<http://www.importnew.com/21136.html>

# 前言

多线程并发编程是Java编程中重要的一块内容，也是面试重点覆盖区域，所以学好多线程并发编程对我们来说极其重要，下面跟我一起开启本次的学习之旅吧。

# 正文

## 线程与进程

1 线程：进程中负责程序执行的执行单元  
线程本身依靠程序进行运行  
线程是程序中的顺序控制流，只能使用分配给程序的资源和环境

2 进程：执行中的程序  
一个进程至少包含一个线程

3 单线程：程序中只存在一个线程，实际上主方法就是一个主线程

4 多线程：在一个程序中运行多个任务  
目的是更好地使用CPU资源

## 线程的实现

### 继承Thread类

在java.lang包中定义, 继承Thread类必须重写run()方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class MyThread extends Thread{      private static int num = 0;        public MyThread(){          num++;      }        @Override      public void run() {          System.out.println("主动创建的第"+num+"个线程");      }  } |

创建好了自己的线程类之后，就可以创建线程对象了，然后通过start()方法去启动线程。注意，不是调用run()方法启动线程，run方法中只是定义需要执行的任务，如果调用run方法，即相当于在主线程中执行run方法，跟普通的方法调用没有任何区别，此时并不会创建一个新的线程来执行定义的任务。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Test {      public static void main(String[] args)  {          MyThread thread = new MyThread();          thread.start();      }  }  class MyThread extends Thread{      private static int num = 0;      public MyThread(){          num++;      }      @Override      public void run() {          System.out.println("主动创建的第"+num+"个线程");      }  } |

在上面代码中，通过调用start()方法，就会创建一个新的线程了。为了分清start()方法调用和run()方法调用的区别，请看下面一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class Test {      public static void main(String[] args)  {          System.out.println("主线程ID:"+Thread.currentThread().getId());          MyThread thread1 = new MyThread("thread1");          thread1.start();          MyThread thread2 = new MyThread("thread2");          thread2.run();      }  }    class MyThread extends Thread{      private String name;        public MyThread(String name){          this.name = name;      }        @Override      public void run() {          System.out.println("name:"+name+" 子线程ID:"+Thread.currentThread().getId());      }  } |

运行结果：

[](http://www.importnew.com/21136.html/1-24)

从输出结果可以得出以下结论：

1）thread1和thread2的线程ID不同，thread2和主线程ID相同，说明通过run方法调用并不会创建新的线程，而是在主线程中直接运行run方法，跟普通的方法调用没有任何区别；

2）虽然thread1的start方法调用在thread2的run方法前面调用，但是先输出的是thread2的run方法调用的相关信息，说明新线程创建的过程不会阻塞主线程的后续执行。

### 实现Runnable接口

在Java中创建线程除了继承Thread类之外，还可以通过实现Runnable接口来实现类似的功能。实现Runnable接口必须重写其run方法。  
下面是一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Test {      public static void main(String[] args)  {          System.out.println("主线程ID："+Thread.currentThread().getId());          MyRunnable runnable = new MyRunnable();          Thread thread = new Thread(runnable);          thread.start();      }  }  class MyRunnable implements Runnable{      public MyRunnable() {      }        @Override      public void run() {          System.out.println("子线程ID："+Thread.currentThread().getId());      }  } |

Runnable的中文意思是“任务”，顾名思义，通过实现Runnable接口，我们定义了一个子任务，然后将子任务交由Thread去执行。注意，这种方式必须将Runnable作为Thread类的参数，然后通过Thread的start方法来创建一个新线程来执行该子任务。如果调用Runnable的run方法的话，是不会创建新线程的，这根普通的方法调用没有任何区别。

事实上，查看Thread类的实现源代码会发现Thread类是实现了Runnable接口的。

在Java中，这2种方式都可以用来创建线程去执行子任务，具体选择哪一种方式要看自己的需求。直接继承Thread类的话，可能比实现Runnable接口看起来更加简洁，但是由于Java只允许单继承，所以如果自定义类需要继承其他类，则只能选择实现Runnable接口。

