链滴

Kafka 的核心原理

作者：vcjmhg

原文链接：https://ld246.com/article/1627732329206

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile com/bing/20200316.jpg?imageView2/1/w/960/h/540/interlace/1/q/100"></p>
<h2 id="概述">概述</h2>
<p>什么是 Kafka？ </p>
<p>这里先引用官网首页的一句话：</p>
<blockquote>
<p>Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.</p>
</blockquote>
<p>翻译成中文，就是 Apache Kafka 是一个开源分布式事件流平台，被数千家公司用于高性能数据道、流分析、数据集成和关键任务应用程序。</p>
<p>网站上给的 Kafka 的定义着重强调的是 Kafka 在流处理方面的应用，但在实际场景中，它更多被用来当做消息引擎，也就是说 Kafka 一款开源的 基于发布订阅模的消息引擎系统 。</p>
<h2 id="消息引擎">消息引擎</h2>
<p>消息引擎，顾名思义，就是用来传递消息的，我们可以把它简单理解成**信箱”**。用户 A（生产者）把消息发给消息引擎，用户 B（消费者）从消息引擎中获取消息，进行费，而消息引擎就在其中起到了一个信箱的作用。</p>
<p>说道这里可能有小伙伴会有疑问了 ：我直接把消息从 A 发到 B 不就好了，中间为啥还要中间放消息引擎多此一举呢？</p>
<p>如果生产者和消费者处理消息的步调一致，我们确实没必要中间加一个消息引擎，而在大部分业中，两者处理消息的步调并不一致，这样可能会造成丢失的情况。我们以秒杀系统为例，在一个秒杀景中，用户下单后会调用订单系统生成对应的订单，而处理该订单的下游可能有 多个子服系统 ，比如调用支付宝接口和微信支付接口，查询登录状态等。显然上游的业务逻辑比较单处理速度较快，且秒杀场景下 tps 很高，而下游处理速度远小于上游服务，如果直接对接，必然会现订单堆积的情形，甚至可能导致下游服务崩溃。</p>
<p>而引入 <code>消息引擎</code> 之后，上游服务和下游服务直接就实现了一种解耦/strong>，上游订单服务不再直接和下游子服务进行交互，当新订单生成后它仅仅向 Kafka Broker 送一条订单消息即可。而下游各个子服务订阅 Kafka 的相应主题，并实时从各自分区里边拿数据进行理，从而实现了上游订单和下游处理服务之间的解耦。这样当出现秒杀业务时，Kafka 能够瞬间增加订单流量全部以消息形式保存到对应的主题中，既不影响上游的 TPS，也给了下游子服务流出充足的间来消费它。</p>
<p>我们可以简单借助两张图来说明，在引入消息队列之前，PV 图是这样色的</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/siyuan/1568182249394/assets/image-20210715164450-a826qy7.png?imageVew2/2/interlace/1/format/jpg"></p>
<p>很明显流量在某个瞬间达到了峰值，然后瞬间又降低了，但引入消息引擎之后，PV 图发生了变：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/siyuan/1568182249394/assets/image-20210715164708-3xxbfu5.png?imageViw2/2/interlace/1/format/jpg"></p>
<p>好像山峰被削平了意义，这就是消息队列的作用即“削峰填谷”，实现流量平滑过渡。</p>
<p>知道了消息引擎的作用，接下来我们聚焦于消息引擎本身，一个消息引擎本质上讲需要做好三件：</p>

定义好消息编码格式
实现消息的传递
实现消息的保存

<p>那 Kafka 是如何解决上边三个问题呢？</p>
<h2 id="消息格式">消息格式</h2>
<p>当前对消息的编码格式其实有许多种成熟的解决方案，比如比如非常普遍的 CSV、JSON 或 YAML，再比如谷歌的 Protocol Buffer 或 Facebook 的 Thrift。这些都是很酷的方

，但 Kafka 上边方法都用 :smile::smiling_imp:。它为了减少消息的体积，节省空间，直接
择的 ByteBuf 这种紧密的二进制存储格式。</p>
<p>具体消息的消息格式被分成两个层次，消息集合和消息：消息集合（message set）以及消息（
essage）。两者的关系可以用一张图来描述：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210715173149-66l31ko.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>一个消息集合中包含若干条日志项（record item），而日志项
是真正封装消息的地方。Kafka 底层的消息日志由一系列消息集合日志项组成。Kafka 通
不会直接操作具体的一条条消息，它总是在消息集合这个层面上进行写入操作。
/p>
<p>消息体之上有了两个字段 <code>offset</code> 以及 <code>message size</code> 其中
中 <code>offset</code> 标志其在分区中的偏移量，这个 <code>offset</code> 是逻辑值而并
实际值，<code>message size</code> 则标识的消息的大小，两者一起构成了日志的头部，固定 1
B。</p>
<h2 id="v0">v0</h2>
<p>具体到消息层面上来说，消息的格式经历过两次演变，对于 Kafka 消息格式的第一个版本，我们
它称之为 v0，在 Kafka 0.10.0 版本之前都是采用的这个消息格式。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210715175122-irk6oeg.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>从 crc32 字段开始算，每个字段含义如下：</p>
<table>
<thead>
<tr>
<th>字段名称</th>
<th>含义</th>
<th>大小</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc32</td>
<td>crc32 校验值，用来做信息校验</td>
<td>4B</td>
</tr>
<tr>
<td>attributes</td>
<td>消息的属性，低 3 位表示压缩类型:
0 表示 NONE、1 表示 GZIP、2
表示 SNAPPY、3 表示 LZ4（LZ4 自 Kafka 0.9.x 引入），其余位保留。
</td>
<td>1B</td>
</tr>
<tr>
<td>magic</td>
<td>消息格式版本号，V0 版本 magic 为 0</td>
<td>1B</td>
</tr>
<tr>
<td>key length</td>
<td>表示消息的 key 的长度。如果为-1，则表示没有设置 key，即 key=null。</td>
<td>4B</td>
</tr>
<tr>
<td>key（可选）</td>

<td>如果没有 key 则无此字段。</td>
<td>非固定</td>
</tr>
<tr>
<td>value length</td>
<td>实际消息体的长度。如果为-1，则表示消息为空。</td>
<td>4B</td>
</tr>
<tr>
<td>value</td>
<td>消息体，可以为空</td>
<td>非固定</td>
</tr>
</tbody>
</table>
<p>从这个表格中，我们不难算出，对于 V0 协议来说，一个消息的最小大小为：crc32 + magic + a
tributes + key length + value length = 4B + 1B + 1B + 4B + 4B =14B，也就是说 v0 版本中一
消息的最小长度为 14B，如果小于这个值，那么这就是一条破损的消息而不被接受。</p>
<h2 id="v1">v1</h2>
<p>V0 某种程度上讲，重要字段都已经包含了，但存在着一个较大的缺陷---没有时间戳，无法删除
些超时，失效的消息。因而 kafka 从 0.10.0 版本开始到 0.11.0 版本消息格式从 V0 升级到了 V1，
两个版本的消息格式之间仅仅就差了一个 <code>timestamp</code> 字段，表示消息的时间戳，V1
版本的消息结构图如下所示</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210715180553-nhvfmxi.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>较之于 v0 版本，v1 版本的消息格式发生了如下变化：</p>
<table>
<thead>
<tr>
<th>字段名称</th>
<th>含义</th>
<th>变化</th>
</tr>
</thead>
<tbody>
<tr>
<td>:point_right: magic</td>
<td>版本号</td>
<td>变成了 1</td>
</tr>
<tr>
<td>:point_right: attributes</td>
<td>消息属性</td>
<td>原本只用了低三位表示压缩类型，现在将第 4 位利用了起来，用来表示 ti
estamp 的含义：
其中 0 表示 timestamp 为 CreateTime 即创建的时间，1 表示 timestamp
型为 LogAppendTime
即日志追加时间，可以理解成是修改时间
</td>
</tr>
<tr>
<td>:heavy_plus_sign: timestamp</td>
<td>时间戳</td>
<td>增加了时间戳属性，大小为 8B</td>
</tr>
</tbody>

