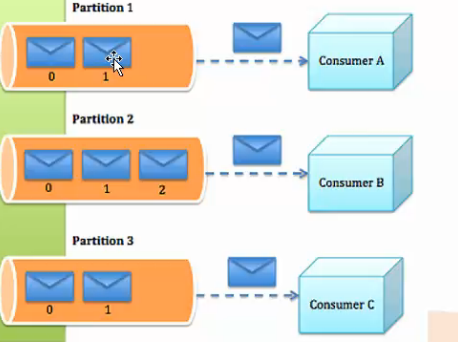
那在kafka里面有很重要的几个名词，其中这个partition相当于就是Topic内部然后存的这个数据是吧？Topic是个逻辑概念，然后patition相当于承载这么一个数据的这么一个角度，那消费的时候有一个特点，这个ConsumerA和ConsumerB和ConsumerC然后和这个partition分别建立了一定的关系（如下图1所示）

图1：



那其中你会发现这个只要是这个ConsumerA和partition1正在连接的过程中当中，然后其他的Consumer是不允许再从这个partition1这里面读数据的，就是说这个ConsumerA正在读这个partition1，那么其他的Consumer从这个partition1上同步读数据，然后这个一个Topic里面这个Partition的这个个数，我们是可以再去创建一个Topic的时候你可以通过配置去配置的，有一个不太好理解的就是有一个Consumer group，这个Consumer group就是说便于多播这么一个引入的概念，就是说原来是一个Consumer 它可以接收所有的数据是吧？那这个时候如果是把一个group当作一个Consumer的时候，那相当于是在这个Consumer内部是只能获得一部分数据，而Consumer之间是不能够有重复的。

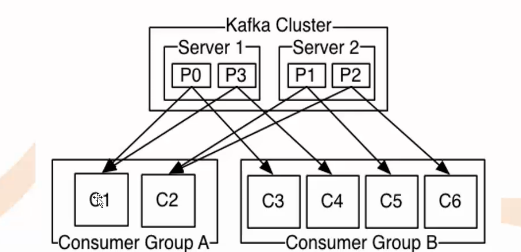
还有一个offser就是说partition当中它是有一定的偏移的是吧？就是你数据存储的时候永远是从最后追加的，然后追加的过程中其实不是每一个offser不是存的真正的数据，而是标识着你当前这个数据相对于你这个partition从0这个节点的一个偏移量（如下图2所示）

图2：



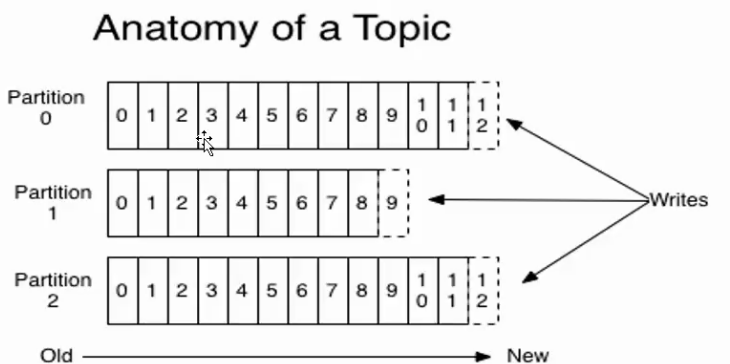
所以就是说kafka是使用了Topic以及partition的概念，并且这个partition是隶属于Topic的，如果一个Topic有多个partition的话，那每个partition是Consumer消费的一个最基本的单元，那如果要是有一个Topic它包含了多个partition，那最多有多少个Consumer同时在一个Consumer group里面可以进行对这个Topic进行消费呢？ 那也就是说有多少partition就是有多少个是不是？如果要是你后面这个Consumer多的话没有办法，因为你每一个partition只能同一时间服务于一个Consumer（如下图3所示）

图3：



另外一个就是offser偏移量（如下图4所示）

图4：



这个偏移量是kafka不会帮你维护的，是为了让整个的系统变的简单，那变得简单，因为你让kafka在去记录每一个Consumer，因为每一个Consumer它读的偏移是不一样的是不是？那你再让kafka去存储这个维护这个消息的话，会势必给这个系统造成了一些更复杂的一些设计，因为没有这么一个复杂的设计在里面的话，所以kafka可以做的更稳定，然后另外kafka它是一个顺序写的这么一个特点，就是体现在了你的partition在顺序去追加这么一个过程，还有就是因为你这个顺序写，所以你写的速度是非常快的，然后另外一个是kafka是无状态的，无状态的意思就是说刚才我说过了就kafka不会保存你的每一个客户端或者每一个消费者的一个偏移量，有了这么几个基本的概念之后，然后我们开始看一下kafka的一些深入的一些内容

那我们看一下第一个是持久化，持久化就是说当你这个partition也就是落地到这个数据文件里面的，那如果说只有一个文件会怎么样呢？就是你的partition只有一个文件，那这个时候数据会直接添加你的尾部。

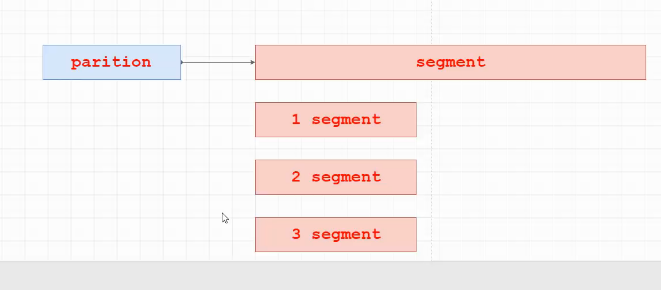
其实partition内部是有一个真正落地的数据，这个数据叫segment，那假设说我一个partition对应着一个文件的话是有什么样的特点呢？就是说你来的这个数据你给partition写的这个数据永远是在这个segment的尾部去追加是不是？因为它不能够随机的去你在任何一个位置去插入数据，那因为你随机去填写，那他就有了一个OE这么一个操作，就是操作非常快，那这是你写数据是非常快的，那假设说你要是去查数据会不会有什么问题呢？那你想随着这个数据的不断的积累，那你这个partition若只对着这个文件的话，这个文件是永远不断的去增长的，随后就增长的非常大（如下图5所示）

