二

13 Java生产者是如何管理TCP连接的?

你好，我是胡夕。今天我要和你分享的主题是：Kafka 的 Java 生产者是如何管理 TCP 连接的。

为何采用 TCP?

Apache Kafka 的所有通信都是基于 TCP 的，而不是基于 HTTP 或其他协议。无论是生产者、消费者，还是 Broker 之间的通信都是如此。你可能会问，为什么 Kafka 不使用 HTTP 作为底层的通信协议呢？其实这里面的原因有很多，但最主要的原因在于 TCP 和 HTTP 之间的区别。

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，比如多路复用请求以及同时轮询多个连接的能力。

所谓的多路复用请求，即 multiplexing request，是指将两个或多个数据流合并到底层单一物理连接中的过程。TCP 的多路复用请求会在一条物理连接上创建若干个虚拟连接，每个虚拟连接负责流转各自对应的数据流。其实严格来说，TCP 并不能多路复用，它只是提供可靠的消息交付语义保证，比如自动重传丢失的报文。

更严谨地说，作为一个基于报文的协议，TCP 能够被用于多路复用连接场景的前提是，上层的应用协议（比如 HTTP）允许发送多条消息。不过，我们今天并不是要详细讨论 TCP 原理，因此你只需要知道这是社区采用 TCP 的理由之一就行了。

除了 TCP 提供的这些高级功能有可能被 Kafka 客户端的开发人员使用之外，社区还发现，目前已知的 HTTP 库在很多编程语言中都略显简陋。

基于这两个原因，Kafka 社区决定采用 TCP 协议作为所有请求通信的底层协议。

Kafka 生产者程序概览

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。

第 1 步：构造生产者对象所需的参数对象。

第 2 步：利用第 1 步的参数对象，创建 KafkaProducer 对象实例。

第 3 步：使用 KafkaProducer 的 send 方法发送消息。

第 4 步：调用 KafkaProducer 的 close 方法关闭生产者并释放各种系统资源。

上面这 4 步写成 Java 代码的话大概是这个样子：

```
Properties props = new Properties ();
props.put(“参数 1”, “参数 1 的值”);
props.put(“参数 2”, “参数 2 的值”);
……
try (Producer<String, String> producer = new KafkaProducer<>(props)) {
            producer.send(new ProducerRecord<String, String>(……), callback);
        ……
}
```

这段代码使用了 Java 7 提供的 try-with-resource 特性，所以并没有显式调用 producer.close() 方法。无论是否显式调用 close 方法，所有生产者程序大致都是这个路数。

现在问题来了，当我们开发一个 Producer 应用时，生产者会向 Kafka 集群中指定的主题（Topic）发送消息，这必然涉及与 Kafka Broker 创建 TCP 连接。那么，Kafka 的 Producer 客户端是如何管理这些 TCP 连接的呢?

何时创建 TCP 连接?

要回答上面这个问题，我们首先要弄明白生产者代码是什么时候创建 TCP 连接的。就上面的那段代码而言，可能创建 TCP 连接的地方有两处：Producer producer = new KafkaProducer(props) 和 producer.send(msg, callback)。你觉得连向 Broker 端的 TCP 连接会是哪里创建的呢？前者还是后者，抑或是两者都有？请先思考 5 秒钟，然后我给出我的答案。

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。其实这种表述也不是很准确，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一个名为 Sender 的线程，该 Sender 线程开始运行时首先会创建与 Broker 的连接。**我截取了一段测试环境中的日志来说明这一点：

> [2018-12-09 09:35:45,620] DEBUG [Producer clientId=producer-1] Initialize

> connection to node localhost:9093 (id: -2 rack: null) for sending metadata request (org.apache.kafka.clients.NetworkClient:1084)

> [2018-12-09 09:35:45,622] DEBUG [Producer clientId=producer-1] Initiating connection to node localhost:9093 (id: -2 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:914)