</table>
<p>该版本协议最小的大小也变成了 14+8=22B</p>
<h2 id="v2">v2</h2>
<p>kafka 从 0.11.0 版本开始所使用的消息格式版本为 v2，这个版本的消息相比于 v0 和 v1 的版本
言改动很大，在消息集层面上看，消息集合的名称发生了变化，V2 之前消息集称之为 <code>Messa
e Set</code> 而 v2，则改成 <code>Record Batch</code>，结果也发生了变化，具体如下图所
：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210715193828-hr62i7c.png?imageVi
w2/2/interlace/1/format/jpg"></p>

CRC 值从消息中移除，被迁移到消息批次中
first offset：表示当前 RecordBatch 的起始位移。
length：计算 partition leader epoch 到 headers 之间的长度。
partition leader epoch：用来确保数据可靠性，详细可以参考 KIP-101
magic：消息格式的版本号，对于 v2 版本而言，magic 等于 2。
attributes：消息属性，注意这里占用了两个字节。低 3 位表示压缩格式，可以参考 v0 和 v1；第
4 位表示时间戳类型；第 5 位表示此 RecordBatch 是否处于事务中，0 表示非事务，1 表示事务。第
 位表示是否是 Control 消息，0 表示非 Control 消息，而 1 表示是 Control 消息，Control 消息用
支持事务功能。
last offset delta：RecordBatch 中最后一个 Record 的 offset 与 first offset 的差值。主要被 broker
用来确认 RecordBatch 中 Records 的组装正确性。
first timestamp：RecordBatch 中第一条 Record 的时间戳。
max timestamp：RecordBatch 中最大的时间戳，一般情况下是指最后一个 Record 的时间戳，和 la
t offset delta 的作用一样，用来确保消息组装的正确性。
producer id：用来支持幂等性，详细可以参考 KIP-98。
producer epoch：和 producer id 一样，用来支持幂等性。
first sequence：和 producer id、producer epoch 一样，用来支持幂等性。
records count：RecordBatch 中 Record 的个数。

<p>从具体的消息格式角度来看，字段的类型借鉴 Protocol Buffer 引入 varint 类型和 ZigZag 编码
</p>
<p>Varints 是使用一个或多个字节来序列化整数的一种方法，数值越小，其所占用的字节数就越少。
igZag 编码以一种锯齿形（zig-zags）的方式来回穿梭于正负整数之间，以使得带符号整数映射为无
号整数，这样可以使得绝对值较小的负数仍然享有较小的 Varints 编码值，比如-1 编码为 1,1 编码为
，-2 编码为^1。详细可以参考：<a href="https://link.l
246.com/forward?goto=https%3A%2F%2Fdevelopers.google.com%2Fprotocol-buffers%2Fdo
s%2Fencoding" target="_blank" rel="nofollow ugc">https://developers.google.com/protocol
buffers/docs/encoding。</p>
<p>v2 版本的消息格式去掉了 crc 字段，将其移动到消息集合字段中（公共部分提到消息集合中，
个消息集合只需要保存一个字段，可以节省空间，提高压缩比），另外增加了 length（消息总长度
、timestamp delta（时间戳增量）、offset delta（位移增量）和 headers 信息，并且 attributes
弃用了,具体变化如下表所示：</p>
<table>
<thead>
<tr>
<th>字段名称</th>
<th>含义</th>
<th>变化</th>
</tr>
</thead>
<tbody>
<tr>
<td>:heavy_plus_sign: length</td>

<td>消息字段总长度</td>
<td>新增</td>
</tr>
<tr>
<td>:heavy_plus_sign: timestamp delta</td>
<td>时间戳增量</td>
<td>新增，通常一个 timestamp 需要占用 8 个字节，在这里
保存和信息集合的时间戳差值的可能进一步节省空间
</td>
</tr>
<tr>
<td>:heavy_plus_sign:offset delta</td>
<td>位移增量</td>
<td>新增，保存和 Recorbatch 起始位移的差值</td>
</tr>
<tr>
<td>:heavy_plus_sign:headers</td>
<td>头信息</td>
<td>新增，用来支持应用级别的扩展，不需要像 v0 和 v1 版本
一样不得不将一些应用级别的性嵌入在消息体里面
</td>
</tr>
<tr>
<td>:heavy_minus_sign:attribute</td>
<td>消息属性</td>
<td>弃用，但在消息格式中仍占据 1B 大小，以备未来扩展</td>
</tr>
</tbody>
</table>
<p>消息批次最小为 61 字节，相比 V0、V1 版本要大很多。当存储多条消息时，由于 V2 版本主要通过可变长度提高了消息格式的空间使用率，并将某些字段移到消息批次中，同时消息批次可容纳多消息，因而整体的大小会比 v1 和 v0 小很多。</p>
<p>为了进一步减小消息长度，节省磁盘空间和带宽，Kafka 也提供了消息压缩的功能。</p>
<h2 id="消息的压缩">消息的压缩</h2>
<p>尤其是在 V2 版本消息格式的变化（比如使用了 varint 类型，将大量公共信息从消息移动到消息合体中），使得消息的压缩方法从原来以消息为单位进行压缩，变成现在以消息集合为单位进行压缩使得压缩效率大幅度提高，下图展示了两者压缩效果的比较图：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/siyuan/1568182249394/assets/image-20210718135614-1wka5xu.png?imageVew2/2/interlace/1/format/jpg"></p>
<h3 id="如何开启压缩">如何开启压缩</h3>
<p>在 Kafka 中压缩可能发生在两个地方：生产者端和 Broker 端。</p>
<p>生产者端：</p>
<p>生产者端如果想要启动压缩，只要配置响应参数即可：</p>
<pre><code class="language-java highlight-chroma">Properties/span> props = <spn class="highlight-k">new Properties();
props.put("botstrap.servers", "locahost:9092");
props.put("acs", "all"<spa class="highlight-o">);

props.<span class="
ighlight-na">put("ke
.serializer", "org.apac
e.kafka.common.serialization.StringSerializer");
props.<span class="
ighlight-na">put("va
ue.serializer", "org.ap
che.kafka.common.serialization.StringSerializer");
// 开启GZIP压缩
props<spa
class="highlight-o">.put<span class="highligh
-o">("compression.type"<span class="highlight
o">, "gzip");

Producer <
span class="highlight-n">String , String > <span c
ass="highlight-n">producer = <span class="hig
light-k">new KafkaProducer<span class="highli
ht-o"><>(props);

</code></pre>

<p>其中第 7 行代码就是配置生产者端 Producer 的压缩算法使用的是 GZIP。</p>
<p>也就是说配置完该参数之后，Kafka 在发送消息之后都会讲消息压缩成 GZIP 方法在
送到 Broker 端。</p>
<p>Broker 端：</p>
<p>同样的 Broker 启动消息压缩的方法也非常简单，只需要将 <code>compression.type</code>
设置响应的压缩值即可，但需要注意的是，这个参数的意思不是说对接收的消息都进行压
，而是指的是接收消息的格式，也就是说在 Broker 端配置这个参数之后，Broker 端如果
到的消息是其想要的压缩类型，则直接接收，如果是其他类型则 Broker 端则需要进行解压
重压缩操作将消息转换成指定的格式。</p>
<h3 id="何时会压缩">何时会压缩</h3>
<p>对于生产者端来说：只要开启了压缩，在发送的消息都会压缩成配置的格式，正如上边的代码，
置消息的压缩类型为** “gzip” **，那么对于发送的所有消息都会先压缩成“gzip”格式然后再发送
</p>
<p>但对 broker 端则不同，下边两种情况会发生压缩：</p>

Broker 端和 Producer 端配置了不同的压缩参数
在这种情况下，Broker 会先将接受到的信息进行解压缩，然后再压缩成其配置的格式。
Broker 端发生了消息格式转换。

