为什么 Thread.stop 被废弃了？

因为其天生是不安全的。停止一个线程会导致其解锁其上被锁定的所有监视器（监视器以在栈顶产生 ThreadDeath 异常的方式被解锁）。如果之前被这些监视器保护的任何对象处于不一致状态，其它线程看到的这些对象就会处于不一致状态。 这种对象被称为 受损的 （damaged）。当线程在受损的对象上进行操作时，会导致任意行为。 这种行为可能微妙且难以检测， 也可能会比较明显。不像其他未受检的 （unchecked ）异常， ThreadDeath 悄无声息的杀死及其他线程。因此，用户得不到程序可能会崩溃的警告。崩溃会在真正破坏发生后的任意时刻显现， 甚至在数小时或数天之后。

难道我不能仅捕获 ThreadDeath 异常来修正受损对象吗？

理论上，也许可以，但书写正确的多线程代码的任务将 极其 复杂。由于两方面的原因，这一任务的将几乎不可能完成：

-
  - 线程可以在几乎任何地方抛出 ThreadDeath 异常。由于这一点， 所有的同步方法和 （代码）块将必须被考虑得事无巨细。
  - 线程在清理第一个 ThreadDeath 异常的时候（在 catch 或 finally 语句中），可能会抛出第二个。 清理工作将不得不重复直到到其成功。 保障这一点的代码将会很复杂。

Thread.stop(Throwable) 会有什么问题？

除了上述所有问题外，此方法还可能产生其目标线程不准备处理的异常（包括若非为实现此方法， 线程不太可能抛出的受检异常） 。例如，下面的方法行为上等同于 Java 的 throw 操作，但是绕开了编译器的努力，即保证要调用的方法已经声明了其所有可能抛出的异常：

```
static void sneakyThrow(Throwable t) {
    Thread.currentThread().stop(t);
}
```

我应该用什么来取代 Thread.stop ？

大多数 stop 的使用， 应当被替换为简单修改某些变量来指示其目标线程将停止运行的代码。目标线程应当有规律的检查这些变量。 并且， 如果这些变量指示其将停止运行， 目标线程应当以某种有序的方式从它的 run 方法返回（这正是 Java Tutorial 一贯建议的方式）。 为了确保停止请求的及时传达，变量必须是 volatile 的（或者变量的访问被同步）。

例如，假设你的 applet 包含了 start 、 stop 和 run 方法：

```
private Thread blinker;

public void start() {
    blinker = new Thread(this);
    blinker.start();
```

```
}

public void stop() {
    blinker.stop();  // UNSAFE!
}

public void run() {
    Thread thisThread = Thread.currentThread();
    while (true) {
        try {
            thisThread.sleep(interval);
        } catch (InterruptedException e){
        }
        repaint();
      }
  }
```

为了避免使用 Thread.stop ，你可以把 applet 的 stop 和 run 方法替换成：

```
  private volatile Thread blinker;

  public void stop() {
      blinker = null;
  }

  public void run() {
      Thread thisThread = Thread.currentThread();
      while (blinker == thisThread) {
          try {
              thisThread.sleep(interval);
          } catch (InterruptedException e){
          }
          repaint();
      }
  }
```

我如何才能停止一个长时间等待的线程（例如，用于输入）？

这正是 Thread.interrupt 方法要做的。与上述相同的“基于状态”的信号传递机制可以被应用，但是状态传递（blinker = null，在上一个例子中）后面可以跟一个 Thread.interrupt 调用，用来中断和等待：

```
  public void stop() {
      Thread moribund = waiter;
      waiter = null;
```

```
    moribund.interrupt();
  }
```

为了让这种技术起作用，　关键 在于，对于任何捕获了中断异常但不准备立即处理的方法，应重新断言异常 。我们说 重新断言 （reasserts ）而不是 重新抛出 （rethorws ），因为要重新抛出异常并非总是可行。 如果捕获 InterruptedException 的方法没有被声明为抛出这种 （受检）异常，那么它应该采用下述咒语来 “重新中断自己 ”：

```
  Thread.currentThread().interrupt();
```

如果线程没有响应 Thread.interrupt 会怎么样？

在某些情况下， 你可以采取特定于应用的技巧。 例如，如果线程在一个已知的 socket 上等待，你可以关闭这个 socket ，来促使线程立即返回。不幸的是，确实没有放之四海皆可工作的通用技术。 应当注意，对于等待线程不响应 Thread.interrupt 的所有情况，它也不会响应 Thread.stop 。这些案例包括有意的拒绝服务攻击， 以及 thread.stop 和 thread.interrupt 不能正常工作的 I/O 操作。

为什么 Thread.suspend 和 Thread.resume 被废弃了？

Thread.suspend 天生容易引起死锁 。如果 目标线程挂起时在保护系统关键资源的监视器上持有锁，那么其他线程在目标线程恢复之前都无法访问这个资源 。如果要恢复目标线程的线程在调用 resume 之前试图锁定这个监视器，死锁就发生了 。这种死锁一般自身表现为 “冻结（frozen ） ”进程。

