# 线程的基本操作

## 新建线程

新建线程很简单。只需要使用new关键字创建一个线程对象，然后调用它的start()启动线程即可。



那么线程start()之后，会干什么呢？线程有个run()方法，start()会创建一个新的线程并让这个线程执行run()方法。

这里需要注意，下面代码也能通过编译，也能正常执行。但是，却不能新建一个线程，而是在当前线程中调用run()方法，将run方法只是作为一个普通的方法调用。



所以，希望大家注意，调用start方法和直接调用run方法的区别。

**start方法是启动一个线程，run方法只会在当前线程中串行的执行run方法中的代码。**

默认情况下， 线程的run方法什么都没有，启动一个线程之后马上就结束了，所以如果你需要线程做点什么，需要把您的代码写到run方法中，所以必须重写run方法。



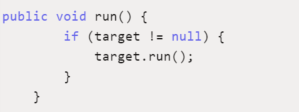
上面是使用匿名内部类实现的，重写了Thread的run方法，并且打印了一条信息。我们可以通过继承Thread类，然后重写run方法，来自定义一个线程。但考虑java是单继承的，从扩展性上来说，我们实现一个接口来自定义一个线程更好一些，java中刚好提供了Runnable接口来自定义一个线程。



Thread类有一个非常重要的构造方法：



我们在看一下Thread的run方法：



当我们启动线程的start方法之后，线程会执行run方法，run方法中会调用Thread构造方法传入的target的run方法。

**实现Runnable接口是比较常见的做法，也是推荐的做法。**

## 终止线程

一般来说线程执行完毕就会结束，无需手动关闭。但是如果我们想关闭一个正在运行的线程，有什么方法呢？可以看一下Thread类中提供了一个stop()方法，调用这个方法，就可以立即将一个线程终止，非常方便。



运行代码，输出：



代码中有个死循环，调用 stop方法之后，线程thread1的状态变为TERMINATED（结束状态），线程停止了。

我们使用idea或者eclipse的时候，会发现这个方法是一个废弃的方法，也就是说，在将来，jdk可能就会移除该方法。

stop方法为何会被废弃而不推荐使用？stop方法过于暴力，强制把正在执行的方法停止了。

大家是否遇到过这样的场景：电力系统需要维修，此时咱们正在写代码，维修人员直接将电源关闭了，代码还没保存的，是不是很崩溃，这种方式就像直接调用线程的stop方法类似。线程正在运行过程中，被强制结束了，可能会导致一些意想不到的后果。可以给大家发送一个通知，告诉大家保存一下手头的工作，将电脑关闭。

## 线程中断

在java中，线程中断是一种重要的线程协作机制，**从表面上理解，中断就是让目标线程停止执行的意思，实际上并非完全如此**。在上面中，我们已经详细讨论了stop方法停止线程的坏处，jdk中提供了更好的中断线程的方法。严格的说，线程中断并不会使线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出了！至于目标线程接收到通知之后如何处理，则完全由目标线程自己决定，这点很重要，如果中断后，线程立即无条件退出，我们又会遇到stop方法的老问题。

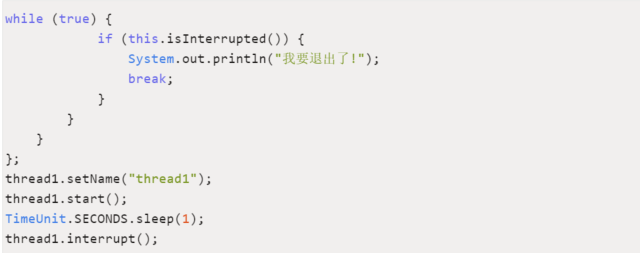
Thread提供了3个与线程中断有关的方法，这3个方法容易混淆，大家注意下：



**interrupt()方法是一个实例方法**，它通知目标线程中断，也就是设置中断标志位为**true**，中断标志位表示当前线程已经被中断了。

**isInterrupted()方法也是一个实例方法**，它判断当前线程是否被中断（通过检查中断标志位）。

**interrupted()方法是一个静态方法**，返回boolean类型，也是用来判断当前线程是否被中断，但是同时会清除当前线程的中断标志位的状态。



上面代码中有个死循环，interrupt()方法被调用之后，线程的中断标志将被置为true，循环体中通过检查线程的中断标志是否为true（ this.isInterrupted()）来判断线程是否需要退出了。

