重点内容：

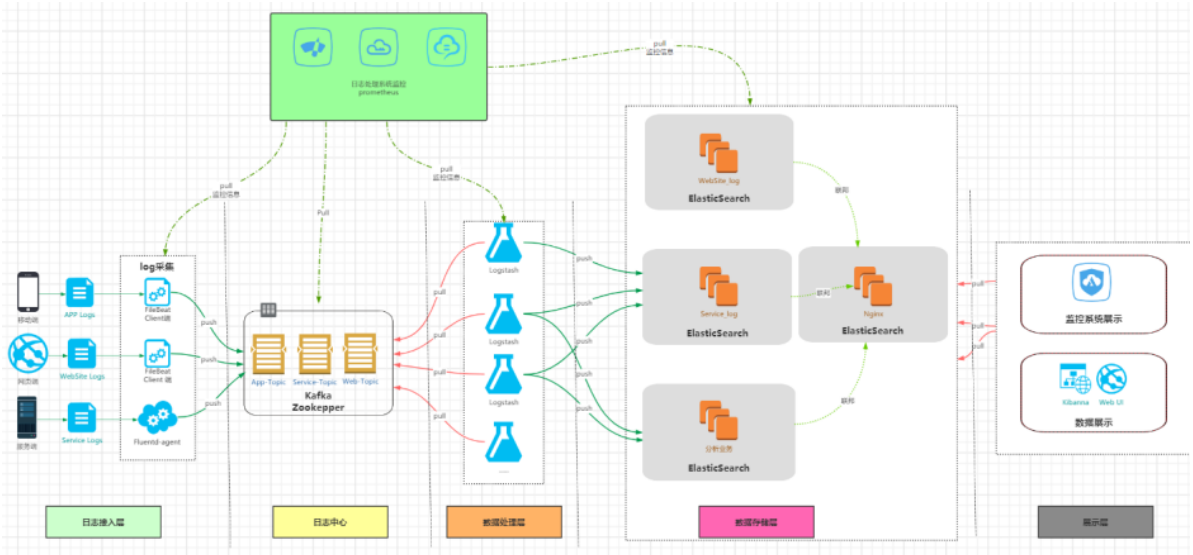
1. kafka解决什么问题
2. kafka框架分析
3. kafka相关术语
4. 生产者如何发布消息
5. 消费者如何消费消息
6. 数据可靠性保证

1 kafka的介绍

Kafka 本质上是一个 MQ（Message Queue），使用消息队列的好处?

- 解耦：允许我们独立的扩展或修改队列两边的处理过程。
- 可恢复性：即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。
- 缓冲：有助于解决生产消息和消费消息的处理速度不一致的情况。
- 灵活性&峰值处理能力：不会因为突发的超负荷的请求而完全崩溃，消息队列能够使关键组件顶住突发的访问压力。
- 异步通信：消息队列允许用户把消息放入队列但不立即处理它。

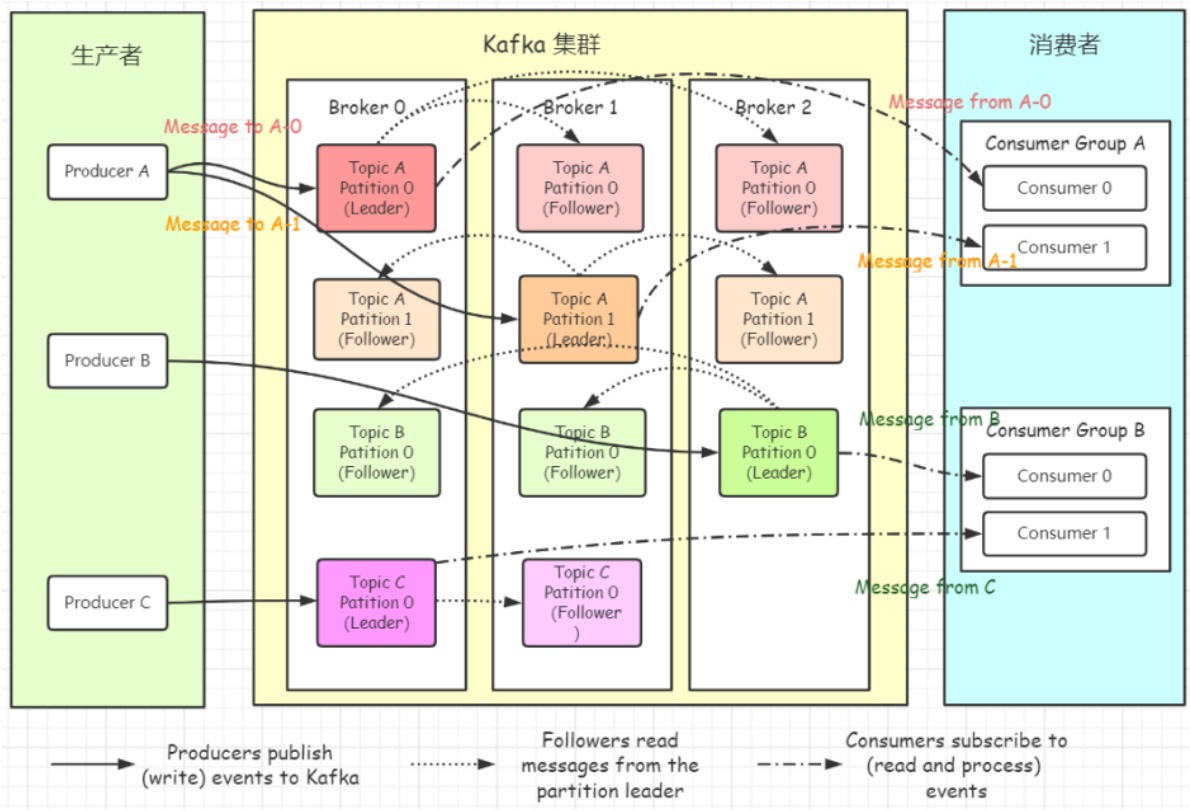
典型应用：原链接https://www.processon.C0m/view/link/60ed48030e3e74074e05f7f9



Logstash是一个开源数据收集引擎，具有实时管道功能。Logstash可以动态地将来自不同数据源的数据统一起来，并将数据标准化到你所选择的目的地

Elast1csearch 是一个分布式、RESTful 风格的搜索和数据分析引擎，能够解决不断涌现出的各种用例。作为 Elast1c Stack 的核心，它集中存储您的数据，帮助您发现意料之中以及意料之外的情况。