图5：



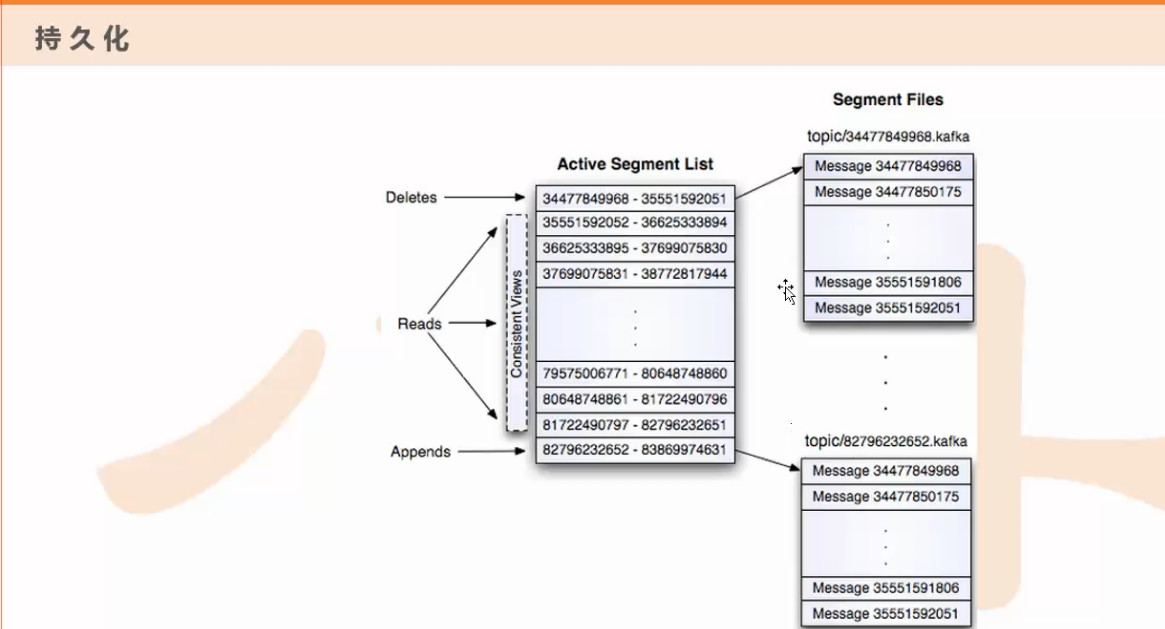
那这个时候你想查这个文件，你在这个大的文件里面想查某一个数据怎么查呢？因为你这个数据是从头往后写的是吧？那你查的话最简单的方式就从前往后进行顺序查找，而且这个顺序查找是非常慢，那有什么方法可以把这个数据查找变得更快一点，所以kafka内部是实际上是这样做的，它是用多个segment去把这个大的segment给它拆分，这样的话比如说我先写1号segment，写完之后再写2号segment，2号写完了写3号，所以你会发现一个特点，这个1号segment，2号segment，3号segment当然它还在不断的产生数据，而且产生的时候这个segment这个文件大小基本上也都是一致的，那随着这个数据的不断积累，那你这个segment也是不断的增多是吧？但是你越旧的数据或者越早的数据，是存到了前面数字越小的segment里面去，越新的数据就存在了这个数字的越大的segment这里面去（如下图6所示）

图6：



那所以你想查一个历史的数据的时候，然后你可以知道了这个数据的一个offser，但是你不知道这个offser到底是具体在哪一个segment上，那这个时候用二分法，这个类似于我们之前弄的那个IP查找是吧？思路有点像，那你一旦你定位到了某一个segment里面之后，好了相当于是每一个segment里面它其实这个数据都不是特别大，那接下来怎么快速定位你真正的数据呢？那这个时候没办法，也只能通过顺序查找的方式，所以在真正的partition设计里面，一个partition它会对应着多个segment文件，而且每一个segment文件，这下面有一个图（如下图7所示）

图7：



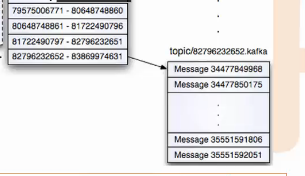
比如那个上图7中最大的那个图是segment的list，那这上图7中有三个segment 文件，然后每一个segment文件相当于是你的头部有一个索引号，尾部有一个消息号，那在这个消息范围内的所有的数据都是落到这个文件里面（如下图8所示）

