35-两阶段终止模式：如何优雅地终止线程？

前面两篇文章我们讲述的内容，从纯技术的角度看，都是**启动**多线程去执行一个异步任务。既启动，那又该如何终止呢？今天咱们就从技术的角度聊聊如何优雅地**终止**线程，正所谓有始有终。

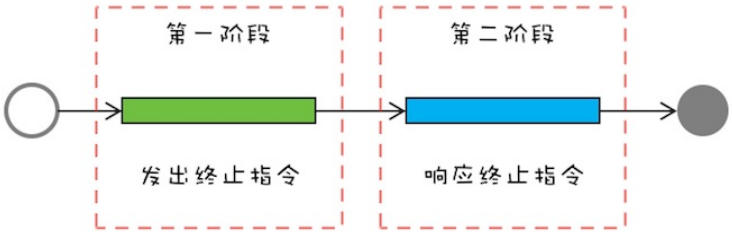
在《09 | Java线程（上）：Java线程的生命周期》中，我曾讲过：线程执行完或者出现异常就会进入终止状态。这样看，终止一个线程看上去很简单啊！一个线程执行完自己的任务，自己进入终止状态，这的确很简单。不过我们今天谈到的“优雅地终止线程”，不是自己终止自己，而是在一个线程T1中，终止线程T2；这里所谓的“优雅”，指的是给T2一个机会料理后事，而不是被一剑封喉。

Java语言的Thread类中曾经提供了一个stop()方法，用来终止线程，可是早已不建议使用了，原因是这个方法用的就是一剑封喉的做法，被终止的线程没有机会料理后事。

既然不建议使用stop()方法，那在Java领域，我们又该如何优雅地终止线程呢？

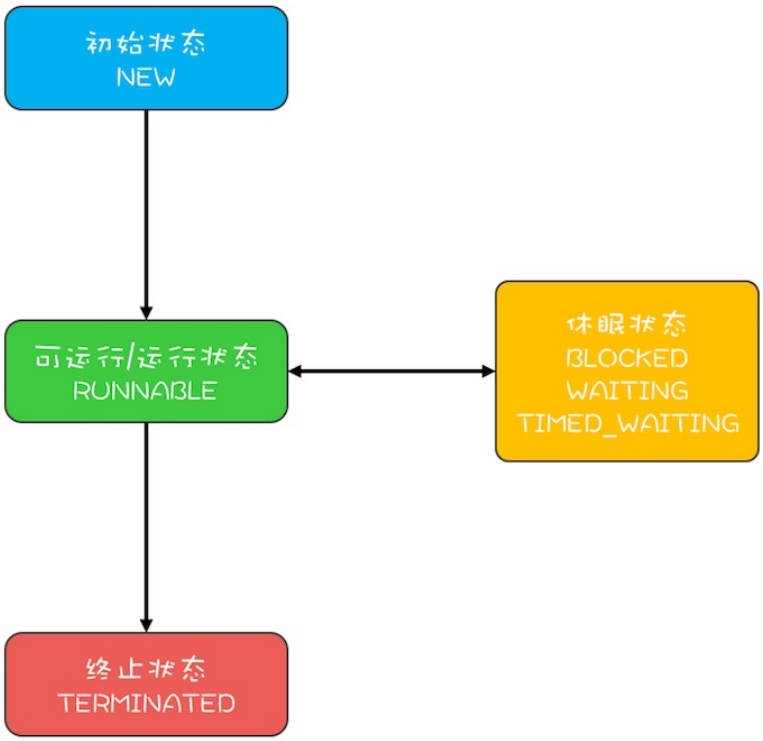
如何理解两阶段终止模式

前辈们经过认真对比分析，已经总结出了一套成熟的方案，叫做**两阶段终止模式**。顾名思义，就是将终止过程分成两个阶段，其中第一个阶段主要是线程T1向线程T2**发送终止指令**，而第二阶段则是线程T2**响应终止指令**。



两阶段终止模式示意图

那在Java语言里，终止指令是什么呢？这个要从Java线程的状态转换过程说起。我们在《09 | Java线程（上）：Java线程的生命周期》中曾经提到过Java线程的状态转换图，如下图所示。



Java中的线程状态转换图

从这个图里你会发现，Java线程进入终止状态的前提是线程进入RUNNABLE状态，而实际上线程也可能处在休眠状态，也就是说，我们要想终止一个线程，首先要把线程的状态从休眠状态转换到RUNNABLE状态。如何做到呢？这个要靠Java Thread类提供的**interrupt()方法**，它可以将休眠状态的线程转换到RUNNABLE状态。

