Apache Kafka 协议中文编程指南

本文基于 A Guide To The Kafka Protocol 2017-06-14 的版本 v114 进行翻译的。

简介

本文档涵盖了 Kafka 0.8 及更高版本的通信协议实现。它旨在提供一个可读的，涵盖可请求的协议及其二进制格式，以及如何正确使用他们来实现一个客户端的协议指南。本文假设您已经了解了 Kafka 的基本设计以及术语。

0.7 及更早的版本所使用的协议与此类似，但我们选择（希望）在兼容性方面做一次中断，以便清理原有设计上的缺陷，并且对一些概念进行概括。

概述

Kafka 协议是相当简单的，只有六个核心客户端请求 API：

1. 元数据（Metadata）– 描述当前可用的 brokers，brokers 的主机和端口信息，并提供了哪个 broker 托管了哪些分区的信息；
2. 发送（Send）– 发送消息到 broker；
3. 获取（Fetch）– 从 broker 上获取消息。主要分三类：一个用于获取数据，一个用于获取集群的元数据，还有一个用于获取 topic 的偏移量信息；

4. 偏移量（Offsets） – 获取给定 topic 分区的可用偏移量信息；
5. 提交偏移量（Offset Commit） – 提交消费者组（Consumer Group）的一组偏移量；
6. 获取偏移量（Offset Fetch） – 为消费者组获取一组偏移量

上述的概念都将在下面进行详细的说明。此外，从 0.9 版本开始，Kafka 支持为消费者和 Kafka 连接进行分组管理。客户端 API 包括五个请求：

1. 分组协调者（GroupCoordinator） – 用来定位一个分组的当前协调者。
2. 加入分组（JoinGroup） –
   成为某个分组的成员，当分组不存在（没有一个成员时）则创建分组。
3. 同步分组（SyncGroup） – 同步分组中所有成员的状态（例如分发分区分配信息(Partition Assignments)到各个组员）。
4. 心跳（Heartbeat） – 保持组内成员的活跃状态。
5. 离开分组（LeaveGroup） – 直接离开一个组。

最后，有几个管理 API，可用于监控/管理 Kafka 集群（详情请参见 KIP-4）：

1. 描述消费者组（DescribeGroups） –
   用于检查一组群体的当前状态（如：查看消费者分区分配）。
2. 列出组（ListGroups） – 列出某个 broker 当前管理的所有组

开始（Preliminaries）

网络（Network）

Kafka 使用基于 TCP 的二进制协议。该协议将所有 API
定义成请求及响应消息对。所有消息都是有长度限制的，并且由后面描述的基本类型组成。

客户端启动一个 socket 连接，然后写入一系列的请求消息并读回相应的响应消息。连接和断开时均不需要握手消息。如果你保持长连接，那么 TCP 协议本身会节省很多的 TCP
握手时间，但如果真的需要重新建立连接，那么代价也相当小。

客户端可能需要维持多个 broker 的连接，因为数据是分区的，而客户端需要和存储这些数据的 broker 服务器进行通讯。当然，通常不需要在单个客户端实例中维护与单个 broker
的多个连接（即连接池）。

服务器保证在单一的 TCP
连接中，请求将按照顺序进行处理，响应也将按该顺序返回。为保证这种顺序， broker
的请求处理仅允许单个连接同时处理一个请求指令。请注意，客户端可以（也应该）使用非阻塞 IO 实现请求管道（pipelining），从而实现更高的吞吐量。也就是说，客户可以在等待上次请求应答的同时发送下一个请求，因为待完成的请求将会在底层操作系统套接字缓冲区进行缓冲。除非特别说明，所有的请求是由客户端启动，并从服务器获取到相应的响应消息。

服务器能够配置请求大小的最大限制，超过这个限制将导致 socket 连接被断开。

分区和引导（Partitioning and bootstrapping）

Kafka 是一个分区系统，所以不是所有的服务器都有完整的数据集。Topic 被分为 P（预先定义的分区数量）个分区，每个分区复制成 N（复制因子）份，Topic Partition 本身仅仅是编号为 0，1，...，P 的“提交日志”。

所有具有这种特性的系统都会存在如何将特定数据分配给特定分区的问题。Kafka 客户端直接控制此分配策略，broker 则没有特别的语义来决定消息发布到哪个分区。相反，生产者直接将消息发送到某个特定的分区；当获取消息时，消费者也直接从某个特定的分区获取。如果两个生产者要使用相同的分区方案，那么他们必须用同样的方法来计算 Key 到分区映射关系。

这些发布或获取数据的请求必须发送到指定分区的 leader broker 中。此条件同时也会由 broker 检查，发送到不正确的 broker 的请求将会返回 NotLeaderForPartition 错误代码（后文所描述的）。

那么客户端如何找出存在的 topic ？这些 topic 都有哪些分区，以及这些分区都保存在哪些 broker 上，以便它可以直接将请求发送到正确的主机上？这个信息是动态的，因此你不能为每个客户端配置一些静态映射。相反，所有的 Kafka broker 都可以回答描述集群当前状态的数据请求：有哪些 topic，这些 topic 都有哪些分区，哪个 broker 是这些分区的 Leader，以及这些 broker 主机的地址和端口信息。

换句话说，客户端只需要找到一个 broker，而这个 broker 将会告知客户端所有其他存在的 broker，以及这些 broker 上面的所有分区。这个 broker 本身也可能会掉线，因此客户端的最佳做法是在启动的时候提供两到三个 broker 地址。用户可以选择使用负载均衡器或只是在客户端静态地配置两个或三个 Kafka 主机。

客户端并不需要轮询地查看集群是否已经改变；它可以在初始化的时候获取元数据然后将其缓存起来，等到它遇到所用元数据过期的错误信息时再更新元数据。这种错误有两种形式：（1）socket 错误指示客户端不能与特定的 broker 进行通信，
（2）请求响应表明该 broker 不再是其请求数据分区的 Leader 的错误。

1. 轮询初始 Kafka 的 URL 列表，直到我们找到一个可以连接到的 broker，获取集群元数据。
2. 处理获取数据或者生产消息请求，根据这些请求所发送的 topic 和分区，将这些请求发送到合适的 broker。
3. 如果我们得到一个相应的错误(比如显示元数据已经过期)，刷新元数据，然后再试一次。

分区策略（Partitioning Strategies）

上面提到消息的分区分配是由生产者客户端控制的，那么这个功能是如何暴露给最终用户的？

在 Kafka 中分区主要有两个目的：

1. 它平衡了 broker 的数据和请求的负载
2. 它允许消费者多进程处理分发消息，同时也能够维护本地状态，并且在分区中维持消息的

顺序。我们称之为语义的分区（semantic partitioning）。

对于给定的使用场景下，你可能只关心其中的一个或两个。

为了实现简单的负载均衡，一个简单的策略是客户端对所有 broker 进行轮询请求(round robin requests)。另一种选择，在那些生产者比 brokers
多的场景下，给每个客户机随机选择一个分区并发布消息到该分区。后一种策略能够使用较少的 TCP 连接。