<p>所谓的消息格式转换主要是为了兼容老版本的消费者程序。还记得之前说过的 V1、V2 版本吧?
一个生产环境中，Kafka 集群中同时保存多种版本的消息格式非常常见。为了兼容老版本的格式，Bro
er 端会对新版本消息执行向老版本格式的转换。这个过程中会涉及消息的解压缩和重新压缩。一般情
下这种消息格式转换对性能是有很大影响的，除了这里的压缩之外，它还让 Kafka 丧失了引以为豪的
特性。</p>
<h3 id="何时解压缩">何时解压缩</h3>
<p>有压缩必然有解压缩，通常来说压缩是发生在消费者程序中，也就是说消费者端从 Bro
er 端获取到消息之后，需要自行解压还原成之前的消息。</p>
<p>可能会有小伙伴问了，Consumer 端是如何知道具体的消息类型的呢？</p>
<p>其实就是在消息中，Kafka 会将启用何种压缩算法封装到消息集合中，这样 Consumer 获取到
息集合之后就知道了这些消息使用了哪种压缩算法。</p>

<p>如果简单总结一下，三者一个关系其实就是：生产者端压缩，Broker 端<st
ong>尽量保持，消费者端解压缩。</p>
<h3 id="最佳实践">最佳实践</h3>
<p>前面一些小节对压缩的时机原理进行了简单的描述，但具体我们在什么情况下启用压缩呢？</p>
<p>你现在已经知道 Producer 端完成的压缩，那么启用压缩条件就是：</p>

Producer 程序运行机器上的 CPU 资源要很充足。如果 Producer 运行机器本身 CPU 已经消耗
尽了，那么启用消息压缩无疑是雪上加霜，只会适得其反。
另外如果我们带宽资源有限，我们也要尽量开启压缩

<p>消息的压缩过程我们简单了解了，但消息在不同端之间传输的时候又经历了什么呢？</p>
<h2 id="消息传输">消息传输</h2>
<p>在具体讲消息传输部分之前，我们应该先看一张 Kafka 的一张整体架构图<span class="langua
e-math">^4：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/siyuan/1568182249394/assets/58c35d3ab0921bf0476e3ba14069d291-20210718150817
gpveb2d.jpg?imageView2/2/interlace/1/format/jpg"></p>
<p>整个架构简单来分的话可以分成三端，生产者、消费者和 Broker 端。</p>
<p>向主题发布消息的客户端应用程序称为生产者（Producer），生产者程序通常持续不断地向一
或多个主题发送消息，而订阅这些主题消息的客户端应用程序就被称为消费者（Consumer）。和生
者类似，消费者也能够同时订阅多个主题的消息。我们把生产者和消费者统称为客户端（Clients）。
/p>
<p>而与之相对的是 Brokor 端服务器端，Kafka 的服务器端由被称为 Broker 的服务进程
成，即一个 Kafka 集群由多个 Broker 组成，Broker 负责接收和处理客户端发送过来的
求，以及对消息进行持久化。其实一个 Broker 我们可以简单理解为一个 kafka 服务器端
序进程。为了保证 Kafka 的高可用，一般将多个 Broker 进行运行在多个不同的主机上。<
p>
<p>当然为了提高 Broker 集群的可用性，Broker 引入了备份机制，备份的思想非常简单也就是把<st
ong>相同的数据拷贝到多台机器上，这些被拷贝的数据就称之为副本。在 Kafaka 中由
类副本：领导者副本和追随者副本，前者对外提供服务，后者被动的追随领导者副本进行数据同步。
可能跟我们接触的 mysql 集群不同。</p>
<p>虽然有了副本可以解决数据持久化或者消息不丢失的问题，但是却没有解决伸缩性问题（动态扩
的问题），什么伸缩性问题呢？简单来说随着数据规模的增大，领导者副本积累了太多的数据以至于
台 Broker 机器都无法容纳了，此时应该怎么办呢？一个最简单的思想就是把数据分成多块，存储到
同的 Broker 上。</p>
<p>Kafka 中的分区机制指的是将每个主题划分成多个分区（Partition），每
分区是一组有序的消息日志。生产者生产的每条消息只会被发送到一个分区中，也就是说如果向一个
分区的主题发送一条消息，这条消息要么在分区 0 中，要么在分区 1 中。如你所见，Kafka 的分区编
是从 0 开始的，如果 Topic 有 100 个分区，那么它们的分区号就是从 0 到 99。</p>
<p>通过前面的描述我们可以简单总结出 kafka 的消息架构：</p>

第一层是主题层，每个主题可以配置 M 个分区，而每个分区又可以配置 N 个副本。
第二层是分区层，每个分区的 N 个副本中只能有一个充当领导者角色，对外提供服务；其他 N-1
个副本是追随者副本，只是提供数据冗余之用。
第三层是消息层，分区中包含若干条消息，每条消息的位移从 0 开始，依次递增。最后，客户端
序只能与分区的领导者副本进行交互。

<p>了解 kafka 的基础概念之后，我们简单总结下消息产生到消费的整个过程。</p>
<p>一般来说一个消息从被产生到被消费会经历如下一个过程：</p>
<p>首先生产者会产生指定主题的消息（当然该消息根据配置可能会被压缩），多个消息构成一个 <c
de>Record Batch</code> 然后通过建立的网络连接发送到 Broker 端，Broker 端接受到消息之后
对应的 <code>Record Bath</code> 写入到指定主题的主副本（如何确定）的对应分区中（如何
位），Follow 副本会主动从 Leader 副本中拉取消息实现副本同步，在同步完成（完成标志？）之后
该主题订阅者中对应该分区的消费者会拉去消息进行消费，整个过程完成。</p>

```
<p>当然上边只是粗略的描述了一下消息所经历的过程，可能小伙伴听完之后可能会有很多疑问：</
>
<ul>
<li>消费者是如何与 Broker 端建立网络连接的？会建立多少网络连接？</li>
<li>Broker 端接收到消息之后会选择哪个分区来写入消息呢？</li>
<li>Broker 端如何确定主题对应的主副本呢？</li>
<li>主从副本之间是如何实现消息同步的呢？</li>
<li>....</li>
</ul>
<p>这些问题，我们下面的章节都会逐个回答。现在我们先将话题切回本章主题，也即<strong>消
是如何建立连接进行传输的？</strong></p>
<h2 id="Java生产者如何管理TCP连接">Java 生产者如何管理 TCP 连接</h2>
<p>首先我们要知道 Kafka 所有的通信都是基于 TCP 的，而不是基于 HTTP 或者其他协议，具体这
设计原因，从社区的角度来看，在开发客户端时，人们能够利用 TCP 本身提供的一些高级功能，比
多路复用请求以及同时轮询多个连接的能力。</p>
<p>Kafka 的 Java 生产者 API 主要的对象就是 KafkaProducer。通常我们开发一个生产者的步骤有 4
步：</p>
<ul>
<li>第 1 步：构造生产者对象所需的参数对象。</li>
<li>第 2 步：利用第 1 步的参数对象，创建 KafkaProducer 对象实例。</li>
<li>第 3 步：使用 KafkaProducer 的 send 方法发送消息。</li>
<li>第 4 步：调用 KafkaProducer 的 close 方法关闭生产者并释放各种系统资源。</li>
</ul>
<p>具体代码如下：</p>
<pre><code class="language-java highlight-chroma">
<span class="highlight-n">Properties</span> <span class="highlight-n">props</span> <s
an class="highlight-o">=</span> <span class="highlight-k">new</span> <span class="hig
light-n">Properties</span> <span class="highlight-o">();</span>
<span class="highlight-n">props</span><span class="highlight-o">.</span><span class="
ighlight-na">put</span><span class="highlight-o">(</span><span class="highlight-err">
</span><span class="highlight-n">参数1</span><span class="highlight-err">” </span><s
an class="highlight-o">,</span> <span class="highlight-err"> “</span><span class="highli
ht-n">参数1的值</span><span class="highlight-err">” </span><span class="highlight-o">
</span><span class="highlight-err">; </span>
<span class="highlight-n">props</span><span class="highlight-o">.</span><span class="
ighlight-na">put</span><span class="highlight-o">(</span><span class="highlight-err">
</span><span class="highlight-n">参数2</span><span class="highlight-err">” </span><s
an class="highlight-o">,</span> <span class="highlight-err"> “</span><span class="highli
ht-n">参数2的值</span><span class="highlight-err">” </span><span class="highlight-o">
</span><span class="highlight-err">; </span>
<span class="highlight-err">......</span>
<span class="highlight-k">try</span> <span class="highlight-o">(</span><span class="hi
hlight-n">Producer</span><span class="highlight-o">&lt;</span><span class="highlight-n
>String</span><span class="highlight-o">,</span> <span class="highlight-n">String</spa
><span class="highlight-o">&gt;</span> <span class="highlight-n">producer</span> <sp
n class="highlight-o">=</span> <span class="highlight-k">new</span> <span class="highl
ght-n">KafkaProducer</span><span class="highlight-o">&lt;&gt;(</span><span class="hi
hlight-n">props</span><span class="highlight-o">))</span> <span class="highlight-o">{<
span>
        <span class="highlight-n">producer</span><span class="highlight-o">.</span><s
an class="highlight-na">send</span><span class="highlight-o">(</span><span class="high
ight-k">new</span> <span class="highlight-n">ProducerRecord</span><span class="highl
ght-o">&lt;</span><span class="highlight-n">String</span><span class="highlight-o">,</
pan> <span class="highlight-n">String</span><span class="highlight-o">&gt;(</span><s
```

