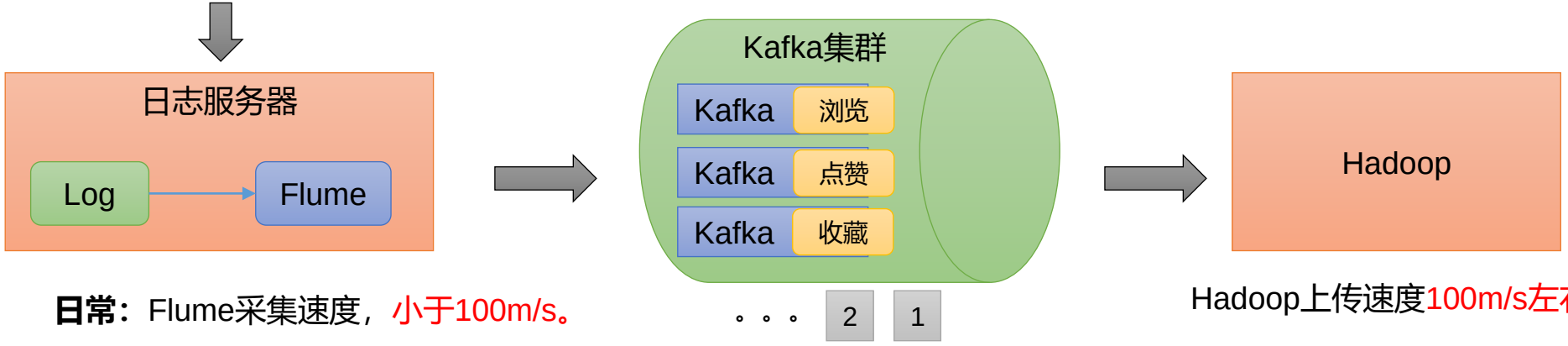
Kafka定义

前端埋点记录用户购买海购人参丸的行为数据（浏览、点赞、收藏、评论等）。

Kafka传统定义：Kafka是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域。

发布/订阅：消息的发布者不会将消息直接发送给特定的订阅者，而是将发布的消息分为不同的类别，订阅者只接收感兴趣的消息。

Kafka最新定义：Kafka是一个开源的分布式事件流平台（Event Streaming Platform），被数千家公司用于高性能数据管道、流分析、数据集成和关键任务应用。



日常：Flume采集速度，小于100m/s。

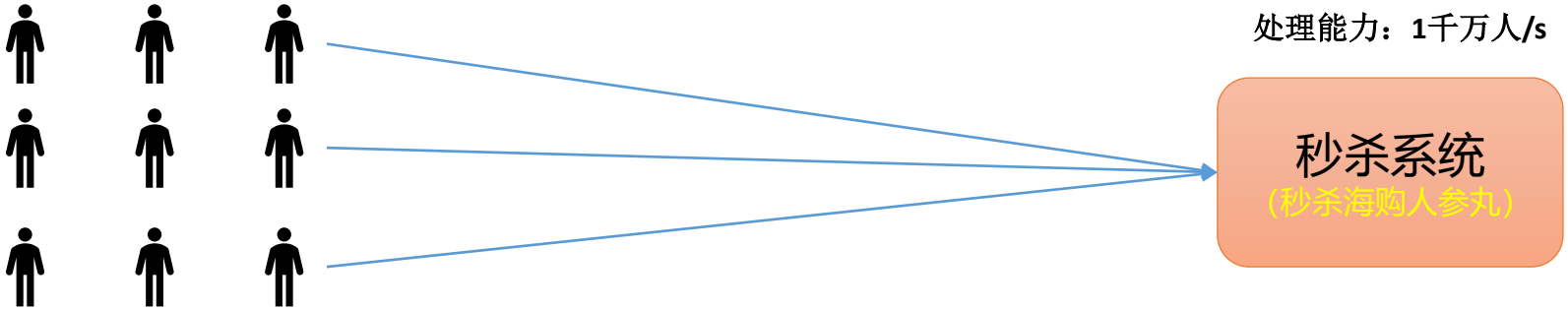
11.11 活动：Flume采集速度，大于200m/s。

Hadoop上传速度100m/s左右。

消息队列的应用场景——缓冲/消峰

缓冲/消峰：有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致的情况。

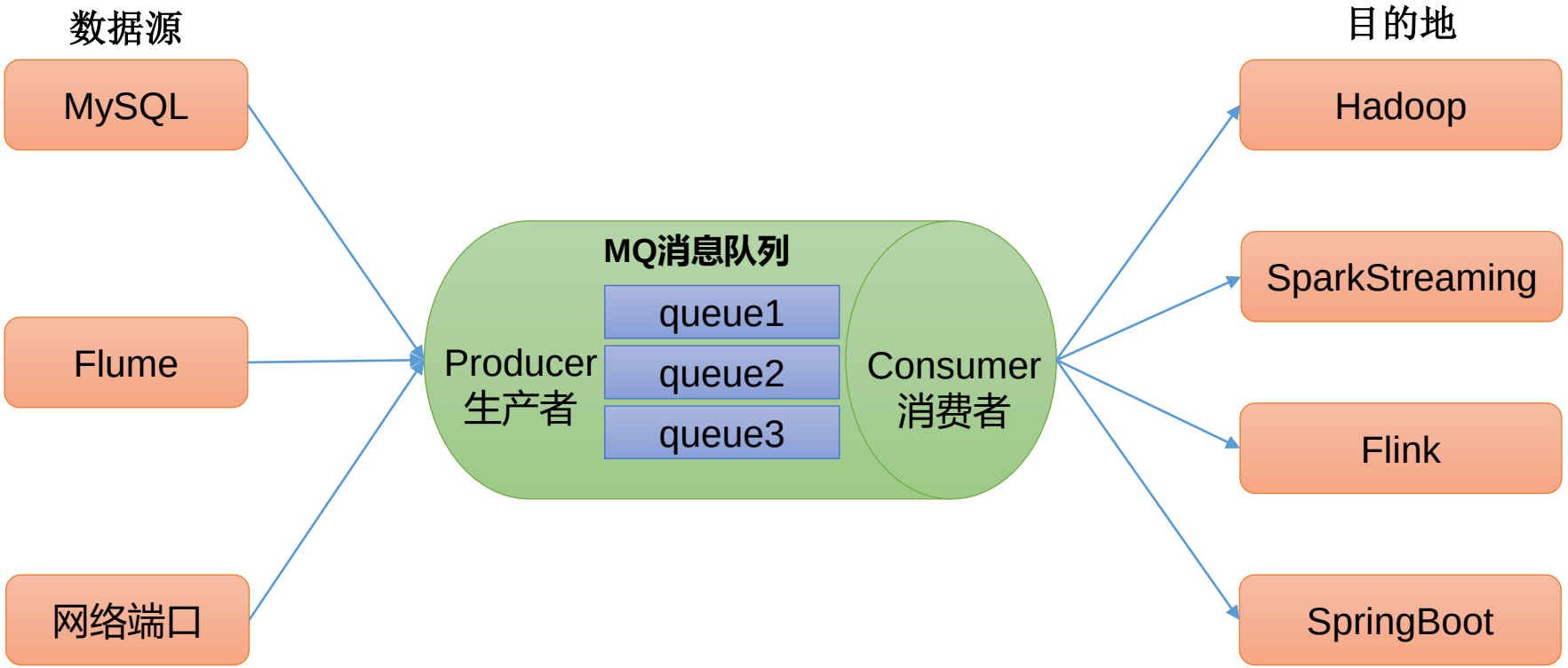
双十一参与用户：10亿人/s



双十一参与用户：10亿人/s

消息队列的应用场景——解耦

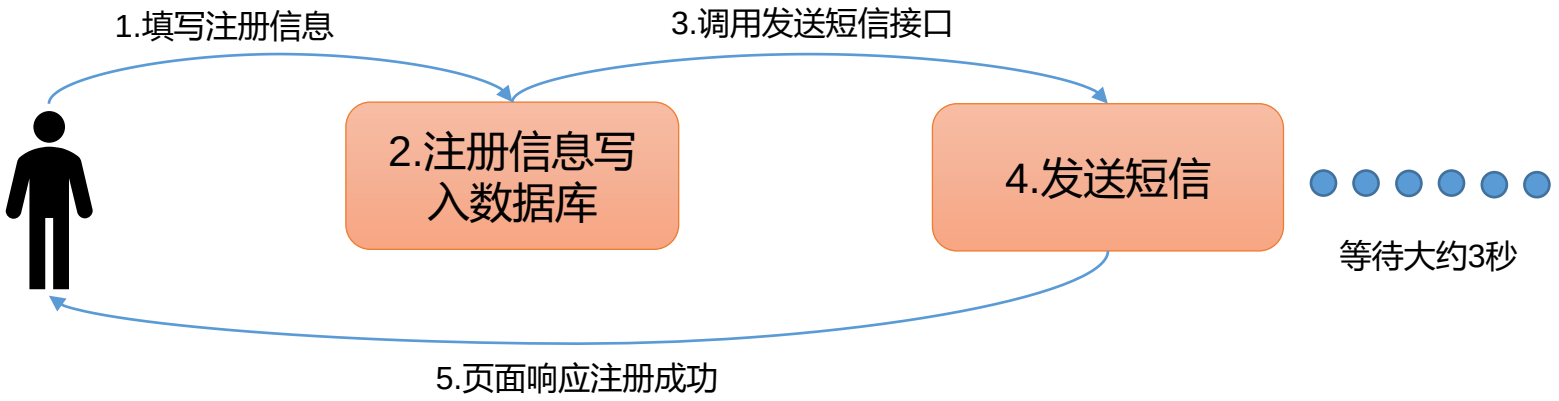
解耦：允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。



