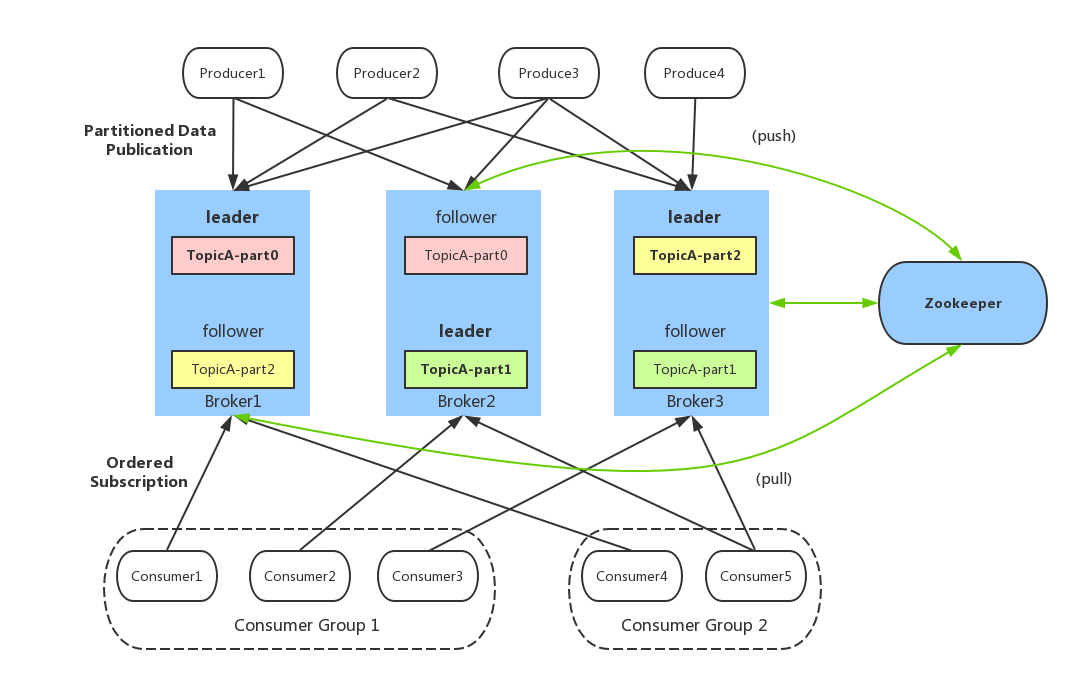
Apache Kafka:下一代分布式消息系统

Apache Kafk是分布式发布-订阅消息系统，其内部实现是分布式、分区的和可复制的提交日志服务，与传统消息挺相比：

* 被设计为一个分布式系统，易于向外扩展
* 同时为发布和订阅提供高吞吐量
* 支持多订阅者，当失败时能自动平衡消费者
* 将消息持久化磁盘，因此可用于批量消费，例如ETL以及实时应用程序等

# 1. Kafka架构

Kafka的系统架构图如下所示：



它的架构包括以下组件：

1. Topic(话题)，特定类型的消息流，消息是字节的有效负载（Payload），话题是消息的分类名和种子（Feed）名
2. Producer(生产者)，能够发布消息到话题的任何对象，生产者可以选择序列化方法对消息内容编码
3. 已发布的消息保存在一组服务器中，它们被称为代理（Broker）或Kafka集群
4. Consumer(消费者)，可以订阅一个或者多个话题，并从Broker拉数据，从而消费这些已经发布的消息。
5. Consumer Group，在Kakfa中每个Consumer都属于一个Consumer Group，每条消息只能被Group中的一个Consumer消费，但是可以被多个Consumer Group消费
6. Patition，物理上的概念，每个Topic包含一个或者多个Partition
7. Replica，为了保证可用性，Partition可以有多个副本
8. Leader，Partition有多个副本时，选择其中一个作为Leader，Producer和Consumer只和Leader交互。Follower从Leader中复制数据。
9. Controller，Kafka其中一个服务器，用来进行Leader Election以及各种Failover。

