

## 19 | TimingWheel: 探究Kafka定时器背后的高效时间轮算法

2020-06-06 胡夕

Kafka核心源码解读

[进入课程 >](#)



讲述：胡夕

时长 22:40 大小 20.76M



你好，我是胡夕。今天，我们开始学习 Kafka 延时请求的代码实现。

延时请求（Delayed Operation），也称延迟请求，是指因未满足条件而暂时无法被处理的 Kafka 请求。举个例子，配置了 `acks=all` 的生产者发送的请求可能一时无法完成，因为 Kafka 必须确保 ISR 中的所有副本都要成功响应这次写入。因此，通常情况下，这些请求没法被立即处理。只有满足了条件或发生了超时，Kafka 才会把该请求标记为完成状态。这就是所谓的延时请求。

今天，我们的重点是弄明白请求被延时处理的机制——分层时间轮算法。



时间轮的应用范围非常广。很多操作系统的定时任务调度（如 Crontab）以及通信框架（如 Netty 等）都利用了时间轮的思想。几乎所有的时间任务调度系统都是基于时间轮算

法的。Kafka 应用基于时间轮算法管理延迟请求的代码简洁精炼，而且和业务逻辑代码完全解耦，你可以从 0 到 1 地照搬到你自己的项目工程中。

## 时间轮简介

在开始介绍时间轮之前，我想先请你思考这样一个问题：“如果是你，你会怎么实现 Kafka 中的延时请求呢？”

针对这个问题，我的第一反应是使用 Java 的 DelayQueue。毕竟，这个类是 Java 天然提供的延时队列，非常适合建模延时对象处理。实际上，Kafka 的第一版延时请求就是使用 DelayQueue 做的。

但是，DelayQueue 有一个弊端：它插入和删除队列元素的时间复杂度是  $O(\log N)$ 。对于 Kafka 这种非常容易积攒几十万个延时请求的场景来说，该数据结构的性能是瓶颈。当然，这一版的设计还有其他弊端，比如，它在清除已过期的延迟请求方面不够高效，可能会出现内存溢出的情形。后来，社区改造了延时请求的实现机制，采用了基于时间轮的方案。

时间轮有简单时间轮（Simple Timing Wheel）和分层时间轮（Hierarchical Timing Wheel）两类。两者各有利弊，也都有各自的使用场景。Kafka 采用的是分层时间轮，这是我们重点学习的内容。

关于分层时间轮，有很多严谨的科学论文。不过，大多数的论文读起来晦涩难懂，而且偏理论研究。然而，我们并非是要完整系统地学习这套机制，我们关心的是如何将其应用于实践当中。要做到这一点，结合着源码来学习就是一个不错的途径。你需要关注，在代码层面，Kafka 是如何实现多层时间轮的。

“时间轮”的概念稍微有点抽象，我用一个生活中的例子，来帮助你建立一些初始印象。

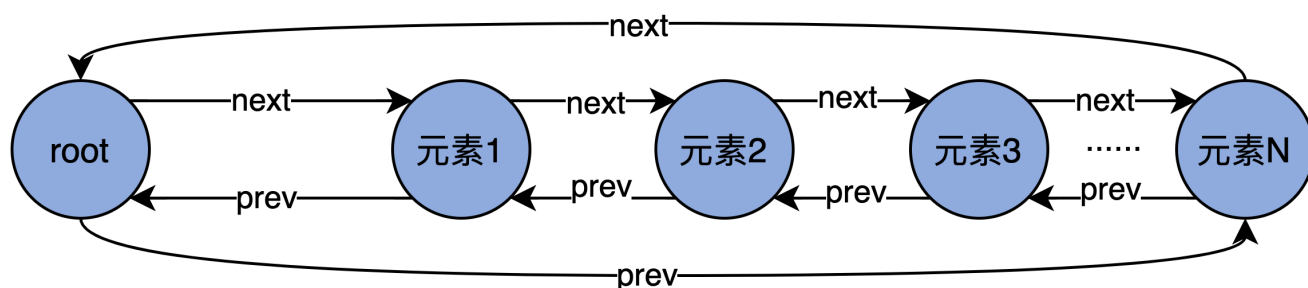
想想我们生活中的手表。手表由时针、分针和秒针组成，它们各自有独立的刻度，但又彼此相关：秒针转动一圈，分针会向前推进一格；分针转动一圈，时针会向前推进一格。这就是典型的分层时间轮。

和手表不太一样的是，Kafka 自己有专门的术语。在 Kafka 中，手表中的“一格”叫“一个桶（Bucket）”，而“推进”对应于 Kafka 中的“滴答”，也就是 tick。后面你在阅读

