https://mp.weixin.qq.com/s/o4iZp4eR\_4zCzz6QJXRKiw

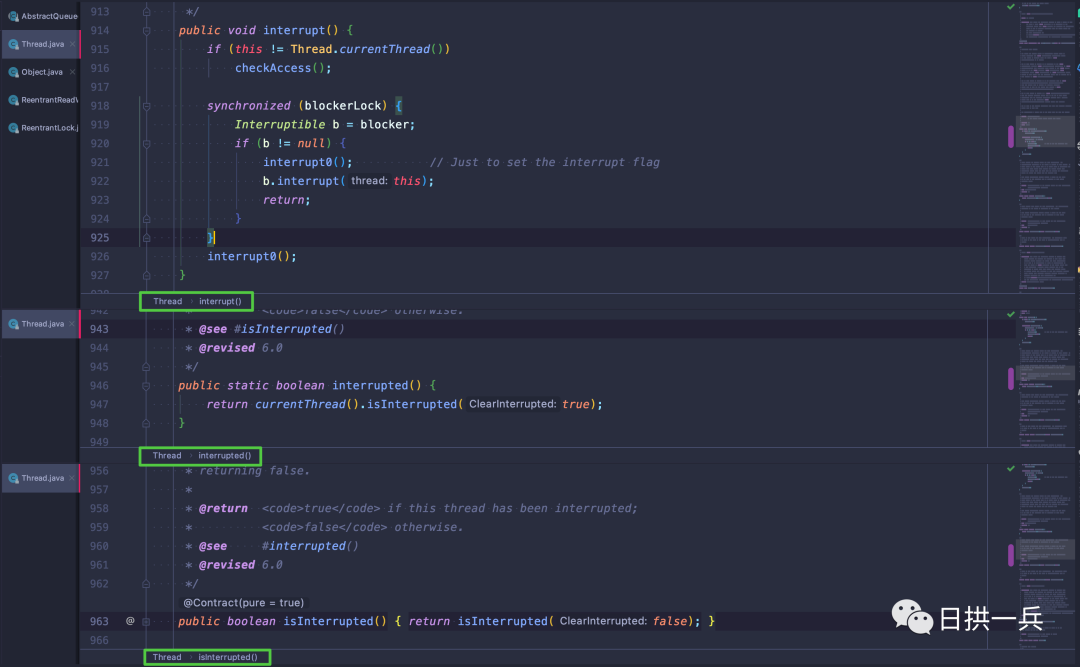
# Java并发编程的中断机制

在学习/编写并发程序时，总会听到/看到如下词汇：

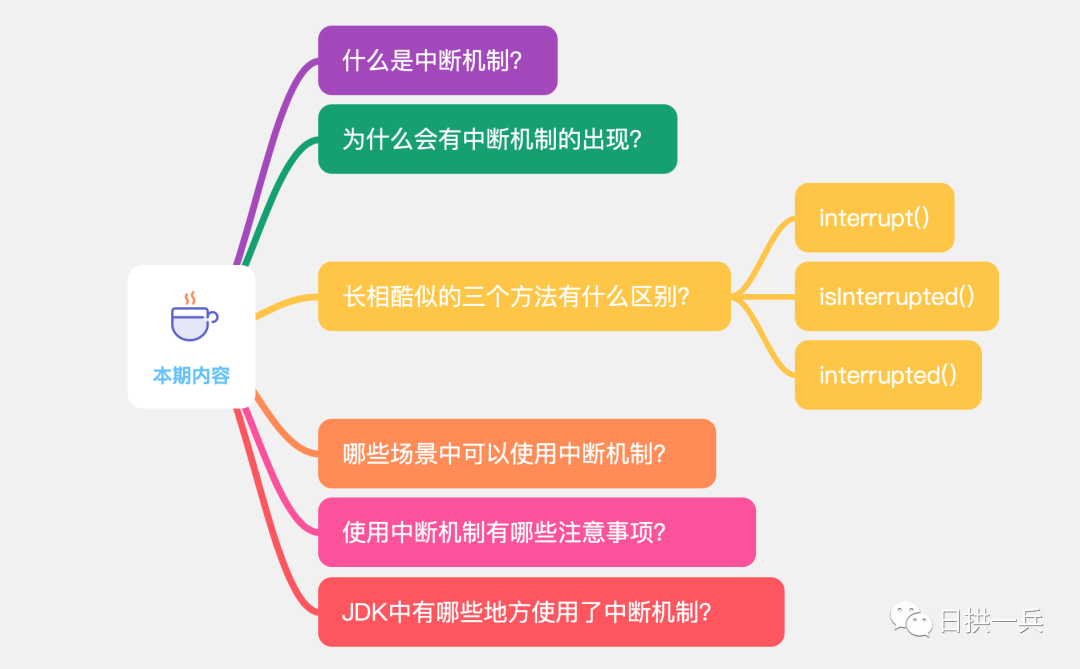
* 线程被中断或抛出InterruptedException
* 设置了中断标识
* 清空了中断标识
* 判断线程是否被中断

