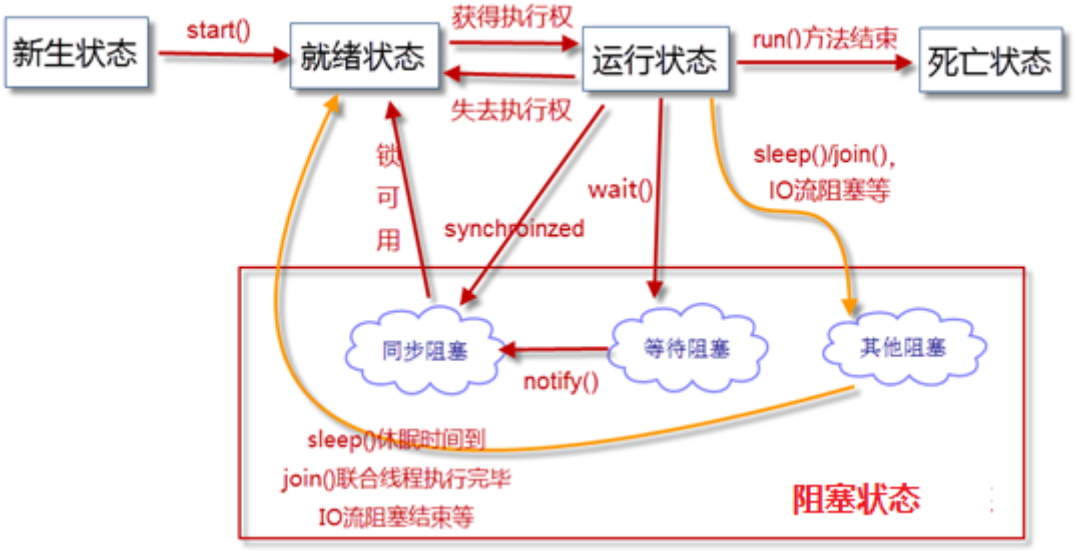


- 新生状态(New)

用new关键字建立一个线程对象后，该线程对象就处于新生状态。处于新生状态的线程有自己的**内存空间**，通过调用start**方法**进入就绪状态。

- 就绪状态(Runnable)

处于就绪状态的线程已经具备了运行条件，但是还**没有被分配到**CPU，处于“线程就绪队列”，等待系统为其分配CPU。就绪状态并不是执行状态，当系统选定一个等待执行的Thread对象后，它就会进入执行状态。一旦获得CPU时间片，线程就进入运行状态并**自动调用自己的run方法**。有4中原因会导致线程进入就绪状态：

1. 新建线程：调用start()方法，进入就绪状态；
2. 阻塞线程：**阻塞解除**，进入就绪状态；
3. 线程让步：Thread.yield() **方法**，暂停正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况

下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。

4. 运行线程：JVM将CPU资源从本线程切换到其他线程，失去执行权。

- **运行状态(Running)**

在运行状态的线程执行自己run方法中的代码，直到调用其他方法而终止或等待某资源而阻塞或完成任务而死亡。如果**在给定的时间片内没有执行结束，就会被系统给换下来回到就绪状态**。也可能由于某些“导致阻塞的事件”而进入阻塞状态。jvm只能看到runnable状态，不能看到running状态。

- **阻塞状态(Blocked)**

阻塞指的是暂停一个线程的执行以等待某个条件发生(如某资源就绪)。有4种原因会导致阻塞：

1. 执行sleep(int millsecond)方法，使当前线程休眠，进入阻塞状态。当指定的时间到了后，线程进入就绪状态。

```
   try  {
                Thread.sleep(2000);//调用线程的
sleep()方法；
            } catch  (InterruptedException e) {
                e.printStackTrace();
            }
```

2. 线程运行时，某个操作进入阻塞状态，比如**执行IO流操作(read()/write()方法本身就是阻塞的方法)**。只有当引起该操作阻塞的原因消失后，线程进入就绪状态。

