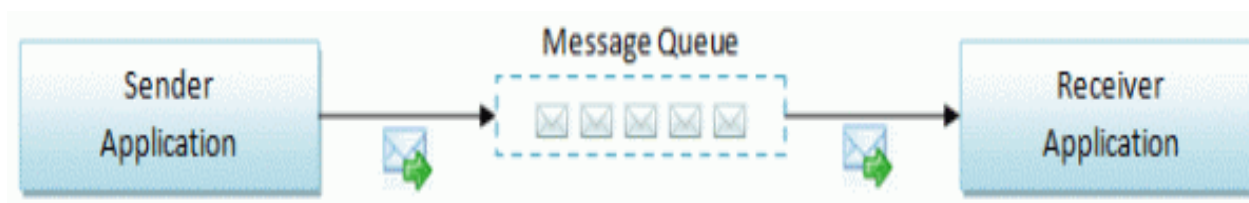




kafka技术分享

## 消息队列 (Message Queue)

### ➤ MQ的模型



- 1、解耦合
- 2、提高系统的响应时间

```
订单支付成功的方法(){
    1、修改订单状态
    2、计算会员积分
    3、通知物流进行配送
}
```

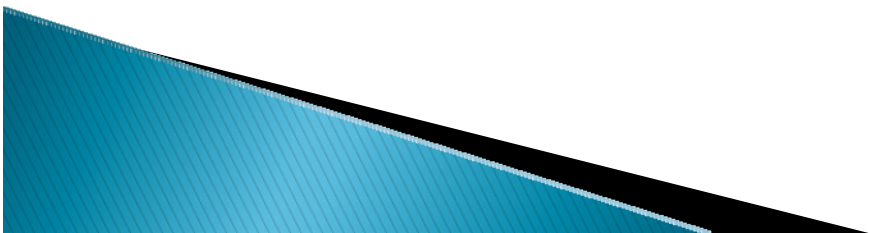
## 消息队列分类

### ➤ 点对点:

- 消息生产者生产消息发送到queue中，然后消息消费者从queue中取出并且消费消息。
- 注意:
  - 消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。
  - Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

### ➤ 发布/订阅:

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到topic的消息会被所有订阅者消费。



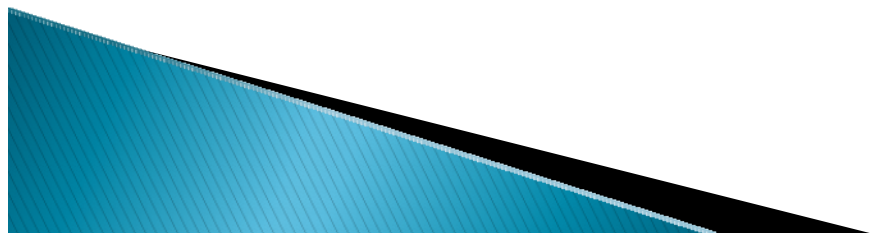
## 消息队列MQ对比

- **RabbitMQ**: 支持的协议多, 非常重量级消息队列, 对路由、负载均衡或者数据持久化都有很好的支持。
- **ZeroMQ**: 号称最快的消息队列系统, 尤其针对大吞吐量的需求场景, 擅长的高级/复杂的队列, 但是技术也复杂, 并且只提供非持久性的队列。
- **ActiveMQ**: Apache下的一个子项, 类似ZeroMQ, 能够以代理人和点对点的技术实现队列。
- **Redis**: 是一个key-Value的NOsql数据库, 但也支持MQ功能, 数据量较小, 性能优于RabbitMQ, 数据超过10K就慢的无法忍受
- **rocketMQ**: RocketMQ作为一款纯java、分布式、队列模型的开源消息中间件 (经历了淘宝双十一的洗礼, 在功能和性能上据说是远超ActiveMQ。性能好, 高吞吐, 高可用性, 支持大规模分布式。
- **kafka**: 是一个高吞吐量的分布式消息系统, 具有高性能, 快速持久化, 无消息确认, 无消息遗漏, 可能会有重复消息, 依赖于zookeeper, 成本高的特点。

## Kafka简介

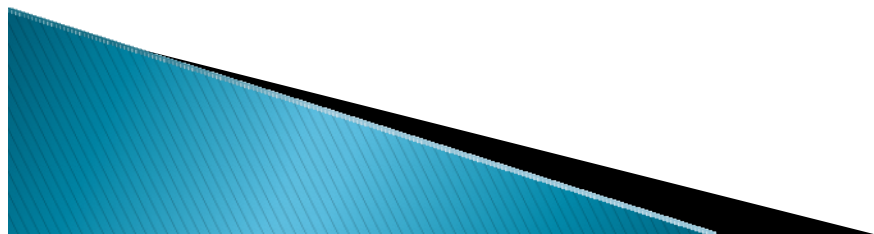
Kafka 是**分布式发布-订阅消息系统**。它最初由 LinkedIn 公司开发，使用 **Scala**语言编写,之后成为 **Apache** 项目的一部分。Kafka 是一个**分布式的，可划分的，多订阅者,冗余备份的持久性的日志服务**。以可**水平扩展和高吞吐率**而被广泛使用。目前越来越多的开源分布式处理系统如 **Cloudera、Apache Storm、Spark**等都支持与**Kafka**集成。**Kafka**凭借着自身的优势，越来越受到互联网企业的青睐，唯品会也采用**Kafka**作为其内部核心消息引擎之一。它主要用于**处理活跃的流式数据**。

在**0.10**版本之前，**Kafka**仅仅作为一个消息系统，主要用来解决应用解耦、异步消息、流量削峰等问题。不过在 **0.10**版本之后，**Kafka**提供了连接器与流处理的能力，它也从分布式的消息系统逐渐成为一个**流式的数据平台**。

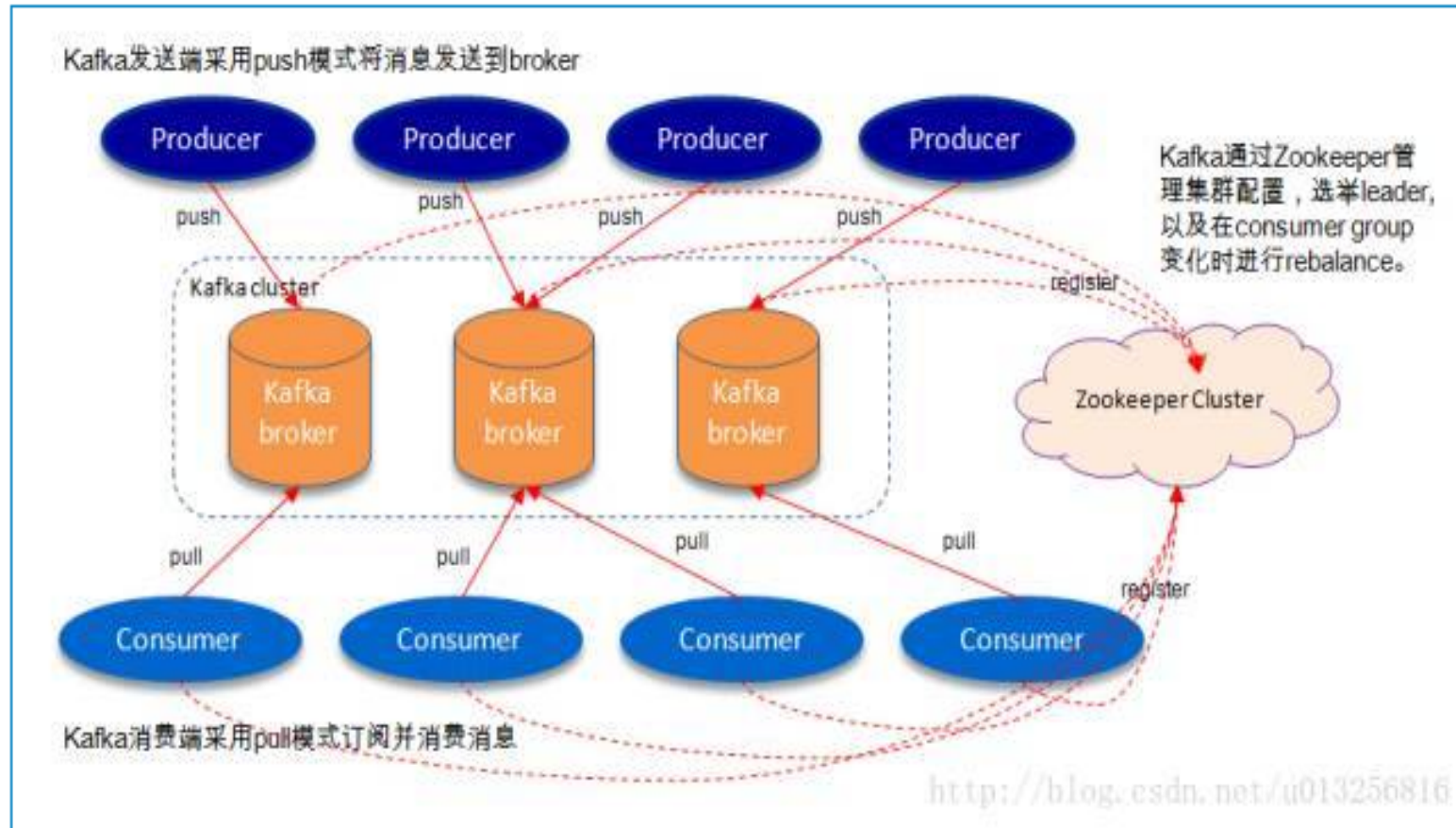


## Kafka的特点

- 同时为发布和订阅提供高吞吐量。据了解，Kafka 每秒可以生产约 25 万消息（50 MB），每秒处理 55 万消息（110 MB），它的延迟最低只有几毫秒。
- 可进行持久化操作。将消息持久化到磁盘，支持数据备份防止数据丢失（Kafka直接将数据写入到日志文件中，以追加的形式写入，采用的是顺序写磁盘，因此效率非常高，经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）
- 具有很高的容错性，通过健壮的副本(replication)策略，可以使得Kafka在性能和可靠性之间运转的游刃有余
- 分布式系统，易于向外扩展。所有的 producer、broker 和 consumer 都会有多个，均为分布式的。无需停机即可扩展机器。
- 消息被处理的状态是在 consumer 端维护，而不是由 server 端维护。当失败时能自动平衡。

