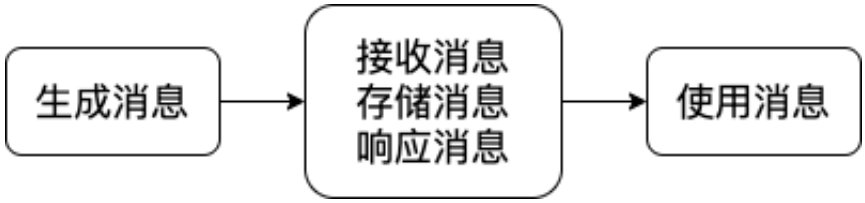
Kafka

kafka 使用场景

1. 活动跟踪，例如用户点击网站，根据用户行为生成活动报告或内容推荐
2. 传递消息，基本功能，几乎所有应用场景都是传递消息
3. 度量指标收集
4. 日志收集
5. 流处理

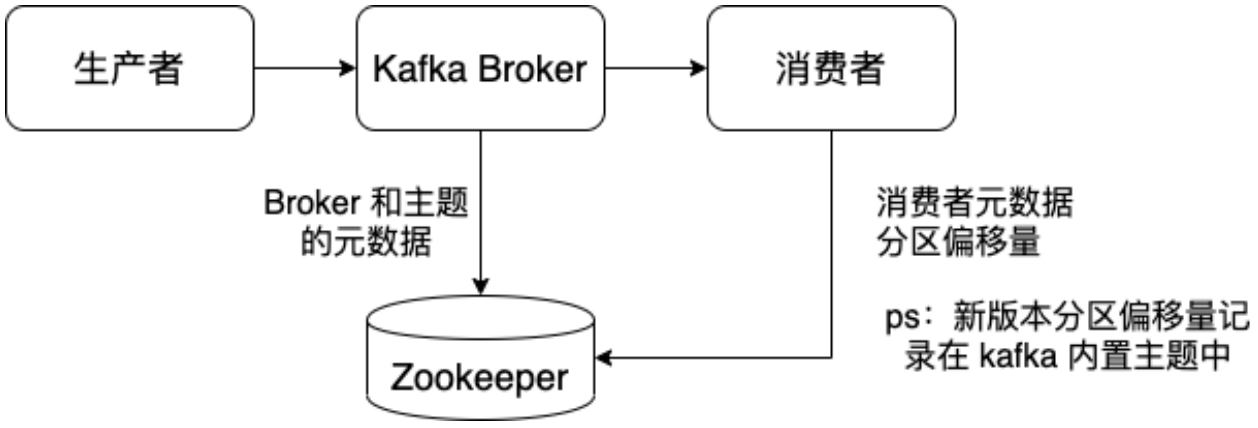


说明：

1. 这些使用场景的共同特征是：都有产生数据的地方，也都有使用数据的地方，因此要求 kafka 具备接收数据，存储数据，对数据请求进行响应的功能
2. kafka 具备接收消息，存储消息，响应消息的功能

kafka 总览

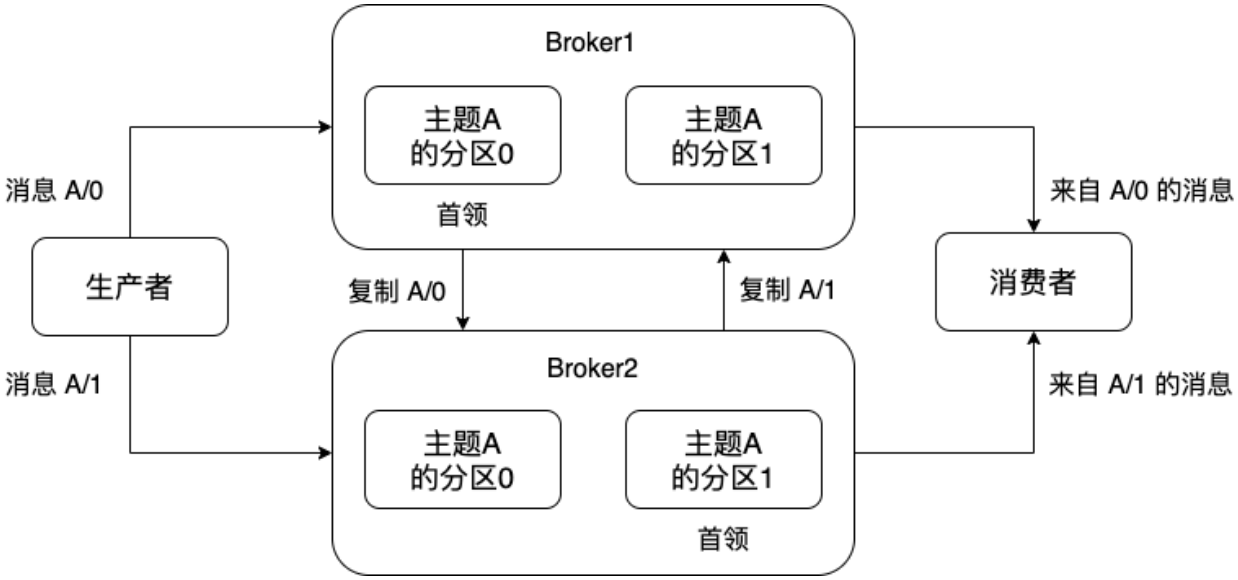
kafka 基本组成元素



1. 一个 Kafka 集群由多个 Broker 组成，一个 Broker 在逻辑上为在一台服务端机器上运行的进程。Broker 是 Kafka 集群中的服务端节点，负责存储数据、协调分区副本、管理消费者群组、处理客户端请求等工作。
2. Kafka 集群通过 Zookeeper 维护集群节点，进行控制器选举以及集群配置管理等活动，是 Kakfa 的基础依赖。

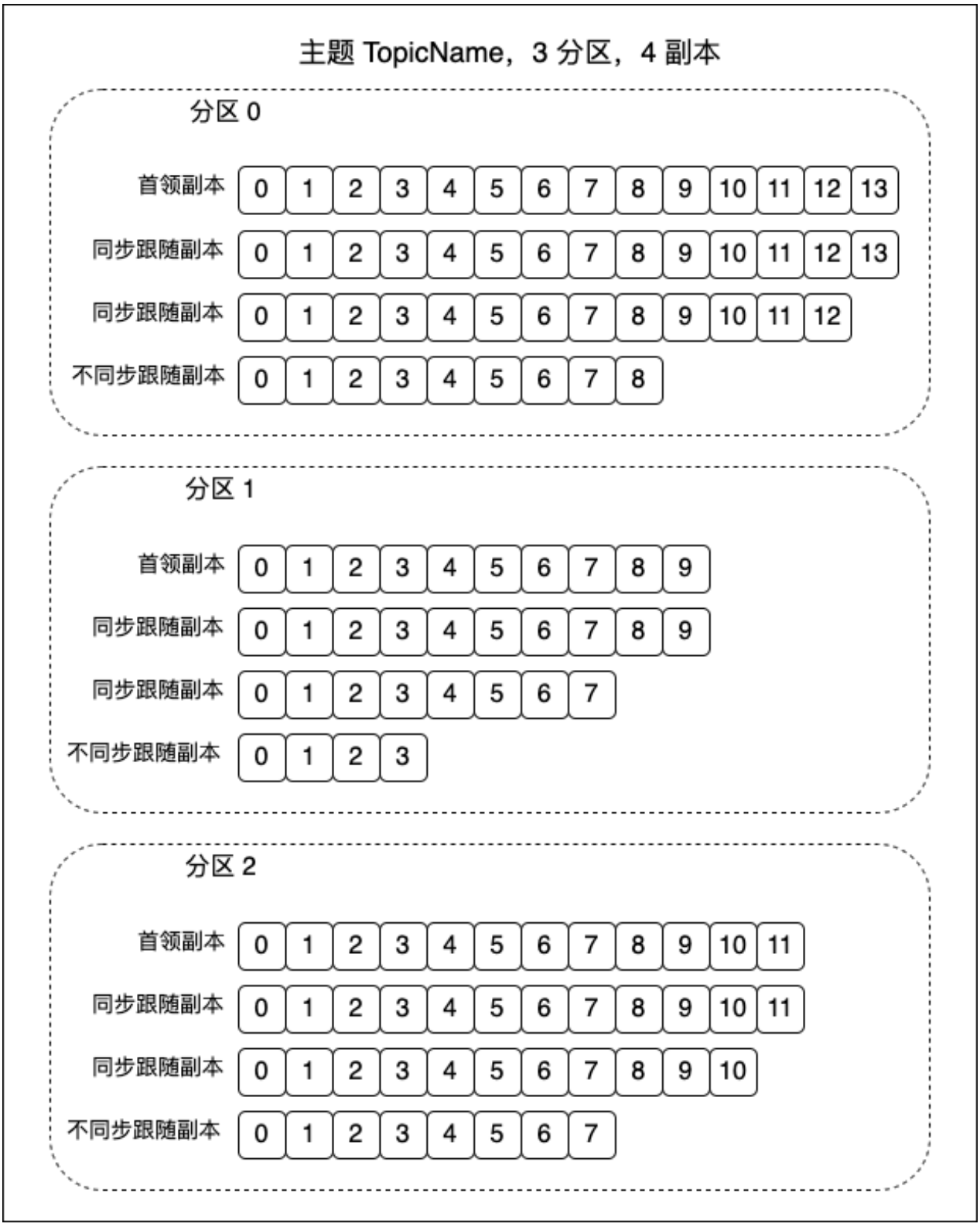
3. Client 为开发人员使用的客户端，不属于 Kafka 集群的节点，但是是组成 Kafka 集群的重要角色，分为生产者与消费者两种类型。
    - Producer：生产者是集群中生产数据的地方，通过 push 的方式将数据推送到集群服务端存储。
    - Consumer：消费者是集群中消费数据的地方，通过 pull 的方式将数据从集群服务器中拉取并消费。

