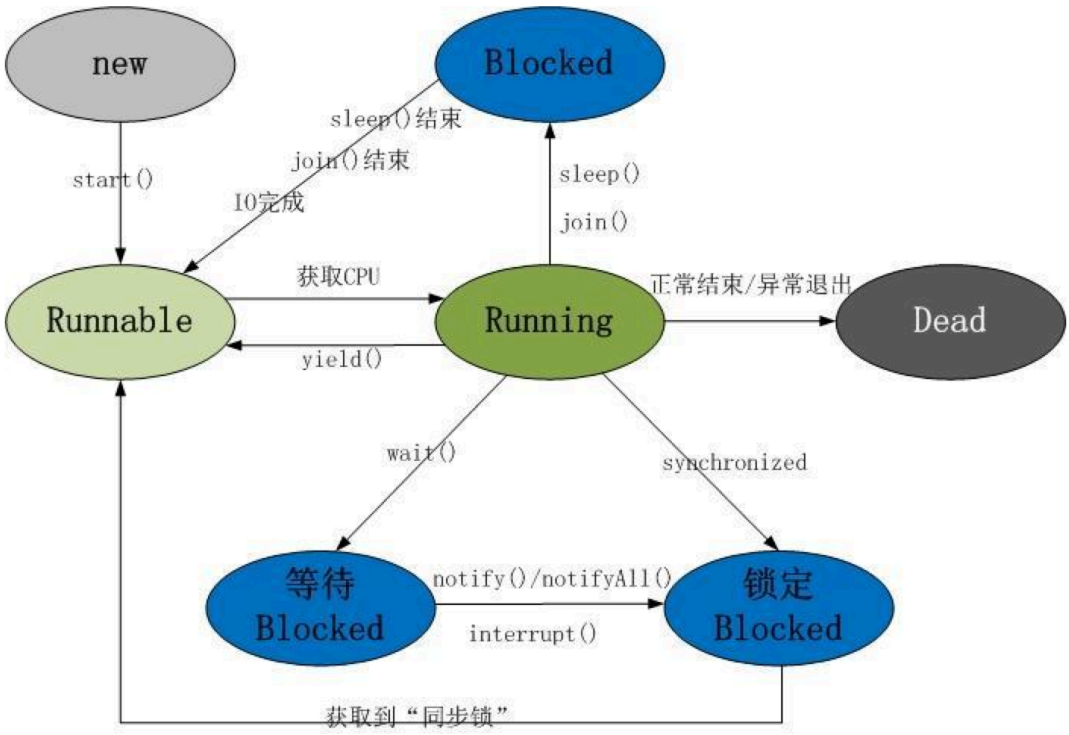
- 线程的生命周期一般会经历：新建(new)、就绪(Runnable)、运行(Running)、阻塞(Blocked)和死亡。
- 在阻塞状态的线程只能进入就绪状态，无法直接进入运行状态。而就绪和运行状态之间的转换通常不受程序控制，而是由系统线程调度所决定。当处于就绪状态的线程获得处理器资源时，该线程进入运行状态；当处于运行状态的线程失去处理器资源时，该线程进入就绪状态。
- 由一个方法例外，调用yield()方法可以让运行状态的线程转入就绪状态。



线程状态转换关系

1. 新建（New）状态

当程序用**new**关键字创建一个线程后，该线程就处于新建状态，此时线程情况如下：

1. 此时JVM为其分配内存，并初始化其成员变量的值;
2. 此时线程对象没有表现出任何线程的动态特征，程序也不会执行线程的run方法。

2. 就绪（Runnable）状态

当线程对象调用了`start`方法后，该线程就处于就绪状态，此时线程的情况如下：

1. 此时JVM会为其 **创建方法调用栈和程序计数器**;
2. 该状态的线程一直存在于 **线程就绪队列** 中，线程并没有开始运行;
3. 此时的线程正在等待系统为其分配，并不是说执行了start方法，线程就可以立即执行。