在 Java Thread 类又提供了长相酷似，让人傻傻分不清的三个方法来处理并发中断问题：

* interrupt()
* interrupted()
* isInterrupted()



看到这我不禁会问自己：



## **什么是中断机制？**



刚刚接触【中断】这个词时，先入为主的概念就是“直接中断/打断”正在做的事，使其停止。我的理解是这样的：

你：在打游戏

女朋友：别打游戏了，赶快过来吃饭

你：听到女朋友招呼之后立马中断手中的游戏乖乖过去吃饭

在多线程编程中，中断是一种【协同】机制，怎么理解这么高大上的词呢？就是女朋友叫你吃饭，你收到了中断游戏通知，**但是否马上放下手中的游戏去吃饭看你心情** 。在程序中怎样演绎这个心情就看具体的业务逻辑了，Java 的中断机制就是这么简单。

## **为什么会有中断机制？**

中断是一种协同机制，我觉得就是解决【当局者迷】的状况

现实中，你努力忘我没有昼夜的工作，如果再没有人告知你中断，你身体是吃不消的。

在多线程的场景中，有的线程可能迷失在怪圈无法自拔（自旋浪费资源），这时就可以用其他线程在**恰当**的时机给它个中断通知，被“中断”的线程可以选择在**恰当**的时机选择跳出怪圈，最大化的利用资源

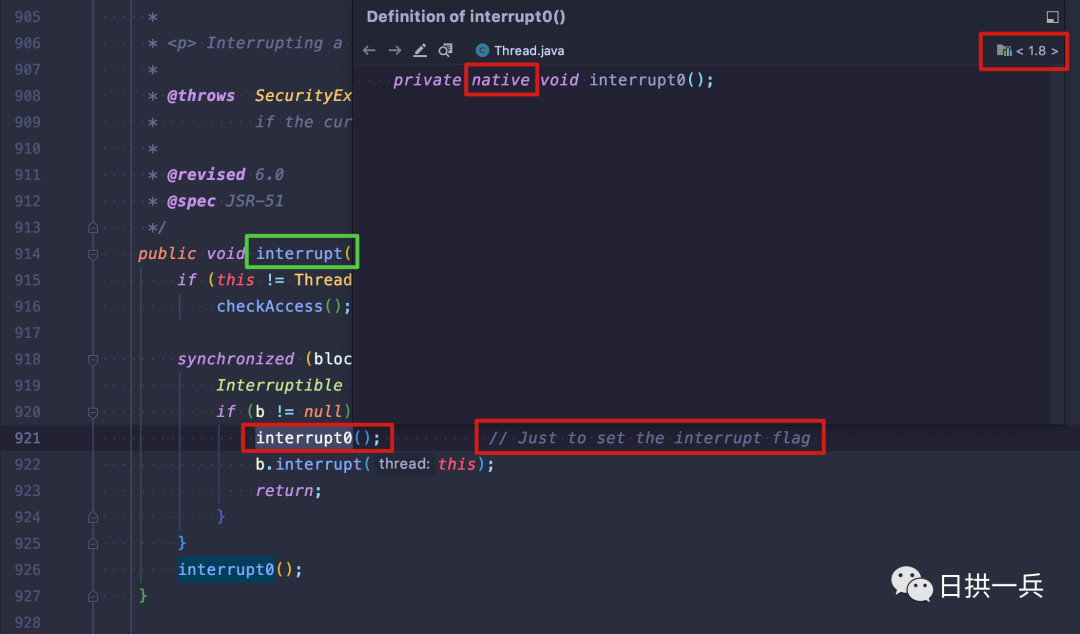
那程序中如何中断？怎样识别是否中断？又如何处理中断呢？这就与上文提到的三个方法有关了

