20 | 多线程开发消费者实例

2019-07-18 胡夕

你好，我是胡夕。今天我们来聊聊Kafka Java Consumer端多线程消费的实现方案。

目前，计算机的硬件条件已经大大改善，即使是在普通的笔记本电脑上，多核都已经是标配了，更不用说专业的服务器了。如果跑在强劲服务器机器上的应用程序依然是单线程架构，那实在是有点暴殄天物了。不过，Kafka Java Consumer就是单线程的设计，你是不是感到很惊讶。所以，探究它的多线程消费方案，就显得非常必要了。

Kafka Java Consumer设计原理

在开始探究之前，我先简单阐述下Kafka Java Consumer为什么采用单线程的设计。了解了这一点，对我们后面制定多线程方案大有裨益。

谈到Java Consumer API，最重要的当属它的入口类KafkaConsumer了。我们说KafkaConsumer是单线程的设计，严格来说这是不准确的。因为，从Kafka 0.10.1.0版本开始，KafkaConsumer就变为了双线程的设计，即**用户主线程和心跳线程**。

所谓用户主线程，就是你启动Consumer应用程序main方法的那个线程，而新引入的心跳线程（Heartbeat Thread）只负责定期给对应的Broker机器发送心跳请求，以标识消费者应用的存活性（liveness）。引入这个心跳线程还有一个目的，那就是期望它能将心跳频率与主线程调用KafkaConsumer.poll方法的频率分开，从而解耦真实的消息处理逻辑与消费者组成员存活性管理。

不过，虽然有心跳线程，但实际的消息获取逻辑依然是在用户主线程中完成的。因此，在消费消息的这个层面上，我们依然可以安全地认为KafkaConsumer是单线程的设计。

其实，在社区推出Java Consumer API之前，Kafka中存在着一组统称为Scala Consumer的API。这组API，或者说这个Consumer，也被称为老版本Consumer，目前在新版的Kafka代码中已经被完全移除了。

我之所以重提旧事，是想告诉你，老版本Consumer是多线程的架构，每个Consumer实例在内部为所有订阅的主题分区创建对应的消息获取线程，也称Fetcher线程。老版本Consumer同时也是阻塞式的（blocking），Consumer实例启动后，内部会创建很多阻塞式的消息获取迭代器。但在很多场景下，Consumer端是有非阻塞需求的，比如在流处理应用中执行过滤（filter）、连接（join）、分组（group by）等操作时就不能是阻塞式的。基于这个原因，社区为新版本Consumer设计了单线程+轮询的机制。这种设计能够较好地实现非阻塞式的消息获取。

除此之外，单线程的设计能够简化Consumer端的设计。Consumer获取到消息后，处理消息的逻辑是否采用多线程，完全由你决定。这样，你就拥有了把消息处理的多线程管理策略从Consumer端代码中剥离的权利。

另外，不论使用哪种编程语言，单线程的设计都比较容易实现。相反，并不是所有的编程语言都能够很好地支持多线程。从这一点上来说，单线程设计的Consumer更容易移植到其他语言上。毕竟，Kafka社区想要打造上下游生态的话，肯定是希望出现越来越多的客户端的。

多线程方案

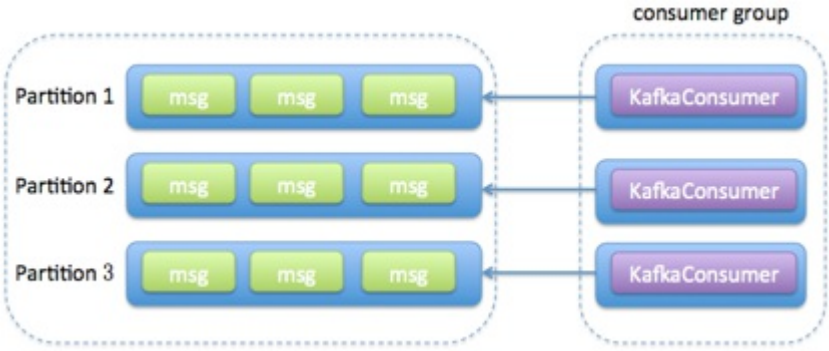
了解了单线程的设计原理之后，我们来具体分析一下KafkaConsumer这个类的使用方法，以及如何推演出对应的多线程方案。