### 使用ExecutorService、Callable、Future实现有返回结果的多线程

多线程后续会学到，这里暂时先知道一下有这种方法即可。

ExecutorService、Callable、Future这个对象实际上都是属于Executor框架中的功能类。想要详细了解Executor框架的可以访问[http://www.javaeye.com/topic/366591](http://www.javaeye.com/topic/366591" \t "_blank) ，这里面对该框架做了很详细的解释。返回结果的线程是在JDK1.5中引入的新特征，确实很实用，有了这种特征我就不需要再为了得到返回值而大费周折了，而且即便实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。下面提供了一个完整的有返回结果的多线程测试例子，在JDK1.5下验证过没问题可以直接使用。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | /\*\*  \* 有返回值的线程  \*/  @SuppressWarnings("unchecked")  public class Test {  public static void main(String[] args) throws ExecutionException,      InterruptedException {     System.out.println("----程序开始运行----");     Date date1 = new Date();       int taskSize = 5;     // 创建一个线程池     ExecutorService pool = Executors.newFixedThreadPool(taskSize);     // 创建多个有返回值的任务     List<Future> list = new ArrayList<Future>();     for (int i = 0; i < taskSize; i++) {      Callable c = new MyCallable(i + " ");      // 执行任务并获取Future对象      Future f = pool.submit(c);      // System.out.println(">>>" + f.get().toString());      list.add(f);     }     // 关闭线程池     pool.shutdown();       // 获取所有并发任务的运行结果     for (Future f : list) {      // 从Future对象上获取任务的返回值，并输出到控制台      System.out.println(">>>" + f.get().toString());     }       Date date2 = new Date();     System.out.println("----程序结束运行----，程序运行时间【"       + (date2.getTime() - date1.getTime()) + "毫秒】");  }  }    class MyCallable implements Callable<Object> {  private String taskNum;    MyCallable(String taskNum) {     this.taskNum = taskNum;  }    public Object call() throws Exception {     System.out.println(">>>" + taskNum + "任务启动");     Date dateTmp1 = new Date();     Thread.sleep(1000);     Date dateTmp2 = new Date();     long time = dateTmp2.getTime() - dateTmp1.getTime();     System.out.println(">>>" + taskNum + "任务终止");     return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";  }  } |

代码说明：  
上述代码中Executors类，提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。  
public static ExecutorService newFixedThreadPool(int nThreads)  
创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()  
创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()  
创建一个单线程化的Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  
创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

## 线程的状态

在正式学习Thread类中的具体方法之前，我们先来了解一下线程有哪些状态，这个将会有助于后面对Thread类中的方法的理解。

* 创建（new）状态: 准备好了一个多线程的对象
* 就绪（runnable）状态: 调用了start()方法, 等待CPU进行调度
* 运行（running）状态: 执行run()方法
* 阻塞（blocked）状态: 暂时停止执行, 可能将资源交给其它线程使用
* 终止（dead）状态: 线程销毁

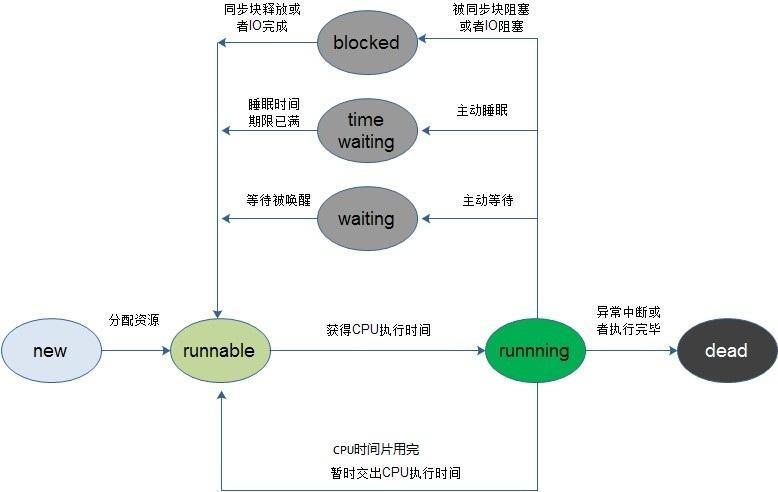
当需要新起一个线程来执行某个子任务时，就创建了一个线程。但是线程创建之后，不会立即进入就绪状态，因为线程的运行需要一些条件（比如内存资源，在前面的JVM内存区域划分一篇博文中知道程序计数器、Java栈、本地方法栈都是线程私有的，所以需要为线程分配一定的内存空间），只有线程运行需要的所有条件满足了，才进入就绪状态。