kafka 功能说明

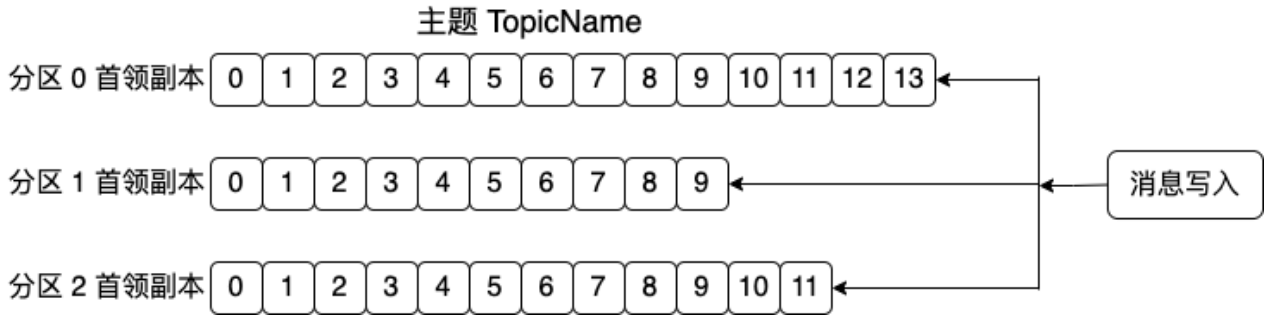


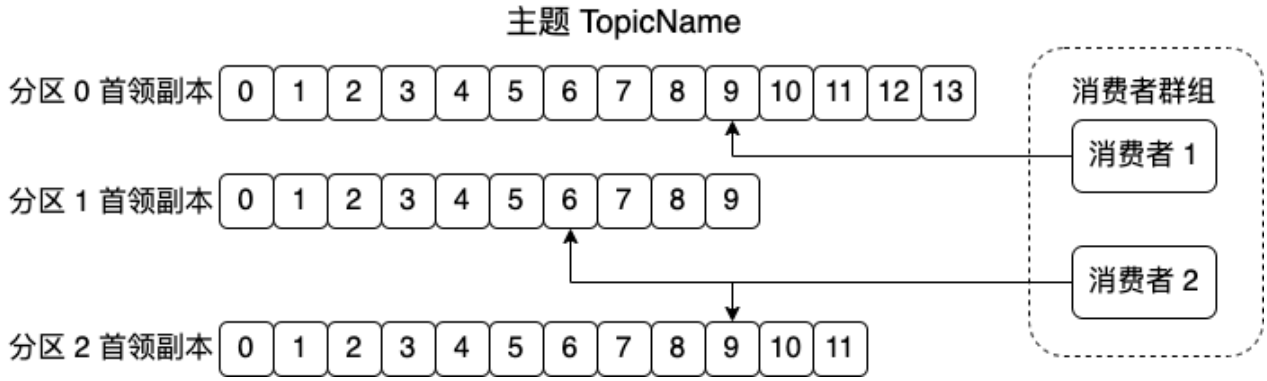
kafka 是如何进行接收消息，存储消息，响应消息的?

1. kafka 通过多个 **broker** 组成高可用集群，接收从生产者发送过来的消息，响应消费者获取消息的请求。
2. kafka 通过**主题**对消息进行分类，不同类型，不同功能的消息发送到不同的主题上。使 kafka 成为通用的消息传递组件。
3. kafka 将主题分为若干个**分区**，一个分区就是一个提交日志，kafka 通过分区来实现伸缩性。
4. 分区可以有多个**副本**，kafka 通过副本实现数据高可用。副本分为**首领副本**和**跟随副本**，跟随副本有同步和不同步两种状态。



5. 分区的所有副本按照一定的规则被分布在不同的 broker 中。
6. 每个分区只有首领副本会响应客户端（生产者和消费者）请求。





Broker 配置

1. **broker.id**

每个 broker 都需要有一个标识符，使用 broker.id 来表示。它的默认值是 0，也可以被设置成其他任意整数。这个值在整个 Kafka 集群里必须是唯一的。这个值可以任意选定，如果出于维护的需要，可以在服务器节点间交换使用这些 ID。建议把它们设置成与机器名具有相关性的整数，这样在进行维护时，将 ID 号映射到机器名就没那么麻烦了。例如，如果机器名包含唯一性的数字(比如host1.example.com、host2.example.com)， 那么用这些数字来设置 broker.id 就再好不过了。

---

2. **port**

如果使用配置样本来启动 Kafka，它会监听 9092 端口。修改 port 配置参数可以把它设置成其他任意可用的端口。要注意，如果使用 1024 以下的端口，需要使用 root 权限启动 Kafka，不过不建议这么做。

---

3. **zookeeper.connect**

用于保存 broker 元数据的 Zookeeper 地址是通过 zookeeper.connect 来指定的。localhost:2181 表示这个 Zookeeper 是运行在本地的 2181 端口上。该配置参数是用冒号分隔的一组 hostname:port/path 列表，每一部分的含义如下：

- hostname 是 Zookeeper 服务器的机器名或 IP 地址；
- port 是 Zookeeper 的客户端连接端口；
- /path 是可选的 Zookeeper 路径，作为 Kafka 集群的 chroot 环境。如果不指定，默认使用根路径。如果指定的 chroot 路径不存在，broker 会在启动的时候创建它。

---

4. **log.dirs**

Kafka 把所有消息都保存在磁盘上，存放这些日志片段的目录是通过 log.dirs 指定的。它是一组用逗号分隔的本地文件系统路径。如果指定了多个路径，那么 broker 会根据“最少使用”原则，把同一个分区的日志片段保存到同一个路径下。要注意，broker 会往拥有最少数目分区的路径新增分区，而不是往拥有最小磁盘空间的路径新增分区。

---

5. **num.recovery.threads.per.data.dir**

对于如下 3 种情况，Kafka 会使用可配置的线程池来处理日志片段：

- 服务器正常启动，用于打开每个分区的日志片段；
- 服务器崩溃后重启，用于检查和截短每个分区的日志片段
- 服务器正常关闭，用于关闭日志片段。

默认情况下，每个日志目录只使用一个线程。因为这些线程只是在服务器启动和关闭时会用到，所以完全可以设置大量的线程来达到并行操作的目的。特别是对于包含大量分区的服务器来说，一旦发生崩溃，在进行恢复时使用并行操作可能会省下数小时的时间。设置此参数时需要注意，所配置的数字对应的是 log.dirs 指定的单个日志目录。也就是说，如果 num.recovery.threads.per.data.dir 被设为 8，并且 log.dir 指定了 3 个路径，那么总共需要 24 个线程。

说明：

1. 服务器正常启动和正常关闭也许会花很长的时间，因为服务器的内部线程会对日志片段进行相关的处理
2. 服务器崩溃后重启，线程会对日志片段做什么检查，如何判断这些检查是否正常，崩溃后无法重启也许和这些检查不通过有关系
3. 如何干预这些线程的执行，有哪些参数可以设置，如何查看线程的执行结果等

