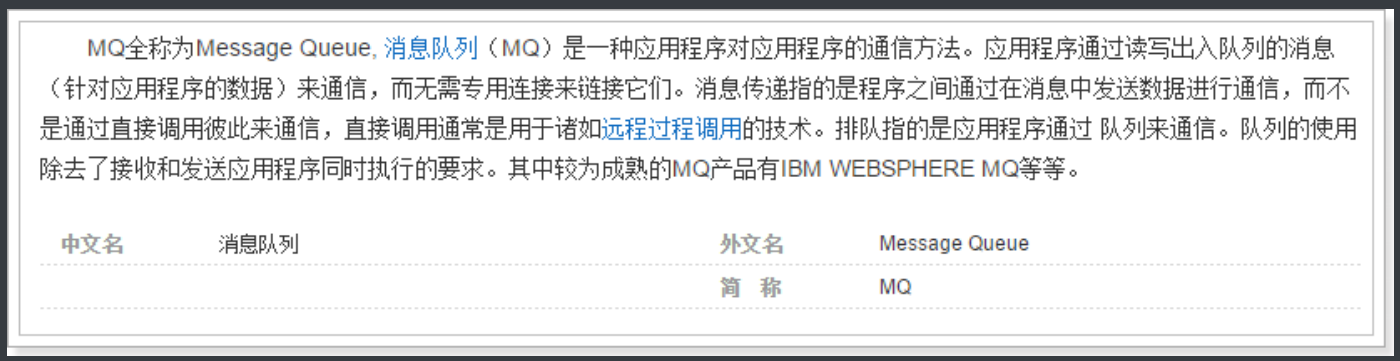
1.消息队列概述

1.1什么是消息队列

消息队列即MQ(Message Queue)，从字面意思上看，本质是个队列，FIFO 先入先出，只不过队列中存放的内容是Message而已。消息队列还是一种跨进程的通信机制（是应用程序对应用程序的通信），用于上下游传递消息。

在互联网架构中，MQ 是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。使用了 MQ 之后，消息发送上游只需要依赖 MQ，不用依赖其他服务。



MQ全称为Message Queue, 消息队列（MQ）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用连接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于诸如远程过程调用的技术。排队指的是应用程序通过 队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。其中较为成熟的MQ产品有IBM WEBSPHERE MQ等等。

中文名	消息队列	外文名	Message Queue
		简　称	MQ

消息队列是典型的：生产者、消费者模型。生产者不断向消息队列中生产消息，消费者不断的从队列中获取消息。因为消息的生产和消费都是异步的，而且只关心消息的发送和接收，没有业务逻辑的侵入，这样就实现了生产者和消费者的解耦。

1.2消息队列作用

> 使用场景

消息队列的作用有三个

- 系统**解耦**
  - 生产者直接发消息给消费者，两个系统之间存在耦合
- **异步**处理

- 流量**削峰**

1.2.1流量消峰

举个例子，如果订单系统最多能处理一万次订单，这个处理能力应付正常时段的下单时绰绰有余，正常时段我们下单一秒后就能返回结果。但是在高峰期，如果有两万次下单操作系统是处理不了的，只能限制订单超过一万后不允许用户下单，这样的用户体验不是特别好。使用消息队列做缓冲，我们可以取消这个限制，把一秒内下的订单分散成一段时间来处理，这时有些用户可能在下单十几秒后才能收到下单成功的操作，但是比不能下单的体验要好。