2 架构



注意：版面原因这里没有画上zookeeper， broker都是由zookeeper管理。

Kafka 存储的消息来自任意多被称为 Producer 生产者的进程。数据从而可以被发布到不同的 topic 主题下的不同 Partition 分区。

在一个分区内，这些消息被索引并连同时间戳存储在一起。其它被称为 Consumer 消费者的进程可以从分区订阅消息。

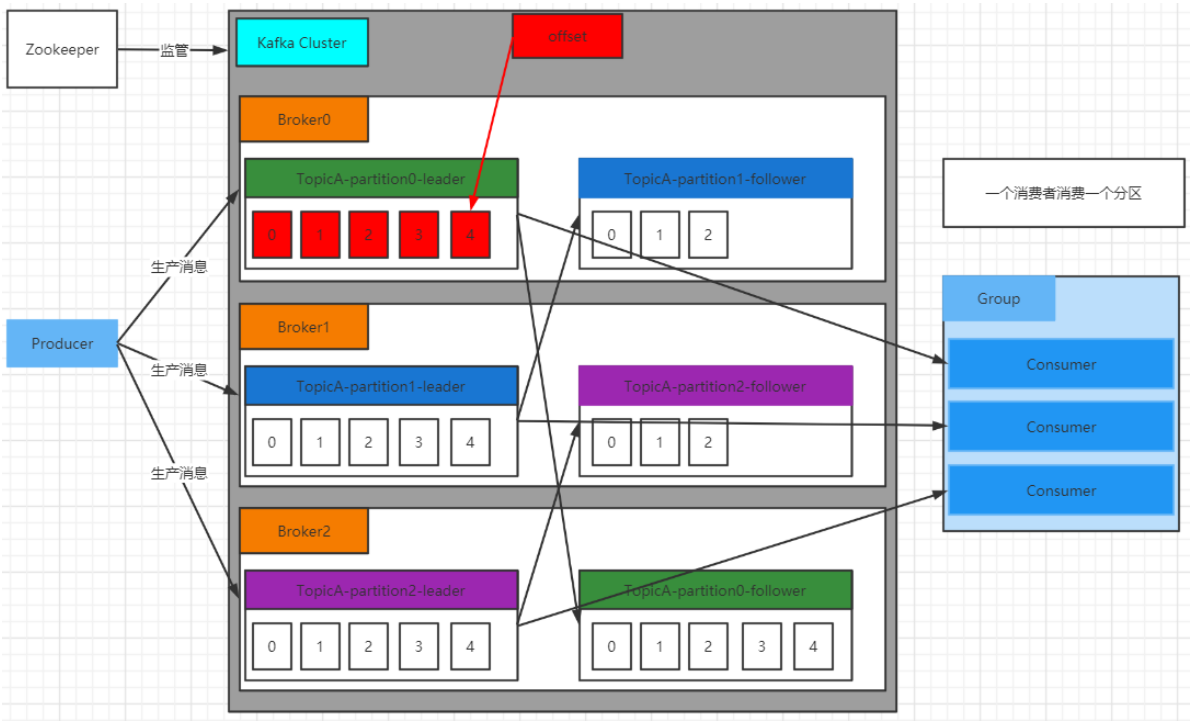
Kafka 运行在一个由一台或多台服务器组成的集群上，并且分区可以跨集群结点分布。

下面给出 Kafka 一些重要概念，让大家对 Kafka 有个整体的认识和感知，后面还会详细的解析每一个概念的作用以及更深入的原理：

- **Producer**：消息生产者，向 Kafka Broker 发消息的客户端。
- **Consumer**：消息消费者，从 Kafka Broker 取消息的客户端。
- **Consumer Group**：**消费者组（CG）**，消费者组内每个消费者负责消费不同分区的数据，提高消费能力。一个分区只能由组内一个消费者消费，消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
- **Broker**：**一台 Kafka 机器就是一个 Broker**。一个集群(**kafka C1uster**)由多个 Broker 组成。一个 Broker 可以容纳多个 topic。
- **topic**：可以理解为一个队列，topic 将消息分类，生产者和消费者面向的是同一个 topic。
- **Partition**：为了**实现扩展性，提高并发能力**，一个非常大的 topic 可以分布到多个 Broker （即服务器）上，一个 topic 可以分为多个 Partition，同一个topic在不同的分区的数据是不重复的，每个 Partition 是一个有序的队列，其表现形式就是一个一个的文件夹。
- **Replication**：每一个分区都有多个副本，副本的作用是做备胎。当主分区（Leader）故障的时候会选择一个备胎（Follower）上位，成为Leader。在kafka中默认副本的最大数量是10个，且副本的数量不能大于Broker的数量，follower和leader绝对是在不同的机器，同一机器对同一个分区也只可能存放一个副本（包括自己）。
- **Message**：消息，每一条发送的消息主体。
- **Leader**：每个分区多个副本的“主”副本，生产者发送数据的对象，以及消费者消费数据的对象，都是 Leader。
- **Follower**：每个分区多个副本的“从”副本，实时从 Leader 中同步数据，保持和 Leader 数据的同步。Leader 发生故障时，某个 Follower 还会成为新的 Leader。

- **Offset：消费者消费的位置信息**，监控数据消费到什么位置，当消费者挂掉再重新恢复的时候，可以从消费位置继续消费。
- **ZooKeeper**：Kafka 集群能够正常工作，需要依赖于 ZooKeeper，ZooKeeper 帮助 Kafka 存储和管理集群信息。

2.1 工作流程



不同的Partition的offset 是独立的。

Kafka 中消息是以 topic 进行分类的，生产者生产消息，消费者消费消息，面向的都是同一个 topic。

topic 是逻辑上的概念，而 Partition 是物理上的概念，**每个 Partition 对应于一个 log 文件**，该 log 文件中存储的就是 Producer 生产的数据。

