

## 20 | DelayedOperation: Broker是怎么延时处理请求的？

2020-06-09 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 26:22 大小 24.15M



你好，我是胡夕。

上节课，我们学习了分层时间轮在 Kafka 中的实现。既然是分层时间轮，那就说明，源码中构造的时间轮是有多个层次的。每一层所表示的总时长，等于该层 Bucket 数乘以每个 Bucket 涵盖的时间范围。另外，该总时长自动成为下一层单个 Bucket 所覆盖的时间范围。

举个例子，目前，Kafka 第 1 层的时间轮固定时长是 20 毫秒（interval），即有 20 个 Bucket（wheelSize），每个 Bucket 涵盖 1 毫秒（tickMs）的时间范围。第 2 层的总时长是 400 毫秒，同样有 20 个 Bucket，每个 Bucket 20 毫秒。依次类推，那么第 3 层的时间轮时长就是 8 秒，因为这一层单个 Bucket 的时长是 400 毫秒，共有 20 个 Bucket。

基于这种设计，每个延迟请求需要根据自己的超时时间，来决定它要被保存于哪一层时间轮上。我们假设在  $t=0$  时创建了第 1 层的时间轮，那么，该层第 1 个 Bucket 保存的延迟请求就是介于  $[0, 1)$  之间，第 2 个 Bucket 保存的是介于  $[1, 2)$  之间的请求。现在，如果有两个延迟请求，超时时刻分别在 18.5 毫秒和 123 毫秒，那么，第 1 个请求就应该被保存在第 1 层的第 19 个 Bucket（序号从 1 开始）中，而第 2 个请求，则应该被保存在第 2 层时间轮的第 6 个 Bucket 中。

这基本上就是 Kafka 中分层时间轮的实现原理。Kafka 不断向前推动各个层级的时间轮的时钟，按照时间轮的滴答时长，陆续接触到 Bucket 下的各个延迟任务，从而实现了请求的延迟处理。

但是，如果你仔细查看的话，就会发现，到目前为止，这套分层时间轮代码和 Kafka 概念并无直接的关联，比如分层时间轮里并不涉及主题、分区、副本这样的概念，也没有和 Controller、副本管理等 Kafka 组件进行直接交互。但实际上，延迟处理请求是 Kafka 的重要功能之一。你可能会问，到底是 Kafka 的哪部分源码负责创建和维护这套分层时间轮，并将它集成到整体框架中去的呢？答案就是接下来要介绍的两个类：Timer 和 SystemTimer。

## Timer 接口及 SystemTimer

这两个类的源码位于 `utils.timer` 包下的 `Timer.scala` 文件。其中，**Timer 接口定义了管理延迟操作的方法，而 SystemTimer 是实现延迟操作的关键代码**。后续在学习延迟请求类 `DelayedOperation` 时，我们就会发现，调用分层时间轮上的各类操作，都是通过 `SystemTimer` 类完成的。

### Timer 接口

接下来，我们就看下它们的源码。首先是 `Timer` 接口类，代码如下：

 复制代码

```
1 trait Timer {
2   // 将给定的定时任务插入到时间轮上，等待后续延迟执行
3   def add(timerTask: TimerTask): Unit
4   // 向前推进时钟，执行已达过期时间的延迟任务
5   def advanceClock(timeoutMs: Long): Boolean
6   // 获取时间轮上总的定时任务数
7   def size: Int
8   // 关闭定时器
```

```
9     def shutdown(): Unit
10 }
```

该 Timer 接口定义了 4 个方法。

add 方法：将给定的定时任务插入到时间轮上，等待后续延迟执行。

advanceClock 方法：向前推进时钟，执行已达过期时间的延迟任务。

size 方法：获取当前总定时任务数。

shutdown 方法：关闭该定时器。

其中，最重要的两个方法是 **add** 和 **advanceClock**，它们是**完成延迟请求处理的关键步骤**。接下来，我们结合 Timer 实现类 SystemTimer 的源码，重点分析这两个方法。

## SystemTimer 类

SystemTimer 类是 Timer 接口的实现类。它是一个定时器类，封装了分层时间轮对象，为 Purgatory 提供延迟请求管理功能。所谓的 Purgatory，就是保存延迟请求的缓冲区。也就是说，它保存的是因为不满足条件而无法完成，但是又没有超时的请求。

下面，我们从定义和方法两个维度来学习 SystemTimer 类。

### 定义

首先是该类的定义，代码如下：

 复制代码

```
1 class SystemTimer(executorName: String,
2                   tickMs: Long = 1,
3                   wheelSize: Int = 20,
4                   startMs: Long = Time.SYSTEM.hiResClockMs) extends Timer {
5     // 单线程的线程池用于异步执行定时任务
6     private[this] val taskExecutor = Executors.newFixedThreadPool(1,
7       (runnable: Runnable) => KafkaThread.nonDaemon("executor-" + executorName,
8     // 延迟队列保存所有Bucket，即所有TimerTaskList对象
9     private[this] val delayQueue = new DelayQueue[TimerTaskList]()
10    // 总定时任务数
11    private[this] val taskCounter = new AtomicInteger(0)
12    // 时间轮对象
13    private[this] val timingWheel = new TimingWheel(
```

```
14     tickMs = tickMs,
15     wheelSize = wheelSize,
16     startMs = startMs,
17     taskCounter = taskCounter,
18     delayQueue
19 )
20 // 维护线程安全的读写锁
21 private[this] val readWriteLock = new ReentrantReadWriteLock()
22 private[this] val readLock = readWriteLock.readLock()
23 private[this] val writeLock = readWriteLock.writeLock()
24 .....
25 ~
```