源码的时候，会频繁地看到 Bucket、tick 字眼，你可以把它们理解成手表刻度盘面上的“一格”和“向前推进”的意思。

除此之外，每个 Bucket 下也不是白板一块，它实际上是一个双向循环链表（Doubly Linked Cyclic List），里面保存了一组延时请求。

我先用一张图帮你理解下双向循环链表。

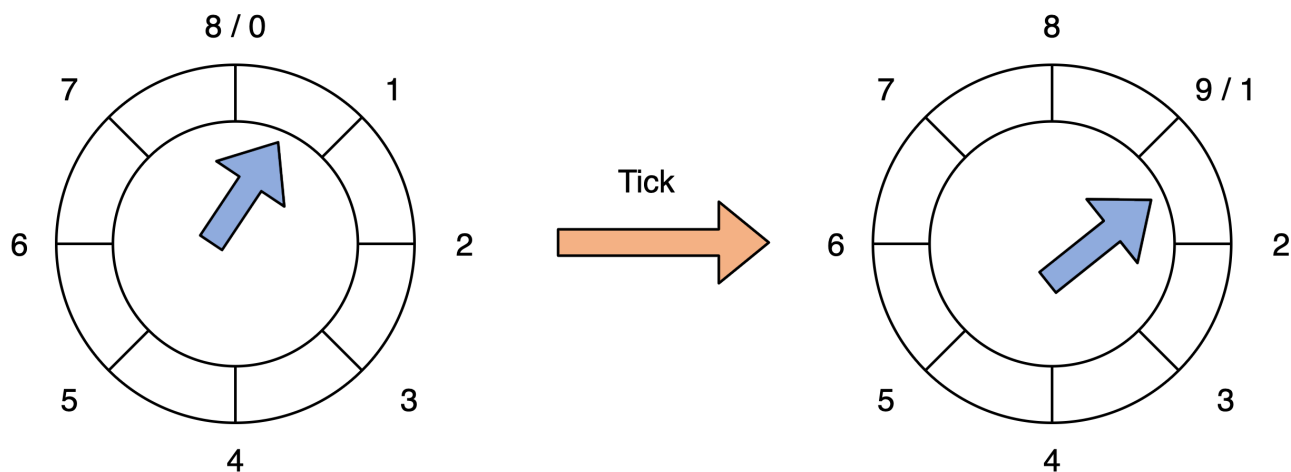


图中的每个节点都有一个 next 和 prev 指针，分别指向下一个元素和上一个元素。Root 是链表的头部节点，不包含任何实际数据。它的 next 指针指向链表的第一个元素，而 prev 指针指向最后一个元素。

由于是双向链表结构，因此，代码能够利用 next 和 prev 两个指针快速地定位元素，因此，在 Bucket 下插入和删除一个元素的时间复杂度是  $O(1)$ 。当然，双向链表要求同时保存两个指针数据，在节省时间的同时消耗了更多的空间。在算法领域，这是典型的用空间去换时间的优化思想。

## 源码层级关系

在 Kafka 中，具体是怎么应用分层时间轮实现请求队列的呢？



图中的时间轮共有两个层级，分别是 Level 0 和 Level 1。每个时间轮有 8 个 Bucket，每个 Bucket 下是一个双向循环链表，用来保存延迟请求。

在 Kafka 源码中，时间轮对应 `utils.timer` 包下的 `TimingWheel` 类，每个 Bucket 下的链表对应 `TimerTaskList` 类，链表元素对应 `TimerTaskEntry` 类，而每个链表元素里面保存的延时任务对应 `TimerTask`。

在这些类中，`TimerTaskEntry` 与 `TimerTask` 是 1 对 1 的关系，`TimerTaskList` 下包含多个 `TimerTaskEntry`，`TimingWheel` 包含多个 `TimerTaskList`。

我画了一张 UML 图，帮助你理解这些类之间的对应关系：



## 时间轮各个类源码定义

掌握了这些基础知识，下面我就结合这些源码，来解释下延迟请求是如何被这套分层时间轮管理的。根据调用关系，我采用自底向上的方法给出它们的定义。

### TimerTask 类

首先是 `TimerTask` 类。该类位于 `utils.timer` 包下的 `TimerTask.scala` 文件中。它的代码只有几十行，非常容易理解。

```
1 trait TimerTask extends Runnable {  
2     val delayMs: Long // 通常是request.timeout.ms参数值  
3     // 每个TimerTask实例关联一个TimerTaskEntry  
4     // 就是说每个定时任务需要知道它在哪个Bucket链表下的哪个链表元素上  
5     private[this] var timerTaskEntry: TimerTaskEntry = null  
6     // 取消定时任务，原理就是将关联的timerTaskEntry置空  
7     def cancel(): Unit = {  
8         synchronized {  
9             if (timerTaskEntry != null) timerTaskEntry.remove()  
10            timerTaskEntry = null  
11        }  
12    }  
13    // 关联timerTaskEntry，原理是给timerTaskEntry字段赋值  
14    private[timer] def setTimerTaskEntry(entry: TimerTaskEntry)  
15        : Unit = {  
16        synchronized {  
17            if (timerTaskEntry != null && timerTaskEntry != entry)  
18                timerTaskEntry.remove()  
19            timerTaskEntry = entry  
20        }  
21    }  
22    // 获取关联的timerTaskEntry实例  
23    private[timer] def getTimerTaskEntry(): TimerTaskEntry = {  
24        timerTaskEntry  
25    }  
26 }  
27
```

从代码可知，TimerTask 是一个 Scala 接口 (Trait) 。每个 TimerTask 都有一个 delayMs 字段，表示这个定时任务的超时时间。通常来说，这就是客户端参数 request.timeout.ms 的值。这个类还绑定了一个 timerTaskEntry 字段，因为，每个定时任务都要知道，它存放在哪个 Bucket 链表下的哪个链表元素上。