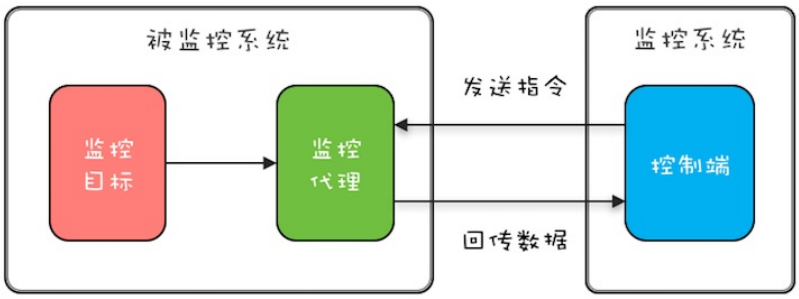
线程转换到RUNNABLE状态之后，我们如何再将其终止呢？RUNNABLE状态转换到终止状态，优雅的方式是让Java线程自己执行完 run() 方法，所以一般我们采用的方法是**设置一个标志位**，然后线程会在合适的时机检查这个标志位，如果发现符合终止条件，则自动退出run()方法。这个过程其实就是我们前面提到的第二阶段：**响应终止指令**。

综合上面这两点，我们能总结出终止指令，其实包括两方面内容：**interrupt()方法**和**线程终止的标志位**。

理解了两阶段终止模式之后，下面我们看一个实际工作中的案例。

用两阶段终止模式终止监控操作

实际工作中，有些监控系统需要动态地采集一些数据，一般都是监控系统发送采集指令给被监控系统的监控代理，监控代理接收到指令之后，从监控目标收集数据，然后回传给监控系统，详细过程如下图所示。出于对性能的考虑（有些监控项对系统性能影响很大，所以不能一直持续监控），动态采集功能一般都会有终止操作。



动态采集功能示意图

下面的示例代码是**监控代理**简化之后的实现，start()方法会启动一个新的线程rptThread来执行监控数据采集和回传的功能，stop()方法需要优雅地终止线程rptThread，那stop()相关功能该如何实现呢？

```
class Proxy {
  boolean started = false;
  //采集线程
  Thread rptThread;
  //启动采集功能
  synchronized void start(){
    //不允许同时启动多个采集线程
    if (started) {
      return;
    }
    started = true;
    rptThread = new Thread(()->{
      while (true) {
        //省略采集、回传实现
        report();
        //每隔两秒钟采集、回传一次数据
        try {
          Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
      }
      //执行到此处说明线程马上终止
      started = false;
    });
    rptThread.start();
  }
  //终止采集功能
  synchronized void stop(){
    //如何实现?
  }
}
```

按照两阶段终止模式，我们首先需要做的就是将线程rptThread状态转换到RUNNABLE，做法很简单，只需要在调用 `rptThread.interrupt()` 就可以了。线程rptThread的状态转换到RUNNABLE之后，如何优雅地终止呢？下面的示例代码中，我们选择的标志位是线程的中断状态：`Thread.currentThread().isInterrupted()`，需要注意的是，我们在捕获Thread.sleep()的中断异常之后，通过 `Thread.currentThread().interrupt()` 重新设置了线程的中断状态，因为JVM的异常处理会清除线程的中断状态。

```
class Proxy {
  boolean started = false;
  //采集线程
  Thread rptThread;
  //启动采集功能
  synchronized void start(){
    //不允许同时启动多个采集线程
    if (started) {
      return;
    }
    started = true;
    rptThread = new Thread(()->{
      while (!Thread.currentThread().isInterrupted()){
        //省略采集、回传实现
        report();
        //每隔两秒钟采集、回传一次数据
        try {
          Thread.sleep(2000);
        } catch (InterruptedException e){
          //重新设置线程中断状态
          Thread.currentThread().interrupt();
        }
      }
      //执行到此处说明线程马上终止
      started = false;
    });
    rptThread.start();
  }
  //终止采集功能
  synchronized void stop(){
    rptThread.interrupt();
  }
}
```