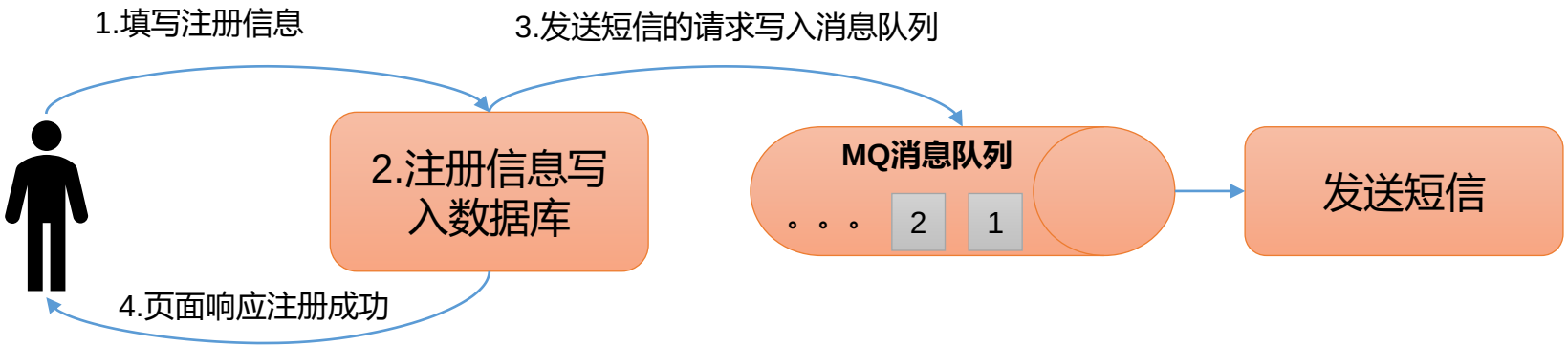
消息队列的应用场景——异步通信

异步通信：允许用户把一个消息放入队列，但并不立即处理它，然后在需要的时候再去处理它们。

同步处理



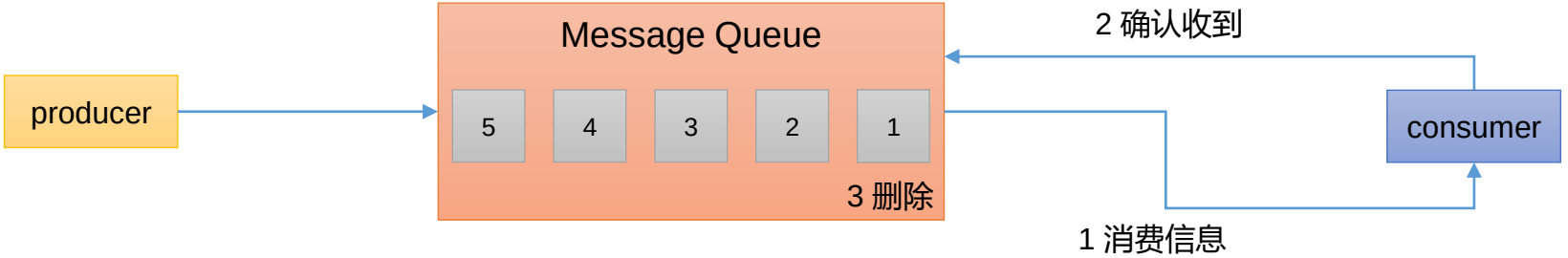
异步处理



消息队列的两种模式

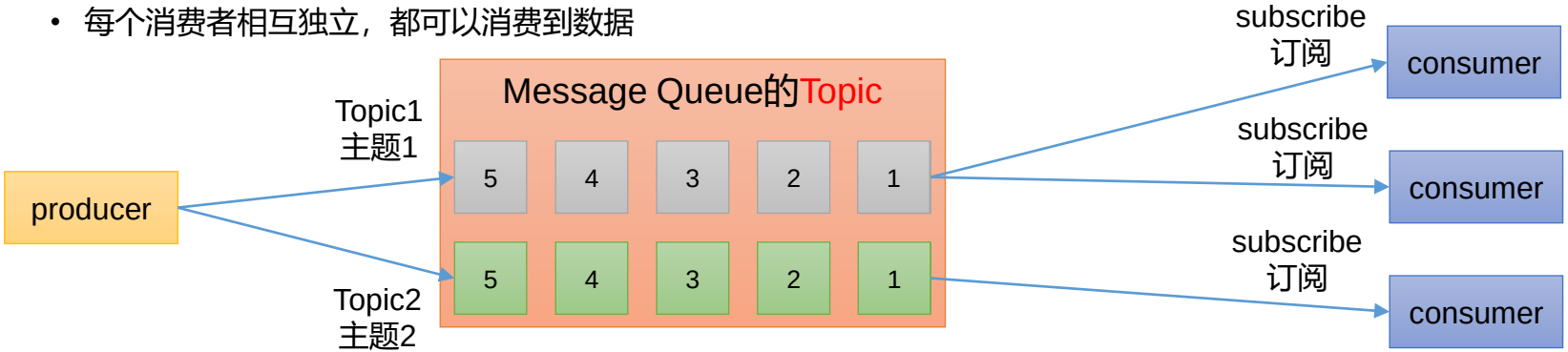
1）点对点模式

- 消费者主动拉取数据，消息收到后清除消息



2）发布/订阅模式

- 可以有多个topic主题（浏览、点赞、收藏、评论等）
- 消费者消费数据之后，不删除数据
- 每个消费者相互独立，都可以消费到数据



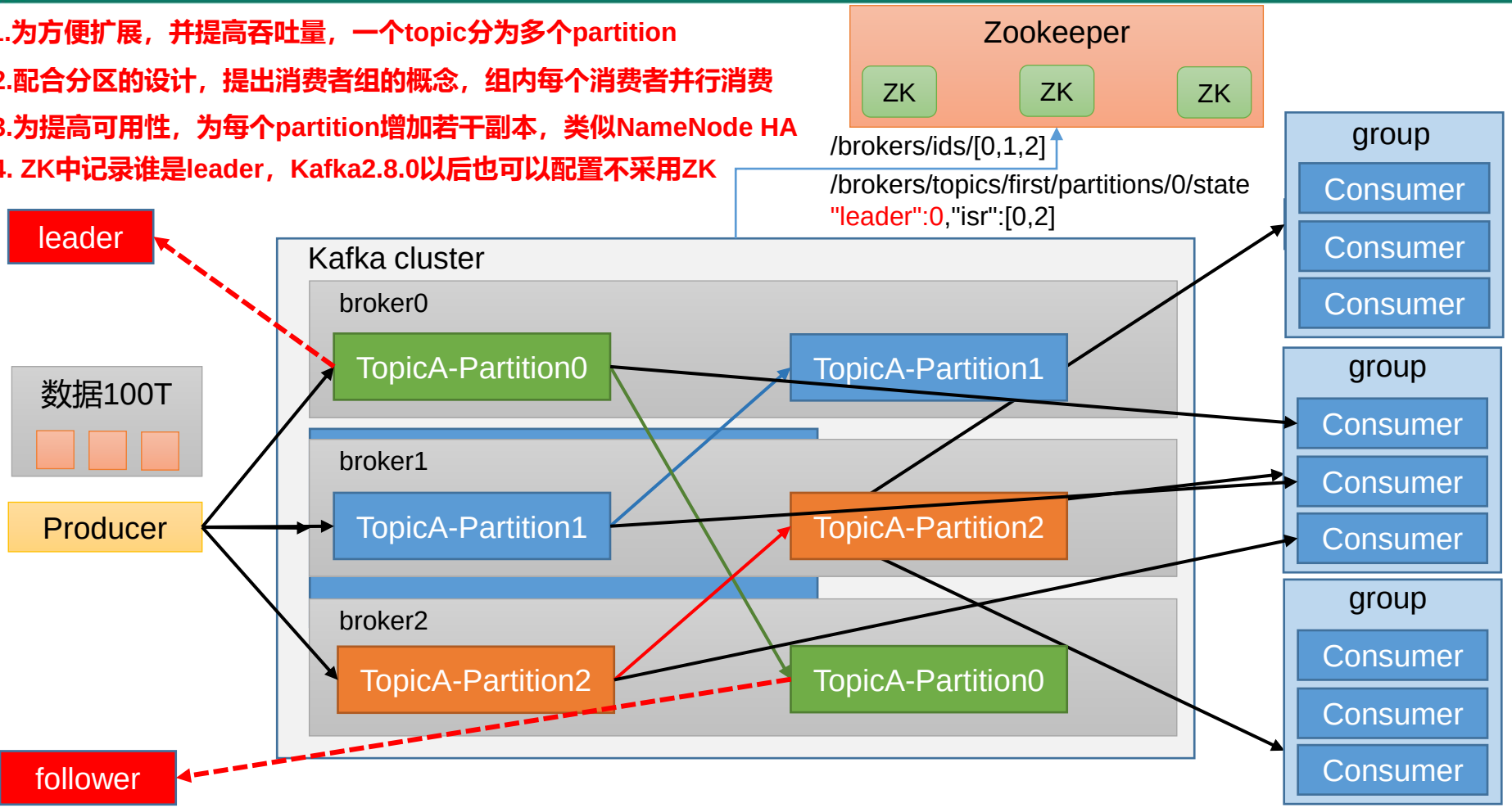
Kafka 基础架构

1.为方便扩展，并提高吞吐量，一个topic分为多个partition

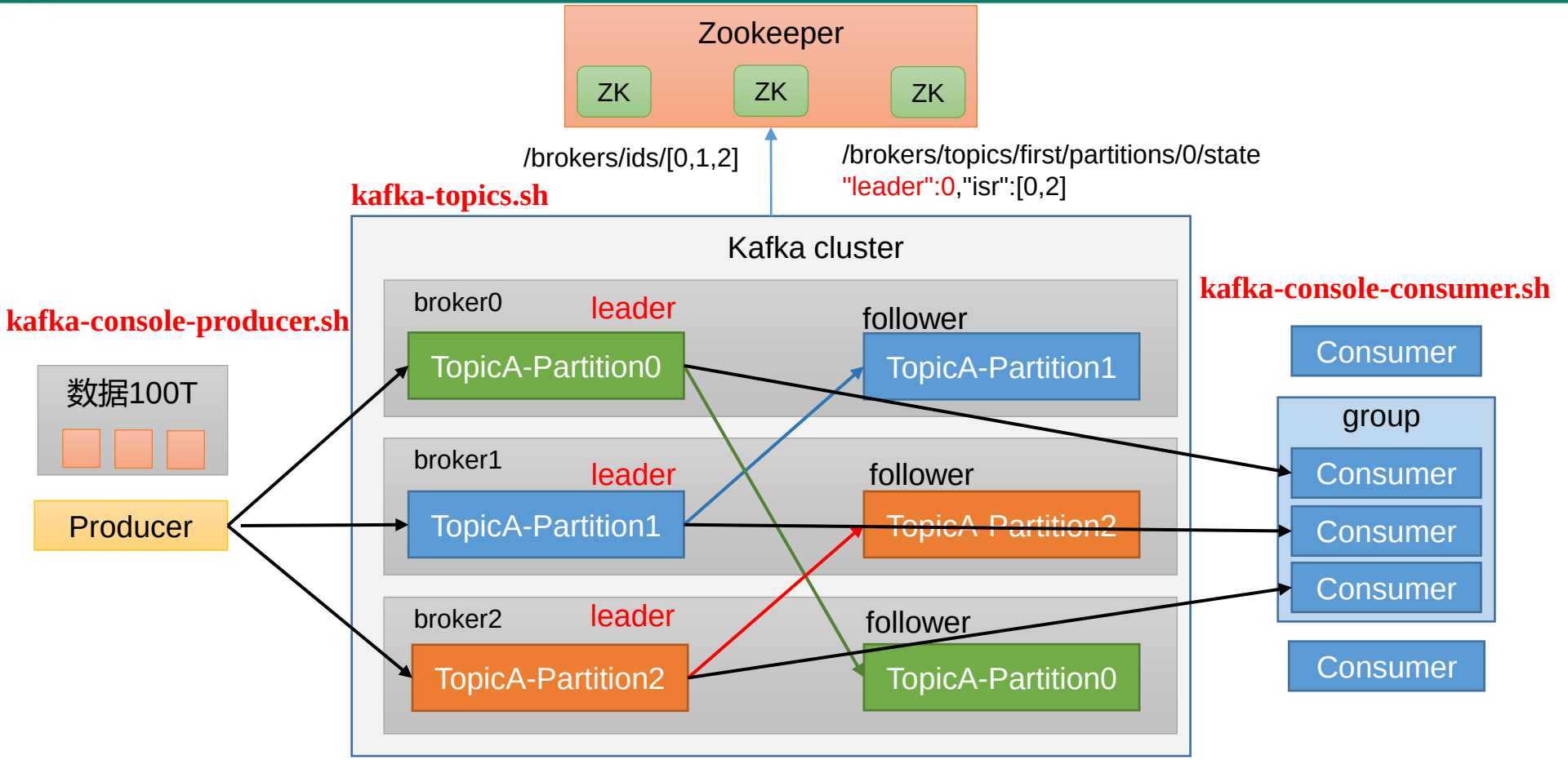
2.配合分区的设计，提出消费者组的概念，组内每个消费者并行消费

3.为提高可用性，为每个partition增加若干副本，类似NameNode HA

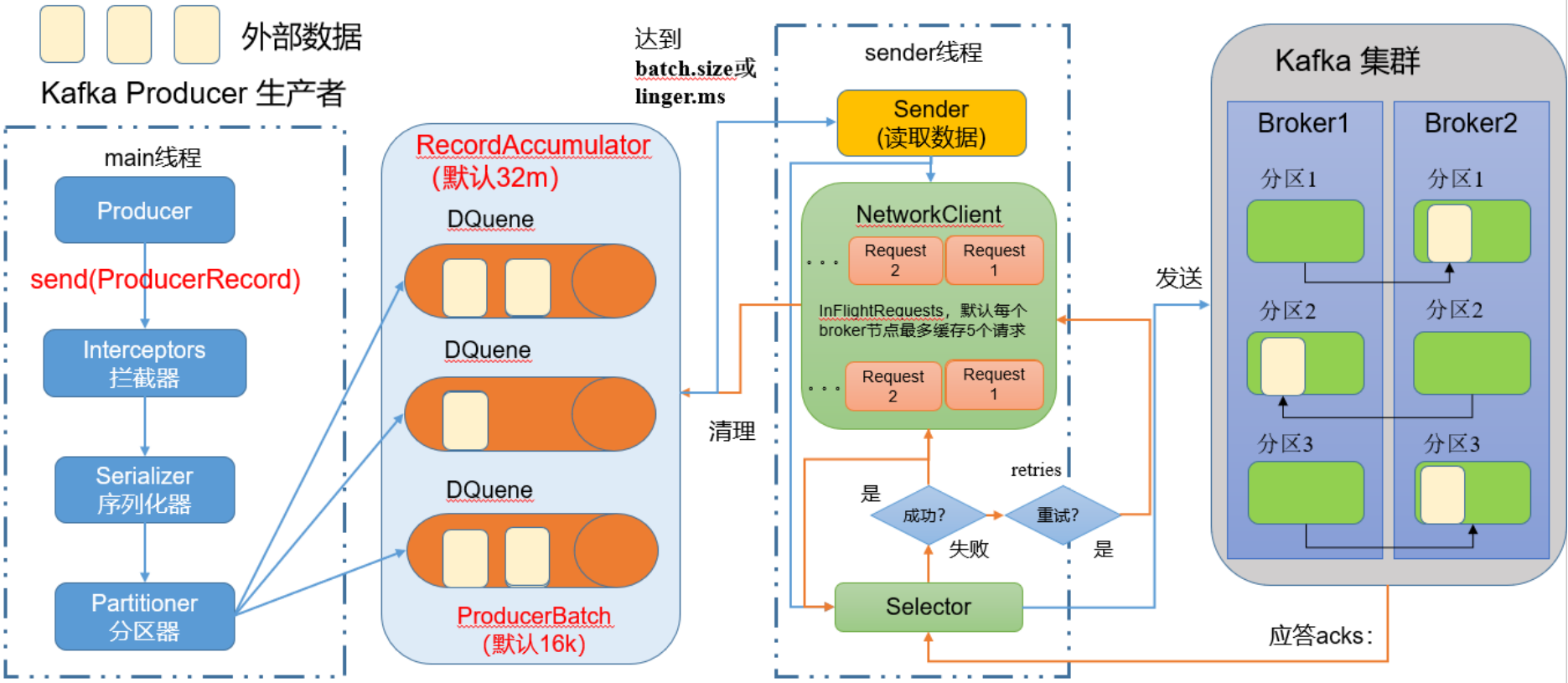
4. ZK中记录谁是leader，Kafka2.8.0以后也可以配置不采用ZK



Kafka 基础架构


Zookeeper
ZK
ZK
ZK
/brokers/ids/[0,1,2]
/brokers/topics/first/partitions/0/state
"leader":0,"isr":[0,2]
kafka-topics.sh
Kafka cluster
kafka-console-producer.sh
kafka-console-consumer.sh
数据100T
Producer
broker0
leader
TopicA-Partition0
follower
TopicA-Partition1
broker1
leader
TopicA-Partition1
follower
TopicA-Partition2
broker2
leader
TopicA-Partition2
follower
TopicA-Partition0
Consumer
group
Consumer
Consumer
Consumer
Consumer

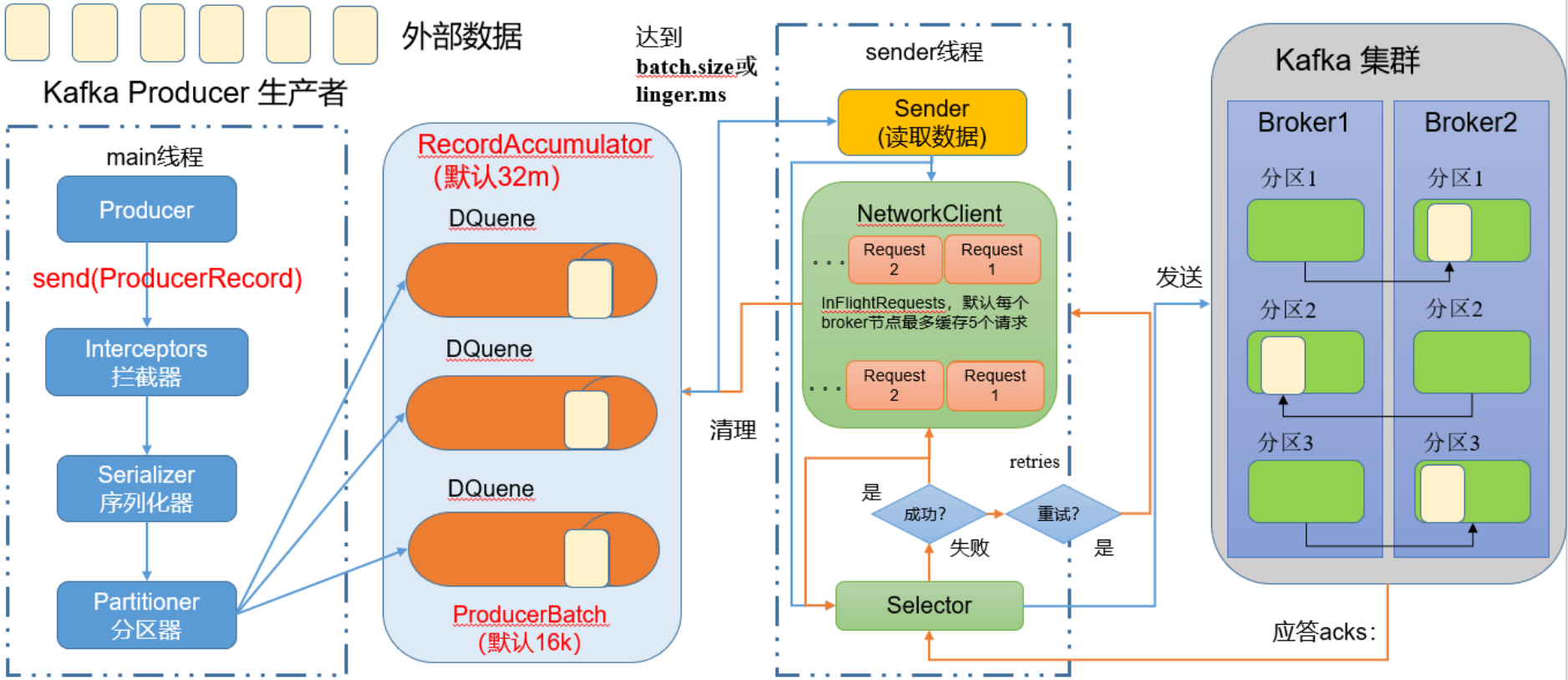
发送流程



- **batch.size**：只有数据积累到batch.size之后，sender才会发送数据。默认16k
- **linger.ms**：如果数据迟迟未达到batch.size，sender等待linger.ms设置的时间到了之后就会发送数据。单位ms，默认值是0ms，表示没有延迟。

- 0：生产者发送过来的数据，不需要等数据落盘应答。
- 1：生产者发送过来的数据，Leader收到数据后应答。
- -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。-1和all等价。

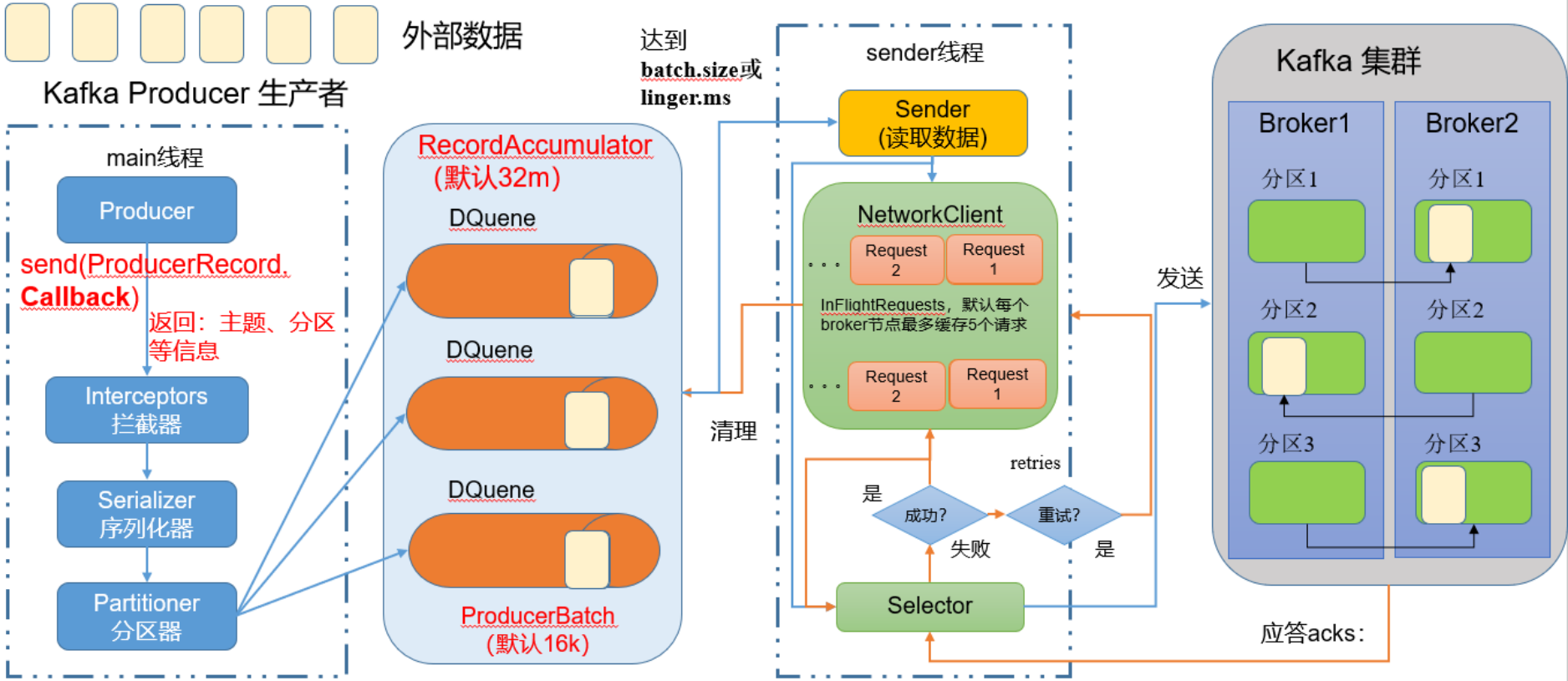
异步发送流程



- **batch.size**：只有数据积累到batch.size之后，sender才会发送数据。默认16k
- **linger.ms**：如果数据迟迟未达到batch.size，sender等待linger.ms设置的时间到了之后就会发送数据。单位ms，默认值是0ms，表示没有延迟。

- 0：生产者发送过来的数据，不需要等数据落盘应答。
- 1：生产者发送过来的数据，Leader收到数据后应答。
- -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。-1和all等价。

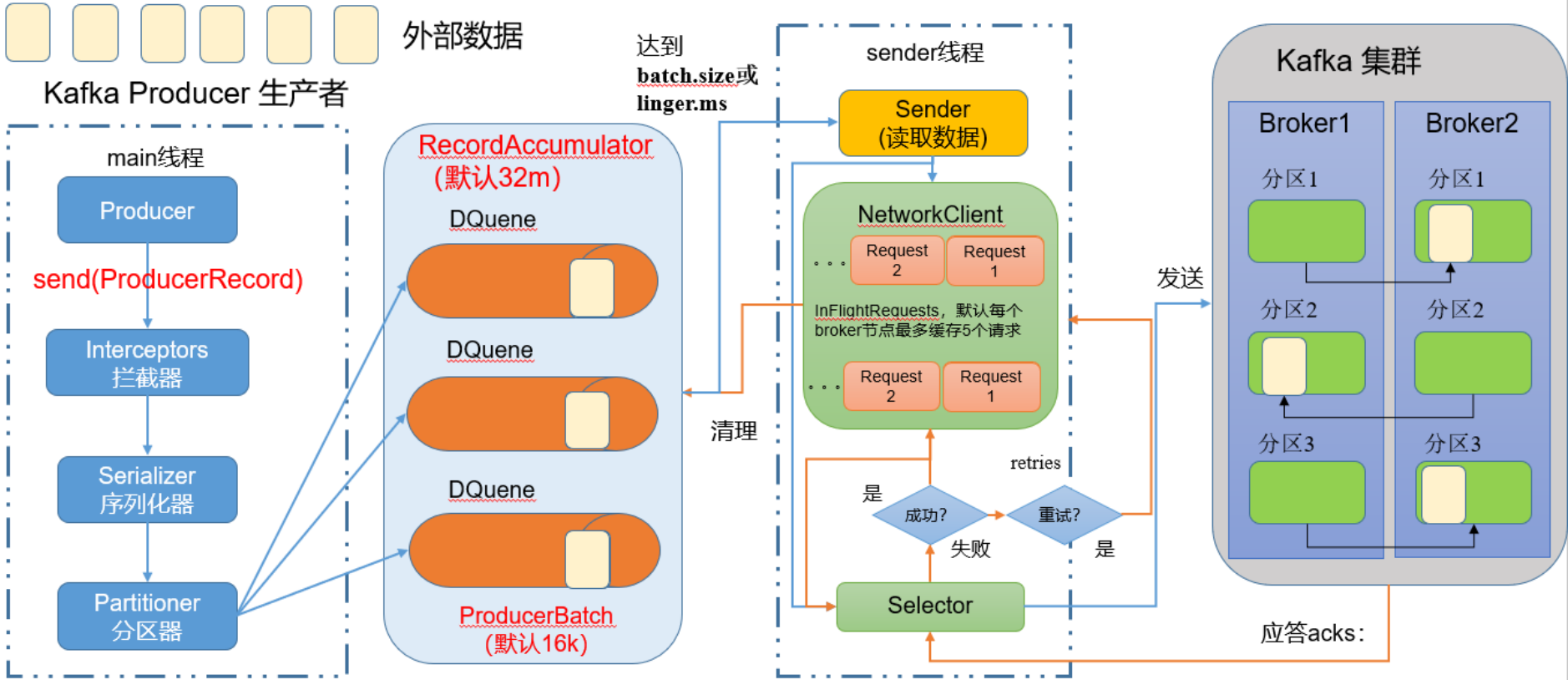
带回调函数的异步发送流程



- **batch.size**：只有数据积累到batch.size之后，sender才会发送数据。默认16k
- **linger.ms**：如果数据迟迟未达到batch.size，sender等待linger.ms设置的时间到了之后就会发送数据。单位ms，默认值是0ms，表示没有延迟。

- 0：生产者发送过来的数据，不需要等数据落盘应答。
- 1：生产者发送过来的数据，Leader收到数据后应答。
- -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。-1和all等价。

同步发送流程



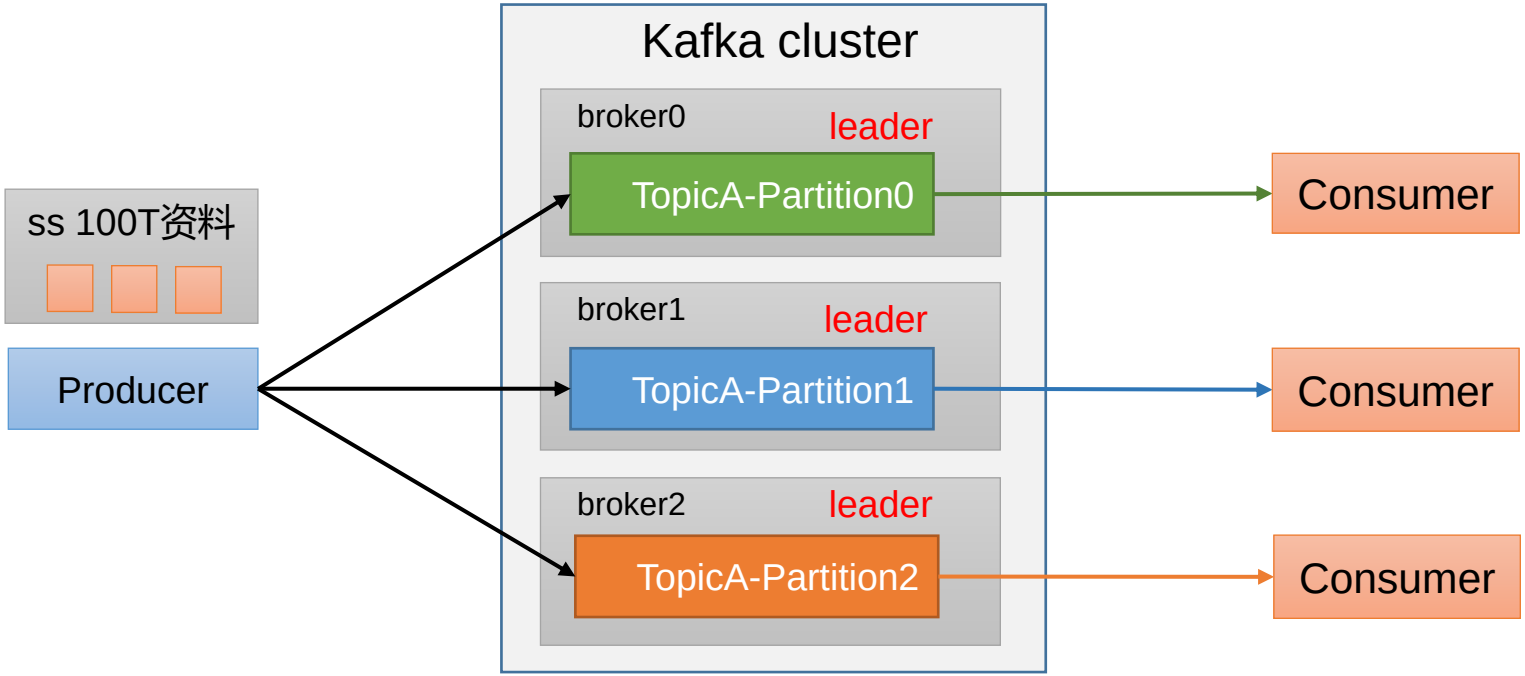
- **batch.size**：只有数据积累到batch.size之后，sender才会发送数据。默认16k
- **linger.ms**：如果数据迟迟未达到batch.size，sender等待linger.ms设置的时间到了之后就会发送数据。单位ms，默认值是0ms，表示没有延迟。

- 0：生产者发送过来的数据，不需要等数据落盘应答。
- 1：生产者发送过来的数据，Leader收到数据后应答。
- -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。-1和all等价。

Kafka 分区好处

（1）**便于合理使用存储资源**，每个Partition在一个Broker上存储，可以把海量的数据按照分区切割成一块一块数据存储在多台Broker上。合理控制分区的任务，可以实现负载均衡的效果。

（2）**提高并行度**，生产者可以以分区为单位发送数据；消费者可以以分区为单位进行消费数据。



生产经验——生产者如何提高吞吐量

ss家仓库,默认32m

一次拉一个，来了就走

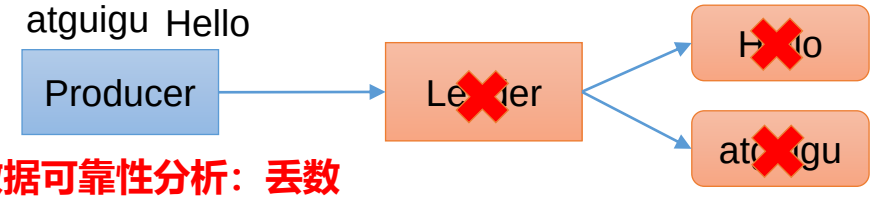
broker

- batch.size：批次大小，默认16k
- linger.ms：等待时间，修改为5-100ms
- compression.type：压缩snappy
- RecordAccumulator：缓冲区大小，修改为64m