语义分区是指使用消息中的 key 来决定分配的分区。例如，如果你正在处理一个点击消息流时，你可能想通过用户ID来划分流，使得特定用户的所有数据会被单个消费者消费。要做到这一点，客户端可以采取与消息相关联的 key，并使用 key 的某种 Hash 值来选择要发送的分区。

批处理（Batching）

我们的 API 鼓励将小的请求批量处理以提高效率。我们发现这能非常显著地提升性能。我们用来发送消息和获取消息的API，总是鼓励处理一连串的消息，而不是单一的消息。聪明的客户端可以利用这一点，并支持“异步”操作模式，以此使用批处理方式来发送那些单独发送的消息，并以更大的块形式发送。我们可以进一步允许跨多个主题和分区的批处理，所以生产请求可能包含追加到许多分区的数据，一个读取请求可以一次性从多个分区提取数据的。

当然，如果他们喜欢，客户端实现者可以选择忽略这一点，一次只发送一条消息。

版本和兼容性（Versioning and Compatibility）

该协议旨在达到向后兼容的基础上实现渐进演化。我们的版本是基于每个 API
基础之上，每个版本包括一个请求和响应对。每个请求包含 API Key，里面包含了被调用 API 的标识，以及表示请求和响应格式的版本号。

这样做的目的是允许客户端实现相应特定版本的请求，并在其请求中指明此版本。我们的目标主要是为了在不允许停机及客户端和服务器端不能同时更改的环境下进行 API 演变。

服务器会拒绝它不支持的版本的请求，并始终返回它期望收到的能够完成请求响应的版本的协议格式。建议的升级路径方式是，新功能将首先部署到服务器（老客户端无法完全利用他们的新功能），然后随着新的客户端的部署，这些新功能将逐步被利用。

目前，所有版本从0开始，当我们介绍这些 API 时，我们将分别显示每个版本的格式。

协议（The Protocol）

协议基本类型（Protocol Primitive Types）

该协议是建立在以下基本类型之上。

1. 定长基本类型（Fixed Width Primitives）

int8, int16, int32, int64 – 具有以大端（Big Endiam）顺序存储的给定精度（以位为单位）的有符号整数。
2. 变长基本类型（Variable Length Primitives）
bytes, string – 这些类型由一个表示长度的带符号整数 N 以及后续 N 字节的内容组成。长度如果为 -1 表示空（null）。string 使用 int16 作为其大小，bytes 使用 int32
3. 数组（Arrays）
这个类型用来处理重复结构的数据。这些将始终被编码为 int32 大小，其包含长度 N，接着是结构的 N 次重复。这些结构自身是由其他的基本数据类型组成。我们后面会用 BNF 语法展示一个 foo 的结构体数组[foo]

阅读请求格式语法的注意事项

后面的 BNF 明确地以上下文无关的语法展示了请求和响应的二进制格式。每个 API 我都会给出请求和响应的定义，以及所有的子定义（sub-definitions）。BNF 使用没有经过缩写的便于阅读的名称（比如我使用一个符号化了的名称来定义了一个 production 错误码，即便它只是 int16 整数）。一般在 BNF 中，一个 production 序列表示一个连接，所以下面给出的 MetadataRequest 将是一个含有 VersionId，然后 clientId，然后 TopicNames 数组（每一个都有其自身的定义）。自定义类型一般使用驼峰法拼写，基本类型使用全小写方式拼写。当存在多种可能的自定义类型时，使用'|'符号分割，并且用括号表示分组。顶级定义不缩进，后续的子部分会被缩进。

通用的请求和响应格式

所有请求和响应都源自以下语法，该语法将通过本文档的其余部分逐步描述：

```
RequestOrResponse => Size (RequestMessage | ResponseMessage)
  Size => int32
```

域（FIELD）	描述
MessageSize	MessageSize 域给出了后续请求或响应消息的字节(bytes)长度。客户端可以先读取4字节的内容，得到长度 N，然后读取并解析后续的 N 字节请求内容。

请求（Requests）

所有请求的格式如下：

```
RequestMessage => ApiKey ApiVersion CorrelationId ClientId RequestMessage
  ApiKey => int16
  ApiVersion => int16
  CorrelationId => int32
  ClientId => string
  RequestMessage => MetadataRequest | ProduceRequest | FetchRequest | OffsetRequest | O
```

ffsetCommitRequest | OffsetFetchRequest

域（FIELD）	描述
ApiKey	这是一个表示所调用 API 的数字 id（即它表示是一个元数据请求？生产请求？获取请求等）.
ApiVersion	这是该 API 的一个数字版本号。我们为每个 API 定义一个版本号，该版本号允许服务器根据版本号正确地解释请求内容。响应消息也始终对应于所述请求的版本的格式。
ClientId	这是为客户端应用程序的自定义的标识。用户可以使用他们喜欢的任何标识符，他们会被用在记录错误，监测统计信息等场景。例如，你可能不仅想要监视每秒的总体请求，还要根据客户端应用程序进行监视，那它就可以被用上（其中每一个都将驻留在多个服务器上）。这个 ID 作为特定的客户端对所有的请求的逻辑分组。
CorrelationId	这是一个用户提供的整数。它将会被服务器原封不动地回传给客户端。用于匹配客户机和服务器之间的请求和响应。

下面将描述各种请求和响应消息。

响应（Responses）

Response => CorrelationId ResponseMessage
CorrelationId => int32
ResponseMessage => MetadataResponse | ProduceResponse | FetchResponse | OffsetResponse | OffsetCommitResponse | OffsetFetchResponse

域（FIELD）	描述
CorrelationId	服务器传回给客户端所提供用作关联请求和响应消息的任何整数。

所有响应都与请求成对匹配（例如，我们发送 MetadataResponse ，会得到 MetadataRequest 响应）

消息集（Message sets）

消息集结构是生产和获取请求共用的结构。在 Kafka 中，消息是由键值对以及少量相关的元数据组成。消息集仅仅是一个有偏移量和大小信息的消息序列。这种格式正好可以用于 broker 上的磁盘存储，也可以用在线上数据交换。

消息集也是 Kafka 中的压缩单元，我们也允许消息递归包含压缩消息从而允许批量压缩。

注意, 在通讯协议中，消息集的前一部分数据没有类似的其他数组元素的 int32。

```
MessageSet => [Offset MessageSize Message]
  Offset => int64
  MessageSize => int32
```

消息格式

```
v0
Message => Crc MagicByte Attributes Key Value
  Crc => int32
  MagicByte => int8
  Attributes => int8
  Key => bytes
  Value => bytes

v1 (supported since 0.10.0)
Message => Crc MagicByte Attributes Key Value
  Crc => int32
  MagicByte => int8
  Attributes => int8
  Timestamp => int64
  Key => bytes
  Value => bytes
```