当线程进入就绪状态后，不代表立刻就能获取CPU执行时间，也许此时CPU正在执行其他的事情，因此它要等待。当得到CPU执行时间之后，线程便真正进入运行状态。

线程在运行状态过程中，可能有多个原因导致当前线程不继续运行下去，比如用户主动让线程睡眠（睡眠一定的时间之后再重新执行）、用户主动让线程等待，或者被同步块给阻塞，此时就对应着多个状态：time waiting（睡眠或等待一定的事件）、waiting（等待被唤醒）、blocked（阻塞）。

当由于突然中断或者子任务执行完毕，线程就会被消亡。

下面这副图描述了线程从创建到消亡之间的状态：

[](http://www.importnew.com/21136.html/2-19)

在有些教程上将blocked、waiting、time waiting统称为阻塞状态，这个也是可以的，只不过这里我想将线程的状态和Java中的方法调用联系起来，所以将waiting和time waiting两个状态分离出来。

注:sleep和wait的区别:

* *sleep是Thread类的方法,wait是Object类中定义的方法.*
* *Thread.sleep不会导致锁行为的改变, 如果当前线程是拥有锁的, 那么Thread.sleep不会让线程释放锁.*
* *Thread.sleep和Object.wait都会暂停当前的线程. OS会将执行时间分配给其它线程. 区别是, 调用wait后, 需要别的线程执行notify/notifyAll才能够重新获得CPU执行时间.*

## 上下文切换

对于单核CPU来说（对于多核CPU，此处就理解为一个核），CPU在一个时刻只能运行一个线程，当在运行一个线程的过程中转去运行另外一个线程，这个叫做线程上下文切换（对于进程也是类似）。

由于可能当前线程的任务并没有执行完毕，所以在切换时需要保存线程的运行状态，以便下次重新切换回来时能够继续切换之前的状态运行。举个简单的例子：比如一个线程A正在读取一个文件的内容，正读到文件的一半，此时需要暂停线程A，转去执行线程B，当再次切换回来执行线程A的时候，我们不希望线程A又从文件的开头来读取。

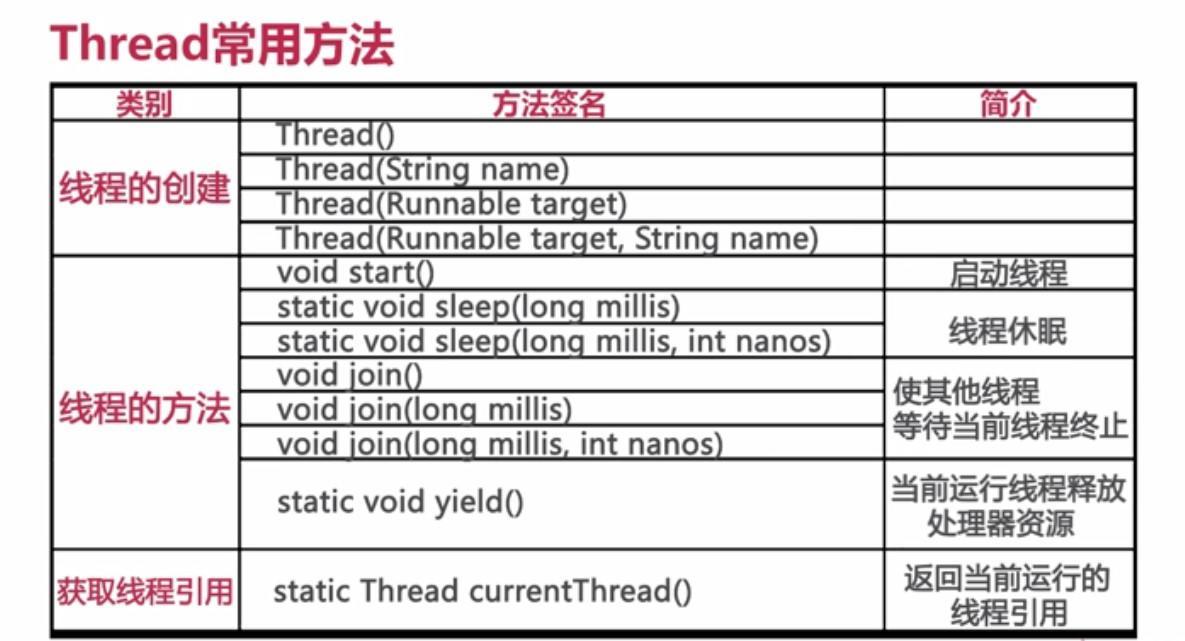
因此需要记录线程A的运行状态，那么会记录哪些数据呢？因为下次恢复时需要知道在这之前当前线程已经执行到哪条指令了，所以需要记录程序计数器的值，另外比如说线程正在进行某个计算的时候被挂起了，那么下次继续执行的时候需要知道之前挂起时变量的值时多少，因此需要记录CPU寄存器的状态。所以一般来说，线程上下文切换过程中会记录程序计数器、CPU寄存器状态等数据。