Producer 生产的数据会不断追加到该 log 文件末端，且每条数据都有自己的 Offset。

消费者组中的每个消费者，都会实时记录自己消费到了哪个 Offset，以便出错恢复时，从上次的位置继续消费。
日志默认在：/tmp/kafka-logs

2.2 副本原理

副本机制（Replication），也可以称之为备份机制，通常是指分布式系统在多台网络互联的机器上保存有相同的数据拷贝。副本机制的好处在于：

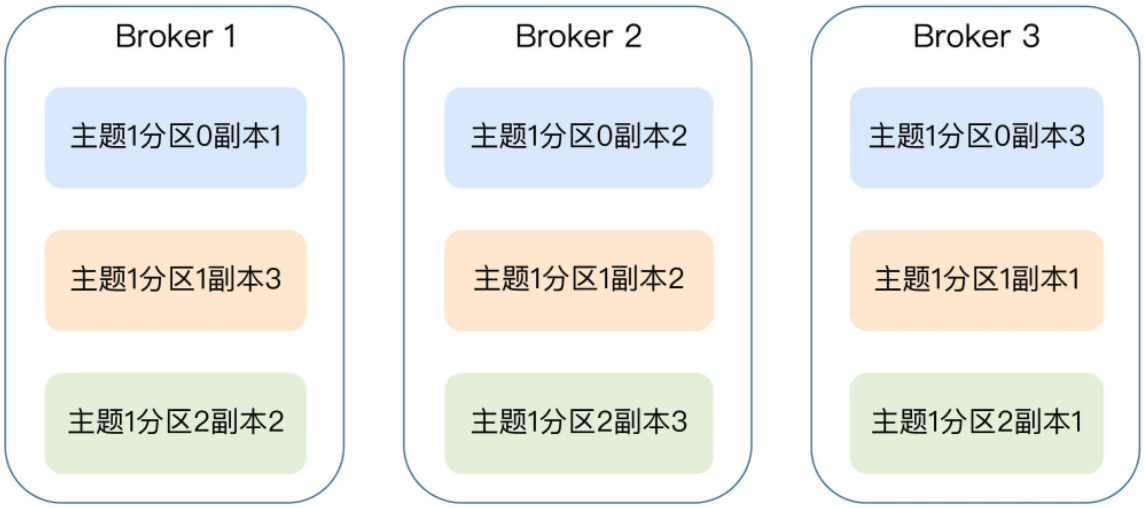
1. **提供数据冗余。**在一部分节点宕机的时候，系统仍能继续工作（即提高可用性）。
2. 提供高伸缩性。支持扩大机器数量，从而可以支撑更高的**读请求量**，比如fastdfs、mongodb。

   kafka是否支持通过副本机制提高读的请求量? -》不支持这样的机制
3. 改善数据局部性。允许将数据放入与用户地理位置相近的地方，**从而降低系统延时。 kafka也不支持。**

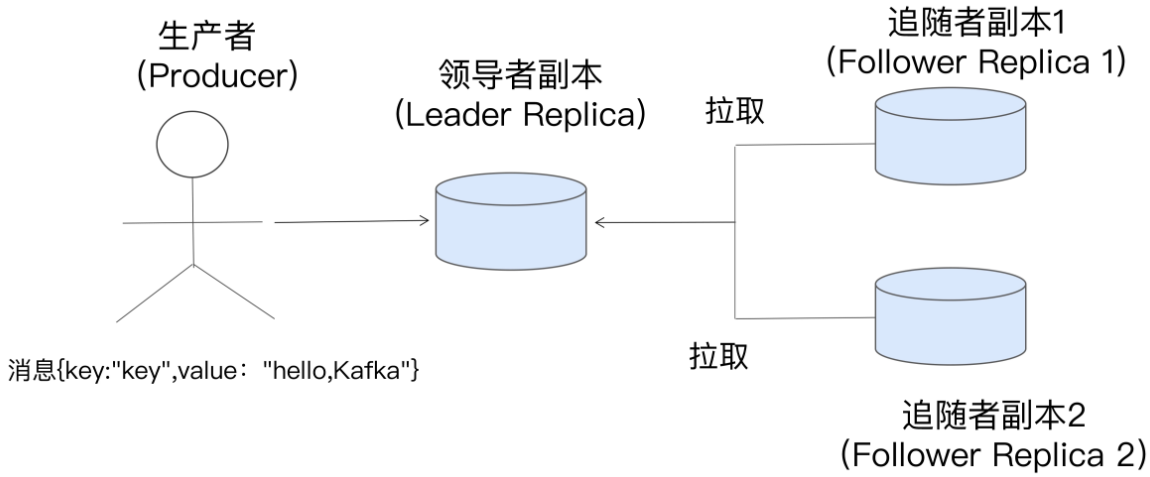
目前Kafka只实现了副本机制带来的第 1 个好处，即是提供数据冗余实现高可用性和高持久性。

在kafka生产环境中，每台 Broker 都可能保存有各个主题下不同分区的不同副本，因此，单个 Broker 上存有成百上于个副本的现象是非常正常的。

下图展示了一个有 3 台 Broker 的 Kafka 集群上的副本分布情况。从图中可以看到，主题 1 分区 0 的 3 个副本分散在 3 台 Broker 上，其他主题分区的副本也都散落在不同的 Broker 上，从而实现数据冗余。



基于领导者（Leader-based）的副本机制



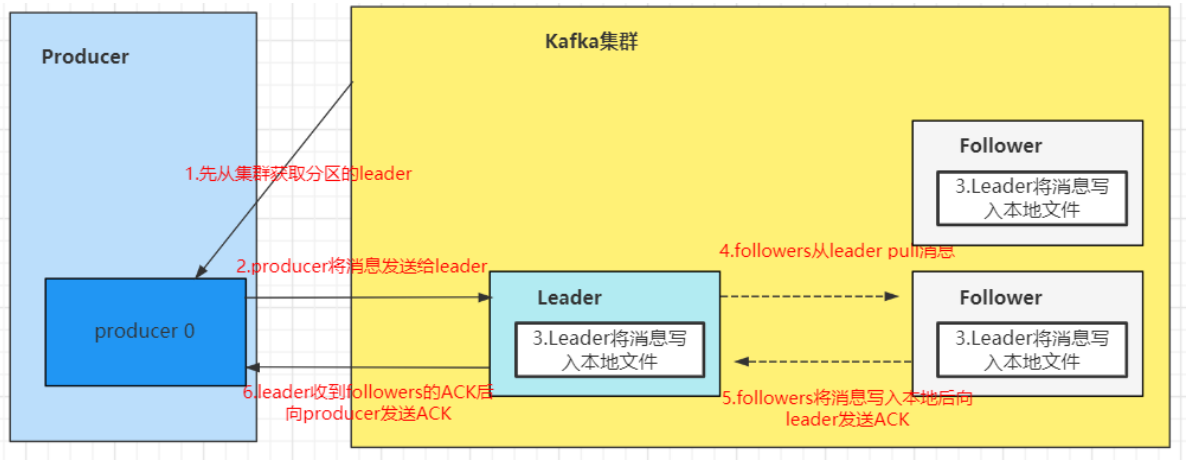
1. 在 Kafka 中，副本分成两类：领导者副本（Leader Replica）和追随者副本（Follower Replica）。每个分区在创建时都要选举一个副本，称为领导者副本，其余的副本自动称为追随者副本。
2. Kafka 副本机制中的追随者副本是不对外提供服务的，不同于Fastdfs、MongdoDB等。
3. 当领导者副本挂掉了，或领导者副本所在的 Broker 宕机时，Kafka 依托于 ZooKeeper 提供的监控功能能够实时感知到，并立即开启新一轮的领导者选举，从追随者副本中选一个作为新的领导者。老 Leader 副本重启回来后，只能作为追随者副本加入到集群中。