ACK应答级别

acks：

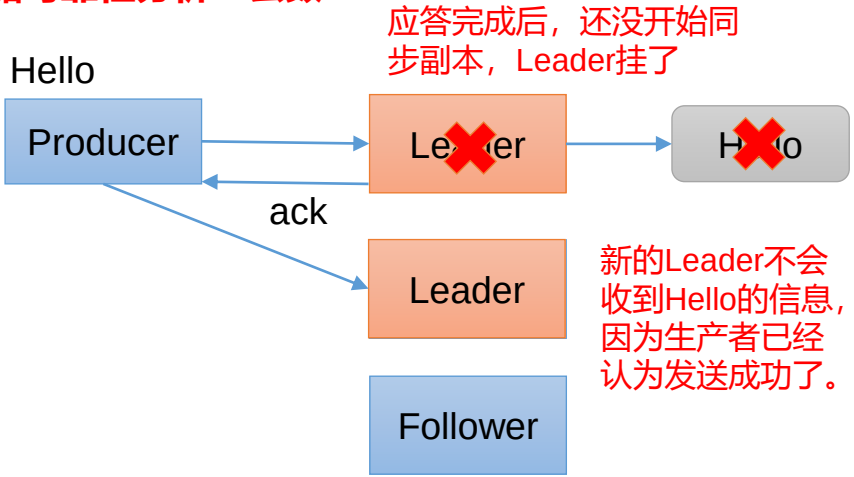
0：生产者发送过来的数据，不需要等数据落盘应答



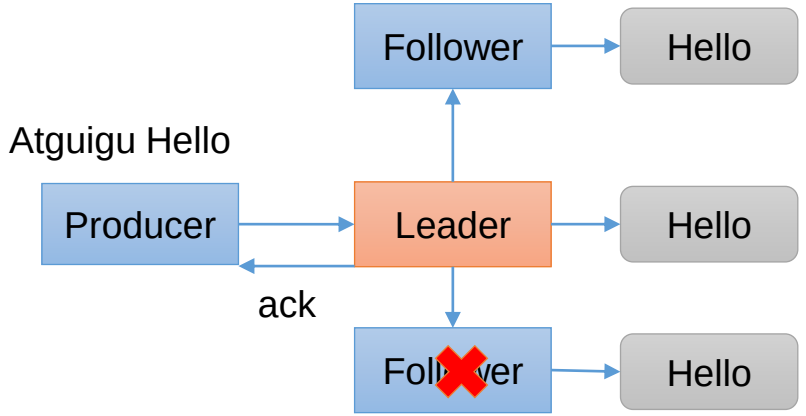
数据可靠性分析：丢数

1：生产者发送过来的数据，**Leader**收到数据后应答。

数据可靠性分析：丢数



-1（all）：生产者发送过来的数据，**Leader和ISR队列里面的所有节点收齐数据后应答。**

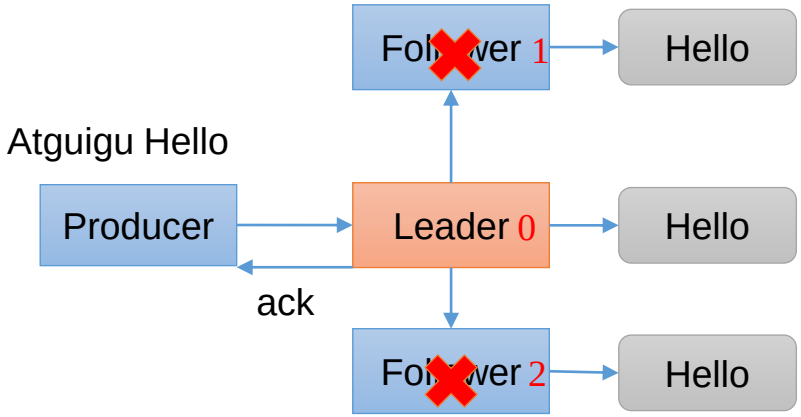


思考：Leader收到数据，所有Follower都开始同步数据，但有一个Follower，因为某种故障，迟迟不能与Leader进行同步，那这个问题怎么解决呢？

ACK应答级别

acks：

-1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。



思考：Leader收到数据，所有Follower都开始同步数据，但有一个Follower，因为某种故障，迟迟不能与Leader进行同步，那这个问题怎么解决呢？

Leader维护了一个动态的in-sync replica set（**ISR**），意为和Leader保持同步的Follower+Leader集合(leader：0，isr:0,1,2)。

如果Follower长时间未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR。该时间阈值由**replica.lag.time.max.ms**参数设定，默认30s。例如2超时，(leader:0, isr:0,1)。

这样就不用等长期联系不上或者已经故障的节点。

数据可靠性分析：

如果分区副本设置为1个，或者ISR里应答的最小副本数量（min.insync.replicas 默认为1）设置为1，和ack=1的效果是一样的，仍然有丢数的风险（leader：0，isr:0）。

- **数据完全可靠条件 = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2**

可靠性总结：

acks=0，生产者发送过来数据就不管了，可靠性差，效率高；

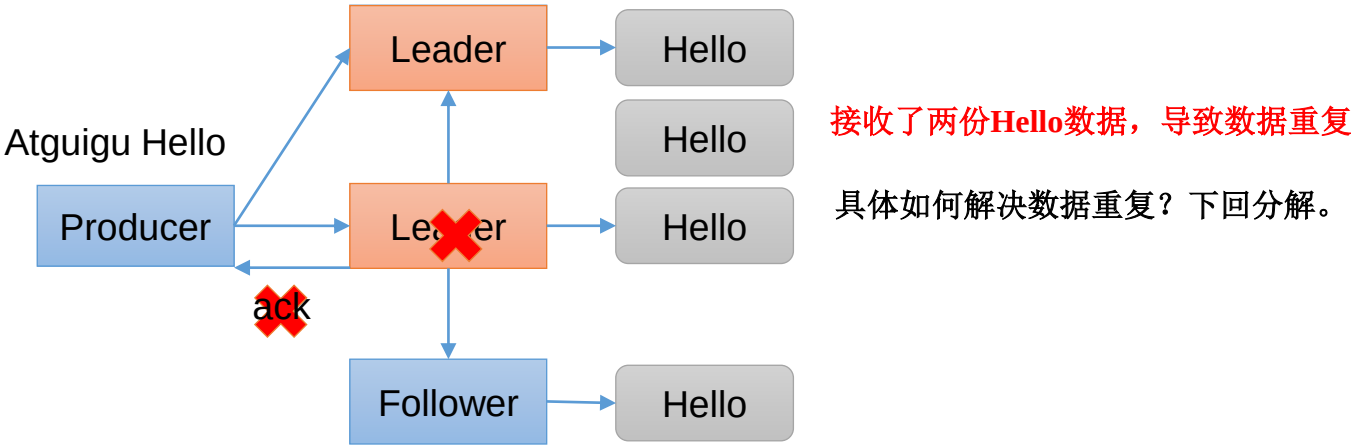
acks=1，生产者发送过来数据Leader应答，可靠性中等，效率中等；

acks=-1，生产者发送过来数据Leader和ISR队列里面所有Follwer应答，可靠性高，效率低；

在生产环境中，acks=0很少使用；acks=1，一般用于传输普通日志，允许丢个别数据；acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景。

数据重复分析：

acks：　-1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答。



接收了两份Hello数据，导致数据重复

具体如何解决数据重复？下回分解。

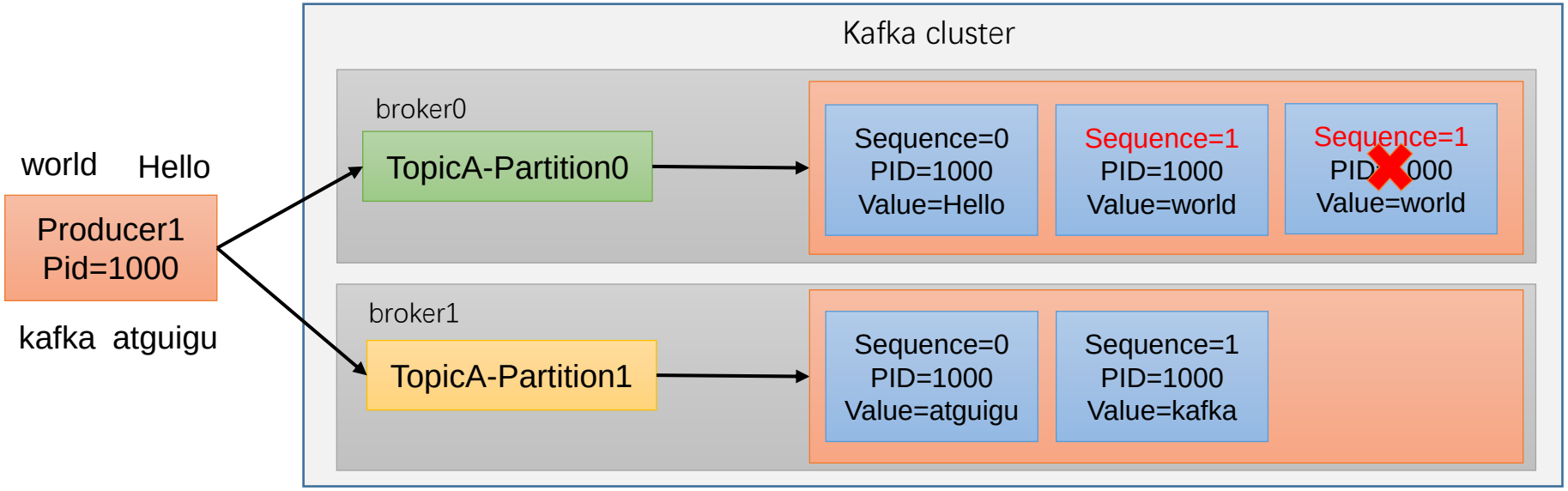
幂等性原理

幂等性就是指Producer不论向Broker发送多少次重复数据，Broker端都只会持久化一条，保证了不重复。

精确一次（Exactly Once）= 幂等性 + 至少一次（ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2）。

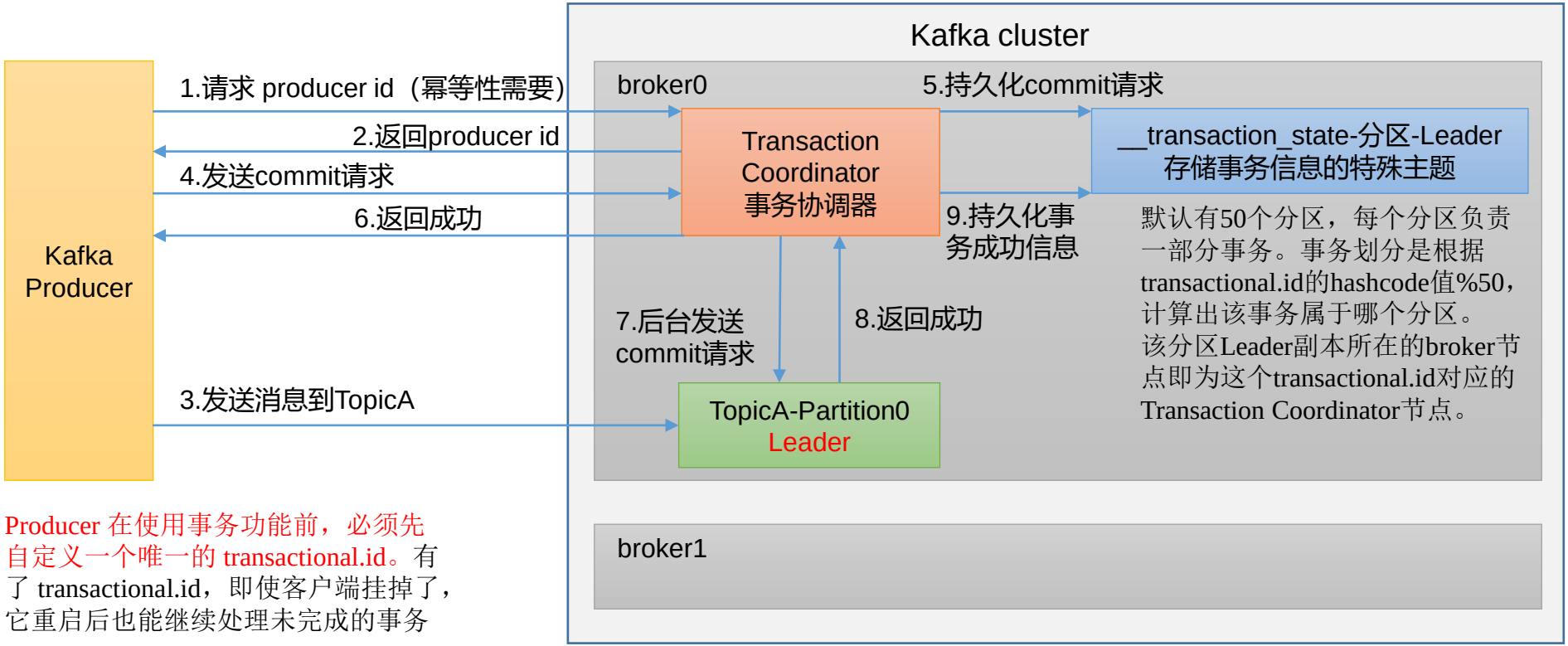
重复数据的判断标准：具有<PID，Partition，SeqNumber>相同主键的消息提交时，Broker只会持久化一条。其中PID是Kafka每次重启都会分配一个新的；Partition 表示分区号；Sequence Number是单调自增的。

所以幂等性只能保证的是在单分区单会话内不重复。



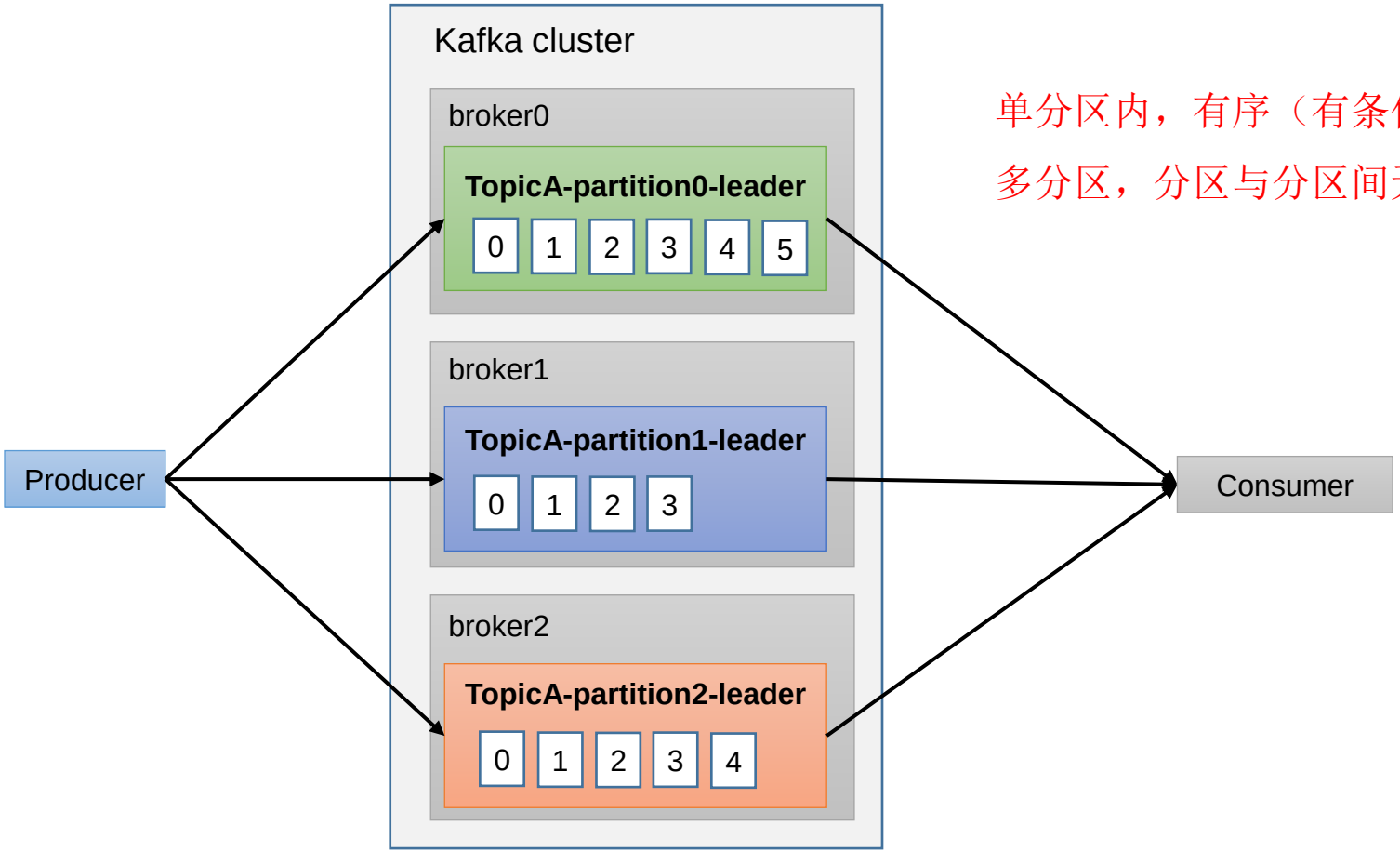
Kafka 事务原理

说明：开启事务，必须开启幂等性。



Producer 在使用事务功能前，必须先自定义一个唯一的 transactional.id。有了 transactional.id，即使客户端挂掉了，它重启后也能继续处理未完成的事务

生产经验——数据有序



单分区内，有序（有条件的，详见下节）；

多分区，分区与分区间无序；

生产经验——数据乱序

1）kafka在1.x版本之前保证数据单分区有序，条件如下：

max.in.flight.requests.per.connection=1（不需要考虑是否开启幂等性）。

2）kafka在1.x及以后版本保证数据单分区有序，条件如下：

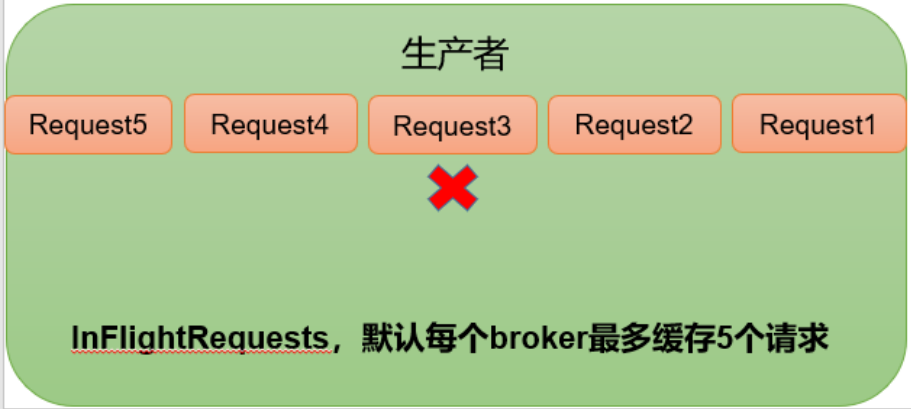
（1）未开启幂等性

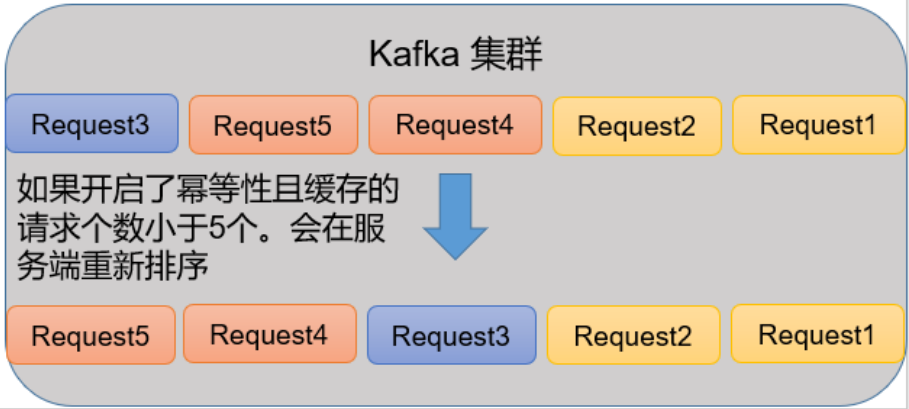
max.in.flight.requests.per.connection需要设置为1。

（2）开启幂等性

max.in.flight.requests.per.connection需要设置小于等于5。

原因说明：因为在kafka1.x以后，启用幂等后，kafka服务端会缓存producer发来的最近5个request的元数据，故无论如何，都可以保证最近5个request的数据都是有序的。