调用`start`方法和**`run`**方法的区别：

1. **调用start()方法来启动线程，系统会把该run()方法当成线程执行体来处理**,但如果直接执行run()方法，系统会把线程对象当成一个普通的对象，而run()方法就是一个普通的方法，而不是线程执行体;
2. **调用了run()方法后，该线程已经不在处于新建状态**，不要再次调用start方法，否则会出现IllegalThreadStateExccption异常。

如何让子线程调用`start()`方法之后立即执行而非"等待执行"：

程序可以使用Thread.sleep(1) 来让当前运行的线程（主线程）睡眠1毫秒，1毫秒就够了，因为在这1毫秒内CPU不会空闲，它会去执行另一个处于就绪状态的线程，这样就可以让子线程立即开始执行；

3. 运行（Running）状态

1. 当CPU开始调度处于"就绪状态"的线程时，此时线程获取了CPU时间片才得以真正开始执行**`run`**()方法，则该线程处于"运行状态"。

2. 处于运行状态的线程不可能一直处于"运行状态"，线程在运行过程中需要被中断，目的是使其他线程获得执行的机会，这需要取决于底层平台所采用的策略。

1. **抢占式策略**：系统为每个可执行线程分配一个时间片处理任务，当时间片执行完后，系统就会剥夺该线程所占用的资源，让其他线程获得执行的机会。此时线程就会从**运行状态**变为**就绪状态**，重新等待系统分配资源。
2. **协作式策略**: 只有当一个线程调用了它的yield()方法后，才会放弃所占用的资源，也就是说必须由本线程主动放弃所占资源，线程才会从**运行状态**变为**就绪状态**。

4. 阻塞（Blocked）状态

处于运行状态的线程在某些状态下，会让出CPU并暂时停止自己的运行，进入阻塞状态。场景如下：

1. 线程执行sleep()方法，主动放弃CPU资源，进入中断状态（**不会释放持有的对象锁**），时间到后等待系统分配CPU继续执行；
2. 程序调用了线程的suspend方法将线程挂起；
3. 线程调用wait，等待notify/notifyAll唤醒时(**会释放持有的对象锁**)；

阻塞状态分类：

1. **等待阻塞**: 运行状态中的 **线程执行wait()方法**, 使本线程进入等待阻塞状态；
2. **同步阻塞**: 线程在 **获取synchronized同步锁失败**(因为锁被其他线程占用)，它会进入同步阻塞状态；
3. **其他阻塞**: 通过调用线程的 **sleep()或join()或发出I/O请求** 时, 线程会进入到阻塞状态，当sleep()状态超时、join()等待线程终止或者超时、I/O处理完毕时，线程重新转入就绪状态；

4.1 等待（waiting）状态

线程处于 无限制等待状态，等待一个特殊的事件来重新唤醒。

1. 通过wait()方法进行等待的线程等待一个notify()或者notifyAll()方法.
2. 通过join()方法进行等待的线程等待目标线程运行结束而唤醒.

以上两种一旦通过相关事件唤醒线程，线程就进入了"就绪状态" 继续运行。

4.2 时限等待(time_waiting)状态

线程进入了一个 时限等待状态，如

sleep(3000)，等待3秒后线程重新进行 就绪（RUNNABLE）状态 继续运行.