an class="highlight-err">......), callback);
 ......
}
</code></pre>
<p>:question::question: 从代码上，好像并没有显示的用代码建立连接,<stro g>那它的连接是什么时候建立的呢？</p>
<p>首先，生产者应用在创建 KafkaProducer 实例时是会建立与 Broker 的 TCP 连接的。其实这种述也不是很准确，应该这样说：在创建 KafkaProducer 实例时，**生产者应用会在后台创建并启动个名为 Sender 的线程，该 Sender 线程开始运行时首先会创建与 Broker 的连接。**我们结合一段志来看：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https: /b3logfile.com/siyuan/1568182249394/assets/image-20210718172004-4tz9z44.png?imageVi w2/2/interlace/1/format/jpg"></p>
<p>从第一段日志上可以看到，KafkaProducer 会先发送一个元数据请求（sending metadata requ st），而这个请求就是我们前边说的 Sender 线程建立发起的。</p>
<p>这个时候可能会有小伙伴问了，如果有多个 Broker 怎么办？</p>
<p>这个就设计到 Producer 创建时 <code>bootstrap.servers</code> 参数了，该参数是 Produc r 的核心参数之一，指定了这个 Producer 启动时要连接的 Broker 地址。因而 Sender 线程在建立连时，会和 bootstrap.servers 设置的所有 Broker 建立连接。因此如果你在</str ng> bootstrap.servers中设置了多个 100 个 Broker，那么你的 Producer 在启动的时候会和这 100 个 Broker 建立 TCP 连接。</p>
<p>在实际生产中，是不太建议在 bootstrap.servers 参数设置所有集群信息的，因为这样建立连接效率很慢而且网络代价很大，通常情况下设置 3 到 4 台就可以了，因为实际上只要 Produc r 建立起与任意一台 Broker 连接之后就能拿到整个 Broker 的集群信息，因而没必要在该数中配置所有的 Broker。</p>
<p>从第 3~4 行日志，可以看到 Producer 获取元数据后，开始于其他未建立起连接的 Broker 建立连接。</p>
<p>从前边的分析，我们可以看到 Producer 和 Broker 建立连接的地方有两处：</p>

先建立起 bootstap.servers 中 Broker 的连接
获取集群元数据后建立起其他未连接的 Broker 建立起连接（可能，因为如已经和所有集群中 Broker 连接完成了，则不会再建立新连接）

<p>除此之外，实际上 TPC 连接还可能在发送消息的时候，当 Prod cer 发现尚不存在与目标 Broker 的连接，也会创建一个。</p>
<p>纵观它的连接过程，其实我们可以发现其不足之处，首先在 Producer 创建连接的时候使用的 Ka kaProducer 是线程安全的类，但其在创建 KafkaProducer 实例的时候去创建连接，这样会有风险：对象构造器中器启动线程会有指针逃逸的风险。
另一问题，就是 KafkaProducer 建立连接的时候有必要跟所有的 Broker 都建立连接吗？</p>
<p>试想一下，在一个有着 1000 台 Broker 的集群中，你的 Producer 可能只会与其中的 3～5 台 B oker 长期通信，但是 Producer 启动后依次创建与这 1000 台 Broker 的 TCP 连接。一段时间之后大约有 995 个 TCP 连接又被强制关闭。这难道不是一种资源浪费吗？很显然，这里是有改善和优化空间的。 比如可以将连接延后到实际需要的时候再建立呢？:smile:</p>
<p>:question::question: 有连接的建立必然就会有连接的关闭，那 TPC 连接是何时关闭呢？</p>
<p>Producer 关闭连接的方式有两种：</p>
<p>第一种是用户主动关闭：</p>
<p>比如调用 <code>producer.close()</code> 主动关闭或者调用 <code>kill命令</code> 杀线程</p>
<p>第二种是 Kafaka 自动关闭：</p>
<p>与 Producer 端参数 <code>connections.max.idle.ms</code> 的值有关。默认情况下该参数是 9 分钟，即如果在 9 分钟内没有任何请求“流过”某个 TCP 连接，那么 Kafka 会主动帮你把该 TC 连接关闭。用户可以在 Producer 端设置 connections.max.idle.ms=-1 禁掉这种机制。一旦被设置

 -1，TCP 连接将成为永久长连接。当然这只是软件层面的“长连接”机制，由于 Kafka 创建的这些 S
cket 连接都开启了 keepalive，因此 keepalive 探活机制还是会遵守的。</p>
<h2 id="Java消费者如何管理TCP连接">Java 消费者如何管理 TCP 连接</h2>
<p>消费者要进行消费，构建的实例是 KafkaConsumer，但与 KafkaProducer 不同的是，<strong
构建 KafkaConsumer 时不会创建任何 TCP 连接，也就是说当执行完 <code>neww Kafk
Consumer(properties)</code> 之后不会创建任何 TCP 连接。
而是在调用 <code>KafkaConsumer.poll()</code> 方法时被创建，这个设计也就避免了我们前边
的 this 指针逃逸的问题 :smile::smile:。从这点看，这样的设计可能会更好。
具体来说在调用 <code>poll()</code> 方法内部会有三个时机创建 TCP 连接：</p>

发起 FindCoordinator 请求时。
消费者端有个组件叫 <code>协调者（Coordinator）</code>，它驻留在 Broker 内存中，负责消
者组的组成员管理和各个消费者的位移提交管理。当消费者程序首次调用 <code>poll()</code> 时
向 Kafka 发送一个名为 <code>FindCorrdinator</code> 的请求，去获取哪个 Broker
管理它的协调者。
但消费者应该向哪个 Broker 发起这个请求呢？理论上任何一个 Broker 都可以回答这个问题，也就
说消费者端可以向任意服务器发送 FindCorrdinator 请求，但出于负
均衡考虑，<code>KafaConsumer</code> 会向消费者端所连接的 Broker 中负载（<st
ong>待发送请求最少的 Broker）最小的那个发送该请求。当　然，这种评估是 Broker
单项评估，可能并不是最优。
连接协调者时
Broker 处理完上一步发送的 FindCoordinator 请求之后，会返还对应的响应结果（Response），
式地告诉消费者哪个 Broker 是真正的协调者，因此在这一步，消费者知晓了真正的协调者后，会创
连向该 Broker 的 Socket 连接。只有成功连入协调者，协调者才能开启正常的组协调操作，比如加
组、等待组分配方案、心跳请求处理、位移获取、位移提交等。
消费数据时
消费者会为每一个要消费的分区创建与该分区领导者副本所在的 Broker 连接的 TCP</stro
g>，也就是说也和所消费分区的领导者副本所在的 Broker 全都建立起连接，举个例子，假设消费者
消费 5 个分区的数据，这 5 个分区各自的领导者副本分布在 4 台 Broker 上，那么该消费者在消费时
创建与这 4 台 Broker 的 Socket 连接。