域（FIELD）	描述
Offset	这是在 Kafka 中作为日志序列号使用的偏移量。当生产者发送非压缩消息，这时候它可以填写任意值。当生产者发送压缩消息，为了避免服务端重新压缩，每个压缩消息的内部消息的偏移量应该从0开始，然后依次增加（kafka压缩消息详见后面的描述）。
Crc	CRC是消息字节的剩余部分的 CRC32 值。broker和消费者可用来检查信息的完整性。
MagicByte	这是用于允许向后兼容的消息二进制格式演变的版本ID。当前值是0。
Attributes	这个字节保存有关消息的元数据属性。 最低的3位包含用于消息的压缩编解码器。 第四位表示时间戳类型，0代表 CreateTime，1代表 LogAppendTime。生产者必须把这个位设成0。 所有其他位必须被设置为0。

域（FIELD）	描述
Timestamp	消息的时间戳。时间戳类型在 Attributes 域中体现。单位是从纪元（1970年1月1日午夜（UTC））开始以来的毫秒数。
Key	Key 是一个可选项，主要用于分区分配。Key 可以为 null。
Value	Value 是消息的实际内容，类型是字节数组。Kafka 支持消息递归包含，因此这里存储的也可能是一个消息集。消息可以为 null。

在 Kafka 0.11 中，MessageSet 和 Message 的结构有重大修改。不仅添加了新字段来支持新功能，例如恰好一次语义（exactly once semantics）和记录头（record headers）；而且消除了先前版本消息格式的递归性质，有利于扁平结构。MessageSet 现在称为 RecordBatch，其中包含一个或多个 Records（不是 Messages）。当启用压缩时，RecordBatch 的头信息并没有压缩，但是 Records 本身是压缩的。此外，Record 中的多个字段是 varint 编码的，这为大批量数据节省了大量空间。

新消息格式的 Magic 值为2.其结构如下：

```
RecordBatch =>
  FirstOffset => int64
  Length => int32
  PartitionLeaderEpoch => int32
  Magic => int8
  CRC => int32
  Attributes => int16
  LastOffsetDelta => int32
  FirstTimestamp => int64
  MaxTimestamp => int64
  ProducerId => int64
  ProducerEpoch => int16
  FirstSequence => int32
  Records => [Record]

Record =>
  Length => varint
  Attributes => int8
  TimestampDelta => varint
  OffsetDelta => varint
  KeyLen => varint
  Key => data
  ValueLen => varint
  Value => data
  Headers => [Header]
```

```
Header => HeaderKey HeaderVal
  HeaderKeyLen => varint
  HeaderKey => string
  HeaderValueLen => varint
  HeaderValue => data
```

新添加字段的语义描述如下：

域（FIELD）	描述
FirstOffset	表示 RecordBatch 中的第一个偏移量。
LastOffsetDelta	批次中每个记录的 offsetDelta 将相对于 FirstOffset 来计算。 具体的，批量消息中每个 Record 的偏移量是 OffsetDelta + FirstOffset。 RecordBatch 中最后一条消息的偏移量。 broker 会使用它来确保正确的行为，即使批量中的记录被压缩。
PartitionLeaderEpoch	在 KIP-101 里面引入的。这是由 broker 在收到生产者请求时设置的，用于确保在 leader 发生变更同时日志正在截断时不会丢失数据。
FirstTimeStamp	批量消息中第一个 Record 的时间戳。RecordBatch 中其他 Record 的时间戳计算公式是其 TimestampDelta + FirstTimestamp。
RecordBatch Attributes	这个字节保存有关消息的元数据属性。 最低的3位包含用于消息的压缩编解码器。 第四位表示时间戳类型，0代表 CreateTime，1代表 LogAppendTime。生产者必须把这个位设成0。（自 0.10.0 起） 第五位表示 RecordBatch 是否是事务的一部分。0 表示 RecordBatch 不是事务性的，而 1 表示它是事务性的。 (自 0.11.0 起) 第六位表示 RecordBatch 是否包含控制消息。1表示 RecordBatch 包含控制消息，0表示不包含控制消息。 控制消息用于启用 Kafka 中的事务并由 broker 生成。 客户端不应将控制批次（即设置了此位）返回给应用程序。（自 0.11.0.0 起）
Record Attributes	记录级别的属性，目前未使用。
MaxTimestamp	批次中最后一个 Record 的时间戳。 broker 会使用它来确保正确的行为，即使批量中的记录被压缩。

域（FIELD）	描述
ProducerId	在 0.11.0.0 中的 KIP-98 引入，这是由 broker 指定的 producerId，并由 InitProducerId 请求接收。 想要支持幂等消息传递和事务的客户端必须设置此字段。
ProducerEpoch	在 0.11.0.0 中的 KIP-98 引入，这是由 broker 指定的 producerEpoch，并由 InitProducerId 请求接收。 想要支持幂等消息传递和事务的客户端必须设置此字段。
FirstSequence	在 0.11.0.0 中的 KIP-98 引入，这是由生产者分配的序列号，broker 使用它来删除重复的消息。 想要支持幂等消息传递和事务的客户端必须设置此字段。 RecordBatch 中每个 Record 的序列号是它的 OffsetDelta + FirstSequence。
Headers	在 0.11.0.0 中的 KIP-82 引入，Kafka 现在支持应用程序级别记录级别的头信息。 Producer 和 Consumer 的 APIS 已进行了相应的更新，以便支持编读写这些头信息。

压缩（Compression）

Kafka 支持压缩多条消息以便提高效率。当然，这比压缩一条原始消息要来得复杂。因为单条消息可能没有足够的冗余信息以达到良好的压缩比，压缩的多条信息必须以特殊方式批量发送（当然，如果真的需要的话，你可以自己压缩批处理的一个消息）。要被发送的消息被包装（未压缩）在一个 MessageSet 结构中，然后将其压缩并存储在具有适当压缩编解码器集的单条消息的值字段中。接收系统通过解压缩得到实际的消息集。外层 MessageSet 应该只包含一个压缩的“消息”（详情见Kafka-1718）。

Kafka目前支持以下两种压缩算法：

压缩算法（COMPRESSION）	编码器编号（CODEC）
None	0
GZIP	1
Snappy	2

API

本节将给出每个 API 的用法、二进制格式，以及字段的含义等细节。

元数据 API（Metadata API）

这个 API 解决以下几个问题：

- 存在哪些 Topic？
- 每个主题有几个分区（Partition）？

- 每个分区的 Leader 分别是哪个 broker？
- 这些 broker 的地址和端口分别是什么？

这是唯一一个能发往集群中任意一个 broker 的请求。

由于可能有许多
topic，客户端可以提供一个主题名称的可选列表，以便仅返回这些主题的元数据。

返回的元数据信息是分区级别的信息，为了方便和避免冗余，以 topic
为组集中在一起。每个分区的元数据中包含了 leader 以及所有副本以及正在同步的副本的信息。

