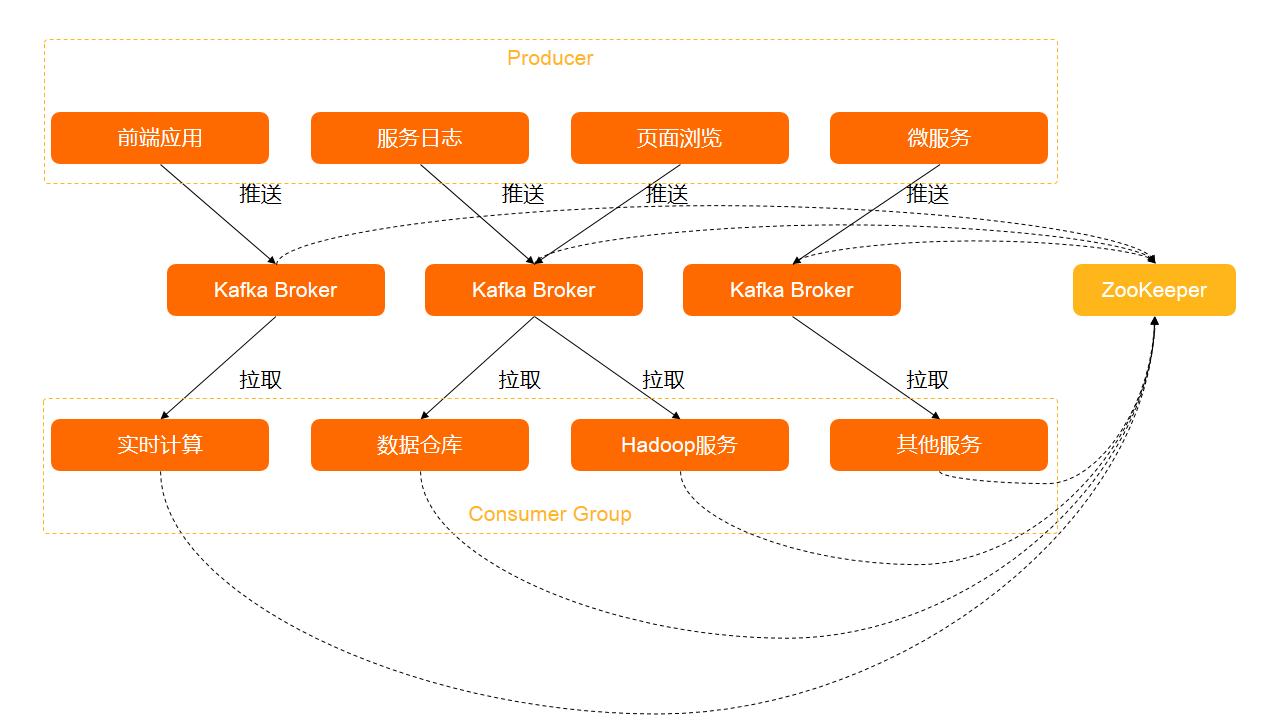
**消息队列Kafka版系统架构**

如[图 1](https://help.aliyun.com/document_detail/68152.html?spm=a2c4g.11186623.6.546.274e49bf3JIXcs" \l "fig-h4w-luj-jjq" \o ")所示，一个消息队列Kafka版集群包括Producer、Kafka Broker、Consumer Group、ZooKeeper。

图 1. 消息队列Kafka版的系统架构[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/2334843951/p129320.png)