<p>通过上面的描述，我们思考这样一个问题：</p>
<blockquote>
<p>假设有个 Kafka 集群由 2 台 Broker 组成，有个主题有 5 个分区，当一个消费该主题的消费者
序启动时，你认为该程序会创建多少个 Socket 连接？为什么？</p>
</blockquote>
<p>很显然，一共会创建四个 TCP，稳定后会变成三个，具体来说：</p>

第一类：先以发送携带“假”ID 的 <code>FindCorrdinator</code>，请求获取协调者信息，
续该请求会因超时而被断开
然后连接对应主题的协调者（Corrdinator）
与分区所在的两个 Broker 建立两个连接。

<p>:question::question: 同样的，TCP 连接何时关闭呢？</p>
<p>和生产者类似，消费者关闭 Socket 也分为主动关闭和 Kafka 自动关闭。</p>

主动关闭：调用 KafkaConsumer.close()方法来关闭，或者执行 kill 命令
自动关闭：消费者端参数 connection.max.idle.ms 控制的，该参数现在
默认值是 9 分钟，即如果某个 Socket 连接上连续 9 分钟都没有任何请求“过境”的话，那么消费者
强行“杀掉”这个 Socket 连接。

<p>另外和生产者不同的是，消费者端的连接可能会被动关闭，比如如果 Broker 端 <code>heartbe
t.interval.ms</code> 参数设置过小，Consumer 端两次轮询时间大于该参数值，则 KafaConsumer
和 Broker 端建立的连接可能会被中断。</p>

<p>另一方面，消费者端的设计可能也有些问题，第一类 TCP 连接仅仅是为了首次获取元数据而创
的，后面就会被废弃掉。最根本的原因是，消费者在启动时还不知道 Kafka 集群的信息，只能使用一
“假”的 ID 去注册，即使消费者获取了真实的 Broker ID，它依旧无法区分这个“假”ID 对应的是
台 Broker，因此也就无法重用这个 Socket 连接，只能再重新创建一个新的连接。</p>
<p>造成这样的原因，目前 Kafka 仅仅通过 ID 这一个维度的数据来表征 Socket 信息，这点信息显
不能够确定连接的是那一台 Broker，也许未来，社区应该考虑使用 <code><主机名，端口，ID&g
;</code> 三元组的方式来定位 Socket 的资源，这样或许能够让消费者程序少创建一些 TCP 连接。<
p>
<p>连接建立之后还不算完，我们需要保证消息能够正确的被送达，kafka 是如何做到这
点的呢？</p>
<h2 id="Kafka消息交付的可靠性保障">Kafka 消息交付的可靠性保障</h2>
<p>所谓的消息交付可靠性保障，是指 Kafka 对 Producer 和 Consumer 要处理的消息提供什么样
承诺。常见的承诺有以下三种：</p>

最多一次（at most once）：消息可能会丢失，但绝不会被重复发送。</li

至少一次（at least once）：消息不会丢失，但有可能被重复发送。
精确一次（exactly once）：消息不会丢失，也不会被重复发送。

<p>其中 Kafka 默认情况下支持的第二种可靠性保障，即能够保证消息不会丢失，这一点比较容易，
要保证发送失败后增加重试机制即可。</p>
<p>但我们也可以配置 Kafka 支持第三种可靠性保证，即精确一次，
afa 是如何实现这一点呢？</p>
<p>简单来说，kafka 是通过两种机制来保证的，<code>幂等性</code> 和 <code>事务</code
。</p>
<h3 id="幂等性Producer">幂等性 Producer</h3>
<p>什么是幂等性呢？</p>
<p>在数学中，指的是 <code>f(f(x)) = f(x)</code>，即 x 的重复性嵌套输入不影响输出的结果值
迁移到软件开发中，一次请求某个资源或者多次请求某个资源 必须有一致性的结果</stron
> 。例如，在针对同一个订单，一次下单请求和多次下单请求的结果应该是一致的，最终都是该订单
单成功。</p>
<p>幂等性有很多好处，其最大的优势在于我们可以安全地重试任何幂等性操作，反正它
也不会破坏我们的系统状态。</p>
<p>在 kafka 中 Producer 默认是不支持幂等性的，但我们可以创建幂等性 Producer，它其实是 0.11
0.0 版本引入的新功能。在此之前，Kafka 向分区发送数据时，可能会出现同一条消息被发送了多次
导致消息重复的情况。在 0.11 之后，指定 Producer 幂等性的方法很简单，仅需要设置一个参数即
，即 <code>props.put(“enable.idempotence”, ture)</code>，或 <code>props.put(Produce
Config.ENABLE_IDEMPOTENCE_CONFIG， true)</code>。</p>
<p>底层实现方式也非常典型：</p>
<p>首先给 Broker 增加了 ProducerID，每个 Producer 初始化时都会赋予一个唯一的 ID 用来表征
 Producer，该字段对客户端不可见。另外通过字段 <code>producer epoch</code> 来表征唯一
消息集合。</p>
<p>在 Broker 实际写入消息之前，通过前边的两个字段从查询当前写入的消息的状态，如果已经写
则直接返回 ack，而不需要重复写入。如果未写入过该消息，先将消息放到缓存中，然后执行写入逻
。</p>
<p>看上去，幂等性 Producer 的功能很酷，使用起来也很简单，仅仅设置一个参数就能保证消息不
复了，但实际上，我们必须要了解幂等性 Producer 的作用范围：</p>

首先，它只能保证单分区上的幂等性，即一个幂等性 Producer 能够保证某个主题的一个分区上
出现重复消息，它无法实现多个分区的幂等性。
其次，它只能实现单会话上的幂等性，不能实现跨会话的幂等性。这里的会话，你可以理解为 Pr
ducer 进程的一次运行。当你重启了 Producer 进程之后，这种幂等性保证就丧失了。</st
ong>

<p>如果我们想要实现多分区多会话上消息无重复，则需要使用 <code>事务（transcation）</cod
> 或者 <code>依赖事务型Producer</code></p>
<h3 id="事务">事务</h3>
<p>Kafka 的事务概念类似于我们熟知的数据库提供的事务。在数据库领域，事务提供的安全性保障
经典的 ACID，即原子性（Atomicity）、一致性 (Consistency)、隔离性 (Isolation) 和持久性 (Durab
lity)。
Kafka 自 0.11 版本开始也提供了对事务的支持，目前主要是在 read committed 隔离级别上做事情
它能保证多条消息原子性地写入到目标分区，同时也能保证 Consumer 只能看到事务成功提交的消息
。</p>
<p>在讲完传输部分内容之后，接下来我们了解一下，消息引擎的另一个重点知识，即消
是如何被保存的。</p>
<h3 id="事务型-Producer">事务型 Producer</h3>
<blockquote>
<p>事务型 Producer 能够保证将消息原子性地写入到多个分区中。这批消息要么全部写入成功，要
全部失败。另外，事务型 Producer 也不惧进程的重启。Producer 重启回来后，Kafka 依然保证它
发送消息的精确一次处理。</p>
</blockquote>
<p>设置事务型 Producer 的方法也很简单，满足两个要求即可：</p>

和幂等性 Producer 一样，开启 enable.idempotence = true。
设置 Producer 端参数 transactional. id。最好为其设置一个有意义的名字。