每个 `SystemTimer` 类定义了 4 个原生字段，分别是 `executorName`、`tickMs`、`wheelSize` 和 `startMs`。

`tickMs` 和 `wheelSize` 是构建分层时间轮的基础，你一定要重点掌握。不过上节课我已经讲过了，而且我在开篇还用具体数字带你回顾了它们的用途，这里就不重复了。另外两个参数不太重要，你只需要知道它们的含义就行了。

`executorName`：Purgatory 的名字。Kafka 中存在不同的 Purgatory，比如专门处理生产者延迟请求的 Produce 缓冲区、处理消费者延迟请求的 Fetch 缓冲区等。这里的 Produce 和 Fetch 就是 `executorName`。

`startMs`：该 `SystemTimer` 定时器启动时间，单位是毫秒。

除了原生字段，`SystemTimer` 类还定义了一些其他一些字段属性。我介绍 3 个比较重要的。这 3 个字段与时间轮都是强相关的。

1. **delayQueue 字段**。它保存了该定时器下管理的所有 Bucket 对象。因为是 DelayQueue，所以只有在 Bucket 过期后，才能从该队列中获取到。`SystemTimer` 类的 `advanceClock` 方法正是依靠了这个特性向前驱动时钟。关于这一点，一会儿我们详细说。
2. **timingWheel**。TimingWheel 是实现分层时间轮的类。`SystemTimer` 类依靠它来操作分层时间轮。
3. **taskExecutor**。它是单线程的线程池，用于异步执行提交的定时任务逻辑。

## 方法

说完了类定义与字段，我们看下 SystemTimer 类的方法。

该类总共定义了 6 个方法：add、addTimerTaskEntry、reinsert、advanceClock、size 和 shutdown。

其中，size 方法计算的是给定 Purgatory 下的总延迟请求数，shutdown 方法则是关闭前面说到的线程池，而 addTimerTaskEntry 方法则是将给定的 TimerTaskEntry 插入到时间轮中。如果该 TimerTaskEntry 表征的定时任务没有过期或被取消，方法还会将已过期的定时任务提交给线程池，等待异步执行该定时任务。至于 reinsert 方法，它会调用 addTimerTaskEntry 重新将定时任务插入回时间轮。

其实，SystemTimer 类最重要的方法是 add 和 advanceClock 方法，因为它们**真正对外提供服务的**。我们先说 add 方法。add 方法的作用，是将给定的定时任务插入到时间轮中进行管理。代码如下：

```
1 def add(timerTask: TimerTask): Unit = {
2     // 获取读锁。在没有线程持有写锁的前提下，
3     // 多个线程能够同时向时间轮添加定时任务
4     readLock.lock()
5     try {
6         // 调用addTimerTaskEntry执行插入逻辑
7         addTimerTaskEntry(new TimerTaskEntry(timerTask, timerTask.delayMs + Time.S'
8     } finally {
9         // 释放读锁
10        readLock.unlock()
11    }
12 }
```

 复制代码

add 方法就是调用 addTimerTaskEntry 方法执行插入动作。以下是 addTimerTaskEntry 的方法代码：

```
1 private def addTimerTaskEntry(timerTaskEntry: TimerTaskEntry): Unit = {
2     // 视timerTaskEntry状态决定执行什么逻辑：
3     // 1. 未过期未取消：添加到时间轮
4     // 2. 已取消：什么都不做
5     // 3. 已过期：提交到线程池，等待执行
6     if (!timingWheel.add(timerTaskEntry)) {
7         // 定时任务未取消，说明定时任务已过期
```

 复制代码



```
8 // 否则timingWheel.add方法应该返回True
9 if (!timerTaskEntry.cancelled)
10     taskExecutor.submit(timerTaskEntry.timerTask)
11 }
12 }
```

TimingWheel 的 add 方法会在定时任务已取消或已过期时，返回 False，否则，该方法会将定时任务添加到时间轮，然后返回 True。因此，addTimerTaskEntry 方法到底执行什么逻辑，取决于给定定时任务的状态：