## **interrupt() VS isInterrupted() VS interrupted()**

Java 的每个线程对象里都有一个 boolean 类型的标识，代表是否有中断请求，可你寻遍 Thread 类你也不会找到这个标识，因为这是通过底层 native 方法实现的。

### **interrupt()**

interrupt() 方法是 **唯一一个** 可以将上面提到中断标志设置为 true 的方法，从这里可以看出，这是一个 Thread 类 public 的对象方法，所以可以推断出任何线程对象都可以调用该方法，进一步说明就是可以一个线程 interrupt 其他线程，也可以 interrupt 自己。其中，中断标识的设置是通过 native 方法 interrupt0 完成的



在 Java 中，线程被中断的反应是不一样的，脾气不好的直接就抛出了 InterruptedException() ，

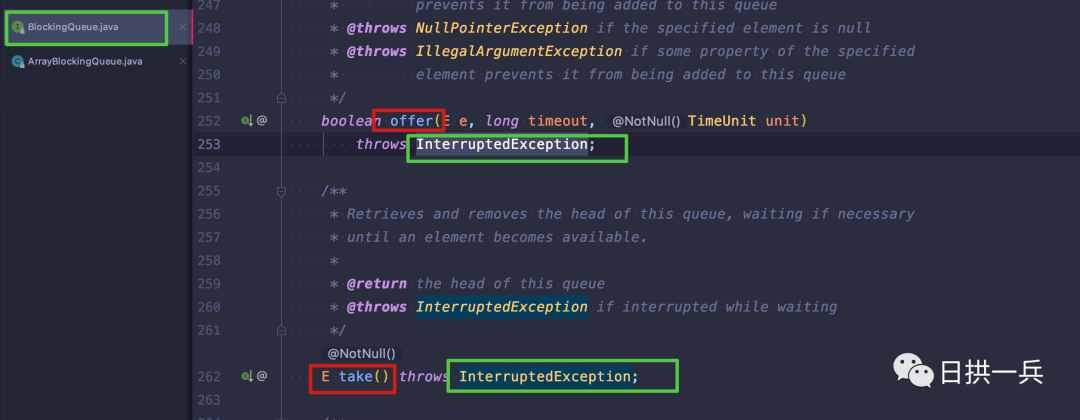


该方法注释上写的很清楚，当线程被阻塞在：

1. wait()
2. join()
3. sleep()

