

【多易教育】 Kafka 教程



JUST DO IT
多易教育

1 基本概念

1.1 什么是 kafka

Kafka 它最初由 LinkedIn 公司开发，之后成为 Apache 项目的一部分。

Kafka 是一个分布式消息中间件,支持分区的、多副本的、多订阅者的、基于 zookeeper 协调的分布式消息系统。

通俗来说: kafka 就是一个存储系统, 存储的数据形式为“消息”;
它的主要作用类似于蓄水池, 起到一个缓冲作用;

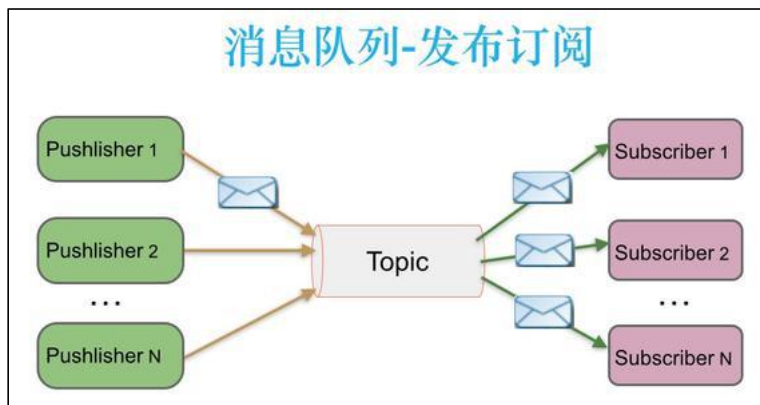
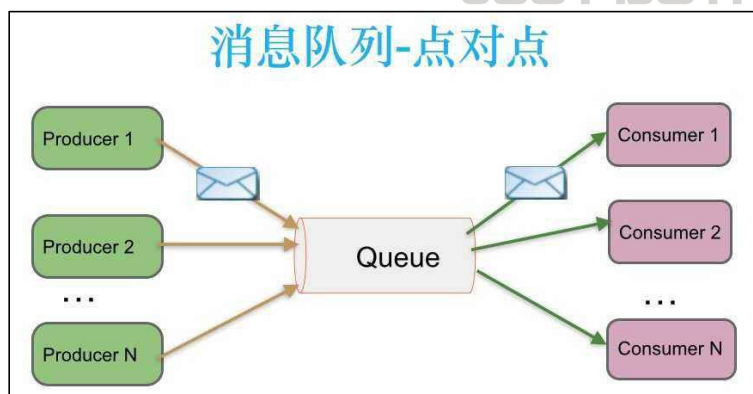
【扩展阅读】怎样理解存储系统

1.1.1 消息队列介绍

常见的消息队列有 activemq , rabbitmq , rocketmq;

消息队列常用于两个系统之间的数据传递;

分布式消息传递基于可靠的消息队列, 在客户端应用和消息系统之间异步传递消息。
有两种主要的消息传递模式: 点对点传递模式、发布-订阅模式。

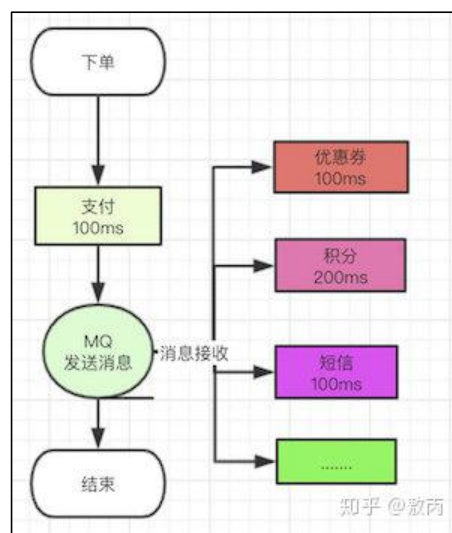
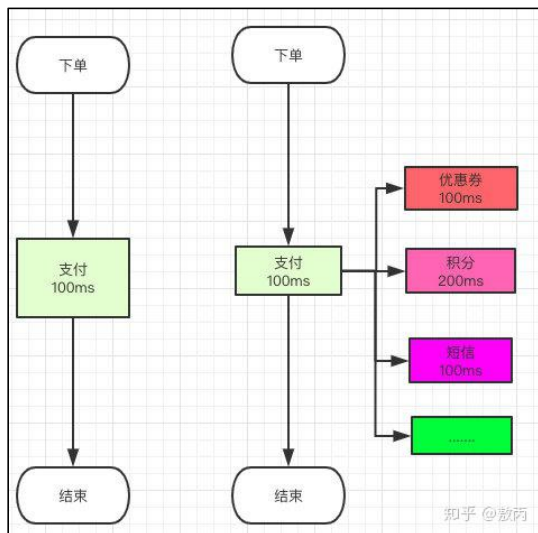
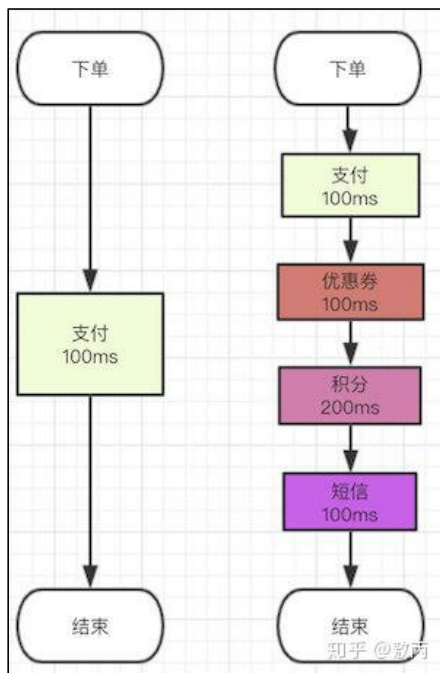


大部分的消息系统选用发布-订阅模式。

kafka 是发布-订阅模式。

1.1.2 为什么使用消息队列

- 异步
- 削峰
- 解耦



详细说明：

(1) 利用消息队列对原系统进行**解耦**（解耦数据的处理过程）：

- ① 提高扩展性：因为消息队列解耦了处理过程，有新增需求时只要另外增加处理过程即可。不需要改变原系统代码；
- ② 提高峰值处理能力：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请求而完全崩溃；
- ③ 提高系统的可恢复性：系统的一部分组件时效时，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回复后被处理

【扩展阅读】什么是解耦？

(2) **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。想从队列中放多少消息就放多少，然后在需要的时候再去处理它们。

【扩展阅读】什么是同步处理，什么是异步处理？

(3) 增加数据冗余和安全性：有些情况下，处理数据的过程会失败，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。

(4) 顺序保证：在大多使用场景下，数据的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 只能保证一个 partition 内的消息有序性**

(5) 缓冲：在任何中重要的系统中，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。

(6) 用于数据流：在一个分布式系统里，要得到一个关于用户操作会用多长时间及其原因的总体印象，是个巨大的挑战。消息系列通过消息处理的频繁，来方便的辅助确定那些表现不佳的处理过程或领域，这些地方的数据流不够优化。

1.2 kafka 的特点

- 高吞吐量、低延迟：kafka 每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个 topic 可以分多个 partition，由多个 consumer group 对 partition 进行 consume 操作。
- 可扩展性：kafka 集群支持热扩展
- 持久性、可靠性：消息被持久化到本地磁盘，并且**支持数据备份**防止数据丢失
- 容错性：允许**集群中节点失败**（若副本数量为 n,则允许 n-1 个节点失败）
- 高并发：支持数千个客户端同时读写

1.3 kafka 的使用场景

主要用于数据处理系统中的缓冲！（尤其是实时流式数据处理）

- 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。
- 消息系统：解耦和生产者和消费者、缓存消息等。
- 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。
- 运营指标：kafka 也经常用来记录运维监控数据。包括收集各种分布式应用的数据，各种操作的集中反馈，比如报警和报告。
- 流式数据处理：比如 spark streaming 和 Flink

1.4 扩展阅读

【扩展阅读】什么是存储系统

- 什么是存储系统？

存数据的系统，比如 mysql、hdfs、hbase、redis、elastic search，本地文件系统

- 什么是存储视图？

所存储的数据的展现形式

— 比如，HDFS 中存储的数据以文件形式存在：/aaa/bbb/a.txt

文件系统，通常是各类“高级”存储系统的底层基础；

Hadoop Overview Datanodes Datanode Volume Failures Snapshot Startup Progress Utilities ▾

Browse Directory

/

Show entries Search:

<input type="checkbox"/>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	<input type="checkbox"/>
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Sep 14 16:21	0	0 B	a_data	
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Sep 14 16:22	0	0 B	a_data_backup	
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Oct 25 15:15	0	0 B	bulkload	
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Sep 03 14:49	0	0 B	dict_data	
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Oct 12 15:36	0	0 B	dicts	
<input type="checkbox"/>	drwxr-xr-x	root	supergroup	0 B	Oct 16 14:52	0	0 B	doit17job	

— 比如，mysql 存储的数据以二维表的形式存在，当然，mysql 的数据最终也是存储在文件系统中；

对象 student @realtimedw (doite...				
开始事务 备注 筛选 排序 导入 导出				
id	name	age	sex	modify_time
1	tom	33	male	2020-10-14 11:31:46
2	jack	18	female	2020-10-14 11:31:53
3	lucy	28	male	2020-10-15 16:31:57
4	brown	30	female	2020-10-15 11:37:16
5	kitty	26	male	2020-10-15 13:37:30

— 比如，kafka 消息系统，它也是一个数据存储系统，对外展现出的视图，不是文件，不是二维表，而是“消息”，当然，这些所谓的“消息”最终也是存储在文件系统上

所谓消息：就是一条数据；kafka 中的“消息”由一个 key 和一个 value 组成。而大量消息可以根据主题、用途等划分到不同的“topic”中；

0	byte[]
1	byte[]
2	byte[]
3	byte[]
4	byte[]
5	byte[]
...	...

扩展问题：

什么是运算系统？

什么是存储+运算系统？

什么是消息中间件？（消息队列）

什么是服务？什么是客户端？什么是协议？

【扩展阅读】同步解耦？

// 版本 1

```
class BImpl{
    public void method(String param){
        // SOME LOGIC
    };
}
```

```
class Aimpl{

    BImpl b = new BImpl();

    public void doSomething() {
        b.method("param");
    }
}
```

// 版本 2

```
interface BInterface{
    public void method(String param);
}
```

```
class BImp1 implements BInterface {
    public void method(String param){
        // SOME LOGIC 1
    };
}
```

```
class BImp2 implements BInterface {
    public void method(String param){
```



```
// SOME LOGIC 2
};
}

class AImpl{
    public static void main(String[] args) {
        // 通过配置文件获取所需要的实现类名
        String className = PropertyUtil.get("calssName");
        BInterface b = ReflectUtil.getInstance(className);
        b.method("param");
    }
}
```

【扩展阅读】同步处理，异步处理，异步解耦？

同步调用

```
/**
 * 下游：数据处理者
 */
class MsgProcessor {
    public void processMsg(String msg){
        // 将收到的消息进行合法性检查
        // 将检查合格的消息按照确定规范进行格式化处理
        // 根据消息中的某指定字段去数据库中查找匹配信息
        // 将匹配到的信息添加到“消息”中
        // .....
    }
}

/**
 * 上游：数据生成者
 */
class GenMessage{
    MsgProcessor processor = new MsgProcessor();
    Random random = new Random();
    public void genMas(){
        while(true){
            // 模拟生成数据
            String msg = "msg "+ random.nextInt(100);
            // 处理数据，此处的处理方案，即为“同步调用”：调用 MsgProcessor 类 processMsg 方法
```



```
// 同步调用最核心特点：调用了目标方法后得等到目标方法返回，自己逻辑才能继续执行
processor.processMsg(msg);

// 处理完成后的其他事情
// .....

    }
}
}
```

异步调用

```
/**
 * 中游，缓冲池
 */
class MsgBuffer{
    ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(100);

    public void put(String msg) throws Exception {
        queue.put(msg);
    }

    public String take() throws Exception {
        return queue.take();
    }
}

/**
 * 上游：数据生成者
 */
class GenMessage2{
    MsgBuffer buffer;
    public GenMessage2(MsgBuffer buffer){
        this.buffer = buffer;
    }

    public void genMsg() throws Exception {
        Random random = new Random();
        while(true){
            // 模拟生成数据
            String msg = "msg "+ random.nextInt(100);

            // 将生成的数据放入缓冲
        }
    }
}
```

```
        buffer.put(msg);

        // 继续处理其他事情
        // .....
    }
}

/**
 * 下游：数据处理者
 */
class MsgProcessor2 {
    MsgBuffer buffer;
    public MsgProcessor2(MsgBuffer buffer){
        this.buffer = buffer;
    }
    public void processMsg() throws Exception {
        while(true) {
            // 从缓冲池中获取数据
            String msg = buffer.take();

            // 将收到的消息进行合法性检查
            // 将检查合格的消息按照确定规范进行格式化处理
            // 根据消息中的某指定字段去数据库中查找匹配信息
            // 将匹配到的信息添加到“消息”中
            // .....
        }
    }
}

public class CallDemo{
    public static void main(String[] args) throws Exception {
        MsgBuffer msgBuffer = new MsgBuffer();

        // 启动生产者，生成数据，只管自己生成，不用等待后续的处理
        new Thread(new Runnable() {
            @Override
            public void run() {
                GenMessage2 genner = new GenMessage2(msgBuffer);
                try {
                    // 生产数据

```

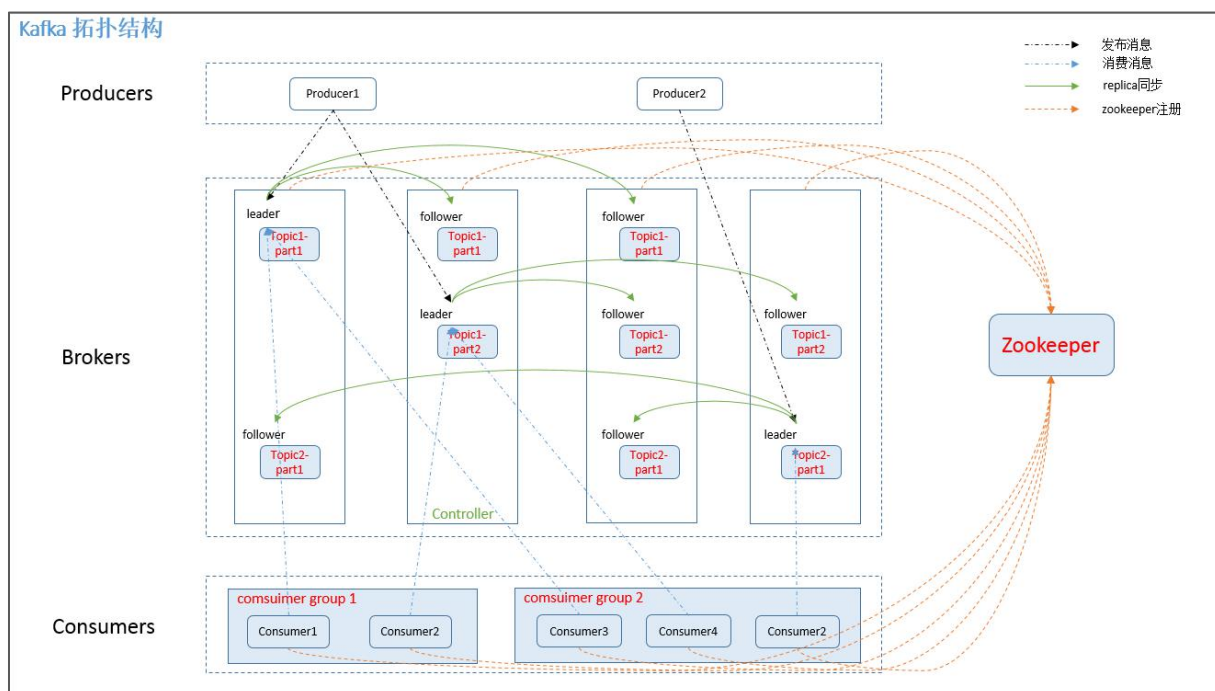
```
        genner.genMsg();
    } catch (Exception e) {}
    }
}).start();

// 启动消费者，处理数据，只管自己处理，不用关心数据的生产
new Thread(new Runnable() {
    @Override
    public void run() {
        MsgProcessor2 processor = new MsgProcessor2(msgBuffer);
        try {
            // 处理数据
            processor.processMsg();
        } catch (Exception e) {}
    }
}).start();

// 接下来，这里可以干任何事情
}
}
```

JUST DO IT

2 kafka 系统的架构（基础重点）



Kafka 架构分为以下几个部分

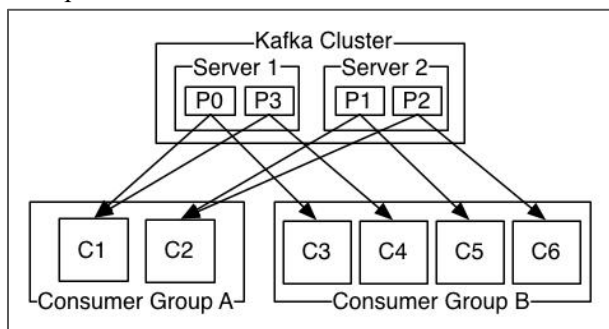
2.1 producer

消息生产者，就是向 kafka broker 发消息的客户端。

2.2 consumer

consumer：消息消费者，从 kafka broker 取消息的客户端。

consumer group：单个或多个 consumer 可以组成一个 consumer group；这是 kafka 用来实现消息的广播(发给所有的 consumer)和单播(发给任意一个 consumer)的手段。一个 topic 可以有多个 Consumer Group。



JUST DO IT
多易教育

2.3 topic

- 数据的逻辑分类；

你可以理解为数据库中“表”的概念；

- partition

topic 中数据的具体管理单元；（你可以理解为 hbase 中表的“region”概念）

一个 topic 可以划分为多个 partition，分布到多个 broker 上管理；

每个 partition 由一个 kafka broker 服务器管理；

partition 中的每条消息都会被分配一个递增的 id (offset)；

每个 partition 是一个有序的队列，kafka 只保证按一个 partition 中的消息的顺序，不保证一个 topic 的整体（多个 partition 间）的顺序。

每个 partition 都可以有多个副本；

- broker

一台 kafka 服务器就是一个 broker。
一个 kafka 集群由多个 broker 组成。
一个 broker 可以容纳多个 topic 的多个 partition。

分区对于 kafka 集群的好处是：实现 topic 数据的负载均衡。分区对于消费者来说，可以提高并发度，提高效率。

- offset

消息在底层存储中的索引位置，kafka 底层的存储文件就是以文件中第一条消息的 offset 来命名的，通过 offset 可以快速定位到消息的具体存储位置；

2.4 Leader

partition replica 中的一个角色，producer 和 consumer 只跟 leader 交互（负责读写）。

2.5 副本 Replica

partition 的副本，保障 partition 的高可用（replica 副本数目不能大于 kafka broker 节点的数目，否则报错）。

每个 partition 的所有副本中，必包括一个 leader 副本，其他的就是 follower 副本

2.6 Follower

partition replica 中的一个角色，从 leader 中拉取复制数据（只负责备份）。

如果 leader 所在节点宕机，follower 中会选举出新的 leader；

2.7 偏移量 Offset

每一条数据都有一个 offset，是数据在该 partition 中的唯一标识（其实就是消息的索引号）。

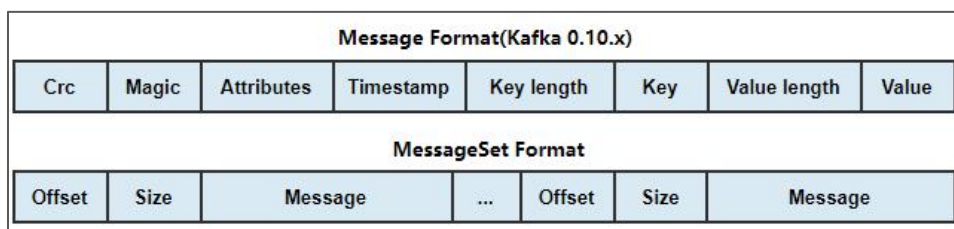
各个 consumer 会保存其消费到的 offset 位置，这样下次可以从该 offset 位置开始继续消费；

consumer 的消费 offset 保存在一个专门的 topic（__consumer_offsets）中；（0.10.x 版本以前是保存在 zk 中）

2.8 消息 Message

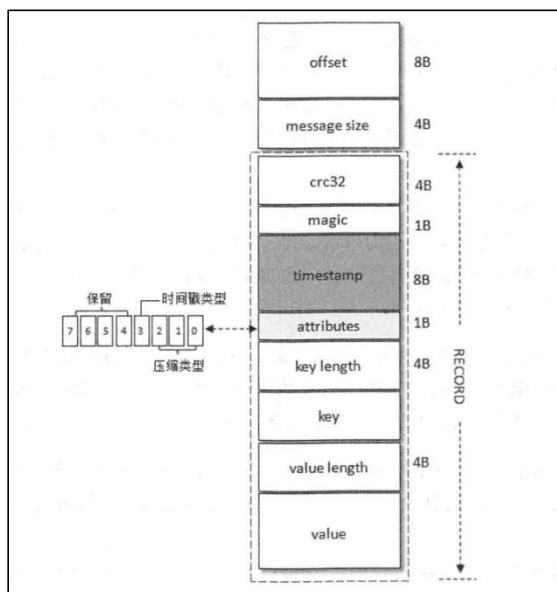
在客户端编程代码中，消息的类叫做 `ProducerRecord`、`ConsumerRecord`；
简单来说，kafka 中的每个 message 由一对 key-value 构成

