

# 优极限

“极限教育，挑战极限”

[www.yjxxt.com](http://www.yjxxt.com)

极限教育，挑战极限。优极限是一个让 95% 的学生年薪过 18 万的岗前培训公司，让我们的学员具备优秀的互联网技术和职业素养，勇攀高薪，挑战极限。公司位于上海浦东，拥有两大校区，共万余平。累计培训学员超 3 万名。我们的训练营就业平均月薪 19000，最高年薪 50 万。

核心理念：让学员学会学习，拥有解决问题的能力，拿到高薪职场的钥匙。

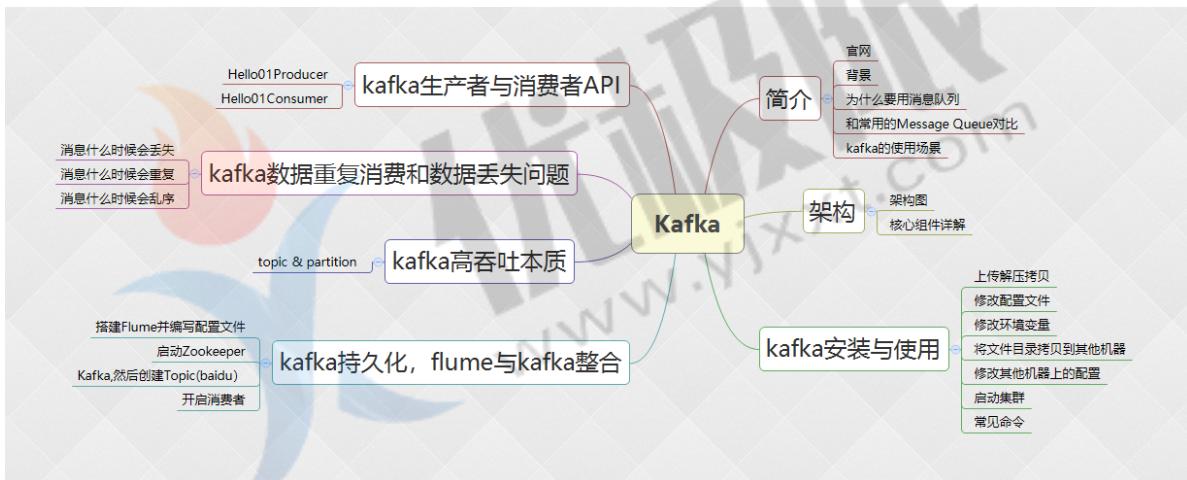
项目驱动式团队协作、一对一服务、前瞻性思维、教练式培养模型-培养你成为就业明星。首创的老学员项目联盟给学员充分的项目、技术支撑，利用优极限平台这根杠杆，不断挑战极限，勇攀高薪，开挂人生。

扫码关注优极限微信公众号：

(获取最新技术相关资讯及更多源码笔记)



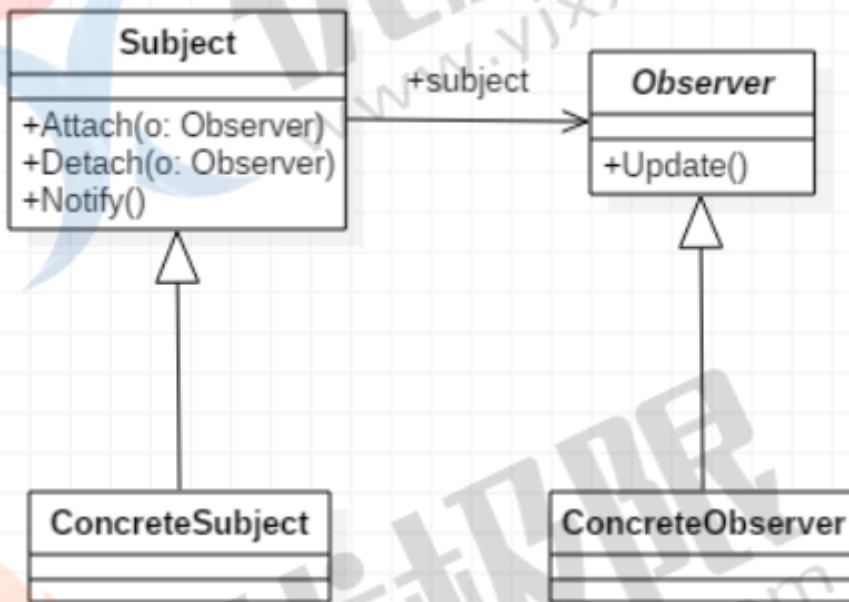
# Kafka



## 1. 异步通信原理

### 1.1. 观察者模式

- 观察者模式 (Observer) , 又叫发布-订阅模式 (Publish/Subscribe)
- 定义对象间一种一对多的依赖关系, 使得每当一个对象改变状态, 则所有依赖于它的对象都会得到通知并自动更新。
- 一个对象 (目标对象) 的状态发生改变, 所有的依赖对象 (观察者对象) 都将得到通知。
- 



- 现实生活中的应用场景
  - 京东到货通知

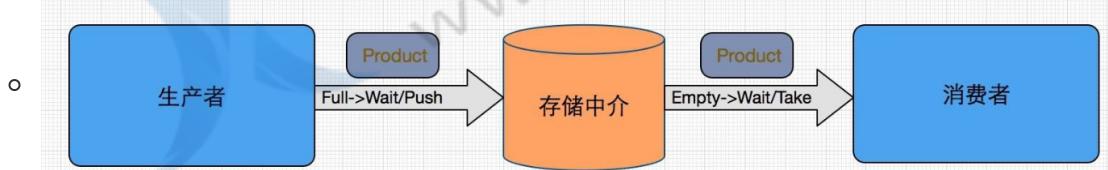


- 《鸡毛信》



## 1.2. 生产者消费者模式

- 传统模式
  - 生产者直接将消息传递给指定的消费者
  - 耦合性特别高，当生产者或者消费者发生变化，都需要重写业务逻辑
- 生产者消费者模式
  - 通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而是通过阻塞队列来进行通讯



- 数据传递流程
  - 生产者消费者模式，即N个线程进行生产，同时N个线程进行消费，两种角色通过内存缓冲区进行通信，
  - 生产者负责向缓冲区里面添加数据单元
  - 消费者负责从缓冲区里面取出数据单元
    - 一般遵循先进先出的原则

## 1.3. 缓冲区

- 解耦
  - 假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产生依赖
- 支持并发
  - 生产者直接调用消费者的某个方法过程中函数调用是同步的
  - 万一消费者处理数据很慢，生产者就会白白糟蹋大好时光
- 支持忙闲不均
  - 缓冲区还有另一个好处。如果制造数据的速度时快时慢，缓冲区的好处就体现出来了。
  - 当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。
  - 等生产者的制造速度慢下来，消费者再慢慢处理掉。

