

创建线程几种方式

为什么实现 Runnable 接口比继承 Thread 类实现线程要好?

interrupt讲解

如何使用interrupt停止线程:

为什么说 volatile 修饰标记位停止方法是错误的:

stop()、suspend() 和 resume()这几个停止线程方法为什么被弃用了:

线程的6种状态

New 新创建:

Runnable 可运行:

阻塞状态:

Blocked 被阻塞:

Waiting 等待:

Timed Waiting 限期等待:

Terminated 终止:

注意点:

wait/notify/notifyAll 讲解

wait/notify/notifyAll 示例:

虚假唤醒:

为什么 if会出现虚假唤醒:

为什么 wait/notify/notifyAll 被定义在 Object 类中, 而 sleep 定义在 Thread 类中?

wait/notify 和 sleep 方法的异同?

线程的sleep()方法和yield()方法有什么区别?

wait/notify、Condition、BlockingQueue 实现生产者消费者模式

生产者消费者模式介绍:

如何用 BlockingQueue 实现生产者消费者模式:

如何用 Condition 实现生产者消费者模式:

如何用 wait/notify 实现生产者消费者模式

中断总结:

join用法和原理

join的作用

join的原理

## 创建线程几种方式

### 1. 实现 Runnable 接口

```
1 public class RunnableThread implements Runnable {
2     @Override
3     public void run() {
4         System.out.println("implents Runnable ...");
5     }
6     public static void main(String[] args) {
7         new Thread(new RunnableThread()).start();//也是使用new Thread();
8     }
9 }
```

### 2. 继承 Thread 类

```

1 public class RunnableThread extends Thread {
2     @Override
3     public void run() {
4         System.out.println("implents Runnable ...");
5     }
6     public static void main(String[] args) {
7         new Thread(new RunnableThread()).start();//也是使用new Thread();
8     }
9 }

```

```

public
class Thread implements Runnable {
    /* Make sure registerNatives is the first thing <clinit>
    private static native void registerNatives();
    static {
        registerNatives();
    }

    private volatile String name;
    private int priority;

    /* Whether or not the thread is a daemon thread. */
    private boolean daemon = false;

    /* Fields reserved for exclusive use by the JVM */
    private boolean stillborn = false;
    private long eetop;

    /* What will be run. */
    private Runnable target;

```

3. 线程池创建线程（实现ThreadFactory接口）
4. 有返回值的 Callable 创建线程（运行时会有返回值，上面几个都是没有返回值的）
5. 定时器 Timer

实现线程只有一种：new Thread();

## 为什么实现 Runnable 接口比继承 Thread 类实现线程要好？

1. 实现了 Runnable 与 Thread 类的解耦
2. 可以提高性能，可以不用每次都创建线程。（使用实现 Runnable 接口的方式，就可以把任务直接传入线程池）
3. 拓展性好，Java 语言不支持双继承，如果我们的类一旦继承了 Thread 类，那么它后续就没有办法再继承其他的类

## interrupt讲解

## 如何使用interrupt停止线程：

```
1 while (!Thread.currentThread().isInterrupted() && more work to do) {
2     do more work
3 }
4
5 //isInterrupted源码，如果这个线程的中断标记位就会被设置成 true，就说明有程序想终止该线程
6 public boolean isInterrupted() {
7     return isInterrupted(false);
8 }
```

sleep 期间能否感受到中断，遇到了 InterruptedException，应该如何处理

当线程sleep期间出现 interrupt，这时会抛出InterruptedException，但线程并没有真正中断，需要以下方法解决：

```
1 //1.方法直接抛异常
2 void subTask2() throws InterruptedException {
3     Thread.sleep(1000);
4 }
5 //2.再次中断
6 private void reInterrupt() {
7     try {
8         Thread.sleep(2000);
9     } catch (InterruptedException e) {
10         Thread.currentThread().interrupt();
11         e.printStackTrace();
12     }
13 }
```

## 为什么说 volatile 修饰标记位停止方法是错误的：

使用volatile 停止线程：

```
1 public class MyThread implements Runnable{
2     private volatile boolean flag;
3     public void stop() {
4         flag = false;
5     }
6     @Override
7     public void run() {
8         while(true)
9             System.out.println("a");
10    }
11 }
12 public class Test_thread {
13     public static void main(String[] args) {
14         MyThread mt = new MyThread();
15         mt.run();
16         mt.stop();
17     }
18 }
19 }
```

volatile 这种方法在某些特殊的情况下，比如线程被长时间阻塞的情况，就无法及时感受中断，所以 volatile 是不够全面的停止线程的方法。

## stop()、suspend() 和 resume()这几个停止线程方法为什么被弃用了：

1. **stop()**：会直接把线程停止，这样就没有给线程足够的时间来处理想要在停止前保存数据的逻辑，任务戛然而止，会导致出现数据完整性等问题。
2. **suspend()和resume()**：它并不会释放锁，就开始进入休眠，但此时有可能仍持有锁，这样就容易导致死锁问题，因为这把锁在线程被 resume() 之前，是不会被释放的。

假设线程 A 调用了 suspend() 方法让线程 B 挂起，线程 B 进入休眠，而线程 B 又刚好持有一把锁，此时假设线程 A 想访问线程 B 持有的锁，但由于线程 B 并没有释放锁就进入休眠了，所以对于线程 A 而言，此时拿不到锁，也会陷入阻塞，那么线程 A 和线程 B 就都无法继续向下执行。