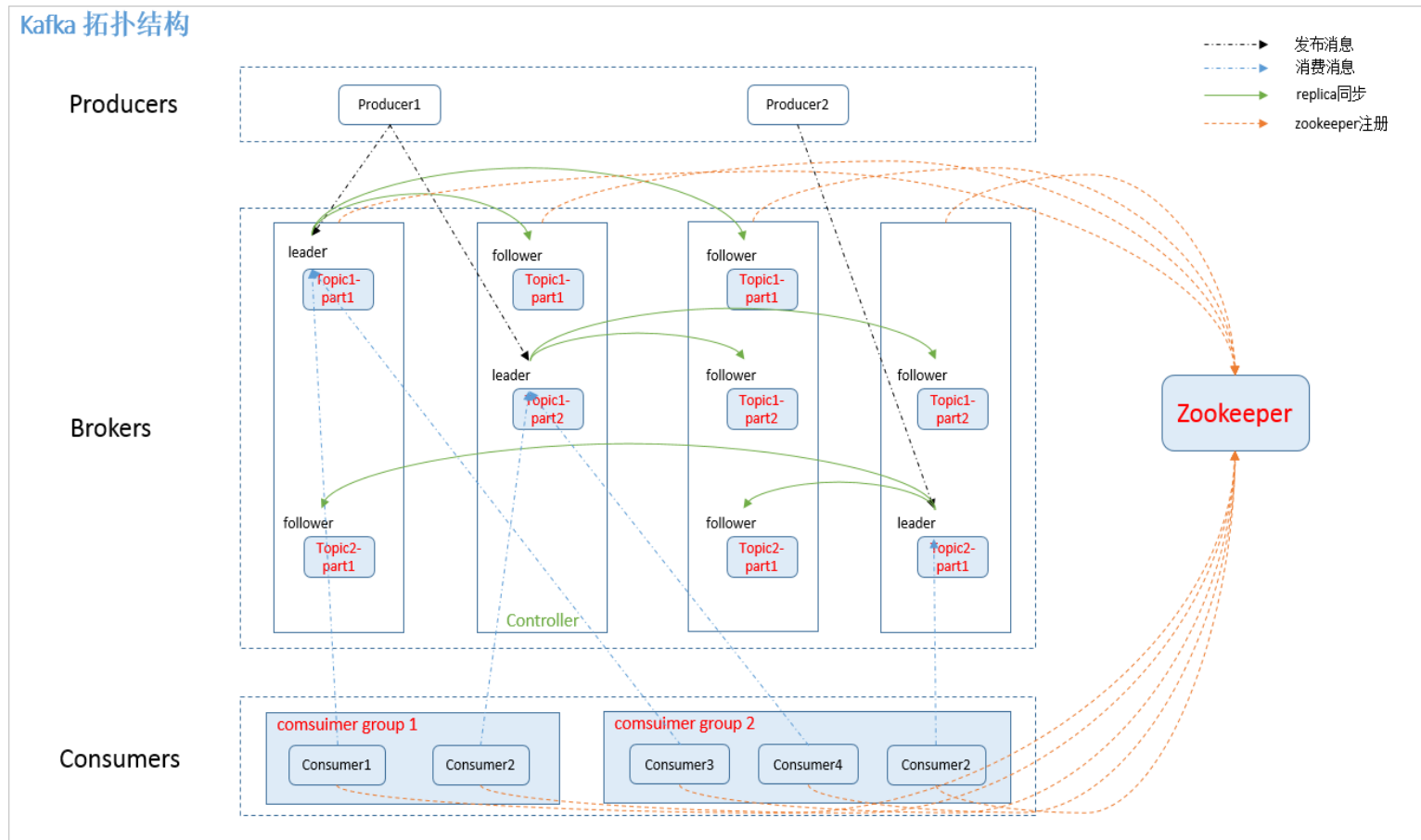


# Kafka的架构



# Kafka的架构

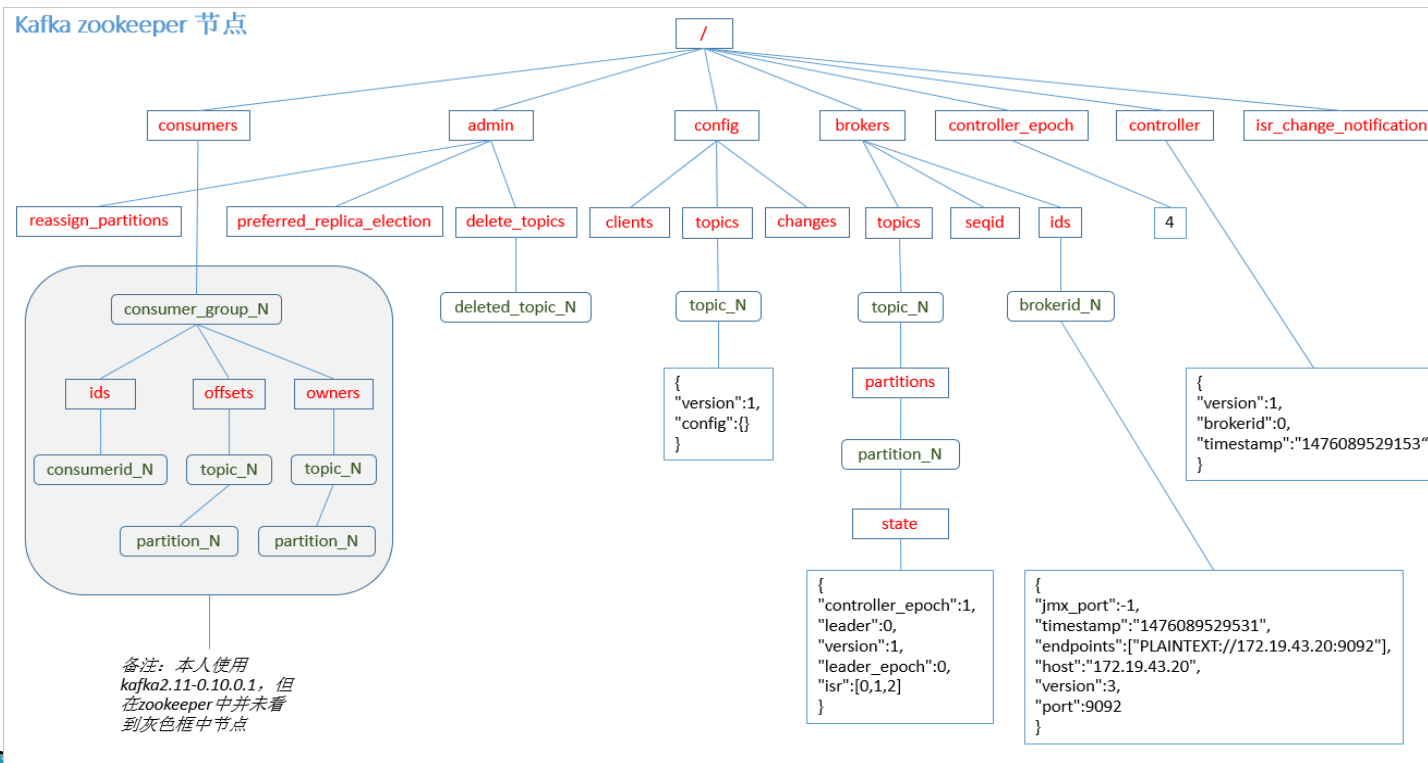
Kafka 拓扑结构





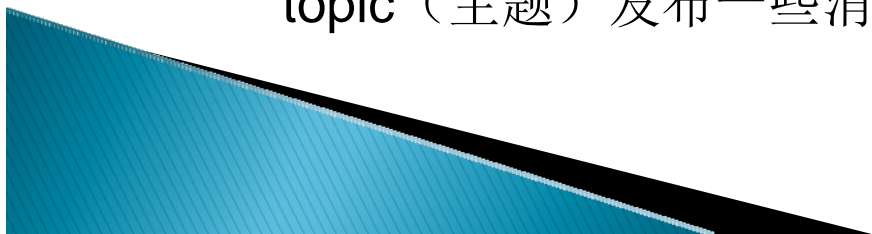
# zookeeper

- 1) 管理broker与consumer的动态加入与离开。
- 2) 触发负载均衡，当broker或consumer加入或离开时会触发负载均衡算法，使得一个consumer group内的多个consumer的订阅负载平衡。
- 3) 维护消费关系及每个partition的消费信息。



## Kafka的核心概念

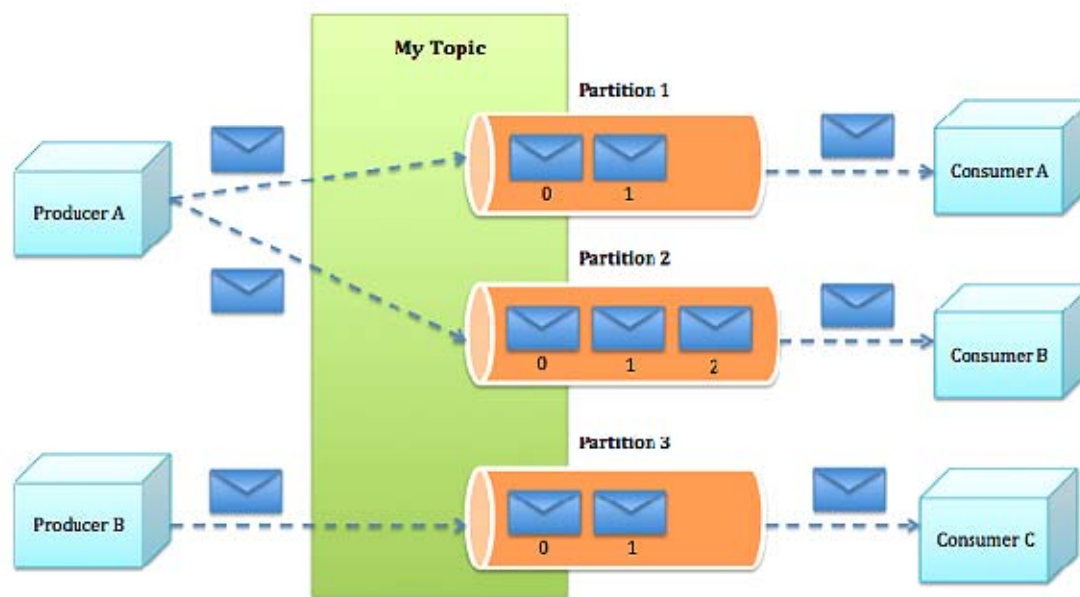
- 服务代理(Broker): 消息中间件处理节点, 一个kafka节点就是一个broker, 一个或多个broker可以组成一个kafka集群。
- 话题(Topic): kafka根据topic对消息进行归类, 发布到kafka集群的每条消息都需要指定一个topic
- 生产者(Producer): 消息生产者, 向broker发送消息的客户端(push)
- 消费者(Consumer): 消息消费者, 从broker读取消息的客户端(pull)
- 消费者组(Consumer Group): 每个consumer属于一个特定的consumer Group, 一条消息可以发送到多个不同的consumer Group, 但是一个consumer Group中只能有一个consumer能够消费该消息.
- 分区(Partition): 物理上的概念, 一个topic可以分为多个partition, 每个partition内部是有序的
- 消息(Message): 是通信的基本单位, 每个 producer 可以向一个topic (主题) 发布一些消息。



# producer发布消息

## 1、消息的分区

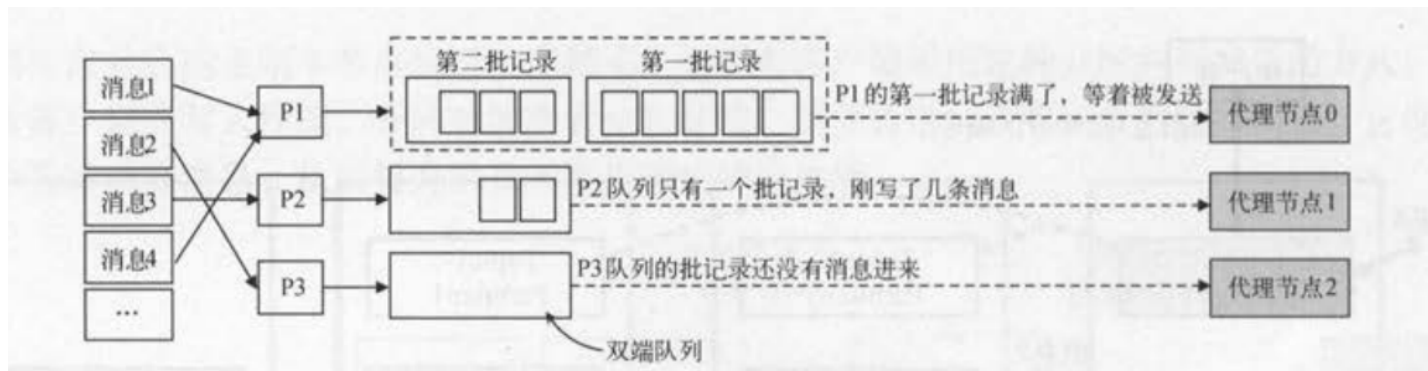
kafka通过将主题分成多个分区的语意来实现**并行处理**，生产者可以将一批消息分成多个分区，每个分区写入不同的服务端节点，生产者客户端采用这种分区并行发送的方式，从而**提升生产者客户端的写入性能**。分区对消费组也有好处，消费组指定获取一个主题的消息，它也可以同时从多个分区读取消息，从而**提升消费者客户端的读取性能**



# producer发布消息

## 2、producer的消息集

生产者发送的消息先在客户端缓存到记录收集器中，等到一定时机再由发送线程**Sender**批量的写入**kafka**集群。生产者每生产一条消息，就想记录收集器中追加一条消息，最佳方法的返回值表示批记录是否满了；如果满了则开始发送这一批数据。如果没有满则继续等待收集到足够的消息在发送



## producer发布消息

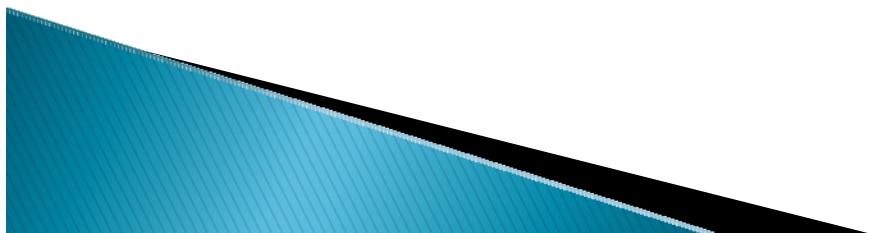
### 3、消息的写入方式

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率）。

### 4、消息的路由

producer 发送消息到 broker 时，会根据分区算法选择将其存储到哪一个 partition。其路由机制为：

1. 指定了 partition，则直接使用；
2. 未指定 partition 但指定 key，通过对 key 的 value 进行hash 选出一个 partition
3. partition 和 key 都未指定，使用轮询选出一个 partition。