注意: 如果 broker 配置中设置了 auto.create.topics.enable，当查找的 topic 不存在时，topic
元数据请求将会以默认的复制因子和默认的分区数为参数创建该 topic。

topic元数据请求（Topic Metadata Request）

```
TopicMetadataRequest => [TopicName]
  TopicName => string
```

域（FIELD）	描述
TopicName	要获取元数据的主题数组。 如果为空，就返回所有主题的元数据

元数据响应（Metadata Response）

响应包含的每个分区的元数据，这些分区元数据以 topic 为组组装在一起。该元数据以 broker id
来指向具体的 broker。每个 broker 有一个地址和端口。

```
MetadataResponse => [Broker][TopicMetadata]
  Broker => NodeId Host Port  (any number of brokers may be returned)
    NodeId => int32
    Host => string
    Port => int32
  TopicMetadata => TopicErrorCode TopicName [PartitionMetadata]
    TopicErrorCode => int16
  PartitionMetadata => PartitionErrorCode PartitionId Leader Replicas Isr
    PartitionErrorCode => int16
    PartitionId => int32
    Leader => int32
    Replicas => [int32]
    Isr => [int32]
```

域（FIELD）	描述
Leader	该分区作为 Leader 节点的 Kafka broker id。如果正处于 Leader

域（FIELD）	描述
	选举过程中，这时候没有 Leader ，则返回 -1。
Replicas	该分区中，其他活着的作为 slave 的节点集合。
Isr	副本集合中，所有处在与 Leader 跟随（“caught up”，表示数据已经完全复制到这些节点）状态的子集
Broker	kafka broker 节点的 id, 主机名, 端口信息

可能的错误码（Possible Error Codes）

* UnknownTopic (3)
* LeaderNotAvailable (5)
* InvalidTopic (17)
* TopicAuthorizationFailed (29)

生产 API（Produce API）

生产 API 主要用于将消息集发送到服务器。为了提高效率，它允许在单个请求中发送用于许多主题分区的消息集。

生产 API 使用通用的消息集格式，但由于发送时还没有被分配偏移量，因此可以任意填写该值。

生产请求（Produce Request）

v0, v1 (supported in 0.9.0 or later) and v2 (supported in 0.10.0 or later)
ProduceRequest => RequiredAcks Timeout [TopicName [Partition MessageSetSize MessageSet]]
  RequiredAcks => int16
  Timeout => int32
  Partition => int32
  MessageSetSize => int32

v1 及其之后的 Produce Request 表示客户端可以解析 Produce Response 中的配额限制时间（quota throttle time）。
v2 及其之后的 Produce Request 表示客户端可以解析 Produce Response 中的时间戳字段（timestamp field）。

域（FIELD）	描述
RequiredAcks	这个值表示服务端收到多少确认后才发送反馈消息给请求端。如果设置为0，那么服务端将不发送 response（这是服务端不会回复请求的唯一情况）。如果这个值为1，那么服务器将等到数

域（FIELD）	描述
	据写入到本地日志之后发送 response。如果这个值是-1，那么服务端将阻塞，直到这个消息被所有的同步副本写入后再发送 response。
Timeout	这提供了服务端等待接收 RequiredAcks 确认数量的最长时间（以毫秒为单位）。超时并非一个确切的限制，有以下原因：（1）不包括网络延迟；（2）计时器在此请求处理开始时开始，如果有很多请求，由于服务器负载而导致的排队等待时间将不被包括在内；（3）如果本地写入时间超过这个设置，我们将不会终止本地写操作，这样这个超时时间就不会得到遵守。要获得此类型的硬超时，客户端应使用套接字超时。
TopicName	发布数据的主题名称。
Partition	发布数据的分区
MessageSetSize	后续消息集的长度，字节为单位
MessageSet	上面描述的标准格式消息集合

生产响应（Produce Response）

```
v0
ProduceResponse => [TopicName [Partition ErrorCode Offset]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  Offset => int64

v1 (supported in 0.9.0 or later)
ProduceResponse => [TopicName [Partition ErrorCode Offset]] ThrottleTime
  TopicName => string
  Partition => int32
  ErrorCode => int16
  Offset => int64
  ThrottleTime => int32

v2 (supported in 0.10.0 or later)
ProduceResponse => [TopicName [Partition ErrorCode Offset Timestamp]] ThrottleTime
  TopicName => string
  Partition => int32
  ErrorCode => int16
  Offset => int64
  Timestamp => int64
  ThrottleTime => int32
```

域	描述
Topic	此响应对应的主题。

域	描述
Partition	此响应对应的分区。
ErrorCode	如果有，此分区对应的错误信息。错误以分区为单位提供，因为可能存在给定的分区不可用或者被其他的主机维护（非 Leader），但是其他的分区的请求操作成功的情况
Offset	追加到该分区的消息集中第一个消息的偏移量。
Timestamp	如果该主题使用了 LogAppendTime，这个时间戳就是 broker 分配给这个消息集。这个消息集中的所有消息都有相同的时间戳。 如果使用的是 CreateTime，这个域始终是 -1。 如果没有返回错误码，生产者可以假定消息的时间戳已经被 broker 接受。 单位为从UTC标准准时间1970年1月1日0点到所在时间的毫秒数。
ThrottleTime	由于配额违规而限制请求的持续时间，以毫秒为单位。（如果没有违反限额条件，此值为0）

Possible Error Codes: (TODO)

Fetch API

获取消息接口用于获取一些 topic 分区的一个或多个的日志块。逻辑上根据指定topic，分区和消息起始偏移量开始获取一批消息。在一般情况下，返回消息的偏移量将大于或等于开始偏移量。然而，如果是压缩消息，有可能返回的消息的偏移量比起始偏移量小。这类的消息的数量通常较少，并且调用者必须负责过滤掉这些消息。

获取数据指令请求遵循一个长轮询模型，如果没有足够数量的消息可用，它们可以阻塞一段时间。

作为优化，服务器被允许在消息集的末尾返回一个消息的一部分，客户端应处理这种情况。

有一点要注意的是，获取消息 API 需要指定消费的分区。现在的问题是如何让消费者知道消费哪个分区？特别地，作为一组消费者，如何使得每个消费者获取分区的一个子集，并且平衡这些分区。我们使用 zookeeper 动态地为 Scala 和 Java
客户端完成这个任务。这种方法的缺点是，它需要一个相当胖的客户端并且需要客户端与 zookeeper 连接。我们尚未创建一个 Kafka 接口，使得该功能移动到服务器端并被更方便地访问。一个简单的消费者客户端可以通过配置指定访问的分区，但这样将不能在某些消费者失效后做到分区的动态重新分配。我们希望能在下一个主要版本解决这一空白。

Fetch Request

```
FetchRequest => ReplicaId MaxWaitTime MinBytes [TopicName [Partition FetchOffset MaxBytes]]
  ReplicaId => int32
  MaxWaitTime => int32
```

```
  MinBytes => int32
  TopicName => string
  Partition => int32
  FetchOffset => int64
  MaxBytes => int32
```

