二

21 Java 消费者是如何管理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 消费者是如何管理 TCP 连接的。

在专栏[第 13 讲]中，我们专门聊过“Java**生产者**是如何管理 TCP 连接资源的”这个话题，你应该还有印象吧？今天算是它的姊妹篇，我们一起来研究下 Kafka 的 Java**消费者**管理 TCP 或 Socket 资源的机制。只有完成了今天的讨论，我们才算是对 Kafka 客户端的 TCP 连接管理机制有了全面的了解。

和之前一样，我今天会无差别地混用 TCP 和 Socket 两个术语。毕竟，在 Kafka 的世界中，无论是 ServerSocket，还是 SocketChannel，它们实现的都是 TCP 协议。或者这么说，Kafka 的网络传输是基于 TCP 协议的，而不是基于 UDP 协议，因此，当我今天说到 TCP 连接或 Socket 资源时，我指的是同一个东西。

何时创建 TCP 连接?

我们先从消费者创建 TCP 连接开始讨论。消费者端主要的程序入口是 KafkaConsumer 类。**和生产者不同的是，构建 KafkaConsumer 实例时是不会创建任何 TCP 连接的**，也就是说，当你执行完 new KafkaConsumer(properties) 语句后，你会发现，没有 Socket 连接被创建出来。这一点和 Java 生产者是有区别的，主要原因就是生产者入口类 KafkaProducer 在构建实例的时候，会在后台默默地启动一个 Sender 线程，这个 Sender 线程负责 Socket 连接的创建。

从这一点上来看，我个人认为 KafkaConsumer 的设计比 KafkaProducer 要好。就像我在第 13 讲中所说的，在 Java 构造函数中启动线程，会造成 this 指针的逃逸，这始终是一个隐患。

如果 Socket 不是在构造函数中创建的，那么是在 KafkaConsumer.subscribe 或 KafkaConsumer.assign 方法中创建的吗？严格来说也不是。我还是直接给出答案吧：**TCP 连接是在调用 KafkaConsumer.poll 方法时被创建的。**再细粒度地说，在 poll 方法内部有 3 个时机可以创建 TCP 连接。

1.**发起 FindCoordinator 请求时。**

还记得消费者端有个组件叫协调者（Coordinator）吗？它驻留在 Broker 端的内存中，负责消费者组的组成员管理和各个消费者的位移提交管理。当消费者程序首次启动调用 poll 方法时，它需要向 Kafka 集群发送一个名为 FindCoordinator 的请求，希望 Kafka 集群告诉它哪个 Broker 是管理它的协调者。

不过，消费者应该向哪个 Broker 发送这类请求呢？理论上任何一个 Broker 都能回答这个问题，也就是说消费者可以发送 FindCoordinator 请求给集群中的任意服务器。在这个问题上，社区做了一点点优化：消费者程序会向集群中当前负载最小的那台 Broker 发送请求。负载是如何评估的呢？其实很简单，就是看消费者连接的所有 Broker 中，谁的待发送请求最少。当然了，这种评估显然是消费者端的单向评估，并非是站在全局角度，因此有的时候也不一定是最优解。不过这不并影响我们的讨论。总之，在这一步，消费者会创建一个 Socket 连接。

2.**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），显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，会创建连向该 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，比如加入组、等待组分配方案、心跳请求处理、位移获取、位移提交等。

3.**消费数据时。**

消费者会为每个要消费的分区创建与该分区领导者副本所在 Broker 连接的 TCP。举个例子，假设消费者要消费 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会创建与这 4 台 Broker 的 Socket 连接。

创建多少个 TCP 连接?