这些方法时，如果被中断，就会抛出 InterruptedException 受检异常（也就是必须要求我们 catch 进行处理的）

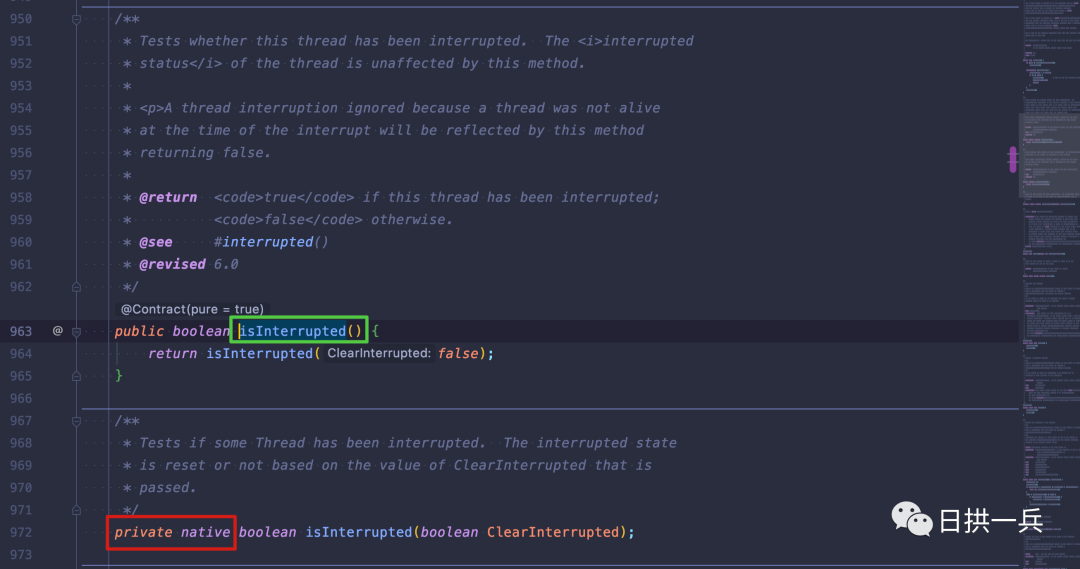
熟悉 JUC 的朋友可能知道，其实被中断抛出 InterruptedException 的远远不止这几个方法，比如：



反向推理，这些可能阻塞的方法如果声明有 throws InterruptedException ， 也就暗示我们它们是可中断的

调用 interrput() 方法后，中断标识就被设置为 true 了，那我们怎么利用这个中断标识，来判断某个线程中断标识到底什么状态呢？

### **isInterrupted()**



这个方法名起的非常好，因为比较符合我们 bean boolean 类型字段的 get 方法规范，没错，该方法就是返回中断标识的结果：

* true：线程被中断，
* false：线程没被中断或被清空了中断标识（如何清空我们一会儿看）

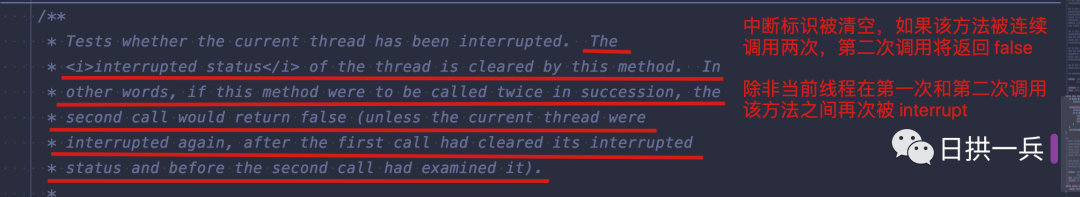
拿到这个标识后，线程就可以判断这个标识来执行后续的逻辑了。有起名好的，也有起名不好的，就是下面这个方法：

### **interrupted()**

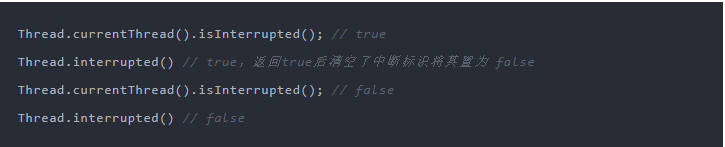
按照常规翻译，过去时时态，这就是“被打断了/被打断的”，其实和上面的 isInterrupted() 方法差不多，两个方法都是调用 private 的 isInterrupted() 方法， 唯一差别就是会清空中断标识（这是从方法名中怎么也看不出来的）



