Kafka 基础学习笔记

1. 相关概念

1.1 Broker

使用Kafka前，我们都会启动Kafka服务进程，这里的Kafka服务进程我们一般会称之为Kafka Broker或Kafka Server。因为Kafka是分布式消息系统，所以在实际的生产环境中，是需要多个服务进程形成集群提供消息服务的。所以每一个服务节点都是一个broker，而且在Kafka集群中，为了区分不同的服务节点，每一个broker都应该有一个不重复的全局ID，称之为broker.id，这个ID可以在kafka软件的配置文件server.properties中进行配置。

1.2 Controller

* Kafka是分布式消息传输系统，所以存在多个Broker服务节点，但是它的软件架构采用的是分布式系统中比较常见的主从（Master – Slave）架构，也就是说需要从多个Broker中找到一个用于管理整个Kafka集群的Master节点，这个节点，我们就称之为Controller。
* 如果在运行过程中，Controller节点出现了故障，那么Kafka会依托于ZooKeeper软件选举其他的节点作为新的Controller，让Kafka集群实现高可用。

1.2.1 Controller节点的选举的过程

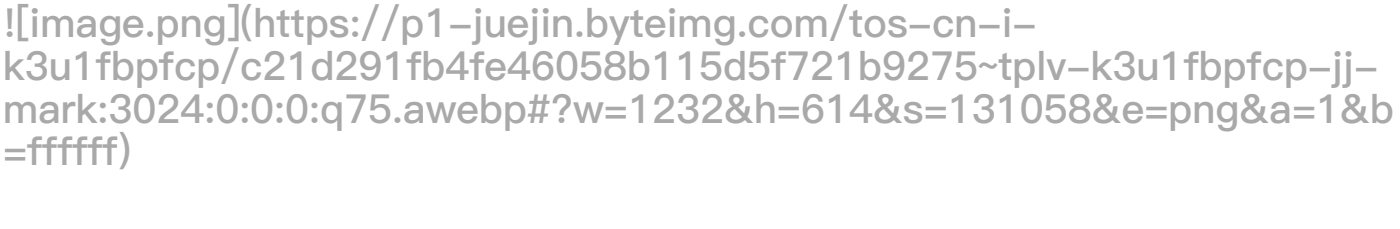
1. 第一次启动Kafka集群时，会同时启动多个Broker节点，每一个Broker节点就会连接ZooKeeper，并尝试创建一个临时节点 **/controller**
2. 因为ZooKeeper中一个节点不允许重复创建，所以多个Broker节点，最终只能有一个Broker节点可以创建成功，那么这个创建成功的Broker节点就会自动作为Kafka集群控制器节点，用于管理整个Kafka集群。
3. 没有选举成功的其他Slave节点会创建Node监听器，用于监听**/controller**节点的状态变化。
4. 一旦Controller节点出现故障或挂掉了，那么对应的ZooKeeper客户端连接就会中断。ZooKeeper中的 **/controller** 节点就会自动被删除，而其他的那些Slave节点因为增加了监听器，所以当监听到 **/controller** 节点被删除

后，就会马上向ZooKeeper发出创建 **/controller** 节点的请求，一旦创建成功，那么该Broker就变成了新的Controller节点了。

1.3 **Topic**

* Kafka采用的数据传输方式为发布-订阅模式，也就是说由消息的生产者发布消息，消费者订阅消息后获取数据。为了对消费者`订阅的消息进行区分`，所以对消息在`逻辑上进行了分类`，这个分类我们称之为主题：`Topic`。消息的生产者必须将消息数据发送到某一个主题，而消费者必须从某一个主题中获取消息，并且消费者可以同时消费一个或多个主题的数据。Kafka集群中可以存放多个主题的消息数据。
* 为了防止主题的名称和监控指标的名称产生冲突，官方推荐主题的名称中不要同时包含`下划线`和`点`。

1.4 **分区：Partition**

* 缺点：消息生产者将数据发送到一个主题时，假如发送给这个主题的`数据非常多`，那么主题所在broker节点的`负载和吞吐量`就会受到极大的考验，甚至有可能因为`热点问题`引起broker节点故障，导致服务不可用。
* 解决：一个主题从物理上分成几块，然后`将不同的数据块均匀地分配到不同的broker节点上`，这样就可以缓解单节点的负载问题。这就是分区。
* topic主题的每个分区都会用一个编号进行标记，一般是`从0开始`的连续整数数字。Partition分区是物理上的概念，也就意味着会以数据文件的方式真实存在。每个topic包含一个或多个partition，`每个partition都是一个有序的队列(消息队列)`。partition中每条消息都会分配一个有序的ID，称之为偏移量：Offset。


1.5 **副本：Replication**

* 如果一个topic划分了多个分区partition，那么这些分区就会均匀地分布在不同的broker节点上，一旦某一个broker节点出现了问题，那么在这个节点上的分区就会出现问题，那么Topic的数据就不完整了。所以一般情况下，为了防止出现数据丢失的情况，我们会给分区数据设定多个备份，这里的备份，我们称之为：副本`Replication`。
* Kafka支持多副本，使得主题topic可以做到更多容错性，牺牲性能与空间去

换取更高的可靠性。

![image.png](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/5a187e67e2ad488b89d6ad8ee1ce272a~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1144&h=704&s=150119&e=png&a=1&b=fefdfd)
注意： 这里不能将多个备份放置在同一个broker中，因为一旦出现故障，多个副本就都不能用了，那么副本的意义就没有了。

1.5.1 **副本类型：Leader & Follower**

* 在Kafka中，`所有的文件都称之为副本`，只不过会选择其中的一个文件作为主文件，称之为：`Leader(主导)副本`，其他的文件作为备份文件，称之为：`Follower（追随）副本`。在Kafka中，这里的文件就是`分区`，每一个分区都可以存在1个或多个副本，`只有Leader副本才能进行数据的读写`，Follower副本`只做备份`使用。
*