正是因为有这样的风险，所以 suspend() 和 resume() 组合使用的方法也被废弃了。

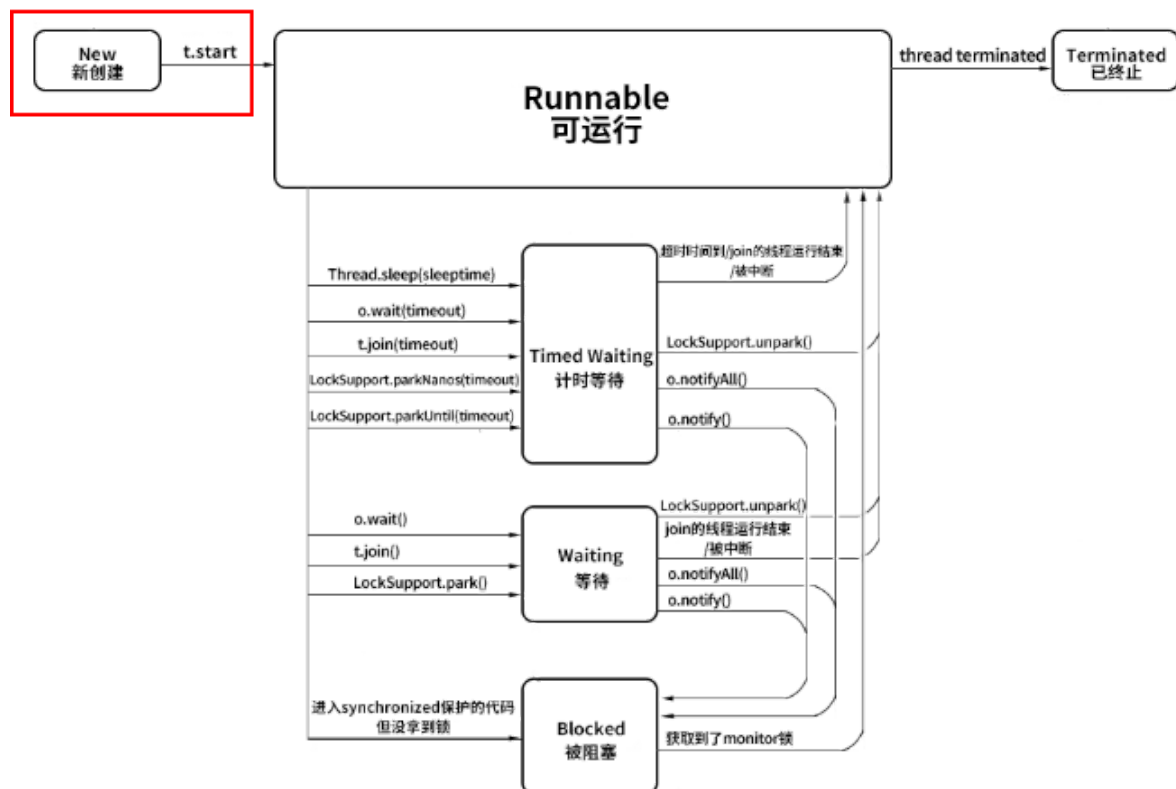
## 线程的6种状态

1. New (新创建)
2. Runnable (可运行)
3. Blocked (被阻塞)
4. Waiting (等待)
5. Timed Waiting (计时等待)
6. Terminated (被终止)

如果想要确定线程当前的状态，可以通过 getState() 方法，并且线程在任何时刻只可能处于 1 种状态。

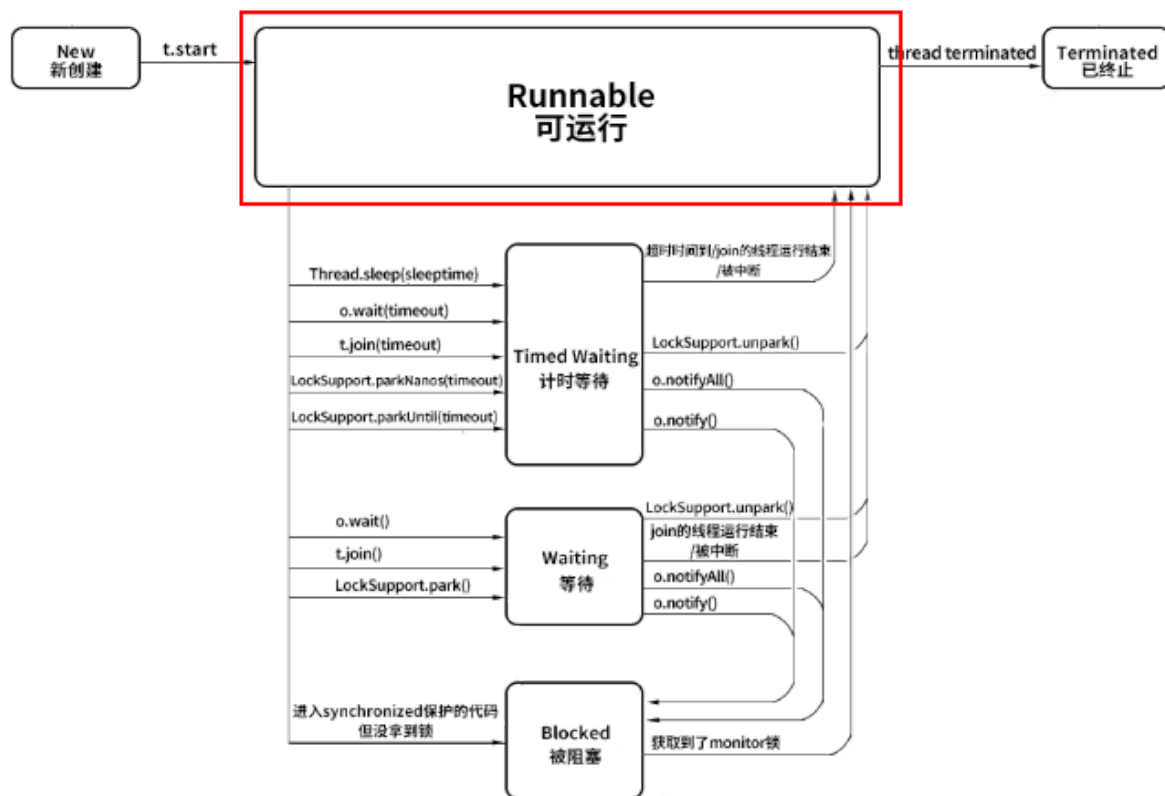
### New 新创建：

下面我们逐个介绍线程的 6 种状态，如图所示，首先来看下左上角的 New 状态。



New 表示线程被创建但尚未启动的状态：当我们用 new Thread() 新建一个线程时，如果线程没有开始运行 start() 方法，所以也没有开始执行 run() 方法里面的代码，那么此时它的状态就是 New。而一旦线程调用了 start()，它的状态就会从 New 变成 Runnable，也就是状态转换图中中间的这个大方框里的内容。

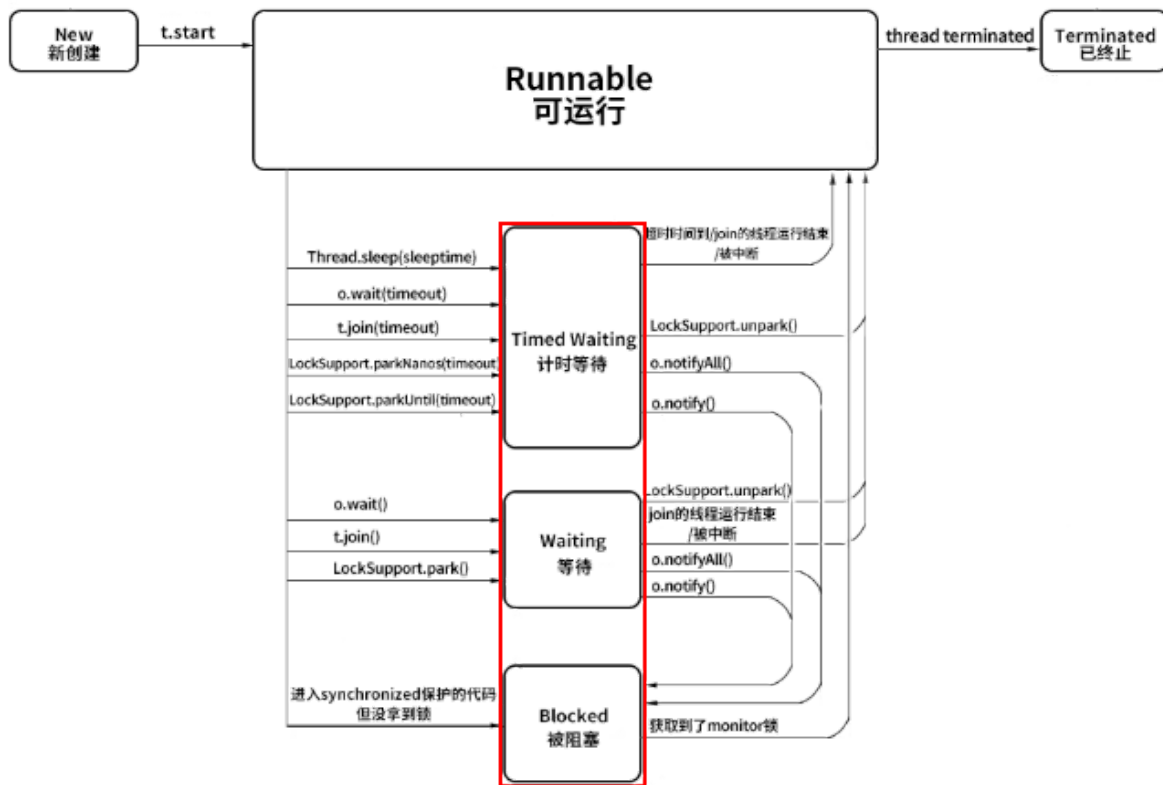
## Runnable 可运行:



Java 中的 Runnable 状态对应操作系统线程状态中的两种状态，分别是 Running 和 Ready，也就是说，Java 中处于 Runnable 状态的线程有可能正在执行，也有可能没有正在执行，正在等待被分配 CPU 资源。

所以，如果一个正在运行的线程是 Runnable 状态，当它运行到任务的一半时，执行该线程的 CPU 被调度去做其他事情，导致该线程暂时不运行，它的状态依然不变，还是 Runnable，因为它有可能随时被调度回来继续执行任务。

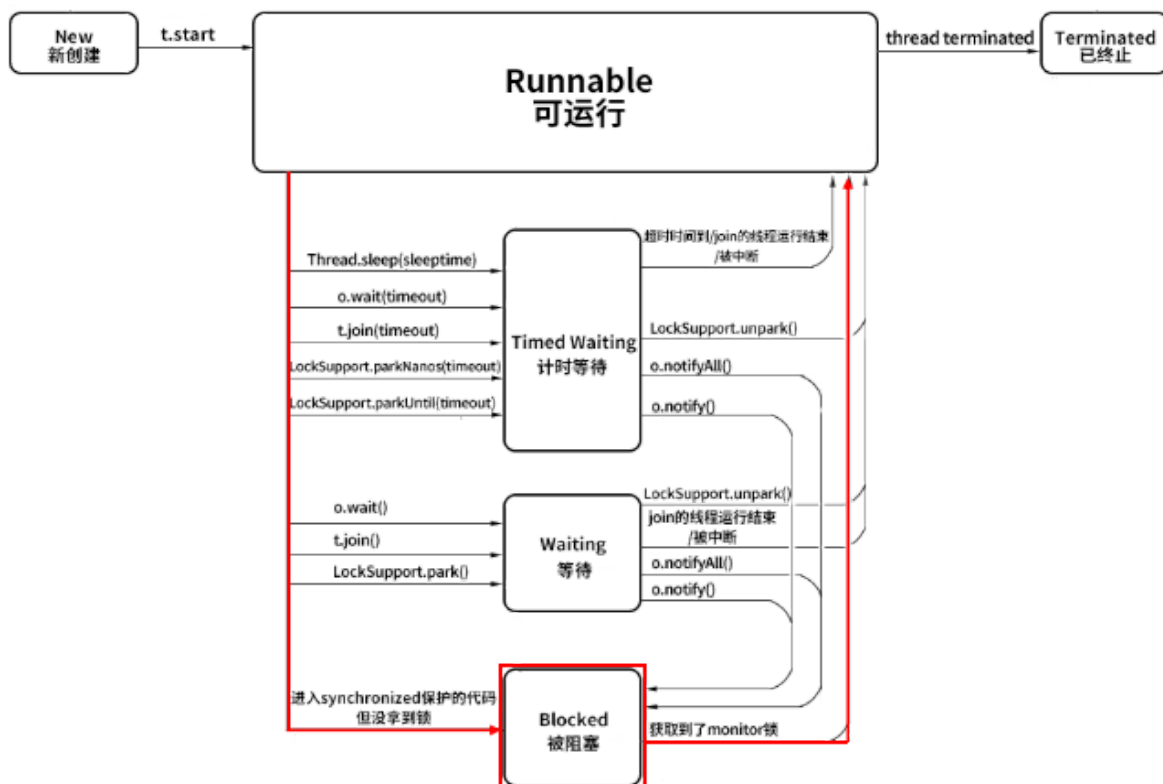
## 阻塞状态:



接下来，我们来看下 Runnable 下面的三个方框，它们统称为阻塞状态，在 Java 中阻塞状态通常不仅仅是 Blocked，实际上它包括三种状态，分别是 Blocked(被阻塞)、Waiting(等待)、Timed Waiting(计时等待)，这三种状态统称为阻塞状态，下面我们来看看这三种状态具体是什么含义。

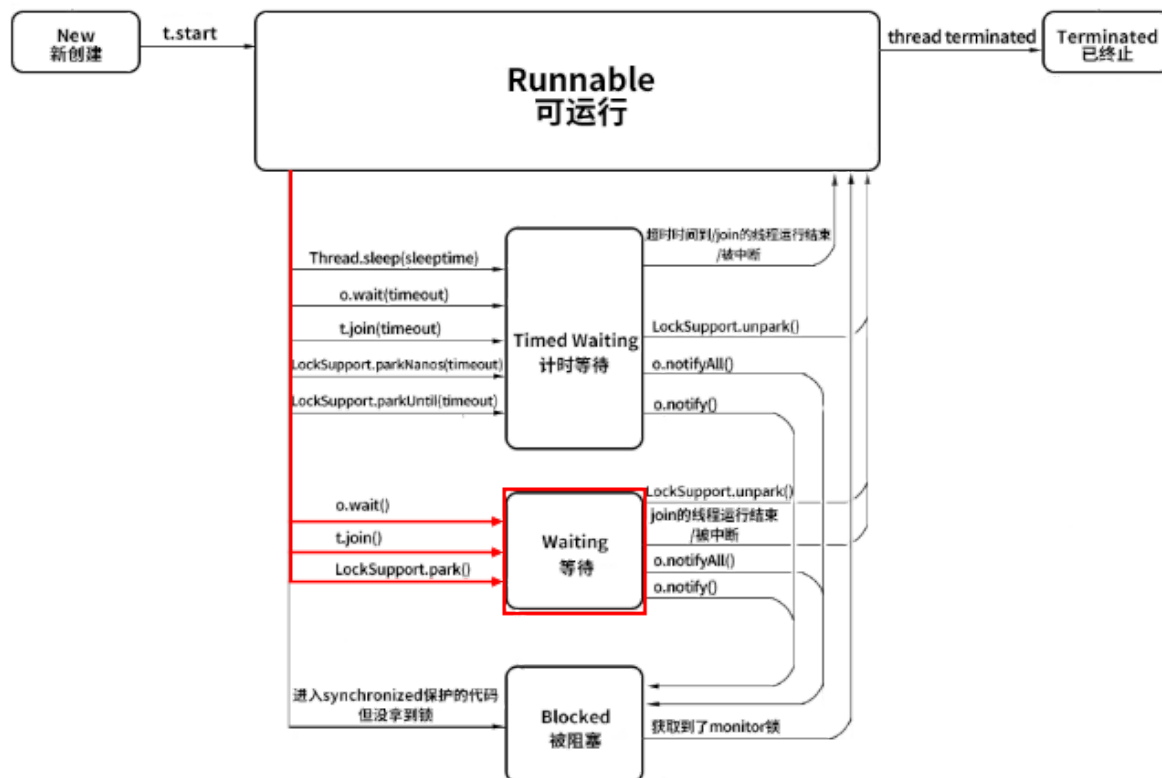
### Blocked 被阻塞：

首先来看最简单的 Blocked，从箭头的流转方向可以看出，从 Runnable 状态进入 Blocked 状态只有一种可能，就是进入 synchronized 保护的代码时没有抢到 monitor 锁，无论是进入 synchronized 代码块，还是 synchronized 方法，都是一样。



我们再往右看，当处于 Blocked 的线程抢到 monitor 锁，就会从 Blocked 状态回到 Runnable 状态。

## Waiting 等待：



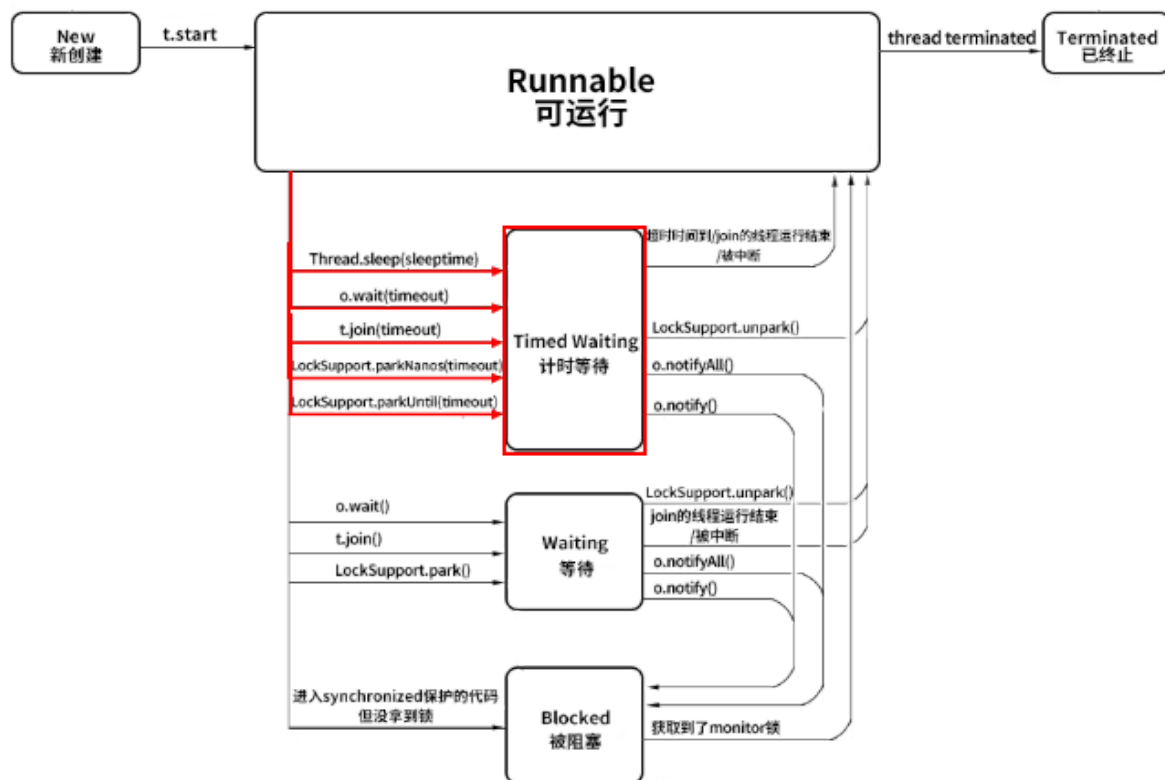
我们再看看 Waiting 状态，线程进入 Waiting 状态有三种可能性。

1. 没有设置 Timeout 参数的 `Object.wait()` 方法。
2. 没有设置 Timeout 参数的 `Thread.join()` 方法。
3. `LockSupport.park()` 方法。

刚才强调过，Blocked 仅仅针对 synchronized monitor 锁，可是在 Java 中还有很多其他的锁，比如 ReentrantLock，如果线程在获取这种锁时没有抢到该锁就会进入 Waiting 状态，因为本质上它执行了 `LockSupport.park()` 方法，所以会进入 Waiting 状态。同样，`Object.wait()` 和 `Thread.join()` 也会让线程进入 Waiting 状态。

Blocked 与 Waiting 的区别是 Blocked 在等待其他线程释放 monitor 锁，而 Waiting 则是在等待某个条件，比如 join 的线程执行完毕，或者是 `notify()/notifyAll()`。