因为调用该方法，会返回当前中断标识，同时会清空中断标识，就有了那一段有点让人迷惑的方法注释：



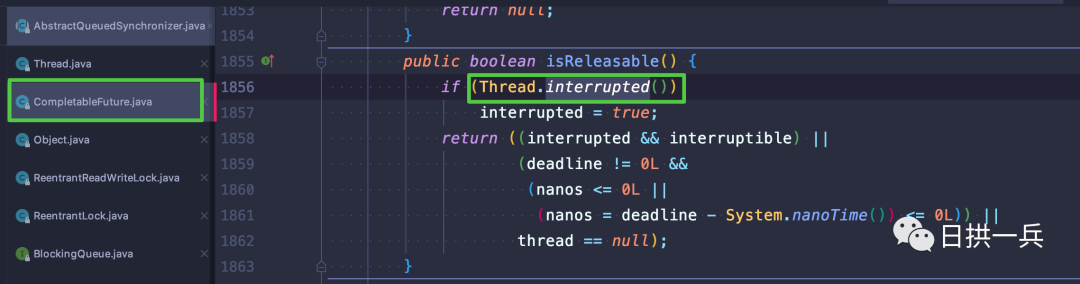
来段程序你就会明白上面注释的意思了：



这个方法总觉得很奇怪，现实中有什么用呢？

当你可能要被大量中断并且你想确保只处理一次中断时，就可以使用这个方法了

该方法在 JDK 源码中应用也非常多，比如（后续文章会具体分析，这里知道该方法的作用和使用场景就好）：



相信到这里你已经能明确分辨三胞胎都是谁，并发挥怎样的作用了，那么有哪些场景我们可以使用中断机制呢？

## **中断机制的使用场景**

通常，中断的使用场景有以下几个

* 点击某个桌面应用中的关闭按钮时（比如你关闭 IDEA，不保存数据直接中断好吗？）；
* 某个操作超过了一定的执行时间限制需要中止时；
* 多个线程做相同的事情，只要一个线程成功其它线程都可以取消时；
* 一组线程中的一个或多个出现错误导致整组都无法继续时；

因为中断是一种协同机制，提供了更优雅中断方式，也提供了更多的灵活性，所以当遇到如上场景等，我们就可以考虑使用中断机制了

## **使用中断机制有哪些注意事项**

其实使用中断机制无非就是注意上面说的两项内容：

1. 中断标识
2. InterruptedException

前浪已经将其总结为两个通用原则，我们后浪直接站在肩膀上用就可以了，来看一下这两个原则是什么：

### **原则-1**

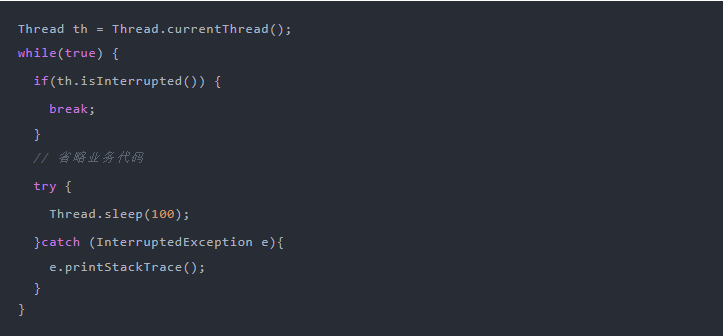
如果遇到的是可中断的阻塞方法, 并抛出 InterruptedException，可以继续向方法调用栈的上层抛出该异常；如果检测到中断，则可清除中断状态并抛出 InterruptedException，使当前方法也成为一个可中断的方法