2.3 分区和主题的关系

- 一个分区只能属于一个主题
- 一个主题可以有多个分区
- 同一主题的不同分区内容不一样，每个分区有自己独立的**offset**
- 同一主题不同的分区能够被放置到不同节点的broker
- 分区规则设置得当可以使得同一主题的消息**均匀落在**不同的分区

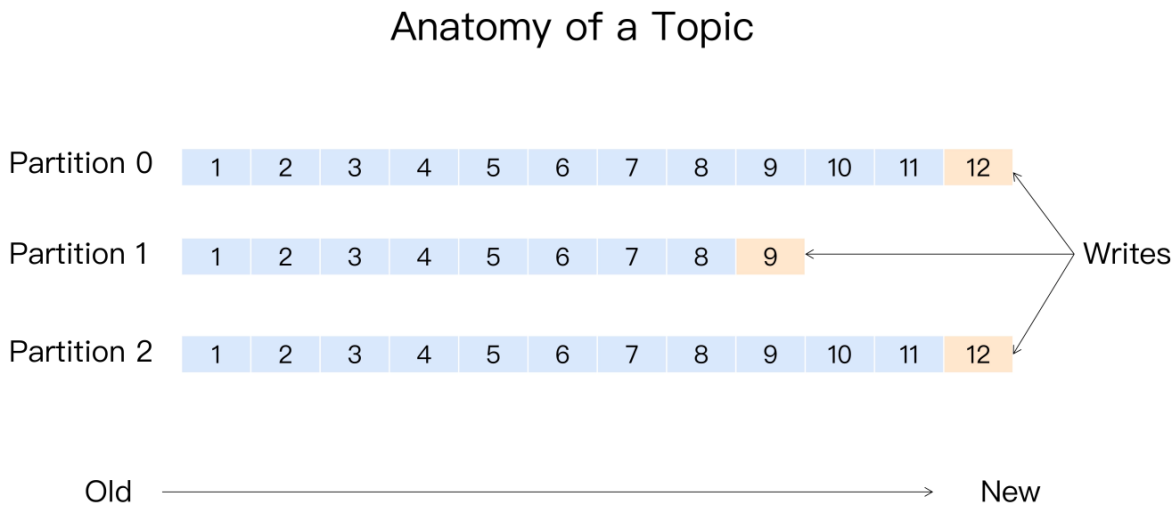
2.4 生产者

producer就是生产者，是数据的入口。Producer在写入数据的时候**永远的找leader**，不会直接将数据写入follower。



2.4.1 为什么分区-可以水平扩展

Kafka 的消息组织方式实际上是三级结构：主题 - 分区 - 消息。主题下的每条消息只会保存在某一个分区中，而不会在多个分区中被保存多份。如下所示：



分区的作用主要**提供负载均衡的能力，能够实现系统的高伸缩性（Scalability）**。不同的分区能够被放置到不同节点的机器上，而数据的读写操作也都是针对分区这个粒度而进行的，这样每个节点的机器都能独立地执行各自分区的读写请求处理。这样，当性能不足的时候可以通过添**加新的节点机器来增加整体系统的吞吐量。**

分区原则：我们需要将 Producer 发送的数据封装成一个 ProducerRecord对象。

该**对象需要指定一些参数**：

- topic：string 类型，NotNull。
- partition：int 类型，可选。
- timestamp：long 类型，可选。
- **key**：string 类型，可选。
- value：string 类型，可选。
- headers：array 类型，Nullable。

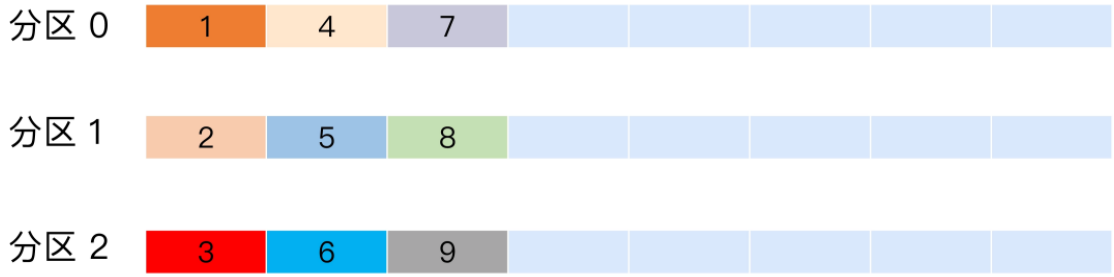
2.4.2 分区策略

所谓分区策略是决定生产者将消息发送到哪个分区的算法。Kafka 为我们提供了默认的分区策略，同时它也支持你自定义分区策略。

- 轮询策略
- 随机策略
- 按消息键
- 默认分区

1. 轮询策略

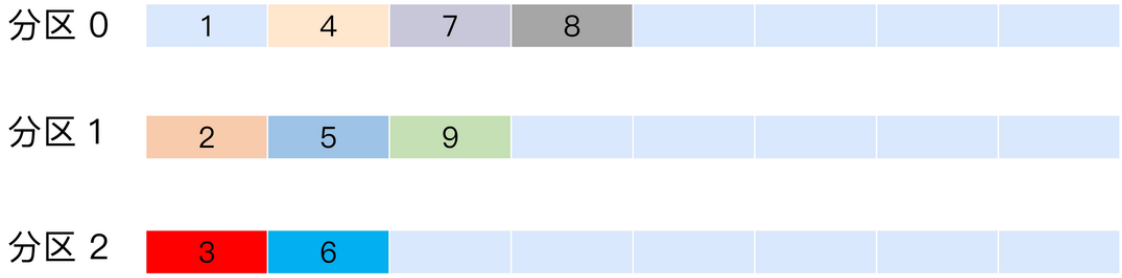
Round-robin 策略，即顺序分配。比如一个主题下有 3 个分区，那么第一条消息被发送到分区 0，第二条被发送到分区 1，第三条被发送到分区 2，以此类推。当生产第 4 条消息时又会重新开始，即将其分配到分区 0