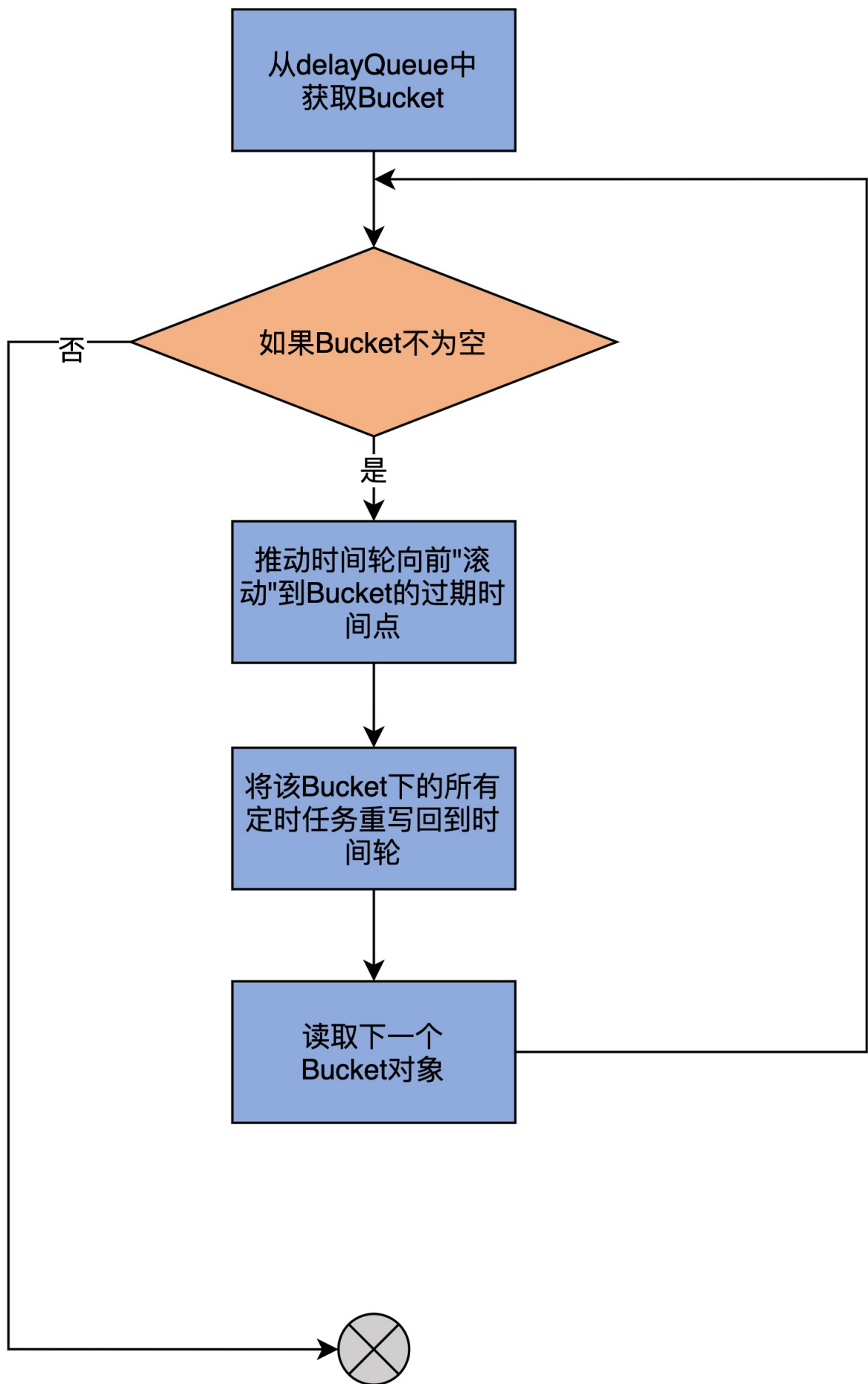
1. 如果该任务既未取消也未过期，那么，addTimerTaskEntry 方法将其添加到时间轮；
2. 如果该任务已取消，则该方法什么都不做，直接返回；
3. 如果该任务已经过期，则提交到相应的线程池，等待后续执行。

另一个关键方法是 advanceClock 方法。顾名思义，它的作用是**驱动时钟向前推进**。我们看下代码：

 复制代码

```
1 def advanceClock(timeoutMs: Long): Boolean = {
2     // 获取delayQueue中下一个已过期的Bucket
3     var bucket = delayQueue.poll(
4         timeoutMs, TimeUnit.MILLISECONDS)
5     if (bucket != null) {
6         // 获取写锁
7         // 一旦有线程持有写锁，其他任何线程执行add或advanceClock方法时会阻塞
8         writeLock.lock()
9         try {
10             while (bucket != null) {
11                 // 推动时间轮向前"滚动"到Bucket的过期时间点
12                 timingWheel.advanceClock(bucket.getExpiration())
13                 // 将该Bucket下的所有定时任务重写回到时间轮
14                 bucket.flush(reinsert)
15                 // 读取下一个Bucket对象
16                 bucket = delayQueue.poll()
17             }
18         } finally {
19             // 释放写锁
20             writeLock.unlock()
21         }
22         true
23     } else {
24         false
25     }
```