## Timed Waiting 限期等待：



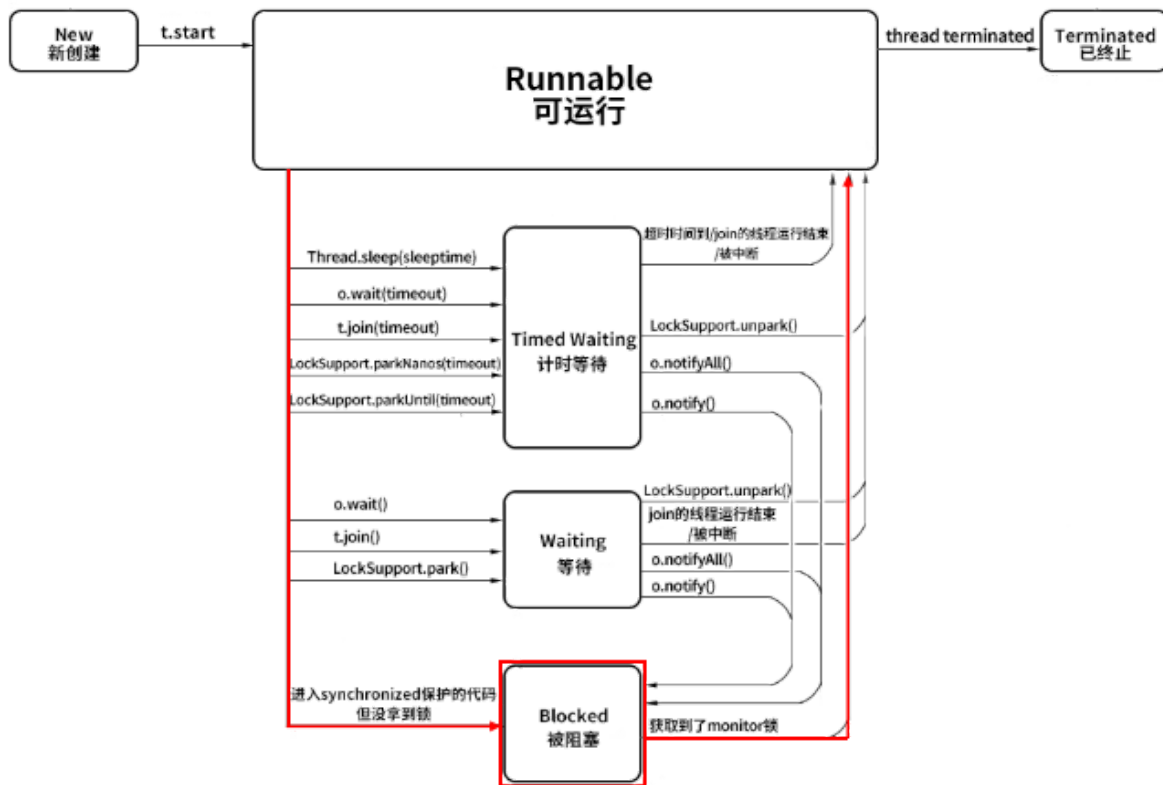
在 Waiting 上面是 Timed Waiting 状态，这两个状态是非常相似的，区别仅在于有没有时间限制，Timed Waiting 会等待超时，由系统自动唤醒，或者在超时前被唤醒信号唤醒。

以下情况会让线程进入 Timed Waiting 状态。

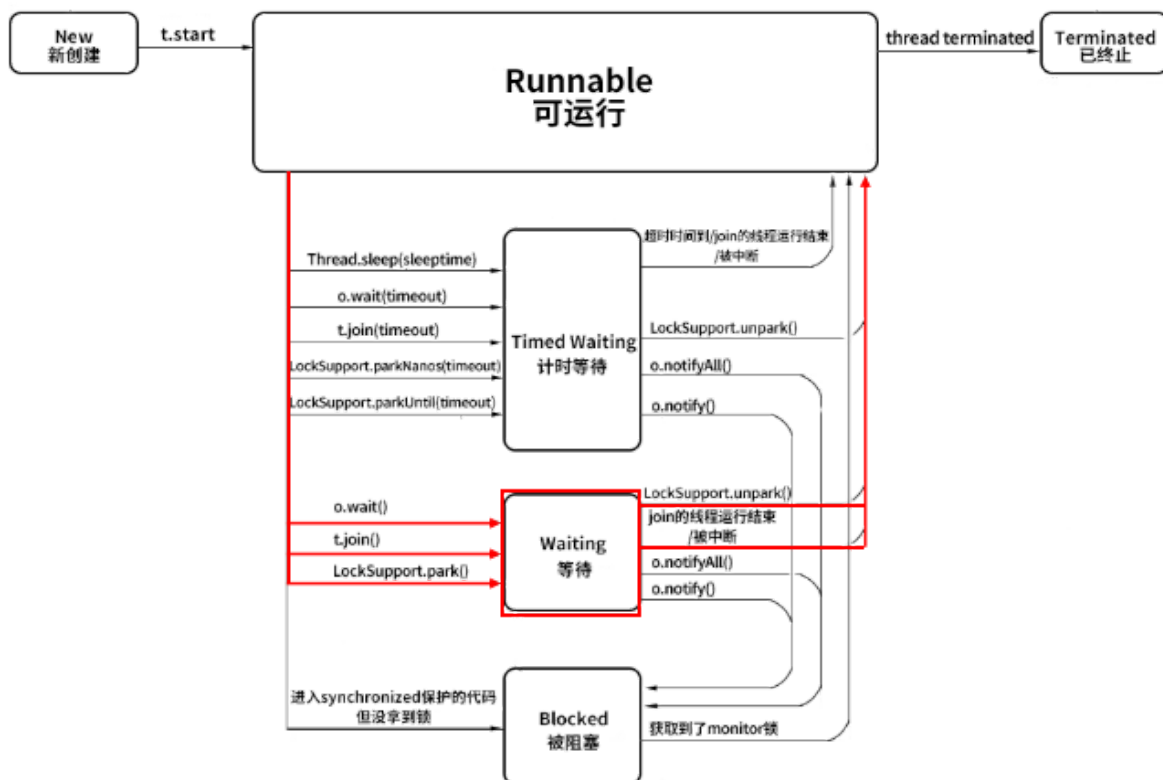
1. 设置了时间参数的 Thread.sleep(long millis) 方法；
2. 设置了时间参数的 Object.wait(long timeout) 方法；
3. 设置了时间参数的 Thread.join(long millis) 方法；
4. 设置了时间参数的 LockSupport.parkNanos(long nanos) 方法和 LockSupport.parkUntil(long deadline) 方法。