图8：



然后还有一个范围内的数据都是落到了下面这个文件里面（如下图9所示）

图9：

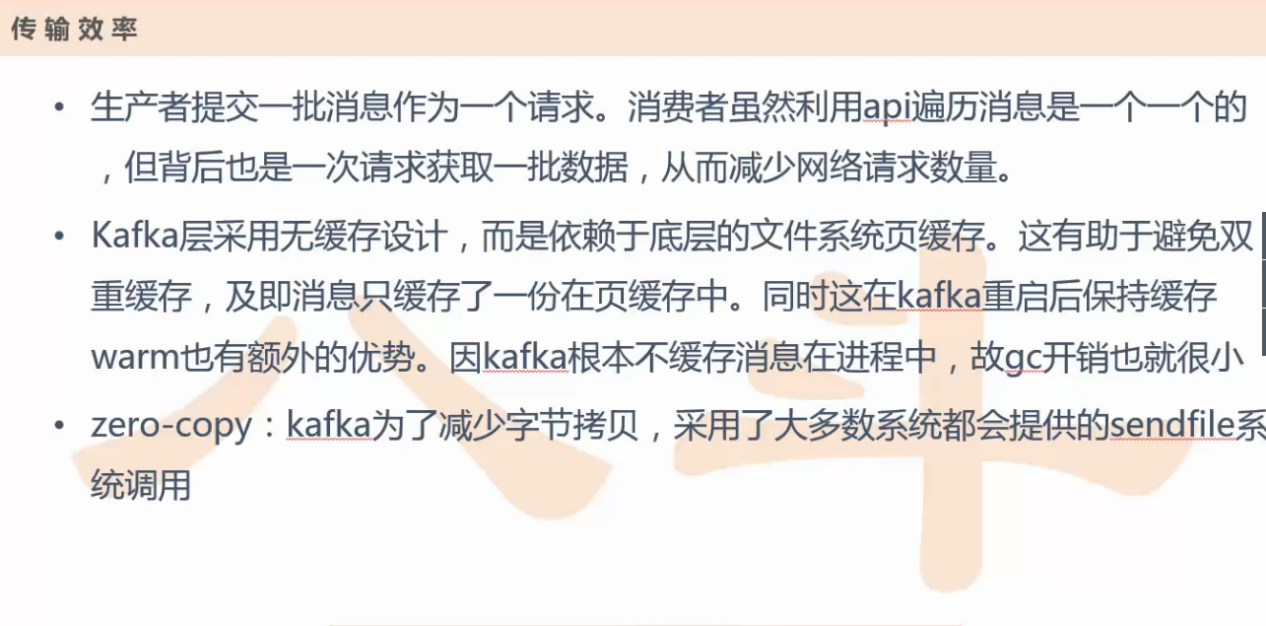


（ps：注意上图8和图9都是头和尾是不一样的，所以是错的图，这个段是不一样的，所以学习仅供参考！）

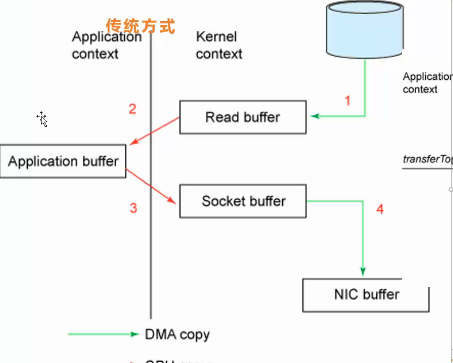
有点类似于我们之前弄的那个ip查找很像，左边一个数字右边一个数字，其实总体来说是保持一个全局排序的这么一个特点，那这个时候你要是知道了某一个消息的话，那你知道这个消息号你可以通过这个id去二分法去命中到具体它是存在于哪个segment里面是吧？然后在这个segment开始从前往后开始遍历，它是这么一个过程，所以这个segment就是一个逻辑日志，每一个partition都对应着一个逻辑日志，这个逻辑日志相当于是由多个实际的叫segment files来去组成的，然后当然了你来了一个新的消息它会找到一个最新的那个segment然后尾部进行插入，然后在与传统的这个消息不太一样，kafka系统它不会有明确的一个消息id这么一个概念，就是说kafka内部它这里面是没有一个明确的消息id，它为了让你这个数据最后能够得到公开是完全是依赖于你这里面有一个offset这个偏移量，偏移量就是一个partition里面的一个位置。

好了然后我们继续往下，这是一个以上说的是持久化的一个做法，然后另外一个就是一个传输效率，就是在传输效率这一块kafka也会做了一些优化（如下图10所示）

图10：



就是说它在用api去遍历消息的时候，就是说它在去遍历这个segment files里面的消息的时候并不是每一次请求一个消息然后来去遍历的，它背后是通过批量的获取一批数据，然后这个时候通过多次的，然后减少了数据的一个请求的数量，那这样的话可以减少网络请求的数量是这么一个情况，然后另外一个这kafka它采用了一个无缓存的一个设计，那缓存我们通常理解为缓存都是一个内存是吧？这内存通常是一个应用层的一个内存，然后这里面有一个图，那我们通过这个图来看一下通常我们在去利用比如说底层的一个设备的时候然后怎么去应用一个过程（如下图11所示）

图11：  


但是这是传统的一个应用方式，假设说这是有一个数据，这个数据是来自于外部是吧？当这个数据我们开始要对它进行读的时候，那其实它首先这个数据是通过底层的一个系统级别的设备然后进行对接过来，那首先第一次1代表是把数据先拷贝到系统级别的一个Readbuffer里面去，然后为了你的应用层程序可以能够读到这样的一个集体数据，所以需要从你的内核(kemelcontext)，就是系统缓存里面然后读到你的应用层(Applicationcontext)的缓存里面，所以这里面也会经历了数据的复制，好了，那应用层想把这个数据再写回到进行输出，比如说你输出到外面的磁盘上或者是通过往IO的形式进行向外传输，那你必须也得通过一个应用层（Application buffer）把这个数据写回到你的系统层（Socket buffer），然后系统层再对接到外部的设备上(NIC buffer)，所以整个的传统方式你从数据源头从进到出中间要经历过四个复制，那其实在复制的过程中还是很耗时间和性能的，所以在这个kafka设计的时候它就能够，因为它这个kafka就是一个数据传递这么一个过程对吧？所以它是可以把这个数据直接在系统级别层然后进行直接读入和读出（如下图12所示）

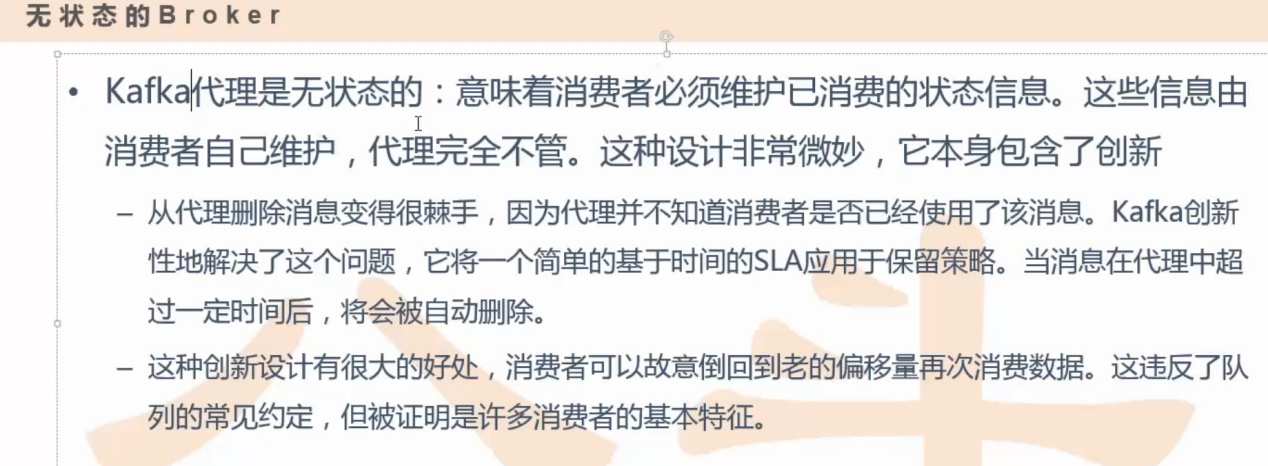
图12：



所以减少了向应用层的两次复制，所以这个时候就加快了你整个的一个传输的效率，那这个上图12的特点就是一个zero-copy，就是0拷贝，目的就是减少你的字节拷贝，然后通过系统应用的方式把这个数据源从源头的设备(Zero-copy)然后写到一个输出的设备(NIC buffer)上是这么一个过程。

这样的话原来是有四次一个拷贝，然后变成现在的一个两次拷贝，之前我们说过kafka有一个预读功能还有一个是后写功能，那预读相当于就是它利用你操作系统级别的一个功能，然后能够加快了你的这个数据的读取，因为你是对这个文件如果是顺序读取的话，它会把你没有读到的数据提前放到你的缓存里面去，那这个缓存其实就是在这一块上图12中的Read buffer里面，然后这个无状态刚才也说过了，就是说你这个消费者你必须自己维护你自己的一个读到的数据这么一个偏移（如下图13所示）