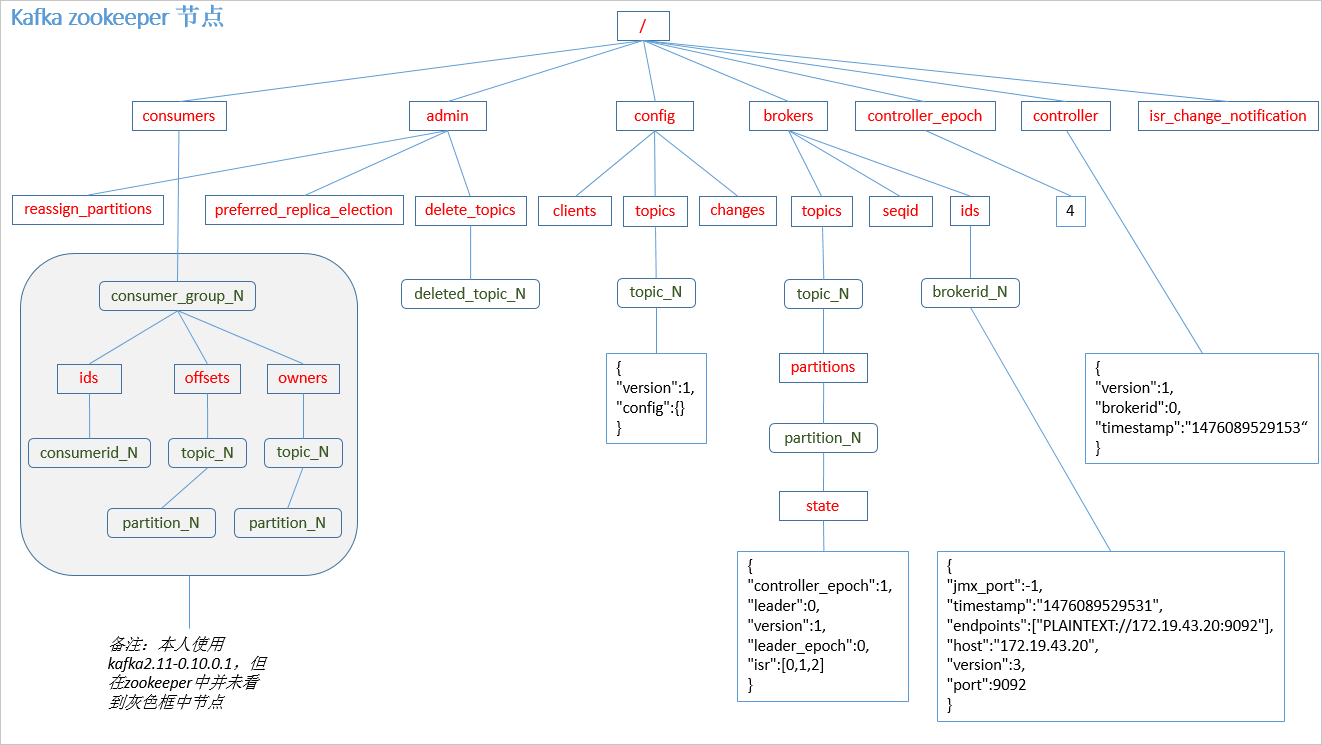
在Kafka集群中，通常包括多个代理，为了负载均衡，将话题分成多个分区，每个代理存储一个或者多个分区。多个生产者和消费者同时生产和获取消息。

# 2. Zookeeper

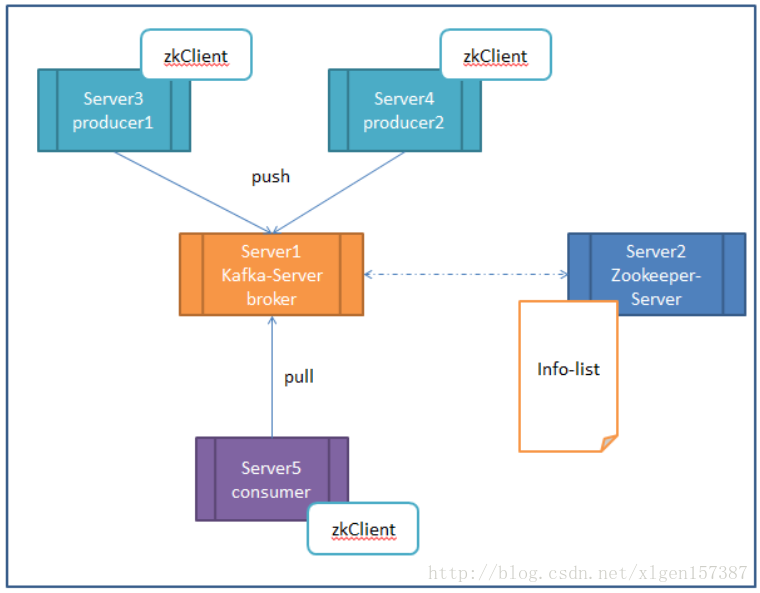
Apache Kafka为多个服务器的分布式系统，其使用Zookeeper构建可靠、高可用、容错和分布式的协调服务。

