RabbitMQ入门与进阶

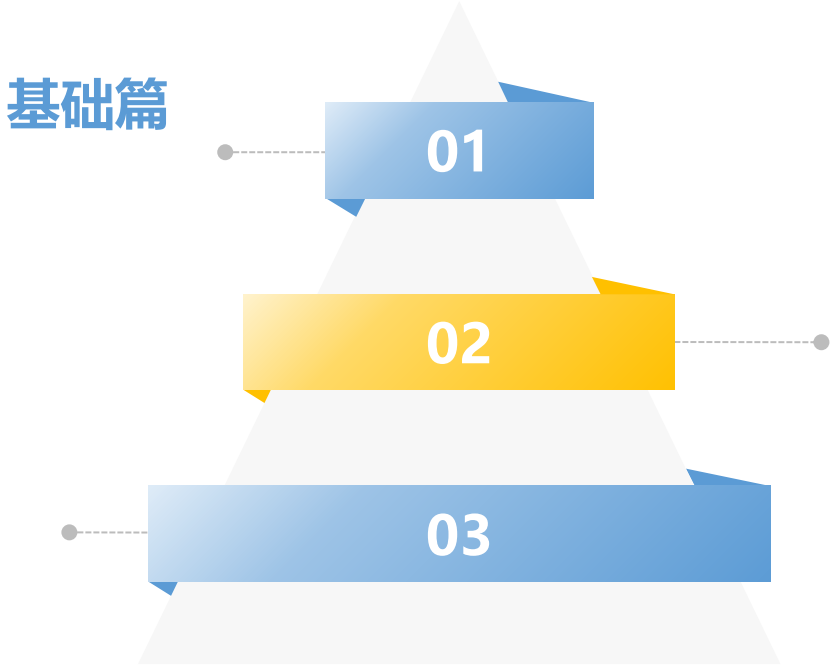
——基于RabbitMQ 3.13.0——

讲师：封捷

Part 01
前言

课程内容、前置要求、课件形式

课程内容



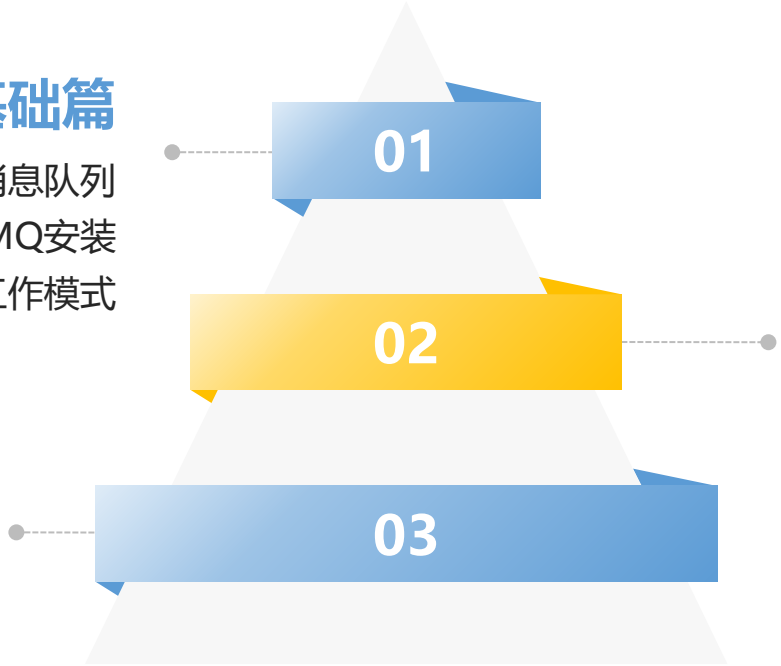
课程内容

基础篇

什么是消息队列

RabbitMQ安装

RabbitMQ的各种工作模式



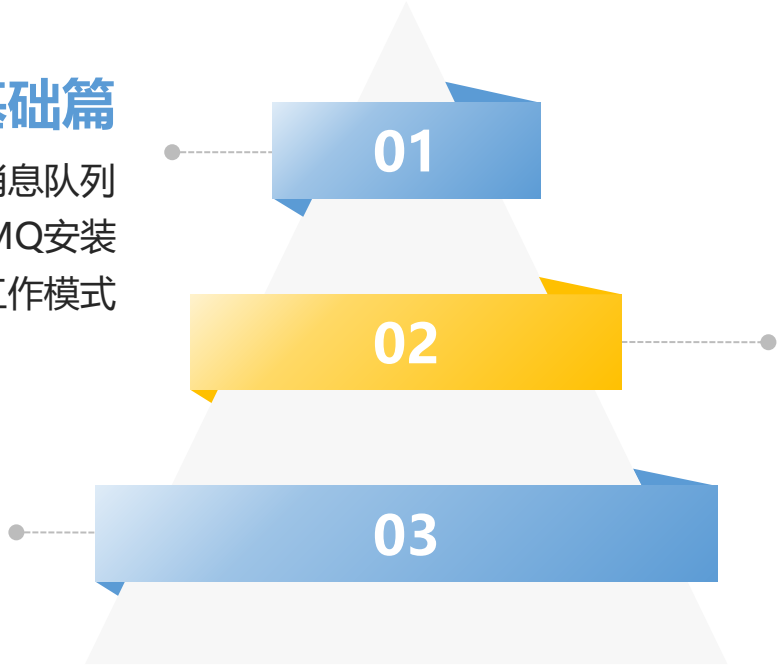
课程内容

基础篇

什么是消息队列

RabbitMQ安装

RabbitMQ的各种工作模式

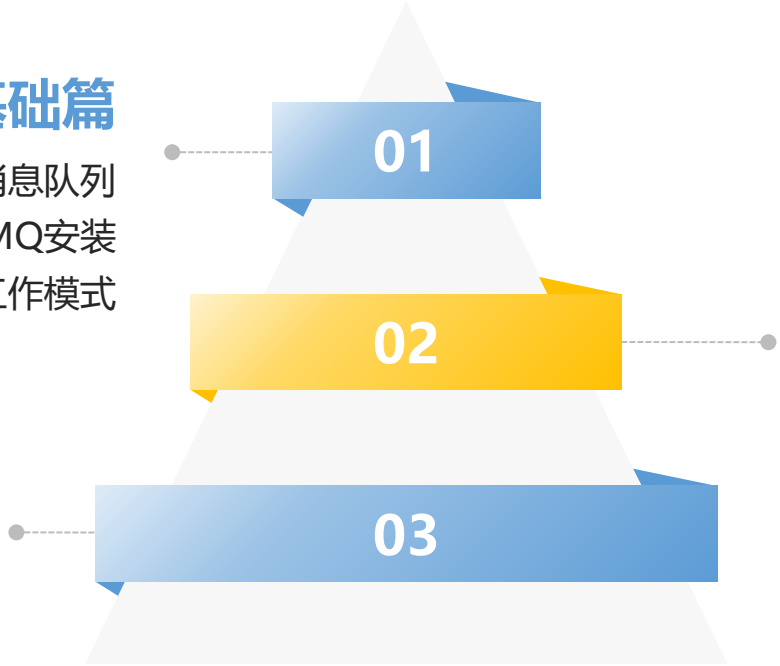


进阶篇

课程内容

基础篇

什么是消息队列
RabbitMQ安装
RabbitMQ的各种工作模式



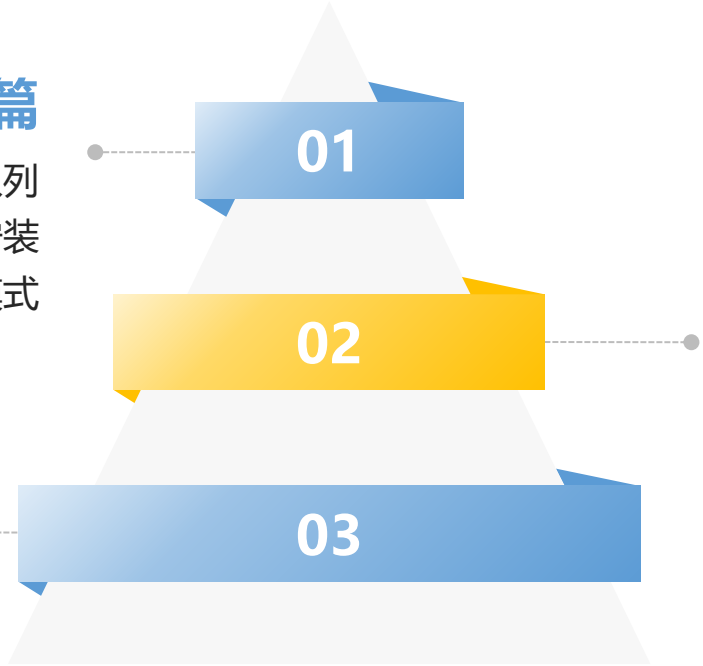
进阶篇

消息可靠性投递
消费端限流
消息超时
死信队列
延迟队列
事务消息
惰性队列
优先级队列

课程内容

基础篇

什么是消息队列
RabbitMQ安装
RabbitMQ的各种工作模式



进阶篇

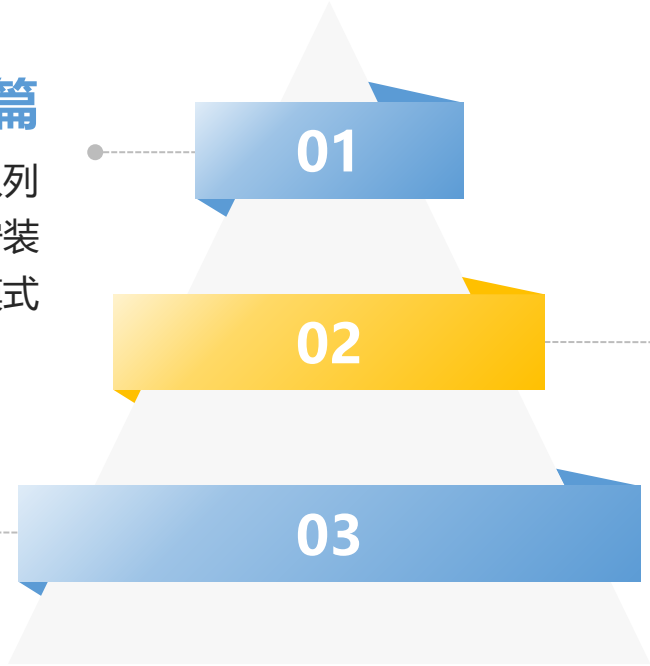
消息可靠性投递
消费端限流
消息超时
死信队列
延迟队列
事务消息
惰性队列
优先级队列

集群篇

课程内容

基础篇

什么是消息队列
RabbitMQ安装
RabbitMQ的各种工作模式



进阶篇

消息可靠性投递
消费端限流
消息超时
死信队列
延迟队列
事务消息
惰性队列
优先级队列

集群篇

原理与搭建
HA-Proxy负载均衡
Quorum队列(3.8)
Stream队列(3.9)
异地容灾机制

前置要求

前置要求

SpringBoot

SpringCloud

前置要求

SpringBoot

SpringCloud

docker

Maven™

前置要求

课件形式

理解

课件形式

理解

操作

Part 02
基础篇

什么是消息队列、RabbitMQ安装、RabbitMQ的各种工作模式

目录

1 为什么需要消息队列?

2 什么是消息队列?

3 RabbitMQ简介

4 RabbitMQ安装

5 HelloWorld

6 RabbitMQ工作模式

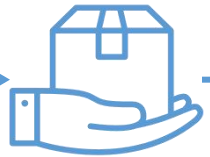
尚硅谷

1 为什么需要消息队列?

Why we need message queue?