讲完如何进入这三种状态，我们再来看下如何从这三种状态流转到下一个状态。

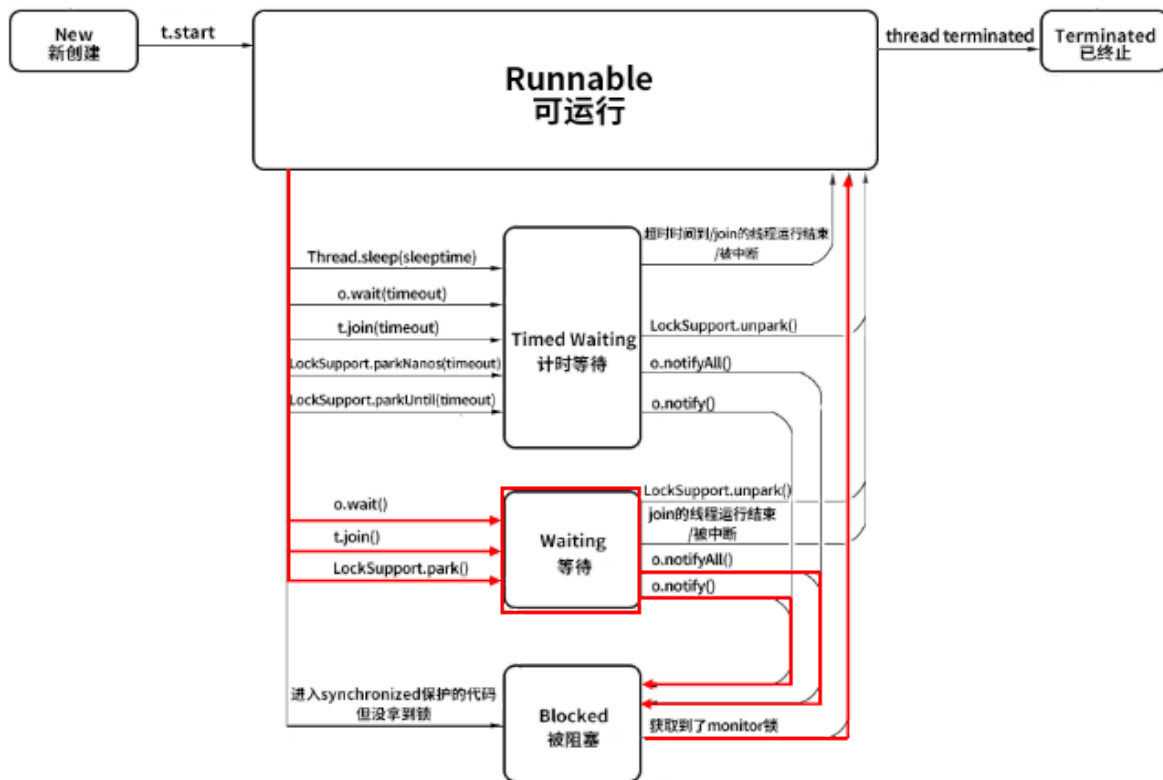




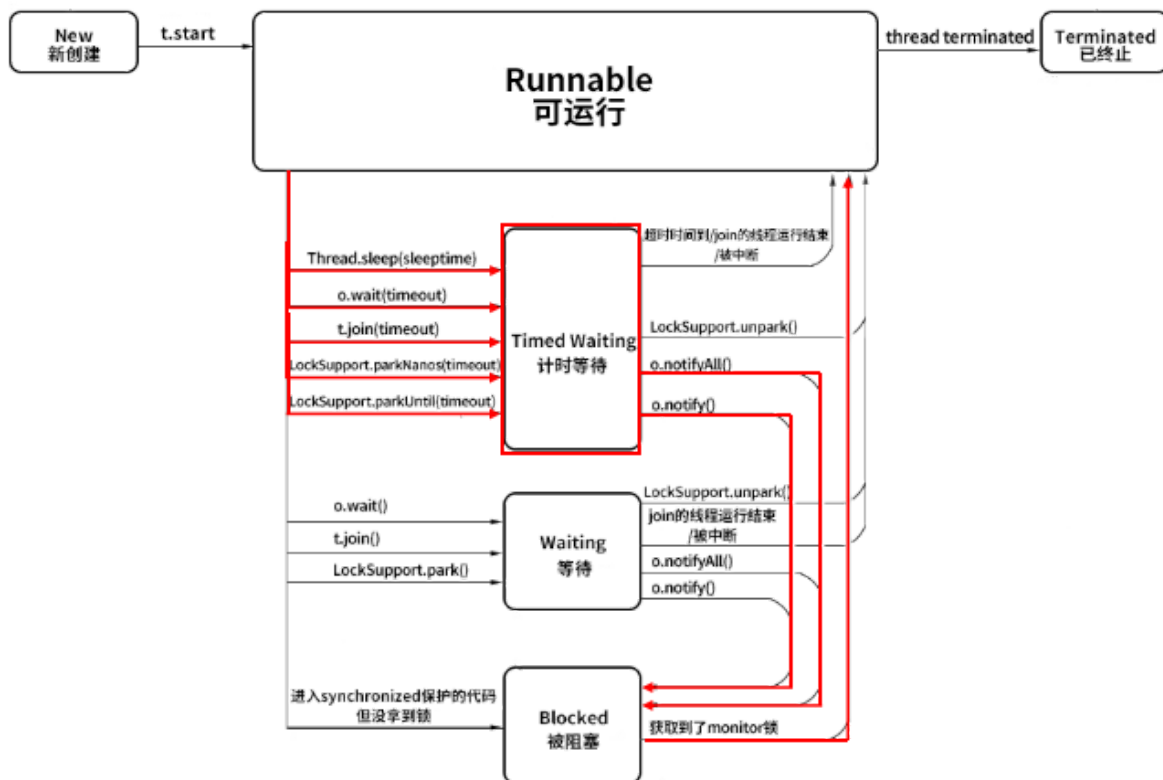
想要从 Blocked 状态进入 Runnable 状态，要求线程获取 monitor 锁，而从 Waiting 状态流转到其他状态则比较特殊，因为首先 Waiting 是无限时的，也就是说无论过了多长时间它都不会主动恢复。



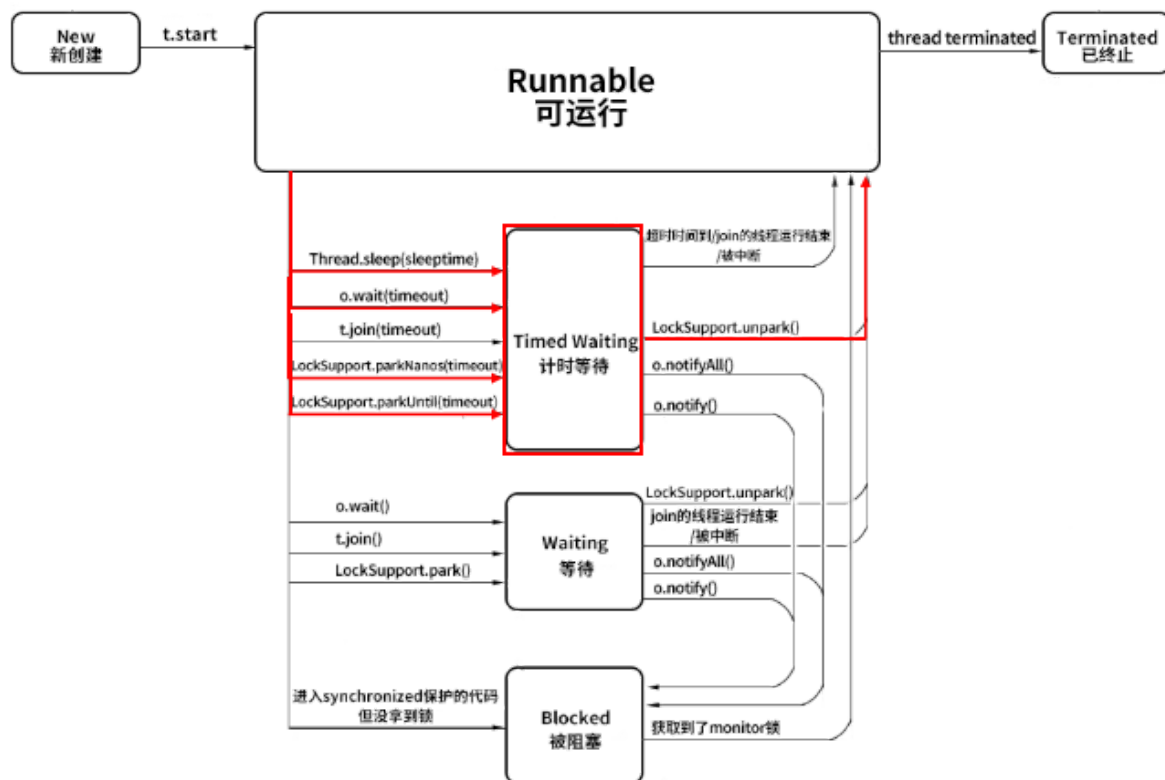
只有当执行了 `LockSupport.unpark()`，或者 `join` 的线程运行结束，或者被中断时才可以进入 Runnable 状态。



如果其他线程调用 `notify()` 或 `notifyAll()` 来唤醒它，它会直接进入 Blocked 状态，这是为什么呢？因为唤醒 Waiting 线程的线程如果调用 `notify()` 或 `notifyAll()`，要求必须首先持有该 monitor 锁，所以处于 Waiting 状态的线程被唤醒时拿不到该锁，就会进入 Blocked 状态，直到执行了 `notify()/notifyAll()` 的唤醒它的线程执行完毕并释放 monitor 锁，才可能轮到它去抢夺这把锁，如果它能抢到，就会从 Blocked 状态回到 Runnable 状态。

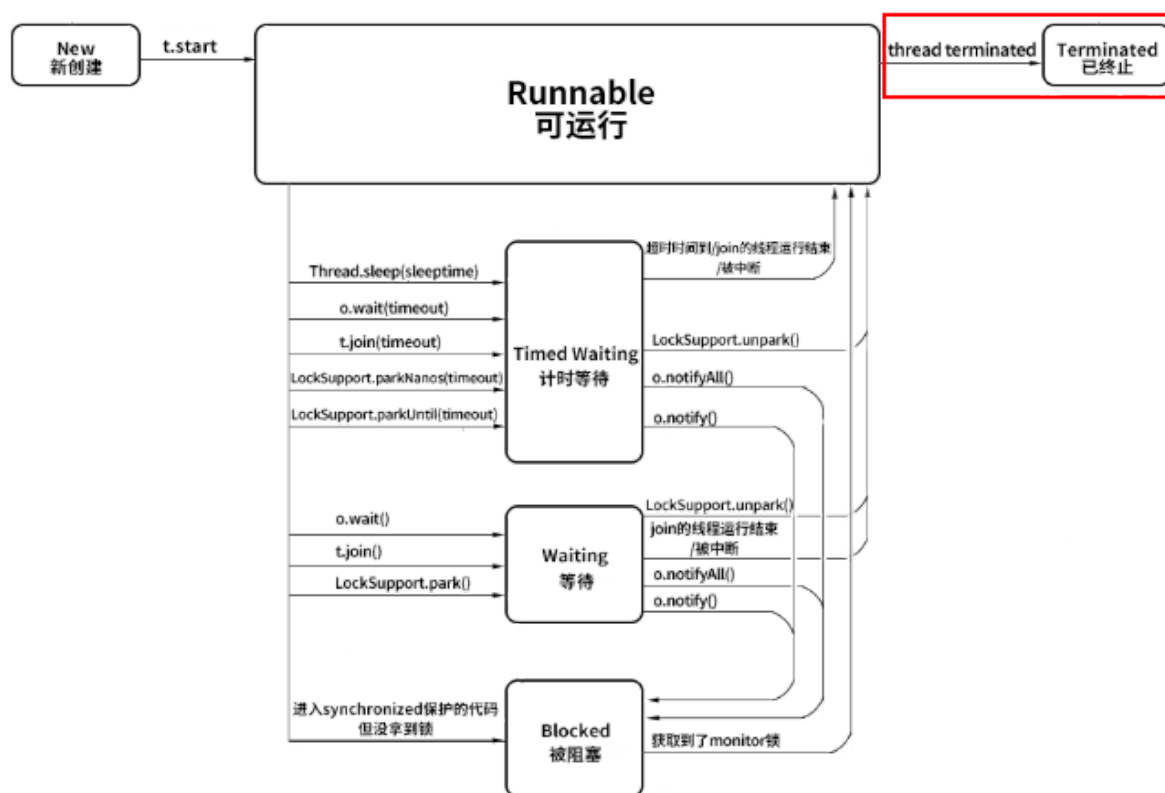


同样在 Timed Waiting 中执行 `notify()` 和 `notifyAll()` 也是一样的道理，它们会先进入 Blocked 状态，然后抢夺锁成功后，再回到 Runnable 状态。