![image.png](https://p1-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/de698df941954606bd21ddbbcca663b1~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1424&h=422&s=158907&e=png&a=1&b=015ce5)

2. 发送消息
=======

2.1 如何保证数据不丢失?
----

* ACK应答

2.1.1 **ACK = 0**

* 当生产数据时，生产者对象将数据通过网络客户端将数据发送到网络数据流中的时候，Kafka就对当前的数据请求进行了响应（确认应答），如果是同步发送数据，此时就可以发送下一条数据了。如果是异步发送数据，回调方法就会被触发。
* ![image.png](https://p6-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/ed43c3ef431c4f69afdbb92cc372789e~tplv-k3u1fbpfcp-jj-

mark:3024:0:0:0:q75.awebp#?w=2415&h=1440&s=10451045&e=png&b=2a81e4)

2.1.2 **ACK = 1**

* 当生产数据时，Kafka Leader副本将数据接收到并写入到了日志文件后，就会对当前的数据请求进行响应（确认应答），如果是同步发送数据，此时就可以发送下一条数据了。如果是异步发送数据，回调方法就会被触发。
* ![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/e737604576f046ceb4a5c3d9f5c9c02f~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=2416&h=1440&s=10455377&e=png&b=2a81e4)

2.1.3 **ACK = -1(默认)**

* 当生产数据时，Kafka Leader副本和Follower副本都已经将数据接收到并写入到了日志文件后，再对当前的数据请求进行响应（确认应答），如果是同步发送数据，此时就可以发送下一条数据了。如果是异步发送数据，回调方法就会被触发。
![image.png](https://p9-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/71abd20536114f56b205136296798a49~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=2432&h=1440&s=10524593&e=png&b=2981e5)
* `注意`：如果假设我们的分区有5个follower副本，编号为1，2，3，4，5。但是我们的In Syn Replica，简称为ISR设置的是（1，2，3，4），就不需要通知5号副本，通知1-4号就可以返回ack了。

2.2 如何解决 消息去重 & 有序 问题

2.2.1 消息重复

* Kafka并没有成功将ACK应答信息发送给Producer（网络超时），Producer会尝试对超时的请求数据进行重试(**retry**)操作。通过重试操作尝试将数据再次发送给Kafka。数据就会重复

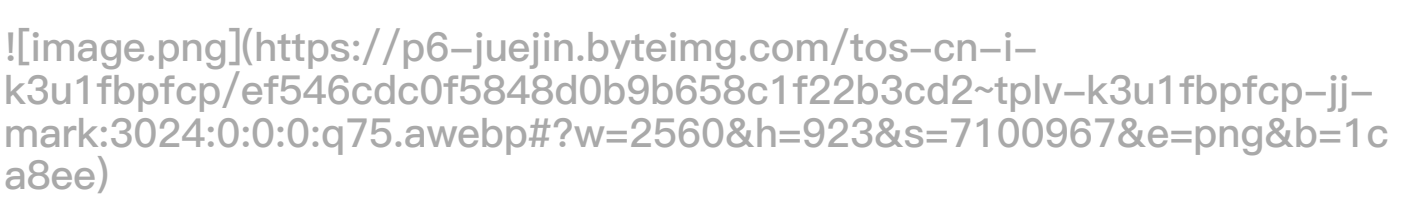
2.2.2 数据乱序

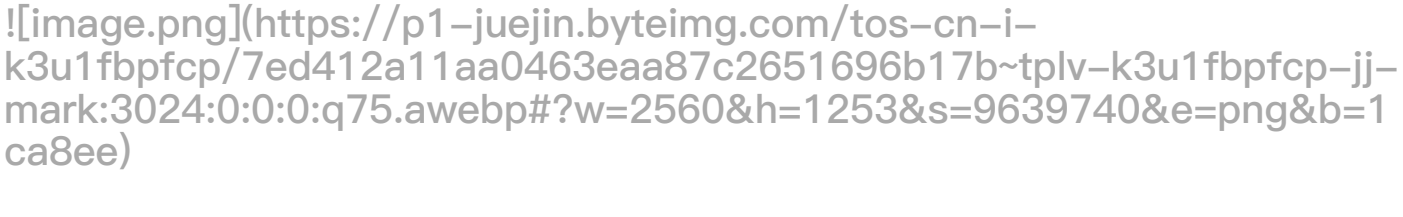
* 我们需要将编号为1，2，3的三条连续数据发送给Kafka。每条数据会对应于

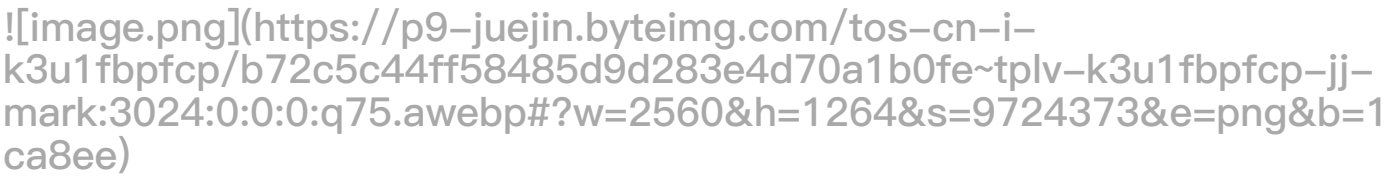
一个连接请求。第1条retry了，其余正常发送到Kafka。顺序则变为了：231