## 1.4. 数据单元

- 关联到业务对象
  - 数据单元必须关联到某种业务对象
- 完整性
  - 就是在传输过程中，要保证该数据单元的完整
- 独立性
  - 就是各个数据单元之间没有互相依赖
  - 某个数据单元传输失败不应该影响已经完成传输的单元；也不应该影响尚未传输的单元。
- 颗粒度
  - 数据单元需要关联到某种业务对象。那么数据单元和业务对象应该处于的关系（一对一？一对多）
  - 如果颗粒度过小会增加数据传输的次数
  - 如果颗粒度过大会增加单个数据传输的时间，影响后期消费

## 2. 消息系统原理

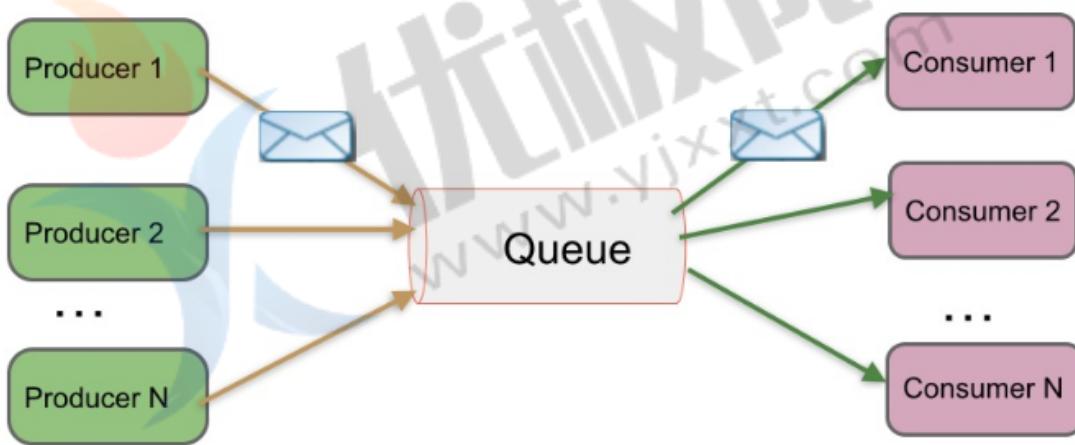
一个消息系统负责将数据从一个应用传递到另外一个应用，应用只需关注于数据，无需关注数据在两个或多个应用间是如何传递的。

### 2.1. 点对点消息传递

- 在点对点消息系统中，消息持久化到一个队列中。此时，将有一个或多个消费者消费队列中的数据。但是一条消息只能被消费一次。
- 当一个消费者消费了队列中的某条数据之后，该条数据则从消息队列中删除。
- 该模式即使有多个消费者同时消费数据，也能保证数据处理的顺序。
- 基于推送模型的消息系统，由消息代理记录消费状态。

- 消息代理将消息推送(push)到消费者后，标记这条消息为已经被消费，但是这种方式无法很好地保证消费的处理语义。

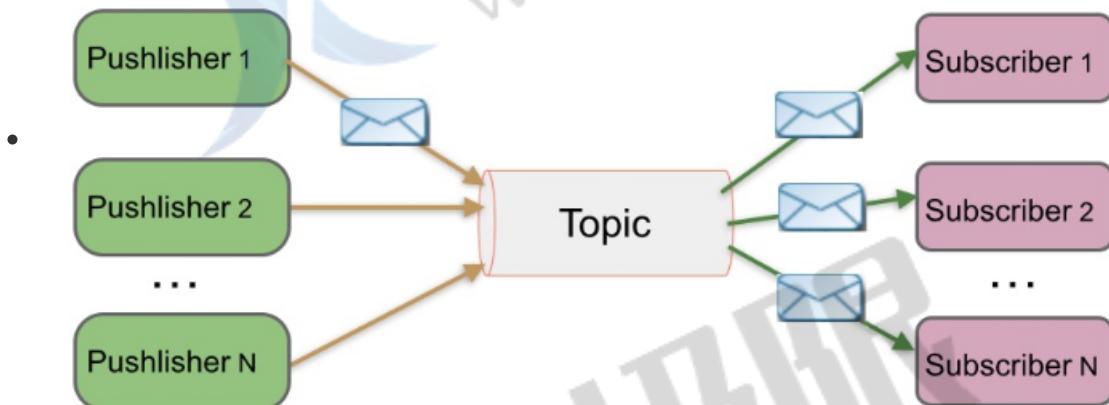
## 消息队列-点对点



### 2.2. 发布订阅消息传递

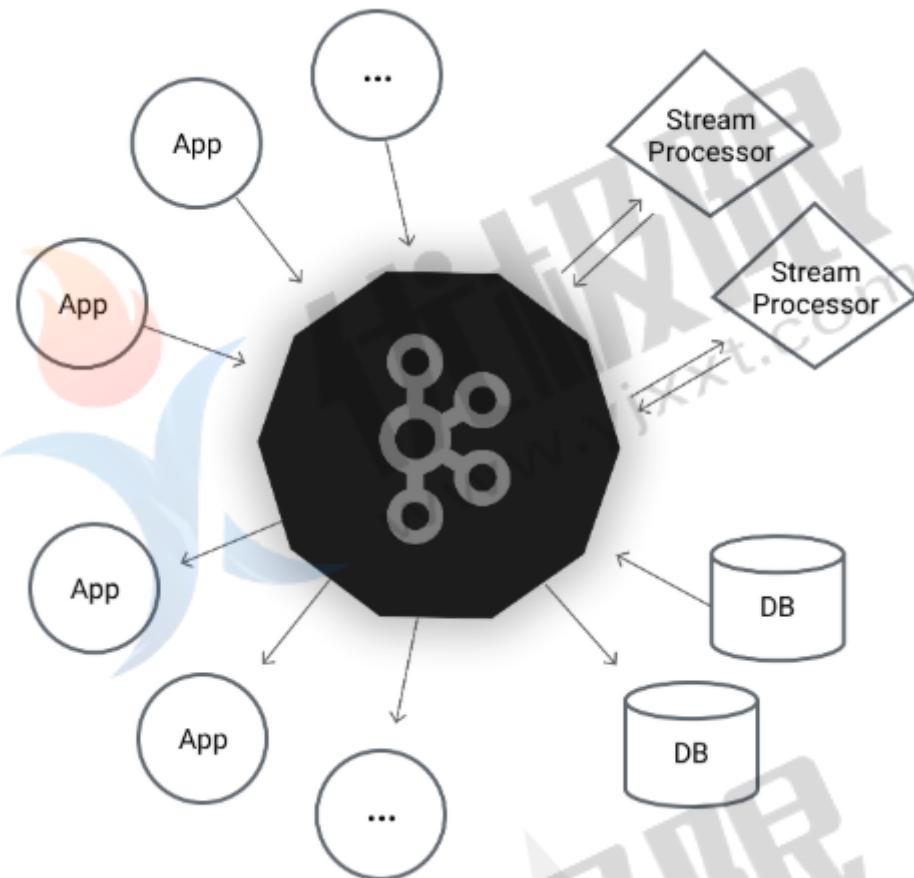
- 在发布-订阅消息系统中，消息被持久化到一个topic中。
- 消费者可以订阅一个或多个topic，消费者可以消费该topic中所有的数据，同一条数据可以被多个消费者消费，数据被消费后不会立马删除。
- 在发布-订阅消息系统中，消息的生产者称为发布者，消费者称为订阅者。
- Kafka 采取拉取模型(Poll)，由自己控制消费速度，以及消费的进度，消费者可以按照任意的偏移量进行消费。