# producer发布消息

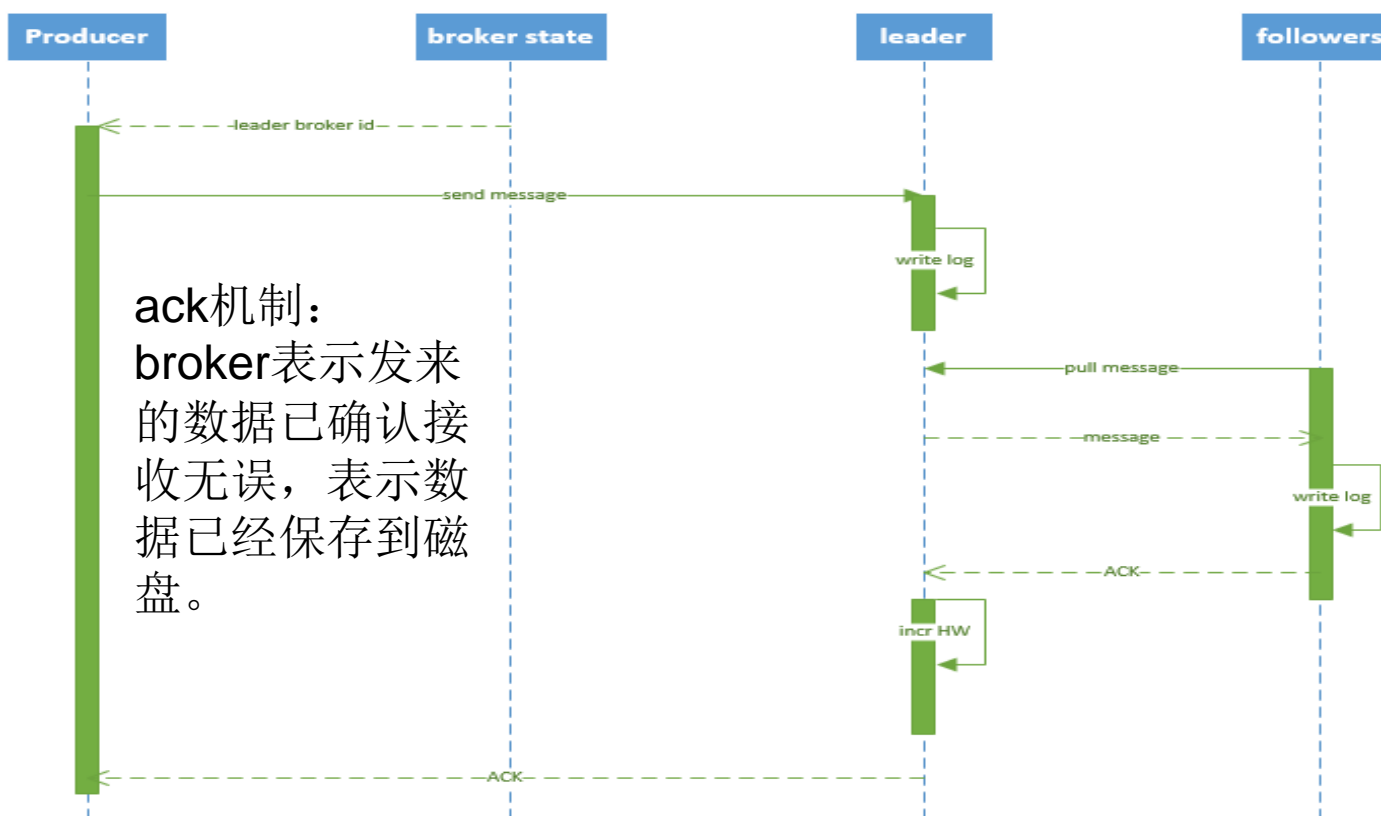
## 5、消息分区的选择

```
/**
 * Compute the partition for the given record.
 * 为消息选择分区编号
 * @param topic The topic name
 * @param key The key to partition on (or null if no key)
 * @param keyBytes serialized key to partition on (or null if no key)
 * @param value The value to partition on or null
 * @param valueBytes serialized value to partition on or null
 * @param cluster The current cluster metadata
 */
public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
    //获取主题所有的分区，用来实现消息的负载均衡
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size();
    //消息没有key则均衡分布
    if (keyBytes == null) {
        //计数器递增
        int nextValue = nextValue(topic);
        List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);
        if (availablePartitions.size() > 0) {
            int part = Utils.toPositive(nextValue) % availablePartitions.size();
            return availablePartitions.get(part).partition();
        } else {
            // no partitions are available, give a non-available partition
            return Utils.toPositive(nextValue) % numPartitions;
        }
    } else {
        // hash the keyBytes to choose a partition
        //有key的话，对消息的key进行散列化后取模
        return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
}
```

为了保证消息的均衡的负载到各个服务端的每个节点上，对于没有key的消息，通过计数器自增轮询的方式依次将消息分配到不同的分区上，对于有键的消息，对键计算散列值，然后和主题的分区分数进行取模得到分区编号

## producer发布消息

### 6、消息的写入流程





## producer发布消息

### 7、消息的写入流程说明

1. producer 先从 zookeeper 的 `"/brokers/.../state"` 节点找到该 partition 的 leader
2. producer 将消息发送给该 leader
3. leader 将消息写入本地 log
4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK
5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW (high watermark, 最后 commit 的 offset) 并向 producer 发送 ACK

客户端对消息的读写操作只针对leader，follower只有副本的作用，不能进行任何读写操作。



## broker 保存消息

### 1、存储方式

物理上把 topic 分成一个或多个 partition（对应 server.properties 中的 num.partitions=3 配置），每个 partition 物理上对应一个文件夹（该文件夹存储该 partition 的所有消息和索引文件），如下：

```
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-0/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-1/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-2/
-rw-r--r--  1 root root   0 Oct 10 00:48 .lock
-rw-r--r--  1 root root  54 Oct 10 00:48 meta.properties
-rw-r--r--  1 root root 1254 Oct 10 04:56 recovery-point-offset-checkpoint
-rw-r--r--  1 root root 1256 Oct 10 04:57 replication-offset-checkpoint
feng@ubuntu:/tmp/kafka/kafka-logs-1$ cd demoTopic2-0
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$ ll
total 12
drwxr-xr-x  2 root root  4096 Oct 10 01:54 ./
drwxr-xr-x 56 root root  4096 Oct 10 04:57 ../
-rw-r--r--  1 root root 10485760 Oct 10 01:54 00000000000000000000.index
-rw-r--r--  1 root root    188 Oct 10 02:28 00000000000000000000.log
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$
```

## broker 保存消息

### 2、存储策略

无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

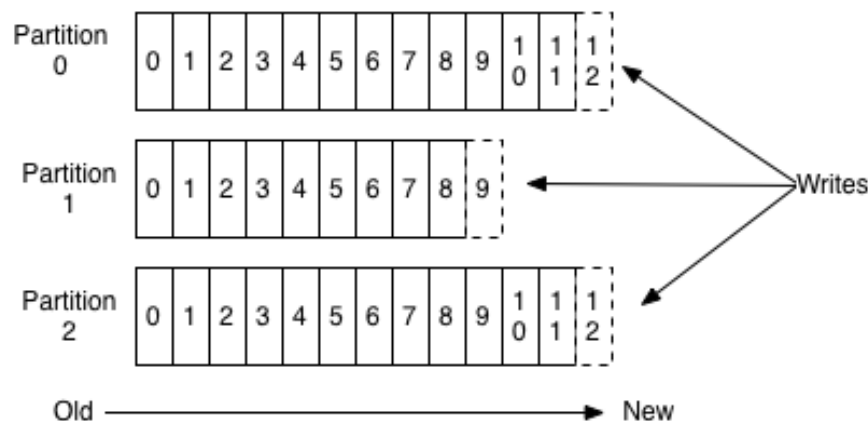
1. 基于时间：log.retention.hours=168
2. 基于大小：log.retention.bytes=1073741824

需要注意的是，因为Kafka读取特定消息的时间复杂度为 $O(1)$ ，即与文件大小无关，所以这里删除过期文件与提高Kafka 性能无关。

## Kafka的持久化

一个Topic可以认为是一类消息，每个topic将被分成多partition(区),每个partition在存储层面是append log文件。任何发布到此partition的消息都会被直接追加到log文件的尾部，每条消息在文件中的位置称为offset（偏移量），partition是以文件的形式存储在文件系统中。Logs文件根据broker中的配置要求,保留一定时间后删除来释放磁盘空间。

### Anatomy of a Topic



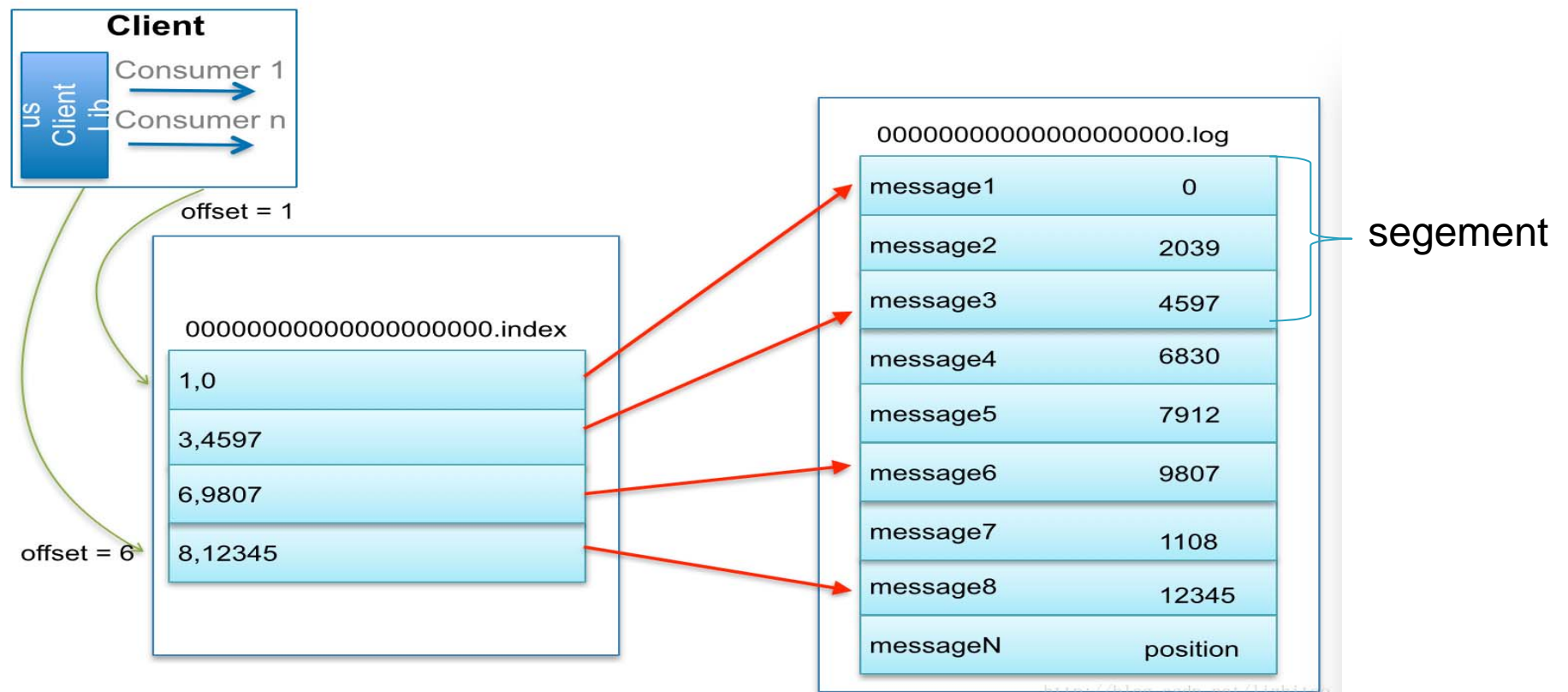
### Partition:

Topic 物理上的分组，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。

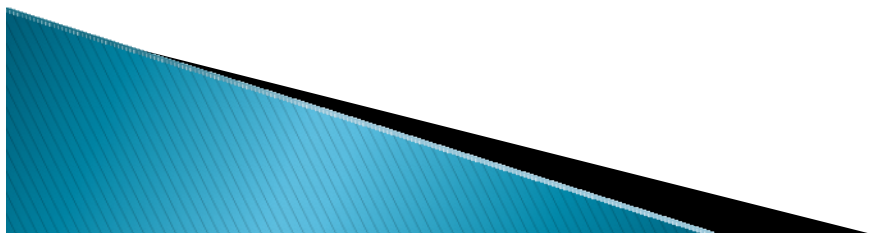
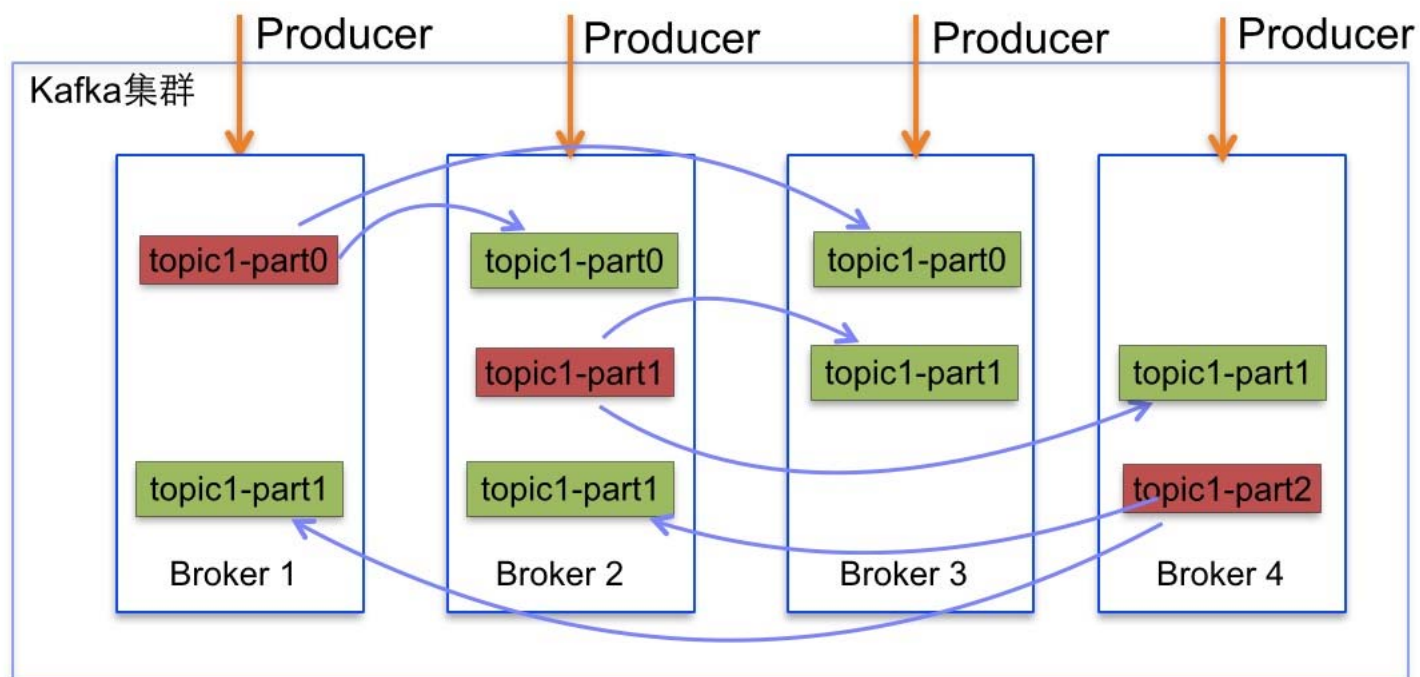
partition 中的每条消息都会被分配一个有序 id (offset)。