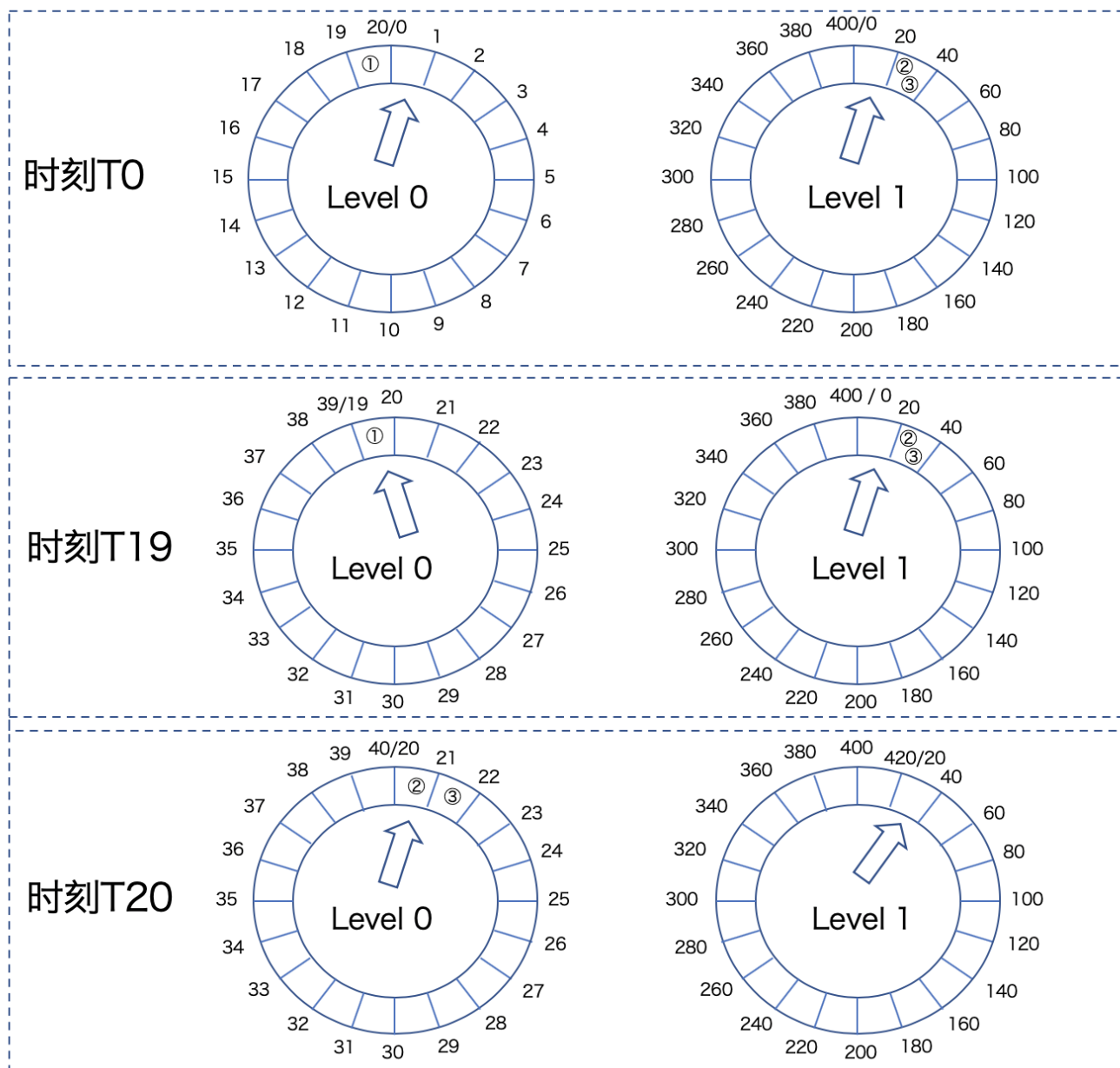
由于代码逻辑比较复杂，我再画一张图来展示一下：





advanceClock 方法要做的事情，就是遍历 delayQueue 中的所有 Bucket，并将时间轮的时钟依次推进到它们的过期时间点，令它们过期。然后，再将这些 Bucket 下的所有定时任务全部重新插入回时间轮。

我用一张图来说明这个重新插入过程。



从这张图中，我们可以看到，在 T0 时刻，任务①存放在 Level 0 的时间轮上，而任务②和③存放在 Level 1 的时间轮上。此时，时钟推进到 Level 0 的第 0 个 Bucket 上，以及 Level 1 的第 0 个 Bucket 上。

当时间来到 T19 时刻，时钟也被推进到 Level 0 的第 19 个 Bucket，任务①会被执行。但是，由于一层时间轮是 20 个 Bucket，因此，T19 时刻 Level 0 的时间轮尚未完整走完一

圈，此时，Level 1 的时间轮状态没有发生任何变化。

当 T20 时刻到达时，Level 0 的时间轮已经执行完成，Level 1 的时间轮执行了一次滴答，向前推进一格。此时，Kafka 需要将任务②和③插入到 Level 0 的时间轮上，位置是第 20 个和第 21 个 Bucket。这个将高层时间轮上的任务插入到低层时间轮的过程，是由 advanceClock 中的 reinsert 方法完成。

至于为什么要重新插入回低层次的时间轮，其实是因为，随着时钟的推进，当前时间逐渐逼近任务②和③的超时时间点。它们之间差值的缩小，足以让它们被放入到下一层的时间轮中。

总的来说，SystemTimer 类实现了 Timer 接口的方法，**它封装了底层的分层时间轮，为上层调用方提供了便捷的方法来操作时间轮**。那么，它的上层调用方是谁呢？答案就是 DelayedOperationPurgatory 类。这就是我们建模 Purgatory 的地方。

不过，在了解 DelayedOperationPurgatory 之前，我们要先学习另一个重要的类：DelayedOperation。前者是一个泛型类，它的类型参数恰恰就是 DelayedOperation。因此，我们不可能在不了解 DelayedOperation 的情况下，很好地掌握 DelayedOperationPurgatory。

## DelayedOperation 类

这个类位于 server 包下的 DelayedOperation.scala 文件中。它是所有 Kafka 延迟请求类的抽象父类。我们依然从定义和方法这两个维度去剖析它。

### 定义

首先来看定义。代码如下：

 复制代码

```
1 abstract class DelayedOperation(override val delayMs: Long,  
2                               lockOpt: Option[Lock] = None)  
3   extends TimerTask with Logging {  
4     // 标识该延迟操作是否已经完成  
5     private val completed = new AtomicBoolean(false)  
6     // 防止多个线程同时检查操作是否可完成时发生锁竞争导致操作最终超时  
7     private val tryCompletePending = new AtomicBoolean(false)  
8     private[server] val lock: Lock = lockOpt.getOrElse(new ReentrantLock)
```

DelayedOperation 类是一个抽象类，它的构造函数中只需要传入一个超时时间即可。这个超时时间通常是**客户端发出请求的超时时间**，也就是客户端参数 `request.timeout.ms` 的值。这个类实现了上节课学到的 TimerTask 接口，因此，作为一个建模延迟操作的类，它自动继承了 TimerTask 接口的 cancel 方法，支持延迟操作的取消，以及 TimerTaskEntry 的 Getter 和 Setter 方法，支持将延迟操作绑定到时间轮相应 Bucket 下的某个链表元素上。