说简单点的：对于线程的上下文切换实际上就是 **存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行**。

虽然多线程可以使得任务执行的效率得到提升，但是由于在线程切换时同样会带来一定的开销代价，并且多个线程会导致系统资源占用的增加，所以在进行多线程编程时要注意这些因素。

## 线程的常用方法

| **编号** | **方法** | **说明** |
| --- | --- | --- |
| 1 | public void start() | 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。 |
| 2 | public void run() | 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| 3 | public final void setName(String name) | 改变线程名称，使之与参数 name 相同。 |
| 4 | public final void setPriority(int priority) | 更改线程的优先级。 |
| 5 | public final void setDaemon(boolean on) | 将该线程标记为守护线程或用户线程。 |
| 6 | public final void join(long millisec) | 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | public void interrupt() | 中断线程。 |
| 8 | public final boolean isAlive() | 测试线程是否处于活动状态。 |
| 9 | public static void yield() | 暂停当前正在执行的线程对象，并执行其他线程。 |
| 10 | public static void sleep(long millisec) | 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 11 | public static Thread currentThread() | 返回对当前正在执行的线程对象的引用。 |

[](http://www.importnew.com/21136.html/3-15)

### 静态方法

### currentThread()方法

currentThread()方法可以返回代码段正在被哪个线程调用的信息。

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Run1{      public static void main(String[] args){      System.out.println(Thread.currentThread().getName());      }  } |

### sleep()方法

方法sleep()的作用是在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行）。这个“正在执行的线程”是指this.currentThread()返回的线程。

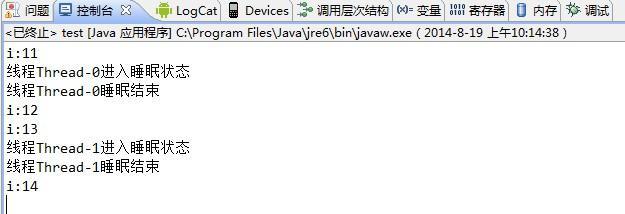
sleep方法有两个重载版本：

|  |  |
| --- | --- |
| 1  2 | sleep(long millis)     //参数为毫秒  sleep(long millis,int nanoseconds)    //第一参数为毫秒，第二个参数为纳秒 |

sleep相当于让线程睡眠，交出CPU，让CPU去执行其他的任务。  
但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。看下面这个例子就清楚了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class Test {        private int i = 10;      private Object object = new Object();        public static void main(String[] args) throws IOException  {          Test test = new Test();          MyThread thread1 = test.new MyThread();          MyThread thread2 = test.new MyThread();          thread1.start();          thread2.start();      }        class MyThread extends Thread{          @Override          public void run() {              synchronized (object) {                  i++;                  System.out.println("i:"+i);                  try {                      System.out.println("线程"+Thread.currentThread().getName()+"进入睡眠状态");                      Thread.currentThread().sleep(10000);                  } catch (InterruptedException e) {                      // TODO: handle exception                  }                  System.out.println("线程"+Thread.currentThread().getName()+"睡眠结束");                  i++;                  System.out.println("i:"+i);              }          }      }  } |

输出结果：

[](http://www.importnew.com/21136.html/4-13)

从上面输出结果可以看出，当Thread-0进入睡眠状态之后，Thread-1并没有去执行具体的任务。只有当Thread-0执行完之后，此时Thread-0释放了对象锁，Thread-1才开始执行。

注意，如果调用了sleep方法，必须捕获InterruptedException异常或者将该异常向上层抛出。当线程睡眠时间满后，不一定会立即得到执行，因为此时可能CPU正在执行其他的任务。所以说调用sleep方法相当于让线程进入阻塞状态。