既然绑定了这个字段，就要提供相应的 Setter 和 Getter 方法。Getter 方法仅仅是返回这个字段而已，Setter 方法要稍微复杂一些。在给 timerTaskEntry 赋值之前，它必须要先考虑这个定时任务是否已经绑定了其他的 timerTaskEntry，如果是的话，就必须先取消绑定。另外，Setter 的整个方法体必须由 monitor 锁保护起来，以保证线程安全性。

这个类还有个 cancel 方法，用于取消定时任务。原理也很简单，就是将关联的 timerTaskEntry 置空。也就是说，把定时任务从链表上摘除。

总之，TimerTask 建模的是 Kafka 中的定时任务。接下来，我们来看 TimerTaskEntry 是如何承载这个定时任务的，以及如何在链表中实现双向关联。

## TimerTaskEntry 类

如前所述，TimerTaskEntry 表征的是 Bucket 链表下的一个元素。它的主要代码如下：

 复制代码

```
1 private[timer] class TimerTaskEntry(val timerTask: TimerTask, val expirationMs
2     @volatile
3     var list: TimerTaskList = null    // 绑定的Bucket链表实例
4     var next: TimerTaskEntry = null  // next指针
5     var prev: TimerTaskEntry = null  // prev指针
6     // 关联给定的定时任务
7     if (timerTask != null) timerTask.setTimerTaskEntry(this)
8     // 关联定时任务是否已经被取消了
9     def cancelled: Boolean = {
10         timerTask.getTimerTaskEntry != this
11     }
12     // 从Bucket链表中移除自己
13     def remove(): Unit = {
14         var currentList = list
15         while (currentList != null) {
16             currentList.remove(this)
17             currentList = list
18         }
19     }
20     .....
21 }
```

该类定义了 TimerTask 类字段，用来指定定时任务，同时还封装了一个过期时间戳字段，这个字段值定义了定时任务的过期时间。

举个例子，假设有个 PRODUCE 请求在当前时间 1 点钟被发送到 Broker，超时时间是 30 秒，那么，该请求必须在 1 点 30 秒之前完成，否则将被视为超时。这里的 1 点 30 秒，就是 expirationMs 值。

除了 TimerTask 类字段，该类还定义了 3 个字段：list、next 和 prev。它们分别对应于 Bucket 链表实例以及自身的 next、prev 指针。注意，list 字段是 volatile 型的，这是因为，Kafka 的延时请求可能会被其他线程从一个链表搬移到另一个链表中，因此，**为了保证必要的内存可见性**，代码声明 list 为 volatile。



该类的方法代码都很直观，你可以看下我写的代码注释。这里我重点解释一下 remove 方法的实现原理。

remove 的逻辑是将 TimerTask 自身从双向链表中移除掉，因此，代码调用了 TimerTaskList 的 remove 方法来做这件事。那这里就有一个问题：“怎么算真正移除掉呢？”其实，这是根据“TimerTaskEntry 的 list 是否为空”来判断的。一旦置空了该字段，那么，这个 TimerTaskEntry 实例就变成了“孤儿”，不再属于任何一个链表了。从这个角度来看，置空就相当于移除的效果。

需要注意的是，置空这个动作是在 TimerTaskList 的 remove 中完成的，而这个方法可能会被其他线程同时调用，因此，上段代码使用了 while 循环的方式来确保 TimerTaskEntry 的 list 字段确实被置空了。这样，Kafka 才能安全地认为此链表元素被成功移除。

## TimerTaskList 类

说完了 TimerTask 和 TimerTaskEntry，就轮到链表类 TimerTaskList 上场了。我们先看它的定义：

复制代码

```
1 private[timer] class TimerTaskList(taskCounter: AtomicInteger) extends Delayed
2   private[this] val root = new TimerTaskEntry(null, -1)
3   root.next = root
4   root.prev = root
5   private[this] val expiration = new AtomicLong(-1L)
6   .....
7 }
```

TimerTaskList 实现了刚刚那张图所展示的双向循环链表。它定义了一个 Root 节点，同时还定义了两个字段：

taskCounter，用于标识当前这个链表中的总定时任务数；

expiration，表示这个链表所在 Bucket 的过期时间戳。

就像我前面说的，每个 Bucket 对应于手表表盘上的一格。它有起始时间和结束时间，因而也就有时间间隔的概念，即“结束时间 - 起始时间 = 时间间隔”。同一层的 Bucket 的时

间间隔都是一样的。只有当前时间越过了 Bucket 的起始时间，这个 Bucket 才算是过期。而这里的起始时间，就是代码中 expiration 字段的值。

除了定义的字段之外，TimerTaskList 类还定义一些重要的方法，比如 expiration 的 Getter 和 Setter 方法、add、remove 和 flush 方法。

我们先看 expiration 的 Getter 和 Setter 方法。


 复制代码

```
1 // Setter方法
2 def setExpiration(expirationMs: Long): Boolean = {
3     expiration.getAndSet(expirationMs) != expirationMs
4 }
5
6 // Getter方法
7 def getExpiration(): Long = {
8     expiration.get()
9 }
```

我重点解释下 Setter 方法。代码使用了 AtomicLong 的 CAS 方法 getAndSet 原子性地设置了过期时间戳，之后将新过期时间戳和旧值进行比较，看看是否不同，然后返回结果。

