# 线程的创建

1、继承Thread类，覆盖run方法。

2、构建一个实现Runnable接口的类，然后创建一个Thread类对象并传递Runnable对象作为构造参数。

3、只有调用Thread对象的start方法，才能启动一个新的执行线程。

# 线程信息

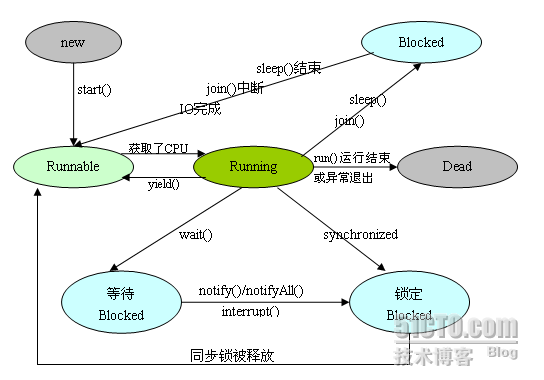
**Thread类对象中保存了一些属性信息，能够帮我们辨别每一个线程：**

1. ID：每个线程的独特标示；
2. Name：线程的名称；
3. Priority：线程对象的优先级。1是最低级，10是最高级
4. Status：线程状态Java中，线程有6种状态。
   1. New：新建状态
   2. Runnable：就绪状态，线程对象创建后，其他线程调用了该对象的start方法。该状态的线程位于可运行线程池中，变得可运行，等待获取cpu的使用权。
   3. Running：运行状态，就绪状态的线程获取了cpu，执行程序代码。
   4. Blocked：阻塞状态，线程因为某种原因放弃cpu使用权，暂时停止运行。直到线程进去就绪状态，才有机会状态运行状态。
   5. Dead：死亡状态，线程执行完毕或者因为异常退出了run方法，该线程结束生命周期。

**阻塞状态分为三种：**

1. 等待阻塞：运行的线程执行wait方法，jvm会把该线程放入等待池中。
2. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则jvm会把该线程放入锁池中。
3. 其他阻塞：运行的线程执行sleep或join方法，或发出了i/o请求时，jvm会把该线程置为阻塞状态。当sleep状态超时、join等待线程终止或超时、或i/o处理完毕时，线程重新进入就绪状态。

**线程状态图**



# 线程中断

1. 多线程应用程序，只有当全部的线程执行结束时（更具体的说，是所有非守护线程结束或者摸个线程调用System.exit方法的时候），它才会结束运行。
2. Java提供中断机制来通知线程，表明我们想要结束它。
3. Thread类提供interrupted()静态方法和isInterrupted()方法，可以检查线程是否被中断，两个方法有很重要的区别。isInterrupted方法不会改变interrupted属性值；interrupted()方法会设置为false，是一个静态方法。建议使用isInterrupted()方法。
4. Thread类的stop方法强行终止线程，但是已经过期，不要使用。**Stop方法不安全的原因举例：**threadA线程拥有了监视器，这些监视器负责保护某些临界资源，比如说银行的转账的金额。当正在转账过程中，main线程调用 threadA.stop()方法。结果导致监视器被释放，其保护的资源（转账金额）很可能出现不一致性。比如，A账户减少了100，而B账户却没有增加100。

# 操作线程的中断机制

1. 中断一个线程只是为了引起该线程的注意，被中断的线程可以决定如果应对中断。
2. 对于sleep、join等操作的线程，如果被调用了interrupt()方法后，会抛出InterruptedException，然后线程的中断标志位会由true重置为false，因为线程为了处理异常已经重新处于就绪状态。
3. Thread类的关于中断的三个方法：
   1. public static boolean **interrupted:** 测试当前线程是否已经中断。线程的*中断状态* 由该方法清除。换句话说，如果连续两次调用该方法，则第二次调用将返回 false。
   2. public boolean **isInterrupted**():测试线程是否已经中断。线程的*中断状态* 不受该方法的影响。
   3. public void **interrupt**():中断线程。
4. 总结：其实，java的中断是一种协作机制，也就是说调用线程对象的interrupt方法并不一定就中断了正在运行的线程，它只是要求线程自己在合适的时机中断自己。每个线程都有一个boolean的中断状态，interrupt方法只是将改状态置为true。（对正常运行的线程调用interrupt方法并不能终止它，只是该表了interrupt标识符）一般来说，如果一个方法抛出InterruptedException，表示该方法是可中断的,比如wait,sleep,join，也就是说可中断方法会对interrupt调用做出响应（例如sleep响应interrupt的操作包括清除中断状态，抛出InterruptedException）,异常都是由可中断方法自己抛出来的，并不是直接由interrupt方法直接引起的。
5. Object.wait, Thread.sleep方法，会不断的轮询监听 interrupted 标志位，发现其设置为true后，会停止阻塞并抛出 InterruptedException异常。