## 消息队列-发布订阅



## 3. Kafka简介

- 官网：<http://kafka.apache.org/>
- Kafka是由Apache软件基金会开发的一个开源流处理平台，由Scala和Java编写。Kafka是一种高吞吐量的分布式发布订阅消息系统，它可以处理消费者在网站中的所有动作流数据。



### 3.1. 设计目标

- 以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间的访问性能。
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条消息的传输。
- 支持Kafka Server间的消息分区，及分布式消费，同时保证每个partition内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- 支持在线水平扩展

### 3.2. Kafka的优点

- 解耦：

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这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。

- 冗余

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的“插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

- 扩展性

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

- 灵活性&峰值处理能力

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

- 可恢复性

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以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。

- 顺序保证

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka保证一个Partition内的消息的有序性。

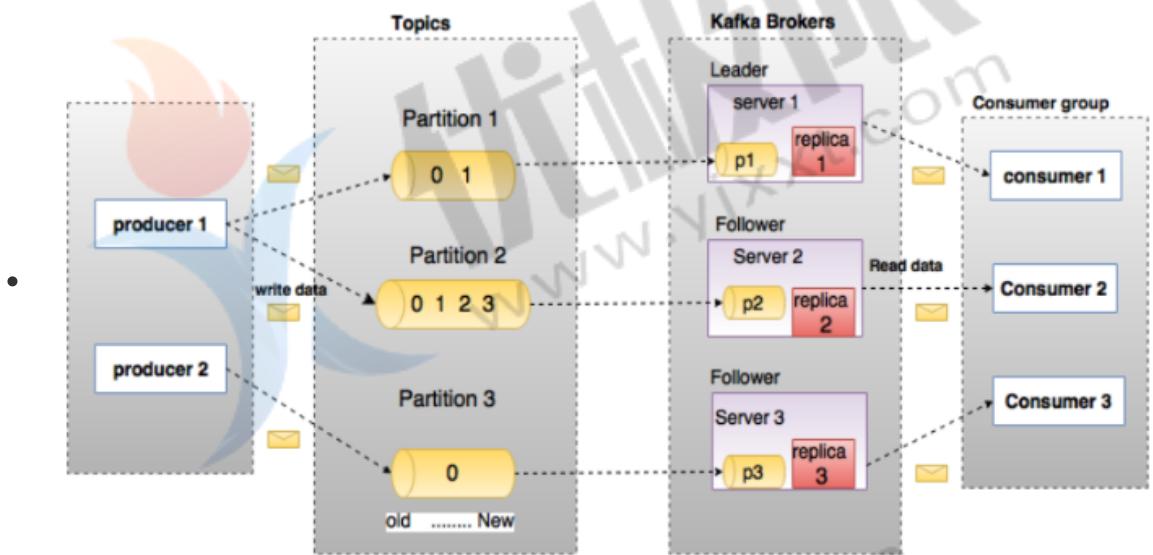
- 缓冲

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行——写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

- 异步通信

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

## 4. Kafka系统架构



### 4.1. Broker

- Kafka 集群包含一个或多个服务器，服务器节点称为broker。

### 4.2. Topic

- 每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic。
- 类似于数据库的表名或者ES的Index
- 物理上不同Topic的消息分开存储
- 逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户只需指定消息的Topic即可生产或消费数据而不必关心数据存于何处)
- 创建流程

- 1. controller在zooKeeper的/brokers/topics节点上注册watcher, 当topic被创建, 则controller会通过watch得到该topic的partition(replica)分配。
- 2. controller从/brokers/ids读取当前所有可用的broker列表, 对于set\_p中的每一个partition:
  - 2.1从分配给该partition的所有replica(称为AR)中任选一个可用的broker作为新的leader, 并将AR设置为新的ISR
  - 2.2将新的leader和ISR写入/brokers/topics/[topic]/partitions/[partition]/state
- 3. controller通过RPC向相关的broker发送LeaderAndISRRequest。

- 删除流程

- 1. controller在zooKeeper的/brokers/topics节点上注册watcher, 当topic被删除, 则controller会通过watch得到该topic的partition(replica)分配。
- 2. 若delete.topic.enable=false, 结束; 否则controller注册在/admin/delete\_topics上的watch被fire, controller通过回调向对应的broker发送StopReplicaRequest。

### 4.3. Partition



- topic中的数据分割为一个或多个partition。
- 每个topic至少有一个partition, 当生产者产生数据的时候, 根据分配策略选择分区, 然后将消息追加到指定的分区的末尾(队列)

- ## Partition数据路由规则
  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。

- 每条消息都会有一个自增的编号
  - 标识顺序
  - 用于标识消息的偏移量
- 每个partition中的数据使用多个segment文件存储。
- partition中的数据是有序的, 不同partition间的数据丢失了数据的顺序。
- 如果topic有多个partition, 消费数据时就不能保证数据的顺序。严格保证消息的消费顺序的场景下, 需要将partition数目设为1。

## 4.4. Leader

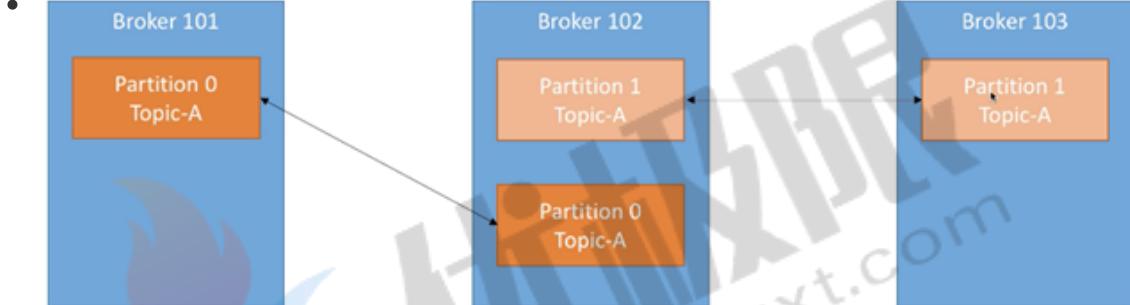
- 每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的partition。
- 1. producer 先从 zookeeper 的 "/brokers/.../state" 节点找到该 partition 的 leader
  2. producer 将消息发送给该 leader
  3. leader 将消息写入本地 log
  4. followers 从 leader pull 消息，写入本地 log 后 leader 发送 ACK
  5. leader 收到所有 ISR 中的 replica 的 ACK 后，增加 HW (high watermark, 最后 commit 的 offset) 并向 producer 发送 ACK