---

6. **auto.create.topics.enable**

默认情况下，Kafka 会在如下几种情形下自动创建主题：

- 当一个生产者开始往主题写入消息时；
- 当一个消费者开始从主题读取消息时；
- 当任意一个客户端向主题发送元数据请求时。

很多时候，这些行为都是非预期的。而且，根据 Kafka 协议，如果一个主题不先被创建，根本无法知道它是否已经存在。如果显式地创建主题，不管是手动创建还是通过其他配置系统来创建，都可以把 auto.create.topics.enable 设为 false。该功能默认是启用的。

说明：

1. 客户端有很多种：生产者，消费者，集群中的其他 Broker，broker 也是 Zookeeper 的客户端
2. 在客户端执行相关操作时，需要获取元数据信息，以便确定如何执行相关操作
3. 客户端会缓存元数据信息，因此需要配置一定的内存空间缓存这些数据，并且需要配置一些参数去更新元数据信息
4. 元数据信息有哪些
5. 元数据信息一部分存储在 Zookeeper 中，一部分存储在内部主题中

主题默认配置

1. **num.partitions**

num.partitions 参数指定了新创建的主题将包含多少个分区。如果启用了主题自动创建功能(该功能默认是启用的)，主题分区的个数就是该参数指定的值。该参数的默认值是 1。要注意，我们可以增加主题分区的个数，但不能减少分区的个数。所以，如果要让一个主题的分区个数少于 nun.partitions 指定的值，需要手动创建该主题。Kafka 集群通过分区对主题进行横向扩展，所以当有新的 broker 加入集群时，可以通过分区个数来实现集群的负载均衡。当然， 这并不是说，在存在多个主题的情况下(它们分布在多个 broker 上)， 为了能让分区分布到所有 broker 上，主题分区的个数必须要大于 broker 的个数。不过，拥有大量消息的主题如果要进行负载分散，就需要大量的分区。

说明：

1. 如何选定主题的分区数
2. 主题的分区和副本如何分布在不同的 broker 上
3. 如果消息是根据 key 或者自定义的分区器写入分区的，那么新增分区可能不起作用甚至会导致错误
4. 单个 broker 对分区的个数是有限制的，分区个数越多，占用的内存，磁盘，带宽也就越多，完成首领选举的时间也越长
5. 如何观测每个分区的吞吐量，如何观察每个分区的大小
6. 分区的个数理论上是主题的吞吐量除以消费者的吞吐量
7. 理论上分区的大小不要超过 25G，超过 25G 后操作磁盘就从顺序 IO 变成了随机 IO
8. 顺序 IO 是 kafka 实现高性能的一个因素

---

2. **log.retention.ms**

Kafka 通常根据时间来决定数据可以被保留多久。默认使用 log.retention.ms 参数来配置时间，默认值为 168 小时，也就是一周。除此以外，还有其他两个参数 log.retention.minutes 和 log.retention.hours。这 3 个参数的作用是一样的，都是决定消息多久以后会被删除，不过还是推荐使用 log.retention.ms。如果指定了不止一个参数，Kafka 会优先使用具有最小值的那个参数。

说明：

1. 根据时间保留数据是通过检查磁盘上日志片段文件的最后修改时间来实现的。一般来说，最后修改时间指的就是日志片段的关闭时间，也就是文件里最后一个消息的时间戳。不过，如果使用管理工具在服务器间移动分区，最后修改时间就不准确了。时间误差可能导致这些分区过多地保留数据。
2. 根据时间保留数据，当保留时间达到后，是否会删除这个日志片段文件中的所有数据。如果删除显然是不合理的。因此每个消息需要导游时间戳信息，表示该条消息是何时到达 broker 的，新版本实现了这个功能。
3. 时间戳功能还允许根据时间重置偏移量。

---

3. **log.retention.bytes**

另一种方式是通过保留的消息字节数来判断消息是否过期。它的值通过参数 log.retention.bytes 来指定，作用在每一个分区上。也就是说，如果有一个包含 8 个分区的主题，并且 log.retention.bytes 被设为 1GB，那么这个主题最多可以保留 8GB 的数据。所以，当主题的分区个数增加时，整个主题可以保留的数据也随之增加。

如果同时指定了 log.retention.bytes 和 log.retention.ms，只要任意一个条件得到满足，消息就会被删除。例如，假设 log.retention.ms 设置为 86 400 000 (也就是 1 天)，log.retention.bytes 设置为 1000 000000 (也就是 1GB)，如果消息字节总数在不到一天的时间就超过了 1GB，那么多出来的部分就会被删除。相反，如果消息字节总数小于 1GB，那么一天之后这些消息也会被删除，尽管分区的数据总量小于 1GB。

4. **log.segment.bytes**

以上的设置都作用在日志片段上，而不是作用在单个消息上。当消息到达 broker 时，它们被追加到分区的当前日志片段上。当日志片段大小达到 log.segment.bytes 指定的上限(默认是 1 GB)时，当前日志片段就会被关闭，一个新的日志片段被打开。如果一个日志片段被关闭，就开始等待过期。这个参数的值越小，就会越频繁地关闭和分配新文件，从而降低磁盘写入的整体效率。

如果主题的消息量不大，那么如何调整这个参数的大小就变得尤为重要。例如，如果一个主题每天只接收 100MB 的消息，而 log.segment.bytes 使用默认设置，那么需要 10 天时间才能填满一个日志片段。因为在日志片段被关闭之前消息是不会过期的，所以如果 log.retention.ms 被设为 604 800 000 (也就是1周)，那么日志片段最多需要 17 天才会过期。这是因为关闭日志片段需要 10 天的时间，而根据配置的过期时间，还需要再保留 7 天时间(要等到日志片段里的最后一个消息过期才能被删除)。

日志片段的大小会影响使用时间戳获取偏移量。在使用时间戳获取日志偏移量时，Kafka 会检查分区里最后修改时间大于指定时间戳的日志片段(已经被关闭的)，该日志片段的前一个文件的最后修改时间小于指定时间戳。然后，Kafka 返回该日志片段(也就是文件名)开头的偏移量。对于使用时间戳获取偏移量的操作来说，日志片段越小，结果越准确。

---

5. **log.segment.ms**

另一个可以控制日志片段关闭时间的参数是 log.segment.ms，它指定了多长时间之后日志片段会被关闭。就像 log.retention.bytes 和 log.retention.ms 这两个参数一样，log.segment.bytes 和 log.segment.ms 这两个参数之间也不存在互斥问题。日志片段会在大小或时间达到上限时被关闭，就看哪个条件先得到满足。默认情况下，log.segment.ms 没有设定值，所以只根据大小来关闭日志片段。

在使用基于时间的日志片段时，要着重考虑并行关闭多个日志片段对磁盘性能的影响。如果多个分区的日志片段永远不能达到大小的上限，就会发生这种情况，因为 broker 在启动之后就开始计算日志片段的过期时间，对于那些数据量小的分区来说，日志片段的关闭操作总是同时发生。

---

6. **message.max.bytes**

broker 通过设置 message.max.bytes 参数来限制单个消息的大小，默认值是 1 000 000，也就是 1MB。如果生产者尝试发送的消息超过这个大小，不仅消息不会被接收，还会收到 broker 返回的错误信息。跟其他与字节相关的配置参数一样，该参数指的是压缩后的消息大小，也就是说，只要压缩后的消息小于 message.max.bytes 指定的值，消息的实际大小可以远大于这个值。