***Producer***

通过push模式向消息队列Kafka版的Kafka Broker发送消息。发送的消息可以是网站的页面访问、服务器日志，也可以是CPU和内存相关的系统资源信息。

***Kafka Broker***

用于存储消息的服务器。Kafka Broker支持水平扩展。Kafka Broker节点的数量越多，Kafka集群的吞吐率越高。

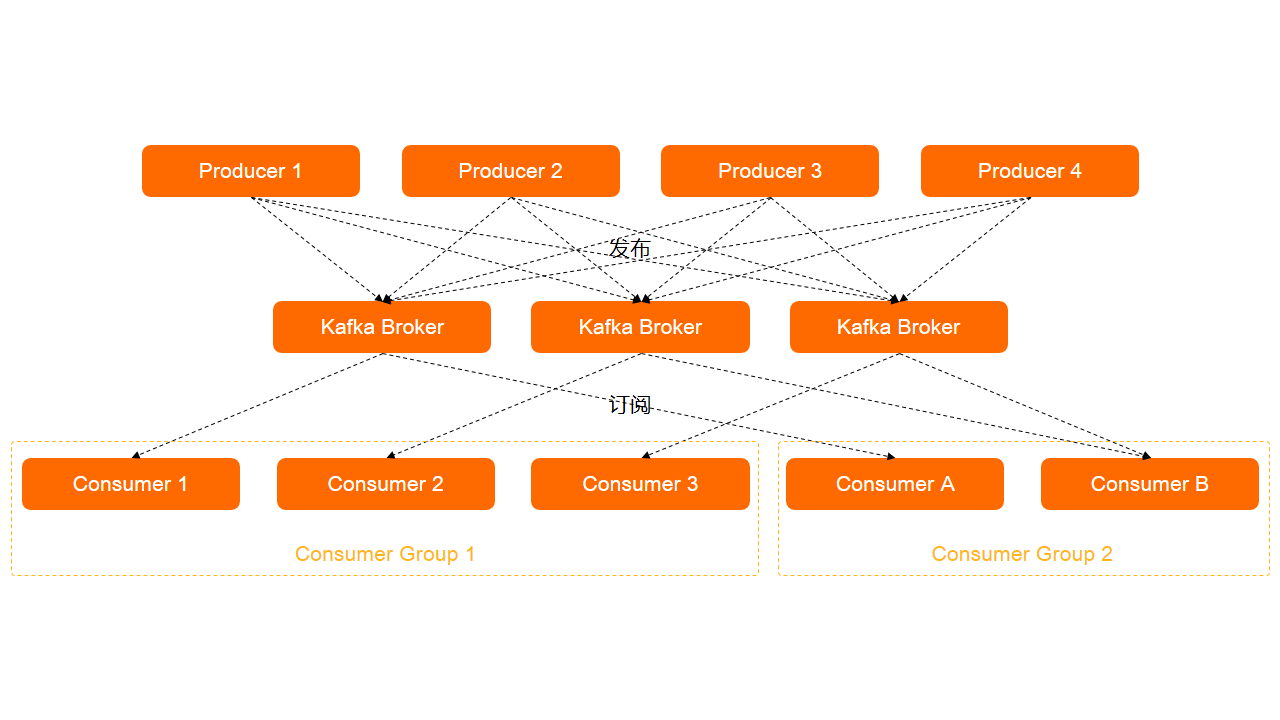
***Consumer Group***

通过pull模式从消息队列Kafka版Broker订阅并消费消息。

***Zookeeper***

管理集群的配置、选举leader分区，并且在Consumer Group发生变化时，进行负载均衡。

**消息队列Kafka版的发布/订阅模型**

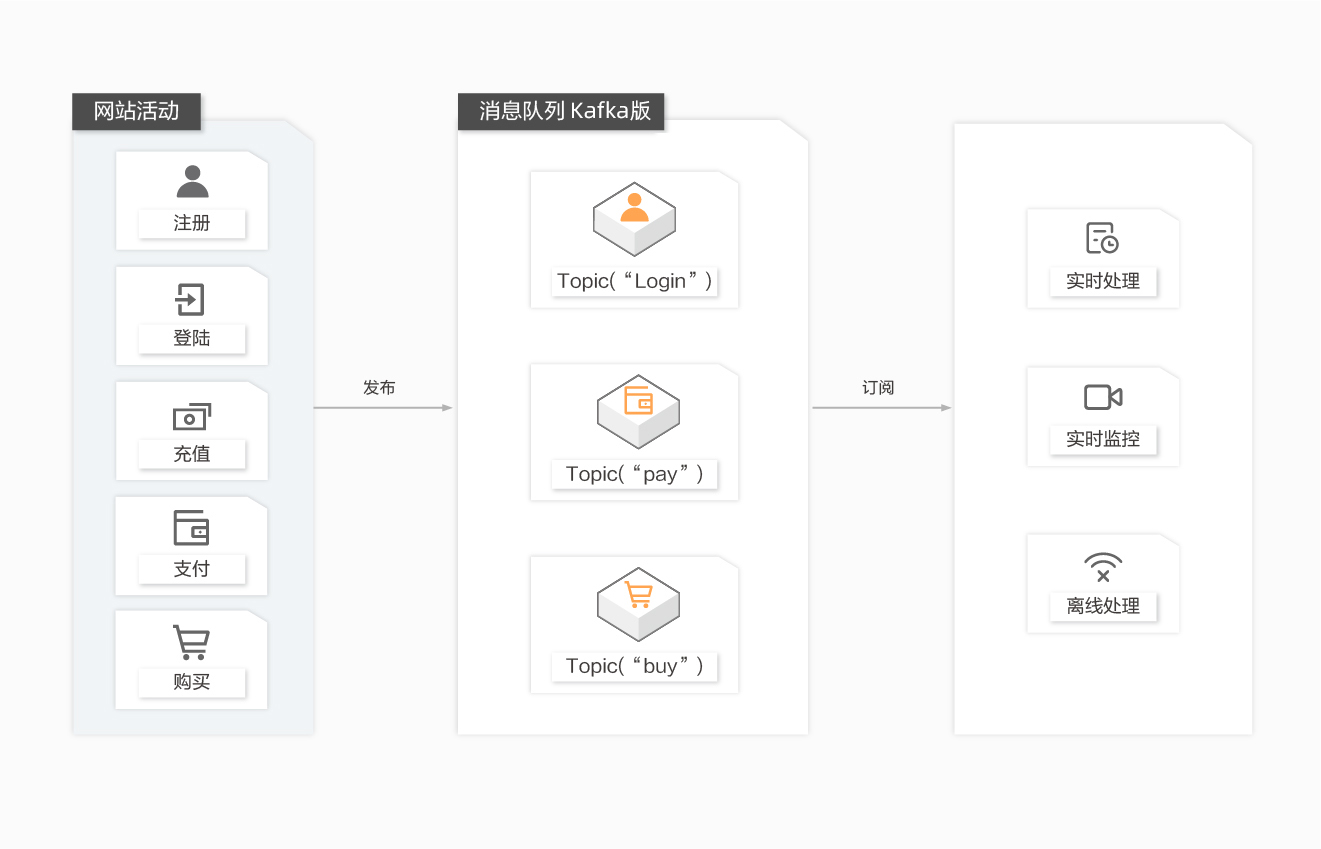
如[图 2](https://help.aliyun.com/document_detail/68152.html?spm=a2c4g.11186623.6.546.274e49bf3JIXcs" \l "fig-clp-z8q-239" \o ")所示，消息队列Kafka版采用发布/订阅模型[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/3334843951/p129319.png)Consumer Group和Topic的对应关系是N : N，即一个Consumer Group可以同时订阅多个Topic，一个Topic也可以被多个Consumer Group同时订阅。