这里为什么要比较新旧值是否不同呢？这是因为，目前 Kafka 使用一个 DelayQueue 统一管理所有的 Bucket，也就是 TimerTaskList 对象。随着时钟不断向前推进，原有 Bucket 会不断地过期，然后失效。当这些 Bucket 失效后，源码会重用这些 Bucket。重用的方式就是重新设置 Bucket 的过期时间，并把它们加回到 DelayQueue 中。这里进行比较的目的，就是用来判断这个 Bucket 是否要被插入到 DelayQueue。

此外，TimerTaskList 类还提供了 add 和 remove 方法，分别实现将给定定时任务插入到链表、从链表中移除定时任务的逻辑。这两个方法的主体代码基本上就是我们在数据结构课上学过的链表元素插入和删除操作，所以这里我就不具体展开讲了。你可以将这些代码和数据结构书中的代码比对下，看看它们是不是长得很像。

 复制代码

```
1 // add方法
2 def add(timerTaskEntry: TimerTaskEntry): Unit = {
3     var done = false
4     while (!done) {
```



```

5    // 在添加之前尝试移除该定时任务，保证该任务没有在其他链表中
6    timerTaskEntry.remove()
7    synchronized {
8        timerTaskEntry.synchronized {
9            if (timerTaskEntry.list == null) {
10                val tail = root.prev
11                timerTaskEntry.next = root
12                timerTaskEntry.prev = tail
13                timerTaskEntry.list = this
14                // 把timerTaskEntry添加到链表末尾
15                tail.next = timerTaskEntry
16                root.prev = timerTaskEntry
17                taskCounter.incrementAndGet()
18                done = true
19            }
20        }
21    }
22 }
23 }
24 // remove方法
25 def remove(timerTaskEntry: TimerTaskEntry): Unit = {
26     synchronized {
27         timerTaskEntry.synchronized {
28             if (timerTaskEntry.list eq this) {
29                 timerTaskEntry.next.prev = timerTaskEntry.prev
30                 timerTaskEntry.prev.next = timerTaskEntry.next
31                 timerTaskEntry.next = null
32                 timerTaskEntry.prev = null
33                 timerTaskEntry.list = null
34                 taskCounter.decrementAndGet()
35             }
36         }
37     }
38 }


```

最后，我们看看 flush 方法。它的代码如下：

```

1 def flush(f: (TimerTaskEntry)=>Unit): Unit = {
2     synchronized {
3         // 找到链表第一个元素
4         var head = root.next
5         // 开始遍历链表
6         while (head ne root) {
7             // 移除遍历到的链表元素
8             remove(head)
9             // 执行传入参数f的逻辑
10            f(head)
11            head = root.next

```


 复制代码

```
12     }
13     // 清空过期时间设置
14     expiration.set(-1L)
15 }
16 }
```

基本上，flush 方法是清空链表中的所有元素，并对每个元素执行指定的逻辑。该方法用于将高层次时间轮 Bucket 上的定时任务重新插入回低层次的 Bucket 中。具体为什么要这么做，下节课我会给出答案，现在你只需要知道它的大致作用就可以了。

## TimingWheel 类

最后，我们再来看下 TimingWheel 类的代码。先看定义：

 复制代码

```
1 private[timer] class TimingWheel(
2     tickMs: Long, wheelSize: Int,
3     startMs: Long, taskCounter: AtomicInteger,
4     queue: DelayQueue[TimerTaskList]) {
5     private[this] val interval = tickMs * wheelSize
6     private[this] val buckets = Array.tabulate[TimerTaskList](wheelSize) { _, _ => null }
7     private[this] var currentTime = startMs - (startMs % tickMs)
8     @volatile private[this] var overflowWheel: TimingWheel = null
9     .....
10 }
11
```

每个 TimingWheel 对象都定义了 9 个字段。这 9 个字段都非常重要，每个字段都是分层时间轮的重要属性。因此，我来逐一介绍下。

tickMs：滴答一次的时长，类似于手表的例子中向前推进一格的时间。对于秒针而言，tickMs 就是 1 秒。同理，分针是 1 分，时针是 1 小时。在 Kafka 中，第 1 层时间轮的 tickMs 被固定为 1 毫秒，也就是说，向前推进一格 Bucket 的时长是 1 毫秒。

wheelSize：每一层时间轮上的 Bucket 数量。第 1 层的 Bucket 数量是 20。

startMs：时间轮对象被创建时的起始时间戳。

taskCounter：这一层时间轮上的总定时任务数。

queue: 将所有 Bucket 按照过期时间排序的延迟队列。随着时间不断向前推进, Kafka 需要依靠这个队列获取那些已过期的 Bucket, 并清除它们。

interval: 这层时间轮总时长, 等于滴答时长乘以 wheelSize。以第 1 层为例, interval 就是 20 毫秒。由于下一层时间轮的滴答时长就是上一层的总时长, 因此, 第 2 层的滴答时长就是 20 毫秒, 总时长是 400 毫秒, 以此类推。

buckets: 时间轮下的所有 Bucket 对象, 也就是所有 TimerTaskList 对象。

currentTime: 当前时间戳, 只是源码对它进行了一些微调整, 将它设置成小于当前时间的最大滴答时长的整数倍。举个例子, 假设滴答时长是 20 毫秒, 当前时间戳是 123 毫秒, 那么, currentTime 会被调整为 120 毫秒。

overflowWheel: Kafka 是按需创建上层时间轮的。这也就是说, 当有新的定时任务到达时, 会尝试将其放入第 1 层时间轮。如果第 1 层的 interval 无法容纳定时任务的超时时间, 就现场创建并配置好第 2 层时间轮, 并再次尝试放入, 如果依然无法容纳, 那么, 就再创建和配置第 3 层时间轮, 以此类推, 直到找到适合容纳该定时任务的第 N 层时间轮。