这个值对性能有显著的影响。值越大，那么负责处理网络连接和请求的线程就需要花越多的时间来处理这些请求。它还会增加磁盘写入块的大小，从而影响 IO 吞吐量。

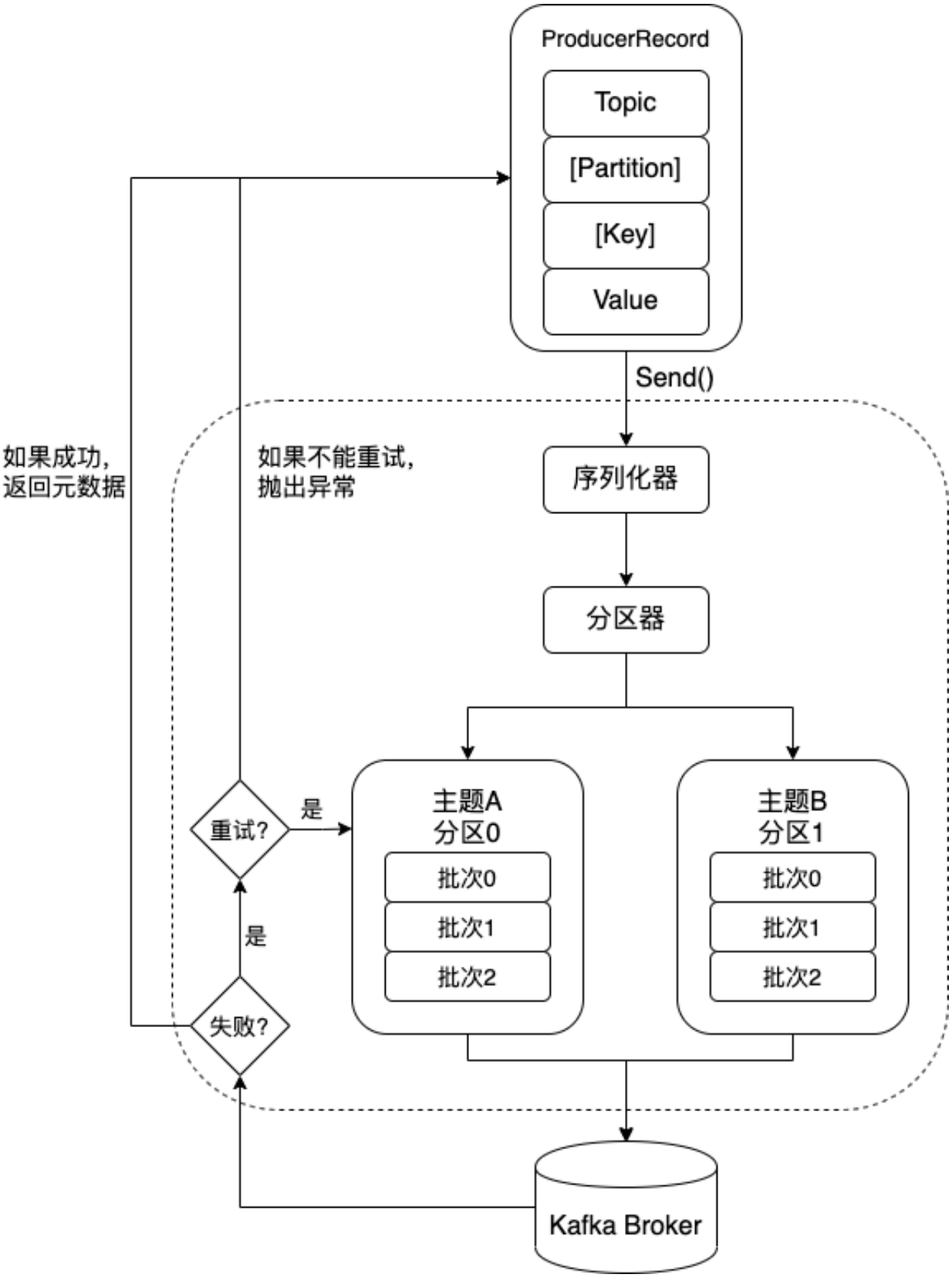
在服务端和客户端之间协调消息大小的配置，消费者客户端设置的 fetch.message.max.bytes 必须与服务器端设置的消息 大小进行协调。如果这个值比 message.max.bytes 小，那么消费者就无法读取比较大的消息，导致出现消费者被阻塞的情况。在为集群里的 broker 配置 replica.fetch.max.bytes 参数时，也遵循同样的原则。

生产者

producer 的首要功能就是向某个 topic 的某个分区发送一条消息，要想实现这个功能，需要解决如下问题：

1. 消息到底是什么，怎么组成一条消息
2. 到底要向哪个 topic 的哪个分区写入消息
3. 寻找这个分区对应的 leader，也就是该分区首领副本所在的 Kafka broker
4. 网络是如何传输的
5. broker 接收到消息后如何存储，存储的格式
6. 消费者获取消息，broker 如何判断哪些消息消费者已经获取

producer 实现过程如下所示：



消息组成

1. 目标主题
2. 目标分区
3. key
4. Value，消息本身
5. Timestamp（新版本中加入的功能）

主题和分区

到底要向哪个 topic 的哪个分区写入消息：

1. 创建一个 `ProducerRecord` 对象，代表即将要发送的消息，ProducerRecord 对象包含了`目标主题`和要`发送的内容`，这样就确定了向哪个主题发送消息。
2. 对于如何确定分区
    1. 目标分区
    2. key hash
    3. 自定义分区器
    4. 默认分区器（将消息均匀发送到各个分区）

元数据

寻找目标分区对应的 leader 所在的 broker

1. kafka 集群内部需要进行分区副本首领选举
2. 将选举结构存储在 Zookeeper 中
3. 生产者请求元数据（分区 leader 副本所在的 Kafka broker）
4. 生产者缓存元数据，并按照一定规则重新请求进行更新

网络传输

1. 序列化
    1. 默认序列化器和自定义序列化器
    2. 序列化 key 和 value
    3. 消费者反序列化
2. Kafka 自定义二进制传输协议和零复制技术
3. 批次
    1. 批次的大小，指定内存的缓存空间
    2. 批次发送的频率
    3. 压缩
4. 发送数据失败的重试机制

代码示例

向 Kafka 发送数据，从创建一个 `ProducerRecord` 对象开始，ProducerRecord 对象需要包含`目标主题`和要`发送的内容`。还可以指定`键`或`分区`。在发送 ProducerRecord 对象时，生产者要先把键和值对象`序列化`成字节数组，这样它们才能够在网络上传输。

接下来，数据被传给`分区器`。如果之前在 ProducerRecord 对象里指定了分区，那么分区器就不会再做任何事情，直接把指定的分区返回。如果没有指定分区，那么分区器会根据 ProducerRecord 对象的键来选择一个分区。选好分区以后，生产者就知道该往哪个主题和分区发送这条记录了。

紧接着，这条记录被添加到一个`记录批次`里，这个批次里的所有消息会被发送到相同的主题和分区上。有一个独立的线程负责把这些记录批次发送到相应的 broker 上。

服务器在收到这些消息时会返回一个响应。如果消息成功写入 Kafka，就返回一个 `RecordMetaData` 对象，它包含了主题和分区信息，以及记录在分区里的偏移量。如果写入失败，则会返回一个错误。生产者在收到错误之后会尝试重新发送消息，几次之后如果还是失败，就返回错误信息。

```
Properties kafkaProps = new Properties();
// 生产者配置
kafkaProps.put("bootstrap.servers", "");
kafkaProps.put("key.serializer", "");
kafkaProps.put("value.serializer", "");

// 根据配置创建生产者对象
KafkaProducer producer = new KafkaProducer(kafkaProps);

// 创建消息对象，指定 "topic", "key", "value"
ProducerRecord record = new ProducerRecord("topic", "key", "value");
try {
    // 使用 send() 方法向 broker 发送消息
    producer.send(record);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 最后关闭生产者对象
    producer.close()
}
```

说明：

1. Send() 方法在内部获取了元数据，并且处理了网络请求相关的细节。
2. 代码中没有对 send() 的返回结果进行处理。

对 send() 返回结果有如下几种处理方式：

1. **同步发送**

```
// 创建消息对象，指定 "topic", "key", "value"
ProducerRecord record = new ProducerRecord("topic", "key", "value");
try {
    // 使用 send() 方法向 broker 发送消息
    // 同步发送数据，阻塞线程直到 broker 返回结果或者抛出异常
    producer.send(record).get();
} catch (Exception e) {
    e.printStackTrace();
}
```

2. **异步发送**

```
// 定义消息发送成功时的回调
```

```
private class DemoProducerCallback implements callback {
    public void onCompletion(RecordMetadata recordMetadata, Exception e){
        // recordMetadata 中包含了消息在分区中的偏移量
    }
}

// 创建消息对象，指定 "topic", "key", "value"
ProducerRecord record = new ProducerRecord("topic", "key", "value");
try {
    // 使用 send() 方法向 broker 发送消息
    // 异步发送数据
    producer.send(record, new DemoProducerCallback());
} catch (Exception e) {
    e.printStackTrace();
}
```

生产者配置

1. **acks**

acks 参数指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入是成功的。这个参数对消息丢失的可能性有重要影响。该参数有如下选项。