域	描述
ReplicaId	副本ID的是发起这个请求的副本节点ID。普通消费者客户端应该始终将其指定为-1，因为他们没有节点ID。其他broker设置他们自己的节点ID。基于调试目的，以非代理身份模拟副本broker发出获取数据指令请求时，这个值填-2。
MaxWaitTime	如果没有足够的数据可发送时，最大阻塞等待时间，以毫秒为单位。
MinBytes	返回响应消息的最小字节数目，必须设置。如果客户端将此值设为0，服务器将会立即返回，但如果没有新的数据，服务端会返回一个空消息集。如果它被设置为1，则服务器将在至少一个分区收到一个字节的数据的情况下立即返回，或者等到超时时间达到。通过设置较高的值，结合超时设置，消费者可以在牺牲一点实时性能的情况下通过一次读取较大的字节的数据块从而提高的吞吐量（例如，设置MaxWaitTime至100毫秒，设置MinBytes为64K，将允许服务器累积数据达到64K前等待长达100ms再响应）。
TopicName	topic名称
Partition	获取数据的Partition id
FetchOffset	获取数据的起始偏移量
MaxBytes	此分区返回消息集所能包含的最大字节数。这有助于限制响应消息的大小。

Fetch Response

```
v0
FetchResponse => [TopicName [Partition ErrorCode HighwaterMarkOffset MessageSetSize Mes
sageSet]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  MessageSetSize => int32

v1 (supported in 0.9.0 or later) and v2 (supported in 0.10.0 or later)
FetchResponse => ThrottleTime [TopicName [Partition ErrorCode HighwaterMarkOffset Messa
geSetSize MessageSet]]
  ThrottleTime => int32
```

```
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  MessageSetSize => int32
```

域	描述
ThrottleTime	由于限额冲突而导致的时间延迟长度，以毫秒为单位。（如果没有违反限额条件，此值为0）
TopicName	返回消息所对应的Topic名称。
Partition	返回消息所对应的分区id。
HighwaterMarkOffset	此分区日志中最末尾的偏移量。此信息可被客户端用来确定后面还有多少条消息。
MessageSetSize	此分区中消息集的字节长度
MessageSet	此分区获取到的消息集，格式与之前描述相同

Fetch Response v1 只包含 v0 格式的消息。
Fetch Response v2 既包含 v0 又包含 v1 格式的消息。

可能的错误码（Possible Error Codes）

* OFFSET_OUT_OF_RANGE (1)
* UNKNOWN_TOPIC_OR_PARTITION (3)
* NOT_LEADER_FOR_PARTITION (6)
* REPLICA_NOT_AVAILABLE (9)
* UNKNOWN (-1)

偏移量 API（又称 ListOffset）（Offset API）

此 API 描述了一组主题分区可用的有效偏移范围。生产者和获取数据 API 的请求必须发送到分区 Leader 所在的 broker 上，这需要通过使用元数据的 API 来确定。

对于版本0，响应包含请求分区每个段的起始偏移量以及日志结束偏移量，即，将被追加到给定分区中的下一个消息的偏移量。在版本1（0.10.1.0 开始支持的）中，Kafka 支持按消息中使用的时间戳搜索偏移的时间索引，并对此API进行了更改以支持此特性。注意这个 API 只支持开启 0.10 消息格式的 topic，否则返回 UNSUPPORTED_FOR_MESSAGE_FORMAT 错误。

Offset Request

```
// v0
ListOffsetRequest => ReplicaId [TopicName [Partition Time MaxNumberOfOffsets]]
  ReplicaId => int32
  TopicName => string
  Partition => int32
```

```
  Time => int64
  MaxNumberOfOffsets => int32


// v1 (supported in 0.10.1.0 and later)
ListOffsetRequest => ReplicaId [TopicName [Partition Time]]
  ReplicaId => int32
  TopicName => string
  Partition => int32
  Time => int64
```

域	描述
Time	用来请求一定时间(毫秒)前的所有消息。这里有两个特殊取值：-1表示获取最后一个offset（也就是后面即将到来消息的offset值）； -2表示获取最早的有效偏移量。注意，因为获取到偏移值都是降序排序，因此请求最早Offset的请求将总是返回一个值

Offset Response

```
// v0
OffsetResponse => [TopicName [PartitionOffsets]]
  PartitionOffsets => Partition ErrorCode [Offset]
  Partition => int32
  ErrorCode => int16
  Offset => int64


// v1
ListOffsetResponse => [TopicName [PartitionOffsets]]
  PartitionOffsets => Partition ErrorCode Timestamp [Offset]
  Partition => int32
  ErrorCode => int16
  Timestamp => int64
  Offset => int64
```

可能的错误码（Possible Error Codes）

```
* UNKNOWN_TOPIC_OR_PARTITION (3)
* NOT_LEADER_FOR_PARTITION (6)
* UNKNOWN (-1)
* UNSUPPORTED_FOR_MESSAGE_FORMAT (43)
```

偏移量提交/获取接口（Offset Commit/Fetch API）

这些 API 允许集中管理偏移。阅读偏移量管理了解更多。根据对 KAFKA-993 的评论，这些 API 调用在Kafka 0.8.1.1 之前的版本中并不完全正常。 它将在 0.8.2 版本中提供。

组协调者请求（Group Coordinator Request）

给定消费者组（consumer group）的偏移量由称为组协调者的特定 broker
维护。比如消费者需要向这个特定的 broker
提交和获取偏移量。可以通过发出一组协调者发现请求从而获得当前协调者信息。

```
GroupCoordinatorRequest => GroupId
  GroupId => string
```

组协调者响应（Group Coordinator Response）

```
GroupCoordinatorResponse => ErrorCode CoordinatorId CoordinatorHost CoordinatorPort
  ErrorCode => int16
  CoordinatorId => int32
  CoordinatorHost => string
  CoordinatorPort => int32
```

可能的错误码（Possible Error Codes）

* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* GROUP_AUTHORIZATION_FAILED (30)

偏移量提交请求（Offset Commit Request）

```
v0 (supported in 0.8.1 or later)
OffsetCommitRequest => ConsumerGroupId [TopicName [Partition Offset Metadata]]
  ConsumerGroupId => string
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string
```

v1 (supported in 0.8.2 or later)
OffsetCommitRequest => ConsumerGroupId ConsumerGroupGenerationId ConsumerId [TopicName [Partition Offset TimeStamp Metadata]]
  ConsumerGroupId => string
  ConsumerGroupGenerationId => int32
  ConsumerId => string
  TopicName => string
  Partition => int32
  Offset => int64
  TimeStamp => int64
  Metadata => string

v2 (supported in 0.9.0 or later)
OffsetCommitRequest => ConsumerGroup ConsumerGroupGenerationId ConsumerId RetentionTime [TopicName [Partition Offset Metadata]]
  ConsumerGroupId => string
  ConsumerGroupGenerationId => int32
  ConsumerId => string
  RetentionTime => int64
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string