Apache Zookeeper是一个分布式、分层级的文件系统，能促进客户端间的松耦合，并提供最终一致的，类似于传统文件系统中文件和目录的Znode视图，它提供了事件驱动模型，客户端能观察特定的Znode的变化，例如Znode增加子节点。使用Zookeeper管理和协调Kafka Broker，当Kafka系统中新增代理或者某个代理故障失效时，Zookeeper服务将通知生产者和消费者，据此开始于其他代理协调工作。

在Kafka中使用Zookeeper来存储一些meta信息（集群broker/topic/parition等），并使用Zookeeper watch机制来发现meta信息的变更并做相应的动作（比如consumer失效，触发负载均衡），Kafka在Zookeeper中的存储结构如下图所示：



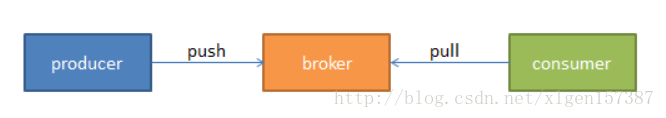
无论是Kafka集群，还是Producer和Consumer都依赖于Zookeeper来保证系统可用性集群保存meta信息，kafka使用zookeeper作为其分布式协调框架，将消息生产、消息存储和消息消费结合在一起，建立起生产者和消费者的订阅关系，并实现生产者和消费者的负载均衡，如下图所示：



1. Producer端使用zookeeper用来发现broker列表以及和Topic下每个Partition Leader建立Socket连接并发送消息。
2. Broker端使用ZK来注册broker信息，监测partition Leader的存活性
3. Consumer端使用Zookeeper来注册Consumer信息，其中包括Consumer消费的partition列表等，同时也用来发现broker列表，并和partition Leader建立Socket连接，并获取消息。

# 3. Kafka设计原理

Apache Kafka需要支持较大的数据量，且具备良好的容错能力，其基本执行：



生产者将数据生产出来，交给broker进行存储，消费者需要消费数据就从broker中获取数据，然后完成一系列对数据的处理操作。

## 3.1 数据存储设计

Apache Kafka的存储布局非常简单，Topic的每个分区对应逻辑日志，物理上一个日志为相同大小的一组分段文件。每次生产者发布消息到一个分区，Broker就将消息追加到最后一个Segement文件中。当发布的消息数量达到设定值或者经过一定时间后，Segment文件真正写入磁盘中，写入完成后，消息公开给消费者。日志文件的配置目录由参数：

*log.dirs=/tmp/kafka-logs*

默认目录/tmp/kafka-logs，其文件结构如下：

*[topic2-0]$ tree*

.

├── 00000000000000000000.index

├── 00000000000000000000.log

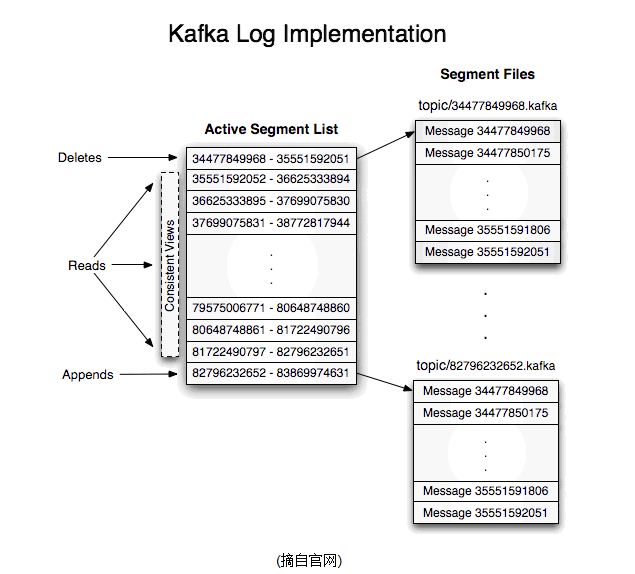
├── 00000000000000000000.timeindex

与传统的消息系统不同的时，Kafka系统中存储的消息没有明确的消息id。

消息通过日志中的逻辑偏移量来公开，这样就避免维护配套密集寻址，用于映射消息ID到实际消息地址的随机存取索引结构的开销。消息ID是增量的，但不连续，要计算下一消息的ID，可以在其逻辑偏移的基础上加上当前消息的长度。

消费者始终从特定分区顺序地获取消息，如果消费者知道特定消息的偏移量，也就说明消费者已经消费了之前所有消息。消费者向代理发出一步拉请求，准备字节缓冲区用于消费，每个异步拉请求都包含要消费的消息偏移量。Kafka 利用SendFile API高效地从代理的日志段文件中分发字节给消费者。

