        false,
        nil,
    )

    //第四步，消费消息
    messages, err := r.channel.Consume(
        q.Name,
        "",
        true,
        false,
        false,
        false,
        nil,
    )

    forever := make(chan bool)
    go func() {
        for d := range messages {
            fmt.Printf("小杜同学写的订阅模式收到的消息：%s\n", d.Body)
        }
    }()

    fmt.Println("订阅模式消费者已开启，退出请按 CTRL+C\n")
    <-forever

}
```

2. 生产者代码

```
package main

import (
```

```
    "fmt"
    "rabbitmq20181121/RabbitMq"
    "strconv"
    "time"
)

func main() {
    rabbitmq := RabbitMq.NewRabbitMqSubscription("duexchangeName")
    for i := 0; i < 100; i++ {
        rabbitmq.PublishSubscription("订阅模式生产第" + strconv.Itoa(i) + "条数据")
        fmt.Printf("订阅模式生产第" + strconv.Itoa(i) + "条数据\n")
        time.Sleep(1 * time.Second)
    }
}
```

3. 消费者代码

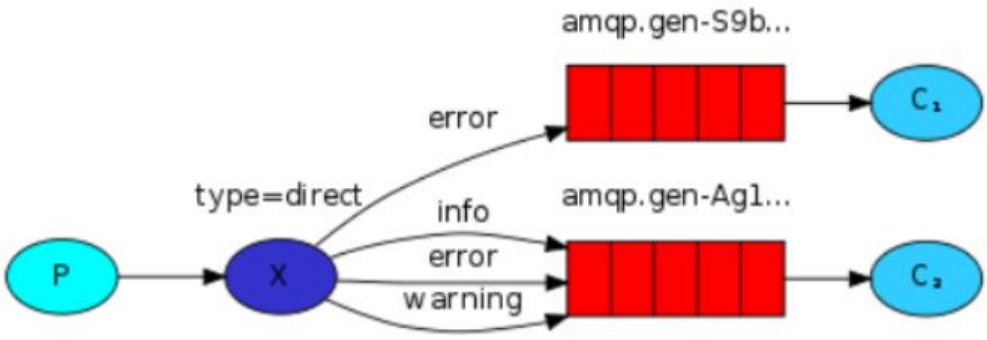
```
package main

import "rabbitmq20181121/RabbitMq"

func main() {
    rabbitmq := RabbitMq.NewRabbitMqSubscription("duexchangeName")
    rabbitmq.ConsumeSbuscription()
}
```

路由模式

根据routing key 和 binding key 完全匹配的路由规则进行分发。



代码：

1. 实现

```
package main

import "rabbitmq20181121/RabbitMq"

func main() {
    rabbitmq := RabbitMq.NewRabbitMqSubscription("duexchangeName")
    rabbitmq.ConsumeSbuscription()
```

```
}1 package RabbitMq

import (
    "github.com/streadway/amqp"
    "log"
)

//rabbitmq的路由模式。
//主要特点不仅一个消息可以被多个消费者消费还可以由生产端指定消费者。
//这里相对比订阅模式就多了一个routingkey的设计，也是通过这个来指定消费者的。
//创建exchange的kind需要是"direct",不然就不是roting模式了。

//创建rabbitmq实例，这里有了routingkey为参数了。
func NewRabbitMqRouting(exchangeName string, routingKey string) *RabbitMQ {
    rabbitmq := NewRabbitMQ("", exchangeName, routingKey)
    var err error
    //获取connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.MqUrl)
    rabbitmq.failOnErr(err, "创建rabbit的路由实例的时候连接出现问题")
    //获取channel
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "创建rabbitmq的路由实例时获取channel出错")
    return rabbitmq
}