在 v0 和 v1 版本中，每个分区的时间戳被定义为提交时间戳，并且偏移协调者将保留提交的偏移量，直到其在 broker
配置中指定的提交时间戳+偏移保留时间为止；如果未设置时间戳字段，broker
将在提交偏移量之前将提交时间作为接收时间，用户可以显式设置提交时间戳，如果他们希望在broker上保留更长的提交偏移量而不是配置的偏移保留时间。

在 v2 版本中，我们移除了时间戳字段，但是增加了一个全局保存时间域（详情参见 KAFKA-1634）；broker 会设置提交时间戳为接收到请求的时间，但是提交的偏移量能被保存到提交请求中用户指定的保存时间，如果这个保存时间没有设值(-1)，那么 broker 会使用默认的保存时间。

注意，当这个 API 在 simple consumer 模式下使用，其并非消费者组的一部分，那么这时候 generationId 必须被设置成-1，并且 memberId
必须为空（非null）。另外，如果有一个活动的消费者组有同样的
groupId，那么提交偏移量的请求将会被拒绝（一般会返回 UNKNOWN_MEMBER_ID 或者 ILLEGAL_GENERATION 错误）。

偏移量提交响应（Offset Commit Response）

v0, v1 and v2:

```
OffsetCommitResponse => [TopicName [Partition ErrorCode]]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
```

可能的错误码（Possible Error Codes）

* OFFSET_METADATA_TOO_LARGE (12)
* GROUP_LOAD_IN_PROGRESS (14)
* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* NOT_COORDINATOR_FOR_GROUP (16)
* ILLEGAL_GENERATION (22)
* UNKNOWN_MEMBER_ID (25)
* REBALANCE_IN_PROGRESS (27)
* INVALID_COMMIT_OFFSET_SIZE (28)
* TOPIC_AUTHORIZATION_FAILED (29)
* GROUP_AUTHORIZATION_FAILED (30)

偏移量获取请求（Offset Fetch Request）

根据 KAFKA-1841 的评论，v0 和 v1 是相同的，但 v0（0.8.1或更高版本支持）从 zookeeper 读取的偏移量，而 v1（0.8.2或更高版本支持）从 Kafka 读偏移量。

```
v0 and v1 (supported in 0.8.2 or after):
OffsetFetchRequest => ConsumerGroup [TopicName [Partition]]
  ConsumerGroup => string
  TopicName => string
  Partition => int32
```

偏移量获取响应（Offset Fetch Response）

```
v0 and v1 (supported in 0.8.2 or after):
OffsetFetchResponse => [TopicName [Partition Offset Metadata ErrorCode]]
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string
  ErrorCode => int16
```

请注意，消费者组中一个 topic 的分区如果没有偏移量，broker
不会设定一个错误码（因为它不是一个真正的错误），但会返回空的元数据并将偏移字段为-1。

偏移量获取请求 v0 和 v1 版本之间格式上并没有什么区别。功能实现上来说，v0 版本从
zookeeper 获取偏移量，v1 版本从 Kafka 中获取偏移量。

可能的错误码（Possible Error Codes）

* UNKNOWN_TOPIC_OR_PARTITION (3) <- 仅适用于 v0 版本
* GROUP_LOAD_IN_PROGRESS (14)
* NOT_COORDINATOR_FOR_GROUP (16)
* ILLEGAL_GENERATION (22)
* UNKNOWN_MEMBER_ID (25)
* TOPIC_AUTHORIZATION_FAILED (29)
* GROUP_AUTHORIZATION_FAILED (30)

组成员管理 API（Group Membership API）

客户端使用这些请求参与由 Kafka
管理的客户端组。从较高层面上看，集群中的每个组都被分配一个 broker（即组协调者）以便于组管理。一旦得到了组协调者地址（使用上面的消费者组协调者请求），组成员可以加入该组，同步状态，然后用心跳消息保持在组中的活跃状态。当客户端关闭时，它会使用离开组请求从消费者组中注销。此协议的语义在 Kafka 客户端分配协议中有详细描述。

组建管理接口的主要使用场景是消费者组，但这些请求被设计的很通用，以便支持其他应用场景（例如，Kafka Connect 组）。这种设计带来的代价就是一些特定的组语义（group semantics）被推到了客户端实现。例如，下面定义的 JoinGroup 和 SyncGroup 请求无明确定义的字段以支持消费者组分区分配。相反，它们在其中包含有一些通用的字节数组（byte
arrays），用这些字节数组就可以使得分区分配切入在消费者客户端实现。

但是，虽然这允许每个客户端实现定义自己的嵌入式模式，但与 Kafka
工具的兼容性要求客户端使用 Kafka 标准的嵌入式模式。例如，consumer-groups.sh 这个应用程序会假定用这种格式来显示分区分配。因此，我们建议客户遵循相同的模式，使这些工具对所有客户端实现都可以正常工作。

加入组请求（Join Group Request）

加入组请求用于让客户端成为组的成员。当新成员加入一个现有的组，之前加入的所有成员必须发送一个新的加入组请求来重新加入组。当成员第一次加入该组，memberId
将是空的（即“”），但重新加入的成员应该使用与之前生成的相同 memberId。

v0 supported in 0.9.0.0 and greater
JoinGroupRequest => GroupId SessionTimeout MemberId ProtocolType GroupProtocols
  GroupId => string
  SessionTimeout => int32

```
  MemberId => string
  ProtocolType => string
  GroupProtocols => [ProtocolName ProtocolMetadata]
    ProtocolName => string
    ProtocolMetadata => bytes

v1 supported in 0.10.1.0 and greater
JoinGroupRequest => GroupId SessionTimeout MemberId ProtocolType GroupProtocols
  GroupId => string
  SessionTimeout => int32
  RebalanceTimeout => int32
  MemberId => string
  ProtocolType => string
  GroupProtocols => [ProtocolName ProtocolMetadata]
    ProtocolName => string
    ProtocolMetadata => bytes
```

SessionTimeout 字段指示客户端的存活。如果组协调者在 session 过期前没有收到一个心跳，那么组员会被移除组。在 0.10.1 之前版本， session timeout也被用作完成 rebalance 的超时时间。一旦组管理者开始 rebalance， 每一个组员会触发 session timeout 以便发送新的 JoinGroup 请求。 如果它们失败了，它们会从组中移除。 在 0.10.1 中，新版 JoinGroup 会使用一个独立的 RebalanceTimeout 来创建，一旦 rebanlance 开始，每个客户端触发过期以便重新加入，但是请注意，如果 session timeout 小于 rebalance timeout, 客户端还是会持续发送 heatbeat。

ProtocolType 字段定义了该组实现的嵌入协议。组协调器确保该组中的所有成员都支持相同的协议类型。组中包含的协议（GroupProtocols）字段中的协议名称和元数据的含义取决于协议类型。请注意，加入群请求允许多协议/元数据对。这使得滚动升级时无需停机。协调器会选择所有成员支持的一种协议，升级后的成员既包括新版本和老版本的协议，一旦所有成员都升级，协调器将选择列在数组中最前面的组协议（GroupProtocol）。