## 4.5. Follower

- Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与Leader保持数据同步。
- 如果Leader失效，则从Follower中选举出一个新的Leader。
- 当Follower挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas” (ISR) 列表中删除，重新创建一个Follower。

## 4.6. replication

- 数据会存放到topic的partation中，但是有可能分区会损坏
- 我们需要对分区的数据进行备份（备份多少取决于你对数据的重视程度）
- 我们将分区的分为Leader(1)和Follower(N)
  - Leader负责写入和读取数据
  - Follower只负责备份
  - 保证了数据的一致性
- 备份数设置为N，表示主+备=N(参考HDFS)
  - ## Kafka 分配 Replica 的算法如下
    1. 将所有 broker (假设共 n 个 broker) 和待分配的 partition 排序
    2. 将第 i 个 partition 分配到第 ( $i \bmod n$ ) 个 broker 上
    3. 将第 i 个 partition 的第 j 个 replica 分配到第 ( $(i + j) \bmod n$ ) 个 broker 上

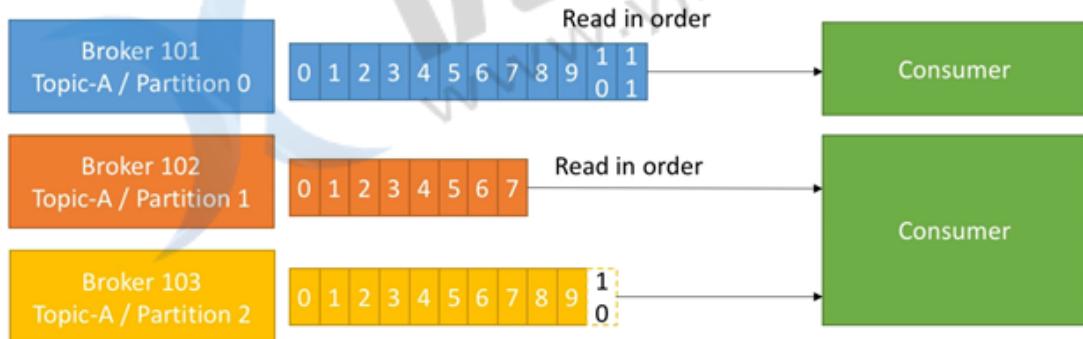


## 4.7. producer

- 生产者即数据的发布者，该角色将消息发布到Kafka的topic中。
- broker接收到生产者发送的消息后，broker将该消息追加到当前用于追加数据的segment文件中。
- 生产者发送的消息，存储到一个partition中，生产者也可以指定数据存储的partition。

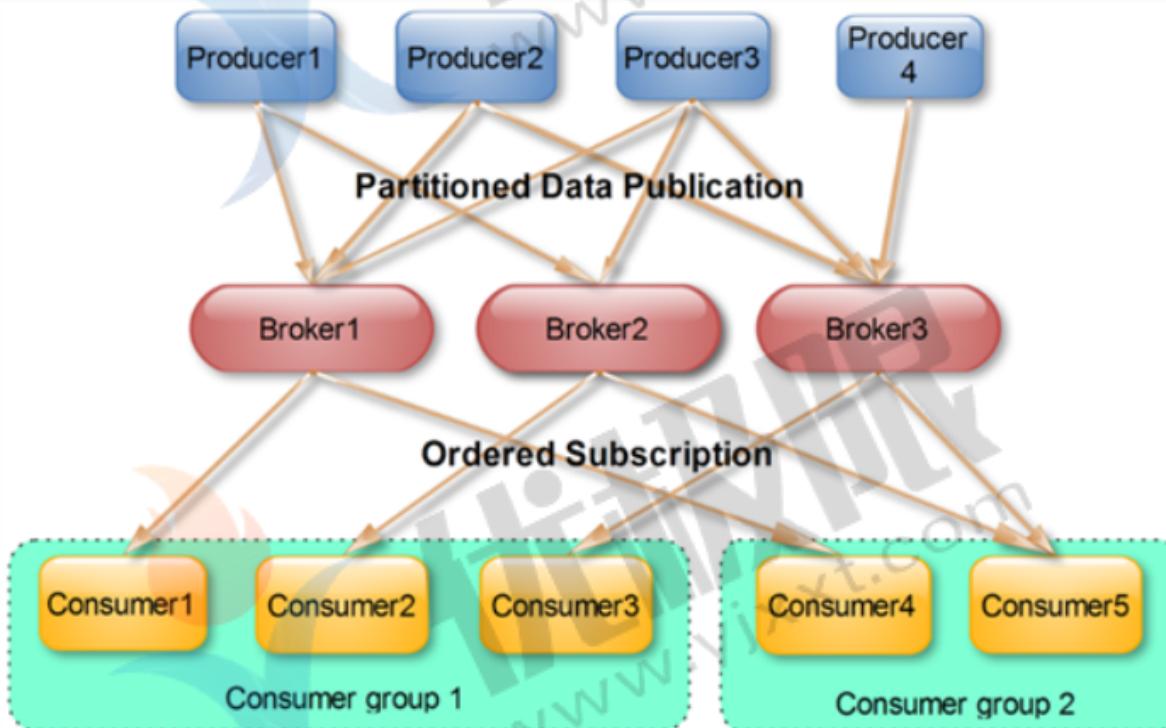
## 4.8. consumer

- 消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。
- kafka 提供了两套 consumer API:
  - 1. The high-level Consumer API
  - 2. The SimpleConsumer API
- high-level consumer API 提供了一个从 kafka 消费数据的高层抽象，而 SimpleConsumer API 则需要开发人员更多地关注细节。



## 4.9. Consumer Group

- 每个Consumer属于一个特定的Consumer Group (可为每个Consumer指定group name, 若不指定group name则属于默认的group)。
- 将多个消费者集中到一起去处理某一个Topic的数据，可以更快的提高数据的消费能力
- 整个消费者组共享一组偏移量(防止数据被重复读取)，因为一个Topic有多个分区