除此之外，DelayedOperation 类额外定义了两个字段：**completed** 和 **tryCompletePending**。

前者理解起来比较容易，它就是**表征这个延迟操作是否完成的布尔变量**。我重点解释一下 tryCompletePending 的作用。

这个参数是在 1.1 版本引入的。在此之前，只有 completed 参数。但是，这样就可能存在这样一个问题：当多个线程同时检查某个延迟操作是否满足完成条件时，如果其中一个线程持有了锁（也就是上面的 lock 字段），然后执行条件检查，会发现不满足完成条件。而与此同时，另一个线程执行检查时却发现条件满足了，但是这个线程又没有拿到锁，此时，该延迟操作将永远不会有再次被检查的机会，会导致最终超时。

加入 tryCompletePending 字段目的，就是**确保拿到锁的线程有机会再次检查条件是否已经满足**。具体是怎么实现的呢？下面讲到 maybeTryComplete 方法时，我会再带你进行深入的分析。

关于 DelayedOperation 类的定义，你掌握到这个程度就可以了，重点是学习这些字段是如何在方法中发挥作用的。

## 方法

DelayedOperation 类有 7 个方法。我先介绍下它们的作用，这样你在读源码时就可以心中有数。

**forceComplete**: 强制完成延迟操作，不管它是否满足完成条件。每当操作满足完成条件或已经过期了，就需要调用该方法完成该操作。

**isCompleted**: 检查延迟操作是否已经完成。源码使用这个方法来决定后续如何处理该操作。比如如果操作已经完成了，那么通常需要取消该操作。

**onExpiration**: 强制完成之后执行的过期逻辑回调方法。只有真正完成操作的那个线程才有资格调用这个方法。

**onComplete**: 完成延迟操作所需的处理逻辑。这个方法只会在 **forceComplete** 方法中被调用。

**tryComplete**: 尝试完成延迟操作的顶层方法，内部会调用 **forceComplete** 方法。

**maybeTryComplete**: 线程安全版本的 **tryComplete** 方法。这个方法其实是社区后来才加入的，不过已经慢慢地取代了 **tryComplete**，现在外部代码调用的都是这个方法了。

**run**: 调用延迟操作超时后的过期逻辑，也就是组合调用 **forceComplete** + **onExpiration**。

我们说过，**DelayedOperation** 是抽象类，对于不同类型的延时请求，**onExpiration**、**onComplete** 和 **tryComplete** 的处理逻辑也各不相同，因此需要子类来实现它们。

其他方法的代码大多短小精悍，你一看就能明白，我就不做过多解释了。我重点说下 **maybeTryComplete** 方法。毕竟，这是社区为了规避因多线程访问产生锁争用导致线程阻塞，从而引发请求超时问题而做的努力。先看方法代码：

 复制代码

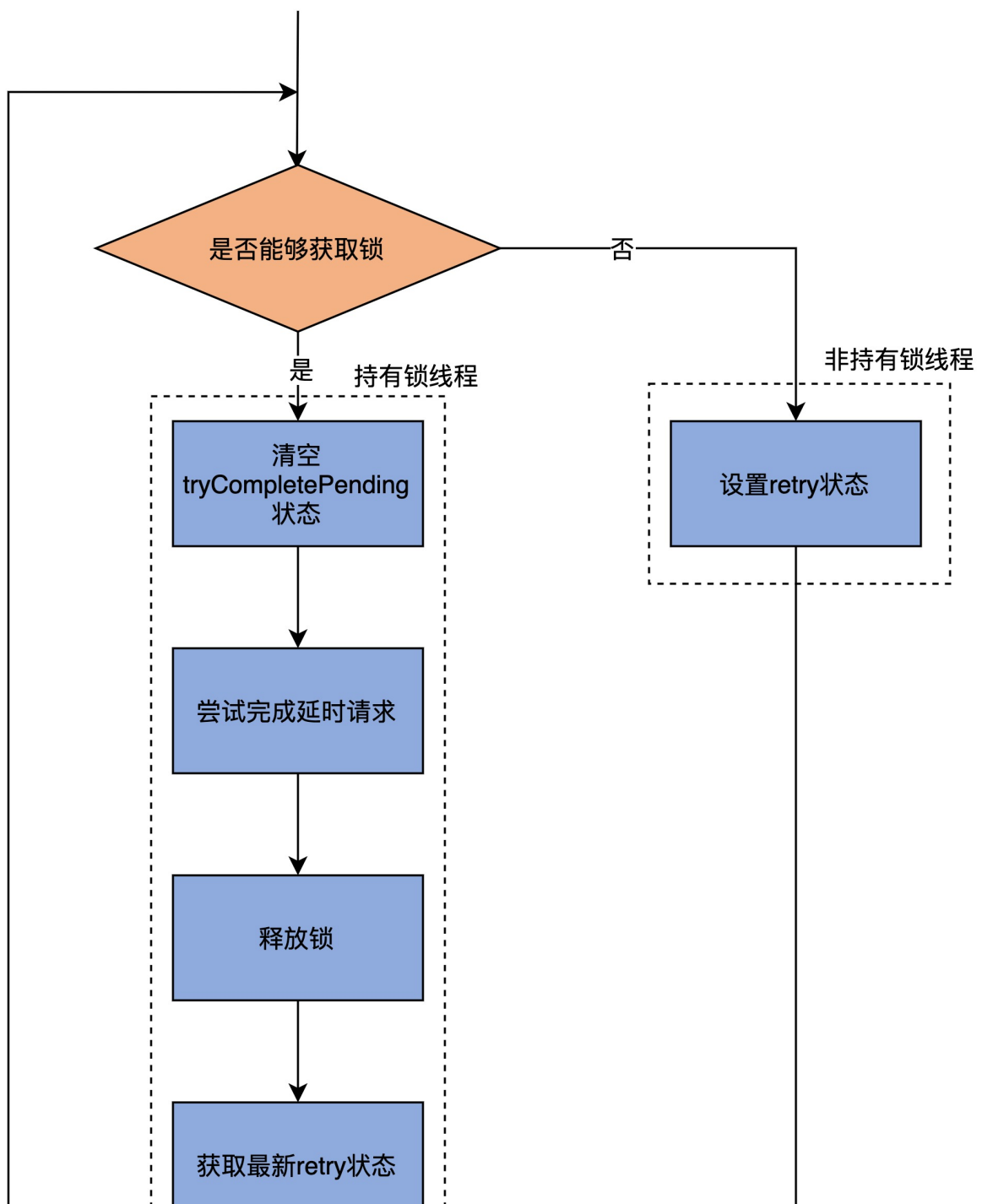
```
1 private[server] def maybeTryComplete(): Boolean = {
2   var retry = false // 是否需要重试
3   var done = false  // 延迟操作是否已完成
4   do {
5     if (lock.tryLock()) { // 尝试获取锁对象
6       try {
7         tryCompletePending.set(false)
8         done = tryComplete()
9       } finally {
10        lock.unlock()
11      }
12      // 运行到这里的线程持有锁，其他线程只能运行else分支的代码
13      // 如果其他线程将maybeTryComplete设置为true，那么retry=true
14      // 这就相当于其他线程给了本线程重试的机会
```