下面我们来说说消费者创建 TCP 连接的数量。你可以先思考一下大致需要的连接数量，然后我们结合具体的 Kafka 日志，来验证下结果是否和你想的一致。

我们来看看这段日志。

> *[2019-05-27 10:00:54,142] DEBUG [Consumer clientId=consumer-1, groupId=test] Initiating connection to node localhost:9092 (id: -1 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:944)*

> *…*

> *[2019-05-27 10:00:54,188] DEBUG [Consumer clientId=consumer-1, groupId=test] Sending metadata request MetadataRequestData(topics=[MetadataRequestTopic(name='t4')], allowAutoTopicCreation=true, includeClusterAuthorizedOperations=false, includeTopicAuthorizedOperations=false) to node localhost:9092 (id: -1 rack: null) (org.apache.kafka.clients.NetworkClient:1097)*

> *…*

> *[2019-05-27 10:00:54,188] TRACE [Consumer clientId=consumer-1, groupId=test] Sending FIND_COORDINATOR {key=test,key_type=0} with correlation id 0 to node -1 (org.apache.kafka.clients.NetworkClient:496)*

> *[2019-05-27 10:00:54,203] TRACE [Consumer clientId=consumer-1, groupId=test] Completed receive from node -1 for FIND_COORDINATOR with correlation id 0, received {throttle_time_ms=0,error_code=0,error_message=null, node_id=2,host=localhost,port=9094} (org.apache.kafka.clients.NetworkClient:837)*

> *…*

> *[2019-05-27 10:00:54,204] DEBUG [Consumer clientId=consumer-1, groupId=test] Initiating connection to node localhost:9094 (id: 2147483645 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:944)*

> *…*

> *[2019-05-27 10:00:54,237] DEBUG [Consumer clientId=consumer-1, groupId=test] Initiating connection to node localhost:9094 (id: 2 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:944)*

> *[2019-05-27 10:00:54,237] DEBUG [Consumer clientId=consumer-1, groupId=test] Initiating connection to node localhost:9092 (id: 0 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:944)*

> *[2019-05-27 10:00:54,238] DEBUG [Consumer clientId=consumer-1, groupId=test] Initiating connection to node localhost:9093 (id: 1 rack: null) using address localhost/127.0.0.1 (org.apache.kafka.clients.NetworkClient:944)*

这里我稍微解释一下，日志的第一行是消费者程序创建的第一个 TCP 连接，就像我们前面说的，这个 Socket 用于发送 FindCoordinator 请求。由于这是消费者程序创建的第一个连接，此时消费者对于要连接的 Kafka 集群一无所知，因此它连接的 Broker 节点的 ID 是 -1，表示消费者根本不知道要连接的 Kafka Broker 的任何信息。

值得注意的是日志的第二行，消费者复用了刚才创建的那个 Socket 连接，向 Kafka 集群发送元数据请求以获取整个集群的信息。

日志的第三行表明，消费者程序开始发送 FindCoordinator 请求给第一步中连接的 Broker，即 localhost:9092，也就是 nodeId 等于 -1 的那个。在十几毫秒之后，消费者程序成功地获悉协调者所在的 Broker 信息，也就是第四行标为橙色的“node_id = 2”。

完成这些之后，消费者就已经知道协调者 Broker 的连接信息了，因此在日志的第五行发起了第二个 Socket 连接，创建了连向 localhost:9094 的 TCP。只有连接了协调者，消费者进程才能正常地开启消费者组的各种功能以及后续的消息消费。

在日志的最后三行中，消费者又分别创建了新的 TCP 连接，主要用于实际的消息获取。还记得我刚才说的吗？要消费的分区的领导者副本在哪台 Broker 上，消费者就要创建连向哪台 Broker 的 TCP。在我举的这个例子中，localhost:9092，localhost:9093 和 localhost:9094 这 3 台 Broker 上都有要消费的分区，因此消费者创建了 3 个 TCP 连接。

看完这段日志，你应该会发现日志中的这些 Broker 节点的 ID 在不断变化。有时候是 -1，有时候是 2147483645，只有在最后的时候才回归正常值 0、1 和 2。这又是怎么回事呢？