- 如果 acks=0，生产者在成功写入消息之前不会等待任何来自服务器的响应。也就是说， 如果当中出现了问题，导致服务器没有收到消息，那么生产者就无从得知，消息也就丢失了。不过，因为生产者不需要等待服务器的响应，所以它可以以网络能够支持的最大速度发送消息，从而达到很高的吞吐量。
- 如果 acks=1，只要集群的首领副本收到消息，生产者就会收到一个来自服务器的成功响应。如果消息无法到达首领副本（比如首领节点崩溃，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息。不过，如果一个没有收到消息的节点成为新首领，消息还是会丢失（这种情况主要是元数据更新不及时造成的）。这个时候的吞吐量取决于使用的是同步发送还是异步发送。如果让发送客户端等待服务器的响应（通过调用 Future 对象的 get() 方法），显然会增加延迟（在网络上传输一个来回的延迟)。如果客户端使用回调，延迟问题就可以得到缓解，不过吞吐量还是会受发送中消息数量的限制（比如，生产者在收到服务器响应之前可以发送多少个消息)。
- 如果 acks=all，只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。这种模式是最安全的，它可以保证不止一个服务器收到消息，就算有服务器发生崩溃，整个集群仍然可以运行。不过，它的延迟比 acks=1 时更高，因为我们要等待不只一个服务器节点接收消息。

---

2. **buffer.memory**

该参数用来设置生产者内存缓冲区的大小，生产者用它缓冲要发送到服务器的消息。如果应用程序发送消息的速度超过发送到服务器的速度，会导致生产者空间不足。这个时候， send() 方法调用要么被阻塞，要么抛出异常，取决于如何设置 `block.on.buffer.full` 参数 (在 0.9.0.0 版本里被替换成了 `max.block.ms`，表示在抛出异常之前可以阻塞一段时间）。

---

3. **compression.type**

默认情况下，消息发送时不会被压缩。该参数可以设置为 `snappy`、`gzip`或 `lz4`，它指定了消息被发送给 broker 之前使用哪一种压缩算法进行压缩。使用压缩可以降低网络传输开销和存储开销，而这往往是向 Kafka 发送消息的瓶颈所在。

---

4. **retries**

生产者从服务器收到的错误有可能是临时性的错误（比如分区找不到首领）。在这种情况 下，retries 参数的值决定了生产者可以重发消息的次数，如果达到这个次数，生产者会放弃重试并返回错误。默认情况下，生产者会在每次重试之间等待 100ms，不过可以通过 `retry.backoff.ms` 参数来改变这个时间间隔。建议在设置重试次数和重试时间间隔之前， 先测试一下恢复一个崩溃节点需要多少时间（比如所有分区选举出首领需要多长时间），让总的重试时间比 Kafka 集群从崩溃中恢复的时间长，否则生产者会过早地放弃重试。不过有些错误不是临时性错误，没办法通过重试来解决（比如“消息太大”错误）。一般情况下，因为生产者会自动进行重试，所以就没必要在代码逻辑里处理那些可重试的错误。你只需要处理那些不可重试的错误或重试次数超出上限的情况。

---

5. **batch.size**

当有多个消息需要被发送到同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算（而不是消息个数)。当批次被填满，批次里的所有消息会被发送出去。不过生产者并不一定都会等到批次被填满才发送，半满的批次，甚至只包含一个消息的批次也有可能被发送。所以就算把批次大小设置得很大， 也不会造成延迟，只是会占用更多的内存而已。但如果设置得太小，因为生产者需要更频繁地发送消息，会增加一些额外的开销。

---

6. **linger.ms**

该参数指定了生产者在发送批次之前等待更多消息加入批次的时间。KafkaProducer 会在批次填满或 linger.ms 达到上限时把批次发送出去。默认情况下，只要有可用的线程，生产者就会把消息发送出去，就算批次里只有一个消息。把 linger.ms 设置成比 0 大的数， 让生产者在发送批次之前等待一会儿，使更多的消息加入到这个批次。虽然这样会增加延迟，但也会提升吞吐量（因为一次性发送更多的消息，每个消息的开销就变小了）。

---

7. **client.id**

该参数可以是任意的字符串，服务器会用它来识别消息的来源，还可以用在日志和配额指标里。

---

8. **max.in.flight.requests.per.connection**

该参数指定了生产者在收到服务器响应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量。把它设为 1 可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。

---

9. **timeout.ms**、**request.timeout.ms** 和 **metadata.fetch.timeout.ms**

`request.timeout.ms` 指定了生产者在发送数据时等待服务器返回响应的时间，`metadata.fetch.timeout.ms` 指定了生产者在获取元数据（比如目标分区的首领是谁）时等待服务器返回响应的时间。如果等待响应超时，那么生产者要么重试发送数据，要么返回一个错误(抛出异常或执行回调）。`timeout.ms` 指定了 broker 等待同步副本返回消息确认的时间，与 acks 的配置相匹配——如果在指定时间内没有收到同步副本的确认，那么 broker 就会返回一个错误。

---

10. **max.block.ms**

该参数指定了在调用 `send()`方法或使用 `partitionFor()` 方法获取元数据时生产者的阻塞 时间。当生产者的发送缓冲区已满，或者没有可用的元数据时，这些方法就会阻塞。在阻塞时间达到 max.block.ms 时，生产者会抛出超时异常。

---

11. **max.request.size**

该参数用于控制生产者发送的请求大小。它可以指能发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。例如，假设这个值为 1MB，那么可以发送的单个最大消息为 1MB，或者生产者可以在单个请求里发送一个批次，该批次包含了 1000 个消息，每个消息大小为 1KB。另外，broker 对可接收的消息最大值也有自己的限制（`message.max.bytes`），所以两边的配置最好可以匹配，避免生产者发送的消息被 broker 拒绝。

---

12. **receive.buffer.bytes** 和 **send.buffer.bytes**

这两个参数分别指定了 TCP Socket 接收和发送数据包的缓冲区大小。如果它们被设为 -1，就使用操作系统的默认值。如果生产者或消费者与 broker 处于不同的数据中心，那么可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

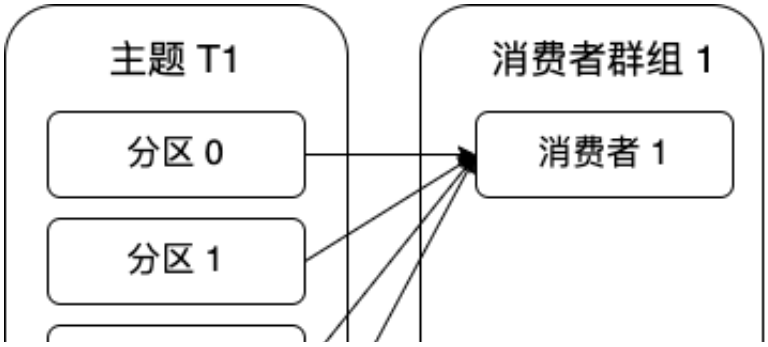
消费者

消费者群组

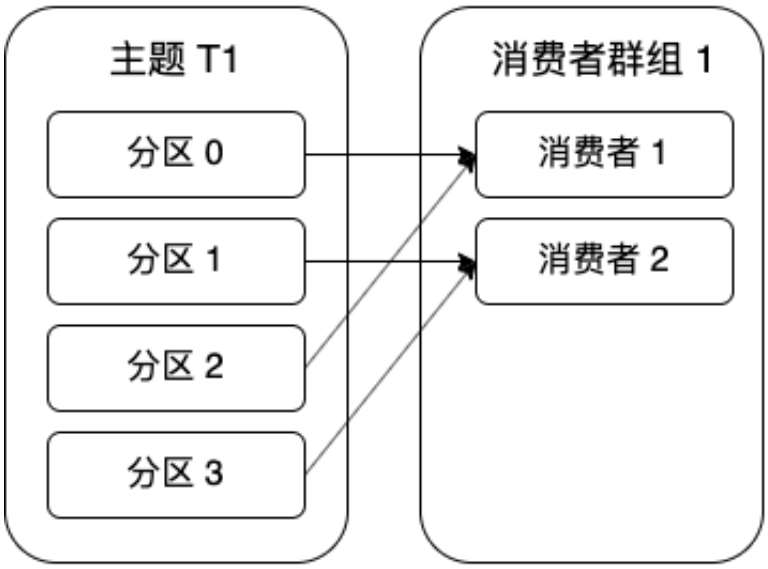
消费者使用一个消费者组名(即 `group.id`)来标记自己，topic 的每条消息都只会被发送到每个订阅它的消费者组的一个消费者实例上。