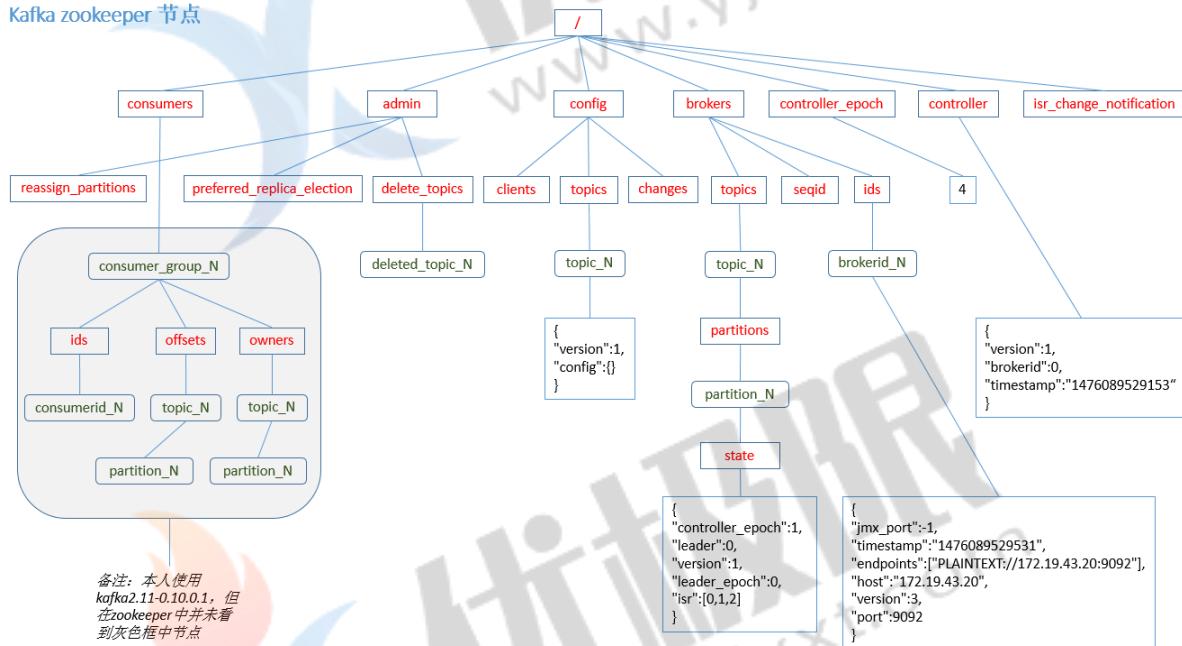
## 4.10. offset偏移量

- 可以唯一的标识一条消息
- 偏移量决定读取数据的位置，不会有线程安全的问题，消费者通过偏移量来决定下次读取的消息
- 消息被消费之后，并不被马上删除，这样多个业务就可以重复使用kafka的消息
- 我们某一个业务也可以通过修改偏移量达到重新读取消息的目的，偏移量由用户控制
- 消息最终还是会被删除的，默认生命周期为1周（7\*24小时）