> [2018-12-09 09:35:45,814] DEBUG [Producer clientId=producer-1] Initialize connection to node localhost:9092 (id: -1 rack: null) for sending metadata request (org.apache.kafka.clients.NetworkClient:1084)

> [2018-12-09 09:35:45,815] DEBUG [Producer clientId=producer-1] Initiating connection to node localhost:9092 (id: -1 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:914)

> [2018-12-09 09:35:45,828] DEBUG [Producer clientId=producer-1] Sending metadata request (type=MetadataRequest, topics=) to node localhost:9093 (id: -2 rack: null) (org.apache.kafka.clients.NetworkClient:1068)

你也许会问：怎么可能是这样？如果不调用 send 方法，这个 Producer 都不知道给哪个主题发消息，它又怎么能知道连接哪个 Broker 呢？难不成它会连接 bootstrap.servers 参数指定的所有 Broker 吗？嗯，是的，Java Producer 目前还真是这样设计的。

我在这里稍微解释一下 bootstrap.servers 参数。它是 Producer 的核心参数之一，指定了这个 Producer 启动时要连接的 Broker 地址。请注意，这里的“启动时”，代表的是 Producer 启动时会发起与这些 Broker 的连接。因此，如果你为这个参数指定了 1000 个 Broker 连接信息，那么很遗憾，你的 Producer 启动时会首先创建与这 1000 个 Broker 的 TCP 连接。

在实际使用过程中，我并不建议把集群中所有的 Broker 信息都配置到 bootstrap.servers 中，通常你指定 3～4 台就足以了。因为 Producer 一旦连接到集群中的任一台 Broker，就能拿到整个集群的 Broker 信息，故没必要为 bootstrap.servers 指定所有的 Broker。

让我们回顾一下上面的日志输出，请注意我标为橙色的内容。从这段日志中，我们可以发现，在 KafkaProducer 实例被创建后以及消息被发送前，Producer 应用就开始创建与两台 Broker 的 TCP 连接了。当然了，在我的测试环境中，我为 bootstrap.servers 配置了 localhost:9092、localhost:9093 来模拟不同的 Broker，但是这并不影响后面的讨论。另外，日志输出中的最后一行也很关键：它表明 Producer 向某一台 Broker 发送了 METADATA 请求，尝试获取集群的元数据信息——这就是前面提到的 Producer 能够获取集群所有信息的方法。

讲到这里，我有一些个人的看法想跟你分享一下。通常情况下，我都不认为社区写的代码或做的设计就一定是对的，因此，很多类似的这种“质疑”会时不时地在我脑子里冒出来。

拿今天的这个 KafkaProducer 创建实例来说，社区的官方文档中提及 KafkaProducer 类是线程安全的。我本人并没有详尽地去验证过它是否真的就是 thread-safe 的，但是大致浏览一下源码可以得出这样的结论：KafkaProducer 实例创建的线程和前面提到的 Sender 线程共享的可变数据结构只有 RecordAccumulator 类，故维护了 RecordAccumulator 类的线程安全，也就实现了 KafkaProducer 类的线程安全。

你不需要了解 RecordAccumulator 类是做什么的，你只要知道它主要的数据结构是一个 ConcurrentMap<TopicPartition, Deque>。TopicPartition 是 Kafka 用来表示主题分区的 Java 对象，本身是不可变对象。而 RecordAccumulator 代码中用到 Deque 的地方都有锁的保护，所以基本上可以认定 RecordAccumulator 类是线程安全的。

说了这么多，我其实是想说，纵然 KafkaProducer 是线程安全的，我也不赞同创建 KafkaProducer 实例时启动 Sender 线程的做法。写了《Java 并发编程实践》的那位布赖恩·格茨（Brian Goetz）大神，明确指出了这样做的风险：在对象构造器中启动线程会造成 this 指针的逃逸。理论上，Sender 线程完全能够观测到一个尚未构造完成的 KafkaProducer 实例。当然，在构造对象时创建线程没有任何问题，但最好是不要同时启动它。

好了，我们言归正传。针对 TCP 连接何时创建的问题，目前我们的结论是这样的：**TCP 连接是在创建 KafkaProducer 实例时建立的。**那么，我们想问的是，它只会在这个时候被创建吗？

当然不是！**TCP 连接还可能在两个地方被创建：一个是在更新元数据后，另一个是在消息发送时。**为什么说是可能？因为这两个地方并非总是创建 TCP 连接。当 Producer 更新了集群的元数据信息之后，如果发现与某些 Broker 当前没有连接，那么它就会创建一个 TCP 连接。同样地，当要发送消息时，Producer 发现尚不存在与目标 Broker 的连接，也会创建一个。

接下来，我们来看看 Producer 更新集群元数据信息的两个场景。

场景一：当 Producer 尝试给一个不存在的主题发送消息时，Broker 会告诉 Producer 说这个主题不存在。此时 Producer 会发送 METADATA 请求给 Kafka 集群，去尝试获取最新的元数据信息。

场景二：Producer 通过 metadata.max.age.ms 参数定期地去更新元数据信息。该参数的默认值是 300000，即 5 分钟，也就是说不管集群那边是否有变化，Producer 每 5 分钟都会强制刷新一次元数据以保证它是最及时的数据。

讲到这里，我们可以“挑战”一下社区对 Producer 的这种设计的合理性。目前来看，一个 Producer 默认会向集群的所有 Broker 都创建 TCP 连接，不管是否真的需要传输请求。这显然是没有必要的。再加上 Kafka 还支持强制将空闲的 TCP 连接资源关闭，这就更显得多此一举了。

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 Broker 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的空间的。

何时关闭 TCP 连接?

说完了 TCP 连接的创建，我们来说说它们何时被关闭。

Producer 端关闭 TCP 连接的方式有两种：**一种是用户主动关闭；一种是 Kafka 自动关闭。**

我们先说第一种。这里的主动关闭实际上是广义的主动关闭，甚至包括用户调用 kill -9 主动“杀掉”Producer 应用。当然最推荐的方式还是调用 producer.close() 方法来关闭。

第二种是 Kafka 帮你关闭，这与 Producer 端参数 connections.max.idle.ms 的值有关。默认情况下该参数值是 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 TCP 连接关闭。用户可以在 Producer 端设置 connections.max.idle.ms=-1 禁掉这种机制。一旦被设置成 -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 Socket 连接都开启了 keepalive，因此 keepalive 探活机制还是会遵守的。

值得注意的是，在第二种方式中，TCP 连接是在 Broker 端被关闭的，但其实这个 TCP 连接的发起方是客户端，因此在 TCP 看来，这属于被动关闭的场景，即 passive close。被动关闭的后果就是会产生大量的 CLOSE_WAIT 连接，因此 Producer 端或 Client 端没有机会显式地观测到此连接已被中断。

小结

我们来简单总结一下今天的内容。对最新版本的 Kafka（2.1.0）而言，Java Producer 端管理 TCP 连接的方式是：

1. KafkaProducer 实例创建时启动 Sender 线程，从而创建与 bootstrap.servers 中所有 Broker 的 TCP 连接。

2. KafkaProducer 实例首次更新元数据信息之后，还会再次创建与集群中所有 Broker 的 TCP 连接。
3. 如果 Producer 端发送消息到某台 Broker 时发现没有与该 Broker 的 TCP 连接，那么也会立即创建连接。
4. 如果设置 Producer 端 connections.max.idle.ms 参数大于 0，则步骤 1 中创建的 TCP 连接会被自动关闭；如果设置该参数 =-1，那么步骤 1 中创建的 TCP 连接将无法被关闭，从而成为“僵尸”连接。

上一页 下一页