1.2.2.应用解耦

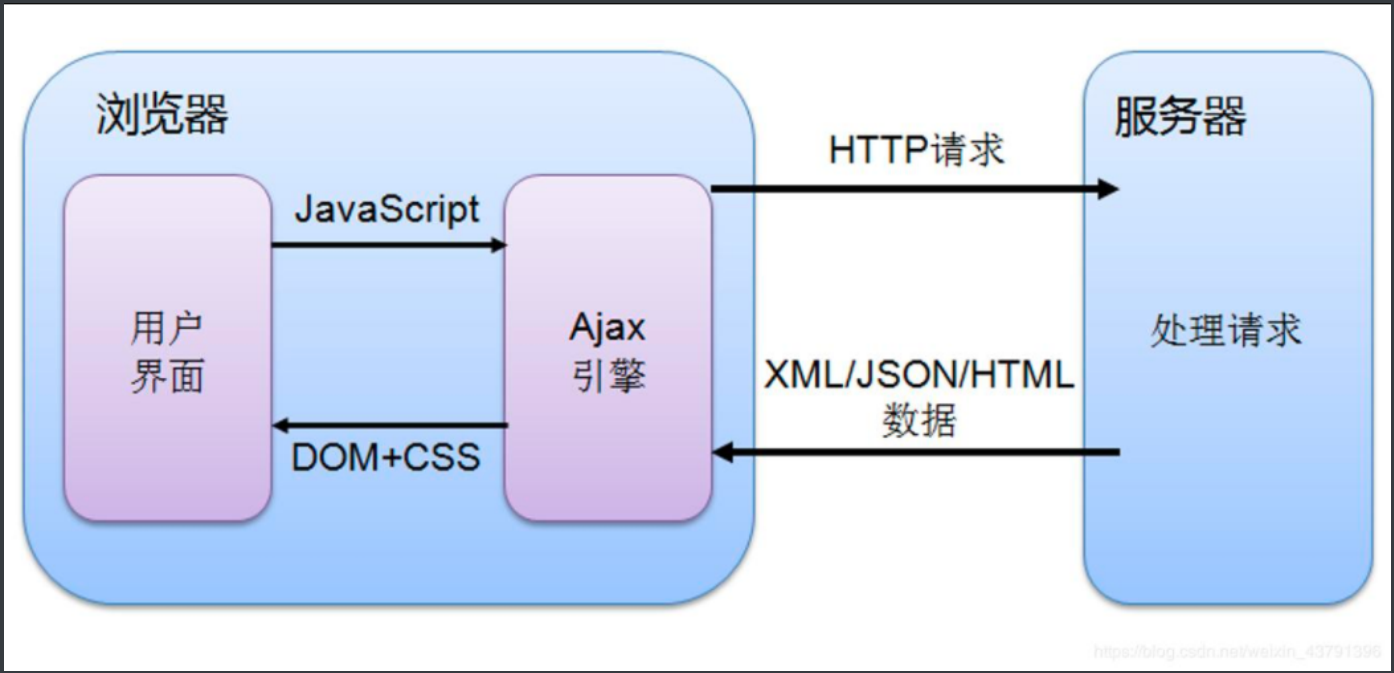
以电商应用为例，应用中有订单系统、库存系统、物流系统、支付系统（天下合久必分，分久必合）。用户创建订单后，如果耦合调用库存系统、物流系统、支付系统，任何一个子系统出了故障，都会造成下单操作异常。当转变成基于消息队列的方式后，系统间调用的问题会减少很多，比如物流系统因为发生故障，需要几分钟来修复。在这几分钟的时间里，物流系统要处理的内存被缓存在消息队列中，用户的下单操作可以正常完成。当物流系统恢复后，继续处理订单信息即可，中单用户感受不到物流系统的故障，提升系统的可用性。

1.2.3异步处理

有些服务间调用是异步的，例如 A 调用 B，B 需要花费很长时间执行，但是 A 需要知道 B 什么时候可以执行完，以前一般有两种方式，

- A 过一段时间去调用 B 的查询 api 查询。
- A 提供一个 callback api， B 执行完之后调用 api 通知 A 服务。

这两种方式都不是很优雅，使用消息总线，可以很方便解决这个问题，A 调用 B 服务后，只需要监听 B 处理完成的消息，当 B 处理完成后，会发送一条消息给 MQ，MQ 会将此消息转发给 A 服务。这样 A 服务既不用循环调用 B 的查询 api，也不用提供 callback api。同样B 服务也不用做这些操作。A 服务还能及时的得到异步处理成功的消息。