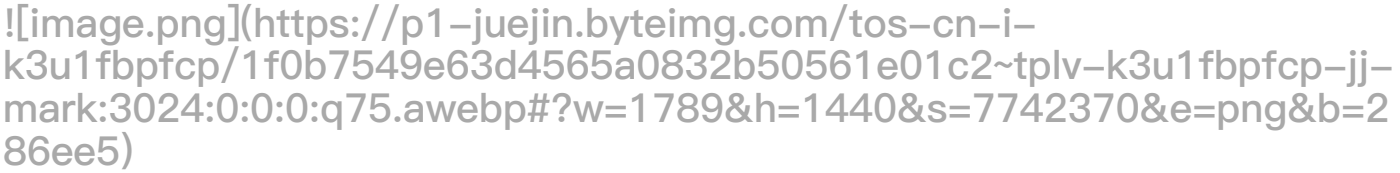
2.2.3 **数据幂等性**

* 默认幂等性是`未开启`的，所以如果想要使用幂等性操作，只需要在生产者对象的配置中开启幂等性配置即可
* `开启幂等的要求：1.ACK=-1 2.在途请求缓冲区=5 3.开启重试机制`
* `开启幂等 只能对一个分区起作用，多个分区不能保证`
* 开启幂等性后，为了保证数据不会重复，那么就需要`给每一个请求批次的数据增加唯一性标识`，kafka中，这个标识采用的是连续的序列号数字 sequencenum
* 但是`不同的生产者Producer可能序列号是一样的`，所以仅仅靠seqnum还无法唯一标记数据，所以还需要同时对生产者进行区分
* 所以Kafka采用申请生产者ID（producerid）的方式对生产者进行区分。
* 这样，在发送数据前，我们就需要提前申请producerid以及序列号 sequencenum
* `Broker`中会给每一个分区记录生产者的`生产状态`(采用队列的方式缓存最近的5个批次数据。)
* 队列中的数据按照seqnum进行升序排列。这里的数字5是经过压力测试，均衡空间效率和时间效率所得到的值，所以为固定值，无法配置且不能修改。
* 如果新的请求批次数据，`存在Borker当前在缓存的5个旧的批次中`，那么说明有重复，当前批次数据不做任何处理。
* 如果Broker当前的请求批次数据在缓存中`没有相同的`，那么判断当前`新的请求批次的序列号`是否为缓存的最后一个批次的序列号`加1`，如果是，说明是连续的，顺序没乱。那么继续，如果不是，那么说明数据已经乱了，发生异常。
* Broker根据异常返回响应，通知Producer进行重试。Producer重试前，需要在缓冲区中将数据重新排序，保证正确的顺序后。再进行重试即可。
* 如果请求批次不重复，且有序，那么更新缓冲区中的批次数据。`将当前的批次放置在队列的结尾`，`将队列的第一个移除`，保证队列中缓冲的数据`最多5个`。
*







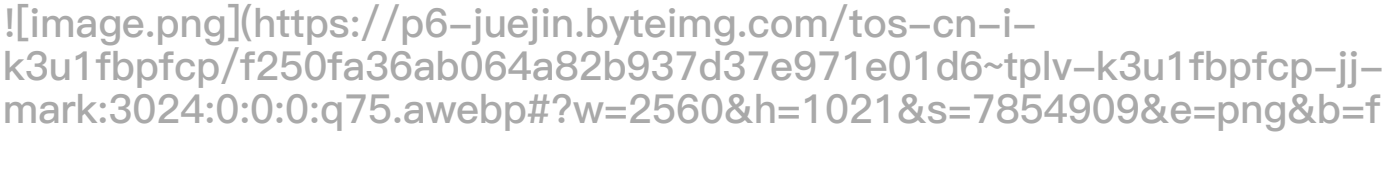


从上面的流程可以看出，Kafka的幂等性是通过消耗时间和性能的方式提升了数据传输的有序和去重，在一些对数据敏感的业务中是十分重要的。但是通过原理，咱们也能明白，这种幂等性还是有缺陷的：

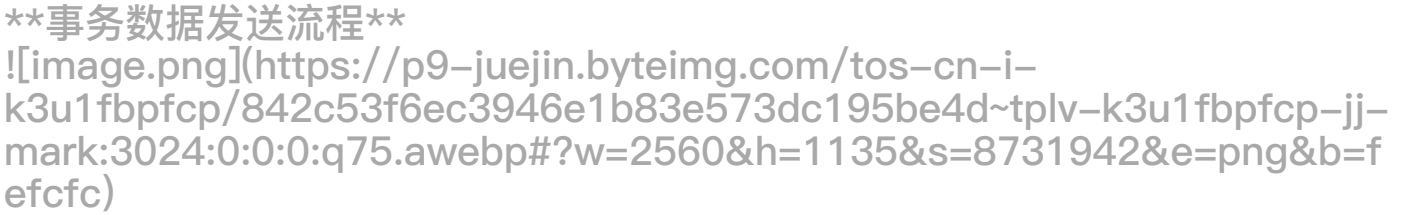
* 幂等性的producer仅做到单分区上的幂等性，即单分区消息有序不重复，`多分区无法保证幂等性。`
* 只能保持生产者单个会话的幂等性，无法实现跨会话的幂等性，也就是说`如果一个producer挂掉再重启，那么重启前和重启后的producer对象会被当成两个独立的生产者`，从而获取两个不同的独立的生产者ID，导致broker端无法获取之前的状态信息，所以无法实现跨会话的幂等。要想解决这个问题，可以采用后续的事务功能。

2.2.4 数据事务

* 对于幂等性的缺陷，kafka可以采用事务的方式解决跨 会话 的幂等性。基本的原理就是`通过事务功能管理生产者ID`，保证事务开启后，生产者对象总能获取一致的生产者ID。
* 为了实现事务，Kafka引入了事务协调器（TransactionCoodinator）负责事务的处理，所有的事务逻辑包括分派PID等都是由TransactionCoodinator负责实施的。
* TransactionCoodinator 会将事务状态持久化到该主题中。
* 事务基本的实现思路就是通过配置的事务ID，将生产者ID进行绑定，然后存储在Kafka专门管理事务的内部主题 `__transaction_state`中
* 这个协调器对象有点类似于咱们数据发送时的那个副本Leader。其实这种设计是很巧妙的，因为kafka将事务ID和生产者ID看成了消息数据，然后将数据发送到一个内部主题中。这样，使用事务处理的流程和咱们自己发送数据的流程是很像的。接下来，我们就把这两个流程简单做一个对比。