同步

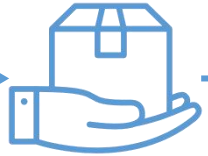
等待



开会

异步

无需等待



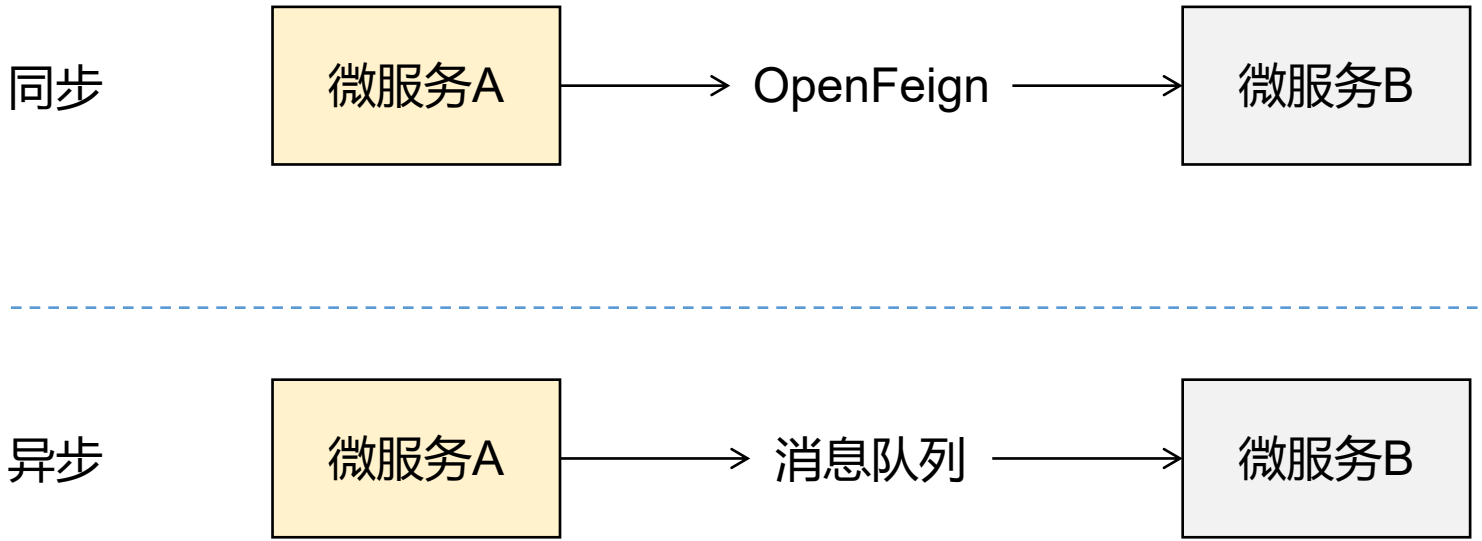
丰巢柜

开会

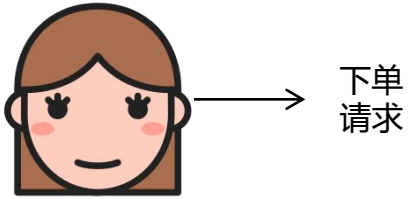
丰巢柜

散会

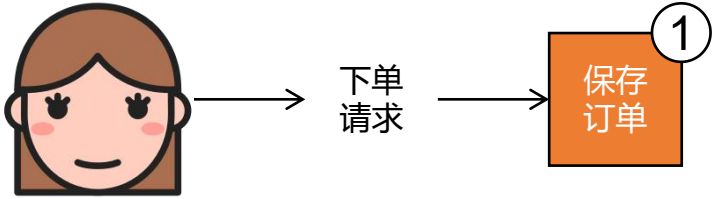
同步


同步
微服务A
OpenFeign
微服务B
异步
微服务A
消息队列
微服务B

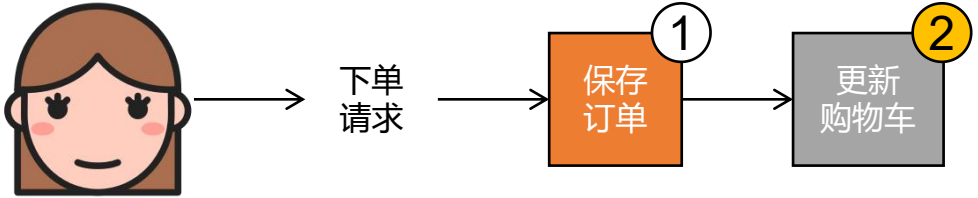
下单功能【同步】



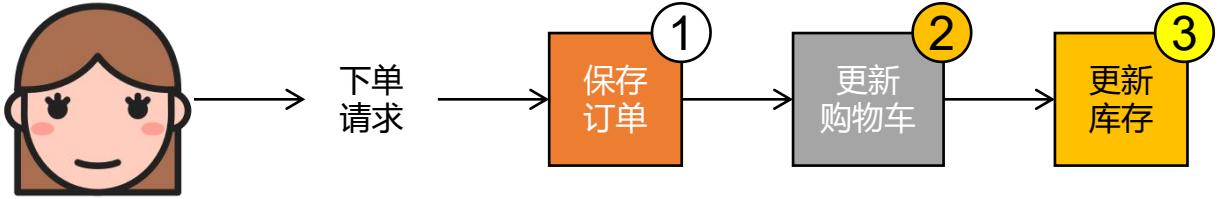
下单功能【同步】



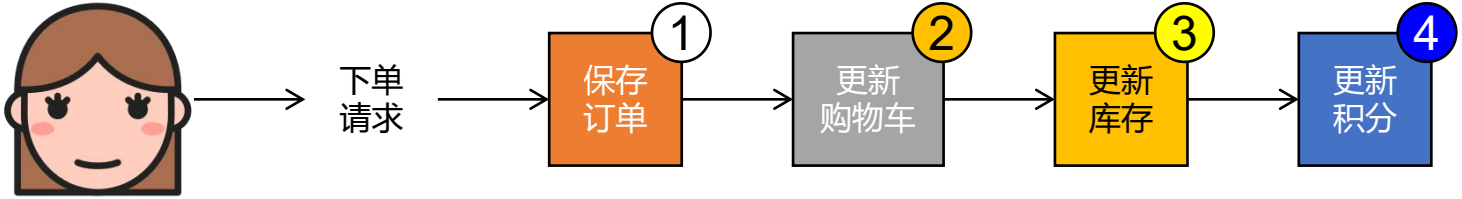
下单功能【同步】



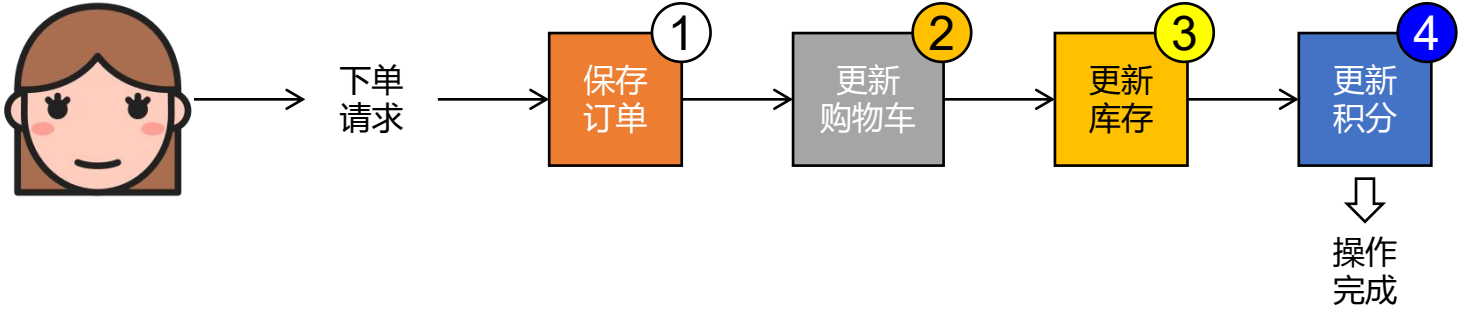
下单功能【同步】



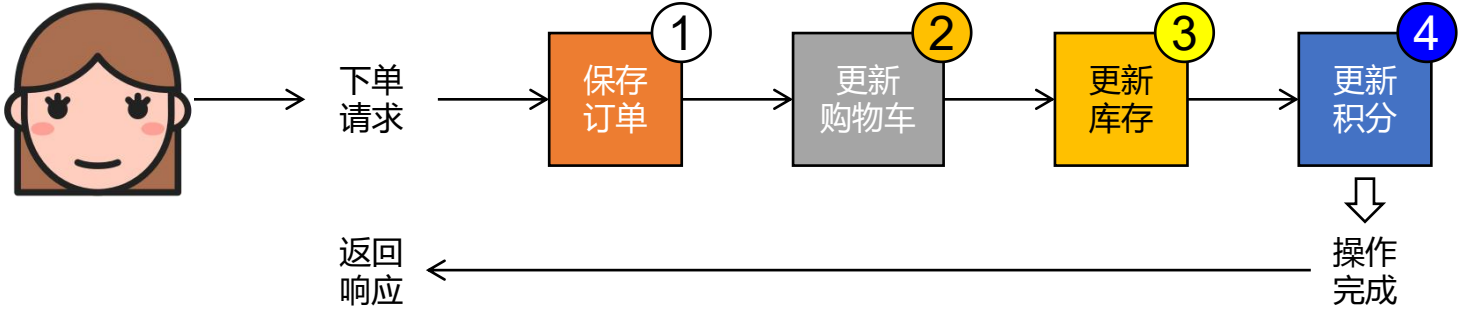
下单功能【同步】



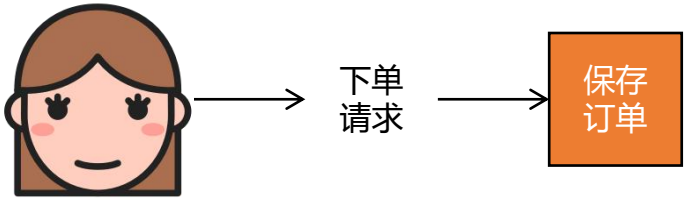
下单功能【同步】



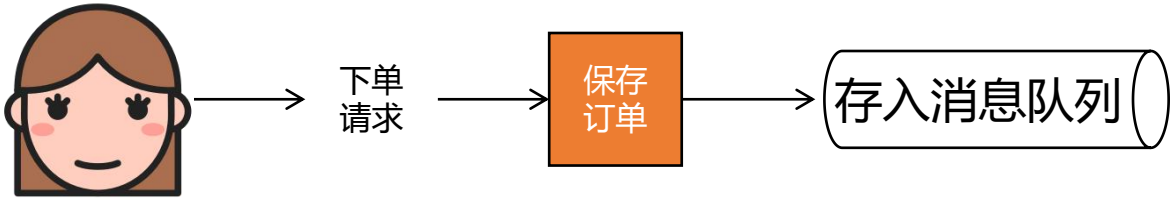
下单功能【同步】



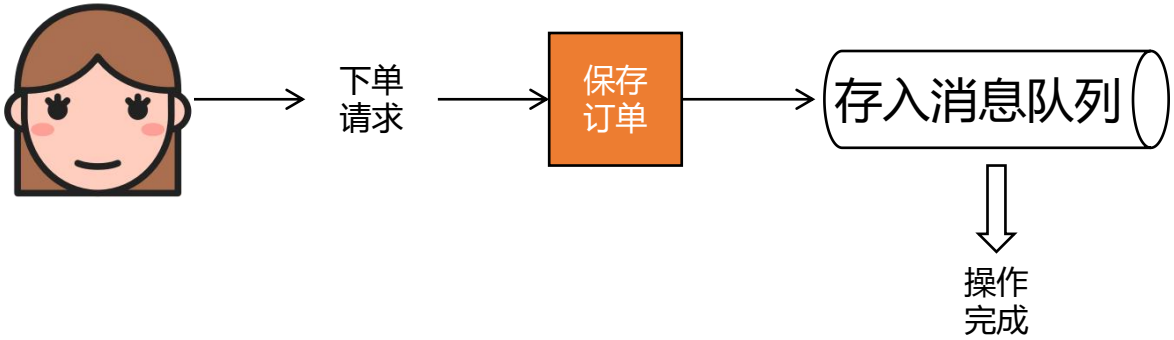
下单功能【异步】



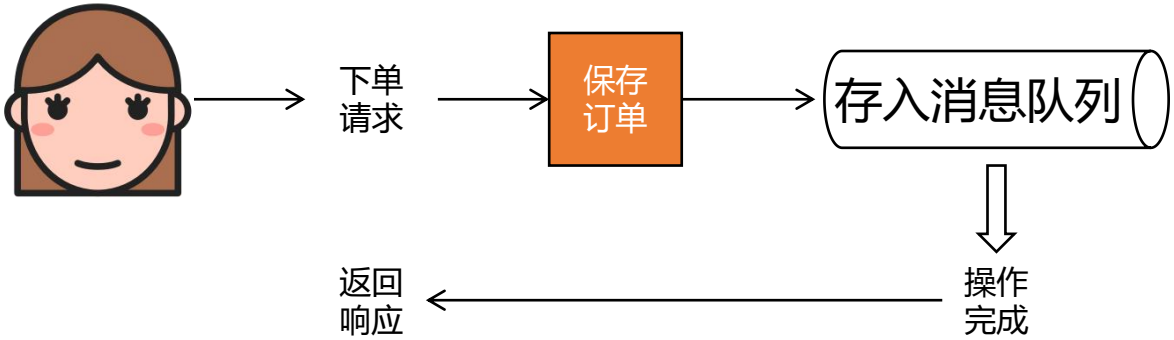
下单功能【异步】



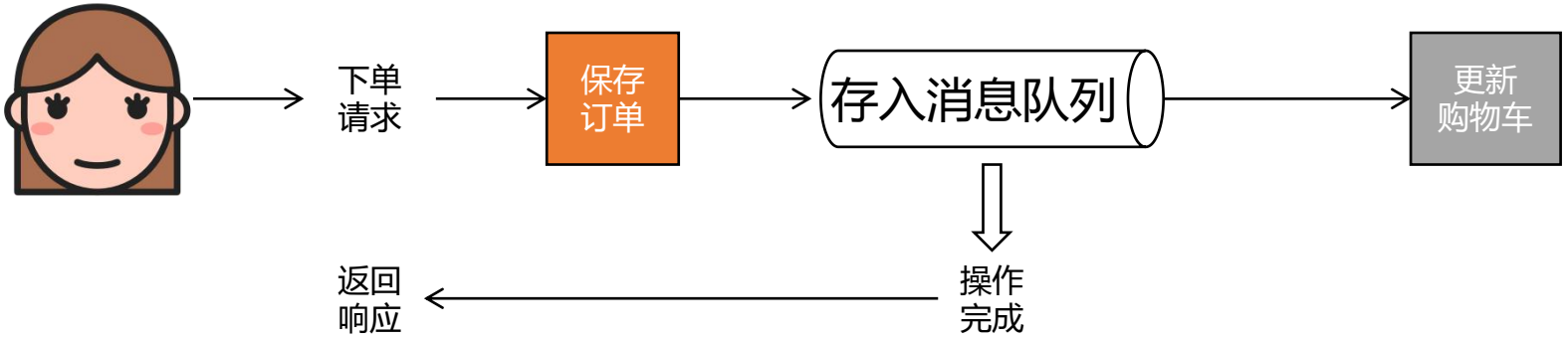
下单功能【异步】



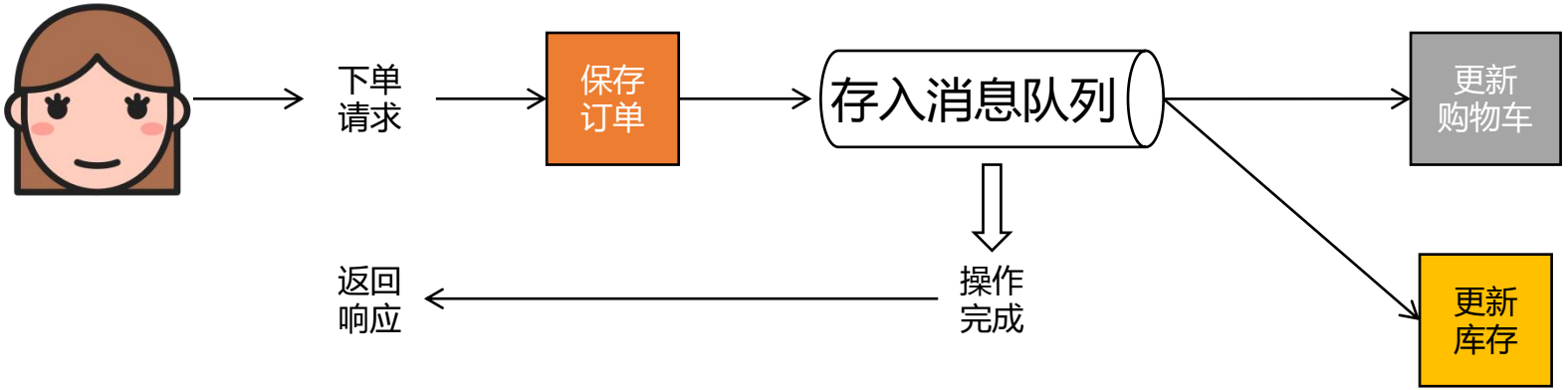
下单功能【异步】



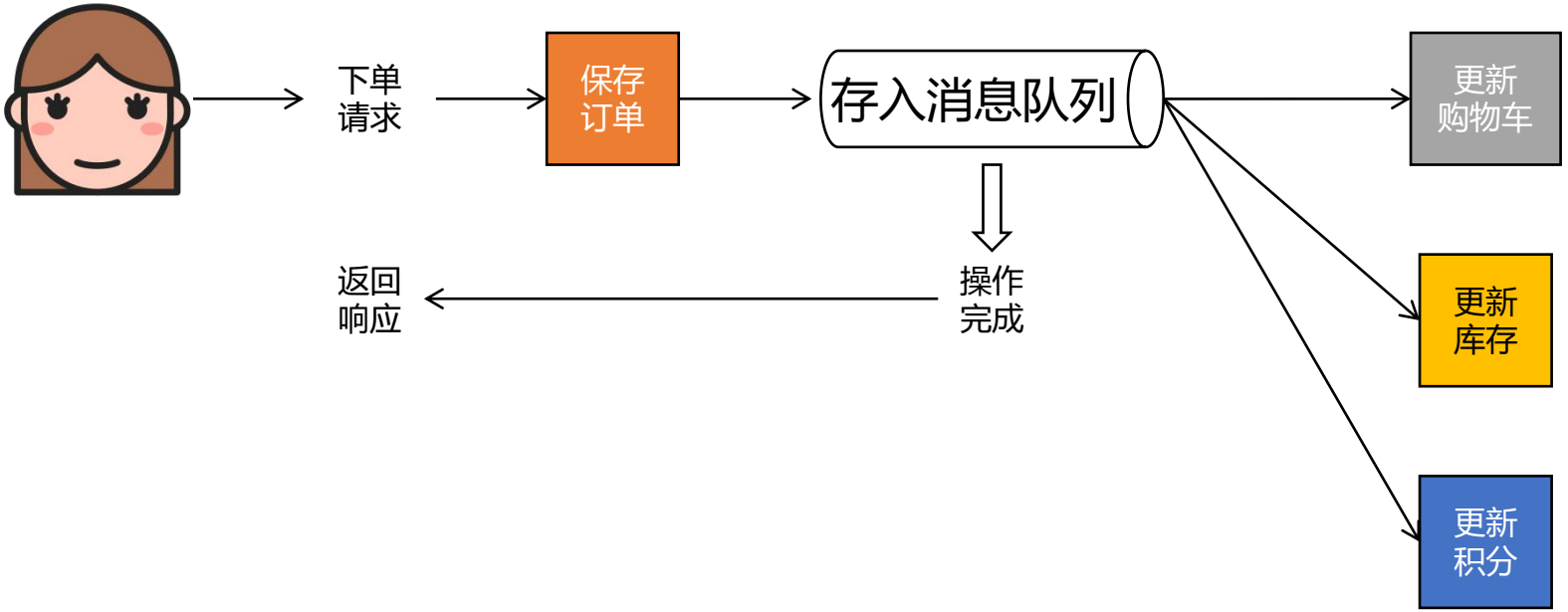
下单功能【异步】



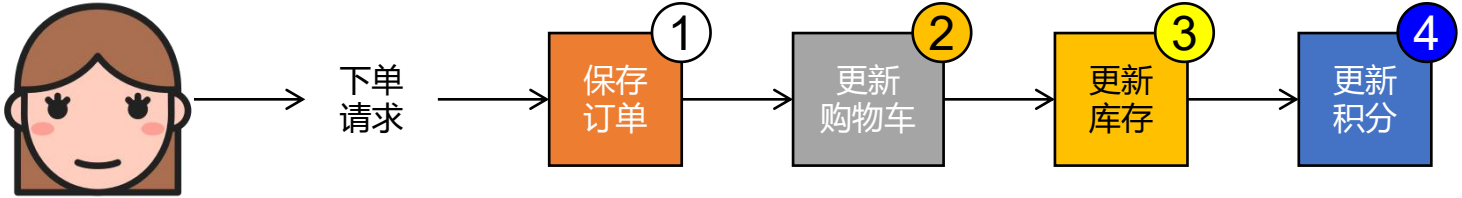
下单功能【异步】



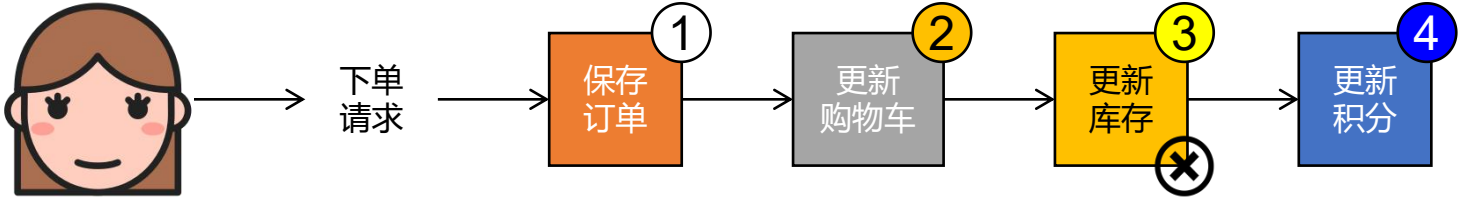
下单功能【异步】



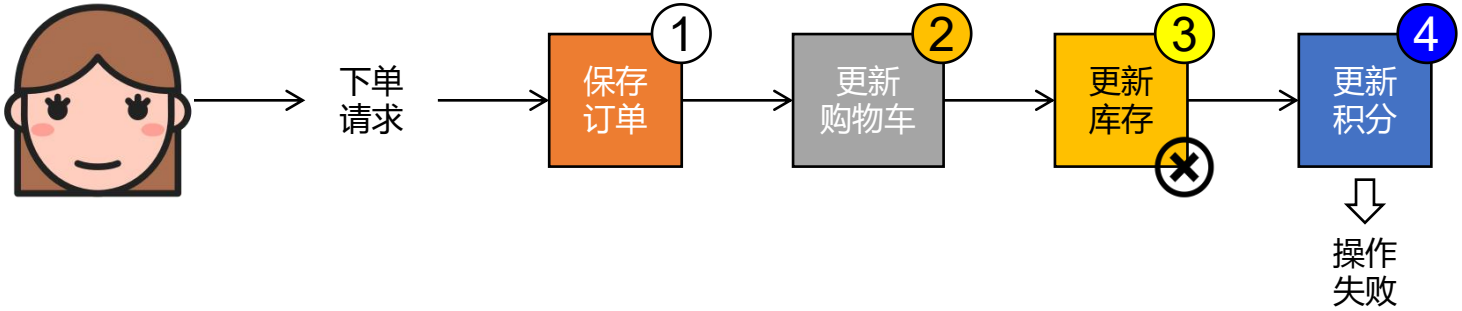
下单功能【同步】问题1：功能耦合度高



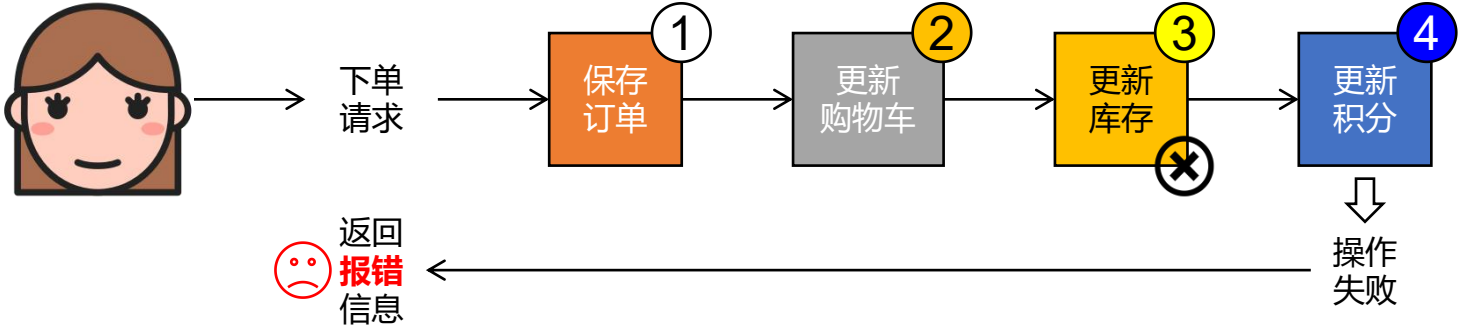
下单功能【同步】问题1：功能耦合度高



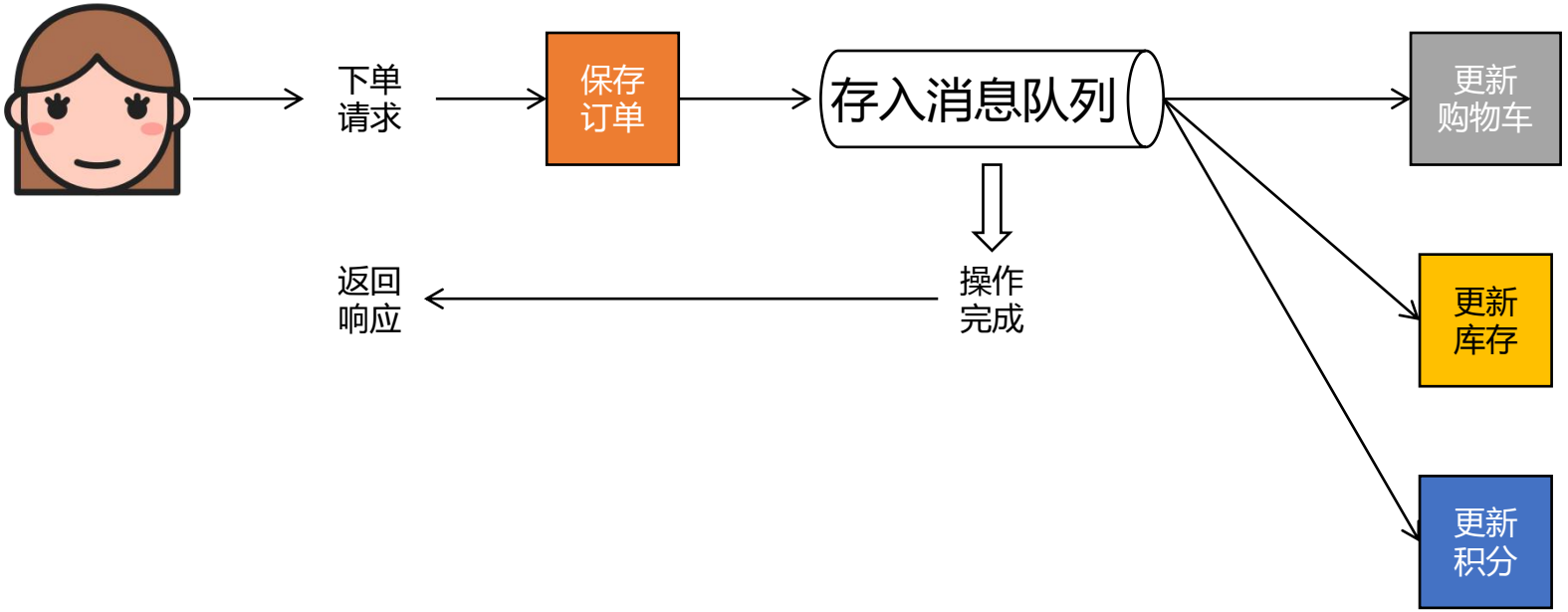
下单功能【同步】问题1：功能耦合度高



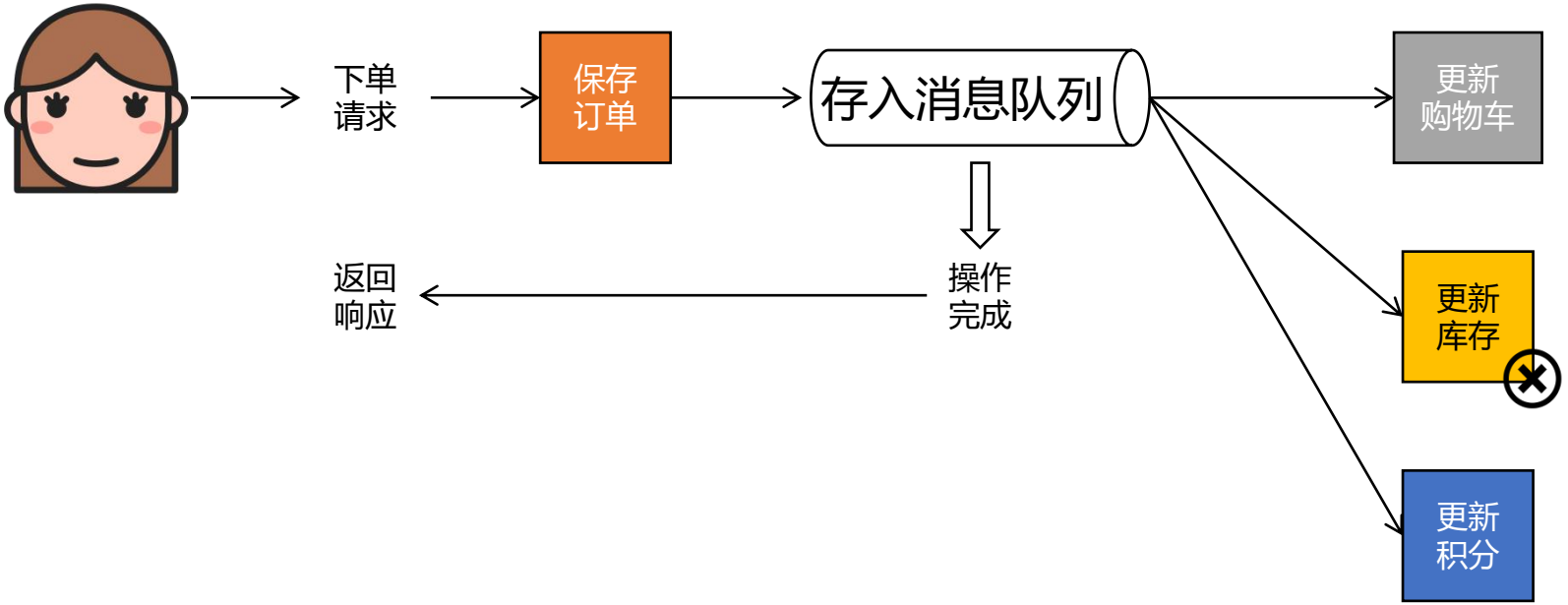
下单功能【同步】问题1：功能耦合度高



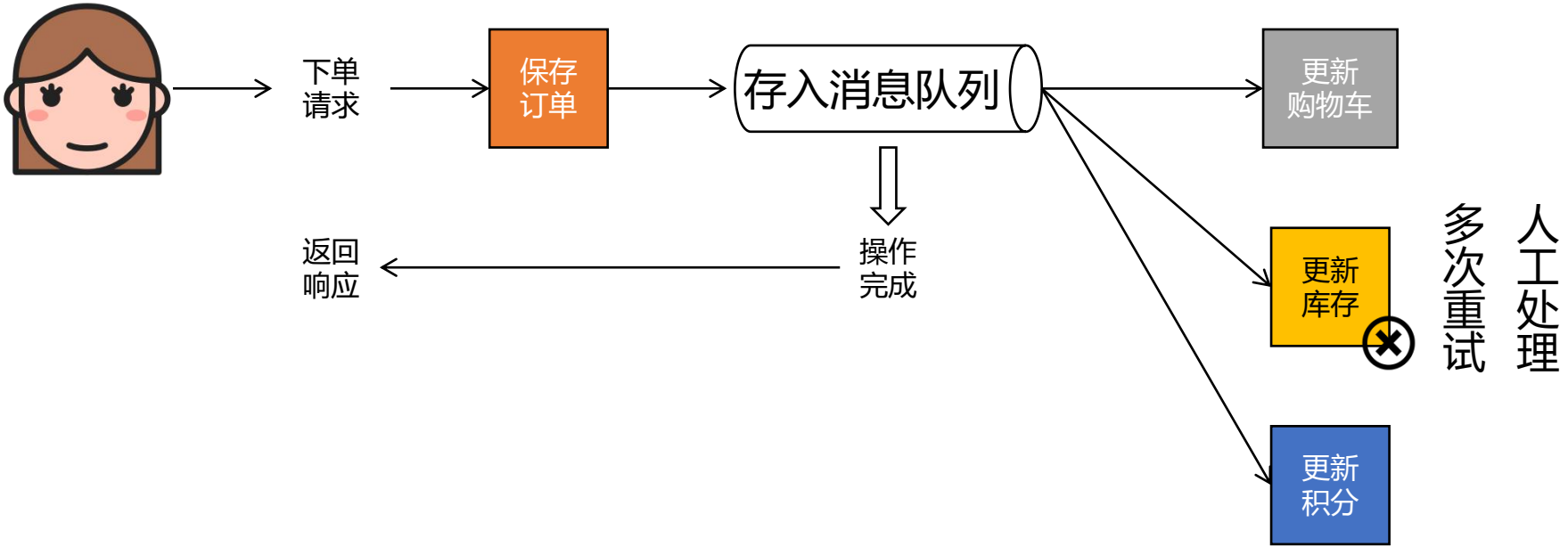
下单功能【异步】 功能解耦



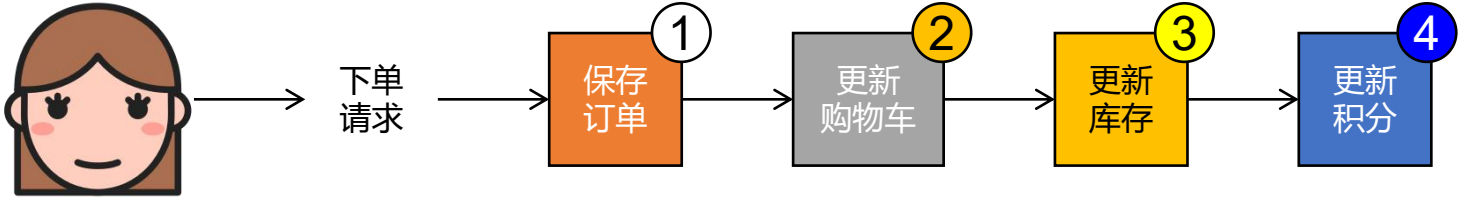
下单功能【异步】 功能解耦



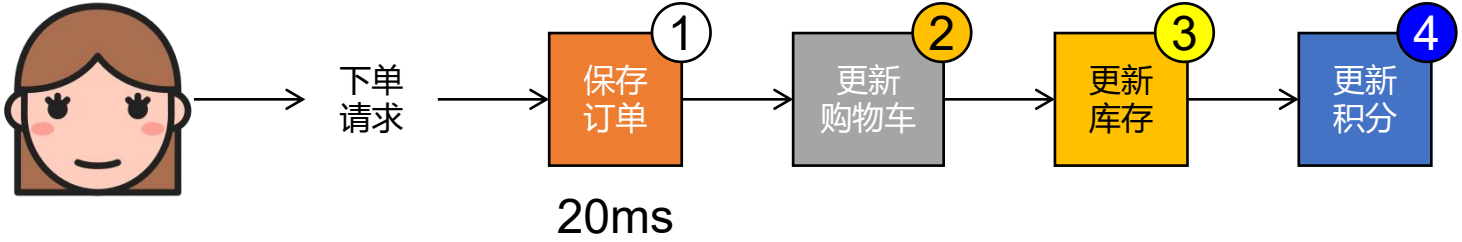
下单功能【异步】 功能解耦



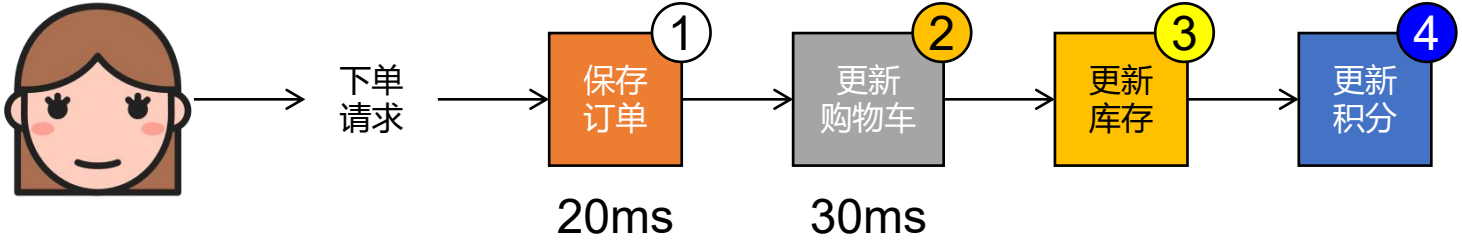
下单功能【同步】问题2：响应时间长



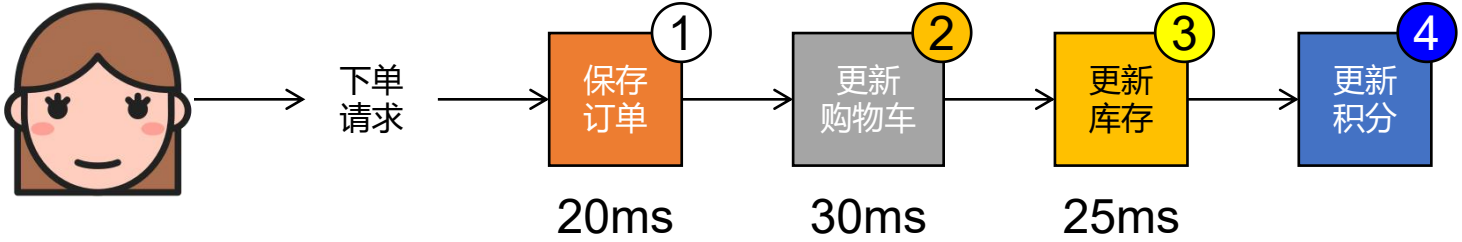
下单功能【同步】问题2：响应时间长



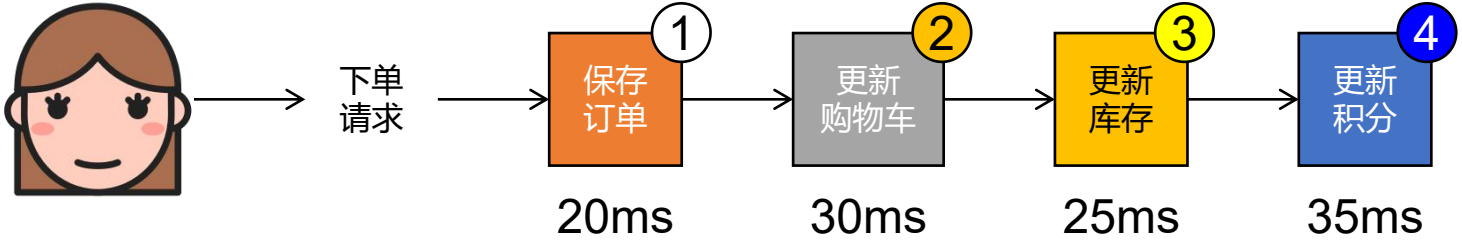
下单功能【同步】问题2：响应时间长



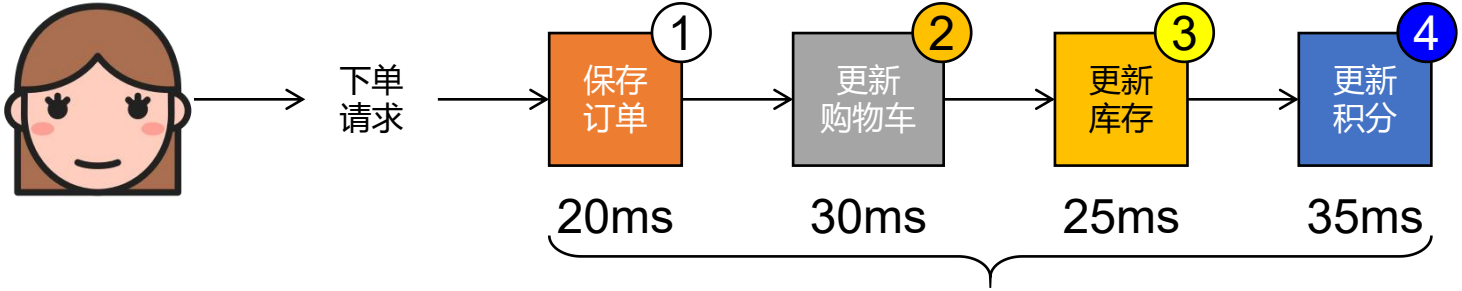
下单功能【同步】问题2：响应时间长



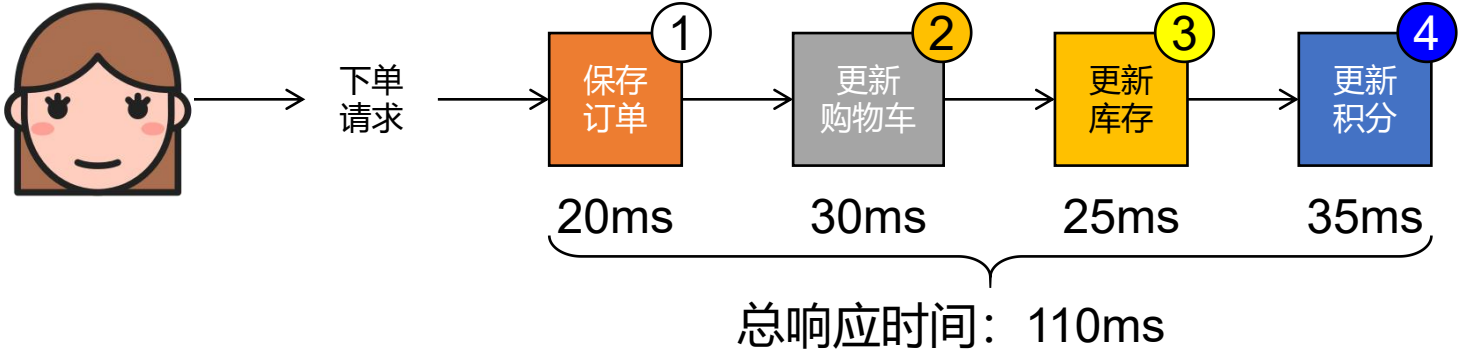
下单功能【同步】问题2：响应时间长



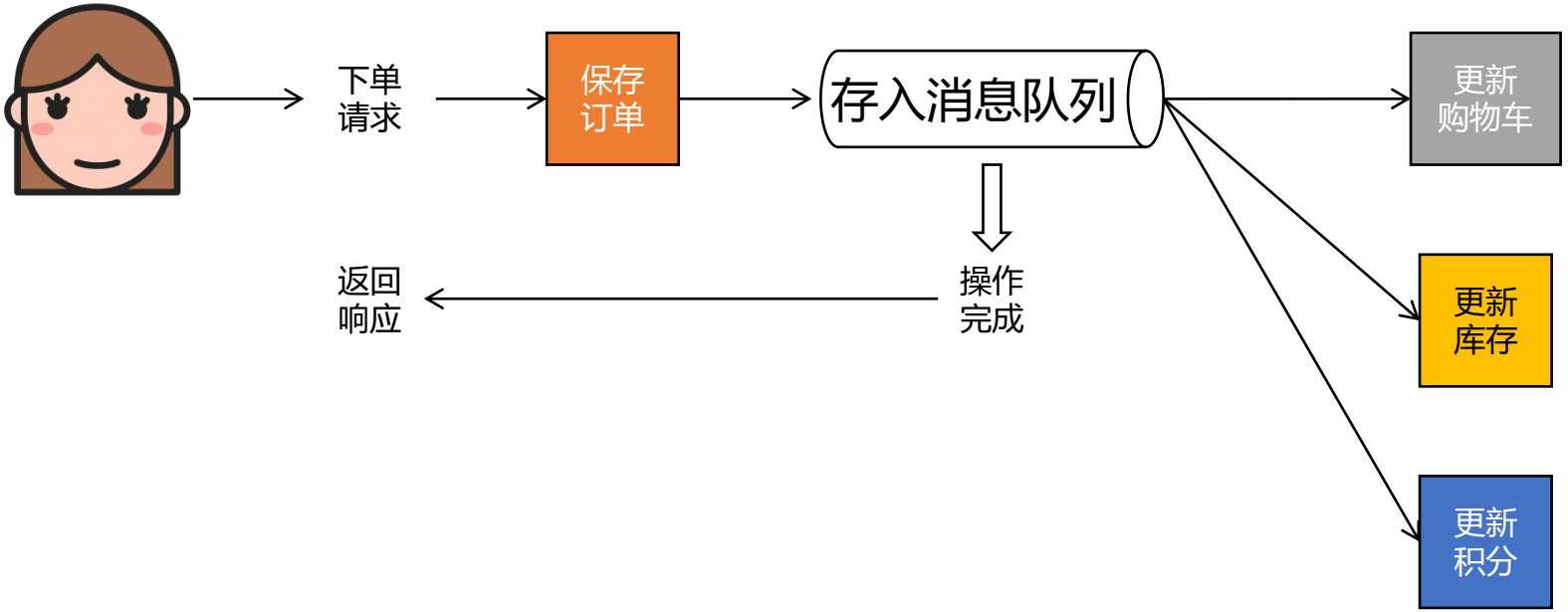
下单功能【同步】问题2：响应时间长



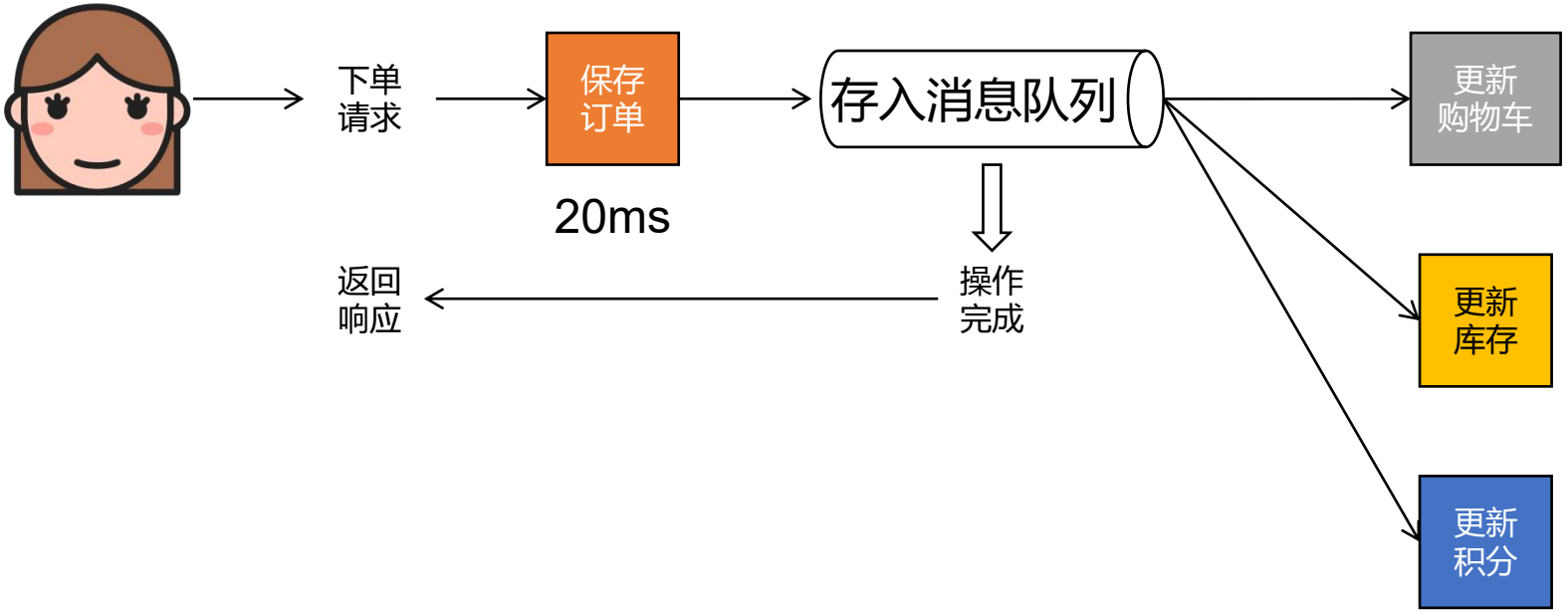
下单功能【同步】问题2：响应时间长



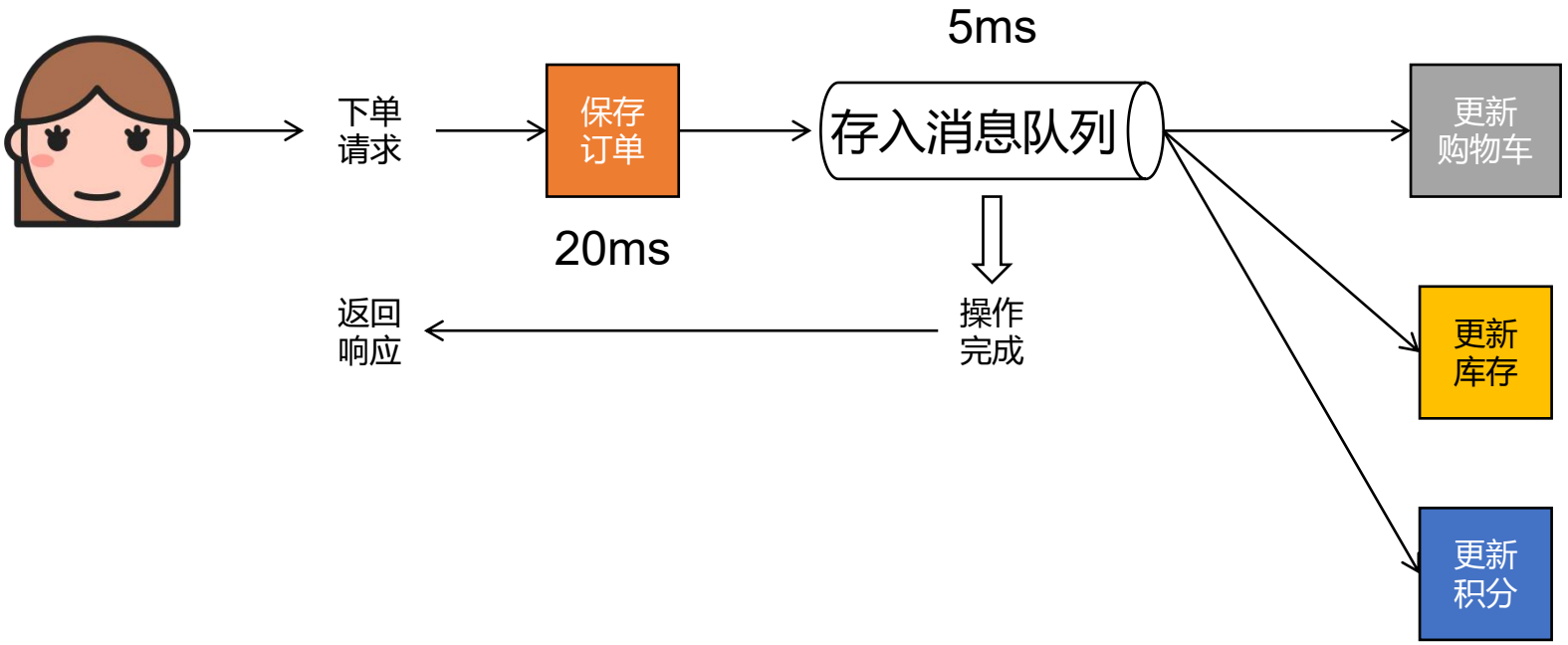
下单功能【异步】 快速响应



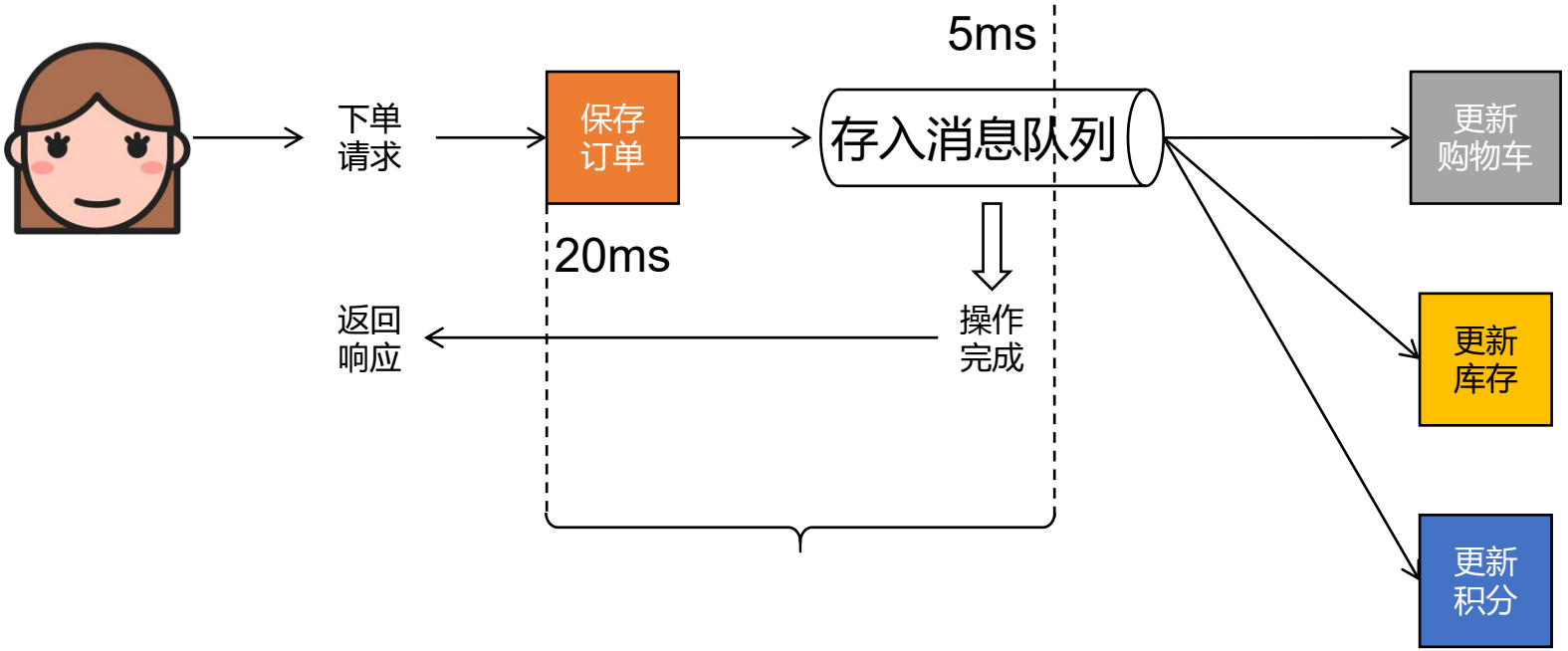
下单功能【异步】 快速响应



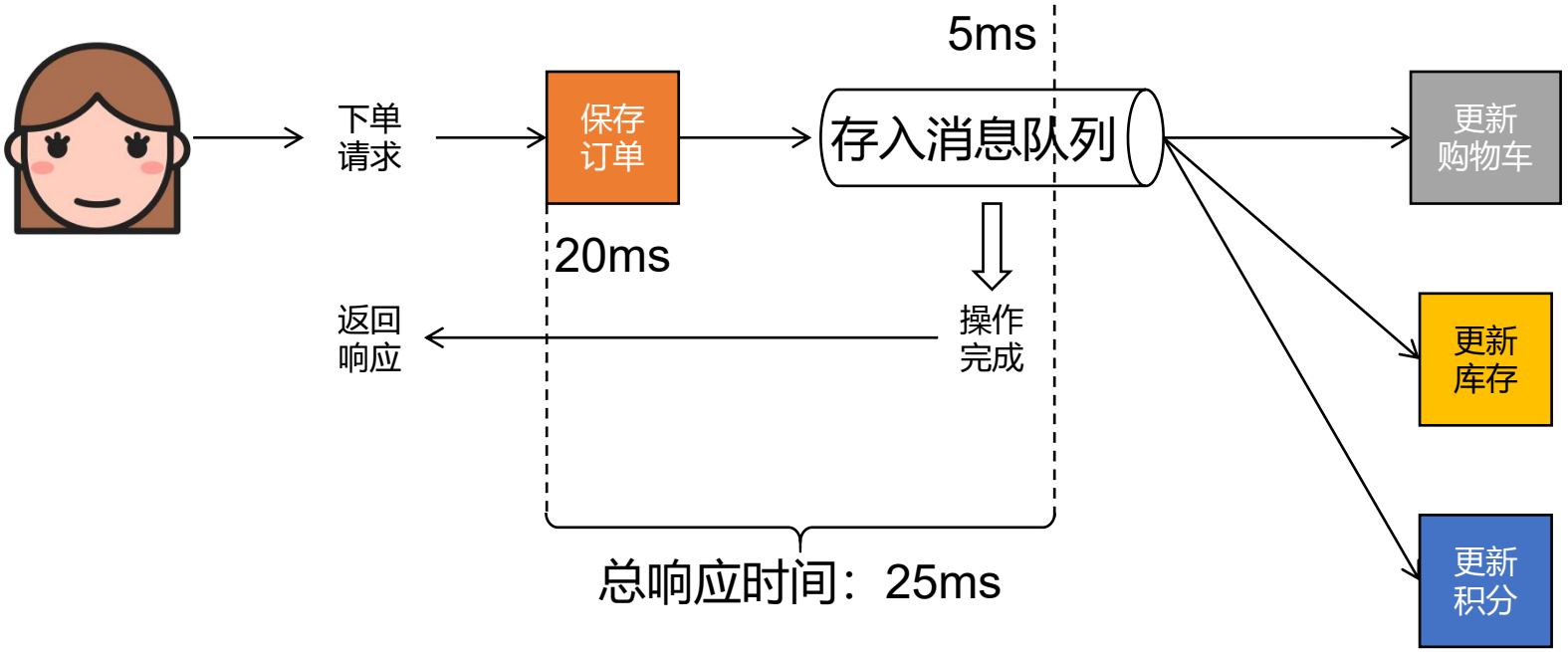
下单功能【异步】 快速响应



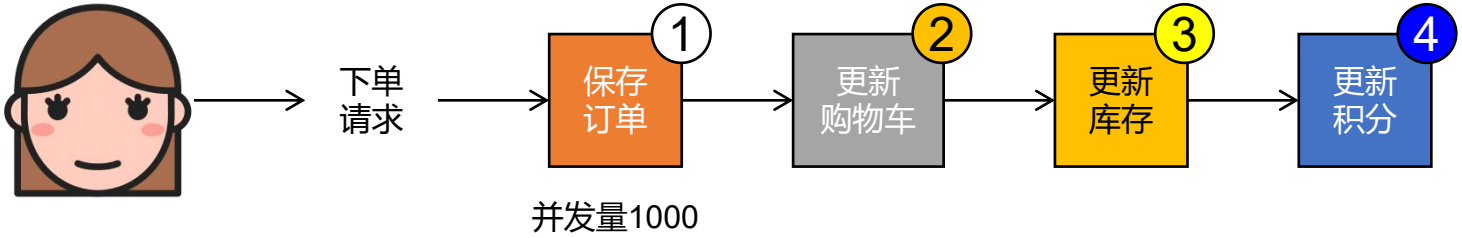
下单功能【异步】 快速响应



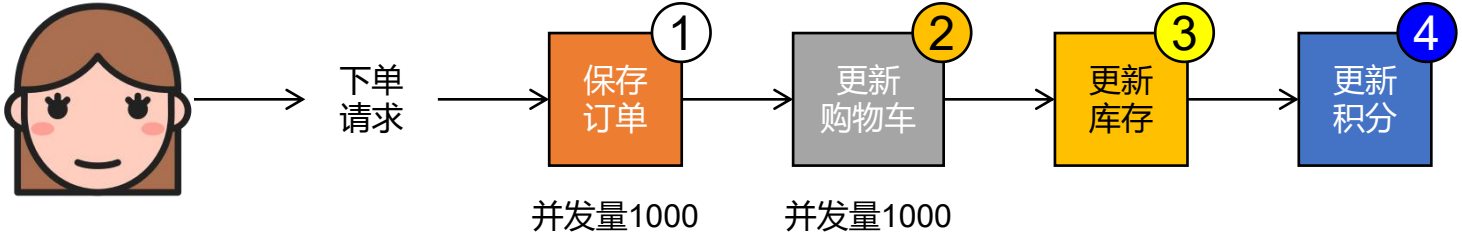
下单功能【异步】 快速响应


5ms
下单
请求
保存
订单
存入消息队列
更新
购物车
20ms
返回
响应
操作
完成
更新
库存
更新
积分
总响应时间：25ms

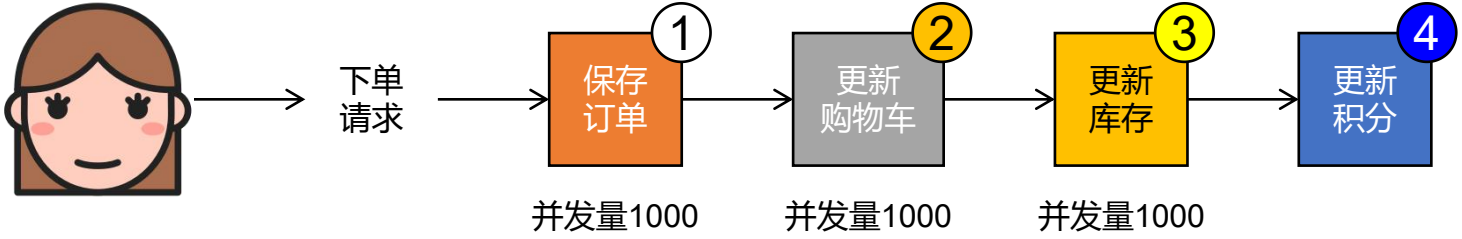
下单功能【同步】问题3：并发压力传递



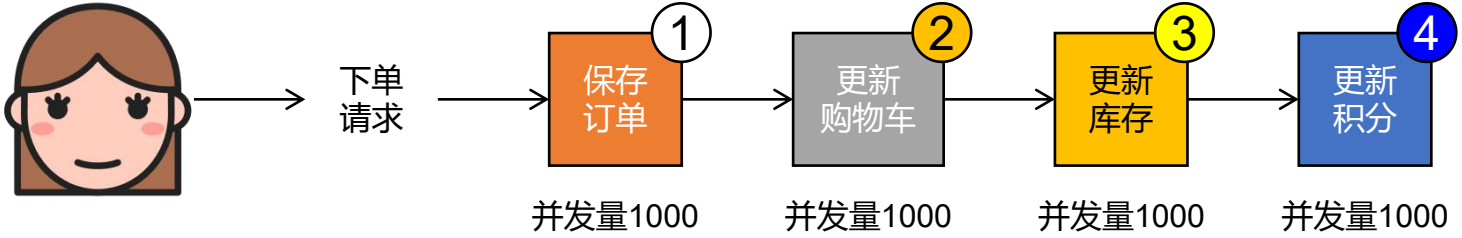
下单功能【同步】问题3：并发压力传递



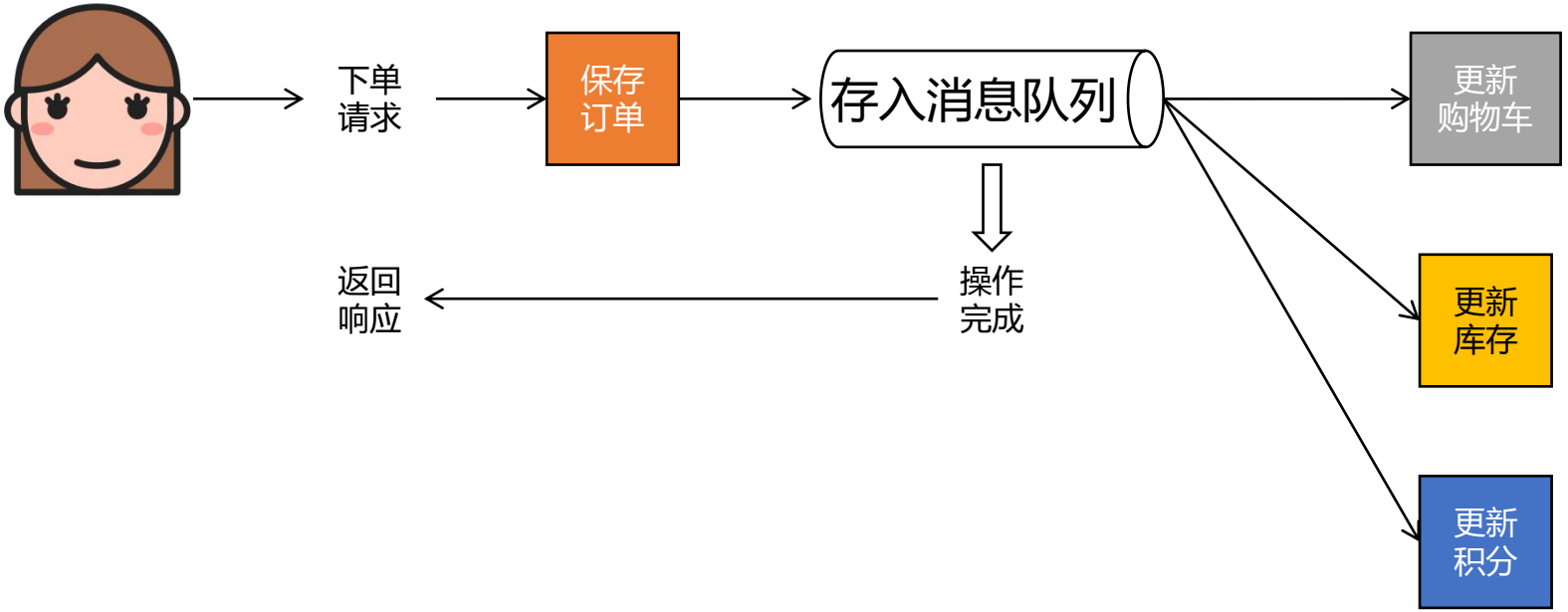
下单功能【同步】问题3：并发压力传递



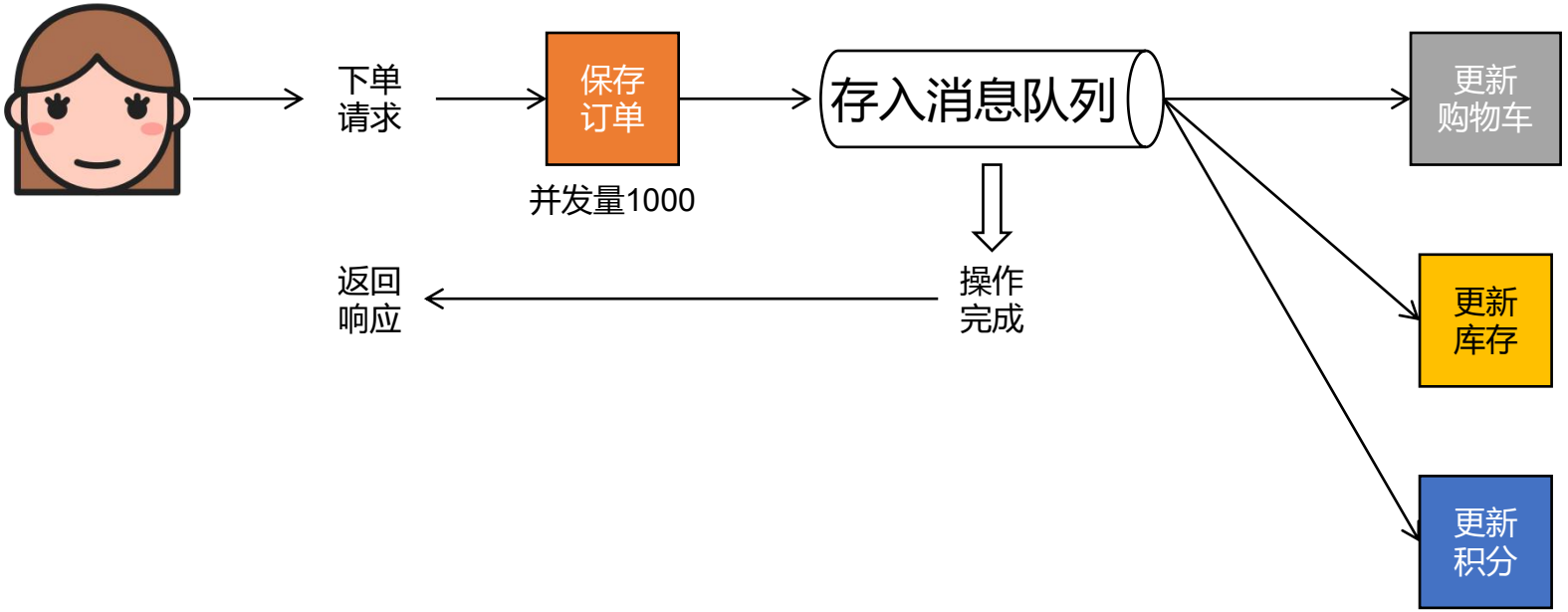
下单功能【同步】问题3：并发压力传递



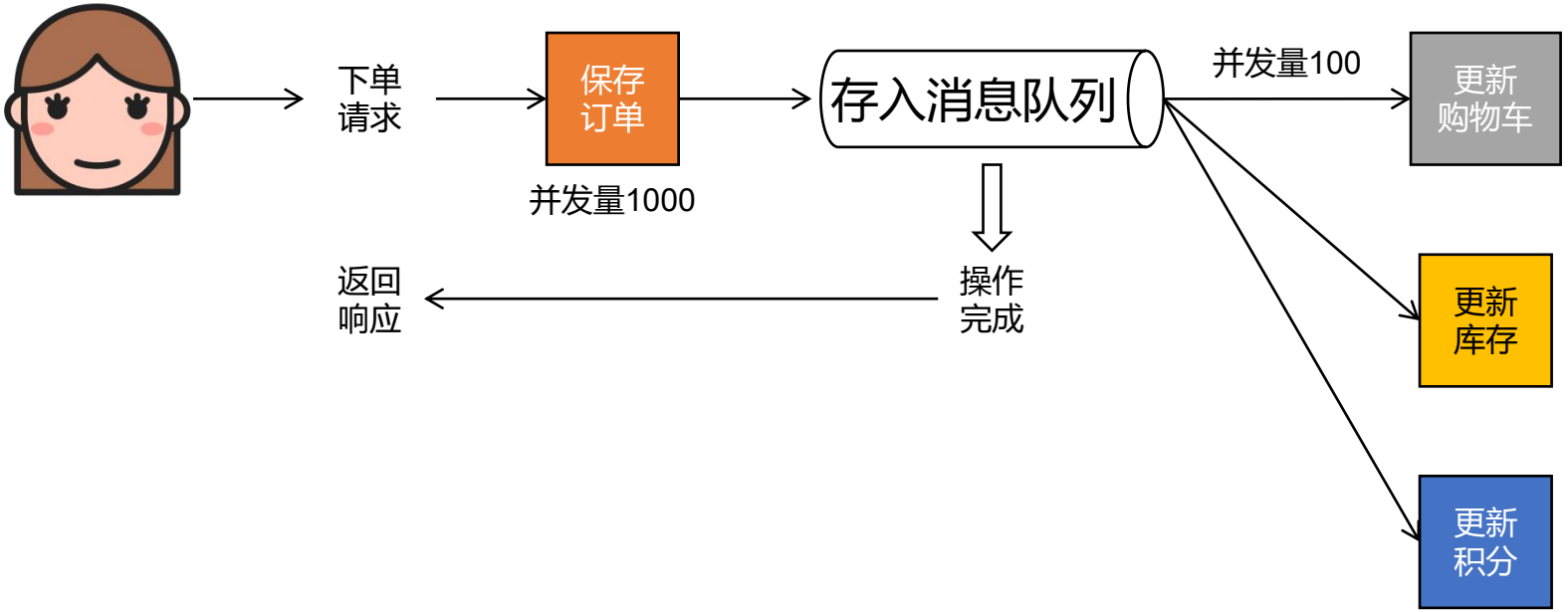
下单功能【异步】 削峰限流



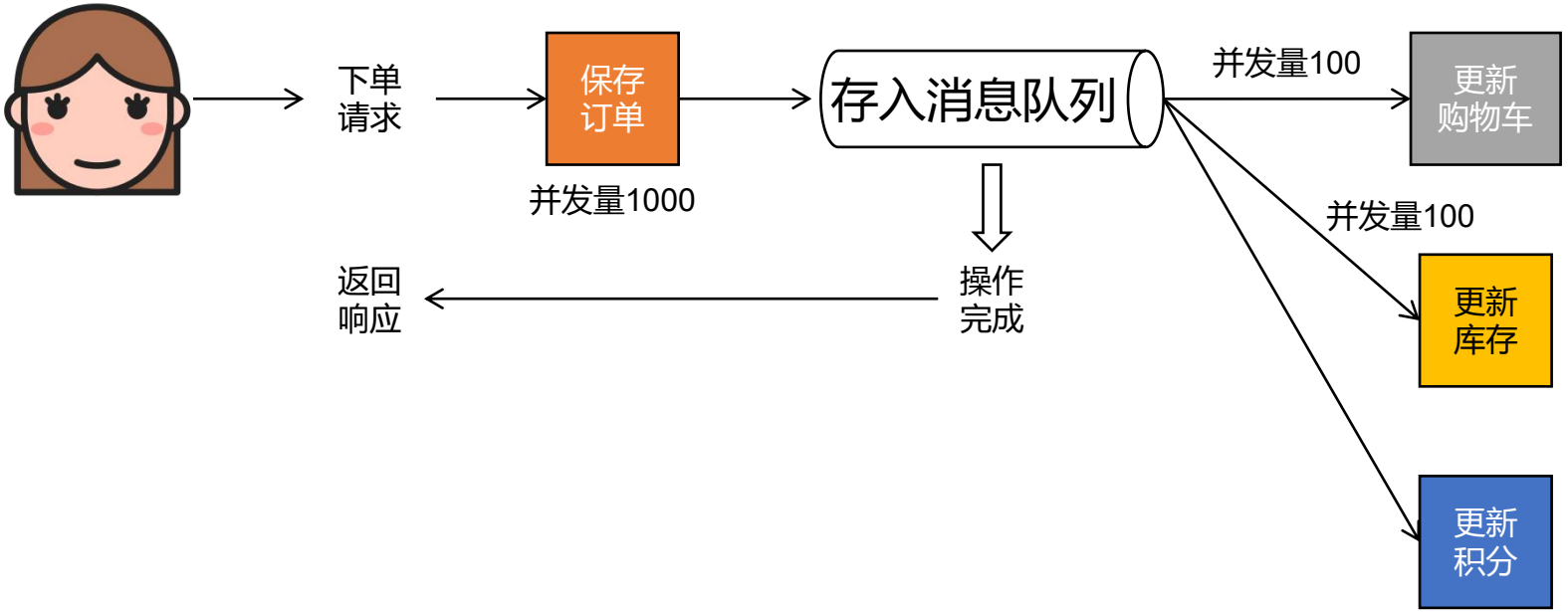
下单功能【异步】 削峰限流



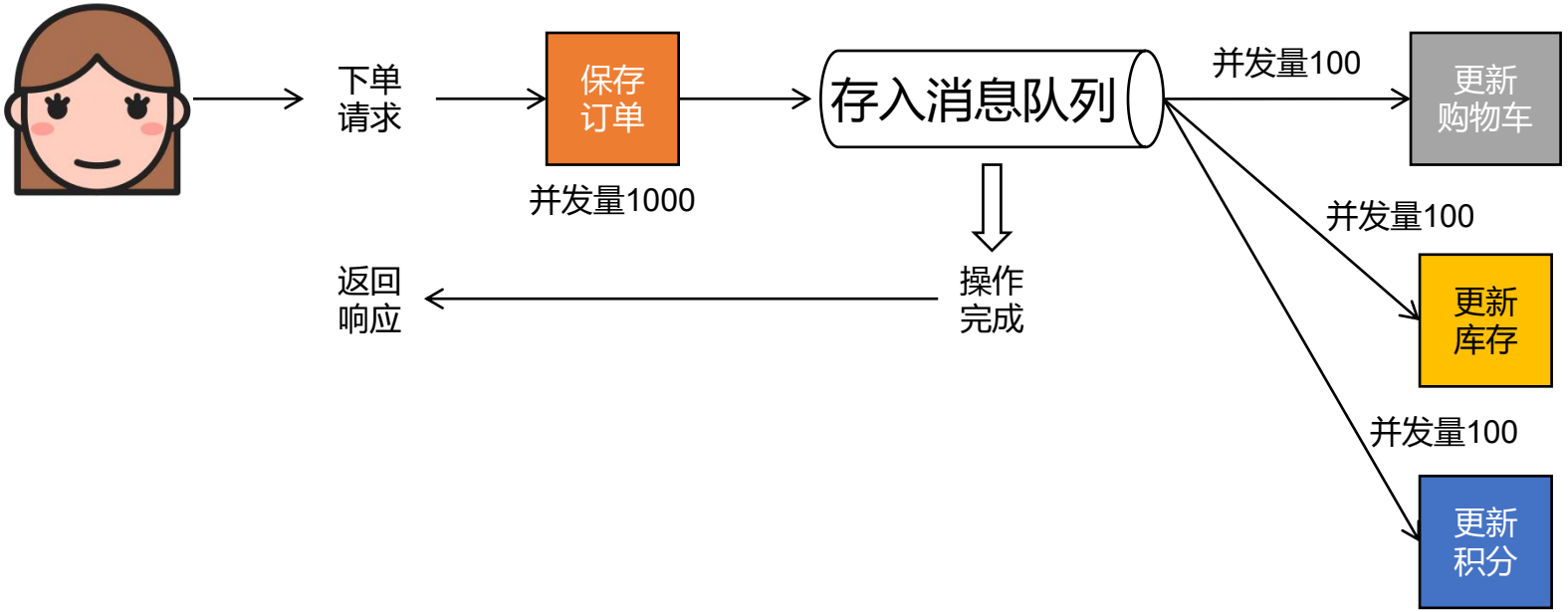
下单功能【异步】 削峰限流



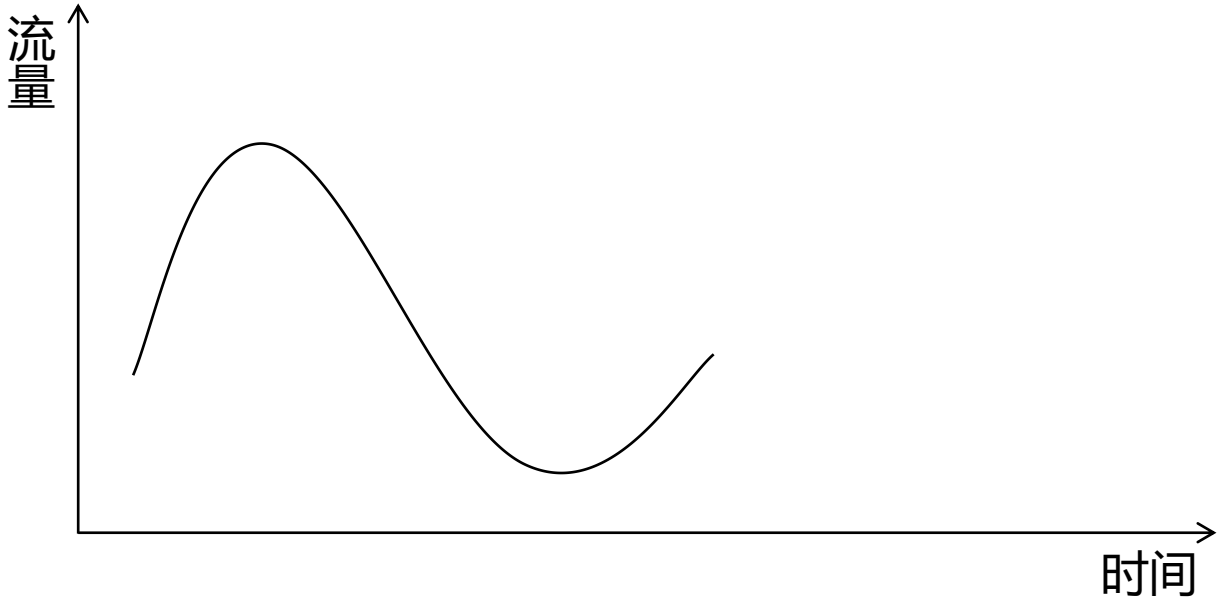
下单功能【异步】 削峰限流


下单
请求
保存
订单
并发量1000
存入消息队列
并发量100
更新
购物车
并发量100
更新
库存
更新
积分
操作
完成
返回
响应

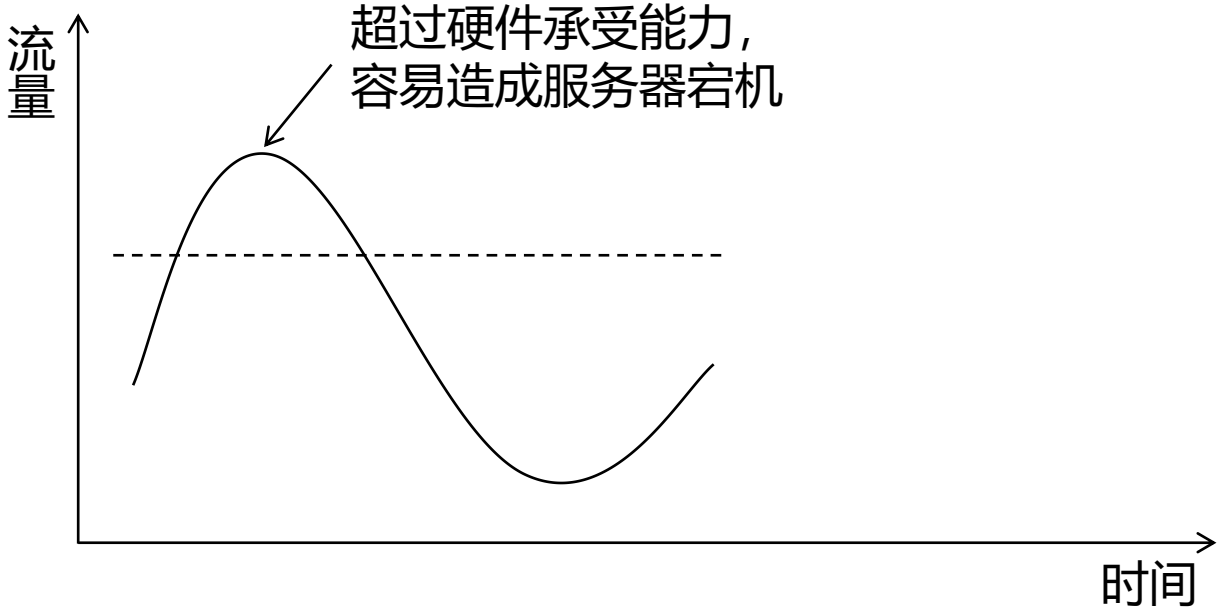
下单功能【异步】 削峰限流


下单
请求
保存
订单
并发量1000
存入消息队列
并发量100
更新
购物车
并发量100
更新
库存
并发量100
更新
积分
操作
完成
返回
响应

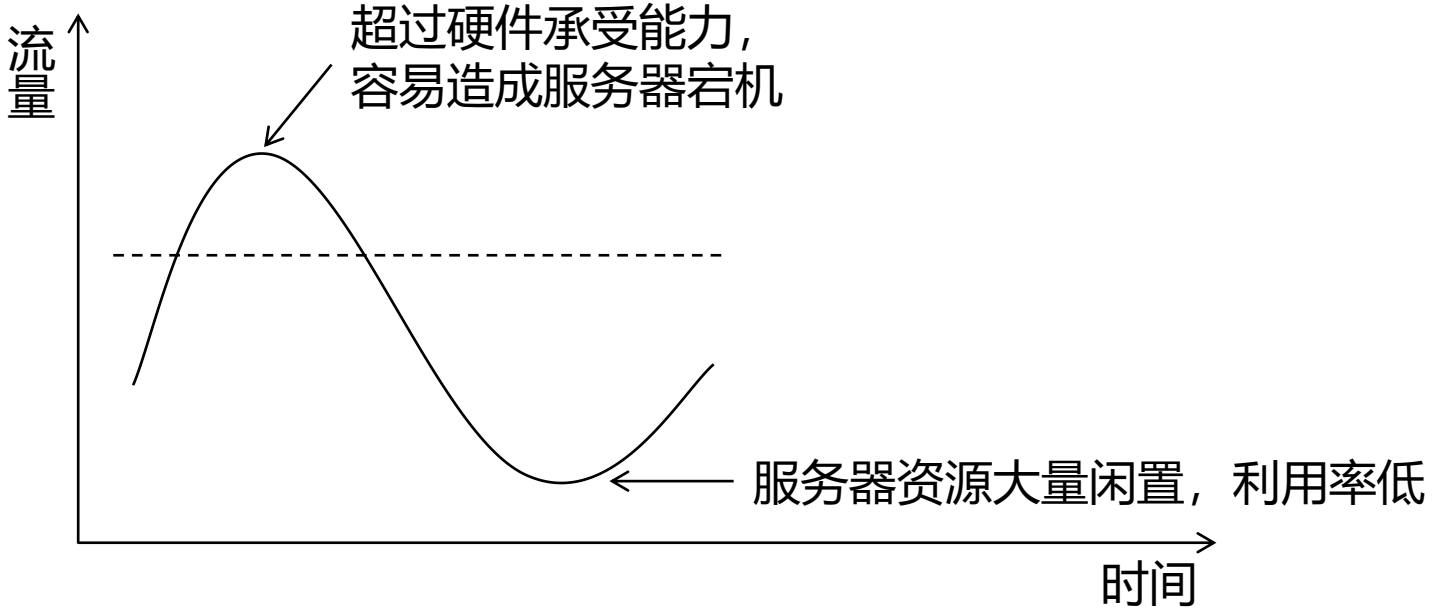
流量起伏过大的危害



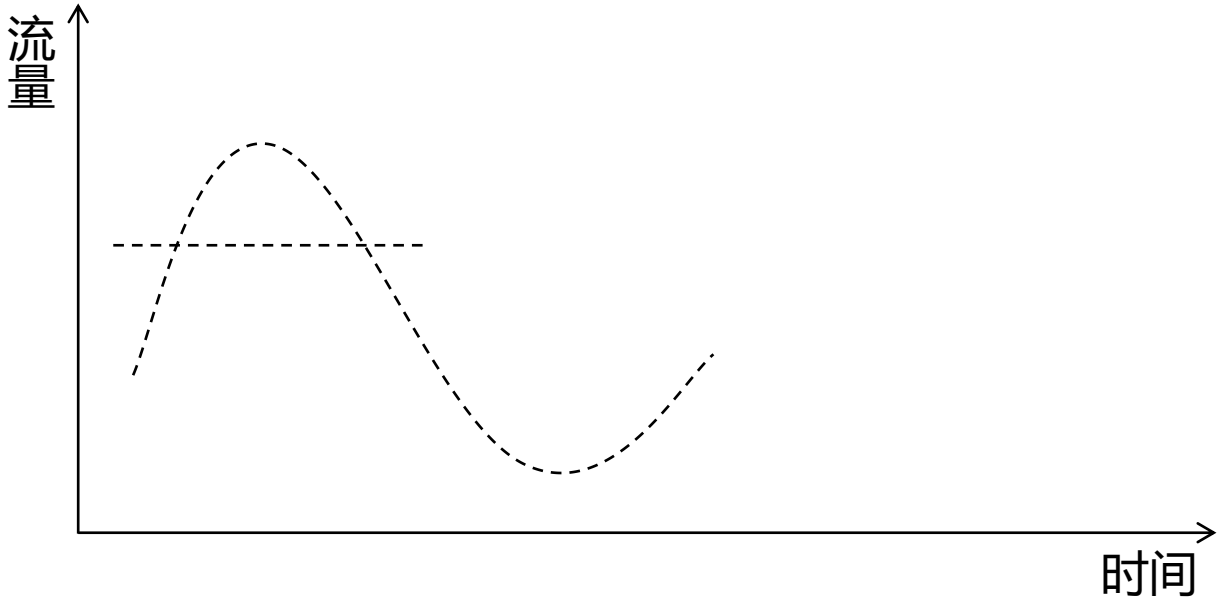
流量起伏过大的危害



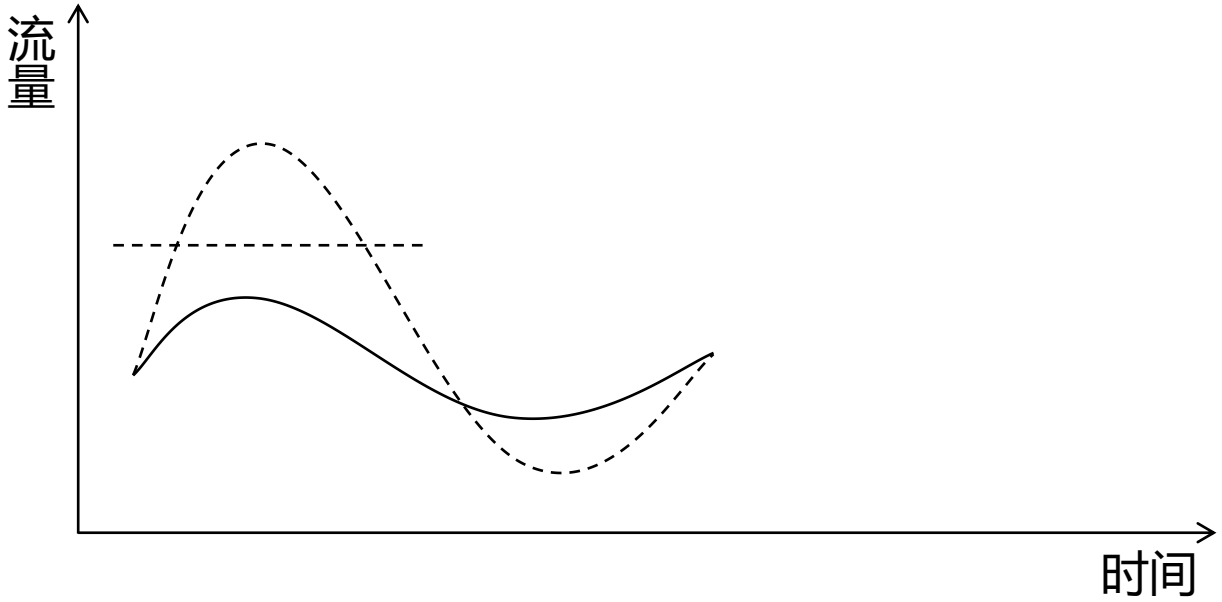
流量起伏过大的危害



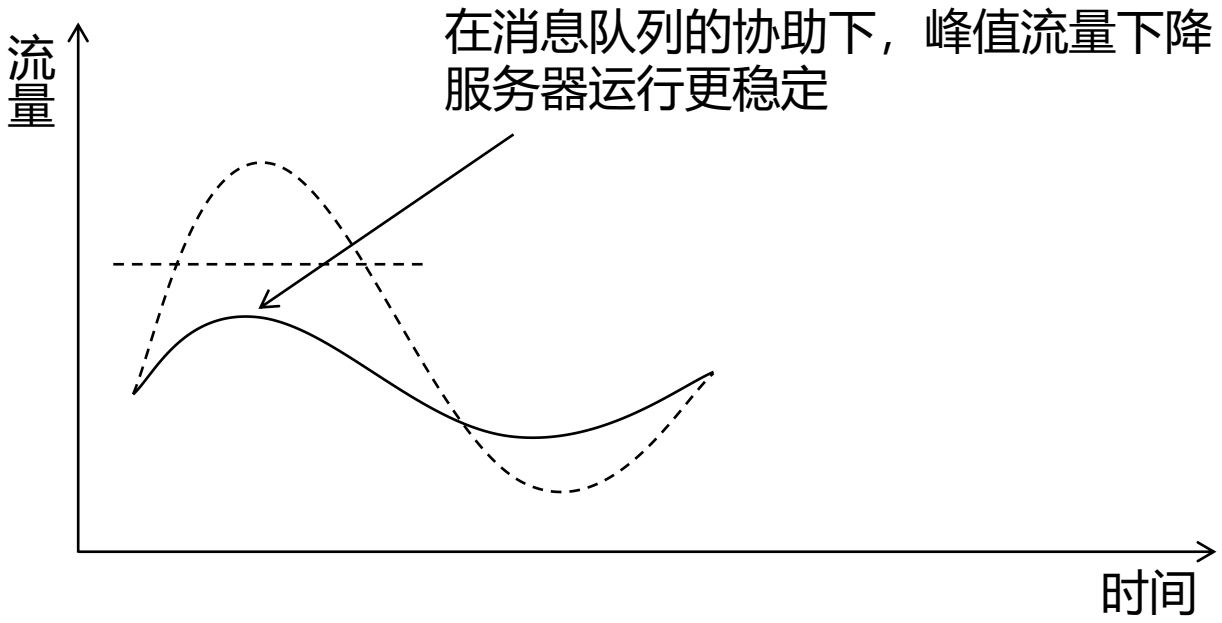
削峰限流的好处



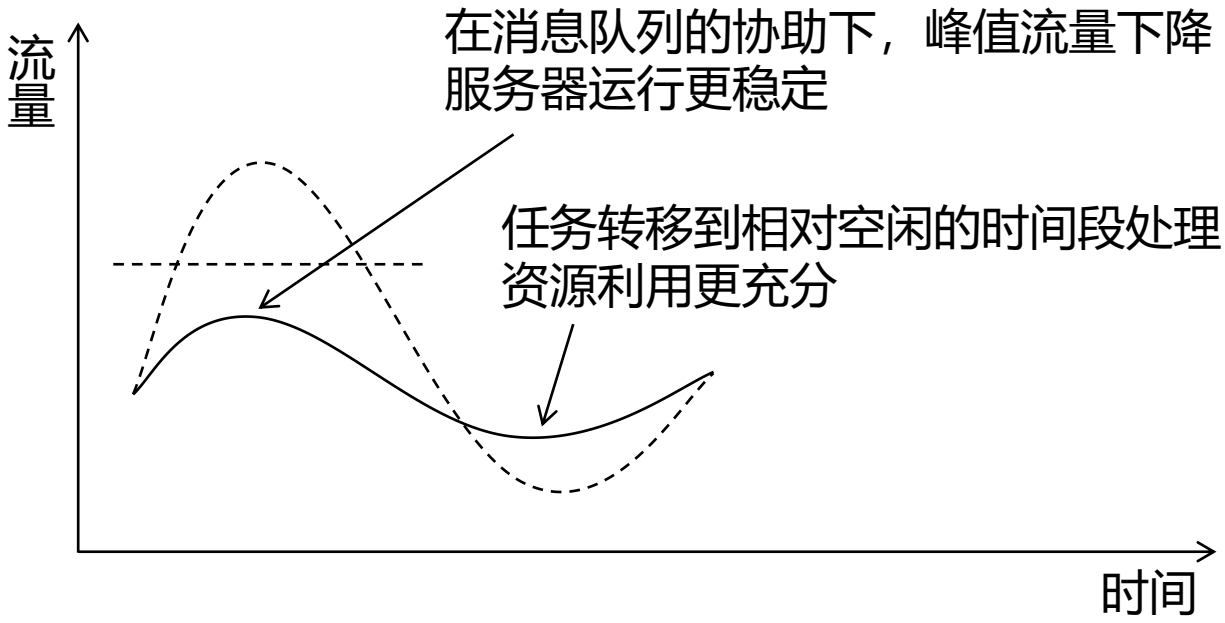
削峰限流的好处



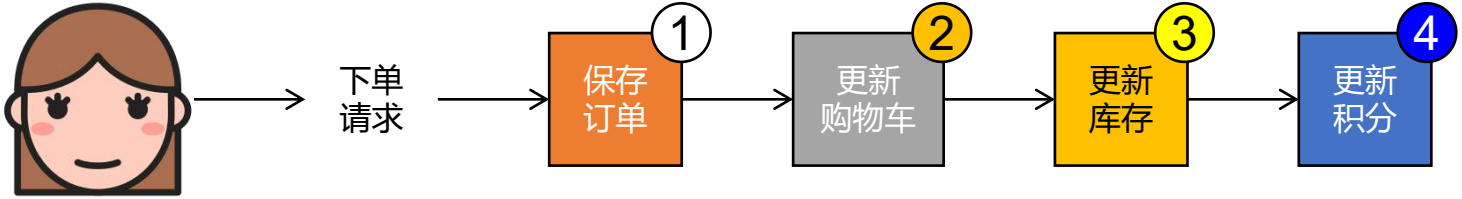
削峰限流的好处



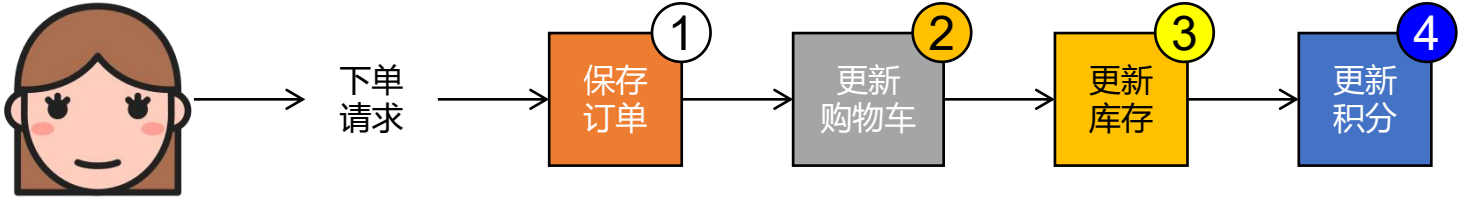
削峰限流的好处



下单功能【同步】问题4：系统结构弹性不足

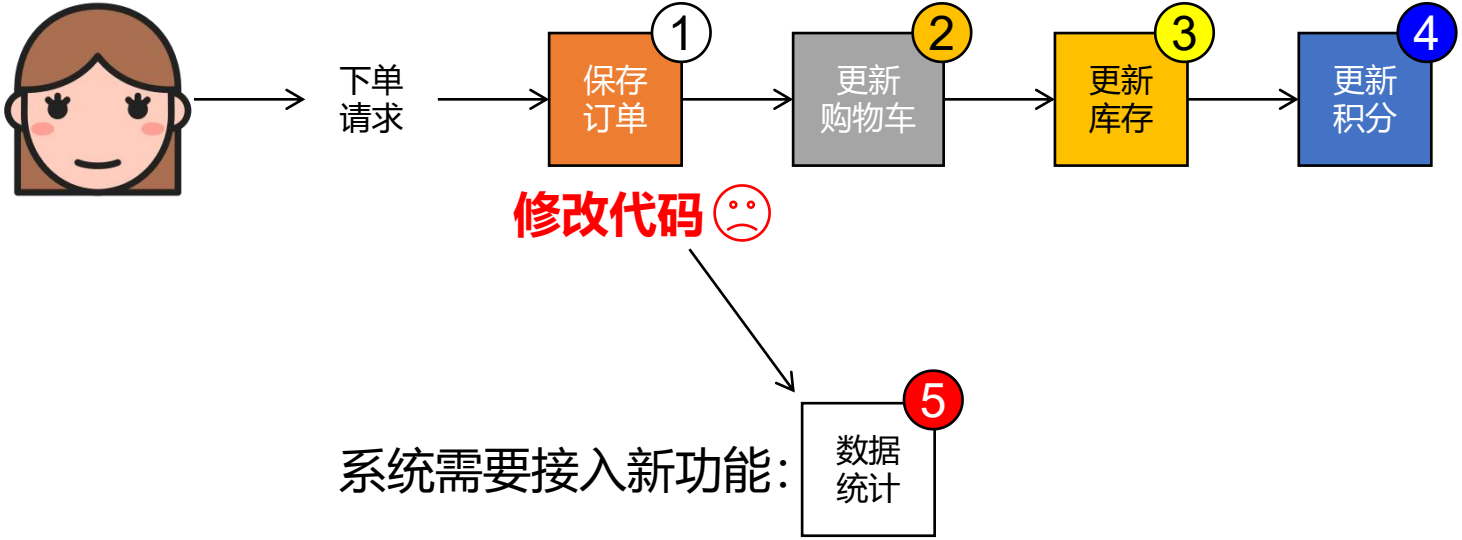


下单功能【同步】问题4：系统结构弹性不足

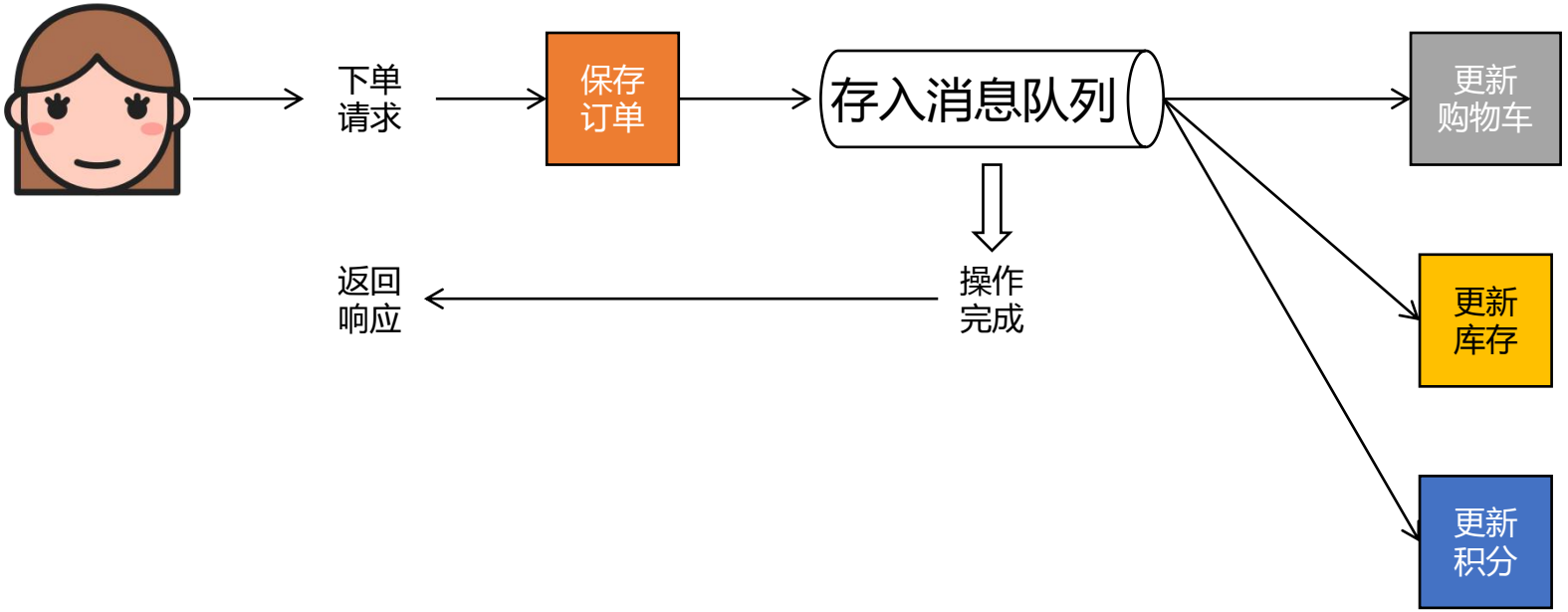


系统需要接入新功能：

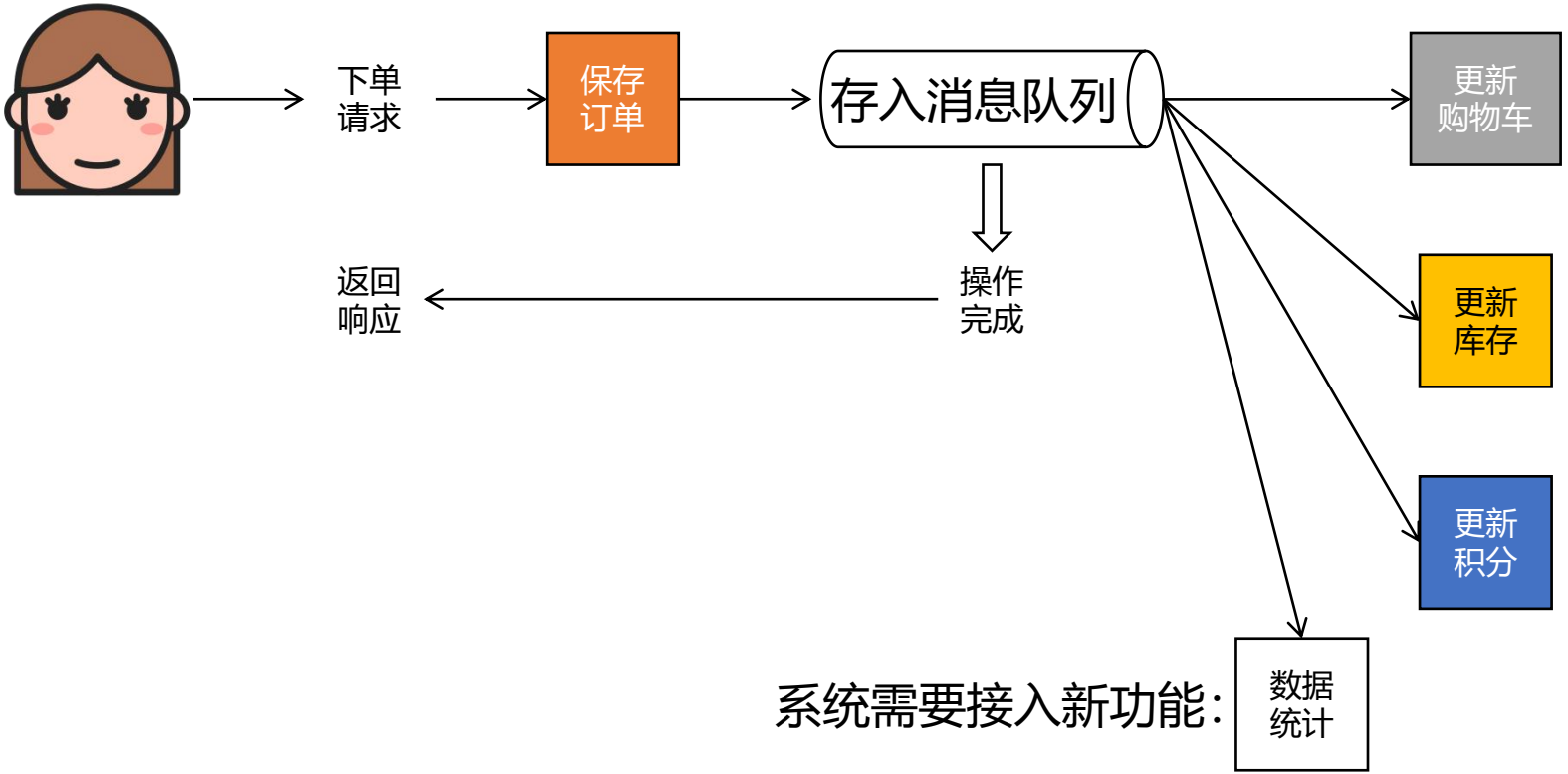
下单功能【同步】问题4：系统结构弹性不足



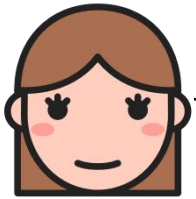
下单功能【异步】便于扩展



下单功能【异步】便于扩展



下单功能【异步】便于扩展



下单
请求

保存
订单

存入消息队列

更新
购物车

更新
库存

更新
积分

数据
统计

代码没有任何修改 ☺

操作
完成

返回
响应

系统需要接入新功能：

小结

- 系统耦合度高
- 并发压力持续向后续服务传导
- 系统结构缺乏弹性，可扩展性差
- 响应时间长

同步

小结

同步

- 系统耦合度高
- 并发压力持续向后续服务传导
- 系统结构缺乏弹性，可扩展性差
- 响应时间长

异步

- 参与的各功能模块相对独立，耦合度低
- 借助消息队列实现流量削峰填谷
- 各功能模块对接消息队列，系统功能扩容方便
- 快速响应

小结

同步

- 系统耦合度高
- 并发压力持续向后续服务传导
- 系统结构缺乏弹性，可扩展性差
- 响应时间长

异步

- 参与的各功能模块相对独立，耦合度低
- 借助消息队列实现流量削峰填谷
- 各功能模块对接消息队列，系统功能扩展方便
- 快速响应

注意

并不是把所有交互方式都改成异步

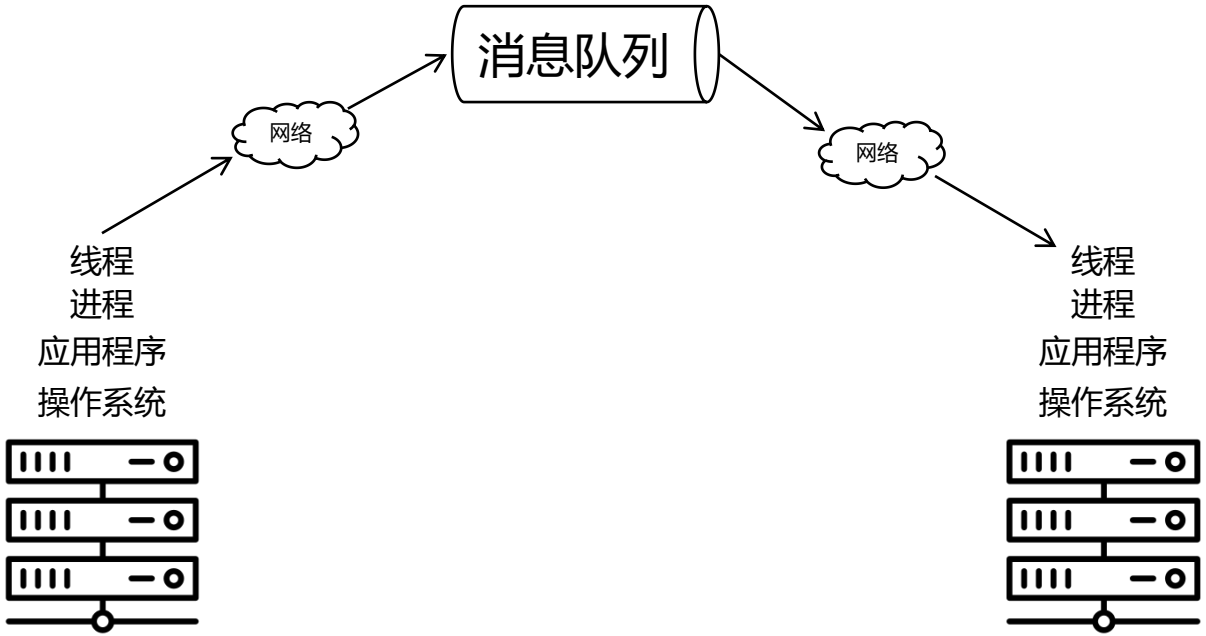
- **强关联**调用还是通过OpenFeign进行同步调用
- **弱关联**、**可独立拆分出来**的功能使用消息队列进行异步调用

2 什么是消息队列?

What is the message queue?

什么是消息队列?