轮询策略有非常优秀的负载均衡表现，它总是能保证消息最大限度地被平均分配到所有分区上，**故默认情况下它是最合理的分区策略**，也是我们最常用的分区策略之一。

2. 随机策略-Randomness 策略

随机就是我们随意地将消息放置到任意一个分区上，如下面这张图所示。



随机策略是老版本生产者使用的分区策略，在新版本中已经改为轮询了。

3. 按消息键保序策略-Key-ordering 策略

Kafka 允许为每条消息定义消息键，简称为 Key。这个 Key 的作用非常大，它可以是一个有着明确业务含义的字符串，比如客户代码、部门编号或是业务 ID 等；也可以用来表征消息元数据。特别是在 Kafka 不支持时间戳的年代，在一些场景中，工程师们都是直接将消息创建时间封装进 Key 里面的。**一旦消息被定义了 Key**，那么你就可以保证**同一个 Key 的所有消息都进入到相同的分区里面**，由于每个分区下的消息处理都是有顺序的，故这个策略被称为按消息键保序策略。

key1 --- 落在同一分区

key2 --- 落在同一个分区

4. 默认分区规则

消息是按照三种策略进入分区:

1. 指明 Partition 的情况下，直接将给定的 Value 作为 Partition 的值。
2. 没有指明 Partition 但有 Key 的情况下，**将 Key 的 Hash 值与分区数取余得到 Partition 值。**
3. 既没有 Partition 有没有 Key 的情况下，第一次调用时随机生成一个整数（后面每次调用都在这个整数上自增），将这个值与可用的分区数取余，得到 Partition 值，也就是常说的 **Round-Robin 轮询算法。**

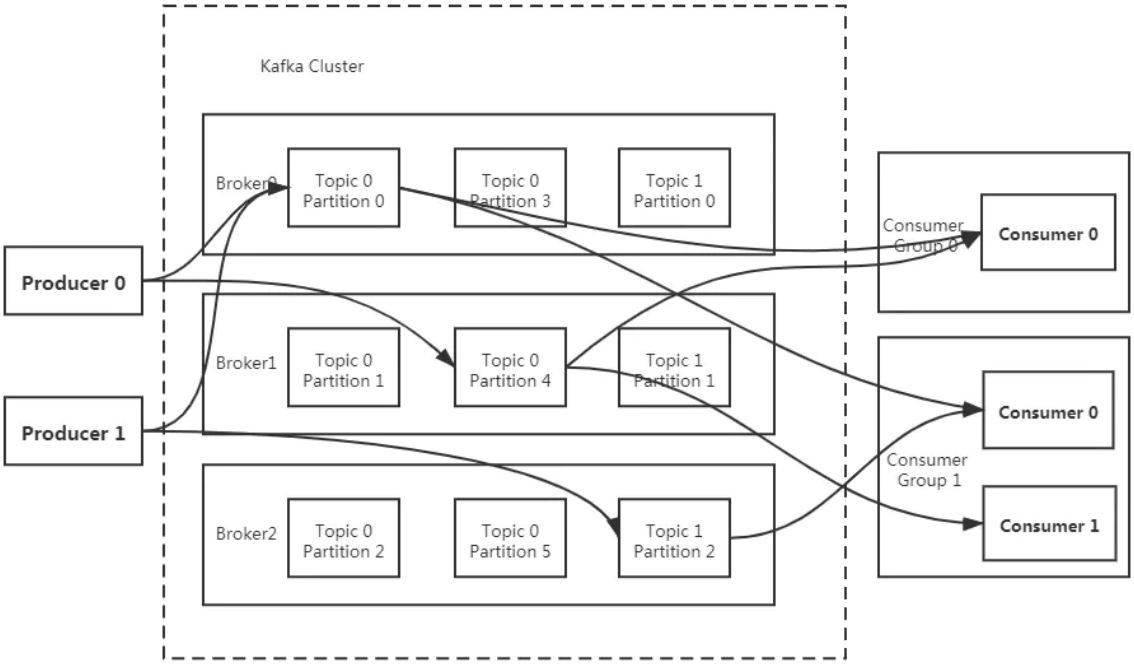
2.5 消费者

传统的消息队列模型的缺陷在于**消息一旦被消费，就会从队列中被删除（zeromq）**，而且只能被下游的一个 Consumer 消费。严格来说，这一点不算是缺陷，只能算是它的一个特性。

但很显然，这种模型的伸缩性（scalability）很差，因为下游的多个 Consumer 都要“抢”这个共享消息队列的消息。发布 / 订阅模型倒是允许消息被多个 Consumer 消费，但它的问题也是伸缩性不高，因为每个订阅者都必须要订阅主题的所有分区。这种全量订阅的方式既不灵活，也会影响消息的真实投递效果。

当 Consumer Group 订阅了多个主题后，组内的每个实例不要求一定要订阅主题的所有分区，它只会消费部分分区中的消息。Consumer Group 之间彼此独立，互不影响，它们能够订阅相同的一组主题而互不干涉。再加上 Broker 端的消息留存机制，Kafka 的 Consumer Group 完美地规避了上面提到的伸缩性差的问题。可以这么说，Kafka 仅仅使用 Consumer Group 这一种机制，却同时实现了传统消息引擎系统的两大模型：

1. 如果所有实例（消费者）都属于同一个 Group，那么它实现的就是点对点消息队列模型；
2. 如果所有实例（消费者）分别属于不同的 **Group，那么它实现的就是发布 / 订阅模型。**



2.5.1 消费方式

Consumer 采用 **Pull（拉取）**模式从 Broker 中读取数据。

Pull 模式则可以根据 Consumer 的消费能力以适当的速率消费消息。Pull 模式不足之处是，如果 Kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。