* 虽然一个Topic可以被多个Consumer Group同时订阅，但该Topic的消息只能被同一个Consumer Group内的任意一个Consumer消费。

**本文介绍消息队列Kafka版的典型应用场景，包括网站活动跟踪、日志聚合、流计算处理、数据中转枢纽。**

网站活动跟踪

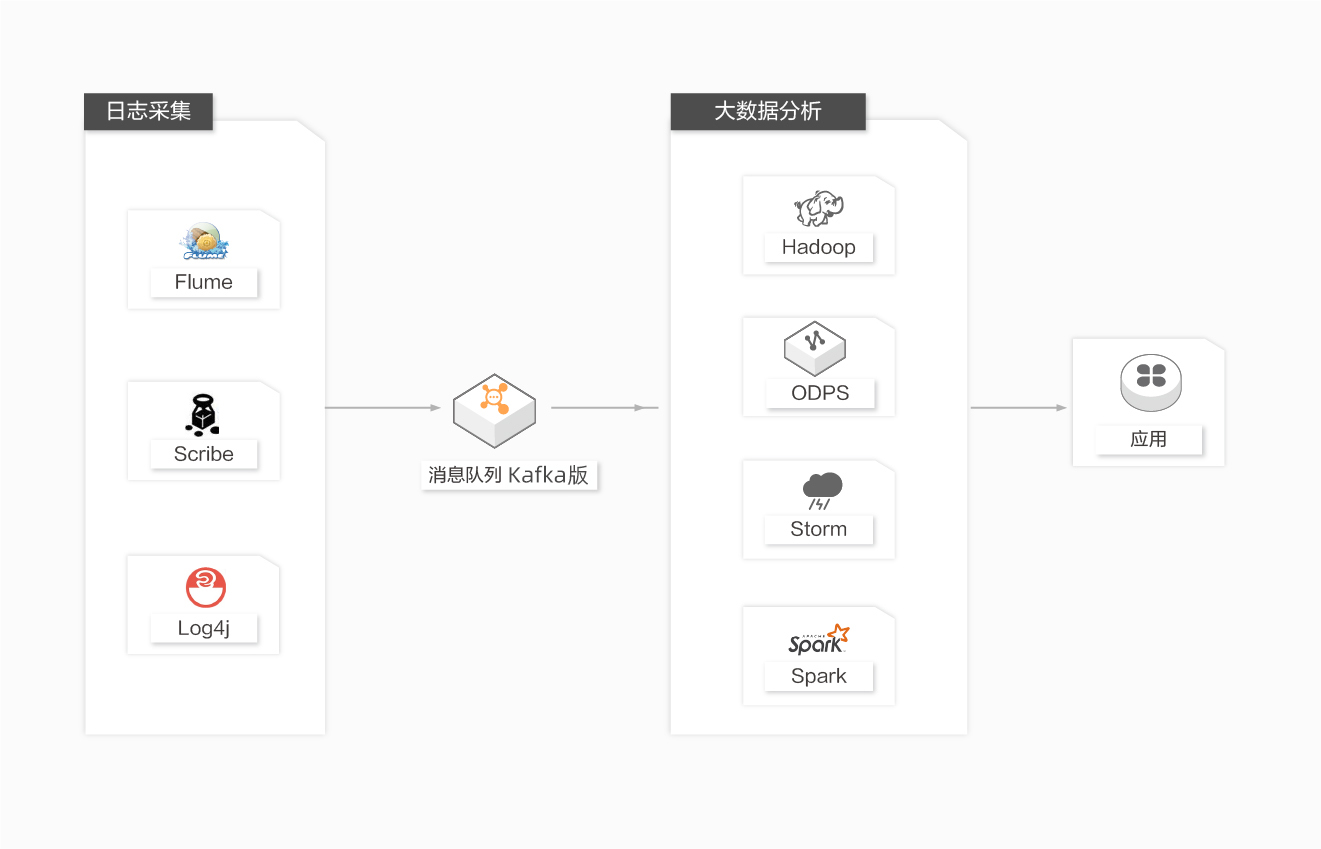
成功的网站运营需要对站点的用户行为进行分析。通过消息队列Kafka版的发布/订阅模型，您可以实时收集网站活动数据（例如注册、登录、充值、支付、购买），根据业务数据类型将消息发布到不同的Topic，然后利用订阅消息的实时投递，将消息流用于实时处理、实时监控或者加载到Hadoop、MaxCompute等离线数据仓库系统进行离线处理。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/3242611951/p98246.jpg)