我应该用什么来取代 Thread.suspend 和 Thread.resume ？

与 Thread.stop, 类似，谨慎的方式，是 让 “目标线程 ”轮询一个指示线程期望状态（活动或挂起）的变量。当期望状态是挂起时，线程用 Object.wait 来等待；当恢复时，用 Object.notify 来通知目标线程 。

（类似生产者消费者）

例如，假设你的 applet 包含下面的 mousePressed 事件句柄，用来切换一 个被称为 blinker 的线程的状态。

```
  private boolean threadSuspended;

Public void mousePressed(MouseEvent e) {
    e.consume();

    if (threadSuspended)
        blinker.resume();
    else
        blinker.suspend();  // DEADLOCK-PRONE!
```

```
        threadSuspended = !threadSuspended;
    }
```

要避免使用 Thread.suspend 和 Thread.resume ，你可以把上述事件句柄替换为：

```
    public synchronized void mousePressed(MouseEvent e) {
        e.consume();

        threadSuspended = !threadSuspended;

        if (!threadSuspended)
            notify();
    }
```

并把下述代码增加到 “运行循环 ”中：

```
            synchronized(this) {
                while (threadSuspended)
                    wait();
            }
```

wait 方法抛出 InterruptedException ，因此他必须在一个 “ try ... catch ”语句中。使用 sleep 方法时，也可以将其放入同样的语句中。检查应该在 sleep 方法后（而不是先于），以便当线程恢复的时候窗口被立即重绘 ?。修改后的 run 方法如下：

```
    public void run() {
        while (true) {
            try {
                Thread.currentThread().sleep(interval);

                synchronized(this) {
                    while (threadSuspended)
                        wait();
                }
            } catch (InterruptedException e){
            }
            repaint();
        }
    }
```

注意， mousePressed 方法中的 Notify 和 run 方法中的 wait 都是在 synchronized 语句块中的。这是语言的要求，也确保了 wait 和 notify 被正确串行化执行。从实际效果来看，这消除了竞争条件，避免了不确定的 “挂起 ”线程丢失 notify 消息而仍保持挂起。

虽然随着平台的成熟 Java 的同步开销正在减少， 但其永远都不会是免费的。 有一个简单的技巧，可以用于移除我们加入到 “运行循环 ”每次迭代中的同步。加入的同步块被替换为稍微有点复杂的代码片段，只有当线程真正被挂起的时候后才会进入同步块：

```
            if (threadSuspended) {
                synchronized(this) {
                    while (threadSuspended)
```

```
                wait();
            }
        }
```

由于缺少显式同步，threadSuspended 必须被指定为 volatile 来保证挂起请求被迅速传递。

修改后的 run 方法如下：

```
    private boolean volatile threadSuspended;

    public void run() {
        while (true) {
            try {
                Thread.currentThread().sleep(interval);

                if (threadSuspended) {
                    synchronized(this) {
                        while (threadSuspended)
                            wait();
                    }
                }
            } catch (InterruptedException e){
            }
            repaint();
        }
    }
```

我可以结合两种技术来产生可以安全 “停止”或“挂起”的线程吗？

是的，这这相当直观。有一个不易察觉的地方，那就是当目标线程可能已经挂起的时候，另外一个线程试图停止它。如果 stop 方法只是将状态变量（ blinker ）设置为 null ，目标线程将仍然处于挂起状态（等待监视器），而不是它所应该的优雅退出。如果 applet 被重启，多个线程可能会同时结束在 monitor 上等待，从而导致奇怪的行为。

为了矫正这种状况，stop 方法必须保证挂起的目标线程迅速恢复。一旦目标线程恢复，它必须立即认识到它已经被停止了，并且优雅的退出。这里是修改过的 run 和 stop 方法：

```
    public void run() {
        Thread thisThread = Thread.currentThread();
        while (blinker == thisThread) {
            try {
                thisThread.sleep(interval);

                synchronized(this) {
                    while (threadSuspended && blinker==thisThread)
                        wait();
                }
```

```
        } catch (InterruptedException e){
        }
        repaint();
    }
}

public synchronized void stop() {
    blinker = null;
    notify();
  }
```

如果 stop 方法调用 Thread.interrupt ，如前所述，它也不需要调用 notify 了，但是他仍然需要被同步。这确保了目标线程不会因竞争条件而丢失中断。

关于 Thread.destroy 如何呢？

Thread.destroy 从未被实现 。如果它被实现了， 它将和 Thread.suspend 一样易于死锁（事实上，它大致上等同于没有后续 Thread.resume 的 Thread.suspend ）。我们现在既没有实现，也没有废除它（防止将来它被实现）。虽然它确实易于发生死锁，有人争论过，在有些情况下程序可能愿意冒死锁的险而不是直接退出。

为什么 Runtime.runFinalizersOnExit 被废弃了？ ?

由于其天生不安全。它 可能导致终结器（ finallizers ）被在活动对象上被调用 ，而其他线程正在并发操作这些对象，导致奇怪的行为或死锁。然而， 如果正在被终结对象的类被编码为 “防御 ”这种调用，这个问题可以避免 。大多数程序员都 不会 阻止它。它们假设当终结器被调用的时候对象已经死亡。

而且， 这个调用不是 “线程安全 ”的，因为它设置了一个 VM全局标志。这迫使每个带有终结器的类防御活动对象的终结！