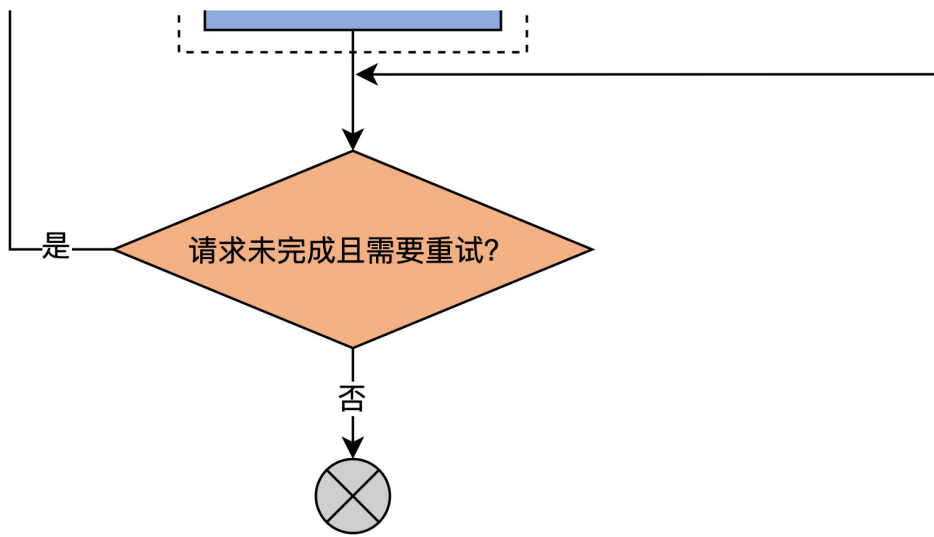
```

15     retry = tryCompletePending.get()
16 } else {
17     // 运行到这里的线程没有拿到锁
18     // 设置tryCompletePending=true给持有锁的线程一个重试的机会
19     retry = !tryCompletePending.getAndSet(true)
20 }
21 } while (!isCompleted && retry)
22 done
23

```

为了方便你理解，我画了一张流程图说明它的逻辑：





从图中可以看出，这个方法可能会被多个线程同时访问，只是不同线程会走不同的代码分支，分叉点就在**尝试获取锁的 if 语句**。

如果拿到锁对象，就依次执行清空 `tryCompletePending` 状态、完成延迟请求、释放锁以及读取最新 `retry` 状态的动作。未拿到锁的线程，就只能设置 `tryCompletePending` 状态，来间接影响 `retry` 值，从而给获取到锁的线程一个重试的机会。这里的重试，是通过 `do...while` 循环的方式实现的。

好了，`DelayedOperation` 类我们就说到这里。除了这些公共方法，你最好结合一两个具体子类的方法实现，体会下具体延迟请求类是如何实现 `tryComplete` 方法的。我推荐你从 `DelayedProduce` 类的 **`tryComplete` 方法** 开始。

我们之前总说，`acks=all` 的 `PRODUCE` 请求很容易成为延迟请求，因为它必须等待所有的 `ISR` 副本全部同步消息之后才能完成，你可以顺着这个思路，研究下 `DelayedProduce` 的 `tryComplete` 方法是如何实现的。

## DelayedOperationPurgatory 类

接下来，我们补上延迟请求模块的最后一块“拼图”：`DelayedOperationPurgatory` 类的源码分析。

该类是实现 `Purgatory` 的地方。从代码结构上看，它是一个 `Scala` 伴生对象。也就是说，源码文件同时定义了 `DelayedOperationPurgatory Object` 和 `Class`。Object 中仅仅定义了 `apply` 工厂方法和一个名为 `Shards` 的字段，这个字段是 `DelayedOperationPurgatory`