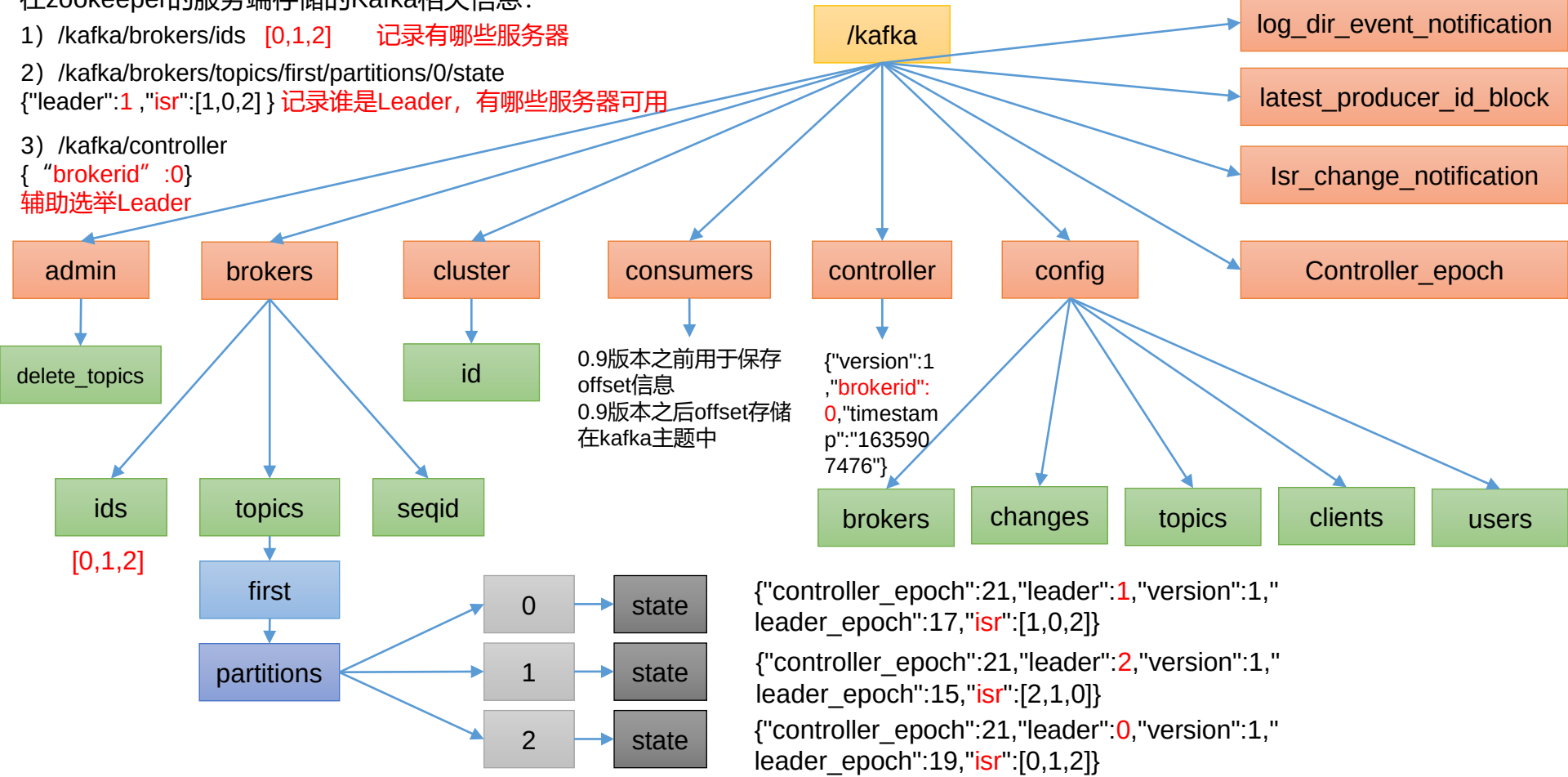
Zookeeper中存储的Kafka 信息

在zookeeper的服务端存储的Kafka相关信息：

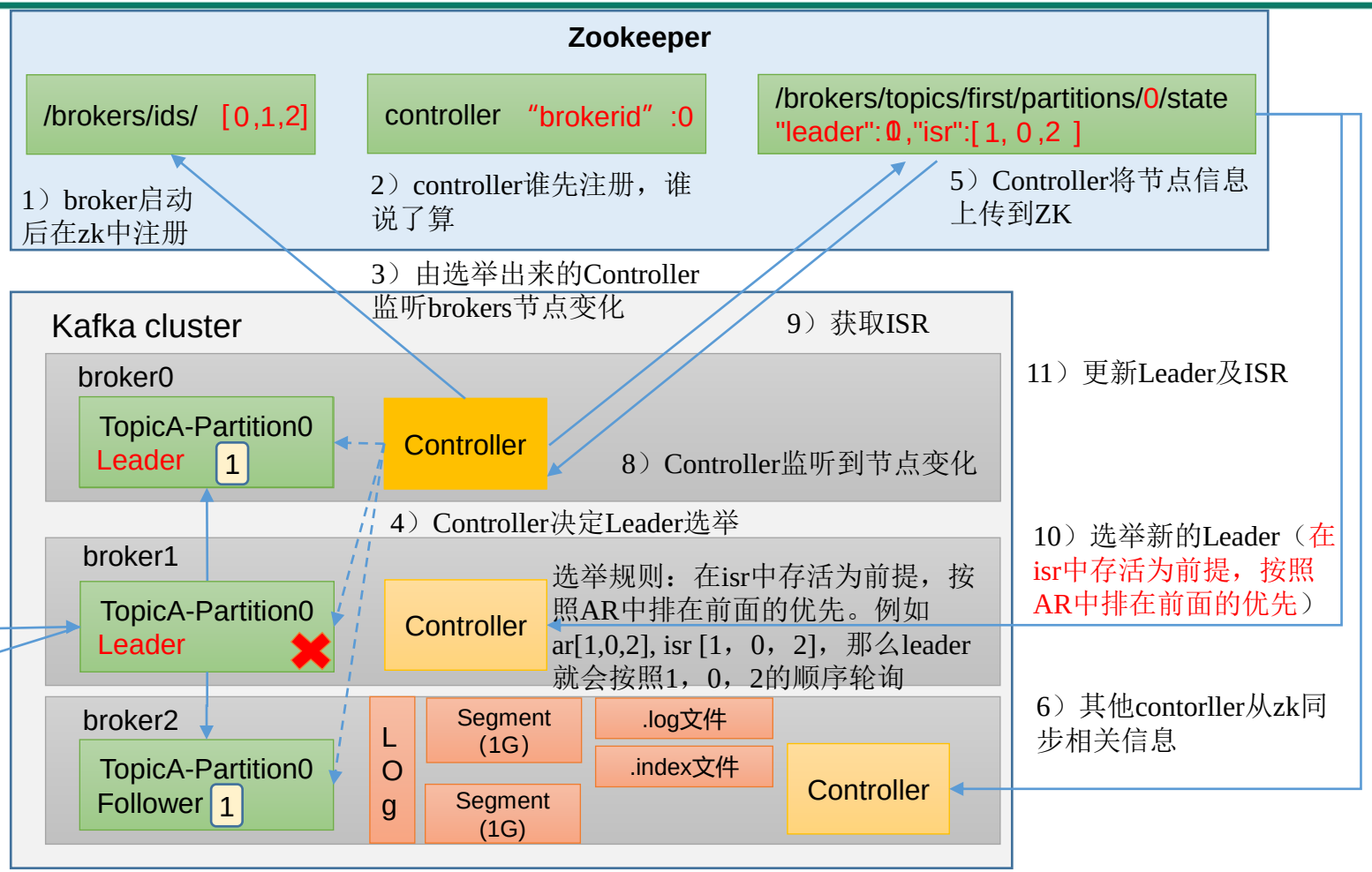
1）/kafka/brokers/ids [0,1,2] 记录有哪些服务器

2）/kafka/brokers/topics/first/partitions/0/state
{"leader":1 ,"isr":[1,0,2] } 记录谁是Leader，有哪些服务器可用

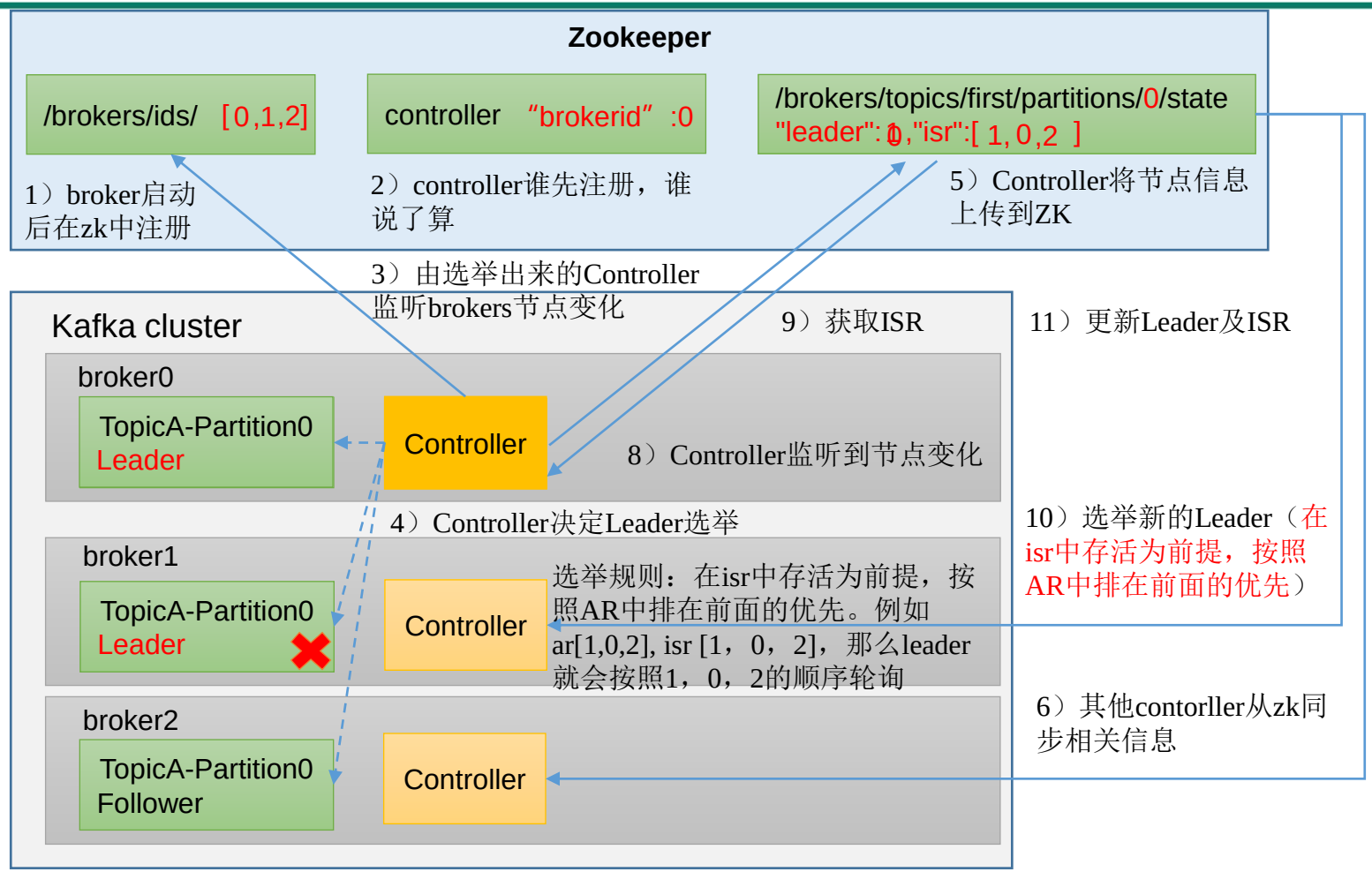
3）/kafka/controller
{ “brokerid” :0}
辅助选举Leader



Kafka Broker总体工作流程



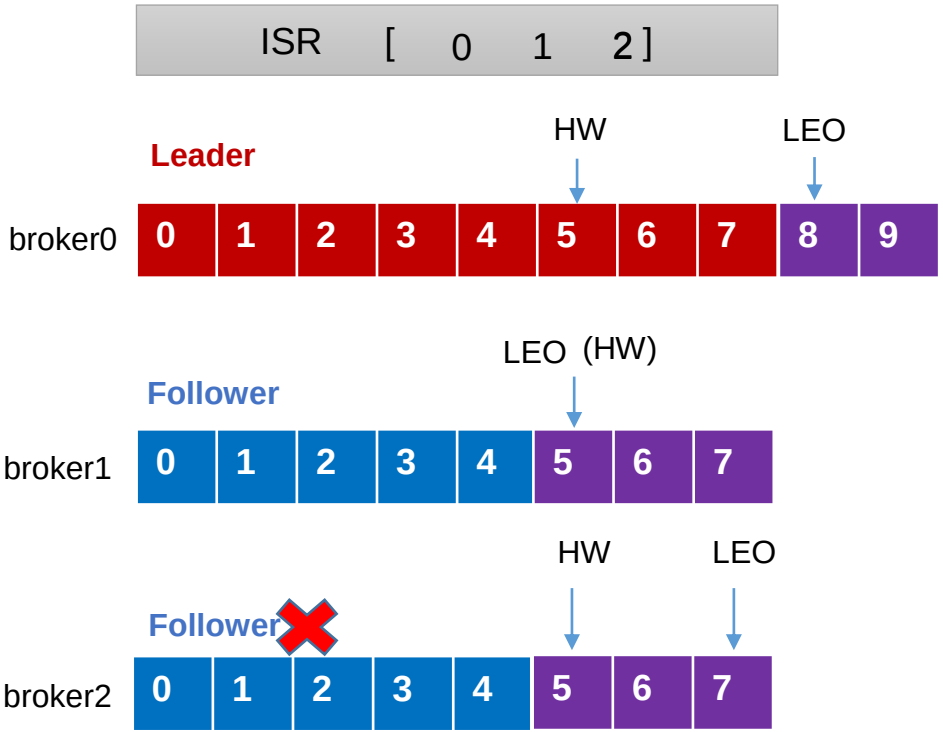
Leader选举流程


Zookeeper
/brokers/ids/ [0,1,2]
controller "brokerid" :0
/brokers/topics/first/partitions/0/state
"leader":1,"isr":[1,0,2]
1）broker启动后在zk中注册
2）controller谁先注册，谁说了算
5）Controller将节点信息上传到ZK
3）由选举出来的Controller监听brokers节点变化
AR：Kafka分区中的所有副本统称
Kafka cluster
9）获取ISR
11）更新Leader及ISR
broker0
TopicA-Partition0
Leader
Controller
8）Controller监听到节点变化
4）Controller决定Leader选举
10）选举新的Leader（在isr中存活为前提，按照AR中排在前面的优先）
broker1
TopicA-Partition0
Leader
Controller
选举规则：在isr中存活为前提，按照AR中排在前面的优先。例如ar[1,0,2], isr [1，0，2]，那么leader就会按照1，0，2的顺序轮询
7）假设Broker1中Leader挂了
broker2
TopicA-Partition0
Follower
Controller
6）其他contorller从zk同步相关信息

Follower故障处理细节

LEO（Log End Offset）：每个副本的最后一个offset，LEO其实就是最新的offset + 1。

HW（High Watermark）：所有副本中最小的LEO 。



1）Follower故障

（1） Follower发生故障后会被临时踢出ISR

（2） 这个期间Leader和Follower继续接收数据

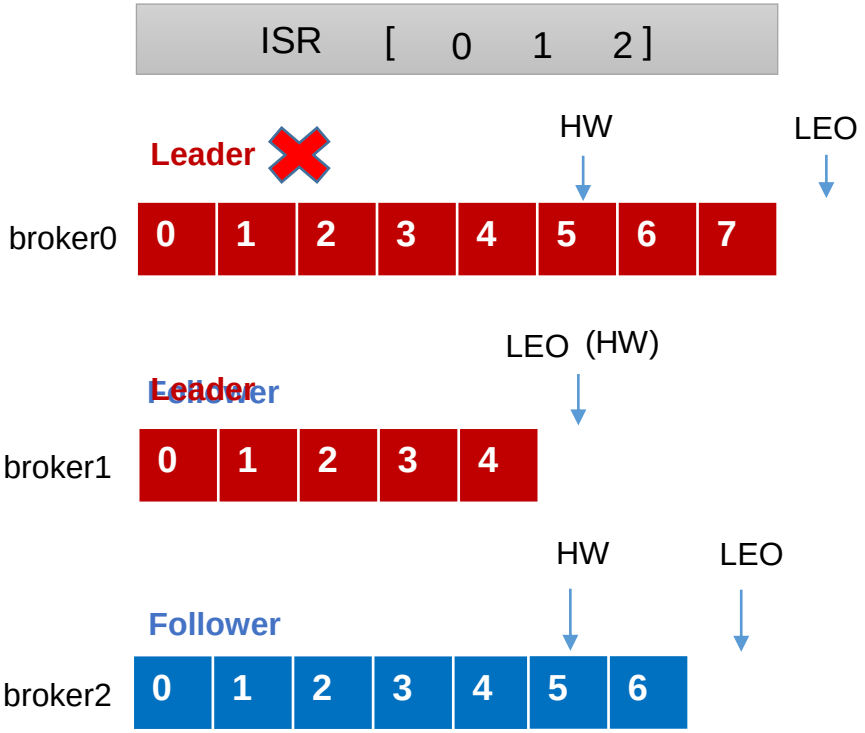
（3）待该Follower恢复后，Follower会读取本地磁盘记录的上次的HW，并将log文件高于HW的部分截取掉，从HW开始向Leader进行同步。

（4）等该**Follower的LEO大于等于该Partition的HW**，即Follower追上Leader之后，就可以重新加入ISR了。

Leader故障处理细节

LEO（Log End Offset）：每个副本的最后一个offset，LEO其实就是最新的offset + 1

HW（High Watermark）：所有副本中最小的LEO



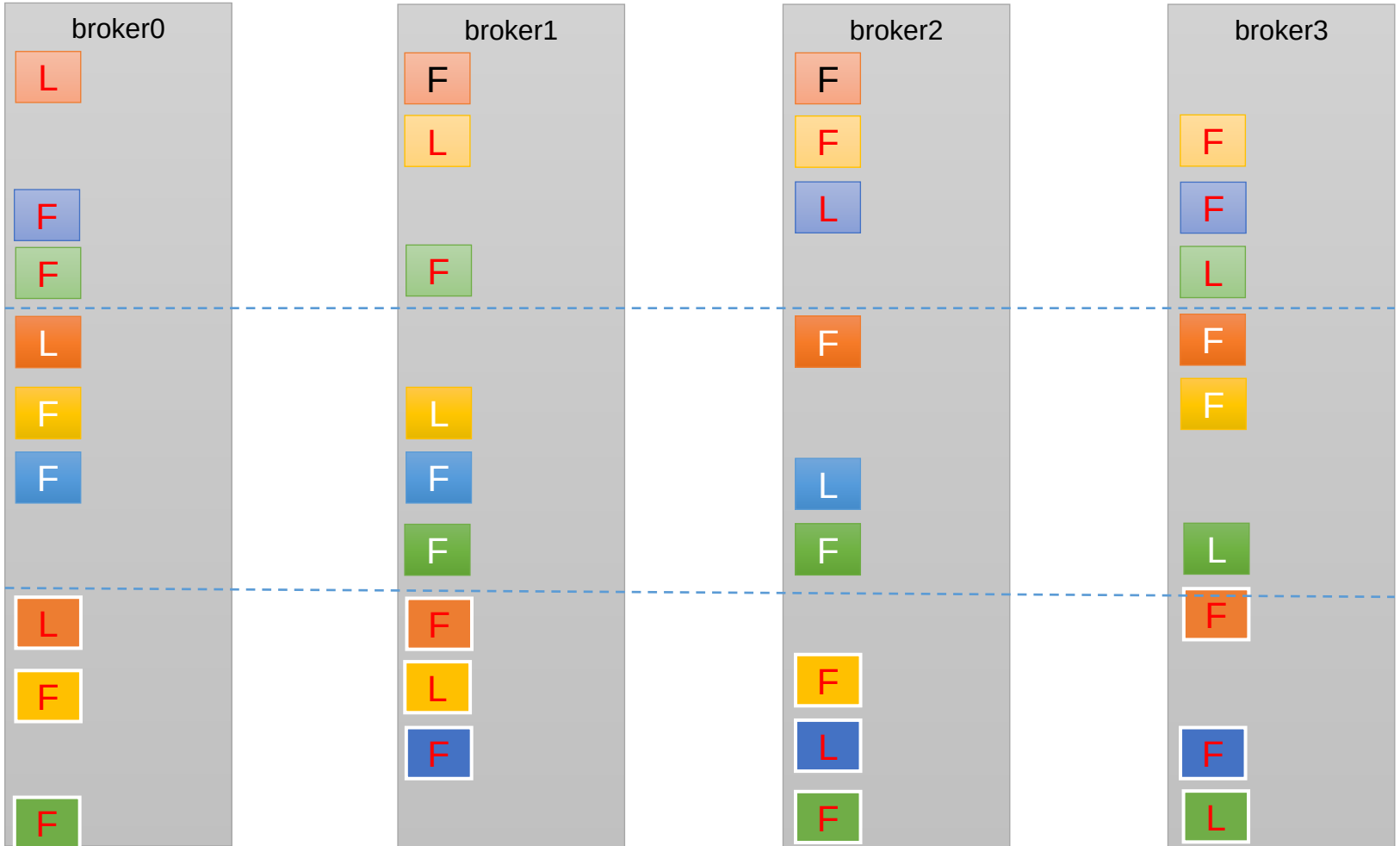
1）Leader故障

（1） Leader发生故障之后，会从ISR中选出一个新的Leader

（2）为保证多个副本之间的数据一致性，其余的Follower会先将各自的log文件高于HW的部分截掉，然后从新的Leader同步数据。

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

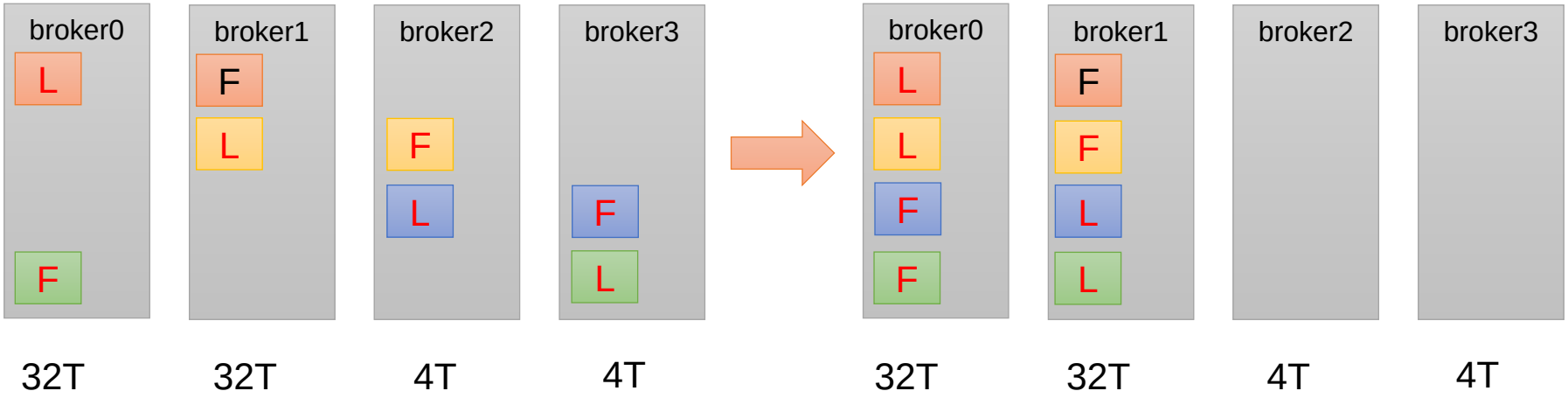
分区副本分配


broker0
broker1
broker2
broker3
L
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F
F
L
F
F