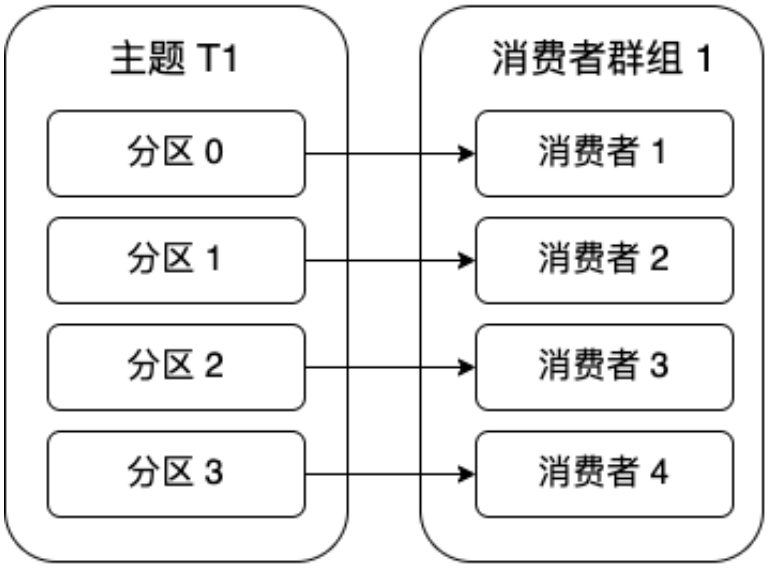
所以消费者群组具有如下特点：

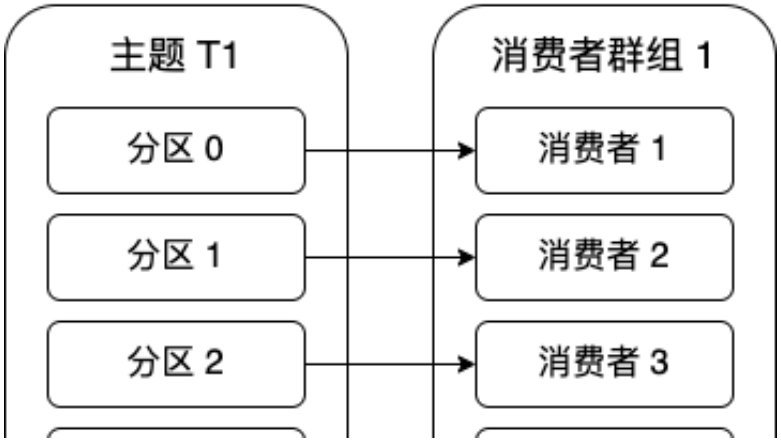
1. 一个 consumer group 可能有若干个 consumer 实例(一个 group 只有一个实例也是允许的)
2. 对于同一个 group 而言，topic 的每条消息只能被发送到 group 下的一个 consumer 实例上，同一个 group 中的 consumer 实例可以消费多个分区上的消息。
3. topic 消息可以被发送到多个 group 中。

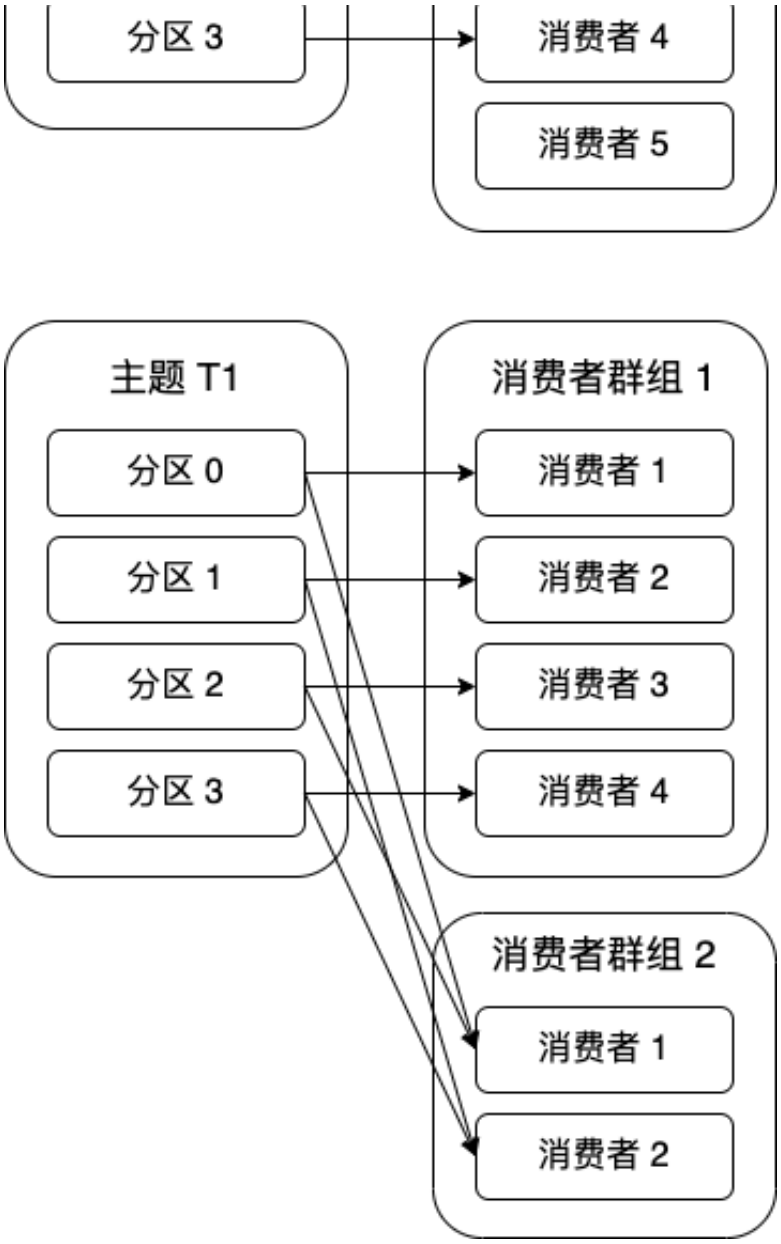



分区 2
分区 3


主题 T1
分区 0
分区 1
分区 2
分区 3
消费者群组 1
消费者 1
消费者 2


主题 T1
分区 0
分区 1
分区 2
分区 3
消费者群组 1
消费者 1
消费者 2
消费者 3
消费者 4


主题 T1
分区 0
分区 1
分区 2
消费者群组 1
消费者 1
消费者 2
消费者 3



为什么需要 consumer group

consumer group 是用于实现高伸缩性、高容错性的 consumer 机制。group 组内多个 consumer 实例可以同时读取 Kafka 消息，而且一旦有某个 consumer 宕机了，consumer group 会立即将已崩溃 consumer 负责的分区转交给其他 consumer 来负责，从而保证整个 group 可以继续工作，不会丢失数据。

消费者群组再均衡

消费者组再平衡(consumer group rebalance)本质上是一种协议，规定了一个 consumer group 下所有 consumer 如何达成一致来分配订阅 topic 的所有分区。

举个例子，假设我们有一个 consumer group，它有 20 个 consumer 实例。该 group 订阅了一个具有 100 个分区的 topic。那么正常情况下，consumer group 平均会为每个 consumer 分配 5 个分区，即每个 consumer 负责读取 5 个分区的数据。这个分配过程就被称作 rebalance。

什么时候触发消费者群组再均衡：

1. 当有新的消费者群组被创建
2. 当消费者群组中加入新的 consumer 实例

3. 当消费者群组中的 consumer 实例崩溃了
4. 当一个主题的分区数增加了

代码示例

```
Properties kafkaProps = new Properties();
// 消费者配置
kafkaProps.put("bootstrap.servers", "");
kafkaProps.put("group.id", "");  // 消费者群组
kafkaProps.put("key.deserializer", "");
kafkaProps.put("value.deserializer", "");

// 根据配置创建消费者对象
KafkaConsumer consumer = new KafkaConsumer(kafkaProps);

// 订阅主题，可以订阅多个主题
consumer.subscribe(Collections.SingletonList(""));

try {
    while(true) {
        /*
         * poll() 参数为超时时间
         * 如果该参数设置为 0，poll() 会立即返回，
         * 否则它会在指定的毫秒数内一直等待 broker 返回数据
         */
        ConsumerRecords records = consumer.poll(100);
        for(ConsumerRecords record : records) {
            // 业务逻辑
            record.topic();
            record.partition();
            record.offset();
            record.key();
            record.value();
        }
    }
} finally {
    consumer.close();
}
```

