# [线程中断](https://javadoop.com/post/AbstractQueuedSynchronizer-2/)

中断不是类似 linux 里面的命令 kill -9 pid，不是说我们中断某个线程，这个线程就停止运行了。**中断代表线程状态，每个线程都关联了一个中断状态，是一个 true 或 false 的 boolean 值，初始值为 false**。

关于中断状态，我们需要重点关注 Thread 类中的以下几个方法：

// Thread 类中的实例方法，持有线程实例引用即可检测线程中断状态
public boolean isInterrupted() {}

// Thread 中的静态方法，检测调用这个方法的线程是否已经中断
// 注意：这个方法返回中断状态的同时，会将此线程的中断状态重置为 false
// 所以，如果我们连续调用两次这个方法的话，第二次的返回值肯定就是 false 了
public static boolean interrupted() {}

// Thread 类中的实例方法，用于设置一个线程的中断状态为 true
public void interrupt() {}

我们说中断一个线程，其实就是设置了线程的 interrupted status 为 true，至于说被中断的线程怎么处理这个状态，那是那个线程自己的事。如以下代码：

while (!Thread.interrupted()) {
 doWork();
 System.out.println("我做完一件事了，准备做下一件，如果没有其他线程中断我的话");
}

这种代码就是会响应中断的，它会在干活的时候先判断下中断状态，不过，除了 JDK 源码外，其他用中断的场景还是比较少的，毕竟 JDK 源码非常讲究。

当然，中断除了是线程状态外，还有其他含义，否则也不需要专门搞一个这个概念出来了。如果线程处于以下三种情况，那么当线程被中断的时候，能自动感知到：

* 来自 Object 类的 wait()、wait(long)、wait(long, int)，来自 Thread 类的join()、join(long)、join(long, int)、sleep(long)、sleep(long, int) > 这几个方法的相同之处是，方法上都有: throws InterruptedException > 如果线程阻塞在这些方法上（我们知道，这些方法会让当前线程阻塞），这个时候如果其他线程对这个线程进行了中断，那么这个线程会从这些方法中立即返回，抛出 InterruptedException 异常，同时重置中断状态为 false。
* 实现了 InterruptibleChannel 接口的类中的一些 I/O 阻塞操作，如 DatagramChannel 中的 connect 方法和 receive 方法等 >如果线程阻塞在这里，中断线程会导致这些方法抛出 ClosedByInterruptException 并重置中断状态。
* Selector 中的 select 方法 >一旦中断，方法立即返回

对于以上 3 种情况是最特殊的，因为他们能自动感知到中断（这里说自动，当然也是基于底层实现），并且在做出相应的操作后都会重置中断状态为 false。

那是不是只有以上 3 种方法能自动感知到中断呢？不是的，如果线程阻塞在 LockSupport.park(Object obj) 方法，也叫挂起，**这个时候的中断也会导致线程唤醒，但是唤醒后不会重置中断状态，所以唤醒后去检测中断状态将是 true**。

## InterruptedException 概述

它是一个特殊的异常，不是说 JVM 对其有特殊的处理，而是它的使用场景比较特殊。通常，我们可以看到，像 Object 中的 wait() 方法，ReentrantLock 中的 lockInterruptibly() 方法，Thread 中的 sleep() 方法等等，这些方法都带有 throws InterruptedException，我们通常称这些方法为阻塞方法（blocking method）。

阻塞方法一个很明显的特征是，它们需要花费比较长的时间（不是绝对的，只是说明时间不可控），还有它们的方法结束返回往往依赖于外部条件，如 wait 方法依赖于其他线程的 notify，lock 方法依赖于其他线程的 unlock 等等。

当我们看到方法上带有 throws InterruptedException 时，我们就要知道，这个方法应该是阻塞方法，我们如果希望它能早点返回的话，我们往往可以通过中断来实现。

除了几个特殊类（如 Object，Thread等）外，**感知中断并提前返回是通过轮询中断状态来实现的**。我们自己需要写可中断的方法的时候，就是通过在合适的时机（通常在循环的开始处）去判断线程的中断状态，然后做相应的操作（通常是方法直接返回或者抛出异常）。当然，我们也要看到，如果我们一次循环花的时间比较长的话，那么就需要比较长的时间才能感知到线程中断了。

## 处理中断

一旦中断发生，我们接收到了这个信息，然后怎么去处理中断呢？本小节将简单分析这个问题。

我们经常会这么写代码：

try {
 Thread.sleep(10000);
} catch (InterruptedException e) {
 // ignore
}
// go on

当 sleep 结束继续往下执行的时候，我们往往都不知道这块代码是真的 sleep 了 10 秒，还是只休眠了 1 秒就被中断了。这个代码的问题在于，我们将这个异常信息吞掉了。（对于 sleep 方法，我相信大部分情况下，我们都不在意是否是中断了，这里是举例）

AQS 的做法很值得我们借鉴，我们知道 ReentrantLock 有两种 lock 方法：

public void lock() {
 sync.lock();
}

public void lockInterruptibly() throws InterruptedException {
 sync.acquireInterruptibly(1);
}

前面我们提到过，lock() 方法不响应中断。如果 thread1 调用了 lock() 方法，过了很久还没抢到锁，这个时候 thread2 对其进行了中断，thread1 是不响应这个请求的，它会继续抢锁，当然它不会把“被中断”这个信息扔掉。我们可以看以下代码：

public final void acquire(int arg) {
 if (!tryAcquire(arg) &&
 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
 // 我们看到，这里也没做任何特殊处理，就是记录下来中断状态。
 // 这样，如果外层方法需要去检测的时候，至少我们没有把这个信息丢了
 selfInterrupt();// Thread.currentThread().interrupt();
}

而对于 lockInterruptibly() 方法，因为其方法上面有throws InterruptedException ，这个信号告诉我们，如果我们要取消线程抢锁，直接中断这个线程即可，它会立即返回，抛出 InterruptedException 异常。

在并发包中，有非常多的这种处理中断的例子，提供两个方法，分别为响应中断和不响应中断，对于不响应中断的方法，记录中断而不是丢失这个信息。如 Condition 中的两个方法就是这样的：

void await() throws InterruptedException;
void awaitUninterruptibly();