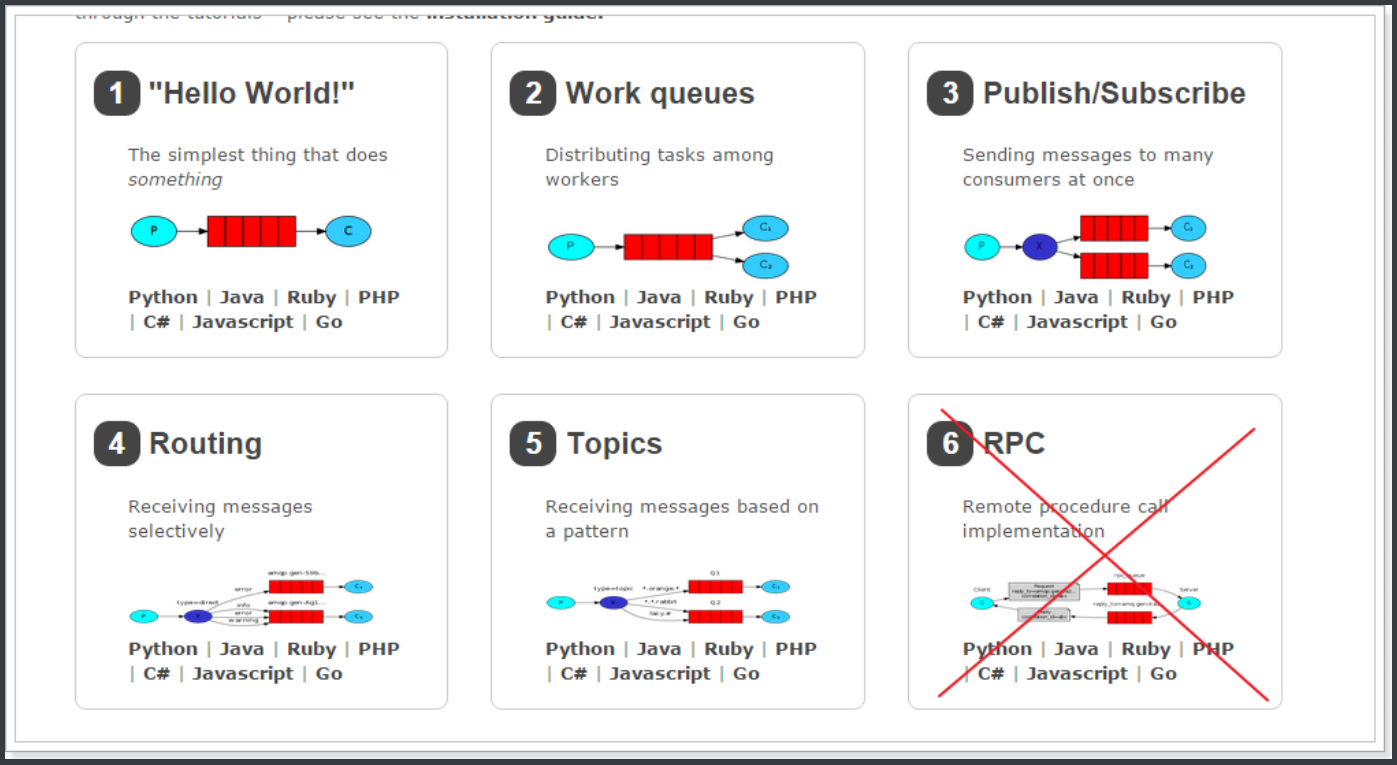


2.RabbitMQ介绍

2.1 RabbitMQ四大核心概念

- 生产者　　产生数据发送消息的程序是生产者
- 交换机　　交换机是 RabbitMQ 非常重要的一个部件，一方面它接收来自生产者的消息，另一方面它将消息推送到队列中。交换机必须确切知道如何处理它接收到的消息，是将这些消息推送到特定队列还是推送到多个队列，亦或者是把消息丢弃，这个得有交换机类型决定
- 队列　　队列是 RabbitMQ 内部使用的一种数据结构，尽管消息流经 RabbitMQ 和应用程序，但它们只能存储在队列中。队列仅受主机的内存和磁盘限制的约束，本质上是一个大的消息缓冲区。许多生产者可以将消息发送到一个队列，许多消费者可以尝试从一个队列接收数据。这就是我们使用队列的方式
- 消费者　　消费与接收具有相似的含义。消费者大多时候是一个等待接收消息的程序。请注意生产者，消费者和消息中间件很多时候并不在同一机器上。同一个应用程序既可以是生产者又是可以是消费者。

2.2RabbitMQ的消息模型



2.3 名词介绍

- **Broker**：接收和分发消息的应用，RabbitMQ Server 就是 Message Broker
- **Virtual host**：出于多租户和安全因素设计的，把 AMQP 的基本组件划分到一个虚拟的分组中，类似

  于网络中的 namespace 概念。当多个不同的用户使用同一个 RabbitMQ server 提供的服务时，可以划分出

  多个 vhost，每个用户在自己的 vhost 创建 exchange / queue 等
- **Connection**：publisher / consumer 和 broker 之间的 TCP 连接
- **Channel**：如果每一次访问 RabbitMQ 都建立一个 Connection，在消息量大的时候建立 TCP Connection 的开销将是巨大的，效率也较低。channel 是在 Connection 内部建立的逻辑连接，如果应用程

  序支持多线程，通常每个 Thread 创建单独的 channel 进行通讯，AMQP method 包含了 channel id 帮助客户端和 message broker 识别 channel，所以 channel 之间是完全隔离的。**Channel 作为轻量级的**