消息队列是实现**应用程序**和**应用程序**之间**通信**的中间件产品



消息队列底层实现的两大主流方式

- 由于消息队列执行的是跨应用的信息传递，所以制定**底层通信标准**非常必要

- 目前主流的消息队列通信协议标准包括：
  - AMQP(**A**dvanced **M**essage **Q**ueuing **P**rotocol)：**通用**协议，IBM公司研发
  - JMS(**J**ava **M**essage **S**ervice)：**专门**为**Java**语言服务，SUN公司研发，一组由Java接口组成的Java标准

AMQP和JMS对比

七层网络模型

●AMQP对应**传输层**与**会话层**
●JMS对应**应用层**

消息模型

●AMQP支持**多种**消息模型，包括点对点（P2P）和发布/订阅（Pub/Sub）。
●JMS主要支持**点对点**和**发布/订阅**两种消息模型。

支持的编程语言和平台

●AMQP支持**多种编程语言**和平台，包括Java、C++、Python等
●JMS主要针对**Java**平台，因此在其他编程语言和平台上的支持相对较少

可靠性

●AMQP提供了**强大**的消息可靠性保证，包括消息持久化、事务性消息和消息确认机制。
●JMS也支持消息持久化和事务性消息，但具体实现**取决于**消息传递系统的**提供者**。

传输协议

●AMQP使用**二进制协议**进行消息传递，提供了高效、可靠的消息投递机制。
●JMS使用一种面向**文本**的**协议**（如HTTP或TCP），消息的传输效率可能较低。

拓展性和兼容性

●AMQP具有**很好的拓展性和兼容性**，可以在不同的消息代理之间交互操作。
●JMS在Java环境中有较好的拓展性和兼容性，但在与**非Java环境**集成时**受到限制**。

各主流MQ产品对比

	RabbitMQ	ActiveMQ	RocketMQ	Kafka
研发团队	Rabbit(公司)	Apache(社区)	阿里(公司)	Apache(社区)
开发语言	Erlang	Java	Java	Scala&Java
核心机制	基于AMQP的消息队列模型 使用生产者-消费者模式，将消息发布到队列中，然后被消费者订阅和处理	基于JMS的消息传递模型 支持点对点模型和发布-订阅模型	分布式的消息队列模型 采用主题(Topic)和标签(Tag)的方式进行消息的分类和过滤	分布式流平台， 通过发布-订阅模型进行高吞吐量的消息处理
协议支持	XMPP STOMP SMTP	XMPP STOMP OpenWire REST	自定义协议	自定义协议 社区封装了HTTP协议支持
客户端支持语言	官方支持Erlang、Java、Ruby等 社区产出多种API，几乎支持所有语言	Java C/C++ Python PHP Perl .NET等	Java C++不成熟	官方支持Java 社区产出多种API，如PHP、Python等
可用性	镜像队列、仲裁队列	主从复制	主从复制	分区和副本
单机吞吐量	每秒十万左右级别	每秒数万级	每秒十万+级(双十一)	每秒百万级
消息延迟	微秒级	毫秒级	毫秒级	毫秒以内
消息确认	完整的消息确认机制		内置消息表，消息保存到数据库实现持久化	
功能特性	并发能力强， 性能极好， 延时低， 社区活跃， 管理界面丰富	老牌产品 成熟度高 文档丰富	MQ功能比较完备 扩展性佳	只支持主要的MQ功能 毕竟是专门为大数据领域服务的

3 RabbitMQ简介

RabbitMQ Introduce

RabbitMQ介绍

- 官网地址：https://www.rabbitmq.com/
- Logo： 
- RabbitMQ是一款**基于AMQP**、**由Erlang语言开发**的消息队列产品，2007年Rabbit技术公司发布了它的1.0版本

RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用

消息的发送端，也可以称为消息的生产者

Producer

RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用

消息的接收端，也可以称为消息的消费者

Producer

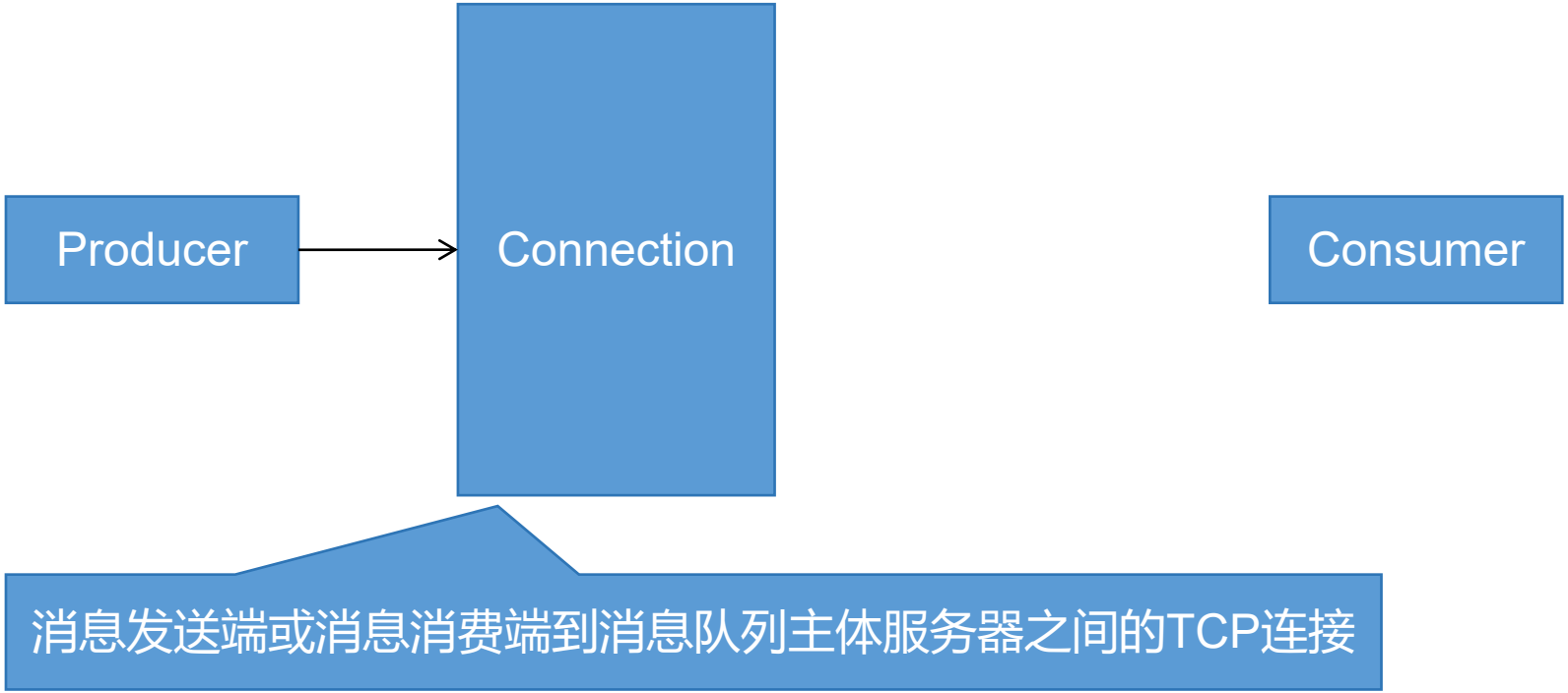
Consumer

RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用

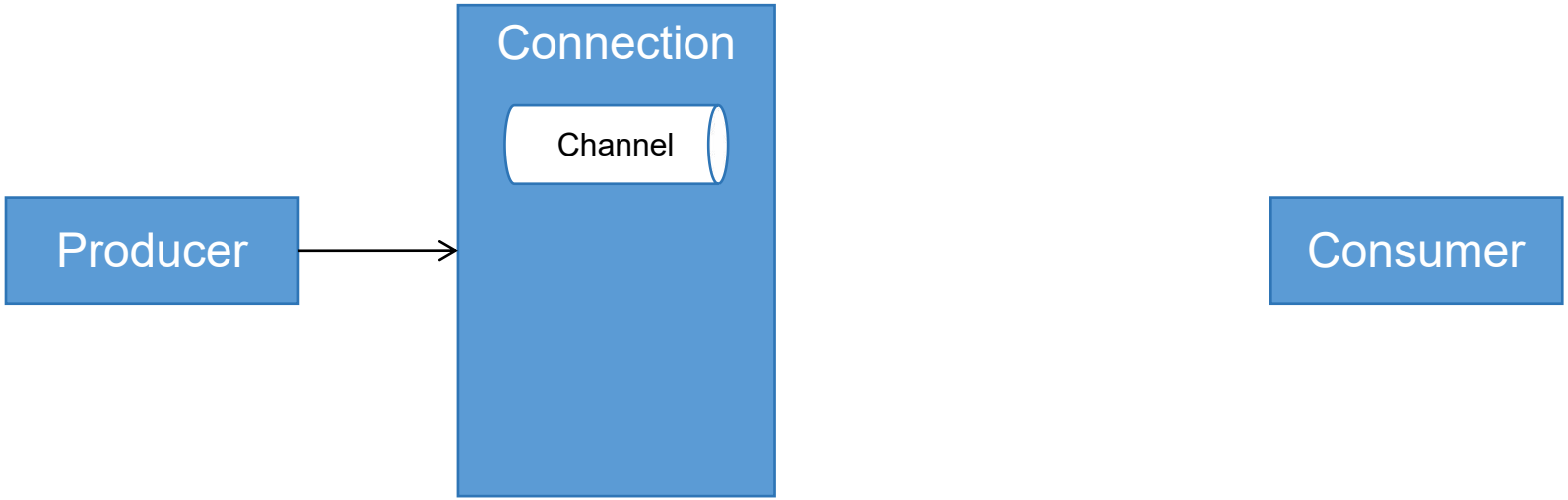
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



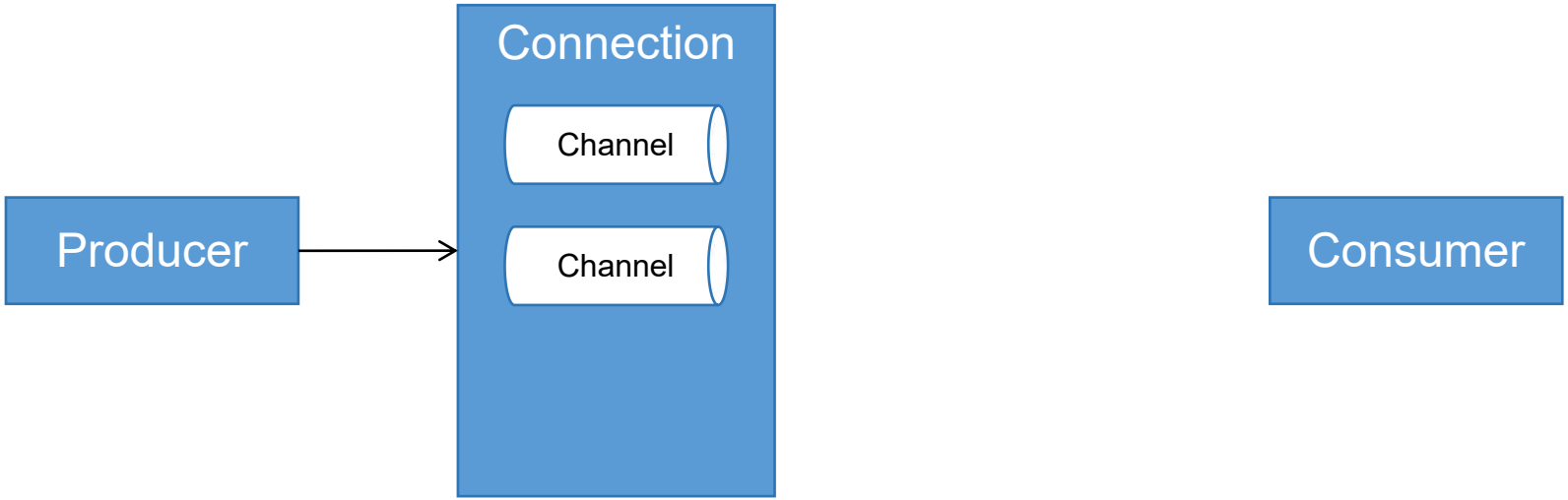
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



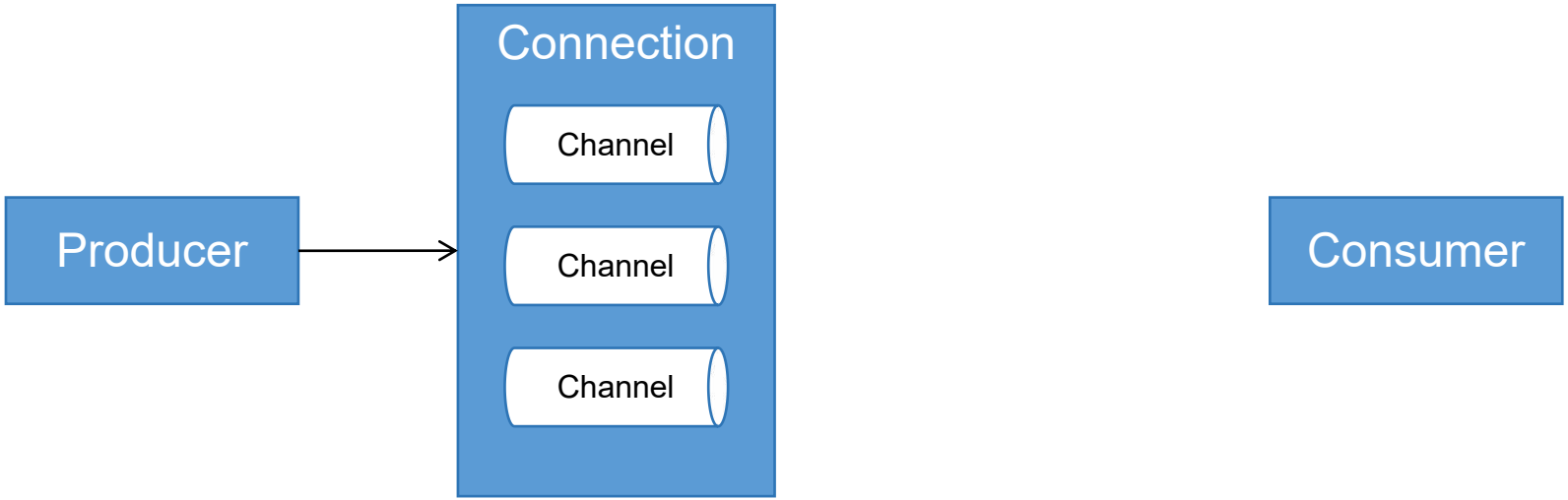
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



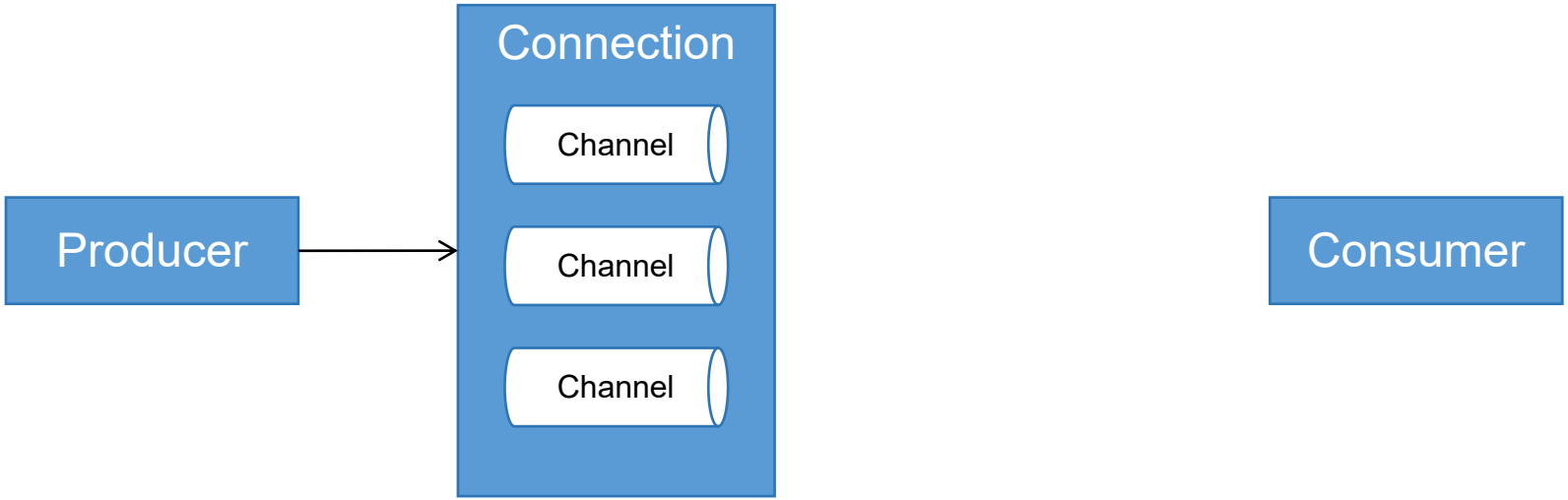
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



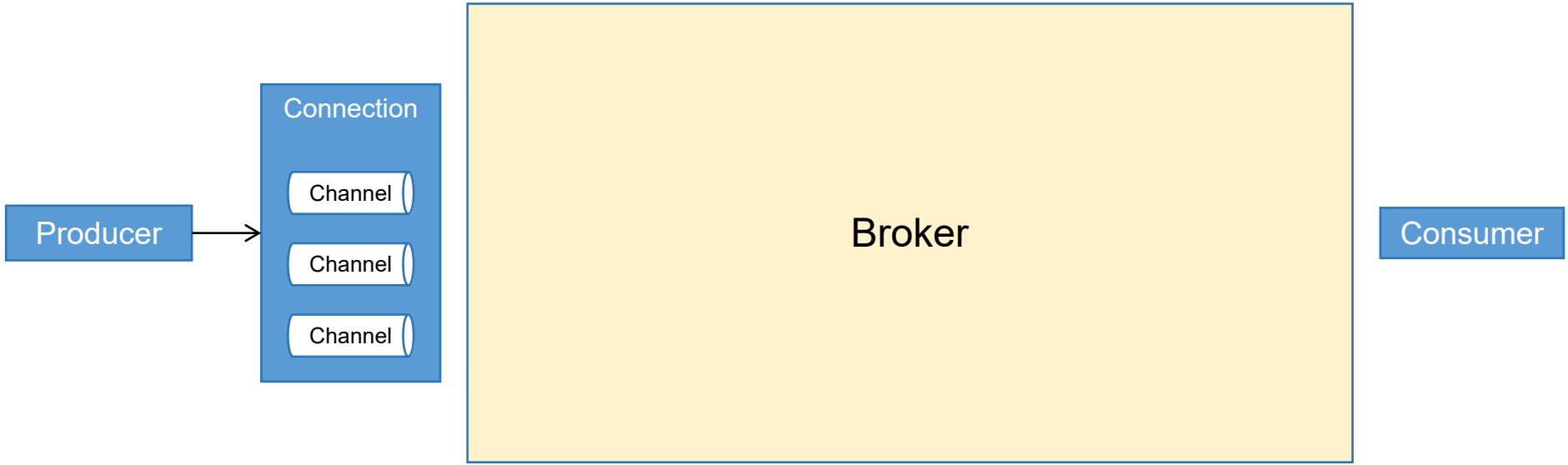
建立TCP连接需要三次握手，反复确认。
所以如果每一次访问RabbitMQ服务器都建立一个Connection开销会极大，效率低下。
所以Channel就是在一个已经建立的Connection中建立的逻辑连接。
如果应用程序支持多线程，那么每个线程创建一个单独的Channel进行通讯。
每个Channel都有自己的id，Channel之间是完全隔离的。
总之核心点就是：实现**Connection复用**

RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用

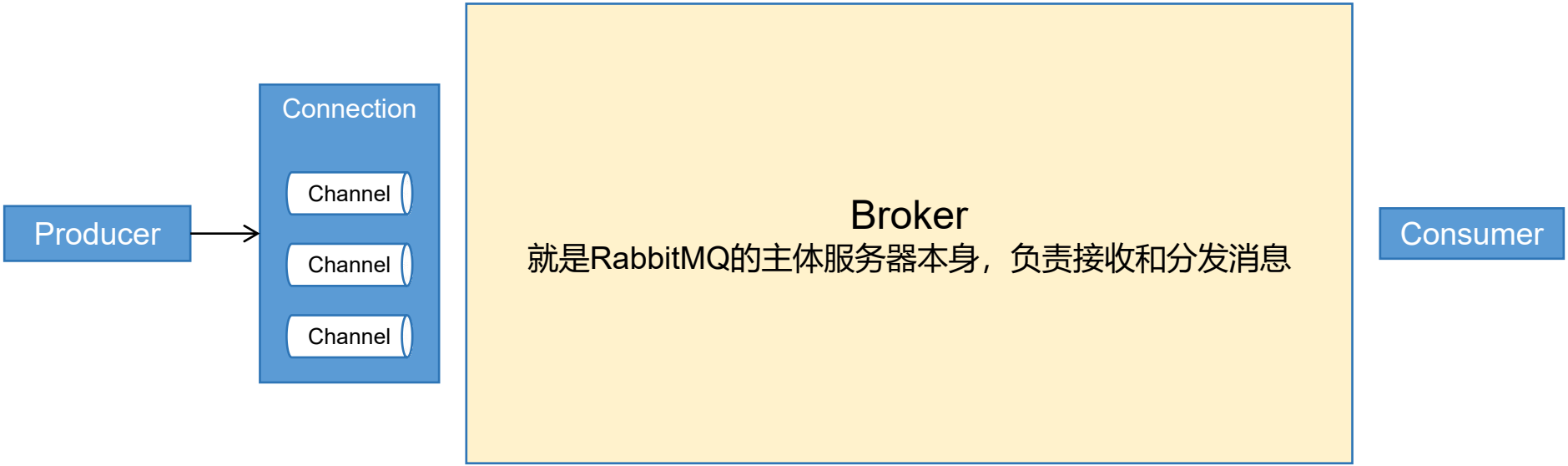
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



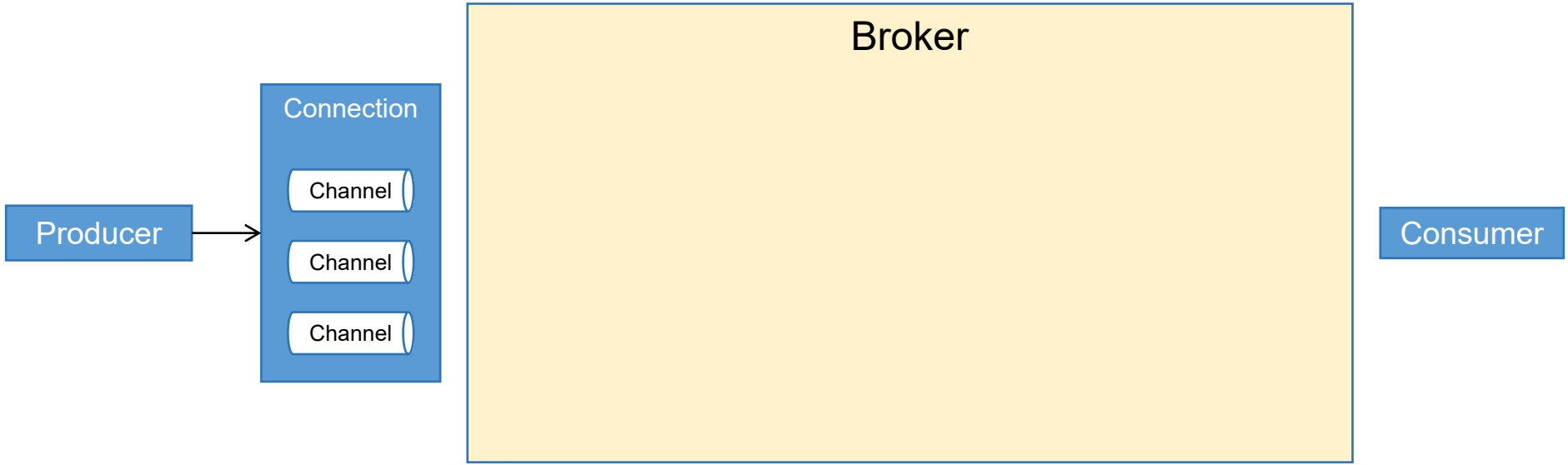
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



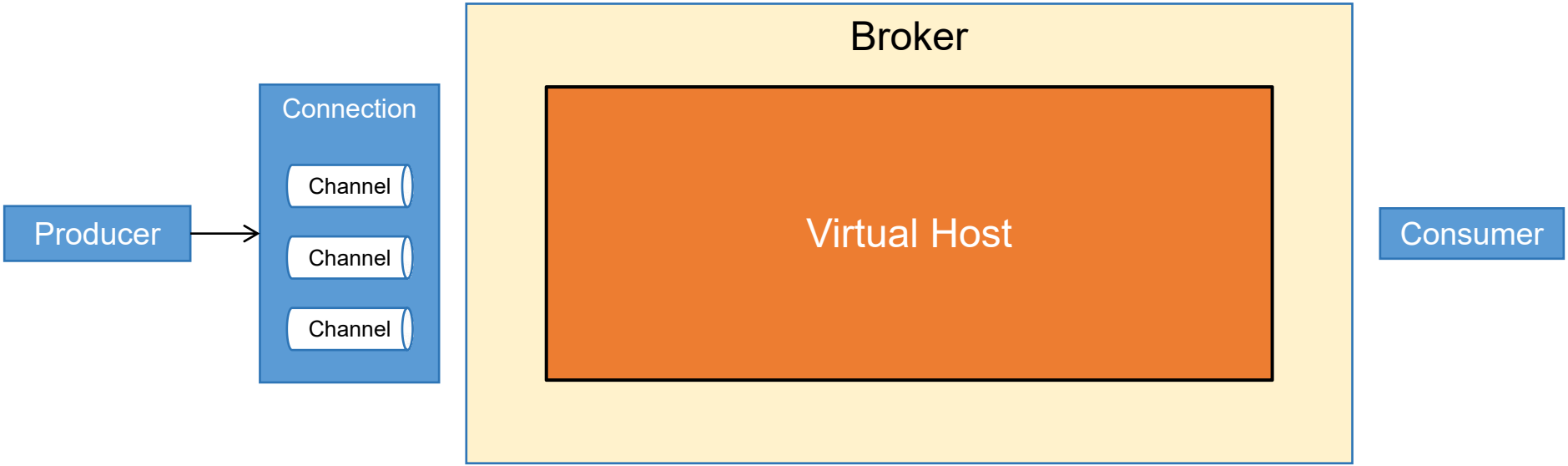
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



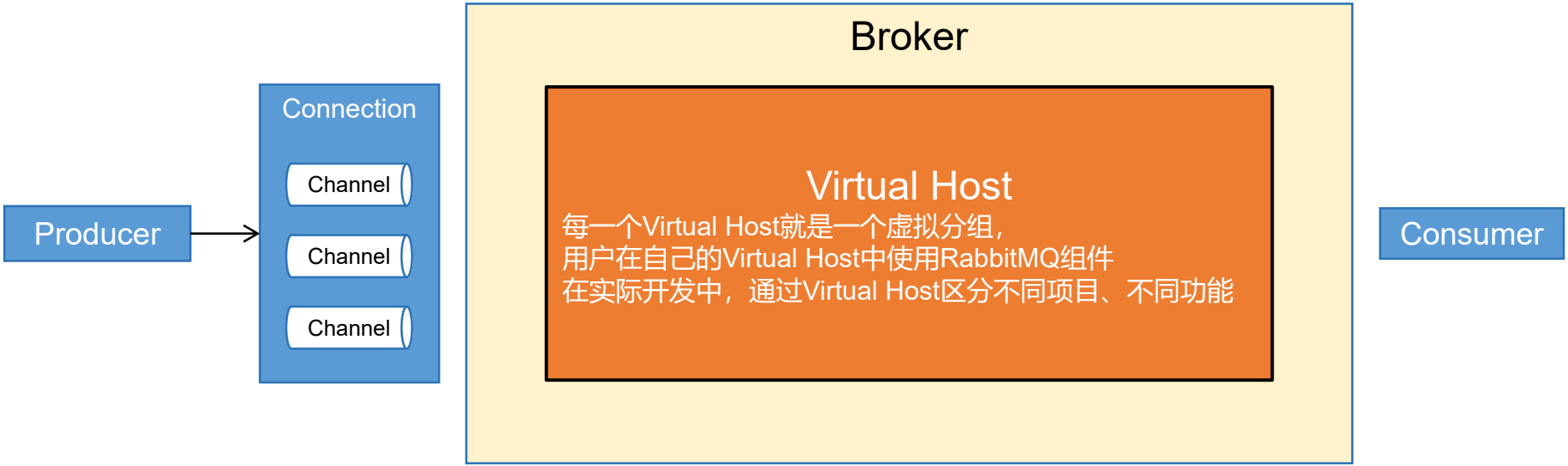
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



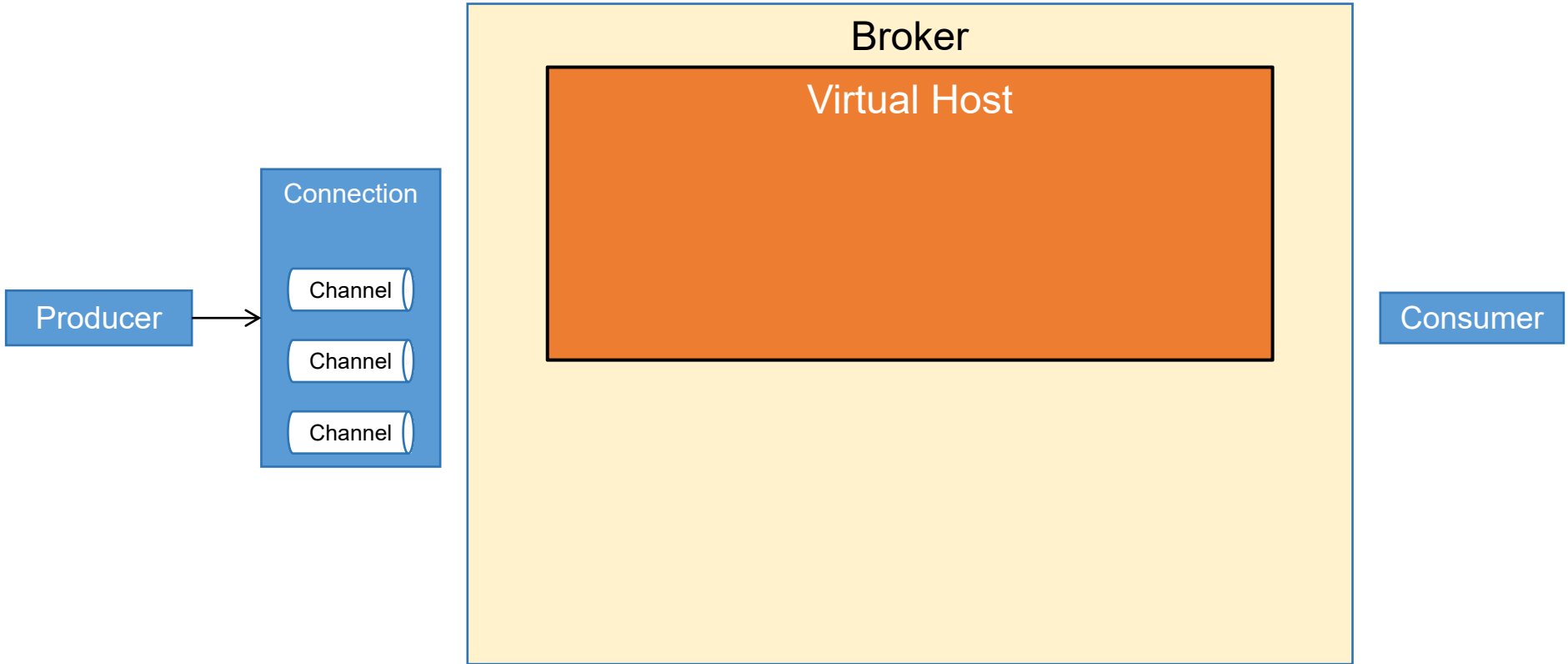
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



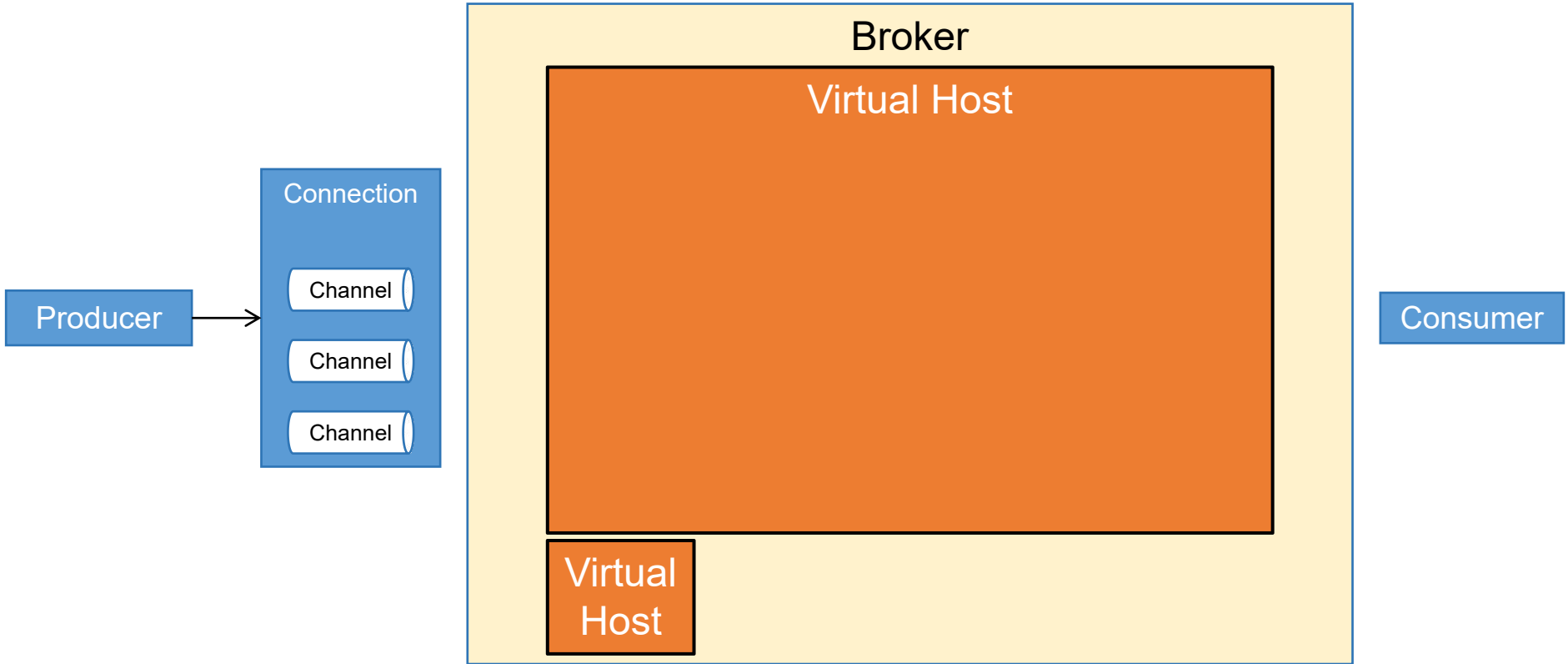
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



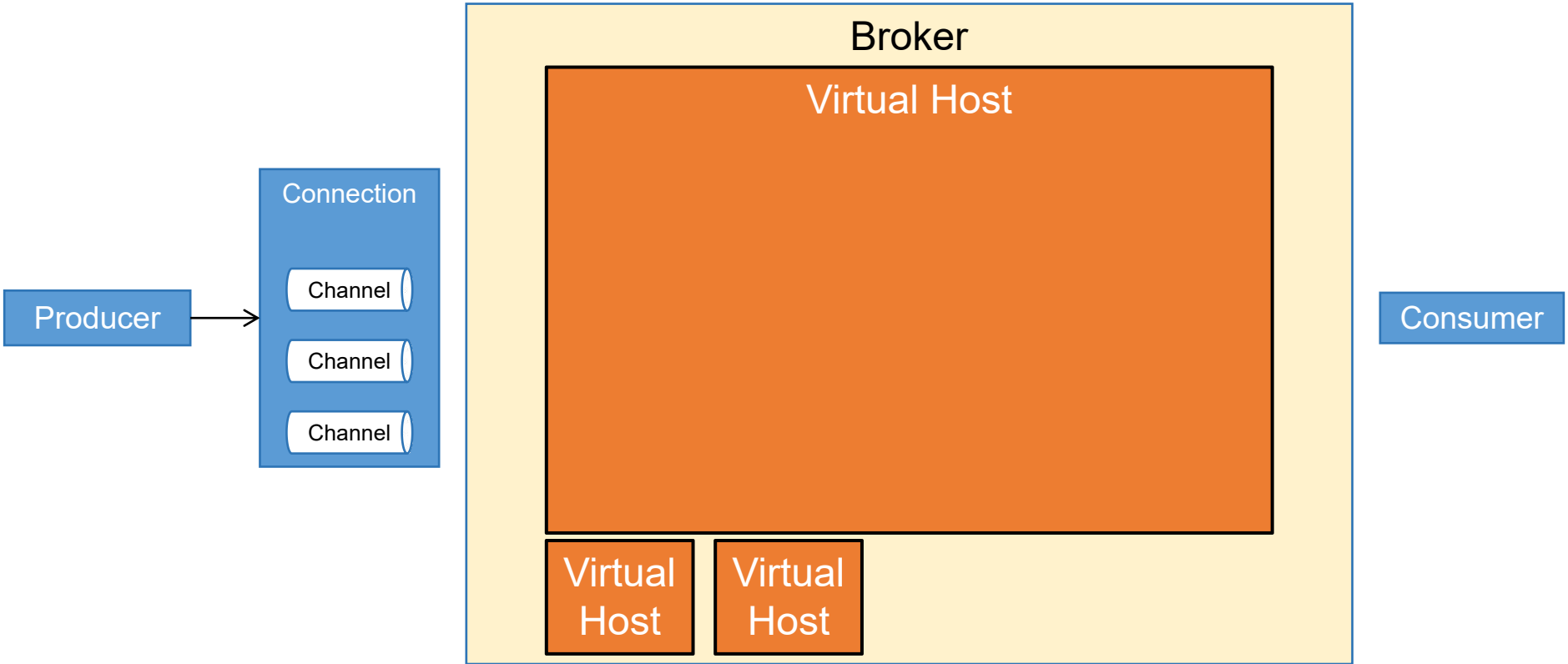
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



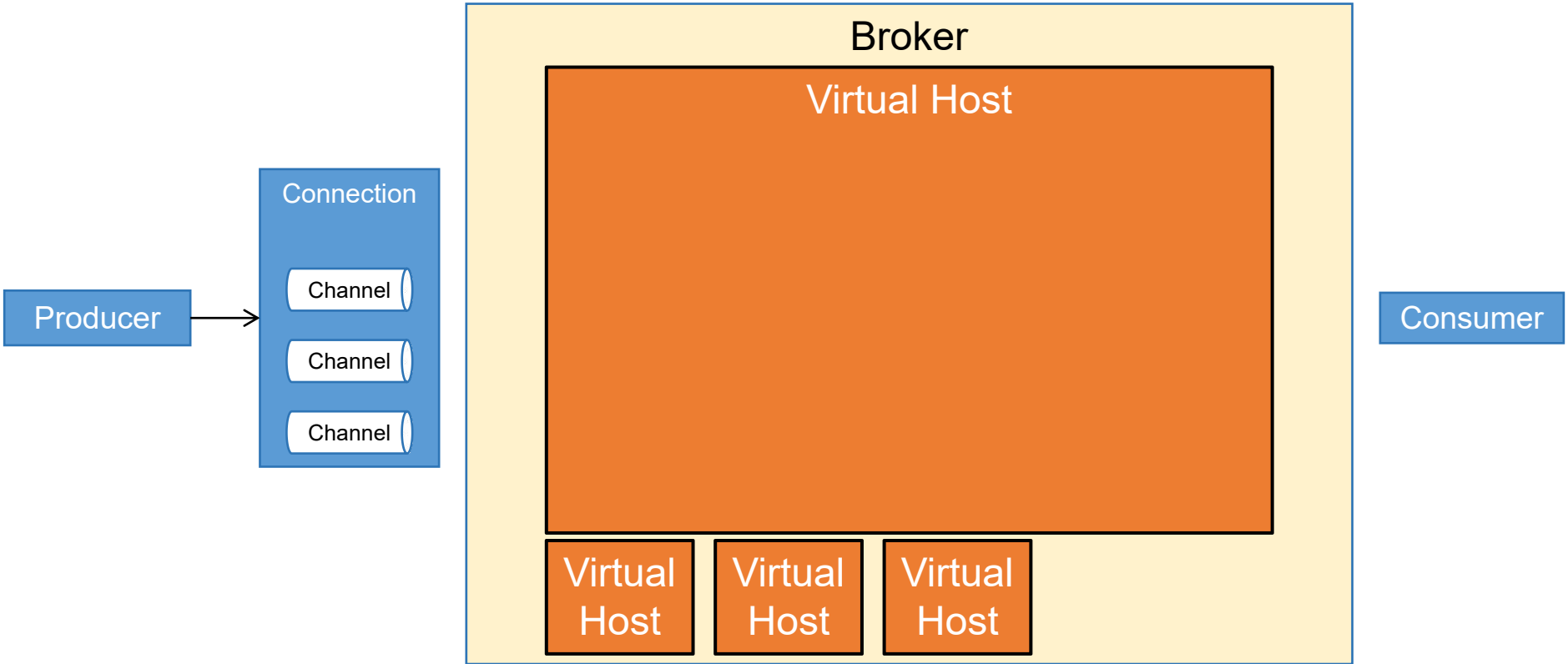
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



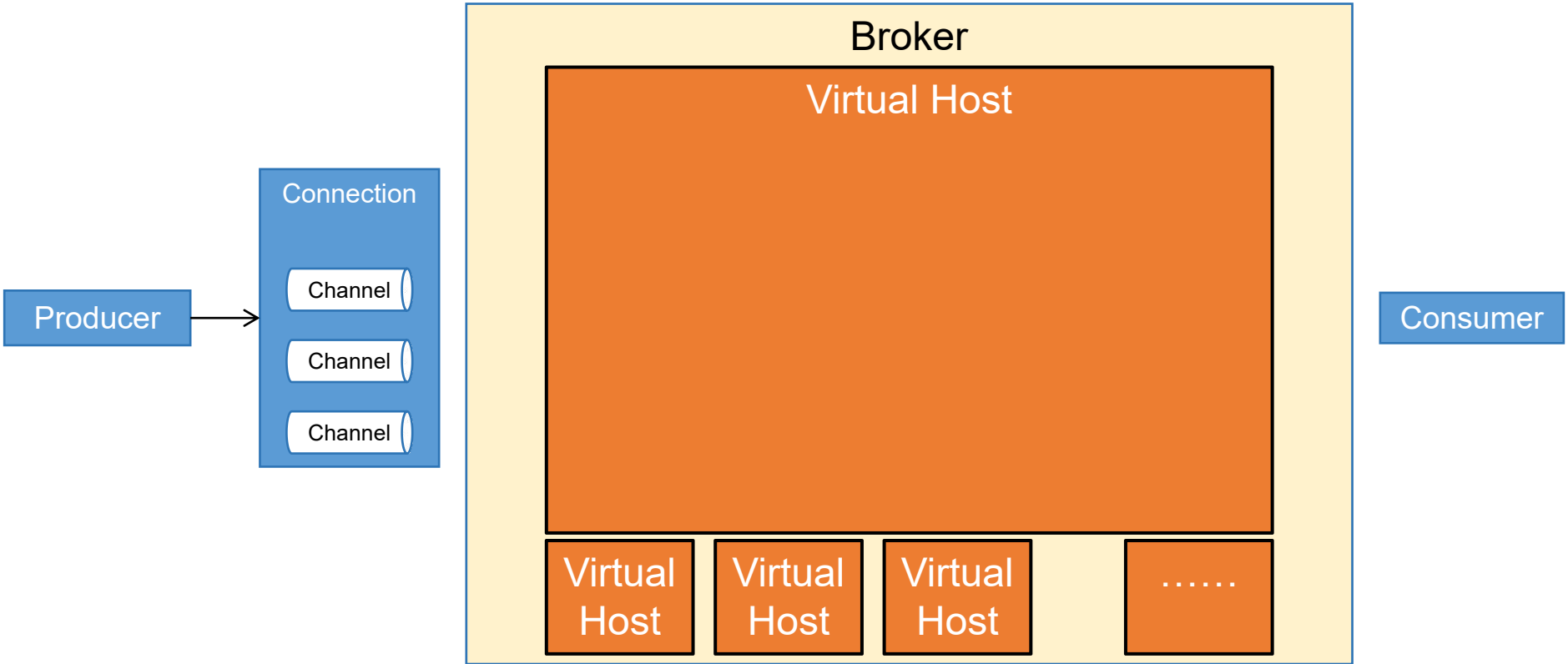
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



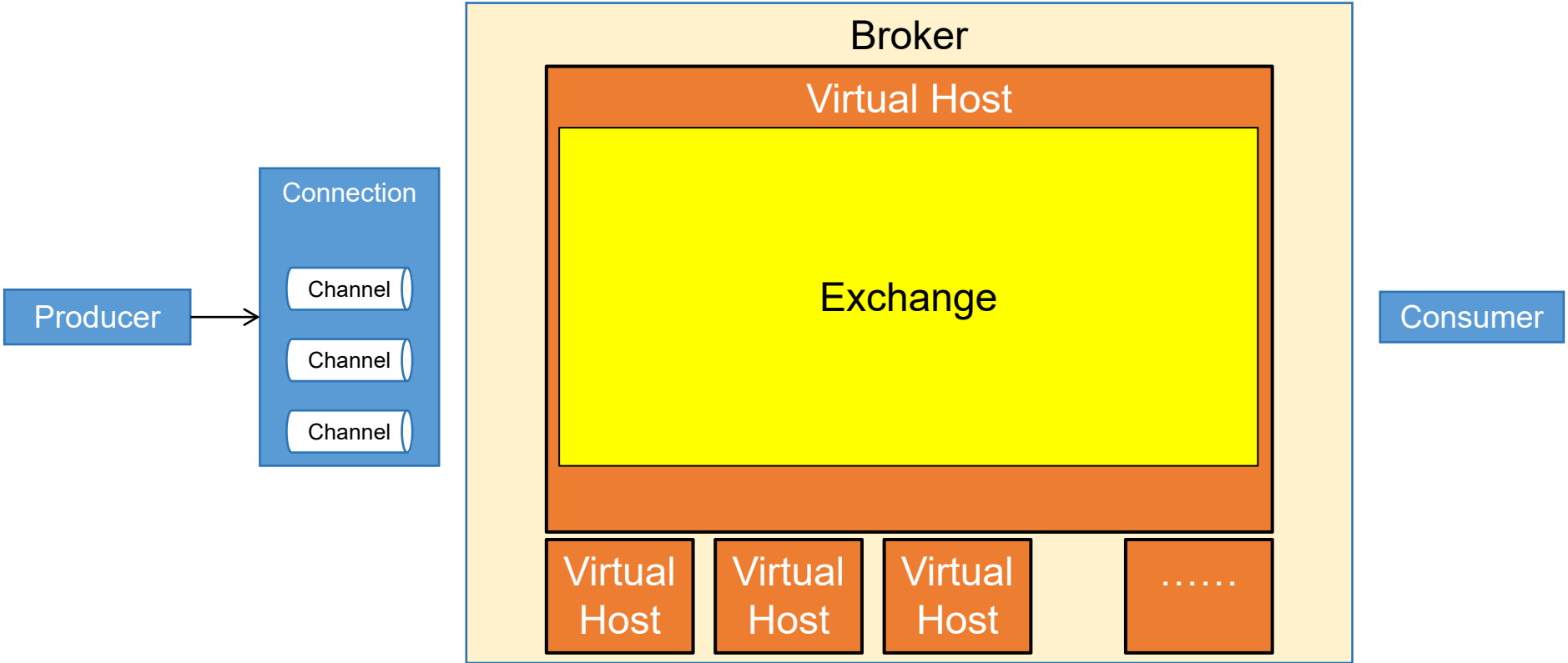
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



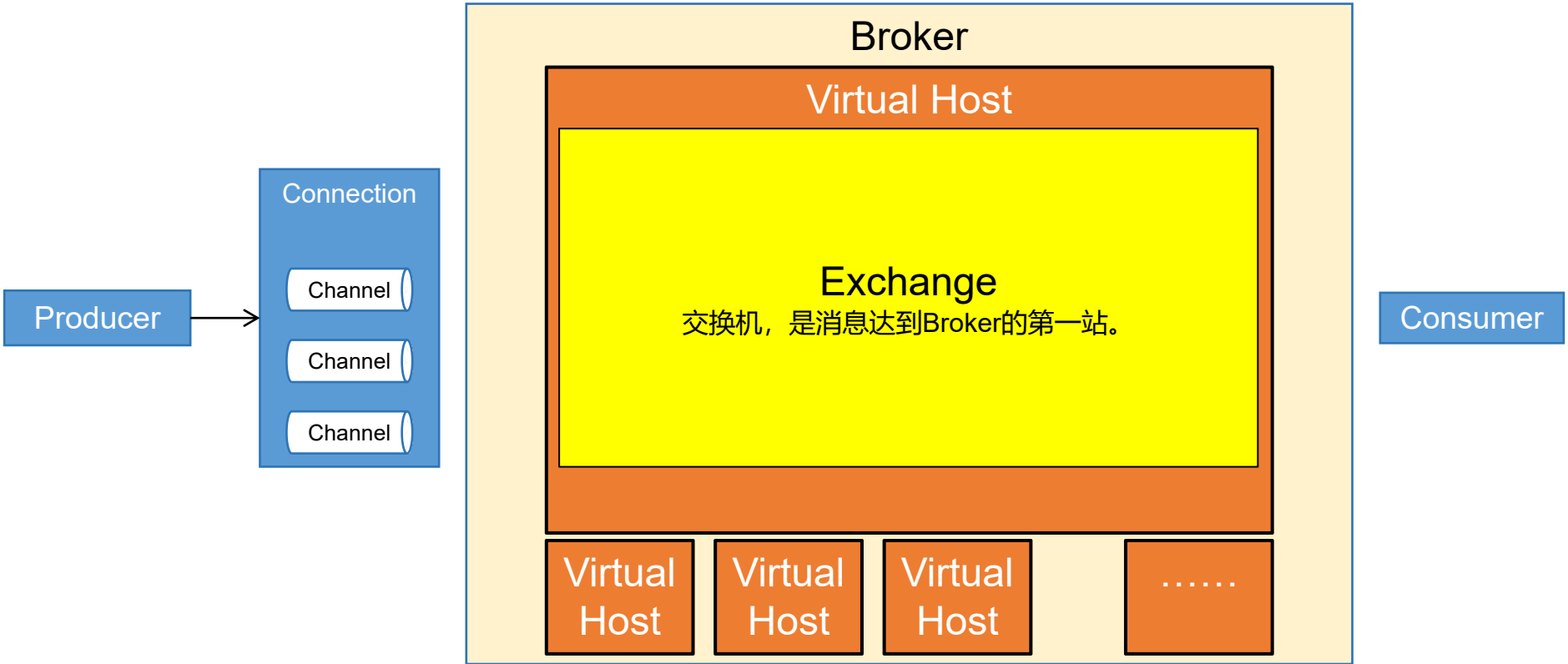
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



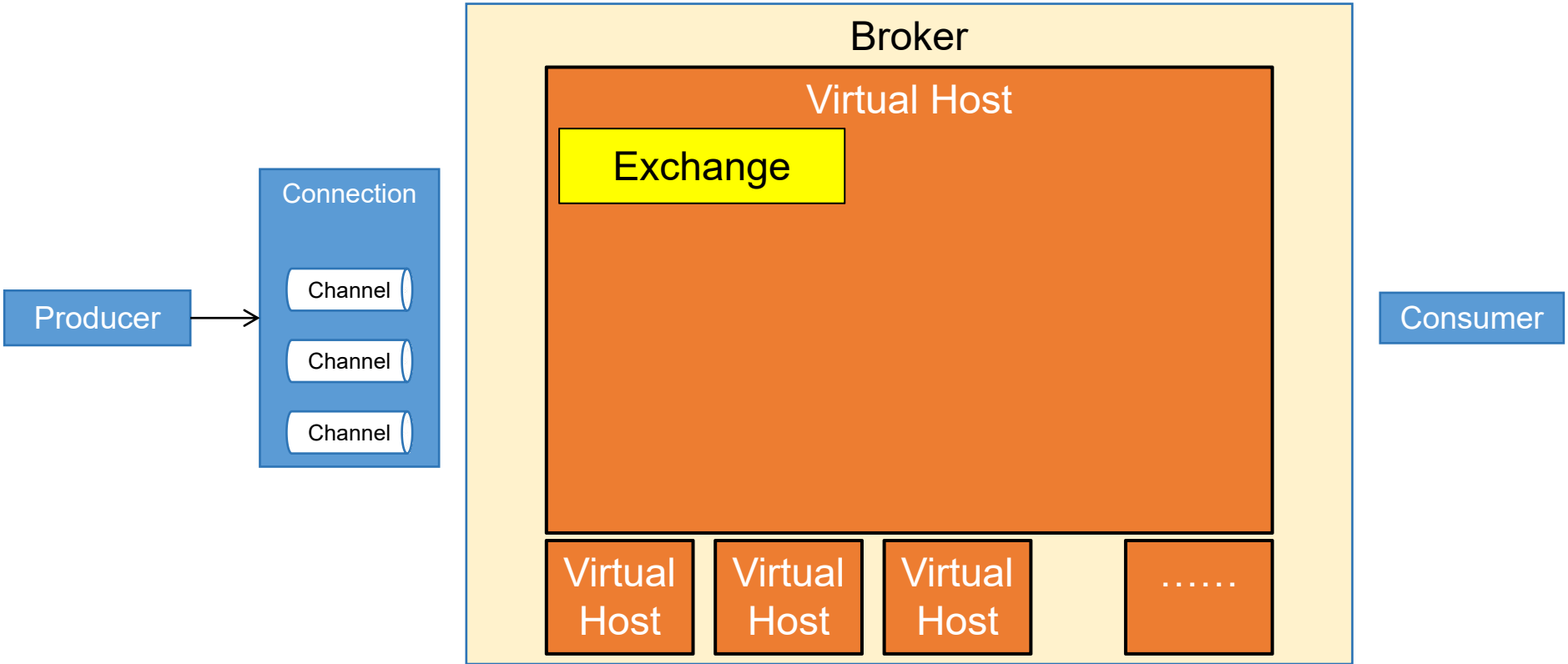
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



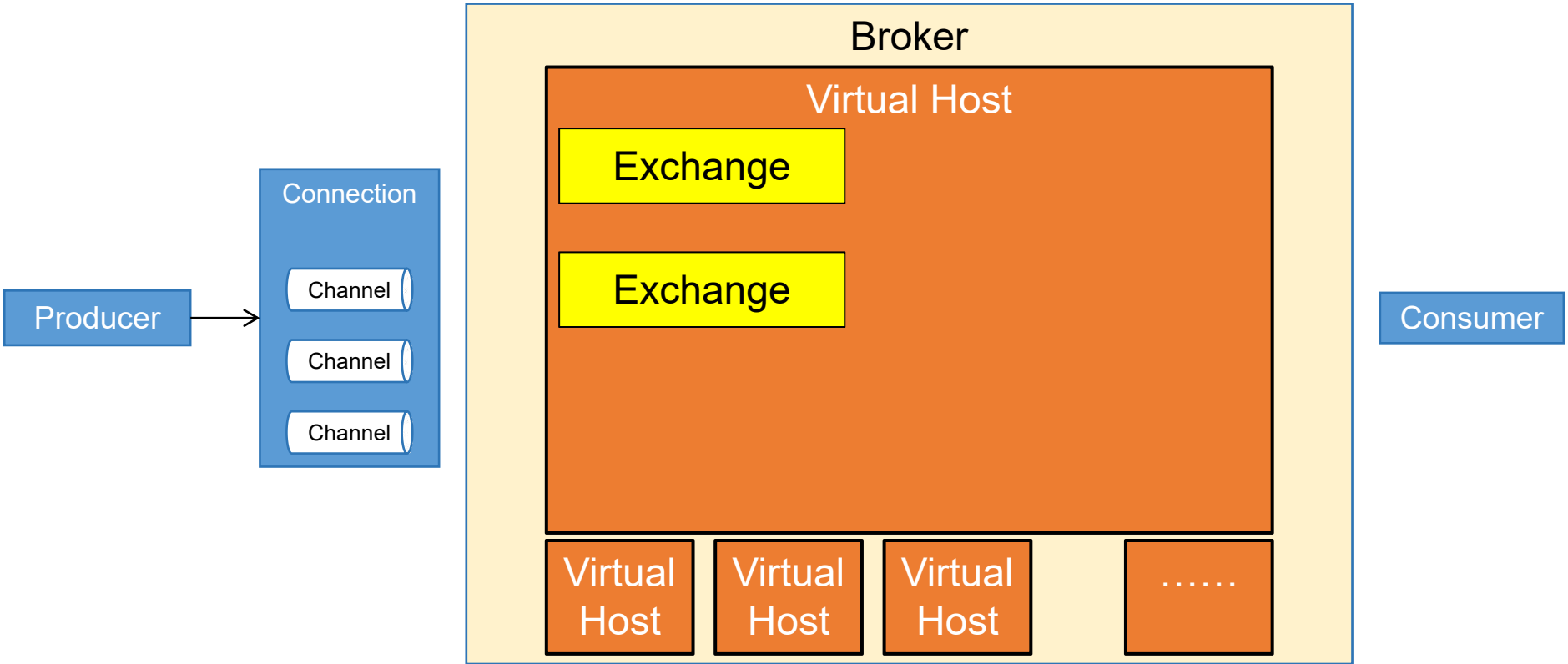
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



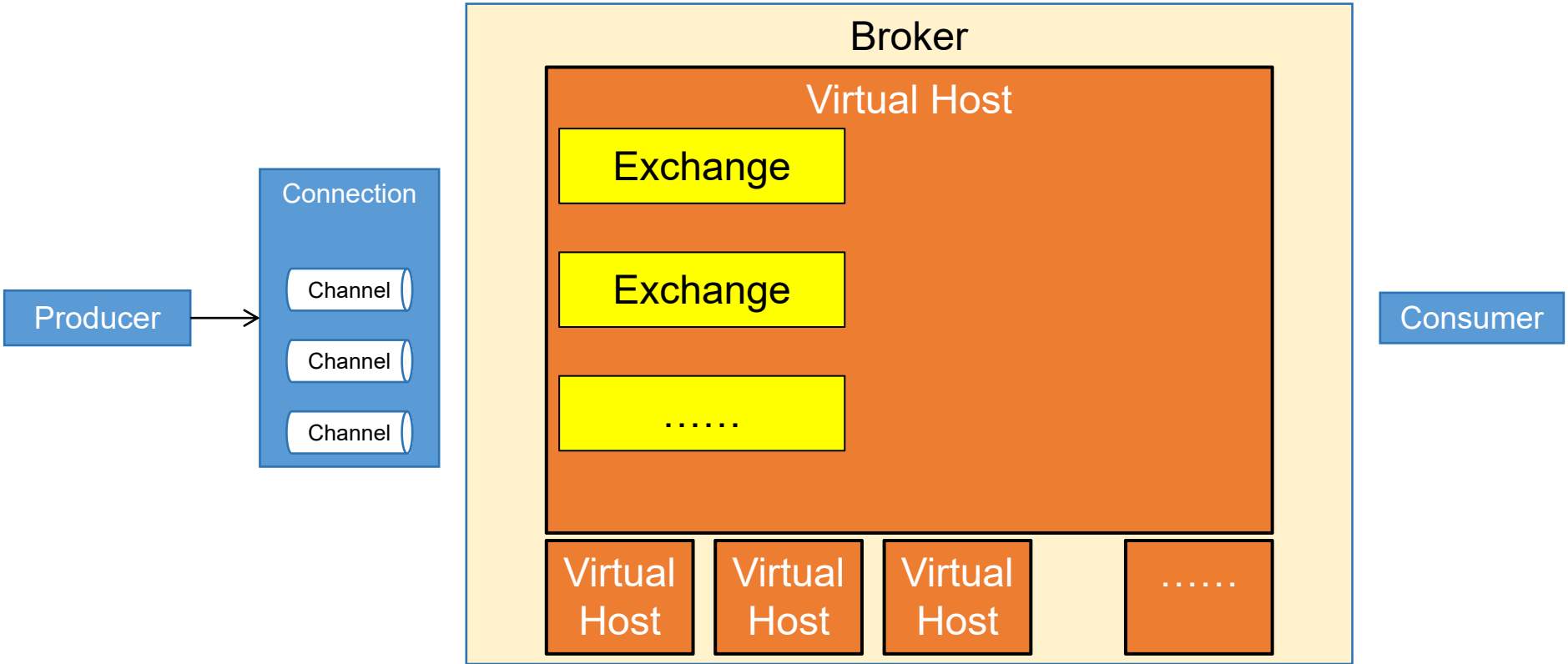
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



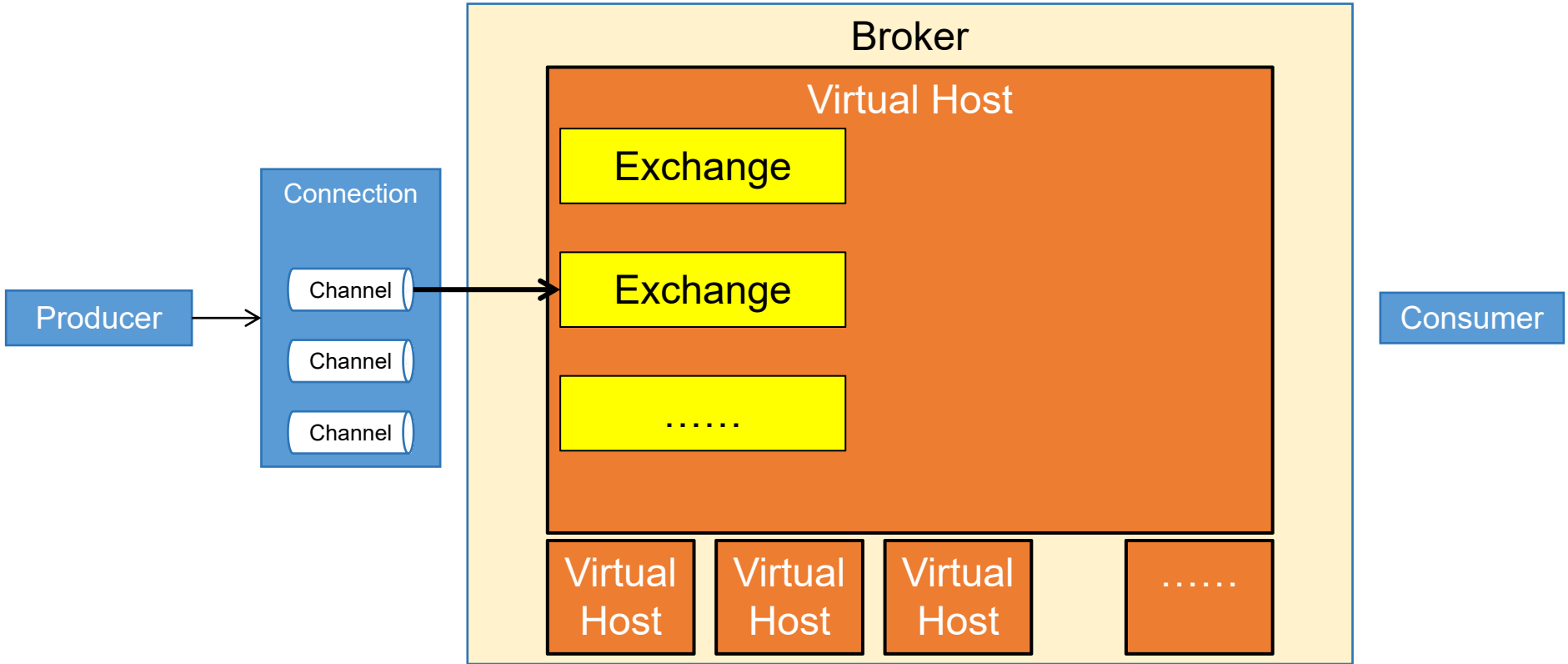
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



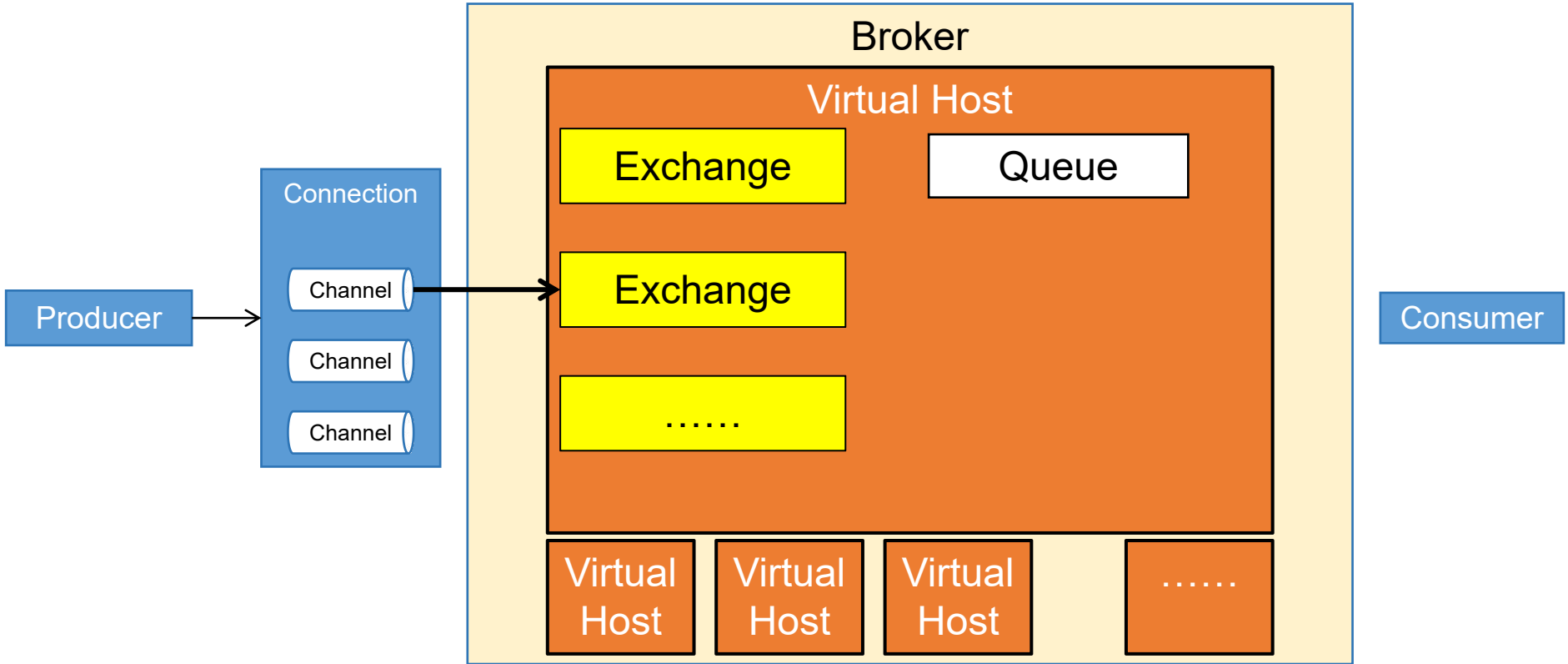
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



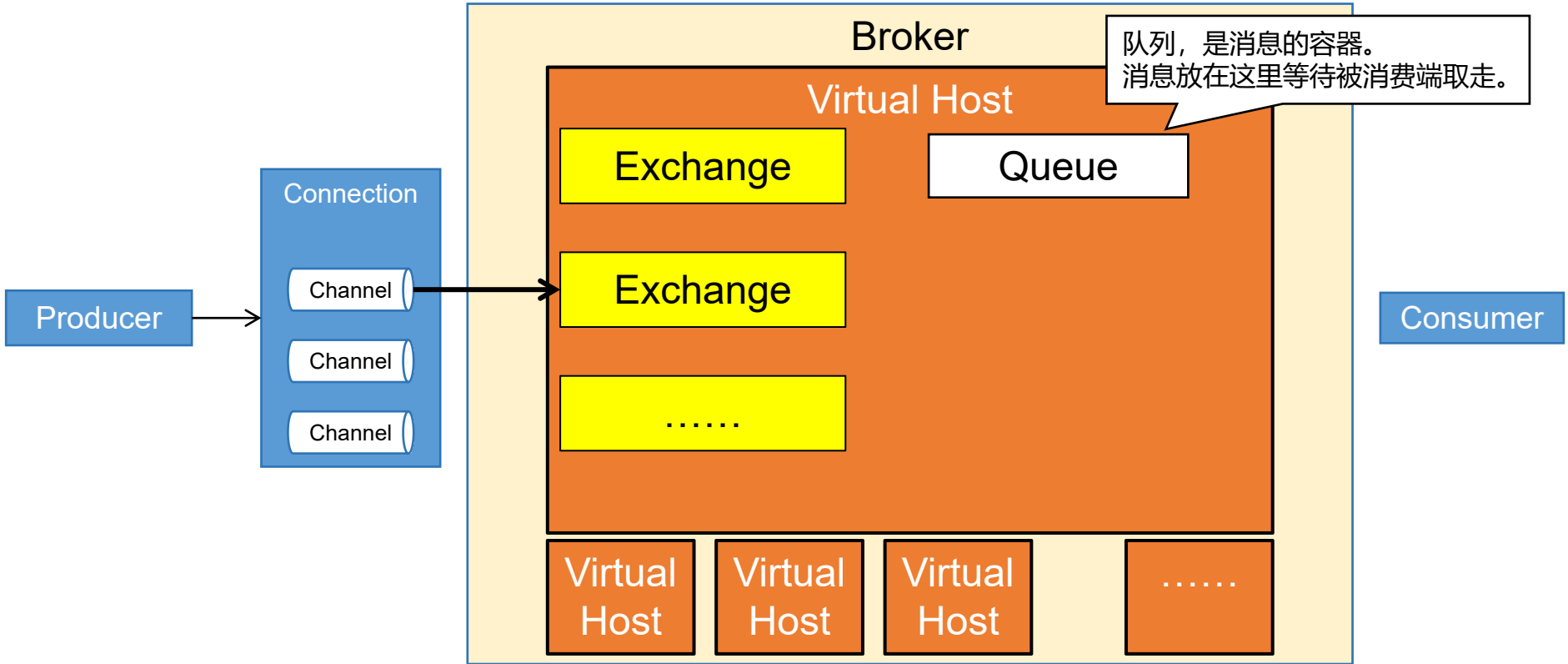
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



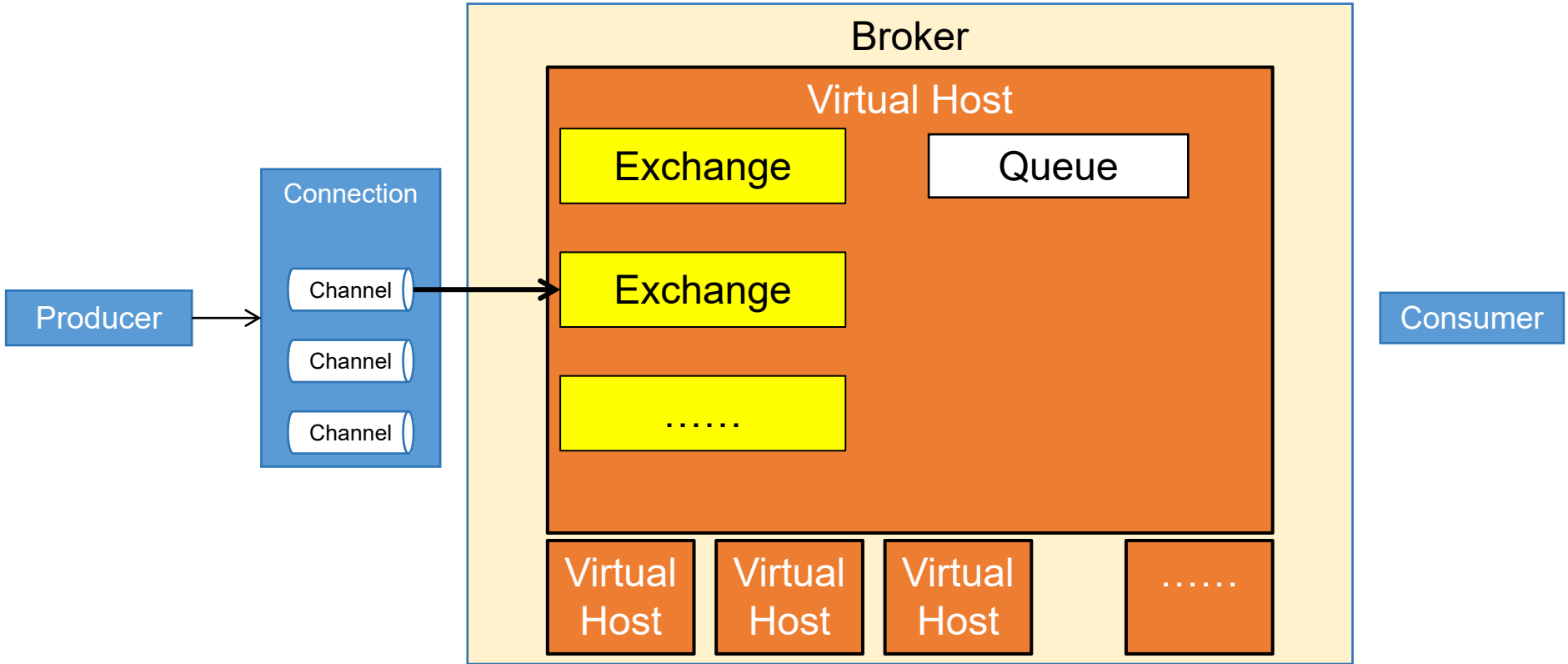
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



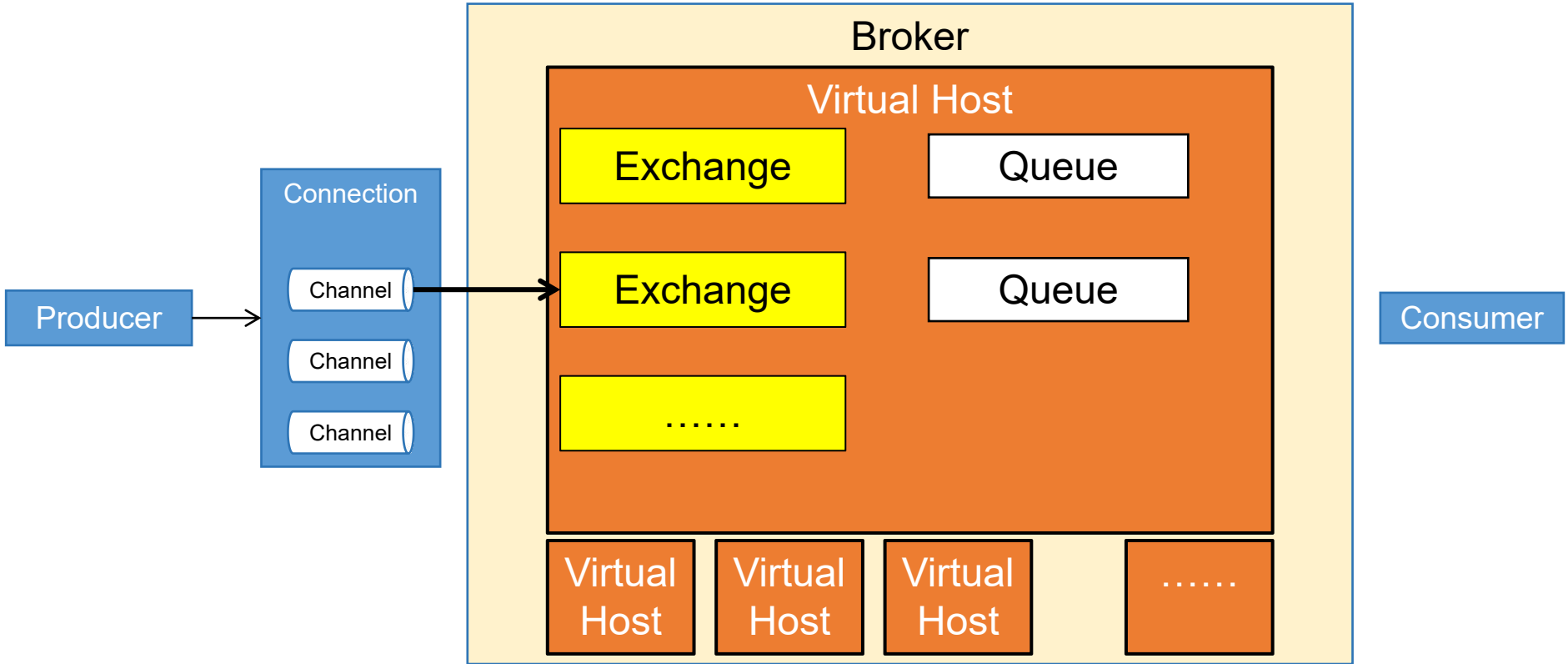
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



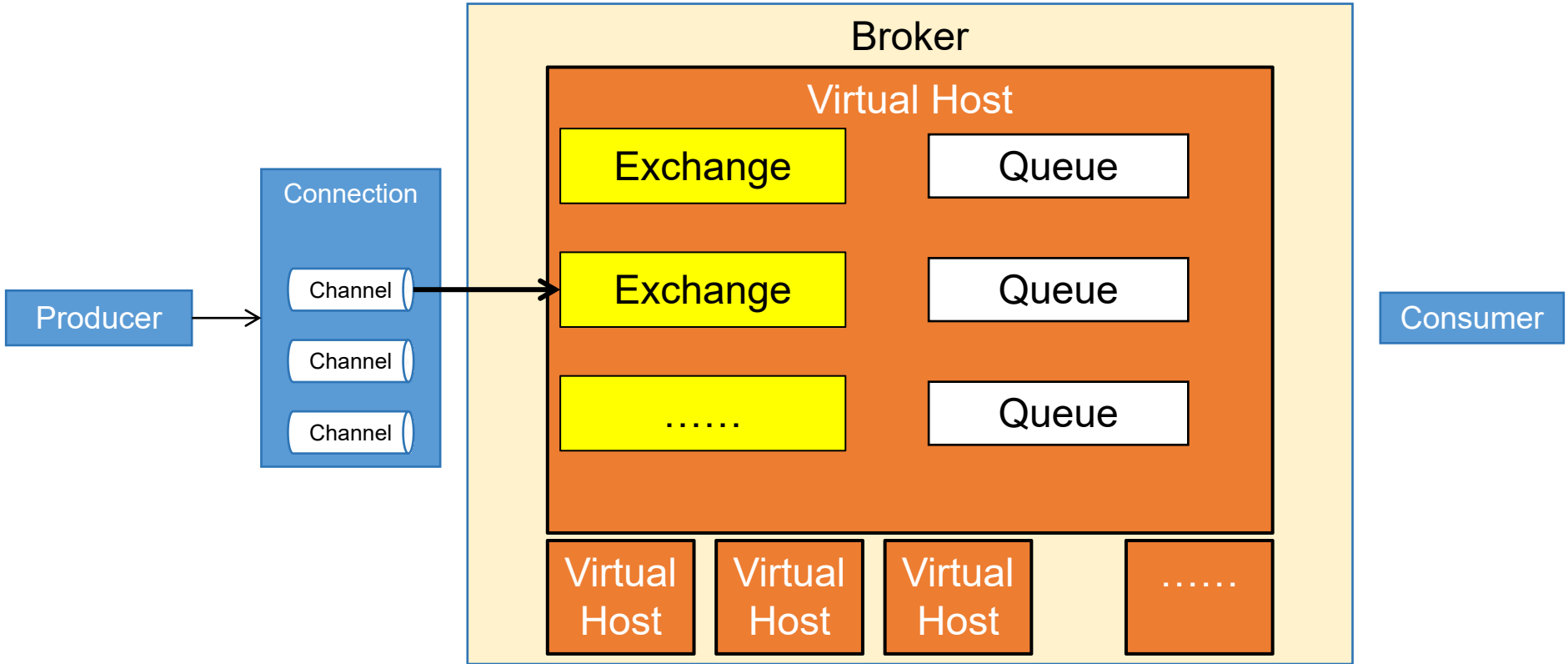
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



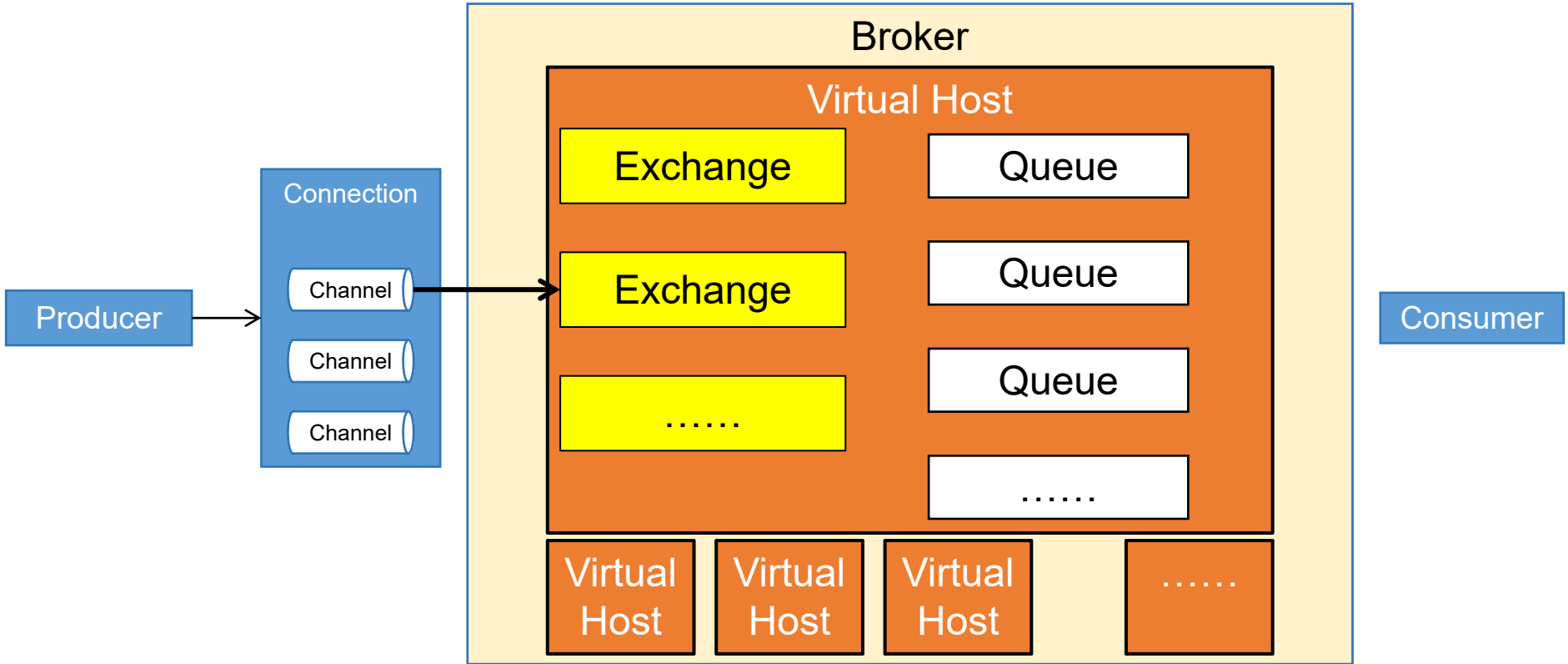
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



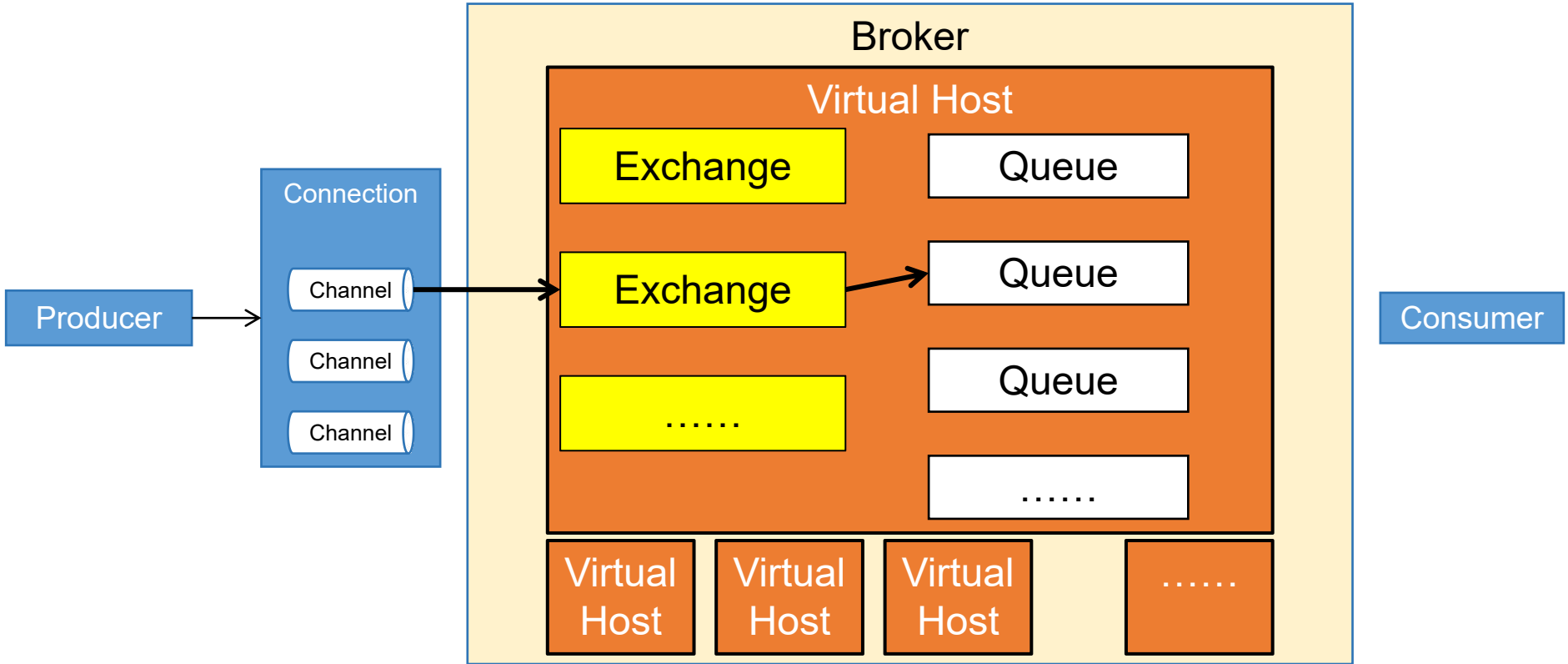
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



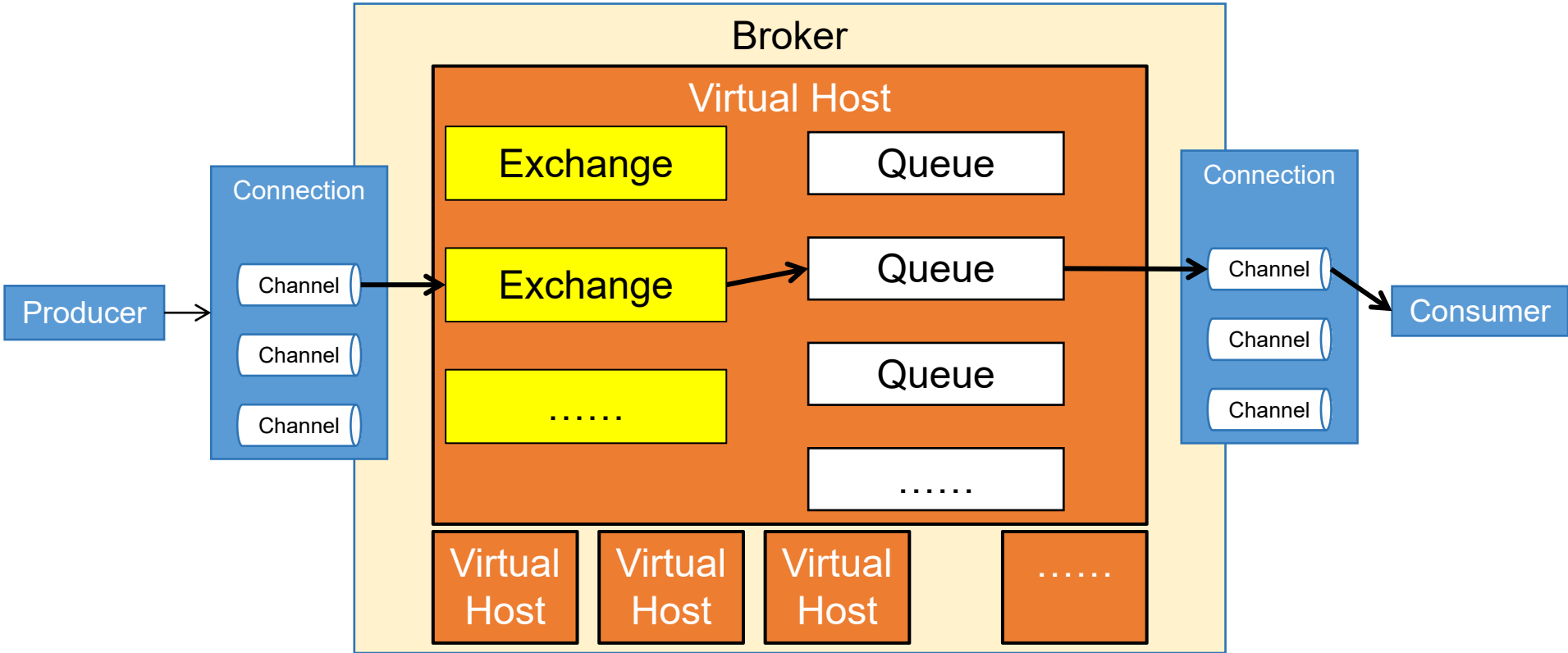
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



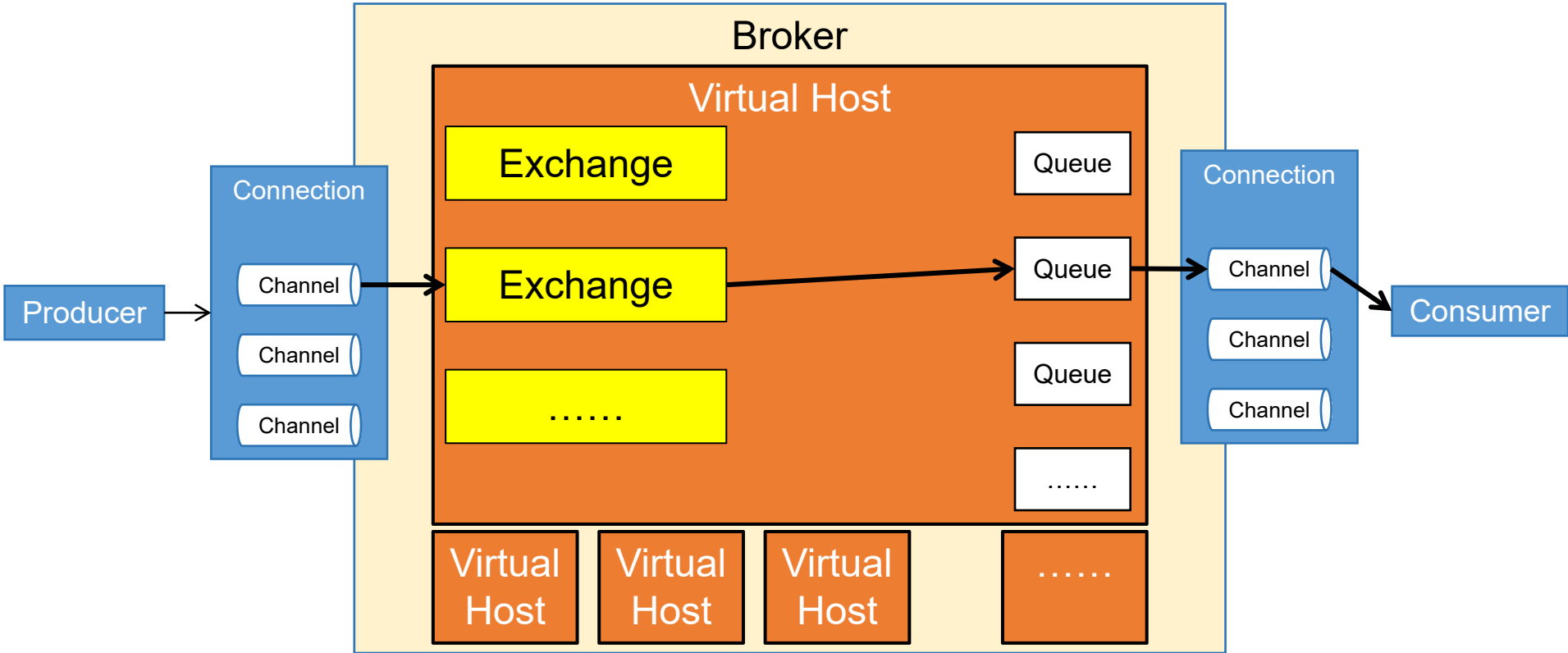
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



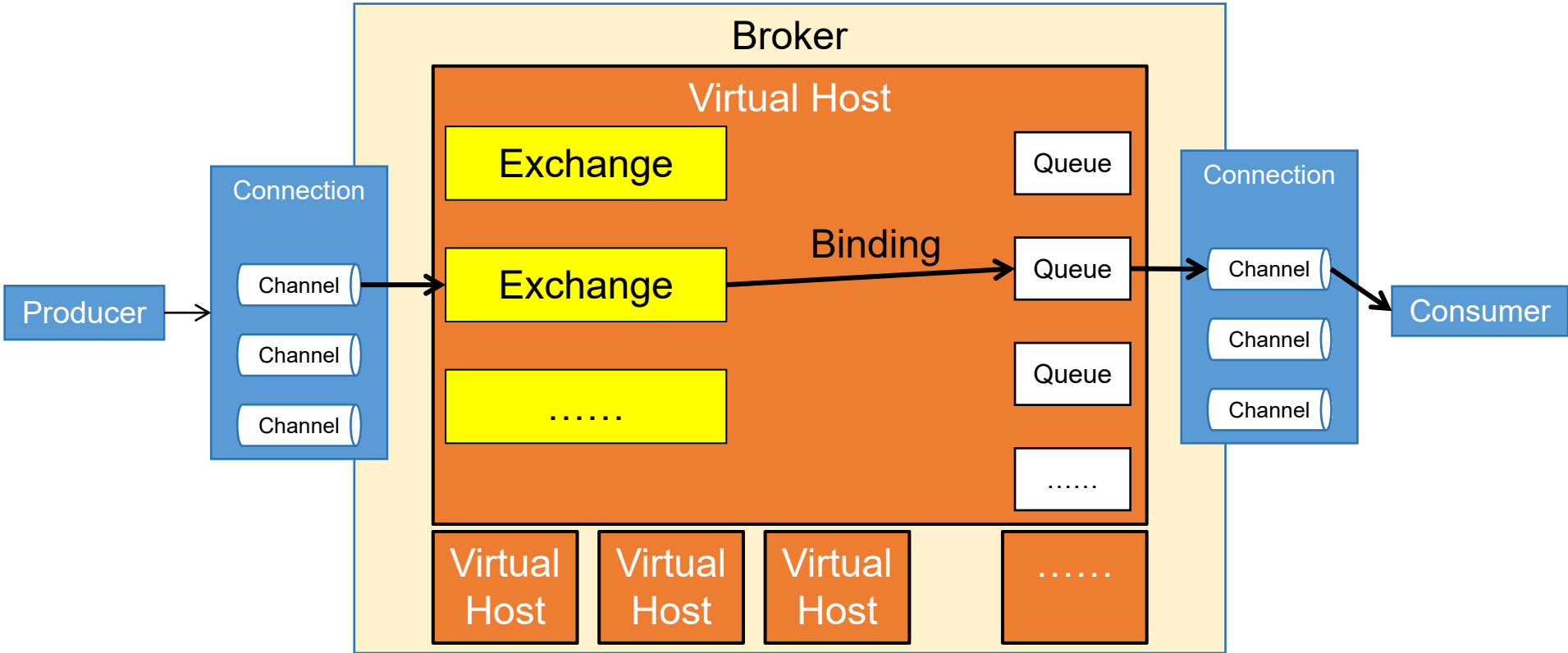
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



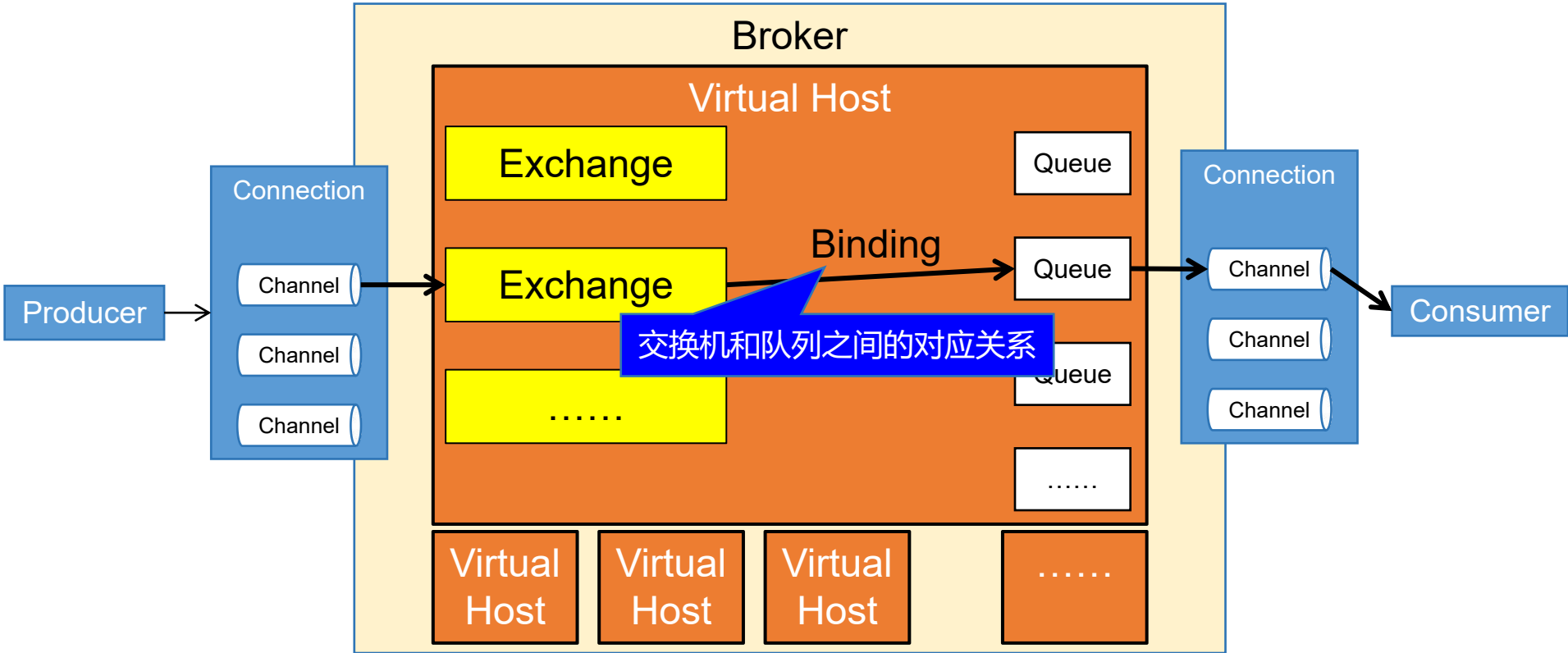
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



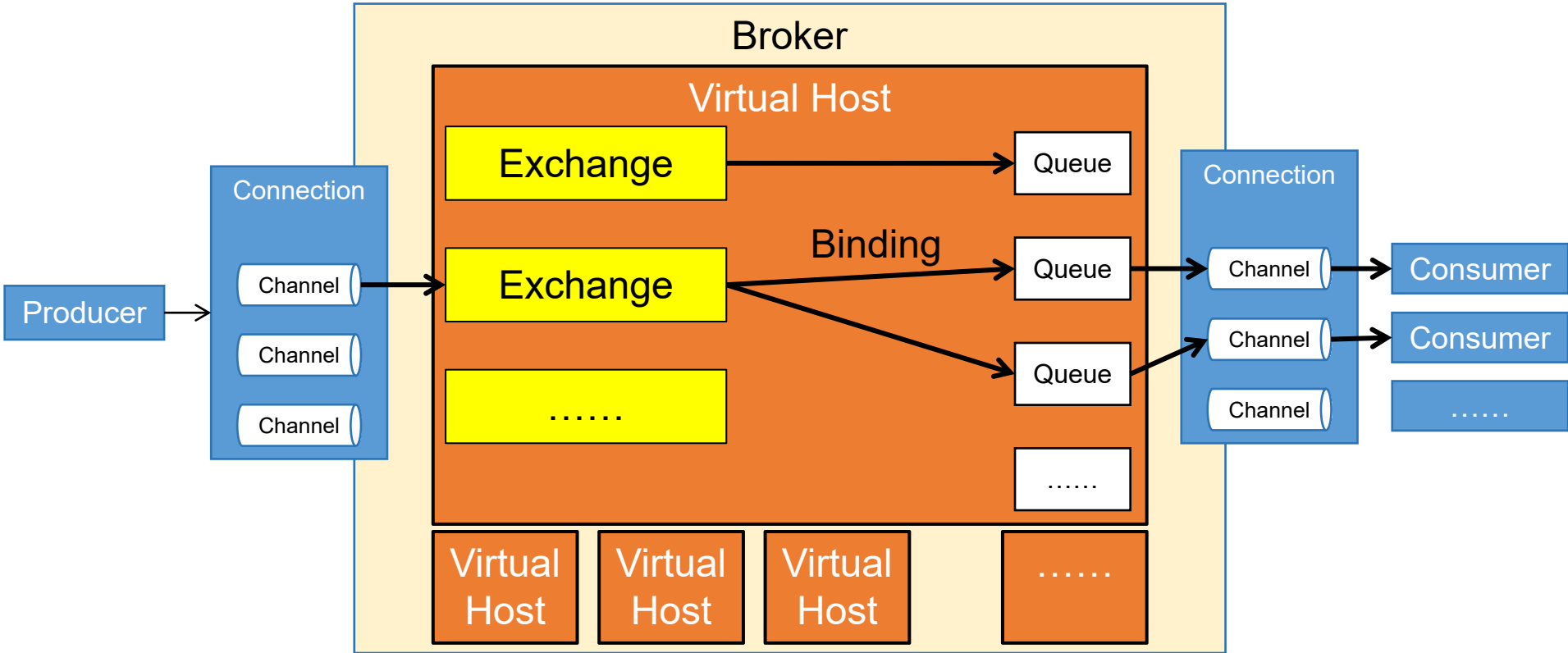
RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



RabbitMQ体系结构介绍

- **重要**：对体系结构的理解直接关系到后续的操作和使用



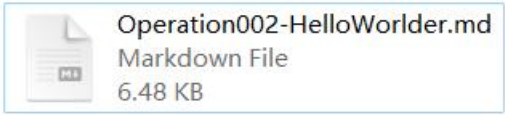
4 RabbitMQ安装

RabbitMQ Installation

Operation001-Install.md
Markdown File
1.04 KB

5 HelloWorld

First Blood



6 RabbitMQ经典用法(工作模式)

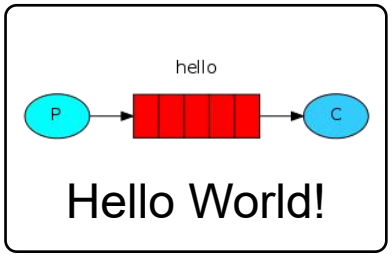
The classic way to use RabbitMQ

总述

- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html

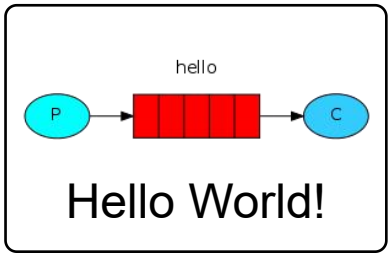
总述

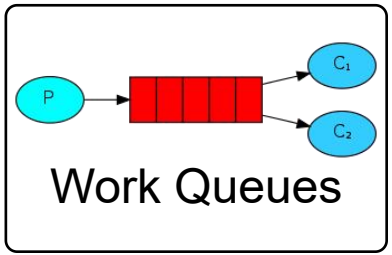
- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html



总述

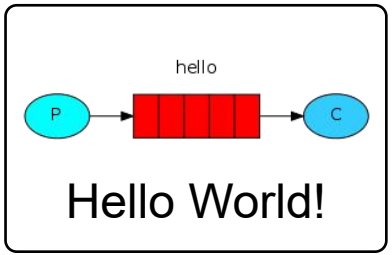
- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html

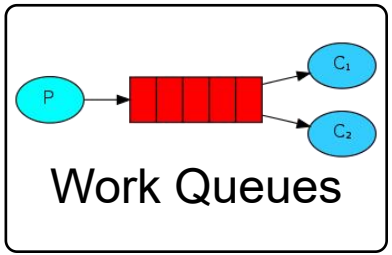


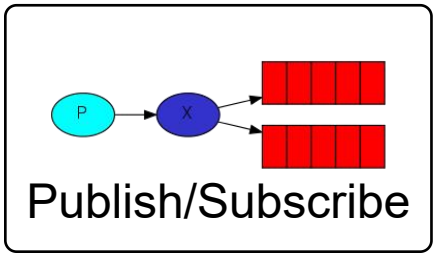


总述

- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html

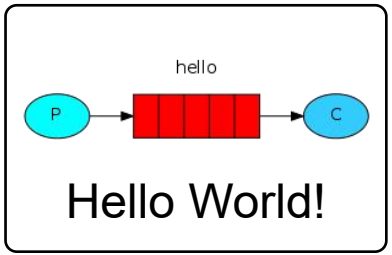


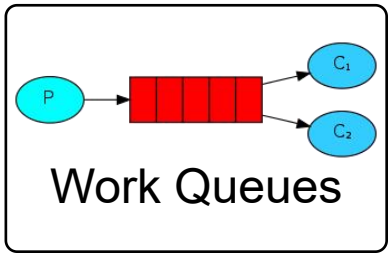


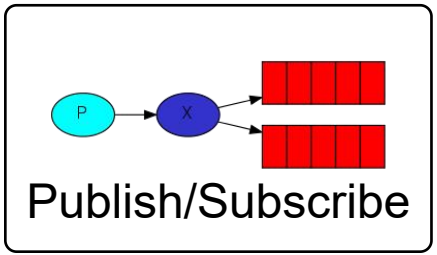


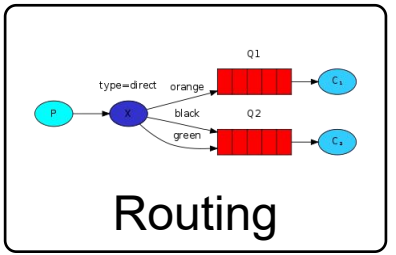
总述

- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html



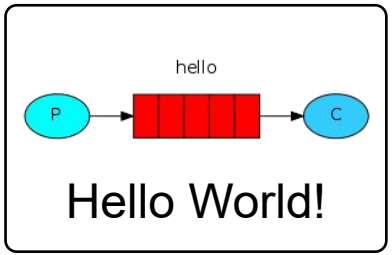


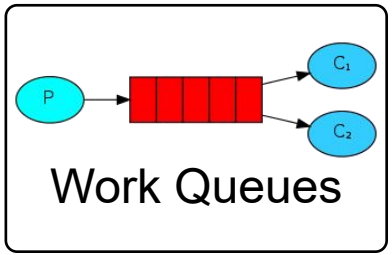


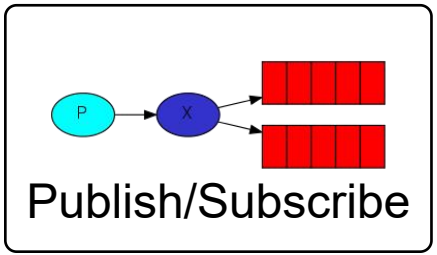


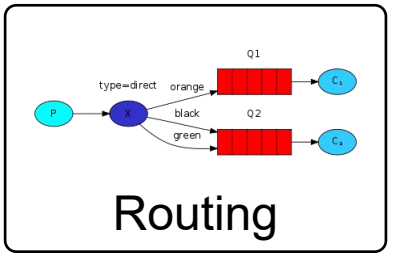
总述

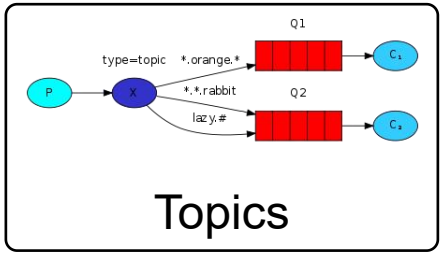
- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html





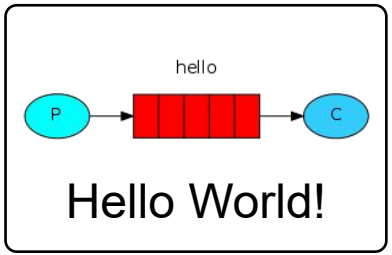


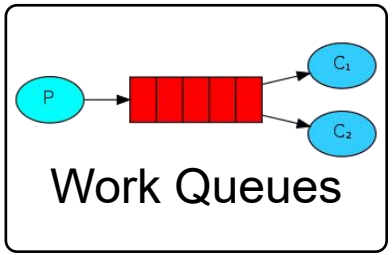


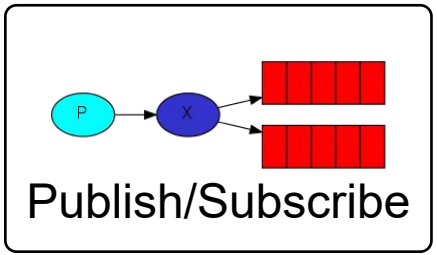


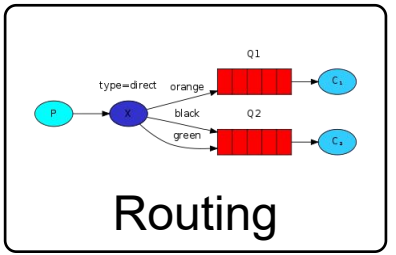
总述

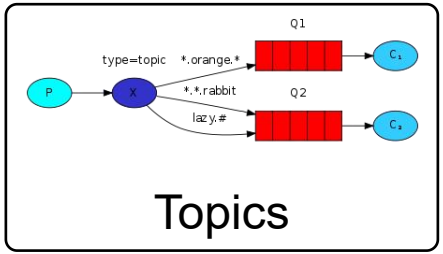
- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html

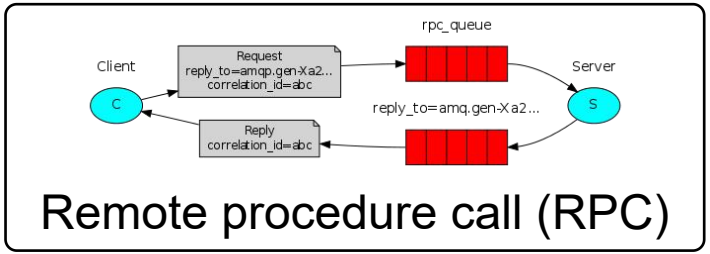






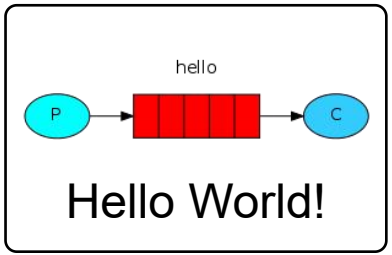


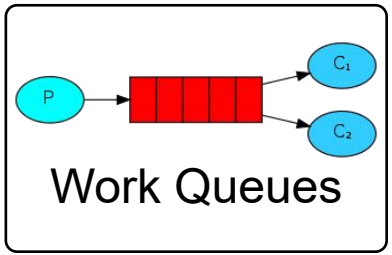


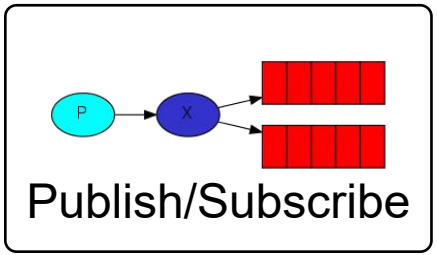


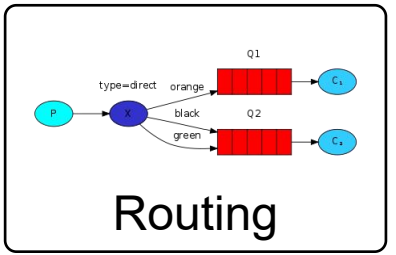
总述

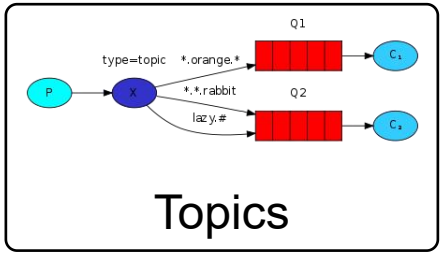
- RabbitMQ官网，通过教程的形式，给我们列举了7种RabbitMQ用法
- 网址：https://www.rabbitmq.com/getstarted.html

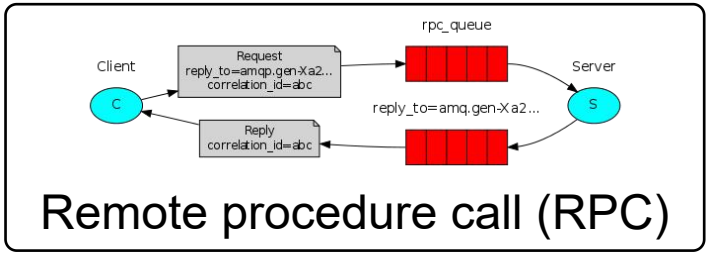


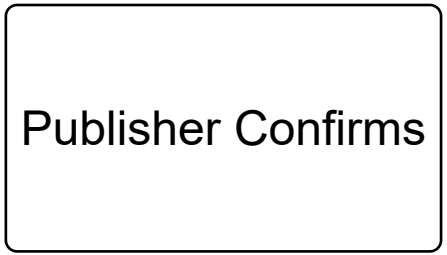












Work Queues

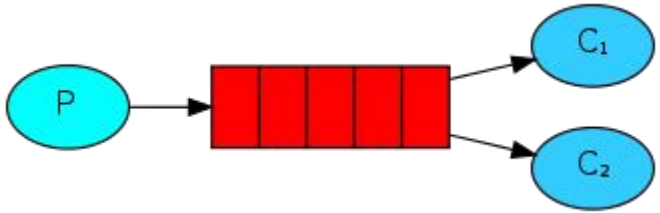
本质上我们刚刚写的HelloWorld程序就是这种模式，只是简化到了最简单的情况：

- 生产者只有一个
- 发送一个消息
- 消费者也只有一个，消息也只能被这个消费者消费

所以HelloWorld也称为简单模式。

现在我们还原一下常规情况：

- 生产者发送多个消息
- 由多个消费者来竞争
- 谁抢到算谁的

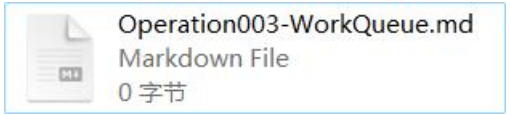


结论：

- 多个消费者监听同一个队列，则各消费者之间对同一个消息是**竞争**的关系。
- Work Queues工作模式适用于任务较重或任务较多的情况，多消费者分摊任务可以提高消息处理的效率。

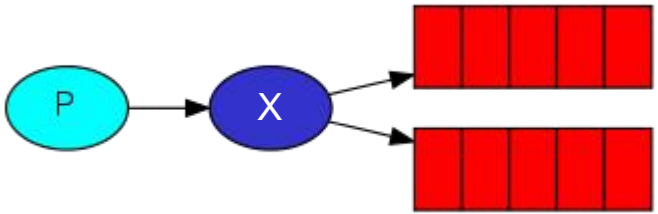
Work Queues

操作文档：

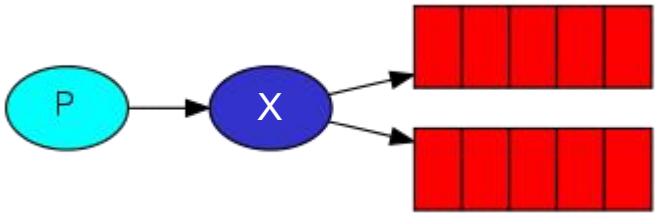


Publish/Subscribe

Publish/Subscribe

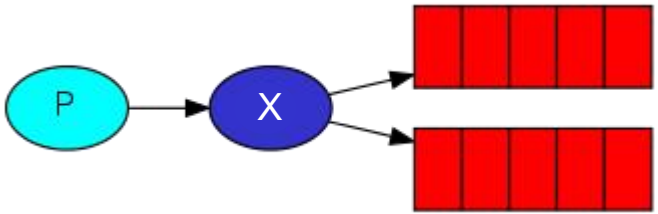


Publish/Subscribe 引入新角色：交换机



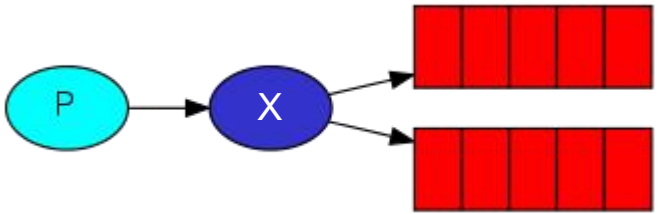
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机



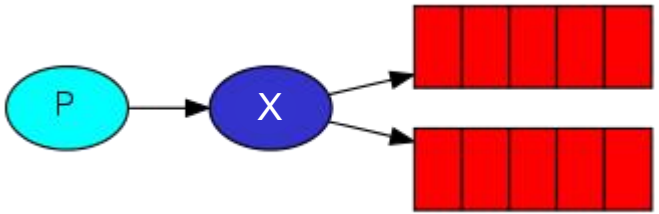
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型



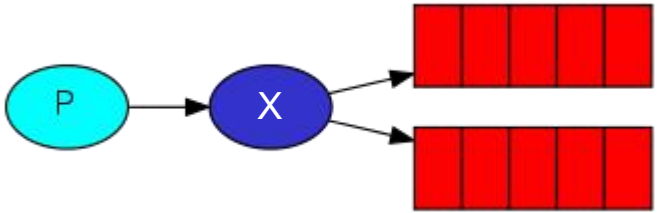
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型
- 交换机有如下3种常见类型



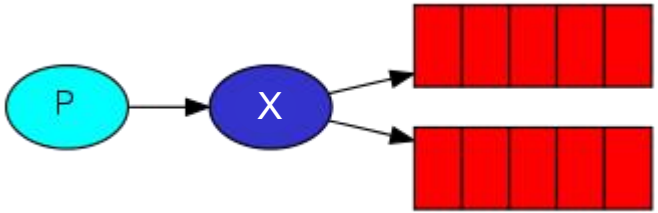
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型
- 交换机有如下3种常见类型
  - Fanout：广播，将消息发送给所有绑定到交换机的队列



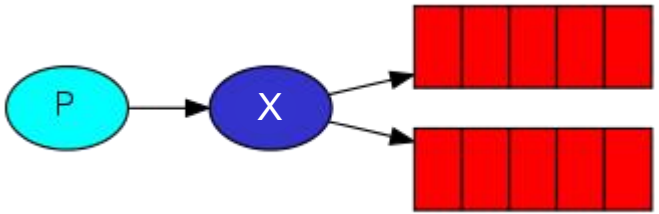
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型
- 交换机有如下3种常见类型
  - Fanout：广播，将消息发送给所有绑定到交换机的队列
  - Direct：定向，把消息交给符合指定routing key的队列



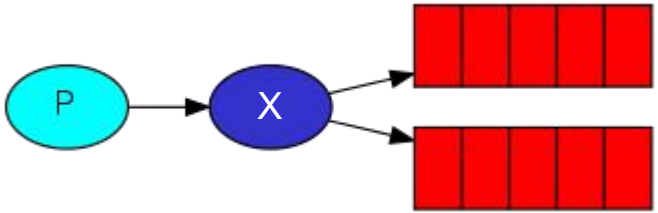
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型
- 交换机有如下3种常见类型
  - Fanout：广播，将消息发送给所有绑定到交换机的队列
  - Direct：定向，把消息交给符合指定routing key的队列
  - Topic：通配符，把消息交给符合routing pattern（路由模式）的队列



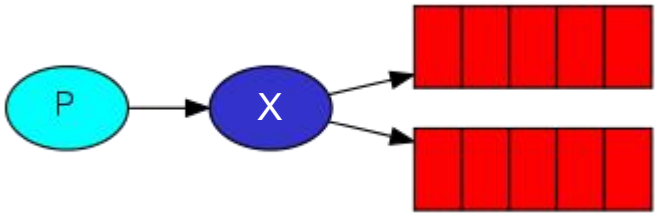
Publish/Subscribe 引入新角色：交换机

- 生产者不是把消息直接发送到队列，而是发送到交换机
- 交换机接收消息，而如何处理消息取决于交换机的类型
- 交换机有如下3种常见类型
  - Fanout：广播，将消息发送给所有绑定到交换机的队列
  - Direct：定向，把消息交给符合指定routing key的队列
  - Topic：通配符，把消息交给符合routing pattern（路由模式）的队列
- 注意：Exchange（交换机）**只负责转发**消息，**不具备存储**消息的能力，因此如果没有任何队列与Exchange绑定，或者没有符合路由规则的队列，那么消息会**丢失**！



Publish/Subscribe 模式说明

- 组件之间关系：
  - 生产者把消息发送到交换机
  - 队列直接和交换机绑定
- 工作机制：消息发送到交换机上，就会以**广播**的形式发送给所有已绑定队列
- 理解概念：
  - Publish：发布，这里就是把消息发送到交换机上
  - Subscribe：订阅，这里只要把队列和交换机绑定，事实上就形成了一种订阅关系

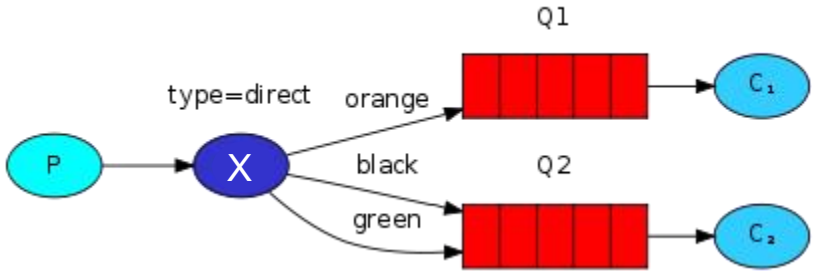


Publish/Subscribe

操作文档：

Routing

- 通过『路由绑定』的方式，把交换机和队列关联起来
- 交换机和队列通过路由键进行绑定
- 生产者发送消息时不仅要指定交换机，还要指定路由键
- 交换机接收到消息会发送到路由键绑定的队列
- 在编码上与 Publish/Subscribe发布与订阅模式的区别：
  - 交换机的类型为：Direct
  - 队列绑定交换机的时候需要指定routing key。



Routing

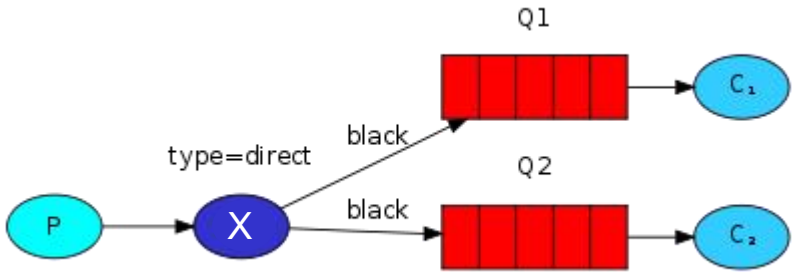
操作文档：

Operation005-Routing.md
Markdown File
0 字节

Routing

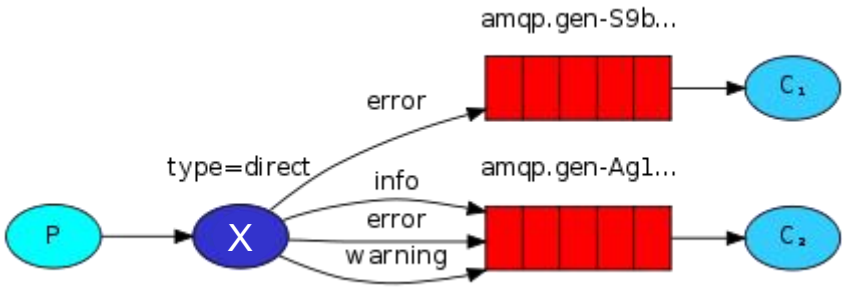
- 如果一个交换机通过相同的routing key绑定了多个队列，就会有广播效果
- 官网说明的原文是：

It is perfectly legal to bind multiple queues with the same binding key. In our example we could add a binding between X and Q1 with binding key black. In that case, the direct exchange will behave like fanout and will broadcast the message to all the matching queues. A message with routing key black will be delivered to both Q1 and Q2.



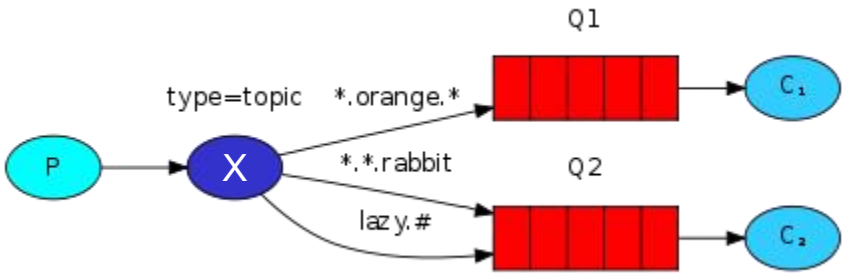
Routing

- 综合起来如下图所示：



Topics

- Topic类型与Direct相比，都是可以根据RoutingKey把消息路由到不同的队列。只不过Topic类型Exchange可以让队列在绑定Routing key的时候使用通配符
- Routingkey一般都是由一个或多个单词组成，多个单词之间以“.”分割，例如：item.insert
- 通配符规则：
  - #：匹配零个或多个词
  - *：匹配一个词

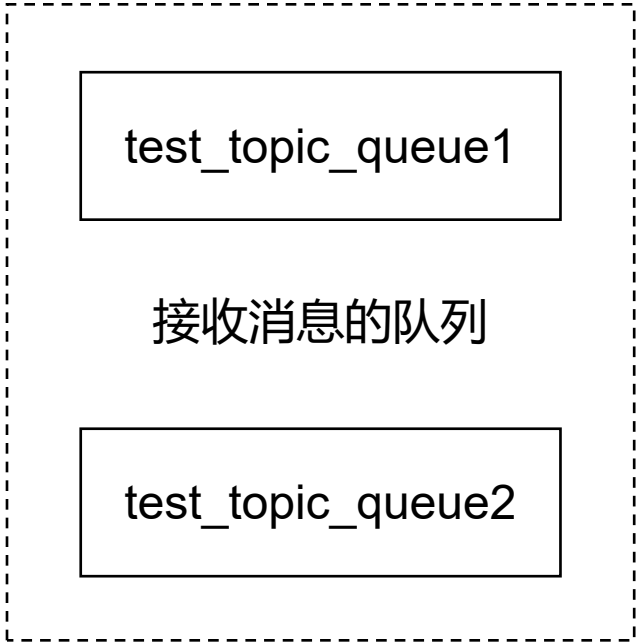


Topics

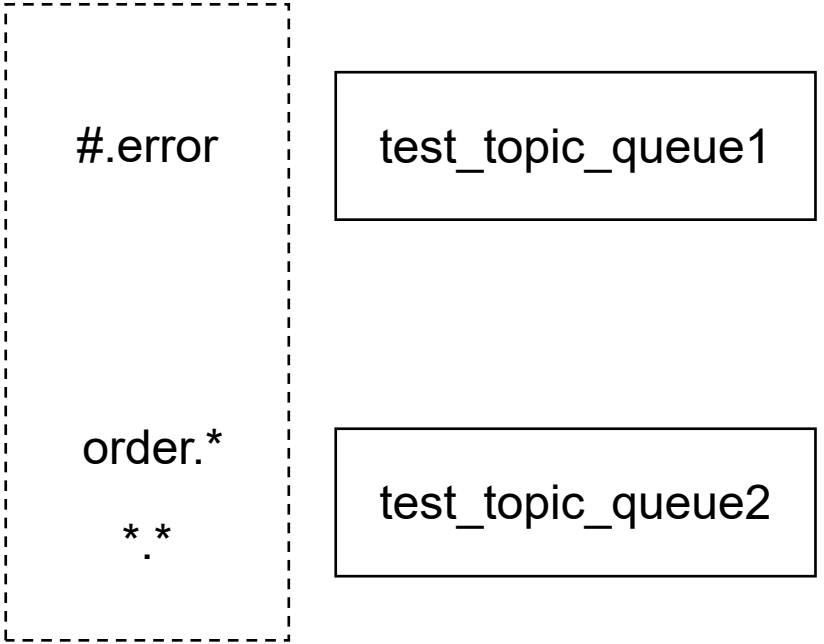
test_topic_queue1

test_topic_queue2

Topics



Topics


#.error
order.*
.
test_topic_queue1
test_topic_queue2
队列绑定的routing key

Topics

#.error

test_topic_queue1

order.info

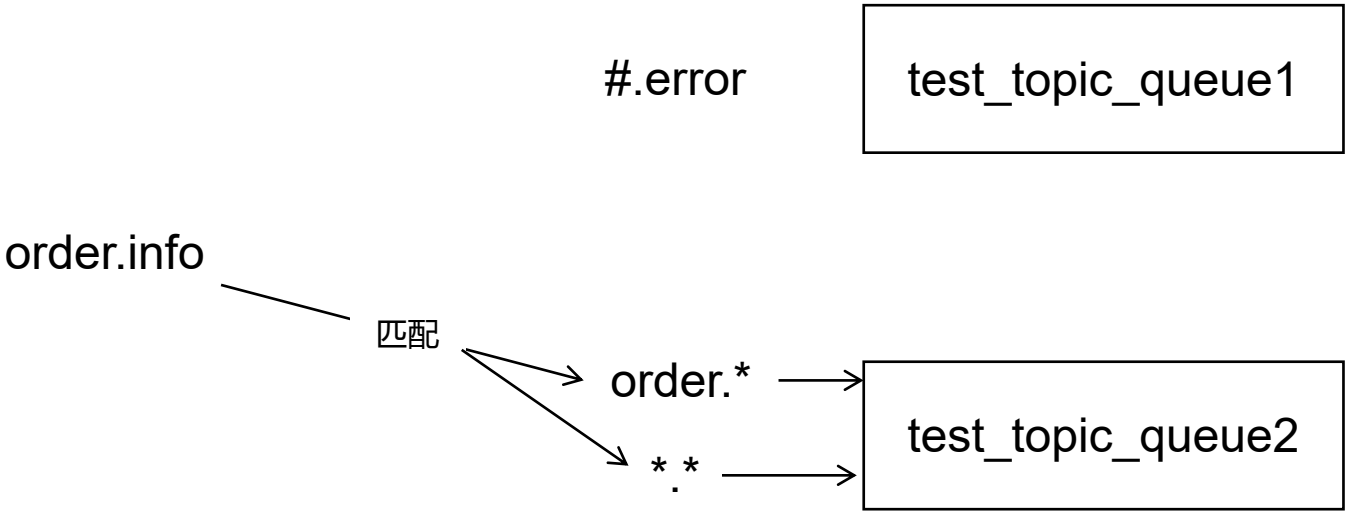
发送消息时指定的routing key

order.*

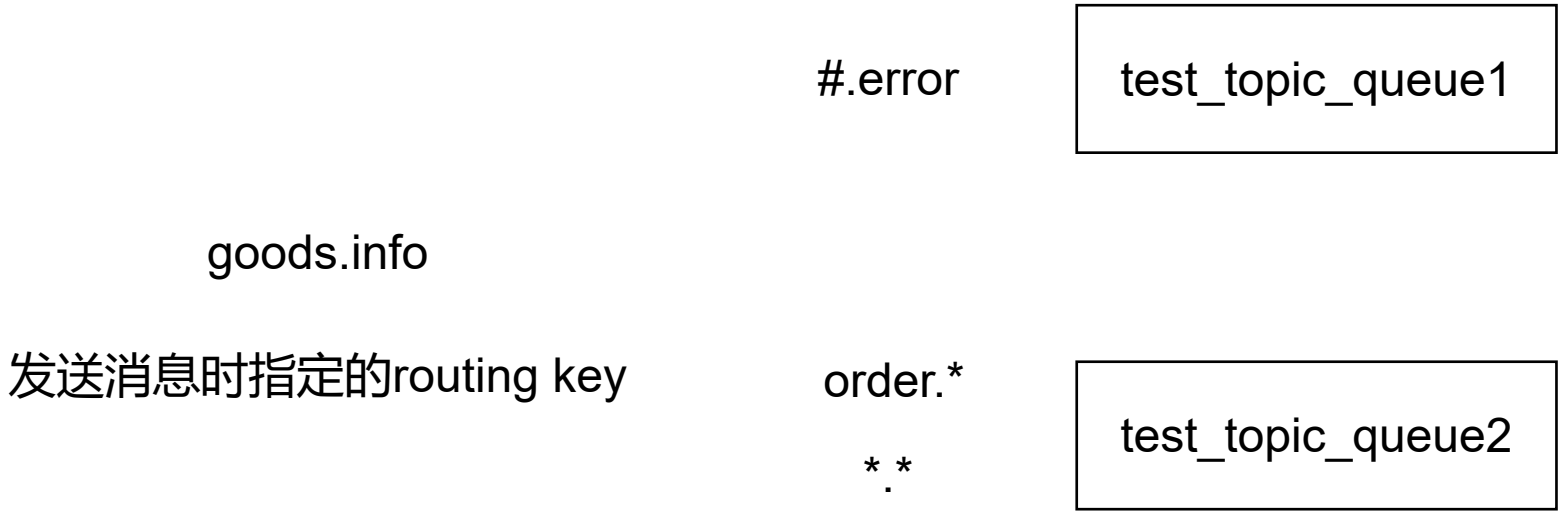
.

test_topic_queue2

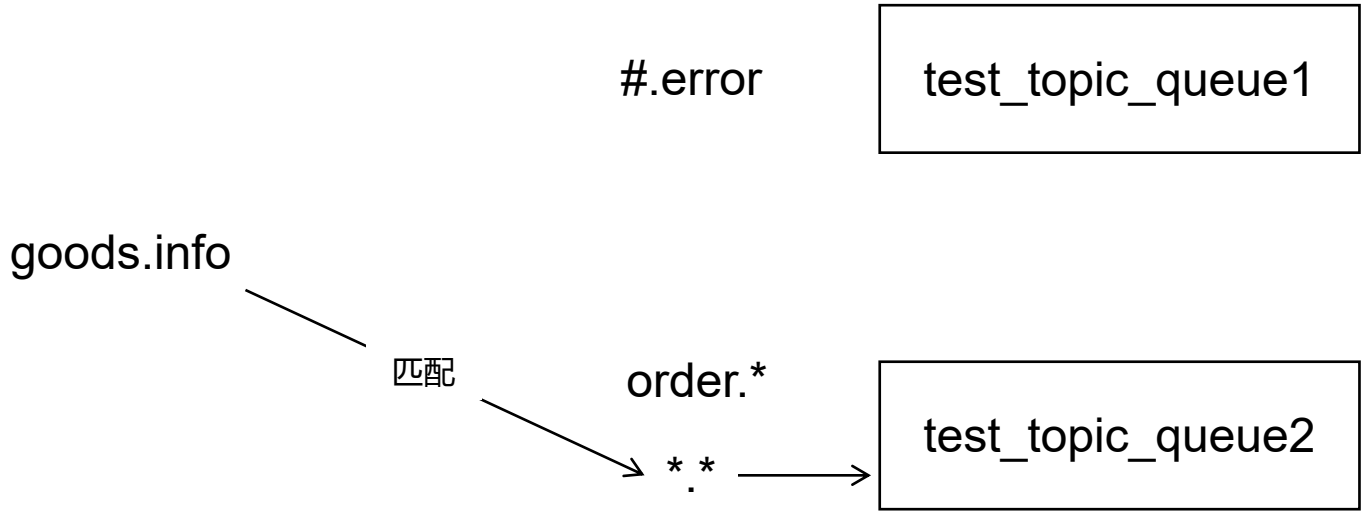
Topics


#.error
test_topic_queue1
order.info
匹配
order.*
.
test_topic_queue2

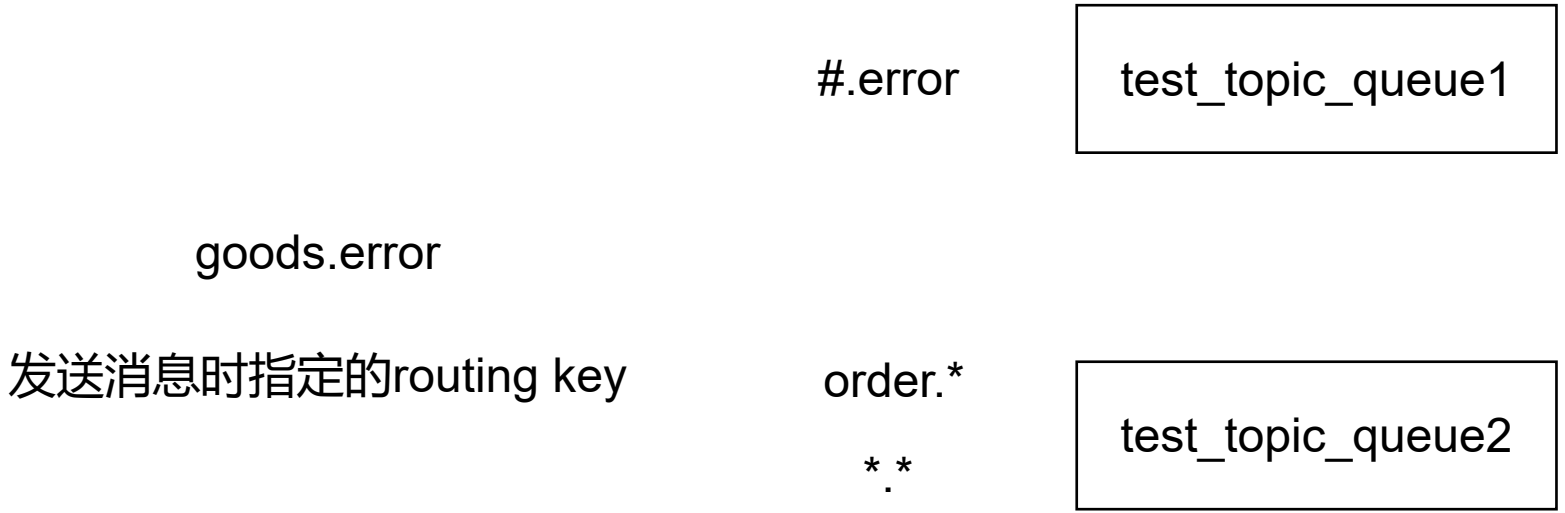
Topics


#.error
test_topic_queue1
goods.info
发送消息时指定的routing key
order.*
.
test_topic_queue2

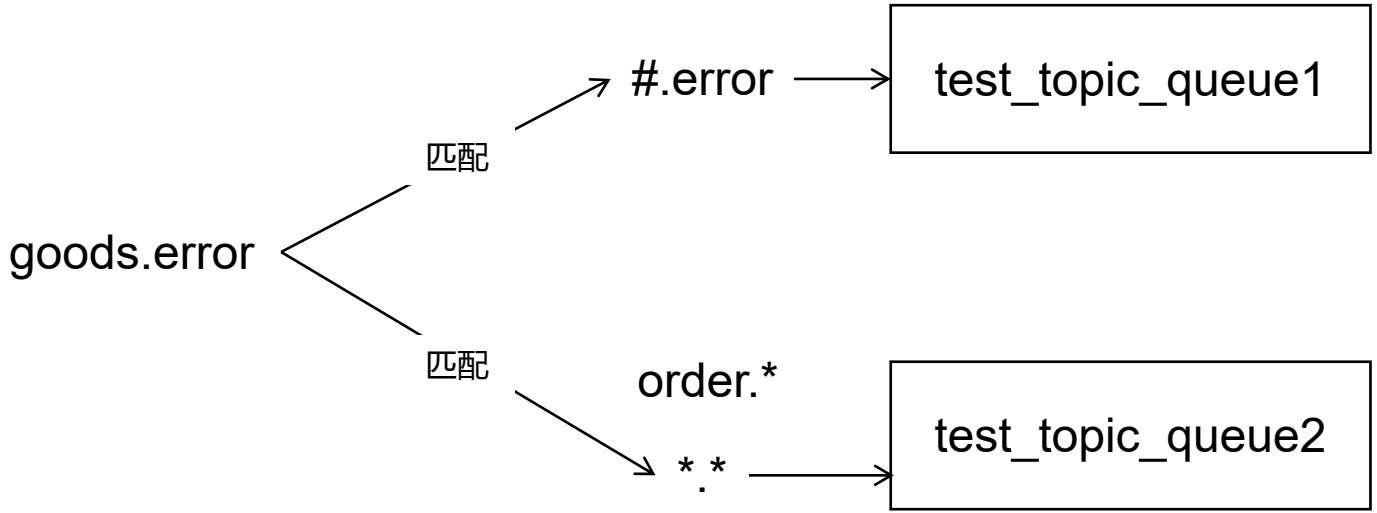
Topics


#.error
test_topic_queue1
goods.info
匹配
order.*
.
test_topic_queue2

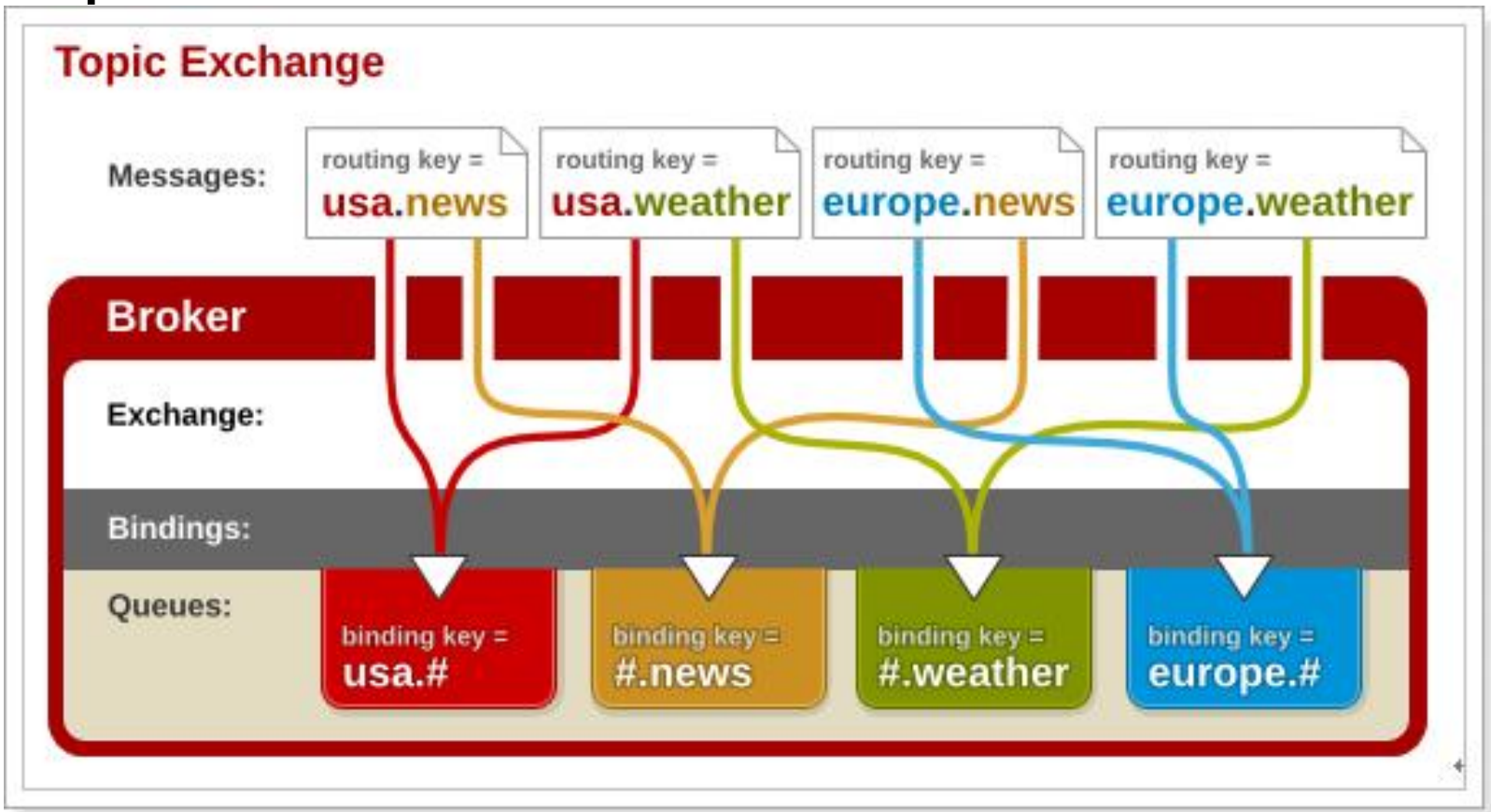
Topics


#.error
test_topic_queue1
goods.error
发送消息时指定的routing key
order.*
.
test_topic_queue2

Topics


#.error
test_topic_queue1
匹配
goods.error
匹配
order.*
.
test_topic_queue2

Topics


Topic Exchange
Messages:
routing key = usa.news
routing key = usa.weather
routing key = europe.news
routing key = europe.weather
Broker
Exchange:
Bindings:
Queues:
binding key = usa.#
binding key = #.news
binding key = #.weather
binding key = europe.#

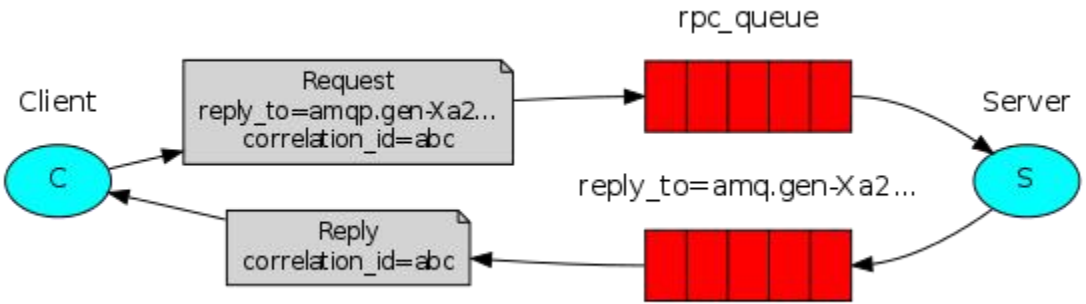
Topics

操作文档：

Operation006-Topics.md
Markdown File
0 字节

RPC

- 远程过程调用，本质上是同步调用，和我们使用OpenFeign调用远程接口一样
- 所以这不是典型的消息队列工作方式，我们就不展开说明了



Publisher Confirms

- 发送端消息确认，是我们在进阶篇要探讨的『消息可靠性投递』的一部分

工作模式小结

- 直接发送到队列：底层使用了默认交换机
- 经过交换机发送到队列
  - Fanout：没有Routing key直接绑定队列
  - Direct：通过Routing key绑定队列，消息发送到绑定的队列上
    - 一个交换机绑定一个队列：定点发送
    - 一个交换机绑定多个队列：广播发送
  - Topic：针对Routing key使用通配符

Part 03
进阶篇

消息可靠性投递、死信队列、延迟消息、惰性队列……

目录

1 客户端整合SpringBoot ★
2 消息可靠性投递 ★
3 消费端限流 ★
4 消息超时 ★
5 死信和死信队列 ★
6 延迟队列 ★
7 事务消息
8 惰性队列
9 优先级队列

尚硅谷

1 RabbitMQ整合SpringBoot

RabbitMQ And SpringBoot Integration

RabbitMQ整合SpringBoot：基本思路

- 搭建环境
- 基础设定：交换机名称、队列名称、绑定关系
- 发送消息：使用RabbitTemplate
- 接收消息：使用@RabbitListener注解

Operation007-SpringBoot.md
Markdown File
0 字节

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener(bindings = @QueueBinding(

    value = @Queue(value = QUEUE_NAME, durable = "true"),

    exchange = @Exchange(value = EXCHANGE_DIRECT),

    key = {ROUTING_KEY}

) )
public void processMessage( String dateString,
                            Message message,
                            Channel channel) {
    System.out.println(dateString);
}
```

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener(bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage( String dateString,
                    Message message,
                    Channel channel) {
    System.out.println(dateString);
}
```

设定基本信息的注解

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener(bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage( String dateString,
                            Message message,
                            Channel channel) {
    System.out.println(dateString);
}
```

监听并且接收消息的方法

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener( bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage( String dateString,
                            Message message,

                            Channel channel ) {

    System.out.println(dateString);
}
```

← 消息数据本身

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener( bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage( String dateString,

                    Message message,

                    Channel channel ) {

    System.out.println(dateString);
}
```

消息对象

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener( bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage( String dateString,

                    Message message,

                    Channel channel ) {

    System.out.println(dateString);
}
```

频道对象 → Channel channel

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener( bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
public void processMessage(String dateString,Message message,Channel channel) {

    System.out.println(dateString);
}
```

RabbitMQ整合SpringBoot：接收消息

```
@RabbitListener( bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
) )
```

RabbitMQ整合SpringBoot：接收消息

```
bindings = @QueueBinding(
    value = @Queue(value = QUEUE_NAME, durable = "true"),
    exchange = @Exchange(value = EXCHANGE_DIRECT),
    key = {ROUTING_KEY}
)
```

RabbitMQ整合SpringBoot：接收消息

value = @Queue(value = QUEUE_NAME, durable = "true"),

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}

RabbitMQ整合SpringBoot：接收消息

指定队列信息

value = @Queue(value = QUEUE_NAME, durable = "true"),

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}

RabbitMQ整合SpringBoot：接收消息

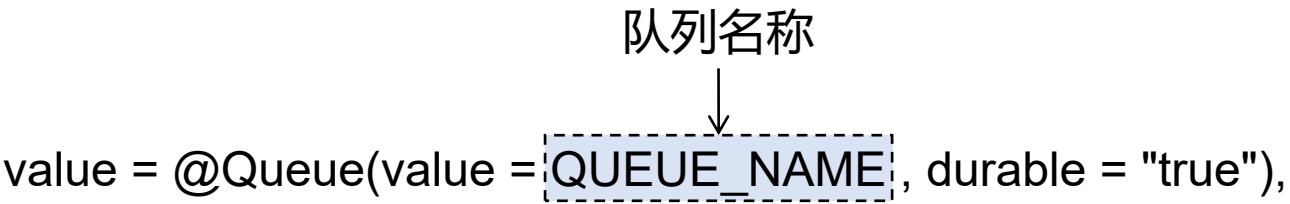
给value属性赋值

```
value = @Queue(value = QUEUE_NAME, durable = "true"),

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}
```

RabbitMQ整合SpringBoot：接收消息



RabbitMQ整合SpringBoot：接收消息

持久化（后面说）

```
value = @Queue(value = QUEUE_NAME , durable = "true"),

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}
```

RabbitMQ整合SpringBoot：接收消息

value = @Queue(value = QUEUE_NAME , durable = "true"),

指定交换机信息 ⟶ exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}

RabbitMQ整合SpringBoot：接收消息

value = @Queue(value = QUEUE_NAME, durable = "true"),

给exchange属性赋值

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}

RabbitMQ整合SpringBoot：接收消息

```
value = @Queue(value = QUEUE_NAME , durable = "true" ),

exchange = @Exchange(value = EXCHANGE_DIRECT ),

key = {ROUTING_KEY}
```

交换机名称

RabbitMQ整合SpringBoot：接收消息

value = @Queue(value = QUEUE_NAME , durable = "true"),

exchange = @Exchange(value = EXCHANGE_DIRECT),

指定路由键信息 → key = {ROUTING_KEY}

RabbitMQ整合SpringBoot：接收消息

```
value = @Queue(value = QUEUE_NAME , durable = "true" ),

exchange = @Exchange(value = EXCHANGE_DIRECT),

key = {ROUTING_KEY}
```

字符串数组

RabbitMQ整合SpringBoot：发送消息

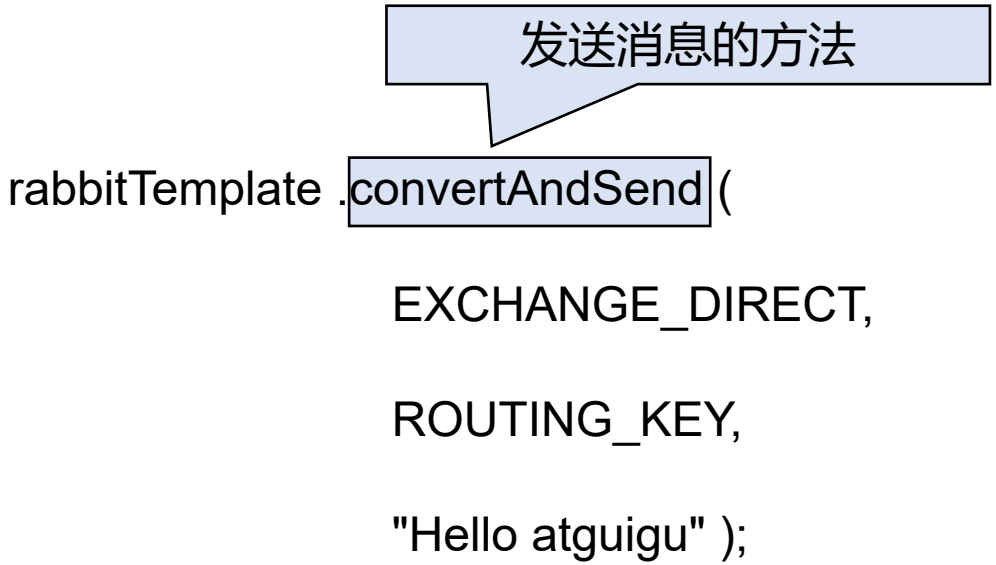
```
rabbitTemplate .convertAndSend (
                    EXCHANGE_DIRECT,
                    ROUTING_KEY,
                    "Hello atguigu" );
```

RabbitMQ整合SpringBoot：发送消息

@Autowired装配进来使用

```
rabbitTemplate.convertAndSend (
                EXCHANGE_DIRECT,
                ROUTING_KEY,
                "Hello atguigu" );
```

RabbitMQ整合SpringBoot：发送消息



RabbitMQ整合SpringBoot：发送消息

```
rabbitTemplate .convertAndSend (
        EXCHANGE_DIRECT,
        ROUTING_KEY,
        "Hello atguigu" );
```

交换机名称

RabbitMQ整合SpringBoot：发送消息

```
rabbitTemplate .convertAndSend (
        EXCHANGE_DIRECT,
        ROUTING_KEY,
        "Hello atguigu" );
```

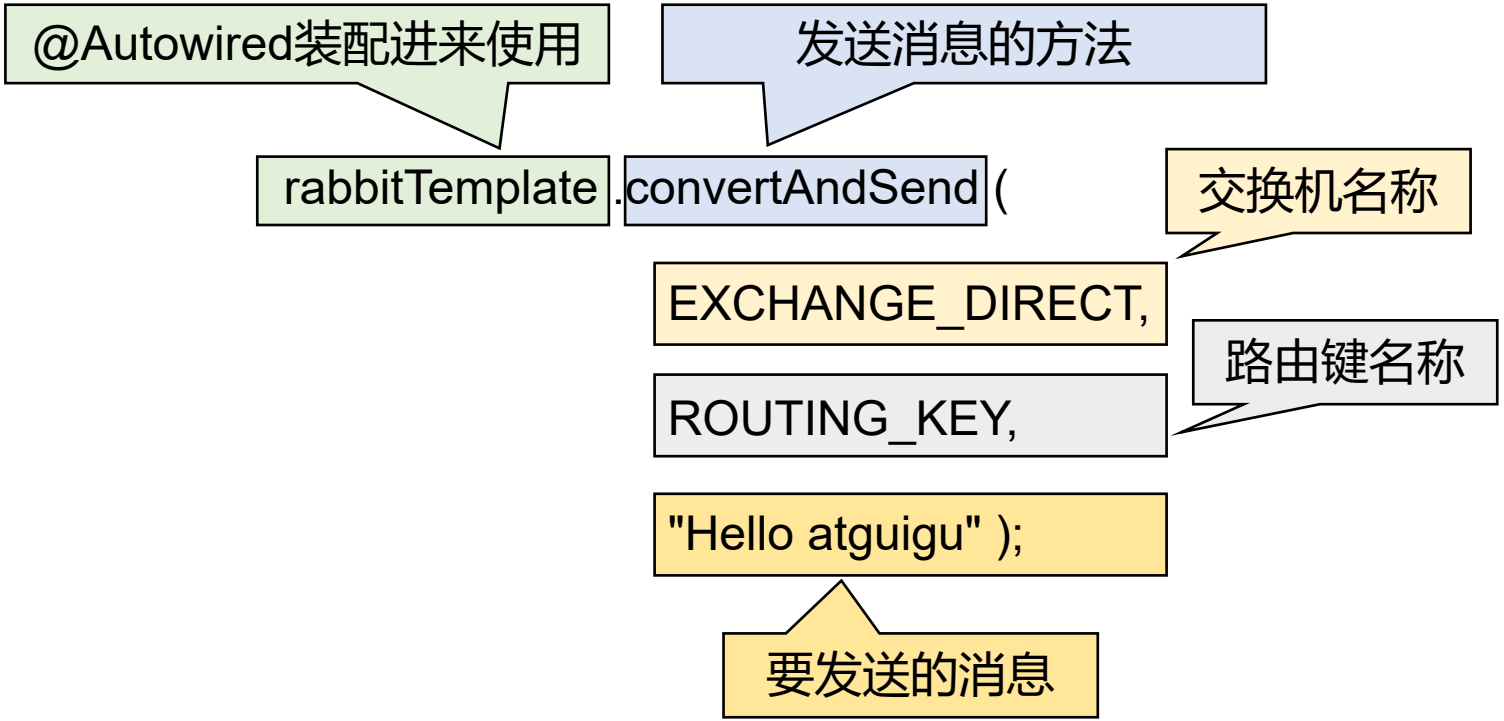
路由键名称

RabbitMQ整合SpringBoot：发送消息

```
rabbitTemplate .convertAndSend (
                    EXCHANGE_DIRECT,
                    ROUTING_KEY,
                    "Hello atguigu" );
```

要发送的消息

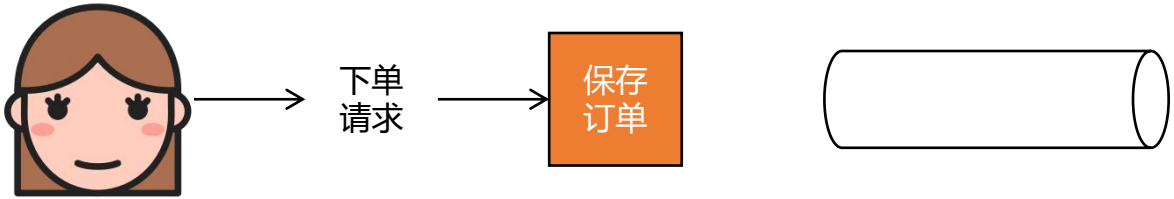
RabbitMQ整合SpringBoot：发送消息



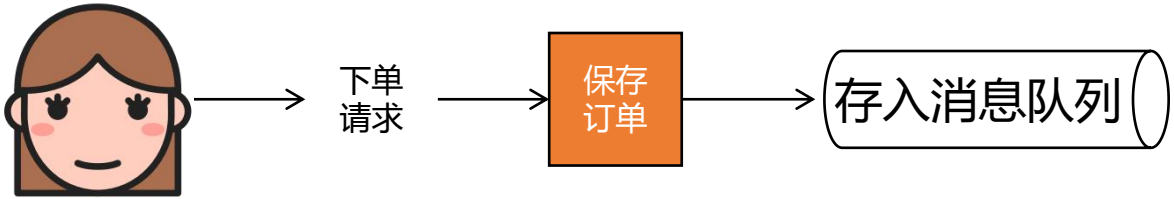
2 消息可靠性投递

Message reliability delivery

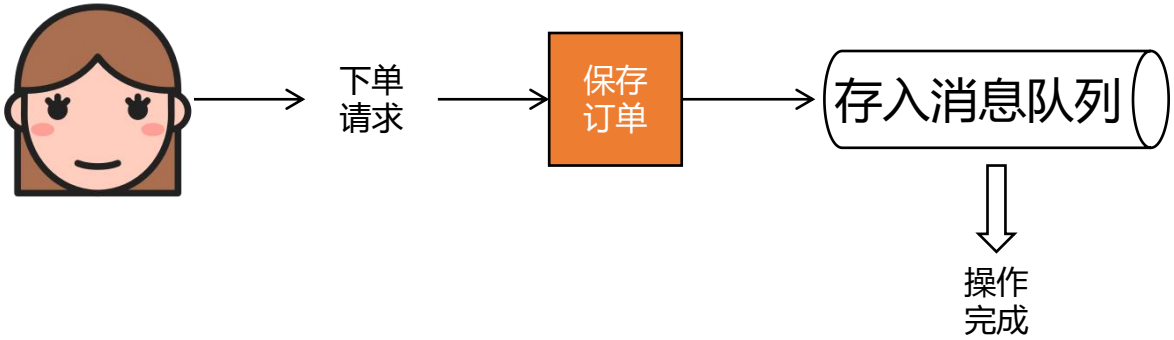
提出问题：下单操作的正常流程



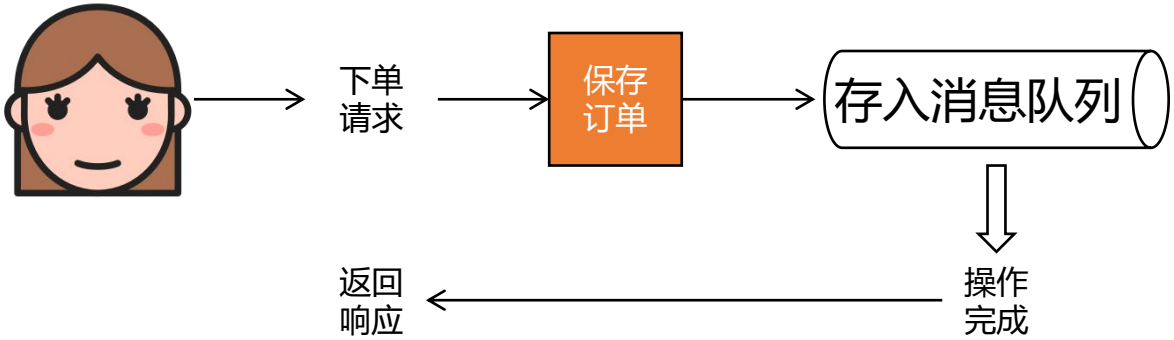
提出问题：下单操作的正常流程



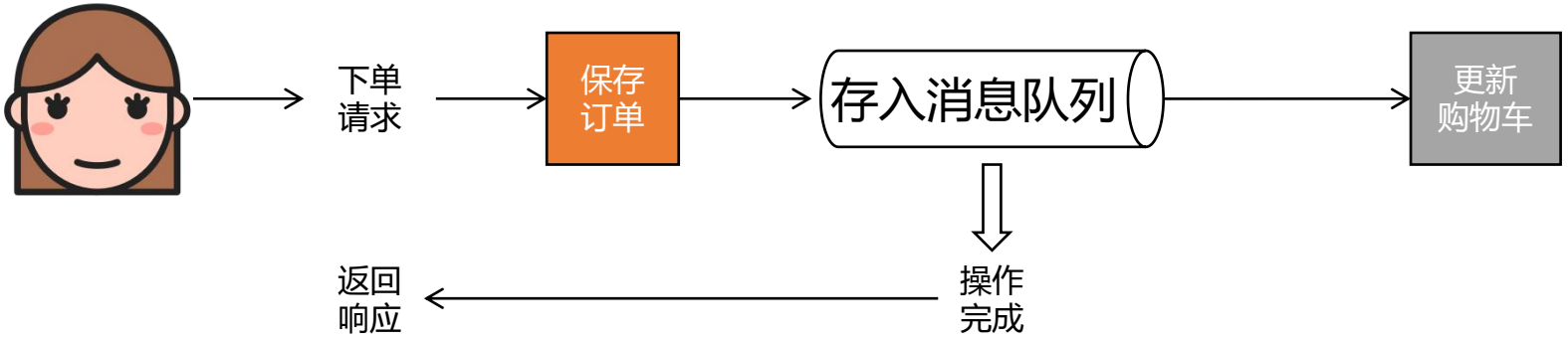
提出问题：下单操作的正常流程



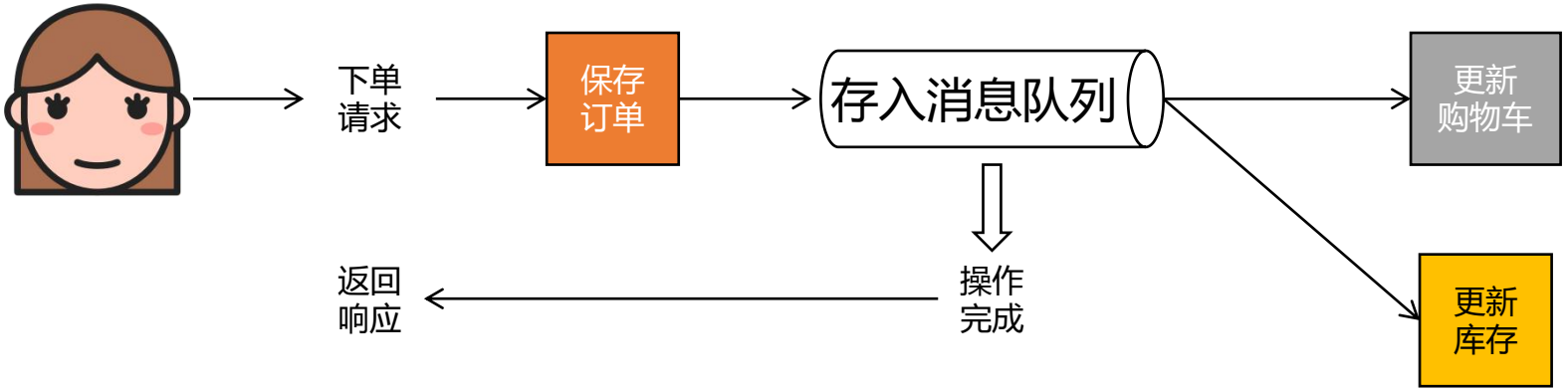
提出问题：下单操作的正常流程



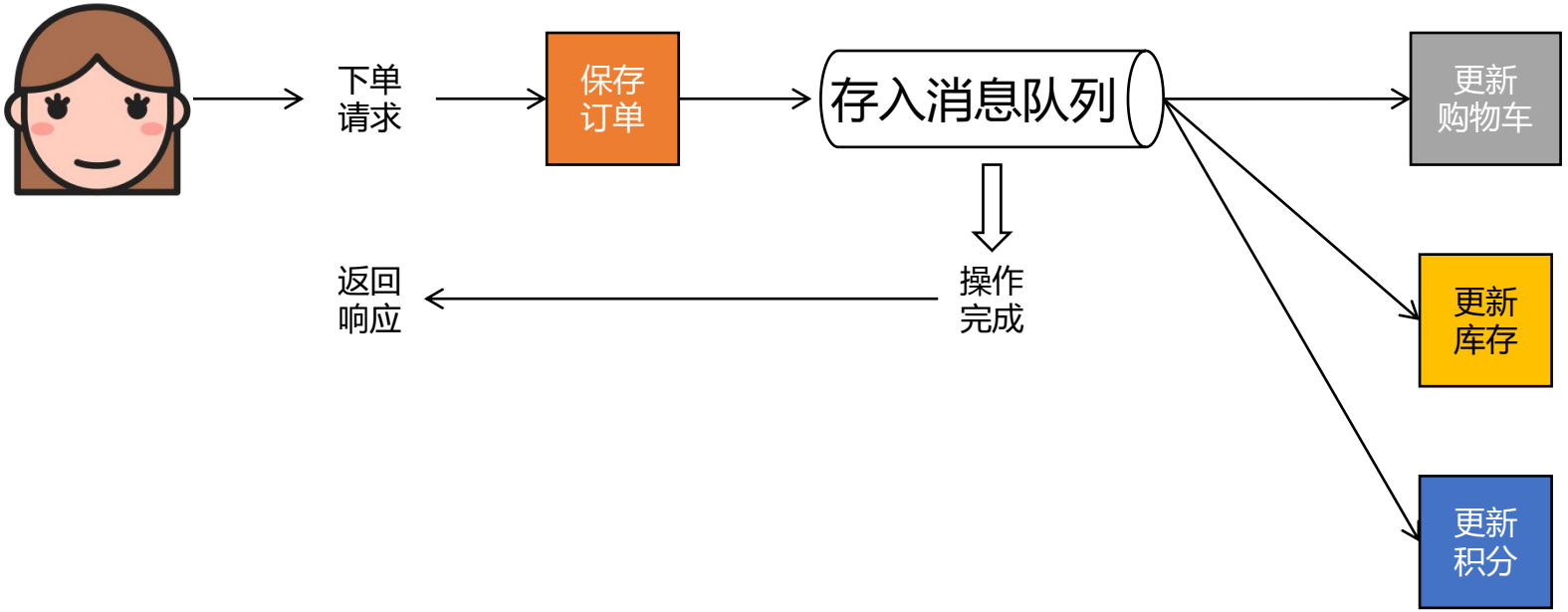
提出问题：下单操作的正常流程



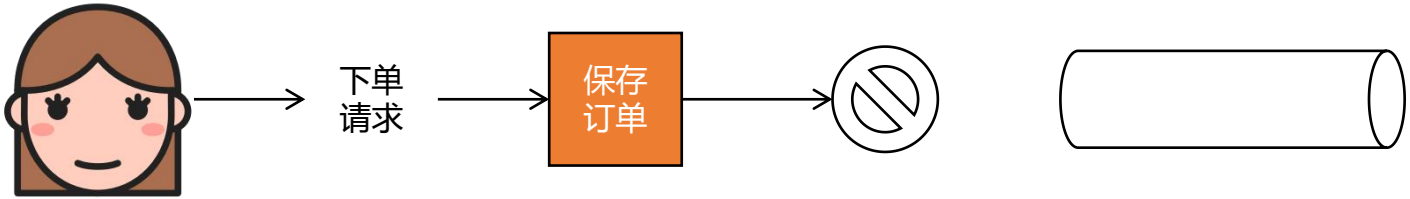
提出问题：下单操作的正常流程



提出问题：下单操作的正常流程



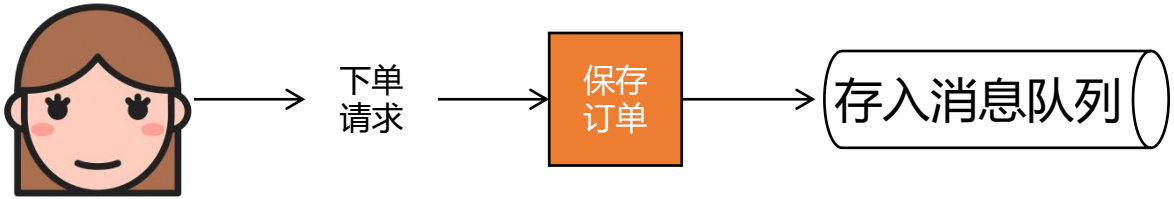
提出问题：故障情况1



消息没有发送到消息队列上

后果：消费者拿不到消息，业务功能缺失，数据错误

提出问题：故障情况2



消息成功存入消息队列

提出问题：故障情况2

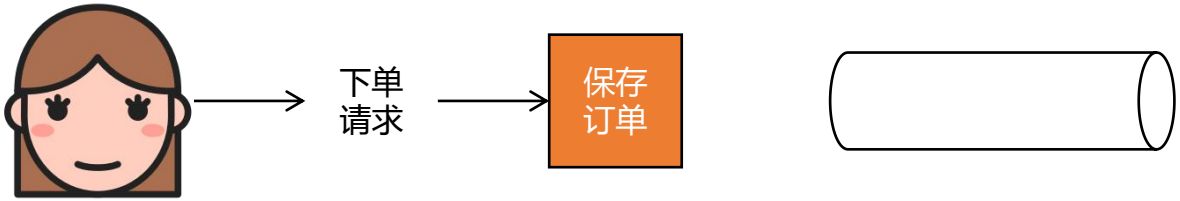
消息成功存入消息队列，但是消息队列服务器宕机了

提出问题：故障情况2

消息成功存入消息队列，但是消息队列服务器宕机了

原本保存在**内存中的消息**也**丢失**了

提出问题：故障情况2

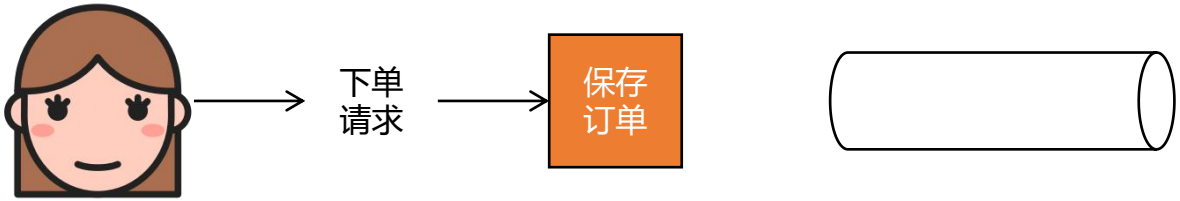


消息成功存入消息队列，但是消息队列服务器宕机了

原本保存在**内存中的消息**也**丢失**了！！！

即使服务器重新启动，消息也找不回来了

提出问题：故障情况2



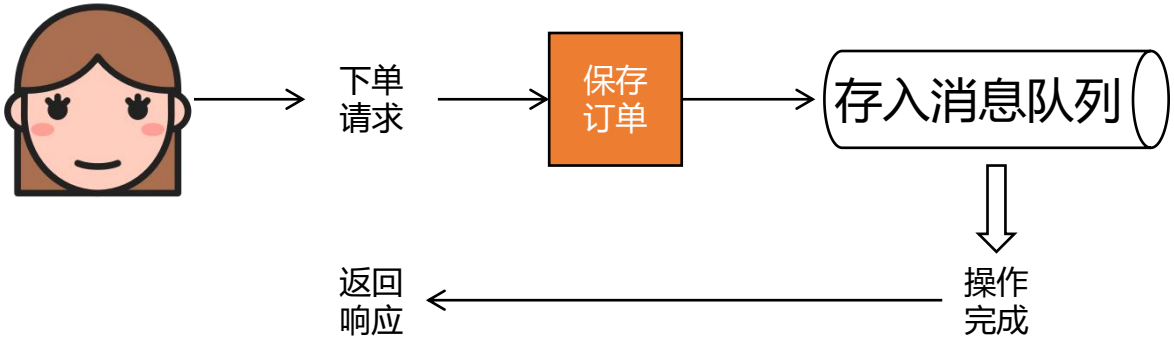
消息成功存入消息队列，但是消息队列服务器宕机了

原本保存在**内存中的消息**也**丢失**了！！！

即使服务器重新启动，消息也找不回来了

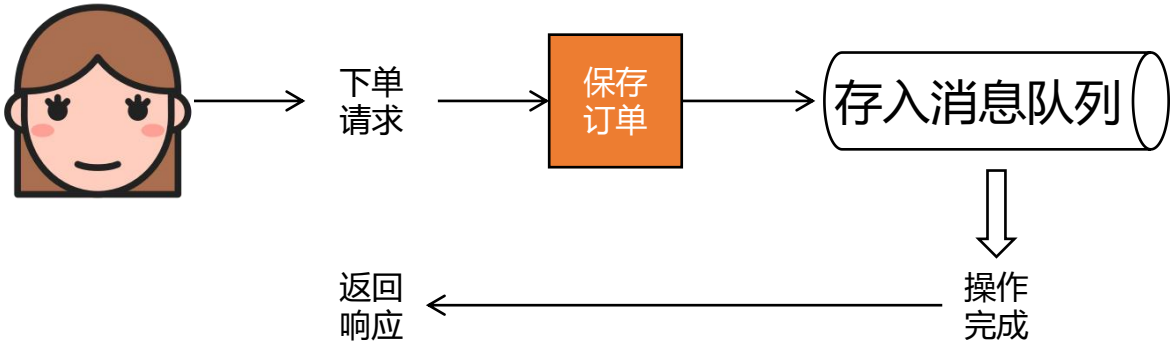
后果：消费者拿不到消息，业务功能缺失，数据错误

提出问题：故障情况3



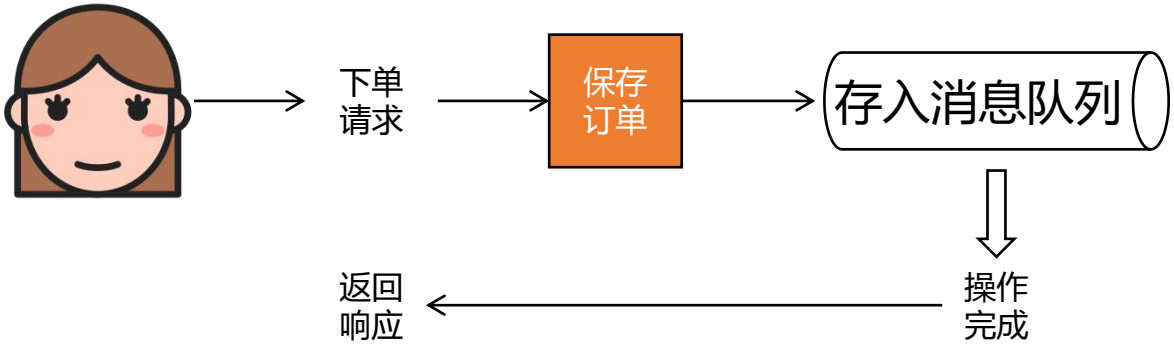
消息成功存入消息队列

提出问题：故障情况3



消息成功存入消息队列，但是消费端出现问题，例如：宕机、抛异常等等

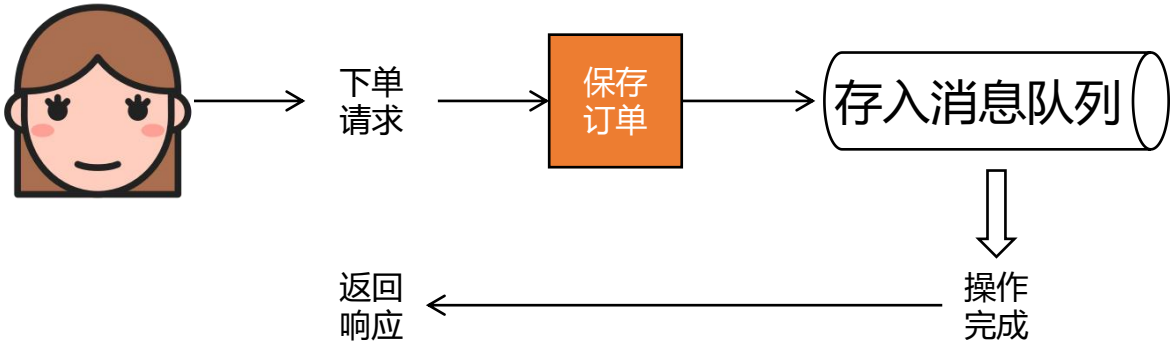
提出问题：故障情况3



宕机

消息成功存入消息队列，但是消费端出现问题，例如：宕机、抛异常等等

提出问题：故障情况3



宕机

消息成功存入消息队列，但是消费端出现问题，例如：宕机、抛异常等等

后果：业务功能缺失，数据错误

对症下药

对症下药

- 故障情况1：消息没有发送到消息队列

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失
  - 解决思路：**消息持久化**到硬盘上，哪怕服务器重启也不会导致消息丢失

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失
  - 解决思路：**消息持久化**到硬盘上，哪怕服务器重启也不会导致消息丢失
- 故障情况3：消费端宕机或抛异常导致消息没有成功被消费

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失
  - 解决思路：**消息持久化**到硬盘上，哪怕服务器重启也不会导致消息丢失
- 故障情况3：消费端宕机或抛异常导致消息没有成功被消费
  - 消费端消费消息**成功**，给服务器返回**ACK信息**，然后消息队列删除该消息

对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失
  - 解决思路：**消息持久化**到硬盘上，哪怕服务器重启也不会导致消息丢失
- 故障情况3：消费端宕机或抛异常导致消息没有成功被消费
  - 消费端消费消息**成功**，给服务器返回**ACK信息**，然后消息队列删除该消息
  - 消费端消费消息**失败**，给服务器端返回**NACK信息**，同时把消息恢复为**待消费**的状态，这样就可以再次取回消息，**重试**一次（当然，这就需要消费端接口支持幂等性）

对症下药

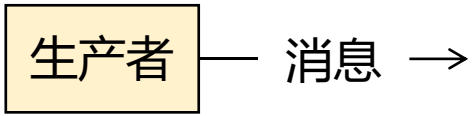
- 故障情况1应对方式操作文档：Operation008-Confirm01-A-Producer.md
- 故障情况1应对方式操作文档：Operation008-Confirm01-B-BackupEx.md
- 故障情况2应对方式操作文档：Operation008-Confirm02-Duration.md
- 故障情况3应对方式操作文档：Operation008-Confirm03-Consumer.md

备份交换机原理

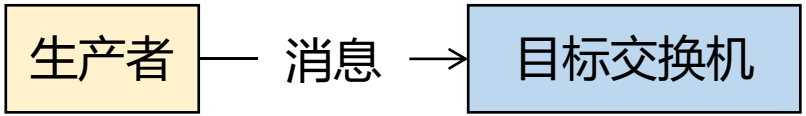
备份交换机原理

生产者

备份交换机原理

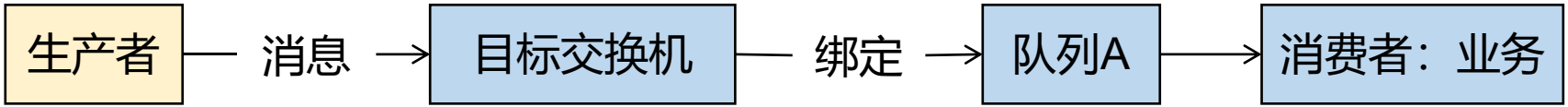


备份交换机原理

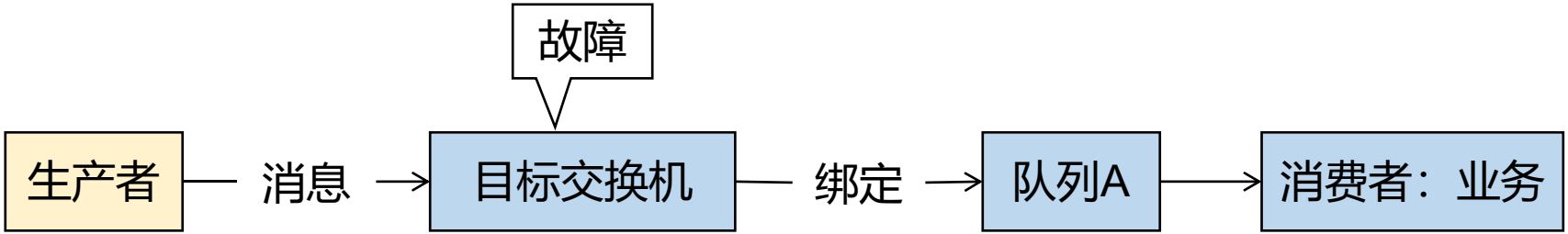


备份交换机原理

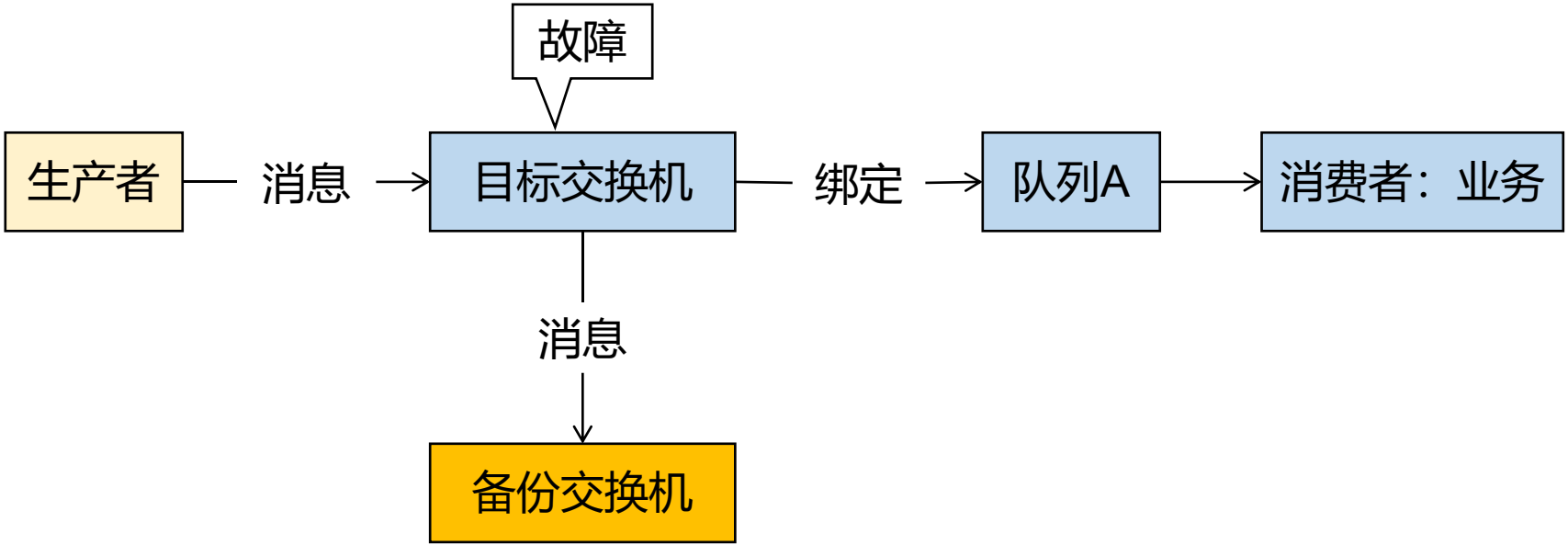
备份交换机原理



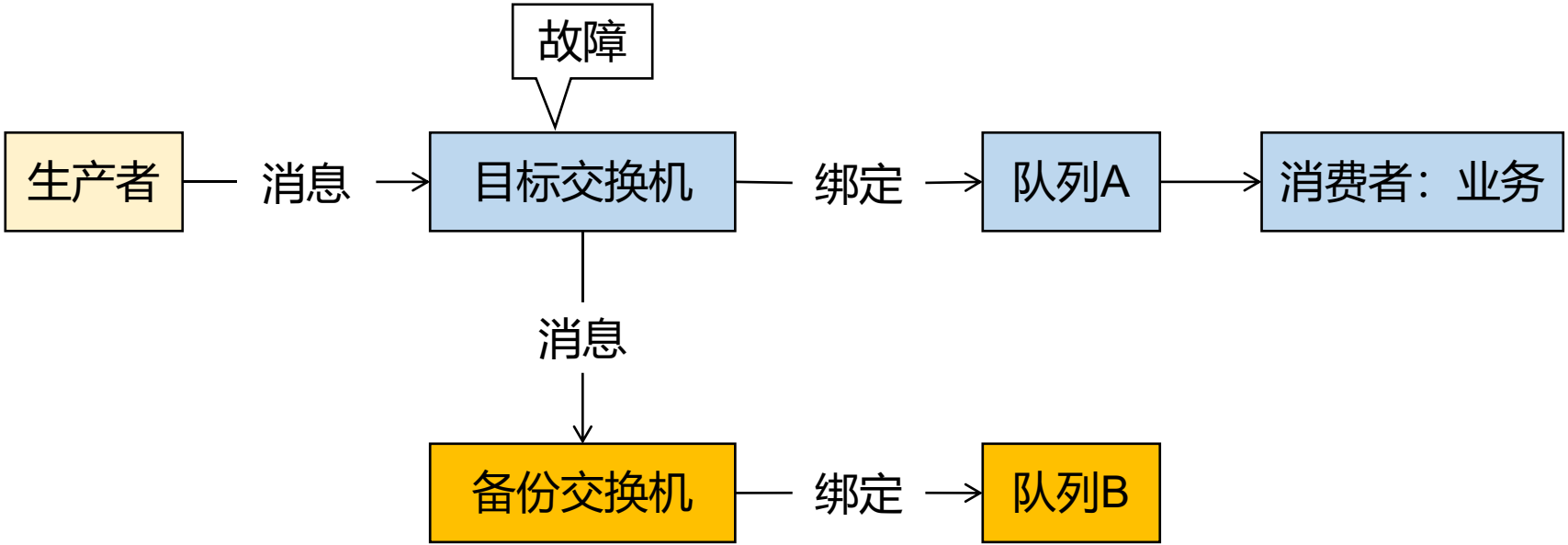
备份交换机原理



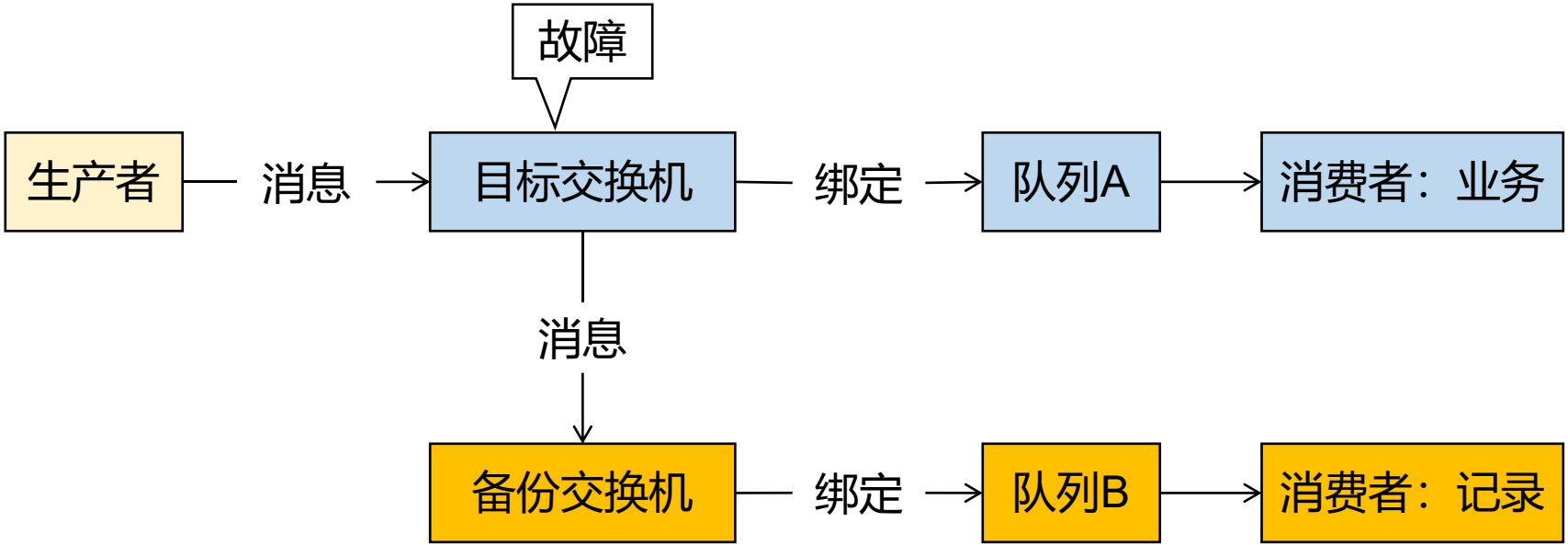
备份交换机原理



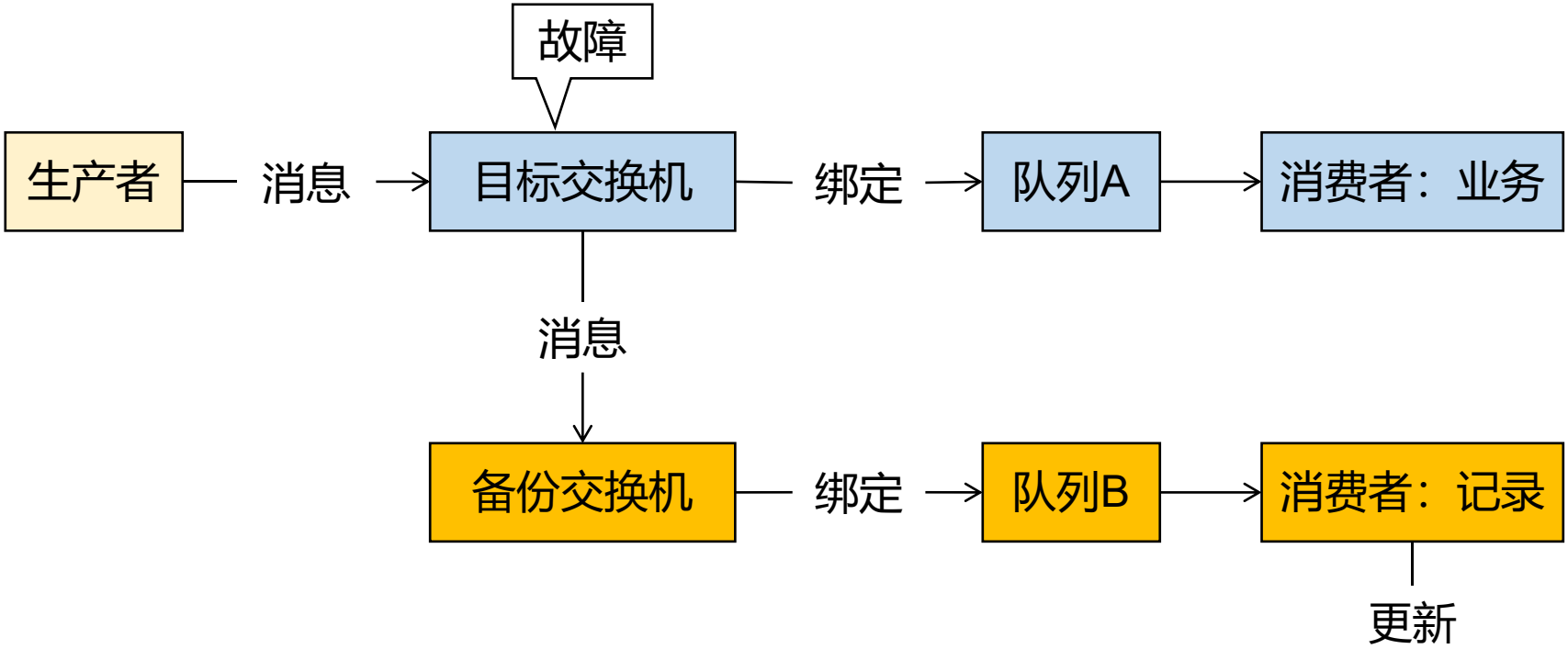
备份交换机原理



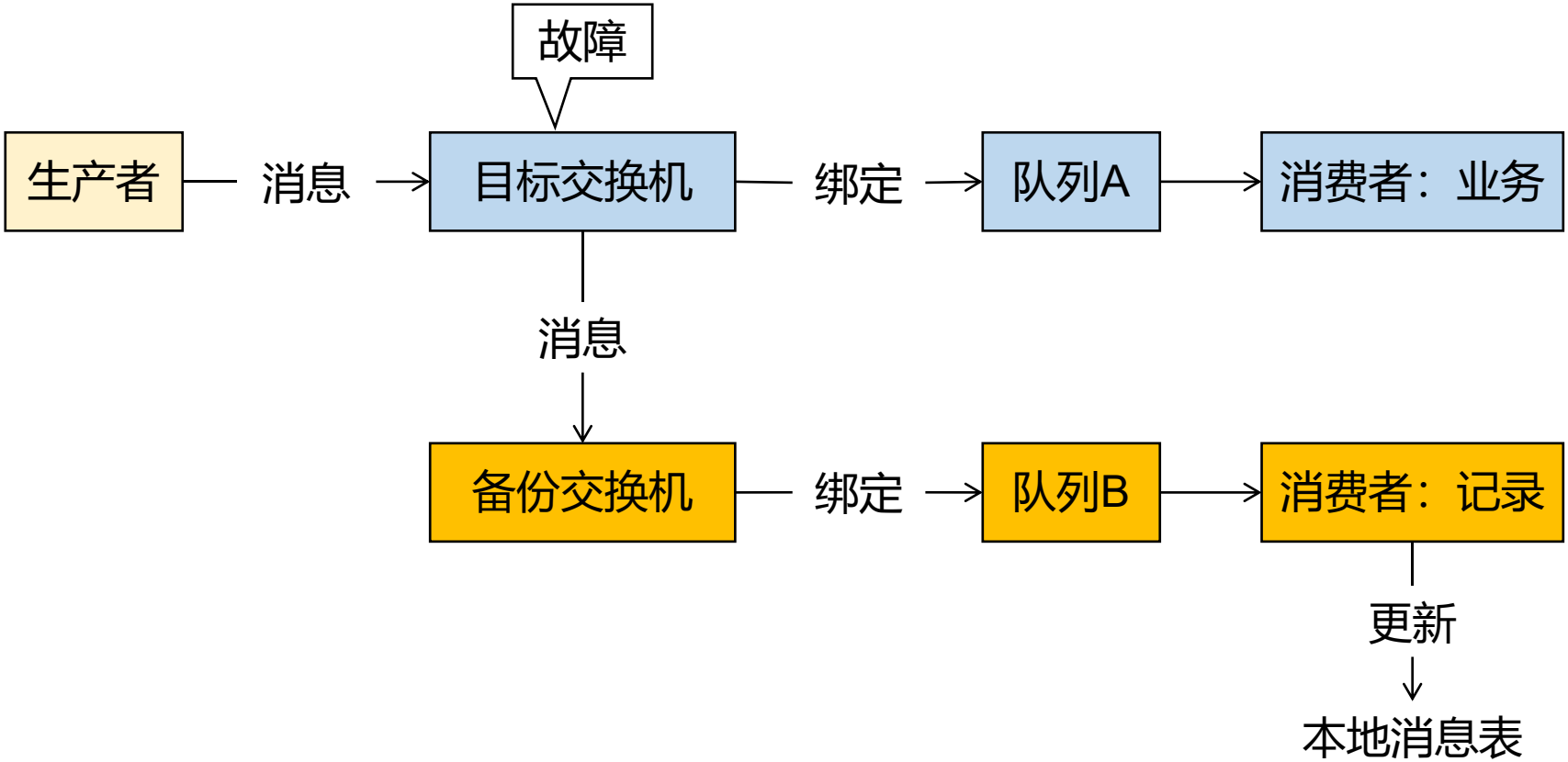
备份交换机原理



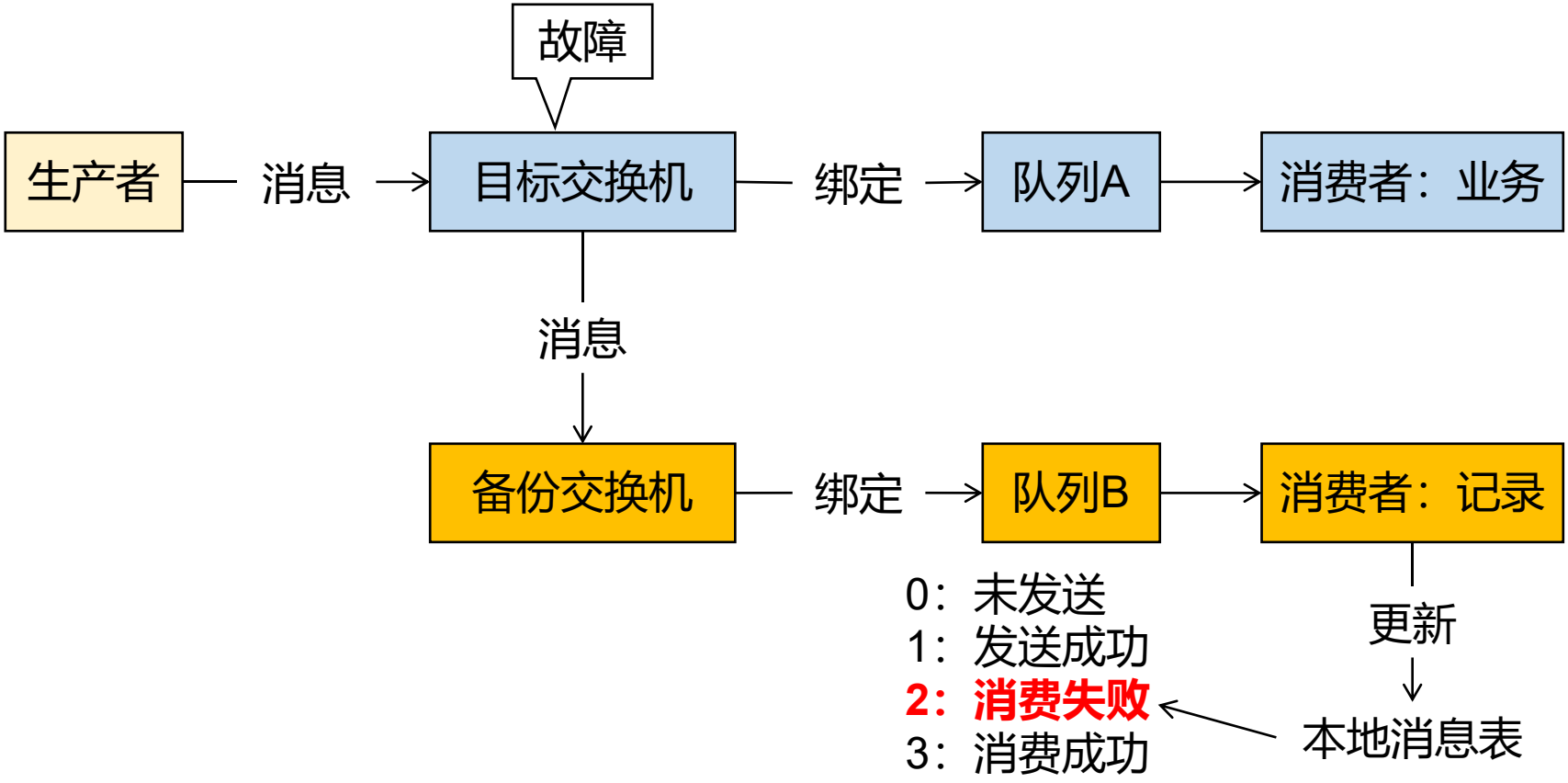
备份交换机原理



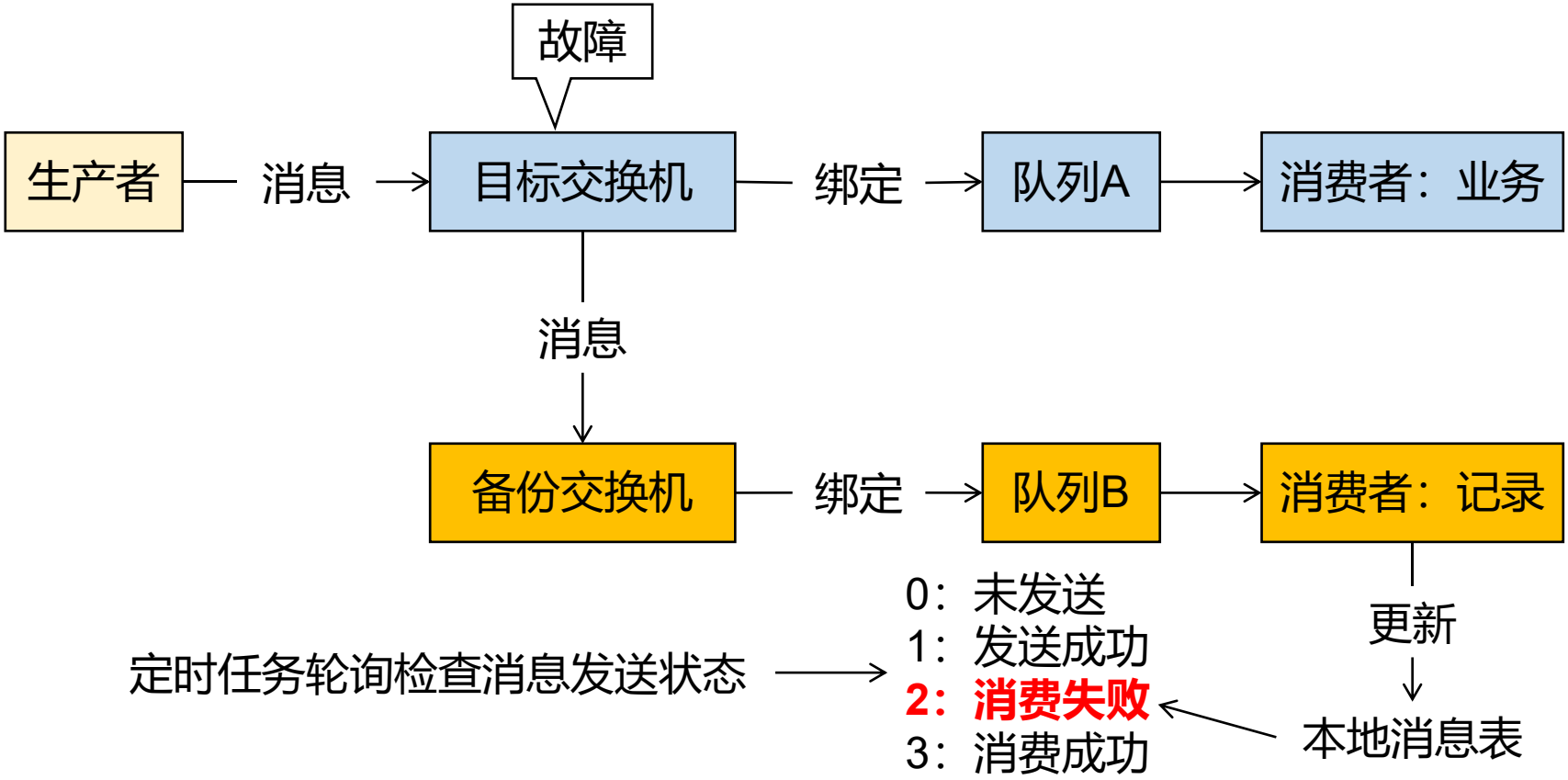
备份交换机原理



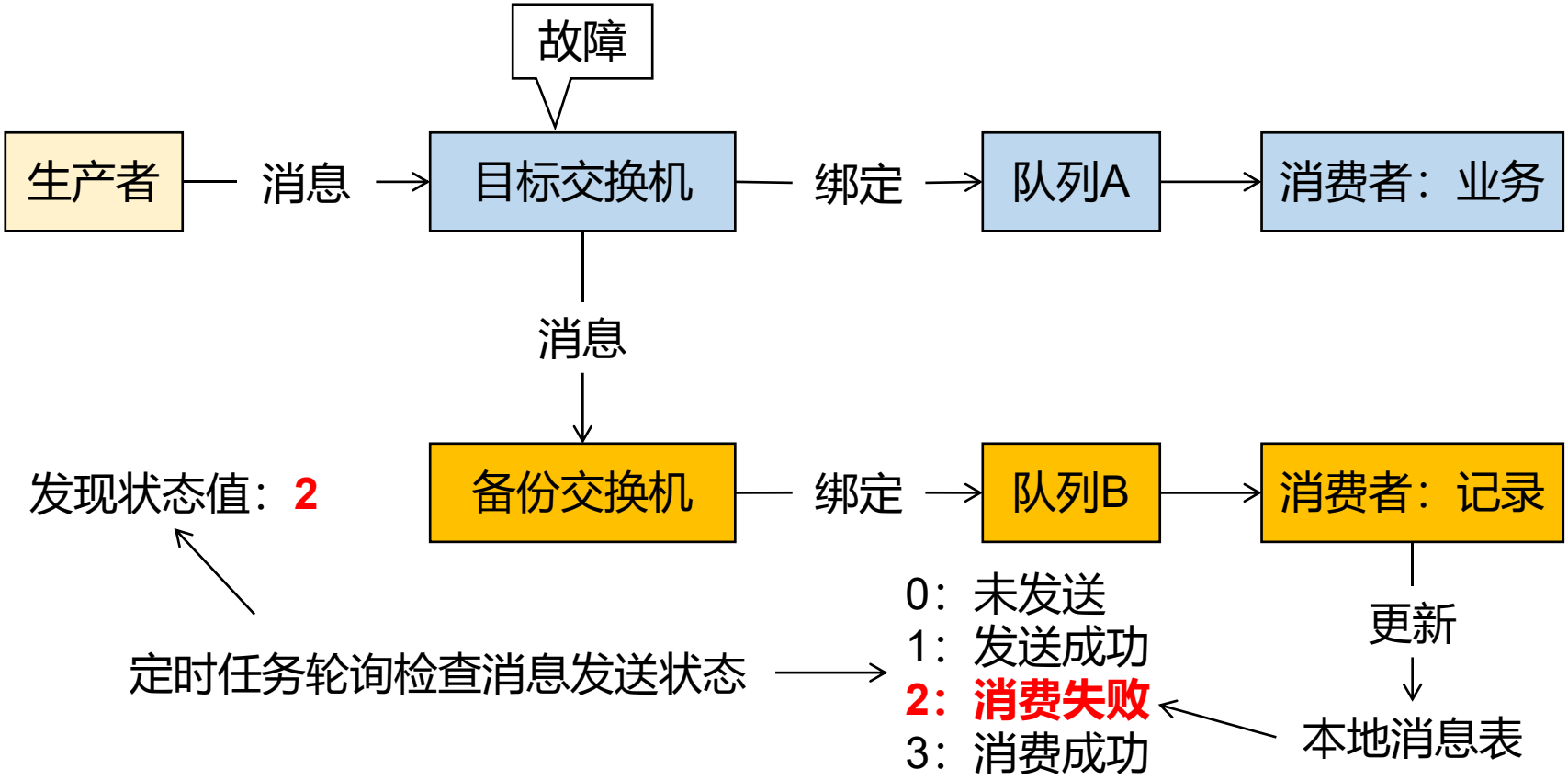
备份交换机原理


故障
生产者
消息
目标交换机
绑定
队列A
消费者：业务
消息
备份交换机
绑定
队列B
消费者：记录
0：未发送
1：发送成功
2：消费失败
3：消费成功
更新
本地消息表

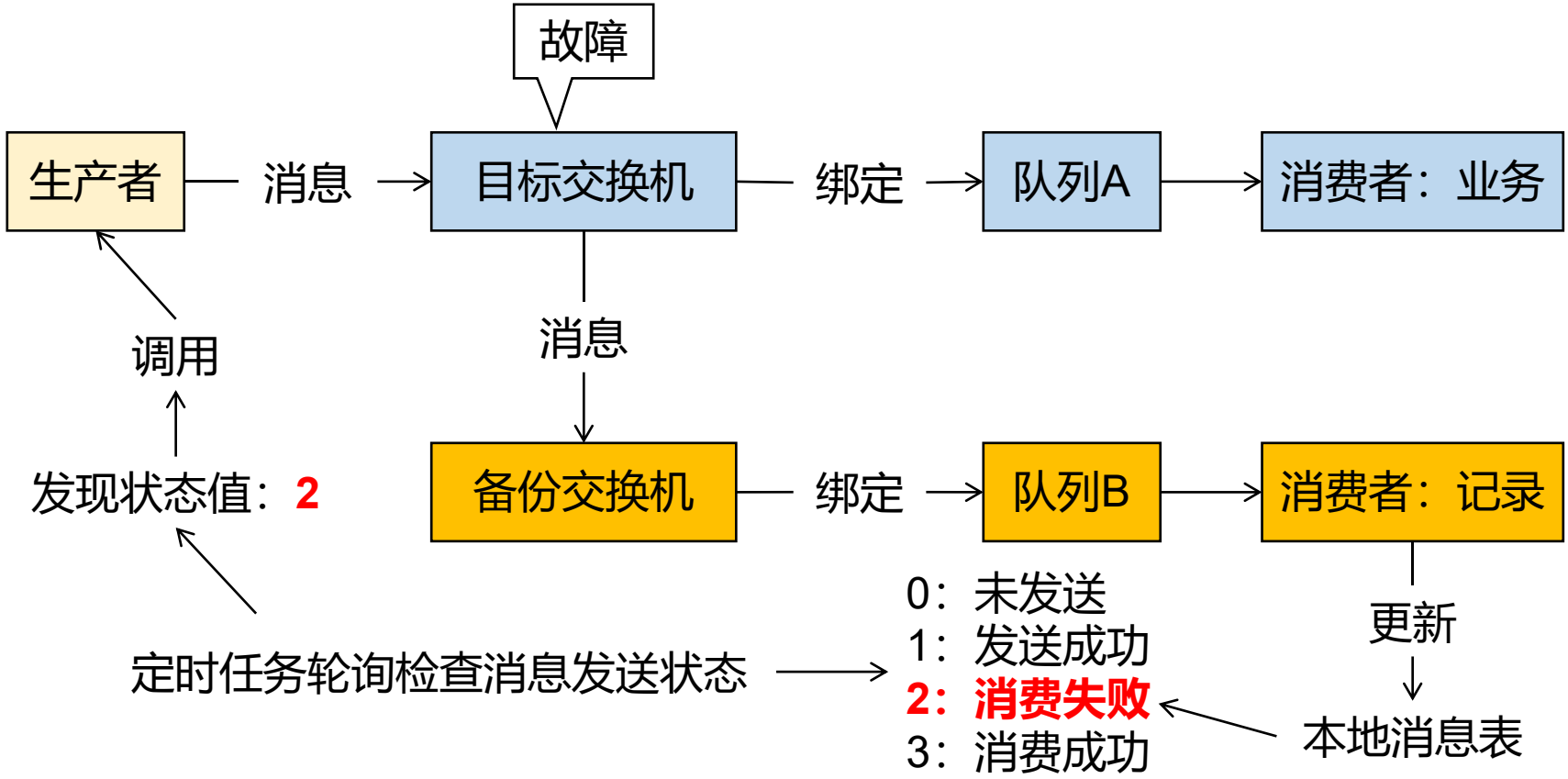
备份交换机原理


故障
生产者
消息
目标交换机
绑定
队列A
消费者：业务
消息
备份交换机
绑定
队列B
消费者：记录
更新
本地消息表
定时任务轮询检查消息发送状态
0：未发送
1：发送成功
2：消费失败
3：消费成功

备份交换机原理


故障
生产者
消息
目标交换机
绑定
队列A
消费者：业务
消息
发现状态值：2
备份交换机
绑定
队列B
消费者：记录
0：未发送
1：发送成功
2：消费失败
3：消费成功
定时任务轮询检查消息发送状态
更新
本地消息表

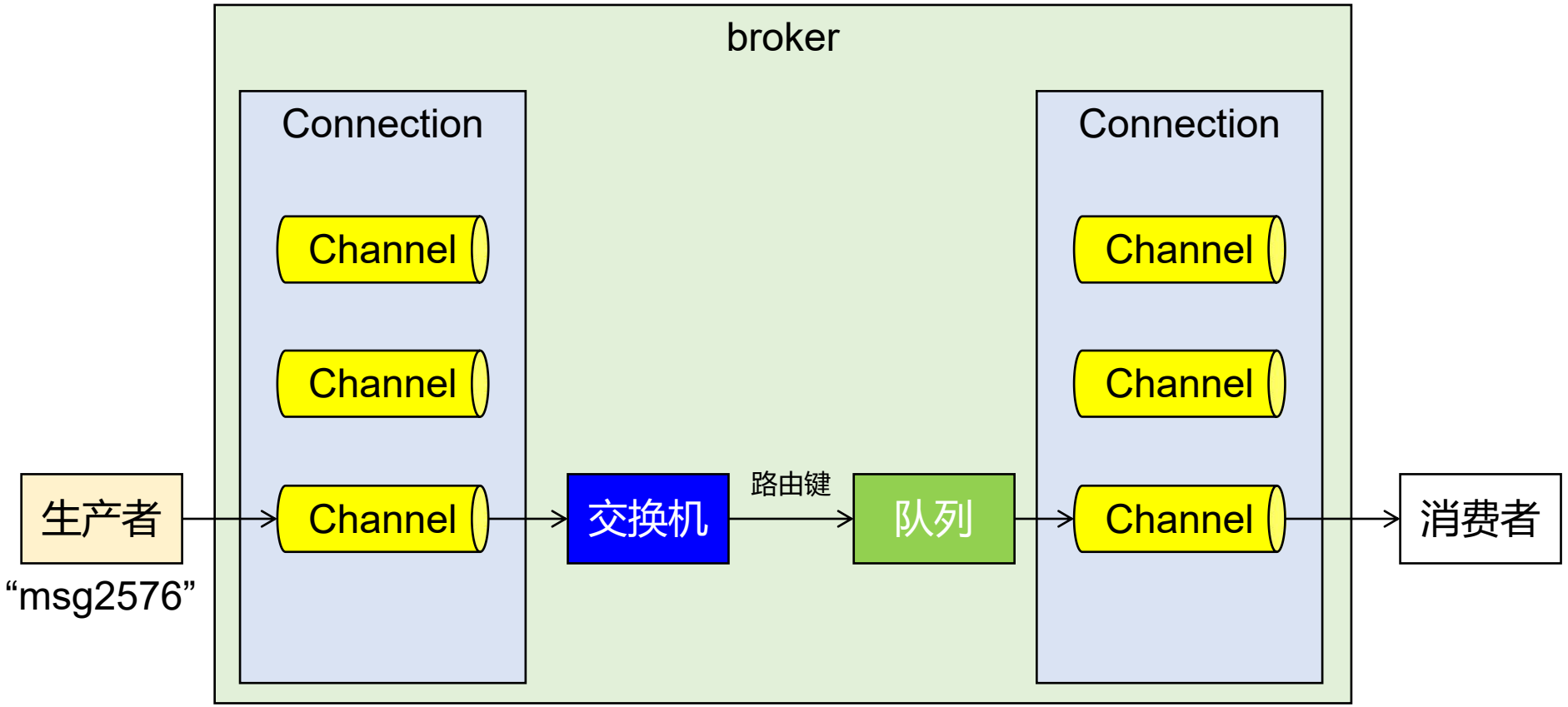
备份交换机原理


故障
生产者
消息
目标交换机
绑定
队列A
消费者：业务
调用
消息
发现状态值：2
备份交换机
绑定
队列B
消费者：记录
0：未发送
1：发送成功
2：消费失败
3：消费成功
更新
本地消息表
定时任务轮询检查消息发送状态

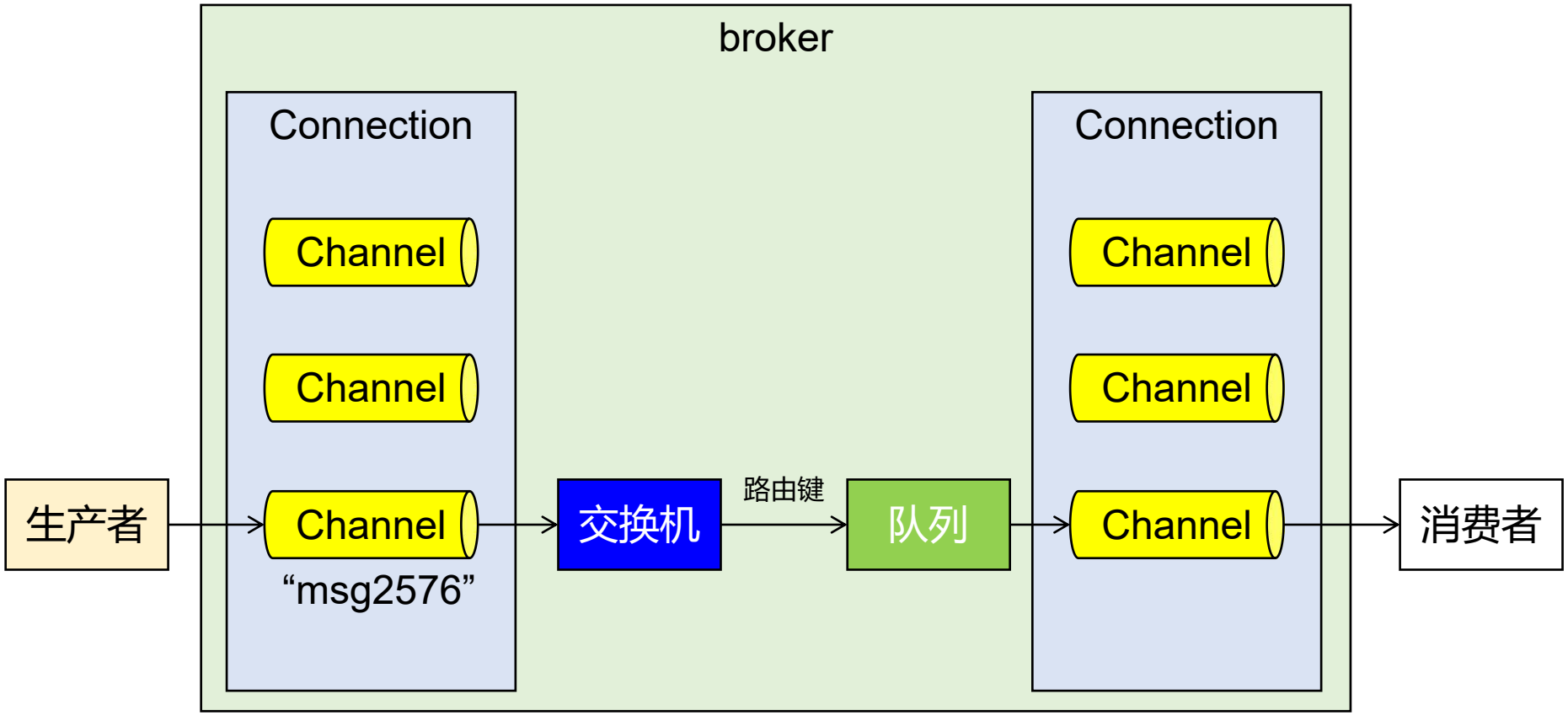
对症下药

- 故障情况1：消息没有发送到消息队列
  - 解决思路A：在**生产者端**进行**确认**，具体操作中我们会分别针对**交换机**和**队列**来确认，如果没有成功发送到消息队列服务器上，那就可以尝试重新发送
  - 解决思路B：为目标交换机指定**备份交换机**，当目标交换机投递失败时，把消息投递至备份交换机
- 故障情况2：消息队列服务器宕机导致内存中消息丢失
  - 解决思路：**消息持久化**到硬盘上，哪怕服务器重启也不会导致消息丢失
- 故障情况3：消费端宕机或抛异常导致消息没有成功被消费
  - 消费端消费消息**成功**，给服务器返回**ACK信息**，然后消息队列删除该消息
  - 消费端消费消息**失败**，给服务器端返回**NACK信息**，同时把消息恢复为**待消费**的状态，这样就可以再次取回消息，**重试**一次（当然，这就需要消费端接口支持幂等性）

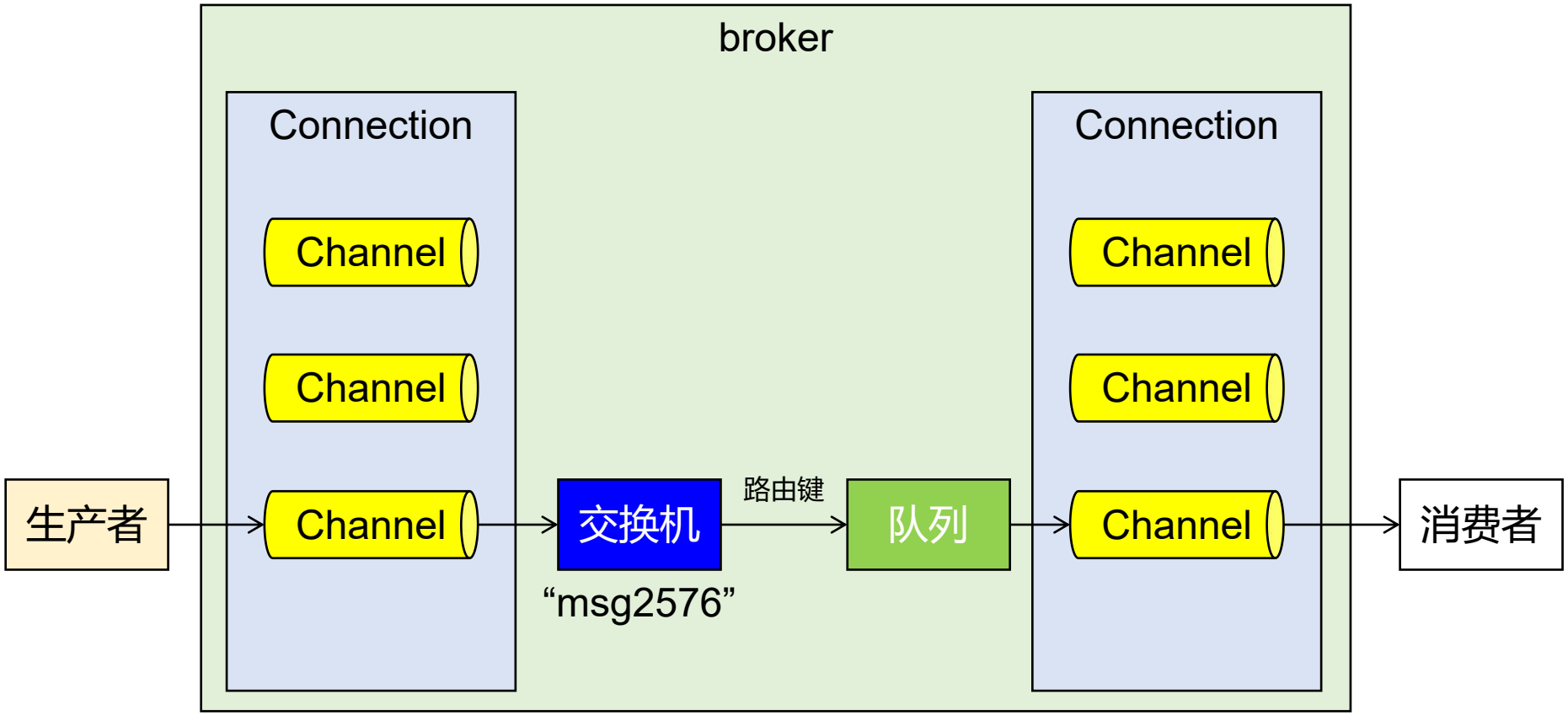
deliveryTag：交付标签机制



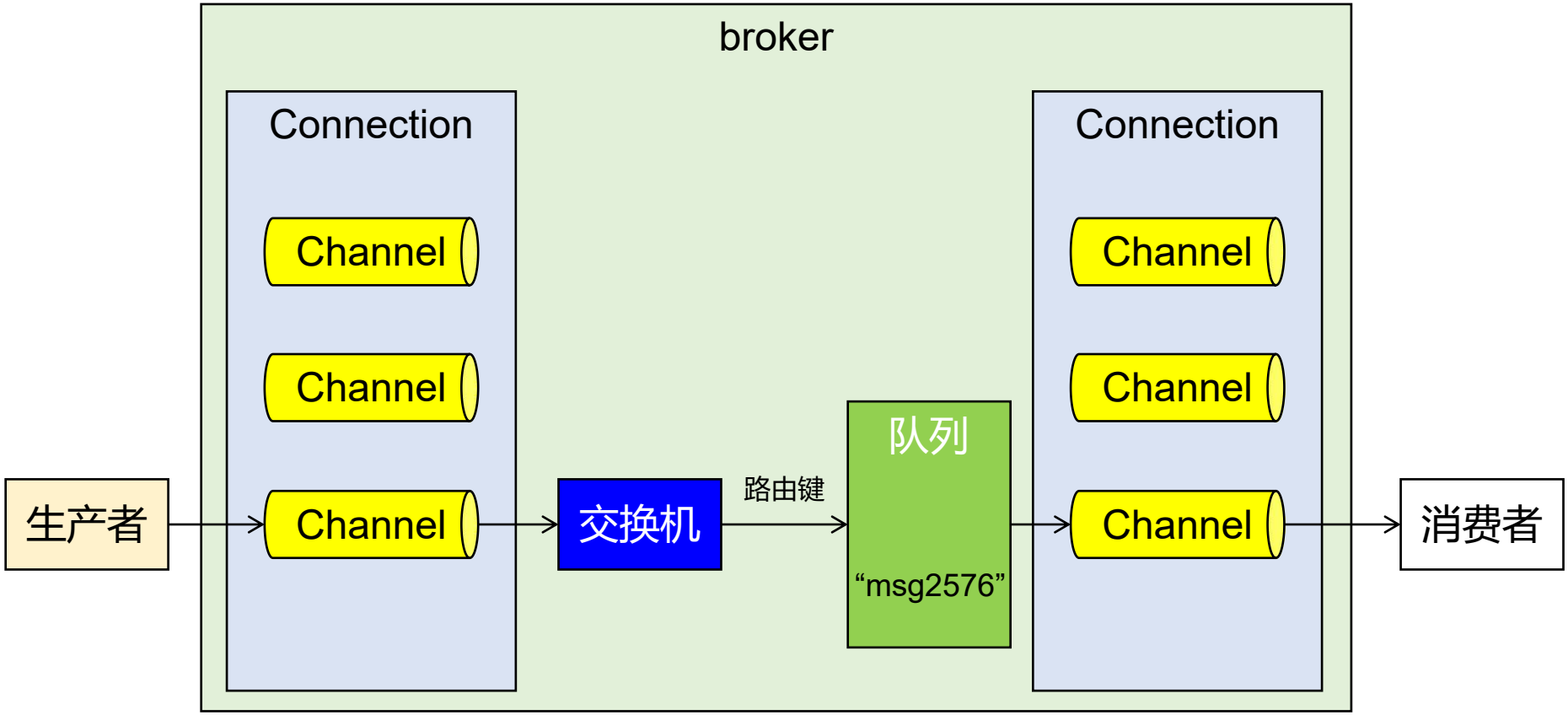
deliveryTag：交付标签机制



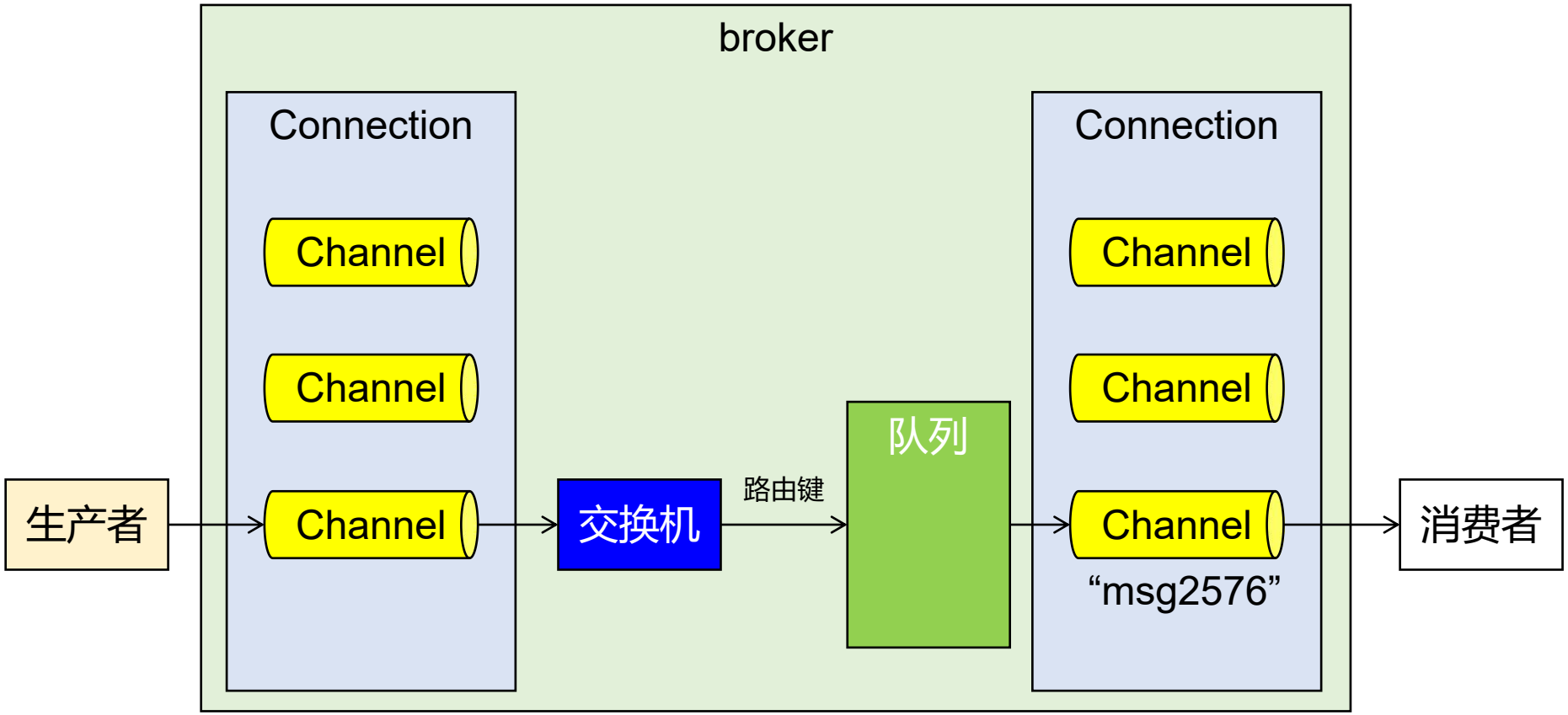
deliveryTag：交付标签机制



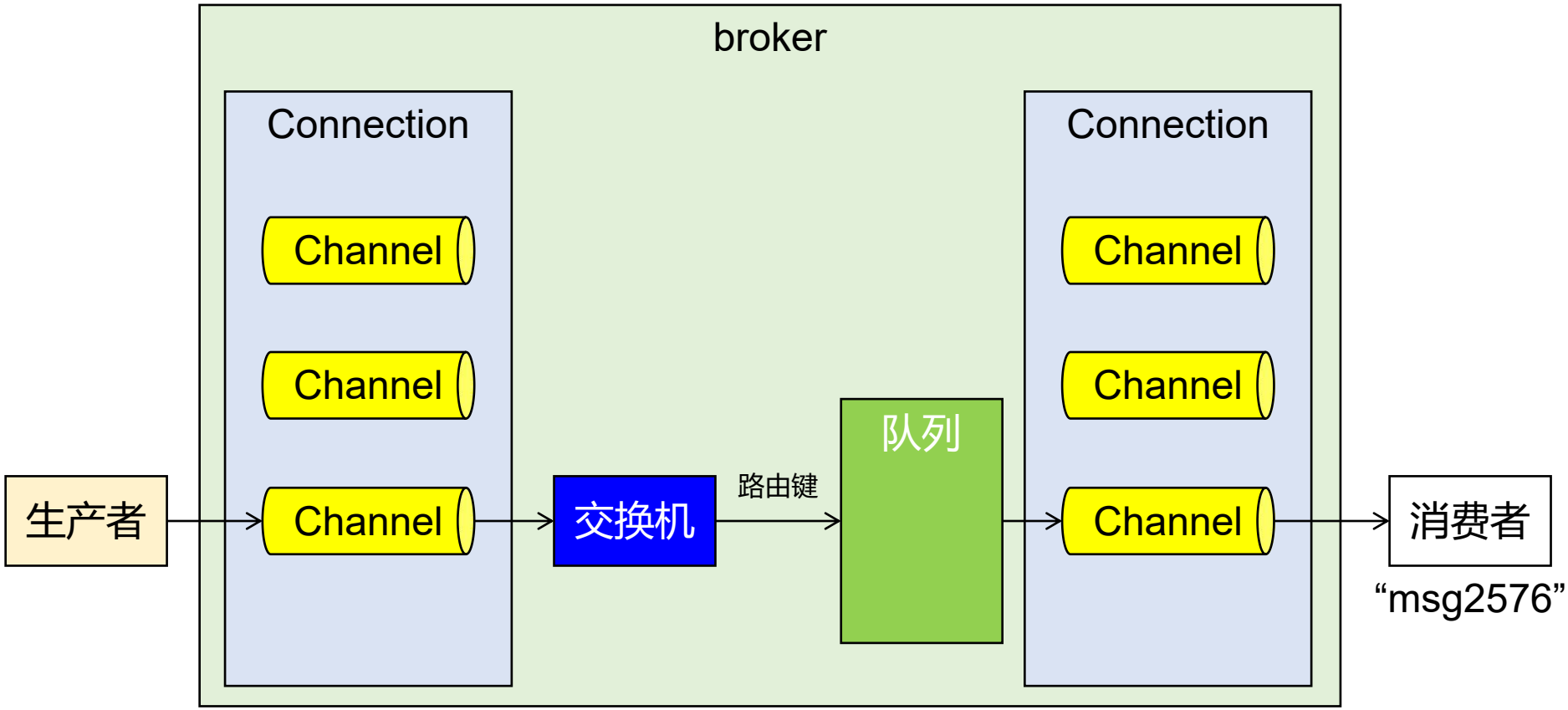
deliveryTag：交付标签机制



deliveryTag：交付标签机制



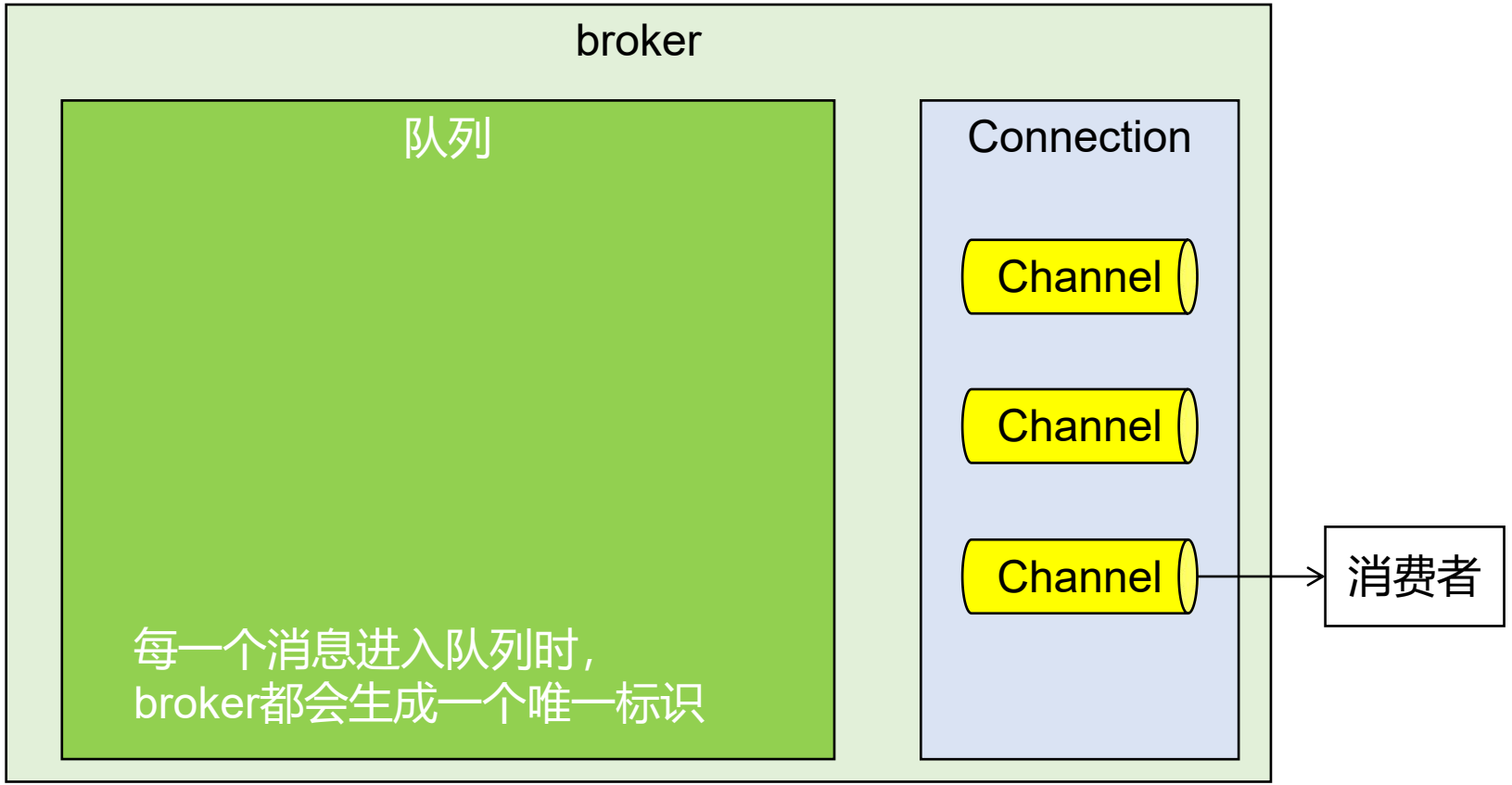
deliveryTag：交付标签机制



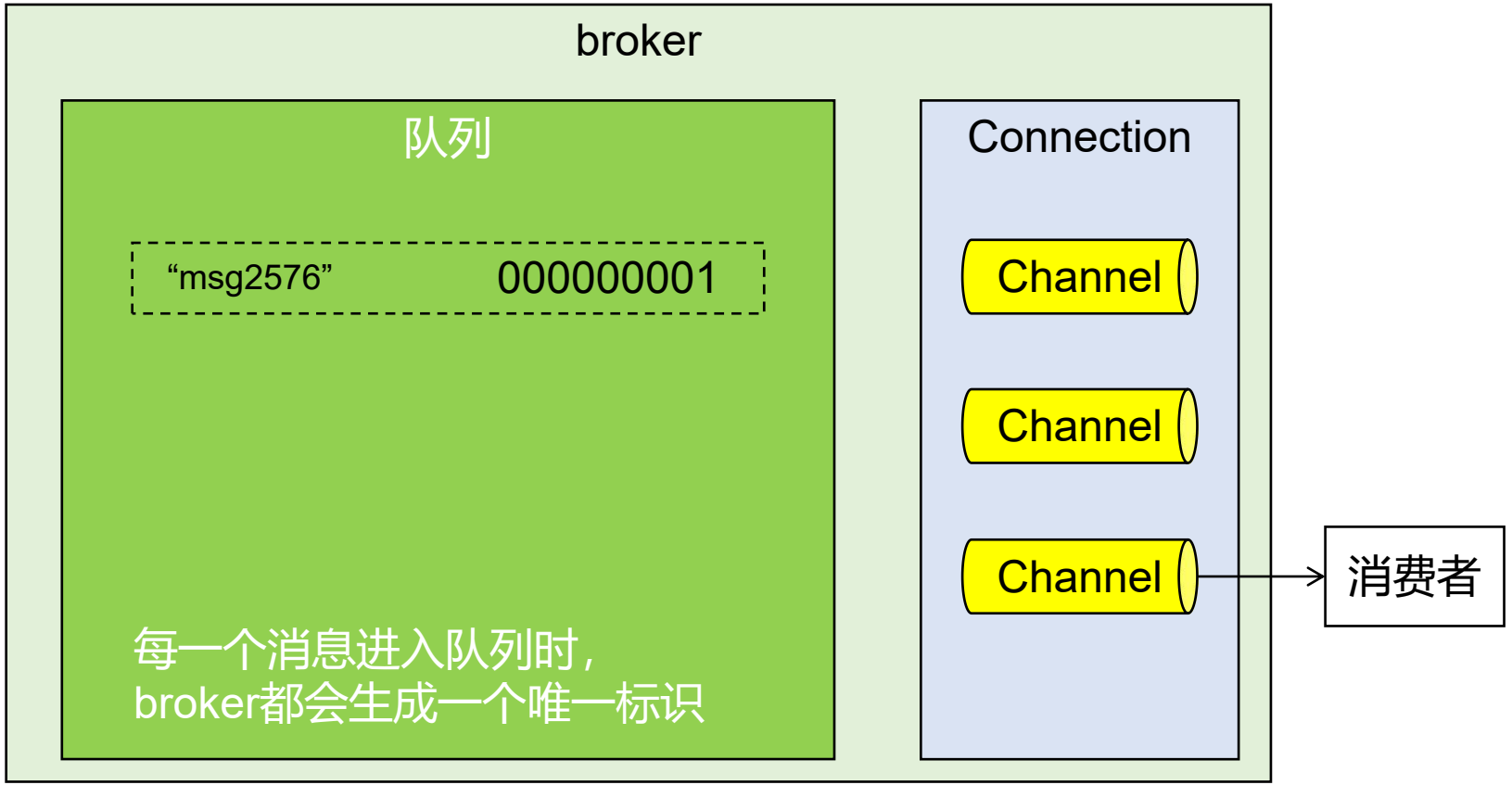
deliveryTag：交付标签机制

引入细节：deliveryTag

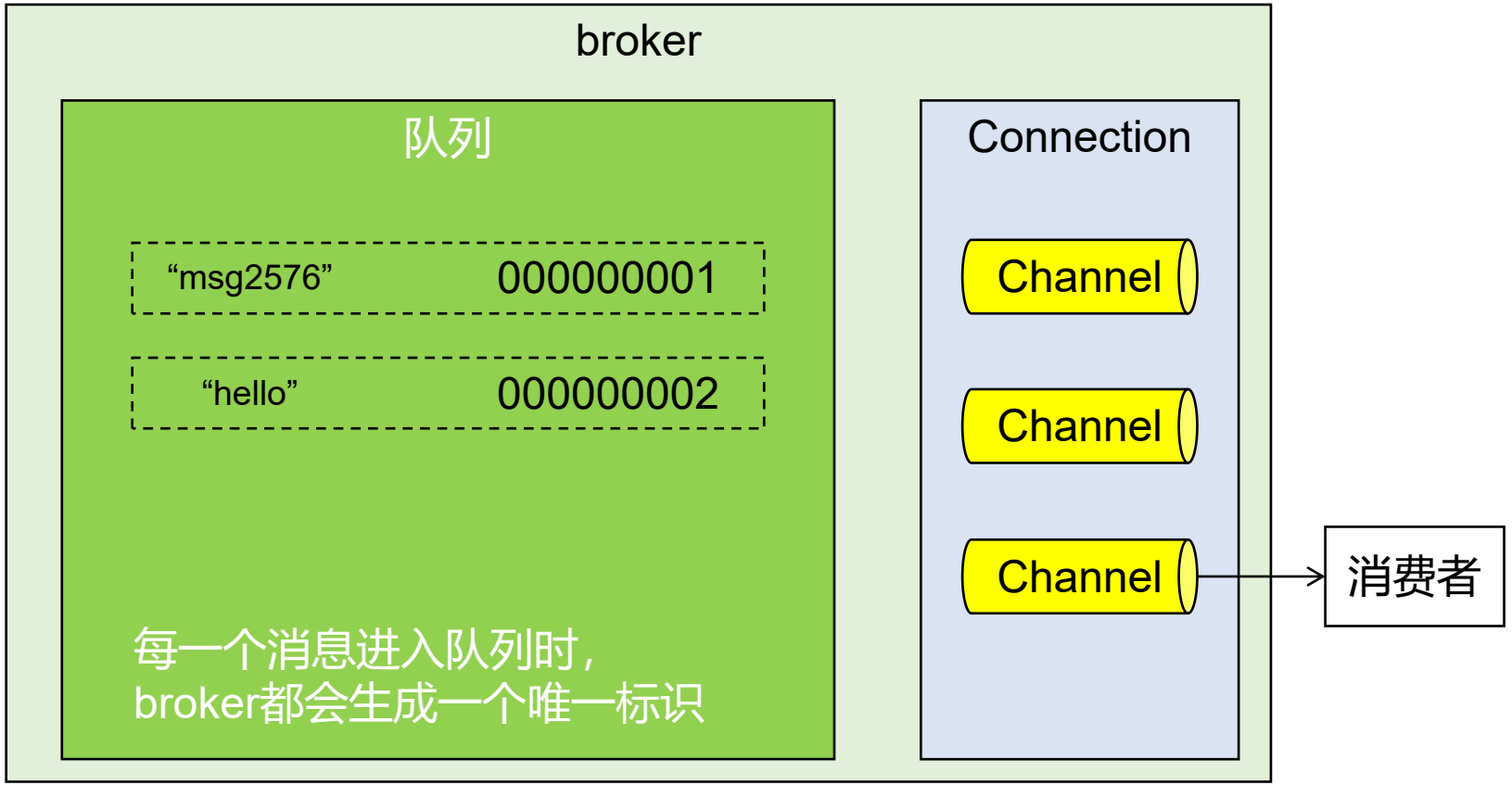
deliveryTag：交付标签机制



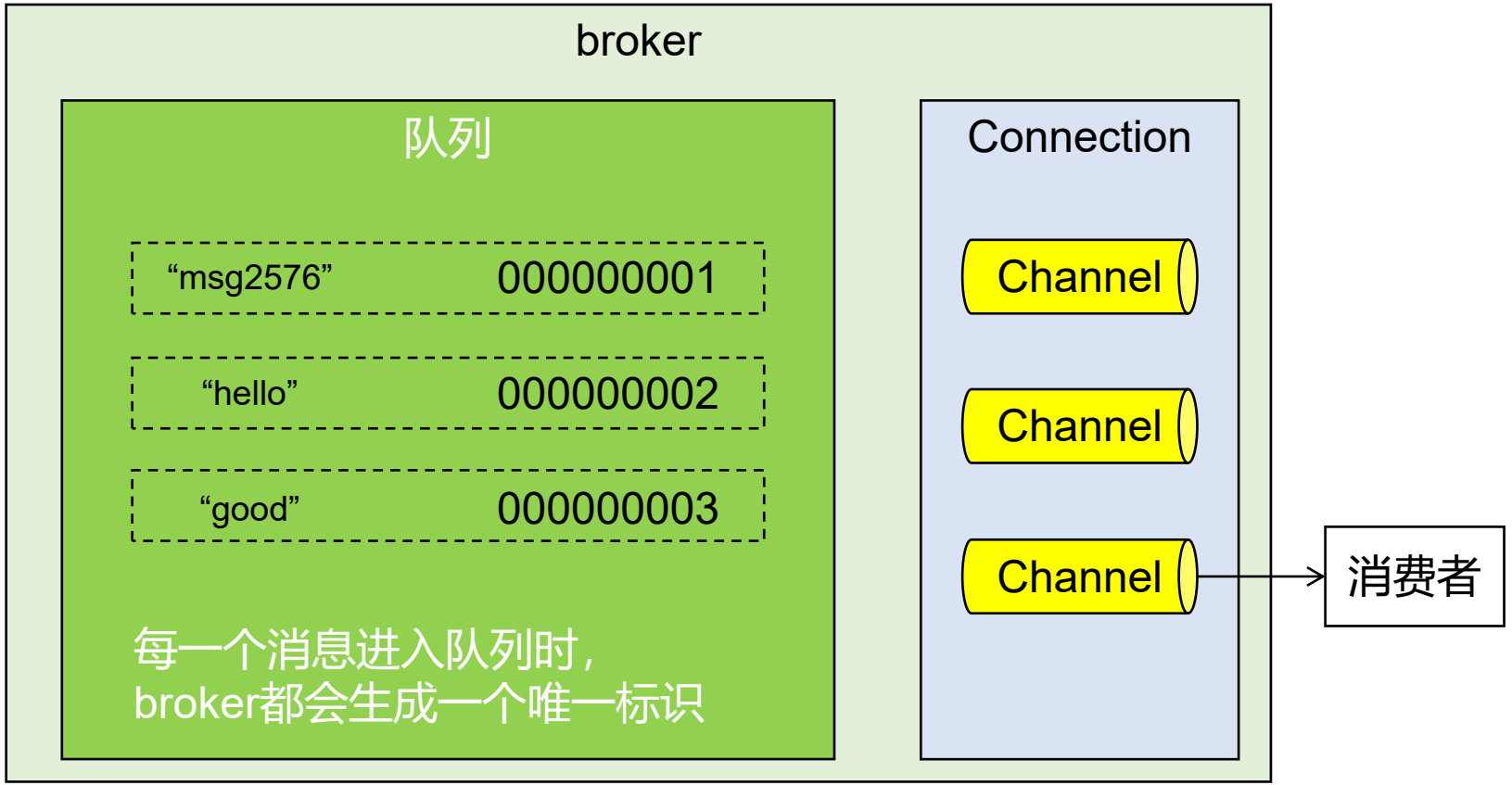
deliveryTag：交付标签机制



deliveryTag：交付标签机制



deliveryTag：交付标签机制



deliveryTag：交付标签机制

这个唯一标识就是deliveryTag(交付标签)

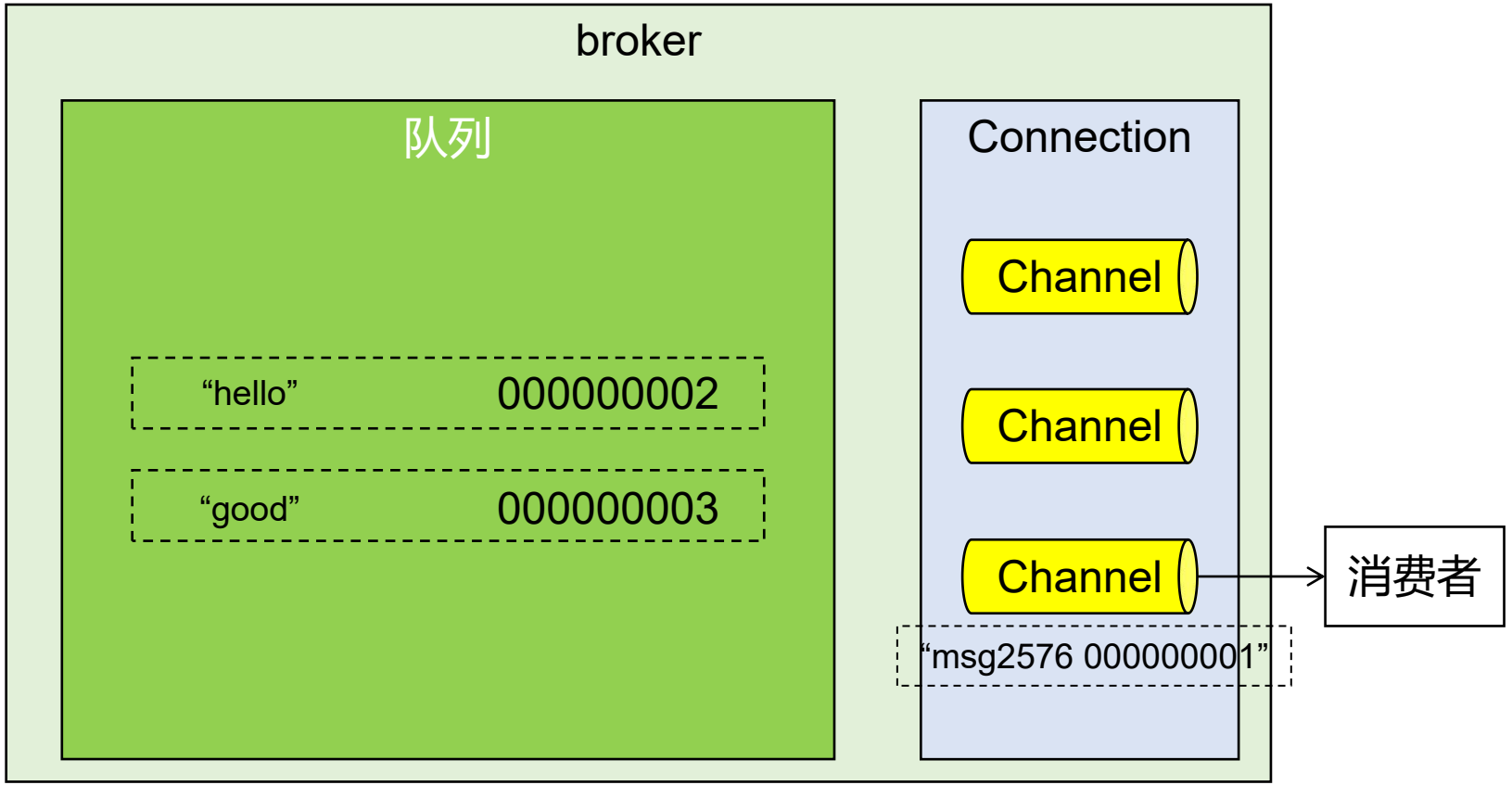
deliveryTag：交付标签机制

deliveryTag是一个64位整数

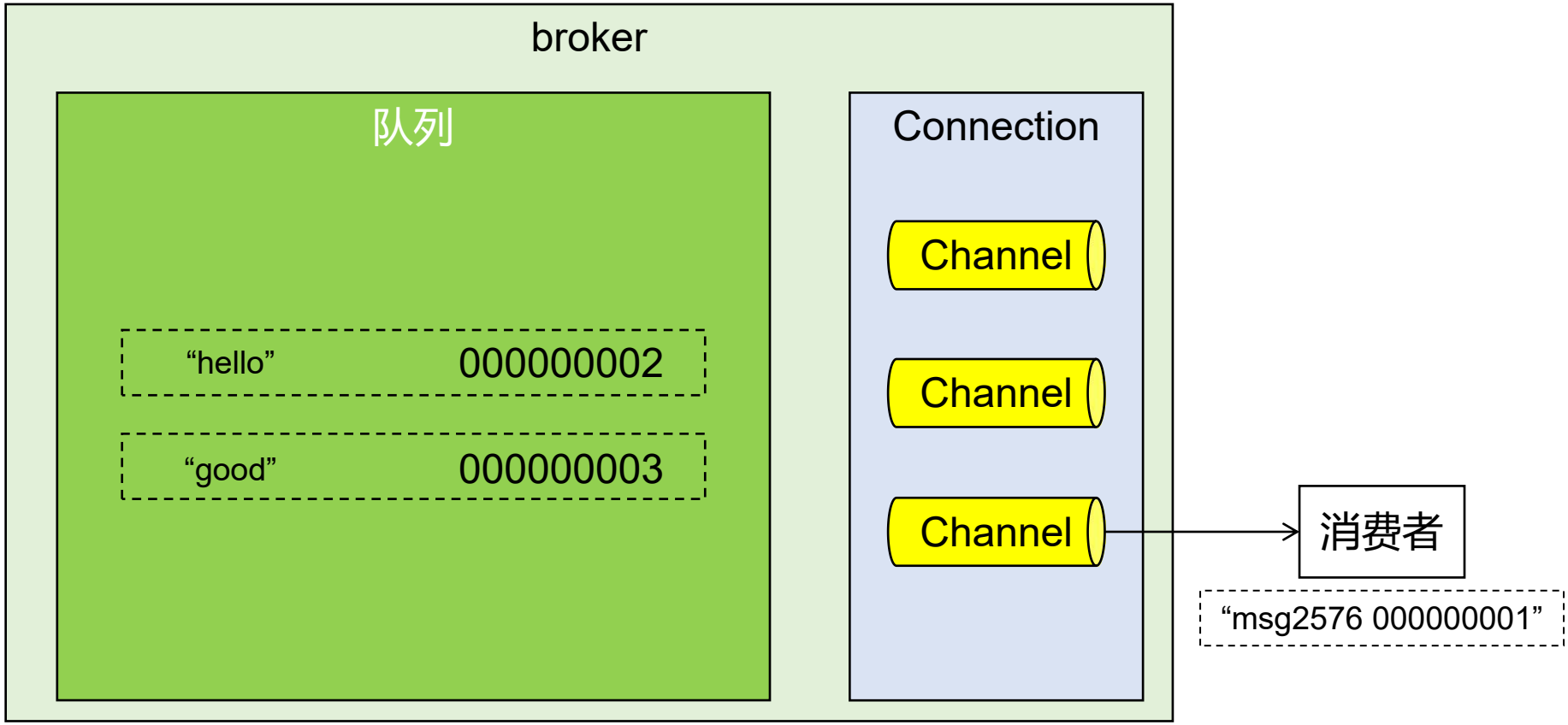
deliveryTag：交付标签机制

消息往消费端投递时，会携带交付标签

deliveryTag：交付标签机制



deliveryTag：交付标签机制



deliveryTag：交付标签机制

提问：交付标签有啥用?

deliveryTag：交付标签机制

答：消费端把消息处理结果ACK、NACK、Reject等返回给Broker之后，Broker需要对对应的消息执行后续操作，例如删除消息、重新排队或标记为死信等等。那么Broker就必须知道它现在要操作的消息具体是哪一条。而deliveryTag作为消息的唯一标识就很好的满足了这个需求。

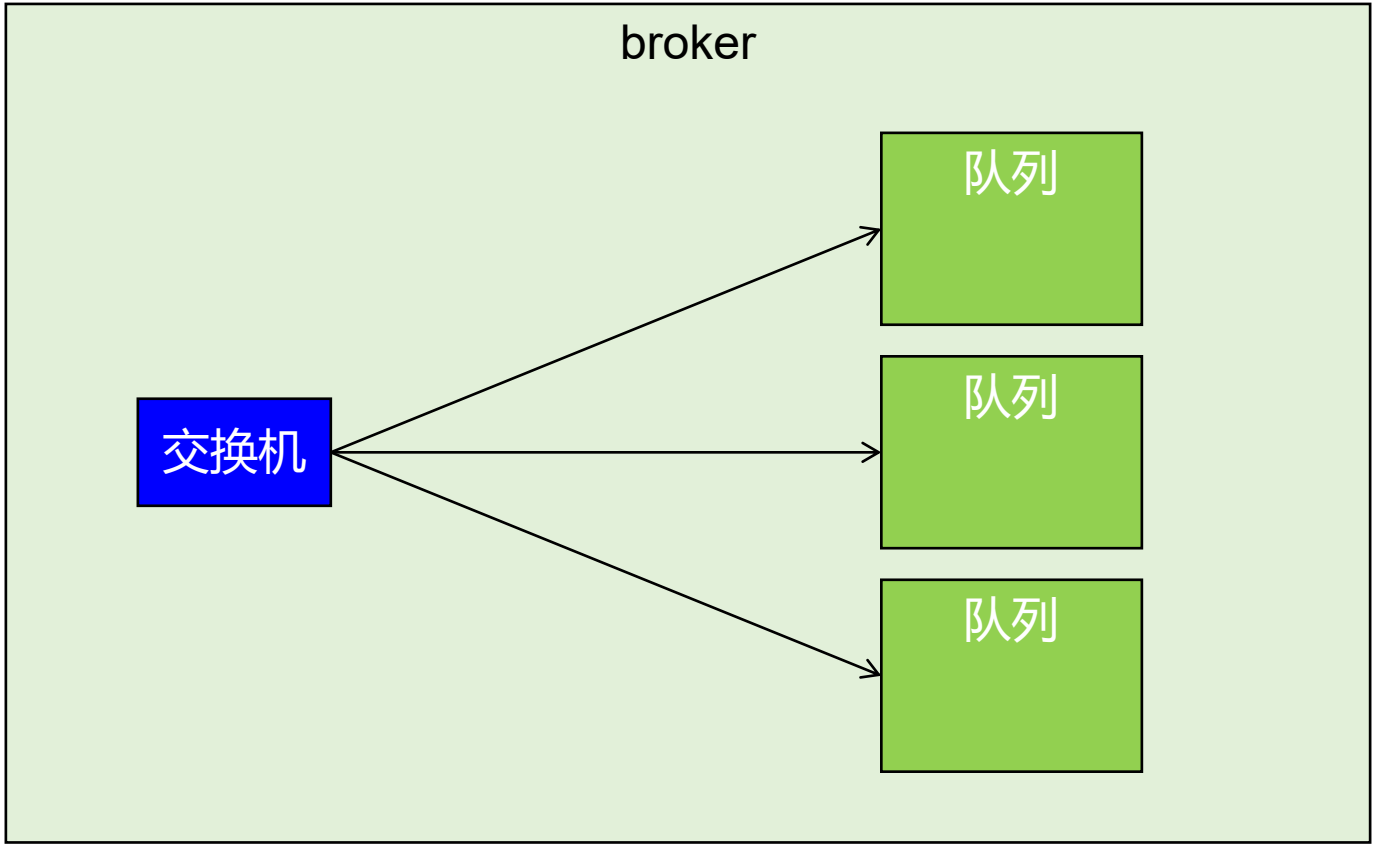
deliveryTag：交付标签机制

提问：如果交换机是Fanout模式，同一个消息广播到了不同队列，deliveryTag会重复吗?

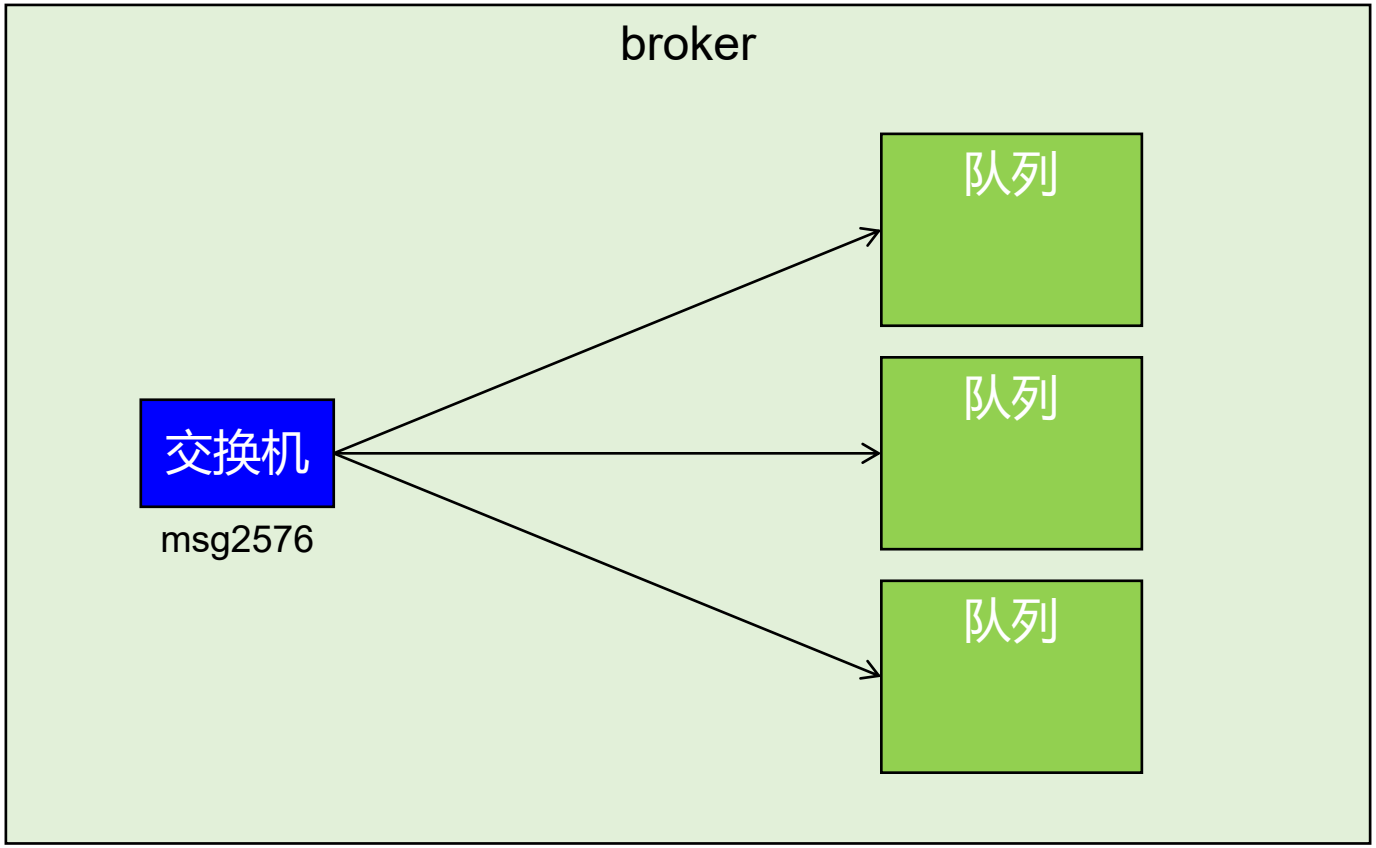
deliveryTag：交付标签机制

答：不会，deliveryTag在Broker范围内唯一

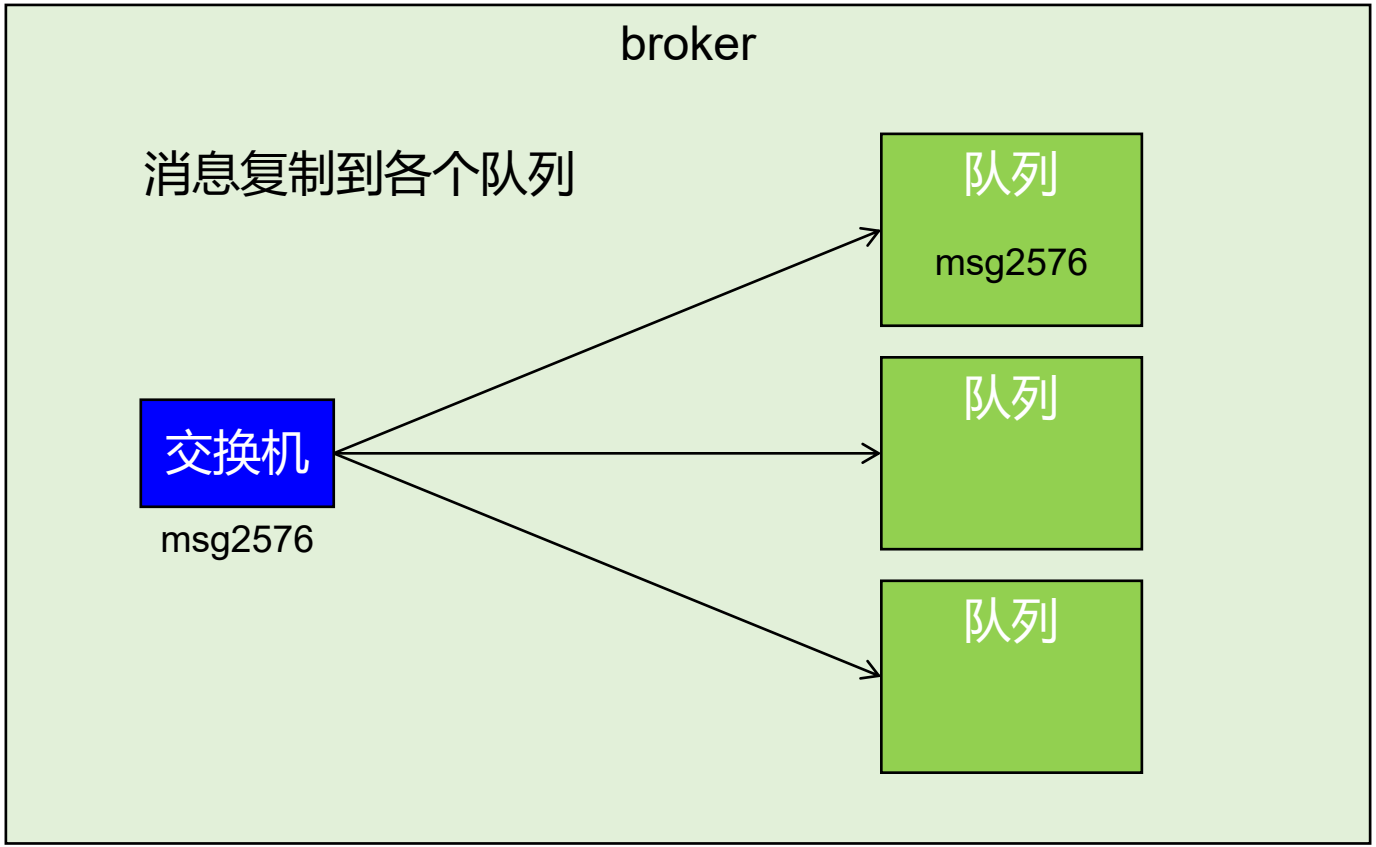
deliveryTag：交付标签机制



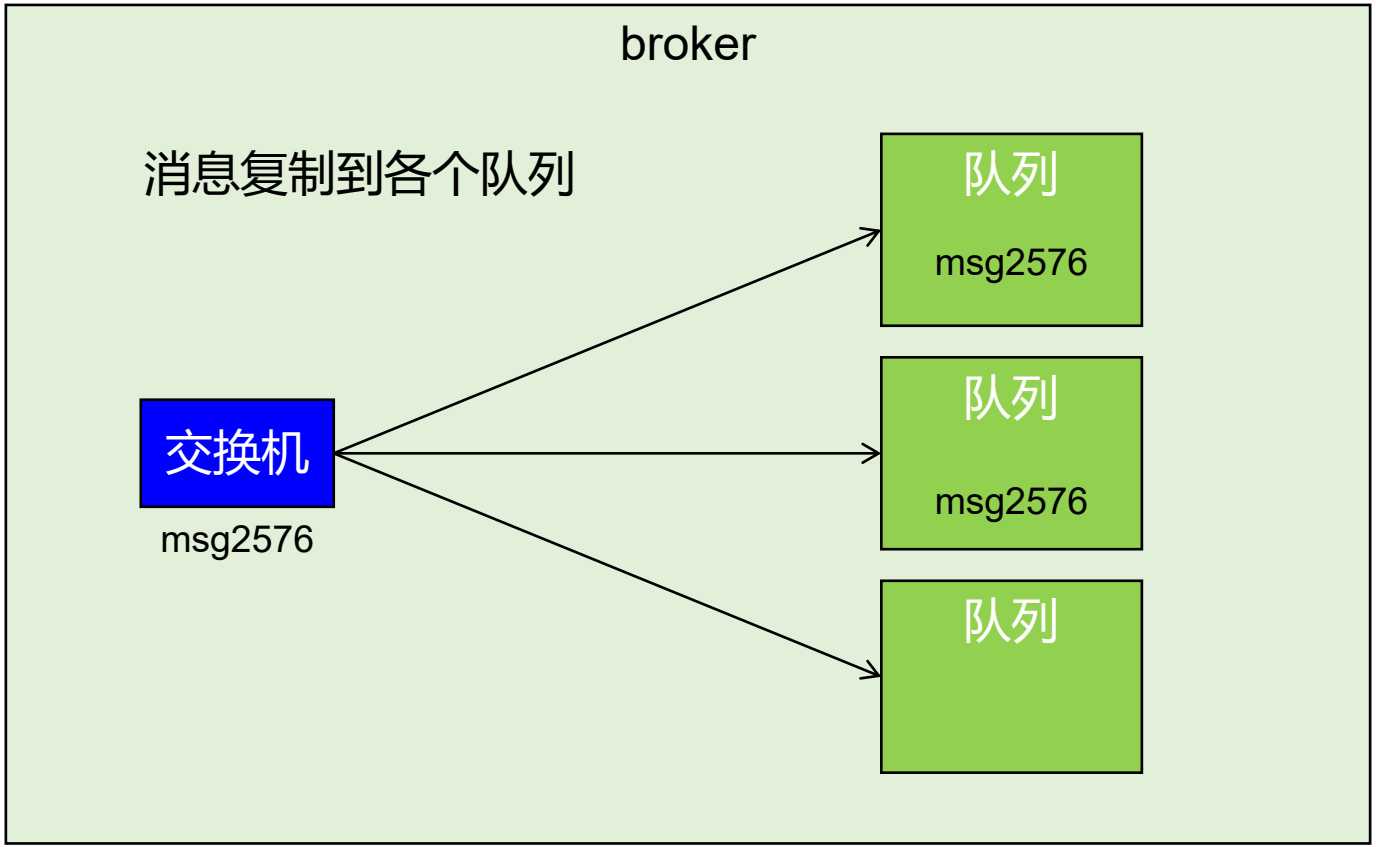
deliveryTag：交付标签机制



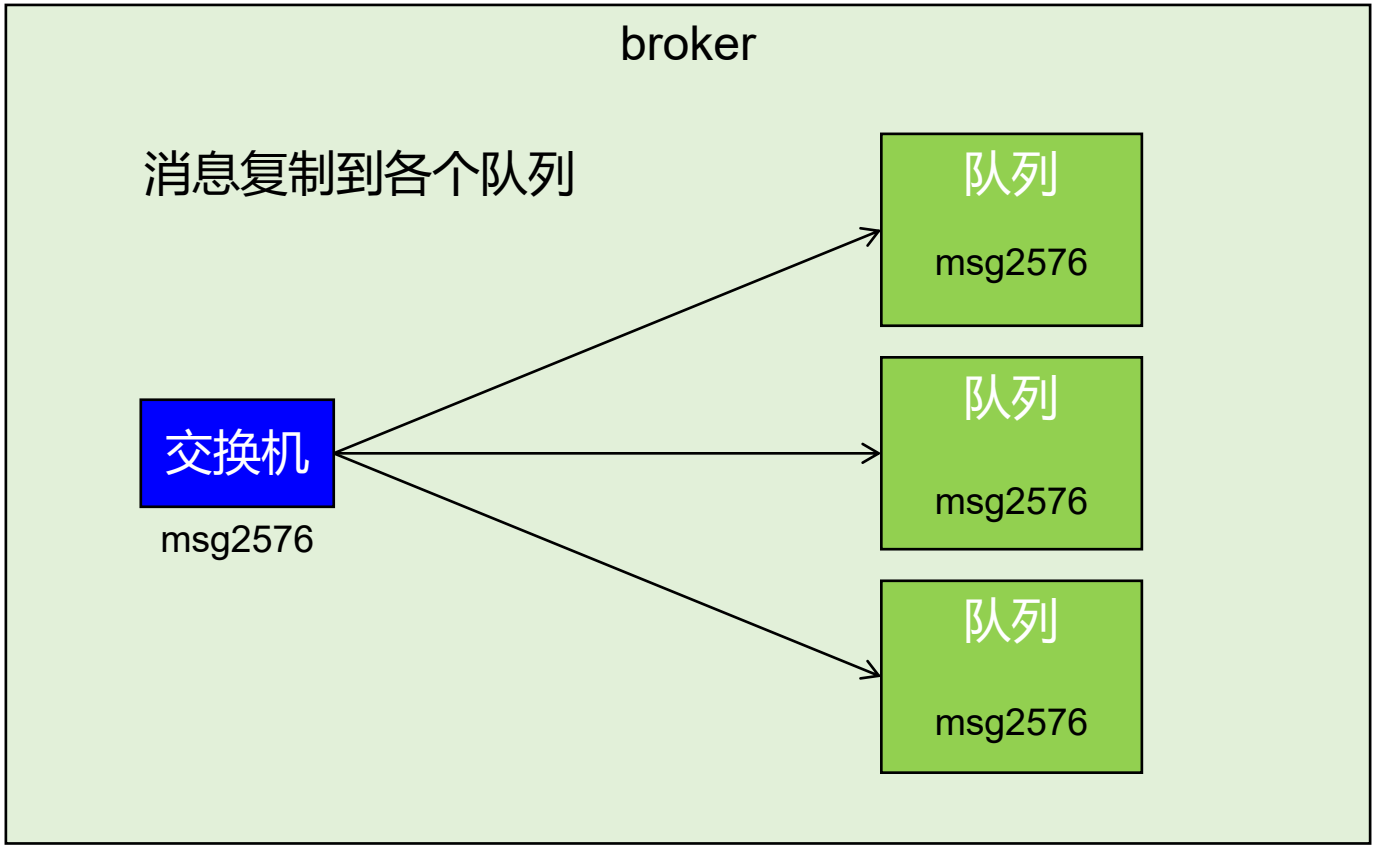
deliveryTag：交付标签机制



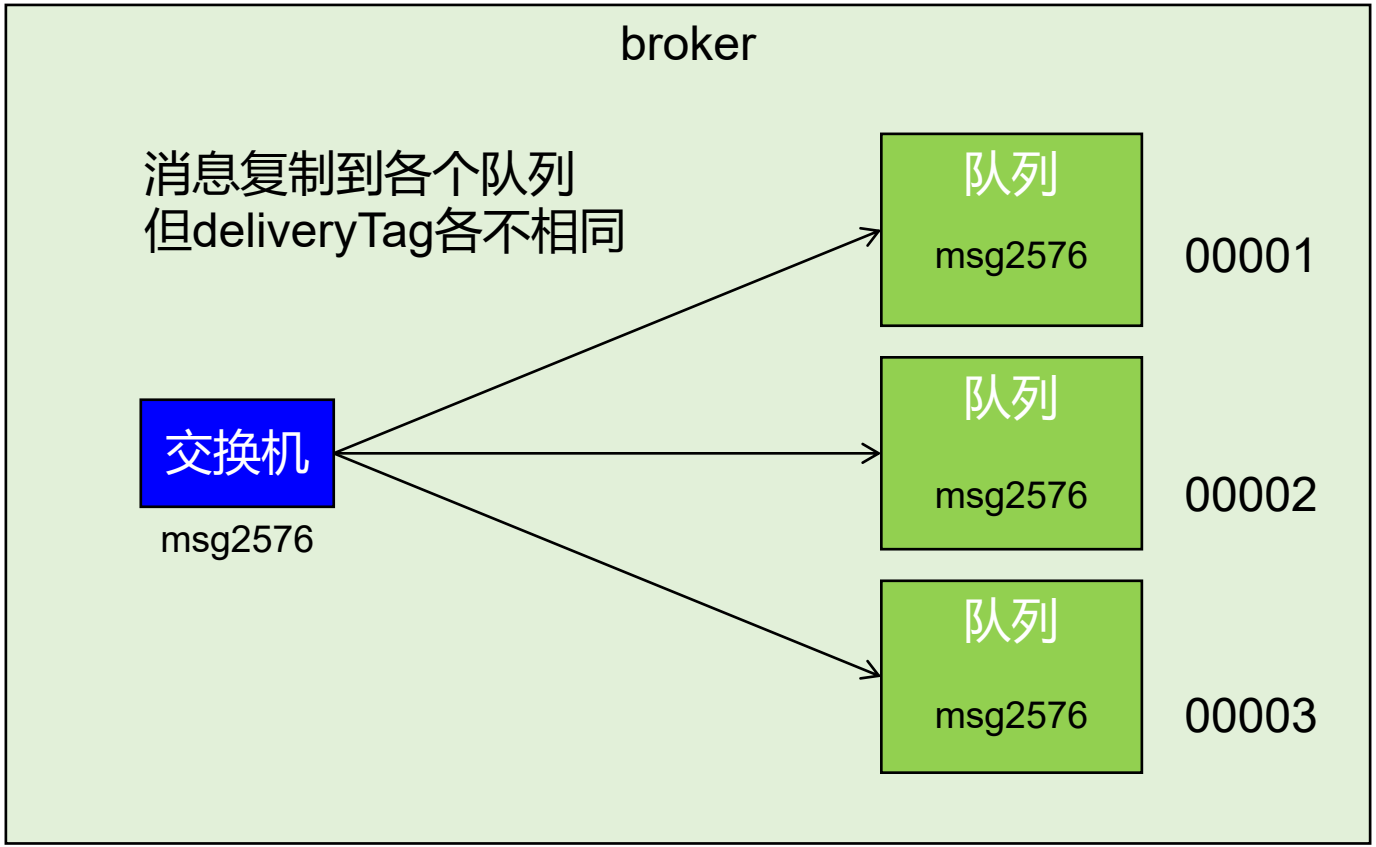
deliveryTag：交付标签机制



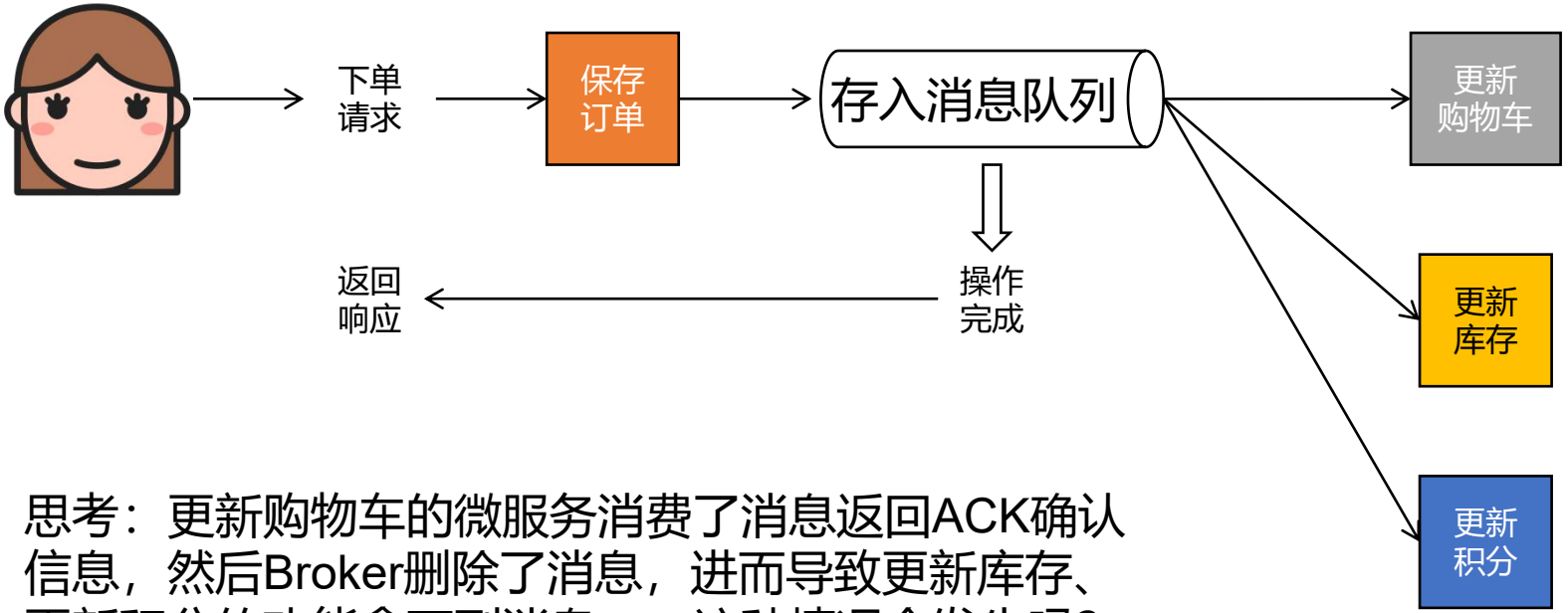
deliveryTag：交付标签机制



deliveryTag：交付标签机制



deliveryTag：交付标签机制



思考：更新购物车的微服务消费了消息返回ACK确认信息，然后Broker删除了消息，进而导致更新库存、更新积分的功能拿不到消息——这种情况会发生吗？

消息确认相关方法参数说明：multiple

deliveryTag	消息内容本身
000001	“hello atguigu”
000002	“good morning”
000003	“吃饭了吗？您呐？”
000004	“256823464654754552346444454”
000005	“372466505980328409384093 89”

消息确认相关方法参数说明：multiple

deliveryTag	消息内容本身
000001	“hello atguigu”
000002	“good morning”
000003	“吃饭了吗？您呐？”
000004	“256823464654754552346444454”
000005	“372466505980328409384093 89”

指定某个deliveryTag

消息确认相关方法参数说明：multiple

multiple为true时

deliveryTag	消息内容本身
000001	“hello atguigu”
000002	“good morning”
000003	“吃饭了吗？您呐？”
000004	“256823464654754552346444454”
000005	“372466505980328409384093 89”

指定某个deliveryTag

消息确认相关方法参数说明：multiple

multiple为true时

批量处理

指定某个deliveryTag

deliveryTag	消息内容本身
000001	“hello atguigu”
000002	“good morning”
000003	“吃饭了吗？您呐？”
000004	“256823464654754552346444454”
000005	“37246650598032840938409389”

消息确认相关方法参数说明：multiple

multiple为false时

单独处理

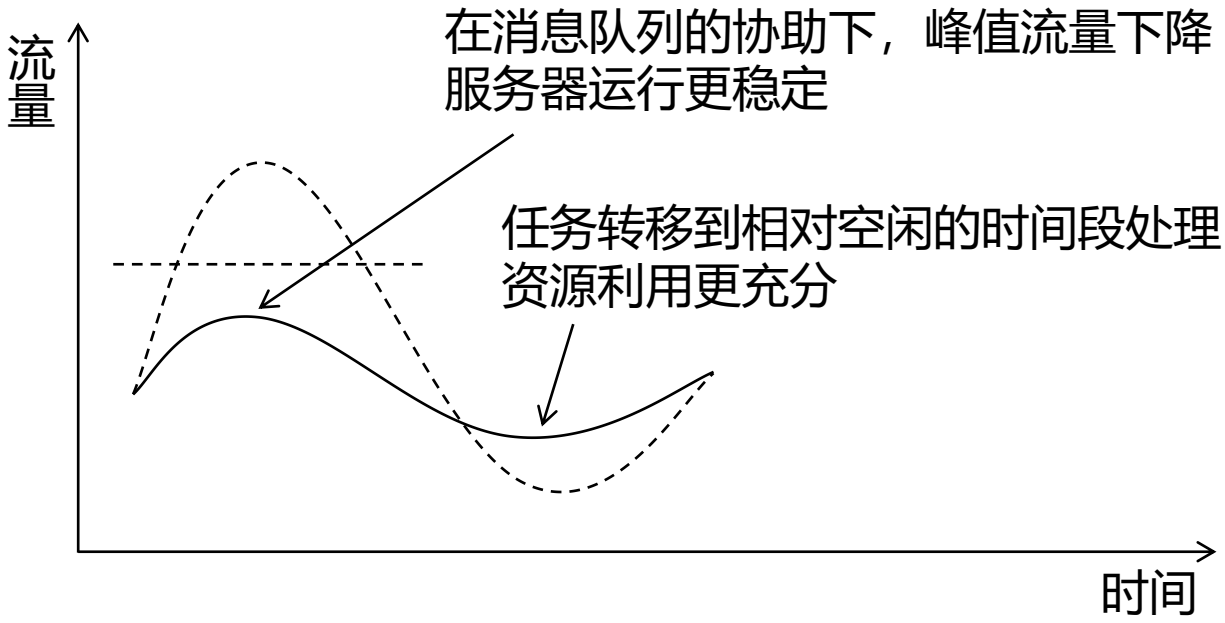
指定某个deliveryTag

deliveryTag	消息内容本身
000001	“hello atguigu”
000002	“good morning”
000003	“吃饭了吗？您呐？”
000004	“256823464654754552346444454”
000005	“37246650598032840938409389”

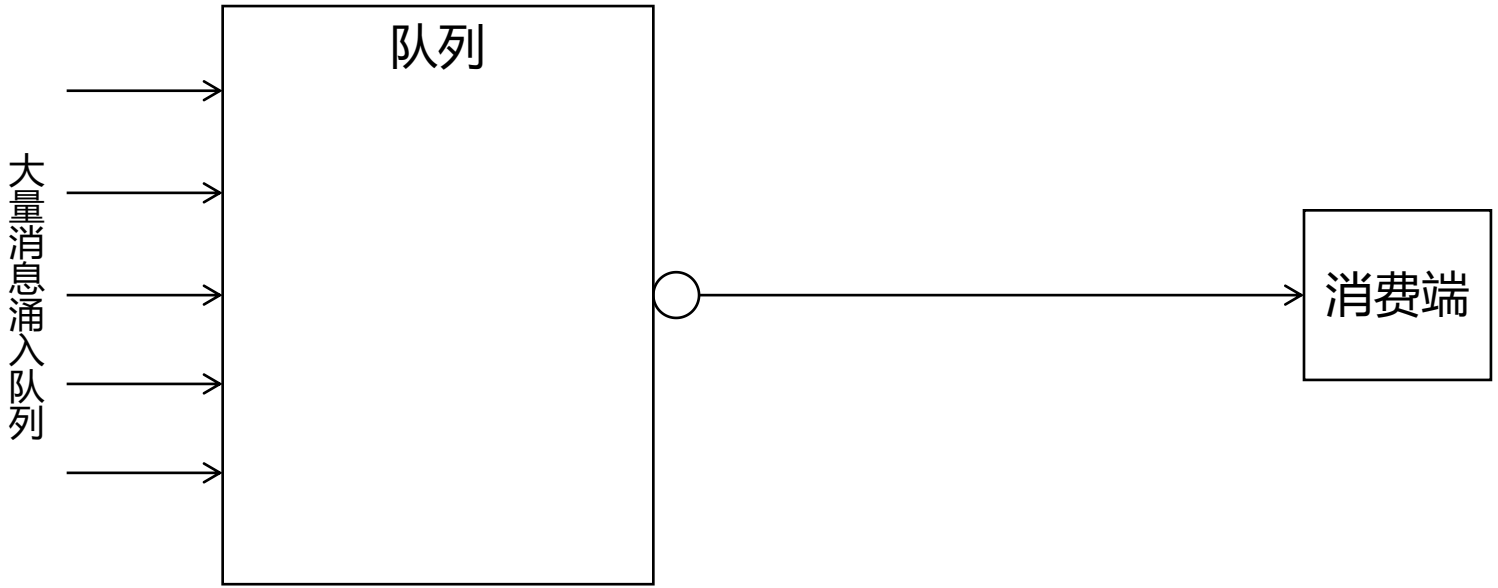
3 消费端限流

Consumer end flow restriction

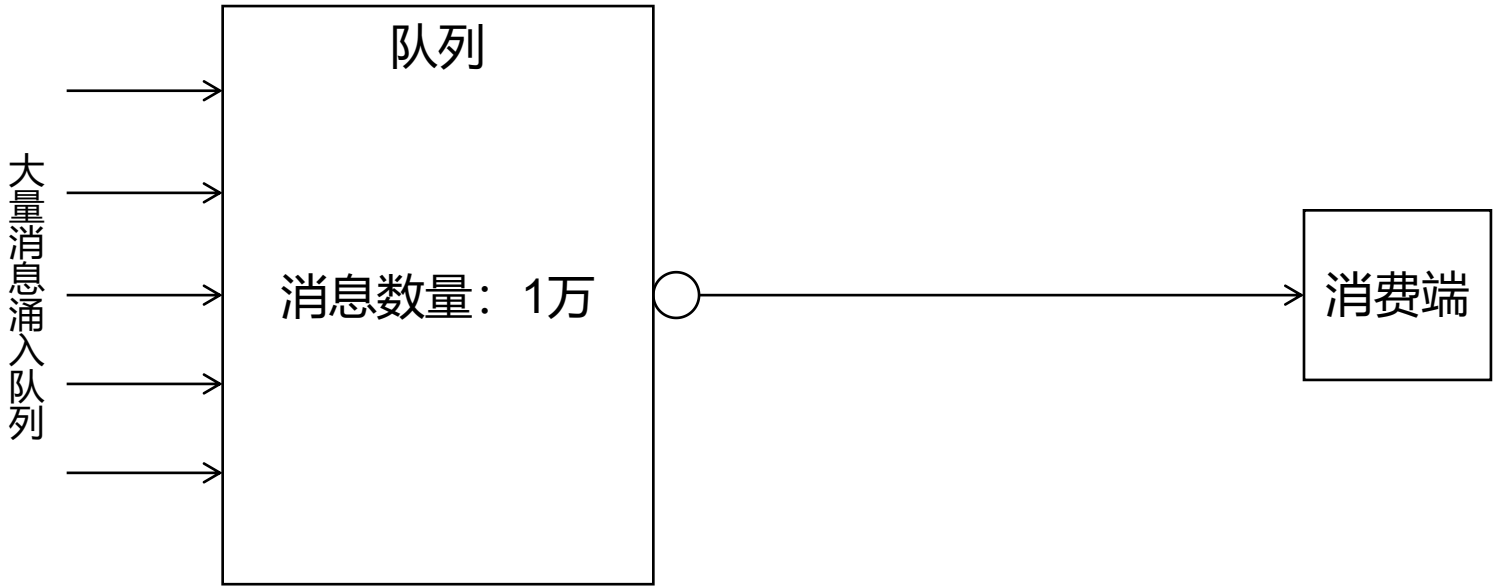
削峰限流的好处



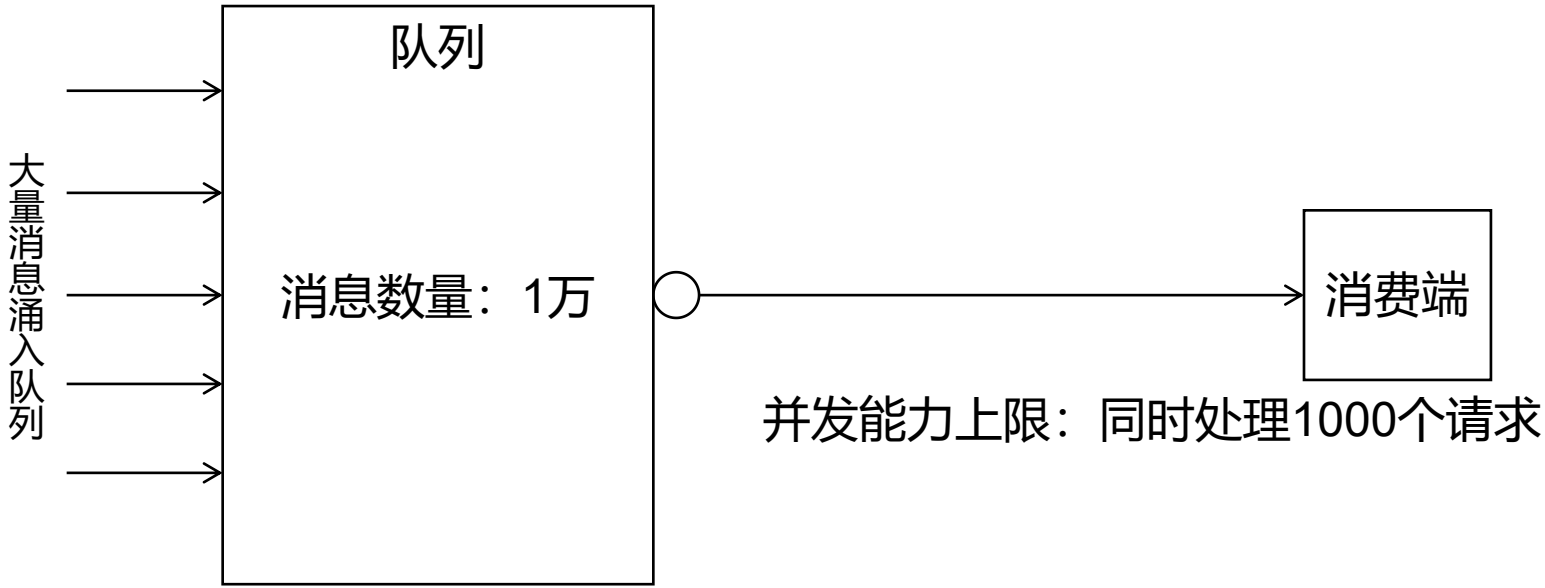
图示



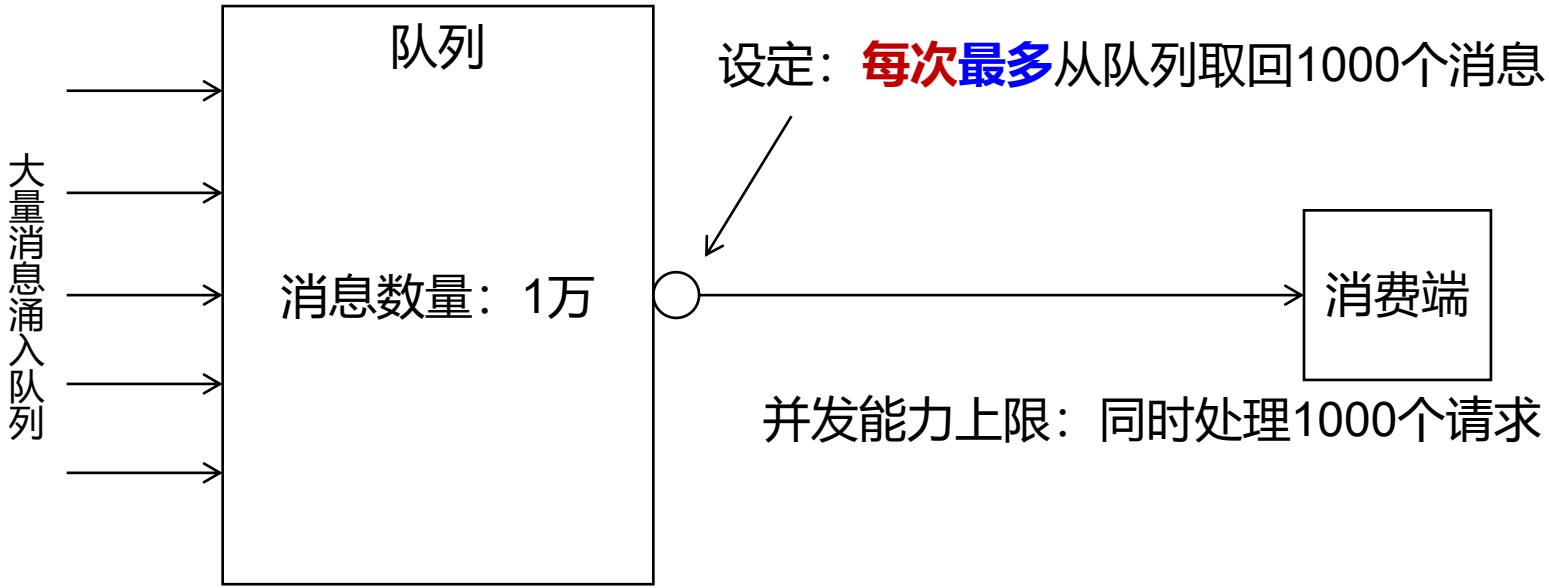
图示



图示



图示


队列
大量消息涌入队列
消息数量：1万
设定：每次最多从队列取回1000个消息
消费端
并发能力上限：同时处理1000个请求

设置方式

- 非常简单，就是一个参数：**prefetch**
- 操作文档：Operation009-Prefetch.md

4 消息超时

Message Timeout

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**
- 我们可以从两个层面来给消息设定过期时间：

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**
- 我们可以从两个层面来给消息设定过期时间：
  - **队列层面**：在队列层面设定**消息的过期时间**，并不是队列的过期时间。意思是这个队列中的消息全部使用**同一个**过期时间。

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**
- 我们可以从两个层面来给消息设定过期时间：
  - **队列层面**：在队列层面设定**消息的过期时间**，并不是队列的过期时间。意思是这个队列中的消息全部使用**同一个**过期时间。
  - **消息本身**：给具体的某个消息设定过期时间

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**
- 我们可以从两个层面来给消息设定过期时间：
  - **队列层面**：在队列层面设定**消息的过期时间**，并不是队列的过期时间。意思是这个队列中的消息全部使用**同一个**过期时间。
  - **消息本身**：给具体的某个消息设定过期时间
- 如果两个层面都做了设置，那么哪个时间短，哪个生效

相关说明

- 给**消息**设定一个**过期时间**，超过这个时间没有被取走的消息就会被**删除**
- 我们可以从两个层面来给消息设定过期时间：
  - **队列层面**：在队列层面设定**消息的过期时间**，并不是队列的过期时间。意思是这个队列中的消息全部使用**同一个**过期时间。
  - **消息本身**：给具体的某个消息设定过期时间
- 如果两个层面都做了设置，那么哪个时间短，哪个生效
- 操作文档：Operation010-MessageTimeout.md

5 死信和死信队列

Dead letter and dead letter queue

死信

死信

- 概念：当一个消息无法被消费，它就变成了死信。

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费
- 死信的处理方式大致有下面三种：

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费
- 死信的处理方式大致有下面三种：
  - **丢弃**：对不重要的消息直接丢弃，不做处理

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费
- 死信的处理方式大致有下面三种：
  - **丢弃**：对不重要的消息直接丢弃，不做处理
  - **入库**：把死信写入数据库，日后处理

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费
- 死信的处理方式大致有下面三种：
  - **丢弃**：对不重要的消息直接丢弃，不做处理
  - **入库**：把死信写入数据库，日后处理
  - **监听**：消息变成死信后进入死信队列，我们专门设置消费端监听死信队列，做后续处理（通常采用）

死信

- 概念：当一个消息无法被消费，它就变成了死信。
- 死信产生的原因大致有下面三种：
  - **拒绝**：消费者拒接消息，basicNack()/basicReject()，并且不把消息重新放入原目标队列，requeue=false
  - **溢出**：队列中消息数量到达限制。比如队列最大只能存储10条消息，且现在已经存储了10条，此时如果再发送一条消息进来，根据先进先出原则，队列中最早的消息会变成死信
  - **超时**：消息到达超时时间未被消费
- 死信的处理方式大致有下面三种：
  - **丢弃**：对不重要的消息直接丢弃，不做处理
  - **入库**：把死信写入数据库，日后处理
  - **监听**：消息变成死信后进入死信队列，我们专门设置消费端监听死信队列，做后续处理（通常采用）
- 操作文档：Operation011-DeadLetter.md

6 延迟队列

Delay Queue

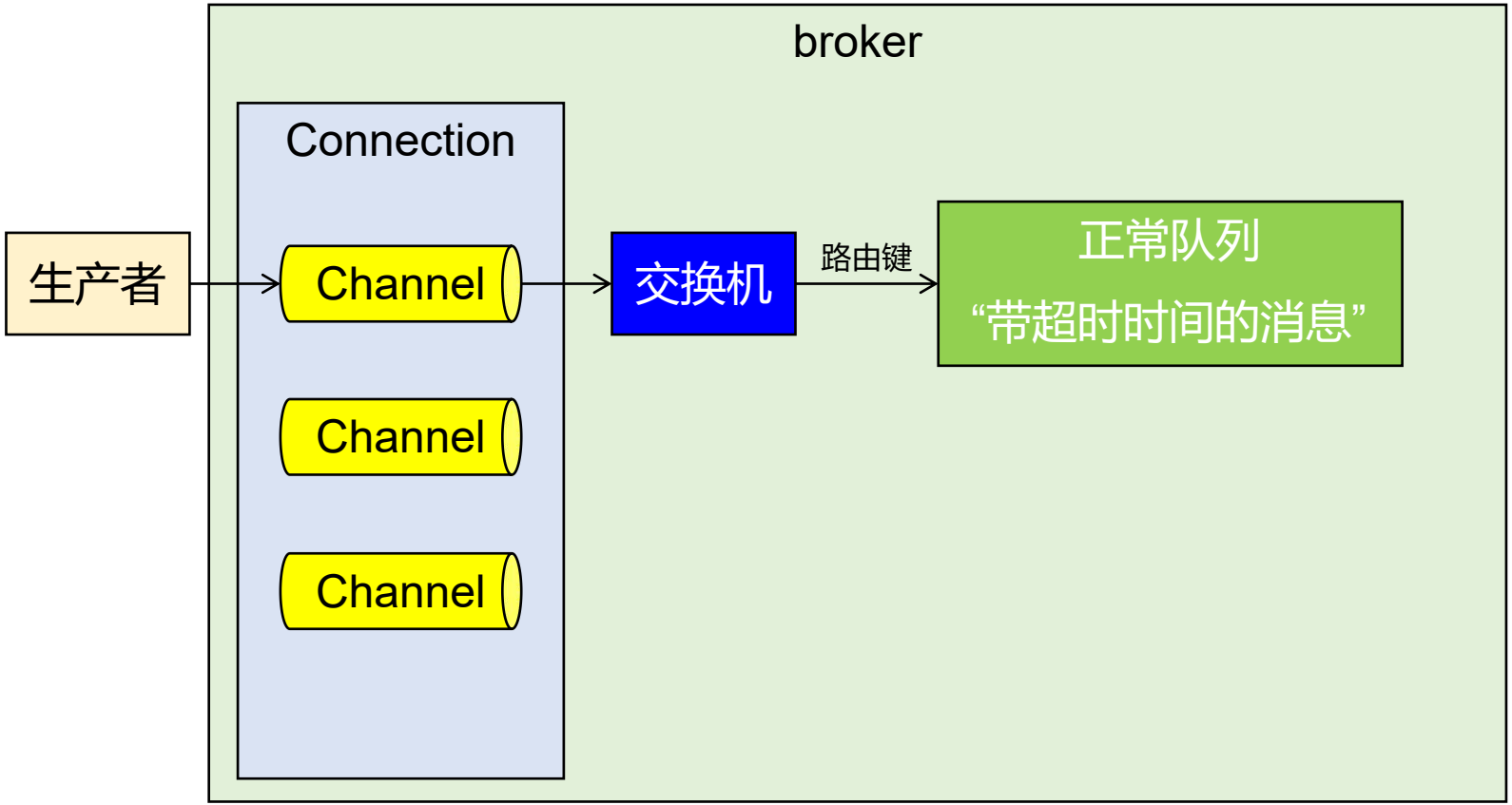
应用场景

推迟一段时间后执行指定操作

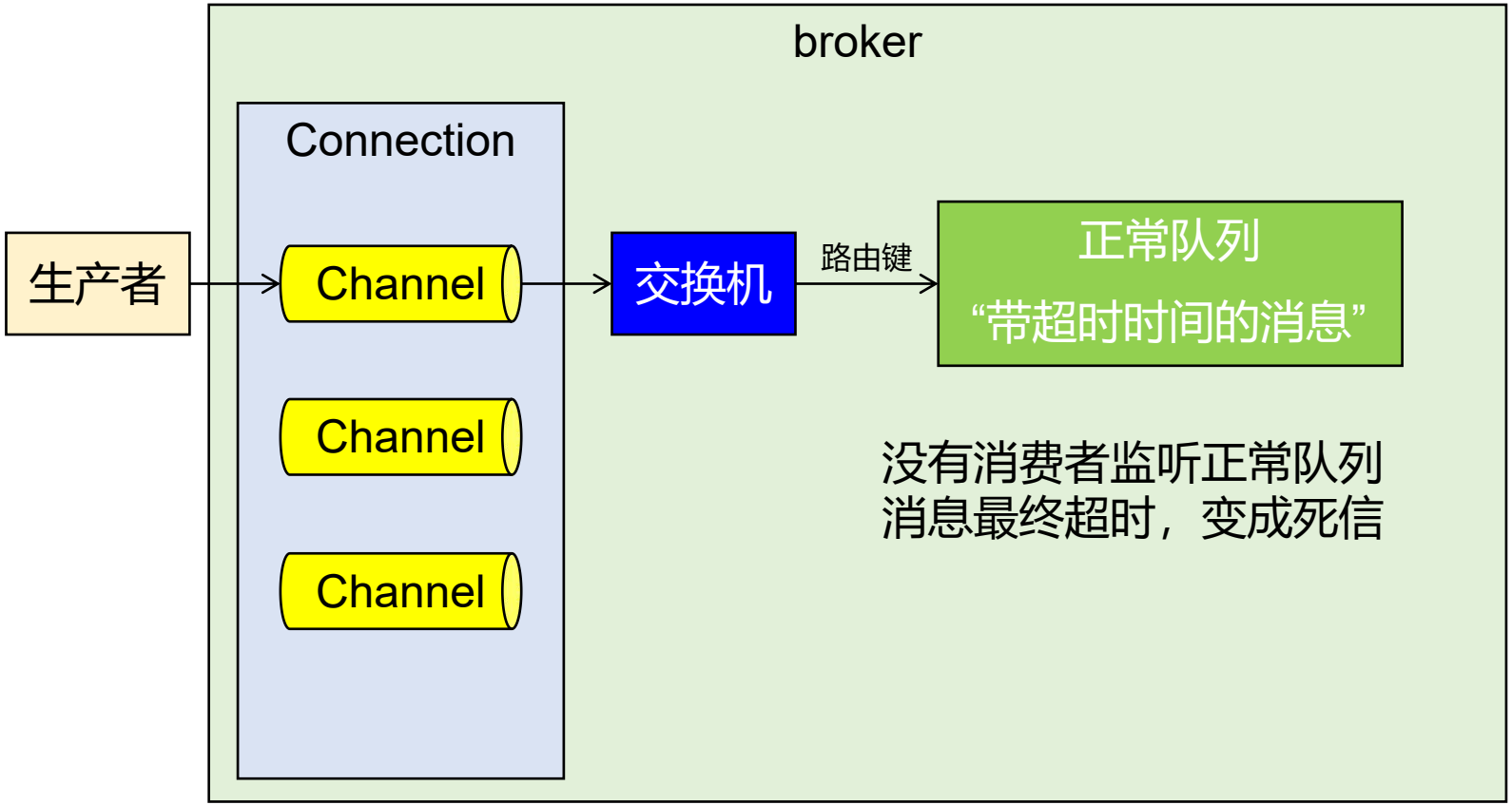
实现思路

- 方案1：借助消息超时时间+死信队列（就是刚刚我们测试的例子）
- 方案2：给RabbitMQ安装插件

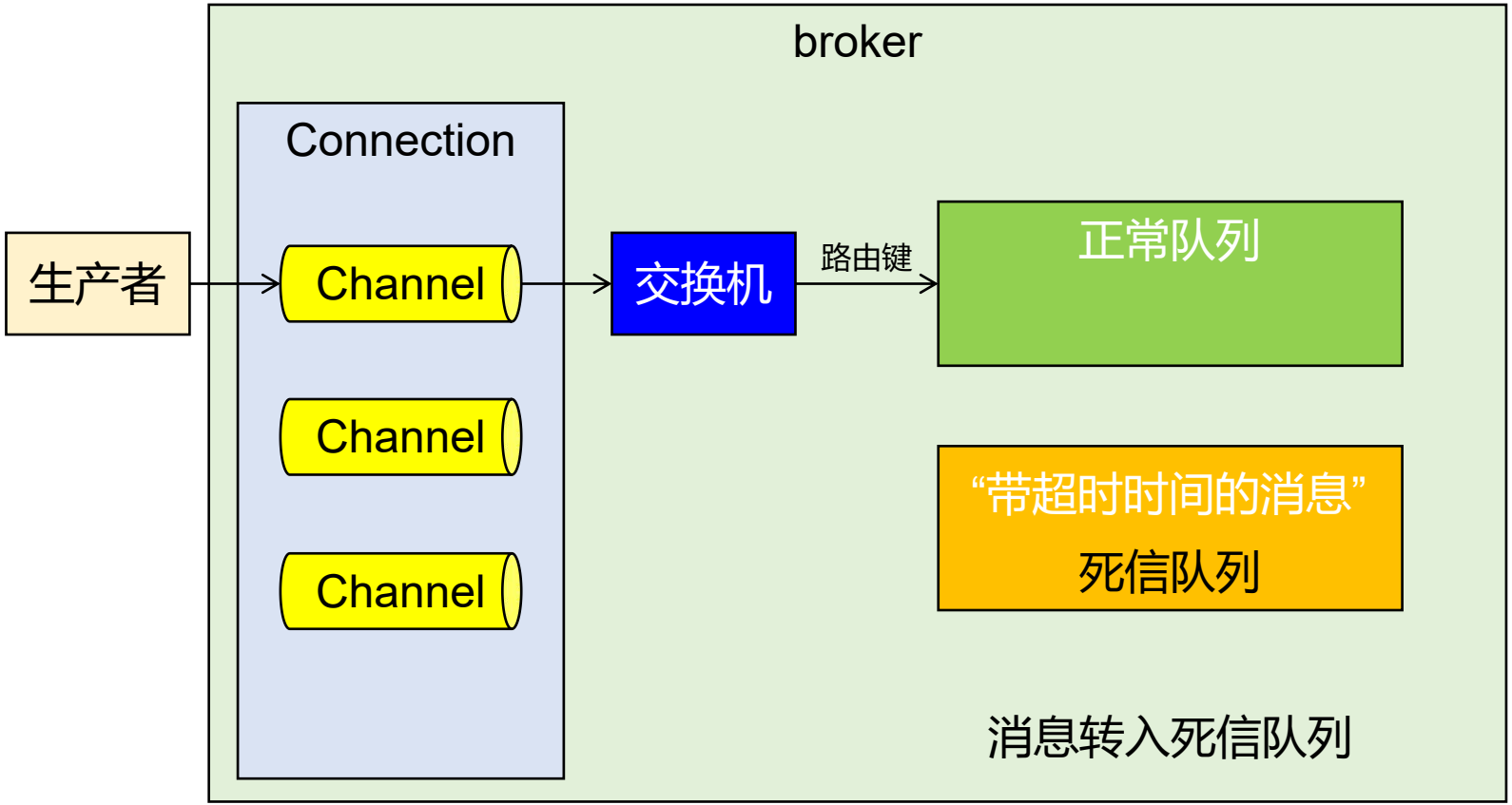
基于死信的延迟队列


broker
Connection
生产者
Channel
Channel
Channel
交换机
路由键
正常队列
“带超时时间的消息”

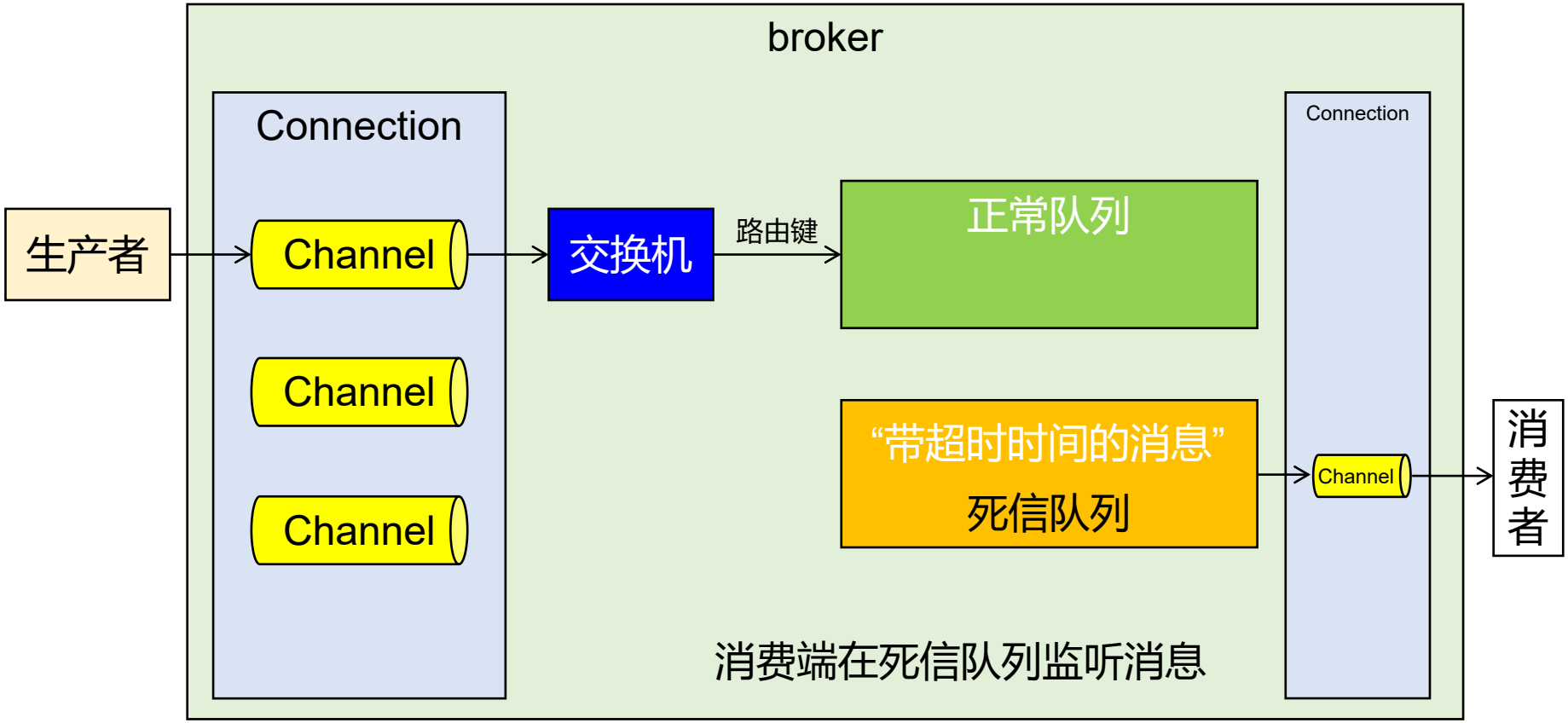
基于死信的延迟队列


broker
Connection
生产者
Channel
Channel
Channel
交换机
路由键
正常队列
“带超时时间的消息”
没有消费者监听正常队列
消息最终超时，变成死信

基于死信的延迟队列


broker
Connection
生产者
Channel
Channel
Channel
交换机
路由键
正常队列
“带超时时间的消息”
死信队列
消息转入死信队列

基于死信的延迟队列


broker
Connection
生产者
Channel
Channel
Channel
交换机
路由键
正常队列
“带超时时间的消息”
死信队列
Connection
Channel
消费者
消费端在死信队列监听消息

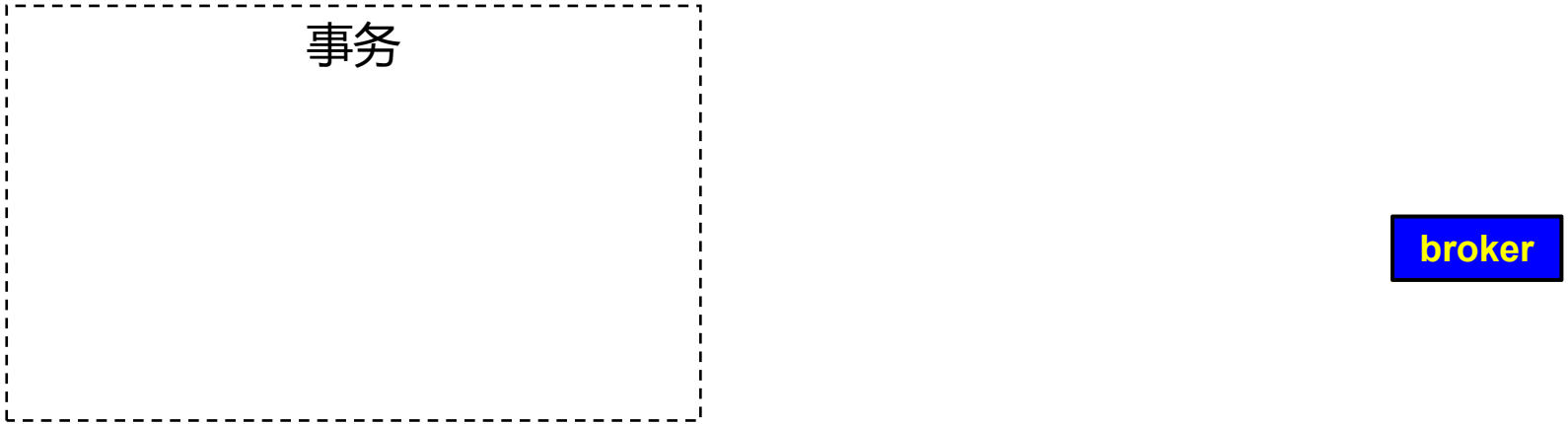
基于插件的延迟队列

- 操作文档：Operation012-DelayPlugin.md

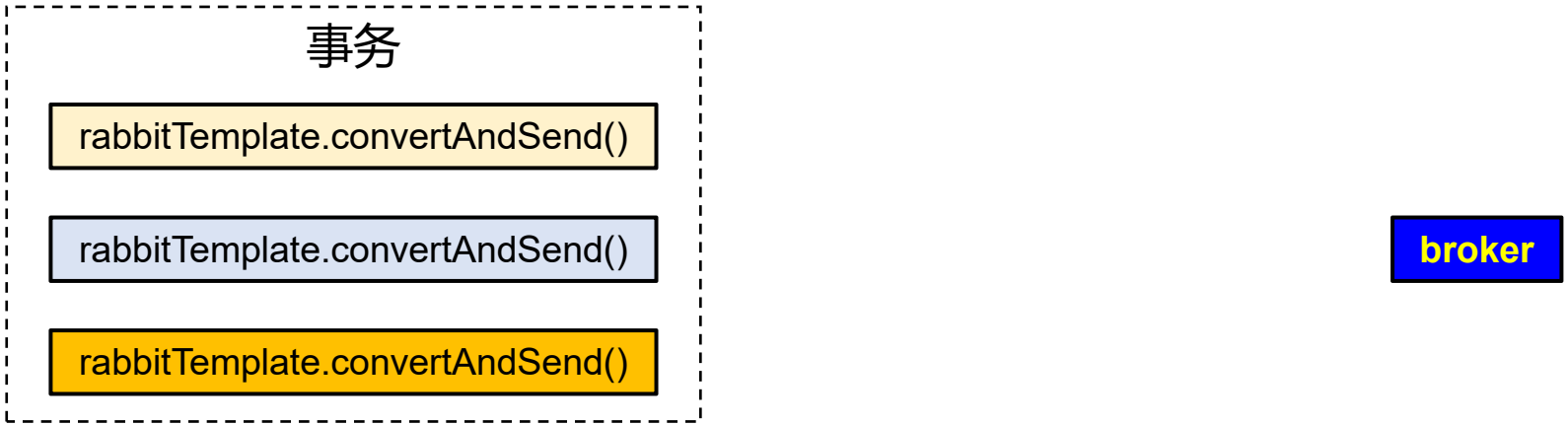
7 事务消息

Transactional Message

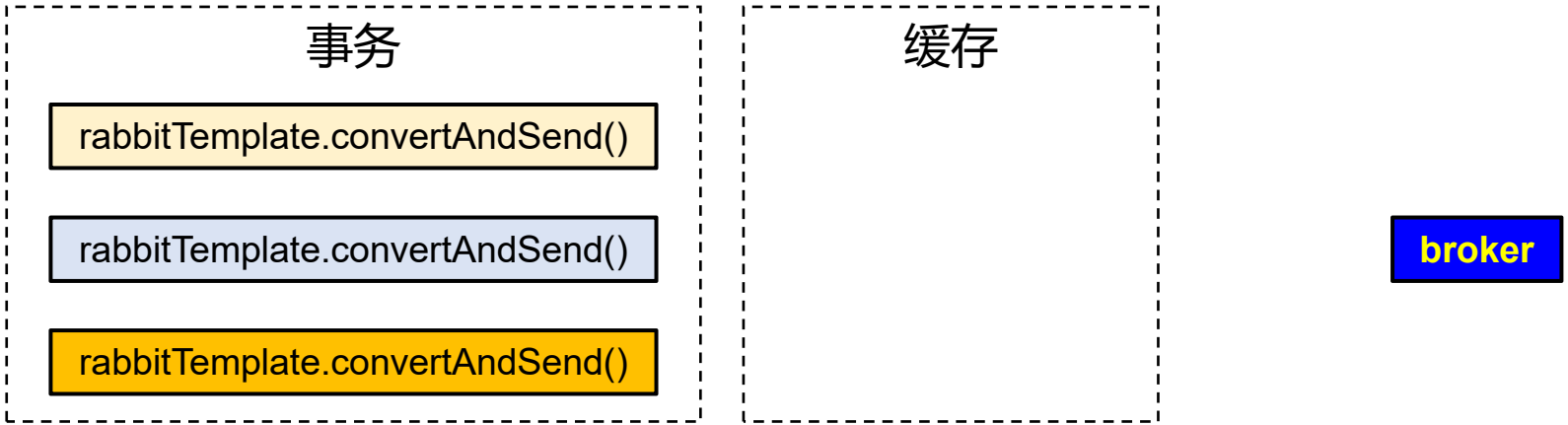
事务消息的机制说明：生产者端



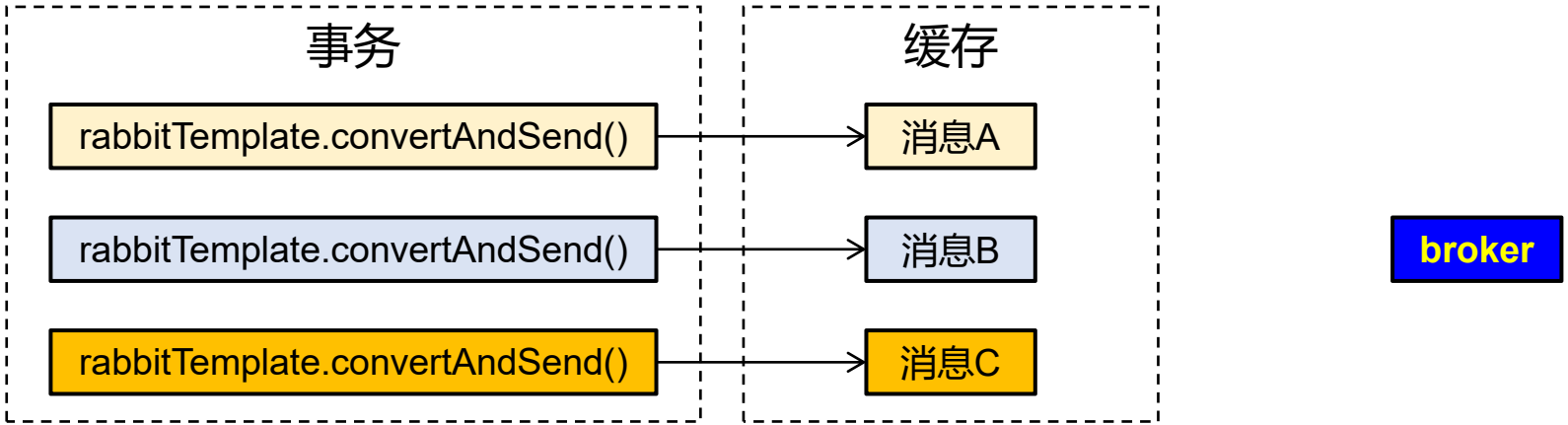
事务消息的机制说明：生产者端



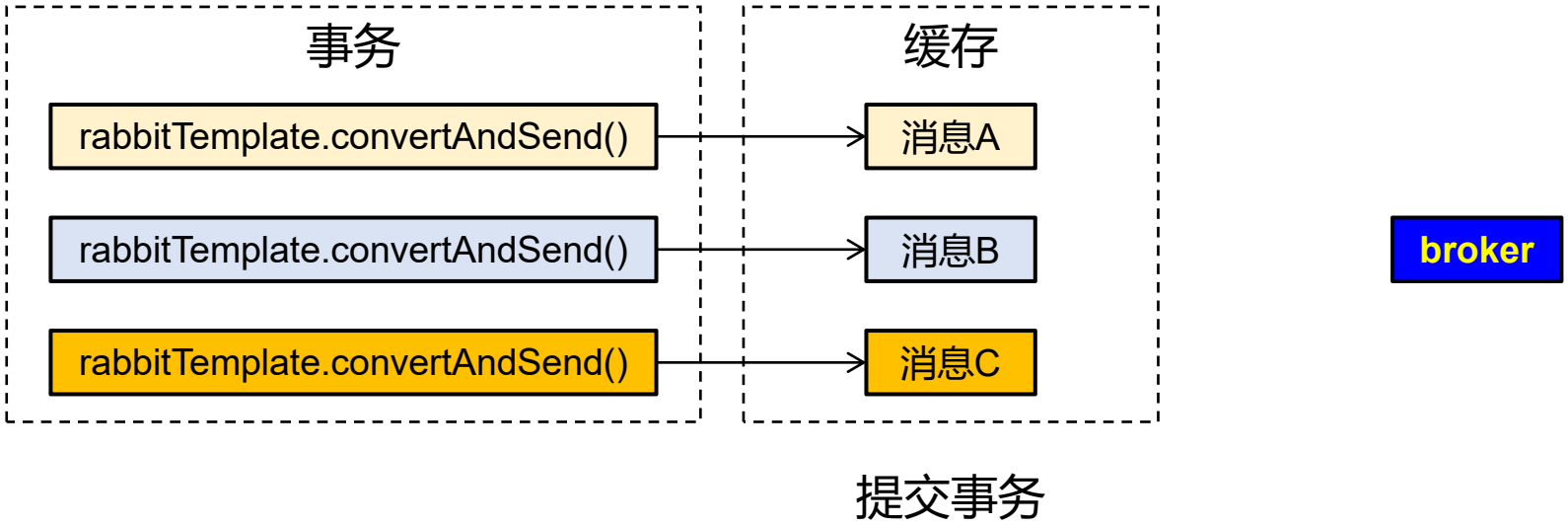
事务消息的机制说明：生产者端



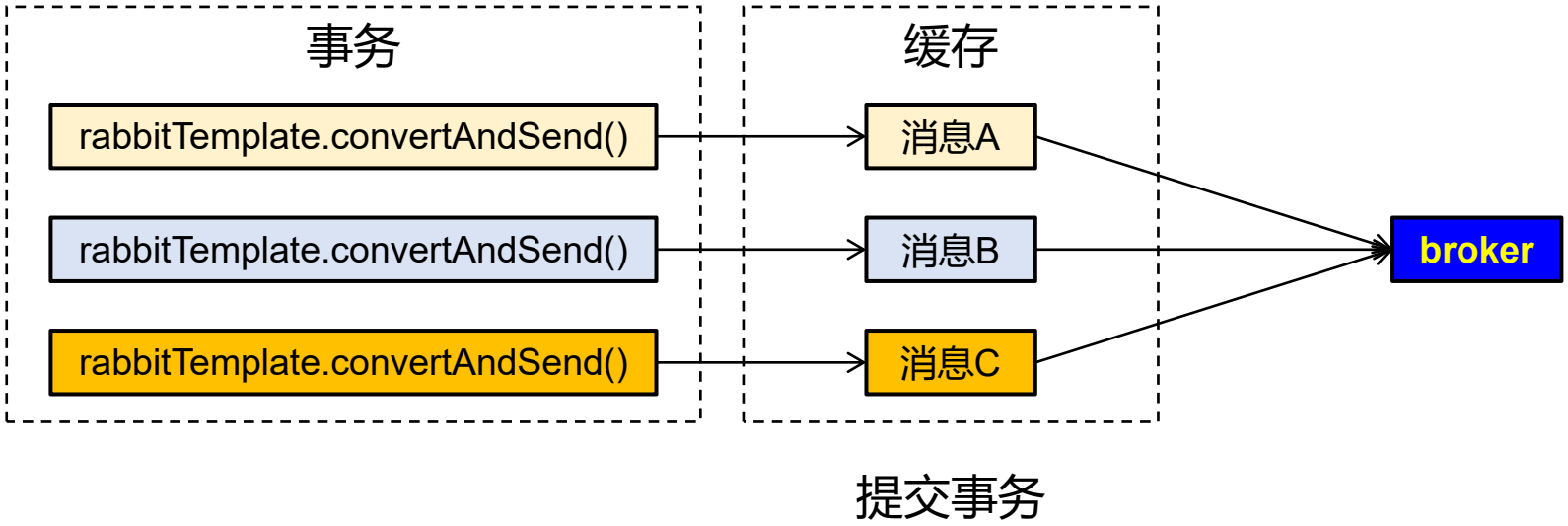
事务消息的机制说明：生产者端