当然对于 Timed Waiting 而言，如果它的超时时间到了且能直接获取到锁/join的线程运行结束/被中断/调用了LockSupport.unpark(), 会直接恢复到 Runnable 状态，而无需经历 Blocked 状态。

### Terminated 终止：



再看看最后一种状态，Terminated 终止状态，要想进入这个状态有两种可能。

- run() 方法执行完毕，线程正常退出。
- 出现一个没有捕获的异常，终止了 run() 方法，最终导致意外终止。

### 注意点：

最后我们再看线程转换的两个注意点。

1. 线程的状态是需要按照箭头方向来走的，比如线程从 New 状态是不可以直接进入 Blocked 状态的，它需要先经历 Runnable 状态。
2. 线程生命周期不可逆：一旦进入 Runnable 状态就不能回到 New 状态；一旦被终止就不可能再有任何状态的变化。所以一个线程只能有一次 New 和 Terminated 状态，只有处于中间状态才可以相互转换。

## wait/notify/notifyAll 讲解

wait 方法必须在 synchronized 保护的同步代码中使用，如果不这样的话，如下代码，如果在 while (buffer.isEmpty())

```
1  class BlockingQueue {
2      Queue<String> buffer = new LinkedList<String>();
3      public void give(String data) {
4          buffer.add(data);
5          notify(); // Since someone may be waiting in take
6      }
7      public String take() throws InterruptedException {
8          while (buffer.isEmpty()) {
9              wait();
10         }
11         return buffer.remove();
12     }
13 }
```

这段代码没有受 synchronized 保护，于是便有可能发生以下场景：

1. 首先，消费者线程调用 take 方法并判断 buffer.isEmpty 方法是否返回 true，若为 true 代表buffer是空的，则线程希望进入等待，但是在线程调用 wait 方法之前，就被调度器暂停了，所以此时还没来得及执行 wait 方法。
2. 此时生产者开始运行，执行了整个 give 方法，它往 buffer 中添加了数据，并执行了 notify 方法，但 notify 并没有任何效果，因为消费者线程的 wait 方法没来得及执行，所以没有线程在等待被唤醒。
3. 此时，刚才被调度器暂停的消费者线程回来继续执行 wait 方法并进入了等待，有可能陷入无穷无尽的等待，因为它错过了刚才 give 方法内的 notify 的唤醒。

**wait 方法会释放 monitor 锁**，这也要求我们必须首先进入到 synchronized 内持有这把锁。

**总结：**wait 方法没有加synchronized 的话，如果在执行wait之前出现了暂停，这时执行 notify方法，后面再执行wait会出现无穷无尽的等待（因为notify已经执行过了）。

## wait/notify/notifyAll 示例：

```
1  public class NotifyThread extends Thread {
2      private final Object lock;
3      public NotifyThread(Object lock) {
4          super();
5          this.lock = lock;
6      }
7      @Override
```

```

8      public void run() {
9          synchronized (lock) {
10              System.out.println("开始 notify time= " + System.currentTimeMillis());
11              lock.notify();
12              System.out.println("结束 notify time= " + System.currentTimeMillis());
13          }
14      }
15  }
16  public class WaitThread extends Thread {
17      private final Object lock;
18      public WaitThread(Object lock) {
19          super();
20          this.lock = lock;
21      }
22      @Override
23      public void run() {
24          try {
25              synchronized (lock) {
26                  long start = System.currentTimeMillis();
27                  lock.wait();
28                  long end = System.currentTimeMillis();
29              }
30          } catch (InterruptedException e) {
31              e.printStackTrace();
32          }
33      }
34  }
35  public class TestWaitNotify {
36      public static void main(String[] args) {
37          Object lock = new Object();
38          WaitThread t1 = new WaitThread(lock);
39          t1.start();
40          NotifyThread t2 = new NotifyThread(lock);
41          t2.start();
42      }
43  }

```

## 虚假唤醒：

### 为什么 if 会出现虚假唤醒：

- 1 因为 if 只会执行一次，执行完会接着向下执行 if ( ) 外边的
- 2 而 while 不会，直到条件满足才会向下执行 while ( ) 外边的

### 为什么会出现虚假唤醒：

线程可能在既没有被 notify/notifyAll，也没有被中断或者超时的情况下被唤醒，这种唤醒是我们不希望看到的。

### 虚假唤醒解决方案：

在 wait() 里面加 while() 判断，不满足条件就继续 wait。

### 虚假唤醒示例：

- 1 反例：
- 2 `if(num<=0) {`

```

3         System.out.println("库存已空，无法卖货");
4         try {
5             this.wait();
6         } catch (InterruptedException e) {
7             e.printStackTrace();
8         }
9     }
10    System.out.println(Thread.currentThread().getName()+" : "+(num--));
11    this.notifyAll();
12    正例：
13    while(num<=0) {
14        System.out.println("库存已空，无法卖货");
15        try {
16            this.wait();
17        } catch (InterruptedException e) {
18            e.printStackTrace();
19        }
20    }
21    System.out.println(Thread.currentThread().getName()+" : "+(num--));
22    this.notifyAll();

```

资源num==0：此时两个消费者线程都wait。

生产者执行num++后，唤醒了所有等待的线程。

此时这两个消费者线程抢占资源后立马执行wait之后的操作，即num--，就会出现产品为负的情况。

while () 的话两次线程互不干扰，分开判断，if只判断一次，第二个线程可以直接跳过判断。

## 为什么 wait/notify/notifyAll 被定义在 Object 类中，而 sleep 定义在 Thread 类中？

我们来看第二个问题，为什么 wait/notify/notifyAll 方法被定义在 Object 类中？而 sleep 方法定义在 Thread 类中？主要有两点原因：

1. 因为 Java 中每个对象都有一把称之为 monitor 监视器的锁，由于每个对象都可以上锁，这就要求在对象头中有一个用来保存锁信息的位置。这个锁是对象级别的，而非线程级别的，wait/notify/notifyAll 也都是锁级别的操作，它们的锁属于对象，所以把它们定义在 Object 类中是最合适，因为 Object 类是所有对象的父类。（因为锁是对象级别的）
2. 因为如果把 wait/notify/notifyAll 方法定义在 Thread 类中，会带来很大的局限性，比如一个线程可能持有多把锁，以便实现相互配合的复杂逻辑，假设此时 wait 方法定义在 Thread 类中，如何实现让一个线程持有多把锁呢？又如何明确线程等待的是哪把锁呢？既然我们是让当前线程去等待某个对象的锁，自然应该通过操作对象来实现，而不是操作线程。

**总结：**因为wait需要配合锁使用，如果锁是线程级的，会带来很大的局限性，比如一个线程可能持有多把锁，以便实现相互配合的复杂逻辑，如何实现让一个线程持有多把锁呢？又如何明确线程等待的是哪把锁这个比较难操作。

## wait/notify 和 sleep 方法的异同？

第三个问题是对比 wait/notify 和 sleep 方法的异同，主要对比 wait 和 sleep 方法，

我们先说相同点：

1. 它们都可以让线程阻塞。
2. 它们都可以响应 interrupt 中断：在等待的过程中如果收到中断信号，都可以进行响应，并抛出 InterruptedException 异常。

但是它们也有很多的不同点：

1. wait 方法必须在 synchronized 保护的代码中使用，而 sleep 方法并没有这个要求。
2. 在同步代码中执行 sleep 方法时，并不会释放 monitor 锁，但执行 wait 方法时会主动释放 monitor 锁。
3. sleep 方法中会要求必须定义一个时间，时间到期后会主动恢复，而对于没有参数的 wait 方法而言，意味着永久等待，直到被中断或被唤醒才能恢复，它并不会主动恢复。
4. wait/notify 是 Object 类的方法，而 sleep 是 Thread 类的方法。