Kafka 中的 message 格式经历了 3 个版本的变化了：version0 、 version1 、 version2



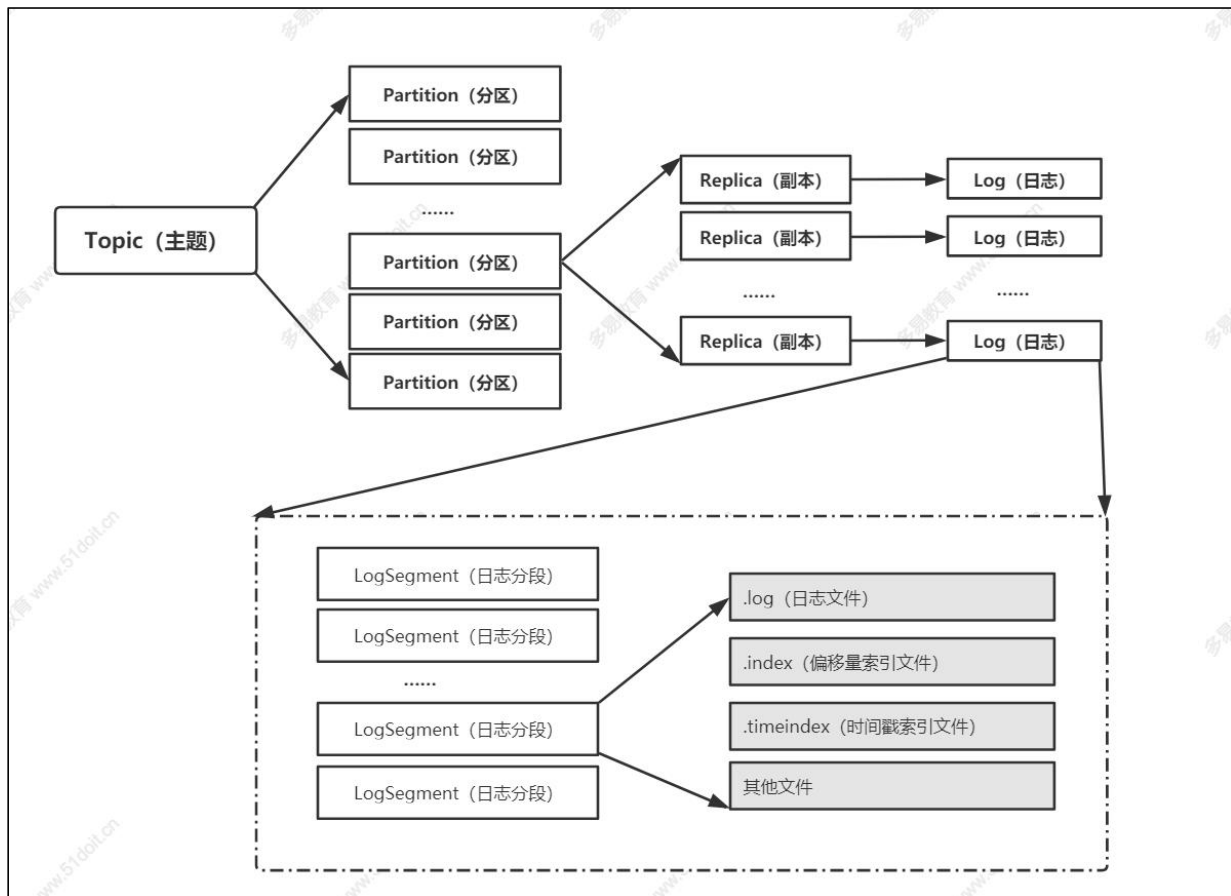
各个字段的含义介绍如下：

- **crc**: 占用 4 个字节，主要用于校验消息的内容；
- **magic**: 这个占用 1 个字节，主要用于标识 Kafka 版本。Kafka 0.10.x magic 默认值为 1
- **attributes**: 占用 1 个字节，这里面存储了消息压缩使用的编码以及 Timestamp 类型。目前 Kafka 支持 gzip、snappy 以及 lz4（0.8.2 引入）三种压缩格式；后四位如果是 0001 则表示 gzip 压缩，如果是 0010 则是 snappy 压缩，如果是 0011 则是 lz4 压缩，如果是 0000 则表示没有使用压缩。第 4 个 bit 位如果为 0，代表使用 create time；如果为 1 代表 append time；其余位（第 5~8 位）保留；
- **key length**: 占用 4 个字节。主要标识 Key 的内容的长度；
- **key**: 占用 N 个字节，存储的是 key 的具体内容；
- **value length**: 占用 4 个字节。主要标识 value 的内容的长度；
- **value**: value 即是消息的真实内容，在 Kafka 中这个也叫做 payload。



3 kafka 的数据存储结构

3.1 kafka 的整体存储结构



3.2 服务器存储结构示例

```
[root@doitedu01 kafka-logs]# ll
total 16
drwxr-xr-x. 2 root root 141 Nov 5 13:51 ATLAS_HOOK-0
-rw-r--r--. 1 root root 0 Nov 5 13:51 cleaner-offset-checkpoint
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-1
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-10
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-13
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-16
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-19
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-22
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-25
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-28
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-31
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-34
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-37
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-4
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-40
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-43
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-46
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-49
drwxr-xr-x. 2 root root 141 Nov 5 13:51 consumer_offsets-7
-rw-r--r--. 1 root root 4 Nov 5 13:52 log-start-offset-checkpoint
-rw-r--r--. 1 root root 54 Nov 5 13:51 meta.properties
-rw-r--r--. 1 root root 461 Nov 5 13:52 recovery-point-offset-checkpoint
-rw-r--r--. 1 root root 461 Nov 5 13:53 replication-offset-checkpoint
drwxr-xr-x. 2 root root 141 Nov 5 13:51 t1-0
drwxr-xr-x. 2 root root 141 Nov 5 13:51 t1-1
drwxr-xr-x. 2 root root 141 Nov 5 13:51 topic2-0
drwxr-xr-x. 2 root root 141 Nov 5 13:51 topic2-1
```

消费者偏移量记录 topic

普通 topic 数据存储目录

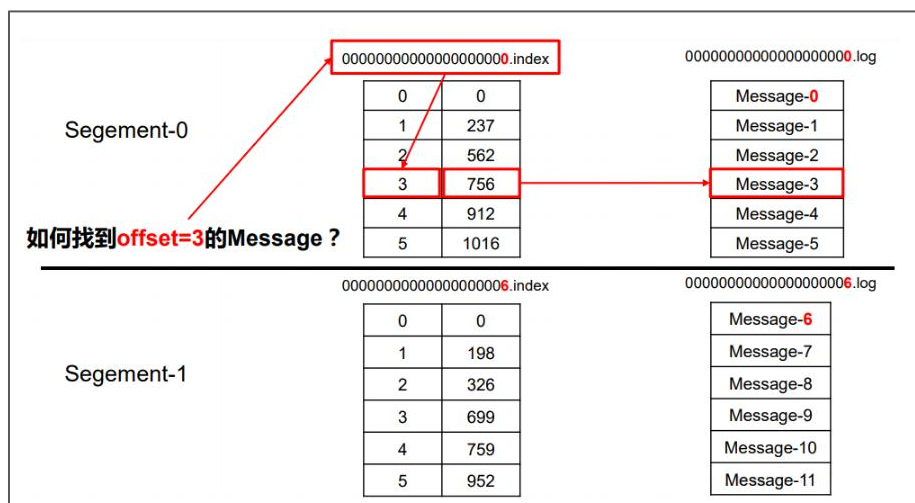
注：“t1”即为一个 topic 的名称；

而“t1-0 / t1-1”则表明这个目录是 t1 这个 topic 的哪个 partition；

```
[root@doitedu01 t1-0]# pwd
/tmp/kafka-logs/t1-0
[root@doitedu01 t1-0]# ll
total 20480
-rw-r--r--. 1 root root 10485760 Nov 5 13:51 00000000000000000000.index
-rw-r--r--. 1 root root 0 Nov 5 13:51 00000000000000000000.log
-rw-r--r--. 1 root root 10485756 Nov 5 13:51 00000000000000000000.timeindex
-rw-r--r--. 1 root root 0 Nov 5 13:51 leader-epoch-checkpoint
[root@doitedu01 t1-0]#
```

由于生产者生产的消息会不断追加到 log 文件末尾，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 对应两个文件：“.index”文件和“.log”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。

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



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据，索引文件中的元数据指向对应数

据文件中 message 的物理偏移地址。

Kafka 中的索引文件以稀疏索引（sparse index）的方式构造消息的索引，它并不保证每个消息在索引文件中都有对应的索引；每当写入一定量（由 broker 端参数 `log.index.interval.bytes` 指定，默认值为 4096，即 4KB）的消息时，偏移量索引文件和时间戳索引文件分别增加一个偏移量索引项和时间戳索引项，增大或减小 `log.index.interval.bytes` 的值，对应地可以增加或缩小索引项的密度；

偏移量索引文件中的偏移量是单调递增的，查询指定偏移量时，使用二分查找法来快速定位偏移量的位置。

4 安装部署

4.1 安装 zookeeper 集群

- 上传安装包

- 移动到指定文件夹

```
mv zookeeper-3.4.6.tar.gz /opt/apps/
```

- 解压

```
tar -zxvf zookeeper-3.4.6.tar.gz
```

- 修改配置文件

（1）进入配置文件目录

```
cd /opt/apps/zookeeper-3.4.6/conf
```

（2）修改配置文件名称

```
mv zoo_sample.cfg zoo.cfg
```

（3）编辑配置文件

```
vi zoo.cfg
```

```
dataDir=/opt/apps/data/zkdata
```

```
server.1=doitedu01:2888:3888
```

```
server.2=doitedu02:2888:3888
```

```
server.3=doitedu03:2888:3888
```

● 创建数据目录

```
mkdir -p /opt/apps/data/zkdata
```

● 在各个节点的数据存储目录中，生成一个 myid 文件，内容为它的 id

```
echo 1 > /opt/apps/data/zkdata/myid
```

```
echo 2 > /opt/apps/data/zkdata/myid
```

```
echo 3 > /opt/apps/data/zkdata/myid
```

● 分发安装包

```
scp -r zookeeper-3.4.6 doitedu02:/opt/apps/ 单独分发
```

```
for i in {2..3};
```

```
do scp -r zookeeper-3.4.6 linux0$i:$PWD;
```

```
done 使用 for 循环分发
```

多易教育

● 配置环境变量

```
vi /etc/profile
```

```
#ZOOKEEPER_HOME
```

```
export ZOOKEEPER_HOME=/opt/apps/zookeeper-3.4.6
```

```
export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

```
source /etc/profile
```

注意：还需要分发环境变量

● 启停集群

```
bin/zkServer.sh start zk 服务启动
```

```
bin/zkServer.sh status zk 查看服务状态
```

```
bin/zkServer.sh stop    zk 停止服务
```

- 脚本启停

- (1) 脚本启动

```
#!/bin/bash

for i in 1 2 3
do
ssh linux0${i} "source /etc/profile;/opt/apps/zookeeper-3.4.6/bin/zkServer.sh start"
done
```

- (2) 脚本停止

```
#!/bin/bash

for i in 1 2 3
do
ssh linux0${i} "source /etc/profile;/opt/apps/zookeeper-3.4.6/bin/zkServer.sh stop"
done
```

4.2 安装 kafka 集群

JUST DO IT
多易教育

- 上传安装包
- 移动到指定文件夹

```
mv kafka_2.11-2.2.2.tgz /opt/apps/
```

- 解压

```
tar -zxvf kafka_2.11-2.2.2.tgz
```

- 修改配置文件

- (1) 进入配置文件目录

```
cd /opt/apps/kafka_2.11-2.2.2/config
```

(2) 编辑配置文件

```
vi server.properties
```

```
#为依次增长的：0、1、2、3、4，集群中唯一 id
```

```
broker.id=0
```

```
#数据存储的目录
```

```
log.dirs=/opt/apps/data/kafkadata
```

```
#指定 zk 集群地址
```

```
zookeeper.connect=doitedu01:2181,doitedu02:2181,doitedu03:2181
```

● 分发安装包

```
for i in {2..3}
```

```
do
```

```
scp -r kafka_2.11-2.2.2 linux0$i:$PWD
```

```
done
```

多易教育

● 配置环境变量

```
vi /etc/profile
```

```
export KAFKA_HOME=/opt/apps/kafka_2.11-2.2.2
```

```
export PATH=$PATH:$KAFKA_HOME/bin
```

```
source /etc/profile
```

注意：还需要分发环境变量

● 分别在 doitedu02 和 doitedu03 上修改配置文件/opt/apps/kafka_2.11-2.2.2/server.properties 中

```
broker.id=1
```

```
broker.id=2 (broker.id 不能重复)
```

- 启停集群（在各个节点上启动）

```
bin/kafka-server-start.sh -daemon /opt/apps/kafka_2.11-2.2.2/config/server.properties
```

停止集群

```
bin/kafka-server-stop.sh stop
```

5 命令行工具

5.1 概述

Kafka 中提供了许多命令行工具（位于\$KAFKA_HOME/bin 目录下）用于管理集群的变更。

kafka-configs.sh	用于配置管理
kafka-console-consumer.sh	用于消费消息
kafka-console-producer.sh	用于生产消息
kafka-consumer-perf-test.sh	用于测试消费性能
kafka-topics.sh	用于管理主题
kafka-dump-log.sh	用于查看日志内容
kafka-server-stop.sh	用于关闭 Kafka 服务
kafka-preferred-replica-election.sh	用于优先副本的选举
kafka-server-start.sh	用于启动 Kafka 服务
kafka-producer-perf-test.sh	用于测试生产性能
kafka-reassign-partitions.sh	用于分区重分配

5.2 topic 操作：kafka-topics

5.2.1 创建 topic

- 基本方式

```
./kafka-topics.sh --zookeeper doitedu01:2181,doitedu02:2181,doitedu03:2181 --create --replication-factor 3  
--partitions 3 --topic test
```

参数解释：

--replication-factor 副本数量

--partitions 分区数量

--topic topic 名称

- 手动指定副本的存储位置

```
bin/kafka-topics.sh --create --topic tpc_1 --zookeeper doitedu01:2181 --replica-assignment 0:1,1:2
```

该方式下，命令会自动判断所要创建的 topic 的分区数及副本数

5.2.2 删除 topic

```
bin/kafka-topics.sh --zookeeper doitedu01:2181,doitedu02:2181,doitedu03:2181 --delete --topic test
```

删除 topic，需要一个参数处于启用状态：`delete.topic.enable = true`

使用 `kafka-topics.sh` 脚本删除主题的行为本质上只是在 ZooKeeper 中的 `/admin/delete_topics` 路径下 建一个与待删除主题同名的节点，以标记该主题为待删除的状态。与创建主题相同的是，真正删除主题的动作也是由 Kafka 的控制器负责完成的。

5.2.3 查看 topic

(1) 列出当前系统中的所有 topic

```
bin/kafka-topics.sh --zookeeper doitedu01:2181,doitedu02:2181,doitedu03:2181 --list
```

(2) 查看 topic 详细信息

```
bin/kafka-topics.sh --zookeeper doitedu01:2181,doitedu02:2181,doitedu03:2181 --describe --topic test
```

Topic:tpc_1	PartitionCount:2	ReplicationFactor:2	Configs:
Topic: tpc_1	Partition: 0	Leader: 0	Replicas: 0,1 Isr: 0,1


```
Topic: tpc_1    Partition: 1    Leader: 1    Replicas: 1,2    Isr: 1,2
```

从上面的结果中,可以看出,topic 的分区数量,以及每个分区的副本数量,以及每个副本所在的 broker 节点,以及每个分区的 leader 副本所在 broker 节点,以及每个分区的 ISR 副本列表;

ISR: in sync replicas 同步副本(当然也包含 leader 自身)

OSR: out of sync replicas 失去同步的副本(数据与 leader 之间的差距超过配置的阈值)

5.2.4 增加分区数

```
bin/kafka-topics.sh --describe --topic tpc_1 --zookeeper doitedu01:2181
```

Kafka 只支持增加分区,不支持减少分区

原因是:减少分区,代价太大(数据的转移,日志段拼接合并)

如果真的需要实现此功能,则完全可以重新创建一个分区数较小的主题,然后将现有主题中的消息按照既定的逻辑复制过去;

5.2.5 动态配置 topic 参数

通过管理命令,可以为已创建的 topic 增加、修改、删除 topic level 参数

- 添加、修改配置参数

```
bin/kafka-configs.sh --zookeeper doitedu01:2181 --entity-type topics --entity-name tpc_1 --alter --add-config compression.type=gzip
```

- 删除配置参数

```
bin/kafka-configs.sh --zookeeper doitedu01:2181 --entity-type topics --entity-name tpc_1 --alter --delete-config compression.type
```

5.3 生产者: kafka-console-producer

```
bin/kafka-console-producer.sh --broker-list doitedu01:9092 --topic test
```

```
>hello word
```

```
>kafka
```

```
>nihao
```

5.4 消费者：kafka-console-consumer

(1) 消费消息

```
bin/kafka-console-consumer.sh --bootstrap-server doitedu01:9092 --from-beginning --topic test
```

(2) #指定要消费的分区，和要消费的起始 offset

```
bin/kafka-console-consumer.sh --bootstrap-server doitedu01:9092,doitedu02:9092,doitedu03:9092 --topic doit14 --offset 2 --partition 0
```

5.5 配置管理 kafka-configs

kafka-configs.sh 脚本是专门用来对配置进行操作的，这里的操作是运行状态修改原有的配置，如此可以达到动态变更的目的；

kafka-configs.sh 脚本包含：变更 alter、查看 describe 这两种指令类型。同使用 kafka-topics.sh 脚本变更配置一样，增、删、改的行为都可以看做变更操作，不过 kafka-configs.sh 脚本不仅可支持操作主题相关的配置，还支持操 broker、用户和客户端这 3 个类型的配置。

kafka-configs.sh 脚本使用 entity-type 参数来指定操作配置的类型，并且使 entity-name 参数来指定操作配置的名称。

比如查看 topic 的配置可以按如下方式执行：

```
bin/kafka-configs.sh zookeeper doitedu01: 2181 --describe --entity-type topics --entity-name tpc_2
```

比如查看 broker 的动态配置可以按如下方式执行：

```
bin/kafka-configs.sh zookeeper doitedu01: 2181 --describe --entity-type brokers --entity-name 0 --zookeeper doitedu01:2181
```

entity-type 和 entity-name 的对应关系

entity-type 的释义	entity-name 的释义
主题类型的配置，取值为 topics	指定主题的名称
broker 类型的配置，取值为 brokers	指定 brokerId 值，即 broker 中 broker.id 参数配置的值
客户端类型的配置，取值为 clients	指定 clientId 值，即 KafkaProducer 或 KafkaConsumer 的 client.id 参数配置的值
用户类型的配置，取值为 users	指定用户名

示例：添加 topic 级别参数

```
bin/kafka-configs.sh --zookeeper localhost:2181 --alter --entity-type topics --entity-name tpc_2 --add-config cleanup.policy=compact ,
max.message.bytes=10000
```

使用 `kafka-configs.sh` 脚本来变更（`alter`）配置时，会在 ZooKeeper 中创建一个命名形式为：`/config/<entity-type>/<entity name>` 的节点，并将变更的配置写入这个节点

6 API 开发：producer 生产者

6.1 生产者 api 示例

一个正常的生产逻辑需要具备以下几个步骤

- （1）配置生产者客户端参数及创建相应的生产者实例
- （2）构建待发送的消息
- （3）发送消息
- （4）关闭生产者实例

首先，引入 maven 依赖

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.0.0</version>
</dependency>
```

JUST DO IT
多易教育

采用默认分区方式将消息散列的发送到各个分区当中

```
import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class MyProducer {

    public static void main(String[] args) throws InterruptedException {

        Properties props = new Properties();

        //设置 kafka 集群的地址

        props.put("bootstrap.servers", "doitedu01:9092,doitedu02:9092,doitedu03:9092");
```

```
//ack 模式, 取值有 0, 1, -1 (all) , all 是最慢但最安全的

props.put("acks", "all");

//失败重试次数 (有可能会造成数据的乱序)

props.put("retries", 3);

//数据发送的批次大小

props.put("batch.size", 10);

//数据发送请求的最大缓存数

props.put("max.request.size", 10);

//消息在缓冲区保留的时间, 超过设置的值就会被提交到服务端

props.put("linger.ms", 10000);

//整个 Producer 用到总内存的大小, 如果缓冲区满了会提交数据到服务端

//buffer.memory 要大于 batch.size, 否则会报申请内存不足的错误

props.put("buffer.memory", 10240);

//序列化器

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++)

    producer.send(new ProducerRecord<String, String>("test", Integer.toString(i),

"dd:" + i));

//Thread.sleep(1000000);

producer.close();

}

}
```

消息对象 `ProducerRecord`, 它并不是单纯意义上的消息, 它包含了多个属性, 原本需要发送的与业务关的消息体只是其中的一个 `value` 属性, 比 “Hello, doitedu!” 只是 `ProducerRecord` 对象的一个属性。 `ProducerRecord` 类的定义如下:

```
public class ProducerRecord<K, V> {
    private final String topic;
    private final Integer partition;
```

```
private final Headers headers;  
private final K key;  
private final V value;  
private final Long timestamp;
```