5. 死亡（dead）状态

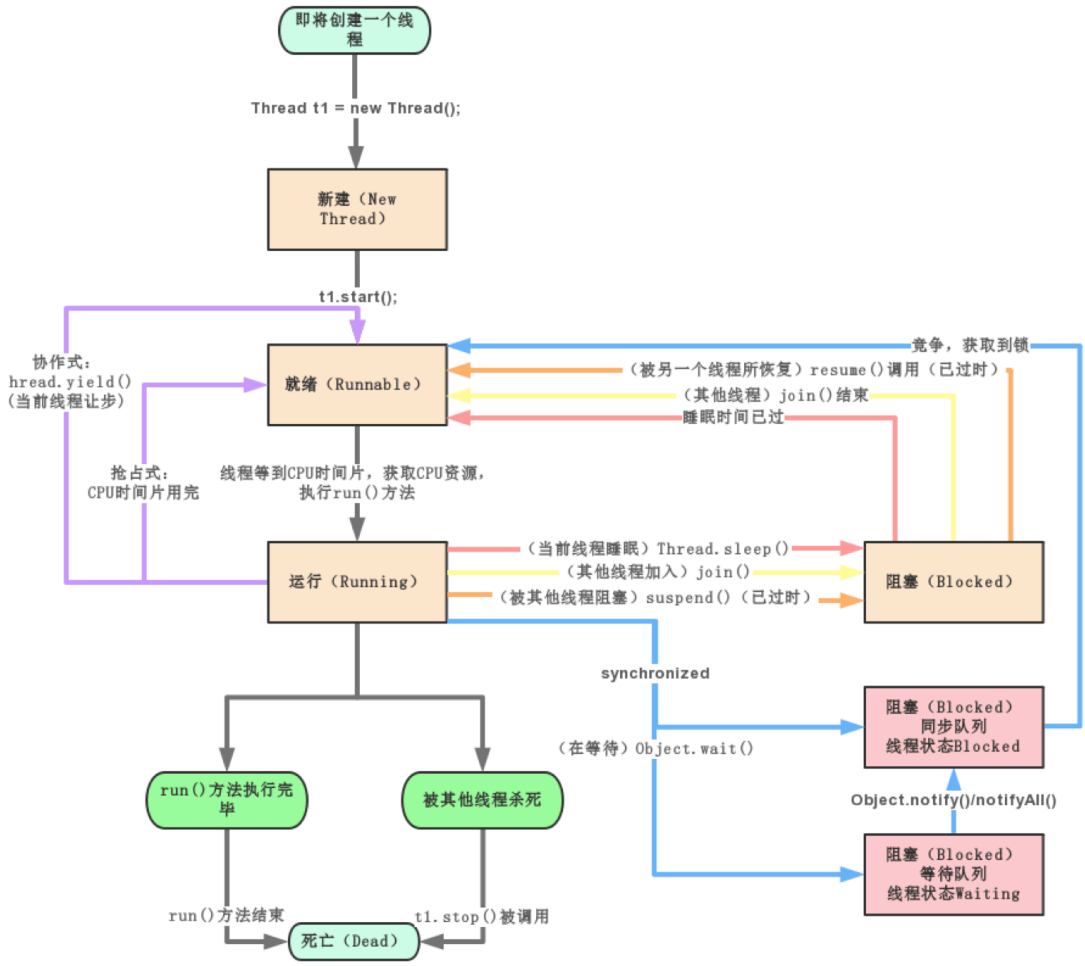
线程会以如下3种方式结束，结束后就处于 死亡状态：

1. run()或call()方法执行完成，线程正常结束；
2. 线程抛出一个未捕获的Exception或Error；
3. 直接调用该线程stop()方法来结束该线程—该方法容易导致死锁, 通常不推荐使用。

5.1 终止（terminated）状态

线程执行完毕后，进入终止（TERMINATED）状态。

6. 线程相关的方法



6.1 线程就绪、运行和死亡状态转换

- 就绪状态转换为运行状态：此线程得到CPU资源；
- 运行状态转换为就绪状态：此线程主动调用yield()方法或在运行过程中失去CPU资源。
- 运行状态转换为死亡状态：此线程执行执行完毕或者发生了异常；

注意:

> 当调用线程中的yield()方法时，线程从运行状态转换为就绪状态，但接下来CPU调度就绪状态中的那个线程具有一定的随机性，因此，可能会出现A线程调用了yield()方法后，接下来CPU仍然调度了A线程的情况.

6.2 run & start

> 1. start() : 线程的启动
> 2. run() : 线程的执行体

6.3 sleep & yield

sleep()： 通过**sleep**(millis)使线程进入休眠一段时间，**该方法在指定的时间内无法被唤醒，同时也不会释放对象锁**.

需要注意的是：

- sleep方法之后，线程是进入阻塞状态的，只有当睡眠的时间结束，才会重新进入到就绪状态，而就绪状态进入到运行状态，是由系统控制的，我们不可能精准的去干涉它。*所以如果调用Thread.sleep(1000)使得线程睡眠1秒，可能结果会大于1秒。*

yield(): 也是Thread类提供的一个静态的方法，它也可以让当前正在执行的线程暂停，让出CPU资源给其他的线程.但是和**sleep**()方法不同的是，它不会进入到阻塞状态，而是进入到就绪状态.