图13：

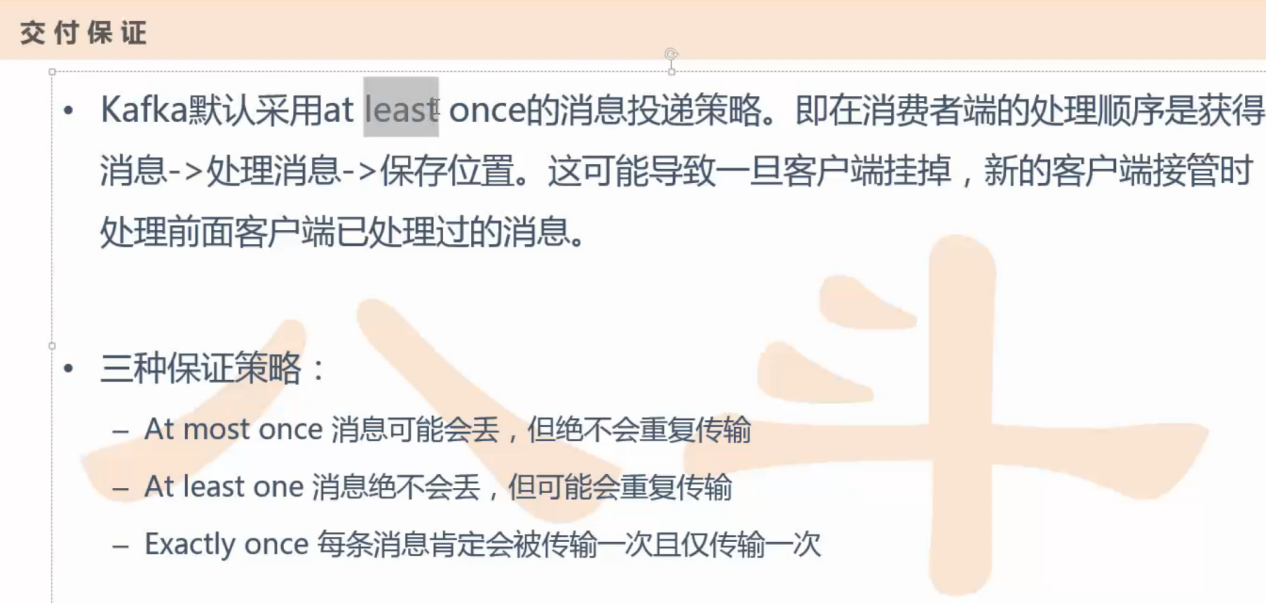


然后对于kafka来说它是不管的，那为了这样的设计，它会让你整个的设计会变得简单，所以kafka也就变得稳定起来，那这块 可能大家会有疑问，那如果是这么做的话，那我这个数据什么时候就需要缓存多长时间是吧？所以这个时候kafka会给你做一定的妥协，就是说我这个数据默认情况下是在我的相当于我这个kafka如果是作为一个存储的话，我这个数据是在我的存储里面要存7天，你在这个7天内你想读的时候，你就可以过来读，但一旦这个时间过了之后，我数据不可能给你保留太长的时间，所以这叫一个time-based消息保留策略，那这里面提到的这个SLA就是类似这么一个策略的一个表达。所以这样设计的好处就是说，消费者是可以在这个时间范围内它是可以进行回赎一些老的偏移量，然后读它过去某一个时刻的一个数据，但是通常消息队列是让你顺序的读取，如果你要是想读其中的某一个时刻的数据其实这是不太符合你消息队列这么一个特点是吧？

所以它就相当于是违反了这个消息队列这么一个常见的约定，但是因为真正为了消息队列是赋予整个线上的系统，这个线上系统你不可能保证你前端这个检索引擎是永远不会出问题的对不对？但你这个检索引擎一旦出现故障的时候，那必须要把这个引擎重新重启一遍对吧？那如果要是之前你从kafka消息队列里面读的数据都往内存里面去维护的话，那你前端这个检索引擎重启的时候，你内存是被清空的一个状态，所以内存是没有这个数据，那这个时候就可以重新对接kafka的某历史的某一个时间段然后把数据再进行一定的恢复，这么一个过程，就是超过了这个SLA设定的时间，那这个消息就会被删除。

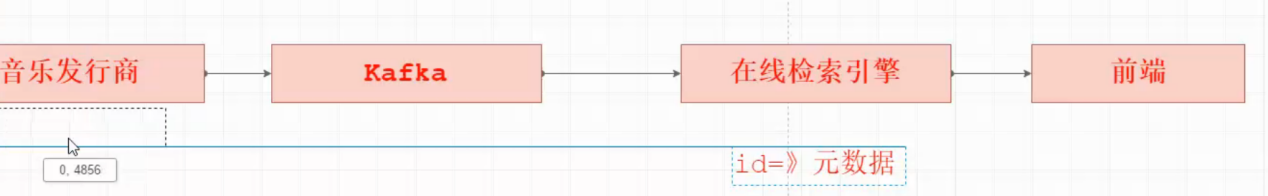
还有一个就是说kafka默认采用的是at least once这么一个消息投递策略（如下图14所示）

图14：



那这个消息投递策略意思就是说你的消息至少发送一次，如果消息未能接收成功，就会出现重发的可能，但是这种行为虽然它消息可以重发了，但是这可以保证你的消息不丢失，就是说比如你是做一个服务的话，那开始呢kafka给了你一个消息，那这个时候这个kafka突然某一个Broker出现了一些问题是吧？然后这个时候它就认为它没有得到你的应答，就是说当消费者进行获得消息的这么一个过程，然后获得消息之后你肯定要对这个消息进行处理，然后再保存你当前这个消息所对应的偏移量也就是offser，那如果是这个消息能够成功被处理的话，那这个时候你需要再去保存offset，这保存offset应该保存哪里呢？因为你这个Consumer可以和zookeeper进行连接的，所以zookeeper是可以帮你保存这个offset，那如果说你消息者在获取数据的过程中，已经获取到了这样的消息，但是你这个偏移量还没有及时的提交给zookeeper，就是说你先拿数据，然后再去标记一下你这个数据的偏移量，就是说你拿的数据你已经拿到了，但是你这个偏移量没有汇报给zookeeper，那这个时候相当于zookeeper会通知这个kafka，或者内部有一个机制，kafka会把这个数据重新给你发送，直到你这个offset重新写到你的zookepper里面去，所以通过一个at least这么一个方式，它可以保证你消息不丢失，但是你消息有可能会重发，但是通常这样的情况，消息重发如果是前一个消息和后面某一个消息是重复的话，这个通常在比如线上引擎存储的时候也没有太大的问题，这里面也可以举个例子，就是说我有一个kafka和一个在线检索引擎和一个音乐发行商和前端（如下图15所示）