6.2 必要的参数配置

在创建真正的生产者实例前需要配置相应的参数，比如需要连接的 Kafka 集群地址。在 Kafka 生产者客户端 `KafkaProducer` 中有 3 个参数是必填的。

- `bootstrap.servers`
- `key.serializer`
- `value.serializer`

为了防止参数名字符串书写错误，可以使用如下方式进行设置：

```
props.setProperty(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, ProducerInterceptorPrefix.class.getName());  
props.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "doitedu01:9092,doitedu02:9092");  
props.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());  
props.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
```

JUST DO IT
多易教育

6.3 发送消息

创建生产者实例和构建消息之后 就可以开始发送消息了。发送消息主要有 3 种模式：

- 发后即忘 (fire-and-forget)

发后即忘，它只管往 Kafka 发送，并不关心消息是否正确到达。

在大多数情况下，这种发送方式没有问题；

不过在某些时候（比如发生不可重试异常时）会造成消息的丢失。

这种发送方式的性能最高，可靠性最差。

```
Future<RecordMetadata> send = producer.send(rcd);
```

- 同步发送 (sync)

```
try {  
    producer.send(rcd).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

0.8.x 前，有一个参数 `producer.type=sync|async` 来决定生产者的发送模式；现已失效（新版中，`producer` 在底层只有异步）

- 异步发送（`async`）

回调函数会在 `producer` 收到 `ack` 时调用，为异步调用，该方法有两个参数，分别是 `RecordMetadata` 和 `Exception`，如果 `Exception` 为 `null`，说明消息发送成功，如果 `Exception` 不为 `null`，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

代码示例

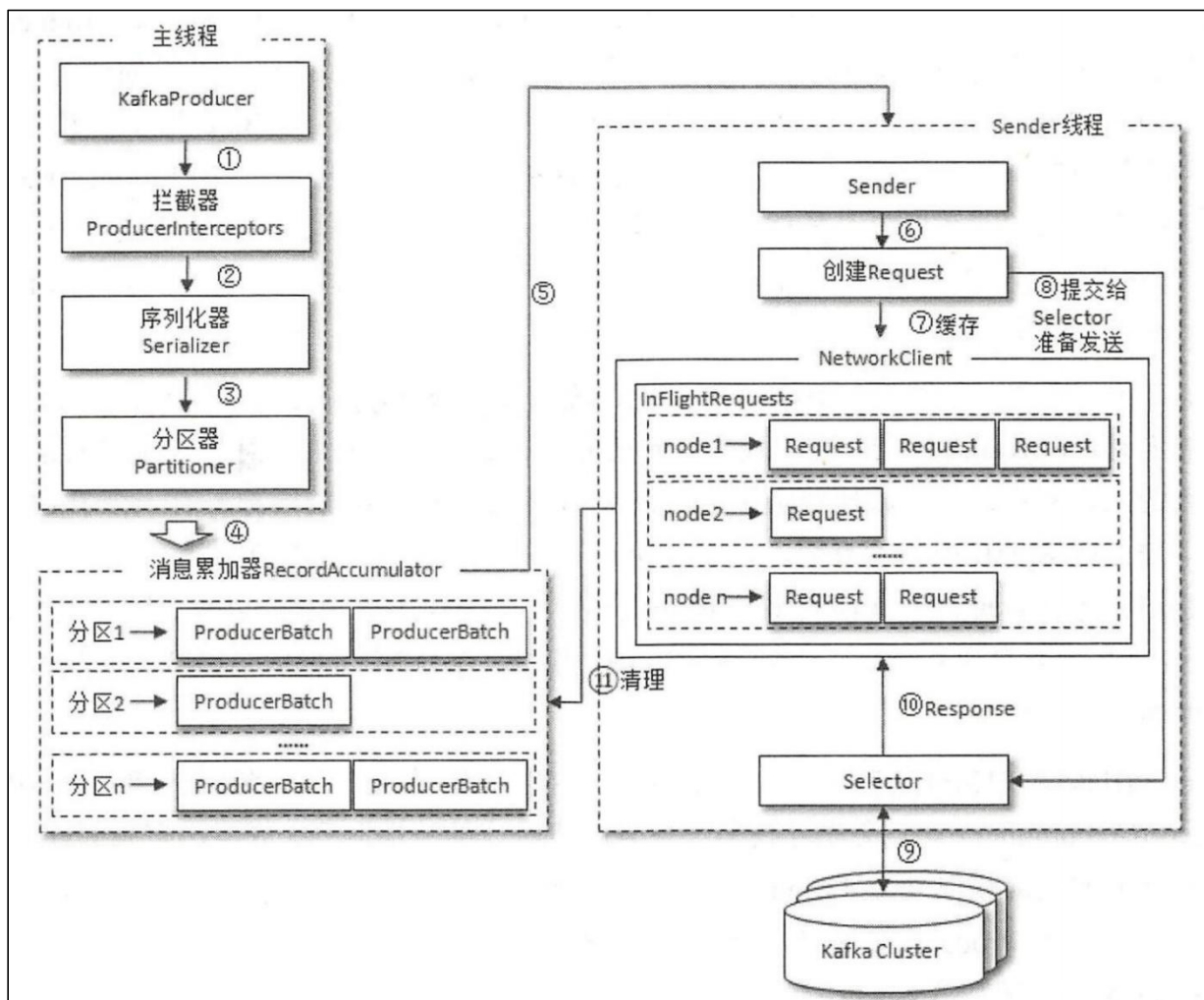
```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class MyProducer {
    public static void main(String[] args) throws InterruptedException {
        Properties props = new Properties();
        // Kafka 服务端的主机名和端口号
        props.put("bootstrap.servers", "doitedu01:9092,doitedu02:9092,doitedu03:9092");
        // 等待所有副本节点的应答
        props.put("acks", "all");
        // 消息发送最大尝试次数
        props.put("retries", 0);
        // 一批消息处理大小
        props.put("batch.size", 16384);
        // 增加服务端请求延时
        props.put("linger.ms", 1);
        // 发送缓存区内存大小
        props.put("buffer.memory", 33554432);
        // key 序列化
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // value 序列化
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<String, String> kafkaProducer = new KafkaProducer<>(props);
        for (int i = 0; i < 50; i++) {
            kafkaProducer.send(new ProducerRecord<String, String>("test", "hello" + i), new Callback() {
                @Override
                public void onComplete(RecordMetadata metadata, Exception exception) {
                    if (metadata != null) {
                        System.out.println(metadata.partition() + "---" + metadata.offset());
                    }
                }
            });
        }
    }
}
```

```
kafkaProducer.close();  
}  
}
```

6.4 生产者原理解析



一个生产者客户端由两个线程协调运行，这两个线程分别为**主线程**和 **Sender** 线程。

在主线程中由 `kafkaProducer` 创建消息，然后通过可能的拦截器、序列化器和分区器的作用之后缓存到消息累加器（`RecordAccumulator`，也称为消息收集器）中。

Sender 线程负责从 `RecordAccumulator` 获取消息并将其发送到 `Kafka` 中；

`RecordAccumulator` 主要用来缓存消息以便 `Sender` 线程可以批量发送，进而减少网络传输的资源消耗以提升性能。`RecordAccumulator` 缓存的大小可以通过生产者客户端参数 `buffer.memory` 配置，默认值为 33554432B，即 32M。如果生产者发送消息的速度超过发送到服务器的速度，则会导致生产者空间不足，这个时候 `KafkaProducer.send()` 方法调用要么被阻塞，要么抛出异常，这个取决于参数

`max.block.ms` 的配置，此参数的默认值为 60000,即 60 秒。

主线程中发送过来的消息都会被迫加到 `RecordAccumulator` 的某个双端队列（`Deque`）中，`RecordAccumulator` 内部为每个分区都维护了一个双端队列，即 `Deque<ProducerBatch>`。消息写入缓存时，追加到双端队列的尾部；

`Sender` 读取消息时，从双端队列的头部读取。注意：`ProducerBatch` 是指一个消息批次；与此同时，会将较小的 `ProducerBatch` 凑成一个较大 `ProducerBatch`，也可以减少网络请求的次数以提升整体的吞吐量。

`ProducerBatch` 大小和 `batch.size` 参数也有着密切的关系。当一条消息（`ProducerRecord`）流入 `RecordAccumulator` 时，会先寻找与消息分区所对应的双端队列（如果没有则新建），再从这个双端队列的尾部获取一个 `ProducerBatch`（如果没有则新建），查看 `ProducerBatch` 中是否还可以写入这个 `ProducerRecord`，如果可以写入，如果不可以则需要创建一个新的 `ProducerBatch`。在新建 `ProducerBatch` 时评估这条消息的大小是否超过 `batch.size` 参数大小，如果不超过，那么就以 `batch.size` 参数的大小来创建 `ProducerBatch`。

如果生产者客户端需要向很多分区发送消息，则可以将 `buffer.memory` 参数适当调大以增加整体的吞吐量。

`Sender` 从 `RecordAccumulator` 获取缓存的消息之后，会进一步将 `<分区,Deque<ProducerBatch>>` 的形式转变成 `<Node,List<ProducerBatch>>` 的形式，其中 `Node` 表示 Kafka 集群 broker 节点。对于网络连接来说，生产者客户端是与具体 broker 节点建立的连接，也就是向具体的 broker 节点发送消息，而并不关心消息属于哪一个分区；而对于 `KafkaProducer` 的应用逻辑而言，我们只关注向哪个分区中发送哪些消息，所以在这里需要做一个应用逻辑层面到网络 I/O 层面的转换。

在转换成 `<Node,List<ProducerBatch>>` 的形式之后，`Sender` 会进一步封装成 `<Node,Request>` 的形式，这样就可以将 `Request` 请求发往各个 `Node` 了，这里的 `Request` 是 Kafka 各种协议请求；

请求在从 `sender` 线程发往 Kafka 之前还会保存到 `InFlightRequests` 中，`InFlightRequests` 保存对象的具体形式为 `Map<NodeId,Deque<request>>`，它的主要作用是缓存了已经发出去但还没有收到服务端响应的请求（`NodeId` 是一个 `String` 类型，表示节点的 `id` 编号）。与此同时，`InFlightRequests` 还提供了许多管理类的方法，并且通过配置参数还可以限制每个连接（也就是客户端与 `Node` 之间的连接）最多缓存的请求数。这个配置参数为 `max.in.flight.request.per.connection`，默认值为 5，即每个连接最多只能缓存 5 个未响应的请求，超过该数值之后就不能再向这个连接发送更多的请求了，除非有缓存的请求收到了响应（`Response`）。通过比较 `Deque<Request>` 的 `size` 与这个参数的大小来判断对应的 `Node` 中是否已经堆积了很多未响应的消息，如果真是如此，那么说明这个 `Node` 节点负载较大或网络连接有问题，再继续其发送请求会增大请求超时的可能。

6.5 重要的生产者参数

6.5.1 acks

acks	含义
0	Producer 往集群发送数据不需要等到集群的返回，不确保消息发送成功。安全性最低但是效率最高。
1	Producer 往集群发送数据只要 Leader 成功写入消息就可以发送下一条，只确保 Leader 接收成功。
-1 或 all	Producer 往集群发送数据需要所有的 ISR Follower 都完成从 Leader 的同步才会发送下一条，确保 Leader 发送成功和所有的副本都成功接收。安全性最高，但是效率最低。

6.5.2 max.request.size

这个参数用来限制生产者客户端能发送的消息的最大值，默认值为 1048576B，即 1MB。一般情况下，这个默认值就可以满足大多数的应用场景了。

这个参数还涉及一些其它参数的联动，比如 broker 端的 `message.max.bytes` 参数，如果配置错误可能会引起一些不必要的异常；比如将 broker 端的 `message.max.bytes` 参数配置为 10，而 `max.request.size` 参数配置为 20，那么当发送一条大小为 15B 的消息时，生产者客户端就会报出异常。

JUST DO IT
多易教育

6.5.3 compression.type

这个参数用来指定消息的压缩方式，默认值为 "none"，即默认情况下，消息不会被压缩。

该参数还可以配置为 "gzip"，"snappy" 和 "lz4"。

对消息进行压缩可以极大地减少网络传输、降低网络 I/O，从而提高整体的性能。

消息压缩是一种以时间换空间的优化方式，如果对时延有一定的要求，则不推荐对消息进行压缩；

6.5.4 retries 和 retry.backoff.ms

`retries` 参数用来配置生产者重试的次数，默认值为 0，即在发生异常的时候不进行任何重试动作。

消息在从生产者发出到成功写入服务器之前可能发生一些临时性的异常，比如网络抖动、leader 副本的选举等，这种异常往往是可以自行恢复的，生产者可以通过配置 `retries` 大于 0 的值，以此通过内部重试来恢复而不是一味地将异常抛给生产者的应用程序。如果重试达到设定的次数，那么生产者就会放弃重试并返回异常。

重试还和另一个参数 `retry.backoff.ms` 有关，这个参数的默认值为 100，它用来设定两次重试之间的时间间隔，避免无效的频繁重试。

Kafka 可以保证同一个分区中的消息是有序的。如果生产者按照一定的顺序发送消息，那么这些消息也会顺序地写入分区，进而消费者也可以按照同样的顺序消费它们。对于某些应用来说，顺序性非常重要，比如 MySQL binlog 的传输，如果出现错误就会造成非常严重的后果：

如果将 `acks` 参数配置为非零值，并且 `max.in.flight.requests.per.connection` 参数配置为大于 1 的值，那可能会出现错序的现象：如果第一批次消息写入失败，而第二批次消息写入成功，那么生产者会重试发送第一批次的消息，此时如果第一次的消息写入成功，那么这两个批次的消息就出现了错序。

一般而言，在保证消息顺序的场合建议把参数 `max.in.flight.requests.per.connection` 配置为 1，而不是把 `acks` 配置为 0，不过这样也会影响整体的吞吐。

6.5.5 batch.size

每个 Batch 要存放 `batch.size` 大小的数据后，才可以发送出去。比如说 `batch.size` 默认值是 16KB，那么里面凑够 16KB 的数据才会发送。

理论上来说，提升 `batch.size` 的大小，可以允许更多的数据缓冲在里面，那么一次 Request 发送出去的数据量就更多了，这样吞吐量可能会有所提升。

但是 `batch.size` 也不能过大，要是数据老是缓冲在 Batch 里迟迟不发送出去，那么发送消息的延迟就会很高。

一般可以尝试把这个参数调节大些，利用生产环境发消息负载测试一下。

6.5.6 linger.ms

这个参数用来指定生产者发送 `ProducerBatch` 之前等待更多消息（`ProducerRecord`）加入 `ProducerBatch` 时间，默认值为 0。

生产者客户端会在 `ProducerBatch` 填满或等待时间超过 `linger.ms` 值时发送出去。增大这个参数的值会增加消息的延迟，但是同时能提升一定的吞吐量。

6.5.7 enable.idempotence

是否开启幂等性功能，详见后续原理加强：

幂等性，就是一个操作重复做，每次的结果都一样！

在 kafka 中，就是，生产者生产的一条消息，如果多次重复发送，在服务器中的结果还是只有一条！

6.5.8 partitioner.classes

用来指定分区器，默认：org.apache.kafka.internals.DefaultPartitioner

自定义 partitioner 需要实现 org.apache.kafka.clients.producer.Partitioner 接口

7 API 开发：consumer 消费者

7.1 消费者 Api 示例

一个正常的消费逻辑需要具备以下几个步骤：

- (1) 配置消费者客户端参数及创建相应的消费者实例；
- (2) 订阅主题；
- (3) 拉取消息并消费；
- (4) 提交消费位移 offset；
- (5) 关闭消费者实例。



```
import org.apache.kafka.clients.consumer.*;
import java.util.Arrays;
import java.util.Properties;

public class MyConsumer {

    public static void main(String[] args) {
        Properties props = new Properties();
        // 定义 kakfa 服务的地址，不需要将所有 broker 指定上
        props.put("bootstrap.servers", "doitedu01:9092");
        // 制定 consumer group
        props.put("group.id", "g1");
        // 是否自动提交 offset
        props.put("enable.auto.commit", "true");
        // 自动提交 offset 的时间间隔
        props.put("auto.commit.interval.ms", "1000");
```

```
// key 的反序列化类
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
// value 的反序列化类
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
// 如果没有消费偏移量记录, 则自动重设为起始 offset: latest, earliest, none
props.put("auto.offset.reset", "earliest");

// 定义 consumer
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

// 消费者订阅的 topic, 可同时订阅多个
consumer.subscribe(Arrays.asList("first", "test", "test1"));

while (true) {
    // 读取数据, 读取超时时间为 100ms
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(),
record.value());
    }
}
```

7.2 必要参数配置

也可以使用如下形式:

```
Properties props = new Properties();
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
props.put(ConsumerConfig.GROUP_ID_CONFIG, groupid);
props.put(ConsumerConfig.CLIENT_ID_CONFIG, clientid);
```

7.3 subscribe 订阅主题

subscribe 有如下重载方法:

```
public void subscribe(Collection<String> topics, ConsumerRebalanceListener listener)

public void subscribe(Collection<String> topics)
```

```
public void subscribe(Pattern pattern, ConsumerRebalanceListener listener)

public void subscribe(Pattern pattern)
```

- 指定集合方式订阅主题

```
consumer.subscribe(Arrays.asList(topic1 ));

consumer.subscribe(Arrays.asList(topic2))
```

- 正则方式订阅主题

如果消费者采用的是正则表达式的方式（`subscribe(Pattern)`）订阅，在之后的过程中，如果有人又创建了新的主题，并且主题名字与正则表达式相匹配，那么这个消费者就可以消费到新添加的主题中的消息。如果应用程序需要消费多个主题，并且可以处理不同的类型，那么这种订阅方式就很有效。

正则表达式的方式订阅的示例如下

```
consumer.subscribe(Pattern.compile ("topic.*" ));
```

利用正则表达式订阅主题，可实现动态订阅；

7.4 assign 订阅主题

消费者不仅可以通过 `KafkaConsumer.subscribe()` 方法订阅主题，还可直接订阅某些主题的指定分区；

在 `KafkaConsumer` 中提供了 `assign()` 方法来实现这些功能，此方法的具体定义如下：

```
public void assign(Collection<TopicPartition> partitions)
```

这个方法只接受参数 `partitions`，用来指定需要订阅的分区集合。

示例如下：

```
consumer.assign(Arrays.asList(new TopicPartition ("tpc_1" , 0),new TopicPartition("tpc_2",1))) ;
```

7.5 subscribe 与 assign 的区别

- 通过 `subscribe()` 方法订阅主题具有消费者自动再均衡功能；

在多个消费者的情况下可以根据分区分配策略来自动分配各个消费者与分区的关系。当消费组的消费者增加或减少时，分区分配关系会自动调整，以实现消费负载均衡及故障自动转移。

- `assign()` 方法订阅分区时，是不具备消费者自动均衡的功能的；

其实这一点从 `assign()` 方法参数可以看出端倪，两种类型 `subscribe()` 都有 `ConsumerRebalanceListener` 类型参数的方法，而 `assign()` 方法却没有。

7.6 取消订阅

既然有订阅，那么就有取消订阅；

可以使用 `KafkaConsumer` 中的 `unsubscribe()` 方法取消主题的订阅，这个方法既可以取消通过 `subscribe(Collection)` 方式实现的订阅；

也可以取消通过 `subscribe(Pattern)` 方式实现的订阅，还可以取消通过 `assign(Collection)` 方式实现的订阅。示例码如下：

```
consumer.unsubscribe();
```

如果将 `subscribe(Collection)` 或 `assign(Collection)` 集合参数设置为空集合，作用与 `unsubscribe()` 方法相同，如下示例中三行代码的效果相同：

```
consumer.unsubscribe();  
  
consumer.subscribe(new ArrayList<String>());  
  
consumer.assign(new ArrayList<TopicPartition>());
```

7.7 消息的消费模式

Kafka 中的消费是基于拉取模式的。消息的消费一般有两种模式：推送模式和拉取模式。推模式是服务端主动将消息推送给消费者，而拉模式是消费者主动向服务端发起请求来拉取消息。

Kafka 中的消息消费是一个不断轮询的过程，消费者所要做的就是重复地调用 `poll()` 方法，`poll()` 方法返回的是所订阅的主题（分区）上的一组消息。

对于 `poll()` 方法而言，如果某些分区中没有可供消费的消息，那么此分区对应的消息拉取的结果就为空。如果订阅的所有分区中都没有可供消费的消息，那么 `poll()` 方法返回为空的集合；

`poll()` 方法具体定义如下：

```
public ConsumerRecords<K, V> poll(final Duration timeout)
```

超时时间参数 `timeout`，用来控制 `poll()` 方法的阻塞时间，在消费者的缓冲区里没有可用数据时会发生阻塞。如果消费者程序只用来单纯拉取并消费数据，则为了提高吞吐率，可以把 `timeout` 设置为 `Long.MAX_VALUE`；

消费者消费到的每条消息的类型为 `ConsumerRecord`

```
public class ConsumerRecord<K, V> {  
    public static final long NO_TIMESTAMP = RecordBatch.NO_TIMESTAMP;  
    public static final int NULL_SIZE = -1;  
    public static final int NULL_CHECKSUM = -1;  
  
    private final String topic;  
    private final int partition;  
    private final long offset;  
    private final long timestamp;  
    private final TimestampType timestampType;  
    private final int serializedKeySize;  
    private final int serializedValueSize;  
    private final Headers headers;  
    private final K key;  
    private final V value;  
  
    private volatile Long checksum;
```

`topic` `partition` 这两个字段分别代表消息所属主题的名称和所在分区的编号。

`offset` 表示消息在所属分区的偏移量。

`timestamp` 表示时间戳，与此对应的 `timestampType` 表示时间戳的类型。

`timestampType` 有两种类型 `CreateTime` 和 `LogAppendTime`，分别代表消息创建的时间戳和消息追加到日志的时间戳。

headers 表示消息的头部内容。

key value 分别表示消息的键和消息的值，一般业务应用要读取的就是 value ；

serializedKeySize、serializedValueSize 分别表示 key、value 经过序列化之后的大小，如果 key 为空，则 serializedKeySize 值为 -1，同样，如果 value 为空，则 serializedValueSize 的值也会为 -1；checksum 是 CRC32 的校验值。

示例代码片段