消费者组:
下文我们定义了消费者组使用的嵌入协议。我们建议所有消费者客户端实现遵循这个格式，以便 Kafka 工具能够对所有的客户端正常工作

```
ProtocolType => "consumer"

ProtocolName => AssignmentStrategy
  AssignmentStrategy => string

ProtocolMetadata => Version Subscription UserData
  Version => int16
  Subscription => [Topic]
```

```
  Topic => string
UserData => bytes
```

UserData 域可以用来自定义分配策略。例如，在一个粘性分区策略实现中（sticky partitioning implementation），这个字段可以包含之前的分配。在基于资源的分配策略（resource-based assignment strategy），也可以包括每个运行消费者主机上的CPU个数等信息。

Kafka Connect 使用 connect 协议类型，协议细节也是基于 Connect 的内部实现。

加入组响应（Join Group Response）

接收到来自该组中的所有成员组的加入组请求后，协调者将选择一个成员作为
Leader，并且选择所有成员支持的协议。Leader 将收到组员的完整列表与选择的协议相关的元数据。其他追随者成员，会收到一个空会员数组。Leader
需要检查每个成员的元数据，并且使用下文中描述的 SyncGroup 请求来分配状态。

一旦加入组阶段完成，协调器会增加该组的 GenerationId，这个 Id 是发送给每个成员的
response 中的一个字段，同时也会在心跳和偏移量提交请求中。当协调器重新 rebalance 了一个组，协调器将发送一个错误码，表示客户端成员需要重新加入组。如果重新平衡完成前成员未重入组（rejoin），那么它将有一个旧 generationId，在新的请求使用这个旧Id时，这将导致 ILLEGAL_GENERATION 错误。

```
v0 and v1 supported in 0.9.0 and greater
JoinGroupResponse => ErrorCode GenerationId GroupProtocol LeaderId MemberId Members
  ErrorCode => int16
  GenerationId => int32
  GroupProtocol => string
  LeaderId => string
  MemberId => string
  Members => [MemberId MemberMetadata]
    MemberId => string
    MemberMetadata => bytes
```

消费者组: 协调者负责选择所有成员都兼容协议（即分区分配策略），Leader 是实际执行分配的成员，加入群请求可以包含多个分配策略，从而支持现有版本升级或者更改不同的分配策略。

可能的错误码（Possible Error Codes）

```
* GROUP_LOAD_IN_PROGRESS (14)
* GROUP_COORDINATOR_NOT_AVAILABLE (15)
```

* NOT_COORDINATOR_FOR_GROUP (16)
* INCONSISTENT_GROUP_PROTOCOL (23)
* UNKNOWN_MEMBER_ID (25)
* INVALID_SESSION_TIMEOUT (26)
* GROUP_AUTHORIZATION_FAILED (30)

同步组请求（Sync Group Request）

组 leader
使用同步组请求将状态（例如，分区分配）分配给当前所有成员。所有成员在加入组后立即发送 SyncGroup，但只有 leader 提供组的分配。

```
SyncGroupRequest => GroupId GenerationId MemberId GroupAssignment
  GroupId => string
  GenerationId => int32
  MemberId => string
  GroupAssignment => [MemberId MemberAssignment]
    MemberId => string
    MemberAssignment => bytes
```

消费者组：消费者组的 MemberAssignment 字段的格式如下所示

```
MemberAssignment => Version PartitionAssignment
  Version => int16
  PartitionAssignment => [Topic [Partition]]
    Topic => string
    Partition => int32
  UserData => bytes
```

使用使用者协议类型的所有客户端实现都应支持此模式。

同步组响应（Sync Group Response）

组中的每个成员都会接收到 leader 发出的同步组响应。

```
SyncGroupResponse => ErrorCode MemberAssignment
  ErrorCode => int16
  MemberAssignment => bytes
```

可能的错误码（Possible Error Codes）

* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* NOT_COORDINATOR_FOR_GROUP (16)
* ILLEGAL_GENERATION (22)
* UNKNOWN_MEMBER_ID (25)
* REBALANCE_IN_PROGRESS (27)
* GROUP_AUTHORIZATION_FAILED (30)

心跳请求（Heartbeat Request）

一旦成员加入并同步完成，它将开始定期发送心跳以使自己保持在组中。
当协调者在配置的会话超时时间内没有它的收到心跳请求，则该成员将被踢出该组。

```
HeartbeatRequest => GroupId GenerationId MemberId
  GroupId => string
  GenerationId => int32
  MemberId => string
```

心跳响应（Heartbeat Response）

```
HeartbeatResponse => ErrorCode
  ErrorCode => int16
```

可能的错误码（Possible Error Codes）

* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* NOT_COORDINATOR_FOR_GROUP (16)
* ILLEGAL_GENERATION (22)
* UNKNOWN_MEMBER_ID (25)
* REBALANCE_IN_PROGRESS (27)
* GROUP_AUTHORIZATION_FAILED (30)

离开组请求（LeaveGroup Request）

如果想明确离开组，客户端可以发送离开组请求。这优先于会话超时，因为它能使该组快速再平衡，这对于消费者而言这意味着可以用更短的时间将分区分配到一个活动的成员。

```
LeaveGroupRequest => GroupId MemberId
  GroupId => string
  MemberId => string
```

离开组响应（LeaveGroup Response）

```
LeaveGroupResponse => ErrorCode
  ErrorCode => int16
```

可能的错误码（Possible Error Codes）

* GROUP_LOAD_IN_PROGRESS (14)
* CONSUMER_COORDINATOR_NOT_AVAILABLE (15)
* NOT_COORDINATOR_FOR_CONSUMER (16)
* UNKNOWN_CONSUMER_ID (25)
* GROUP_AUTHORIZATION_FAILED (30)

管理 API（Administrative API）

组列表请求（ListGroups Request）

这个 API 可用于查找 broker 管理的当前组。 要获取集群中所有组的列表，必须给所有的broker发送 ListGroup。

```
ListGroupsRequest =>
```

组列表响应（ListGroups Response）

```
ListGroupsResponse => ErrorCode Groups
  ErrorCode => int16
  Groups => [GroupId ProtocolType]
    GroupId => string
    ProtocolType => string
```

可能的错误码（Possible Error Codes）

* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* AUTHORIZATION_FAILED (29)

组描述请求（DescribeGroups Request）

```
DescribeGroupsRequest => [GroupId]
  GroupId => string
```

组描述响应（DescribeGroups Response）

```
DescribeGroupsResponse => [ErrorCode GroupId State ProtocolType Protocol Members]
  ErrorCode => int16
  GroupId => string
  State => string
  ProtocolType => string
  Protocol => string
  Members => [MemberId ClientId ClientHost MemberMetadata MemberAssignment]
    MemberId => string
    ClientId => string
    ClientHost => string
    MemberMetadata => bytes
    MemberAssignment => bytes
```