//路由模式，产生消息。
func (r *RabbitMQ) PublishRouting(message string) {
    //第一步，尝试创建交换机，与pub/sub模式不同的是这里的kind需要是direct
    err := r.channel.ExchangeDeclare(r.ExChange, "direct", true, false, false, false, nil)
    r.failOnErr(err, "路由模式，尝试创建交换机失败")
    //第二步，发送消息
    err = r.channel.Publish(
        r.ExChange,
        r.Key,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//路由模式，消费消息。
func (r *RabbitMQ) ConsumerRouting() {
    //第一步，尝试创建交换机，注意这里的交换机类型与发布订阅模式不同，这里的是direct
    err := r.channel.ExchangeDeclare(
        r.ExChange,
        "direct",
        true,
        false,
        false,
        false,
        nil,
    )
```

```
    r.failOnErr(err, "路由模式，创建交换机失败。")

    //第二步，尝试创建队列,注意这里队列名称不用写，这样就会随机产生队列名称
    q, err := r.channel.QueueDeclare(
        "",
        false,
        false,
        true,
        false,
        nil,
    )
    r.failOnErr(err, "路由模式，创建队列失败。")

    //第三步，绑定队列到exchange中
    err = r.channel.QueueBind(q.Name, r.Key, r.ExChange, false, nil)

    //第四步，消费消息。
    messages, err := r.channel.Consume(q.Name, "", true, false, false, false, nil)
    forever := make(chan bool)
    go func() {
        for d := range messages {
            log.Printf("小杜同学写的路由模式(routing模式)收到消息为：%s。\n", d.Body)
        }
    }()
    <-forever
}
```

2. 生产者代码

```
package main

import (
    "fmt"
    "rabbitmq20181121/RabbitMq"
    "strconv"
    "time"
)

func main() {
    rabbitmq1 := RabbitMq.NewRabbitMqRouting("duExchangeName", "one")
    rabbitmq2 := RabbitMq.NewRabbitMqRouting("duExchangeName", "two")
    rabbitmq3 := RabbitMq.NewRabbitMqRouting("duExchangeName", "three")
    for i := 0; i < 100; i++ {
        rabbitmq1.PublishRouting("路由模式one" + strconv.Itoa(i))
        rabbitmq2.PublishRouting("路由模式two" + strconv.Itoa(i))
        rabbitmq3.PublishRouting("路由模式three" + strconv.Itoa(i))
        time.Sleep(1 * time.Second)
        fmt.Printf("在路由模式下，routingKey为one,为two,为three的都分别生产了%d条消息\n", i)
    }
}
```

3. 消费者代码

```
package main
```

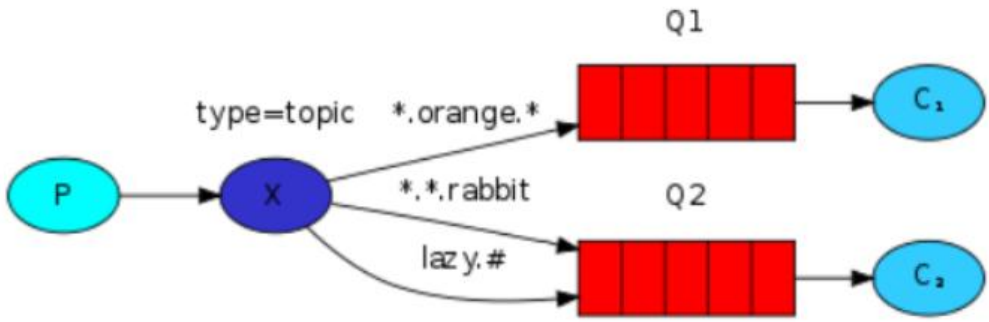
```
import "rabbitmq20181121/RabbitMq"

func main() {
    one := RabbitMq.NewRabbitMqRouting("duExchangeName", "one")
    one.ConsumerRouting()

    two:= RabbitMq.NewRabbitMqRouting("duExchangeName", "two")
    two.ConsumerRouting()
}
```

主题模式

也就是topic类型的交换类型,与路由模式相比，可进行模糊匹配，如果Exchange没有发现能与routing key 匹配的队列，则会丢弃消息。



代码：

1. 实现