消费者参数

1. **fetch.min.bytes**

该属性指定了消费者从服务器获取记录的最小字节数。broker 在收到消费者的数据请求时，如果可用的数据量小于 fetch.min.bytes 指定的大小，那么它会等到有足够的可用数据时才把它返回给消费者。这样可以降低消费者和 broker 的工作负载，因为它们在主题不是很活跃的时候（或者一天里的低谷时段）就不需要来来回回地处理消息。如果没有很多可用数据，但消费者的 CPU 使用率却很高，那么就需要把该属性的值设得比默认值大。如果消费者的数量比较多，把该属性的值设置得大一点可以降低

broker 的工作负载。

---

2. **fetch.max.wait.ms**

我们通过 fetch.min.bytes 告诉 Kafka，等到有足够的数据时才把它返回给消费者。而 fetch.max.wait.ms 则用于指定 broker 的等待时间， 默认是 500ms。如果没有足够的数据流入 Kafka， 消费者获取最小数据量的要求就得不到满足，最终导致 500ms 的延迟。如果要降低潜在的延迟， 可以把该参数值设置得小一些。如果 fetch.max.wait.ms 被设为 100ms， 并且 fetch.min.bytes 被设为 1MB， 那么 Kafka 在收到消费者的请求后， 要么返回 1MB 数据，要么在 100ms 后返回所有可用的数据，就看哪个条件先得到满足。

---

3. **max.partition.fetch.bytes**

该属性指定了服务器从每个分区里返回给消费者的最大字节数。它的默认值是 1MB，也就是说，KafkaConsumer.poll() 方法从每个分区里返回的记录最多不超过 max.partition.fetch.bytes 指定的字节。如果一个主题有 20 个分区和 5 个消费者，那么每个消费者需要至少 4MB 的可用内存来接收记录。在为消费者分配内存时，可以给它们多分配一些，因为如果群组里有消费者发生崩溃，剩下的消费者需要处理更多的分区。max.partition.fetch.bytes 的值必须比 broker 能够接收的最大消息的字节数（通过 `message.max.bytes` 属性配置）大，否则消费者可能无法读取这些消息，导致消费者一直挂起重试。在设置该属性时，另一个需要考虑的因素是消费者处理数据的时间。消费者需要频繁调用 poll() 方法来避免会话过期和发生分区再均衡，如果单次调用 poll() 返回的数据太多，消费者需要更多的时间来处理，可能无法及时进行下一个轮询来避免会话过期。如果出现这种情况，可以把 max.partition.fetch.bytes 值改小，或者延长会话过期时间。

---

4. **session.timeout.ms**

该属性指定了消费者在被认为死亡之前可以与服务器断开连接的时间，默认是 3s。如果消费者没有在 session.timeout.ms 指定的时间内发送心跳给群组协调器，就被认为已经死亡，协调器就会触发再均衡，把它的分区分配给群组里的其他消费者。该属性与 `heartbeat.interval.ms` 紧密相关。heartbeat.interval.ms 指定了 poll() 方法向协调器发送心跳的频率，session.timeout.ms 则指定了消费者可以多久不发送心跳。所以，一般需要同时修改这两个属性，heartbeat.interval.ms 必须比 session.timeout.ms 小。一般是 session.timeout.ms 的三分之一。如果 session.timeout.ms 是 3s，那么 heartbeat.interval.ms 应该是 1s。把 session.timeout.ms 值设得比默认值小，可以更快地检测和恢复崩溃的节点，不过长时间的轮询或垃圾收集可能导致非预期的再均衡。把该属性的值设置得大一些，可以减少意外的再均衡，不过检测节点崩溃需要更长的时间。

---

5. **auto.offset.reset**

该属性指定了消费者在读取一个没有偏移量的分区或者偏移量无效的情况下（因消费者长时间失效，包含偏移量的记录已经过时并被删除）该作何处理。它的默认值是 latest，意思是说，在偏移量无效的情况下，消费者将从最新的记录开始读取数据（在消费者启动之后生成的记录）。另一个值是 earliest，意思是说，在偏移量无效的情况下，消费者将从起始位置读取分区的记录。

---

6. **enable.auto.commit**

该属性指定了消费者是否自动提交偏移量，默认值是 true。为了尽量避免出现重复数据和数据丢失，可以把它设为 false，由自己控制何时提交偏移量。如果把它设为 true，还可以通过配置 `auto.commit.interval.ms` 属性来控制提交的频率。

---

7. **partition.assignment.strategy**

分区会被分配给群组里的消费者。PartitionAssignor 根据给定的消费者和主题，决定哪些分区应该被分配给哪个消费者。Kafka 有两个默认的分配策略。

- Range

该策略会把主题的若干个连续的分区分配给消费者。假设消费者 C1 和消费者 C2 同时订阅了主题 T1 和主题 T2，并且每个主题有 3 个分区。那么消费者 C1 有可能分配到这两个主题的分区 0 和分区 1，而消费者 C2 分配到这两个主题的分区 2。因为每个主题拥有奇数个分区，而分配是在主题内独立完成的，第一个消费者最后分配到比第二个消费者更多的分区。只要使用了 Range 策略，而且分区数量无法被消费者数量整除，就会出现这种情况。

- RoundRobin

该策略把主题的所有分区逐个分配给消费者。如果使用 RoundRobin 策略来给消费者 C1 和消费者 C2 分配分区，那么消费者 C1 将分到主题 T1 的分区 0 和分区 2 以及主题 T2 的分区 1，消费者 C2 将分配到主题 T1 的分区 1 以及主题 T2 的分区 0 和分区 2。一般来说，如果所有消费者都订阅相同的主题（这种情况很常见），RoundRobin 策略会给所有消费者分配相同数量的分区（或最多就差一个分区）。

可以通过设置 partition.assignment.strategy 来选择分区策略。 默认使用的是 org. apache.kafka.clients.consumer.RangeAssignor， 这个类实现了 Range 策略，不过也可以把它改成 org.apache.kafka.clients.consumer.RoundRobinAssignor。我们还可以使用自定义策略，在这种情况下，partition.assignment.strategy 属性的值就是自定义类的名字。

---

8. **client.id**

该属性可以是任意字符串，broker 用它来标识从客户端发送过来的消息，通常被用在日志、度量指标和配额里。

---

9. **max.poll.records**

该属性用于控制单次调用 poll() 方法能够返回的记录数量，可以帮你控制在轮询里需要处理的数据量。

---

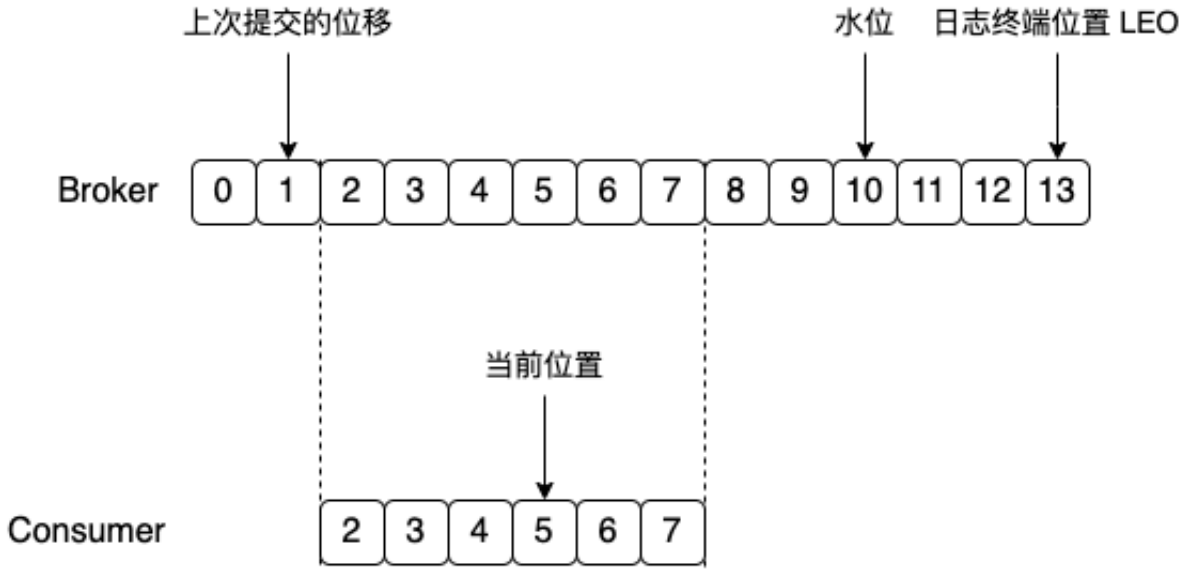
10. **receive.buffer.bytes** 和 **send.buffer.bytes**

socket 在读写数据时用到的 TCP 缓冲区也可以设置大小。如果它们被设为 -1，就使用操作系统的默认值。如果生产者或消费者与 broker 处于不同的数据中心内，可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