前面我们说过了 -1 的来由，即消费者程序（其实也不光是消费者，生产者也是这样的机制）首次启动时，对 Kafka 集群一无所知，因此用 -1 来表示尚未获取到 Broker 数据。

那么 2147483645 是怎么来的呢？它是**由 Integer.MAX_VALUE 减去协调者所在 Broker 的真实 ID 计算得来的**。看第四行标为橙色的内容，我们可以知道协调者 ID 是 2，因此这个 Socket 连接的节点 ID 就是 Integer.MAX_VALUE 减去 2，即 2147483647 减去 2，也就是 2147483645。这种节点 ID 的标记方式是 Kafka 社区特意为之的结果，目的就是要让组协调请求和真正的数据获取请求使用不同的 Socket 连接。

至于后面的 0、1、2，那就很好解释了。它们表征了真实的 Broker ID，也就是我们在 server.properties 中配置的 broker.id 值。

我们来简单总结一下上面的内容。通常来说，消费者程序会创建 3 类 TCP 连接：

1. 确定协调者和获取集群元数据。
2. 连接协调者，令其执行组成员管理操作。
3. 执行实际的消息获取。

那么，这三类 TCP 请求的生命周期都是相同的吗？换句话说，这些 TCP 连接是何时被关闭的呢？

何时关闭 TCP 连接？

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。主动关闭是指你显式地调用消费者 API 的方法去关闭消费者，具体方式就是**手动调用 KafkaConsumer.close() 方法，或者是执行 Kill 命令**，不论是 Kill -2 还是 Kill -9；而 Kafka 自动关闭是由**消费者端参数 connection.max.idle.ms**控制的，该参数现在的默认值是 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，那么消费者会强行“杀掉”这个 Socket 连接。

不过，和生产者有些不同的是，如果在编写消费者程序时，你使用了循环的方式来调用 poll 方法消费消息，那么上面提到的所有请求都会被定期发送到 Broker，因此这些 Socket 连接上总是能保证有请求在发送，从而也就实现了“长连接”的效果。

针对上面提到的三类 TCP 连接，你需要注意的是，**当第三类 TCP 连接成功创建后，消费者程序就会废弃第一类 TCP 连接**，之后在定期请求元数据时，它会改为使用第三类 TCP 连接。也就是说，最终你会发现，第一类 TCP 连接会在后台被默默地关闭掉。对一个运行了一段时间的消费者程序来说，只会有后面两类 TCP 连接存在。

可能的问题

从理论上说，Kafka Java 消费者管理 TCP 资源的机制我已经说清楚了，但如果仔细推敲这里面的设计原理，还是会发现一些问题。

我们刚刚讲过，第一类 TCP 连接仅仅是为了首次获取元数据而创建的，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，只能使用一个“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 对应的是哪台 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。