由于每层时间轮的长度都是倍增的, 因此, 代码并不需要创建太多层的时间轮, 就足以容纳绝大部分的延时请求了。

举个例子, 目前 Clients 端默认的请求超时时间是 30 秒, 按照现在代码中的 wheelSize=20 进行倍增, 只需要 4 层时间轮, 就能容纳 160 秒以内的所有延时请求了。

说完了类声明, 我们再来学习下 TimingWheel 中定义的 3 个方法:

addOverflowWheel、add 和 advanceClock。就像我前面说的, TimingWheel 类字段 overflowWheel 的创建是按需的。每当需要一个新的上层时间轮时, 代码就会调用 addOverflowWheel 方法。我们看下它的代码:

 复制代码

```
1 private[this] def addOverflowWheel(): Unit = {
2     synchronized {
3         // 只有之前没有创建上层时间轮方法才会继续
4         if (overflowWheel == null) {
5             // 创建新的TimingWheel实例
6             // 滴答时长tickMs等于下层时间轮总时长
7             // 每层的轮子数都是相同的
8             overflowWheel = new TimingWheel(
9                 tickMs = interval,
```

```

10         wheelSize = wheelSize,
11         startMs = currentTime,
12         taskCounter = taskCounter,
13         queue
14     )
15 }
16 }
17

```

这个方法就是创建一个新的 TimingWheel 实例，也就是创建上层时间轮。所用的滴答时长等于下层时间轮总时长，而每层的轮子数都是相同的。创建完成之后，代码将新创建的实例赋值给 overflowWheel 字段。至此，方法结束。