生产经验——手动调整分区副本存储

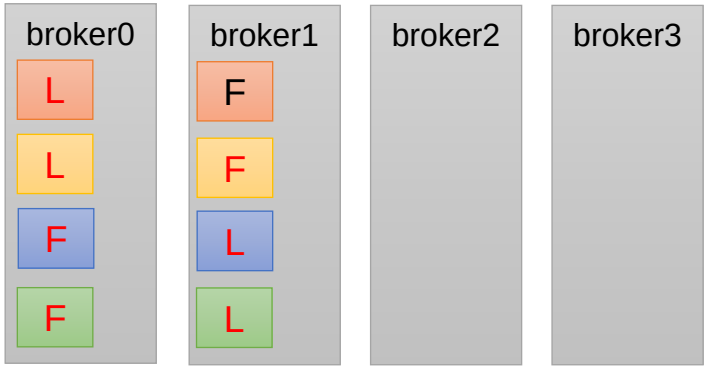
在生产环境中，每台服务器的配置和性能不一致，但是Kafka只会根据自己的代码规则创建对应的分区副本，就会导致个别服务器存储压力较大。所有需要手动调整分区副本的存储。

需求：创建一个新的topic，4个分区，两个副本，名称为three。将该topic的所有副本都存储到broker0和broker1两台服务器上。



Leader Partition自动平衡

正常情况下，Kafka本身会自动把Leader Partition均匀分散在各个机器上，来保证每台机器的读写吞吐量都是均匀的。但是如果某些broker宕机，会导致Leader Partition过于集中在其他少部分几台broker上，这会导致少数几台broker的读写请求压力过高，其他宕机的broker重启之后都是follower partition，读写请求很低，造成集群负载不均衡。



- auto.leader.rebalance.enable，默认是true。自动Leader Partition 平衡
- leader.imbalance.per.broker.percentage，默认是10%。每个broker允许的不平衡的leader的比率。如果每个broker超过了这个值，控制器会触发leader的平衡。
- leader.imbalance.check.interval.seconds，默认值300秒。检查leader负载是否平衡的间隔时间。

下面拿一个主题举例说明，假设集群只有一个主题如下图所示：

```
Topic: atguigu1 Partition: 0    Leader: 0   Replicas: 3,0,2,1   Isr: 3,0,2,1
Topic: atguigu1 Partition: 1    Leader: 1   Replicas: 1,2,3,0   Isr: 1,2,3,0
Topic: atguigu1 Partition: 2    Leader: 2   Replicas: 0,3,1,2   Isr: 0,3,1,2
Topic: atguigu1 Partition: 3    Leader: 3   Replicas: 2,1,0,3   Isr: 2,1,0,3
```

针对broker0节点，分区2的AR优先副本是0节点，但是0节点却不是Leader节点，所以不平衡数加1，AR副本总数是4
所以broker0节点不平衡率为1/4>10%，需要再平衡。

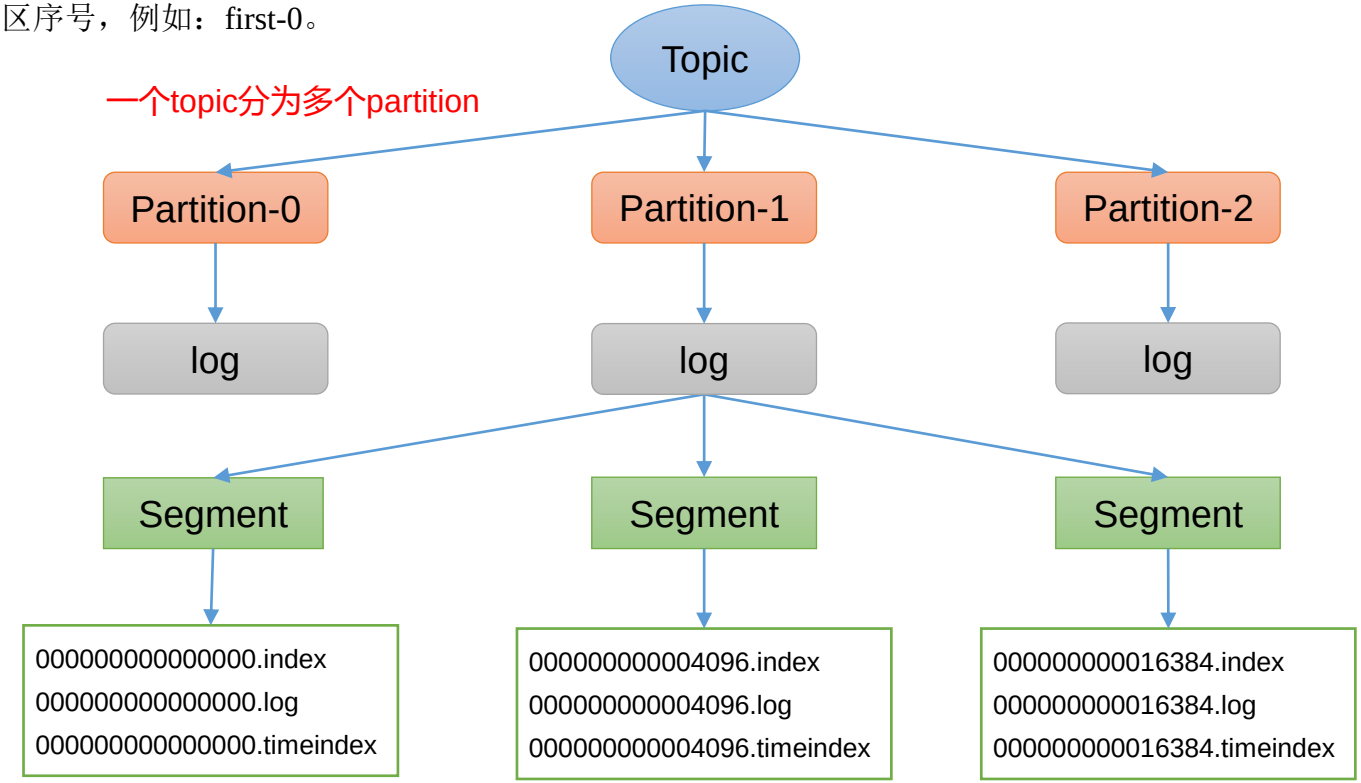
broker2和broker3节点和broker0不平衡率一样，需要再平衡。
Broker1的不平衡数为0，不需要再平衡

Kafka文件存储机制

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生产的数据会被不断追加到该log文件末端，为防止log文件过大导致数据定位效率低下，Kafka采取了分片和索引机制，将每个partition分为多个segment。每个segment包括：“.index”文件、“.log”文件和.timeindex等文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic名称+分区序号，例如：first-0。

一个topic分为多个partition

一个partition分为多个segment



.log 日志文件

.index 偏移量索引文件

.timeindex 时间戳索引文件

其他文件

说明：index和log文件以当前segment的第一条消息的offset命名。

Log文件和Index文件详解

Segement-0 [offset:0-521] 00000000000000000000.index 00000000000000000000.log

Segement-1 [offset:522-1004] 00000000000000000522.index 00000000000000000522.log

绝对Offset	相对Offset	Position
587	65	6410
639	117	13795
691	169	21060
743	221	28367

RecordBatch[baseOffset~lastOffset]		
baseOffset	lastOffset	position
522	522	0
523	523	200
524	536	819
537	562	3092
563	587	6410
588	613	10090
614	639	13795
640	665	17481
666	691	21060
692	717	24781
728	743	28367

1

2

3

4

如何在log文件中定位到offset=600的Record?

1.根据目标offset定位Segment文件

2.找到小于等于目标offset的最大offset对应的索引项

3.定位到log文件

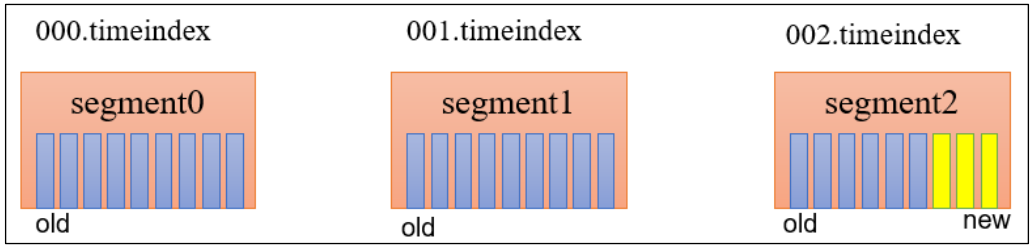
4.向下遍历找到目标Record

注意:

1.index为稀疏索引，大约每往log文件写入4kb数据，会往index文件写入一条索引。参数log.index.interval.bytes默认4kb。

2.Index文件中保存的offset为相对offset，这样能确保offset的值所占空间不会过大，因此能将offset的值控制在固定大小

Segement-2 [offset:1005-] 00000000000000001005.index 00000000000000001005.log


000.timeindex
segment0
old
001.timeindex
segment1
old
002.timeindex
segment2
old
new

compact日志压缩

compact日志压缩：对于相同key的不同value值，只保留最后一个版本。

- log.cleanup.policy = compact 所有数据启用压缩策略

压缩之前的数据

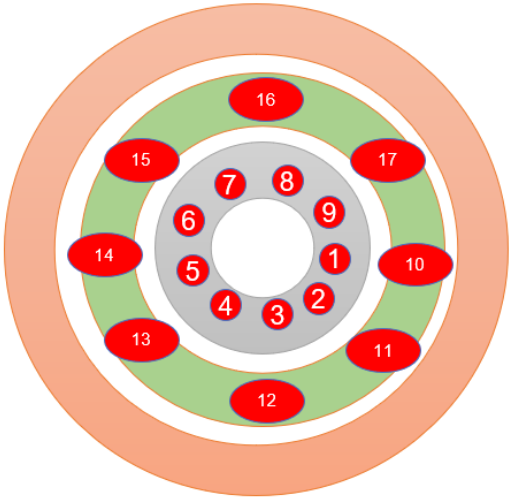
Offset	0	1	2	3	4	5	6	7	8
key	K1	K2	K1	K1	K3	K4	K5	K5	K2
value	V1	V2	V3	V4	V5	V6	V7	V8	V9

压缩之后的数据

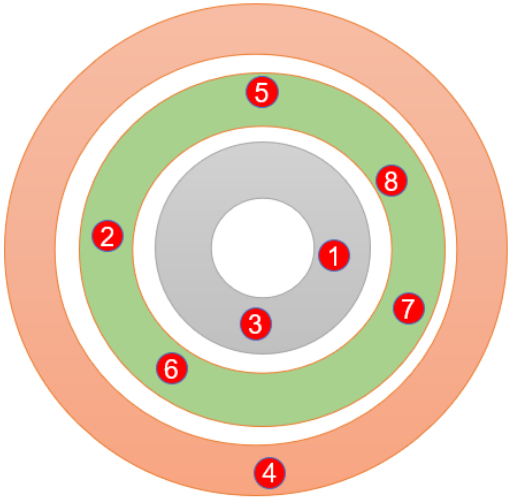
Offset	3	4	5	7	8
keys	k1	K3	K4	K5	K2
values	V4	V5	V6	V8	V9

压缩后的offset可能是不连续的，比如上图中没有6，当从这些offset消费消息时，将会拿到比这个offset大的offset对应的消息，实际上会拿到offset为7的消息，并从这个位置开始消费。

这种策略只适合特殊场景，比如消息的key是用户ID，value是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。



顺序写磁盘能到600M/s



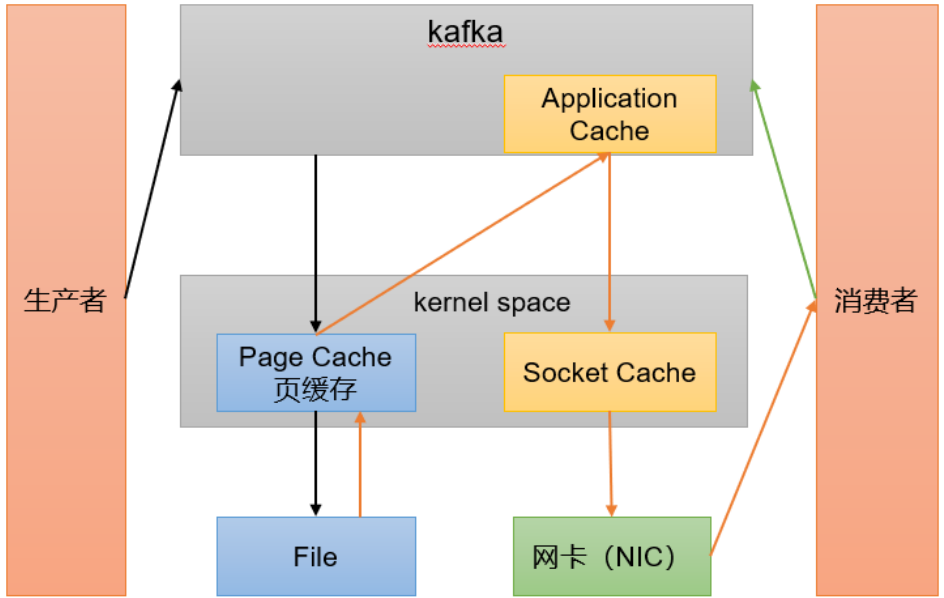
随机写磁盘能到100K/s

页缓存 + 零拷贝技术

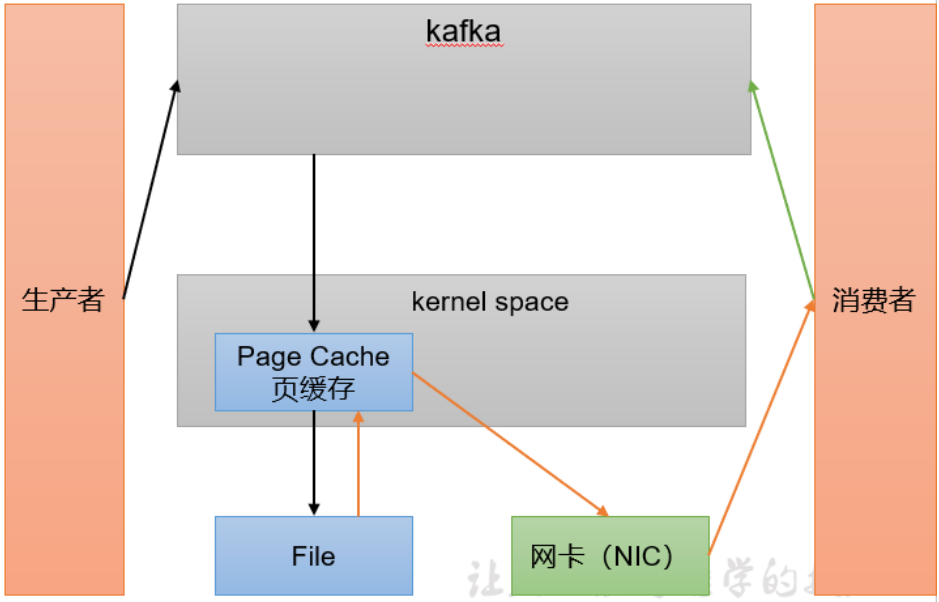
零拷贝：Kafka的数据加工处理操作交由Kafka生产者和Kafka消费者处理。Kafka Broker应用层不关心存储的数据，所以就不用走应用层，传输效率高。

PageCache页缓存：Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache。当读操作发生时，先从PageCache中查找，如果找不到，再去磁盘中读取。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。

非零拷贝工作流程（假设）



零拷贝工作流程（实际）



Kafka 消费方式

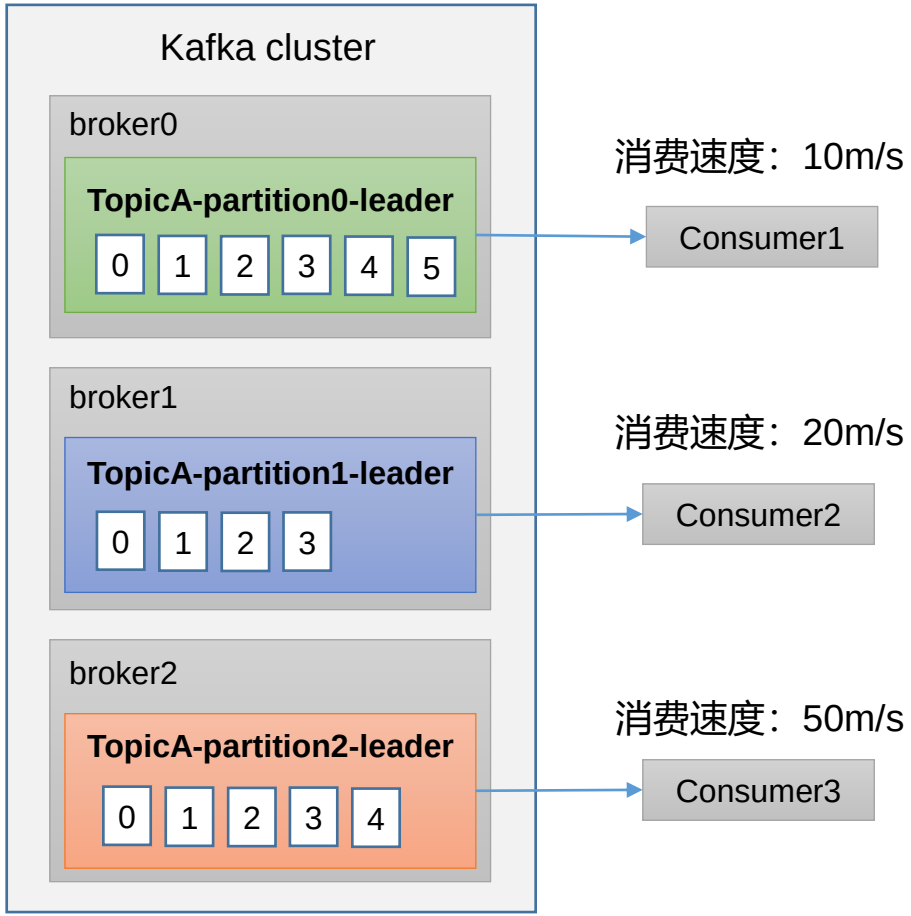
- **pull（拉）模式：**

consumer采用从broker中主动拉取数据。**Kafka采用这种方式。**

- **push（推）模式：**

Kafka没有采用这种方式，因为由broker决定消息发送速率，很难适应所有消费者的消费速率。例如推送的速度是50m/s，Consumer1、Consumer2就来不及处理消息。