上面的示例代码的确能够解决当前的问题，但是建议你在实际工作中谨慎使用。原因在于我们很可能在线程的run()方法中调用第三方类库提供的方法，而我们没有办法保证第三方类库正确处理了线程的中断异常，例如第三方类库在捕获到Thread.sleep()方法抛出的中断异常后，没有重新设置线程的中断状态，那么就会导致线程不能够正常终止。所以强烈建议你**设置自己的线程终止标志位**，例如在下面的代码中，使用isTerminated作为线程终止标志位，此时无论是否正确处理了线程的中断异常，都不会影响线程优雅地终止。

```
class Proxy {
  //线程终止标志位
  volatile boolean terminated = false;
  boolean started = false;
  //采集线程
  Thread rptThread;
  //启动采集功能
  synchronized void start(){
    //不允许同时启动多个采集线程
    if (started) {
      return;
    }
    started = true;
    terminated = false;
    rptThread = new Thread(()->{
```

```
      while (!terminated){
        //省略采集、回传实现
        report();
        //每隔两秒钟采集、回传一次数据
        try {
          Thread.sleep(2000);
        } catch (InterruptedException e){
          //重新设置线程中断状态
          Thread.currentThread().interrupt();
        }
      }
      //执行到此处说明线程马上终止
      started = false;
    });
    rptThread.start();
  }
  //终止采集功能
  synchronized void stop(){
    //设置中断标志位
    terminated = true;
    //中断线程rptThread
    rptThread.interrupt();
  }
}
```

如何优雅地终止线程池

Java领域用的最多的还是线程池，而不是手动地创建线程。那我们该如何优雅地终止线程池呢？

线程池提供了两个方法：**shutdown()和shutdownNow()**。这两个方法有什么区别呢？要了解它们的区别，就先需要了解线程池的实现原理。

我们曾经讲过，Java线程池是生产者-消费者模式的一种实现，提交给线程池的任务，首先是进入一个阻塞队列中，之后线程池中的线程从阻塞队列中取出任务执行。

shutdown()方法是一种很保守的关闭线程池的方法。线程池执行shutdown()后，就会拒绝接收新的任务，但是会等待线程池中正在执行的任务和已经进入阻塞队列的任务都执行完之后才最终关闭线程池。

而shutdownNow()方法，相对就激进一些了，线程池执行shutdownNow()后，会拒绝接收新的任务，同时还会中断线程池中正在执行的任务，已经进入阻塞队列的任务也被剥夺了执行的机会，不过这些被剥夺执行机会的任务会作为shutdownNow()方法的返回值返回。因为shutdownNow()方法会中断正在执行的线程，所以提交到线程池的任务，如果需要优雅地结束，就需要正确地处理线程中断。

如果提交到线程池的任务不允许取消，那就不能使用shutdownNow()方法终止线程池。不过，如果提交到线程池的任务允许后续以补偿的方式重新执行，也是可以使用shutdownNow()方法终止线程池的。《Java并发编程实战》这本书第7章《取消与关闭》的“shutdownNow的局限性”一节中，提到一种将已提交但尚未开始执行的任务以及已经取消的正在执行的任务保存起来，以便后续重新执行的方案，你可以参考一下，方案很简答，这里就不详细介绍了。

其实分析完shutdown()和shutdownNow()方法你会发现，它们实质上使用的也是两阶段终止模式，只是终止指令的范围不同而已，前者只影响阻塞队列接收任务，后者范围扩大到线程池中所有的任务。

总结

两阶段终止模式是一种应用很广泛的并发设计模式，在Java语言中使用两阶段终止模式来优雅地终止线程，需要注意两个关键点：一个是仅检查终止标志位是不够的，因为线程的状态可能处于休眠态；另一个是仅检查线程的中断状态也是不够的，因为我们依赖的第三方类库很可能没有正确处理中断异常。

当你使用Java的线程池来管理线程的时候，需要依赖线程池提供的shutdown()和shutdownNow()方法来终止线程池。不过在使用时需要注意它们的应用场景，尤其是在使用shutdownNow()的时候，一定要谨慎。