位移和位移提交

consumer 端需要为每个它要读取的分区保存消费进度，即分区中当前最新消费消息的位置。该位置就被称为位移(offset) 。consumer 需要定期地向 Kafka 提交自己的位置信息，实际上，这里的位移值通常是下一条待消费的消息的位置。总而言之，offset 就是 consumer 端维护的位置信息。

消费者消费分区消息涉及的相关位置信息：



- 上次提交位移(last committed offset)：consumer 最近一次提交的 offset 值。
- 当前位置(current position)：consumer 已读取但尚未提交时的位置。
- 水位(watermark)：也被称为高水位(highwatermark)，严格来说它不属于 consumer 管理的范围，而是属于分区日志的概念。对于处于水位之下的所有消息，consumer 都是可以读取的，consumer 无法读取水位以上的消息。
- 日志终端位移(Log End Offset，LEO)：也被称为日志最新位移。同样不属于 consumer 范畴，而是属于分区日志管辖。它表示了某个分区副本当前保存消息对应的最大的位移值。值得注意的是，正常情况下 LEO 不会比水位值小。事实上，只有分区所有副本都保存了某条消息，该分区的 leader 副本才会向上移动水位值。

位移提交

1. 自动提交，优点是方便，缺点是可能出现重复数据和数据丢失
2. 手动提交

在处理当前获取到的消息之前提交偏移量

```
while(true) {
    ConsumerRecords records = consumer.poll(100);
    // 在处理当前获取到的消息之前提交偏移量
    consumer.commitSync();
    for(ConsumerRecords record : records) {
        // 业务逻辑
        record.topic();
        record.partition();
        record.offset();
        record.key();
        record.value();
    }
}
```

在处理当前获取到的消息之后提交偏移量

```
while(true) {
    ConsumerRecords records = consumer.poll(100);
    for(ConsumerRecords record : records) {
        // 业务逻辑
        record.topic();
        record.partition();
        record.offset();
        record.key();
        record.value();
    }
    // 在处理当前获取到的消息之后提交偏移量
    consumer.commitSync();
}
```

提交特定偏移量

```
Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap()
int count = 0;
while(true) {
    ConsumerRecords records = consumer.poll(100);
    for(ConsumerRecords record : records) {
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset() + 1, "no metadata")
        );
        if (count % 2 == 0) {
            consumer.commitSync(currentOffsets, null);
        }

        // 业务逻辑
        record.topic();
        record.partition();
        record.offset();
        record.key();
        record.value();

        count++;
    }
}

// 或者

Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap()
int count = 0;
while(true) {
    ConsumerRecords records = consumer.poll(100);
    for(ConsumerRecords record : records) {
        currentOffsets.put(
```

```
                new TopicPartition(record.topic(), record.partition()),
                new OffsetAndMetadata(record.offset() + 1, "no metadata")
            );
            // 业务逻辑
            record.topic();
            record.partition();
            record.offset();
            record.key();
            record.value();

            if (count % 2 == 0) {
                consumer.commitSync(currentOffsets, null);
            }
            count++;
        }
    }
```

异步提交

上述提交采用的都是同步提交的方式，都可以改成异步提交的方式。

```
class OffsetCommitCallback() {
    //提交成功或者抛出异常时进行调用
    public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets,
Exception e) {
        //
    }
}

while(true) {
    ConsumerRecords records = consumer.poll(100);
    for(ConsumerRecords record : records) {
        // 业务逻辑
        record.topic();
        record.partition();
        record.offset();
        record.key();
        record.value();
    }
    // 在处理当前获取到的消息之后异步提交偏移量
    consumer.commitAsync(new OffsetCommitCallback());
}
```

同步提交与异步提交相结合

```
class OffsetCommitCallback() {
    //提交成功或者抛出异常时进行调用
```

```
    public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets,
Exception e) {
        //
    }
}

try {
    while(true) {
        ConsumerRecords records = consumer.poll(100);
        for(ConsumerRecords record : records) {
            // 业务逻辑
            record.topic();
            record.partition();
            record.offset();
            record.key();
            record.value();
        }
        // 在处理当前获取到的消息之后异步提交偏移量
        consumer.commitAsync(new OffsetCommitCallback());
    }
} catch (Exception e) {
    //
} finally {
    try {
        //最后同步提交
        consumer.commitSync();
    } finally {
        consumer.close();
    }
}
```

消费者群组再均衡时监听和回调

```
Properties kafkaProps = new Properties();
// 消费者配置
kafkaProps.put("bootstrap.servers", "");
kafkaProps.put("group.id", "");  // 消费者群组
kafkaProps.put("key.deserializer", "");
kafkaProps.put("value.deserializer", "");

// 根据配置创建消费者对象
KafkaConsumer consumer = new KafkaConsumer(kafkaProps);

class HandleRebalance implements ConsumerRebalanceListener {
    // 消费者停止读取消息之后和再均衡开始之前调用
    public void onPartitionsAssigned(Collections<TopicPartition> partitions) {
        //
    }
```

```
    // 重新分配分区之后和消费者开始读取消息之前调用
    publoc void onPartitionsRevoked(Collections<TopicPartition> partitions) {
        //
    }
}

// 订阅主题，可以订阅多个主题
consumer.subscribe(Collections.SingletonList(""), new HandleRebalance());

try {
    while(true) {
        ConsumerRecords records = consumer.poll(100);
        for(ConsumerRecords record : records) {
            // 业务逻辑
            record.topic();
            record.partition();
            record.offset();
            record.key();
            record.value();
        }
    }
} finally {
    consumer.close();
}
```

从特定的偏移量开始读取数据

使用 poll() 方法从分区获取数据时，broker 都是从上次提交位移之后的位置返回一定数量的消息，使用如下三个方法可以从特定的偏移量开始读取数据。

- `consumer.seekToBeginning(Collection<TopicPartition> tp)` 从分区起始位置开始读取消息
- `consumer.seekToEndllection<TopicPartition> tp)` 从分区末尾开始读取消息
- `consumer.seek(partition, offset)` 从指定位置开始读取消息

例如：重新分配分区之后和消费者开始读取消息之前，调用 `consumer.seek(partition, offset)` 从偏移量为 5 的位置开始读取消息，示例代码如下：

```
Properties kafkaProps = new Properties();
// 消费者配置
kafkaProps.put("bootstrap.servers", "");
kafkaProps.put("group.id", "");  // 消费者群组
kafkaProps.put("key.deserializer", "");
kafkaProps.put("value.deserializer", "");

// 根据配置创建消费者对象
KafkaConsumer consumer = new KafkaConsumer(kafkaProps);
```

```
class HandleRebalance implements ConsumerRebalanceListener {
    // 消费者停止读取消息之后和再均衡开始之前调用
    public void onPartitionsAssigned(Collections<TopicPartition> partitions) {
        //
    }

    // 重新分配分区之后和消费者开始读取消息之前调用
    publoc void onPartitionsRevoked(Collections<TopicPartition> partitions) {
        for(TopicPartition partition: partitions) {
            consumer.seek(partition, 5)
        }
    }
}

// 订阅主题，可以订阅多个主题
consumer.subscribe(Collections.SingletonList(""), new HandleRebalance());

try {
    while(true) {
        ConsumerRecords records = consumer.poll(100);
        for(ConsumerRecords record : records) {
            // 业务逻辑
            record.topic();
            record.partition();
            record.offset();
            record.key();
            record.value();
        }
    }
} finally {
    consumer.close();
}
```

TODO

1. broker 如何组成集群?
2. 分区的所有副本按照一定的规则被分布在不同的 broker 中。如何分布?
3. 如何确定首领副本? 也就是如何进行首领副本选举?
4. 跟随副本如何获取数据?
5. 如何确定跟随副本是否同步?
6. ISR 是什么?
7. 数据在日志目录中的存储格式?
8. 什么是群组协调器，作用是什么?
9. 什么是集群控制器，作用是什么?
10. 选举后的元数据存储在 Zookeeper 中，生产者如何获取，获取的频率
11. 如何检查副本 leader 是否失效，相关的配置选项