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。



Kafka 消费者总体工作流程

Zookeeper

Kafka cluster

offset

每个消费者的offset由消费者提交到系统主题保存

broker0

TopicA-partition0-leader

0 1 2 3 4 5

TopicA-partition1-follower

0 1 2 3

__consumer_offsets

broker1

TopicA-partition1-leader

0 1 2 3

TopicA-partition2-follower

0 1 2 3 4

__consumer_offsets

broker2

TopicA-partition2-leader

0 1 2 3 4

TopicA-partition0-follower

0 1 2 3 4 5

__consumer_offsets

Producer

Consumer

每个分区的数据只能由消费者组中一个消费者消费

group

Consumer

Consumer

Consumer

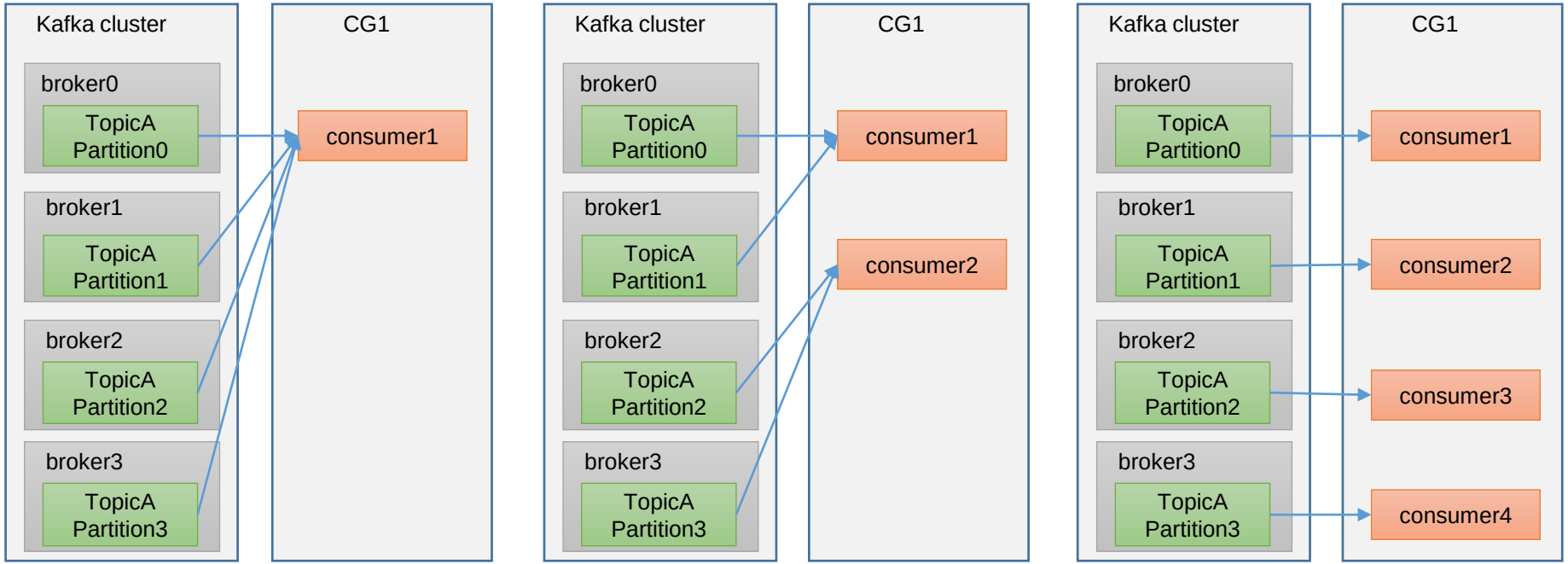
一个消费者可以消费多个分区数据

Consumer

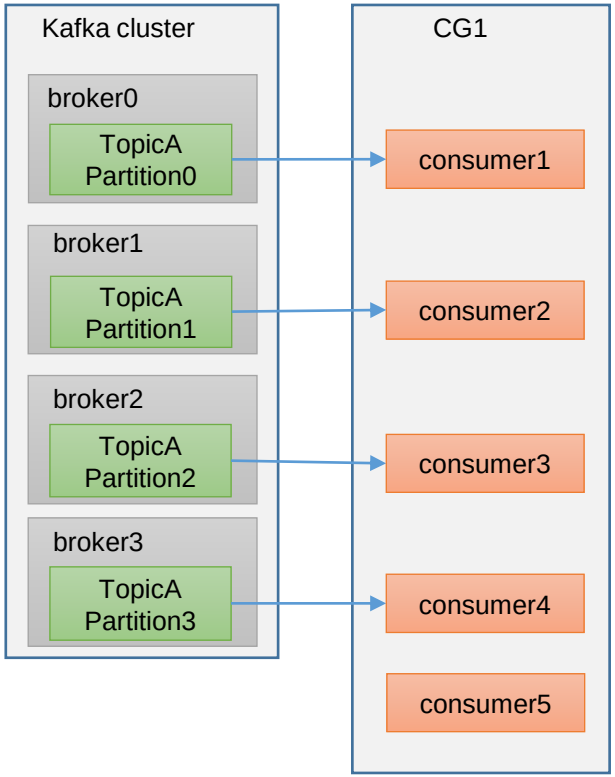
消费者组

Consumer Group（CG）：消费者组，由多个consumer组成。形成一个消费者组的条件，是所有消费者的groupid相同。

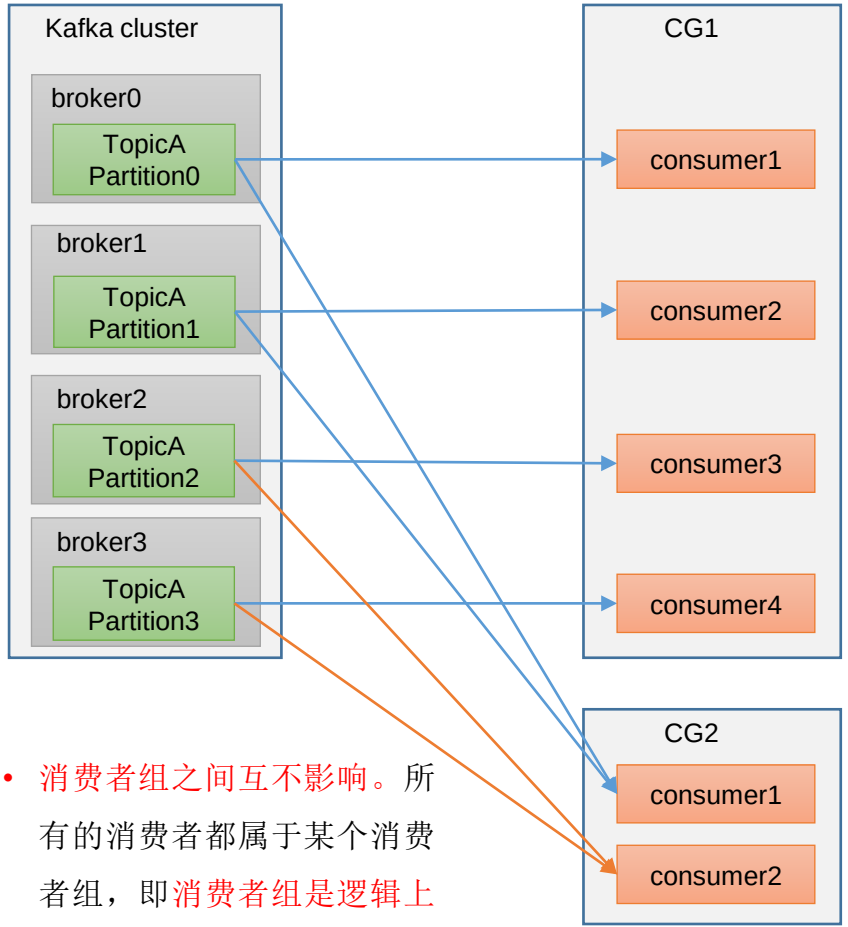
- 消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费。
- 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。



消费者组



- 如果向消费组中添加更多的消费者，超过主题分区数量，则有一部分消费者就会闲置，不会接收任何消息。



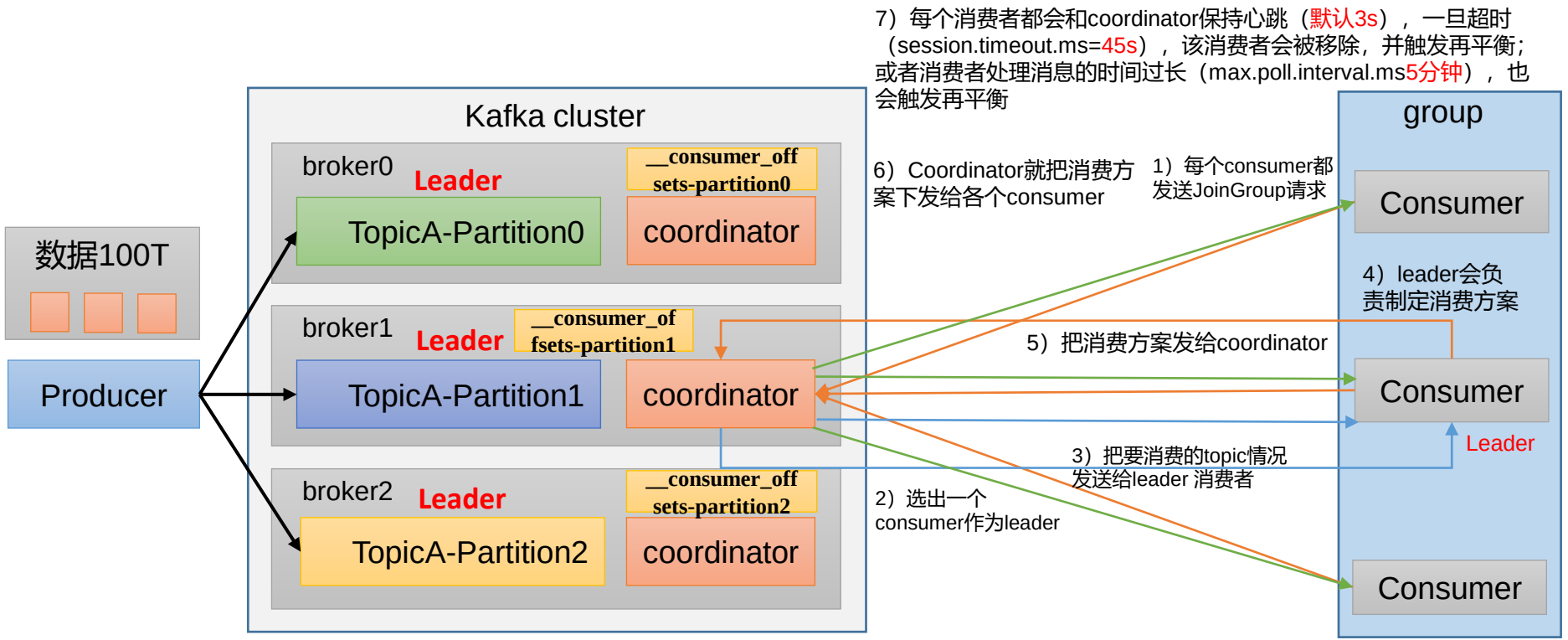
- 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

消费者组初始化流程

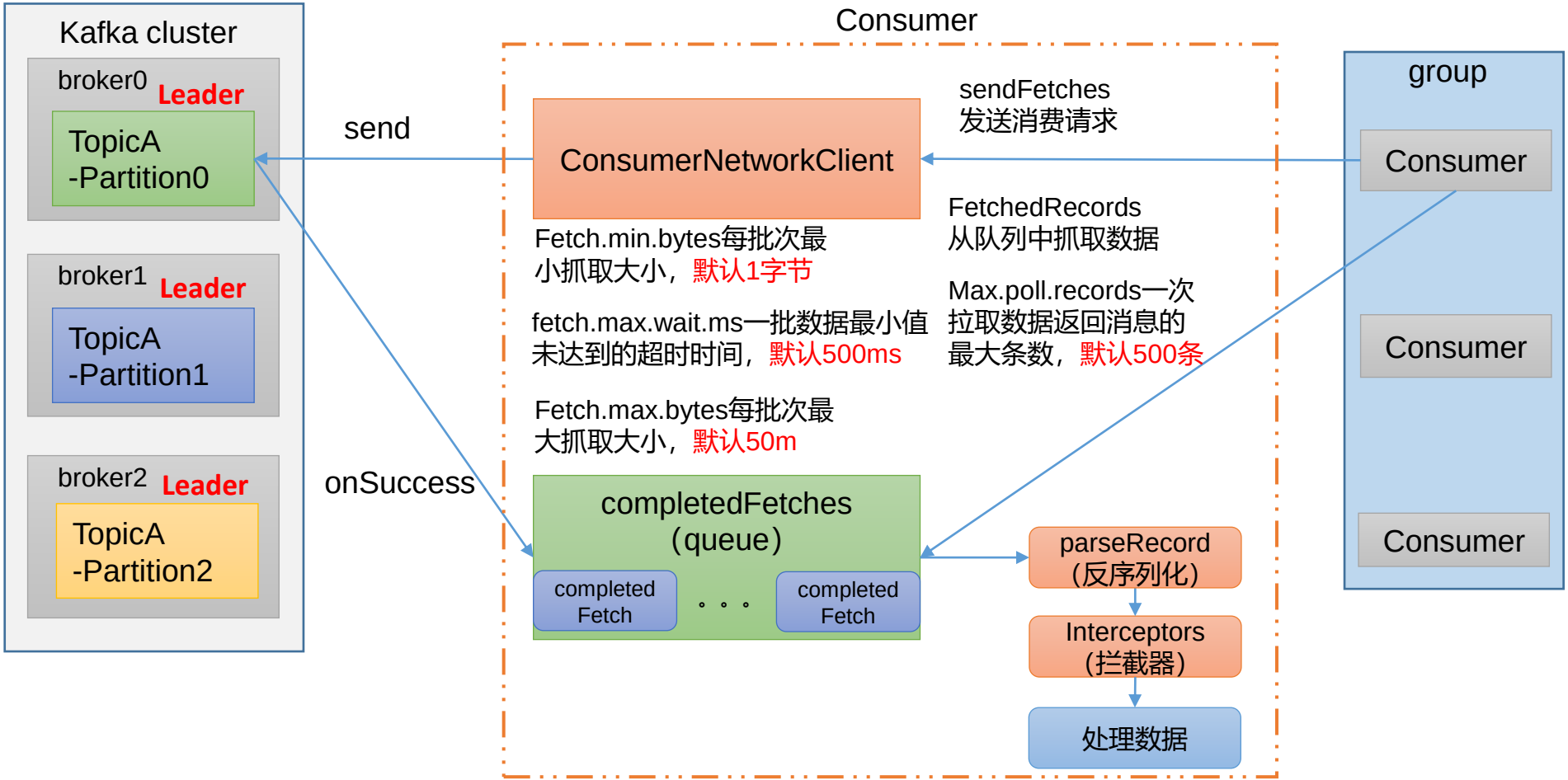
1、coordinator：辅助实现消费者组的初始化和分区的分配。

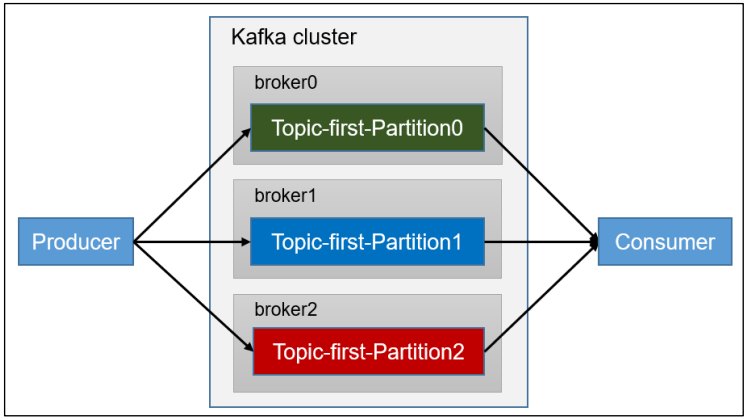
coordinator节点选择 = groupid的hashcode值 % 50（__consumer_offsets的分区数量）

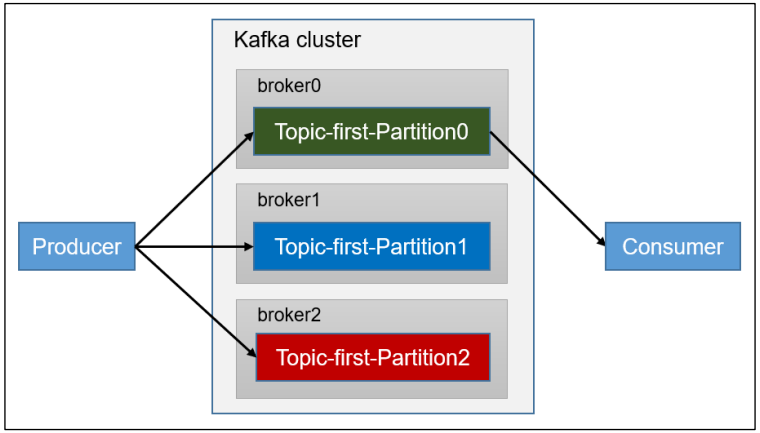
例如： groupid的hashcode值 = 1，1% 50 = 1，那么__consumer_offsets 主题的1号分区，在哪个broker上，就选择这个节点的coordinator作为这个消费者组的老大。消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset。

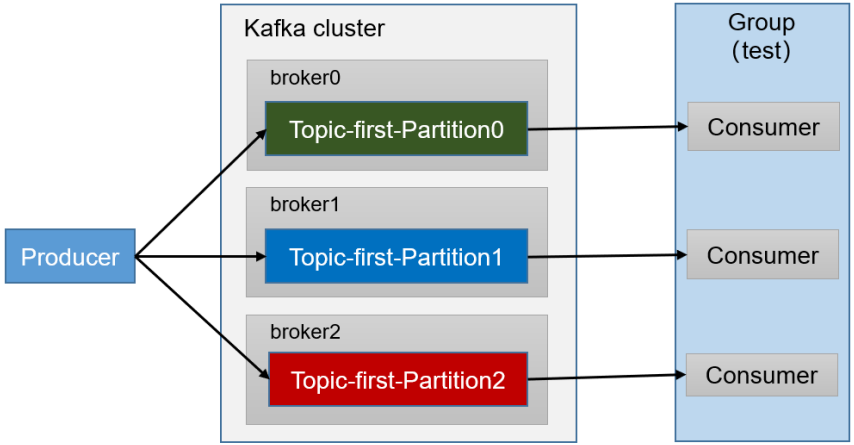


消费者组详细消费流程


Kafka cluster
broker0
Leader
TopicA
-Partition0
broker1
Leader
TopicA
-Partition1
broker2
Leader
TopicA
-Partition2
send
onSuccess
Consumer
ConsumerNetworkClient
sendFetches
发送消费请求
FetchedRecords
从队列中抓取数据
Fetch.min.bytes每批次最小抓取大小，默认1字节
fetch.max.wait.ms一批数据最小值未达到的超时时间，默认500ms
Max.poll.records一次拉取数据返回消息的最大条数，默认500条
Fetch.max.bytes每批次最大抓取大小，默认50m
completedFetches
(queue)
completed Fetch
。。。
completed Fetch
parseRecord
(反序列化)
Interceptors
(拦截器)
处理数据
group
Consumer
Consumer
Consumer


Kafka cluster
broker0
Topic-first-Partition0
broker1
Topic-first-Partition1
broker2
Topic-first-Partition2
Producer
Consumer

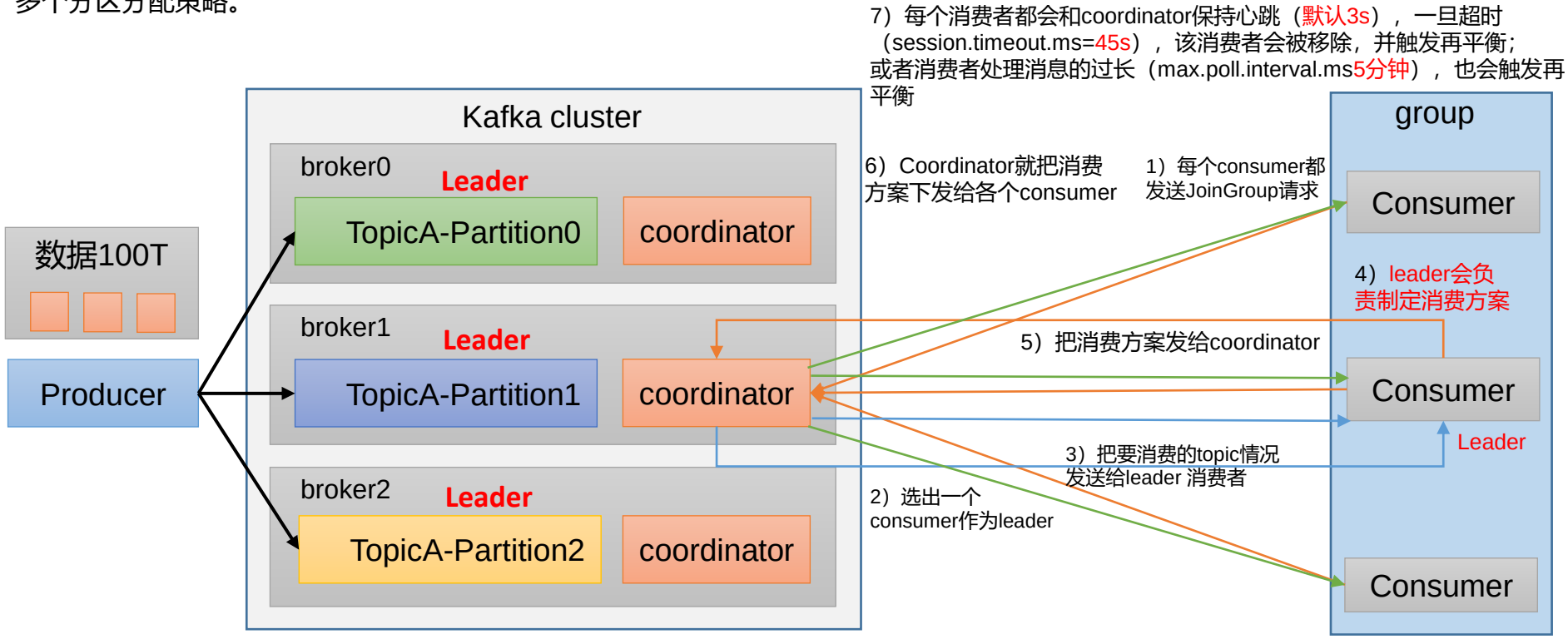

Kafka cluster
broker0
Topic-first-Partition0
broker1
Topic-first-Partition1
broker2
Topic-first-Partition2
Producer
Consumer


Kafka cluster
broker0
Topic-first-Partition0
broker1
Topic-first-Partition1
broker2
Topic-first-Partition2
Producer
Group
(test)
Consumer
Consumer
Consumer

生产经验——分区的分配以及再平衡

1、一个consumer group中有多个consumer组成，一个 topic有多个partition组成，现在的问题是，到底由哪个consumer来消费哪个partition的数据。

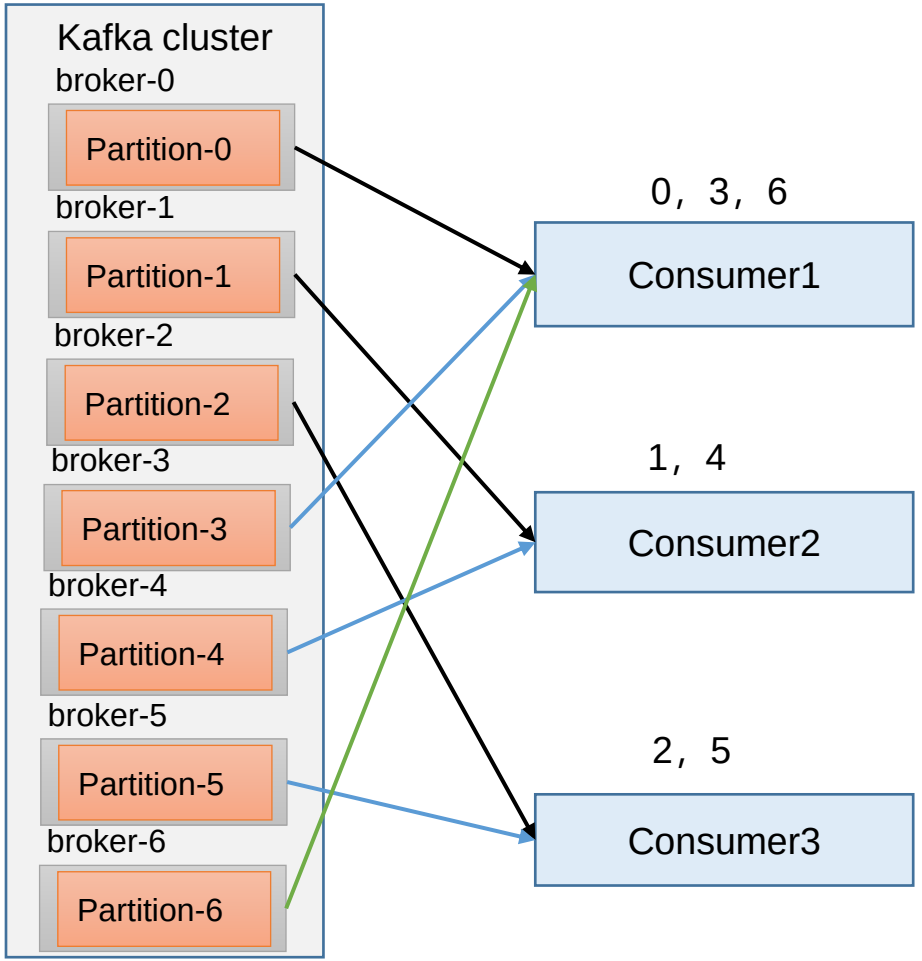
2、Kafka有四种主流的分区分配策略： Range、RoundRobin、Sticky、CooperativeSticky。
可以通过配置参数partition.assignment.strategy，修改分区的分配策略。默认策略是Range + CooperativeSticky。Kafka可以同时使用多个分区分配策略。