可能的错误码（Possible Error Codes）

* GROUP_LOAD_IN_PROGRESS (14)
* GROUP_COORDINATOR_NOT_AVAILABLE (15)
* NOT_COORDINATOR_FOR_GROUP (16)
* AUTHORIZATION_FAILED (29)

常量（Constants）

API 关键字及版本（Api Keys And Current Versions）

下面是请求中 ApiKey 的数字值，用来表示上面所述的请求类型。

接口名称（API NAME）	APIKEY值
ProduceRequest	0

接口名称（API NAME）	APIKEY值
FetchRequest	1
OffsetRequest	2
MetadataRequest	3
Non-user facing control APIs	4-7
OffsetCommitRequest	8
OffsetFetchRequest	9
GroupCoordinatorRequest	10
JoinGroupRequest	11
HeartbeatRequest	12
LeaveGroupRequest	13
SyncGroupRequest	14
DescribeGroupsRequest	15
ListGroupsRequest	16

错误代码（Error Codes）

我们用数字代码表示服务器发生的问题。这些可以由客户端转换为异常或客户端语言中的任何适当的错误处理机制。这是当前正在使用的错误代码表：

错误名称（Error）	编码（Code）	是否可重试（Retriable）	描述
NoError	0		没有错误
Unknown	-1		服务器未知错误
OffsetOutOfRange	1		请求的偏移量超过服务器维护主题分区的偏移量。
InvalidMessage / CorruptMessage	2	Yes	这个错误表示消息的内容与它的CRC校验码不符合。
UnknownTopicOrPartition	3	Yes	broker上不存在所请求的主题或者分区。
InvalidMessageSize	4		消息长度为负数。
LeaderNotAvailable	5	Yes	这个错误会在 leader 选举之间抛出，此时这个分区没有 leader 因此不能被写入。
NotLeaderForPartition	6	Yes	这个错误表示客户端正在把消息发送给副本，而不是分区的 leader。这说明客户端的元数据已经过期。
RequestTimedOut	7	Yes	当这个请求超过了用户自定义的请求时间限制抛出此错误
BrokerNotAvailable	8		这不是面向客户端的错误，一般在 broker 不活

错误名称（Error）	编码（Code）	是否可重试（Retriable）	描述
			动时使用一些工具。
ReplicaNotAvailable	9		当broker希望有副本而实际上并没有时抛出（这个错误可以被安全地忽略）。
MessageSizeTooLarge	10		当服务器配置了一个最大消息长度以避免无限制的内存分配时，客户端产生了一个超过这个最大值的消息会抛出此错误。
StaleControllerEpochCode	11		broker 之间内部通讯是的错误。
OffsetMetadataTooLargeCode	12		如果你赋了一个超过所配置的最大偏移量元数据的字符串时触发。
GroupLoadInProgressCode	14	Yes	broker会在以下情况下返回这个错误：当broker人在加载偏移量时（主题分区的leader发生变化后）请求偏移量获取请求；或者正在反馈组成员请求（比如心跳）时，组的元数据正在被协调器加载。
GroupCoordinatorNotAvailableCode	15	Yes	组协调器请求，偏移量提交和大部分组管理请求时，偏移量主题还没有被建立或者组协调器还没有激活是broker会返回此错误。
NotCoordinatorForGroupCode	16	Yes	非该组协调器的broker接收到一个偏移量获取或提交请求时返回此错误。
InvalidTopicCode	17		请求指令尝试访问一个非法的主题（例如，一个包含非法名称的主题），或者尝试写入一个内部主题（例如消费者偏移量主题）。
RecordListTooLargeCode	18		批处理消息片段数组的长度超过了配置的最大消息片段数。
NotEnoughReplicasCod	19	Yes	当同步中的副本数量小

错误名称（Error）	编码（Code）	是否可重试（Retriable）	描述
e			于配置的最小数量，并且requiredAcks设置为-1时返回此错误
NotEnoughReplicasAfterAppendCode	20	Yes	消息已经写入日志文件，但是同步中的副本数量比请求中要求的数量少时返回此错误码
InvalidRequiredAcksCode	21		请求的requiredAcks非法（任何非-1，1或者0）时返回此错误码。
IllegalGenerationCode	22		组籍管理请求（诸如心跳请求）时generation id不是与当前不一致时返回此错误码
InconsistentGroupProtocolCode	23		加入组请求时成员提供的协议类型或者协议类型组与当前组不兼容时返回。
InvalidGroupIdCode	24		加入组请求时groupId为空或者null是返回。
UnknownMemberIdCode	25		组请求（偏移量提交/获取，心跳等）时memberId不在当前的generation。
InvalidSessionTimeoutCode	26		加入组请求时请求的会话超时超过broker允许的限制。
RebalanceInProgressCode	27		心跳请求时协调器已经开始了组的再平衡，这意味着客户端必须重新加入组。
InvalidCommitOffsetSizeCode	28		这个错意味着偏移量提交因为超过了元数据大小而被拒绝。
TopicAuthorizationFailedCode	29		客户端没有访问请求主题的权限时，broker返回此错误。
GroupAuthorizationFailedCode	30		客户端没有访问特定groupId的权限时，broker返回此错误。
ClusterAuthorizationFailedCode	31		客户端没有权限访问broker之间的接口或者管理接口时，broker返回此错误。

一些常见的哲学问题

有些人问我们为什么不使用
HTTP。有很多原因，最主要的是客户端实现者可以使用一些更高级的 TCP 功能 -
多路复用请求（multiplex
requests）的能力，同时轮询多个连接的能力等。我们还发现许多语言的 HTTP
库还在使用很旧的一些东西。

还有人问我们是否应该支持许多不同的协议。此前的经验是，多协议支持的是很难添加和测试新功能，因为他们要被移植到许多协议实现中。我们感觉大多数用户并不在乎支持多个协议这些特性，他们只是希望在自己选择的语言中实现了良好可靠的客户端。

另一个问题是，为什么我们不采用 XMPP，STOMP，AMQP 或现有的协议。对此的答案因协议而异，但共同的问题是，这些协议确实实现了的大部分内容，但如果我们无法控制协议，我们就无法做我们正在做的事情。我们的信念是，我们可以实现比现有消息系统更好的真正的分布式消息系统，但要做到这一点，我们需要建立不同的工作模式。

最后一个问题是，为什么我们不使用 Protocol Buffers 或 Thrift 来定义我们的请求消息格式。这些库擅长帮助您管理非常多的序列化的消息。然而，我们只有几个消息。而且这些库跨语言的支持是有点参差不齐（取决于软件包）。最后，我们颇为谨慎地管理二进制日志格式和传输协议之间的映射，这对于这些系统来说是不可能的。最后，我们比较喜欢让 API
有明确的版本并且通过检查版来引入原本为空的新值，因为它能更细致地控制兼容性。