Connection 极大减少了操作系统建立 TCP connection 的开销

- **Exchange**：message 到达 broker 的第一站，根据分发规则，匹配查询表中的 routing key，分发消息到 queue 中去。常用的类型有：direct (point-to-point), topic (publish-subscribe) and fanout(multicast)
- **Queue**：消息最终被送到这里等待 consumer 取走
- **Binding**：exchange 和 queue 之间的虚拟连接，binding 中可以包含 routing key，Binding 信息被保存到 exchange 中的查询表中，用于 message 的分发依据

3.RabbitMQ简单模式

RabbitMQ简单模式，非常简单就是生产者往消息队列发送消息，消费者从消息队列取消息。如下图所示：

send

```
package main

import (
  "context"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
)

// 连接失败，错误处理函数
func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
```

```
    }
}

func main() {
    // 连接服务器
    connection, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    failOnError(err, "Failed to connect to RabbitMQ")
    defer connection.Close()

    // 打开通道
    channel, err := connection.Channel()
    failOnError(err, "Failed to open a channel")
    defer channel.Close()

    //设置队列
    queue, err := channel.QueueDeclare(
        "simple", // name
        false,    // durable
        false,    // delete when unused
        false,    // exclusive
        false,    // no-wait
        nil,      // arguments
    )
    failOnError(err, "Failed to declare a queue")

    //发送消息
    msg := "rabbitmq simple msg"
    err = channel.PublishWithContext(
        context.Background(),
        "", queue.Name,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(msg),
        })
```

```
    failOnError(err, "Failed to publish a message")
    log.Printf(" [x] Sent %s\n", msg)
}
```

receive

```
package main

/**
consumer
*/
import (
    amqp "github.com/rabbitmq/amqp091-go"
    "log"
)

// 连接失败，错误处理函数
func failOnError(err error, msg string) {
    if err != nil {
        log.Panicf("%s: %s", msg, err)
    }
}
func main() {
    // 连接服务器
    connection, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    failOnError(err, "Failed to connect to RabbitMQ")
    defer connection.Close()

    // 打开通道
    ch, err := connection.Channel()
    failOnError(err, "Failed to open a channel")
    defer ch.Close()

    // 消息队列
    queue, err := ch.QueueDeclare(
```

```
    "simple", // name
    false,    // durable
    false,    // delete when unused
    false,    // exclusive
    false,    // no-wait
    nil,      // arguments
  )
  failOnError(err, "Failed to declare a queue")

  // 消费消息（接收消息）
  messages, err := ch.Consume(
    queue.Name, // queue
    "",         // consumer
    true,       // auto-ack
    false,      // exclusive
    false,      // no-local
    false,      // no-wait
    nil,        // args
  )
  failOnError(err, "Failed to register a consumer")

  var forever chan struct{}
  go func() {
    for d := range messages {
      log.Printf("Received a message: %s", d.Body)
    }
  }()

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

4.RabbitMQ工作模式

工作队列用来将耗时的任务分发给多个消费者（工作者），主要解决这样的问题：处理资源密集型任务，并且还要等他完成。有了工作队列，我们就可以将具体的工作放到后面去做，将工作封装为一个消息，发送到队列中，一个工作进程就可以取出消息并完成工作。如果启动了多个工作进程，那么工作就可以在多个进程间共享。

> 注意：这里的消息不能被重复消费，一个消息被消费者消费后，不能再被另外一个消费者消费。

producer

```
package producer

import (
  "context"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "os"
  "strings"
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
```

```
  }
}
func bodyFrom(args []string) string {
  var s string
  if (len(args) < 2) || os.Args[1] == "" {
    s = "hello"
  } else {
    s = strings.Join(args[1:], " ")
  }
  return s
}
func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer ch.Close()

  queue, err := ch.QueueDeclare(
    "task_queue",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to declare a queue")

  body := bodyFrom(os.Args)

  err = ch.PublishWithContext(context.Background(), "", queue.Name,
false, false,
    amqp.Publishing{
      DeliveryMode: amqp.Persistent,
```

```
        ContentType:  "text/plain",
        Body:         []byte(body),
      })
    failOnError(err, "Failed to publish a message")
    log.Printf(" [x] Sent %s", body)
  }
```

comsumer