分区分配策略之RoundRobin

RoundRobin 针对集群中所有Topic而言。

RoundRobin 轮询分区策略，是把所有的 partition 和所有的 consumer 都列出来，然后按照 hashcode 进行排序，最后通过轮询算法来分配 partition 给到各个消费者。



offset的默认维护位置

从0.9版本开始，consumer默认将offset保存在Kafka一个内置的topic中，该topic为__consumer_offsets

Kafka0.9版本之前，consumer默认将offset保存在Zookeeper中

Kafka cluster

offset

broker0

TopicA-partition0-leader

0 1 2 3 4 5

2

__consumer_offsets

broker1

TopicA-partition1-leader

0 1 2 3

__consumer_offsets

broker2

TopicA-partition2-leader

0 1 2 3 4

__consumer_offsets

Producer

group

Consumer

Consumer

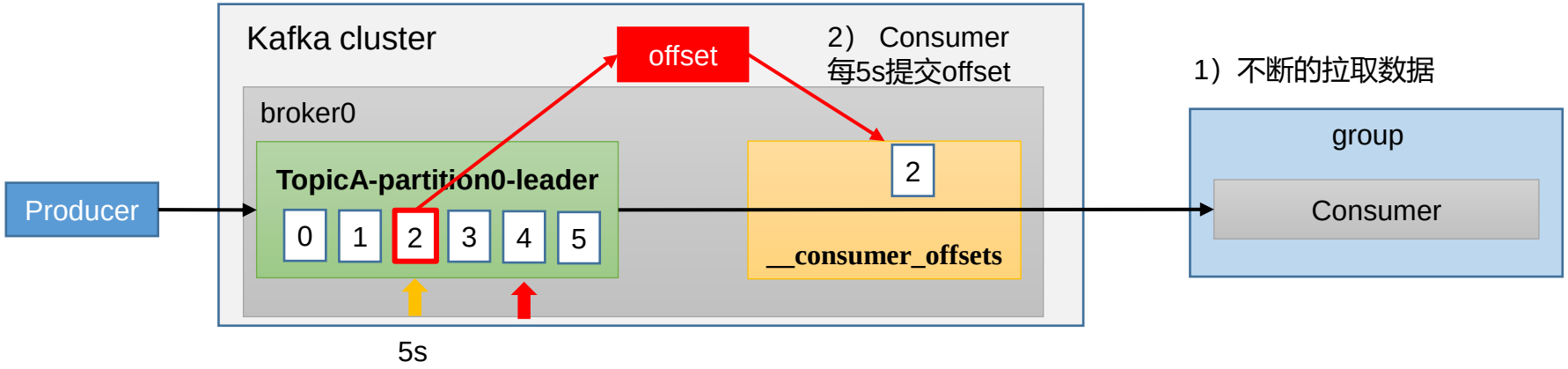
Consumer

自动提交offset

为了使我们能够专注于自己的业务逻辑，Kafka提供了自动提交offset的功能。

自动提交offset的相关参数：

- **enable.auto.commit**：是否开启自动提交offset功能，默认是true
- **auto.commit.interval.ms**：自动提交offset的时间间隔，默认是5s

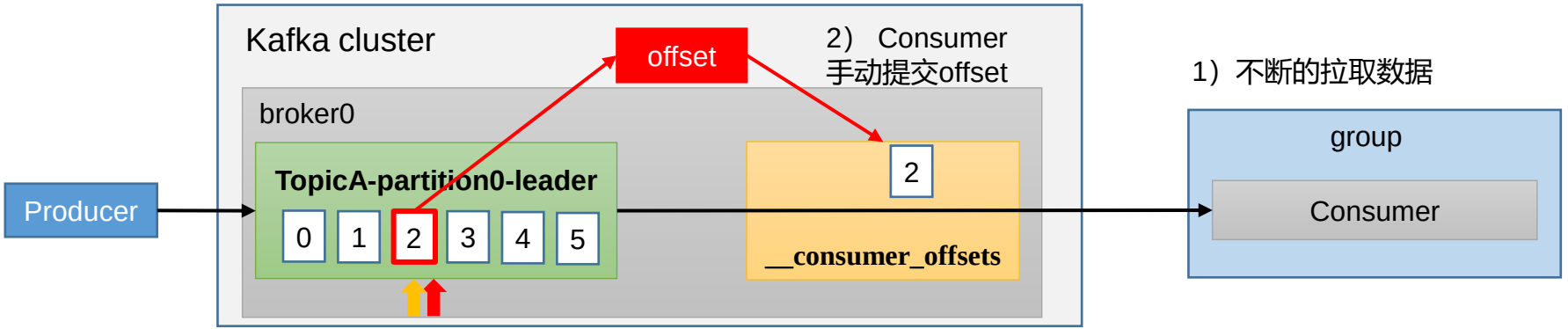


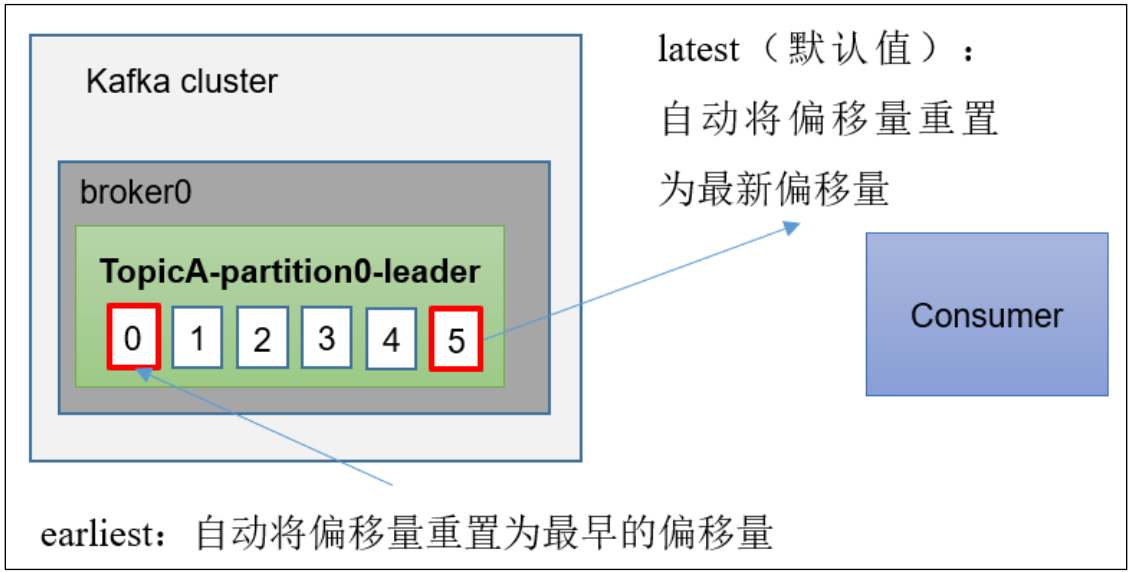
手动提交offset

虽然自动提交offset十分简单便利，但由于其是基于时间提交的，开发人员难以把握offset提交的时机。因此Kafka还提供了手动提交offset的API。

手动提交offset的方法有两种：分别是commitSync（同步提交）和commitAsync（异步提交）。两者的相同点是，都会将本次提交的一批数据最高的偏移量提交；不同点是，同步提交阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而异步提交则没有失败重试机制，故有可能提交失败。

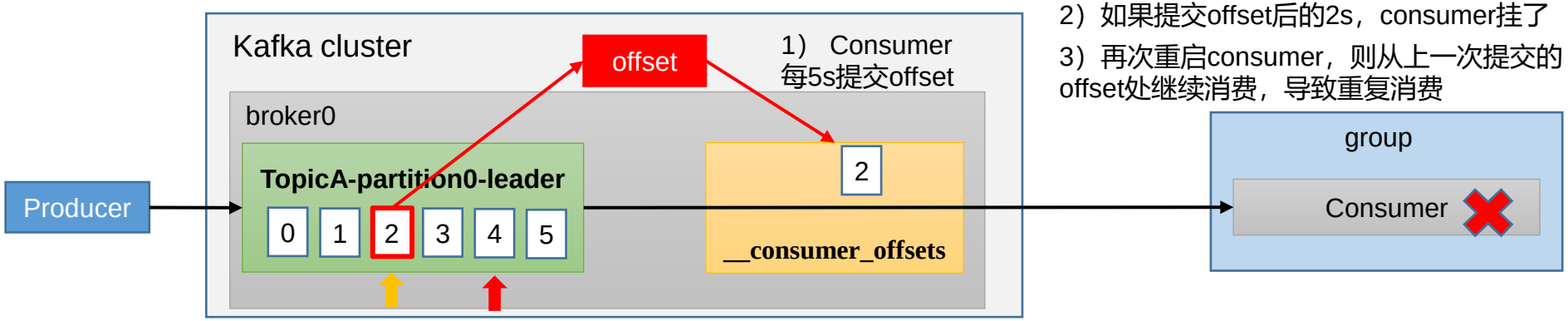
- commitSync（同步提交）：必须等待offset提交完毕，再去消费下一批数据。
- commitAsync（异步提交）：发送完提交offset请求后，就开始消费下一批数据了。



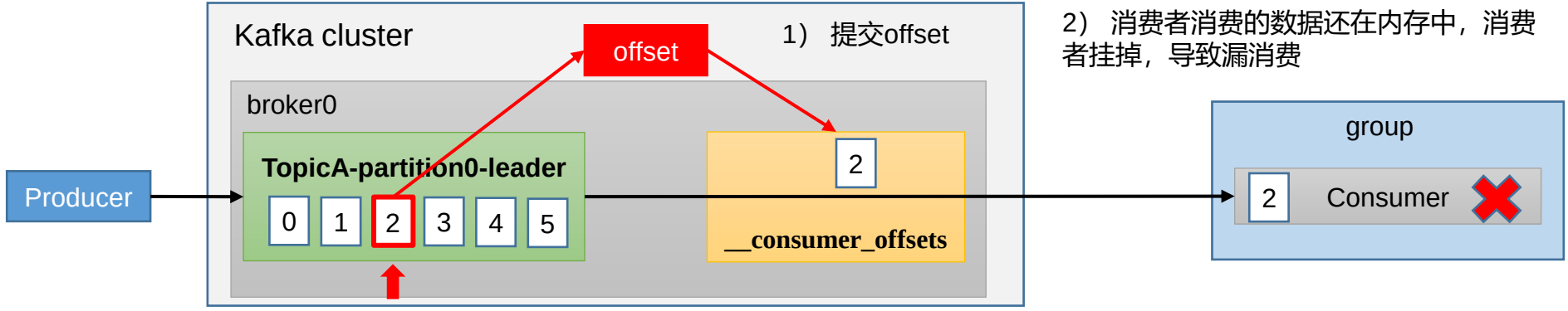

Kafka cluster
broker0
TopicA-partition0-leader
0
1
2
3
4
5
latest（默认值）：
自动将偏移量重置
为最新偏移量
Consumer
earliest：自动将偏移量重置为最早的偏移量

重复消费与漏消费

（1）场景1：重复消费。自动提交offset引起。

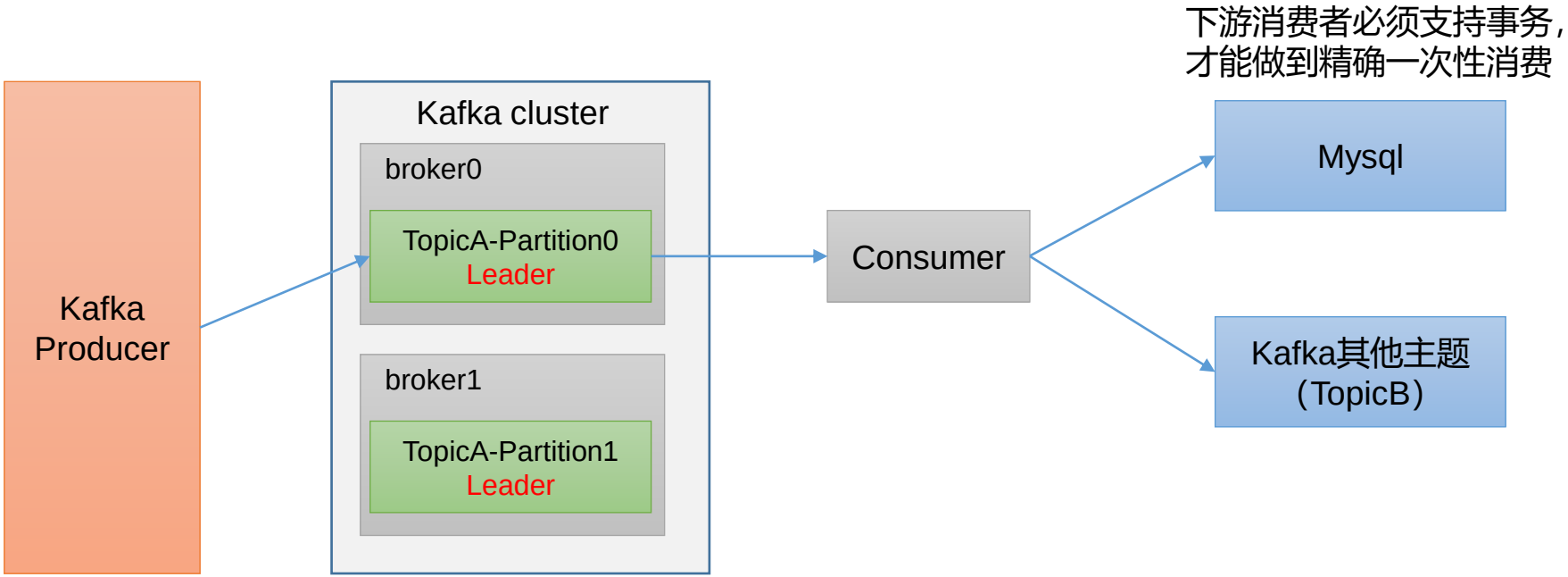


（2）场景1：漏消费。设置offset为手动提交，当offset被提交时，数据还在内存中未落盘，此时刚好消费者线程被kill掉，那么offset已经提交，但是数据未处理，导致这部分内存中的数据丢失。



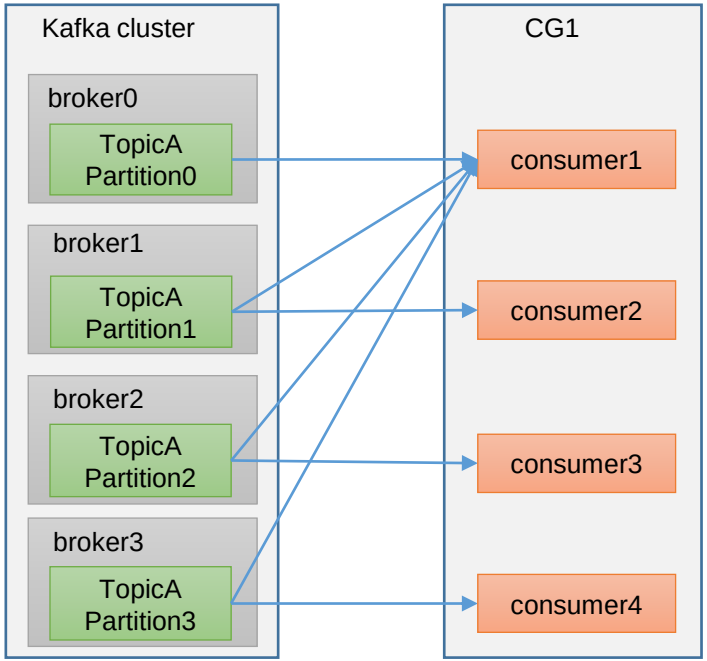
生产经验——消费者事务

如果想完成Consumer端的精准一次性消费，那么需要Kafka消费端将消费过程和提交offset过程做原子绑定。此时我们需要将Kafka的offset保存到支持事务的自定义介质（比如MySQL）。这部分知识会在后续项目部分涉及。

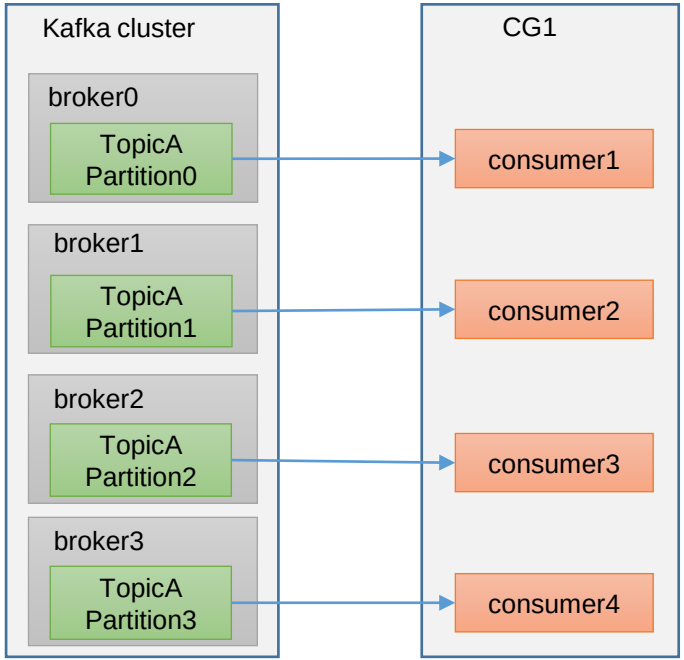


生产经验——数据积压（消费者如何提高吞吐量）

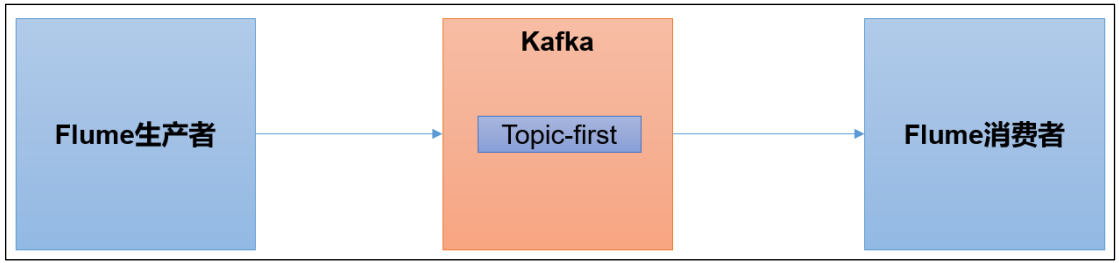
1）如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数 = 分区数。（两者缺一不可）



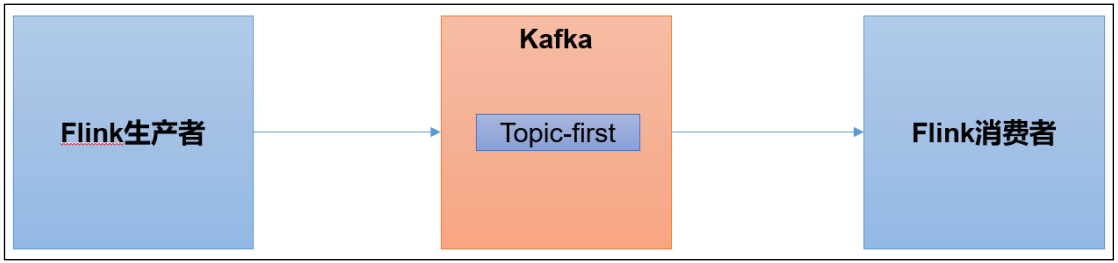
2）如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间 < 生产速度），使处理的数据小于生产的数据，也会造成数据积压。