普通数据发送流程


efdfd)

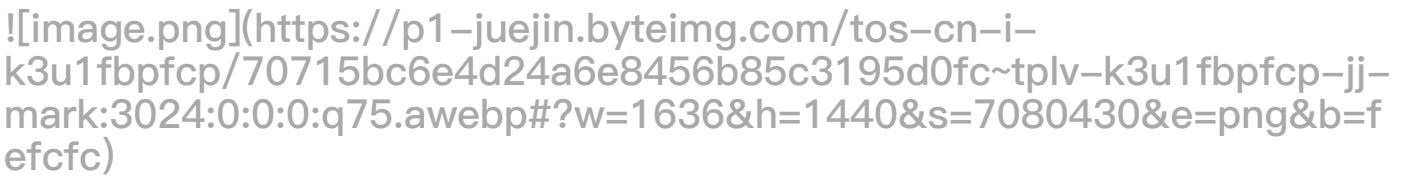
事务数据发送流程


* 通过两张图大家可以看到，基本的事务操作和数据操作是很像的，不过要注意，我们这里只是简单对比了数据发送的过程，其实它们的区别还在于数据发送后的提交过程。
* 普通的数据操作，只要数据写入了日志，那么对于消费者来讲。数据就可以读取到了.
* 但是事务操作中，如果数据写入了日志，但是没有提交的话，其实数据默认情况下也是不能被消费者看到的。只有提交后才能看见数据。

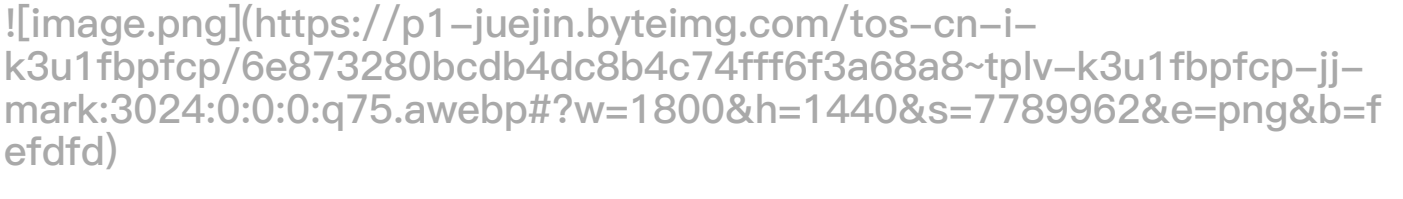
事务提交流程

Kafka中的事务是分布式事务，所以采用的也是`二阶段提交`.

1. 第一个阶段提交事务协调器会告诉生产者事务已经提交了，所以也称之预提交操作，事务协调器会修改事务为`预提交`状态



2. 第二个阶段提交事务协调器会向分区Leader节点中发送数据标记，通知Broker事务已经提交，然后事务协调器会修改事务为`完成提交`状态



3. 存储消息
=======

3.1 数据存储

3.1.1 **ACKS校验**

* **ACKS = 0**: Producer端将数据发送到网络输出流中，此时Kafka就会进行响应。
* **ACKS = 1**: Producer端将数据发送到Broker中，并保存到当前节点的数据日志文件中，Kafka就会进行确认收到数据的响应。（当前Broker节点出现问题而宕掉，数据还是会丢失）
* **ACKS = -1（all）** :在ACKS = 1的基础上，加上Follower也确认成功。
Ø 数据就更加可靠，但是相对，应答时间更长，导致Kafka吞吐量降低。

3.1.2 **ACKS应答及副本数量关系校验**

* Kafka为了数据可靠性更高一些，需要分区的所有副本都能够存储数据，但是分布式环境中难免会出现某个副本节点出现故障，暂时不能同步数据。在Kafka中，能够进行数据同步的所有副本，我们称之为In Sync Replicas，简称`ISR列表`。
* 当生产者Producer要求的数据ACKS应答为-1的时候，那么就必须保证能够同步数据的所有副本能够将数据保存成功后，再进行数据的确认应答。但是一种特殊情况就是，如果当前ISR列表中只有一个Broker存在，那么此时只要这一个Broker数据保存成功了，那么就产生确认应答了，数据依然是不可靠的，那么就失去了设置ACK=all的意义了，所以此时还需要对ISR列表中的副本数量进行约束，至少不能少于2个。这个数量是可以通过配置文件配置的。参数名为：min.insync.replicas。默认值为1（不推荐）
* 所以存储数据前，也需要对ACK应答和最小分区副本数量的关系进行校验。

3.1.3 ***日志文件滚动判断***

* 数据存储到文件中，如果数据文件太大，对于查询性能是会有很大影响的，所以副本数据文件并不是一个完整的大的数据文件，而是根据某些条件分成很多的小文件，每个小文件我们称之为`文件段`。
* 其中的一个条件就是文件大小，参数名为：log.segment.bytes。默认值为1G。
* 如果当前日志段剩余容量可能无法容纳新消息集合，因此有必要创建一个新的日志段来保存待写入的所有消息。此时日志文件就需要滚动生产新的.
* 除了文件大小外，还有时间间隔，如果文件段第一批数据有时间戳，那么当前批次数据的时间戳和第一批数据的时间戳间隔大于滚动阈值，那么日志文件也会滚动生产新的。
* 如果文件段第一批数据没有时间戳，那么就用当前时间戳和文件创建时间戳进