消息队列Kafka版用于网站活动跟踪具备以下优势：

* 高吞吐：网站用户产生的行为信息较为庞大，需要较高的吞吐量来支持。
* 弹性扩容：网站活动导致行为数据激增，云平台可以快速按需扩容。
* 大数据分析：可对接Storm、Spark等实时流计算引擎，亦可对接Hadoop等离线数据仓库系统。
* 日志聚合

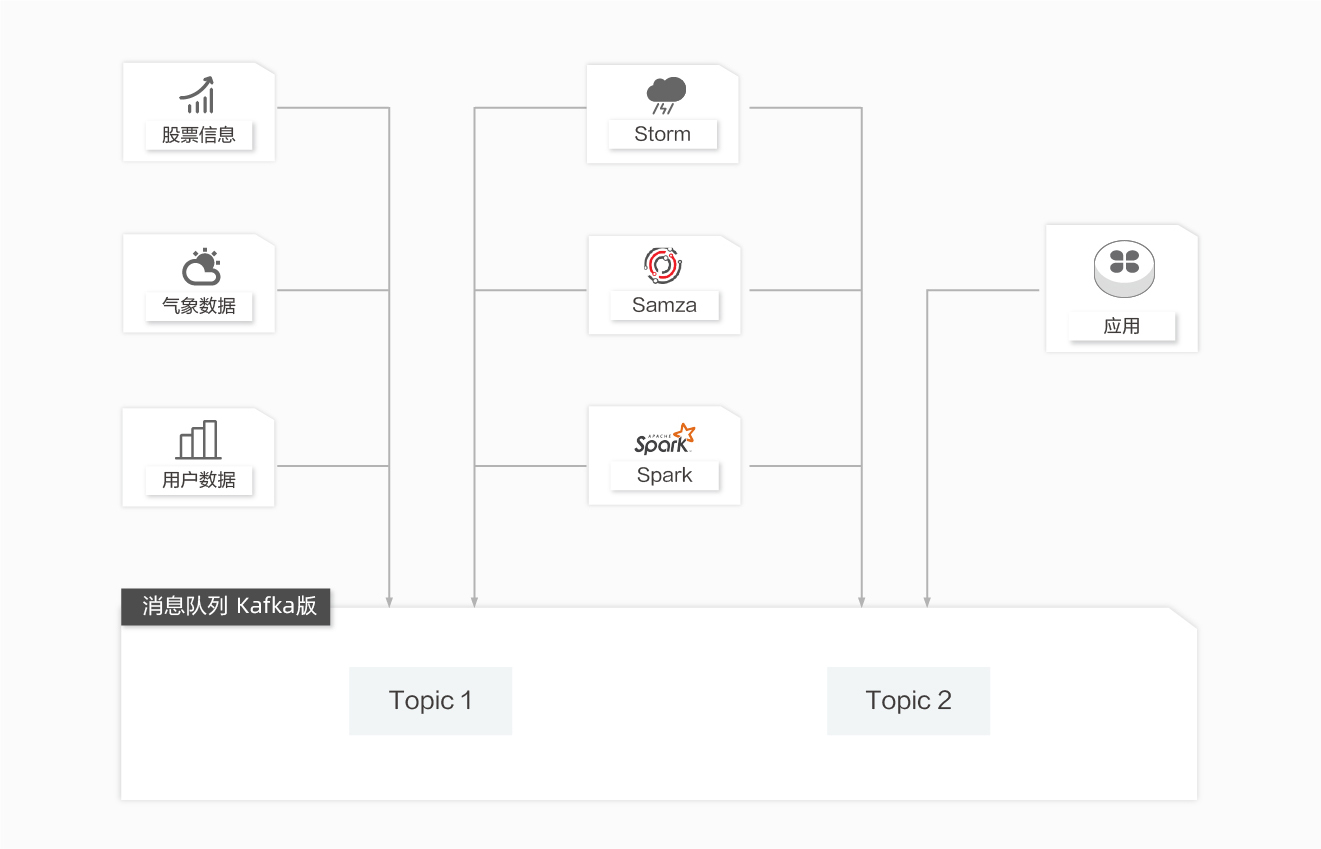
许多公司，例如淘宝、天猫等，每天都会产生大量的日志（一般为流式数据，例如搜索引擎PV、查询等）。相较于以日志为中心的系统，例如Scribe和Flume，消息队列Kafka版在具备高性能的同时，可以实现更强的数据持久化以及更短的端到端响应时间。消息队列Kafka版的这种特性决定它适合作为日志收集中心。消息队列Kafka版忽略掉文件的细节，可以将多台主机或应用的日志数据抽象成一个个日志或事件的消息流，异步发送到消息队列Kafka版集群，从而实现非常低的RT。消息队列Kafka版客户端可批量提交消息和压缩消息，对生产者而言几乎感觉不到性能的开支。消费者可以使用Hadoop、ODPS等离线仓库存储和Strom、Spark等实时在线分析系统对日志进行统计分析。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/3242611951/p98252.jpg)