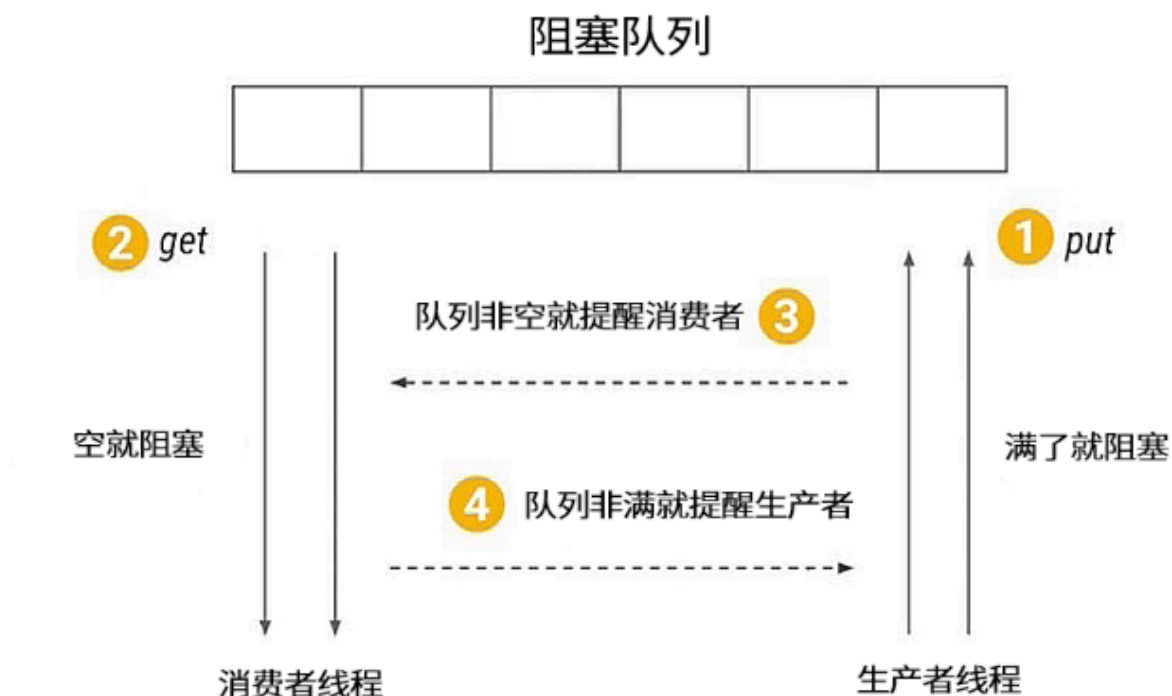
## 线程的sleep()方法和yield()方法有什么区别？

Thread.yield()方法作用是：**暂停当前正在执行的线程对象，并执行其他线程。**

- ① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
- ② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
- ③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
- ④ sleep()方法比yield()方法（跟操作系统CPU 调度相关）具有更好的可移植性。

## wait/notify、Condition、BlockingQueue 实现生产者消费者模式

### 生产者消费者模式介绍：



我们先来看看什么是生产者消费者模式，生产者消费者模式是程序设计中非常常见的一种设计模式，被广泛运用在解耦、消息队列等场景。在现实世界中，我们把生产商品的一方称为生产者，把消费商品的一方称为消费者，有时生产者的生产速度特别快，但消费者的消费速度跟不上，俗称“产能过剩”，又或是多个生产者对应多个消费者时，大家可能会手忙脚乱。如何才能让大家更好地配合呢？这时在生产者和消费者之间就需要一个中介来进行调度，于是便诞生了生产者消费者模式。



使用生产者消费者模式通常需要在两者之间增加一个阻塞队列作为媒介，有了媒介之后就相当于有了一个缓冲，平衡了两者的能力，整体的设计如图所示，最上面是阻塞队列，右侧的 1 是生产者线程，生产者在生产数据后将数据存放在阻塞队列中，左侧的 2 是消费者线程，消费者获取阻塞队列中的数据。而中间的 3 和 4 分别代表生产者消费者之间互相通信的过程，因为无论阻塞队列是满还是空都可能会产生阻塞，阻塞之后就需要在合适的时机去唤醒被阻塞的线程。

那么什么时候阻塞线程需要被唤醒呢？有两种情况。第一种情况是当消费者看到阻塞队列为空时，开始进入等待，这时生产者一旦往队列中放入数据，就会通知所有的消费者，唤醒阻塞的消费者线程。另一种情况是如果生产者发现队列已经满了，也会被阻塞，而一旦消费者获取数据之后就相当于队列空了一个位置，这时消费者就会通知所有正在阻塞的生产者进行生产，这便是对生产者消费者模式的简单介绍。

## 如何用 BlockingQueue 实现生产者消费者模式：

```
1 public static void main(String[] args) {
2
3     BlockingQueue<Object> queue = new ArrayBlockingQueue<>(10);
4     Runnable producer = () -> {
5         while (true) {
6             queue.put(new Object());
7         }
8     };
9     new Thread(producer).start();
10    new Thread(producer).start();
11
12    Runnable consumer = () -> {
13        while (true) {
14            queue.take();
15        }
16    };
17    new Thread(consumer).start();
18    new Thread(consumer).start();
19 }
```

如代码所示，首先，创建了一个 ArrayBlockingQueue 类型的 BlockingQueue，命名为 queue 并将它的容量设置为 10；其次，创建一个简单的生产者，while(true) 循环体中的 queue.put() 负责往队列添加数据；然后，创建两个生产者线程并启动；同样消费者也非常简单，while(true) 循环体中的 queue.take() 负责消费数据，同时创建两个消费者线程并启动。为了代码简洁并突出设计思想，代码里省略了 try/catch 检测，我们不纠结一些语法细节。以上便是利用 BlockingQueue 实现生产者消费者模式的代码。虽然代码非常简单，但实际上 ArrayBlockingQueue 已经在背后完成了很多工作，比如队列满了就去阻塞生产者线程，队列有空就去唤醒生产者线程等。

## 如何用 Condition 实现生产者消费者模式：

```
1 public class MyBlockingQueueForCondition {
2
3     private Queue queue;
4     private int max = 16;
5     private ReentrantLock lock = new ReentrantLock();
6     private Condition notEmpty = lock.newCondition();
7     private Condition notFull = lock.newCondition();
8
9
10    public MyBlockingQueueForCondition(int size) {
11        this.max = size;
12        queue = new LinkedList();
13    }
14 }
```



```

15     public void put(Object o) throws InterruptedException {
16         lock.lock();
17         try {
18             while (queue.size() == max) {
19                 notFull.await();
20             }
21             queue.add(o);
22             notEmpty.signalAll();
23         } finally {
24             lock.unlock();
25         }
26     }
27
28     public Object take() throws InterruptedException {
29         lock.lock();
30         try {
31             while (queue.size() == 0) {
32                 notEmpty.await();
33             }
34             Object item = queue.remove();
35             notFull.signalAll();
36             return item;
37         } finally {
38             lock.unlock();
39         }
40     }
41 }

```

BlockingQueue 实现生产者消费者模式看似简单，背后却暗藏玄机，我们在掌握这种方法的基础上仍需要掌握更复杂的实现方法。我们接下来看如何在掌握了 BlockingQueue 的基础上利用 Condition 实现生产者消费者模式，它们背后的实现原理非常相似，相当于我们自己实现一个简易版的。

如代码所示，首先，定义了一个队列变量 queue 并设置最大容量为 16；其次，定义了一个 ReentrantLock 类型的 Lock 锁，并在 Lock 锁的基础上创建两个 Condition，一个是 notEmpty，另一个是 notFull，分别代表队列没有空和没有满的条件；最后，声明了 put 和 take 这两个核心方法。

因为生产者消费者模式通常是面对多线程的场景，需要一定的同步措施保障线程安全，所以在 put 方法中先将 Lock 锁上，然后，在 while 的条件里检测 queue 是不是已经满了，如果已经满了，则调用 notFull 的 await() 阻塞生产者线程并释放 Lock，如果没有满，则往队列放入数据并利用 notEmpty.signalAll() 通知正在等待的所有消费者并唤醒它们。最后在 finally 中利用 lock.unlock() 方法解锁，把 unlock 方法放在 finally 中是一个基本原则，否则可能会产生无法释放锁的情况。

下面再来看 take 方法，take 方法实际上是与 put 方法相互对应的，同样是通过 while 检查队列是否为空，如果为空，消费者开始等待，如果不为空则从队列中获取数据并通知生产者队列有空余位置，最后在 finally 中解锁。

这里需要注意，我们在 take() 方法中使用 while( queue.size() == 0 ) 检查队列状态，而不能用 if( queue.size() == 0 )。为什么呢？大家思考这样一种情况，因为生产者消费者往往是多线程的，我们假设有两个消费者，第一个消费者线程获取数据时，发现队列为空，便进入等待状态；因为第一个线程在等待时会释放 Lock 锁，所以第二个消费者可以进入并执行 if( queue.size() == 0 )，也发现队列为空，于是第二个线程也进入等待；而此时，如果生产者生产了一个数据，便会唤醒两个消费者线程，而两个线程中只有一个线程可以拿到锁，并执行 queue.remove 操作，另外一个线程因为没有拿到锁而卡在被唤醒的地方，而第一个线程执行完操作后会在 finally 中通过 unlock 解锁，而此时第二个线程便可以拿到被第一个线程释放的锁，继续执行操作，也会去调用 queue.remove 操作，然而这个时候队列已经为空了，所以会抛出 NoSuchElementException 异常，这不符合我们的逻辑。而如果用 while 做检查，当第一个消费者被唤醒得到锁并移除数据之后，第二个线程在执行 remove 前仍会进行 while 检查，发现此时依然满足 queue.size() == 0 的条件，就会继续执行 await 方法，避免了获取的数据为 null 或抛出异常的情况。