行比对，如果大于滚动阈值，那么日志文件也会滚动生产新的。

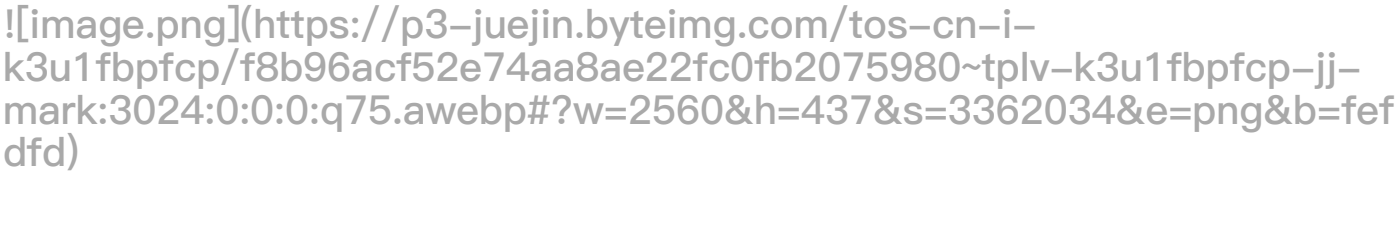
3.1.4 ***请求数据重复性/有序性校验***

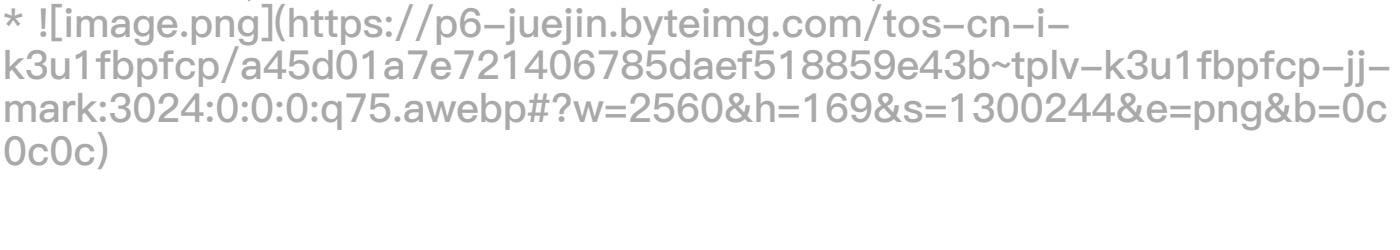
* 因为Kafka允许生产者进行数据重试操作，所以因为一些特殊的情况，就会导致数据请求被Kafka重复获取导致数据重复，所以为了数据的幂等性操作，需要在Broker端对数据进行重复性校验。这里的重复性校验只能对同一个主题分区的5个在途请求中数据进行校验，所以需要在生产者端进行相关配置。

3.1.5 ***数据存储***

* 将数据通过LogSegment中FileChannel对象。将数据写入日志文件，写入完成后，更新当前日志文件的数据偏移量。

3.2 存储文件格式



* 我们所说的数据文件就是以.log作为扩展名的日志文件。文件名长度为20位长度的数字字符串，数字含义为当前日志文件的第一批数据的基础偏移量，也就是文件中保存的第一条数据偏移量。字符串数字位数不够的，前面补0。
* 数据日志文件到达1G才会滚动生产新的文件。那么从1G文件中想要快速获取我们想要的数据，效率还是比较低的。
* 为了定位方便Kafka在提供日志文件保存数据的同时，还提供了用于数据定位的索引文件，索引文件中保存的就是逻辑偏移量和数据物理存储位置（偏移量）的对应关系。
* 
* 某些场景中，我们不想根据顺序（偏移量）获取Kafka的数据，而是想根据时间来获取的数据。这个时候，可没有对应的偏移量来定位数据，那么查找的效率就非常低了，因为kafka还提供了时间索引文件，咱们来看看它的内容是什么
* 

3.3 数据刷写

* 在Linux系统中，当我们把数据写入文件系统之后，其实数据在操作系统的`PageCache（页缓冲）`里面，并`没有刷到磁盘上`。如果操作系统挂了，数据就丢失了。
* 一方面，应用程序可以调用fsync这个系统调用来强制刷盘
* 另一方面，操作系统有后台线程，定时刷盘。
* 频繁调用fsync会影响性能，需要在性能和可靠性之间进行权衡。
* 另一方面，操作系统有后台线程，定时刷盘。频繁调用fsync会影响性能，需要在性能和可靠性之间进行权衡。
+ log.flush.interval.messages ： 达到消息数量时，会将数据flush到日志文件中。
+ log.flush.interval.ms ： 间隔多少时间(ms)，执行一次强制的flush操作。
+ flush.scheduler.interval.ms：所有日志刷新到磁盘的频率
* 官方不建议通过上述的三个参数来强制写盘，数据的可靠性应该通过`replica来保证`，而强制flush数据到磁盘会对整体性能产生影响。

3.4 副本同步

* Kafka中，分区的某个副本会被指定为 Leader，负责响应客户端的读写请求。分区中的其他副本自动成为 Follower，主动拉取（同步）Leader 副本中的数据，写入自己本地日志，确保所有副本上的数据是一致的。

3.5 数据一致性

* 在Kafka中，一旦Leader副本挂了，Follower副本可以选举成为新的Leader副本， 这样就提升了分区可用性，但是相对的，在提升了分区可用性的同时，也就牺牲了数据的一致性。
* 同步出现延迟的情况如下：

* 可以看到两个Follower副本同步的内容不一致，当leader副本因为意外原因宕掉了，那么Kafka为了提高分区可用性，此时会`选择2个follower副本中的一个作为Leader对外提供数据服务`。此时我们就会发现，对于消费者而言，`之前leader副本能访问的数据是D`，但是重新选择leader副本后，`能访问的数据就变成了C`，这样消费者就会认为数据丢失了，也就是所谓的数据不一致了。