```
  package main

  import (
    "bytes"
    amqp "github.com/rabbitmq/amqp091-go"
    "log"
    "time"
  )

  func failOnError(err error, msg string) {
    if err != nil {
      log.Panicf("%s: %s", msg, err)
    }
  }

  func main() {
    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    failOnError(err, "Failed to connect to RabbitMQ")
    defer conn.Close()

    ch, err := conn.Channel()
    failOnError(err, "Failed to open a channel")
    defer ch.Close()

    q, err := ch.QueueDeclare(
      "task_queue", // 队列名称
```

```
    true,         // 是否持久化
    false,        // 是否自动删除
    false,        // 消息是否共享
    false,        // 是否等待
    nil,          // 其他参数
  )
  failOnError(err, "Failed to declare a queue")

  /**
  ch.Qos(1, 0, false)
  prefetch count: 这个参数指定可以从消息代理预取的消息数量。在这个例子中，预取数量被设置为1，这意味着消费者会预先从消息代理那里接收一条消息。
  prefetch size: 这个参数指定可以预取的消息的最大字节大小。在这个例子中，它被设置为0，这意味着不会根据消息的大小来限制预取。
  global: 这个参数指定是否所有的通道都使用这个预取设置。在这个例子中，它被设置为false，这意味着这个预取设置只对当前的通道有效。
  */
  err = ch.Qos(
    1,     // prefetch count
    0,     // prefetch size
    false, // global
  )
  failOnError(err, "Failed to set QoS")

  msgs, err := ch.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")
  var forever chan struct{}
```

```
  go func() {
    for d := range msgs {
      log.Printf("Received a message: %s", d.Body)
      dotCount := bytes.Count(d.Body, []byte("."))
      t := time.Duration(dotCount)
      time.Sleep(t * time.Second)
      log.Printf("Done")
      d.Ack(false)
    }
  }()

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

import (
  "bytes"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "time"
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
```

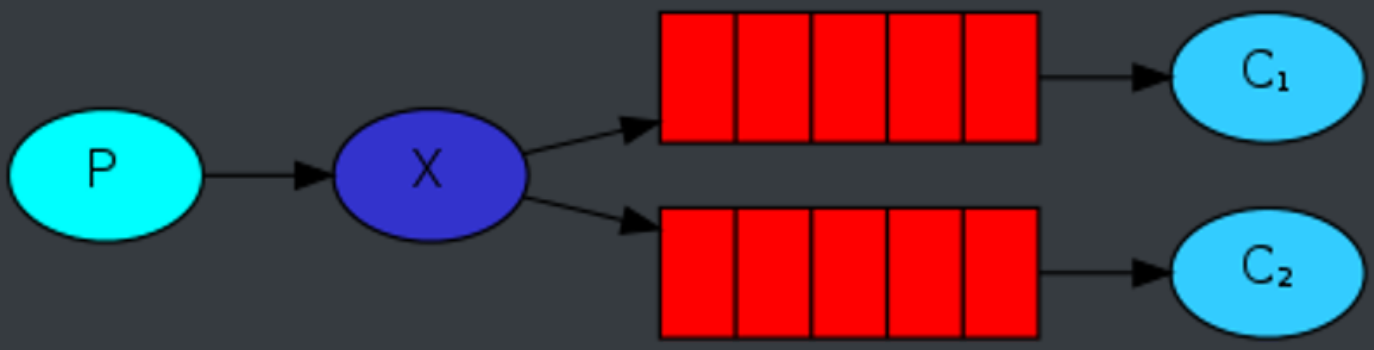
```
  defer ch.Close()

  q, err := ch.QueueDeclare(
    "task_queue", // 队列名称
    true,         // 是否持久化
    false,        // 是否自动删除
    false,        // 消息是否共享
    false,        // 是否等待
    nil,          // 其他参数
  )
  failOnError(err, "Failed to declare a queue")

  /**
  ch.Qos(1, 0, false)
  prefetch count: 这个参数指定可以从消息代理预取的消息数量。在这个例子中，预取数量被设置为1，这意味着消费者会预先从消息代理那里接收一条消息。
  prefetch size: 这个参数指定可以预取的消息的最大字节大小。在这个例子中，它被设置为0，这意味着不会根据消息的大小来限制预取。
  global: 这个参数指定是否所有的通道都使用这个预取设置。在这个例子中，它被设置为false，这意味着这个预取设置只对当前的通道有效。
  */
  err = ch.Qos(
    1,     // prefetch count
    0,     // prefetch size
    false, // global
  )
  failOnError(err, "Failed to set QoS")

  msgs, err := ch.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")
  var forever chan struct{}

  go func() {
    for d := range msgs {
      log.Printf("Received a message: %s", d.Body)
      dotCount := bytes.Count(d.Body, []byte("."))
      t := time.Duration(dotCount)
      time.Sleep(t * time.Second)
      log.Printf("Done")
      /**
      d.Ack(false) 来确认这条消息已经被成功接收并处理。
      这样做的好处是，如果消息在消费者端未被正确处理，
      消息代理可以知道这一点，并可能将其重新发送给其他消费者。