再看一种中断的方法：



代码中通过一个变量isStop来控制线程是否停止。

通过变量控制和线程自带的interrupt方法来中断线程有什么区别呢？

如果一个线程调用了sleep方法，一直处于休眠状态，通过变量控制，还可以中断线程么？大家可以思考一下。

此时只能使用线程提供的interrupt方法来中断线程了。



调用interrupt()方法之后，线程的sleep方法将会抛出 InterruptedException异常。



运行上面的代码，发现程序无法终止。为什么？

代码需要改为：



上面代码可以终止。

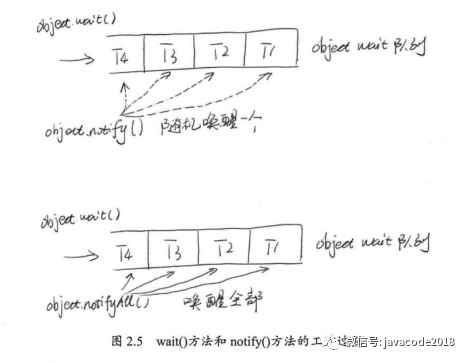
**注意：sleep方法由于中断而抛出异常之后，线程的中断标志会被清除（置为false），所以在异常中需要执行this.interrupt()方法，将中断标志位置为true。**

## 等待（wait）和通知（notify）

为了支持多线程之间的协作，JDK提供了两个非常重要的方法：等待wait()方法和通知notify()方法。这2个方法并不是在Thread类中的，而是在Object类中定义的。这意味着所有的对象都可以调用这两个方法。

当在一个对象实例上调用wait()方法后，当前线程就会在这个对象上等待。这是什么意思？比如在线程A中，调用了obj.wait()方法，那么线程A就会停止继续执行，转为等待状态。等待到什么时候结束呢？线程A会一直等到其他线程调用obj.notify()方法为止，这时，**obj对象成为了多个线程之间的有效通信手段**。

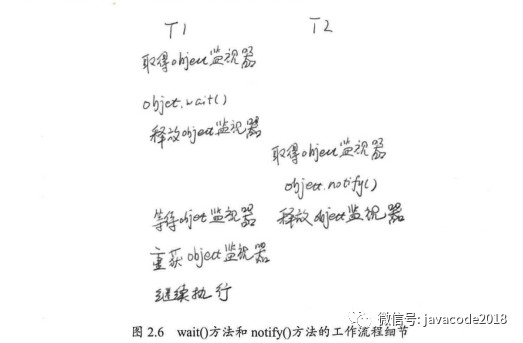
那么wait()方法和notify()方法是如何工作的呢？如图2.5展示了两者的工作过程。如果一个线程调用了object.wait()方法，那么它就会进出object对象的等待队列。这个队列中，可能会有多个线程，因为系统可能运行多个线程同时等待某一个对象。当object.notify()方法被调用时，它就会从这个队列中随机选择一个线程，并将其唤醒。这里希望大家注意一下，**这个选择是不公平的，并不是先等待线程就会优先被选择，这个选择完全是随机的**。



除notify()方法外，Object独享还有一个nofiyAll()方法，它和notify()方法的功能类似，不同的是，它会唤醒在这个等待队列中所有等待的线程，而不是随机选择一个。

这里强调一点，Object.wait()方法并不能随便调用。**它必须包含在对应的synchronize语句块中，无论是wait()方法或者notify()方法都需要首先获取目标独享的一个监视器（若调用对象的wait方法时当前线程未获得该对象的锁，则会抛出 IllegalMonitorStateException 异常）**。图2.6显示了wait()方法和nofiy()方法的工作流程细节。其中T1和T2表示两个线程。T1在正确执行wait()方法前，必须获得object对象的监视器。而wait()方法在执行后，会释放这个监视器。**这样做的目的是使其他等待在object对象上的线程不至于因为T1的休眠而全部无法正常执行。**