# Kafka的持久化

- 为数据文件建索引：稀疏存储，每隔一定字节的数据建立一条索引。  
下图为一个partition的索引示意图：

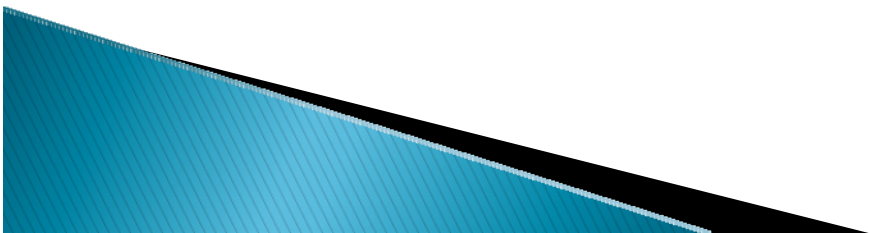


## kafka的副本机制



## kafka的副本机制

- 当某个topic的replication-factor为N且N大于1时，每个Partition都会有N个副本(Replica)。kafka的replica包含leader与follower。
- Replica的个数小于等于Broker的个数，也就是说，对于每个Partition而言，每个Broker上最多只会有一个Replica，因此可以使用Broker id 指定Partition的Replica。
- 所有Partition的Replica默认情况会均匀分布到所有Broker上。



# kafka的副本机制

## kafka副本的分配算法

- 将所有N Broker和待分配的i个Partition排序.
- 将第i个Partition分配到第 $(i \bmod n)$ 个Broker上.
- 将第i个Partition的第j个副本分配到第 $((i + j) \bmod n)$ 个Broker上.

broker: broker1,broker2,broker3  
partition:part1,part2,part3

	broker1	broker2	broker3
leader:	part3	part1	part2
follow1:	part1	part2	part3
follow2:	part2	part3	part1

kafka的副本分配算法: 会首先随机选取第n个broker将第1个分区放置到这个broker,  
然后按照顺序放置其他分区, 根据每个副本依次向前数一个分区



# Kafka的选举机制

## 副本同步列表ISR

副本同步列表(In-Sync Replicas): leader会维护一个与其基本保持同步的Replica列表, 该列表称为ISR(in-sync Replica), 每个Partition都会有一个ISR, 而且是由leader动态维护。

如果一个follower比一个leader落后太多, 或者超过一定时间未发起数据复制请求, 则leader将其从ISR中移除。

当ISR中所有Replica都向Leader发送ACK时, leader才commit

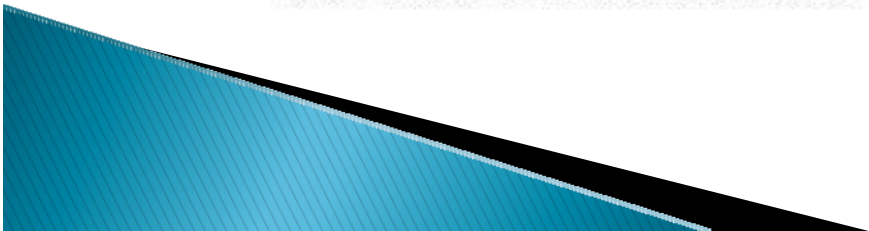
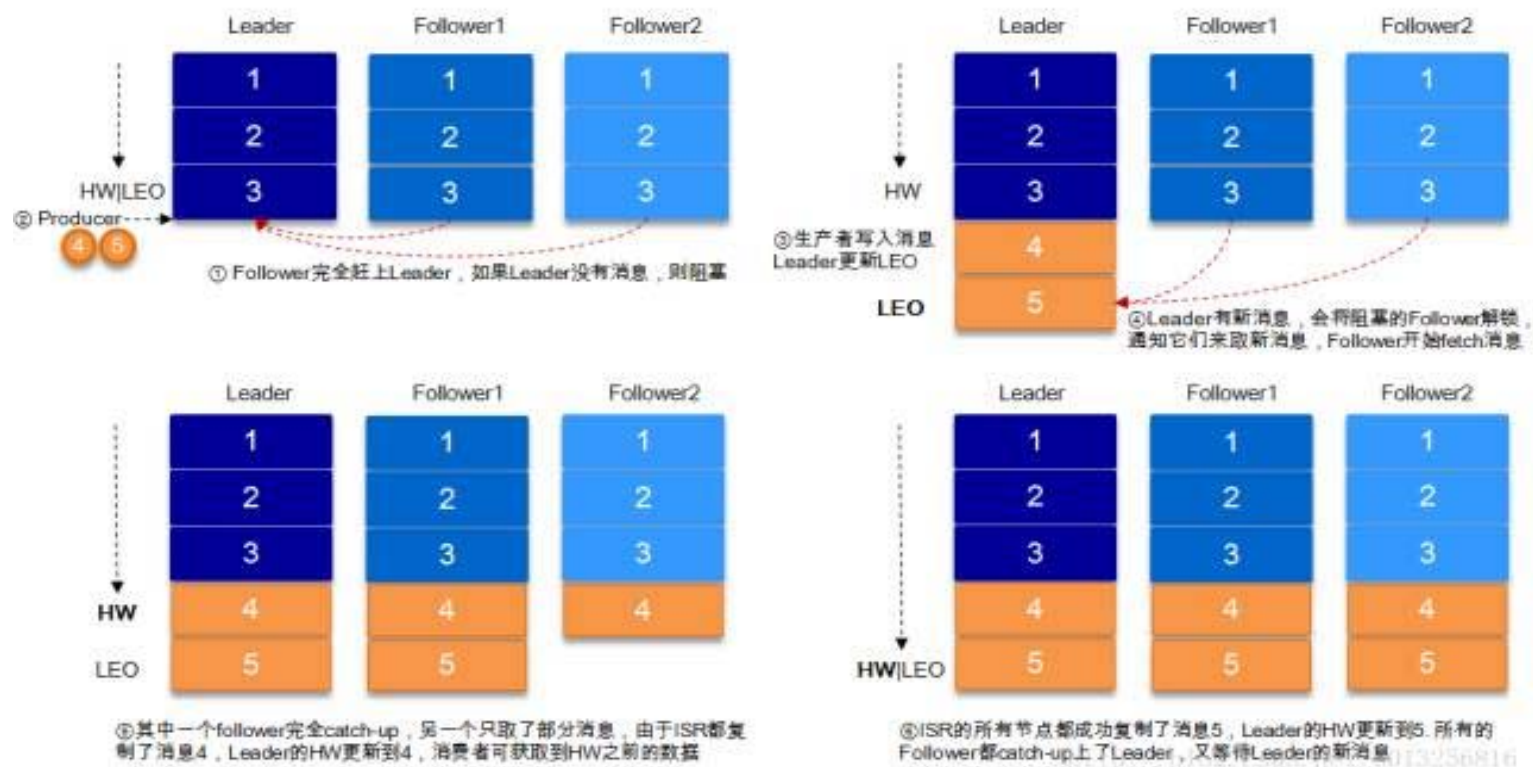
一条消息只有被ISR中的所有follower都从leader复制过去才会被认为已提交。这样就避免了部分数据被写进了leader, 还没来得及被任何follower复制就宕机了, 而造成数据丢失。而对于producer而言, 它可以选择是否等待消息commit, 这可以通过request.required.acks来设置。这种机制确保了只要ISR中有一个或者以上的follower, 一条被commit的消息就不会丢失。





# Kafka的选举机制

## ISR的流转过程

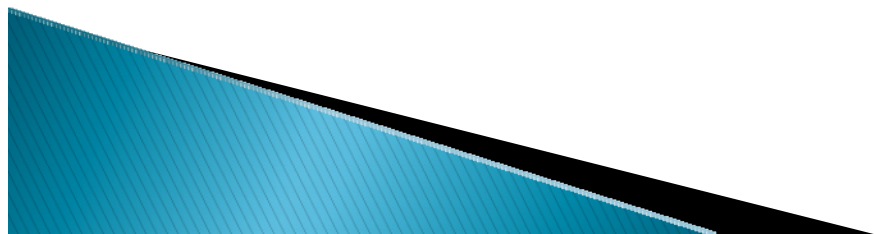


## Kafka的选举机制

Kafka在Zookeeper中为每一个partition动态的维护了一个ISR，这个ISR里的所有replica都跟上了leader，只有ISR里的成员才能有被选为leader的可能(`unclean.leader.election.enable=false`)。在这种模式下，对于 $f+1$ 个副本，一个Kafka topic能在保证不丢失已经commit消息的前提下容忍 $f$ 个副本的失败，在大多数使用场景下，这种模式是十分有利的。

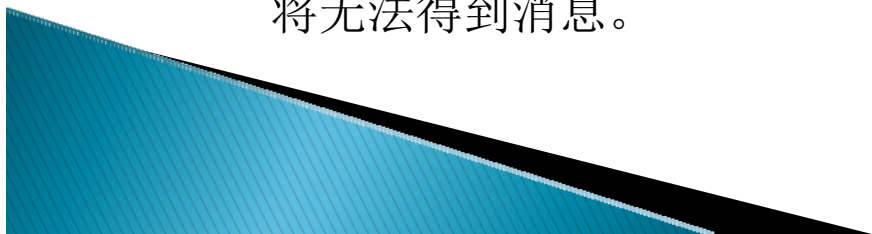
在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某一个partition的所有replica都挂了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

等待ISR中任意一个replica “活” 过来，并且选它作为leader  
选择第一个 “活” 过来的replica(并不一定是在ISR中)作为leader

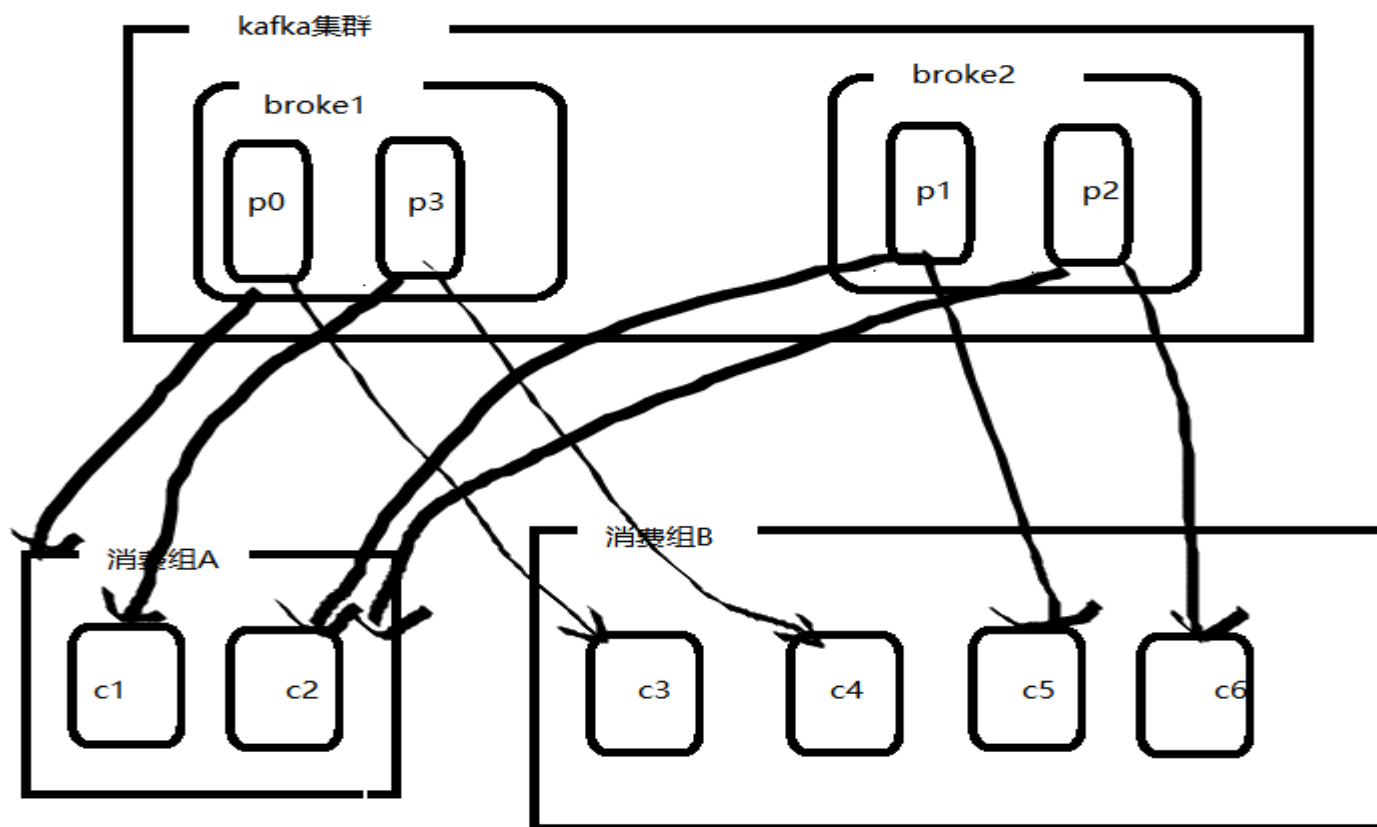


## kafka的消费者、消费组

- 通常情况下，一个group中会包含多个consumer，这样不仅可以提高topic中消息的并发消费能力，而且还能提高故障容错性，如果group中的某个consumer失效那么其消费的partitions将会有其他consumer自动接管。
- 对于Topic中的一条特定的消息，只会被订阅此Topic的每个group中的其中一个consumer消费，此消息不会发送给一个group的多个consumer；那么一个group中所有的consumer将会交错的消费整个Topic，每个group中consumer消息消费互相独立，我们可以认为一个group是一个”订阅”者。
- 在kafka中，一个partition中的消息只会被group中的一个consumer消费(同一时刻)一个Topic中的每个partitions，只会被一个”订阅者”中的一个consumer消费，不过一个consumer可以同时消费多个partitions中的消息。
- kafka的设计原理决定, 对于一个topic，同一个group中不能有多于partitions个数的consumer同时消费，否则将意味着某些consumer将无法得到消息。