### **原则-2**

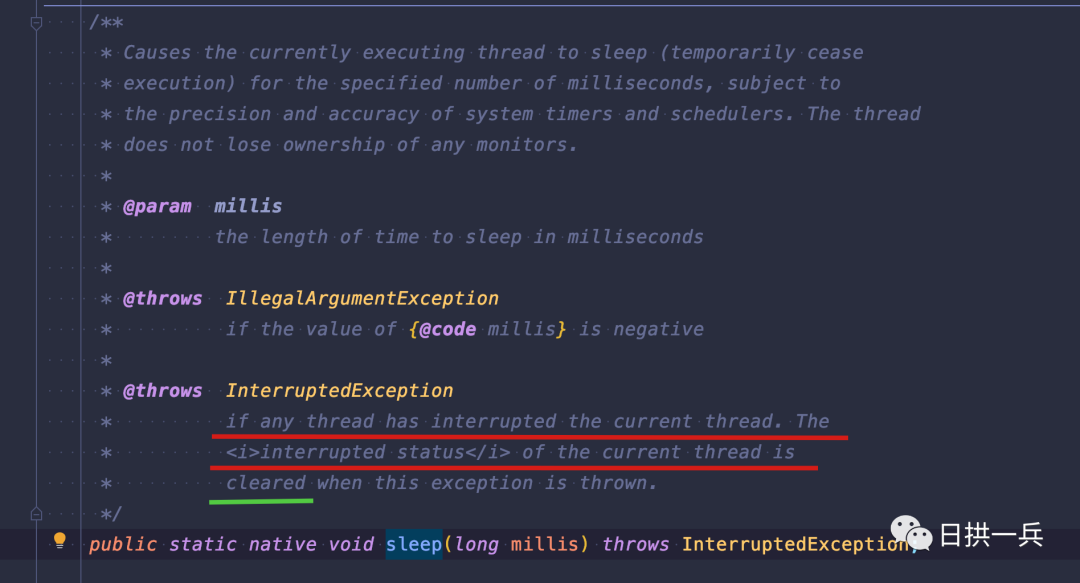
若有时候不太方便在方法上抛出 InterruptedException，比如要实现的某个接口中的方法签名上没有 throws InterruptedException，这时就可以捕获可中断方法的 InterruptedException 并通过 Thread.currentThread.interrupt() 来重新设置中断状态。

再通过个例子来加深一下理解：

本意是当前线程被中断之后，退出while(true),  你觉得代码有问题吗？（先不要向下看）



打开 Thread.sleep 方法：



sleep 方法抛出 InterruptedException后，中断标识也被清空置为 false，我们在catch 没有通过调用 th.interrupt() 方法再次将中断标识置为 true，这就导致无限循环了

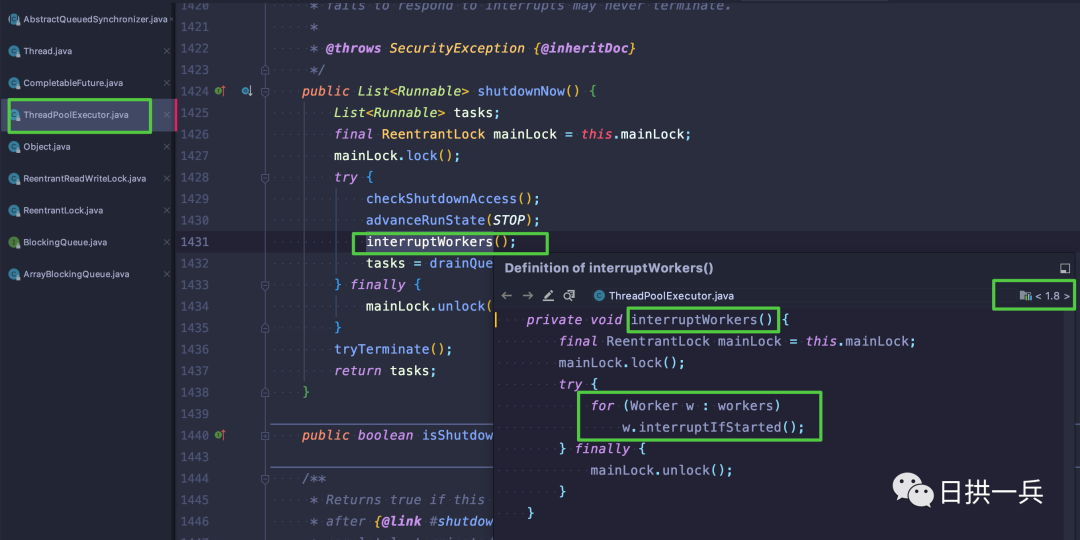
这两个原则很好理解。总的来说，我们应该留意 InterruptedException，当我们捕获到该异常时，绝不可以默默的吞掉它，什么也不做，因为这会导致上层调用栈什么信息也获取不到。其实在编写程序时，捕获的任何受检异常我们都不应该吞掉。