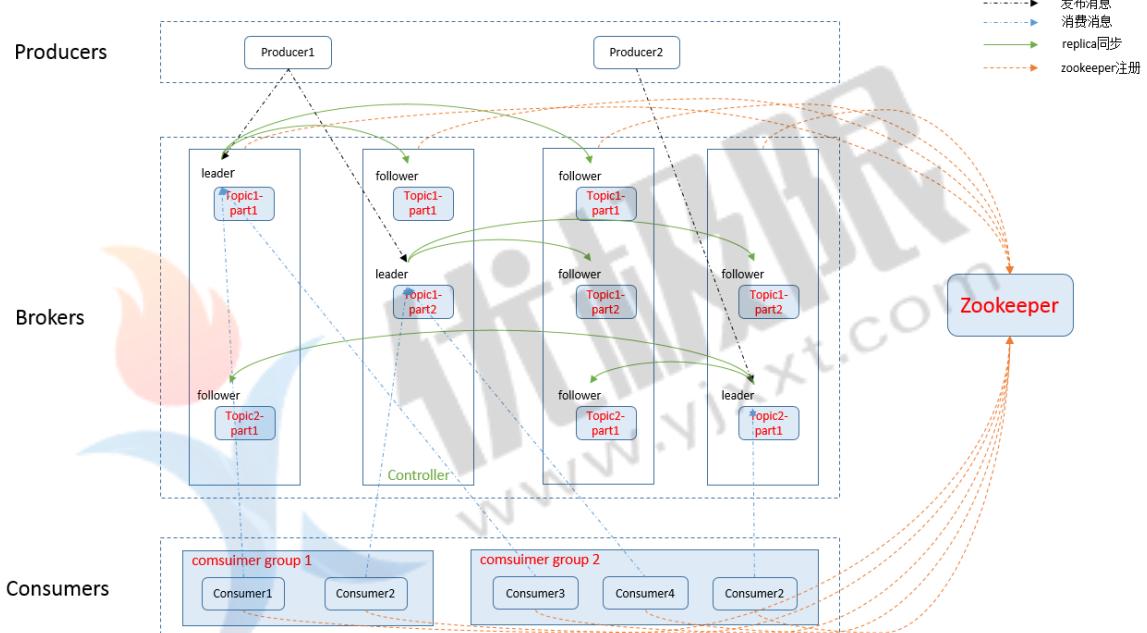
## 4.11. Zookeeper

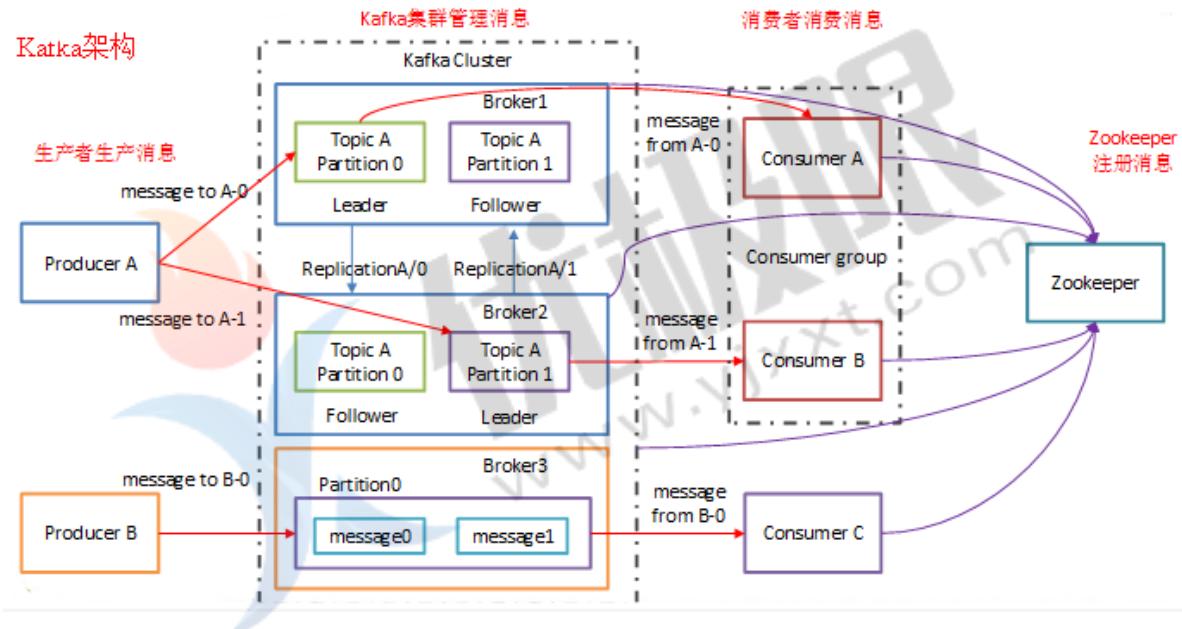
- kafka 通过 zookeeper 来存储集群的 meta 信息。

Kafka zookeeper 节点



Kafka 拓扑结构





## 5. Kafka环境搭建

- 基于Zookeeper搭建并开启
  - 验证ZK的可用性
  - 【123】 zkServer.sh start
- 配置Kafka
  - 基本操作

- 上传解压拷贝

```
[root@bd1601 kafka_2.11-0.8.2.1]# pwd
/opt/bdp/kafka_2.11-0.8.2.1
[root@bd1601 kafka_2.11-0.8.2.1]# ll
total 24
drwxr-xr-x 3 root root 4096 Feb 27 2015 bin
drwxr-xr-x 2 root root 196 Feb 27 2015 config
drwxr-xr-x 2 root root 4096 Feb 27 2015 libs
-rw-r--r-- 1 root root 11358 Feb 27 2015 LICENSE
-rw-r--r-- 1 root root 162 Feb 27 2015 NOTICE
```

- 修改配置文件

- ## vim server.properties
 

```
20 broker.id=0
25 port=9092
58 log.dirs=/var/bdp/kafka-logs
118 zookeeper.connect=node01:2181,node02:2181,node03:2181
```

- 修改环境变量

- ## vim /etc/profile
 

```
export KAFKA_HOME=/opt/lzj/kafka_2.11-0.8.2.1
export PATH=$KAFKA_HOME/bin:$PATH
## 配置文件生效
# source /etc/profile
```

- 将文件目录拷贝到其他机器

- ```
[1]scp -r kafka_2.11-0.8.2.1 root@node02: `pwd`  
[1]scp -r kafka_2.11-0.8.2.1 root@node03: `pwd`  
[1]scp /etc/profile root@node02:/etc/profile  
[1]scp /etc/profile root@node03:/etc/profile  
[123] source /etc/profile
```

- 修改其他机器上的配置

- ```
## vim server.properties  
[2]broker.id=1  
[3]broker.id=2
```

- 启动集群

- `kafka-server-start.sh /opt/lzj/kafka_2.11-0.8.2.1/config/server.properties`

- 常见命令

- ```
//创建主题  
kafka-topics.sh --zookeeper node01:2181,node02:2181,node03:2181 --  
create --replication-factor 2 --partitions 3 --topic userlog  
kafka-topics.sh --zookeeper node01:2181 --create --replication-  
factor 2 --partitions 6 --topic studentlog  
  
kafka-topics.sh --zookeeper node01:2181 --delete --replication-  
factor 2 --partitions 6 --topic baidu  
  
//查看所有主题  
kafka-topics.sh --zookeeper node01:2181,node02:2181,node03:2181 --  
list  
  
//查看主题  
kafka-topics.sh --zookeeper node01:2181,node02:2181,node03:2181 --  
describe --topic userlog  
  
//创建生产者  
kafka-console-producer.sh --broker-list  
node01:9092,node02:9092,node03:9092 --topic userlog  
  
//创建消费者  
kafka-console-consumer.sh --zookeeper  
node01:2181,node02:2181,node03:2181 --from-beginning --topic userlog
```

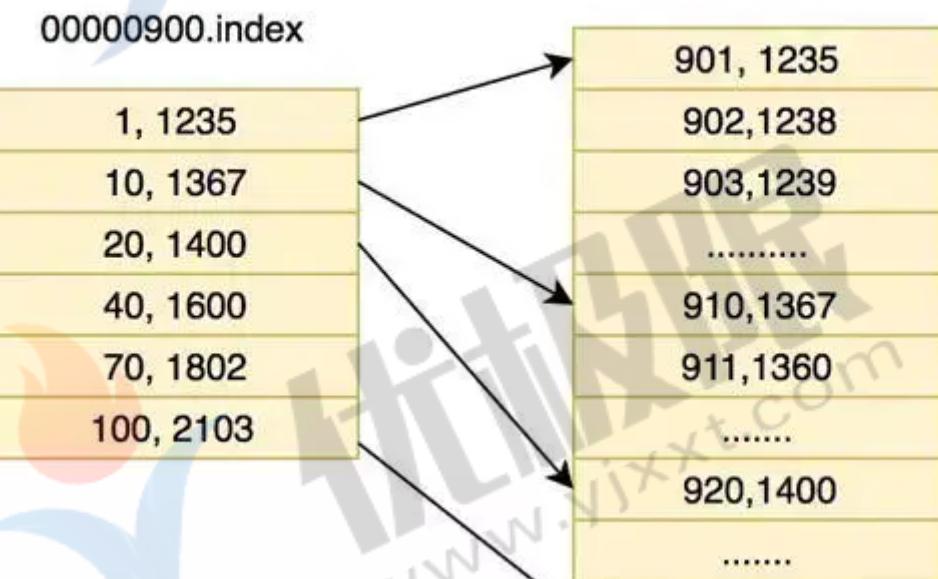
```
[root@bd1601 ~]# kafka-topics.sh --zookeeper bd1601:2181,bd1602:2181,bd1603:2181 --describe --topic userlog  
Topic:userlog PartitionCount:3 ReplicationFactor:2 Configs:  
  Topic: userlog Partition: 0 Leader: 2 Replicas: 2,0 Isr: 2,0  
  Topic: userlog Partition: 1 Leader: 0 Replicas: 0,1 Isr: 0,1  
  Topic: userlog Partition: 2 Leader: 1 Replicas: 1,2 Isr: 1,2  
[root@bd1601 ~]# kafka-topics.sh --zookeeper bd1601:2181,bd1602:2181,bd1603:2181 --describe --topic teacherlog  
Topic:teacherlog PartitionCount:2 ReplicationFactor:1 Configs:  
  Topic: teacherlog Partition: 0 Leader: 0 Replicas: 0 Isr: 0  
  Topic: teacherlog Partition: 1 Leader: 1 Replicas: 1 Isr: 1
```