## kafka的消费者、消费组

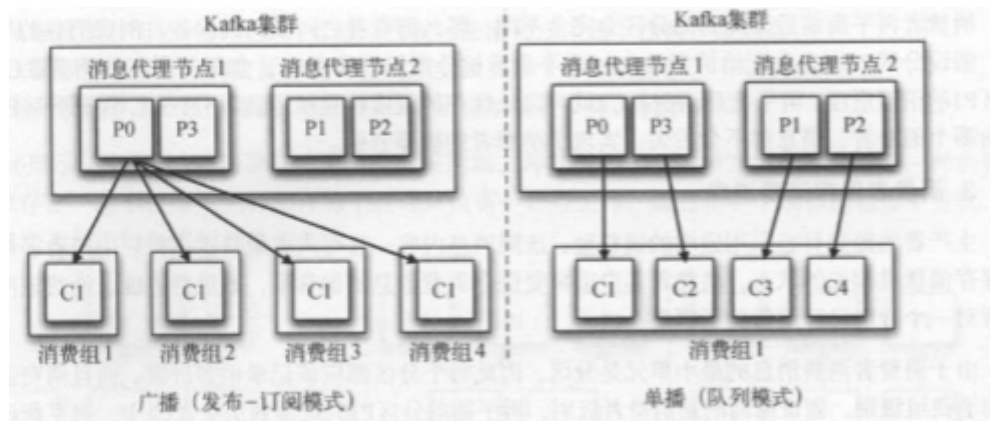


## kafka的消费者、消费组

kafka只能保证一个partition中的消息被某个consumer消费时是顺序的；事实上，从Topic角度来说,当有多个partitions时,消息仍不是全局有序的。

Kafka采用消费组保证了一个分区只可被消费组中的一个消费者所消费，这意味着：

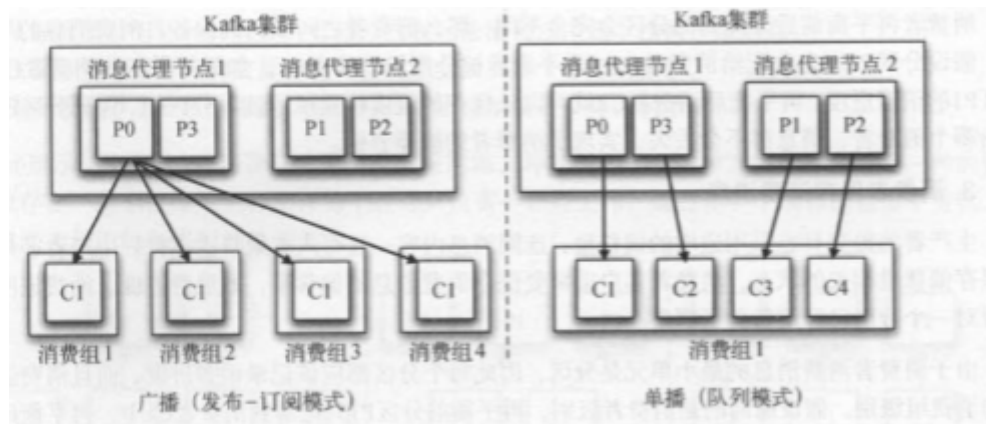
- (1) 在一个消费组中，一个消费者可以消费多个分区。
- (2) 不同的消费者消费的分区一定不会重复，所有消费者一起消费所有的分区。
- (3) 在不同消费组中，每个消费组都会消费所有的分区。
- (4) 同一个消费组下消费者对分区是互斥的，而不同消费组之间是共享的



## kafka的消费者、消费组

### 使用消费组实现消息队列的两种模式

- 发布-订阅模式：同一条消息会被多个消费组消费，每个消费组只有一个消费者，实现广播。
- 队列模式：只有一个消费组、多个消费者 一条消息只被消费组的一个消费者消费，实现单播。



## consumer 消费消息

**Kafka**采用拉取模型，由消费者自己记录消费状态，每个消费者互相独立地顺序读取每个分区的信息。如图1-4所示，有两个消费者（不同消费组）拉取同一个主题的消息，消费者A的消费进度是3，消费者B的消费进度是6。消费者拉取的最大上限通过最高水位（**watermark**）控制，生产者最新写入的消息如果还没有达到备份数量，对消费者是不可见的。这种由消费者控制偏移量的优点是：消费者可以按照任意的顺序消费消息。比如，消费者可以重置到旧的偏移量，重新处理之前已经消费过的消息；或者直接跳到最近的位置，从当前时刻开始消费。

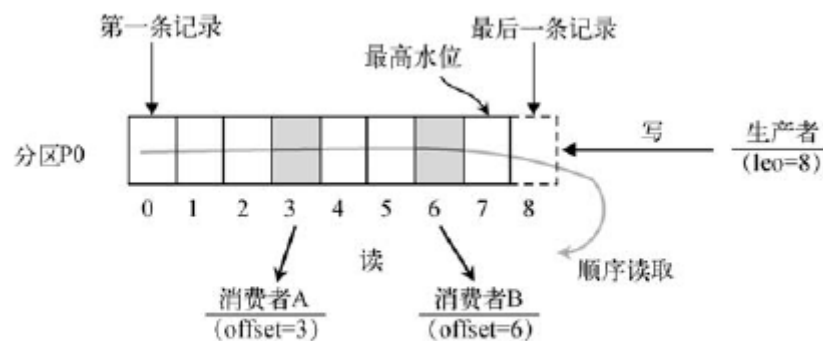


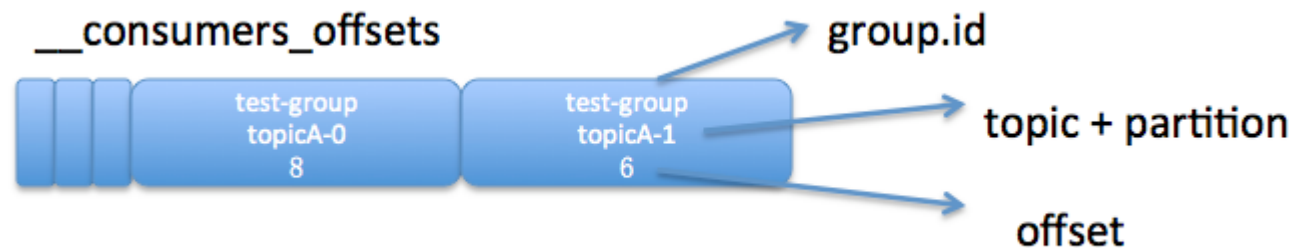
图1-4 采用拉取模型的消费者自己记录消费状态



## kafka的消费者、消费组

### 消费者保存消费进度

- 1、虽然分区是以消费者级别被消费的，但分区的消费进度要保存成消费组级别的。
- 2、每个consumer group保存自己的位移信息
- 3、老版本的位移是提交到zookeeper中的，但是zookeeper其实并不适合进行大批量的读写操作，尤其是写操作。因此kafka提供了另一种解决方案：增加\_\_consumeroffsets topic，将offset信息写入这个topic，摆脱对zookeeper的依赖(指保存offset这件事情)。\_\_consumer\_offsets中的消息保存了每个consumer group某一时刻提交的offset信息。





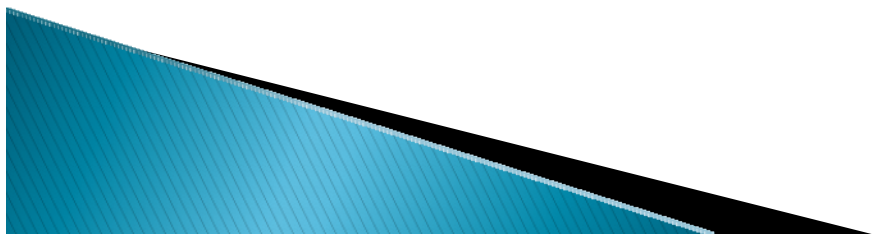
## kafka消费组的再平衡

### 什么是再平衡（rebalance）

**rebalance**本质上是一种协议，规定了一个**consumer group**下的所有**consumer**如何达成一致来分配订阅**topic**的每个分区。比如某个**group**下有20个**consumer**，它订阅了一个具有100个分区的**topic**。正常情况下，**Kafka**平均会为每个**consumer**分配5个分区。这个分配的过程就叫**rebalance**。

### 什么时候再平衡（rebalance）

- 1) 组成员发生变更
- 2) 订阅主题数发生变更——这当然是可能的，如果你使用了正则表达式的方式进行订阅，那么新建匹配正则表达式的**topic**就会触发**rebalance**
- 3) 订阅主题的分区数发生变更

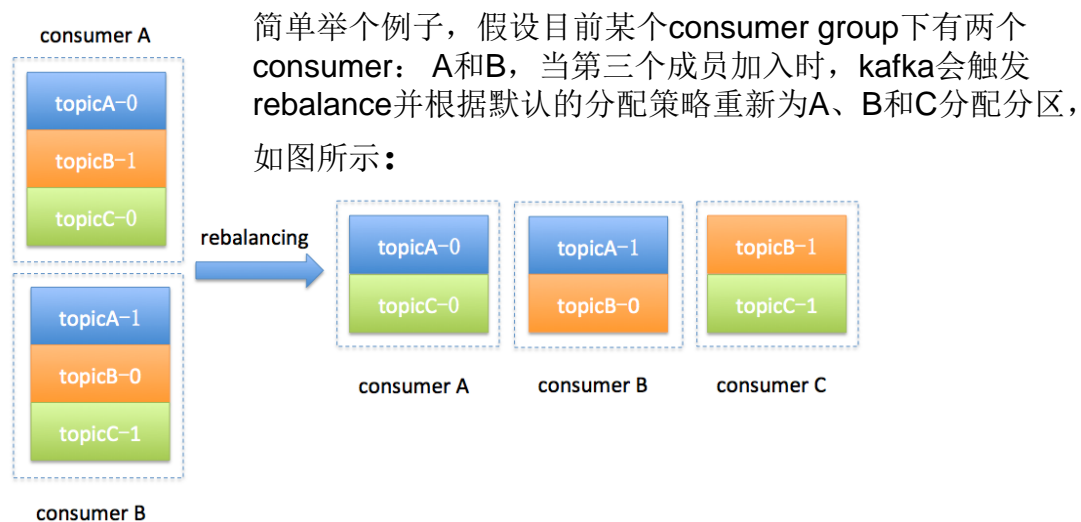


# kafka消费组的再平衡

## 如何进行组内分区分配

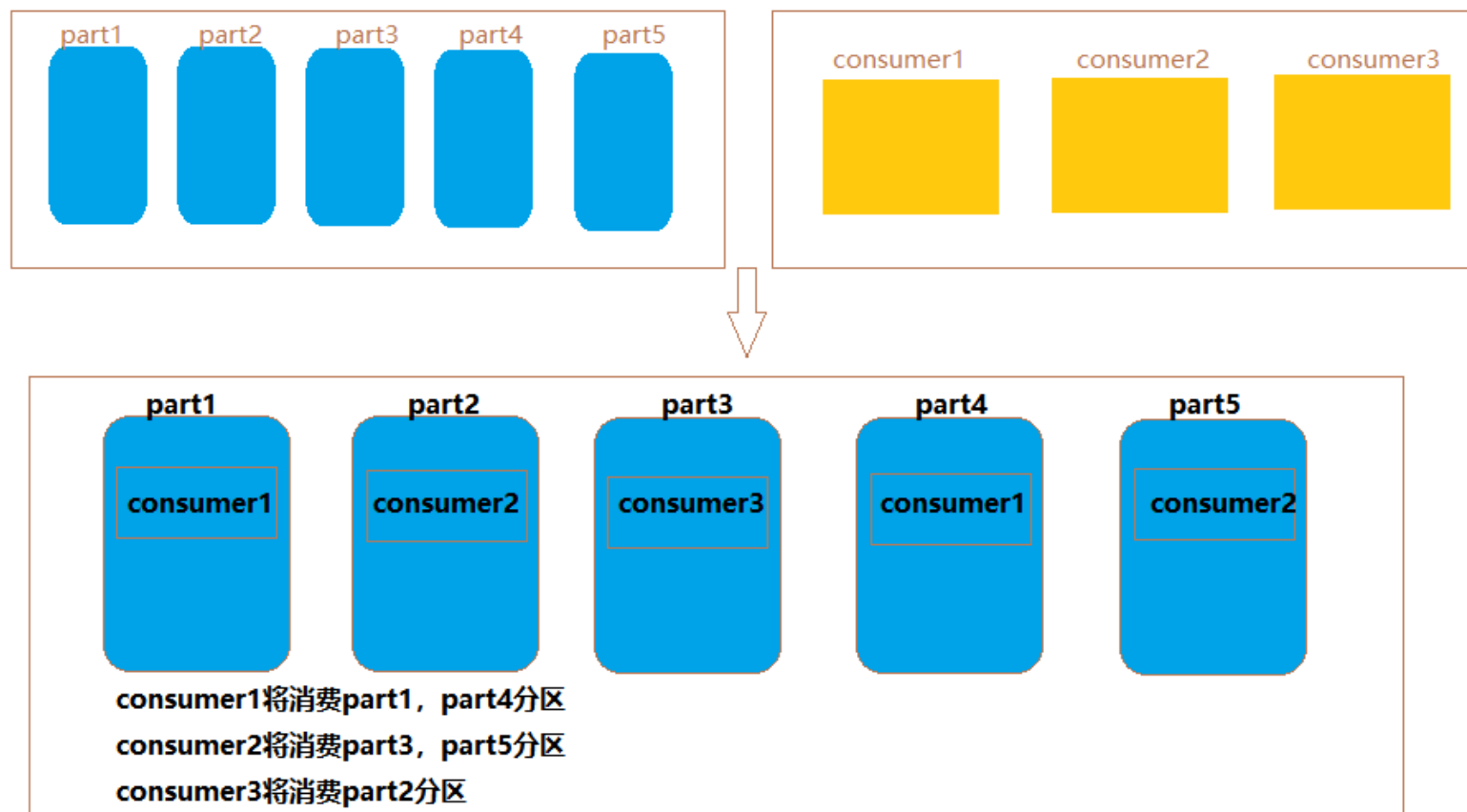
Kafka新版本consumer默认提供了两种分配策略：**range**和**round-robin**。当然Kafka采用了**range**的分配策略。