### yield()方法

调用yield方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。

注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。  
代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class MyThread  extends Thread{      @Override      public void run() {          long beginTime=System.currentTimeMillis();          int count=0;          for (int i=0;i<50000000;i++){              count=count+(i+1);              //Thread.yield();          }          long endTime=System.currentTimeMillis();          System.out.println("用时："+(endTime-beginTime)+" 毫秒！");      }  }    public class Run {      public static void main(String[] args) {          MyThread t= new MyThread();          t.start();      }  } |

执行结果：

|  |  |
| --- | --- |
| 1 | 用时：3 毫秒！ |

如果将 //Thread.yield();的注释去掉，执行结果如下：

|  |  |
| --- | --- |
| 1 | 用时：16080 毫秒！ |

### 对象方法

#### start()方法

start()用来启动一个线程，当调用start方法后，系统才会开启一个新的线程来执行用户定义的子任务，在这个过程中，会为相应的线程分配需要的资源。

#### run()方法

run()方法是不需要用户来调用的，当通过start方法启动一个线程之后，当线程获得了CPU执行时间，便进入run方法体去执行具体的任务。注意，继承Thread类必须重写run方法，在run方法中定义具体要执行的任务。

#### getId()

getId()的作用是取得线程的唯一标识  
代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Test {      public static void main(String[] args) {          Thread t= Thread.currentThread();          System.out.println(t.getName()+" "+t.getId());      }  } |

输出：

|  |  |
| --- | --- |
| 1 | main 1 |

#### isAlive()方法

方法isAlive()的功能是判断当前线程是否处于活动状态  
代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class MyThread  extends Thread{      @Override      public void run() {          System.out.println("run="+this.isAlive());      }  }  public class RunTest {      public static void main(String[] args) throws InterruptedException {          MyThread myThread=new MyThread();          System.out.println("begin =="+myThread.isAlive());          myThread.start();          System.out.println("end =="+myThread.isAlive());      }  } |

程序运行结果：

|  |  |
| --- | --- |
| 1  2  3 | begin ==false  run=true  end ==false |

方法isAlive()的作用是测试线程是否偶处于活动状态。什么是活动状态呢？活动状态就是线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是“存活”的。  
有个需要注意的地方

|  |  |
| --- | --- |
| 1 | System.out.println("end =="+myThread.isAlive()); |

虽然上面的实例中打印的值是true,但此值是不确定的。打印true值是因为myThread线程还未执行完毕，所以输出true。如果代码改成下面这样，加了个sleep休眠：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static void main(String[] args) throws InterruptedException {          MyThread myThread=new MyThread();          System.out.println("begin =="+myThread.isAlive());          myThread.start();          Thread.sleep(1000);          System.out.println("end =="+myThread.isAlive());      } |

则上述代码运行的结果输出为false,因为mythread对象已经在1秒之内执行完毕。

#### join()方法

在很多情况下，主线程创建并启动了线程，如果子线程中药进行大量耗时运算，主线程往往将早于子线程结束之前结束。这时，如果主线程想等待子线程执行完成之后再结束，比如子线程处理一个数据，主线程要取得这个数据中的值，就要用到join()方法了。方法join()的作用是等待线程对象销毁。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class Thread4 extends Thread{      public Thread4(String name) {          super(name);      }      public void run() {          for (int i = 0; i < 5; i++) {              System.out.println(getName() + "  " + i);          }      }      public static void main(String[] args) throws InterruptedException {          // 启动子进程          new Thread4("new thread").start();          for (int i = 0; i < 10; i++) {              if (i == 5) {                  Thread4 th = new Thread4("joined thread");                  th.start();                  th.join();              }              System.out.println(Thread.currentThread().getName() + "  " + i);          }      }  } |

执行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | main  0  main  1  main  2  main  3  main  4  new thread  0  new thread  1  new thread  2  new thread  3  new thread  4  joined thread  0  joined thread  1  joined thread  2  joined thread  3  joined thread  4  main  5  main  6  main  7  main  8  main  9 |