因为消费者从 Broker 主动拉取数据，需要维护一个长轮询，针对这一点， **Kafka 的消费者在消费数据时会传入一个时长参数 timeout。**

如果当前没有数据可供消费，Consumer 会等待一段时间之后再返回，这段时长即为 timeout。

2.5.2 分区分配策略

一个消费者可以订阅多个主题，可以去消费多个分区，一个分区不支持多个消费者（同一个消费组）读取。

一个消费者组中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。当消费者组里面的消费者个数发生改变的时候，也会触发再平衡。

Kafka 有四种分配策略，可以通过参数 partition.assignment.strategy 来配置，默认 Range + CooperativeSticky。

- Range：针对每个 topic。将 topic 中的分区与消费者排序，通过分区数/消费者数决定每个消费者消费几个分区，若除不尽则前面几个消费者会多消费1个分区。注意，如果有N个 topic，容易产生数据倾斜
- RoundRobin：针对集群中的所有 topic。把所有分区和所有的消费者都列出来，然后按照 hashcode 进行排序，最后通过轮训算法来分配分区给到各个消费者
- Sticky：粘性分区从 0.11.x 版本开始引入，首先会尽量均衡的放置分区到消费者上面，在出现同一消费者组内消费者出现问题的时候，会尽量保持原有分配的分
  区不变化
- **CooperativeSticky: 在不停止**消费的情况下进行增量再平衡。这与 Sticky 的逻辑相同，**但具有增量支持**。这种策略可能会产生不平衡的分配。

1 RangeAssignor分配策略

RangeAssignor 分配策略的原理是按照消费者总数和分区总数进行整除运算来获得一个跨度， 然后将分区按照跨度进行平均分配，以保证分区尽可能均匀地分配给所有的消费者。

每一个主题，RangeAssignor策略会将消费组内所有订阅这个主题的消费者按照名称的字典序排序，然后为每个消费者划分固定的分区范围，如果不够平均分配，那么字典序靠前的消费者会被多分配一个分区。

假设n= **分区数** / 消费者数量，m= 分区数%消费者数量，那么前m个消费者每个分配n+1个分区，后面的（消费者数量-m)个消费者每个分配n个分区。

假设消费组内有2个消费者C0和C1都订阅了主题t0和t1，并且每个主题都有4个分区，那 么订阅的所有分区可以标识为： t0p0、t0p1、t0p2、t0p3、t1p0、t1p1、t1p2、t1p3。最终的分配结果为：

- 消费者C0: t0p0、t0p1、t1p0、t1p1
- 消费者C1: t0p2、t0p3、t1p2、t1p3

这样分配得很均匀，那么这个分配策略能够一直保持这种良好的特性吗？我们不妨再来看 另一种情况。假设上面例子中2个主题都只有3个分区，那么订阅的所有分区可以标识为：t0p0、 t0p1、t0p2、t1p0、t1p1、t1p2最终的分配结果为：

- 消费者C0: t0p0、t0p1、t1p0、t1p1
- 消费者C1: t0p2、t1p2

可以明显地看到这样的分配并不均匀，如果将类似的情形扩大，则有可能出现部分消费者过载的情况。对此我们再来看另一种RoundRobinAssignor策略的分配效果如何。

2 RoundRobinAssignor分配策略

RoundRobinAssi gnor 分配策略的原理是将消费组内所有消费者及消费者订阅的所有主题的分 区按照字典序排序，然后通过轮询方式逐个将分区依次分配给每个消费者。RoundRobinAssignor 分配策略对应的Partition.assignment.strategy参数值为org.apache.kafka.C1ients.Consumer.RoundRobinAssignor。

如果同一个消费组内所有的消费者的**订阅信息都是相同的**，那么RoundRobinAssignor分配策略的分区分配会是均匀的。

比如：假设消费组中有2个消费者C0 和C1都订阅了主题 t0和t1, 并且每个主题都有3个分区，那么订阅的所有分区可以标识为：t0p0、t0p1、t0p2、t1p0、 t1p1、t1p2。最终的分配结果为：

- 消费者C0: t0p0、t0p2、t1p1
- 消费者C1: t0p1、t1p0、t1p2

如果同一个消费组内的消费者订阅的信息是不相同的， 那么在执行分区分配的时候就不是完全的轮询分配，有可能导致分区分配得不均匀。 如果某个消费者**没有订阅消费组内的某个主题**， 那么在分配分区的时候此消费者将分配不到这个主题的任何分区。

比如：假设消费组内有3个消费者(C0、 C1和C2), 它们共订阅了3个主题(t0、t1、 t2) , 这 3个主题分别有1、2、3个分区， 即整个消费组订阅了t0p0、 t1p0、 t1p1、 t2p0、 t2p1、 t2p2这6个分区。 具体而言， 消费者 C0 订阅的是主题t0, 消费者C1 订阅的是主题t0和t1, 消费者C2 订阅的是主题t0、t1和t2, 那么最终的分配结果为：

- 消费者C0: t0p0
- 消费者C1: t1p0
- 消费者C2: t1p1、t2p0、 t2p1、t2p2

可以看到RoundRobinAssignor策略也不是十分完美， 这样分配其实并不是最优解， 因为完全可以将分区t1p1 分配给消费者C1。

所以需要注意：如果使用RoundRobinAssignor策略，**则消费者应该订阅相同的主题。**

3 StickyAssignor分配策略

我们再来看一下StickyAssignor分配策略， "sticky"这个单词可以翻译为“ 黏性的", Kafka 从0.11.x版本开始引入这种分配策略， 它主要有两个目的：

(1)分区的分配要尽可能均匀。

(2)分区的分配尽可能与上次分配的保待相同。

当两者发生冲突时， 第一个目标优先于第二个目标。

鉴于这两个目标， StickyAssignor分配策略的具体实现要比RangeAssignor和RoundRobinAssignor这两种分配策略要复杂得多。 我们举 例来看一下StickyAssignor分配策略的实际效果。

假设消费组内有3个消费者(C0、C1和C2)，它们都订阅了4个主题(t0、t1、t2、t3)，并且每个主题有2个分区。 也就是说，整个消费组订阅了t0p0、 t0p1、 t1p0、 t1p1、 t2p0、 t2p1、 t3p0、 t3p1这8个分区。 最终的分配结果如下：