**range策略：**Range策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。然后将分区的个数除于消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区。具体例子可详见kafka副本的分配算法这一节



# kafka消费组的再平衡

## range策略



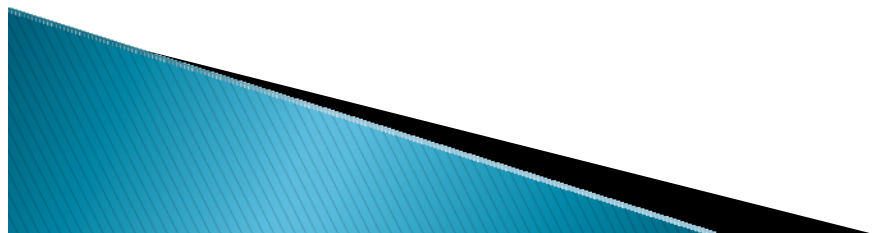
## kafka消费组的再平衡

### 谁来执行rebalance和consumer group管理

Kafka提供了一个角色：coordinator来执行对于consumer group的管理  
详情可见<https://www.cnblogs.com/heidsoft/p/7697974.html>这篇博客

### coordinator

每个consumer group都会被分配一个这样的coordinator用于组管理和位移管理。这个group coordinator比原来承担了更多的责任，比如组成员管理、位移提交保护机制等。当新版本consumer group的第一个consumer启动的时候，它会去和kafka server确定谁是它们组的coordinator。之后该group内的所有成员都会和该coordinator进行协调通信。显而易见，这种coordinator设计不再需要zookeeper了，性能上可以得到很大的提升



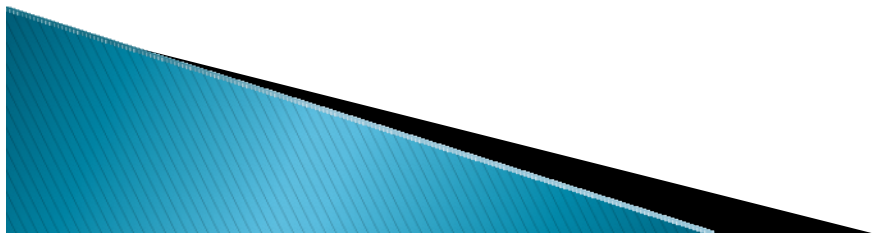
# kafka消费组的再平衡

## Rebalance过程

rebalance分为2步：Join和Sync

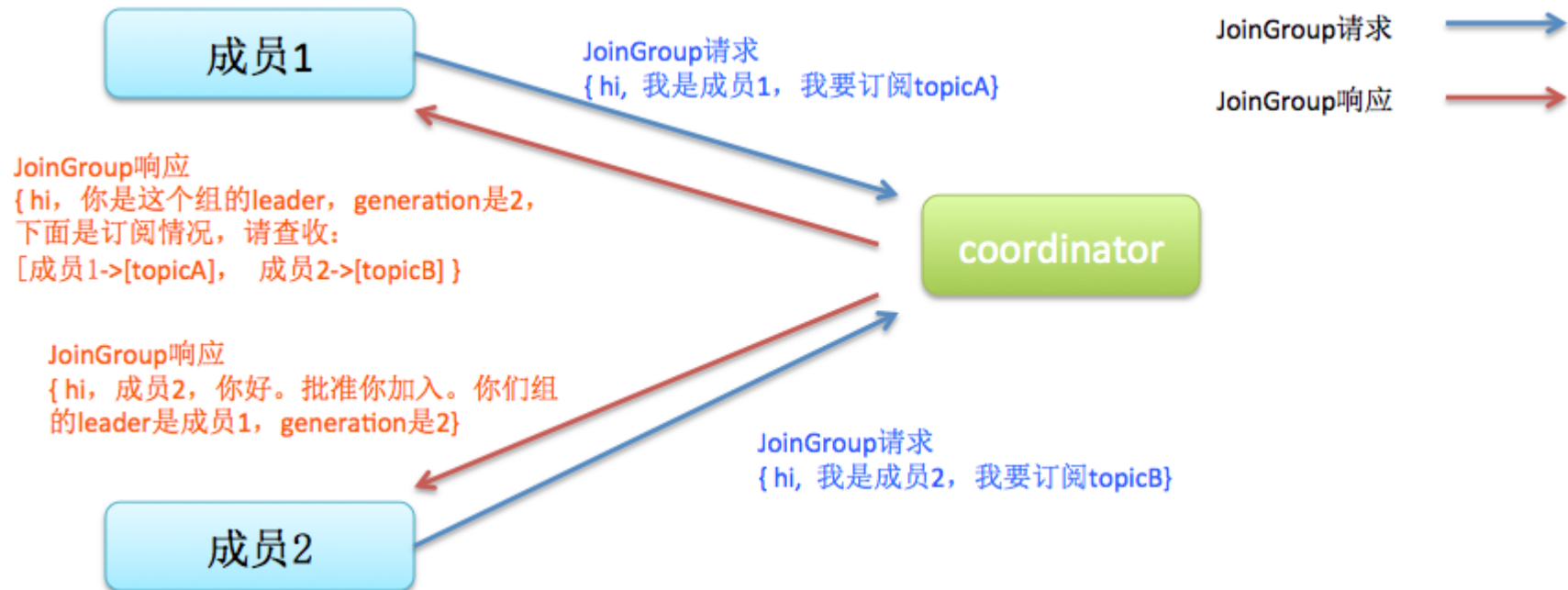
1 Join，顾名思义就是加入组。这一步中，所有成员都向coordinator发送JoinGroup请求，请求入组。一旦所有成员都发送了JoinGroup请求，coordinator会从中选择一个consumer担任leader的角色，并把组成员信息以及订阅信息发给leader——注意leader和coordinator不是一个概念。leader负责消费分配方案的制定。

2 Sync，这一步leader开始分配消费方案，即哪个consumer负责消费哪些topic的哪些partition。一旦完成分配，leader会将这个方案封装进SyncGroup请求中发给coordinator，非leader也会发SyncGroup请求，只是内容为空。coordinator接收到分配方案之后会把方案塞进SyncGroup的response中发给各个consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。



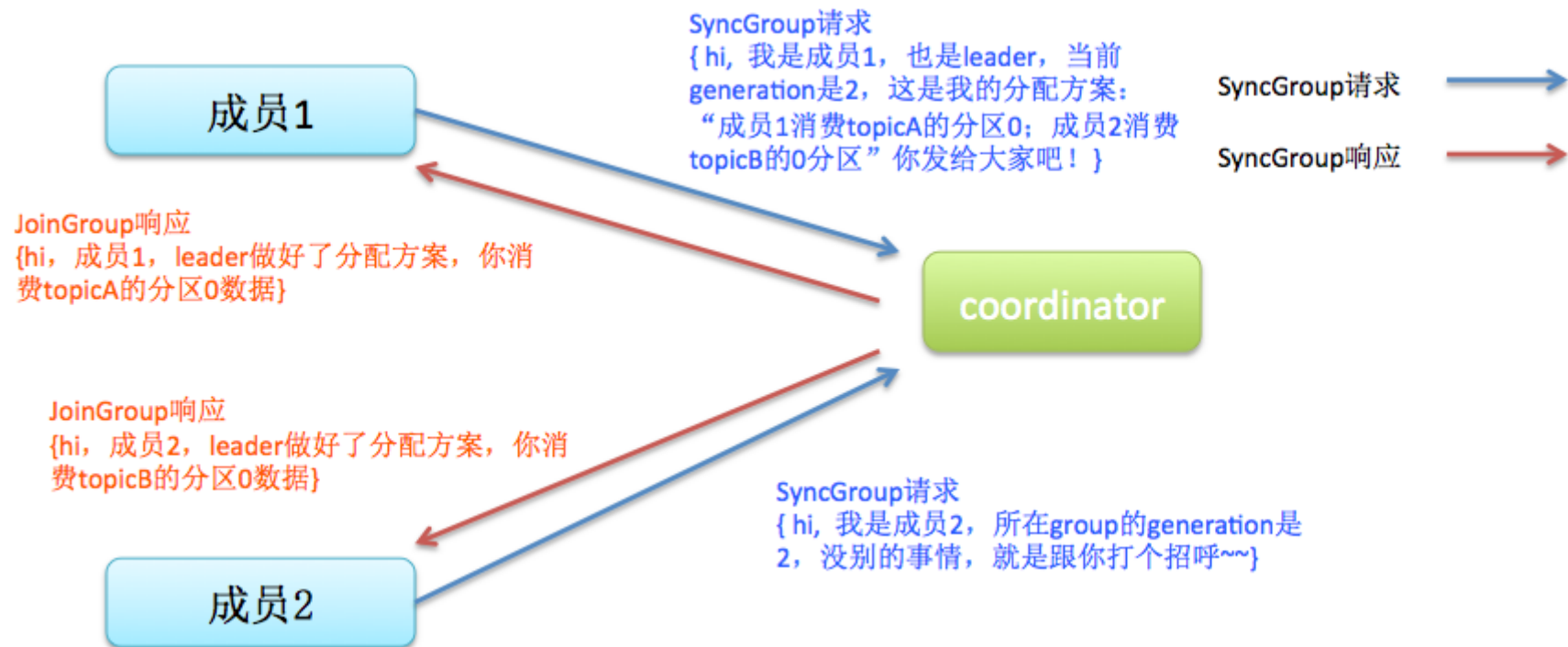
# kafka消费组的再平衡

## Rebalance过程(Join)



# kafka消费组的再平衡

## Rebalance过程(Sync)

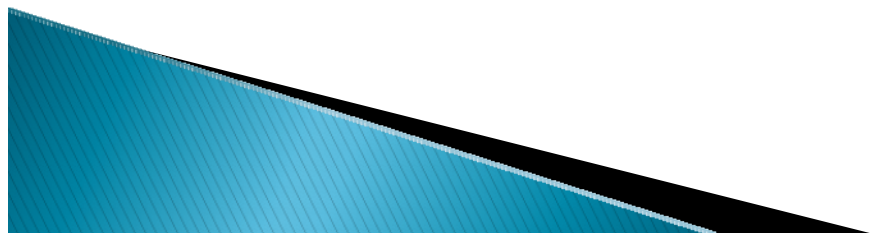


## kafka与大数据

**Kafka**是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者规模的网站中的所有动作流数据。这种动作（网页浏览，搜索和其他用户的行动）是在现代网络上的许多社会功能的一个关键因素。

这些数据通常是由于吞吐量的要求而通过处理日志和日志聚合来解决。对于像**Hadoop**的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

**Kafka**的目的是通过**Hadoop**的并行加载机制来统一线上和离线的消息处理，也是为了通过集群机来提供实时的消费。





## kafka的高可用方案

- 需要配置合适的分区数量：分区数量由topic的并发决定，并发少则一个分区就可以、并发越高，分区数越多，可以提高吞吐量
- 日志保留策略设置：当kafka broker的被写入海量消息后，会生成很多数据文件，占用大量磁盘空间，kafka默认是保留7天，建议根据磁盘情况配置，避免磁盘撑爆。段文件配置1GB，有利于快速回收磁盘空间，重启kafka加载也会加快(如果文件过小，则文件数量比较多，kafka启动时是单线程扫描目录(log.dir)下所有数据文件)
- 文件刷盘策略：为了大幅度提高producer写入吞吐量，需要定期批量写文件。建议配置：每当producer写入10000条消息时，刷数据到磁盘log.flush.interval.messages=10000
- 配置合适的副本数量可大大提高可用性，增强kafka的容错性。注意：副本数量太多同时也会影响kafka的吞吐量
- 消费者数量不能超过分区数量，建议消费者数量小于分区数量
- 可优化kafka的分配算法，是的副本均匀的在每个机器上
- 使用同步策略，但同时也会影响吞吐量