图15：



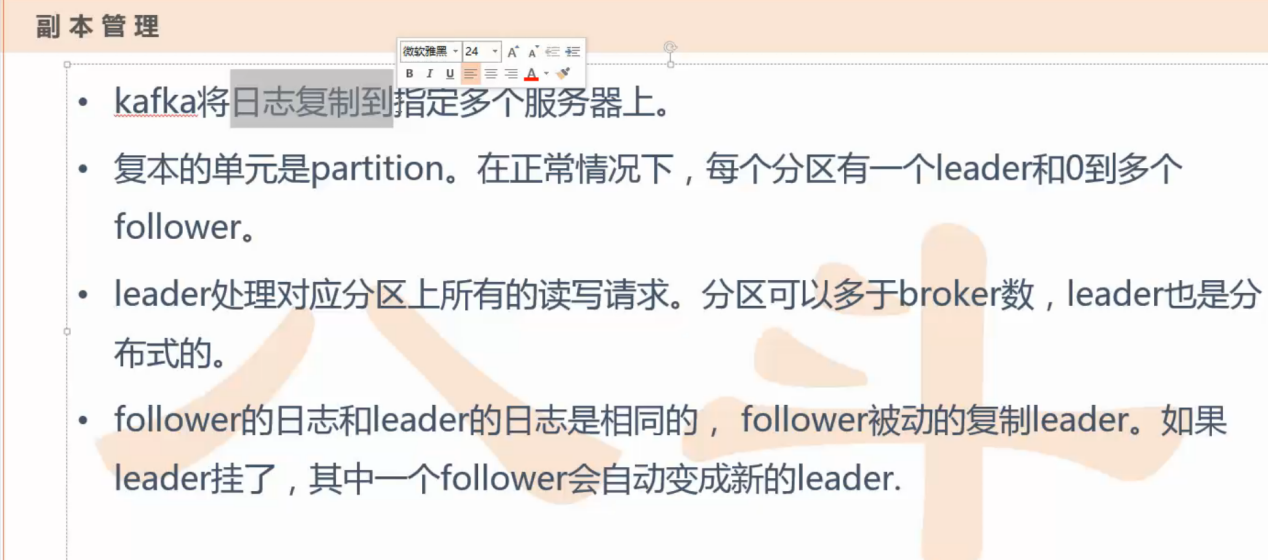
那你前端就是直接对应着你的用户是不是？那你这个用户在去打开页面的时候，比如说想听一首歌，那么这个时候这首歌在页面肯定会有一个ID来表示是吧？比如这首歌有一些描述信息围绕这个音乐要产生一些个信息包括什么歌手的一些信息，这个数据是不可能在前端来保存的，这些数据都是由后端的在线检索引擎这一块给前端提供的是不是？比如说要我听一首歌，那么前端会把这首歌的ID请求给在线引擎，在线引擎会拿这个id在它内存或者是某一个数据库上以id为key去检索这个id背后的比如这个音乐的名字，歌手的名字等等这些元数据然后再返回给你前端做前端页面展示，好了那这个时候比如说你这在线检索引擎里面不管是内存也好还是你的数据库也好我这里面是有一个key和value的一个形式，那这些信息是哪里来的呢？这个信息其实是从音乐发行商开始不断的去传过来的，那这个时候其实很简单了，当这个音乐发行商务刚刚发布了一个新的音乐，这新音乐当然传过来的时候有很多的环节就省略了，当然你这个对新的这个音乐肯定要请求一个它的一个id对吧？然后这个id或者这个音乐以及包含这个所有的元数据信息全部通过这个消息队列kafka传输到了你在线引擎里面，你在线引擎把这个数据往内存去存，就存到了一个id到元数据的这么一个过程，好了那这个时候并不是说你新的音乐就走这么一个流程，老的音乐也可以走这个流程。

比如说某一个老音乐可能前段时间是可以听的，但是最近一段时间由于版权的一些的什么到期之内的，那么某一些音乐可能不能让你听了，那这个时候这个信息是怎么能够让你的前端不能看呢？肯定是在你在线检索引擎元数据里面有某一个字段来标记着你这个音乐下线了，不能够再对提供服务了对不对？那这个时候音乐发行商或者是某一些编辑手动在后台给你设置好说当前这个音乐我要打一个标记，这个标记是不能够被外界所收听的，那这个时候它相当于这个ID本来就在你的检索引擎里面有，那这个时候我这个消息再重新发布一次，这个时候这个原来的元数据就被最新的元数据给覆盖掉，更新掉，所以这整个过程其实不仅仅是一个新的数据才走这流程，你这个旧的数据一旦有修改的需求也要走这样的一个流程，然后刚才说过一个问题就是说大家认为你这个重复发送，我为了我的消息不丢失，然后我又可能有重发的可能，那这个重发对于这种场景来说是完成没有问题的