      如果为 false，则表示消息处理成功，并且不会再次发送。
      如果为 true，则表示消息处理可能会失败，并且消息代理可以在失败时尝试将其发送给
其他消费者。
      */
      d.Ack(false)
    }
  }()

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

5.RabbitMQ发布订阅模式

和工作队列不同的是，消息可以重复消费。就像我有一个公众号，当我发布消息时，关注我公众号的人都可以收到消息。

RabbitMQ发布订阅模式，不直接将消息发送到消息队列，而是发送给交换器。交换器非常简单。一方面它接收来自生产者的消息，另一方面它将它们推送到队列中。交换器必须确切地知道如何处理它收到的消息。是否应该将其附加到特定队列？它应该附加到许多队列中？或者它应该被丢弃。其规则由 *交换类型*定义。



一般常用的交换器类型有四种： fanout,direct,topic ,headers。

fanout:

fanout会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中。

direct：

direct 类型的交换器路由规则也很简单，会把消息路由到那些Bindingkey和Routingkey完全匹配的队列中。

topic

topic类型的交换器在匹配规则上进行了扩展，他与direct类型的交换器相似，也是将消息路由到bindingkey和routingkey 相匹配的队列中，但这里的匹配规则有些不同。

headers:

headers 类型的交换器不依赖与路由键的匹配规则来路由消息，而是根据发送消息的内容中的headers属性进行匹配，在绑定队列和交换器时指定一组键值对，当发送消息到交换器时，rabbitmq会获取到消息的header(也是一个键值对的形式），对比其中的键值对是否完全匹配队列和交换器绑定时指定的键值对，如果完全匹配则消息会路由到该队列，否则不会路由到该队列，headers类型的交换器性能会很差，且不实用，不推荐使用。

producer

```
package main

import (
  "context"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "os"
  "strings"
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func bodyFrom(args []string) string {
  var s string
  if (len(args) < 2) || os.Args[1] == "" {
    s = "hello"
  } else {
    s = strings.Join(args[1:], " ")
  }
  return s
}

func main() {
```

```
    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
    failOnError(err, "Failed to connect to RabbitMQ")
    defer conn.Close()

    ch, err := conn.Channel()
    failOnError(err, "Failed to open a channel")
    defer ch.Close()

    err = ch.ExchangeDeclare(
      "publish-exchange",
      "fanout",
      true,
      false,
      false,
      false,
      nil,
    )
    failOnError(err, "Failed to declare an exchange")

    body := bodyFrom(os.Args)
    err = ch.PublishWithContext(context.Background(), "publish-exchange",
  "", false, false,
      amqp.Publishing{
        ContentType: "text/plain",
        Body:        []byte(body),
      })
    failOnError(err, "Failed to publish a message")

    log.Printf(" [x] Sent %s", body)
  }
```