- 消费者C0: t0p0、t1p1、t3p0
- 消费者C1: t0p1、t2p0、t3p1
- 消费者C2: t1p0、t2p1

这样初看上去似乎与采用RoundRobinAssignor分配策略所分配的结果相同， 但事实是否真的如此呢? 再假设此时消费者 C1脱离了消费组， 那么消费组就会执行再均衡操作，进而消费分区会重新分配。 如果采用RoundRobinAssignor 分配策略， 那么此时的分配结果如下：

- 消费者C0: t0p0、t1p0、t2p0、t3p0
- 消费者C2: t0p1、t1p1、t2p1、t3p1

如分配结果所示，RoundRobinAssignor分配策略会按照消费者C0 和C2进行重新轮询分配。 如果此时使用的是StickyAssignor分配策略，那么分配结果为：

- 消费者C0: t0p0、t1p1、t3p0、**t2p0**
- 消费者C2: t1p0、t2p1、**t0p1**、**t3p1**

可以看到分配结果中保留了上一次分配中对消费者 C0 和C2的所有分配结果，并将原来消费者C1的 “ 负担 “ 分配给了剩余的两个消费者 C0 和C2, 最终 C0 和C2的分配还保持了均衡。

如果发生分区重分配，那么对于同一个分区而言，有可能之前的消费者和新指派的消费者不是同一个，**之前消费者进行到一半的处理还要在新指派的消费者中再次复现一遍**，这显然很浪费系统资源。

StickyAssignor 分配策略如同其名称中的"st1cky" 一样，让分配策略具备一定 的 “ 黏性 ” ，**尽可能地让前后两次分配相同**，进而减少系统资源的损耗及其他异常情况的发生。

到目前为止，我们分析的都是消费者的订阅信息都是相同的情况，我们来看一下订阅信息不同的清况下的处理。

举个例子，同样消费组内有3个消费者(C0、C1和C2) , 集群中有3个主题(t0、t1和 t2) , 这3个主题分别有**1、2、3个分区**。也就是说，集群中有t0p0、 t1p0、 t1p1、 t2p0、 t2p1、 t2p2这6个分区。消费者C0 订阅了主题t0，消费者C1订阅了主题t0和t1, 消费者C2订阅了主题t0、t1和t2。 如果此时采用RoundRobinAssignor分配策略，那么最终的分配结果如RoundRobinAssignor分配策略时的一样

RoundRobinAssignor分配策略的分配结果

- 消费者C0: t0p0
- 消费者C1: t1p0
- 消费者C2: t1p1、t2p0、t2p1、t2p2

如果此时采用的是StickyAssignor分配策略，那么最终的分配结果如下所示。

StickyAssignor分配策略的分配结果

- 消费者C0: t0p0
- 消费者C1: t1p0、t1p1
- 消费者C2: t2p0、t2p1、t2p2

可以看到这才是一个最优解（消费者C0 没有订阅主题t1和t2, 所以不能分配主题t1和t2 中的任何分区给它， 对于消费者C1也可同理推断）。

假如此时消费者C0 脱离了消费组， 那么RoundRobinAssignor分配策略的分配结果为：

- 消费者C1: t0p0、t1p1
- 消费者C2: t1p0、t2p0 、t2p1、 t2p2

可以看到RoundRobinAssignor策略保留了消费者C1和C2中原有的**3个分区**的分配：t2p0、 t2p l和t2p2。

如果采用的是StickyAssignor分配策略， 那么分配结果为：

- 消费者C1: t1p0、t1p1、t0p0
- 消费者C2: t2p0、t2p1、 t2p2

可以看到StickyAssignor分配策略保留了消费者C1和C2中原有的**5个分区**的分配：t1p0、t1p1、t2p0、t2p1、t2p2

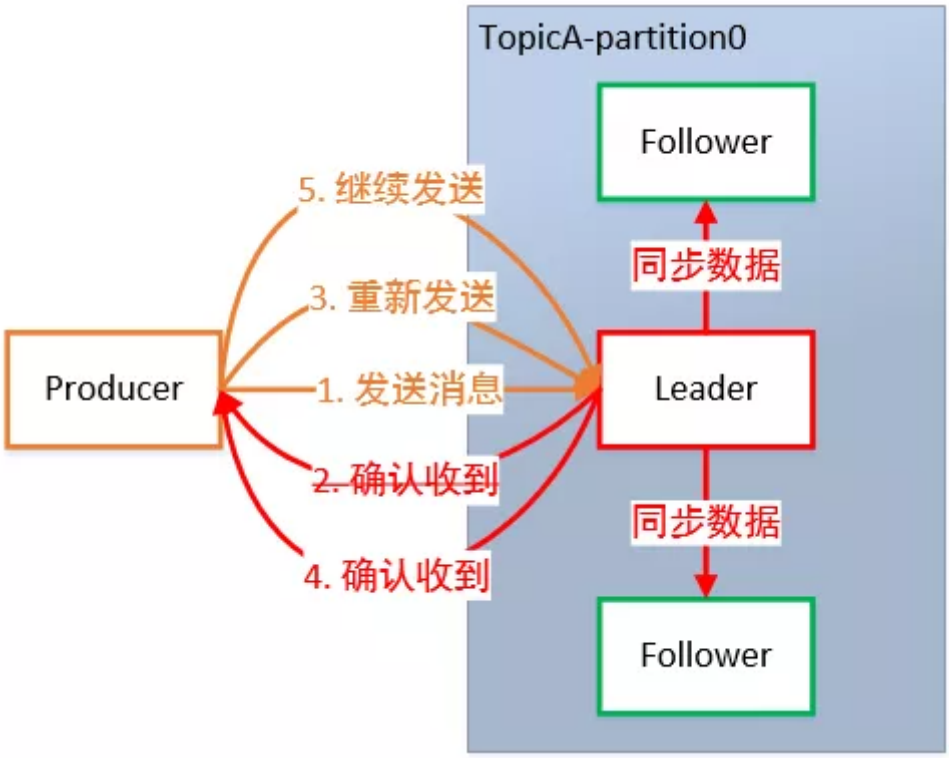
使用StickyAssignor分配策略的一个优点就是可以使分区重分配具备 " 黏性"' **减少不必要的分区移动**（即一个分区剥离之前的消费者，转而分配给另一个新的消费者）。

StickyAssignor分配策略比另外两者分配策略而言显得更加优异，但这个策略的代码实现也异常复杂，

2.6 数据可靠性保证

为保证 Producer 发送的数据，能可靠地发送到指定的 topic，topic 的每个 Partition 收到 Producer 发送的数据后，都需要向 Producer 发送 ACK（ACKnowledge 确认收到）。