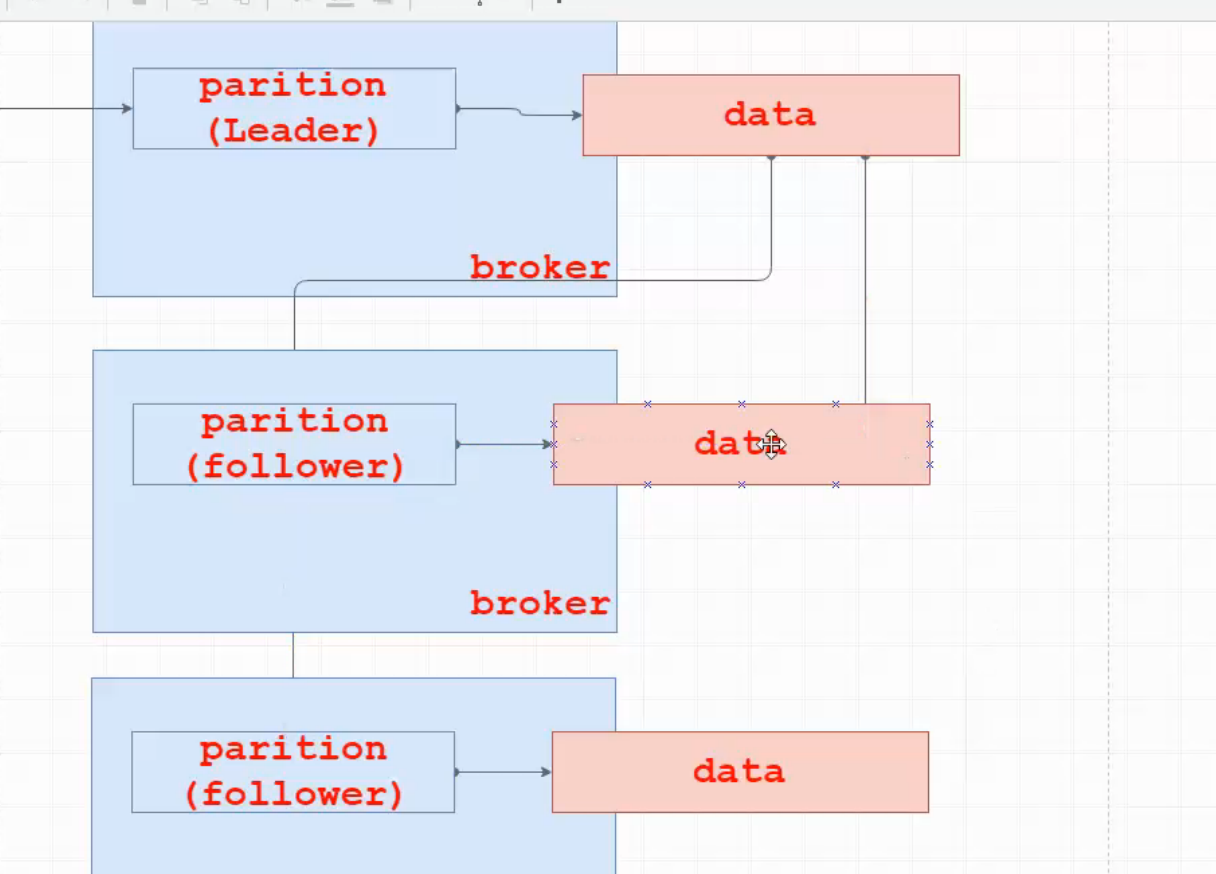
当你前端或者再kafka这一个阶段，消息又重新发布了一次，那即使你元数据中间没有任何数据的变化那你替换一下其实也没有任何问题，所以对于这种场景来说你重发数据对我来说整个系统没有影响的，但是整个系统如果你一旦不发数据，这是有问题的，这样会有丢数据的风险。

当然了除了这种at least once这种消息，它还有其他的消息，但是那几个消息不是一个默认的情况然后也通常不用。

我们看一下副本管理，副本管理这个比较重要，它是对日志来说的，在kafka里面的日志就是数据，所以这个日志副本这么一个策略它是整个kafka可靠的一个核心所在，所以我们看一下这个可靠是怎么得到保证的，那在说这个可靠之前，我们先来回忆一下HDFS的可靠性是通过什么样子的？数据的可靠性，有什么样的特点你的数据不丢失？备份是吧？副本对不对？默认情况下你这个副本是有三份的，当有一台机器挂掉了，那你这个机器上的数据丢了，没关系，我其他机器上也有备份，也可以照样提供服务，所以在kafka里面也有这样的类似的机制。（如下图16所示）

图16：  


但是这个机制相当于跟你hdfs副本很相似，那我们看一下这个是怎么做到的(如下图17所示)

图17：  


我要写数据的时候，我要往某一个partition上写数据的是吧？那万一这个机器挂掉的话，那么这个partition上面的数据也就丢失了对吧？那我为了保证这个数据的可靠性，那我也希望在去写这个partition这个机器上写数据时候的同时，我想把这个数据复制多份，我想把这个数据存到其他broker上，就是这么一个思路，而且这个副本数目也是可以设置的，当然了你这个副本数不可以超过你的broker的数目，比如说你这个broker一共就有三台机器，那你这个副本数设置了四个，那么这是不行的。

好了这里面还有一个细节就是说上图17中三个partition的信息定位是不一样的，我这个第一个partition是一个主架构，其他的都是从，那我写数据的时候只能往Leader上去写，然后一个这个partition会有多个备份，但是这个副本里面只有一个是Leader，不允许有多个Leader，然后这个Leader它是负责了你的数据的读和写，就是说你外部写的时候，Producers去写的时候也要写到Leader上，然后你这个Consumer这个消费者去读的时候也要到Leader上去读，这个是跟你HDFS的设计是不一样的对吧？然后当这个数据再这个Leader上写完之后，那这个数据会同步到每一个partition(follower)上，然后相当于数据通过这样的方式再每一个其他的broker上存了一份是不是？

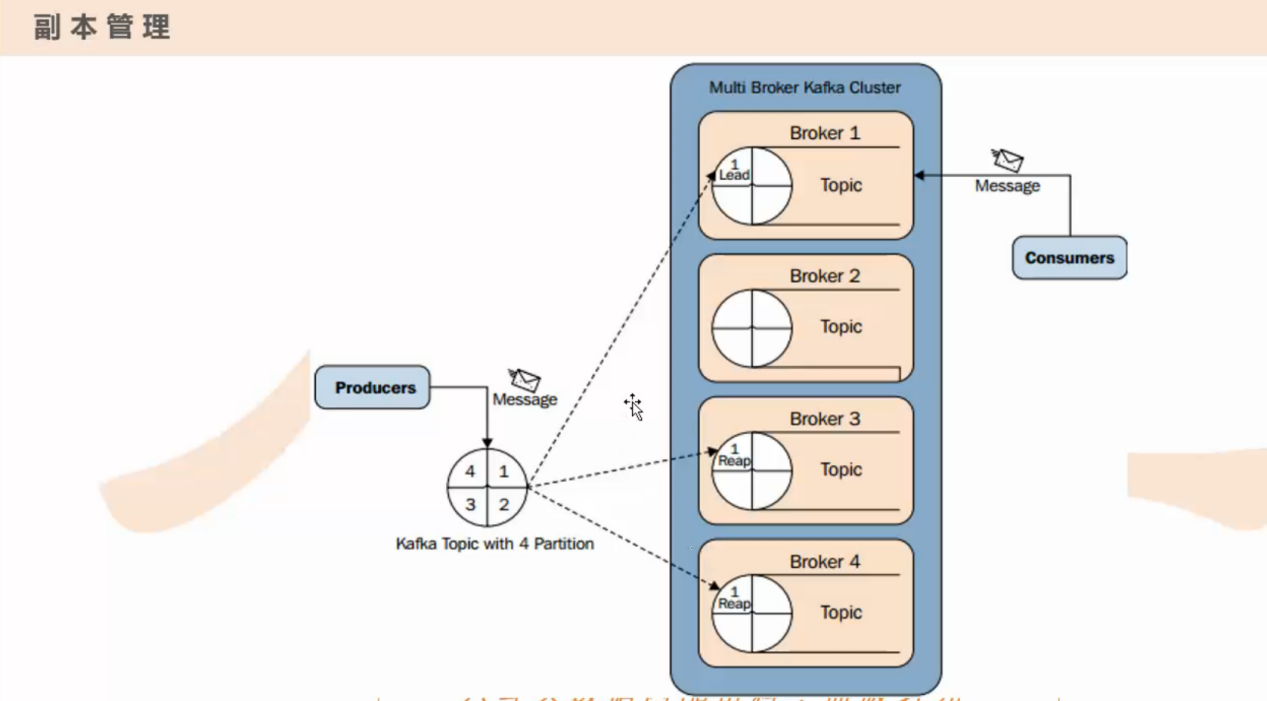
好了这个时候突然Leader机器挂了，那么这个时候就需要在剩下这个follower里面找一个Leader出来，如果你不找一个Leader出来的话，那你外界再有新的数据往里写，再有新的消费者去读数据的时候，那你是无法满足的对吧？