```
/**
 * 订阅与消费方式 2
 */
TopicPartition tp1 = new TopicPartition("x", 0);
TopicPartition tp2 = new TopicPartition("y", 0);
TopicPartition tp3 = new TopicPartition("z", 0);
List<TopicPartition> tps = Arrays.asList(tp1, tp2, tp3);
consumer.assign(tps);

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    for (TopicPartition tp : tps) {
        List<ConsumerRecord<String, String>> rList = records.records(tp);
        for (ConsumerRecord<String, String> r : rList) {
            r.topic();
            r.partition();
            r.offset();
            r.value();
            //do something to process record.
        }
    }
}
```

7.8 指定位移消费

有些时候，我们需要一种更细粒度的掌控，可以让我们从特定的位移处开始拉取消息，而 KafkaConsumer 中的 seek () 方法正好提供了这个功能，让我们可以追前消费或回溯消费。seek () 方法的具体定义如下：

```
public void seek(TopicPartiton partition,long offset)
```

代码示例:

```
// 在调用 seek 方法之前, 需要先调用一次 poll, 以分配到分区
consumer.poll(Duration.ofMillis(1000));

// 获取所分配到的分区信息
Set<TopicPartition> assignment = consumer.assignment();
for (TopicPartition topicPartition : assignment) {
    // 为指定 partition 设置读取起始 offset
    consumer.seek(topicPartition, 80);
}

// 开始正式消费
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    for (ConsumerRecord<String, String> record : records) {
        // do some process
    }
}
```

7.9 再均衡监听器



一个消费组中, 一旦有消费者的增减发生, 会触发消费者组的 **rebalance** 再均衡;

如果 A 消费者消费掉的一批消息还没来得及提交 **offset**, 而它所负责的分区在 **rebalance** 中转移给了 B 消费者, 则有可能发生数据的重复消费处理。此情形下, 可以通过再均衡监听器做一定程度的补救;

代码示例

```
/**
 * 再均衡处理
 */
consumer.subscribe(Collections.singletonList("tpc_5"), new ConsumerRebalanceListener() {
    // 再均衡开始前和消费者停止读取消息之后, 被调用
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> collection) {
        // store the current offset to db
    }

    // 重新分配到分区后和消费者开始读取消息之前, 被调用
    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> collection) {
        // fetch the current offset from db
    }
});
```

7.10 自动位移提交

Kafka 中默认的消费位移的提交方式是自动提交，这个由消费者客户端参数 `enable.auto.commit` 配置，默认值为 `true`。当然这个默认的自动提交不是每消费一条消息就提交一次，而是定期提交，这个定期的周期时间由客户端参数 `auto.commit.interval.ms` 配置，默认值为 5 秒，此参数生效的前提是 `enable.auto.commit` 参数为 `true`。

在默认的方式下，消费者每隔 5 秒会将拉取到的每个分区中最大的消息位移进行提交。自动位移提交的动作是在 `poll()` 方法的逻辑里完成的，在每次真正向服务端发起拉取请求之前会检查是否可以位移提交，如果可以，那么就会提交上一次轮询的位移。

Kafka 消费的编程逻辑中位移提交是一大难点，自动提交消费位移的方式非常简便，它免去了复杂的位移提交逻辑，让编码更简洁。但随之而来的是重复消费和消息丢失的问题。

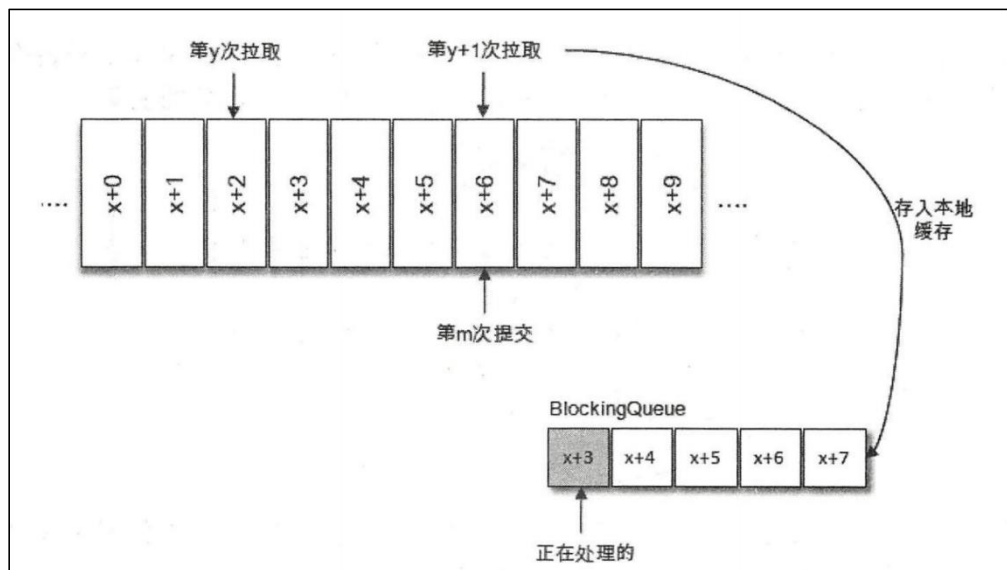
● 重复消费

假设刚刚提交完一次消费位移，然后拉取一批消息进行消费，在下次自动提交消费位移之前，消费者崩溃了，那么又得从上一次位移提交的地方重新开始消费，这样便发生了重复消费的现象（对于再均衡的情况同样适用）。我们可以通过减小位移提交的时间间隔来减小重复消息的窗口大小，但这样并不能避免重复消费的发送，而且也会使位移提交更加频繁。

● 丢失消息

按照一般思维逻辑而言，自动提交是延时提交，重复消费可以理解，那么消息丢失又是在什么情形下会发生的呢？我们来看下图中的情形：

拉取线程不断地拉取消息并存入本地缓存，比如在 `BlockingQueue` 中，另一个处理线程从缓存中读取消息并进行相应的逻辑处理。设目前进行到了第 $y+1$ 次拉取，以及第 m 次位移提交的时候，也就是 $x+6$ 之前的位移已经确认提交了，处理线程却还正在处理 $x+3$ 的消息；此时如果处理线程发生了异常，待其恢复之后会从第 m 次位移提交处，也就是 $x+6$ 的位置开始拉取消息，那么 $x+3$ 至 $x+6$ 之间的消息就没有得到相应的处理，这样便发生消息丢失的现象。



7.11 手动位移提交（调用 kafka api）

自动位移提交的方式在正常情况下不会发生消息丢失或重复消费的现象，但是在编程的世界里异常无可避免；同时，自动位移提交也无法做到精确的位移管理。在 **Kafka** 中还提供了手动位移提交的方式，这样可以使得开发人员对消费位移的管理控制更加灵活。

很多时候并不是说拉取到消息就算消费完成，而是需要将消息写入数据库、写入本地缓存，或者是更加复杂的业务处理。在这些场景下，所有的业务处理完成才能认为消息被成功消费；

手动的提交方式可以让开发人员根据程序的逻辑在合适的地方进行位移提交。开启手动提交功能的前提是消费者客户端参数 `enable.auto.commit` 配置为 `false`，示例如下

```
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
```

手动提交可以细分为同步提交和异步提交，对应于 `KafkaConsumer` 中的 `commitSync()`和 `commitAsync()`两种类型的方法。

- 同步提交的方式

`commitSync()`方法的定义如下：

```
/**
 * 手动提交 offset
 */
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    for (ConsumerRecord<String, String> r : records) {
        //do something to process record.
    }
    consumer.commitSync();
}
```

对于采用 `commitSync()`的无参方法，它提交消费位移的频率和拉取批次消息、处理批次消息的频率是一样的，如果想寻求更细粒度的、更精准的提交，那么就需要使用 `commitSync()`的另一个有参方法，具体定义如下：

```
public void commitSync(final Map<TopicPartition, OffsetAndMetadata> offsets)
```

示例代码如下：

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    for (ConsumerRecord<String, String> r : records) {
        long offset = r.offset();
        //do something to process record.

        TopicPartition topicPartition = new TopicPartition(r.topic(), r.partition());
```

```
consumer.commitSync(Collections.singletonMap(topicPartition,new OffsetAndMetadata(offset+1)));  
}  
}
```

提交的偏移量 = 消费完的 record 的偏移量 + 1

因为，__consumer_offsets 中记录的消费偏移量，代表的是，消费者下一次要读取的位置！！

● 异步提交方式

commitSync() 方法相反，异步提交的方式（commitAsync()）在执行的时候消费者线程不会被阻塞；可能在提交消费位移的结果还未返回之前就开始了新一次的拉取操作。异步提交以便消费者的性能得到一定的增强。commitAsync 方法有一个不同的重载方法，具体定义如下

```
public void commitAsync()  
public void commitAsync(OffsetCommitCallback callback)  
public void commitAsync(final Map<TopicPartition, OffsetAndMetadata> offsets,  
                        OffsetCommitCallback callback)
```

示例代码

```
/**  
 * 异步提交 offset  
 */  
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));  
    for (ConsumerRecord<String, String> r : records) {  
        long offset = r.offset();  
  
        //do something to process record.  
        TopicPartition topicPartition = new TopicPartition(r.topic(), r.partition());  
        consumer.commitSync(Collections.singletonMap(topicPartition,new OffsetAndMetadata(offset+1)));  
        consumer.commitAsync(Collections.singletonMap(topicPartition, new OffsetAndMetadata(offset + 1)), new  
OffsetCommitCallback() {  
            @Override  
            public void onComplete(Map<TopicPartition, OffsetAndMetadata> map, Exception e) {  
                if(e == null ){  
                    System.out.println(map);  
                }else{  
                    System.out.println("error commit offset");  
                }  
            }  
        });  
    }  
}
```

7.12 其他重要参数

`fetch.min.bytes=1B`

一次拉取的最小字节数

`fetch.max.bytes=50M`

一次拉取的最大数据量

`fetch.max.wait.ms=500ms`

拉取时的最大等待时长

`max.partition.fetch.bytes = 1MB`

每个分区一次拉取的最大数据量

`max.poll.records=500`

一次拉取的最大条数

`connections.max.idle.ms=540000ms`

网络连接的最大闲置时长

`request.timeout.ms=30000ms`

一次请求等待响应的最大超时时间

`consumer` 等待请求响应的最长时间



`metadata.max.age.ms=300000`

元数据在限定时间内没有进行更新，则会被强制更新

`reconnect.backoff.ms=50ms`

尝试重新连接指定主机之前的退避时间

`retry.backoff.ms=100ms`

尝试重新拉取数据的重试间隔

`isolation.level=read_uncommitted`

隔离级别！ 决定消费者能读到什么样的数据

`read_uncommitted`: 可以消费到 LSO (LastStableOffset) 位置;

`read_committed`: 可以消费到 HW (High Watermark) 位置

`max.poll.interval.ms`

超过时限没有发起 `poll` 操作，则消费组认为该消费者已离开消费组


```
enable.auto.commit=true
```

开启消费位移的自动提交

```
auto.commit.interval.ms=5000
```

自动提交消费位移的时间间隔

8 API 开发：topic 管理

一般情况下，我们都习惯使用 `kafka-topic.sh` 本来管理主题，如果希望将管理类的功能集成到公司内部系统中，打造集管理、监控、运维、告警为一体的生态平台，那么就需要以程序调用 API 方式去实现。

这种调用 API 方式实现管理主要利用 `KafkaAdminClient` 工具类

`KafkaAdminClient` 不仅可以用来管理 broker、配置和 ACL（Access Control List），还可用来管理主题）它提供了以下方法：

- 创建主题： `CreateTopicsResult createTopics(Collection<NewTopic> newTopics)`。
- 删除主题： `DeleteTopicsResult deleteTopics(Collection<String> topics)`。
- 列出所有可用的主题： `ListTopicsResult listTopics()`。
- 查看主题的信息： `DescribeTopicsResult describeTopics(Collection<String> topicNames)`。
- 查询配置信息： `DescribeConfigsResult describeConfigs(Collection<ConfigResource> resources)`。
- 修改配置信息： `AlterConfigsResult alterConfigs(Map<ConfigResource, Config> configs)`。
- 增加分区： `CreatePartitionsResult createPartitions(Map<String, NewPartitions> newPartitions)`。

构造一个 `KafkaAdminClient`

```
AdminClient adminClient = KafkaAdminClient.create(props);
```

8.1 列出主题

```
ListTopicsResult listTopicsResult = adminClient.listTopics();
Set<String> topics = listTopicsResult.names().get();
System.out.println(topics);
```

8.2 查看主题信息

```
DescribeTopicsResult describeTopicsResult = adminClient.describeTopics(Arrays.asList("tpc_4", "tpc_3"));
Map<String, TopicDescription> res = describeTopicsResult.all().get();
```

```
Set<String> ksets = res.keySet();  
for (String k : ksets) {  
    System.out.println(res.get(k));  
}
```

8.3 创建主题

代码示例:

```
// 参数配置  
Properties props = new Properties();  
props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "doitedu01:9092,doitedu02:9092,doitedu03:9092");  
props.put(AdminClientConfig.REQUEST_TIMEOUT_MS_CONFIG, 3000);  
  
// 创建 admin client 对象  
AdminClient adminClient = KafkaAdminClient.create(props);  
  
// 由服务端 controller 自行分配分区及副本所在 broker  
NewTopic tpc_3 = new NewTopic("tpc_3", 2, (short) 1);  
  
// 手动指定分区及副本的 broker 分配  
HashMap<Integer, List<Integer>> replicaAssignments = new HashMap<>();  
// 分区 0, 分配到 broker0, broker1  
replicaAssignments.put(0, Arrays.asList(0, 1));  
// 分区 1, 分配到 broker0, broker2  
replicaAssignments.put(1, Arrays.asList(0, 2));  
  
NewTopic tpc_4 = new NewTopic("tpc_4", replicaAssignments);  
CreateTopicsResult result = adminClient.createTopics(Arrays.asList(tpc_3, tpc_4));  
  
// 从 future 中等待服务端返回  
try {  
    result.all().get();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
adminClient.close();
```

8.4 删除主题

代码示例：

```
DeleteTopicsResult deleteTopicsResult = adminClient.deleteTopics(Arrays.asList("tpc_1", "tpc_1"));
Map<String, KafkaFuture<Void>> values = deleteTopicsResult.values();
System.out.println(values);
```

8.5 其他管理

除了进行 topic 管理之外，KafkaAdminClient 也可以进行诸如动态参数管理，分区管理等各类管理操作；

9 Kafka 整合

Kafka 和 flume 的整合有 3 种方式：

- 1, 把 kafka 当做 source 的数据源
- 2, 把 kafka 当做 channel
- 3, 把 kafka 作为 sink 的目标存储



9.1 Kafka+Flume

9.1.1 Flume 从 kafka-source 从 kafka 中读取数据

```
a1.sources = r1
a1.channels = c1
a1.sinks = k1

a1.sources.r1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.r1.channels = c1
a1.sources.r1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
a1.sources.r1.kafka.consumer.group.id = g00001
a1.sources.r1.kafka.topics = tpc_2
a1.sources.r1.batchSize = 1000
a1.sources.r1.kafka.consumer.auto.offset.reset = earliest
```

```
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000000
a1.channels.c1.transactionCapacity = 2000

a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
a1.sinks.k1.kafka.topic = tpc_3
a1.sinks.k1.flumeBatchSize = 1000
a1.sinks.k1.kafka.producer.acks = -1
a1.sinks.k1.allowTopicOverride = false
a1.sinks.k1.kafka.producer.linger.ms = 1000
```

9.1.2 Flume 把 kafka 作为 channel



有两种方式:

- 配了一个 source+channel

```
a1.sources = r1
a1.channels = c1

a1.sources.r1.channels = c1
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /root/abc.log

a1.channels.c1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.c1.kafka.topic = flume-channel
a1.channels.c1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
```

- 配了一个 channel+sink

```
a1.channels = c1
a1.sinks = k1

a1.channels.c1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.c1.kafka.topic = flume-channel
a1.channels.c1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092

a1.sinks.k1.channel = c1
```

```
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = hdfs://doitedu01:8020/logdata/%Y-%m-%d/%H/
a1.sinks.k1.hdfs.filePrefix = logdata_
a1.sinks.k1.hdfs.fileSuffix = .log
a1.sinks.k1.hdfs.rollInterval = 0
a1.sinks.k1.hdfs.rollSize = 268435456
a1.sinks.k1.hdfs.rollCount = 0
a1.sinks.k1.hdfs.batchSize = 1000
a1.sinks.k1.hdfs.codeC = gzip
a1.sinks.k1.hdfs.fileType = CompressedStream
```

9.1.3 Flume 把用 kafka-sink 把数据写入 kafka

```
a1.sources = r1
a1.channels = c1
a1.sinks = k1

a1.sources.r1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.r1.channels = c1
a1.sources.r1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
a1.sources.r1.kafka.consumer.group.id = g00001
a1.sources.r1.kafka.topics = tpc_2
a1.sources.r1.batchSize = 1000
a1.sources.r1.kafka.consumer.auto.offset.reset = earliest

a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000000
a1.channels.c1.transactionCapacity = 2000

a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
a1.sinks.k1.kafka.topic = tpc_3
a1.sinks.k1.flumeBatchSize = 1000
a1.sinks.k1.kafka.producer.acks = -1
a1.sinks.k1.allowTopicOverride = false
a1.sinks.k1.kafka.producer.linger.ms = 1000
```

(1) 配置 flume


```
# define
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F -c +0 /root/flume.log
a1.sources.r1.shell = /bin/bash -c

# sink
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = doitedu01:9092,doitedu02:9092,doitedu03:9092
a1.sinks.k1.kafka.topic = test
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1

# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# bind
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```



(2) 启动客户端消费者

(3) 启动 flume，进入 flume bin 目录下

```
./flume-ng agent -c conf/ -n a1 -f jobs/flume-kafka.conf
```

(4) 向日志文件增加数据查看消费情况

```
echo hello >> /root/flume.log
```

9.2 Kafka+sparkStreaming

以 workCount 示意：

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.dstream.{DStream, InputDStream}
import org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WordCount {

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf()
      .setAppName("spark streaming 整合 kafka")
      .setMaster("local[*]")

    val ssc = new StreamingContext(sparkConf, Seconds(1))

    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> "xu01:9092,xu02:9092",
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> "use_a_separate_group_id_for_each_stream",
      "auto.offset.reset" -> "earliest",
      "enable.auto.commit" -> (false: java.lang.Boolean)
    )

    val topics = Array("test")
    // 获取数据
    val stream: InputDStream[ConsumerRecord[String, String]] =
      KafkaUtils.createDirectStream[String, String](
        ssc,
        LocationStrategies.PreferConsistent, // 如果计算节点和 Broker 是同一台节点可以使用 PreferBrokers
        ConsumerStrategies.Subscribe[String, String](topics, kafkaParams)
      )

    stream.foreachRDD(rdd => {
      val words: RDD[Array[String]] = rdd.map(_._value()).map(line => line.split(" "))
    })
  }
}
```



```
val wordAndOne: RDD[(String, Int)] = words.flatMap(arr => arr.map(word => (word, 1)))

// RDD[(K, V)]
val wordCountResult = wordAndOne.reduceByKey(_ + _)
wordCountResult.foreach(println)

})

ssc.start()
ssc.awaitTermination()
}

}
```

10 kafka 原理加强

10.1 日志分段切分条件

日志分段文件切分包含以下几个条件，满足其一即可：

- (1) 当前日志分段文件的大小超过了 broker 端参数 `log.segment.bytes` 配置的值。
`log.segment.bytes` 参数的默认值为 1073741824, 即 1GB
- (2) 当前日志分段中消息的最大时间戳与当前系统的时间戳的差值大于 `log.roll.ms` 或 `log.roll.hours` 参数配置的值。如果同时配置了 `log.roll.ms` 和 `log.roll.hours` 参数，那么 `log.roll.ms` 的优先级高默认情况下，只配置了 `log.roll.hours` 参数，其值为 168, 即 7 天。
- (3) 偏移量索引文件或时间戳索引文件的大小达到 broker 端参数 `log.index.size.max.bytes` 配置的值。`log.index.size.max.bytes` 的默认值为 10485760, 即 10MB
- (4) 追加的消息的偏移量与当前日志分段的偏移量之间的差值大于 `Integer.MAX_VALUE`, 即要追加的消息的偏移量不能转变为相对偏移量 (`offset - baseOffset > Integer.MAX_VALUE`)。

10.2 分区数与吞吐量

Kafka 本身提供用于生产者性能测试的 `kafka-producer-perf-test.sh` 和用于消费者性能测试的 `kafka-consumer-perf-test.sh`, 主要参数如下:

- `topic` 用来指定生产者发送消息的目标主题;

- num-records 用来指定发送消息的总条数
- record-size 用来设置每条消息的字节数;
- producer-props 参数用来指定生产者的配置, 可同时指定多组配置, 各组配置之间以空格分隔与 producer-props 参数对应的还有一个 producer-config 参数, 它用来指定生产者的配置文件;
- throughput 用来进行限流控制, 当设定的值小于 0 时不限流, 当设定的值大于 0 时, 当发送的吞吐量大于该值时就会被阻塞一段时间。

经验: 如何把 kafka 服务器的性能利用到最高, 一般是让一台机器承载 (cpu 线程数*2~3) 个分区
测试环境: 节点 3 个, cpu 2 核 2 线程, 内存 8G , 每条消息 1k

测试结果: topic 在 12 个分区时, 写入、读取的效率都是达到**最高**

写入: 75MB/s , 7.5 万条/s

读出: 310MB/s , 31 万条/s

当分区数>12 或者 <12 时, 效率都比=12 时要低!