如果一个Topic的名称为my\_topic，有2个partition，那么日志将保持在my\_topic\_0和my\_topic\_1两个目录中。日志文件中保存log entries的序列，每个log entry格式为"4个字节的数字N表示消息的长度" + "N个字节的消息内容";每个日志都有一个offset来唯一的标记一条消息,offset的值为8个字节的数字,表示此消息在此partition中所处的起始位置..每个partition在物理存储层面,有多个log file组成(称为segment).segmentfile的命名为"最小offset".kafka.例如"00000000000.kafka";其中"最小offset"表示此segment中起始消息的offset。



1）Segment文件

其中每个partition中所持有的segments列表信息都会存储在zookeeper中。当segment文件尺寸达到一定阈值时（默认1G），将会创建一个新的文件；当Buffer中消息的条数达到阈值时将会触发日志信息flush到日志文件中，同时如果距离最近一次flush的时间差达到阈值时，也会触发flush到日志文件中。如果broker失效，极有可能会丢失那些尚未flush到文件的消息。因为Server意外情况，仍然会导致log文件格式的破坏（文件尾部），那么就要求Server启动时需要检测最后一个Segment的文件结构是否合法并进行必要的修复。在Partition中每条Message包含以下三个属性：

1. Offset，Message在Partition中偏移量，可以认为Offset是Partition中message的Id
2. MessageSize，表示内容data的大小
3. Data，Message的具体内容

Partition的数据文件由以下格式的Message组成，按offset由小到达排列在一起。

获取消息时，需要指定offset和最大chunk尺寸，offset用来表示消息的起始位置，chunk Size用来表示最大获取消息的总长度（间接的表示消息条数）。根据offset，可以找到此消息所在segment文件，然后根据segment的最小offset取差值，得到它在file中的相对位置，直接读取输出即可。

2）数据文件索引

数据文件分段可以在一个较小的数据文件中，查找对应offset的Message，但是依然需要顺序扫描才能找到对应offset的Message。为了进一步的提高查找的效率，Kafka为每个分段后的数据文件建立索引文件，文件后缀名.index。

在索引文件中包含若干个索引条目，每个条目表示数据文件中一条Message的索引，包括两个部分：

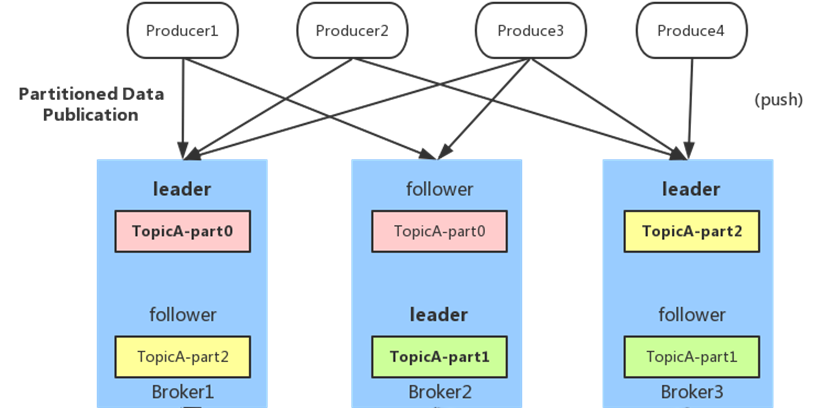
* 相对Offset，分段后数据文件其实Offset不为0，相对offset表示这条Messsage相对于其所属数据文件中最小offset的值
* Position，表示该条message在数据文件中的绝对位置。

但是在index文件中没有为数据文件中的每条message建立索引，而是采用稀疏存储的方式，每隔一定字节的数据建立一条索引，避免索引文件占用过多空间。

日志文件的删除策略，会启动一个后台线程定期扫描log file列表，把保存时间超过阈值的文件直接删除。为了避免删除文件时仍有read操作，采取copy-on-write方式。

## 3.2 生产者设计

Producer将消息发布到指定的Topic中，同时决定该消息属于哪个Partition（基于Round-Robin方式或者其他算法），如下图所示：



1. 负载均衡

Topic由多个Partition组成，均衡分布到不同的Broker上，Producer通过随机或者hash等方式将消息平均发送到多个Partition上，以实现负载均衡。Producer会和Partition Leader保持socket连接，中间不会经过路由层，Producer客户端决定消息被路由到哪个partition上。