首先，我们要明确的是，KafkaConsumer类不是线程安全的(thread-safe)。所有的网络I/O处理都是发生在用户主线程中，因此，你在使用过程中必须要确保线程安全。简单来说，就是你不能在多个线程中共享同一个KafkaConsumer实例，否则程序会抛出ConcurrentModificationException异常。

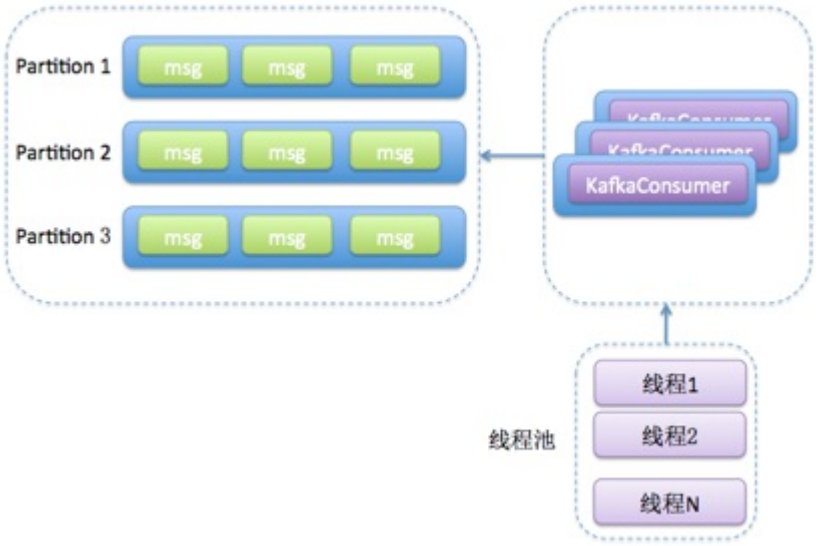
当然了，这也不是绝对的。KafkaConsumer中有个方法是例外的，它就是**wakeup()**，你可以在其他线程中安全地调用**KafkaConsumer.wakeup()**来唤醒Consumer。

鉴于KafkaConsumer不是线程安全的事实，我们能够制定两套多线程方案。

1. **消费者程序启动多个线程，每个线程维护专属的KafkaConsumer实例，负责完整的消息获取、消息处理流程**。如下图所示：



1. **消费者程序使用单或多线程获取消息，同时创建多个消费线程执行消息处理逻辑**。获取消息的线程可以是一个，也可以是多个，每个线程维护专属的KafkaConsumer实例，处理消息则交由**特定的线程池**来做，从而实现消息获取与消息处理的真正解耦。具体架构如下图所示：



总体来说，这两种方案都会创建多个线程，这些线程都会参与到消息的消费过程中，但各自的思路是不一样的。

我们来打个比方。比如一个完整的消费者应用程序要做的事情是1、2、3、4、5，那么方案1的思路是**粗粒度化**的工作划分，也就是说方案1会创建多个线程，每个线程完整地执行1、2、3、4、5，以实现并行处理的目标，它不会进一步分割具体的子任务；而方案2则更**细粒度化**，它会将1、2分割出来，用单线程（也可以是多线程）来做，对于3、4、5，则用另外的多个线程来做。