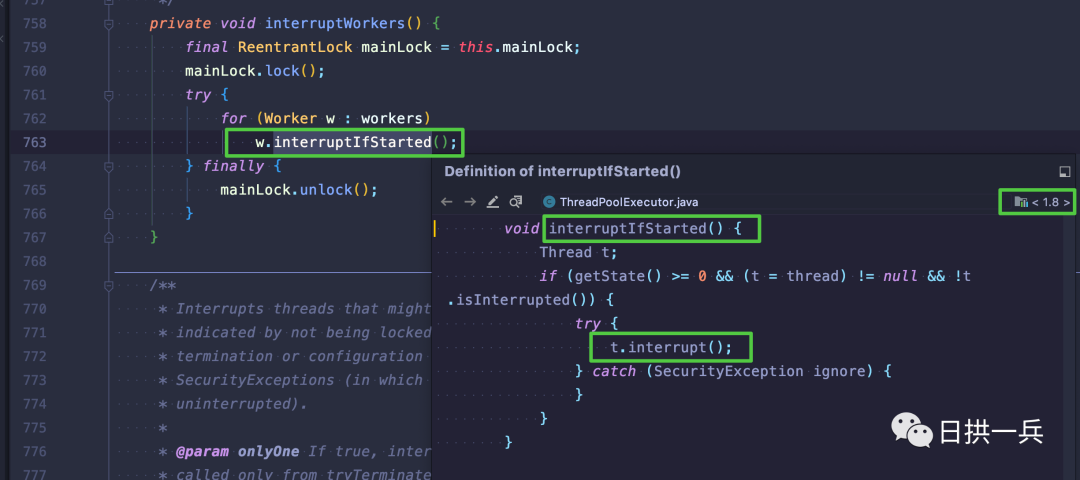
## **JDK 中有哪些使用中断机制的地方呢？**

中断机制贯穿整个并发编程中，这里只简单列觉大家经常会使用的，我们可以通过阅读JDK源码来进一步了解中断机制以及学习如何使用中断机制

### **ThreadPoolExecutor**

ThreadPoolExecutor 中的 shutdownNow 方法会遍历线程池中的工作线程并调用线程的 interrupt 方法来中断线程。





### **FutureTask**

FutureTask 中的 cancel 方法，如果传入的参数为 true，它将会在正在运行异步任务的线程上调用 interrupt 方法，如果正在执行的异步任务中的代码没有对中断做出响应，那么 cancel 方法中的参数将不会起到什么效果。



## **总结**

到这里你应该理解Java 并发编程中断机制的含义了，它是一种协同机制，和你先入为主的概念完全不一样。区分了三个相近方法，说明了使用场景以及使用原则，同时又给出JDK源码一些常见案例，相信你已经胸中有沟壑了，接下来，跟上节奏，我们陆续走进源码吧。

## **灵魂追问**

1. 抛出 InterruptedException 后，中断标识就一定被清空吗？
2. 处在死锁状态的线程是否可以被中断呢？

不可以

1. 进入临界区的线程能否被中断呢？如果不能有什么办法能响应中断吗？
2. 个人感觉interrupted这个方法名称不是特别好，如果你也觉得不好，让你设计这个地方，你有什么想法？