下面，我们再来学习下 add 和 advanceClock 方法。首先是 add 方法，代码及其注释如下：

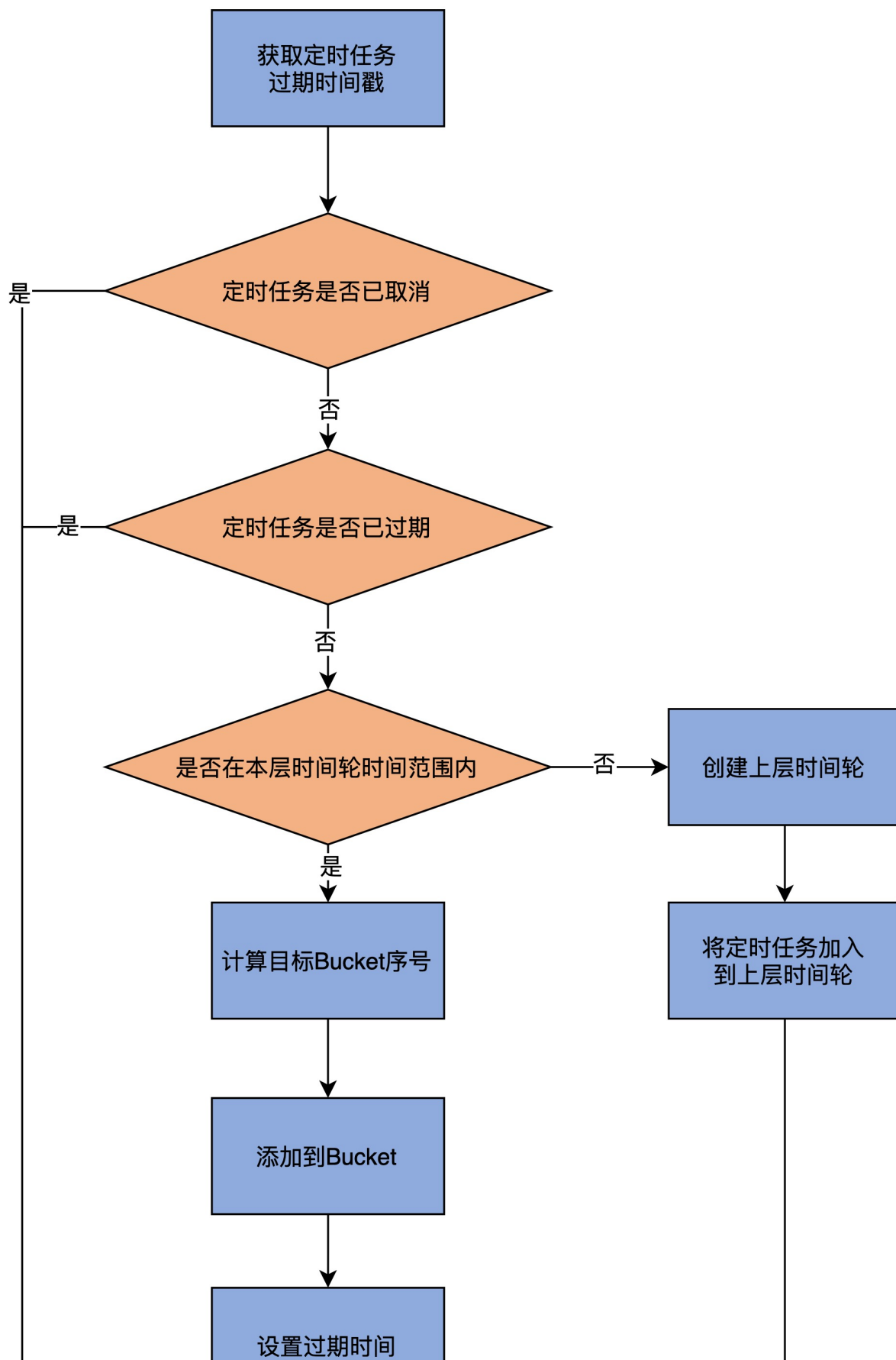
 复制代码

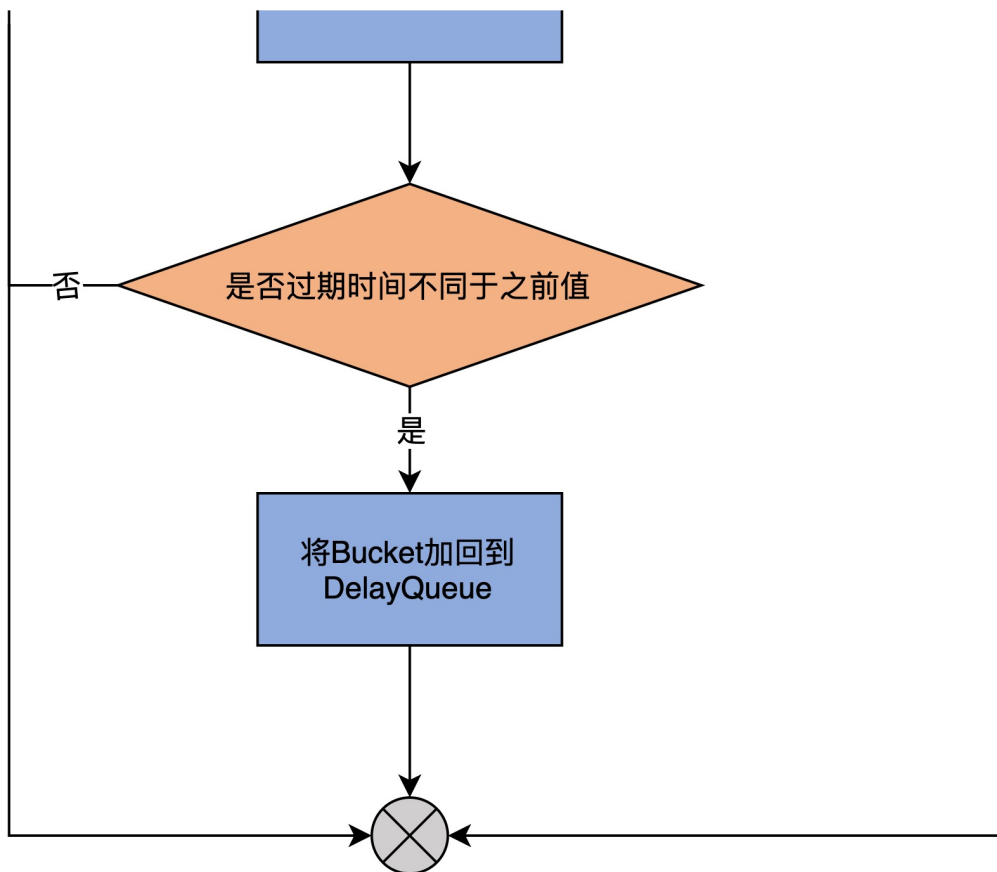
```

1  def add(timerTaskEntry: TimerTaskEntry): Boolean = {
2      // 获取定时任务的过期时间戳
3      val expiration = timerTaskEntry.expirationMs
4      // 如果该任务已然被取消了，则无需添加，直接返回
5      if (timerTaskEntry.cancelled) {
6          false
7      } // 如果该任务超时时间已过期
8      else if (expiration < currentTime + tickMs) {
9          false
10     } // 如果该任务超时时间在本层时间轮覆盖时间范围内
11     else if (expiration < currentTime + interval) {
12         val virtualId = expiration / tickMs
13         // 计算要被放入到哪个Bucket中
14         val bucket = buckets((virtualId % wheelSize.toLong).toInt)
15         // 添加到Bucket中
16         bucket.add(timerTaskEntry)
17         // 设置Bucket过期时间
18         // 如果该时间变更过，说明Bucket是新建或被重用，将其加回到DelayQueue
19         if (bucket.setExpiration(virtualId * tickMs)) {
20             queue.offer(bucket)
21         }
22         true
23     } // 本层时间轮无法容纳该任务，交由上层时间轮处理
24     else {
25         // 按需创建上层时间轮
26         if (overflowWheel == null) addOverflowWheel()
27         // 加入到上层时间轮中
28         overflowWheel.add(timerTaskEntry)
29     }
30 }