课后思考

本文的示例代码中，线程终止标志位isTerminated被声明为volatile，你觉得是否有必要呢？

```
class Proxy {
  //线程终止标志位
  volatile boolean terminated = false;
  ......
}
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言：

- 孙志强 2019-05-18 09:37:06
  有必要，变量被多个线程访问，需要保证可见性 [4赞]

  作者回复2019-05-18 16:30:51
  👍

- ZOU志伟 2019-05-18 17:14:26
  shutdown()调用后，还要再调用awaitTermination方法等待一点时间，线程池里的线程才会终止。 [2赞]

- 晓杰 2019-05-19 14:02:06
  有必要，因为stop方法对isTerminated的修改需要被start方法读取到，保证共享变量的可见性 [1赞]

- WL 2019-05-20 20:27:52
  老师请问一下对"没有办法保证第三方类库正确处理了线程的中断异常"这句话我不是很理解, 例子中的sleep方法不是在本地执行的吗, 为什么会跟第三方类库有关系？

  作者回复2019-05-20 21:10:36
  实际场景都不会这么简单

- WL 2019-05-20 20:21:02
  请问一下老师"JVM 的异常处理会清除线程的中断状态"指的是什么意思, 是指把线程的为true的中断状态改为false吗, JVM是在catch到Interrupt异常的时候重置线程中断状态的吗?

  作者回复2019-05-20 21:11:23
  是的

- 佑儿 2019-05-20 18:28:11
  两阶段终止模式：发送终止指令+响应终止指令。
  终止指令通常可以定义一个终止标识变量(注意并发问题,需要volatie保证可见性)。
  如果线程中调用了可中断方法(wait等)，在发送终止指令的同时需要调用Thread.interrupt()。
  不建议使用线程自身的中断标识作为终止指令，因为项目中第三方的调用无法保证该标志位。

- 佑儿 2019-05-20 17:56:59
  stop和start方法对于terminated访问由于syn关键字，线程安全，但是start中新起了一个线程rptthread，导致stop方法中对于terminated存在可见性问题，因此需要volatie，原子性问题对这个代码段没有影响，所以原子性问题无需关注。

  作者回复2019-05-20 21:12:18
  👍

- 包子 2019-05-20 10:15:31
  老师你好，
  优雅关闭线程最后一个案例代码使用terminated作为线程中断的标志位，那cache住sleep时，不用设置terminated为true吗。不设置线程是关闭不了的啊。
  try {
  Thread.sleep(2000);
  } catch (InterruptedException e){
  // 重新设置线程中断状态
  Thread.currentThread().interrupt();
  }

  作者回复2019-05-20 21:14:03
  stop方法里设置过了

- Liam 2019-05-20 08:21:51
  所以优雅停止线程的关键在于如何优雅地处理中断

- 搏未来 2019-05-20 07:43:47
  有必要，保持可见性.

- 李林杰 2019-05-19 22:41:58
  老师，started是否需要加voliatle保证，多个线程看到的都是最新的值

  作者回复2019-05-20 08:45:46
  synchronized 就不需要了

- 遇见阳光 2019-05-18 23:25:59
  按道理而言，synchronized保证原子性的同时，也能间接的保证可见性啊。感觉可以不加 volatile关键字

  作者回复2019-05-20 09:03:31
  问题是start方法里又启动了一个新的线程，synchronized管不到这个新的线程

- 兔斯基 2019-05-18 15:31:46
  有必要，项目中就是这么设置的，踩过一次了坑了

- ZOU志伟 2019-05-18 12:08:46
  有必要，stop()方法执行后，读terminated的start()方法要再次执行才会可见

  作者回复2019-05-18 16:30:24
  👍

- ack 2019-05-18 10:23:12
  有必要，假设不加volatile， 那么在while (!terminated)时读terminated的值可能会在stop()方法中写terminated值之前，并且stop()方法执行的线程A和在while循环的线程B是不同的，也就是说都有自己变量副本，A线程在自己缓存设置值后不知道什么时候才写回内存，可能导致中断延迟。

- 南北少卿 2019-05-18 10:17:46
  start() 和stop()都是使用的同一把锁this，这样的话，没必要再用volatile声明isTerminated了吧

  作者回复2019-05-18 16:29:13
  有必要

- zero 2019-05-18 09:09:54
  是有必要的，保证可见性

- 锦 2019-05-18 08:48:43
  打卡，有必要

- echo__陈 2019-05-18 08:37:22
  我觉得，在本例子中。stop中，设置终止标识位对interupt是可见的。而interrupt对被中断线程检测到中断事件是可见的……根据传递性原则……我觉得本例子不需要volatile关键字。但平时开发中，一般会加上，主要是因为怕后续开发不注意这些事情导致修改破坏了规则，引起可见性问题产生bug，保险起见会加上volatile

  作者回复2019-05-18 16:42:35
  是的，线程不调用wait,sleep等方法，是无法响应中断的，这个时候基于interrupt的可见性就不成立了，所以工程上这类变量都需要加volatile

- Corner 2019-05-18 07:18:26
  多线程读写共享变量，保证可见性使用volitale没毛病