## kafka的性能测试

### 压测写入消息:

```
./kafka-producer-perf-test.sh --topic test_perf --num-records 1000000 --record-size 1000 --throughput 20000 --producer-props
```

bootstrap.servers=localhost:9092

kafka-producer-perf-test.sh 脚本命令的参数为:

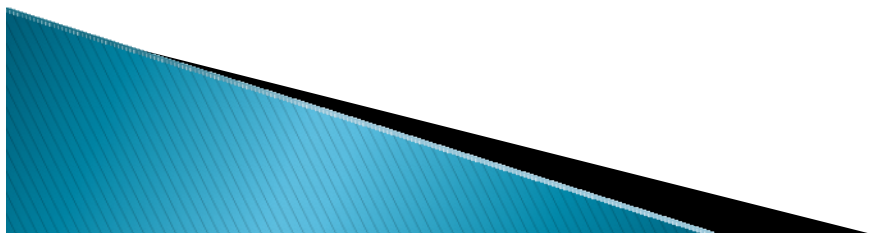
--topic topic名称, 本例为test\_perf--num-records 总共需要发送的消息数, 本例为1000000

--record-size 每个记录的字节数, 本例为1000

--throughput 每秒钟发送的记录数, 本例为20000

--producer-props bootstrap.servers=localhost:9092 发送端的配置信息, 本例只指定了kafka的链接信息

可以看到本例中, 每秒平均向kafka写入了**16.92 MB**的数据, 大概是**17737**条消息, 每次写入的平均延迟为**962.23**毫秒, 最大的延迟为**7415.00**毫秒, **831 ms**内占**50%**;



## kafka的性能测试

### 压测消费消息：

```
./kafka-consumer-perf-test.sh --zookeeper localhost:2181 --topic test_perf --  
fetch-size 1048576 --messages 1000000 --threads 1
```

kafka-consumer-perf-test.sh 脚本命令的参数为：

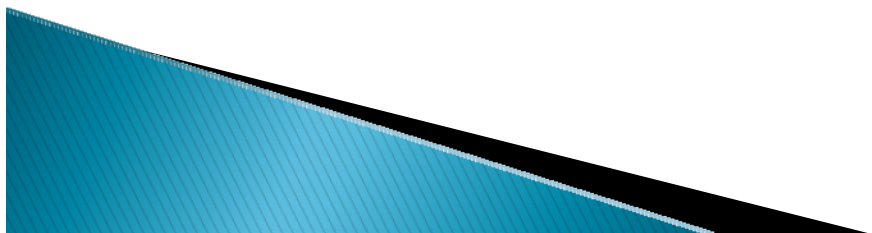
--zookeeper 指定zookeeper的连接信息，本例为localhost:2181，如果使用新的纯java客户端则使用另外的配置

--topic 指定topic的名称，本例为test\_perf

--fetch-size 指定每次fetch的数据的大小，本例为1048576，也就是1M

--messages 总共要消费的消息个数，本例为1000000，100w

可以看到本例中，总共消费了**553.8979M**的数据，每秒为**56.3936**，总共消费了**580804**条消息，每秒为**59132.9668**



# kafka的优劣势以及分别针对优劣势的应用或解决方案

## kafka的优点

kafka具有非常高的吞吐量和非常低的延迟，非常高的容错性和分布式的架构

## kafka的应用：

- 1、kafka被广泛的应用于大数据领域处理流数据进行实时计算
- 2、日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如Hadoop、Hbase、Solr等
- 3、消息系统：解耦和生产者和消费者、缓存消息等；
- 4、用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到Hadoop、数据仓库中做离线分析和挖掘；
- 5、运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告

# kafka的优劣势以及分别针对优劣势的应用或解决方案

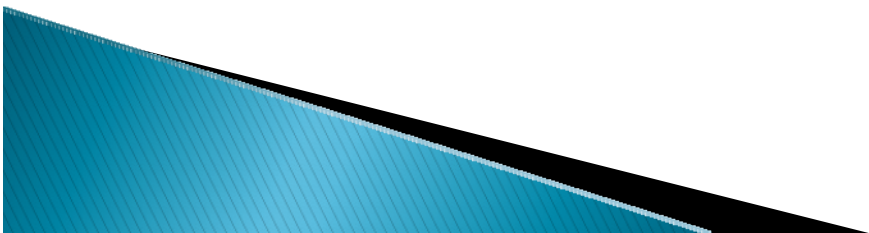
## kafka的缺点及其解决方案

1、重复消息。**Kafka**保证每条消息至少送达一次，虽然几率很小，但一条消息可能被送达多次。

解决方案：将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可

2、消息乱序。**Kafka**某一个固定的**Partition**内部的消息是保证有序的，如果一个**Topic**有多个**Partition**，**partition**之间的消息送达不保证有序。

3、复杂性。**Kafka**需要**Zookeeper**的支持，**Topic**一般需要人工创建，部署和维护比一般**MQ**成本更高。



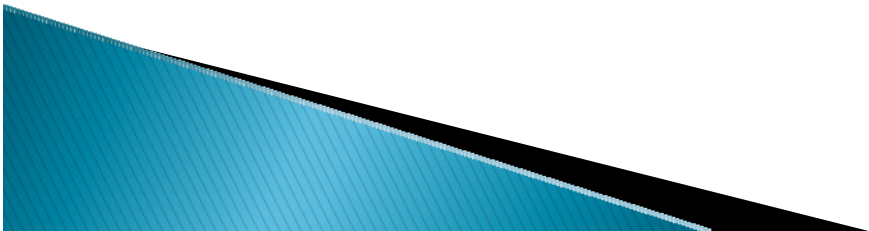
# springboot整合kafka

## 1、引入maven依赖

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>1.1.1.RELEASE</version>
</dependency>
```

## 2、创建消息的实体类

```
public class Message {
    private Long id; //id
    private String msg; //消息
    private Date sendTime; //时间戳
}
```



# springboot整合kafka

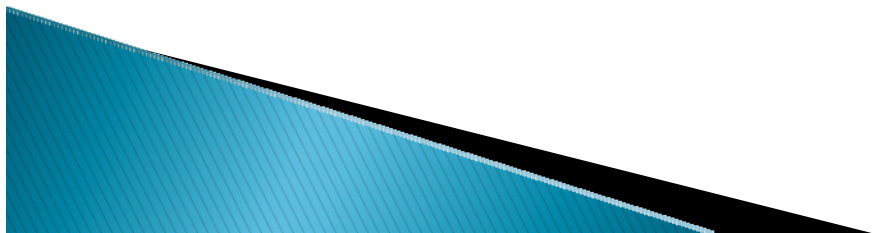
## 3、消息发送类 KafkaSender

@Autowired

private KafkaTemplate<String, String> kafkaTemplate;

//发送消息方法

```
public void send() {  
    Message message = new Message();  
    message.setId(System.currentTimeMillis());  
    message.setMsg(UUID.randomUUID().toString());  
    message.setSendTime(new Date());  
    kafkaTemplate.send("zhisheng", gson.toJson(message));  
}
```





# springboot整合kafka

## 4、消息接收类 KafkaReceiver

```
@Component
@Slf4j
public class KafkaReceiver {

    @KafkaListener(topics = {"zhisheng"})
    public void listen(ConsumerRecord<?, ?> record) {

        Optional<?> kafkaMessage = Optional.ofNullable(record.value());

        if (kafkaMessage.isPresent()) {

            Object message = kafkaMessage.get();

            log.info("----- record =" + record);
            log.info("----- message =" + message);
        }
    }
}
```

# springboot整合kafka

## 5、配置参数application.yml

```
#===== kafka =====
# 指定kafka 代理地址，可以多个
spring.kafka.bootstrap-servers=192.168.153.135:9092

#===== provider =====

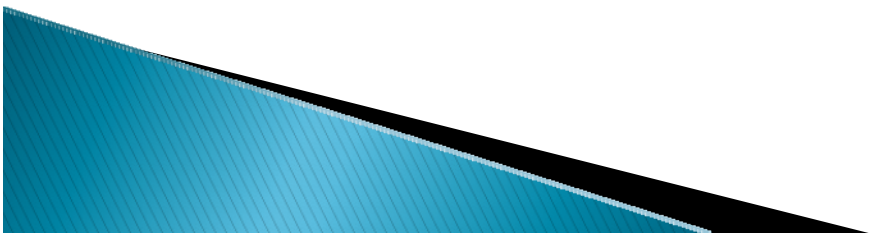
spring.kafka.producer.retries=0
# 每次批量发送消息的数量
spring.kafka.producer.batch-size=16384
spring.kafka.producer.buffer-memory=33554432

# 指定消息key和消息体的编解码方式
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

#===== consumer =====
# 指定默认消费者group id
spring.kafka.consumer.group-id=test-consumer-group

spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.auto-commit-interval=100

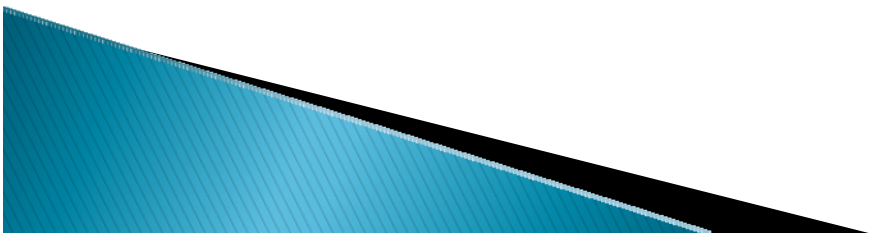
# 指定消息key和消息体的编解码方式
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```



## kafka和前端通信

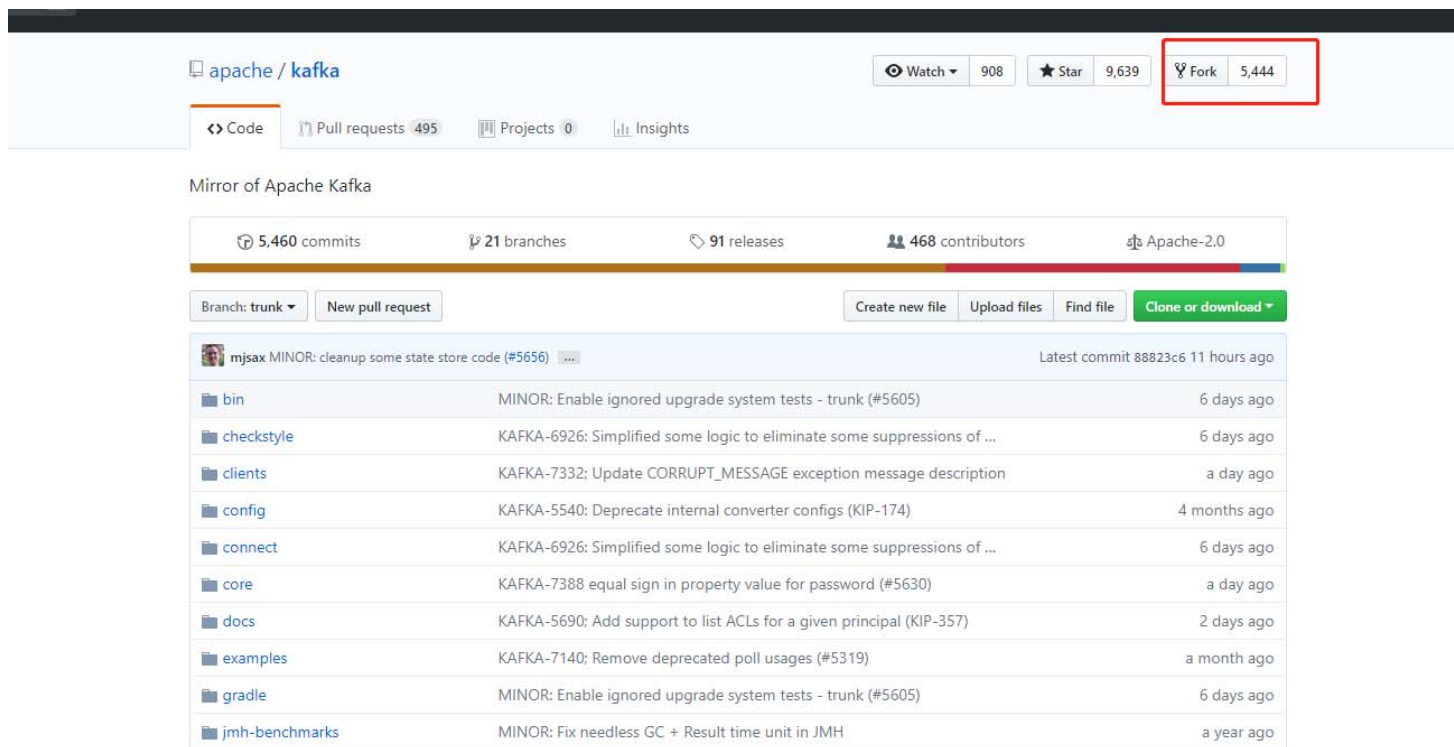
通过使用websocket来使kafka和前端的实时通讯  
可参考以下博客:

<https://www.cnblogs.com/nayt/p/6931499.html>



# kafka源码环境搭建

1、从github上搜索kafka项目，然后fork到自己的账号上



apache / kafka

Watch 908 Star 9,639 Fork 5,444

Code Pull requests 495 Projects 0 Insights

Mirror of Apache Kafka

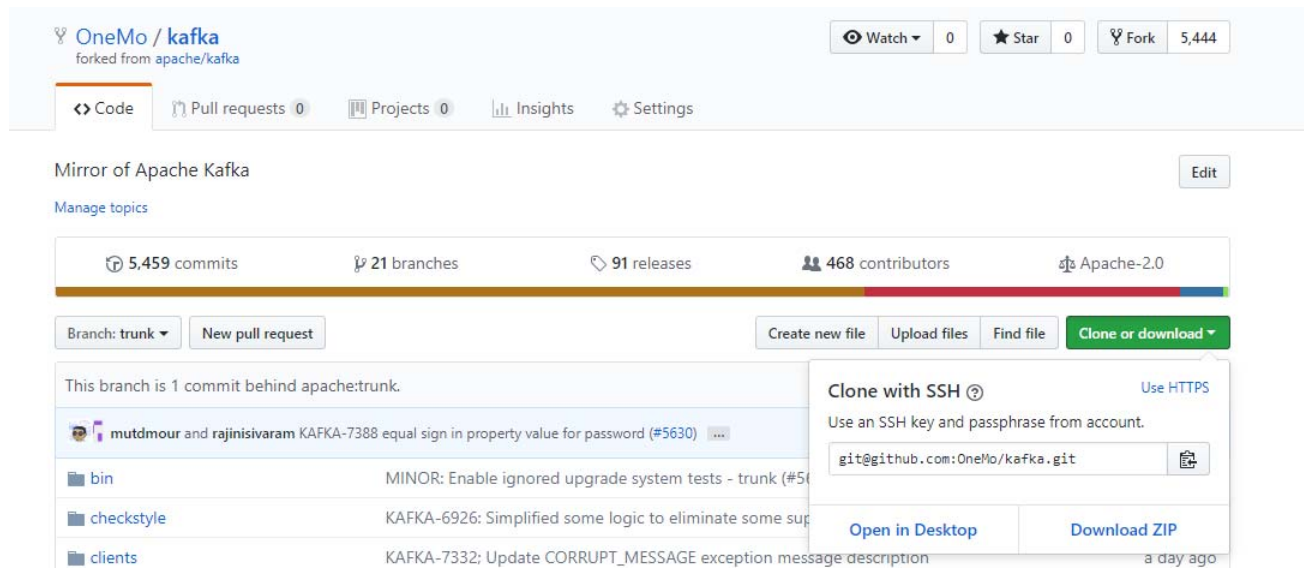
5,460 commits 21 branches 91 releases 468 contributors Apache-2.0

Branch: trunk New pull request Create new file Upload files Find file Clone or download

mjsax	MINOR: cleanup some state store code (#5656)	Latest commit 88823c6 11 hours ago
bin	MINOR: Enable ignored upgrade system tests - trunk (#5605)	6 days ago
checkstyle	KAFKA-6926: Simplified some logic to eliminate some suppressions of ...	6 days ago
clients	KAFKA-7332: Update CORRUPT_MESSAGE exception message description	a day ago
config	KAFKA-5540: Deprecate internal converter configs (KIP-174)	4 months ago
connect	KAFKA-6926: Simplified some logic to eliminate some suppressions of ...	6 days ago
core	KAFKA-7388 equal sign in property value for password (#5630)	a day ago
docs	KAFKA-5690: Add support to list ACLs for a given principal (KIP-357)	2 days ago
examples	KAFKA-7140: Remove deprecated poll usages (#5319)	a month ago
gradle	MINOR: Enable ignored upgrade system tests - trunk (#5605)	6 days ago
jmh-benchmarks	MINOR: Fix needless GC + Result time unit in JMH	a year ago

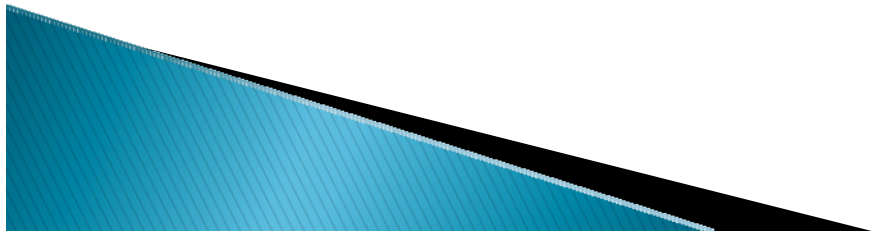
# kafka源码环境搭建

## 2、通过git 克隆到本地



The screenshot shows the GitHub repository page for `OneMo / kafka`, which is a fork of `apache/kafka`. The repository has 5,459 commits, 21 branches, 91 releases, 468 contributors, and 5,444 forks. The 'Code' tab is selected, and the 'Clone or download' button is highlighted. A modal window is open, showing the 'Clone with SSH' option, which uses the SSH key and passphrase from the account. The repository URL is `git@github.com:OneMo/kafka.git`. The modal also shows the 'Open in Desktop' and 'Download ZIP' options.

```
wdfu@DESKTOP-50QS39T MINGW64 ~/Desktop
$ git clone git@github.com:OneMo/kafka.git
```



# kafka源码环境搭建

## 3、搭建gradle环境

因为kafka是采用gradle构建的项目。所以需要搭建gradle的环境，跟搭建maven环境差不多，下载后配置下环境变量就OK了

## 4、修改gradle下载jar地址改为阿里的Maven源

在kafka项目根目录打开build.gradle文件

<http://maven.aliyun.com/nexus/content/groups/public/>

```
buildscript {
    repositories {
        maven { url 'http://maven.aliyun.com/nexus/content/groups/public/' }
        jcenter()
    }
    apply from: file('gradle/buildscript.gradle'), to: buildscript

    dependencies {
        // For Apache Rat plugin to ignore non-Git files
        classpath "org.ajoberstar:grgit:1.9.3"
        classpath 'com.github.ben-manes:gradle-versions-plugin:0.17.0'
        classpath 'org.scoverage:gradle-scoverage:2.3.0'
        classpath 'com.github.jengelman.gradle.plugins:shadow:2.0.4'
        classpath 'org.owasp:dependency-check-gradle:3.1.2'
    }
}

apply from: "$rootDir/gradle/dependencies.gradle"

allprojects {
    repositories {
        maven { url 'http://maven.aliyun.com/nexus/content/groups/public/' }
    }

    apply plugin: 'idea'
    apply plugin: 'org.owasp.dependencycheck'
    apply plugin: 'com.github.ben-manes.versions'
}
```

# kafka源码环境搭建

## 5、build kafka项目

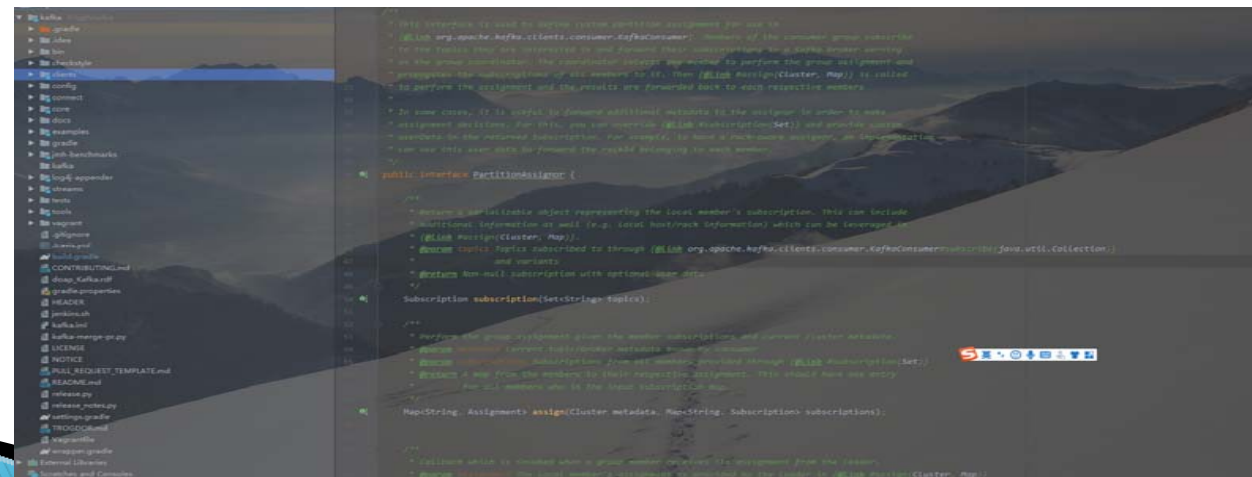
进入kafka项目的根目录在命令行中执行gradle idea命令 然后在通过idea打开项目

```
E:\git\kafka>gradle idea
Starting a Gradle Daemon, 1 busy Daemon could not be reused, use --status for details

> Configure project :
Building project 'core' with Scala version 2.11.12
Building project 'streams-scala' with Scala version 2.11.12

> Task :idea
Generated IDEA project at file:///E:/git/kafka/kafka.ipr

BUILD SUCCESSFUL in 2m 24s
27 actionable tasks: 27 executed
E:\git\kafka>cd ../
```





# Kafka安装

➤ 下载

<http://kafka.apache.org/downloads.html>

➤ 解压

```
tar -zxvf kafka_2.10-0.8.1.1.tgz
```

➤ 启动服务

首先启动zookeeper服务

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

启动Kafka

```
bin/kafka-server-start.sh config/server.properties >/dev/null 2>&1 &
```

➤ 创建topic

创建一个"test"的topic，一个分区一个副本

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

➤ 查看主题

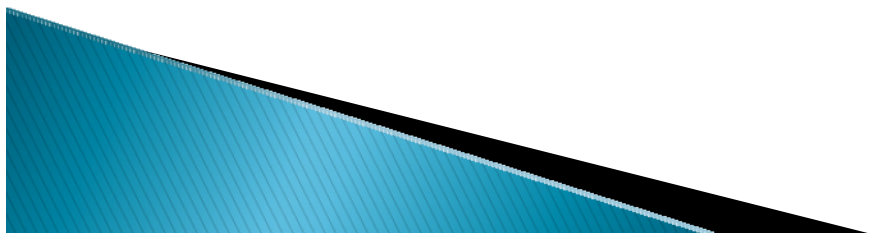
```
bin/kafka-topics.sh --list --zookeeper localhost:2181
```

➤ 查看主题详情

```
bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test
```

➤ 删除主题

```
bin/kafka-run-class.sh kafka.admin.TopicCommand --delete --topic test --zookeeper  
192.168.1.161:2181
```



## Kafka客户端操作

### ➤ 创建生产者 producer

`bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test`

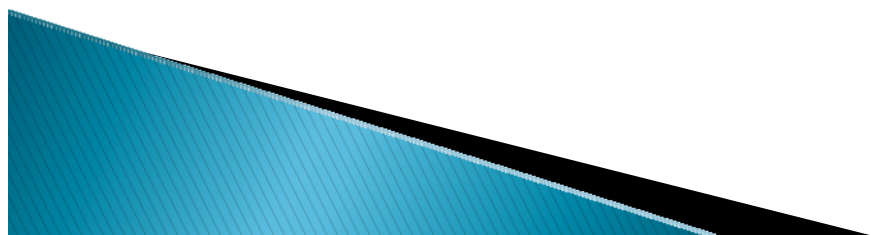
### ➤ 创建消费者 consumer

`bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning`

### ➤ 参数使用帮组信息查看：

生产者参数查看：`bin/kafka-console-producer.sh`

消费者参数查看：`bin/kafka-console-consumer.sh`



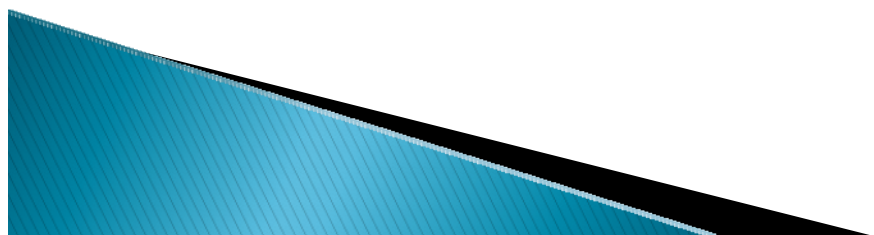
## kafka集群安装

- 安装zk集群
- 修改配置文件详情

broker.id: 唯一, 填数字

host.name: 唯一, 填服务器

zookeeper.connect=192.168.40.134:2181,192.168.40.132:2181,192.168.40.133:2181



# Kafka的测试效果

## 性能测试

目前我已经在虚拟机上做了性能测试。

测试环境：cpu: 双核 内存：2GB 硬盘：60GB

测试指标	性能相关说明	结论
消息堆积压力测试	<p>单个kafka broker节点测试，启动一个kafka broker和Producer，Producer不断向broker发送数据，</p> <p>直到broker堆积数据为18GB为止(停止Producer运行)。启动Consumer，不间断从broker获取数据，</p> <p>直到全部数据读取完成为止，最后查看Producer == Consumer数据，没有出现卡死或broker不响应现象</p>	<p>数据大量堆积不会出现broker卡死或不响应现象</p>
生产者速率	1.200byte/msg,4w/s左右。2.1KB/msg,1w/s左右	性能上是完全满足要求，其性能主要由磁盘决定
消费者速率	1.200byte/msg,4w/s左右。2.1KB/msg,1w/s左右	性能上是完全满足要求，其性能主要由磁盘决定

## kafka学习资源推荐

➤ 书籍： kafka技术内幕 作者： 郑奇煌

➤ 博客：

<https://www.cnblogs.com/huxi2b/>

