线程的生命周期及5种状态

# 线程的生命周期

在Java中，线程的生命周期一共有5个状态：新建、运行、等待、阻塞、停止。

1. 线程状态：Thread.State



Java中的Thread.State是一个**静态的枚举类（enum类型）**。

枚举常量包括：6个。

1. **NEW**： 新建状态，创建了一个线程对象，但是没有调用start方法；
2. **RUNNABLE：运行状态，**拥有CPU的使用权。
3. **WAITING**：调用了wait（）方法的等待状态，需要其他线程利用notify唤醒。
4. **TIMED\_WAITING**：使用sleep（ms）方法后线程的状态。
5. **BLOCKED**：阻塞状态；
6. **TERMINATED**：（终止）死亡状态。



1. 新建的线程在它的一个完整的生命周期中通常要经历4种状态：
2. **新建状态**：当一个Thread类或其子类的对象被声明并创建时，新的线程对象就处于了新建状态。新建状态的线程已经有了**相应的内存空间和其他资源**。
3. **运行状态**：处于新建状态的线程就已经具备了运行的条件，一旦轮到它来使用CPU资源时，即JVM将CPU使用权切换给该线程时，此线程就可以脱离创建它的主线程独立开始自己的生命周期了。

注： 线程创建后，仅仅只是占用内存资源，在JVM管理的线程中还没有该线程，必须通过调用此线程的**start（）方法**告知JVM，这样JVM就知道又有一个线程排队等待切换了。

**线程没有执行完run方法前，不可以再次调用start方法，否则发生java.lang.IllegalThreadStateException异常。**

1. 中断状态：

有4中原因中断：

* 1. JVM把CPU资源从当前线程切换给其他线程，使本线程让出CPU的使用权而处于中断状态；
  2. 线程使用CPU资源期间，执行了**sleep（int ms）方法**，使当前线程处于休眠状态。经过指定的ms毫秒后，该线程就重新进到线程队列中排队等待CPU资源，以便从中断处继续执行。
  3. 线程使用CPU资源期间，执行了**wait（）方法**，使当前线程处于等待状态。等待状态的线程，不会主动进入到线程队列中排队等待CPU资源，必须由其他线程调用**notify（）方法**去通知它，使它重新进到线程队列中排队等待CPU资源，以便从中断处继续执行。
  4. 线程使用CPU资源期间，执行某个操作进入阻塞状态，如执行读写操作时引起阻塞。进入阻塞状态时的线程不能进入排队队列，只有引起阻塞的原因消除时，线程才重新进到线程对列中排队等待CPU资源，以便从原来中断处继续执行。

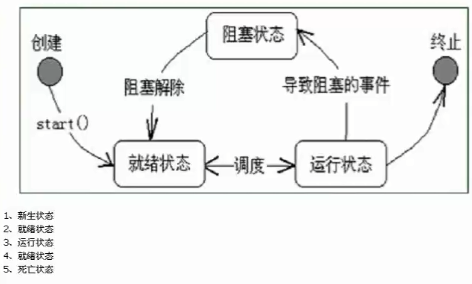
1. 死亡状态：不再具有继续执行的能力。**线程的死亡就是线程释放了实体，释放了线程对象的内存空间。**

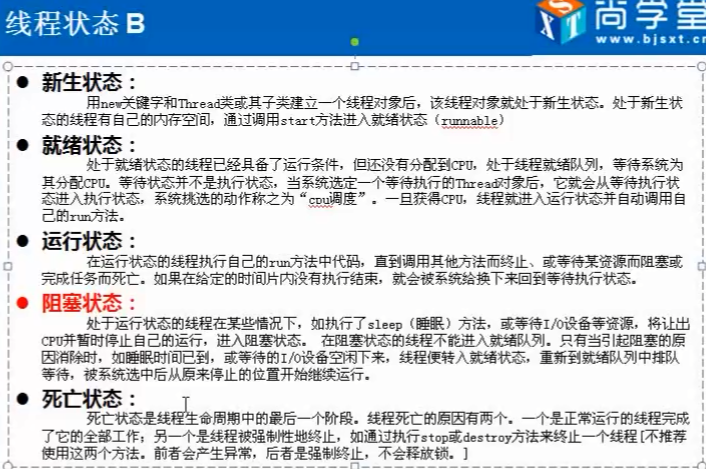
死亡的原因有两个:①**执行完了run方法中的全部语句，结束了run方法，即完成了线程的全部任务**。

**一般建议使用这种方法：通过设置个标志位flag来控制线程的结束。**

②**线程被提前强制性终止**，即强制结束run方法。

1. 线程的生命周期





1. **多线程程序**在不同的计算机上执行结果可能不同，即使是同一台计算机，反复执行的结果也不尽相同，执行结果依赖于当前的CPU资源的使用情况。为了使结果尽量不依赖于当前CPU的使用情况，可以人为的使用sleep（int ms）方法让出CPU的使用权而进入中断状态。
2. 注意：sleep（int ms） 首先释放CPU的使用权，然后等待时间结束，然后排队；wait方法也是首先释放CPU的使用权，但是需要等待其他线程通过notify方法叫醒，然后再进入排队状态。
3. **Sleep（） 方法和wait（） 方法 有什么区别？**