由上可以看出main主线程等待joined thread线程先执行完了才结束的。如果把th.join()这行注释掉，运行结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | main  0  main  1  main  2  main  3  main  4  main  5  main  6  main  7  main  8  main  9  new thread  0  new thread  1  new thread  2  new thread  3  new thread  4  joined thread  0  joined thread  1  joined thread  2  joined thread  3  joined thread  4 |

#### getName和setName

用来得到或者设置线程名称。

#### getPriority和setPriority

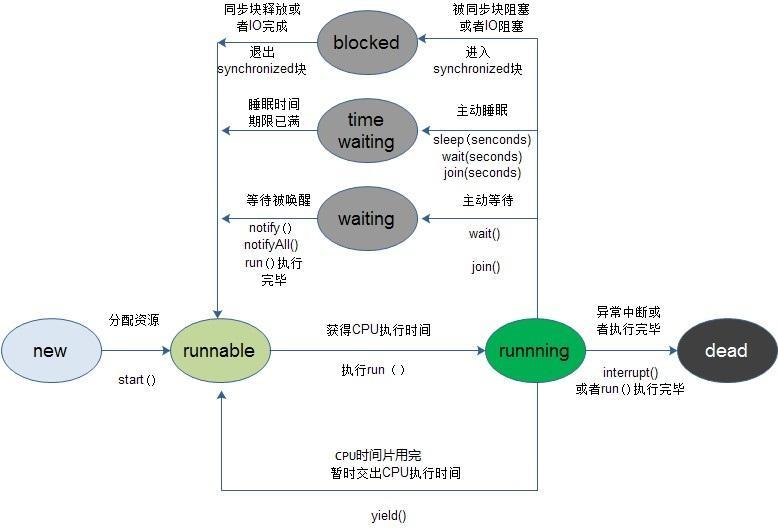
用来获取和设置线程优先级。

#### setDaemon和isDaemon

用来设置线程是否成为守护线程和判断线程是否是守护线程。

守护线程和用户线程的区别在于：守护线程依赖于创建它的线程，而用户线程则不依赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。

**在上面已经说到了Thread类中的大部分方法，那么Thread类中的方法调用到底会引起线程状态发生怎样的变化呢？下面一幅图就是在上面的图上进行改进而来的：**

[](http://www.importnew.com/21136.html/5-11)

## 停止线程

停止线程是在多线程开发时很重要的技术点，掌握此技术可以对线程的停止进行有效的处理。  
停止一个线程可以使用Thread.stop()方法，但最好不用它。该方法是不安全的，已被弃用。  
在Java中有以下3种方法可以终止正在运行的线程：

* 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止
* 使用stop方法强行终止线程，但是不推荐使用这个方法，因为stop和suspend及resume一样，都是作废过期的方法，使用他们可能产生不可预料的结果。
* 使用interrupt方法中断线程，但这个不会终止一个正在运行的线程，还需要加入一个判断才可以完成线程的停止。

## 暂停线程

interrupt()方法

## 线程的优先级

在操作系统中，线程可以划分优先级，优先级较高的线程得到的CPU资源较多，也就是CPU优先执行优先级较高的线程对象中的任务。  
设置线程优先级有助于帮“线程规划器”确定在下一次选择哪一个线程来优先执行。  
设置线程的优先级使用setPriority()方法，此方法在JDK的源码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public final void setPriority(int newPriority) {          ThreadGroup g;          checkAccess();          if (newPriority > MAX\_PRIORITY || newPriority < MIN\_PRIORITY) {              throw new IllegalArgumentException();          }          if((g = getThreadGroup()) != null) {              if (newPriority > g.getMaxPriority()) {                  newPriority = g.getMaxPriority();              }              setPriority0(priority = newPriority);          }      } |

在Java中，线程的优先级分为1~10这10个等级，如果小于1或大于10，则JDK抛出异常throw new IllegalArgumentException()。  
JDK中使用3个常量来预置定义优先级的值，代码如下：

|  |  |
| --- | --- |
| 1  2  3 | public final static int MIN\_PRIORITY = 1;  public final static int NORM\_PRIORITY = 5;  public final static int MAX\_PRIORITY = 10; |

线程优先级特性：

* 继承性  
  比如A线程启动B线程，则B线程的优先级与A是一样的。
* 规则性  
  高优先级的线程总是大部分先执行完，但不代表高优先级线程全部先执行完。
* 随机性  
  优先级较高的线程不一定每一次都先执行完。