如果 Producer 收到 ACK，就会进行下一轮的发送，否则重新发送数据。



2.6.1 副本数据同步策略

何时发送 ACK？确保有 Follower 与 Leader 同步完成，Leader 再发送 ACK，这样才能保证 Leader 挂掉之后，能在 Follower 中选举出新的 Leader 而不丢数据。

多少个 Follower 同步完成后发送 ACK？全部 Follower 同步完成，再发送 ACK。

方案	优点	缺点
半数以上完成同步，就发送ack	延迟低	选举新的leader时，容忍n台节点故障，需要2n+1个副本。
全部完成同步，才发送ack	选举新的leader时，容忍n台节点故障，需要n+1个副本。	延迟高。

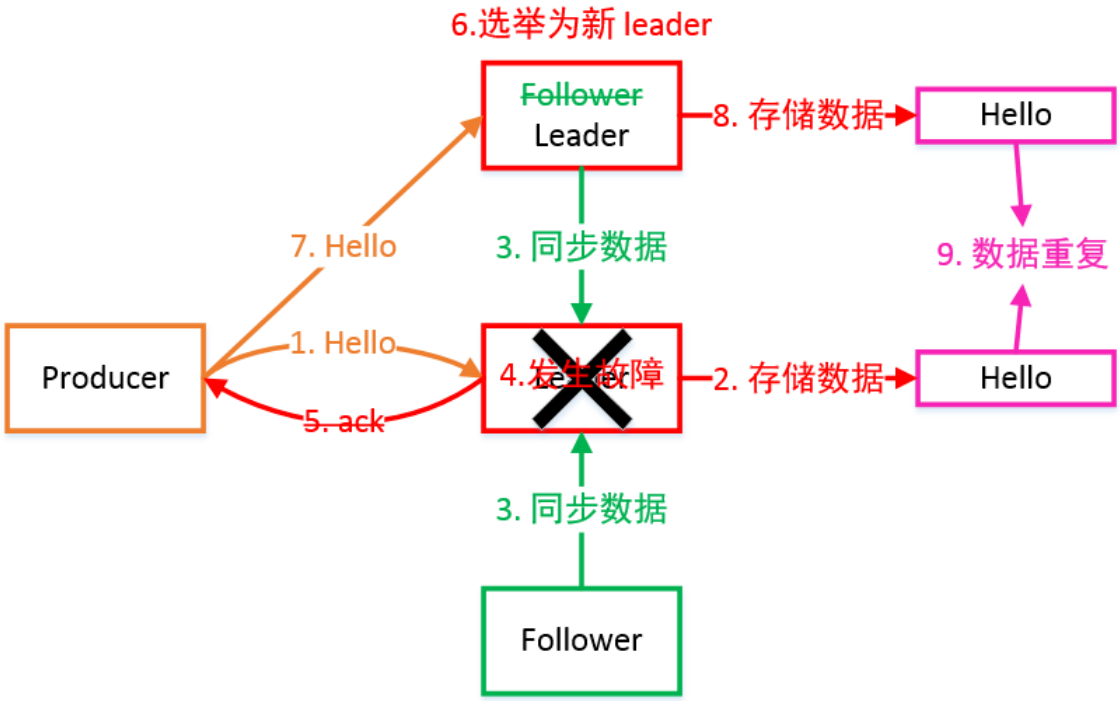
当采用第二种方案时，所有 Follower 完成同步，Producer 才能继续发送数据，设想有一个 Follower 因为某种原因出现故障，那 Leader 就要一直等到它完成同步。这个问题怎么解决？

1. Leader维护了一个动态的** in-sync replica set（ISR）**：和 Leader 保持同步的 Follower 集合。
2. 当 ISR 集合中的 Follower 完成数据的同步之后，Leader 就会给 Follower 发送 ACK。
3. 如果 Follower 长时间未向 Leader 同步数据，则该 Follower 将被踢出 ISR 集合，该时间阈值由 replica.lag.t1me.max.ms 参数设定。Leader 发生故障后，就会从 ISR 中选举出新的 Leader。

2.6.2 ACK 应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等 ISR 中的 Follower 全部接受成功。

所以 Kafka 为用户提供了三种可靠性级别，用户根据可靠性和延迟的要求进行权衡，选择以下的配置。



ACK 参数配置：

- 0：Producer 不等待 Broker 的 ACK，这提供了**最低延迟**，Broker 一收到数据还没有写入磁盘就已经返回，当 Broker 故障时有可能丢失数据。
- 1：Producer 等待 Broker 的 ACK，Partition 的 **Leader 落盘成功后返回 ACK**，如果在 Follower 同步成功之前 Leader 故障，那么将会丢失数据。
- -1（all）：Producer 等待 Broker 的 ACK，Partition 的 Leader 和 Follower 全部落盘成功后才返回 ACK。但是在 Broker 发送 ACK 时，Leader 发生故障，则会造成数据重复。

2.6.3 可靠性指标

没有一个中间件能够做到百分之百的完全可靠，可靠性更多的还是基于几个9的衡量指标，比如4个9、5个9。软件系统的可靠性只能够无限去接近100%，但不可能达到100%。所以kafka如何是实现最大可能的可靠性呢?

- **分区副本**，你可以创建更多的分区来提升可靠性，但是分区数过多也会带来性能上的开销，一般来说，**3个副本**就能满足对大部分场景的可靠性要求
- **ACKS**，生产者发送消息的可靠性，也就是我要保证我这个消息一定是到了broker并且完成了多副本的持久化。参考2.5.2 AC应答机制
- 保障消息到了broker之后，消费者也需要有一定的保证，因为消费者也可能出现某些问题导致消息没有消费到。
- enable.aut0.C0mmit默认为true，也就是自动提交offset，自动提交是批量执行的，有一个时间窗口，这种方式会带来重复提交或者消息丢失的问题，所以对于高可靠性要求的程序，要使用手动提交。 对于高可靠要求的应用来说，宁愿重复消费也不应该因为消费异常而导致消息丢失。

3 参考

2 万字长文深入详解 Kafka，从源码到架构全部讲透

https://mp.weixin.qq.C0m/s/dOiNT0a_dRytwatzdrJNCg

Kafka 日志存储

https://zhuanlan.zhihu.C0m/p/65415304