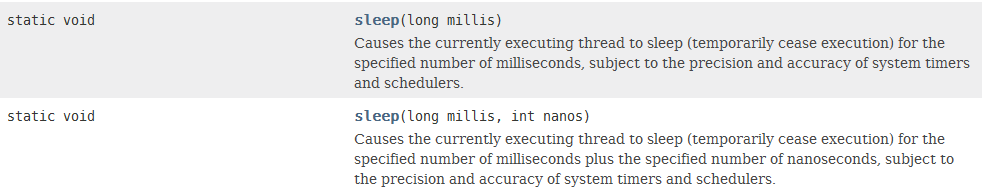
**答： （1）sleep方法是Thread的方法，而wait是Object就有的方法；**

**（2）wait可以指定时间，也可以不指定时间，而sleep必须指定时间；**

**（3）在同步中，对于CPU的执行权和锁的处理不同：wait释放CPU执行权，释放锁，sleep释放CPU执行权，不释放锁（sleep不需要别人叫，wait需要别人叫，所以要释放锁）。**

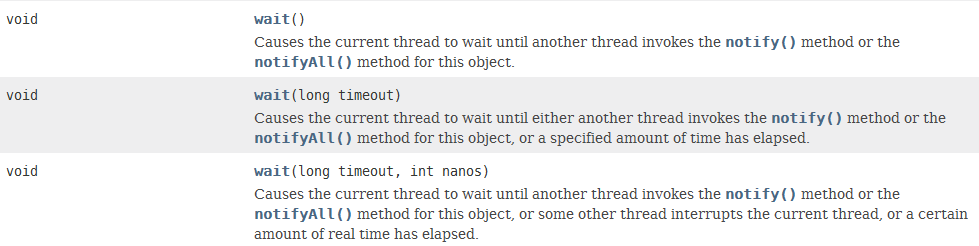
在同步代码块或者同步方法中，可能有多个线程都由于wait语句，而释放了执行权（释放了执行资格和锁），如果有其他线程中利用**notifyAll方法**同时唤醒了这多个线程，此时虽然同时都具备了执行资格，但是由于只有一个线程能够获得锁，因此也只有同时获得了执行资格和锁的线程能够执行同步代码其他线程仍会处于**等待状态**。锁不仅仅是一个门槛，而且是同步的整个过程都必须一直持有锁才可以。在一个同步中，只能有**一个线程**执行，但是 可以**有多个被唤醒着的线程**，只有获得此同步锁的线程才会执行，被唤醒的其他线程因不具有锁而不会执行。

1. sleep方法必须有**时间参数**：



1. wait方法继承于**Object类**：

有三个：**其中wait（）方法等价于wait（0）方法**。



1. 终止线程的方法：
2. 自然终止：run方法运行结束。
3. 外部干涉：

通过设置一些标志，然后对外提供方法终止线程。

1. 线程类中定义控制线程结束的标识；
2. 在run方法中使用该标识；
3. 对外提供改变此标识的方法如stop方法；
4. 外部根据条件调用此方法即可结束线程。

示例：

public class ThreadStop implements Runnable {

private boolean flag = true;

@Override

public void run() {

while(this.flag) {

System.out.println("victory");

}

}

public void stop() {

this.flag = false;

}

}

主方法main:

public static void main(String[] args) {

ThreadStop ts = new ThreadStop();

Thread thread = new Thread(ts,"stop1");

thread.start();

for (int i = 0; i < 400; i++) {

if(300==i) {

ts.stop();

}

System.out.println("main&&&&&&&&&&&&&&&&&&&");

}

}

注意：Thread中有stop方法，但是已经destroy了，是不安全的。

如果通过继承Thread创建线程，建议覆盖其stop方法；

如果是通过实现Runnable接口创建任务类的，则在任务类中提供stop方法即可，注意终止线程时，要利用任务对象调用stop方法。

1. 阻塞状态：
2. **join（）**：阻塞别人，自己执行，别的线程进入了**阻塞状态**。
3. **yield（）方法**：是一个静态方法，**暂停自己线程**。
4. sleep（）方法：释放cpu但是不释放锁。

可以利用sleep方法来：

1. 构造倒计时；见下面。
2. 模拟网络延时。Thread.sleep(long);
3. wait()方法。
4. 利用sleep方法构造倒计时。

private static void DaoJiShi10Seconds() throws InterruptedException {

Date date = new Date(System.currentTimeMillis()+1000\*10);

long end = date.getTime();

int i = 9;

while(true) {

Thread.sleep(1000);

end -= 1000;

System.out.println("倒計時 ："+i--+"秒 ：\t"+new SimpleDateFormat("mm:ss").format(new java.util.Date(end)));

if (System.currentTimeMillis()-10\*1000 > end ) {

break;

} }}

结果:

