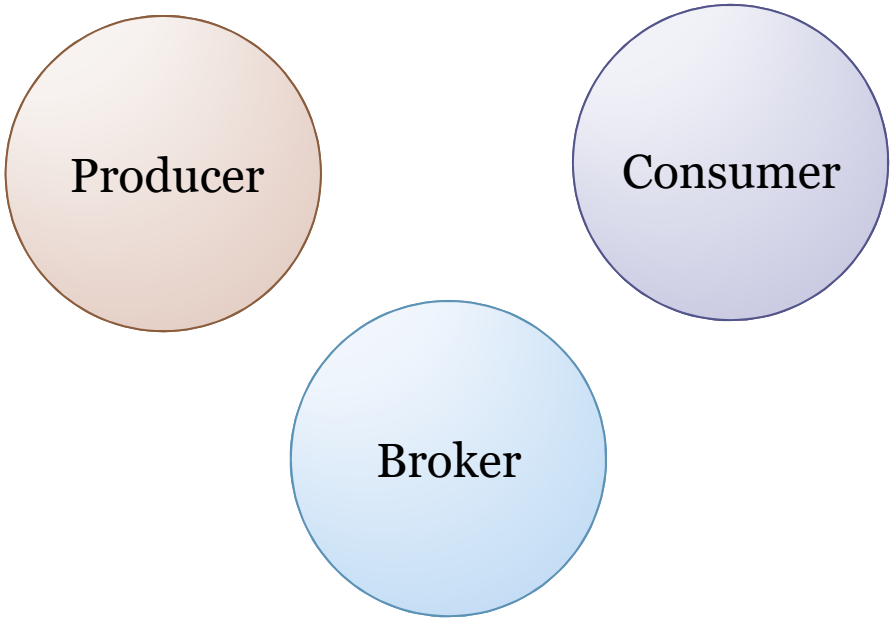
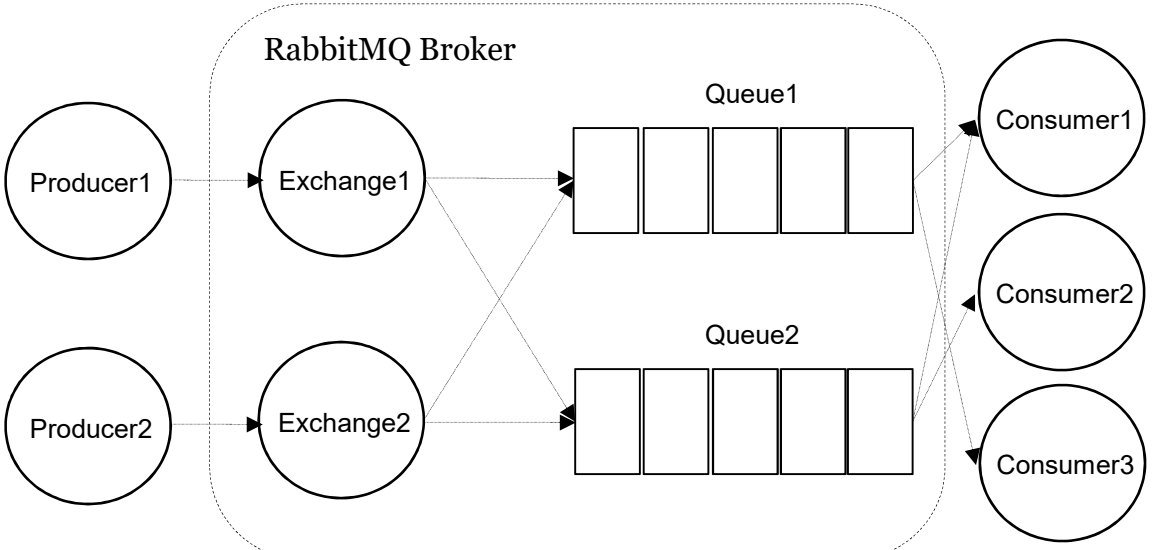
# RabbitMQ大型电商网站实践

朱忠华

# 主要内容



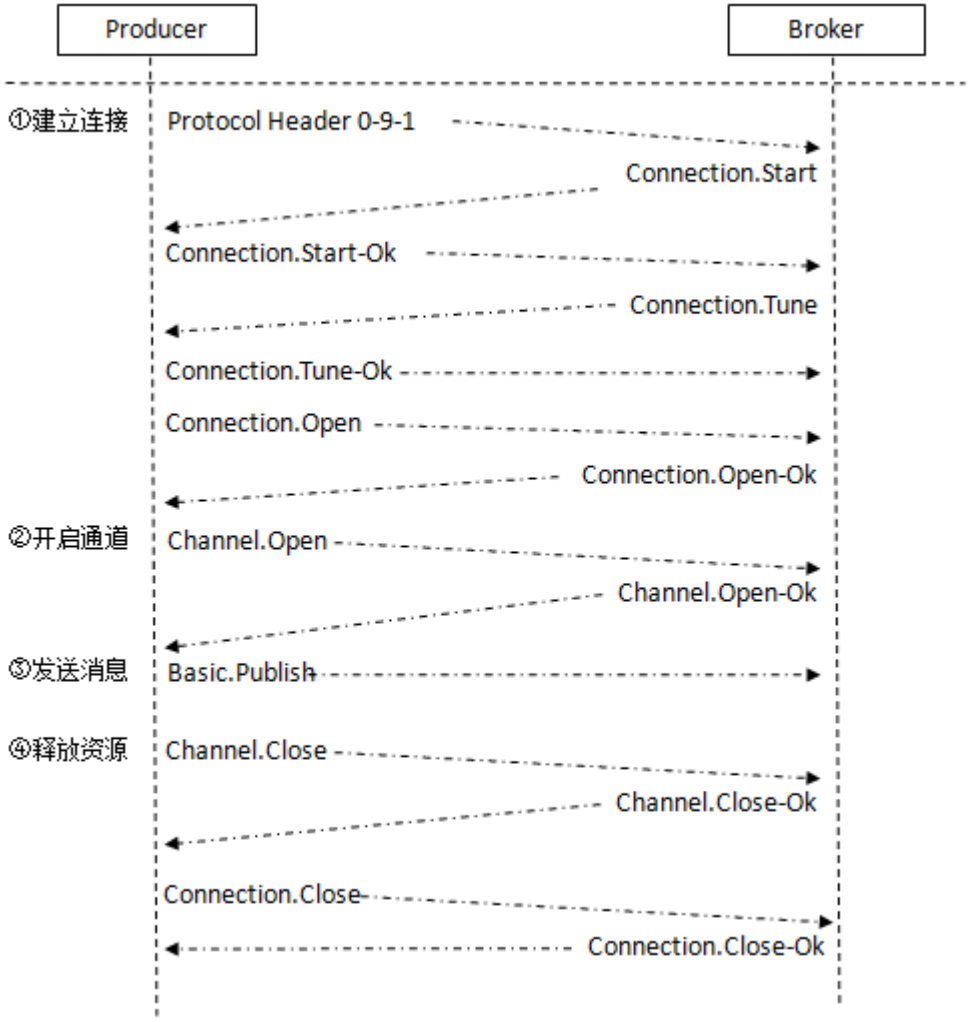
# RabbitMQ简介


RabbitMQ Broker
Queue1
Queue2
Producer1
Producer2
Exchange1
Exchange2
Consumer1
Consumer2
Consumer3

# 客户端——Producer

```
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
String message = "Hello World!";
channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
        MessageProperties.PERSISTENT_TEXT_PLAIN,
        message.getBytes());
channel.close();
connection.close();
```

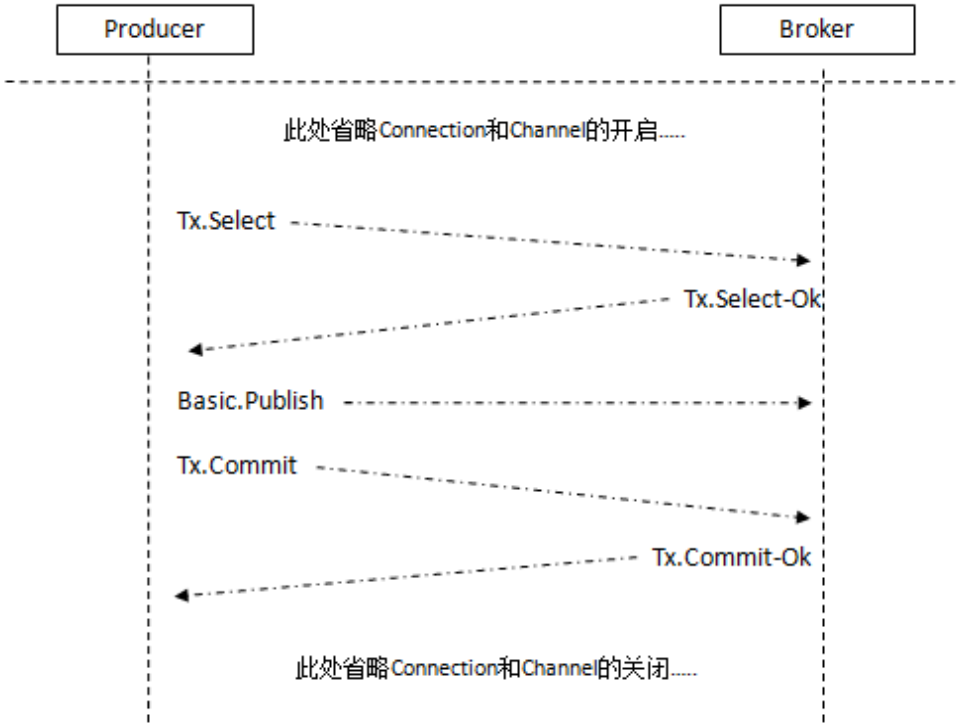
发送的消息是否正确的送达到Broker中？



# 客户端——Producer

解决：AMQP协议——事务机制

```
try {
    channel.txSelect();
    channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
            MessageProperties.PERSISTENT_TEXT_PLAIN,
            message.getBytes());
    channel.txCommit();
} catch (Exception e) {
    e.printStackTrace();
    channel.txRollback();
}
```

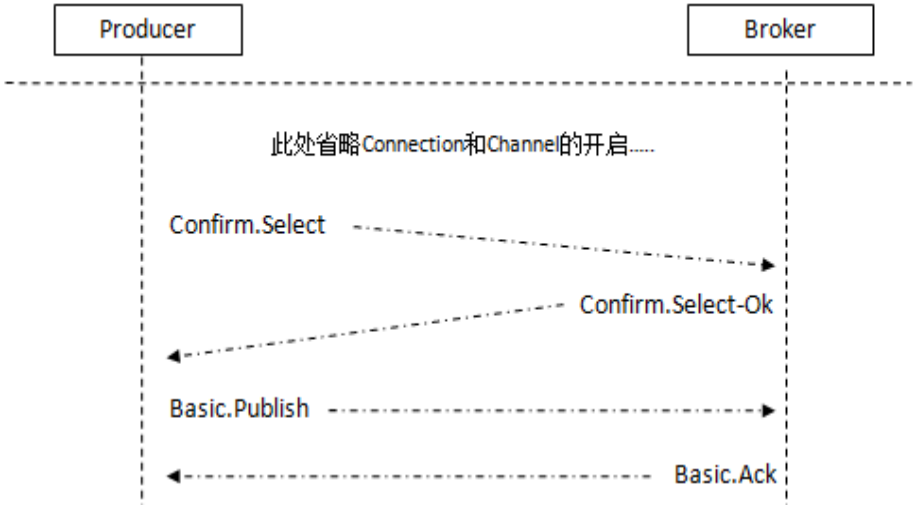



使用事务机制会“吸干”RabbitMQ的性能，
那么有没有更好的解决方案？

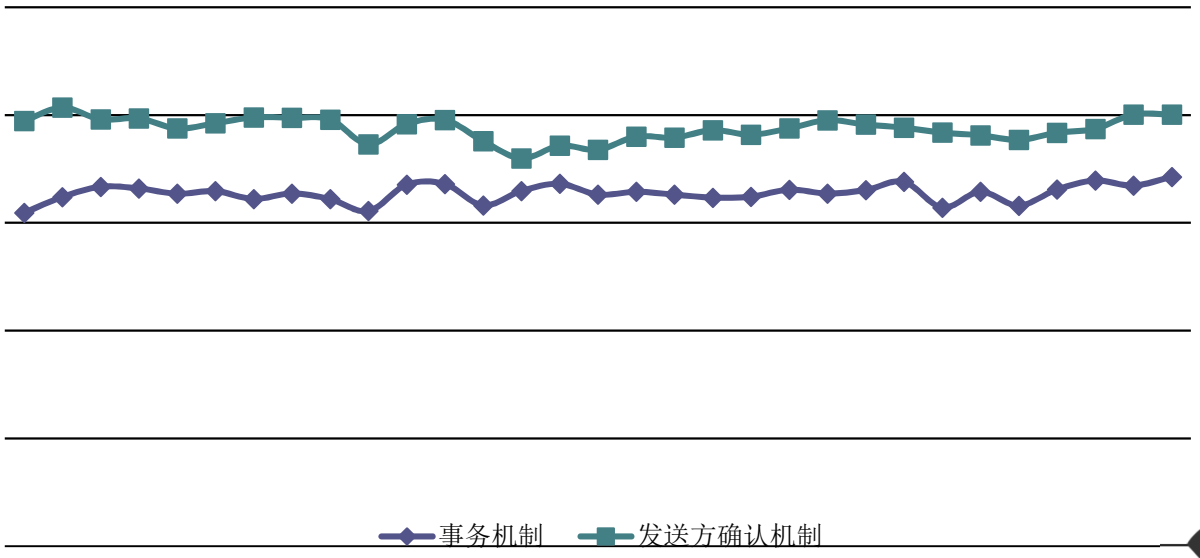
# 客户端——Producer

解决：RabbitMQ提供了一个改进方案——发送方确认（publisher confirm）机制

```
try {
    channel.confirmSelect();
    channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
            MessageProperties.PERSISTENT_TEXT_PLAIN,
            message.getBytes());
    if (!channel.waitForConfirms()) {
        //do something....
    }
} catch (InterruptedException e) {
    //do something else....
}
```

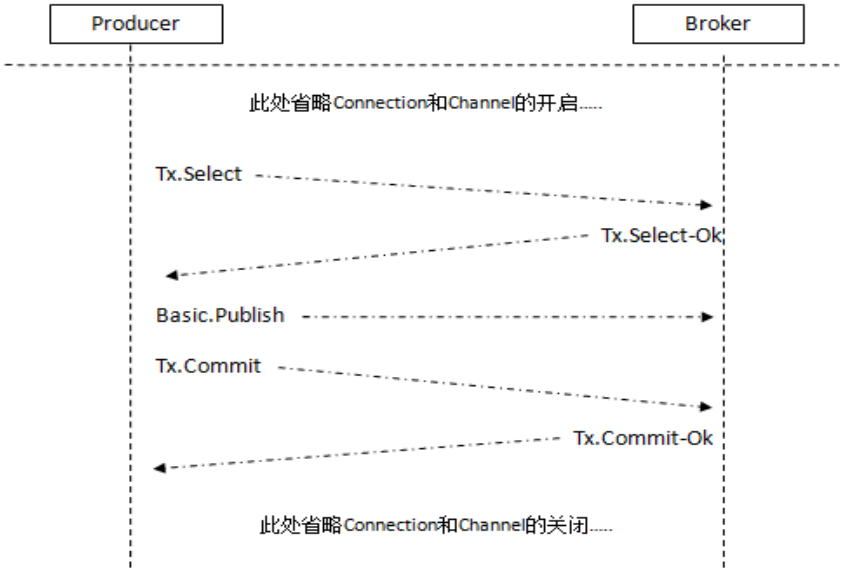


# 客户端——Producer

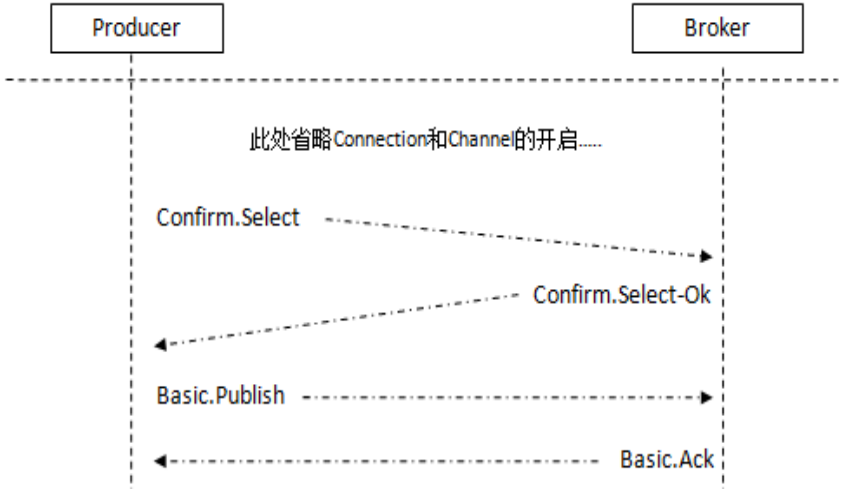



# 客户端——Producer

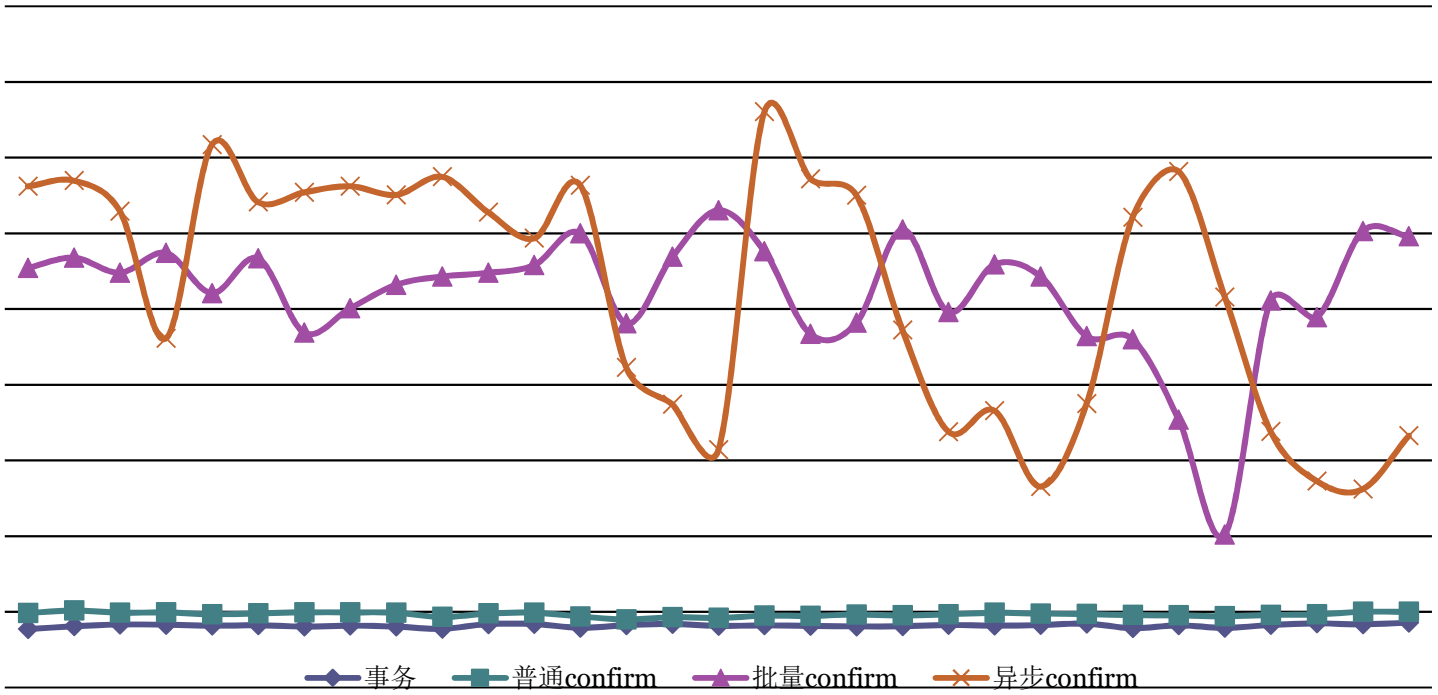
```
try {
    channel.txSelect();
    channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
            MessageProperties.PERSISTENT_TEXT_PLAIN,
            message.getBytes());
    channel.txCommit();
} catch (Exception e) {
    e.printStackTrace();
    channel.txRollback();
}
```




```
try {
    channel.confirmSelect();
    channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
            MessageProperties.PERSISTENT_TEXT_PLAIN,
            message.getBytes());
    if (!channel.waitForConfirms()) {
        //do something....
    }
} catch (InterruptedException e) {
    //do something else....
}
```



# 客户端——Producer

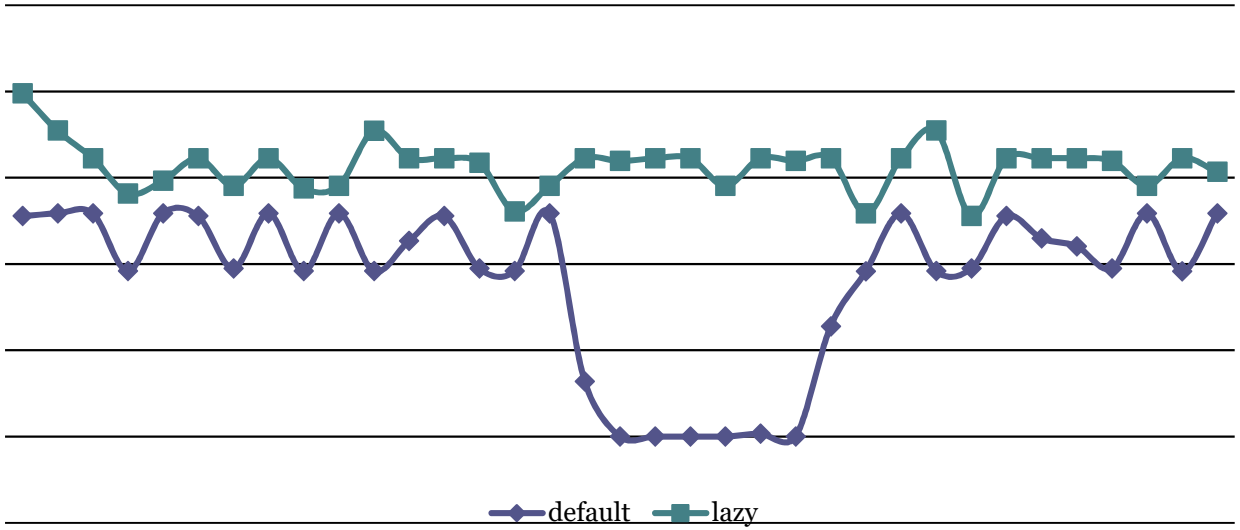



单Queue的QPS达到上限了嚒？还能再优化么？

# 性能优化（1）——合并&压缩

CPU

# 性能优化（2）——Lazy Queue

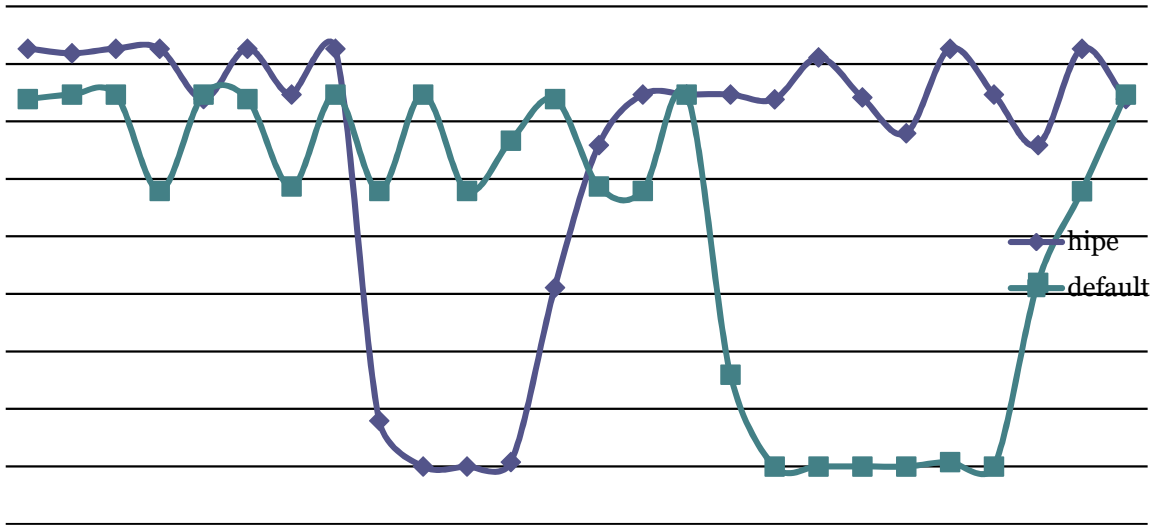


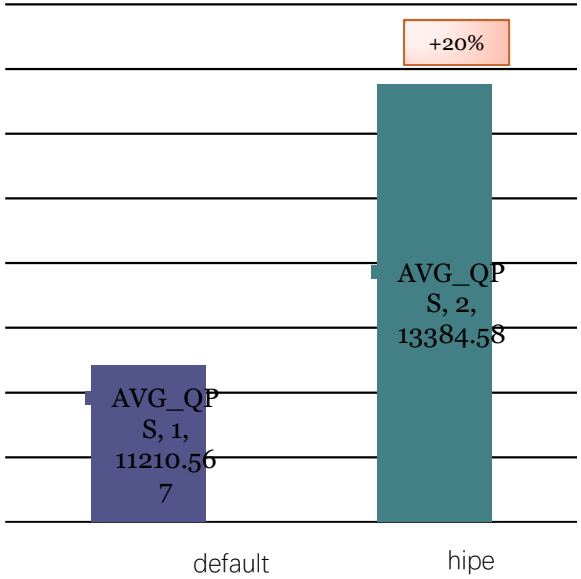

RabbitMQ3.6.0才引入的Lazy Queue

# 性能优化（3）——HiPE

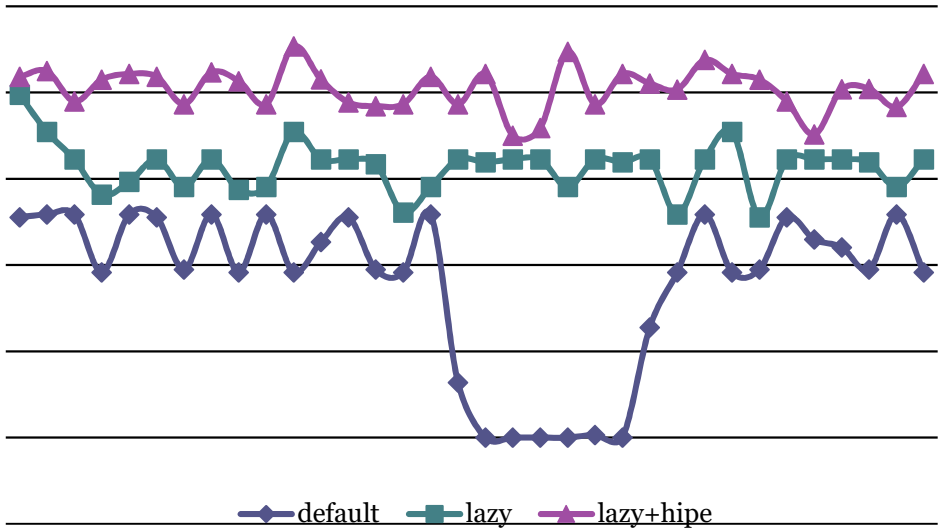
```
[{
    rabbit,[
        {tcp_listeners,[5672]},
        {hipe_compile,true}
    ]

}].
```

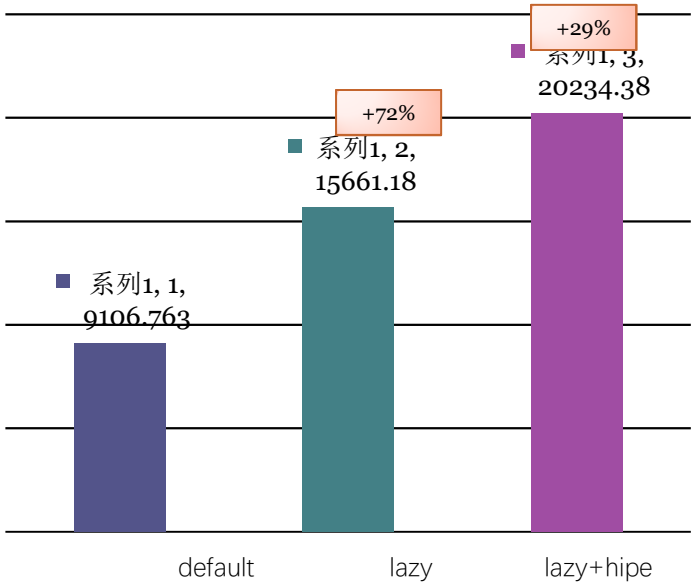
```
=INFO REPORT==== 4-Jan-2018::16:50:15 ===
HiPE in use: compiled 57 modules in 55s.
```



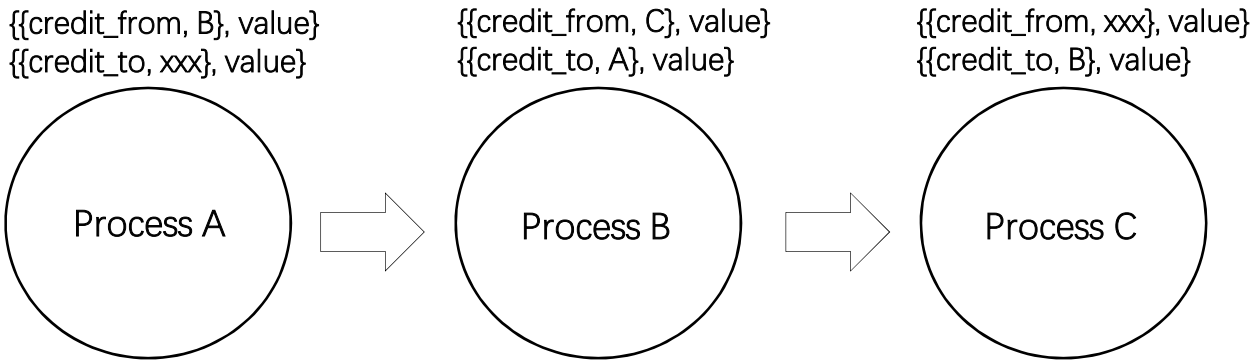
# 性能优化（3）——HiPE



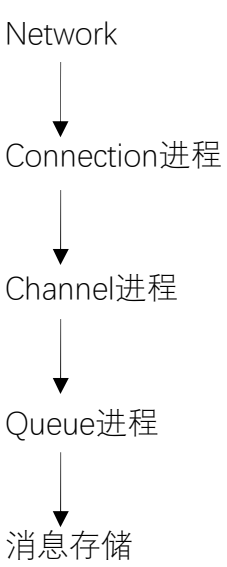



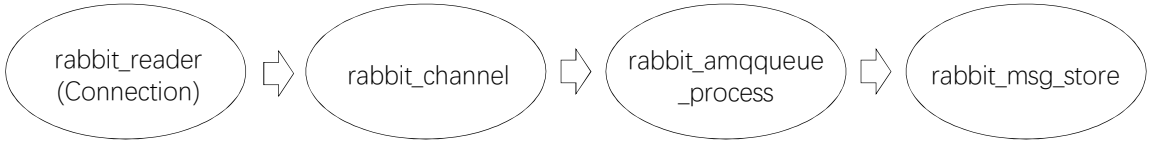

Erlang的版本不能低于18.x

# 性能优化（4）——流控链


{{credit_from, B}, value}
{{credit_to, xxx}, value}
Process A
{{credit_from, C}, value}
{{credit_to, A}, value}
Process B
{{credit_from, xxx}, value}
{{credit_to, B}, value}
Process C

# 性能优化（4）——流控链



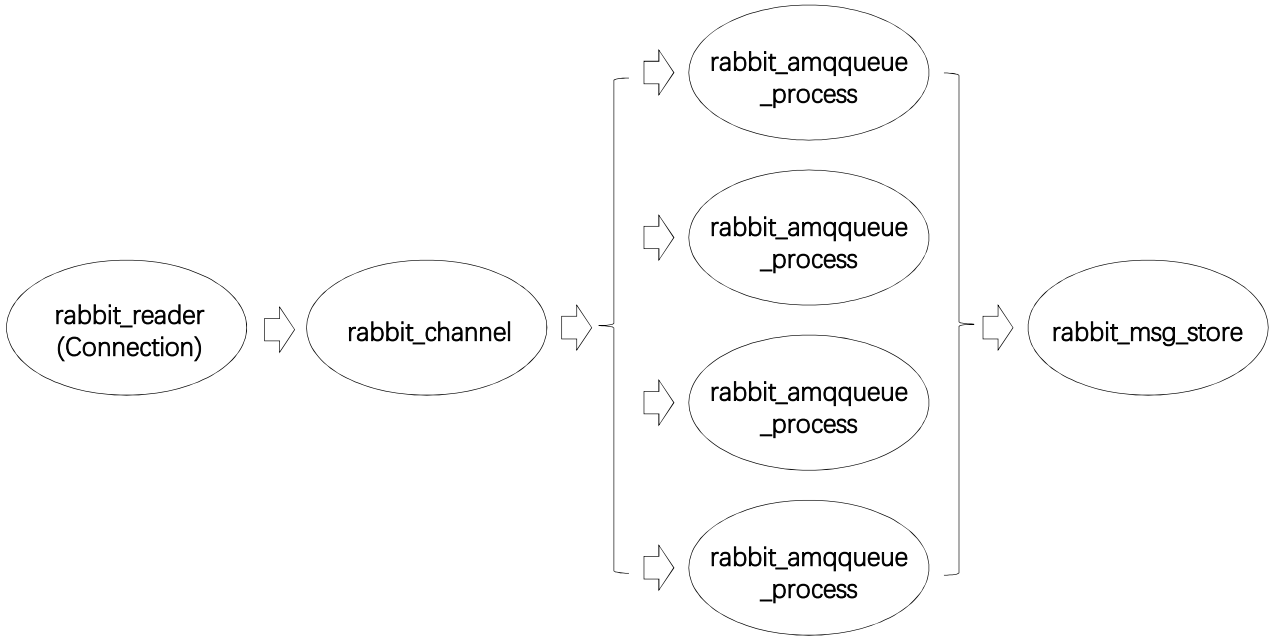

其中的各个进程为：

- rabbit_reader：Connection的处理进程，负责接收、解析AMQP协议数据包等。
- rabbit_channel：Channel的处理进程，负责处理AMQP协议的各种方法、进行路由解析等。
- rabbit_amqqueue_process：队列的处理进程，负责实现队列的所有逻辑。
- rabbit_msg_store：负责实现消息的持久化。

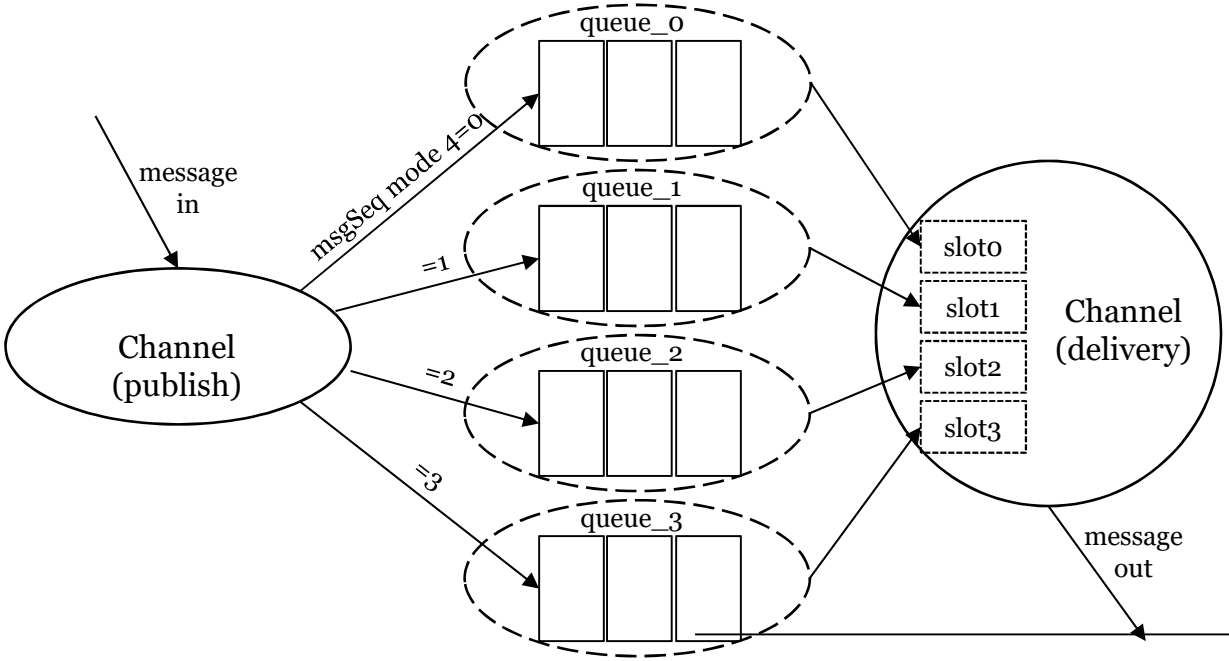
设置的值太大会失去了流控的保护

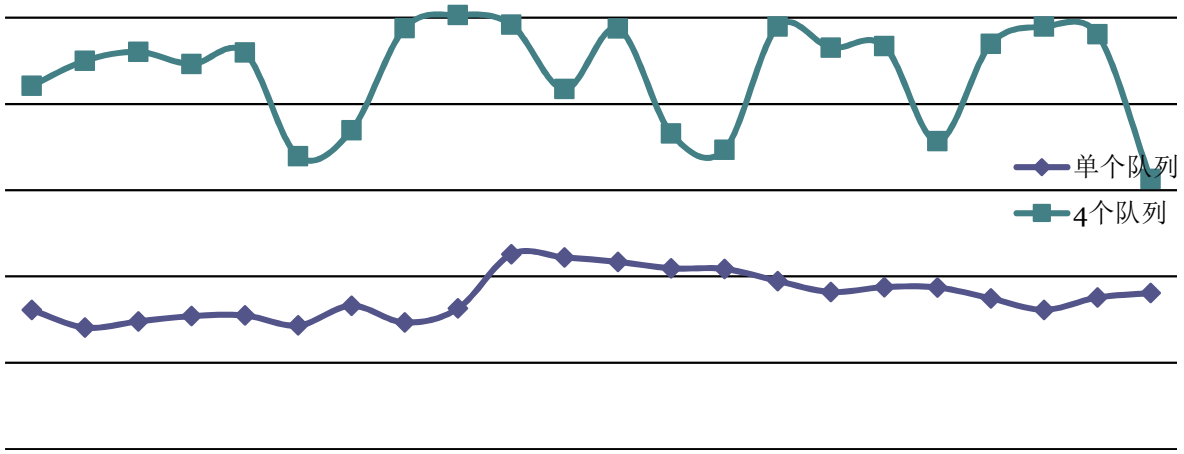
# 性能优化（5）——“曲线救国”

将易造成性能瓶颈的多个rabbitmq_amqqueue_process对外包装成一个队列。类比Kafka中的partitions。



# 客户端——Producer封装



# Basic.Qos的思考

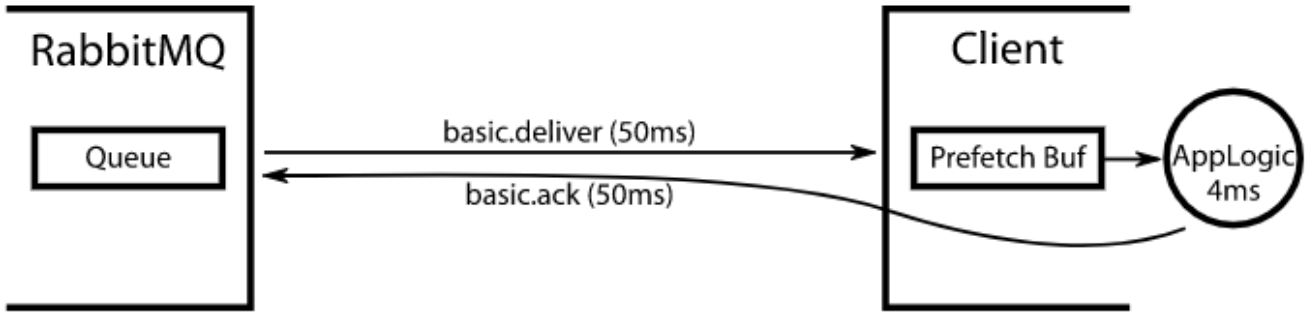
channel.basicQos( int prefetch_count)

## 客户端——Consume

拉模式——Basic.Consume：

```
channel.basicQos(10);
channel.basicConsume(QUEUE_NAME,false,new DefaultConsumer(cha
    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties propertie
                               byte[] body)
```

# Basic.Qos的 思 考




(50ms+50ms+4ms=104ms) vs. (4ms)

消息堆积

# 消息堆积的治理

- 方案1:

  丢弃策略：设置消息保留时间或者保留大小（可以是个数或者占用大小），超过就丢弃。

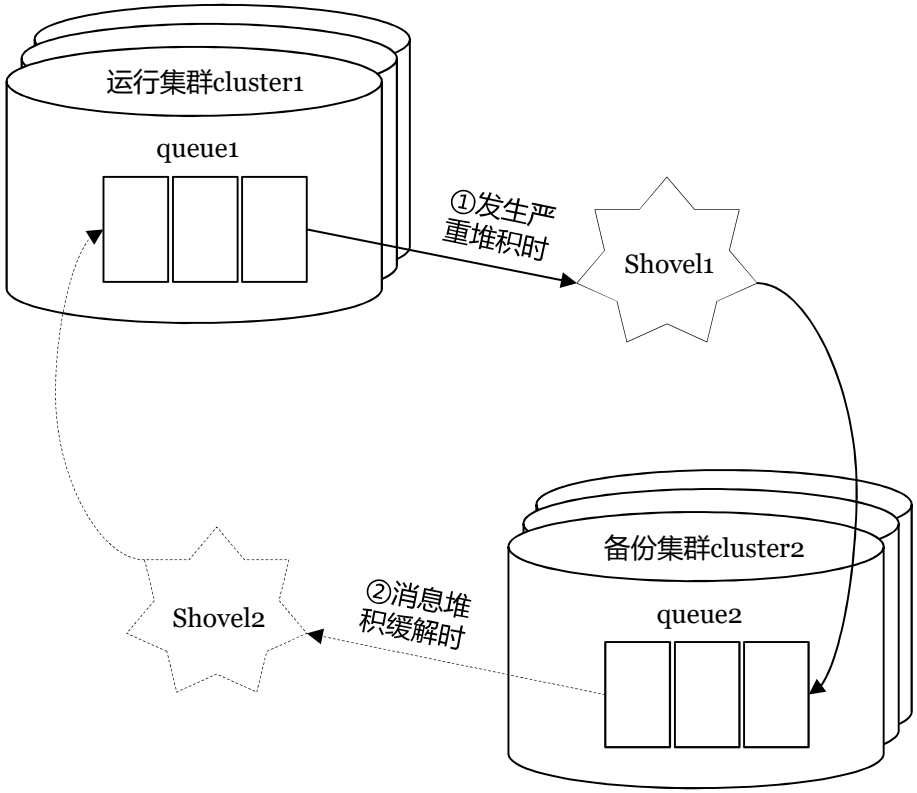
- 方案2:

  Lazy Queue

- 方案3:

  “移花接木”

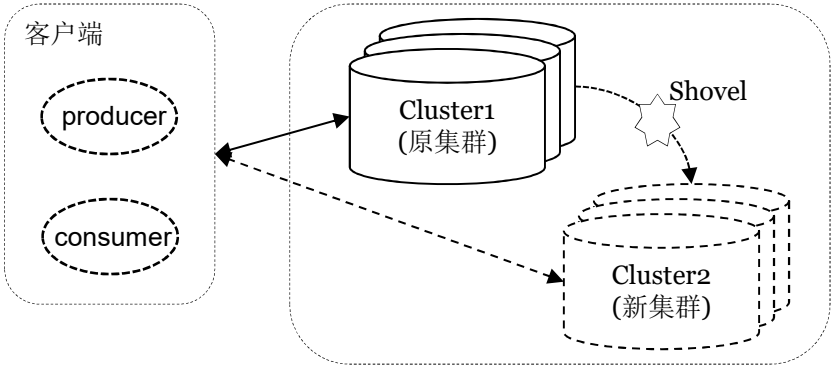
# 消息堆积的治理——“移花接木”




其他用途？

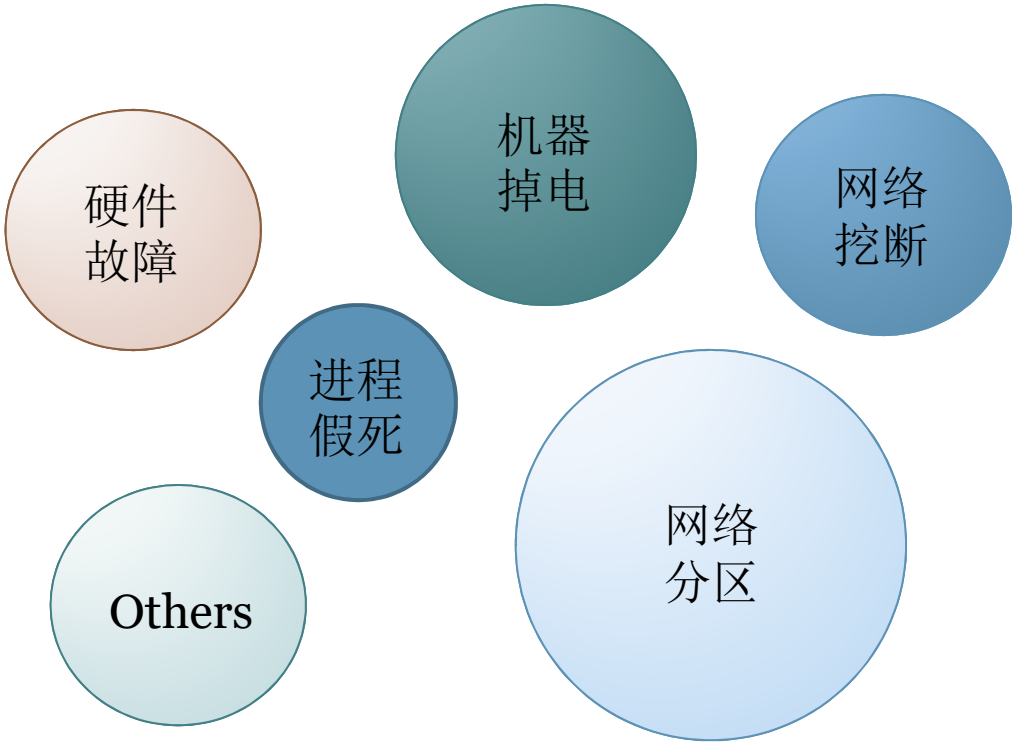
# Shovel应用——集群迁移

对于RabbitMQ运维层面来说，扩容和迁移是比不可少的。



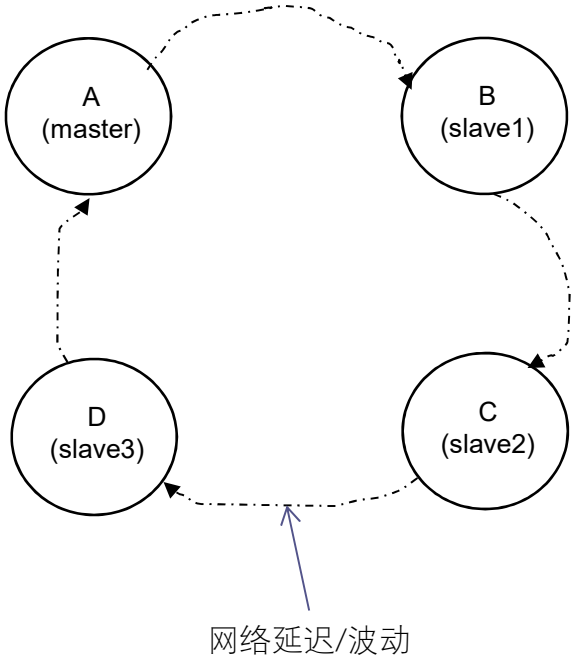

为了消息不丢失，一般先迁移producer的连接，然后再迁移consumer的连接。

# 集群故障



# 网络分区

出现网络分区时，不同分区里的节点会认为不属于自身所在分区的节点都已经挂了（down），对于队列、交换器、绑定的操作仅对当前分区有效。




Network partition detected

Mnesia reports that this RabbitMQ cluster has experienced a network partition. There is a risk of losing data. Please read **RabbitMQ documentation about network partitions and the possible solutions**.

The nature of the partition is as follows:

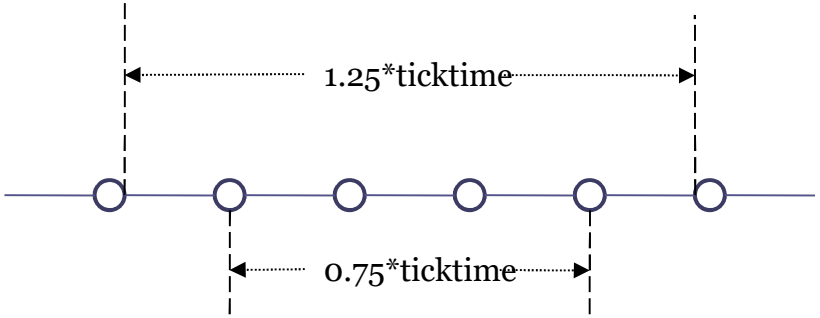
| Node | Was partitioned from |
|---|---|
| rabbit@node1 | rabbit@node2 |
| rabbit@node2 | rabbit@node1<br>rabbit@node3 |
| rabbit@node3 | rabbit@node2 |

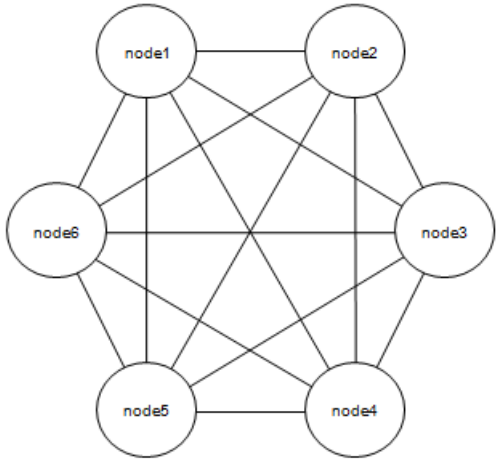
While running in this partitioned state, changes (such as queue or exchange declaration and binding) which take place in one partition will not be visible to other partition(s). Other behaviour is not guaranteed.

如何判定？

# 网络分区——判定

RabbitMQ集群节点内部通信端口默认为25672，两两节点之间都会有信息交互。如果某节点出现网络故障，亦或者是端口不通，则会致使与此节点的交互出现中断，这里就会有个超时判定机制，既而判定网络分区。




将连续4次的tick时间记为T，那么T的取值范围为：0.75*net_ticktime < T < 1.25*net_ticktime。

有何影响？

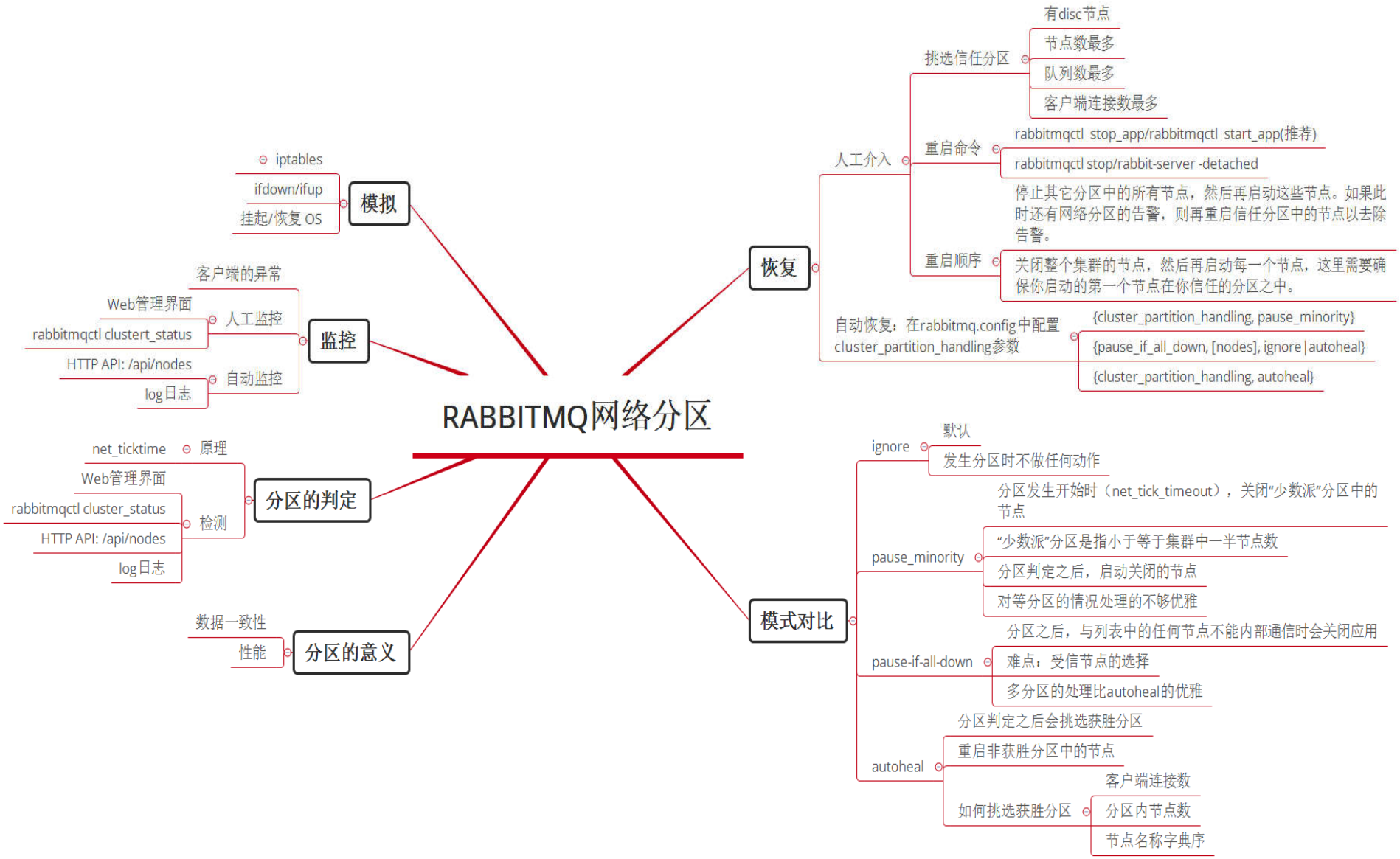
# 网络分区——影响

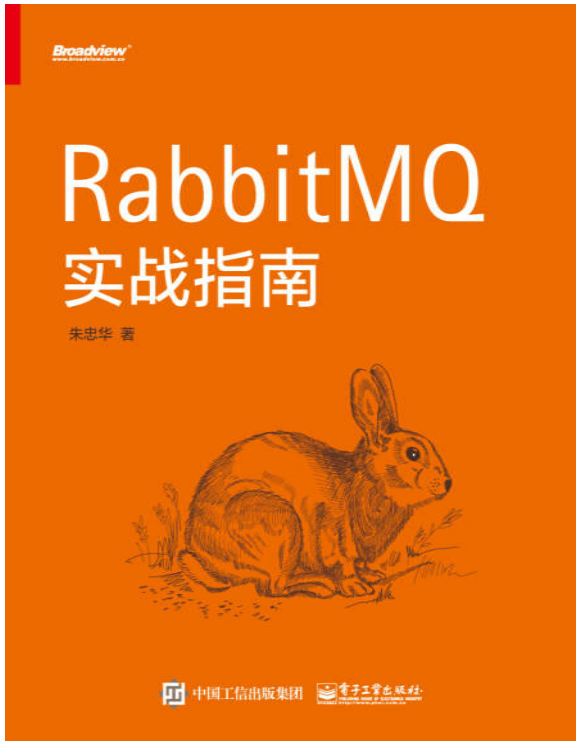
[A,B] => [A],[B]

- client
- broker
- mirror-queue

如何监控？

# 网络分区——总览


RABBITMQ网络分区
模拟
iptables
ifdown/ifup
挂起/恢复 OS
监控
客户端的异常
人工监控
Web管理界面
rabbitmqctl clustert_status
自动监控
HTTP API: /api/nodes
log日志
分区的判定
原理
net_ticktime
检测
Web管理界面
rabbitmqctl cluster_status
HTTP API: /api/nodes
log日志
分区的意义
数据一致性
性能
恢复
人工介入
挑选信任分区
有disc节点
节点数最多
队列数最多
客户端连接数最多
重启命令
rabbitmqctl stop_app/rabbitmqctl start_app(推荐)
rabbitmqctl stop/rabbit-server -detached
停止其它分区中的所有节点，然后再启动这些节点。如果此时还有网络分区的告警，则再重启信任分区中的节点以去除告警。
重启顺序
关闭整个集群的节点，然后再启动每一个节点，这里需要确保你启动的第一个节点在你信任的分区之中。
自动恢复：在rabbitmq.config中配置cluster_partition_handling参数
{cluster_partition_handling, pause_minority}
{pause_if_all_down, [nodes], ignore|autoheal}
{cluster_partition_handling, autoheal}
模式对比
ignore
默认
发生分区时不做任何动作
pause_minority
分区发生开始时（net_tick_timeout），关闭"少数派"分区中的节点
"少数派"分区是指小于等于集群中一半节点数
分区判定之后，启动关闭的节点
对等分区的情况处理的不够优雅
pause-if-all-down
分区之后，与列表中的任何节点不能内部通信时会关闭应用
难点：受信节点的选择
多分区的处理比autoheal的优雅
autoheal
分区判定之后会挑选获胜分区
重启非获胜分区中的节点
如何挑选获胜分区
客户端连接数
分区内节点数
节点名称字典序




# Thank You!