那如果这个时候我再挂掉一台，那最后一个那肯定就是自己是一个Leader了是吧？所以再这个kafka里面它有这么一个特点，就是说针对你partition的副本管理来说，如果一共有f+1个broker的话，允许挂掉f个，只有你有一个broker你就可以存数据，那么这个具体选主是由你的zookeeper来保证的，因为你这个brokerserver是直接可以连到zookeeper上的。

就是说kafka可以将你的这个日志数据可以复制到多个服务器上，也就是多个broker上，但是不是每一个机器上的或者每一个服务的broker都会提供服务，然后副本的单元就是partition，正常情况下，每一个分区就只有一个Leader主或者0或多个follower从。

然后这个Leader的话它是处理所有的读和写，从是不允许和外界进行关联的，你这个Leader和follower的日志是相同的，那么这个时候如果某一个broker上的这个partition这个数据如果是滞后于你这个Leader这个数据比较多的话，那我就认为你这个副本就挂掉了，然后那你这个滞后是由两个纬度来保证的，一个是时间还有一个是数据本身，因为你这个follower从是被动的复制Leader上的数据，所以你这个follower数据肯定是要滞后于Leader的，但是具体滞后多少，那这个可以通过一个配置来配置，比如超过了一个预值，那我认为你滞后的实在是太多了，那我就直接清除掉你，然后另外一个就是如果你的Leader挂了，其中会找一个follower会自动变成你的Leader（如下图18所示）

图18：



左边是你的Producers进行数据的生成，然后因为你有四个partition，所以你需要把这个数据会发到这四个partition里面，这个虚线不是说你这个Producers是往这四个partition上去写的，而是你这个复制是从你的Leader上复制的。

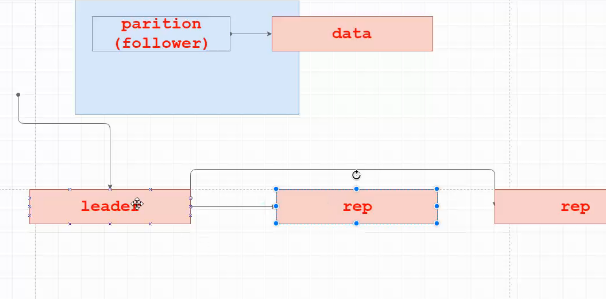
然后你的Consumers是来自于你的Leader从你的去写这么一个过程，那在这里面有一个很重要的概念，就是一个ISR，这个ISR就是说kafka在zookeeper里面动态维护了一个集合，这个集合叫做一个ISR集合，ISR意思就是说同步副本的意思，也就是说再这个集合里面的所有副本，都是跟上了Leader的节奏，就相当于是紧跟着你Leader的步伐去同步数据，也就是说只有你这个ISR里面的这个副本才有当Leader的机会。

然后什么情况下你这个ISR里面的集合副本是可以被去掉呢？ISR需要删除的两个情况，第一个情况就是说消息延迟的问题，第二个就是数据落后太多，消息延迟就是说在一段时间你都没有更新到一个新的数据