3.5.1 **高水位 （HW ： High Watermark）**

4. 消费消息
=======

4.1 **消费者组**

4.1.1 **消费数据的方式：push & pull**

* 如果数据由Kafka进行`推送（push）`，那么多个分区的数据同时推送给消费者进行处理，明显一个消费者的消费能力是有限的，那么消费者无法快速处理数据，就会导致数据的积压，从而导致网络，存储等资源造成极大的压力，影响吞吐量和数据传输效率。

* 如果kafka的分区数据在内部可以存储的时间更长一些，再由消费者根据自己的消费能力向kafka`申请（拉取）数据`，那么整个数据处理的通道就会更顺畅一些。Kafka的Consumer就采用的这种拉取数据的方式。

4.1.2 **消费者组** **Consumer Group**

* 如果主题分区的数据过多，那么消费的时间就会很长。对于kafka来讲，数据就需要长时间的进行存储，那么对Kafka集群资源的压力就非常大。
* 如果希望提高消费者的消费能力，并且减少kafka集群的存储资源压力。所以有必要对消费者进行横向伸缩，从而提高消息消费速率。

* 不过这么做有一个问题，就是每一个消费者是独立，那么`一个消费者就不能消费主题中的全部数据`，简单来讲，就是对于某一个消费者个体来讲，`主题中的部分数据是没有消费到的`，`也就会认为数据丢了，这个该如何解决呢？`那如果我们将这多个消费者当成一个整体，是不是就可以了呢？这就是所谓的`消费者组 Consumer Group`。在kafka中，每个消费者都对应一个消费组，消费者可以是一个线程，一个进程，一个服务实例，如果kafka想要消费消息，那么需要指定消费那个topic的消息以及自己的消费组id(groupId)。

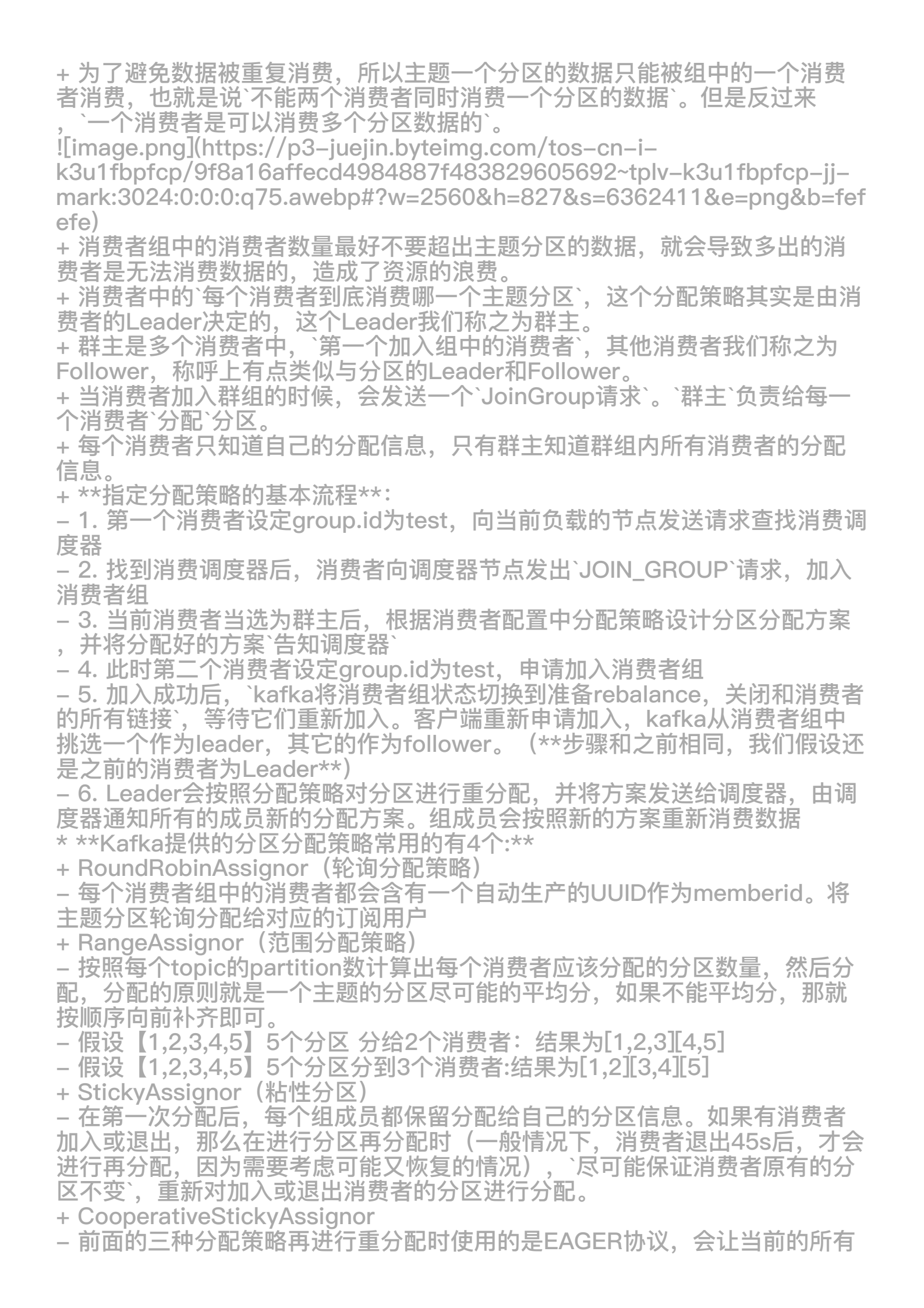
4.2 **调度（协调）器Coordinator**
---

* 消费者想要拉取数据，首先必须要`加入到一个组中`，成为消费组中的一员，同样道理，如果消费者出现了问题，也应该`从消费者组中剥离`。而这种加入组和退出组的处理，都应该由专门的管理组件进行处理，这个组件在kafka中，我们称之为`消费者组调度器（协调）（Group Coordinator）`