消息队列Kafka版用于数据聚合具备以下优势：

* 应用与分析解耦：构建应用系统和分析系统的桥梁，并将它们之间的关联解耦。
* 高可扩展性：具有高可扩展性，即当数据量增加时可通过增加节点快速水平扩展。
* 在线或离线分析系统：支持实时在线分析系统和类似于Hadoop的离线分析系统。
* 流计算处理

在很多领域，如股市走向分析、气象数据测控、网站用户行为分析，由于数据产生快、实时性强且量大，您很难统一采集这些数据并将其入库存储后再做处理，这便导致传统的数据处理架构不能满足需求。与传统架构不同，消息队列Kafka版以及Storm、Samza、Spark等流计算引擎的出现，就是为了更好地解决这类数据在处理过程中遇到的问题，流计算模型能实现在数据流动的过程中对数据进行实时地捕捉和处理，并根据业务需求进行计算分析，最终把结果保存或者分发给需要的组件。

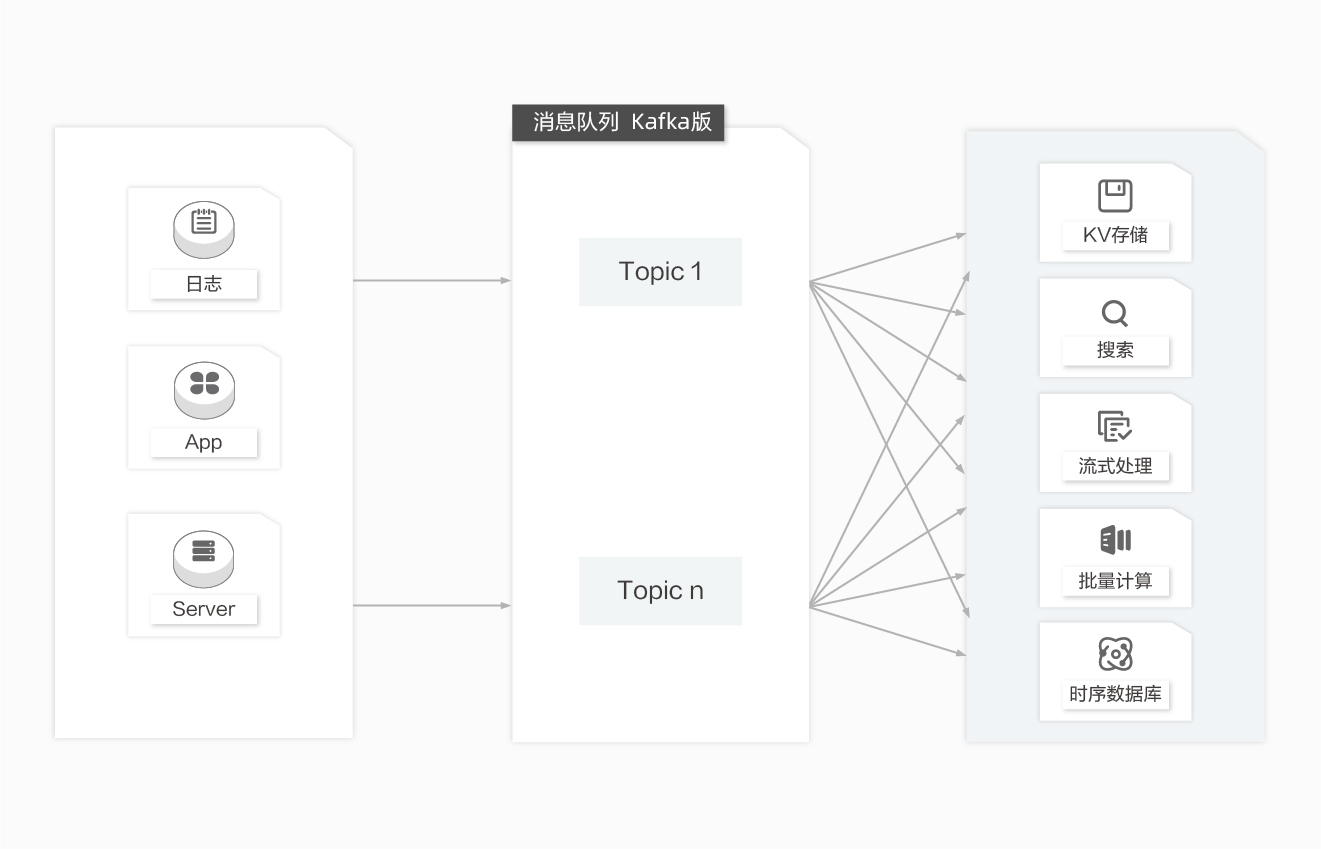
[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/4242611951/p98254.jpg)