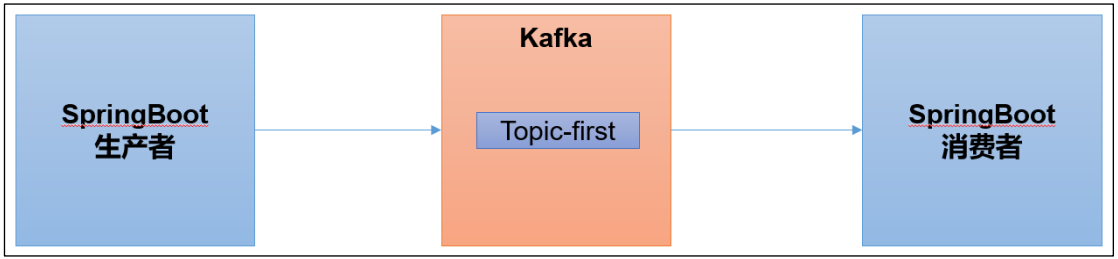


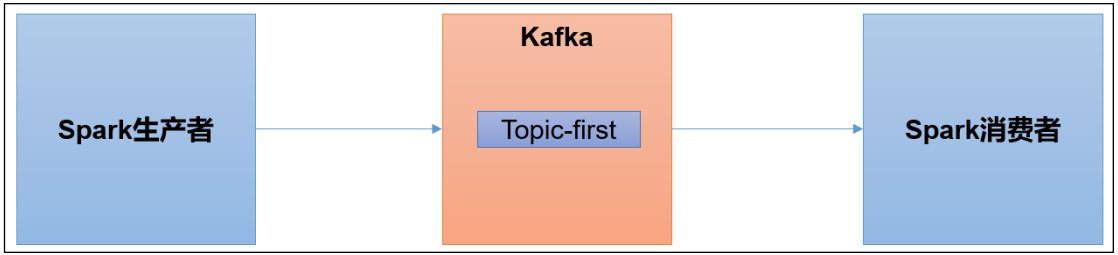
从一次最多拉取500条，调整为一次最多拉取1000条

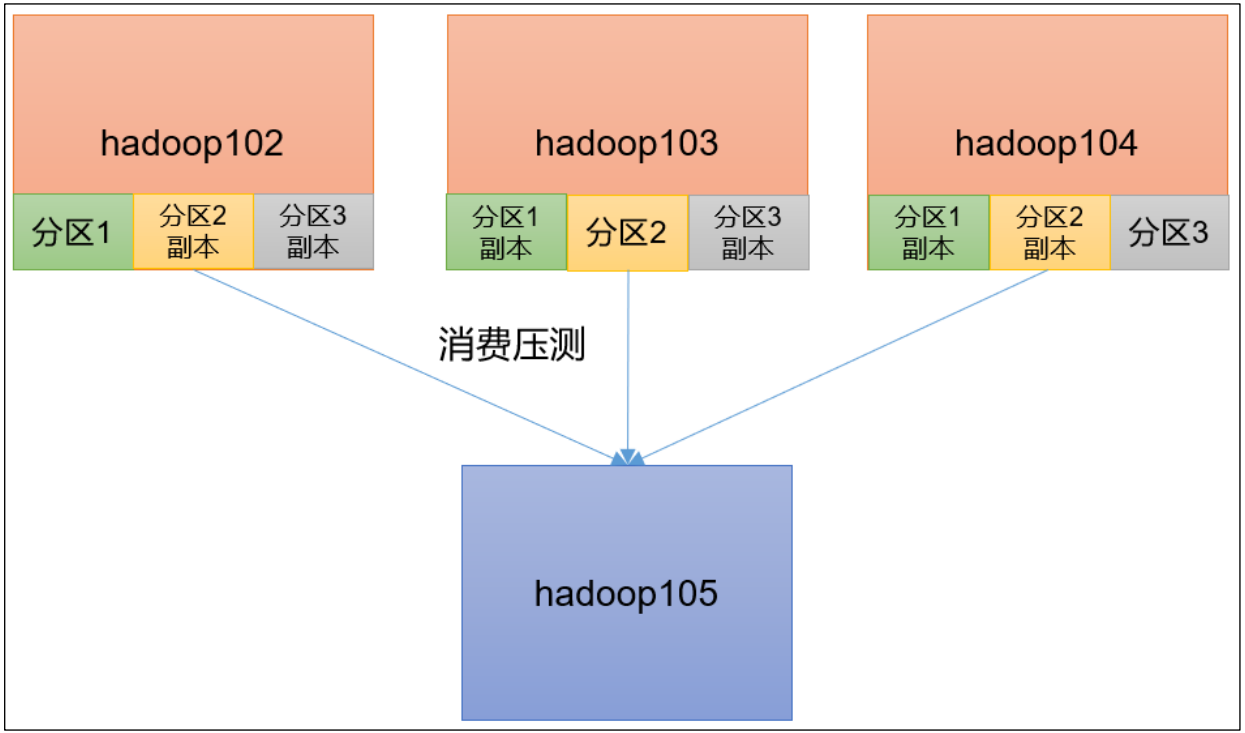

Flume生产者
Kafka
Topic-first
Flume消费者


Linux
/opt/module/
applog/app.*
taildirsource
Flume
kafkasink
memory channel
Kafka
Topic-first
Kafka
消费者

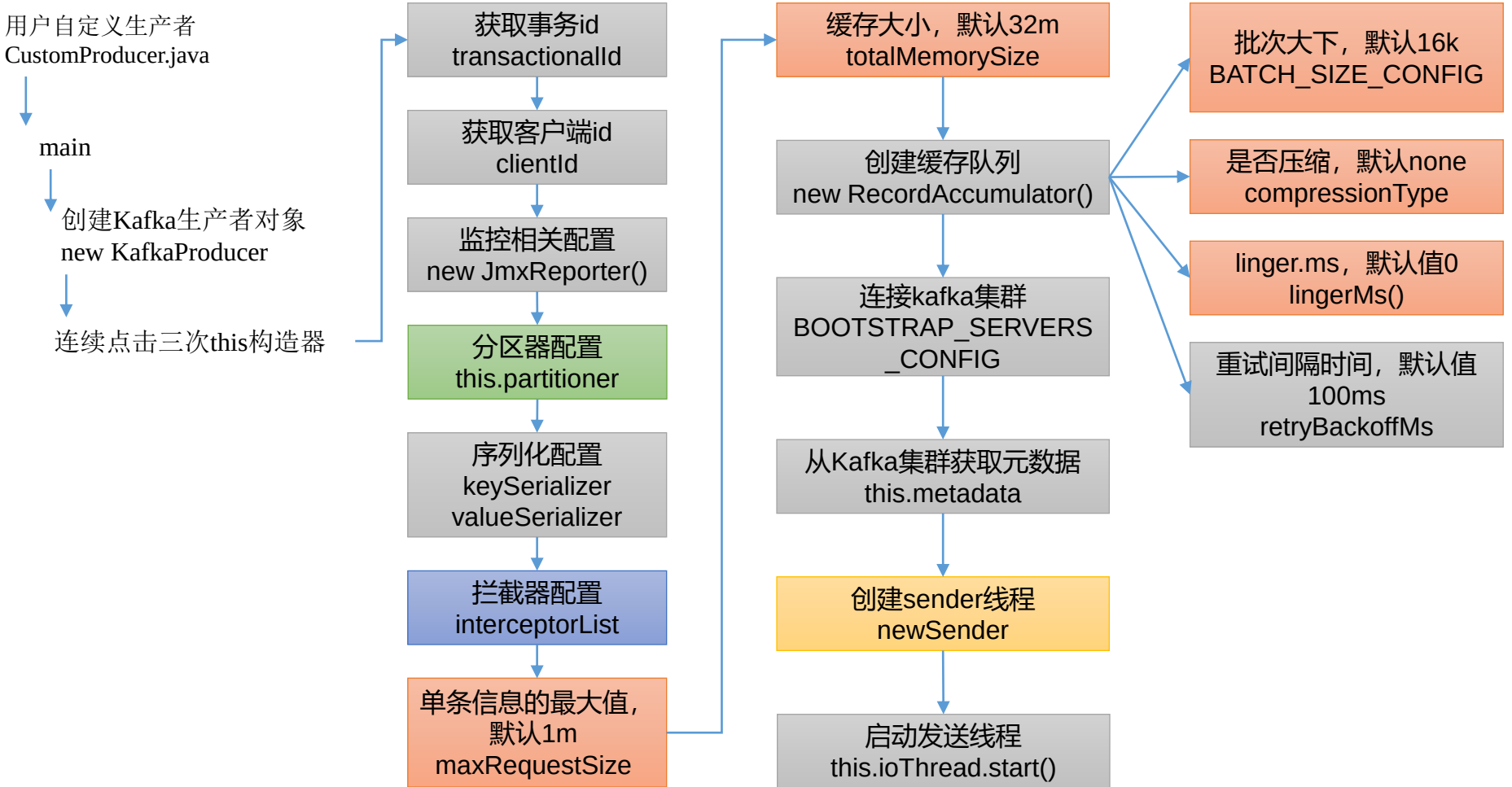

Flink生产者
Kafka
Topic-first
Flink消费者


SpringBoot
生产者
Kafka
Topic-first
SpringBoot
消费者


Spark生产者
Kafka
Topic-first
Spark消费者


hadoop102
分区1
分区2
副本
分区3
副本
hadoop103
分区1
副本
分区2
分区3
副本
hadoop104
分区1
副本
分区2
副本
分区3
消费压测
hadoop105

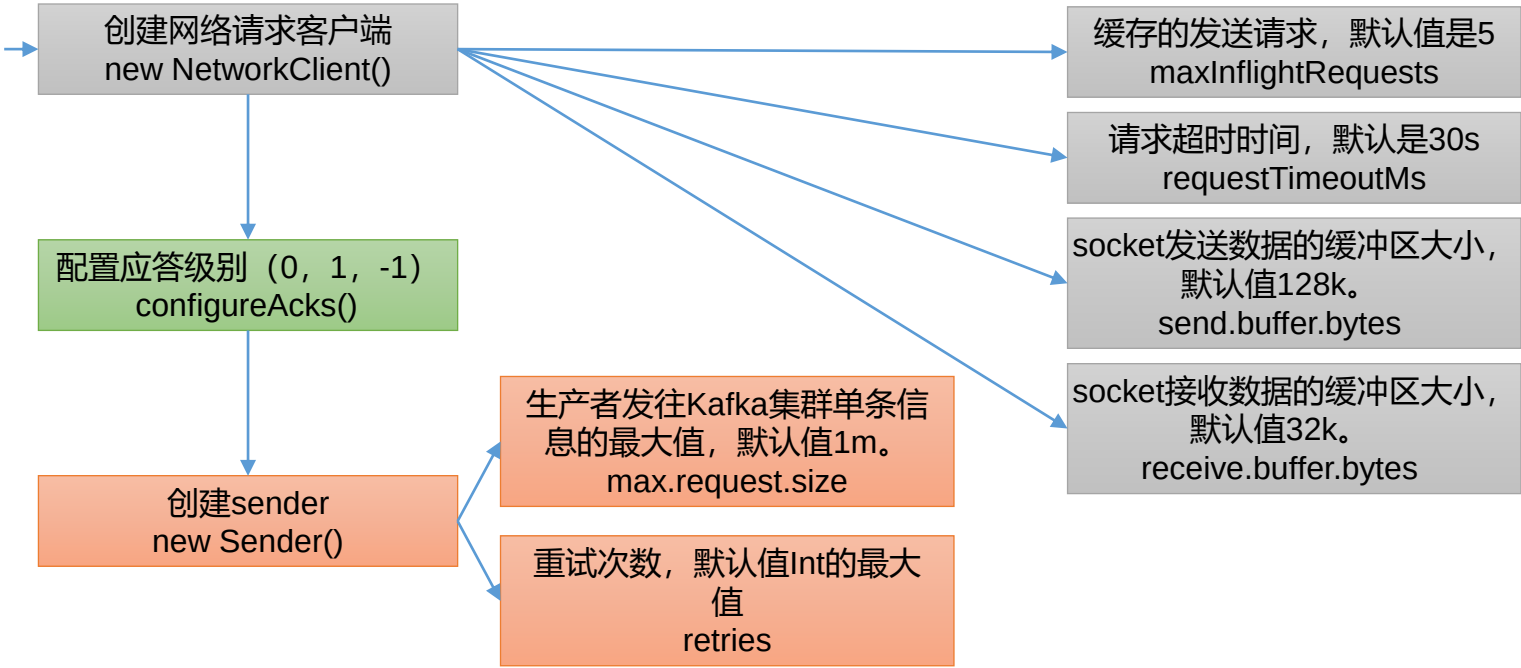
生产者main线程初始化



生产者sender线程初始化

1、初始化sender线程

this.sender = newSender()

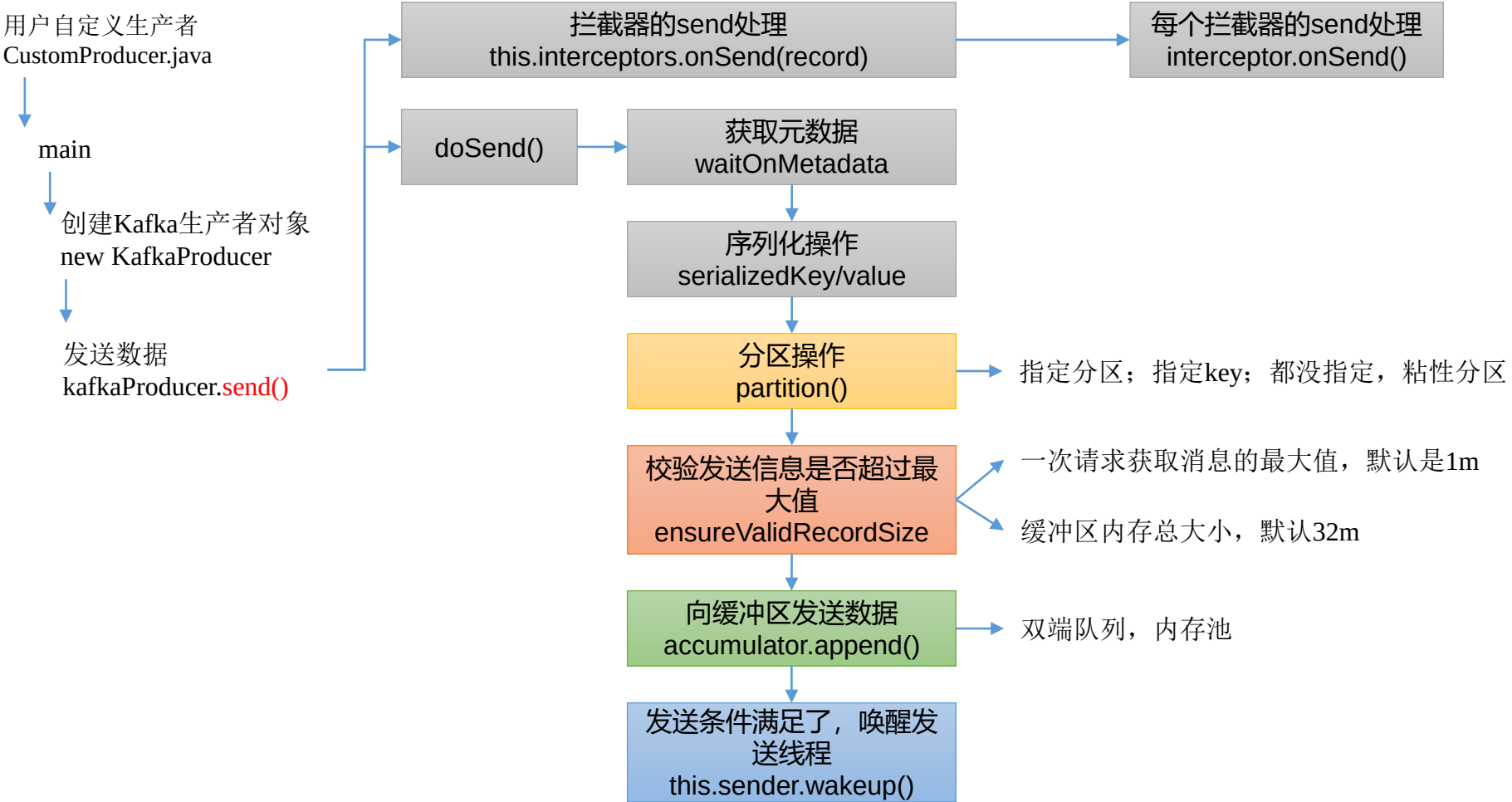


2、启动sender线程

```
this.ioThread = new
KafkaThread(ioThreadName,
this.sender, true);
this.ioThread.start();
```

```
@Override
public void run() {
        while (running) {
                // sender线程从缓冲区准备拉取数据，刚启动拉不到数据
                runOnce();
        }
}
```

发送数据到缓冲区


用户自定义生产者
CustomProducer.java
main
创建Kafka生产者对象
new KafkaProducer
发送数据
kafkaProducer.send()
拦截器的send处理
this.interceptors.onSend(record)
每个拦截器的send处理
interceptor.onSend()
doSend()
获取元数据
waitOnMetadata
序列化操作
serializedKey/value
分区操作
partition()
指定分区；指定key；都没指定，粘性分区
校验发送信息是否超过最大值
ensureValidRecordSize
一次请求获取消息的最大值，默认是1m
缓冲区内存总大小，默认32m
向缓冲区发送数据
accumulator.append()
双端队列，内存池
发送条件满足了，唤醒发送线程
this.sender.wakeup()

sender线程发送数据

唤醒发送请求
this.sender.wakeup();

```
@Override
public void run() {
        while (running) {
            runOnce();
        }
}
```

将准备好的数据发送到服务器端
sendProducerData()

获取元数据
metadata.fetch()

检查32m缓存是否准备好
this.accumulator.ready()

lingerMs时间到

发往同一个broker节点的数据，打包为一个请求批次
this.accumulator.drain()

batches.put(node.id(), ready);

发送请求
sendProduceRequests()

client.newClientRequest

client.send() → doSend()

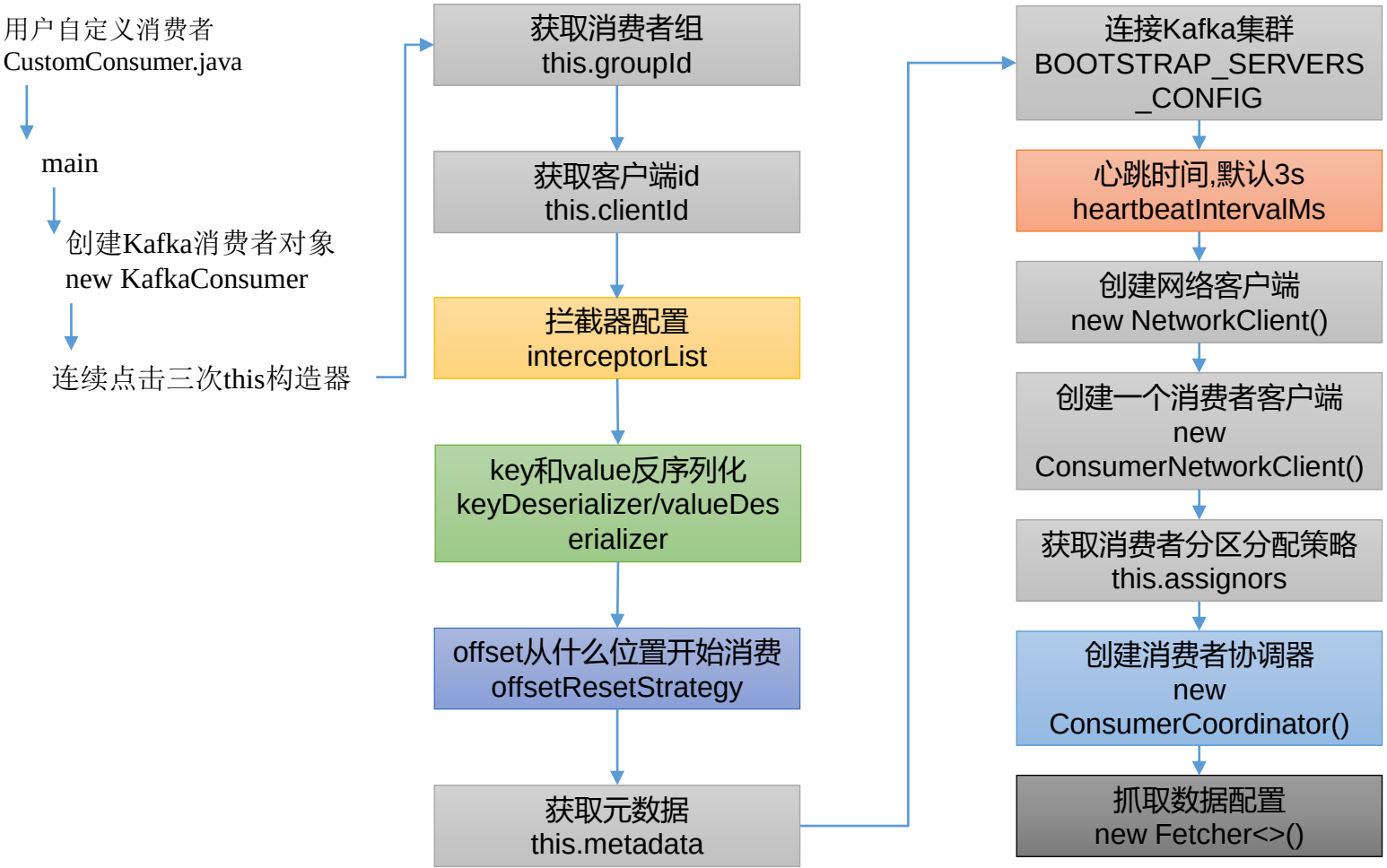
this.inFlightRequests.add(inFlightRequest);

selector.send()

等待发送响应
client.poll()

this.selector.poll()

消费者初始化


用户自定义消费者
CustomConsumer.java
main
创建Kafka消费者对象
new KafkaConsumer
连续点击三次this构造器
获取消费者组
this.groupId
获取客户端id
this.clientId
拦截器配置
interceptorList
key和value反序列化
keyDeserializer/valueDes
erializer
offset从什么位置开始消费
offsetResetStrategy
获取元数据
this.metadata
连接Kafka集群
BOOTSTRAP_SERVERS
_CONFIG
心跳时间,默认3s
heartbeatIntervalMs
创建网络客户端
new NetworkClient()
创建一个消费者客户端
new
ConsumerNetworkClient()
获取消费者分区分配策略
this.assignors
创建消费者协调器
new
ConsumerCoordinator()
抓取数据配置
new Fetcher<>()

消费者订阅主题

用户自定义消费者
CustomConsumer.java

main

创建Kafka消费者对象
new KafkaConsumer

订阅主题

```
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);
```

判断是否需要更改订阅主题，如果需要更改主题，则更新元数据信息
this.subscriptions.subscribe(new HashSet<>(topics), listener)

注册负载均衡监听（例如消费者组中，其他消费者退出触发再平衡）
registerRebalanceListener(listener)

修改订阅主题信息
changeSubscription(topics)

如果订阅的主题和以前订阅的一致，就不需要修改订阅信息。如果不一致，就需要修改

如果订阅的和以前不一致，需要更新元数据信息
metadata.requestUpdateForNewTopics()

消费者拉取和处理数据

用户自定义消费者
CustomConsumer.java

main

1 创建Kafka消费者对象
new KafkaConsumer

2 订阅主题
ArrayList<String> topics = new ArrayList<>();
topics.add("first");
kafkaConsumer.subscribe(topics);

3 消费数据

```
while (true){
    ConsumerRecords<String, String> consumerRecords = kafkaConsumer.poll(Duration.ofSeconds(1));
    for (ConsumerRecord<String, String> consumerRecord : consumerRecords) {
        System.out.println(consumerRecord);
    }
}
```

消费者or消费者组初始化
updateAssignmentMetadataIfNeeded()

协调器交互
coordinator.poll()

获取最新元数据

3s发送一次心跳
pollHeartbeat()

判断是否需要加入消费者组()

拉取数据
pollForFetches()

发送请求并抓取数据
fetcher.sendFetches()

初始化抓取参数
FetchRequest.Builder

发送拉取数据请求
client.send()

监听服务器端返回的数据future.addListener()

拦截器处理消息
interceptors.onConsume()