这两种方案孰优孰劣呢？应该说是各有千秋。我总结了一下这两种方案的优缺点，我们先来看看下面这张表格。

方案	优点	缺点
方案1： **多线程+多KafkaConsumer实例**	方便实现	占用更多系统资源
	速度快，无线程间交互开销	线程数受限于主题分区数，扩展性差
	易于维护分区内的消费顺序	线程自己处理消息容易超时，从而引发Rebalance
方案2： **单线程 + 单KafkaConsumer实例 + 消息处理Worker线程池**	可独立扩展消费获取线程数和Worker线程数	实现难度高
	伸缩性好	难以维护分区内的消息消费顺序
		处理链路拉长，不易于位移提交管理

接下来，我来具体解释一下表格中的内容。

我们先看方案1，它的优势有3点。

1. 实现起来简单，因为它比较符合目前我们使用Consumer API的习惯。我们在写代码的时候，使用多个线程并在每个线程中创建专属的KafkaConsumer实例就可以了。
2. 多个线程之间彼此没有任何交互，省去了很多保障线程安全方面的开销。
3. 由于每个线程使用专属的KafkaConsumer实例来执行消息获取和消息处理逻辑，因此，Kafka主题中的每个分区都能保证只被一个线程处理，这样就很容易实现分区内的消息消费顺序。这对在乎事件先后顺序的应用场景来说，是非常重要的优势。

说完了方案1的优势，我们来看看这个方案的不足之处。

1. 每个线程都维护自己的KafkaConsumer实例，必然会占用更多的系统资源，比如内存、TCP连接等。在资源紧张的系统环境中，方案1的这个劣势会表现得更加明显。
2. 这个方案能使用的线程数受限于Consumer订阅主题的总分区数。我们知道，在一个消费者组中，每个订阅分区都只能被组内的一个消费者实例所消费。假设一个消费者组订阅了100个分区，那么方案1最多只能扩展到100个线程，多余的线程无法分配到任何分区，只会白白消耗系统资源。当然了，这种扩展性方面的局限可以被多机架构所缓解。除了在一台机器上启用100个线程消费数据，我们也可以选择在100台机器上分别创建1个线程，效果是一样的。因此，如果你的机器资源很丰富，这个劣势就不足为虑了。
3. 每个线程完整地执行消息获取和消息处理逻辑。一旦消息处理逻辑很重，造成消息处理速度慢，就很容易出现不必要的Rebalance，从而引发整个消费者组的消费停滞。这个劣势你一定要注意。我们之前讨论过如何避免Rebalance，如果你不记得的话，可以回到专栏第17讲复习一下。

下面我们来说说方案2。

与方案1的粗粒度不同，方案2将任务切分成了**消息获取**和**消息处理**两个部分，分别由不同的线程处理它们。比起方案1，方案2的最大优势就在于它的**高伸缩性**，就是说我们可以独立地调节消息获取的线程数，以及消息处理的线程数，而不必考虑两者之间是否相互影响。如果你的消费获取速度慢，那么增加消费获取的线程数即可；如果是消息的处理速度慢，那么增加Worker线程池线程数即可。

不过，这种架构也有它的缺陷。

1. 它的实现难度要比方案1大得多，毕竟它有两组线程，你需要分别管理它们。
2. 因为该方案将消息获取和消息处理分开了，也就是说获取某条消息的线程不是处理该消息的线程，因此无法保证分区内的消费顺序。举个例子，比如在某个分区中，消息1在消息2之前被保存，那么Consumer获取消息的顺序必然是消息1在前，消息2在后，但是，后面的Worker线程却有可能先处理消息2，再处理消息1，这就破坏了消息在分区中的顺序。还是那句话，如果你在意Kafka中消息的先后顺序，方案2的这个劣势是致命的。
3. 方案2引入了多组线程，使得整个消息消费链路被拉长，最终导致正确位移提交会变得异常困难，结果就是可能会出现消息的重复消费。如果你在意这一点，那么我不推荐你使用方案2。

实现代码示例

讲了这么多纯理论的东西，接下来，我们来看看实际的实现代码大概是什么样子。毕竟，就像Linus说的：“Talk is cheap, show me the code!”