10.2.1 生产者性能测试



tpc_3: 分区数 2, 副本数 1

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-producer-perf-test.sh --topic tpc_3 --num-records 100000  
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=doitedu01:9092 acks=1  
100000 records sent, 26068.821689 records/sec (25.46 MB/sec), 926.82 ms avg latency, 1331.00 ms max latency, 924 ms 50th,  
1272 ms 95th, 1305 ms 99th, 1318 ms 99.9th.
```

tpc_4: 分区数 2, 副本数 2

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-producer-perf-test.sh --topic tpc_4 --num-records 100000  
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=doitedu01:9092 acks=1  
100000 records sent, 25886.616619 records/sec (25.28 MB/sec), 962.06 ms avg latency, 1647.00 ms max latency, 857 ms 50th,  
1545 ms 95th, 1622 ms 99th, 1645 ms 99.9th.
```

tpc_5: 分区数 3, 副本数 1

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-producer-perf-test.sh --topic tpc_5 --num-records 100000  
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=doitedu01:9092 acks=1  
100000 records sent, 28785.261946 records/sec (28.11 MB/sec), 789.29 ms avg latency, 1572.00 ms max latency, 665 ms 50th,  
1502 ms 95th, 1549 ms 99th, 1564 ms 99.9th.
```

tpc_6: 分区数 6, 副本数 1

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-producer-perf-test.sh --topic tpc_6 --num-records 100000  
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=doitedu01:9092 acks=1
```

100000 records sent, 42662.116041 records/sec (41.66 MB/sec), 508.68 ms avg latency, 1041.00 ms max latency, 451 ms 50th, 945 ms 95th, 1014 ms 99th, 1033 ms 99.9th.

tpc_12: 分区数

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-producer-perf-test.sh --topic tpc_12 --num-records 100000
--record-size 1024 --throughput -1 --producer-props bootstrap.servers=doitedu01:9092 acks=1
```

100000 records sent, 56561.085973 records/sec (55.24 MB/sec), 371.42 ms avg latency, 1103.00 ms max latency, 314 ms 50th, 988 ms 95th, 1091 ms 99th, 1093 ms 99.9th.

脚本还包含了许多其他的参数，比如 from latest group、print-metrics、threads 等，篇幅及时间限制，同学们可以自行了解这些参数的使用细节。

例如，加上参数：--print-metrics，则会打印更多信息

```
Metric Name                                     Value
app-info:commit-id:{client-id=producer-1}      : 3402a8361b734732
app-info:version:{client-id=producer-1}        : 2.0.0
kafka-metrics-count:count:{client-id=producer-1} : 106.000
producer-metrics:batch-size-avg:{client-id=producer-1} : 15552.902
producer-metrics:batch-size-max:{client-id=producer-1} : 15556.000
producer-metrics:batch-split-rate:{client-id=producer-1} : 0.000
producer-metrics:batch-split-total:{client-id=producer-1} : 0.000
producer-metrics:buffer-available-bytes:{client-id=producer-1} : 33554432.000
producer-metrics:buffer-exhausted-rate:{client-id=producer-1} : 0.000
producer-metrics:buffer-exhausted-total:{client-id=producer-1} : 0.000
producer-metrics:buffer-total-bytes:{client-id=producer-1} : 33554432.000
producer-metrics:bufferpool-wait-ratio:{client-id=producer-1} : 0.058
producer-metrics:bufferpool-wait-time-total:{client-id=producer-1} : 1922445054.000
producer-metrics:compression-rate-avg:{client-id=producer-1} : 1.000
producer-metrics:connection-close-rate:{client-id=producer-1} : 0.000
producer-metrics:connection-close-total:{client-id=producer-1} : 0.000
producer-metrics:connection-count:{client-id=producer-1} : 3.000
producer-metrics:connection-creation-rate:{client-id=producer-1} : 0.089
producer-metrics:connection-creation-total:{client-id=producer-1} : 3.000
```

JUST DO IT
多易教育

10.2.2 消费者性能测试

```
[root@doitedu01 kafka_2.11-2.0.0]# bin/kafka-consumer-perf-test.sh --topic tpc_3 --messages 100000
--broker-list doitedu01:9092 --consumer.config x.properties
```

结果数据个字段含义：

```
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec, rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec
2020-11-14 15:43:42:22, 2020-11-14 15:43:43:347, 98.1377, 106.0948, 100493, 108641.0811, 13, 912, 107.6071, 110189.6930
```

结果中包含了多项信息，分别对应起始运行时间（start.time）、结束运行时 end.time）、消息总量（data.consumed.in.MB，单位为 MB），按字节大小计算的消费吞吐量（单位为 MB）、消费的消息总数（data.consumed.in.nMsg）、按消息个数计算的吞吐量（nMsg.sec）、再平衡的时间（rebalance.time.ms 单位为 MB/s）、拉取消息的持续时间（fetch.time.ms，单位为 ms）、每秒拉取消息的字节大小（fetch.MB.sec 单位 MB/s）、每秒拉取消息的个数（fetch.nMsg.sec）。其中
 $\text{fetch.time.ms} = \text{end.time} - \text{start.time} - \text{rebalance.time.ms}$

10.2.3 分区数与吞吐量实际测试

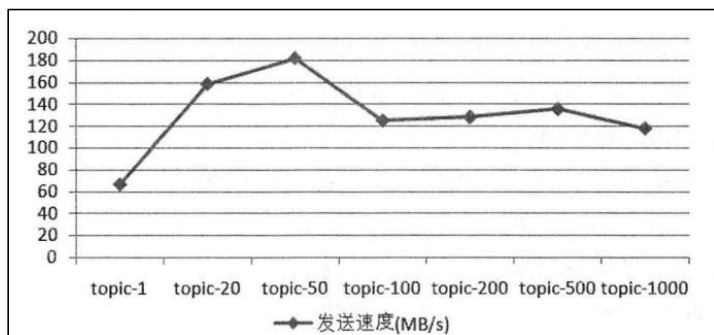
Kafka 只允许单个分区中的消息被一个消费者线程消费，一个消费组的消费并行度完全依赖于所消费的分区的数；

如此看来，如果一个主题中的分区数越多，理论上所能达到的吞吐量就越大，那么事实真的如预想的一样吗？

我们以一个 3 台普通阿里云主机组成的 3 节点 kafka 集群进行测试，每台主机的内存大小为 8GB，磁盘为 40GB，4 核 CPU 16 线程，主频 2600MHZ，JVM 版本为 1.8.0_112，Linux 系统版本为 2.6.32-504.23.4.el6.x86_64。

创建分区数为 1、20、50、100、200、500、1000 的主题，对应的主题名称分别为 topic-1 topic-20 topic-50 topic-100 topic-200 topic-500 topic-1000，所有主题的副本因子都设置为 1。

- 生产者，测试结果如下



- 消费者，测试结果与上图趋势类同

如何选择合适的分区数？从某种意思来说，考验的是决策者的实战经验，更透彻地说，是 Kafka 本身、业务应用、硬件资源、环境配置等多方面的考量而做出的选择。在设定完分区数，或者更确切地说是创建主题之后，还要对其追踪、监控、调优以求更好地利用它。

一般情况下，根据预估的吞吐量及是否与 key 相关的规则来设定分区数即可，后期可以通过增加分区数、增加 broker 或分区重分配等手段来进行改进。

如果一定要给一个准则，则建议将分区数设定为集群中 broker 的倍数，即假定集群中有 3 个 broker 节点，可以设定分区数为 3/6/9 等，至于倍数的选定可以参考预估的吞吐量。

不过，如果集群中的 broker 节点数有很多，比如大几十或上百、上千，那么这种准则也不太适用。

10.3 controller 控制器

在 Kafka 集群中会有一个或者多个 broker, 其中有一个 broker 会被选举为控制器(Kafka Controller), 它负责管理整个集群中所有分区和副本的状态。当某个分区的 leader 副本出现故障时, 由控制器负责为该分区选举新的 leader 副本。当检测到某个分区的 ISR 集合发生变化时, 由控制器负责通知所有 broker 更新其元数据信息。当使用 kafka-topics.sh 脚本为某个 topic 增加分区数量时, 同样还是由控制器负责分区的重新分配。

Kafka 中的控制器选举的工作依赖于 Zookeeper, 成功竞选为控制器的 broker 会在 Zookeeper 中创建 /controller 这个临时 (EPHEMERAL) 节点, 此临时节点的内容参考如下:

```
{"version":1,"brokerid":0,"timestamp":"1529210278988"}
```

其中 version 在目前版本中固定为 1, brokerid 表示称为控制器的 broker 的 id 编号, timestamp 表示竞选成为控制器时的时间戳。

在任意时刻, 集群中有且仅有一个控制器。每个 broker 启动的时候会去尝试去读取 zookeeper 上的 /controller 节点的 brokerid 的值, 如果读取到 brokerid 的值不为 -1, 则表示已经有其它 broker 节点成功竞选为控制器, 所以当前 broker 就会放弃竞选; 如果 Zookeeper 中不存在 /controller 这个节点, 或者这个节点中的数据异常, 那么就会尝试去创建 /controller 这个节点, 当前 broker 去创建节点的时候, 也有可能其他 broker 同时去尝试创建这个节点, 只有创建成功的那个 broker 才会成为控制器, 而创建失败的 broker 则表示竞选失败。每个 broker 都会在内存中保存当前控制器的 brokerid 值, 这个值可以标识为 activeControllerId。



controller 竞选机制: 简单说, 先来先上!

具备控制器身份的 broker 需要比其他普通的 broker 多一份职责, 具体细节如下:

- 监听 partition 相关的变化

为 Zookeeper 中的 /admin/reassign_partitions 节点注册 PartitionReassignmentListener, 用来处理分区重分配的动作。

为 Zookeeper 中的 /isr_change_notification 节点注册 IsrChangeNotificationListener, 用来处理 ISR 集合变更的动作。

为 Zookeeper 中的 /admin/preferred-replica-election 节点添加 PreferredReplicaElectionListener, 用来处理优先副本选举。

- 监听的变化

为 Zookeeper 中的 /brokers/topics 节点添加 TopicChangeListener, 用来处理 topic 增减的变化; 为 Zookeeper 中的 /admin/delete_topics 节点添加 TopicDeletionListener, 用来处理删除 topic 的动作。

- 监听 broker 相关的变化

为 Zookeeper 中的 /brokers/ids/ 节点添加 BrokerChangeListener, 用来处理 broker 增减的变化。

- 更新集群的元数据信息

从 Zookeeper 中读取获取当前所有与 topic、partition 以及 broker 有关的信息并进行相应的管理。对于所有 topic 所对应的 Zookeeper 中的 /brokers/topics/[topic] 节点添加 PartitionModificationsListener, 用来监听 topic 中的分区分配变化。并

将最新信息同步给其他所有 broker。

- 启动并管理分区状态机和副本状态机。
- 如果参数 `auto.leader.rebalance.enable` 设置为 `true`, 则还会开启一个名为“`auto-leader-rebalance-task`”的定时任务来负责维护分区的优先副本的均衡。

10.4 创建 topic 时, partition 是如何分配给 broker 的

客户端请求创建一个 topic 时, 每一个分区副本在 broker 上的分配, 是由集群 controller 来决定; 其分布策略如下:

```
private def assignReplicasToBrokersRackUnaware(nPartitions: Int,
                                              replicationFactor: Int,
                                              brokerList: Seq[Int],
                                              fixedStartIndex: Int,
                                              startPartitionId: Int): Map[Int, Seq[Int]] = {
  val ret = mutable.Map[Int, Seq[Int]]()
  val brokerArray = brokerList.toArray
  val startIndex = if (fixedStartIndex >= 0) fixedStartIndex else rand.nextInt(brokerArray.length)
  var currentPartitionId = math.max(0, startPartitionId)
  var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex else rand.nextInt(brokerArray.length)
  for (_ <- 0 until nPartitions) {
    if (currentPartitionId > 0 && (currentPartitionId % brokerArray.length == 0))
      nextReplicaShift += 1
    val firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.length
    val replicaBuffer = mutable.ArrayBuffer(brokerArray(firstReplicaIndex))
    for (j <- 0 until replicationFactor - 1)
      replicaBuffer += brokerArray(replicaIndex(firstReplicaIndex, nextReplicaShift, j, brokerArray.length))
    ret.put(currentPartitionId, replicaBuffer)
    currentPartitionId += 1
  }
  ret
}

private def replicaIndex(firstReplicaIndex: Int, secondReplicaShift: Int, replicaIndex: Int, nBrokers: Int):
Int = {
  val shift = 1 + (secondReplicaShift + replicaIndex) % (nBrokers - 1)
  (firstReplicaIndex + shift) % nBrokers
}
```

- 副本因子不能大于 Broker 的个数;

- 第 0 个分区（编号为 0）的第一个副本放置位置是随机从 `brokerList` 选择的；
- 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移（也就是如果有 5 个 `Broker`，5 个分区，假设第 0 分区放在第四个 `Broker` 上，那么第 1 分区将会放在第五个 `Broker` 上；第三个分区将会放在第一个 `Broker` 上；第四个分区将会放在第二个 `Broker` 上，依次类推）；
- 剩余的副本相对于第一个副本放置位置其实是由 `nextReplicaShift` 决定的，而这个数也是随机产生的；

10.5 分区 Leader 的选举机制

分区 leader 副本的选举由控制器 `controller` 负责具体实施。

当创建分区（创建主题或增加分区都有创建分区的动作）或 `Leader` 下线（此时分区需要选举一个新的 leader 上线来对外提供服务）的时候都需要执行 leader 的选举动作。

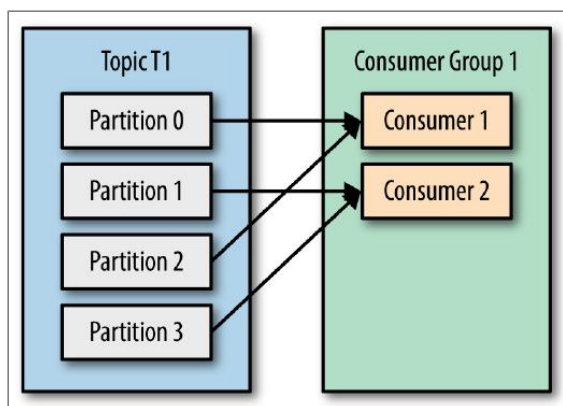
基本思路：

按照 `AR` 集合中副本的顺序查找第一个存活的副本，并且这个副本在 `ISR` 集合中；

一个分区的 `AR` 集合在 `partition` 分配的时候就被指定，并且只要不发生重分配的情况，集合内部副本的顺序是保持不变的，而分区的 `ISR` 集合中副本的顺序可能会改变；

JUST DO IT
多易教育

10.6 消费者组的分区分配策略



在 `kafka` 内部存在两种的分区分配策略：**range（默认）** 和 **round robin**。

当以下事件发生时，kafka 将会进行一次分区分配：

- 同一个 consumer group 内新增消费者
- 消费者离开当前所属的 consumer group，包括 shuts down 或 crashes
- 订阅的主题新增分区

将分区的所有权从一个消费者移到另一个消费者称为再平衡（rebalance），如何 rebalance 也涉及到分区分配策略。

10.6.1 Range Strategy

- 先将消费者按照 client.id 字典排序，然后按 topic 逐个处理；
- 针对一个 topic，将其 partition 总数/消费者数得到 商 n 和 余数 m，则每个 consumer 至少分到 n 个分区，且前 m 个 consumer 每人多分一个分区；

举例说明 2：假设有 TOPIC_A 有 5 个分区，由 2 个 consumer（C1,C2）来消费；

5/2 得到商 2，余 1，则 C1 将分到 3 个分区，C2 为 2 个分区，然后就按照“范围”进行分配：

C1: TOPIC_A-0 TOPIC_A-1 TOPIC_A-2

C2: TOPIC_A-3 TOPIC_A-4

举例说明 2：假设 TOPIC_A 有 5 个分区，TOPIC_B 有 3 个分区，由 2 个 consumer（C1,C2）来消费

- 先分配 TOPIC_A：

5/2 得到商 2，余 1，则 C1 有 3 个分区，C2 有 2 个分区，得到结果

C1: TOPIC_A-0 TOPIC_A-1 TOPIC_A-2

C2: TOPIC_A-3 TOPIC_A-4

- 再分配 TOPIC_B

3/2 得到商 1，余 1，则 C1 有 2 个分区，C2 有 1 个分区，得到结果

C1: TOPIC_B-0 TOPIC_B-1

C2: TOPIC_B-2

- 最终分配结果：

C1: TOPIC_A-0 TOPIC_A-1 TOPIC_A-2 TOPIC_B-0 TOPIC_B-1

C2: TOPIC_A-3 TOPIC_A-4 TOPIC_B-2

10.6.2 Round-Robin Strategy

- 将所有主题分区组成 TopicAndPartition 列表，并对 TopicAndPartition 列表按照其 hashCode 排序
- 然后，以轮询的方式分配给各消费者

以上述“例 2”来举例：

- 先对 TopicAndPartition 的 hashCode 排序，假如排序结果如下：

TOPIC_A-0 TOPIC_B-0 TOPIC_A-1 TOPIC_A-2 TOPIC_B-1 TOPIC_A-3 TOPIC_A-4 TOPIC_B-2

- 然后按轮询方式分配

C1: TOPIC_A-0 TOPIC_A-1 TOPIC_B-1 TOPIC_A-4

C2: TOPIC_B-0 TOPIC_A-2 TOPIC_A-3 TOPIC_B-2

我们可以通过 `partition.assignment.strategy` 参数选择 `range` 或 `roundrobin`。

`partition.assignment.strategy` 参数默认的值是 `range`。

`partition.assignment.strategy=org.apache.kafka.clients.consumer.RoundRobinAssignor`

`partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor`



10.7 group coordinator 组协调器与 rebalance 再均衡

10.7.1 基本概念

每个消费组的子集在服务端对应一个 `GroupCoordinator` 其进行管理, `GroupCoordinator` 是 Kafka 服务端中用于管理消费组的组件。

消费者客户端中由 `ConsumerCoordinator` 组件负责与 `GroupCoordinator` 行交互;

`ConsumerCoordinator` 和 `GroupCoordinator` 最重要的职责就是负责执行消费者 `rebalance` 操作, 包括前面提及的分区分配工作也是在 `rebalance` 期间完成的。

会触发 `rebalance` 的事件可能是如下任意一种:

- 有新的消费者加入消费组。
- 有消费者宕机下线, 消费者并不一定需要真正下线, 例如遇到长时间的 GC、网络延迟导致消费者长时间未向 `Group Coordinator` 发送心跳等情况时, `GroupCoordinator` 会认为消费者已下线。
- 有消费者主动退出消费组 (发送 `LeaveGroupRequest` 请求): 比如客户端调用了 `unsubscribe()` 方法取消对某些主题的订阅。

- 消费组所对应的 GroupCoordinator 节点发生了变更。
- 消费组内所订阅的任一主题或者主题的分区数量发生变化。

10.7.2 消费者加入消费者组的流程

阶段 1: Find Group Coordinator

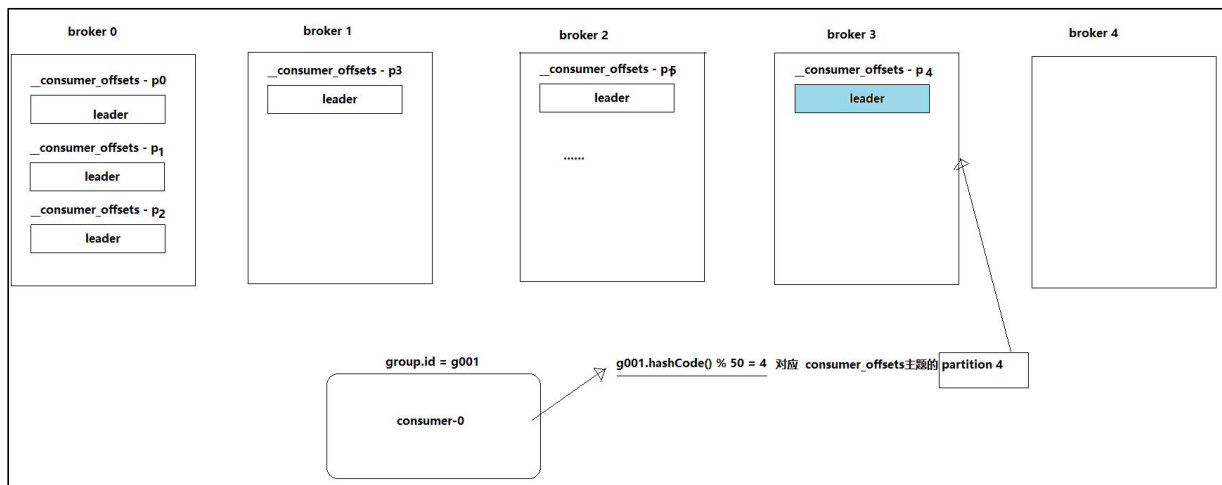
查找 Group Coordinator 的方式:

- 先根据消费组 groupid 的 hash 值计算它应该所在 __consumer_offsets 中的分区编号:

```
Utils.abc(groupId.hashCode()) % groupMetadataTopicPartitionCount
```

groupMetadataTopicPartitionCount 为 __consumer_offsets 的分区总数, 这个可以通过 broker 端参数 offset.topic.num.partitions 来配置, 默认值是 50;

- 找到对应的分区号后, 再寻找此分区 leader 副本所在 broker 节点, 则此节点即为自己的 Grouping Coordinator;

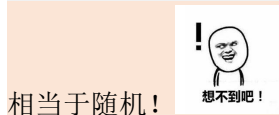


阶段 2: Join The Group

- 此阶段的重要操作之 1: 选举消费组的 leader

```
private val members = new mutable.HashMap[String, MemberMetadata]
```

```
var leaderid = members.keys.head
```



相当于随机!

- 此阶段的重要操作之 2: 选举分区分配策略

最终选举的分配策略基本上可以看作被各个消费者支持的最多的策略, 具体的选举过程如下:

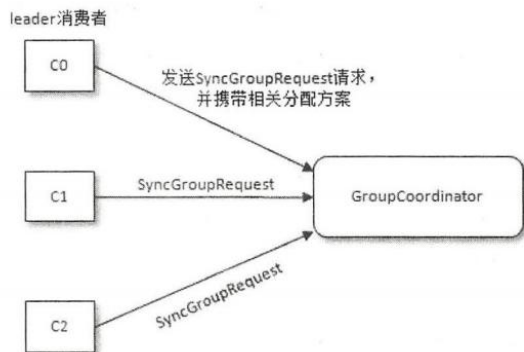
- (1) 收集各个消费者支持的所有分配策略, 组成候选集 candidates

- (2) 每个消费者从候选集 `candidates` 找出第一个自身支持的策略，为这个策略投上一票。
- (3) 计算候选集中各个策略的选票数，选票数最多的策略即为当前消费组的分配策略。

其实，此逻辑并不需要 consumer 来执行，而是由 Group Coordinator 来执行

阶段 3: SYNC Group

此阶段，主要是由消费组 leader 将分区分配方案，通过 Group Coordinator 来转发给组中各消费者



阶段 4: HEART BEAT

进入这个阶段之后，消费组中的所有消费者就会处于正常工作状态。

各消费者在消费数据的同时，保持与 Group Coordinator 的心跳通信；

消费者的心跳间隔时间由参数 `heartbeat.interval.ms` 指定，默认值为 3000，即这个参数必须比 `session.timeout.ms` 参数设定的值要小；一般情况下 `heartbeat.interval.ms` 的配置值不能超过 `session.timeout.ms` 配置值的 1/3。这个参数可以调整得更低，以控制正常重新平衡的预期时间；如果一个消费者发生崩溃，并停止读取消息，那么 GroupCoordinator 会等待一小段时间确认这个消费者死亡之后才会触发再均衡。在这一小段时间内，死掉的消费者并不会读取分区里的消息。这个一小段时间由 `session.timeout.ms` 参数控制，该参数的配置值必须在 broker 端参数 `group.min.session.timeout.ms`（默认值为 6000，即 6 秒）和 `group.max.session.timeout.ms`（默认值为 300000，即 5 分钟）允许的范围内。

10.8 数据可靠性与一致性

Kafka 作为一个商业级消息中间件，消息可靠性的重要性可想而知。可靠性需要从以下两个角度探讨：

- Topic 分区副本机制
- Producer 往 Broker 发送消息

10.8.1 Topic 分区副本

kafka 从 0.8.0 版本开始引入了分区副本。也就是说每个分区可以人为的配置几个副本（创建主题的时候指定 replication-factor，也可以在 broker 级别进行配置 `default.replication.factor`）；

kafka 可以保证单个分区里的事件是有序的，分区可以在线（可用），也可以离线（不可用）。在众多的分区副本里面有一个副本是 Leader，其余的副本是 follower，所有的读写操作都是经过 Leader 进行的，同时 follower 会定期地去 leader 上复制数据。当 Leader 挂了的时候，其中一个 follower 会重新成为新的 Leader。通过分区副本，引入了数据冗余，同时也提供了 kafka 的数据可靠性。

Kafka 的分区多副本架构是 Kafka 可靠性保证的核心，把消息写入多个副本可以使 Kafka 在发生崩溃时仍能保证消息的持久性。

10.8.2 Producer 往 Broker 发送消息

为了让用户设置数据可靠性，kafka 在 producer 里面提供了消息确认机制。我们可以通过配置来决定消息发送到对应分区的几个副本才算消息发送成功。可以在定义 producer 时通过 acks 参数指定（在 0.8.2.X 版本之前是通过 request.required.acks 参数设置的）。这个参数支持以下三种值：

- acks = 0：意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功写入 kafka。在这种情况下还是有可能发生错误，比如发送的对象不能被序列化或者网卡发生故障，但如果是分区离线或整个集群长时间不可用，那就不会收到任何错误。在 acks=0 模式下的运行速度是非常快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不过如果选择了这种模式，一定会丢失一些消息。
- acks = 1：意味着 leader 在收到消息并把它写入到分区数据文件（不一定同步到磁盘上）时会返回确认或错误响应。在这个模式下，如果发生正常的 leader 选举，生产者会在选举时收到一个 LeaderNotAvailableException 异常，如果生产者能恰当地处理这个错误，它会重试发送消息，最终消息会安全到达新的 leader 那里。不过在这个模式下仍然有可能丢失数据，比如消息已经

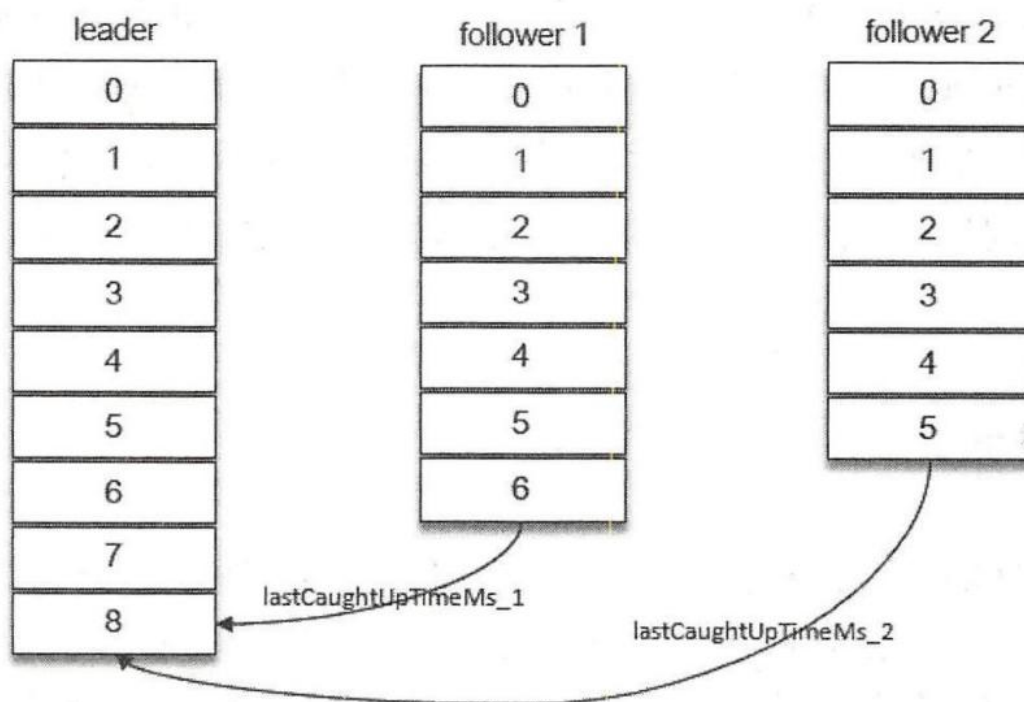
成功写入 leader，但在消息被复制到 follower 副本之前 leader 发生崩溃。

- `acks = all`（这个和 `request.required.acks = -1` 含义一样）：意味着 leader 在返回确认或错误响应之前，会等待所有同步副本都收到消息。如果和 `min.insync.replicas` 参数结合起来，就可以决定在返回确认前至少有多少个副本能够收到消息，生产者会一直重试直到消息被成功提交。不过这也是最慢的做法，因为生产者在继续发送其他消息之前需要等待所有副本都收到当前的消息。

根据实际的应用场景，我们设置不同的 `acks`，以此保证数据的可靠性。

10.8.3 ISR 同步副本列表及 Leader 选举

ISR 概念：（同步副本）。每个分区的 leader 会维护一个 ISR 列表，ISR 列表里面就是 follower 副本的 Borker 编号，只有跟得上 Leader 的 follower 副本才能加入到 ISR 里面，这个是通过 `replica.lag.time.max.ms=10000`（默认值）参数配置的，只有 ISR 里的成员才有被选为 leader 的可能。



前提：

1. `AR = {leader, follower1, follower2}`
2. `replicaMaxLagTimeMs`：由 `replica.lag.time.max.ms` 参数配置，默认为 10000

假设：

1. $\text{now} - \text{lastCaughtUpTimeMs}_1 \leq \text{replicaMaxLagTimeMs}$
2. $\text{now} - \text{lastCaughtUpTimeMs}_2 > \text{replicaMaxLagTimeMs}$

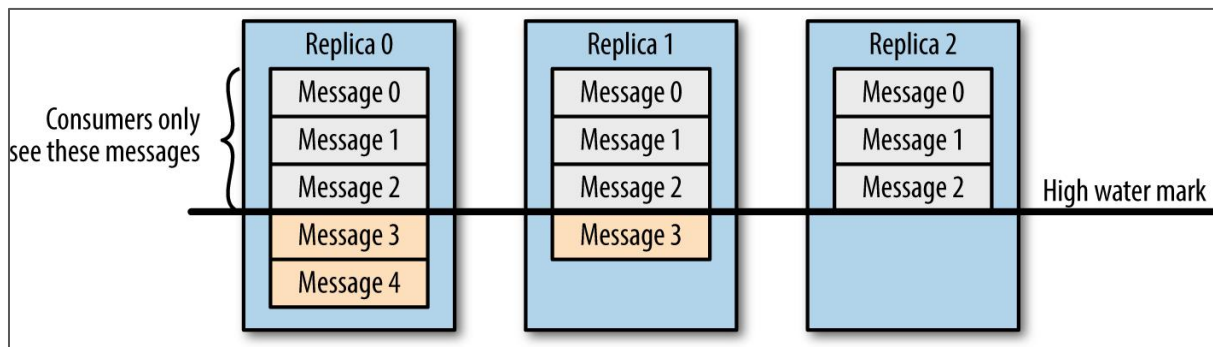
结论：

`ISR = {leader, follower1}`；follower2 失效

所以当 Leader 挂掉了，而且 `unclean.leader.election.enable=false` 的情况下，Kafka 会从 ISR 列表中选择第一个 follower 作为新的 Leader，因为这个分区拥有最新的已经 committed 的消息。通过这个可以保证已经 committed 的消息的数据可靠性。

10.8.4 高水位线 HIGH WATER MARK

数据一致性在此说的是：不论是在旧 Leader 下，还是新选举的 Leader 下，Consumer 都能读到一致的数据。那么 Kafka 是如何实现的呢？

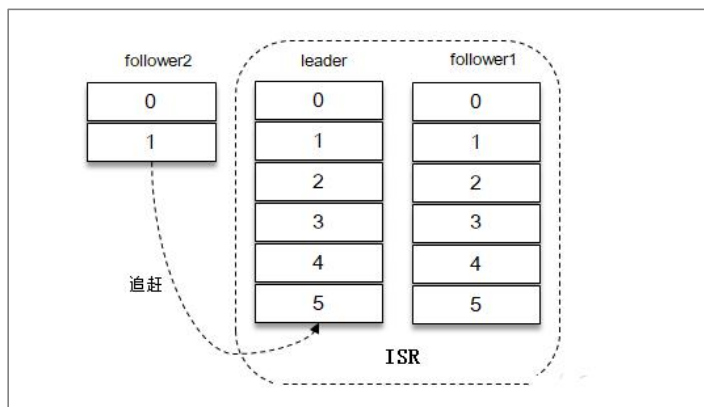


假设分区的副本为 3，其中副本 0 是 Leader，副本 1 和副本 2 是 follower，并且在 ISR 之列。虽然副本 0 已经写入了 Message4，但是 Consumer 只能读取到 Message2。因为所有的 ISR 都同步了 Message2，只有 **High Water Mark (HW 高水位线)** 以上的消息才对 Consumer 可读；而 High Water Mark 取决于 ISR 列表里面偏移量最小的分区，对应于上图的副本 2，这个很类似于木桶原理。

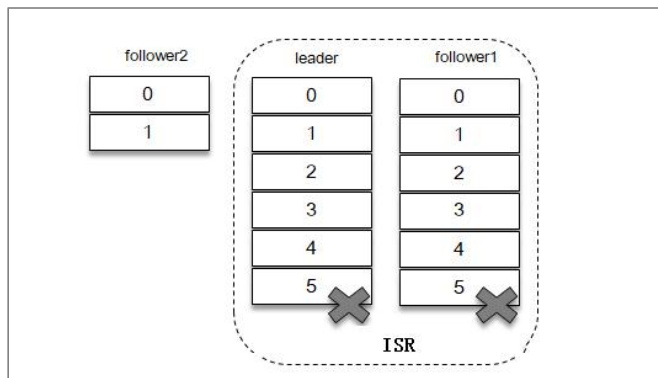
这样做的原因是还没有被足够多副本复制的消息被认为是“不安全”的，如果 Leader 发生崩溃，另一个副本成为新 Leader，那么这些消息很可能丢失。如果我们允许消费者读取这些消息，可能会破坏一致性。试想，一个消费者从当前 Leader（副本 0）读取并处理了 Message4，这个时候 Leader 挂掉了，选举了副本 1 为新的 Leader，这时候另一个消费者再去从新的 Leader 读取消息，发现这个消息其实并不存在，这就导致了数据不一致性问题。

10.8.5 不清洁选举 `unclean.leader.election.enable`

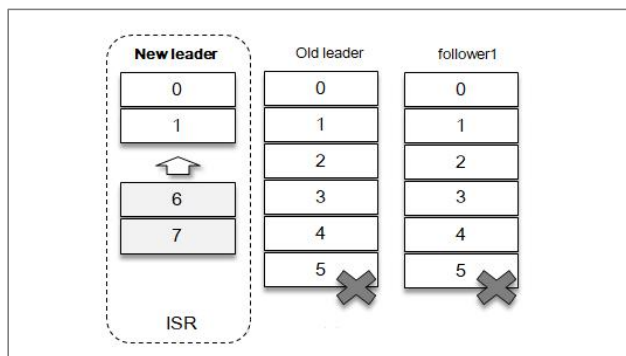
从 Kafka 0.11.0.0 版本开始 `unclean.leader.election.enable` 参数的默认值由原来的 true 改为 false，这个参数背后到底意味着什么，Kafka 的设计者处于什么原因要修改这个默认值？



参考上图，某种状态下，follower2 副本落后 leader 副本很多，并且也不在 leader 副本和 follower1 副本所在的 ISR (In-Sync Replicas) 集合之中。follower2 副本正在努力的追赶 leader 副本以求迅速同步，并且能够加入到 ISR 中。但是很不幸的是，此时 ISR 中的所有副本都突然下线，情形如下图所示：

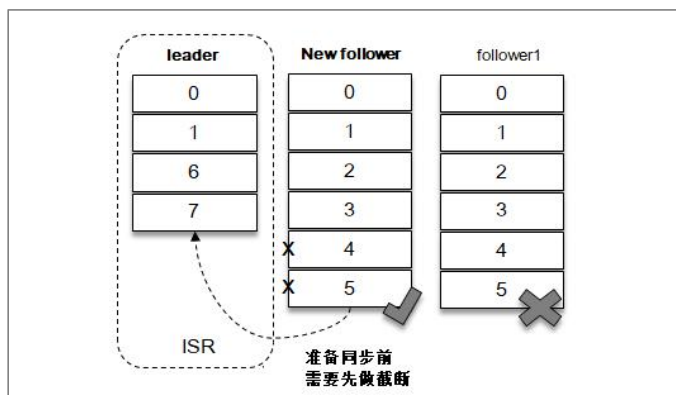


此时 follower2 副本还在，就会进行新的选举，不过在选举之前首先要判断 `unclean.leader.election.enable` 参数的值。如果 `unclean.leader.election.enable` 参数的值为 `false`，那么就意味着非 ISR 中的副本不能够参与选举，此时无法进行新的选举，此时整个分区处于不可用状态。如果 `unclean.leader.election.enable` 参数的值为 `true`，那么可以从非 ISR 集合中选举 follower 副本称为新的 leader。

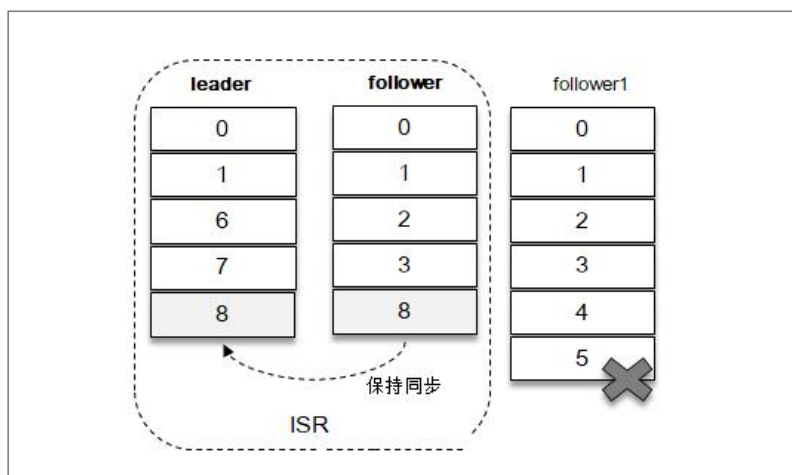


我们进一步考虑 `unclean.leader.election.enable` 参数为 `true` 的情况，在上面的这种情形中 follower2 副本就顺其自然的称为了新的 leader。随着时间的推进，新的 leader 副本从客户端收到了新的消息，如上图所示。

此时，原来的 leader 副本恢复，成为了新的 follower 副本，准备向新的 leader 副本同步消息，但是它发现自身的 LEO (Log End Offset) 比 leader 副本的 LEO 还要大。kafka 中有一个准则，**follower 副本的 LEO 是不能够大于 leader 副本的**，所以新的 follower 副本就需要截断日志至 leader 副本的 LEO 处。



如上图所示，新的 follower 副本需要删除消息 4 和消息 5，之后才能与新的 leader 副本进行同步。之后新的 follower 副本和新的 leader 副本组成了新的 ISR 集合，参考下图。



原本客户端已经成功的写入了消息 4 和消息 5，而在发生日志截断之后就意味着这 2 条消息就丢失了，并且新的 follower 副本和新的 leader 副本之间的消息也不一致。

也就是说，如果 `unclean.leader.election.enable` 参数设置为 `true`，就有可能发生数据丢失和数据不一致的情况，Kafka 的可靠性就会降低；

而如果 `unclean.leader.election.enable` 参数设置为 `false`，Kafka 的可用性就会降低。

具体怎么选择需要读者更具实际的业务逻辑进行权衡，可靠性优先还是可用性优先。从 kafka 0.11.0.0 版本开始将此参数从 `true` 设置为 `false`，可以看出 Kafka 的设计者偏向于可靠性，如果能够容忍 `uncleanLeaderElection` 场景带来的消息丢失和不一致，可将此参数设置为 `true`。

10.8.6 总结综述

综上所述，为了保证数据的可靠性与一致性，我们最少需要配置一下几个参数：

设置 producer 的： `acks=all`（或者 `request.required.acks=-1`）

设置 topic 的： `replication.factor>=3`，并且 `min.insync.replicas>=2`；

设置 `unclean.leader.election.enable=false`

10.9 幂等性

10.9.1 幂等性要点

Kafka 0.11.0.0 版本开始引入了**幂等性**与**事务**这两个特性，以此来实现 EOS (exactly once semantics ，精确一次处理语义)

生产者在发送失败后的重试时 (retries)，有可能会重复写入消息，而使用 **Kafka 幂等性功能**之后就可以避免这种情况。

开启幂等性功能的方式很简单，只需要显式地将生产者客户端参数 `enable.idempotence` 设置为 `true` 即可（这个参数的默认值为 `false`），参考如下：

在开启幂等性功能时，如下几个参数必须正确配置：

- `retries > 0`
- `max.in.flight.requests.per.connection<=5`
- `acks = -1`

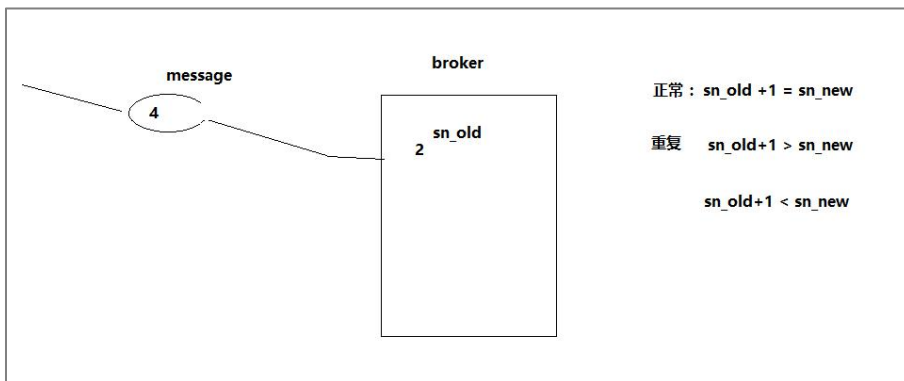
如有违反，则会抛出 `ConfigException` 异常：

10.9.2 kafka 幂等性实现机制

1，每一个 producer 在初始化时会生成一个 `producer_id`，并为每个目标 partition 维护一个“**序列号**”；
2，producer 每发送一条消息，会将`<producer_id,分区>`对应的“序列号”加 1
3，broker 端会为每一对`<producer_id,分区>`维护一个序列号，对于每收到的一条消息，会判断服务端的 `SN_old` 和接收到的消息中的 `SN_new` 进行对比：

- 如果 `SN_OLD+1 = SN_NEW`，正常情况
- 如果 `SN_old+1>SN_new`，说明是重复写入的数据，直接丢弃

- 如果 $SN_old + 1 < SN_new$ ，说明中间有数据尚未写入，或者是发生了乱序，或者是数据丢失，将抛出严重异常：`OutOfOrderSequenceException`



`producer.send("aaa")` 消息 aaa 就拥有了一个唯一的序列号

如果这条消息发送失败，`producer` 内部自动重试（`retry`），此时序列号不变；
`producer.send("aaa")` 消息 aaa 此时又拥有一个新的序列号

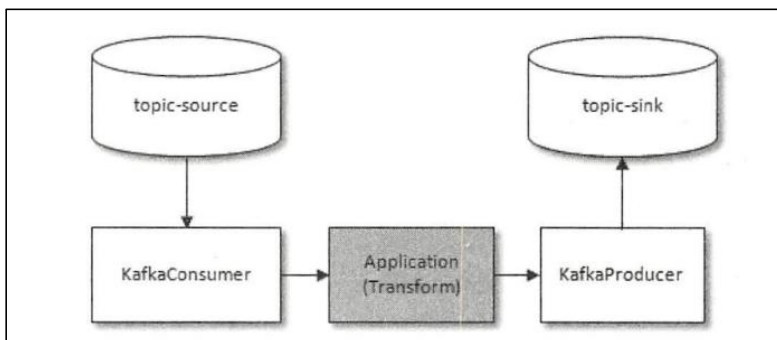
注意：kafka 只保证 `producer` 单个会话中的单个分区幂等；

10.10 kafka 事务

10.10.1 事务要点介绍

JUST DO IT
多易教育

在实际数据处理中，`consume-transform-produce` 是一种常见且典型的场景；



在此场景中，我们往往需要实现，从“读取 source 数据，至业务处理，至处理结果写入 kafka”的整个流程，具备原子性：

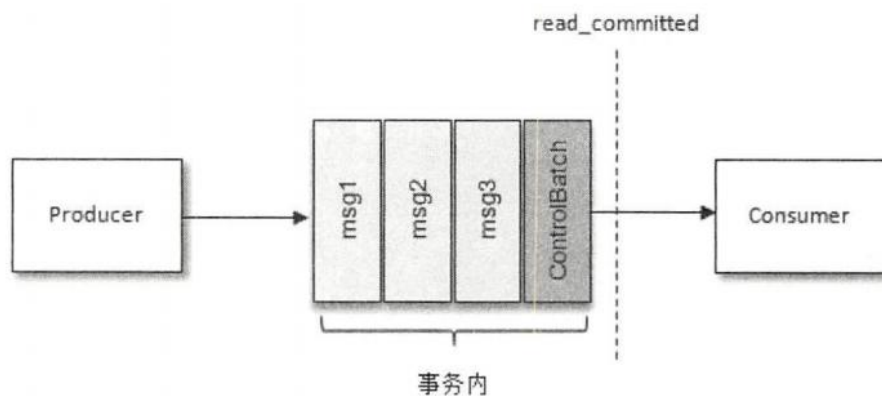
要么全流程成功，要么全部失败！

（也就是说，处理且输出结果成功，才会提交消费端偏移量；
如果处理或输出结果失败，则消费偏移量也不会提交）

要实现上述的需求，可以利用 Kafka 中的**事务机制**：

它可以使应用程序将消费消息、生产消息、提交消费位移当作原子操作来处理，即使该生产或消费会跨多个 topic 分区；

在消费端有一个参数 `isolation.level`，与事务有着莫大的关联，这个参数的默认值为“`read_uncommitted`”，意思是说消费端应用可以看到（消费到）未提交的事务，当然对于已提交的事务也是可见的。这个参数还可以设置为“`read_committed`”，表示消费端应用不可以看到尚未提交的事务内的消息。



控制消息（ControlBatch: COMMIT/ABORT）表征事务是被提交还是被放弃

10.10.2 事务 api 示例

为了实现事务，应用程序必须提供唯一 `transactional.id`，并且开启生产者的幂等性