> 当某个线程调用了yield()方法暂停之后，优先级与当前线程相同，或者优先级比当前线程更高的就绪状态的线程更有可能获得执行的机会，当然，只是有可能，因为我们不可能精确的干涉cpu调度线程。

6.4 join

将几个并行线程的线程合并为一个单线程执行，应用场景是 当一个线程必须等待另一个线程执行完毕才能执行.
Thread类提供了**join**方法来完成这个功能，注意，它不是静态方法。

```
void join()
    当前线程等该加入该线程后面，等待该线程终止。
void join(long millis)
    当前线程等待该线程终止的时间最长为 millis 毫秒。 如果在millis时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待cpu调度
void join(long millis,int nanos)
    等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。如果在millis时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待cpu调度
```

> join方法实现是通过调用wait方法实现。当main线程调用t.join时候，main线程会获得线程对象t的锁（wait 意味着拿到该对象的锁），调用该对象的wait(等待时间)，直到该对象唤醒main线程，比如退出后。这就意味着main 线程调用t.join时，必须能够拿到线程t对象的锁。

6.5 wait & notify/notifyAll

wait & notify/notifyAll这三个都是Object类的方法。使用 wait ，notify 和 notifyAll 前提是先获得调用对象的锁

- 调用 wait 方法后，释放持有的对象锁，线程状态有 Running 变为 Waiting，并将当前线程放置到对象的 等待队列；
- 调用notify 或者 notifyAll 方法后，等待线程依旧不会从 wait 返回，需要调用 noitfy 的线程释放锁之后，等待线程才有机会从 wait 返回；
- notify 方法：将等待队列的一个等待线程从等待队列种移到同步队列中 ，而 notifyAll 方法：将等待队列种所有的线程全部移到同步队列，被移动的线程状态由 Waiting 变为 Blocked。

> 1. 同步队列（锁池）：假设线程A已经拥有了某个对象（注意:不是类）的锁，而其它的线程想要调用这个对象的某个synchronized方法(或者synchronized块)，由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权，但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的同步队列（锁池）中，这些线程状态为Blocked.
> 2. 等待队列（等待池）：假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁，同时线程A就进入到了该对象的等待队列（等待池）中，此时线程A状态为Waiting。如果另外的一个线程调用了相同对象的notifyAll()方法，那么处于该对象的等待池中的线程就会全部进入该对象的同步队列（锁池）中，准备争夺锁的拥有权。如果另外的一个线程调用了相同对象的notify()方法，那么仅仅有一个处于该对象的等待池中的线程（随机）会进入该对象的同步队列（锁池）。

被notify或notifyAll唤起的线程是有规律的，具体如下：

> 1. 如果是通过notify来唤起的线程，那先进入wait的线程会先被唤起来;
> 2. 如果是通过nootifyAll唤起的线程，默认情况是最后进入的会先被唤起来，即LIFO的策略；

6.5 线程优先级

每个线程执行时都有一个优先级的属性，优先级高的线程可以获得较多的执行机会，而优先级低的线程则获得较少的执行机会。与线程休眠类似，线程的优先级仍然无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的也并非没机会执行。

6.6 线程的终止

Thread.stop()、Thread.suspend、Thread.resume，这些终止线程运行的方法已经被废弃了，使用它们是极端不安全的。

> 当suspend的线程持有某个对象锁，而resume它的线程又正好需要使用此锁的时候，死锁就产生了.

想要安全有效的结束一个线程，可以使用下面的方法:

> 1. 正常执行完run方法，然后结束掉.
> 2. 利用break或者某些判断条件的标识符来结束掉线程。(在while循环中)
> 3. 使用interrupt在catch中捕获异常来实现。

interrupt方法做如下说明：

- 每个Thread都有一个中断状状态，默认为false
- 当一个线程处于sleep、wait、join这三种状态之一的时候，如果此时他的中断状态为true，那么它就会抛出一个InterruptedException的异常，并将中断状态重新设置为false。