Timed waiting：在一定时间之后会被系统自动唤醒。

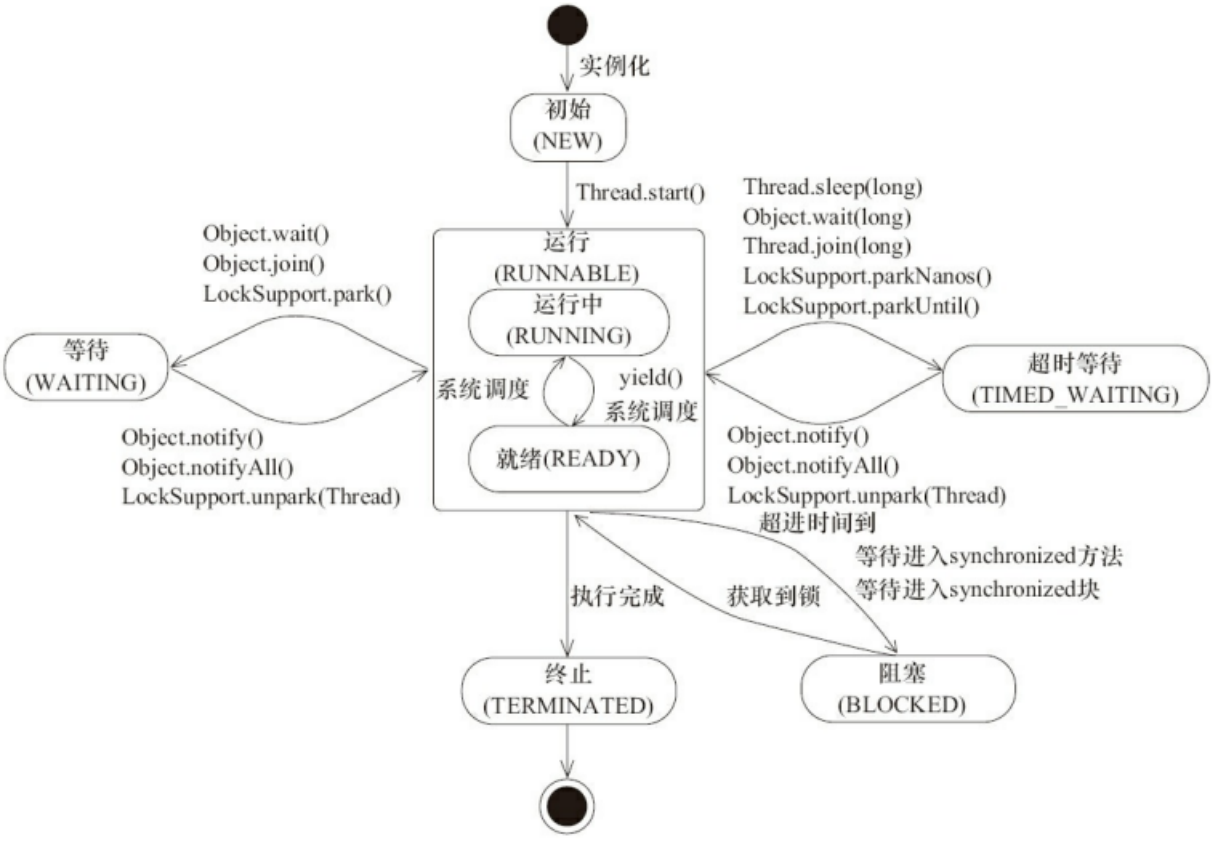
调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。

调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取一个排它锁。而等待是主动的，通过调用 Thread.sleep() 和 Object.wait() 等方法进入。

进入方法	退出方法
Thread.sleep() 方法	时间结束
设置了 Timeout 参数的 Object.wait(long) 方法	时间结束 / Object.notify() / Object.notifyAll()
设置了 Timeout 参数的 Thread.join(long) 方法	时间结束 / 被调用的线程执行完毕
LockSupport.parkNanos() 方法	LockSupport.unpark(Thread)
LockSupport.parkUntil() 方法	LockSupport.unpark(Thread)



3. 执行wait()方法，使当前线程进入阻塞状态。当使用nofity()方法唤醒这个线程后，它进入就绪状态。Obj.wait()，与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作。

Thread类的方法：sleep(),yield()等

Object的方法：wait()和notify()等

```
            try  {
                //wait后，线程会将持有的锁释放，进入阻塞状态；
                //这样其它需要锁的线程就可以获得锁；
                 this.wait();
                 //这里的含义是执行此方法的线程暂停，进入阻塞状态，
                 //等消费者消费了馒头后再生产。
             } catch  (InterruptedException e) {
                 e.printStackTrace();
             }
         }
         // 唤醒在当前对象等待池中等待的第一个线程。notify()调用后，并不是马上就释放对象锁的，而是在相应的synchronized(){}语句块执行结束，自动释放锁后
         //notifyAll叫醒所有在当前对象等待池中等待的所有线程。
         this.notify();
         // 如果不唤醒的话。以后这两个线程都会进入等待线程，没有人唤醒。
```

4. join()线程联合：当某个线程等待另一个线程执行结束后，才能继续执行时，使用join()方法。线程A在运行期间调用线程B的join()方法，线程A就必须等待线程B执行完毕。

```
        try  {
            son.join();       //儿子先执行完
        } catch  (InterruptedException e) {
            e.printStackTrace();
            System.out.println("爸爸出门去找儿子跑哪去了");
            // 结束JVM。如果是0则表示正常结束；如果是非0则表示非正常结束
            System.exit(1);
        }
```

Wating：等待其它线程显式地唤醒，否则不会被分配 CPU 时间片。

进入方法	退出方法
没有设置 Timeout 参数的 Object.wait() 方法	Object.notify() / Object.notifyAll()
没有设置 Timeout 参数的 Thread.join() 方法	被调用的线程执行完毕
LockSupport.park() 方法	LockSupport.unpark(Thread)

- 终止状态(Terminated)

线程生命周期中的最后一个阶段，当一个线程进入终止状态以后，就不能再回到其它状态了。两种情况：

- 正常运行的线程完成了它run()方法内的全部工作;

- 是线程被强制止，如通过执行stop()或destroy()方法来终止一个线程(注：stop()/destroy()方法已经被JDK废弃，不推荐使用)。通常的做法是提供一个**boolean型的终止变量**，当这个变量置为false，则终止线程的运行。

终止线程的典型方法(重要)

```
public class TestThreadCiycle implements Runnable {
    String name;
    boolean live = true;// 标记变量，表示线程是否可中止；
    public TestThreadCiycle(String name) {
        super();
        this.name = name;
    }
    public void run() {
        int i = 0;
        //当live的值是true时，继续线程体；false则结束循环，继而终止线程体；
        while (live) {
            System.out.println(name + (i++));
        }
    }
    public void terminate() {
        live = false;
    }

    public static void main(String[] args) {
        TestThreadCiycle ttc = new TestThreadCiycle("线程A:");
        Thread t1 = new Thread(ttc);// 新生状态
        t1.start();// 就绪状态
        for (int i = 0; i < 100; i++) {
            System.out.println("主线程" + i);
        }
        ttc.terminate();
        System.out.println("ttc stop!");
    }
}
```