```
properties.put ("transactional.id","transactionid00001");  
properties.put ("enable.idempotence",true);
```

kafka 生产者中提供的关于事务的方法如下：

```
m abortTransaction() void  
m beginTransaction() void  
m commitTransaction() void  
m initTransactions() void  
m sendOffsetsToTransaction(Map<TopicParti... void  
Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor Next Tip
```

“消费-处理-生产”典型场景下的代码结构示例：

```
package cn.doitedu.kafka.transaction;  
  
import org.apache.kafka.clients.consumer.*;  
import org.apache.kafka.clients.producer.KafkaProducer;  
import org.apache.kafka.clients.producer.ProducerConfig;  
import org.apache.kafka.clients.producer.ProducerRecord;  
import org.apache.kafka.common.TopicPartition;
```

```
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;

import java.time.Duration;
import java.util.*;

/**
 * @author hunter.d
 * @qq 657270652
 * @wx haitao-duan
 * @date 2020/11/15
 */
public class TransactionDemo {
    public static void main(String[] args) {

        Properties props_p = new Properties();
        props_p.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "doitedu01:9092,doitedu02:9092");
        props_p.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        props_p.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props_p.setProperty(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "transaction_id_001");

        Properties props_c = new Properties();
        props_c.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props_c.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props_c.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "doitedu01:9092,doitedu02:9092");
        props_c.put(ConsumerConfig.GROUP_ID_CONFIG, "groupid01");
        props_c.put(ConsumerConfig.CLIENT_ID_CONFIG, "clientid");
        props_c.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // 构造生产者和消费者
        KafkaProducer<String, String> p = new KafkaProducer<String, String>(props_p);
        KafkaConsumer<String, String> c = new KafkaConsumer<String, String>(props_c);
        c.subscribe(Collections.singletonList("tpc_5"));

        // 初始化事务
        p.initTransactions();

        // consumer-transform-produce 模型业务流程
        while(true){
            // 拉取消息
            ConsumerRecords<String, String> records = c.poll(Duration.ofMillis(1000L));
            if(!records.isEmpty()){
```

```
// 准备一个 hashmap 来记录: "分区-消费位移" 键值对
HashMap<TopicPartition, OffsetAndMetadata> offsetsMap = new HashMap<>();

// 开启事务
p.beginTransaction();

try {
    // 获取本批消息中所有的分区
    Set<TopicPartition> partitions = records.partitions();
    // 遍历每个分区
    for (TopicPartition partition : partitions) {
        // 获取该分区的信息
        List<ConsumerRecord<String, String>> partitionRecords = records.records(partition);
        // 遍历每条消息
        for (ConsumerRecord<String, String> record : partitionRecords) {
            // 执行数据的业务处理逻辑
            ProducerRecord<String, String> outRecord = new ProducerRecord<>("tpc_sink",
record.key(), record.value().toUpperCase());
            // 将处理结果写入 kafka
            p.send(outRecord);
        }

        // 将处理完的本分区对应的消费位移记录到 hashmap 中
        long offset = partitionRecords.get(partitionRecords.size() - 1).offset();
        offsetsMap.put(partition, new OffsetAndMetadata(offset+1));
    }

    // 向事务管理器提交消费位移
    p.sendOffsetsToTransaction(offsetsMap, "groupid");
    // 提交事务
    p.commitTransaction();
} catch (Exception e) {
    // 终止事务
    p.abortTransaction();
}

}
```


10.11 Kafka 速度快的原因之：零拷贝

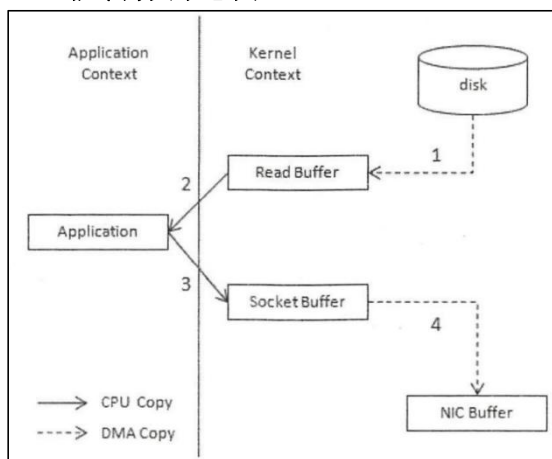
Kafka 速度快的原因：

- 消息顺序追加（磁盘顺序读写比内存的随机读写还快）
- 页缓存等技术（数据交给操作系统的页缓存，并不真正刷入磁盘；而是定期刷入磁盘）
- 使用 Zero-Copy（零拷贝）技术来进一步提升性能；

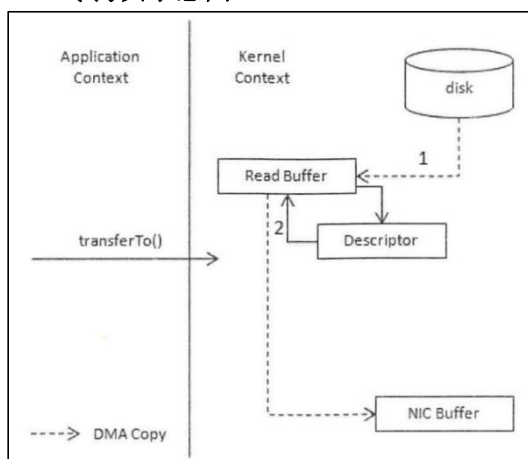
所谓的零拷贝是指将数据直接从磁盘文件复制到网卡设备中，而不需要经由应用程序之手；

零拷贝大大提高了应用程序的性能，减少了内核和用户模式之间的上下文切换；对于 Linux 系统而言，零拷贝技术依赖于底层的 `sendfile()` 方法实现；对应于 Java 语言，`FileChannel.transferTo()` 方法的底层实现就是 `sendfile()` 方法；

● 非零拷贝示意图



● 零拷贝示意图



零拷贝技术通过 DMA (Direct Memory Access) 技术将文件内容复制到内核模式下的 Read Buffer。不过没有数据被复制到 Socket Buffer，只有包含数据的位置和长度的信息的文件描述符被加到 Socket Buffer；DMA 引擎直接将数据从内核模式中传递到网卡设备。

这里数据只经历了 2 次复制就从磁盘中传送出去了，并且上下文切换也变成了 2 次。

零拷贝是针对内核模式而言的，数据在内核模式下实现了零拷贝；

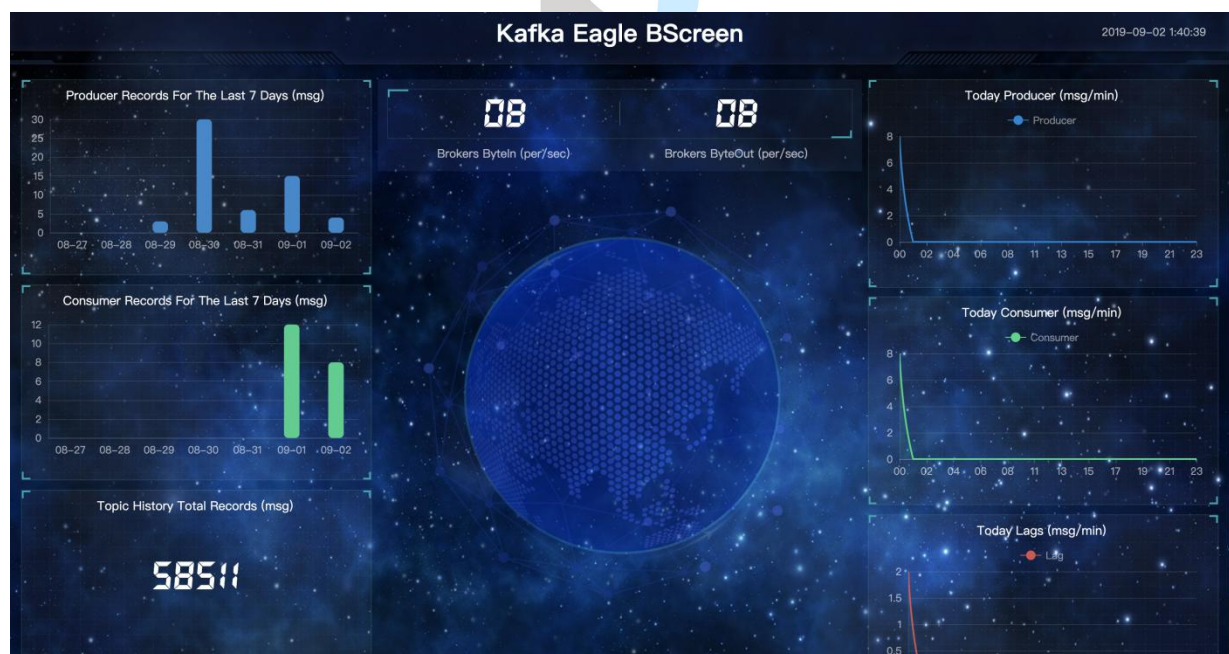
11Kafka 运维监控

11.1 前言

kafka 自身并没有继承监控管理系统，因此对 kafka 的监控管理比较不便，好在有大量的第三方监控管理系统来使用，常见的有：

- Kafka Eagle
- KafkaOffsetMonitor
- Kafka Manager（雅虎开源的 Kafka 集群管理器）
- Kafka Web Console
- 还有 JMX 接口自开发监控管理系统

11.2 Kafka-Eagle 简介



11.3 Kafka-Eagle 安装

安装包下载地址：<https://github.com/smartloli/kafka-eagle-bin/>

官方文档地址: <https://docs.kafka-eagle.org/>

11.3.1 上传, 解压

11.3.2 配置环境变量: JAVA_HOME 和 KE_HOME

```
vi /etc/profile
export JAVA_HOME=/usr/java/jdk1.8
export PATH=$PATH:$JAVA_HOME/bin
export KE_HOME=/data/soft/new/kafka-eagle
export PATH=$PATH:$KE_HOME/bin
```

11.3.3 配置 KafkaEagle



```
cd ${KE_HOME}/conf
```

```
vi system-config.properties
```

```
# Multi zookeeper&kafka cluster list -- The client connection address of the Zookeeper cluster is
set here

kafka.eagle.zk.cluster.alias=cluster1

cluster1.zk.list=tdn1:2181,tdn2:2181,tdn3:2181

#cluster2.zk.list=xtn1:2181,xtn2:2181,xtn3:2181

# Kafka broker nodes online list

cluster1.kafka.eagle.broker.size=3

#cluster2.kafka.eagle.broker.size=20

# Zkcli limit -- Zookeeper cluster allows the number of clients to connect to

kafka.zk.limit.size=25

# Kafka Eagle webui port -- WebConsole port access address

kafka.eagle.webui.port=8048
```

```
# Kafka offset storage -- Offset stored in a Kafka cluster, if stored in the zookeeper, you can not
use this option

cluster1.kafka.eagle.offset.storage=kafka

cluster2.kafka.eagle.offset.storage=kafka

# Whether the Kafka performance monitoring diagram is enabled
kafka.eagle.metrics.charts=false

# Kafka Eagle keeps data for 30 days by default
kafka.eagle.metrics.retain=30

# If offset is out of range occurs, enable this property -- Only suitable for kafka sql
kafka.eagle.sql.fix.error=false

kafka.eagle.sql.topic.records.max=5000

# Delete kafka topic token -- Set to delete the topic token, so that administrators can have the
right to delete

kafka.eagle.topic.token=keadmin

# Kafka sasl authenticate

cluster1.kafka.eagle.sasl.enable=false

cluster1.kafka.eagle.sasl.protocol=SASL_PLAINTEXT

cluster1.kafka.eagle.sasl.mechanism=SCRAM-SHA-256

cluster1.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule
required username="admin" password="admin-secret";

# If not set, the value can be empty

cluster1.kafka.eagle.sasl.client.id=

# Add kafka cluster cgroups

cluster1.kafka.eagle.sasl.cgroup.enable=false

cluster1.kafka.eagle.sasl.cgroup.topics=kafka_ads01,kafka_ads02

# Default use sqlite to store data

#kafka.eagle.driver=org.sqlite.JDBC

# It is important to note that the '/hadoop/kafka-eagle/db' path must exist.

#kafka.eagle.url=jdbc:sqlite:/opt/data/db/ke.db
```

```
#kafka.eagle.username=root

#kafka.eagle.password=smartloli

# <Optional> set mysql address

kafka.eagle.driver=com.mysql.jdbc.Driver

kafka.eagle.url=jdbc:mysql://h3:3306/kedb?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull

kafka.eagle.username=root

kafka.eagle.password=123456
```

如果，数据库选择的是 sqlite，则要手动创建所配置的 db 文件存放目录： /opt/data/db

11.3.4 创建 KafkaEagle 所需的库

```
mysql> CREATE DATABASE IF NOT EXISTS kedb DEFAULT CHARSET utf8 COLLATE utf8_general_ci;
```

11.3.5 启动 KafkaEagle

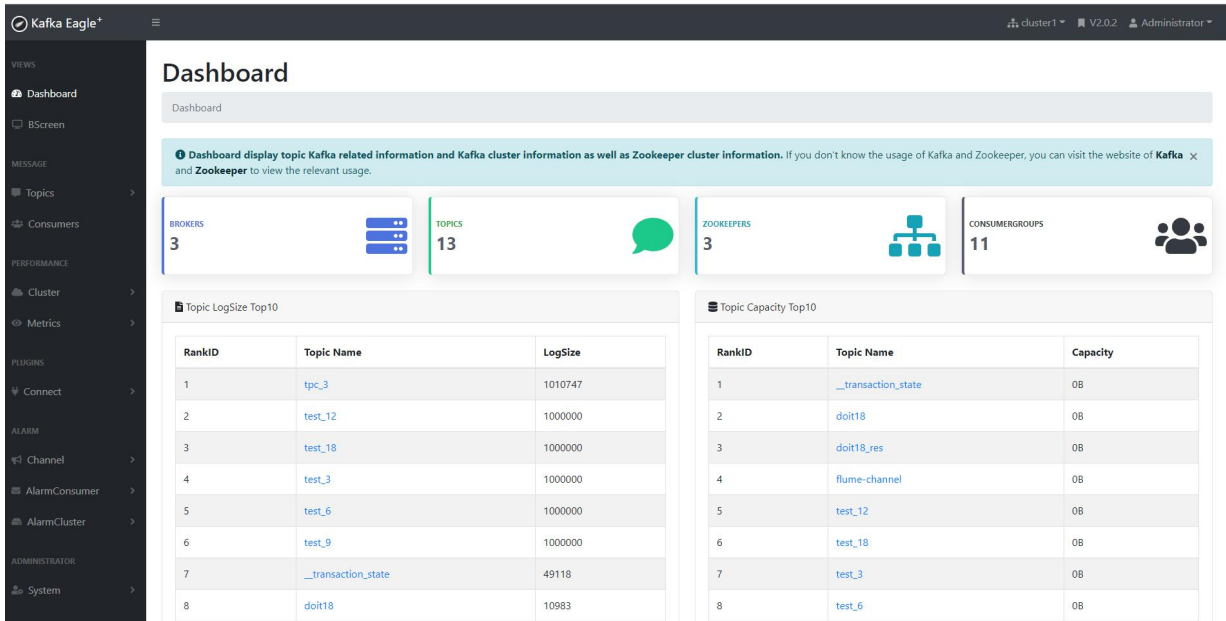
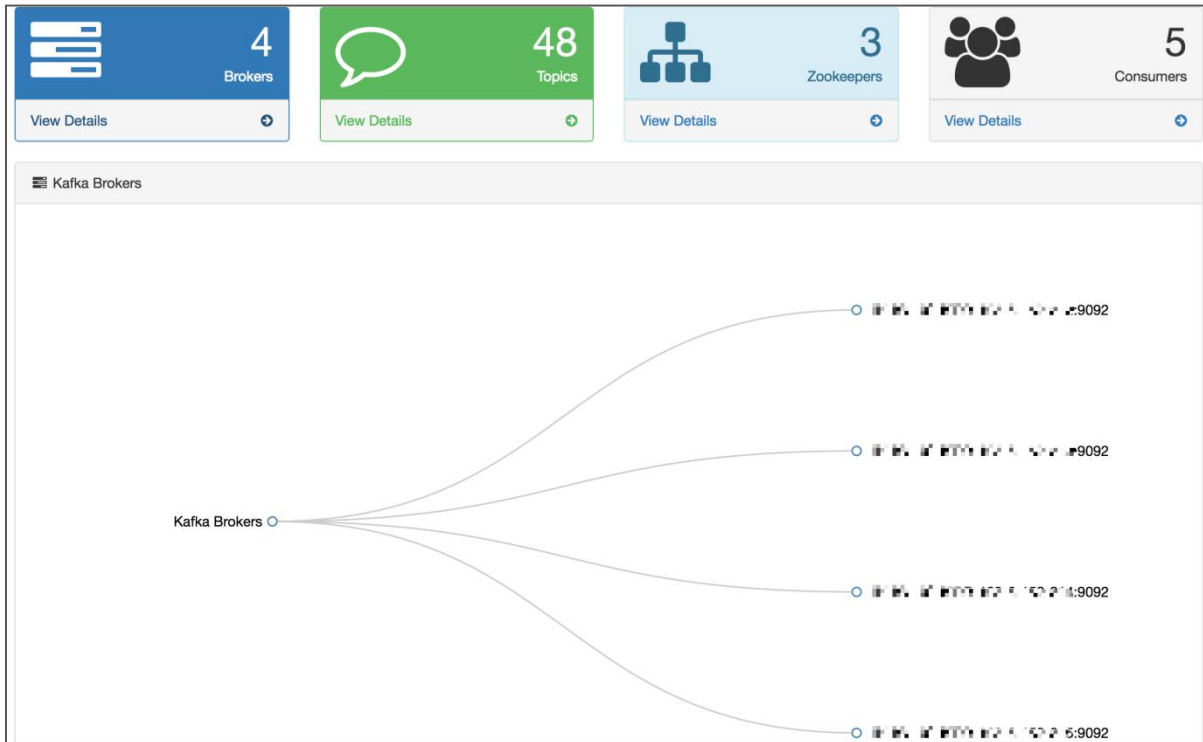