监控列表的数组长度信息。因此，我们还是重点学习 DelayedOperationPurgatory Class 的源码。

前面说过，DelayedOperationPurgatory 类是一个泛型类，它的参数类型是 DelayedOperation 的具体子类。因此，通常情况下，每一类延迟请求都对应于一个 DelayedOperationPurgatory 实例。这些实例一般都保存在上层的管理器中。比如，与消费者组相关的心跳请求、加入组请求的 Purgatory 实例，就保存在 GroupCoordinator 组件中，而与生产者相关的 PRODUCE 请求的 Purgatory 实例，被保存在分区对象或副本状态机中。

## 定义

至于怎么学，还是老规矩，我们先从定义开始。代码如下：

 复制代码

```
1 final class DelayedOperationPurgatory[T <: DelayedOperation](
2     purgatoryName: String,
3     timeoutTimer: Timer,
4     brokerId: Int = 0,
5     purgeInterval: Int = 1000,
6     reaperEnabled: Boolean = true,
7     timerEnabled: Boolean = true) extends Logging with KafkaMetricsGroup {
8     .....
9 }
```

定义中有 6 个字段。其中，很多字段都有默认参数，比如，最后两个参数分别表示是否启动删除线程，以及是否启用分层时间轮。现在，源码中所有类型的 Purgatory 实例都是默认启动的，因此无需特别留意它们。

purgeInterval 这个参数用于控制删除线程移除 Bucket 中的过期延迟请求的频率，在绝大部分情况下，都是 1 秒一次。当然，对于生产者、消费者以及删除消息的 AdminClient 而言，Kafka 分别定义了专属的参数允许你调整这个频率。比如，生产者参数 producer.purgatory.purge.interval.requests，就是做这个用的。

事实上，需要传入的参数一般只有两个：**purgatoryName** 和 **brokerId**，它们分别表示这个 Purgatory 的名字和 Broker 的序号。



而 timeoutTimer，就是我们前面讲过的 SystemTimer 实例，我就不重复解释了。

## Watchers 和 WatcherList

DelayedOperationPurgatory 还定义了两个内置类，分别是 Watchers 和 WatcherList。

**Watchers 是基于 Key 的一个延迟请求的监控链表。**它的主体代码如下：

 复制代码

```
1 private class Watchers(val key: Any) {
2     private[this] val operations =
3         new ConcurrentLinkedQueue[T]()
4     // 其他方法.....
5 }
```

每个 Watchers 实例都定义了一个延迟请求链表，而这里的 Key 可以是任何类型，比如表示消费者组的字符串类型、表示主题分区的 TopicPartitionOperationKey 类型。你不用穷尽这里所有的 Key 类型，你只需要了解，Watchers 是一个通用的延迟请求链表，就行了。Kafka 利用它来**监控保存其中的延迟请求的可完成状态**。

既然 Watchers 主要的数据结构是链表，那么，它的所有方法本质上就是一个链表操作。比如，tryCompleteWatched 方法会遍历整个链表，并尝试完成其中的延迟请求。再比如，cancel 方法也是遍历链表，再取消掉里面的延迟请求。至于 watch 方法，则是将延迟请求加入到链表中。

说完了 Watchers，我们看下 WatcherList 类。它非常短小精悍，完整代码如下：


 复制代码

```
1 private class WatcherList {
2     // 定义一组按照Key分组的Watchers对象
3     val watchersByKey = new Pool[Any, Watchers](Some((key: Any) => new Watchers(|
4     val watchersLock = new ReentrantLock()
5     // 返回所有Watchers对象
6     def allWatchers = {
7         watchersByKey.values
8     }
9 }
```

WatcherList 最重要的字段是 **watchersByKey**。它是一个 Pool，Pool 就是 Kafka 定义的池对象，它本质上就是一个 ConcurrentHashMap。watchersByKey 的 Key 可以是任何类型，而 Value 就是 Key 对应类型的一组 Watchers 对象。

说完了 DelayedOperationPurgatory 类的两个内部类 Watchers 和 WatcherList，我们可以开始学习该类的两个重要方法：tryCompleteElseWatch 和 checkAndComplete 方法。

前者的作用是**检查操作是否能够完成**，如果不能的话，就把它加入到对应 Key 所在的 WatcherList 中。以下是方法代码：

 复制代码

```
1 def tryCompleteElseWatch(operation: T, watchKeys: Seq[Any]): Boolean = {
2   assert(watchKeys.nonEmpty, "The watch key list can't be empty")
3   var isCompletedByMe = operation.tryComplete()
4   // 如果该延迟请求是由本线程完成的，直接返回true即可
5   if (isCompletedByMe)
6     return true
7   var watchCreated = false
8   // 遍历所有要监控的Key
9   for(key <- watchKeys) {
10    // 再次查看请求的完成状态，如果已经完成，就说明是被其他线程完成的，返回false
11    if (operation.isCompleted)
12      return false
13    // 否则，将该operation加入到Key所在的WatcherList
14    watchForOperation(key, operation)
15    // 设置watchCreated标记，表明该任务已经被加入到WatcherList
16    if (!watchCreated) {
17      watchCreated = true
18      // 更新Purgatory中总请求数
19      estimatedTotalOperations.incrementAndGet()
20    }
21  }
22  // 再次尝试完成该延迟请求
23  isCompletedByMe = operation.maybeTryComplete()
24  if (isCompletedByMe)
25    return true
26  // 如果依然不能完成此请求，将其加入到过期队列
27  if (!operation.isCompleted) {
28    if (timerEnabled)
29      timeoutTimer.add(operation)
30    if (operation.isCompleted) {
31      operation.cancel()
32    }
33  }
34  false
```