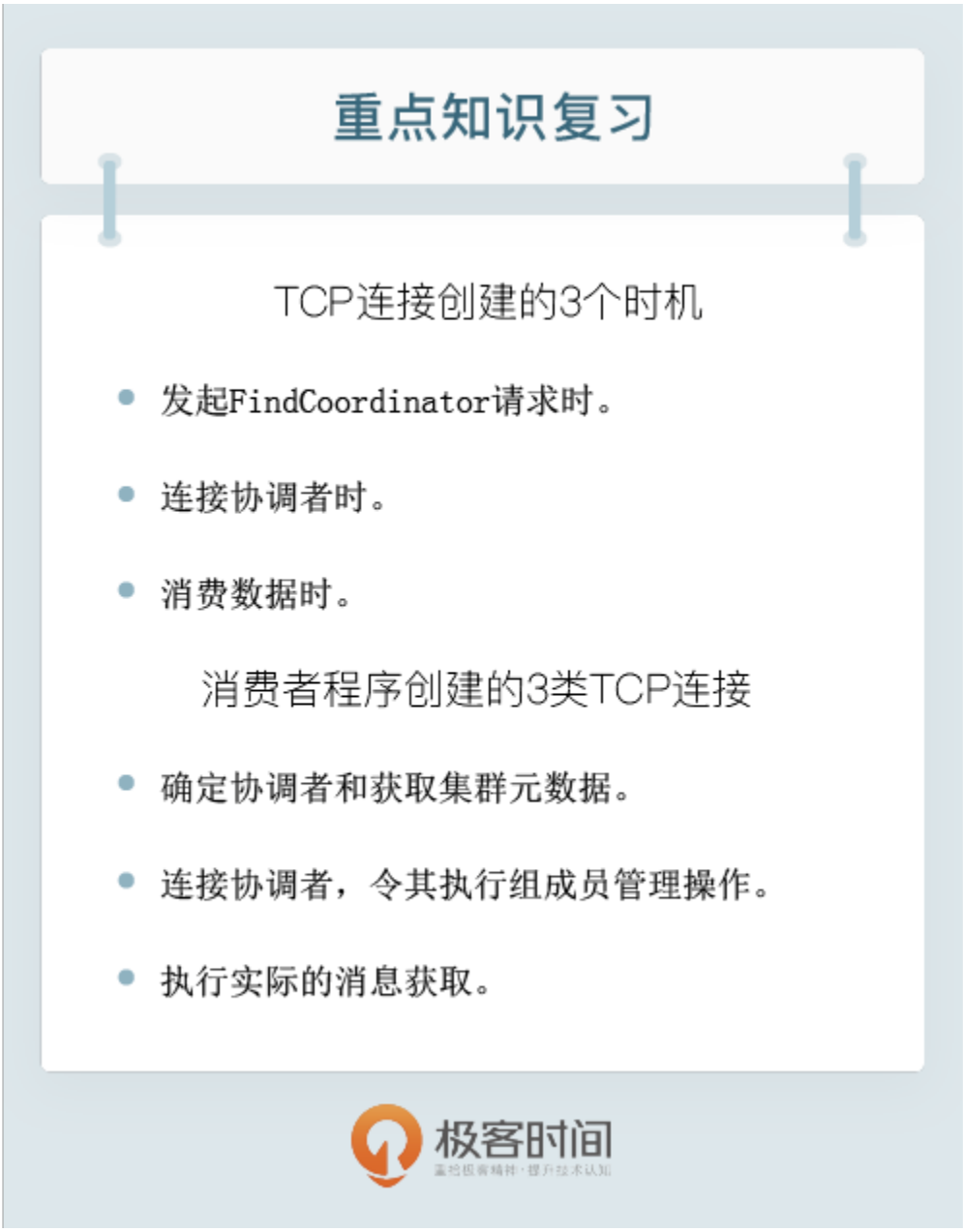
为什么会出现这种情况呢？主要是因为目前 Kafka 仅仅使用 ID 这一个维度的数据来表征 Socket 连接信息。这点信息明显不足以确定连接的是哪台 Broker，也许在未来，社区应该考虑使用**< 主机名、端口、ID>**三元组的方式来定位 Socket 资源，这样或许能够让消费者程序少创建一些 TCP 连接。

也许你会问，反正 Kafka 有定时关闭机制，这算多大点事呢？其实，在实际场景中，我见过很多将 connection.max.idle.ms 设置成 -1，即禁用定时关闭的案例，如果是这样的话，这些 TCP 连接将不会被定期清除，只会成为永久的“僵尸”连接。基于这个原因，社区应该

考虑更好的解决方案。

小结

好了，今天我们补齐了 Kafka Java 客户端管理 TCP 连接的“拼图”。我们不仅详细描述了 Java 消费者是怎么创建和关闭 TCP 连接的，还对目前的设计方案提出了一些自己的思考。希望今后你能将这些知识应用到自己的业务场景中，并对实际生产环境中的 Socket 管理做到心中有数。



上一页 下一页