```
cd ${KE_HOME}/bin
chmod +x ke.sh
./ke.sh start
```

```
Welcome to
Kafka Eagle
Version 1.4.3
*****
* Kafka Eagle Service has started success.
* Welcome, Now you can visit 'http://localhost:8048/ke'
* Account:admin ,Password:123456
*****
* <Usage> ke.sh [start|status|stop|restart|stats] </Usage>
* <Usage> https://www.kafka-eagle.org/ </Usage>
*****
```

11.3.6 访问 web 界面

<http://hl:8048/ke>



Kafka Eagle+

VIEWS

Dashboard

BScreen

MESSAGE

Topics

Create

List

KSQL

Mock

Manager

Hub

Consumers

PERFORMANCE

Cluster

Metrics

PLUGINS

Topic

Topic / Create

Create a new kafka's topic.

Topic Property

Topic Name (*)

Made up of letters and digits or underscores . Such as "demo_kafka_topic_1" .

Partitions (*)

1

Partition parameters must be numeric .

Replication Factor (*)

1

Replication Factor parameters must be numeric . Pay attention to available brokers must be larger than replication factor .

Create

Kafka Eagle+

VIEWS

Dashboard

BScreen

MESSAGE

Topics

Create

List

KSQL

Mock

Manager

Hub

Consumers

PERFORMANCE

Cluster

Metrics

PLUGINS

Connect

ALARM

Topic

Topic / List

List all topic information.

Broker Spread: the higher the coverage, the higher the resource usage of kafka broker nodes.

Broker Skewed: the larger the skewed, the higher the pressure on the broker node of kafka.

Broker Leader Skewed: the higher the leader skewed, the higher the pressure on the kafka broker leader node.

APP

1 (APP)

Topic List Info

ID	Topic Name	Partitions	Broker Spread	Broker Skewed	Broker Leader Skewed	Created
1	test_18	18	66%	0%	0%	2020-11-16 14:54:56
2	eagle_demo1	3	100%	0%	0%	2020-11-18 16:50:01
3	tpc_3	1	33%	0%	0%	2020-11-16 11:30:17
4	flume-channel	1	33%	0%	0%	2020-11-16 11:48:10
5	tpc_2	3	100%	0%	33%	2020-11-16 10:54:29

Kafka Eagle+

ZK & Kafka

ZK & Kafka / Overview

Cluster Information, in the form of tables to demonstrate the Kafka and Zookeeper cluster node IP, port, and its version number. If you don't know the usage of Kafka and Zookeeper, Kafka and Zookeeper to view the relevant usage.

Note: Kafka version is "-1" or JMX Port is "-1" maybe kafka broker jmxport disable.

Kafka Cluster Info

ID	IP	Port	JMX Port	Memory(Used Percent)	CPU	Created	Modify
1	doitedu01	9092	-1	NULL	NULL	2020-11-18 16:29:49	2020-11-18 16:29:49
2	doitedu02	9092	-1	NULL	NULL	2020-11-18 08:55:30	2020-11-18 08:55:30
3	doitedu03	9092	-1	NULL	NULL	2020-11-18 16:29:51	2020-11-18 16:29:51

Showing 1 to 3 of 3 entries

Kafka Zookeeper Info

Brokers Metrics

Brokers Metrics / Overview

Through JMX to obtain data, monitor the Kafka client, the production side, the number of messages, the number of requests, processing time and other data to visualize performance.

Kafka Brokers MBean

Rate	Mean	1 Minute	5 Minute	15 Minute
Messages in /sec	0	0	0	0
Bytes in /sec	00	00	00	00
Bytes out /sec	00	00	00	00
Bytes rejected /sec	00	00	00	00
Failed fetch request /sec	0	0	0	0
Failed produce request /sec	0	0	0	0
Total fetch requests /sec	0	0	0	0
Total produce requests /sec	0	0	0	0
Replication byte in /sec	00	00	00	00
Replication byte out /sec	00	00	00	00
Produce message conversions /sec	0	0	0	0

12 kafka 面试题集

12.1 kafka 都有哪些特点？

- 高吞吐量、低延迟：kafka 每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个 topic 可以分多个 partition, consumer group 对 partition 进行 consume 操作。
- 可扩展性：kafka 集群支持热扩展

- 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据副本防止数据丢失
- 容错性：允许集群中节点失败（若副本数量为 n , 则允许 $n-1$ 个节点失败）
- 高并发：支持数千个客户端同时读写

12.2 请简述你在哪些场景下会选择 kafka?

- 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。
- 消息系统：解耦和生产者和消费者、缓存消息等。
- 用户行为跟踪：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。
- 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。
- 作为流式处理的数据源：比如 spark streaming 和 Flink

12.3 kafka 的设计架构你知道吗?

见第二章



12.4 kafka 分区的目的?

分区对于 kafka 集群的好处是：实现负载均衡。

分区对于消费者和生产者来说，可以提高并行度，提高效率。

12.5 kafka 是如何做到消息的有序性?

kafka 中的每个 partition 中的消息在写入时都是有序的（不断追加），而且单独一个 partition 只能由一个消费者去消费，可以在里面保证消息的顺序性。

但是分区之间的消息是不保证有序的。

12.6 kafka 的高可靠性是怎么实现的？

多副本存储

Producer 发送数据时可配置 `ack=all`

12.7 请谈一谈 kafka 数据一致性原理

一致性指的是不论在什么情况下，Consumer 都能读到一致的数据。

HW 高水位线

LEO 等

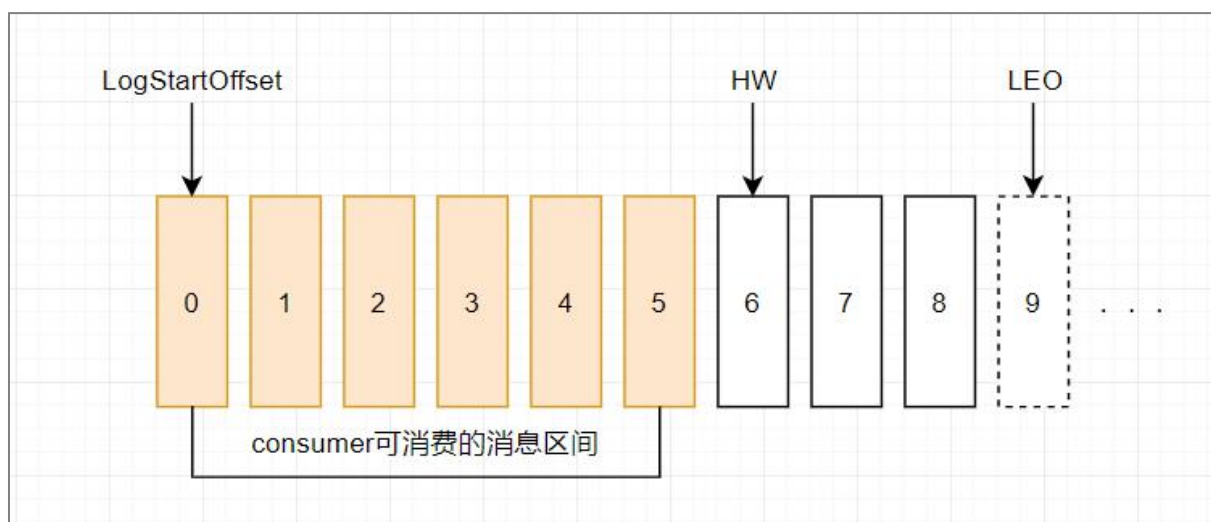
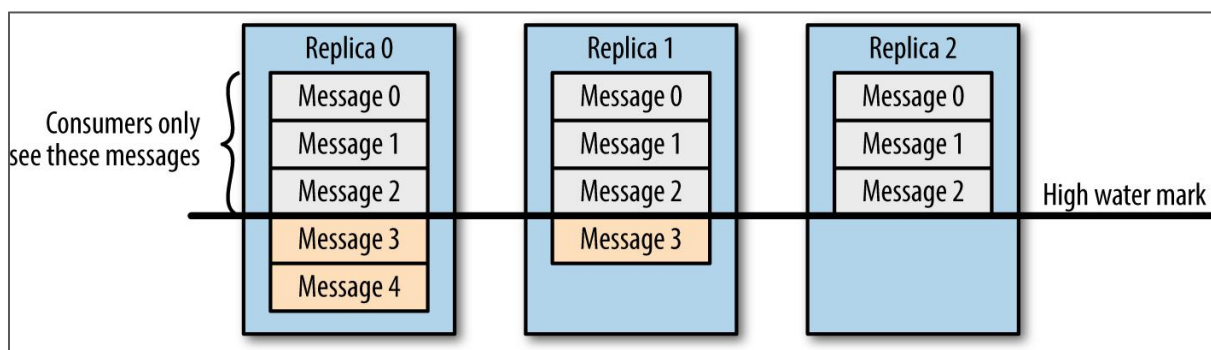
12.8 ISR、OSR、AR 是什么？

- ISR: In-Sync Replicas 同步副本队列
- OSR: Out-of-Sync Replicas
- AR: Assigned Replicas 所有副本

ISR 是由 leader 维护，follower 从 leader 同步数据有一些延迟（具体可以参见 图文了解 Kafka 的副本复制机制），超过相应的阈值会把 follower 剔除出 ISR，存入 OSR（Out-of-Sync Replicas）列表，新加入的 follower 也会先存放在 OSR 中。AR=ISR+OSR。

12.9 LEO、HW、LSO、LW 等分别代表什么

- LEO: 是 LogEndOffset 的简称，代表当前日志文件中末尾下一条待写消息的 offset
- HW: (High Watermark) 俗称高水位，它标识了一个特定的消息偏移量 (offset)，消费者只能拉取到这个 offset 之前的消息。取 partition 对应的 ISR 中最小的 LEO 作为 HW，consumer 最多只能消费到 HW 所在的位置上一条信息。
- LSO: 是 LastStableOffset 的简称，对未完成的事务而言，LSO 的值等于事务中第一条消息的位置 (firstUnstableOffset)，对已完成的事务而言，它的值同 HW 相同
- LW: Low Watermark 低水位，代表 AR 集合中最小的 logStartOffset 值。

JUST DO IT
多易教育

12.10 kafka 在什么情况下会出现消息丢失？

- topic 的副本如果只有 1 个，那么一旦这个副本所在 broker 服务器宕机，则有可能丢失
- producer 往 kafka 写入数据时，如果确认机制参数 acks!=all，也可能造成数据丢失；
- 不清洁选举机制如果开启，也可能造成数据丢失（不清洁选举就是说在所有 ISR 副本全部宕机的情况下，可以让 OSR 副本成为 Leader，而 OSR 中的数据显然不全；那么，就算之前的 Leader 重新上线了，也会被进行日志截断）

12.11 怎么尽可能保证 kafka 的可靠性

副本数

Ack=all

`min.insync.replicas >= 2`

12.12 数据传输的语义有几种？

数据传输的语义通常有以下三种级别：

- 最多一次：消息不会被重复发送，最多被传输一次，但也有可能一次不传输
- 最少一次：消息不会被漏发送，最少被传输一次，但也有可能被重复传输。
- 精确的一次（Exactly once）：不会漏传输也不会重复传输

12.13 kafka 消费者是否可以消费指定分区的信息？

可以

12.14 kafka 消费者是否从指定偏移量开始消费？

可以

12.15 客户端操作 kafka 消息是采用 pull 模式，还是 push 模式？

kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息。

一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。

pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导

致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略。

pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞直到新消息到达（当然也可以阻塞直到消息的数量达到某个特定的量这样就可以批量拉取）

12.16 kafka 的消息格式有了解吗？

V0

V1

crc attributes magic timestamp keylength key valuelength value

V2

参见《2.1.2》

12.17 kafka 高效文件存储设计特点

Kafka 把 topic 中一个 partition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

通过索引信息可以快速定位 message 和确定 response 的最大大小。

通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。

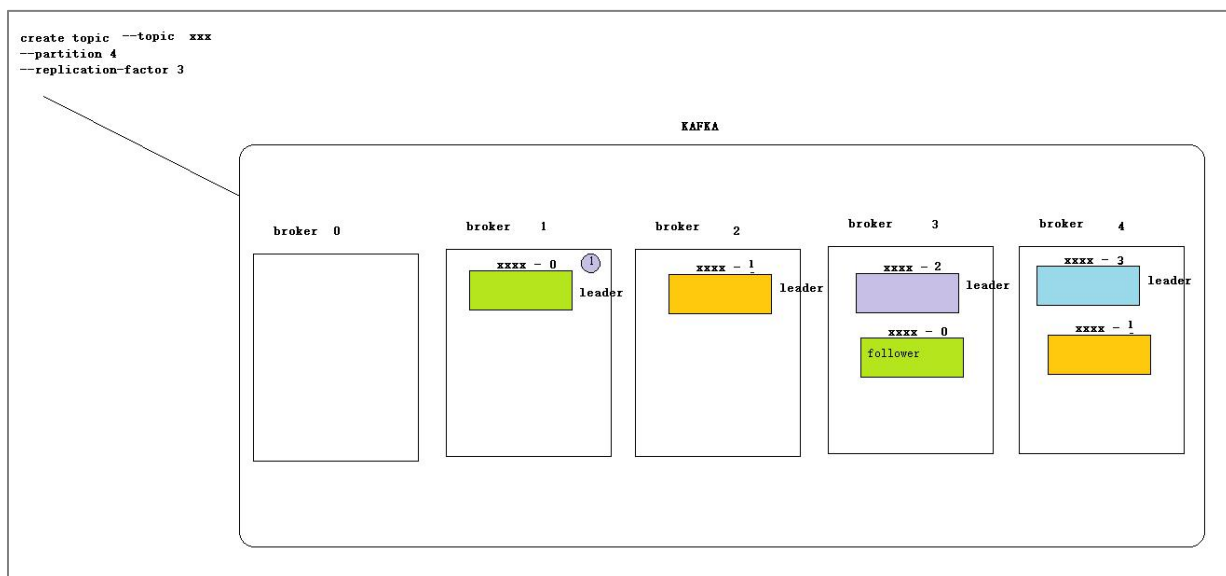
通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小

12.18 kafka 创建 Topic 时如何将分区分配给各 Broker

- 副本因子不能大于 Broker 的个数；
- 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；
- 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个 Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将

会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；

- 剩余的副本相对于第一个副本放置位置其实是由 `nextReplicaShift` 决定的，而这个数也是随机产生的；



12.19 kafka 新建的分区会在哪创建存储目录

我们知道，在启动 Kafka 集群之前，我们需要配置好 `log.dirs` 参数，其值是 kafka 数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

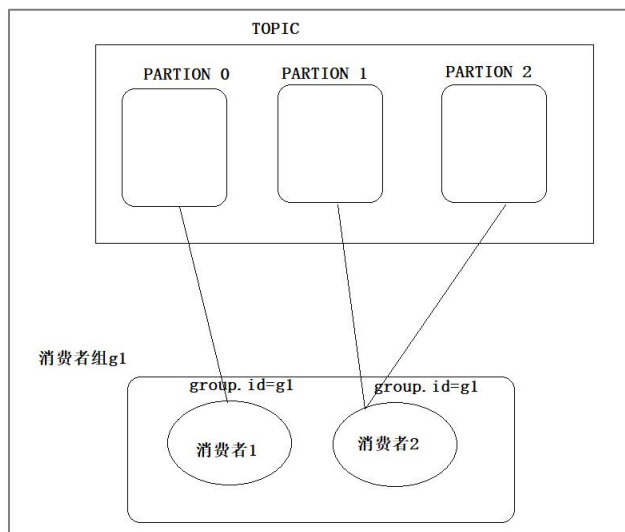
如果 `log.dirs` 参数只配置了一个目录，那么分配到各个 broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。

但是如果 `log.dirs` 参数配置了多个目录，那么 kafka 会在哪个文件夹中创建分区目录呢？答案是：Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic 名+分区 ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给 `log.dirs` 参数新增了一个新的磁盘，新的分区目录肯定是在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

12.20 消费者和消费者组有什么关系？

每个消费者从属于消费组。消费者通过一个参数：`group.id` 来指定所属的组；

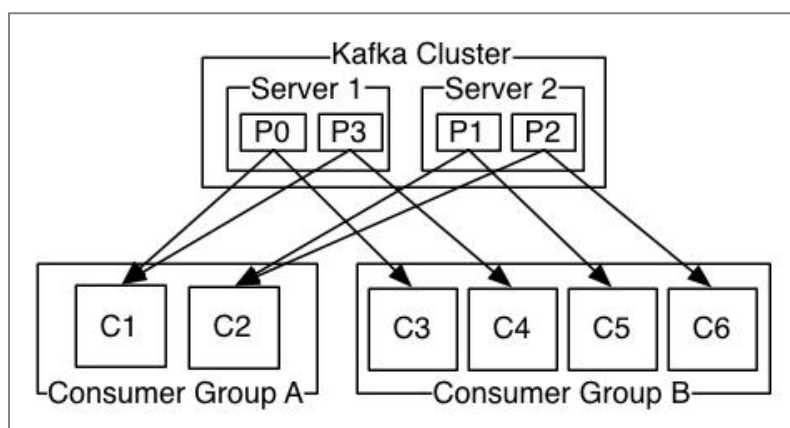
可以把多个消费者的 group.id 设置成同一个值，那么这几个消费者就属于同一个组；
比如，让 c-1, c-2, c-3 的 group.id=“g1”，那么 c-1, c-2, c-3 这 3 个消费者都属于 g1 消费组；



一个消费者，在本质上究竟如何定义：一个消费者可以是一个线程，也可以是一个进程，本质上就是一个 consumer 对象实例！

消费者组的意义：（可以让多个消费者组成一个组，并共同协作来消费数据，提高消费并行度）一个消费组中的各消费者，在消费一个 topic 的数据时，互相不重复！如果 topic 的某分区被组中的一个消费消费，那么，其他消费者就不会再消费这个分区了；

具体关系如下：



12.21 谈一谈 kafka 的消费者组分区分配再均衡

在 Kafka 中，当有新消费者加入或者订阅的 topic 数发生变化时，会触发 rebalance(再均衡：在同一个消费者组当中，分区的所有权从一个消费者转移到另外一个消费者)机制，Rebalance 顾名思义就是重新均衡消费者消费。

Rebalance 的过程如下：

第一步：所有成员都向 coordinator 发送请求，请求入组。一旦所有成员都发送了请求，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。

第二步：leader 开始分配消费方案，指明具体哪个 consumer 负责消费哪些 topic 的哪些 partition。一旦完成分配，leader 会将这个方案发给 coordinator。coordinator 接收到分配方案之后会把方案发给各个 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。

对于 rebalance 来说，coordinator 起着至关重要的作用



12.22 谈谈 kafka 消费者组分区分配策略

Range 策略

Round-Robin 策略

12.23 kafka 是如何实现高吞吐率的？

kafka 是分布式消息系统，需要处理海量的消息，kafka 的设计是把所有的消息都写入速度低容量大的硬盘，以此来换取更强的存储能力，但实际上，使用硬盘并没有带来过多的性能损失。kafka 主要使用了以下几个方式实现了超高的吞吐率：

- 顺序读写；
- 零拷贝（这是 kafka 收发数据时用的一种操作系统上的底层机制）
- 页缓存机制
- 文件分段
- 批量读写
- 数据压缩（不一定能提高吞吐率）

12.24 kafka 监控插件都有哪些？

kafka manager

kafka-offset-monitor：主要做消费者偏移量的监控

kafka-eagle：功能很强大！

12.25 kafka 分区数可以增加或减少吗？为什么？

kafka 允许对 topic 动态增加分区，但不支持减少分区

Kafka 分区数据不支持减少是由很多原因的，比如减少的分区其数据放到哪里去？是删除，还是保留？删除的话，那么这些没消费的消息不就丢了。如果保留这些消息如何放到其他分区里面？追加到其他分区后面的话那么就破坏了 Kafka 单个分区的有序性。如果为了保证删除分区数据插入到其他分区保证有序性，那么实现起来逻辑就会非常复杂。

12.26 kafka 的分区分布策略是怎样的？

分区分布的计算策略如下

- 副本因子不能大于 Broker 的个数；
- 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；
- 其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果有 5 个 Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；
- 剩余副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的；