消息队列Kafka版用于流计算处理具备以下优势：

* 流动的数据：构建应用系统和分析系统的桥梁，并将它们之间的关联解耦。
* 高可扩展性：由于数据产生的速度快且数据量大，需要高可扩展性。
* 流计算引擎：可对接开源Storm、Samza、Spark以及EMR、Blink、StreamCompute等阿里云产品。

## 数据中转枢纽

近10多年来，诸如KV存储（HBase）、搜索（ElasticSearch）、流式处理（Storm、Spark、Samza）、时序数据库（OpenTSDB）等专用系统应运而生。这些系统是为单一的目标而产生的，因其简单性使得在商业硬件上构建分布式系统变得更加容易且性价比更高。通常，同一份数据集需要被注入到多个专用系统内。例如，当应用日志用于离线日志分析时，搜索单个日志记录同样不可或缺，而构建各自独立的工作流来采集每种类型的数据再导入到各自的专用系统显然不切实际，利用消息队列Kafka版作为数据中转枢纽，同份数据可以被导入到不同专用系统中。

[](https://static-aliyun-doc.oss-cn-hangzhou.aliyuncs.com/assets/img/zh-CN/4242611951/p98253.jpg)

消息队列Kafka版作为数据中转枢纽具备以下优势：

* 高容量存储：能在商业硬件上存储高容量的数据，实现可横向扩展的分布式系统。
* 一对多消费模型：发布/订阅模型，支持同份数据集能同时被消费多次。
* 同时支持实时和批处理：支持本地数据持久化和Page Cache，在无性能损耗的情况下能同时传送消息到实时和批处理的消费者。

# 名词解释

本文主要对消息队列Kafka版涉及的专有名词和术语进行定义和解释，方便您更好地理解相关概念并使用该产品。

***Broker*** 消息队列Kafka版集群中的一个独立服务器。

***Consumer*** 消息订阅者，也称为消息消费者，负责向消息队列Kafka版服务器读取消息并进行消费。

***Consumer Group*** 一类Consumer，这类Consumer通常接收并消费同一类消息，且消费逻辑一致。Consumer Group和Topic的关系是N：N，即同一个Consumer Group可以订阅多个Topic，同一个Topic也可以被多个Consumer Group订阅。

***分区顺序消息*** 默认情况下，保证相同Key的消息分布在同一个分区中，且分区内消息按照发送顺序存储。集群中出现机器宕机时，不会造成消息乱序。但是会出现部分分区发送消息失败，等到宕机机器重新上线后即可恢复正常。

***Local存储*** 存储消息的一种方式，使用原生Apache Kafka的ISR复制算法，采用分布式3副本机制，且min.insync.replicas=2。

***Partition*** 分区，物理上的概念。每个Topic包含一个或多个分区。

***Producer*** 消息发布者，负责生产并发送消息到消息队列Kafka版服务器。

***普通消息*** 默认情况下，保证相同Key的消息分布在同一个分区中，且分区内消息按照发送顺序存储。集群中出现机器宕机时，可能会造成消息乱序。

***Topic*** 消息的类型。消息队列Kafka版通过Topic对消息进行分类。一个Topic由一个或多个Partition组成，存储于一个或多个Broker上。

***云存储*** 存储消息的一种方式，底层接入阿里云云盘的多副本能力。在消息队列Kafka版层面，每个分区只需要1个副本。

# 发布者最佳实践

本文介绍消息队列Kafka版发布者的最佳实践，帮助您减少发送消息出错的可能性。

文中的最佳实践基于消息队列Kafka版的Java客户端。对于其它语言的客户端，其基本概念与思想是相通的，但实现细节可能有差异，仅供参考。

* Key：消息的标识。
* Value：消息内容。

为了便于追踪，请为消息设置一个唯一的Key。您可以通过Key追踪某消息，打印发送日志和消费日志，了解该消息的发送和消费情况。

失败重试

分布式环境下，由于网络等原因偶尔发送失败是常见的。导致这种失败的原因可能是消息已经发送成功，但是Ack失败，也有可能是确实没发送成功。

消息队列Kafka版是VIP网络架构，会主动断开空闲连接（30秒没活动），因此，不是一直活跃的客户端会经常收到 "connection rest by peer" 错误，建议重试消息发送。

**重试参数**

* retries，重试次数，建议设置为3。
* retry.backoff.ms，重试间隔，建议设置为1000。

异步发送 发送接口是异步的，如果您想得到发送的结果，可以调用metadataFuture.get(timeout, TimeUnit.MILLISECONDS)。