# 线程的睡眠和恢复

1. sleep方法使线程睡眠，单位：毫秒，sleep时间结束后，jvm会安排它们cpu时间，线程会继续按指令执行。
2. 在sleep时间段内，注意不释放对象锁标志。也就是说如果有synchronized同步快，其他线程仍然不能访问共享数据。sleep使优先级低的线程得到执行的机会，当然也可以让同优先级的线程有执行的机会；
3. 在sleep时间段内，如果Thread是睡眠和中断，方法会立刻抛出InterruptedException异常，并不会一直等待睡眠时间过去。
4. Yield方法也能让线程对象离开cpu，类似sleep，yield方法向jvm表示线程对象可以让cpu执行其他任务。Jvm不保证遵守请求，通常它只是用来调试。不能由用户指定暂停多长时间，并且只能让同优先级的线程有执行的机会。yield()也不会释放锁标志。
5. Join方法，等待调用该方法的线程执行完毕后再往下继续执行。
6. **wait()和notify()、notifyAll()：**
   1. 这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用。synchronized关键字用于保护共享数据，阻止其他线程对共享数据的存取，但是这样程序的流程就很不灵活了，如何才能在当前线程还没退出synchronized数据块时让其他线程也有机会访问共享数据呢？此时就用这三个方法来灵活控制。
   2. wait()方法使当前线程暂停执行并释放对象锁标示，让其他线程可以进入synchronized数据块，当前线程被放入对象等待池中。当调用notify()方法后，将从对象的等待池中移走一个任意的线程并放到锁标志等待池中，只有锁标志等待池中线程能够获取锁标志；如果锁标志等待池中没有线程，则notify()不起作用。
   3. notifyAll()则从对象等待池中移走所有等待那个对象的线程并放到锁标志等待池中。
   4. wait()和notify()因为会对对象的“锁标志”进行操作，他们必需在Synchronized函数或者 synchronized block 中进行调用。如果在non-synchronized 函数或 non-synchronized block 中进行调用，虽然能编译通过，但在运行时会抛出IllegalMonitorStateException的异常。
   5. **注意 这三个方法都是java.lang.Object的方法。**

# 守护线程

1. 用来服务用户线程。
2. setDaemon方法，必须在线程启动之前调用，当线程正在运行时调用会产生异常。
3. 当一个守护线程产生新的线程，默认都是守护线程。
4. Java垃圾回收线程就是一个典型的守护线程。

# ThreadLocal

1. 如果希望各个线程的属性不会被共享，java并发api提供了一个很清楚的机制叫本地线程变量。
2. ThreadLocal可以在线程内实现一个局部变量。可以在线程的任何地方来访问，能够减少参数的传递。
3. InheritableThreadLocal可以在子线程和父线程之间共享实例，也同样是为了减少参数的传递。

# Synchronized

1. 一个执行线程访问了被synchronized关键字声明的静态方法，另一个线程可以访问该类的一个对象中的其他非静态的方法。必须非常注意这一点，因为两个线程可以访问两个不同的同步方法，如果其中一个是静态的而另一个不是。如果这两种方法改变相同的数据,你将会有数据不一致的错误。

# Lock

# ReadWriteLock

# 线程同步工具

## Semaphore

1. 当一个线程想要访问某个共享资源，首先，它必须获得semaphore。如果semaphore的内部计数器的值大于0，那么semaphore减少计数器的值并允许访问共享的资源。计数器的值大于0表示，有可以自由使用的资源，所以线程可以访问并使用它们。
2. 可修改Semaphores的公平性，在默认的情况下信号量的进入是不公平的。如果在初始化的第二个参数设定为true时，则会选择时间等待最久的一个进入。