comsumer

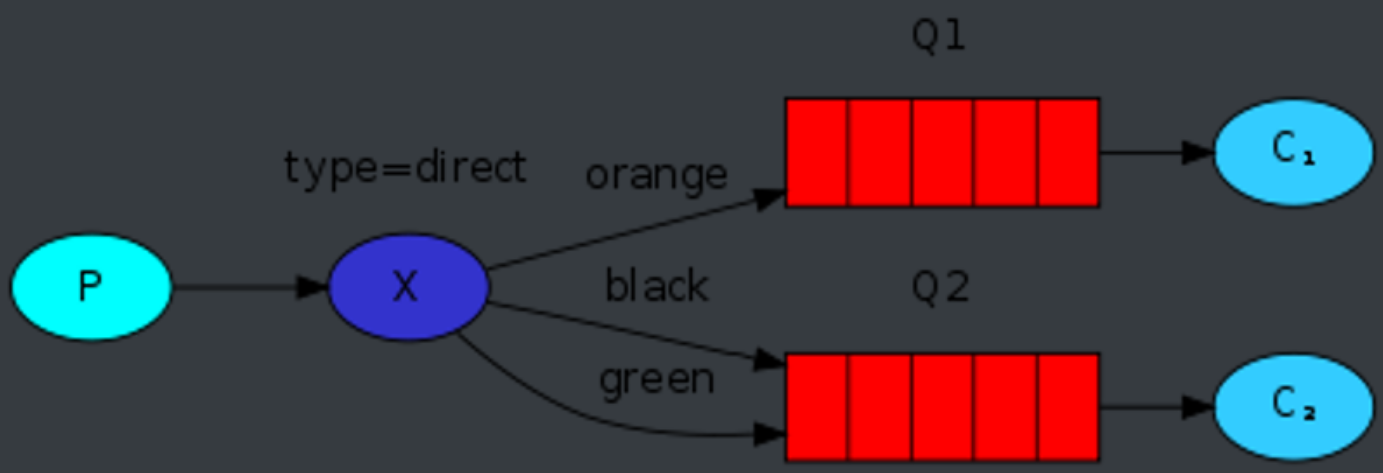
```
  package main

  import (
```

```
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer ch.Close()

  err = ch.ExchangeDeclare(
    "publish-exchange",
    "fanout",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to declare an exchange")

  q, err := ch.QueueDeclare(
    "",    // name
    false, // durable
    false, // delete when unused
    true,  // exclusive
    false, // no-wait
```

```
    nil,   // arguments
  )
  failOnError(err, "Failed to declare a queue")

  err = ch.QueueBind(
    q.Name,
    "",
    "publish-exchange",
    false,
    nil,
  )
  failOnError(err, "Failed to bind a queue")

  consumeMsgs, err := ch.Consume(
    q.Name,
    "",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to register a consumer")
  var forever chan struct{}

  go func() {
    for d := range consumeMsgs {
      log.Printf(" [x] %s", d.Body)
    }
  }()

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

6.RabbitMQ路由模式

发布订阅模式，将消息广播给所有订阅消息的消费者。路由模式仅订阅消息的子集。例如，我们将能够仅将关键错误消息定向到日志文件（以节省磁盘空间），同时仍然能够在控制台上打印所有日志消息。路由模式使用的交换器类型为 `direct` 直接类型。



producer

```
package main

import (
  "context"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "os"
  "strings"
)

/**
路由模式
direct 类型的交换器路由规则也很简单，会把消息路由到那些Bindingkey和Routingkey完全匹配的队列中。
*/
```

```
func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func bodyFrom(args []string) string {
  var s string
  if (len(args) < 3) || os.Args[2] == "" {
    s = "hello"
  } 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
}

func main() {

  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer ch.Close()
```

```
    err = ch.ExchangeDeclare(
      "router-direct",
      "direct",
      true,
      false,
      false,
      false,
      nil,
    )
    failOnError(err, "Failed to declare an exchange")

    body := bodyFrom(os.Args)
    err = ch.PublishWithContext(
      context.Background(),
      "router-direct",
      severityFrom(os.Args),
      false,
      false,
      amqp.Publishing{
        ContentType: "text/plain",
        Body:        []byte(body),
      })

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

    log.Printf(" [x] Sent %s", body)
  }
```

consumer

```
  package main

  import (
    amqp "github.com/rabbitmq/amqp091-go"
    "log"
    "os"
```

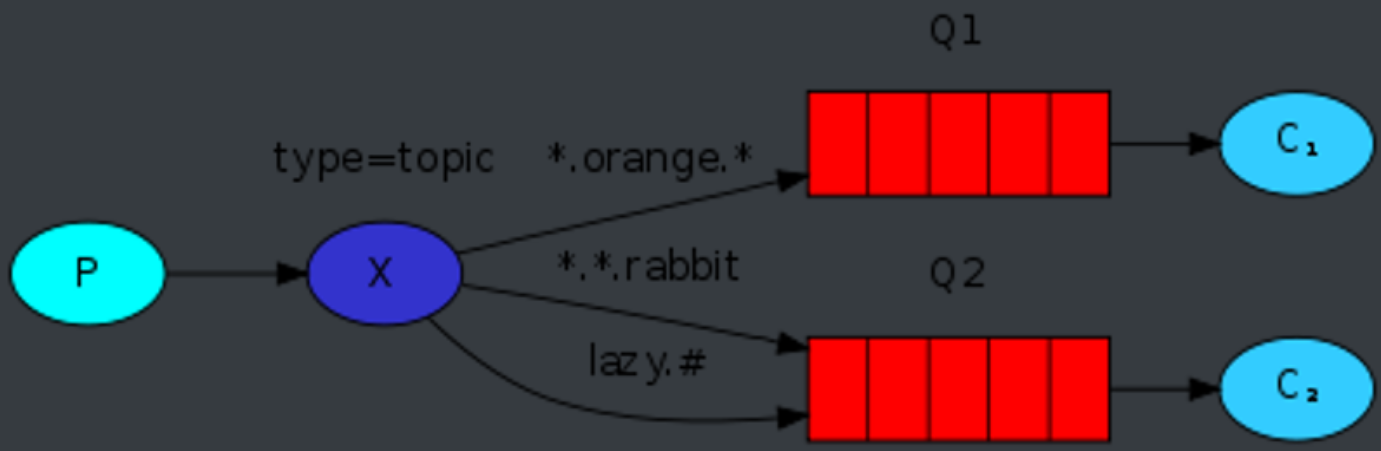
```
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer ch.Close()