## 如何用 wait/notify 实现生产者消费者模式

最后我们再来看看使用 wait/notify 实现生产者消费者模式的方法，实际上实现原理和Condition 是非常类似的，它们是兄弟关系：

```
1  class MyBlockingQueue {
2
3      private int maxSize;
4      private LinkedList<Object> storage;
5
6      public MyBlockingQueue(int size) {
7          this.maxSize = size;
8          storage = new LinkedList<>();
9      }
10
11     public synchronized void put() throws InterruptedException {
12         while (storage.size() == maxSize) {
13             wait();
14         }
15         storage.add(new Object());
16         notifyAll();
17     }
18
19     public synchronized void take() throws InterruptedException {
20         while (storage.size() == 0) {
21             wait();
22         }
23         System.out.println(storage.remove());
24         notifyAll();
25     }
26 }
```

如代码所示，最主要的部分仍是 take 与 put 方法，我们先来看 put 方法，put 方法被 synchronized 保护，while 检查队列是否为满，如果不满就往里放入数据并通过 notifyAll() 唤醒其他线程。同样，take 方法也被 synchronized 修饰，while 检查队列是否为空，如果不为空就获取数据并唤醒其他线程。使用这个 MyBlockingQueue 实现的生产者消费者代码如下：

```
1  /**
2   * 描述：      wait形式实现生产者消费者模式
3   */
4  public class WaitStyle {
5
6      public static void main(String[] args) {
7          MyBlockingQueue myBlockingQueue = new MyBlockingQueue(10);
8          Producer producer = new Producer(myBlockingQueue);
9          Consumer consumer = new Consumer(myBlockingQueue);
10         new Thread(producer).start();
11         new Thread(consumer).start();
12     }
13 }
14
15 class Producer implements Runnable {
16
17     private MyBlockingQueue storage;
18
19     public Producer(MyBlockingQueue storage) {
20         this.storage = storage;
21     }
22 }
```

```

23     @Override
24     public void run() {
25         for (int i = 0; i < 100; i++) {
26             try {
27                 storage.put();
28             } catch (InterruptedException e) {
29                 e.printStackTrace();
30             }
31         }
32     }
33 }
34
35 class Consumer implements Runnable {
36
37     private MyBlockingQueue storage;
38
39     public Consumer(MyBlockingQueue storage) {
40         this.storage = storage;
41     }
42
43     @Override
44     public void run() {
45         for (int i = 0; i < 100; i++) {
46             try {
47                 storage.take();
48             } catch (InterruptedException e) {
49                 e.printStackTrace();
50             }
51         }
52     }
53 }

```

以上就是三种实现生产者消费者模式的讲解，其中，第一种 BlockingQueue 模式实现比较简单，但其背后的实现原理在第二种、第三种实现方法中得以体现，第二种、第三种实现方法本质上是我们自己实现了 BlockingQueue 的一些核心逻辑，供生产者与消费者使用。

## 中断总结：

- 当线程sleep和wait期间出现 interrupt，这时会抛出InterruptedException，但线程并没有真正中断，需要以下方法解决：

```

1  //1.方法直接抛异常
2  void subTask2() throws InterruptedException {
3      Thread.sleep(1000);
4  }
5  //2.再次中断
6  private void reInterrupt() {
7      try {
8          Thread.sleep(2000);
9      } catch (InterruptedException e) {
10         Thread.currentThread().interrupt();
11         e.printStackTrace();
12     }
13 }

```

- volatile 这种方法在某些特殊的情况下，比如线程被长时间阻塞的情况，就无法及时感受中断，所以 volatile 是不够全面的停止线程的方法。
- lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断。

synchronized不可中断的意思是等待获取锁的时候不可中断，拿到锁之后可中断，没获取到锁的情况下，中断操作一直不会生效。

- lock和synchronized获取到锁的线程被中断的时候，会抛出中断异常同时释放持有的锁。

## join用法和原理

join的作用是将另一线程加入到当前线程中，并串行执行

### join的作用

```
1 public class JoinTest {
2     public static void main(String[] args) throws Exception{
3         Thread thread = new Thread(new JoinTask());
4         thread.start();
5         thread.join();
6         System.out.println("Main end.");
7     }
8 }
9 class JoinTask implements Runnable {
10     @Override
11     public void run() {
12         try{
13             System.out.println("Join Task Running...");
14             TimeUnit.SECONDS.sleep(3);
15             System.out.println("Join Task End.");
16         }catch (Exception ex) {
17             ex.printStackTrace();
18         }
19     }
20 }
```

### join的原理

join底层基于wait/notify实现，下面看一下join方法的源码：

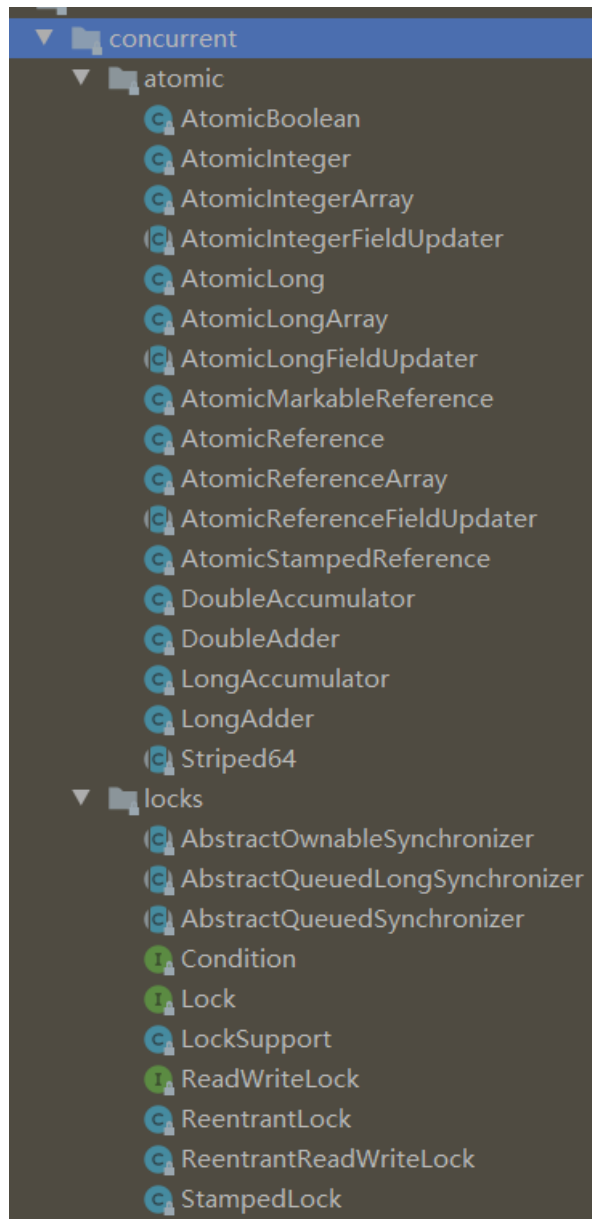
```
1     long base = System.currentTimeMillis();
2     long now = 0;
3
4     if (millis < 0) {
5         throw new IllegalArgumentException("timeout value is negative");
6     }
7
8     if (millis == 0) {
9         while (isAlive()) {
10             wait(0);
11         }
12     }
13 }
```

```


























11     }
12     } else {
13         while (isAlive()) {
14             long delay = millis - now;
15             if (delay <= 0) {
16                 break;
17             }
18             wait(delay);
19             now = System.currentTimeMillis() - base;
20         }
21     }

```

当在线程thread上调用join方法时，就相当于在main线程上调用了thread.wait()方法，当被join的线程执行完成时，会自动在thread对象上调用notify方法，也就是唤醒main线程。



- AbstractExecutorService
- ArrayBlockingQueue
- BlockingDeque
- BlockingQueue
- BrokenBarrierException
- Callable
- CancellationException
- CompletableFuture
- CompletionException
- CompletionService
- CompletionStage
- ConcurrentHashMap
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentMap
- ConcurrentNavigableMap
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- CountDownLatch
- CountedCompleter
- CyclicBarrier
- Delayed
- DelayQueue
- Exchanger
- ExecutionException
- Executor
- ExecutorCompletionService
- Executors
- ExecutorService
- ForkJoinPool
- ForkJoinTask

-  ForkJoinWorkerThread
-  Future
-  FutureTask
-  LinkedBlockingDeque
-  LinkedBlockingQueue
-  LinkedTransferQueue
-  Phaser
-  PriorityBlockingQueue
-  RecursiveAction
-  RecursiveTask
-  RejectedExecutionException
-  RejectedExecutionHandler
-  RunnableFuture
-  RunnableScheduledFuture
-  ScheduledExecutorService
-  ScheduledFuture
-  ScheduledThreadPoolExecutor
-  Semaphore
-  SynchronousQueue
-  ThreadFactory
-  ThreadLocalRandom
-  ThreadPoolExecutor
-  TimeoutException
-  TimeUnit
-  TransferQueue