```

我结合一张图来解释下这个 add 方法要做的事情：





方法的**第 1 步**是获取定时任务的过期时间戳。所谓过期时间戳，就是这个定时任务过期时的时点。

**第 2 步**是看定时任务是否已被取消。如果已经被取消，则无需加入到时间轮中。如果没有被取消，就接着看这个定时任务是否已经过期。如果过期了，自然也不用加入到时间轮中。如果没有过期，就看这个定时任务的过期时间是否能够被涵盖在本层时间轮的时间范围内。如果可以，则进入到下一步。

**第 3 步**，首先计算目标 Bucket 序号，也就是这个定时任务需要被保存在哪个 TimerTaskList 中。我举个实际的例子，来说明一下如何计算目标 Bucket。

前面说过了，第 1 层的时间轮有 20 个 Bucket，每个滴答时长是 1 毫秒。那么，第 2 层时间轮的滴答时长应该也就是 20 毫秒，总时长是 400 毫秒。第 2 层第 1 个 Bucket 的时间范围应该是 $[20, 40)$ ，第 2 个 Bucket 的时间范围是 $[40, 60)$ ，依次类推。假设现在有个延时请求的超时时间戳是 237，那么，它就应该被插入到第 11 个 Bucket 中。



在确定了目标 Bucket 序号之后，代码会将该定时任务添加到这个 Bucket 下，同时更新这个 Bucket 的过期时间戳。在刚刚的那个例子中，第 11 号 Bucket 的起始时间就应该是小于 237 的最大的 20 的倍数，即 220。

**第 4 步**，如果这个 Bucket 是首次插入定时任务，那么，还同时要将这个 Bucket 加入到 DelayQueue 中，方便 Kafka 轻松地获取那些已过期 Bucket，并删除它们。如果定时任务的过期时间无法被涵盖在本层时间轮中，那么，就按需创建上一层时间戳，然后在上一层时间轮上完整地执行刚刚所说的所有逻辑。

说完了 add 方法，我们看下 advanceClock 方法。顾名思义，它就是向前驱动时钟的方法。代码如下：

 复制代码

```
1 def advanceClock(timeMs: Long): Unit = {
2     // 向前驱动到的时点要超过Bucket的时间范围，才是有意义的推进，否则什么都不做
3     // 更新当前时间currentTime到下一个Bucket的起始时点
4     if (timeMs >= currentTime + tickMs) {
5         currentTime = timeMs - (timeMs % tickMs)
6         // 同时尝试为上一层时间轮做向前推进动作
7         if (overflowWheel != null) overflowWheel.advanceClock(currentTime)
8     }
9 }
```

参数 timeMs 表示要把时钟向前推动到这个时点。向前驱动到的时点必须要超过 Bucket 的时间范围，才是有意义的推进，否则什么都不做，毕竟它还在 Bucket 时间范围内。

相反，一旦超过了 Bucket 覆盖的时间范围，代码就会更新当前时间 currentTime 到下一个 Bucket 的起始时点，同时递归地为上一层时间轮做向前推进动作。推进时钟的动作是由 Kafka 后台专属的 Reaper 线程发起的。

今天，我反复提到了删除过期 Bucket，这个操作是由这个 Reaper 线程执行的。下节课，我们会提到这个 Reaper 线程。

## 总结

今天，我简要介绍了时间轮机制，并结合代码重点讲解了分层时间轮在 Kafka 中的代码实现。Kafka 正是利用这套分层时间轮机制实现了对于延迟请求的处理。在源码层级上，