<p>另外事务型 Producer 和普通的 Producer 也有一些不同：</p>
<pre><code class="language-java highlight-chroma">producer<
span>.initTransactions</sp
n>();
try {
producer.<s
an class="highlight-na">beginTransaction();
producer.<s
an class="highlight-na">send(<span class="high
ight-n">record1);
producer.<s
an class="highlight-na">send(<span class="high
ight-n">record2);
producer.<s
an class="highlight-na">commitTransaction();
} catch <span class=
highlight-o">(KafkaException <span class="high
ight-n">e) {
producer.<s
an class="highlight-na">abortTransaction();
}
</code></pre>
<p>和普通 Producer 代码相比，事务型 Producer 的显著特点是调用了一些事务 API，如 initTransa
tion、beginTransaction、commitTransaction 和 abortTransaction，它们分别对应事务的初始化
事务开始、事务提交以及事务终止。</p>
<p>这段代码能够保证 record1 和 record2 能被当做一个事务统一提交到 Kafka 中，那么要么它们
一提交成功，要么统一提交失败。</p>
<p>:question::question: 通过对幂等性和事务的支持，我们可以保证消息交付的可靠性，消息被发到
Broker 集群之后，消息又是被如何保存的呢？</p>
<h2 id="消息保存">消息保存</h2>
<p>Broker 端获取到消息之后，会按照既定的分区策略（定位消息所要发送的
个分区的算法），确定消息要保存的分区，然后向该分区的主分区写入消息其他

Follower 分区从主分区拉取消息进行同步，这就是整个
息保存的过程。</p>
<p>:question::question: 从上边的描述中也可以看出来，Kafka 生产者在确定消息的时候，是按照
前配置好的来确定消息所要保存的分区，那 Kafka 支持哪些分区策略呢？我们该如何配置呢？</p>
<h2 id="分区策略">分区策略</h2>
<p>对数据进行分区主要作用为了提高系统的高伸缩性，分区之后主
区也会分布在不同的 Broker 上，这也也就提高了系统的负载均衡能力。</p>
<h3 id="自定义分区策略">自定义分区策略</h3>
<p>kafka 默认使用了,如果我们想要更换分区策略的话，需要显式地配置生产者端的参数 <code>par
itioner.class</code>。这个参数该怎么设定呢？方法很简单，在编写生产者程序时，你可以编写一
具体的类实现 o <code>rg.apache.kafka.clients.producer.Partitioner</code> 接口。这个借口定
了两个方法：<code>partition()</code> 和 <code>close</code>()，通常你只需要实现最重要的
partition 方法。我们来看看这个方法的方法签名：</p>
<pre><code class="language-java highlight-chroma">int</spa
> partition(<span cl
ss="highlight-n">String topic<span class="highl
ght-o">, Object key
/span>, byte<span c
ass="highlight-o">[] keyBytes<span class="highl
ght-o">, Object valu
, byte[] valueBytes<span class="h
ghlight-o">, Cluster
luster);
</code></pre>
<p>这里的 topic、key、keyBytes、value 和 valueBytes 都属于消息数据，cluster 则是集群信息
比如当前 Kafka 集群共有多少主题、多少 Broker 等）。通过这些参数，我们自定义分区
略代码，返回分区编号结果。只要你自己的实现类定义好了 partition 方法，同时设置 <c
de>partitioner.class</code> 参数为你自己实现类的 Full Qualified Name
那么生产者程序就会按照你的代码逻辑对消息进行分区。</p>
<p>理论上分区的策略多种多样，但常见的分区策略主要有三种：</p>
<h3 id="1--轮询策略">1. 轮询策略</h3>
<p><code>轮询策略</code> 很容易理解，即顺序分配，比如如果一个主题下游三个分区，那么
一条消息会被发送到 0 号分区，第二条消息会被发送到 1 号分区</st
ong>，第三条消息会被发送到 2 号分区，第四条消息会被发送到 4
分区，以此类推....</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719165632-e9rowb1.png?imageV
ew2/2/interlace/1/format/jpg"></p>
<p>轮询策略有非常优秀的负载均衡表现，它总是能保证消息最大限度地被平均分配到所
分区上，故默认情况下它是最合理的分区策略，也是我们最常用的分区策略之一。</p>
<h3 id="2--随机策略">2. 随机策略</h3>
<p>所谓 <code>随机策略</code> 也就是我们随机的把消息放置到任意分区，如下图所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719165820-tpmy326.png?imageV
ew2/2/interlace/1/format/jpg"></p>
<p>如果想要实现分区策略，也非常简单，只需要 <code>partition()</code> 方法中写入下面两
代码：</p>
<pre><code class="language-java highlight-chroma">List</spa
><PartitionInfo<s
an class="highlight-o">> partitions <span cl
ss="highlight-o">= cluster<span class="highlig
t-o">.partitionsForTopic<span class="highlight
o">(topic);

```
<span class="highlight-k">return</span> <span class="highlight-n">ThreadLocalRandom</
pan><span class="highlight-o">.</span><span class="highlight-na">current</span><span
class="highlight-o">().</span><span class="highlight-na">nextInt</span><span class="hig
light-o">(</span><span class="highlight-n">partitions</span><span class="highlight-o">.
/span><span class="highlight-na">size</span><span class="highlight-o">());</span>
</code></pre>
<p>本质上看随机策略也是力求将数据均匀地打散到各个分区，但从实际表现来看，<strong>它要
于轮询策略</strong>，所以如果追求数据的均匀分布，还是使用轮询策略比较好。</p>
<h3 id="3-按消息键保序策略">3.按消息键保序策略</h3>
<p>Kafka 在存的时候是以键值对的形式存的，因而对于 key 按照业务需求来设置不同的值，比如设
成部门 ID，那我们想要具有<strong>相同 key</strong> 的消息都保存在一个分区中，那么我们就
要该策略。也就是说 <code>按消息键保存策略</code> 是根据消息的 <strong>key</strong>
进行分区的，其示意图如下所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719171314-0t3nco3.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>这种策略其实当我们希望消息有序的时候很常用，通过这种策略，我们可以把相同的 key 的消息
到同一个分区中，这样同一个分区中的消息就天然具有了<strong>有序性</strong>。</p>
<p>其具体实现也非常简单，只需要两行代码：</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-n">List</spa
><span class="highlight-o">&lt;</span><span class="highlight-n">PartitionInfo</span><s
an class="highlight-o">&gt;</span> <span class="highlight-n">partitions</span> <span cl
ss="highlight-o">=</span> <span class="highlight-n">cluster</span><span class="highlig
t-o">.</span><span class="highlight-na">partitionsForTopic</span><span class="highlight
o">(</span><span class="highlight-n">topic</span><span class="highlight-o">);</span>
<span class="highlight-k">return</span> <span class="highlight-n">Math</span><span cl
ss="highlight-o">.</span><span class="highlight-na">abs</span><span class="highlight-o
>(</span><span class="highlight-n">key</span><span class="highlight-o">.</span><span
class="highlight-na">hashCode</span><span class="highlight-o">())</span> <span class=
highlight-o">%</span> <span class="highlight-n">partitions</span><span class="highlight
o">.</span><span class="highlight-na">size</span><span class="highlight-o">();</span>
</code></pre>
<p>确定分区之后，消息便可以指定分区的主分区中写入消息，写入完成之后还不算完，还必须要等
跟随者分区数据同步之后才能进行消息，这就引入 Kafka 的。</p>
<h2 id="副本机制">副本机制</h2>
<p>通常来讲一个系统引入副本会提供下边三个好处：</p>
<ol>
<li>数据冗余，提高数据可用性</li>
<li>提供伸缩性，支持横向扩展，能够通过增加机器的方式来提升读性能，进而提高读操作吞吐量。<
li>
<li>提高数据局部性，允许将数据放入与用户地理位置相近的地方，从而降低系统延时。</li>
</ol>
<p>但由于 Kafka 只允许<strong>领导者副本</strong>对外提供服务，其他副本只是被动的同步
据，因而，Kafka 的副本机制只提供了第一个好处，后边两个好处都没有。但另外两种特性，Kafka
过了<strong>分区</strong>来实现（想一想是为什么？）。</p>
<p>Kafka 的副本机制工作原理如下：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719173153-oeb1nle.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<ol>
<li>第一，在 Kafka 中，副本分成两类：领导者副本（Leader Replica）和追随者副本（Follower R
plica）。<strong>每个分区在创建时都要选举一个副本，称为领导者副本</strong>，其余的副本
动称为追随者副本。</li>
<li>只有领导者副本对外提供服务，追随者副本不处理客户端请求，它唯一的任务就是从领导者副本
```