```
package RabbitMq

import (
    "github.com/streadway/amqp"
    "log"
)

//topic模式
//与routing模式不同的是这个exchange的kind是"topic"类型的。
//topic模式的特别是可以以通配符的形式来指定与之匹配的消费者。
//"*"表示匹配一个单词。 “#”表示匹配多个单词，亦可以是0个。

//创建rabbitmq实例
func NewRabbitMqTopic(exchangeName string, routingKey string) *RabbitMQ {
    rabbitmq := NewRabbitMQ("", exchangeName, routingKey)
    var err error
    //获取connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.MqUrl)
    rabbitmq.failOnErr(err, "创建rabbit的topic模式时候连接出现问题")
    //获取channel
```

```
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "创建rabbitmq的topic实例时获取channel出错")
    return rabbitmq
}

//topic模式。生产者。
func (r *RabbitMQ) PublishTopic(message string) {
    //第一步，尝试创建交换机,这里的kind的类型要改为topic
    err := r.channel.ExchangeDeclare(
        r.ExChange,
        "topic",
        true,
        false,
        false,
        false,
        nil,
    )
    r.failOnErr(err, "topic模式尝试创建exchange失败。")

    //第二步，发送消息。
    err = r.channel.Publish(
        r.ExChange,
        r.Key,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//topic模式。消费者。"*"表示匹配一个单词。 “#”表示匹配多个单词，亦可以是0个。
func (r *RabbitMQ) ConsumerTopic() {
    //第一步，创建交换机。这里的kind需要是“topic”类型。
    err := r.channel.ExchangeDeclare(
        r.ExChange,
        "topic",
        true, //这里需要是true
        false,
        false,
        false,
        nil,
    )
    r.failOnErr(err, "topic模式，消费者创建exchange失败。")

    //第二步，创建队列。这里不用写队列名称。
    q, err := r.channel.QueueDeclare(
        "",
        false,
        false,
        true,
        false,
        nil,
    )
```

```
    r.failOnErr(err, "topic模式，消费者创建queue失败。")

    //第三步，将队列绑定到交换机里。
    err = r.channel.QueueBind(
        q.Name,
        r.Key,
        r.ExChange,
        false,
        nil,
    )

    //第四步，消费消息。
    messages, err := r.channel.Consume(
        q.Name,
        "",
        true,
        false,
        false,
        false,
        nil,
    )

    forever := make(chan bool)
    go func() {
        for d := range messages {
            log.Printf("小杜同学写的topic模式收到了消息：%s。\n", d.Body)
        }
    }()
    <-forever

}
```

2. 生产者代码

```
package main

import (
    "fmt"
    "rabbitmq20181121/RabbitMq"
    "strconv"
    "time"
)

func main() {
    one := RabbitMq.NewRabbitMqTopic("exchangeNameTpoic1224", "Singer.Jay")
    two := RabbitMq.NewRabbitMqTopic("exchangeNameTpoic1224", "Persident.XIDADA")
    for i := 0; i < 100; i++ {
        one.PublishTopic("小杜同学，topic模式，Jay," + strconv.Itoa(i))
        two.PublishTopic("小杜同学，topic模式，All," + strconv.Itoa(i))
        time.Sleep(1 * time.Second)
        fmt.Printf("topic模式。这是小杜同学发布的消息%v \n", i)
    }
}
```

3. 消费者代码1

```
package main

import "rabbitmq20181121/RabbitMq"

func main() {
    jay := RabbitMq.NewRabbitMqTopic("exchangeNameTpoic1224", "Singer.*")
    jay.ConsumerTopic()
}
```

4. 消费者代码2

```
package main

import "rabbitmq20181121/RabbitMq"

func main() {
    jay := RabbitMq.NewRabbitMqTopic("exchangeNameTpoic1224", "#")
    jay.ConsumerTopic()
}
```