线程安全 Producer是线程安全的，且可以往任何Topic发送消息。通常情况下，一个应用对应一个Producer就足够了。

Acks

* acks=0：无需服务端的Response、性能较高、丢数据风险较大。
* acks=1：服务端主节点写成功即返回Response、性能中等、丢数据风险中等、主节点宕机可能导致数据丢失。
* acks=all：服务端主节点写成功且备节点同步成功才返回Response、性能较差、数据较为安全、主节点和备节点都宕机才会导致数据丢失。

一般建议选择acks=1，重要的服务可以设置acks=all。

Batch 把消息缓存在内存中，并进行打包发送。Kafka通过Batch来提高吞吐，但同时也会增加延迟，生产时应该对两者予以权衡。 在构建Producer时，需要考虑以下两个参数：

* batch.size : 发往每个分区（Partition）的消息缓存量（消息内容的字节数之和，不是条数）达到这个数值时，就会触发一次网络请求，然后客户端把消息真正发往服务器；
* linger.ms : 每条消息待在缓存中的最长时间。若超过这个时间，就会忽略batch.size的限制，然后客户端立即把消息发往服务器。

由此可见，Kafka客户端什么时候把消息真正发往服务器，是由上面两个参数共同决定的：batch.size有助于提高吞吐，linger.ms有助于控制延迟。您可以根据具体业务需求进行调整。

OOM

结合Kafka的Batch设计思路，Kafka会缓存消息并打包发送，如果缓存太多，则有可能造成OOM（Out of Memory）。

* buffer.memory : 所有缓存消息的总体大小超过这个数值后，就会触发把消息发往服务器。此时会忽略batch.size和linger.ms的限制。
* buffer.memory的默认数值是32 MB，对于单个Producer来说，可以保证足够的性能。 需要注意的是，如果您在同一个JVM中启动多个Producer，那么每个Producer都有可能占用32 MB缓存空间，此时便有可能触发OOM。
* 在生产时，一般没有必要启动多个Producer；如果特殊情况需要，则需要考虑buffer.memory的大小，避免触发OOM。
* 分区顺序

单个分区（Partition）内，消息是按照发送顺序储存的，是基本有序的。

默认情况下，消息队列Kafka版为了提升可用性，并不保证单个分区内绝对有序，在升级或者宕机时，会发生少量消息乱序（某个分区挂掉后把消息Failover到其它分区）。

对于包年包月计费模式下的专业版实例，如果业务要求分区保证严格有序，请在创建Topic时指定保序。

# 订阅者最佳实践

本文主要介绍消息队列Kafka版订阅者的最佳实践，帮助您减少消费消息出错的可能性。消费消息基本流程

消息队列Kafka版订阅者在订阅消息时的基本流程是：

1. Poll数据
2. 执行消费逻辑
3. 再次poll数据

负载均衡

每个Consumer Group可以包含多个消费实例，即可以启动多个消息队列Kafka版 Consumer，并把参数group.id设置成相同的值。属于同一个Consumer Group的消费实例会负载消费订阅的Topic。

举例：Consumer Group A订阅了Topic A，并开启三个消费实例C1、C2、C3，则发送到Topic A的每条消息最终只会传给C1、C2、C3的某一个。Kafka默认会均匀地把消息传给各个消息实例，以做到消费负载均衡。

Kafka负载消费的内部原理是，把订阅的Topic的分区，平均分配给各个消费实例。因此，消费实例的个数不要大于分区的数量，否则会有实例分配不到任何分区而处于空跑状态。这个负载均衡发生的时间，除了第一次启动上线之外，后续消费实例发生重启、增加、减少等变更时，都会触发一次负载均衡。

消息队列Kafka版的每个Topic的分区数量默认是16个，已经足够满足大部分场景的需求，且云上服务会根据容量调整分区数。

多个订阅

一个Consumer Group可以订阅多个Topic。 一个Topic也可以被多个Consumer Group订阅，且各个Consumer Group独立消费Topic下的所有消息。

举例：Consumer Group A订阅了Topic A，Consumer Group B也订阅了Topic A，则发送到Topic A的每条消息，不仅会传一份给Consumer Group A的消费实例，也会传一份给Consumer Group B的消费实例，且这两个过程相互独立，相互没有任何影响。

消费位点

每个Topic会有多个分区，每个分区会统计当前消息的总条数，这个称为最大位点MaxOffset。

消息队列Kafka版 Consumer会按顺序依次消费分区内的每条消息，记录已经消费了的消息条数，称为ConsumerOffset。

剩余的未消费的条数（也称为消息堆积量） = MaxOffset – ConsumerOffset

消费位点提交

消息队列Kafka版消费者有两个相关参数：

* enable.auto.commit：默认值为true。
* auto.commit.interval.ms： 默认值为1000，也即1s。

这两个参数组合的结果就是，每次poll数据前会先检查上次提交位点的时间，如果距离当前时间已经超过参数auto.commit.interval.ms规定的时长，则客户端会启动位点提交动作。