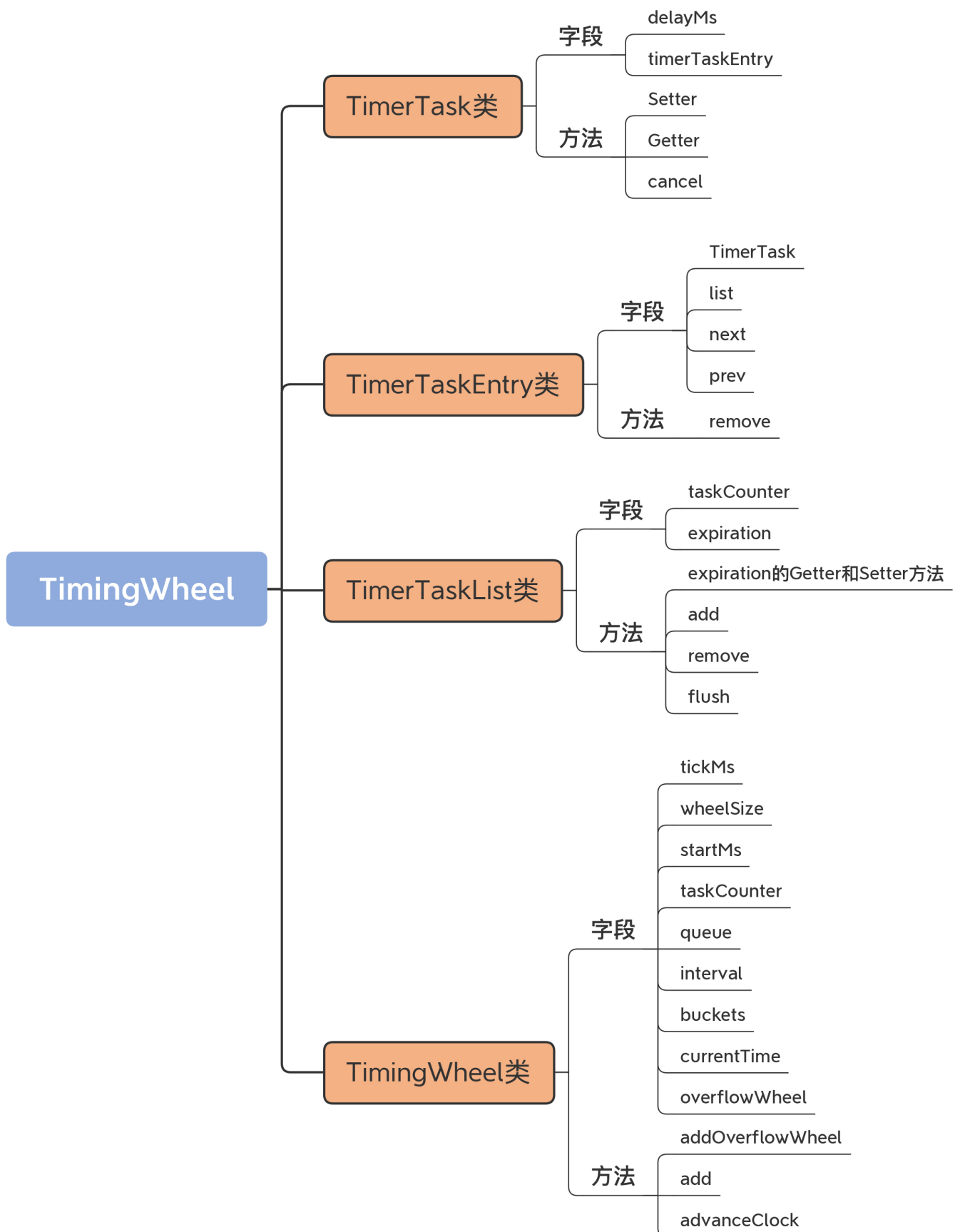
Kafka 定义了 4 个类来构建整套分层时间轮体系。

TimerTask 类：建模 Kafka 延时请求。它是一个 Runnable 类，Kafka 使用一个单独线程异步添加延时请求到时间轮。

TimerTaskEntry 类：建模时间轮 Bucket 下延时请求链表的元素类型，封装了 TimerTask 对象和定时任务的过期时间戳信息。

TimerTaskList 类：建模时间轮 Bucket 下的延时请求双向循环链表，提供  $O(1)$  时间复杂度的请求插入和删除。

TimingWheel 类：建模时间轮类型，统一管理下辖的所有 Bucket 以及定时任务。



解，当请求不能被及时处理时，Kafka 是如何应对的。

在分布式系统中，如何优雅而高效地延迟处理任务是摆在设计者面前的难题之一。我建议你好好学习下这套实现机制在 Kafka 中的应用代码，活学活用，将其彻底私有化，加入到你的工具箱中。

## 课后讨论

TimingWheel 类中的 overflowWheel 变量为什么是 volatile 型的？

欢迎你在留言区畅所欲言，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。

上一篇 18 | PartitionStateMachine：揭秘分区状态机实现原理

下一篇 20 | DelayedOperation：Broker是怎么延时处理请求的？

## 精选留言 (3)



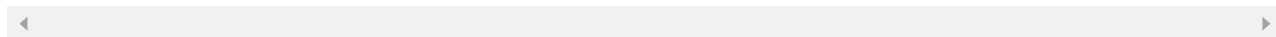
RonnieXie

2020-06-07

写留言

老师，这里有一个疑问，使用哈希表map似乎也可以实现延迟队列，key为时间戳，定期执行任务和删除过期任务，时间复杂度 $O(1)$ ，请问使用分层时间轮和哈希表map的优缺点是什么？

作者回复: 如何解决排序问题呢？



1



空知

2020-06-07

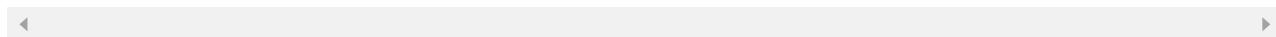
只有当前时间越过了 Bucket 的起始时间，这个 Bucket 才算是过期。而这里的起始时间，就是代码中 expiration 字段的值。这里的起始时间是否应该是 终止时间？

taskCounter：这一层时间轮上的总定时任务数。这里是否是每个Bucket的任务数？

展开

作者回复: 1. 源码中并没有终止时间的提法。一旦时钟越过了Bucket起始时间，该Bucket就被视为过期了

2. 不是单个Bucket的，而是整个时间轮上的



1



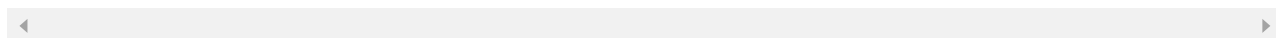
伯安知心

2020-06-06

首先声明TimingWheel不是线程安全的，addOverflowWheel这个方法设计本身要单例模式，但是多个线程执行addOverflowWheel方法，可能出现不一致实例化多个类，如果是volatile限制了指令重排序，就解决了这个问题。首先声明TimingWheel不是线程安全的，addOverflowWheel这个方法设计本身要单例模式，但是多个线程执行addOverflowWheel方法，可能出现不一致实例化多个类，如果是volatile限制了指令重排序，就解决了这个...

展开

作者回复: 不妨写个改进的patch：)



1