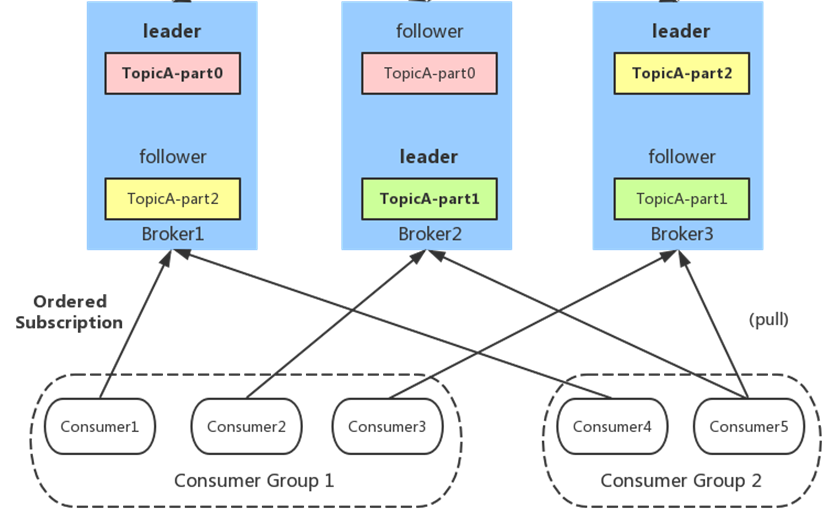
Partition Leader的位置(host: port)注册在zookeeper上，Producer作为zk client，通过watch来监听partition Leader的变更事件。

1. 批量发送

Producer端可以在内存中合并多条消息，批量发送给Broker，减少Broker存储消息的IO次数，从而提高消息的吞吐量，但是一定程度上影响消息的实时性。

## 3.3 消费者设计

Consumer端向broker发送fetch请求，并告知其获得消息的offset，此后消费者会获得一定条数的消息，Consumer端也可以重置offset来重置消费消息，如下图：

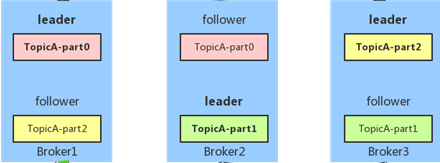


在Kafka中，采用pull方式即Consumer在和broker建立连接之后，主动去pull消息，这种模式下Consumer端可以根据自己的消费能力适时的fetch消息并处理，且可以控制消息消费的进度（offset），此外消费者可以良好的控制消息消费的数量。

Consumer Group是Kafka提供的可扩展且具有容错性的消费者机制，在组内可以有多个消费者或消费者实例，共享一个ID(group ID)。组内所有的消费者协调在一起消费订阅主题的所有分区。当然每个分区只能由同一个消费组内的一个Consumer来消费。

## 3.4 Replication设计

Kafka将每个partition数据复制到多个server上，任何partition有一个leader和多个follower，Replication的个数可以通过broker配置文件来设定，如下图：



Leader处理所有的read-write请求，Follower需要和Leader保持一致，Follower和Consumer一样将消费消息保存在本地日志中。Leader负责跟踪所有的follower状态，如果Follower落后太多或者失败，Leader将其从replicas同步列表中删除。当所有Follower都将一条消息保存成功，此消息才被认为是Committed，此时Consumer才能消费它。当Leader失效时，需在followers中选取出新的Leader，选举时考虑负责均衡，尽可能让Leader均衡分布在多个Broker中。Replica均衡分布到整个集群的算法如下：

* 将所有Broker（n）和待分配的Partition排序
* 将第i个partition分配到（i mod n）个Broker上
* 将第i个partition的j个Replica分配到（(i+j) mod n）个Broker上

## 3.5 HA基本原理

1） Broker HA

Broker集群由Zookeeper维护，并选举出一个Controller，所有partition的Leader选举都有Controller决定，将Leader的变更直接通过rpc方式通知需要为此做出响应的brokers.

Controller负载增删topic和partition replica的重新分配。Controller在Zookeeper上注册watch，一旦broker宕机，其对应在zookeeper上的临时节点自动被删除，Controller对宕机的broker上所有partition重新分配新的leader。如果controller宕机，其他broker通过zk选举出新的controller，然后同样对宕机broker上所有partition重新分配新的leader.

2）Partition HA

Partition Leader所在的broker宕机，broker controller根据动态维护ISR，会重新在剩下的broker机器中选出ISR中新的成员成为leader。在ISR中，至少有一个follwer，则可以确保已经committed的数据不丢失，否则选择任意一个replica作为leader。

https://www.colabug.com/1217880.html

https://blog.csdn.net/boonya/article/details/54783294

https://blog.csdn.net/ronmy/article/details/60467147