我先跟你分享一段方案1的主体代码：

```
public class KafkaConsumerRunner implements Runnable {
    private final AtomicBoolean closed = new AtomicBoolean(false);
    private final KafkaConsumer consumer;



    public void run() {
        try {
            consumer.subscribe(Arrays.asList("topic"));
            while (!closed.get()) {
   ConsumerRecords records =
    consumer.poll(Duration.ofMillis(10000));
                //  执行消息处理逻辑
            }
        } catch (WakeupException e) {
            // Ignore exception if closing
            if (!closed.get()) throw e;
        } finally {
            consumer.close();
        }
    }



    // Shutdown hook which can be called from a separate thread
    public void shutdown() {
        closed.set(true);
        consumer.wakeup();
    }
```

这段代码创建了一个Runnable类，表示执行消费获取和消费处理的逻辑。每个KafkaConsumerRunner类都会创建一个专属的KafkaConsumer实例。在实际应用中，你可以创建多个KafkaConsumerRunner实例，并依次执行启动它们，以实现方案1的多线程架构。

对于方案2来说，核心的代码是这样的：

```
private final KafkaConsumer<String, String> consumer;
private ExecutorService executors;
...


private int workerNum = ...;
executors = new ThreadPoolExecutor(
 workerNum, workerNum, 0L, TimeUnit.MILLISECONDS,
 new ArrayBlockingQueue<>(1000),
 new ThreadPoolExecutor.CallerRunsPolicy());


...
while (true)  {
 ConsumerRecords<String, String> records =
  consumer.poll(Duration.ofSeconds(1));
 for (final ConsumerRecord record : records) {
  executors.submit(new Worker(record));
 }
}
..
```

这段代码最重要的地方是我标为橙色的那个语句：当Consumer的poll方法返回消息后，由专门的线程池来负责处理具体的消息。调用poll方法的主线程不负责消息处理逻辑，这样就实现了方案2的多线程架构。

小结

总结一下，今天我跟你分享了Kafka Java Consumer多线程消费的实现方案。我给出了比较通用的两种方案，并介绍了它们各自的优缺点以及代码示例。我希望你能根据这些内容，结合你的实际业务场景，实现适合你自己的多线程架构，真正做到举一反三、融会贯通，彻底掌握多线程消费的精髓，从而在日后实现更宏大的系统。

开放讨论

今天我们讨论的都是多线程的方案，可能有人会说，何必这么麻烦，我直接启动多个Consumer进程不就得了？那么，请你比较一下多线程方案和多进程方案，想一想它们各自的优劣之处。

欢迎写下你的思考和答案，我们一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋

友。

精选留言

yhh 24

希望老师能讲讲方案2下线程池怎么管理和提交位移！！

2019-07-18

QQ怪 9

老师能否加餐spring-kafka相关知识

2019-07-18

leaning_人生 8

希望老师能对比spring-kafka源码，关于多线程管理consumer谢谢

2019-07-18

小生向北 6

能够用多线程解决的就不要用多进程，毕竟资源有限。方案2的讲解还是太浅了，同希望老师能针对方案2详细讲解一下！方案2里面在异步线程里提交offset，每个线程自己提交自己的，如果中间有offset提交失败，后面的offset又提交成功了咋办呢？而且每个线程都自己提交consumer.commit()就意味着要在多个线程里面使用consumer，如文中所说，这种情况是要报CME错误的，那究竟该如何正确的提交呢，有没有最佳实践呀？

2019-07-18

KEEPUP

2

希望老师讲一下sparkstreaming 消费kafka 消息的情况

2019-07-19

Xiao

2

胡老师，第二种方案我觉得还有个问题就是如果是自动提交，那就会容易出现消息丢失，因为异步消费消息，如果worker线程有异常，主线程捕获不到异常，就会造成消息丢失，这个是不是还需要做补偿机制；如果是手动提交，那offer set也会有可能丢失，消息重复消费，消息重复还好处理，做幂等就行。