异步拉取消息，并提交到自己日志中，实现与领导者副本的同步。
当领导者副本挂掉之后，或者领导者副本所在的 Broker 宕机，Kafka 依托 Zookeeper 的监控
能能够实时感知到，进而进行新一轮的选举，从追随者副本中选择新的领导者副本</stron
>。

<p>Kafka 这样设计副本机制有什么好处呢？</p>

方便实现** “Read-your-writes"**
所谓 Read-your-writes，顾名思义就是，当你使用生产者 API 向 Kafka 成功写入消息后，马上使用
费者 API 去读取刚才生产的消息。比如发微博，我们在发布之后，当然希望马上就被看到，但如果副
对外提供服务，则副本需要时间和领导者副本数据进行同步，这样就不能实现 Read-your-
rites.
方便实现单调读（Monotonic Reads）。
什么是单调读呢？就是对于一个消费者用户而言，在多次消费消息时，它不会看到某条消息一会
存在一会儿不存在。
如果允许追随者副本提供读服务，那么假设当前有 2 个追随者副本 F1 和 F2，它们异步地拉取领导者
本数据。倘若 F1 拉取了 Leader 的最新消息而 F2 还未及时拉取，那么，此时如果有一个消费者先从
1 读取消息之后又从 F2 拉取消息，它可能会看到这样的现象：第一次消费时看到的最新消息在第二
消费时不见了，这就不是单调读一致性。但是，如果所有的读请求都是由 Leader 来处理，那么 Kafka
就很容易实现单调读一致性。

<p>另外由于追随者副本不提供服务，只是定期地异步拉取领导者副本中的数据而已。既然是异步的
就存在着不可能与 Leader 实时同步的风险。那 Kafka 是如何解决该问题的呢？</p>
<p>Kafka 引入了 In-sync Replicas，也就是所谓的 ISR 副本集合。ISR 中的副本都是与 Leader 同
的副本，相反，不在 ISR 中的追随者副本就被认为是与 Leader 不同步的。Leader 副本天
就在 ISR 中。也就是说，ISR 不只是追随者副本集合，它必然包括 Leader 副本。甚至在
些情况下，ISR 只有 Leader 这一个副本。想要进入 ISR 需要满足一定的条件。</p>
<p>这个条件就是 Broker 端参数 <code>replica.lag.time.max.ms </code> 参数值。这个参数的
义是 Follower 副本能够落后 Leader 副本的最长时间间隔，当前默认值是 10 秒。这就是说，只要一
 Follower 副本落后 Leader 副本的时间不连续超过 10 秒，那么 Kafka 就认为该 Follower 副本与 Le
der 是同步的，即使此时 Follower 副本中保存的消息明显少于 Leader 副本中的消息。</p>
<p>我们在前面说过，Follower 副本唯一的工作就是不断地从 Leader 副本拉取消息，然后写入到自
的提交日志中。如果这个同步过程的速度持续慢于 Leader 副本的消息写入速度，那么在 <code>repl
ca.lag.time.max.ms </code> 时间后，此 Follower 副本就会被认为是与 Leader 副本不同步的，因
不能再放入 ISR 中。此时，Kafka 会自动收缩 ISR 集合，将该副本“踢出”ISR。倘若该副本后面慢
地追上了 Leader 的进度，那么它是能够重新被加回 ISR 的。这也表明，ISR 是一个动态调
的集合，而非静态不变的。</p>
<p>而领导者副本在选举的时候，默认情况下都是从 ISR 集合的副本中进行选举。</p>
<p>通过副本机制，进行领导者副本和追随者副本数据之间的同步后，输入便保存成功，成功之后，<
trong>消费者就可进行消费了。:smile::smile:</p>
<h2 id="消息消费">消息消费</h2>
<p>就消息引擎而言，消费者支持的消息有两种:</p>

点对点模式:类似于打电话，一对一传输
发布/订阅模式：和点对点模型不同的是，这个模型可能存在多个发布者向相同的主题发送消息
而订阅者也可能存在多个，它们都能接收到相同主题的消息。跟日常生活中订报纸，非常相似。

<p>Kafka 比较酷，它这两种方式都支持，那它是怎么实现对这两种消息消费
式都支持的呢？</p>
<p>那就不得不提 <code>消费者组</code> 这一 Kafka 非常有亮点的设计，什么是消费者组呢？
一句话概括就是：<code>Consumer Group</code> 是 Kafka 提供的可供扩展的具有容错性的消
者机制。
一个消费者组内包含多个消费者，它们共享同一个消费者组 ID，称之为 Group ID，组内所有消费者
调在一起消费订阅主题的所有分区，每个分区只能由同一个消费者组内的一个 Consumer 实例来消

，具体来说其重要的特性有三个：</p>

Consumer Group 下可以有一个或者多个 Consumer 实例，这里的实例是一个单独的进程，也
以是同一个进程下的线程，在实际场景中，使用进程更为常见。
Group ID 是一个字符串，在一个 Kafka 集群中，它标识唯一的一个 Consumer Group
Consumer Group 下所有实例订阅的的主题的单个分区，只能分配给该消费者组内 Consumer
例消费，同时这个分区也可以被其他消费者组消费。

<p>基于消费者组这些特性，非常容易便实现了对两种消息消费方式的支持，具体来说：</p>
<p>由于消费者组订阅了某个主题，消费者组的实例只消费组内的某个分区，因而在实现两种模型看
时候呢，当所有实例都在一个消费者组，那它就是一个消息队列模型。如下图
示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719202424-ucy4b5c.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>当所有的实例都在不同的消费者组的时候，那就是发布/订阅模型 ：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719202957-nvcctmk.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>在了解了 Consumer Group 以及它的设计亮点之后，你可能会有这样的疑问：在实际使用场景
，我怎么知道一个 Group 下该有多少个 Consumer 实例呢？**理想情况下，Consumer 实例的数量
该等于该 Group 订阅主题的分区总数。**以下图为例：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210719203754-ogi9t22.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>图中分区共有三个，那么消费者组在设置成三个的情况下，可以恰好实现一个消费者
例消费一个分区这一最完美的情形，如果设置成两个消费者，则会出现一个消费者消费两
分区，一个消费者实例消费一个分区；如果设置成四个消费者实例，则会出现一个空闲的消费者实例
消费分区，因而在实际使用时不要设置成消费者实例数目大于订阅分区的情形，这样只能白白浪费资
。</p>
<p>:question::question: 一个消费者组在消费的过程中，也不可能是**“一帆风顺”**的，可能会
生各种各样的意外，比如某些消费者实例挂掉了，消费主题变更了等等这些意料之外的情况，<stron
>遇到这种情况消费者组该怎么办呢？</p>
<p>这就说道了消费者组中的另一个重要概念 <code>重平衡（Rebalance）</code>，Rebalance
本质上是一种协议**，规定了一个 Consumer Group 下的所有 Consumer 如何达成一致，来分配订
 Topic 的每个分区**。比如某个 Group 下有 20 个 Consumer 实例，它订阅了一个具有 100 个分区
 Topic。正常情况下，Kafka 平均会为每个 Consumer 分配 5 个分区。这个分配的过程就叫 Rebalan
e。</p>
<p>消费组在消费消息的过程中，如果发生一些“意外”，就会触发重平衡，
体来说，重平衡的触发条件有三个：</p>

组成员数发生变更，比如有新的 Consumer 加入或者离开
订阅主题数发生变更。Consumer Group 可以使用正则表达式的方式订阅主题，当匹配的主题
目变多时，也会触发重平衡问题
订阅主题的分区数发生变更。当前 kafka 只允许增加一个主题分区。

<p>虽然说通过重平衡可以合理分配消费者组中的消费者和主题的关系，实现一种均衡</st
ong>，但它在重平衡的过程，，所有 Consumer 实例都会停止消费，等待 Re
alance 完成，相当于 JVM 中的 Stop the World，并且这一过程十分之慢。此
其效率不高，当前 Kafka 的设计机制决定了每次 Rebalance 时，Group 下的所有成员都要参与进来
而且通常不会考虑局部性原理，但局部性原理对提升系统性能是特别重要的。</p>
<p>因而在实际开发过程中，我们是要尽量避免重平衡情况的发生，如何避免呢？</stron
></p>
<p>结合前边重平衡触发的条件，其中前两种不可避免，最后一种可以通过一些措施来优