因此，如果将enable.auto.commit设置为true，则需要在每次poll数据时，确保前一次poll出来的数据已经消费完毕，否则可能导致位点跳跃。

如果想自己控制位点提交，请把enable.auto.commit设为false，并调用commit(offsets)函数自行控制位点提交。

消费位点重置

以下两种情况，会发生消费位点重置：

* 当服务端不存在曾经提交过的位点时（例如客户端第一次上线）。
* 当从非法位点拉取消息时（例如某个分区最大位点是10，但客户端却从11开始拉取消息）。

Java客户端可以通过auto.offset.reset来配置重置策略，主要有三种策略：

* "latest"，从最大位点开始消费。
* "earliest"，从最小位点开始消费。
* "none"，不做任何操作，也即不重置。

**说明**

* 建议设置成 “latest”，而不要设置成 “earliest”，避免因位点非法时从头开始消费，从而造成大量重复。
* 如果是您自己管理位点，可以设置成 "none"。

拉取大消息

消费过程是由客户端主动去服务端拉取消息的，在拉取大消息时，需要注意控制拉取速度，注意修改配置：

* "max.poll.records"，如果单条消息超过1 MB，建议设置为1。
* "fetch.max.bytes"，设置比单条消息的大小略大一点。
* "max.partition.fetch.bytes"，设置比单条消息的大小略大一点。

拉取大消息的核心是一条一条拉的。

拉取公网

通过公网消费消息时，通常会因为公网带宽的限制导致连接被干掉，此时需要注意控制拉取速度，修改配置：

1. "fetch.max.bytes", 建议设置成公网带宽的一半（注意这里的单位是bytes，公网带宽的单位是bits）
2. "max.partition.fetch.bytes", 建议设置成fetch.max.bytes的三分之一或者四分之一。

消息重复和消费幂等

消息队列Kafka版消费的语义是 “at least once”， 也就是至少投递一次，保证消息不丢，但是不会保证消息不重复。在出现网络问题、客户端重启时均有可能出现少量重复消息，此时应用消费端如果对消息重复比较敏感（例如说订单交易类），则应该做到消息幂等。

以数据库类应用为例，常用做法是：

* 发送消息时，传入key作为唯一流水号ID；
* 消费消息时，判断key是否已经消费过，如果已经消费过了，则忽略，如果没消费过，则消费一次；

当然，如果应用本身对少量消息重复不敏感，则不需要做此类幂等检查。

消费失败

消息队列Kafka版是按分区一条一条消息顺序向前推进消费的，如果消费端拿到某条消息后执行消费逻辑失败，例如应用服务器出现了脏数据，导致某条消息处理失败，等待人工干预，那么有以下两种处理方式：

* 失败后一直尝试再次执行消费逻辑。这种方式有可能造成消费线程阻塞在当前消息，无法向前推进，造成消息堆积；
* 由于消息队列Kafka版没有处理失败消息的设计，实践中通常会打印失败的消息、或者存储到某个服务（例如创建一个Topic专门用来放失败的消息），然后定时check失败消息的情况，分析失败原因，根据情况处理。
* 消费延迟

消息队列Kafka版的消费机制是由客户端主动去服务端拉取消息进行消费的。因此，一般来说，如果客户端能够及时消费，则不会产生较大延迟。如果产生了较大延迟，请先关注是否有堆积，并注意提高消费速度。

消费阻塞以及堆积

消费端最常见的问题就是消费堆积，最常造成堆积的原因是：

* 消费速度跟不上生产速度，此时应该提高消费速度，详情请参见[提高消费速度](https://help.aliyun.com/document_detail/68166.html?spm=a2c4g.11186623.6.709.397a181ej7EIzc#section-sod-7b4-znp)；
* 消费端产生了阻塞。

消费端拿到消息后，执行消费逻辑，通常会执行一些远程调用，如果这个时候同步等待结果，则有可能造成一直等待，消费进程无法向前推进。

消费端应该竭力避免堵塞消费线程，如果存在等待调用结果的情况，建议设置等待的超时时间，超时后作消费失败处理。

提高消费速度

提高消费速度有以下两个办法：

* 增加Consumer实例个数

可以在进程内直接增加（需要保证每个实例对应一个线程，否则没有太大意义），也可以部署多个消费实例进程；需要注意的是，实例个数超过分区数量后就不再能提高速度，将会有消费实例不工作。

* 增加消费线程。

增加Consumer实例本质上也是增加线程的方式来提升速度，因此更加重要的性能提升方式是增加消费线程，最基本的步骤如下：

* 1. 定义一个线程池。
  2. Poll数据。
  3. 把数据提交到线程池进行并发处理。
  4. 等并发结果返回成功后，再次poll数据执行。

消息过滤

消息队列Kafka版自身没有消息过滤的语义。实践中可以采取以下两个办法：

* 如果过滤的种类不多，可以采取多个Topic的方式达到过滤的目的；
* 如果过滤的种类多，则最好在客户端业务层面自行过滤。

实践中请根据业务具体情况进行选择，也可以综合运用上面两种办法。

消息广播

消息队列Kafka版没有消息广播的语义，可以通过创建不同的Consumer Group来模拟实现。

订阅关系

同一个Consumer Group内，各个消费实例订阅的Topic最好保持一致，避免给排查问题带来干扰。