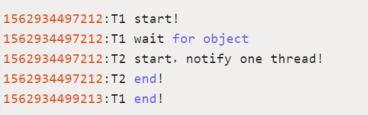
线程T2在notify()方法调用前，也必须获得object对象的监视器。所幸，此时T1已经释放了这个监视器，因此，T2可以顺利获得object对象的监视器。接着，T2执行了notify()方法尝试唤醒一个等待线程，这里假设唤醒了T1。T1在被唤醒后，要做的第一件事并不是执行后续代码，而是要尝试重新获得object对象的监视器，而这个监视器也正是T1在wait()方法执行前所持有的那个。如果暂时无法获得，则T1还必须等待这个监视器。当监视器顺利获得后，T1才可以在真正意义上继续执行。



给大家上个例子：



运行结果：

  
注意下打印结果，T2调用notify方法之后，T1并不能立即继续执行，而是要等待T2释放objec投递锁之后，T1重新成功获取锁后，才能继续执行。因此最后2行日志相差了2秒（因为T2调用notify方法后休眠了2秒）。

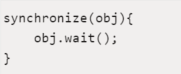
**注意：Object.wait()方法和Thread.sleeep()方法都可以让现场等待若干时间。除wait()方法可以被唤醒外，另外一个主要的区别就是wait()方法会释放目标对象的锁，而Thread.sleep()方法不会释放锁。**

再给大家讲解一下wait()，notify()，notifyAll()，加深一下理解：

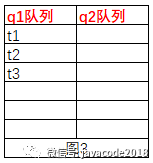
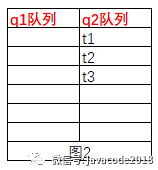
可以这么理解，obj对象上有2个队列，如图1，q1：等待队列，q2：准备获取锁的队列；两个队列都为空。



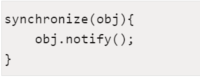
obj.wait()过程：



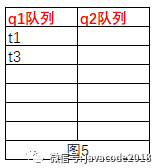
假如有3个线程，t1、t2、t3同时执行上面代码，t1、t2、t3会进入q2队列，如图2，进入q2的队列的这些线程才有资格去争抢obj的锁，假设t1争抢到了，那么t2、t3机型在q2中等待着获取锁，t1进入代码块执行wait()方法，此时t1会进入q1队列，然后系统会通知q2队列中的t2、t3去争抢obj的锁，抢到之后过程如t1的过程。最后t1、t2、t3都进入了q1队列，如图3。



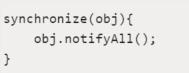
上面过程之后，又来了线程t4执行了notify()方法，如下：



t4会获取到obj的锁，然后执行notify()方法，系统会从q1队列中随机取一个线程，将其加入到q2队列，假如t2运气比较好，被随机到了，然后t2进入了q2队列，如图4，进入q2的队列的锁才有资格争抢obj的锁，t4线程执行完毕之后，会释放obj的锁，此时队列q2中的t2会获取到obj的锁，然后继续执行，执行完毕之后，q1中包含t1、t3，q2队列为空，如图5。



接着又来了个t5队列，执行了notifyAll()方法，如下：



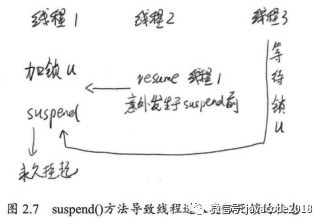
t5会获取到obj的锁，然后执行notifyAll()方法，系统会将队列q1中的线程都移到q2中，如图6，t5线程执行完毕之后，会释放obj的锁，此时队列q2中的t1、t3会争抢obj的锁，争抢到的继续执行，未增强到的带锁释放之后，系统会通知q2中的线程继续争抢索，然后继续执行，最后两个队列中都为空了。



## 挂起（suspend）和继续执行（resume）线程

Thread类中还有2个方法，即线程挂起(suspend)和继续执行(resume)，这2个操作是一对相反的操作，被挂起的线程，必须要等到resume()方法操作后，才能继续执行。**系统中已经标注着2个方法过时了，不推荐使用。**