## 守护线程

在Java线程中有两种线程，一种是User Thread（用户线程），另一种是Daemon Thread(守护线程)。  
Daemon的作用是为其他线程的运行提供服务，比如说GC线程。其实User Thread线程和Daemon Thread守护线程本质上来说去没啥区别的，唯一的区别之处就在虚拟机的离开：如果User Thread全部撤离，那么Daemon Thread也就没啥线程好服务的了，所以虚拟机也就退出了。

守护线程并非虚拟机内部可以提供，用户也可以自行的设定守护线程，方法：public final void setDaemon(boolean on) ；但是有几点需要注意：

* thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。 （备注：这点与守护进程有着明显的区别，守护进程是创建后，让进程摆脱原会话的控制+让进程摆脱原进程组的控制+让进程摆脱原控制终端的控制；所以说寄托于虚拟机的语言机制跟系统级语言有着本质上面的区别）
* 在Daemon线程中产生的新线程也是Daemon的。 （这一点又是有着本质的区别了：守护进程fork()出来的子进程不再是守护进程，尽管它把父进程的进程相关信息复制过去了，但是子进程的进程的父进程不是init进程，所谓的守护进程本质上说就是“父进程挂掉，init收养，然后文件0,1,2都是/dev/null，当前目录到/”）
* 不是所有的应用都可以分配给Daemon线程来进行服务，比如读写操作或者计算逻辑。因为在Daemon Thread还没来的及进行操作时，虚拟机可能已经退出了。

### 同步与死锁

1. 同步代码块  
   在代码块上加上”synchronized”关键字，则此代码块就称为同步代码块
2. 同步代码块格式

|  |  |
| --- | --- |
| 1  2  3 | synchronized(同步对象){   需要同步的代码块;  } |

1. 同步方法  
   除了代码块可以同步，方法也是可以同步的
2. 方法同步格式

|  |  |
| --- | --- |
| 1 | synchronized void 方法名称(){} |

1. synchronized后续会单独来学习。(●’◡’●)

## 面试题

**线程和进程有什么区别？**  
答：一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

**如何在Java中实现线程？**  
答：  
创建线程有两种方式：  
一、继承 Thread 类，扩展线程。  
二、实现 Runnable 接口。

**启动一个线程是调用run()还是start()方法？**  
答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

**Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?**  
答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**线程的sleep()方法和yield()方法有什么区别？**  
答：  
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。

**请说出与线程同步以及线程调度相关的方法。**  
答：

* wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
* sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
* notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
* notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

# 总结

以上就是多线程的一些基础概念，可能总结的不够仔细，多多包涵。后续会针对一些比较重要的知识点单独列出来总结。学好多线程是拿高薪的基础，小伙伴一起加油吧！

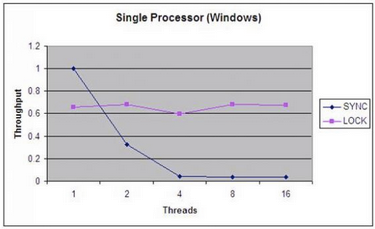
# 参考

该文为本人学习的笔记，方便以后自己跳槽前复习。参考网上各大帖子，取其精华整合自己的理解而成。还有，关注我个人主页的公众号，里面电子书资源有《Java多线程编程核心技术》以及《JAVA并发编程实践》高清版，需要的小伙伴自己取。

《Java多线程编程核心技术》  
《JAVA并发编程实践》  
[Java并发编程：Thread类的使用](http://www.cnblogs.com/dolphin0520/p/3920357.html)  
[关于Java并发编程的总结和思考](http://blog.csdn.net/jackfrued/article/details/44499227)  
[JAVA多线程实现的三种方式](http://blog.csdn.net/aboy123/article/details/38307539)

# 整理的思维导图

个人整理的多线程基础的思维导图,导出的图片无法查看备注的一些信息，所以需要源文件的童鞋可以关注我个人主页上的公众号，回复**多线程基础**即可获取源文件。

[](http://www.importnew.com/21136.html/6-10)

一直觉得自己写的不是技术，而是情怀，一篇篇文章是自己这一路走来的痕迹。靠专业技能的成功是最具可复制性的，希望我的这条路能让你少走弯路，希望我能帮你抹去知识的蒙尘，希望我能帮你理清知识的脉络，希望未来技术之巅上有你也有我。