2019-07-18

james

2

方案2最核心的如何commit老师没有说，难道只能启用自动提交吗？我觉得可以用Cyclicbarrier来实现线程池执行完毕后，由consumer来commit，不用countdownlatch因为它只能记录一次，而cb可以反复用，或者用forkjoin方式，总之要等待多线程都处理完才能commit，风险就是某个消息处理太慢回导致整体都不能commit，而触发rebalance以及重复消费，而重复消费我用布隆过滤器来解决

2019-07-18

来碗绿豆汤

1

对于第二种方案，可以添加一个共享的队列，消费线程消费完一个记录，就写入队列，然后主线程可以读取这个队列，然后依次提交小号的offset

2019-07-22

玉剑冰锋

1

Kafka重启时间比较长，每次重启一台差不多四五十分钟，日志保存12个小时，每台数据量差不多几个T，想请教一下老师有什么可以优化的参数吗？

2019-07-18

作者回复

有可能是要加载的日志段数据太多导致的，可以增加num.recovery.threads.per.data.dir的值

2019-07-19

可以

0

方案2会让我想到Reactor，同样是将拉取数据（上游）消费数据（下游），下游订阅上游的数据，调度器设置上游、下游的线程方式。所以想方案2的代码是否能用Reactor实现呢？还是我理解上有误？望指正

2019-07-31

金hb.Ryan 冷空氣駕到

0

我们方案2也是类似解决方案，主线程poll然后submit任务，多线程消费，如果消费延迟即队列满那么主线程仍然会wait，这样其实commit还是主线程commit逻辑。现在想到是不是可以一个partition一个线程池来保证可以异步同步commit？

2019-07-28

金hb.Ryan 冷空氣駕到

0

一般来说单个partition的获取速度是远远大于单线程的处理速度，所以一个partition consumer是必要有多个线程来并行处理来提高处理速度。当然单线程如果能够跟上那也没什么差别了

2019-07-28

千屿

0

最近用spring cloud做了一个kafka可靠消息微服务组件，有兴趣的朋友可以看看，消费端是多线程模型，消费线程和业务执行分离，使用了mongodb(分片+副本集) 存储消息发送的链路，对发送失败的消息做了补偿机制。https://gitee.com/huacke/mq-kafka，有问题可以联系我。

2019-07-20

z.l

0

请教个问题，如果使用方案1，一个consumer group订阅了2个topic，每个topic都是24个分区，此时最大线程数可以设置为24还是48？

2019-07-20

作者回复

理论上是48，但实际上这么多线程反而是开销，可以采用多进程的方式

2019-07-22

rm -rf

0

思考：
多进程上下文切换成本比较大，没多线程好。

另外，老师我想问问，方案1这种是消费者组吗？启动了多个消费者线程，会自动进行分区分配进行消费吗？

2019-07-20

作者回复

是的

2019-07-22

开水

0

方案一用在需要精确控制消费数量的方案里，比如访问量这种日志什么的。
方案二可以把后面处理通过数据库key做成幂等操作，用在实时处理需要随时增减消费能力的业务上面。

2019-07-19

丘壑

0

对于老师说的第二种多线程处理的方案，我本人觉得在消息量很大的系统中比较常用，只是在使用的时候很担心出现异常后的数据问题，数据应该怎么找回，这块对消费异常设计难度较大，请老师可以考虑分享下这块的手动提交位移及异常处理的经验

2019-07-19

Aaron亚伦

0

我觉得方案2下管理和提交移位跟处理消息的线程池是没有关系的。所以不管是手动提交还是自动提交还是KafkaConsumer的实例完成的。

2019-07-19

Liam

0

其实方案12可以结合，即启动多个consumer，每个consumer也可以分离接收和业务处理

2019-07-19

nightmare

0

方案1 位移提交好管理 方案2 位移提交不好环境 但是扩容更加方便 多进程消耗物理资源

2019-07-18