## 6. Kafka数据检索机制



- topic在物理层面以partition为分组，一个topic可以分成若干个partition
- partition还可以细分为Segment，一个partition物理上由多个Segment组成
  - segment 的参数有两个：
    - log.segment.bytes: 单个segment可容纳的最大数据量，默认为1GB
    - log.segment.ms: Kafka在commit一个未写满的segment前，所等待的时间（默认为7天）
- LogSegment 文件由两部分组成，分别为“.index”文件和“.log”文件，分别表示为 Segment 索引文件和数据文件。
  - partition全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值
  - 数值大小为64位，20位数字字符长度，没有数字用0填充
  - 第一个segment
    - 00000000000000000000.index
    - 00000000000000000000.log
  - 第二个segment，文件命名以第一个segment的最后一条消息的offset组成
    - 000000000000170410.index
    - 000000000000170410.log
  - 第三个segment，文件命名以上一个segment的最后一条消息的offset组成
    - 000000000000239430.index
    - 000000000000239430.log
- 消息都具有固定的物理结构，包括：offset(8 Bytes)、消息体的大小(4 Bytes)、crc32(4 Bytes)、magic(1 Byte)、attributes(1 Byte)、key length(4 Bytes)、key(K Bytes)、payload(N Bytes)等等字段，可以确定一条消息的大小，即读取到哪里截止。

- 



## 7. 数据的安全性

## 7.1. producer delivery guarantee

- - 0. At least one 消息绝不会丢，但可能会重复传输
  - 1. At most once 消息可能会丢，但绝不会重复传输
  - 2. Exactly once 每条消息肯定会被传输一次且仅传输一次
- Producers可以选择是否为数据的写入接收ack，有以下几种ack的选项: request.required.acks
  - acks=0:
    - Producer 在 ISR 中的 Leader 已成功收到的数据并得到确认后发送下一条 Message。
  - acks=1:
    - 这意味着 Producer 无需等待来自 Broker 的确认而继续发送下一批消息。
  - acks=all:
    - Producer 需要等待 ISR 中的所有 Follower 都确认接收到数据后才算一次发送完成，可靠性最高。

## 7.2. ISR机制

- 关键词
  - AR : Assigned Replicas 用来标识副本的全集
  - OSR : out -sync Replicas 离开同步队列的副本
  - ISR : in -sync Replicas 加入同步队列的副本
  - ISR = Leader + 没有落后太多的副本;AR = OSR+ ISR。
- 我们备份数据就是防止数据丢失，当主节点挂掉时，可以启用备份节点
  - producer--push-->leader
  - leader--pull-->follower
  - Follower每隔一定时间去Leader拉取数据，来保证数据的同步
- ISR(in-syncReplica)
  - 当主节点挂点，并不是去Follower选择主，而是从ISR中选择主
  - 判断标准
    - 超过10秒钟没有同步数据
      - replica.lag.time.max.ms=10000
      - 主副节点差4000条数据
        - replica.lag.max.messages=4000
  - 脏节点选举
    - kafka采用一种降级措施来处理：
    - 选举第一个恢复的node作为leader提供服务，以它的数据为基准，这个措施被称为脏leader选举

## 7.3. Broker数据存储机制

- 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

## 7.4. consumer delivery guarantee

- 如果将 consumer 设置为 autocommit, consumer 一旦读到数据立即自动 commit。如果只讨论这一读取消息的过程, 那 Kafka 确保了 Exactly once。
  - 如果 consumer 在 commit 后还没来得及处理消息就 crash 了, 下次重新开始工作后就无法读到刚刚已提交而未处理的消息
  - 这就对应于 At most once
- 读完消息先 commit 再处理消息。
  - 如果 consumer 在 commit 后还没来得及处理消息就 crash 了, 下次重新开始工作后就无法读到刚刚已提交而未处理的消息, 实际上该消息已经被处理过了。
  - 这就对应于 At least once。
- 读完消息先处理再 commit。
  - 如果在处理完消息之后 commit 之前 consumer crash 了, 下次重新开始工作时还会处理刚刚未 commit 的消息, 实际上该消息已经被处理过了。
  - 这就对应于 At least once。
- 如果一定要做到 Exactly once, 就需要协调 offset 和实际操作的输出。
  - 经典的做法是引入两阶段提交。
- Kafka 默认保证 At least once, 并且允许通过设置 producer 异步提交来实现 At most once

## 7.5. 数据的消费

- partition\_num=2, 启动一个consumer进程订阅这个topic, 对应的, stream\_num设为2, 也就是说启两个线程并行处理message。
- 如果auto.commit.enable=true,
  - 当consumer fetch了一些数据但还没有完全处理掉的时候,
  - 刚好到commit interval出发了提交offset操作, 接着consumer crash掉了。
  - 这时已经fetch的数据还没有处理完成但已经被commit掉, 因此没有机会再次被处理, 数据丢失。
- 如果auto.commit.enable=false,
  - 假设consumer的两个fetcher各自拿了一条数据, 并且由两个线程同时处理,
  - 这时线程t1处理完partition1的数据, 手动提交offset, 这里需要着重说明的是, 当手动执行commit的时候,
  - 实际上是对这个consumer进程所占有的所有partition进行commit, kafka暂时还没有提供更细粒度的commit方式,
  - 也就是说, 即使t2没有处理完partition2的数据, offset也被t1提交掉了。如果这时consumer crash掉, t2正在处理的这条数据就丢失了。
- 方法1: (将多线程问题转成单线程)
  - 手动commit offset, 并针对partition\_num启同样数目的consumer进程, 这样就能保证一个consumer进程占有一个partition, commit offset的时候不会影响别的partition的offset。但这个方法比较局限, 因为partition和consumer进程的数目必须严格对应
- 方法2: (参考HDFS数据写入流程)
  - 手动commit offset, 另外在consumer端再将所有fetch到的数据缓存到queue里, 当把queue里所有的数据处理完之后, 再批量提交offset, 这样就能保证只有处理完的数据才被commit。

## 8. JavaAPI

## 8.1. 生产者

- 创建一线程重复的向kafka输入数据

- 创建生产者线程类

```
public class Hello01Producer extends Thread {  
    //创建Kafka的生产者  
    private Producer<String, String> producer;  
  
    /**  
     * 创建构造器  
     */  
  
    public Hello01Producer(String pname) {  
        //设置线程的名字  
        super.setName(pname);  
        //创建配置文件列表  
        Properties properties = new Properties();  
        // kafka地址, 多个地址用逗号分割  
        properties.put("metadata.broker.list",  
"192.168.58.161:9092,192.168.58.162:9092,192.168.58.163:9092");  
        //设置写出数据的格式  
        properties.put("serializer.class", StringEncoder.class.getName());  
        //写出的应答方式  
        properties.put("acks", 1);  
        //批量写出  
        properties.put("batch.size", 16384);  
        //创建生产者对象  
        producer = new Producer<String, String>(new  
kafka.producer.ProducerConfig(properties));  
    }  
  
    @Override  
    public void run() {  
        //初始化一个计数器  
        int count = 0;  
  
        System.out.println("Hello01Producer.run--开始发送数据");  
        //迭代发送消息  
        while (count < 100000) {  
            String key = String.valueOf(++count);  
            String value = Thread.currentThread().getName() + "--" + count;  
            //封装消息对象  
            KeyedMessage<String, String> message = new KeyedMessage<>("userlog",  
key, value);  
            //发送消息到服务器  
            producer.send(message);  
            //打印消息  
            System.out.println("Producer.run--" + key + "--" + value);  
            //每个1秒发送1条  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
    }

    public static void main(String[] args) {
        Hello01Producer producer = new Hello01Producer("上海尚学堂");
        producer.start();
    }
}
```