：</p>
<p>基本思想就是，规避不必要的 Reblance：</p>

因为心跳时间短，未来得及发送心跳实例心跳，导致被移除组
poll.timeout.ms 过小，一些执行较长时间的消费者会被移除。

<p>针对这两种原因，解决方法就是提高超时时间，这里给出参考值：</p>

设置 session.timeout.ms = 6s。
设置 heartbeat.interval.ms = 2s。
要保证 Consumer 实例在被判定为“dead”之前，能够发送至少 3 轮的心跳请求，即 <strong
session.timeout.ms >= 3 * heartbeat.interval.ms。

<p>第二类非必要 Reblance**，需要根据具体业务场景来设置合理的超时时间**，比如如果消费者
费单条时间很长，那么在设置超时时间的时候要保证超时时间内，消费者把数据消费完。</p>
<p>:question::question: 为什么重平衡会发生这么多问题呢？这就要我们结合重平衡的流程来理解
。</p>
<h2 id="重平衡问题">重平衡问题</h2>
<p>首先我们弄清楚一个问题，消费者组和 Broker 端是如何知道该重平衡的呢？</p>
<p>答案就是，靠消费者端的 <code>心跳线程（Heartbeat Thread）</code>。心跳线程，顾名
义，就是用来发送心跳来告知 Broker 端消费者自身的存活情况的，但在 Kafka 中重平衡机制也需要
跳线程来协助完成。
当协调者决定开启新一轮的重平衡后，它会将“<code>REBALANCE_IN_PROGRESS</code>”封
到心跳请求的响应中，发给消费者实例。当消费者实例发现心跳响应中包含了“<code>REBALANCE_
N_PROGRESS</code>”，就能马上知道重平衡开始了，这就是重平衡的一个通知机制</s
rong>。</p>
<p>当重平衡流程启动之后，需要消费者端和 Broker 端两端协调进行工作。下边我们分别展开来说<
p>
<h3 id="消费者端重平衡流程">消费者端重平衡流程</h3>
<p>针对消费者组来说，Kafka 当前设计了一套状态机，来辅助解决重平衡问
。Kafka 为消费者组定义了 5 种状态，它们分别是：<code>Empty</code>、<code>Dead</cod
>、<code>PreparingRebalance</code>、<code>CompletingRebalance </code> 和 <code>
table</code>，其具体含义如下图所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720115846-kzr8cur.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>各个状态的转移图如下所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720120253-4bjjpnx.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>消费者组一开始会处于 Empty 状态，当重平衡开始之后，它会被置为 <code>PreparingRebala
ce</code> 状态等待成员加入，成员加入后变更为 <code>CompletingRebalance</code> 状态
待 <code>消费者Leader</code> 对消费者组中各个消费者需要消费的分区进行划分，分配完成之
进入 <code>Stable</code> 完成重平衡。
当有新成员加入或者已有成员退出时，消费者组会从 <code>Stable</code> 直接跳到 <code>Pre
aringRebalance状态</code>，此时所有现存成员必须申请入组，当所有成员都退组之后，消费者
状态变更程 <code>Empty</code>，开始重平衡流程。</p>
<p>在讲完消费者组重平衡过程中状态的变化之后，接下我们把目光聚焦到消费者实例本身所发生的
一些事情。</p>
<p>Kafka 定期删除过期位移的条件就是，消费者组要处于状态，此时如果消费者组如果停掉了很长
间（超过 7 天），那么 Kafka 很可能就把该组的位移数据删除掉。</p>
<p>在消费者端，重平衡分为两个步骤：分别是加入组和等待领导者消费者（Leader Consumer）
配方案。这两个步骤分别对应两类特定的请求：JoinGroup 请求和 SyncGroup 请求。</p>
<p>当组内成员加入组时，它会向协调者发送 JoinGroup 请求。在该请求中，每个成员都要将自己

阅的主题上报，这样协调者就能收集到所有成员的订阅信息。一旦收集了全部成员的 JoinGroup 请
后，协调者会从这些成员中选择一个担任这个消费者组的领导者。通常情况下，第一个发送
oinGroup 请求的成员自动成为领导者。你一定要注意区分这里的领导者和之前我们介绍
领导者副本，它们不是一个概念。这里的领导者是具体的消费者实例，它既不是副本，也不是协调者
领导者消费者的任务是收集所有成员的订阅信息，然后根据这些信息，制定具体的分区消
分配方案。</p>
<p>选出领导者之后，协调者会把消费者组订阅信息封装到 JoinGroup 请求的响应体中，然后发给
导者，由领导者统一作出分配方案，进入下一步：发送 SyncGroup 请求。</p>
<p>在这一步中，领导者向协调者发送 SyncGroup 请求，将刚刚做出的分配方案发给协调者。值得
意的是，其他成员也会向协调者发送 SyncGroup 请求，只不过请求体中并没有实际的内容。这一步
主要目的是让协调者接收分配方案，然后统一以 SyncGroup 响应的方式分发给所有成员，这样组内
有成员就都知道自己该消费哪些分区了。</p>
<p>下面这张图描述的是 JoinGroup 请求的处理流程。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720130924-ctbqi9s.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>JoinGroup 请求的主要作用是将组成员订阅信息发送给领导者消费者，待领导者制定好分配方案
，重平衡流程进入到 SyncGroup 请求阶段。</p>
<p>下图描述的是 SyncGroup 请求的处理流程：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720131116-8edd6l9.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>SyncGroup 请求的主要目的，就是让协调者把领导者制定的分配方案下发给各个组内成员。当
有成员都成功接收到分配方案后，消费者组进入到 Stable 状态，即开始正常的消费工作</s
rong>。</p>
<h3 id="Broker端重平衡流程">Broker 端重平衡流程</h3>
<p>Broker 端重平衡则需要分四个场景进行讨论（实际上领导者消费者是负责提出解决方案的，下
流程图为了便于理解并未区分领导着消费者和普通消费者）：</p>
<p>场景一：新成员入组。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720131524-jow0svz.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>场景二：组成员主动离组。</p>
<p>何谓主动离组？就是指消费者实例所在线程或进程调用 close() 方法主动通知协调者它要退出。
个场景就涉及到了第三类请求：LeaveGroup 请求。协调者收到 LeaveGroup 请求后，依然会以心跳
应的方式通知其他成员，因此我就不再赘述了，还是直接用一张图来说明。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720131614-zk3y7lh.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>场景三：组成员崩溃离组（崩溃离组是指消费者实例出现严重故
，突然宕机导致的离组）</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720131838-og3h0c7.png?imageV
ew2/2/interlace/1/format/jpg"></p>
<p>场景四：重平衡时协调者对组内成员提交位移的处理。</p>
<p>当重平衡开启时，协调者会给予成员一段缓冲时间，要求每个成员必须在这段时间内快速地上报
己的位移信息，然后再开启正常的 JoinGroup/SyncGroup 请求发送。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210720132025-lfdpkxu.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<h2 id="总结">总结</h2>
<p>本文主要以消息引擎的角度出发，介绍 Kafka 的设计原理，整篇文章分成四个大篇幅：</p>
<p>一：Kafka 消息格式以及演变过程，并且对消息的压缩也做了介绍</p>
<p>二：Kafka 实现消息传输的方式，包含了生产者和消费者如何建立连接，如果保障消息的传输的

靠性</p>
<p>三：消息的保存方法，即 Kafka 是如何通过分区和副本机制来保证 kafka 的高可用</p>
<p>四：消费者组以及重平衡问题即重平衡发生的过程，触发条件，以及可参考的解决方法</p>
<h2 id="参考">参考</h2>

《Kafka 消息规范》
《Kafka 消息格式演进》
《一文看懂 Kaka 消息格式的演变》
https://time.geekbang.org/olumn/article/99318