  err = ch.ExchangeDeclare(
    "router-direct",
    "direct",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to declare an exchange")

  queue, err := ch.QueueDeclare(
    "",
    false,
    false,
    true,
    false,
    nil,
  )
```

```
    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 _, routeringKey := range os.Args[1:] {
      log.Printf("Binding queue %s to exchange %s with routing key %s",
  queue.Name, "router-direct", routeringKey)
      err = ch.QueueBind(
        queue.Name,
        routeringKey,
        "router-direct",
        false,
        nil,
      )
      failOnError(err, "Failed to bind a queue")
    }

    msgs, err := ch.Consume(
      queue.Name,
      "",
      true,
      false,
      false,
      false,
      nil,
    )
    failOnError(err, "Failed to register a consumer")
    var forever chan struct{}

    go func() {
      for d := range msgs {
        log.Printf(" [x] %s", d.Body)
      }
    }()
```

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

7.RabbitMQ主题模式

topics 主题模式跟 routing 路由模式类似，只不过路由模式是指定固定的路由键 routingKey，而主题模式是可以模糊匹配路由键 routingKey，类似于SQL中 = 和 like 的关系。



producer

```
package main

import (
  "context"
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "os"
  "strings"
)
```

```go
func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}

func severityFrom(args []string) string {
  var s string
  if (len(args) < 2) || os.Args[1] == "" {
    s = "anonymous.info"
  } else {
    s = os.Args[1]
  }
  return s
}

func bodyFrom(args []string) string {
  var s string
  if (len(args) < 3) || os.Args[2] == "" {
    s = "hello"
  } else {
    s = strings.Join(args[2:], " ")
  }
  return s
}

func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  channel, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer channel.Close()

  err = channel.ExchangeDeclare(
```

```
    "topic_router",
    "topic",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to declare an exchange")

  body := bodyFrom(os.Args)
  err = channel.PublishWithContext(context.Background(), "topoc_router",
severityFrom(os.Args), false, false, amqp.Publishing{
    ContentType: "text/plain",
    Body:        []byte(body),
  })
  failOnError(err, "Failed to publish a message")

  log.Printf(" [x] Sent %s", body)
}
```

consumer

```
package main

import (
  amqp "github.com/rabbitmq/amqp091-go"
  "log"
  "os"
)

func failOnError(err error, msg string) {
  if err != nil {
    log.Panicf("%s: %s", msg, err)
  }
}
```

```go
func main() {
  conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
  failOnError(err, "Failed to connect to RabbitMQ")
  defer conn.Close()

  ch, err := conn.Channel()
  failOnError(err, "Failed to open a channel")
  defer ch.Close()

  ch.ExchangeDeclare(
    "topic_router",
    "topic",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to declare an exchange")

  queue, err := ch.QueueDeclare(
    "",
    false,
    false,
    true,
    false,
    nil,
  )
  failOnError(err, "fail to declare a queue")

  if len(os.Args) < 2 {
    log.Printf("Usage: %s [binding_key]...", os.Args[0])
    os.Exit(0)
  }
```

```
  for _, s := range os.Args[1:] {
    log.Printf("Binding queue %s to exchange %s with routing key %s",
queue.Name, "topic_router", s)
    err = ch.QueueBind(
      queue.Name,
      s,
      "topic_router",
      false,
      nil,
    )
    failOnError(err, "Failed to bind a queue")
  }

  msgs, err := ch.Consume(
    queue.Name,
    "",
    true,
    false,
    false,
    false,
    nil,
  )
  failOnError(err, "Failed to register a consumer")

  var forever chan struct{}

  go func() {
    for d := range msgs {
      log.Printf(" [x] %s", d.Body)
    }
  }()

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