## 8.2. 消费者

- 创建一线程重复的向kafka消费数据

```
//创建消费者对象
private ConsumerConnector consumer;

/**
 * 创建构造器
 */
public Hello01Consumer(String cname) {
    super.setName(cname);
    //读取配置文件
    Properties properties = new Properties();
    //ZK地址
    properties.put("zookeeper.connect",
    "192.168.58.161:2181,192.168.58.162:2181,192.168.58.163:2181");
    //消费者所在组的名称
    properties.put("group.id", "shsxt-bigdata");
    //ZK超时时间
    properties.put("zookeeper.session.timeout.ms", "400");
    //当消费者第一次消费时，从最低的偏移量开始消费
    properties.put("auto.offset.reset", "smallest");
    //自动提交偏移量
    properties.put("auto.commit.enable", "true");
    //消费者自动提交偏移量的时间间隔
    properties.put("auto.commit.interval.ms", "1000");
    //创建消费者对象
    consumer = Consumer.createJavaConsumerConnector(new
    ConsumerConfig(properties));
}

@Override
public void run() {
    // 描述读取哪个topic，需要几个线程读
    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    topicCountMap.put("userlog", 1);
    //消费者给句配置信息开始读取消息流
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
    consumer.createMessageStreams(topicCountMap);
    // 每个线程对应于一个KafkaStream
    List<KafkaStream<byte[], byte[]>> list = consumerMap.get("userlog");
    // 获取kafkaStream流
    KafkaStream stream0 = list.get(0);
    ConsumerIterator<byte[], byte[]> it = stream0.iterator();
    //开始迭代并获取数据
    while (it.hasNext()) {
```

```

        // 获取一条消息
        MessageAndMetadata<byte[], byte[]> value = it.next();
        int partition = value.partition();
        long offset = value.offset();
        String data = new String(value.message());
        System.out.println("开始" + data + " partition:" + partition + " offset:" + offset);
    }
}

public static void main(String[] args) {
    Hello01Consumer consumer01 = new Hello01Consumer("李毅");
    consumer01.start();
}
}

```

## 8.3. 重复消费和数据的丢失

- 有可能一个消费者取出了一条数据 (offset=88)，但是还没有处理完成，但是消费者被关闭了
  - 如果下次还能从88重新处理就属于完美情况
  - 如果下次数据从86开始，就属于数据的重复消费
  - 如果下次数据从89开始，就是与数据的丢失
  - //消费者自动提交偏移量的时间间隔props.put("auto.commit.interval.ms", "1010");
 提交间隔《单条执行时间》 (重复)
 提交间隔《单条执行时间》 (丢失)

# 9. Kafka优化

## 9.1. Partition 数目

- 一般来说，每个partition能处理的吞吐为几MB/s（仍需要基于根据本地环境测试后获取准确指标），增加更多的partitions意味着：
  - 更高的并行度与吞吐
  - 可以扩展更多的（同一个consumer group中的）consumers
  - 若是集群中有较多的brokers，则可更大程度上利用闲置的brokers
  - 但是会造成Zookeeper的更多选举
  - 也会在Kafka中打开更多的文件
- 调整准则
  - 一般来说，若是集群较小（小于6个brokers），则配置 $2 \times$  broker数的partition数。在这里主要考虑的是之后的扩展。若是集群扩展了一倍（例如12个），则不用担心会有partition不足的现象发生
  - 一般来说，若是集群较大（大于12个），则配置 $1 \times$  broker数的partition数。因为这里不需要再考虑集群的扩展情况，与broker数相同的partition数已经足够应付常规场景。若有必要，则再手动调整
  - 考虑最高峰吞吐需要的并行consumer数，调整partition的数目。若是应用场景需要有20个（同一个consumer group中的）consumer并行消费，则据此设置为20个partition
  - 考虑producer所需的吞吐，调整partition数目（如果producer的吞吐非常高，或是在接下来两年内都比较高，则增加partition的数目）

## 9.2. Replication factor

- 此参数决定的是records复制的数目，建议至少设置为2，一般是3，最高设置为4。
- 更高的replication factor (假设数目为N) 意味着：
  - 系统更稳定 (允许N-1个broker宕机)
  - 更多的副本 (如果acks=all，则会造成较高的延时)
  - 系统磁盘的使用率会更高 (一般若是RF为3，则相对于RF为2时，会占据更多50% 的磁盘空间)
- 调整准则：
  - 以3为起始 (当然至少需要有3个brokers，同时也不建议一个Kafka 集群中节点数少于3个节点)
  - 如果replication 性能成为了瓶颈或是一个issue，则建议使用一个性能更好的broker，而不是降低RF的数目
  - 永远不要在生产环境中设置RF为1

## 9.3. 批量写入

- 为了大幅度提高producer写入吞吐量，需要定期批量写文件

- 每当producer写入10000条消息时，刷数据到磁盘  
`log.flush.interval.messages=10000`

每间隔1秒钟时间，刷数据到磁盘  
`log.flush.interval.ms=1000`

## 10. Flume+Kafka集成

- 搭建Flume并编写配置文件

- `vim /opt/lzj/flume-1.6.0/options/f2k.conf`
  - `#flume-ng agent -n a1 -f /opt/lzj/flume-1.6.0/options/f2k.conf - flume.root.logger=INFO,console`

```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /var/bdp/baidu.ping

# Describe the sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.topic = baidu
a1.sinks.k1.brokerList = node01:9092,node02:9092,node03:9092
a1.sinks.k1.requiredAcks = 1
a1.sinks.k1.batchSize = 10
a1.sinks.k1.channel = c1

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000000
a1.channels.c1.transactionCapacity = 10000
```

```
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

- 启动Zookeeper, Kafka,然后创建Topic(baidu) ,开启消费者
  - 【123】 zkServer.sh start
  - 【123】 kafka-server-start.sh /opt/lzj/kafka\_2.11/config/server.properties
  - 【1】 kafka-topics.sh --zookeeper node01:2181 --create --replication-factor 3 --partitions 3 --topic baidu
  - 【1】 kafka-console-consumer.sh --zookeeper node01:2181 --from-beginning --topic baidu
- 开启Flume
  - 【1】 flume-ng agent -n a1 -f /opt/lzj/apache-flume-1.6.0-bin/options/f2k.conf -Dflume.root.logger=INFO,console
- 开始ping百度的脚本
  - ping [www.baidu.com](http://www.baidu.com) >> /var/bdp/baidu.ping 2>&1 &