该方法的名字折射出了它要做的事情：先尝试完成请求，如果无法完成，则把它加入到 `WatcherList` 中进行监控。具体来说，`tryCompleteElseWatch` 调用 `tryComplete` 方法，尝试完成延迟请求，如果返回结果是 `true`，就说明执行 `tryCompleteElseWatch` 方法的线程正常地完成了该延迟请求，也就不需要再添加到 `WatcherList` 了，直接返回 `true` 就行了。

否则的话，代码会遍历所有要监控的 `Key`，再次查看请求的完成状态。如果已经完成，就说明是被其他线程完成的，返回 `false`；如果依然无法完成，则将该请求加入到 `Key` 所在的 `WatcherList` 中，等待后续完成。同时，设置 `watchCreated` 标记，表明该任务已经被加入到 `WatcherList` 以及更新 `Purgatory` 中总请求数。

待遍历完所有 `Key` 之后，源码会再次尝试完成该延迟请求，如果完成了，就返回 `true`，否则就取消该请求，然后将其加入到过期队列，最后返回 `false`。

总的来看，你要掌握这个方法要做的两个事情：

1. 先尝试完成延迟请求；
2. 如果不行，就加入到 `WatcherList`，等待后面再试。

那么，代码是在哪里进行重试的呢？这就需要用到第 2 个方法 `checkAndComplete` 了。

该方法会**检查给定 `Key` 所在的 `WatcherList` 中的延迟请求是否满足完成条件**，如果是的话，则结束掉它们。我们一起看下源码：

 复制代码

```
1 def checkAndComplete(key: Any): Int = {
2   // 获取给定Key的WatcherList
3   val wl = watcherList(key)
4   // 获取WatcherList中Key对应的Watchers对象实例
5   val watchers = inLock(wl.watchersLock) { wl.watchersByKey.get(key) }
6   // 尝试完成满足完成条件的延迟请求并返回成功完成的请求数
7   val numCompleted = if (watchers == null)
8     0
9   else
10    watchers.tryCompleteWatched()
```

```
11     debug(s"Request key $key unblocked $numCompleted $purgatoryName operations")
12     numCompleted
13 }
```

代码很简单，就是根据给定 Key，获取对应的 WatcherList 对象，以及它下面保存的 Watchers 对象实例，然后尝试完成满足完成条件的延迟请求，并返回成功完成的请求数。

可见，非常重要的步骤就是**调用 Watchers 的 tryCompleteWatched 方法，去尝试完成那些已满足完成条件的延迟请求。**

## 总结

今天，我们重点学习了分层时间轮的上层组件，包括 Timer 接口及其实现类 SystemTimer、DelayedOperation 类以及 DelayedOperationPurgatory 类。你基本上可以认为，它们是逐级被调用的关系，即 **DelayedOperation 调用 SystemTimer 类，DelayedOperationPurgatory 管理 DelayedOperation。**它们共同实现了 Broker 端对于延迟请求的处理，基本思想就是，**能立即完成的请求马上完成，否则就放入到名为 Purgatory 的缓冲区中。**后续，DelayedOperationPurgatory 类的方法会自动地处理这些延迟请求。

我们来回顾一下重点。

SystemTimer 类：Kafka 定义的定时器类，封装了底层分层时间轮，实现了时间轮 Bucket 的管理以及时钟向前推进功能。它是实现延迟请求后续被自动处理的基础。


DelayedOperation 类：延迟请求的高阶抽象类，提供了完成请求以及请求完成和过期后的回调逻辑实现。

DelayedOperationPurgatory 类：Purgatory 实现类，该类定义了 WatcherList 对象以及对 WatcherList 的操作方法，而 WatcherList 是实现延迟请求后续自动处理的关键数据结构。

总的来说，延迟请求模块属于 Kafka 的冷门组件。毕竟，大部分的请求还是能够被立即处理的。了解这部分模块的最大意义在于，你可以学习 Kafka 这个分布式系统是如何异步循环操作和管理定时任务的。这个功能是所有分布式系统都要面临的课题，因此，弄明白了这部分的原理和代码实现，后续我们在自行设计类似的功能模块时，就非常容易了。

## 课后讨论

DelayedOperationPurgatory 类中定义了一个 Reaper 线程，用于将已过期的延迟请求从数据结构中移除掉。这实际上是由 DelayedOperationPurgatory 的 advanceClock 方法完成的。它里面有这样一句：

 复制代码

```
1 val purged = watcherLists.foldLeft(0) {  
2     case (sum, watcherList) => sum + watcherList.allWatchers.map(_.purgeComplete  
3 }
```

你觉得这个语句是做什么用的？

欢迎在留言区写下你的思考和答案，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

### 更多课程推荐

## MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇  
前阿里资深技术专家



涨价倒计时 

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 (1)

写留言



伯安知心

2020-06-09

大概意思就是说 到了purgeInterval这个清除间隔的数量，就会清理watcherLists中已经完成但是一直被监听的触发器个数，而且这个purged 就是已经完成但是还未清理的任务数量。purgeCompleted 进行remove。大概意思就是说 到了purgeInterval这个清除间隔的数量，就会清理watcherLists中已经完成但是一直被监听的触发器个数，而且这个purged 就是已经完成但是还未清理的任务数量。purgeCompleted 进行remove。

展开 ∨