落后太多就好理解是吧？

然后这里面有一个细节（如下图19所示）

图19：

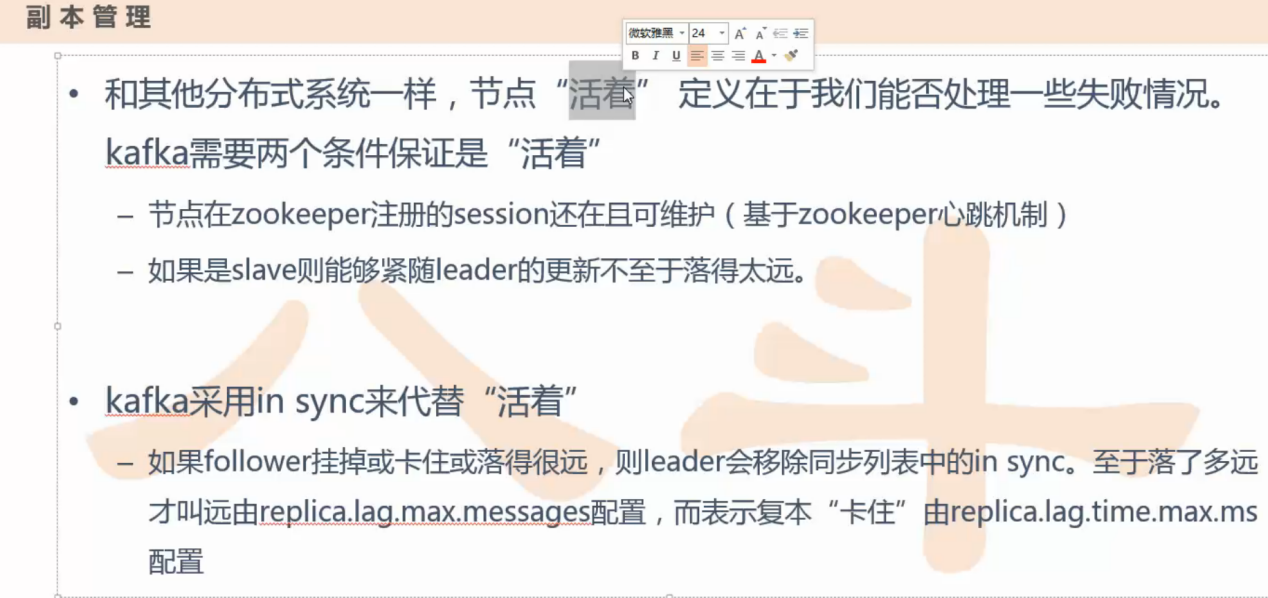


这个时候开始往Leader里面去写，写的时候我什么时候作为Producers的时候我才认为我的数据写成功了呢？就是当你的Leader写完自己的数据之后，那需要让你的其他的从也要同步数据，然后因为你这个数据真正是在你broker里面是要以磁盘的形式去存的对把？，但是你想一下如果是所有的节点你把数据都存在磁盘上，然后在保持一致性返回给你客户端的话，那这个时候效率是会很慢的对不对？然后如果要是说这个Leader你接受完消息之后，那直接返回，通过异步的方式这样就很快，当然数据在内部来说它是彼此之间能够，就是你的这个副本，那我在这里面同步进行复制可能对这个实效性要求就没那么高了对吧？但是虽然你的这个响应很快，然后让你的数据能够交互的更快就是性能会更好，但是这里面可能会有一定的风险，如果是你的Leader真要是挂了，你这个副本真要是没存这个数据，那这个时候就傻眼了是吧？那么这个数据就丢失了。

那么在kafka里面是怎么样的一个进行挨个反馈的呢？也就是说当你的Leader接收完数据之后，它自己落完盘之后，好了那副本就马上从你的数据把这个数据存到每一个副本的内存里面，但是这个时候内存不一定往你相应的副本机器上往磁盘上去写，那只要是这个数据一旦写到了他们各自内存里面的时候，那这些副本都可以向Leader返回一个成功，那至于内存往磁盘上去写，这块是可以让从去慢慢的同步的，所以这块的时间是可以省掉的，然后这个时候如果是所有副本以及这个Leader都准备就绪之后，那它才会返回一个真正的一个提交成功这么一个操作，那么这个在kafka内部，既保证你的一致性又保证了数据的一个可靠性，是通过这样的方式去管理的。

然后说一个活着的定义（如下图20所示）

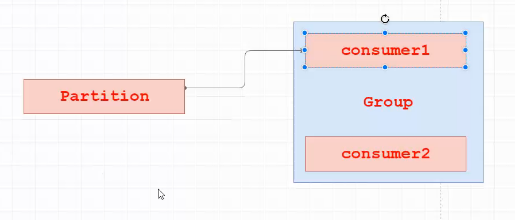
图20：



活着的定义就是说和大部分的分布式系统一样，就是kafka处理失败要明确你这个机器是否真正的活着是吧？这个机器就是broker，那在kafka内部它会有一个配置，这个配置叫做replica.lag.max.messages，这个配置如果设为4的话就表明你的从follower不能落后与主Leader超过3个消息，否则的话从ISR中删除，那一旦从ISR中删除的话，就是说你以后也没有当Leader主的机会了，那我之后也不会给你去同步数据了，那我就判定你这个broker就是挂掉了的状态

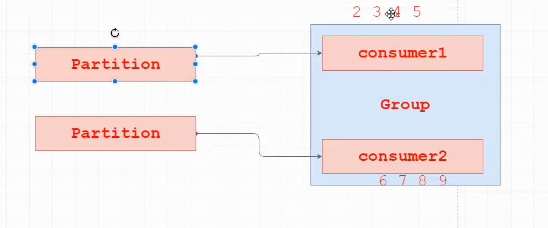
这里面提交就是说为了保证你的这个数据不丢失，就是说刚才我们讲到的这个，就是你的副本就是从你的leader同步完数据之后，那所有的节点都有相应的数据，不管你数据是在磁盘也好还是在内存也好，只要是每个节点都有了，那这个时候就认为你这次数据是可以成功的状态，比如说我这里有一个图（如下图21所示）

图21：



那这个时候Consumer1是往你这个partition里面去读数据，那这个时候比如说你只有一个partition，那你其他的Consumer是不可以去连接的，我只能一个partition同一时间同一时刻我只能服务于Consumer，如果说我这个partition里面的数据是1 2 3 4 5 6 7 8 9，就是说我每一个数字代表一个消息是吧？那如果说我这个group如果是来对这个partition做消费的时候，因为我这个Consumer2是不能够来消费的，所以我只有Consumer1来消费，那这个时候因为我这个消费者数目要多于你的partition的数目，所以这个Consumer2是只能属于空闲的状态，那这个时候我这个Consumer1需要在partition里面的数据都读过滤，那如果你这个partition如果是能够有多个的话，那这个时候相当于把这个消息是拆分了，上面的一半消息给Consumer1读了，然后下面的一半消息给Consumer2读了（如下图22所示）这么一个过程

图22：



所以就相当于把这个消息同步的这个性能能够大大的加快，你一个group里面不是有多个

Consumer吗？但是你Consumer消费的数据是不能够有重复的，但是你所有的Consumer消费的数据全局合并的话，其实是一个完整的一个数据，只不过是每一个Consumer消费了一部分，所以这里面就是涉及到了一个分布式协调的一个过程，就是说kafka中一个topic中不同分区就是不同partition，只能被消费组中的一个消费者来消费，这样的话如果要是有多个消费者的话，多消费者就导致了额外的开销。然后kafka中消费的进程只需要在代理和同组消费者有变化时进行一次协调。

这个是zookeeper在它内部它是可以维护着一个相应的目录数，然后在不同的目录里面他会维护相应的数据以及结构对吧？那在这里面比如说，在这个broker下有id，它就创建一个节点（znode），那你可以把这个broker id写进去（如下图23）

图23：  


那这个id就是认为是你当前这个集群中到底有多少个机器在，就是说你这个kafka集群里到底有多少个集群在运转，然后下面是一个维护topic的信息，并且partition的一些注册的一些信息在这下面。

这是关于Consumer groups它也会有相应的绝对路径一个节点来去维护，还有一个Consumer id，就是你每一个消费者它也有一个id也会维护起来（如下图24所示）

图24：



然后还有一个offser它也会维护了你的Consumer 的一个数据的一个偏移量读取的一个数据偏移它也会帮你维护，然后还有你这个partition相应的一些注册（如下图25所示）

图25：  