4.3 **消费者分配策略Assignor**
---

* 消费者想要拉取主题分区的数据，首先必须要加入到一个组中。

* 但是一个组中有多个消费者的话，那么每一个消费者该如何消费呢，是不是像图中一样的消费策略呢？如果是的话，那假设消费者组中只有2个消费者或有4个消费者，`和分区的数量不匹配，怎么办？`所以这里，我们需要给大家介绍一下，Kafka中基本的消费者组中的`消费者和分区`之间的分配规则：
+ 同一个消费者组的消费者都订阅同一个主题，所以消费者组中的多个消费者可以共同消费一个主题中的所有数据。

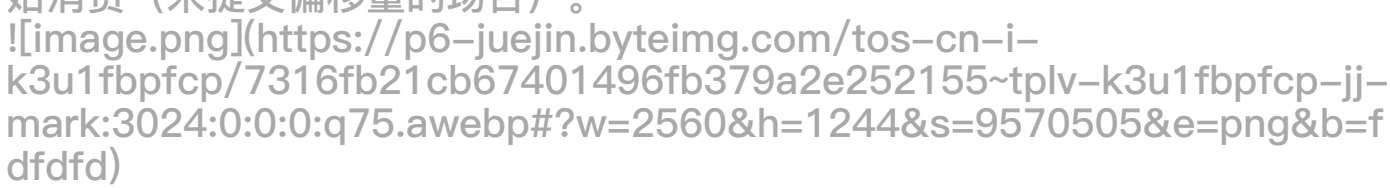
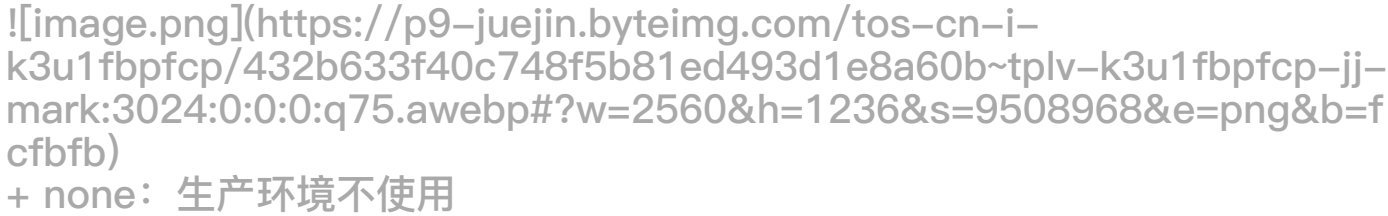
+ 为了避免数据被重复消费，所以主题一个分区的数据只能被组中的一个消费者消费，也就是说`不能两个消费者同时消费一个分区的数据`。但是反过来，`一个消费者是可以消费多个分区数据的`。


消费者放弃当前分区，关闭连接，资源清理，重新加入组和等待分配策略。明显效率是比较低的，所以从Kafka2.4版本开始，在粘性分配策略的基础上，优化了重分配的过程，使用的是COOPERATIVE协议，特点就是在整个再分配的过程中图中可以看出，粘性分区分配策略分配的会更加均匀和高效一些
，COOPERATIVE协议将一次全局重平衡，改成每次小规模重平衡，直至最终收敛平衡的过程。
+ Kafka消费者默认的分区分配就是**RangeAssignor,
CooperativeStickyAssignor***

4.4 **偏移量offset**

默认情况下，消费者如果不指定消费主题数据的偏移量，那么消费者启动消费时，无论当前主题之前存储了多少历史数据，消费者`只能`从`连接成功后`当前主题`最新的`数据偏移位置读取`(LEO)`，而`无法读取之前的任何数据`，如果想要获取之前的数据，就需要设定配置参数或指定数据偏移量。

4.4.1 **起始偏移量**

* 参数取值有3个：
+ earliest：
- 对于同一个消费者组，从头开始消费。就是说如果这个topic有历史消息存在，现在新启动了一个消费者组，且auto.offset.reset=earliest，那将会从头开始消费（未提交偏移量的场合）。

+ latest：
- 对于同一个消费者组，消费者只能消费到连接topic后，新产生的数据（未提交偏移量的场合）。

+ none：生产环境不使用
* 除了从最开始的偏移量或最后的偏移量读取数据以外，Kafka还支持从指定的偏移量的位置开始消费数据。

4.4.2**偏移量提交**

生产环境中，消费者可能因为某些原因或故障重新启动消费，那么如果不知道

之前消费数据的位置，重启后再消费，就可能重复消费（earliest）或漏消费（latest）。

所以Kafka提供了保存消费者偏移量的功能，而这个功能需要由消费者进行提交操作。这样消费者重启后就可以根据之前提交的偏移量进行消费了。`注意，一旦消费者提交了偏移量，那么kafka会优先使用提交的偏移量进行消费。此时，auto.offset.reset参数是不起作用的。`

4.4.2.1 **自动提交**

* 所谓的自动提交就是消费者消费完数据后，无需告知kafka当前消费数据的偏移量，而是由消费者客户端API周期性地将消费的偏移量提交到Kafka中。这个周期默认为5000ms，可以通过配置进行修改。