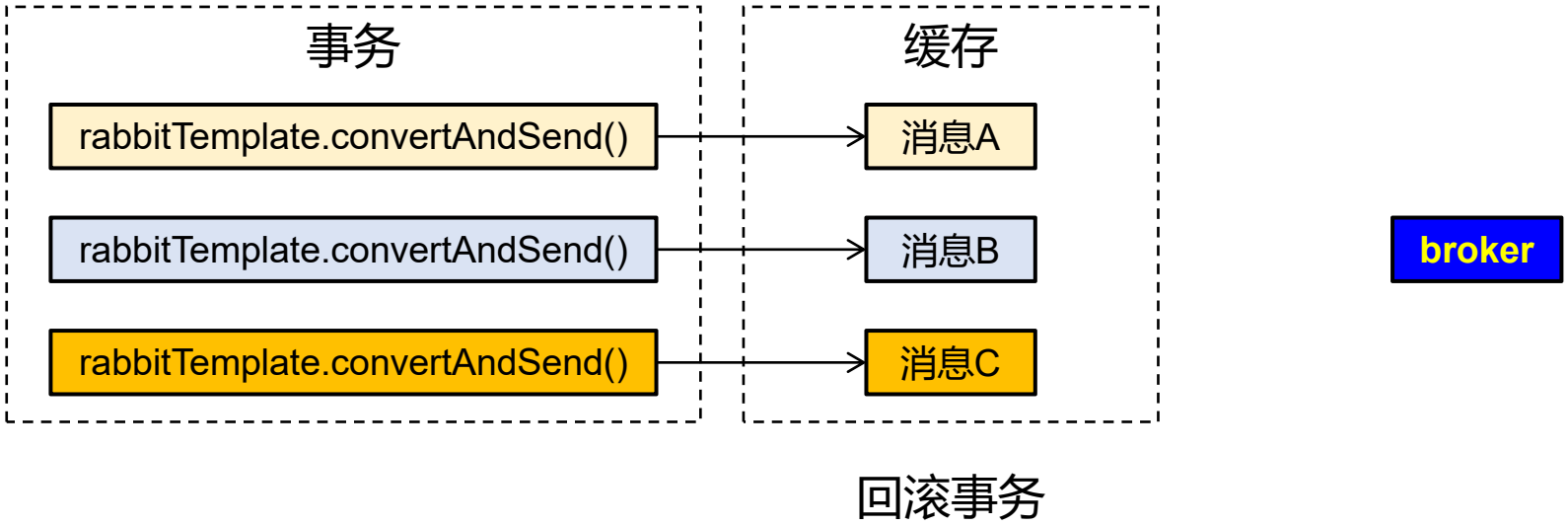
事务消息的机制说明：生产者端



事务消息的机制说明：生产者端



事务消息的机制说明：生产者端



事务消息的机制说明：生产者端

- 总结：
  - 在生产者端使用事务消息和消费端没有关系
  - 在生产者端使用事务消息仅仅是控制事务内的消息是否发送
  - 提交事务就把事务内所有消息都发送到交换机
  - 回滚事务则事务内任何消息都不会被发送
  - 操作文档：Operation013-Tx-Producer.md

事务消息的机制说明：消费端

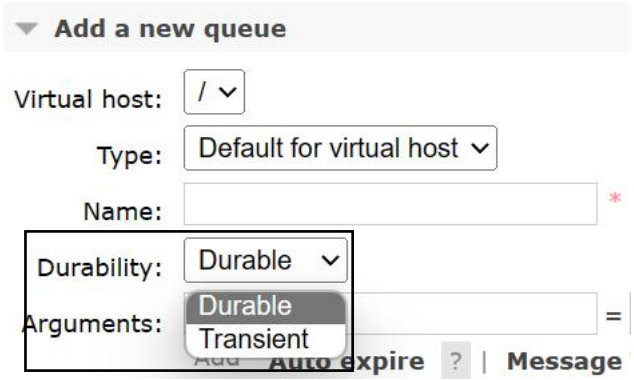
事务控制对消费端**无效**！！！

8 惰性队列

Lazy Queue

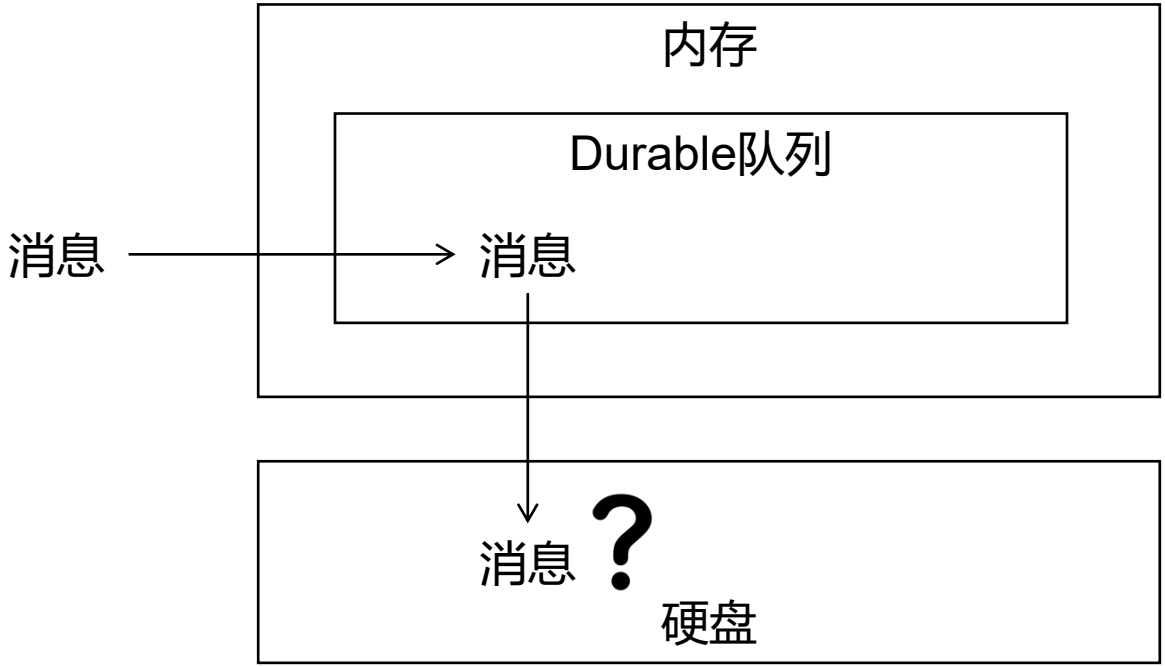
惰性队列：未设置惰性模式时队列的持久化机制

- 创建队列时，在Durability这里有两个选项可以选择
  - Durable：持久化队列，消息会持久化到硬盘上
  - Transient：临时队列，不做持久化操作，broker重启后消息会丢失

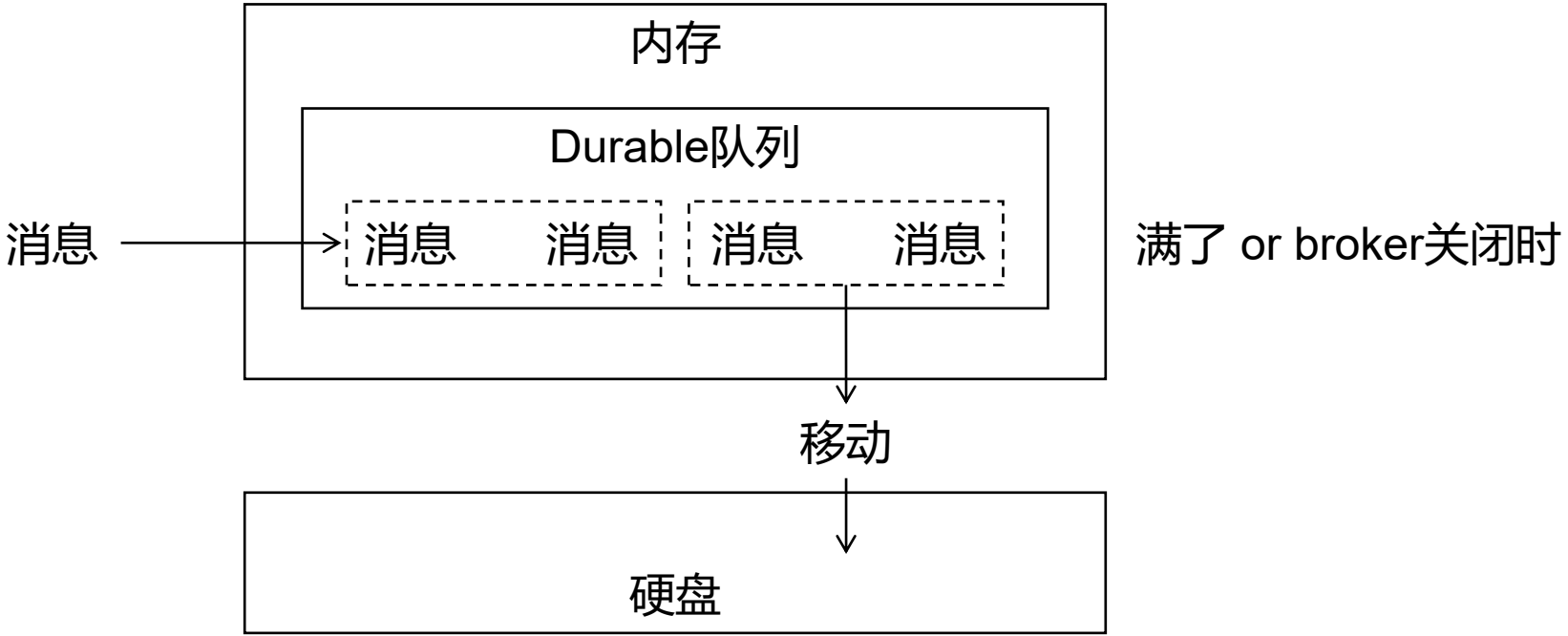


惰性队列：未设置惰性模式时队列的持久化机制

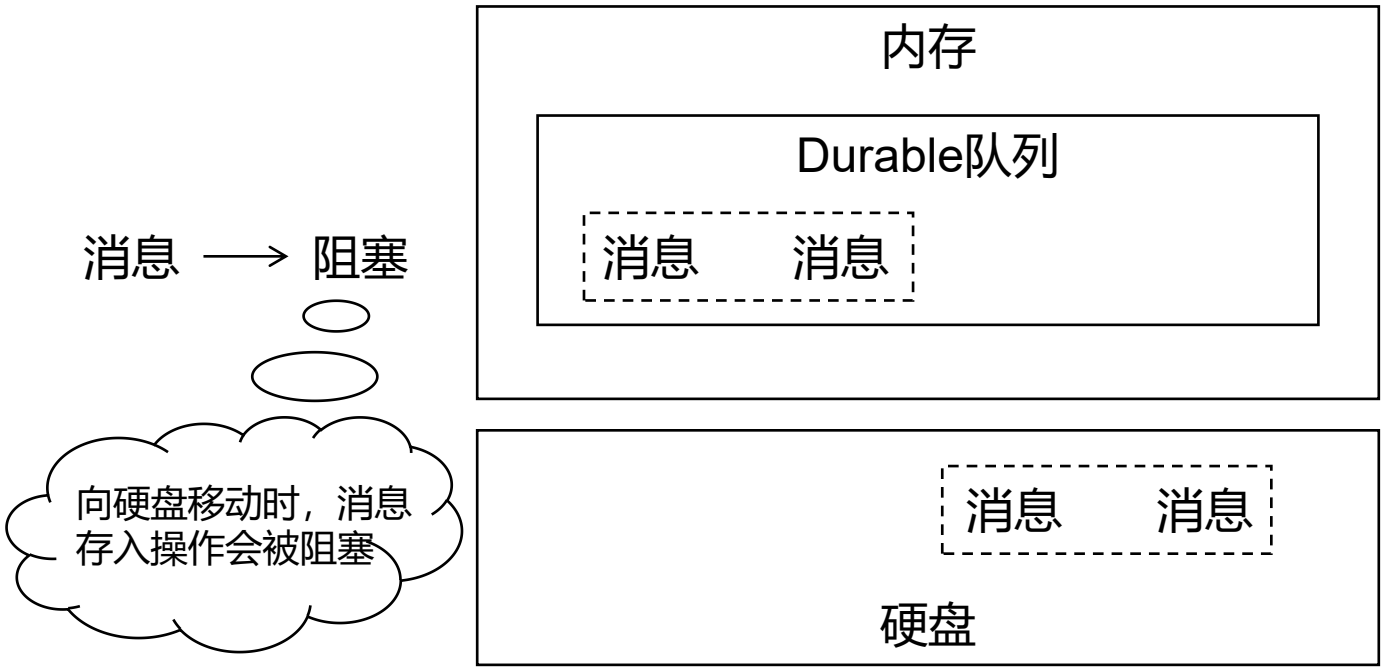
- 那么Durable队列在存入消息之后，是否是立即保存到硬盘呢?



惰性队列：未设置惰性模式时队列的持久化机制



惰性队列：未设置惰性模式时队列的持久化机制



惰性队列：惰性队列的工作方式

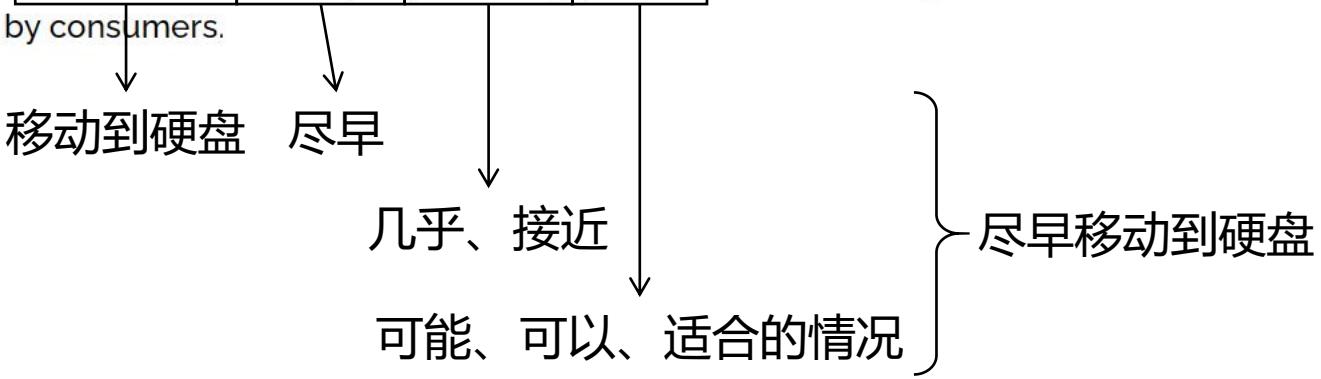
What is a Lazy Queue

A "lazy queue" is a classic queue which is running in `lazy` mode. When the "lazy" queue mode is set, messages in classic queues are moved to disk as early as practically possible. These messages are loaded into RAM only when they are requested by consumers.

惰性队列：惰性队列的工作方式

What is a Lazy Queue

A "lazy queue" is a classic queue which is running in `lazy` mode. When the "lazy" queue mode is set, messages in classic queues are moved to disk as early as practically possible. These messages are loaded into RAM only when they are requested by consumers.



惰性队列：惰性队列的工作方式

- 比较下面两个说法是否是相同的意思：
  - 立即移动到硬盘
  - 尽早移动到硬盘
- 我认为不一样：
  - 立即：消息刚进入队列时
  - 尽早：服务器不繁忙时

惰性队列：惰性队列的工作方式

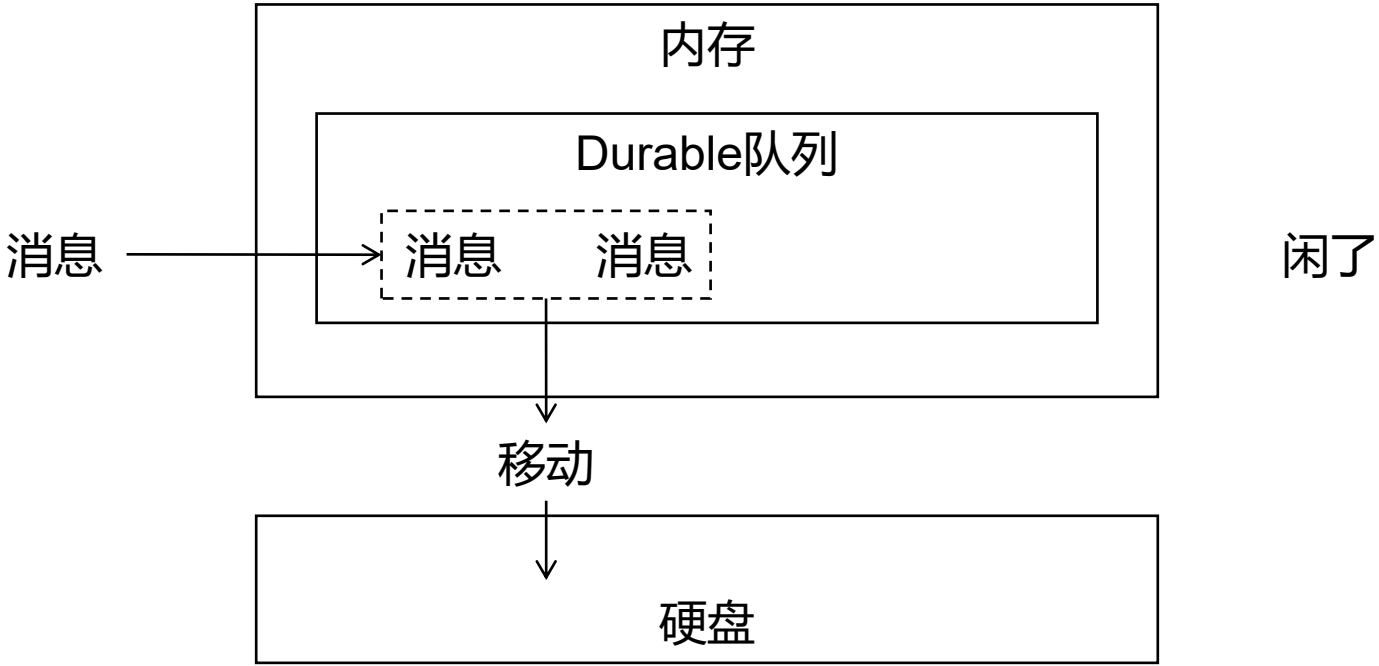
What is a Lazy Queue

A "lazy queue" is a classic queue which is running in `lazy` mode. When the "lazy" queue mode is set, messages in classic queues are moved to disk as early as practically possible. These messages are loaded into RAM only when they are requested by consumers.

↓

服务器相对空闲

惰性队列：惰性队列的工作方式



惰性队列：应用场景

One of the main reasons for using lazy queues is to support very long queues (many millions of messages). Queues can become very long for various reasons:

- consumers are offline / have crashed / are down for maintenance
- there is a sudden message ingress spike, producers are outpacing consumers
- consumers are slower than normal

官网原文

原文翻译：使用惰性队列的主要原因之一是支持非常长的队列（数百万条消息）。
由于各种原因，排队可能会变得很长：

- 消费者离线/崩溃/停机进行维护
- 突然出现消息进入高峰，生产者的速度超过了消费者
- 消费者比正常情况慢

惰性队列：惰性队列的工作方式

- 操作文档：Operation014-LazyQueue.md

9 优先级队列

Priority Queue

优先级队列：机制说明

- 默认情况：基于队列先进先出的特性，通常来说，先入队的先投递
- 设置优先级之后：优先级高的消息更大几率先投递
- 关键参数：x-max-priority

优先级队列：消息的优先级设置

- RabbitMQ允许我们使用一个正整数给消息设定优先级
- 消息的优先级数值取值范围：1~255
- RabbitMQ官网建议在1~5之间设置消息的优先级（优先级越高，占用CPU、内存等资源越多）

优先级队列：队列的优先级设置

- 队列在声明时可以指定参数：x-max-priority
- 默认值：0 此时消息即使设置优先级也无效
- 指定一个正整数值：消息的优先级数值不能超过这个值

- 操作文档：Operation015-PriorityQueue.md

Part 04

集群篇

工作机制、集群搭建、负载均衡、仲裁队列、流式队列……

目录

1 工作机制

2 集群搭建

3 负载均衡

4 仲裁队列

5 流式队列

6 异地容灾

尚硅谷

1 工作机制

Cluster work way

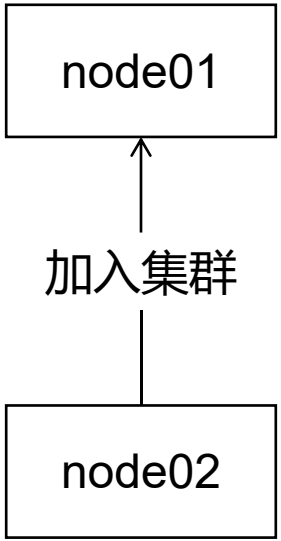
基本诉求

避免单点故障

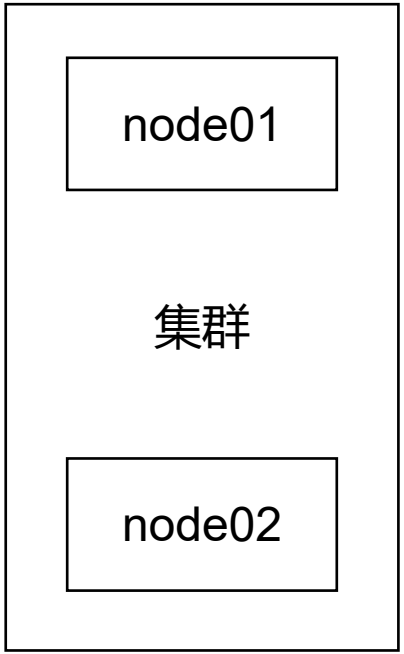
大流量场景分摊负载

数据同步

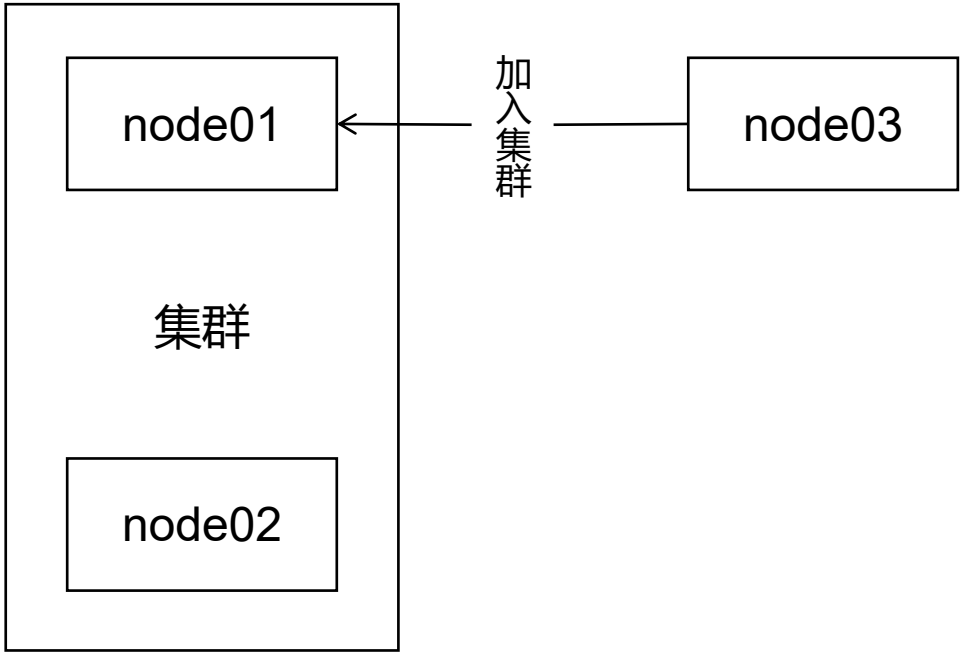
工作机制



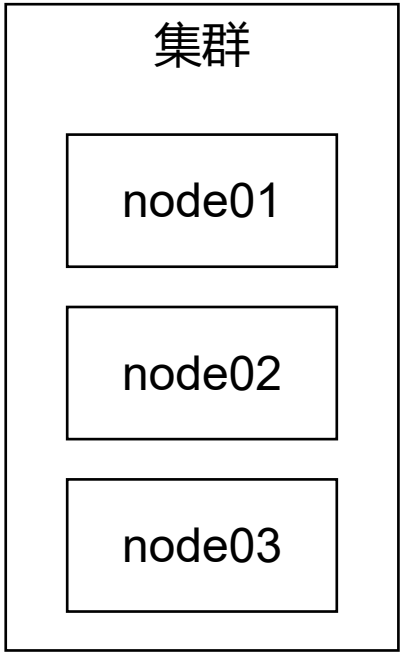
工作机制



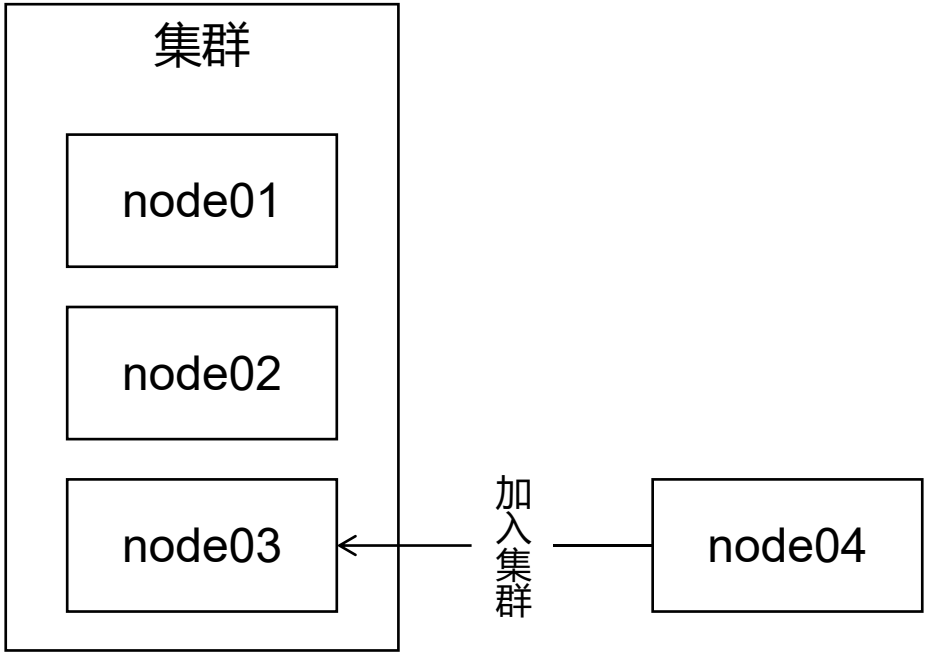
工作机制



工作机制

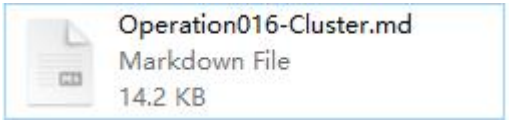


工作机制



2 集群搭建

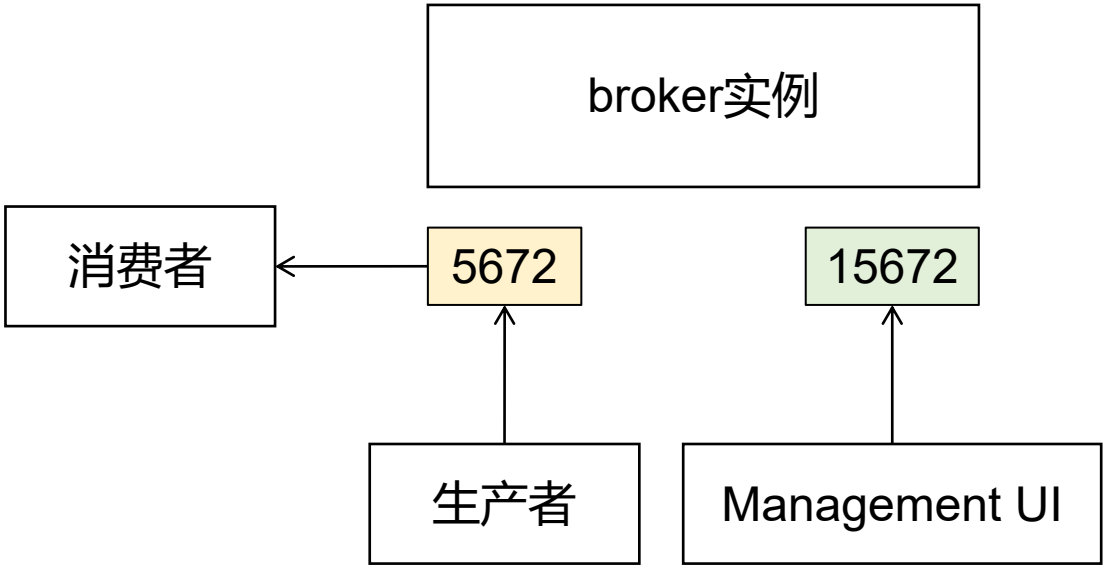
Cluster Foundation



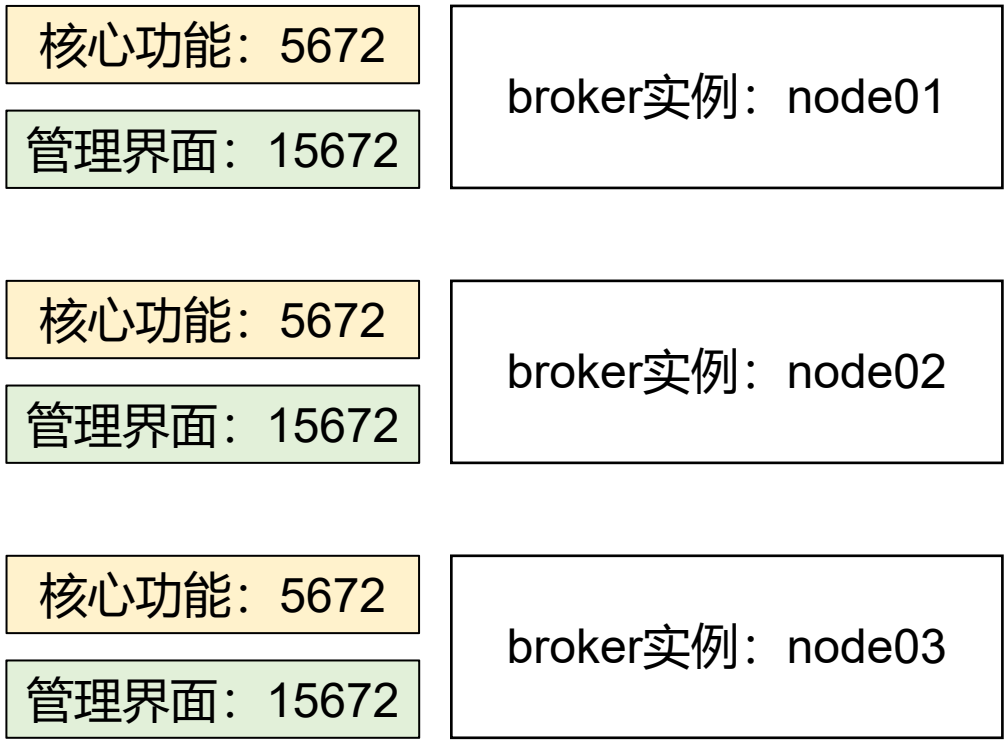
3 负载均衡

Load Balance

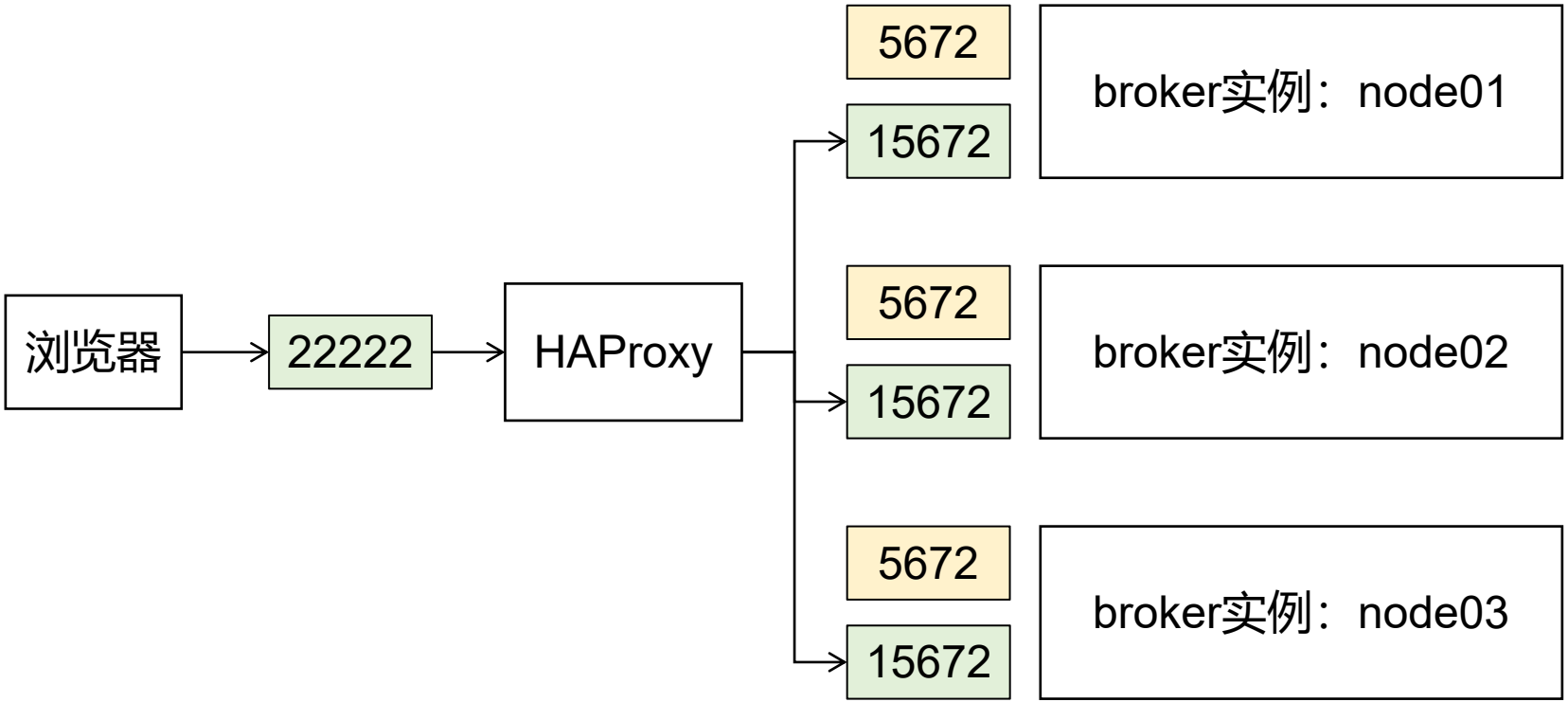
两个需要暴露的端口



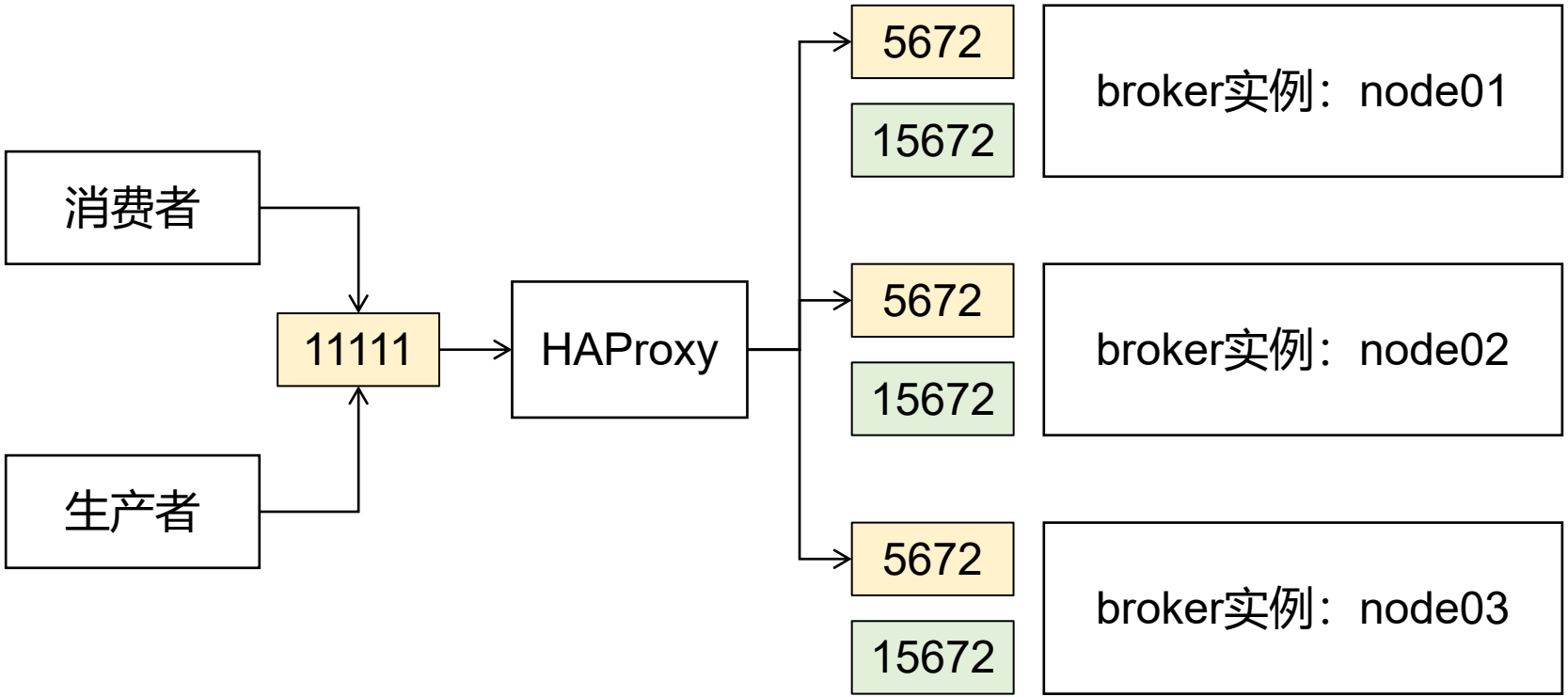
目前集群方案


核心功能：5672
管理界面：15672
broker实例：node01
核心功能：5672
管理界面：15672
broker实例：node02
核心功能：5672
管理界面：15672
broker实例：node03

管理界面负载均衡


浏览器
22222
HAProxy
5672
15672
broker实例：node01
5672
15672
broker实例：node02
5672
15672
broker实例：node03

核心功能负载均衡

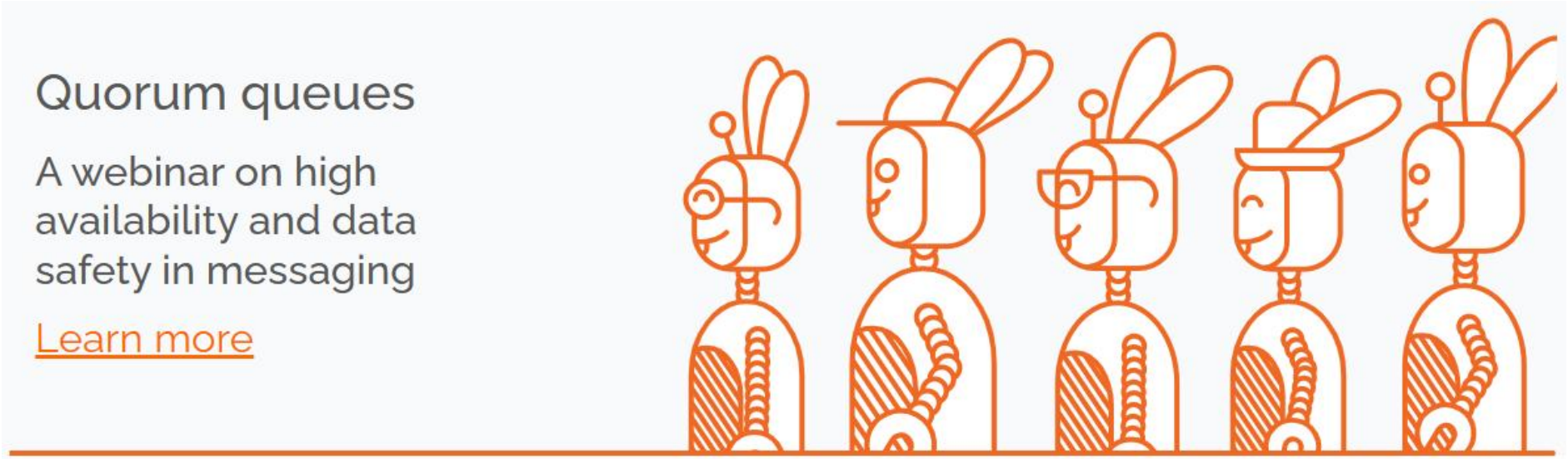

消费者
生产者
11111
HAProxy
5672
15672
broker实例：node01
5672
15672
broker实例：node02
5672
15672
broker实例：node03

4 仲裁队列

Quorum Queue

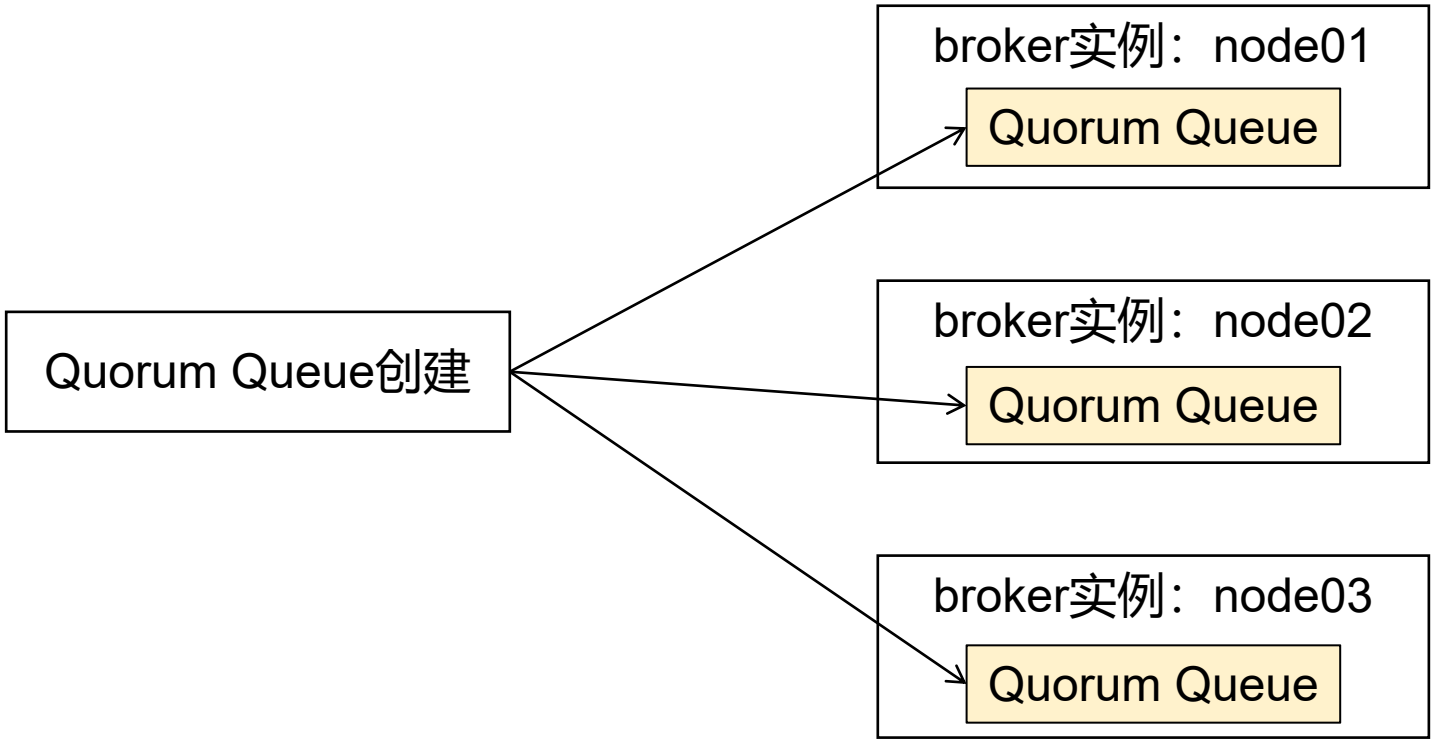
Operation017-Quorum.md
Markdown File
2.30 KB

首页强推 (version≤3.12.x)



RabbitMQ 3.8.x版本的主要更新内容
未来有可能取代Classic Queue

集群化分布

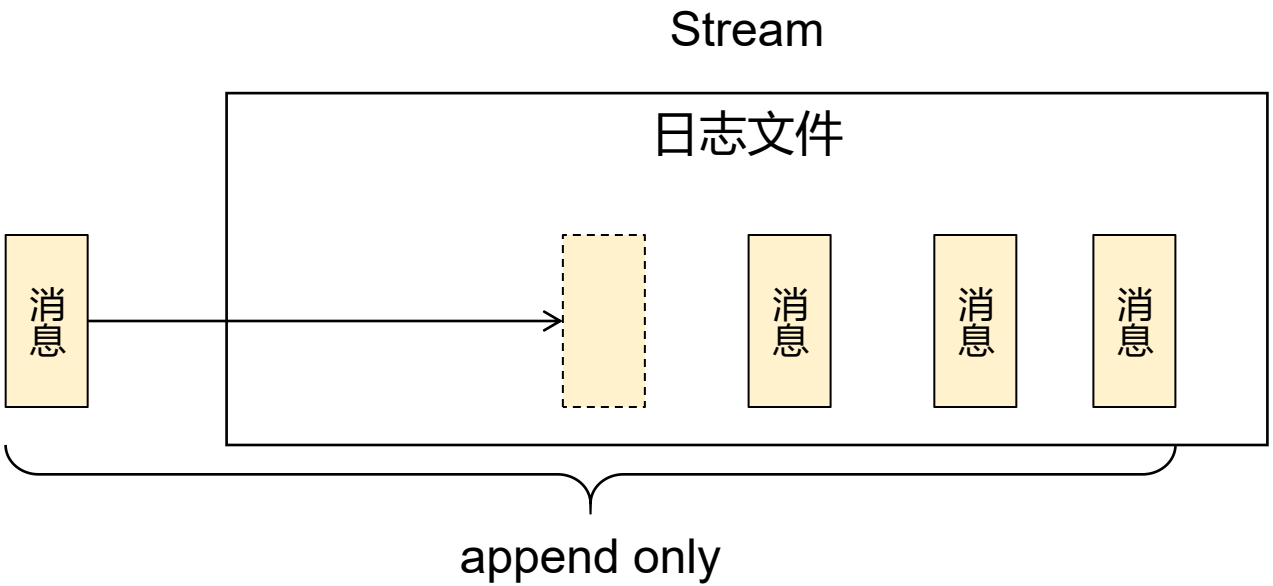

Quorum Queue创建
broker实例：node01
Quorum Queue
broker实例：node02
Quorum Queue
broker实例：node03
Quorum Queue

5 流式队列

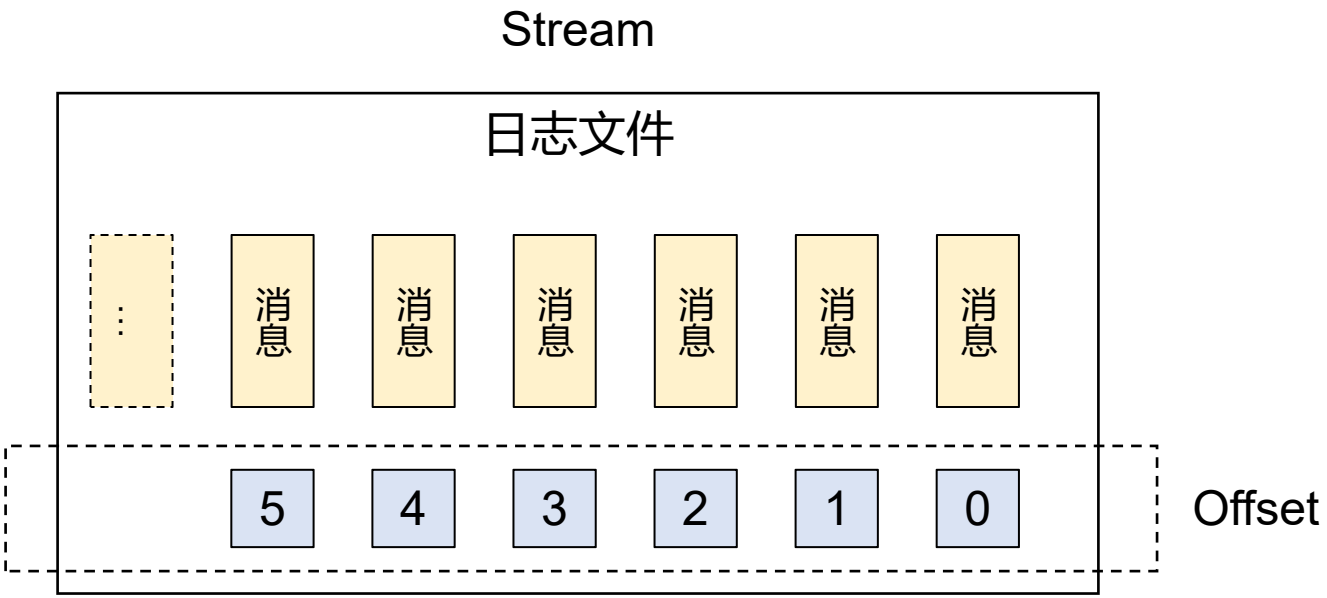
Stream

Operation018-Stream.md
Markdown File
5.23 KB

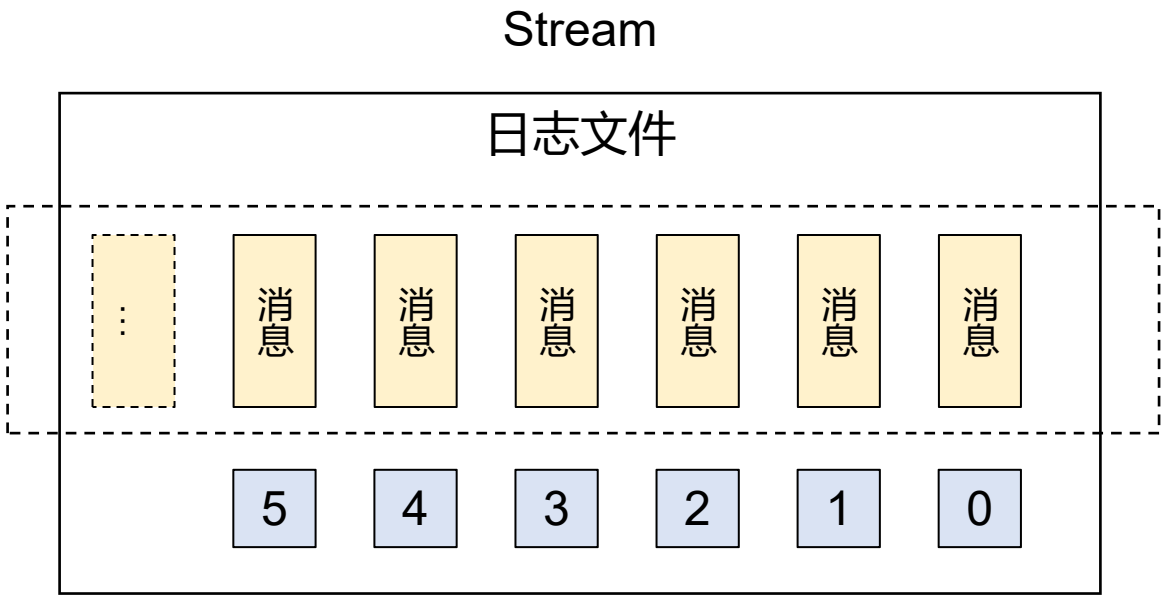
核心机制



核心机制



核心机制



消费之后仍不删除，可以重复消费

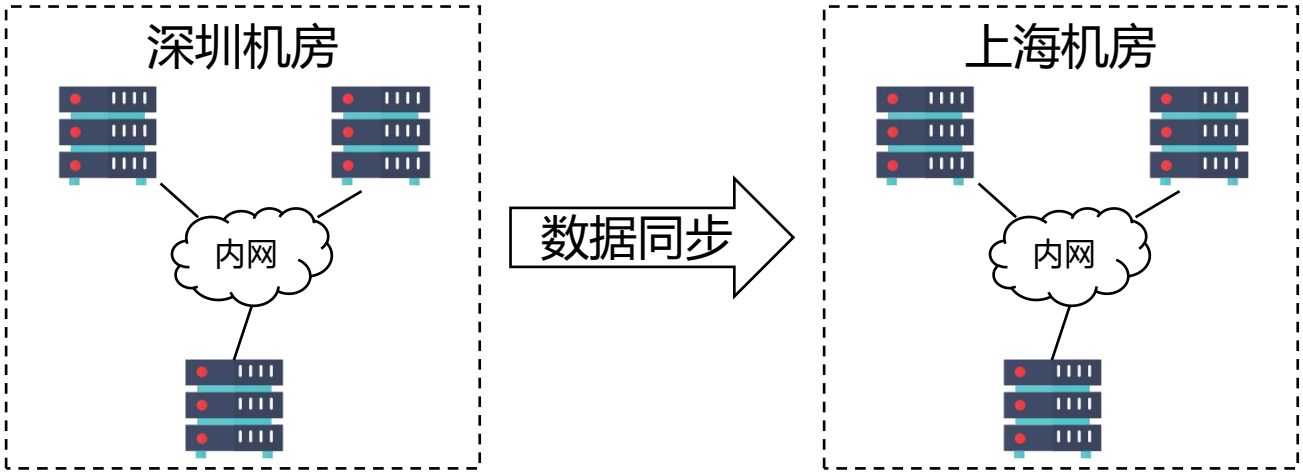
总体评价

- 从客户端支持角度来说，生态尚不健全
- 从使用习惯角度来说，和原有队列用法不完全兼容
- 从竞品角度来说，**像Kafka但远远比不上Kafka**
- 从应用场景角度来说：
  - 经典队列：适用于系统内部异步通信场景
  - 流式队列：适用于系统间跨平台、大流量、实时计算场景（Kafka主场）
- 使用建议：Stream队列在目前企业实际应用非常少，真有特定场景需要使用肯定会倾向于使用Kafka，而不是RabbitMQ Stream
- 未来展望：Classic Queue已经有和Quorum Queue合二为一的趋势，Stream也有加入进来整合成一种队列的趋势，但Stream内部机制决定这很难

6 异地容灾

Disaster Recovery at a Different Location

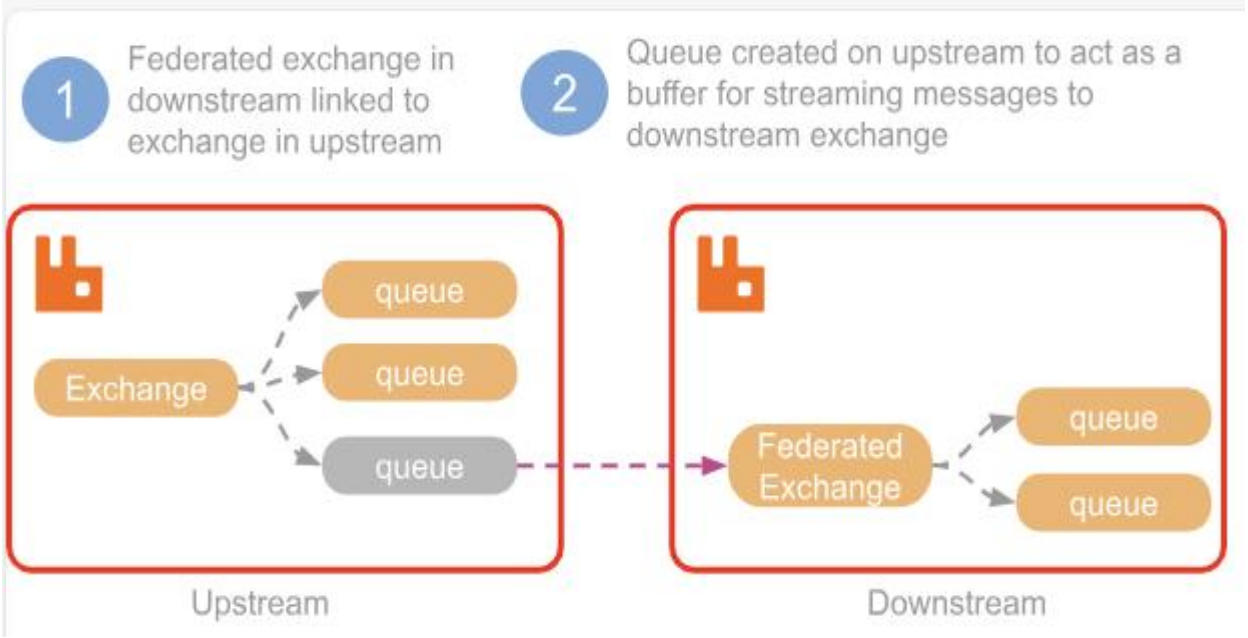
异地容灾的基本思路



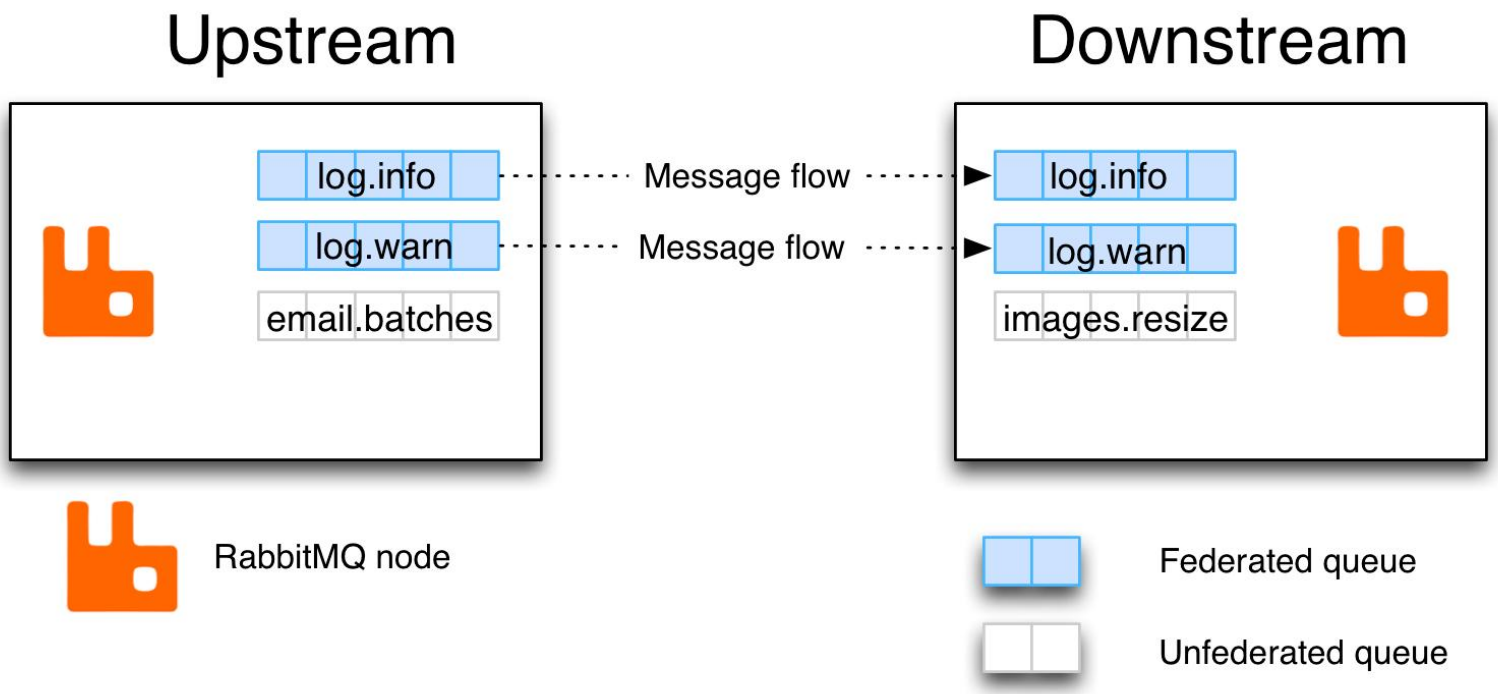
RabbitMQ异地容灾方案

- Federation插件（参见Operation019-Federation.md）
- Shovel插件（参见Operation020-Shovel.md）

Federation插件：联邦交换机



Federation插件：联邦队列



Shovel插件

- Shovel和Federation的主要区别：
  - Shovel更简洁一些
  - Federation更倾向于跨集群使用，而Shovel是否跨集群都可以
  - Shovel源队列中的消息经过数据转移后相当于被消费了


谢谢观看
尚硅谷