**系统不推荐使用suspend()方法去挂起线程是因为suspend()方法导致线程暂停的同时，并不会释放任何锁资源**。此时，其他任何线程想要访问被它占用的锁时，都会被牵连，导致无法正常运行（如图2.7所示）。直到在对应的线程上进行了resume()方法操作，被挂起的线程才能继续，从而其他所有阻塞在相关锁上的线程也可以继续执行。但是，如果resume()方法操作意外地在suspend()方法前就被执行了，那么被挂起的线程可能很难有机会被继续执行了。并且，更严重的是：它所占用的锁不会被释放，因此可能会导致整个系统工作不正常。而且，对于被挂起的线程，从它线程的状态上看，居然还是**Runnable**状态，这也会影响我们队系统当前状态的判断。



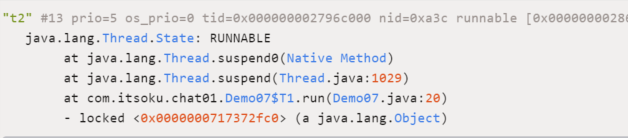
上个例子：



运行代码输出：



我们会发现程序不会结束，线程t2被挂起了，导致程序无法结束，使用jstack命令查看线程堆栈信息可以看到：



发现t2线程在suspend0处被挂起了，t2的状态竟然还是RUNNABLE状态，线程明明被挂起了，状态还是运行中容易导致我们队当前系统进行误判，代码中已经调用resume()方法了，但是**由于时间先后顺序的缘故，resume并没有生效**，这导致了t2永远的被挂起了，并且永远占用了object的锁，这对于系统来说可能是致命的。

## 等待线程结束（join）和谦让（yeild）

很多时候，一个线程的输入可能非常依赖于另外一个或者多个线程的输出，此时，这个线程就需要等待依赖的线程执行完毕，才能继续执行。jdk提供了join()操作来实现这个功能。如下所示，显示了2个join()方法：



第1个方法表示无限等待，它会一直只是当前线程。知道目标线程执行完毕。

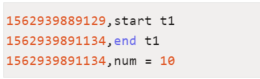
第2个方法有个参数，用于指定等待时间，如果超过了给定的时间目标线程还在执行，当前线程也会停止等待，而继续往下执行。

比如：线程T1需要等待T2、T3完成之后才能继续执行，那么在T1线程中需要分别调用T2和T3的join()方法。

上个示例：



执行结果：



num的结果为10，1、3行的时间戳相差2秒左右，说明主线程等待t1完成之后才继续执行的。

另外一个方法是Thread.yield()，他的定义如下：



yield是谦让的意思，这是一个静态方法，一旦执行，它会让当前线程出让CPU，但需要注意的是，出让CPU并不是说不让当前线程执行了，当前线程在出让CPU后，还会进行CPU资源的争夺，但是能否再抢到CPU的执行权就不一定了。因此，对Thread.yield()方法的调用好像就是在说：我已经完成了一些主要的工作，我可以休息一下了，可以让CPU给其他线程一些工作机会了。

**如果觉得一个线程不太重要，或者优先级比较低，而又担心此线程会过多的占用CPU资源，那么可以在适当的时候调用一下Thread.yield()方法，给与其他线程更多的机会。**

## 总结

1.创建线程的2中方式：继承Thread类；实现Runnable接口

2.启动线程：调用线程的start()方法

3.终止线程：**调用线程的stop()方法，方法已过时，建议不要使用**

4.线程中断相关的方法：调用线程实例interrupt()方法将中断标志置为true；使用线程实例方法isInterrupted()获取中断标志；调用Thread的静态方法interrupted()获取线程是否被中断，此方法调用之后会清除中断标志（将中断标志置为false了）

5.wait、notify、notifyAll方法，这块比较难理解，可以回过头去再理理

6.线程挂起使用线程实例方法suspend()，恢复线程使用线程实例方法resume()，这2个方法都过时了，不建议使用

7.等待线程结束：调用线程实例方法join()

8.出让cpu资源：调用线程静态方法yeild()