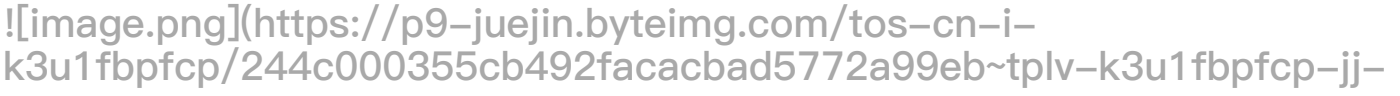
4.4.2.2 **手动提交**

基于时间周期的偏移量提交，是我们无法控制的，一旦参数设置的不合理，或单位时间内数据量消费的很多，却没有来及的自动提交，那么数据就会重复消费。所以Kafka也支持消费偏移量的手动提交，也就是说当消费者消费完数据后，自行通过API进行提交。不过为了考虑效率和安全，kafka同时提供了异步提交和同步提交两种方式供我们选择。注意：需要禁用自动提交auto.offset.reset=false，才能开启手动提交

* **异步提交**：向Kafka发送偏移量offset提交请求后，就可以直接消费下一批数据，因为无需等待kafka的提交确认，所以无法知道当前的偏移量一定提交成功，所以安全性比较低，但相对，消费性能会提高
* **同步提交**：必须等待Kafka完成offset提交请求的响应后，才可以消费下一批数据，一旦提交失败，会进行重试处理，尽可能保证偏移量提交成功，但是依然可能因为以外情况导致提交请求失败。此种方式消费效率比较低，但是安全性高。

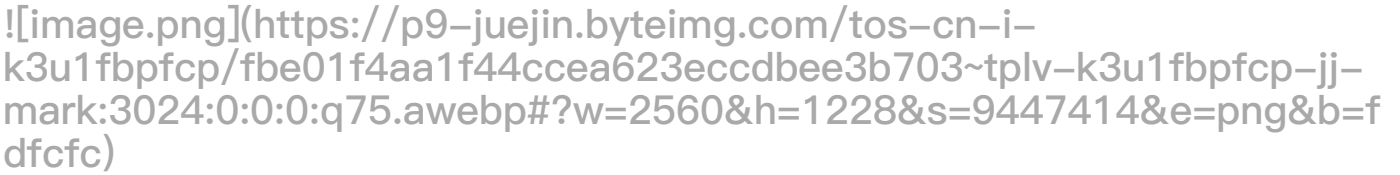
4.5 **消费者事务**

无论偏移量使用自动提交还是，手动提交，特殊场景中数据都有可能会出现重复消费。



mark:3024:0:0:0:q75.awebp#?w=1501&h=1440&s=6496241&e=png&b=fefcfc)

如果提前提交偏移量，再处理业务，又可能出现数据丢失的情况。



4.6 **偏移量的保存**

由于消费者在消费消息的时候可能会由于各种原因而断开消费，当重新启动消费者时我们需要让它接着上次消费的位置offset继续消费，因此消费者需要实时的记录自己以及消费的位置。

0.90版本之前，这个信息是记录在zookeeper内的，在0.90之后的版本，offset保存在__consumer_offsets这个topic内。

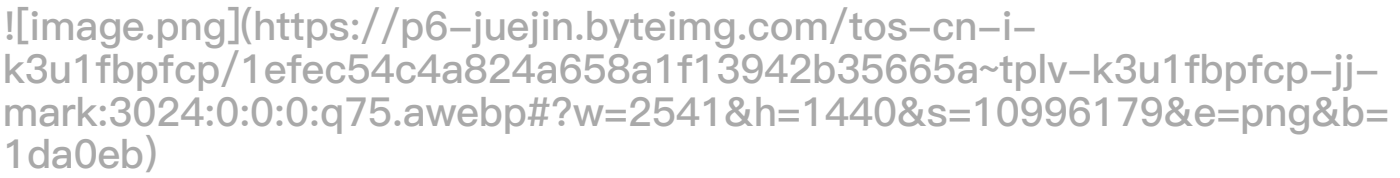
每个consumer会定期将自己消费分区的offset提交给kafka内部topic：__consumer_offsets，提交过去的时候，key是consumerGroupId+topic+分区号。value就是当前offset的值，kafka会定期清理topic里的消息，最后就保留最新的那条数据。

因为__consumer_offsets可能会接收高并发的请求，kafka默认给其分配50个分区(可以通过offsets.topic.num.partitions设置)，均匀分配到Kafka集群的多个Broker中。Kafka采用hash(consumerGroupId) % __consumer_offsets主题的分区数来计算我们的偏移量提交到哪一个分区。因为偏移量也是保存到主题中的，所以保存的过程和生产者生产数据的过程基本相同。

4.7 **消费数据**

因为__consumer_offsets可能会接收高并发的请求，kafka默认给其分配50个分区(可以通过offsets.topic.num.partitions设置)，均匀分配到Kafka集群的多个Broker中。Kafka采用hash(consumerGroupId) % __consumer_offsets主题的分区数来计算我们的偏移量提交到哪一个分区

。因为偏移量也是保存到主题中的，所以保存的过程和生产者生产数据的过程基本相同。



1. 服务端获取到用户拉取数据的请求
* Kafka消费客户端会向Broker发送拉取数据的请求FetchRequest，服务端Broker获取到请求后根据请求标记FETCH交给应用处理接口KafkaApis进行处理。
2. 通过副本管理器拉取数据
* 副本管理器需要确定当前拉取数据的分区，然后进行数据的读取操作
3. 通过副本管理器拉取数据
* 2.4版本前，数据读写的分区都是Leader分区，从2.4版本后，kafka支持Follower副本进行读取。主要原因就是跨机房或者说跨数据中心的场景，为了节约流量资源，可以从当前机房或数据中心的副本中获取数据。这个副本称之未首选副本。
4. 拉取分区数据
* Kafka的底层读取数据是采用日志段LogSegment对象进行操作的。
5. 零拷贝
* 为了提高数据读取效率，Kafka的底层采用nio提供的FileChannel零拷贝技术，直接从操作系统内核中进行数据传输，提高数据拉取的效率。