### 创建线程

* 通过实现 Runnable 接口

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

...

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

* 通过继承 Thread 类本身

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

ThreadDemo( String name) {

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

...

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

* 通过 Callable 和 Future 创建线程

public class CallableThreadTest implements Callable<Integer> {

public static void main(String[] args)

{

CallableThreadTest ctt = new CallableThreadTest();

FutureTask<Integer> ft = new FutureTask<>(ctt);

for(int i = 0;i < 100;i++)

{

System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);

if(i==20)

{

new Thread(ft,"有返回值的线程").start();

}

}

try

{

System.out.println("子线程的返回值："+ft.get());

} catch (InterruptedException e)

{

e.printStackTrace();

} catch (ExecutionException e)

{

e.printStackTrace();

}

}

@Override

public Integer call() throws Exception

{

...

return i;

}

}

**三种方式的对比：**

1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。

2. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。

### 多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU 花费在上下文的切换的时间将多于执行程序的时间！

#### Thread方法

**常用重要方法**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 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。 |

**Thread类的静态方法**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **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。 |
| 3 | **public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

#### Synchronized实现线程同步

#### 使用synchronized时释放锁总结

1、当前线程的同步方法、代码块执行结束的时候释放

2、当前线程在同步方法、同步代码块中遇到break 、 return 终于该代码块或者方法的时候释放。

3、出现未处理的error或者exception导致异常结束的时候释放

4、程序执行了 同步对象 wait 方法 ，当前线程暂停，释放锁

**在以下两种情况不会释放锁：**

1、代码块中使用了 Thread.sleep()  Thread.yield() 这些方法暂停线程的执行，不会释放。

2、线程执行同步代码块时，其他线程调用 suspend 方法将该线程挂起，该线程不会释放锁 ，所以我们应该避免使用 suspend 和 resume 来控制线程 。

### 线程生命周期

线程的生命周期

线程的生命周期包含5个阶段，包括：新建、就绪、运行、阻塞、销毁。

**新建**：就是刚使用new方法，new出来的线程；

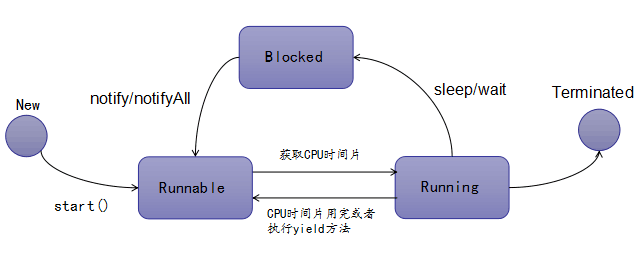
**就绪**：就是调用的线程的start()方法后，这时候线程处于等待CPU分配资源阶段，谁先抢的CPU资源，谁开始执行;

**运行**：当就绪的线程被调度并获得CPU资源时，便进入运行状态，run方法定义了线程的操作和功能;

**阻塞**：在运行状态的时候，可能因为某些原因导致运行状态的线程变成了阻塞状态，比如sleep()、wait()之后线程就处于了阻塞状态，这个时候需要其他机制将处于阻塞状态的线程唤醒，比如调用notify或者notifyAll()方法。唤醒的线程不会立刻执行run方法，它们要再次等待CPU分配资源进入运行状态;

**销毁**：如果线程正常执行完毕后或线程被提前强制性的终止或出现异常导致结束，那么线程就要被销毁，释放资源;

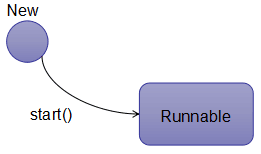
**完整的生命周期图如下**：



#### 新建状态

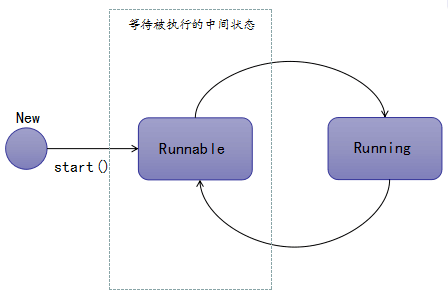
Thread t1 = new Thread();

以上代码代表Thread被创建，而在操作系统层面，真正的线程还没有被创建。只有当我们调用了 start() 方法之后，该线程才会被创建出来，进入Runnable状态。只有当我们调用了 start() 方法之后，该线程才会被创建出来。

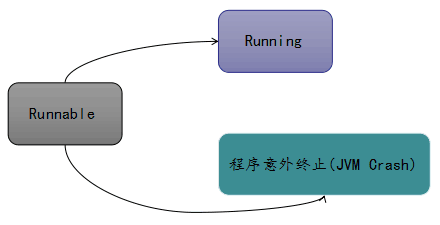


#### 就绪状态

调用start()方法后，JVM 进程会去创建一个新的线程，而此线程不会马上被 CPU 调度运行，进入Running状态，这里会有一个中间状态，就是Runnable状态，你可以理解为等待被 CPU 调度的状态



那么处于Runnable状态的线程能发生哪些状态转变？

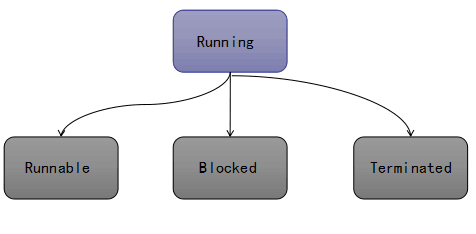


Runnable状态的线程无法直接进入Blocked状态和Terminated状态的。只有处在Running状态的线程，换句话说，只有获得CPU调度执行权的线程才有资格进入Blocked状态和Terminated状态，Runnable状态的线程要么能被转换成Running状态，要么被意外终止。

#### 运行状态

当CPU调度发生，并从任务队列中选中了某个Runnable线程时，该线程会进入Running执行状态，并且开始调用run()方法中逻辑代码。

那么处于Running状态的线程能发生哪些状态转变？



* 被转换成Terminated状态，比如调用 stop() 方法;
* 被转换成Blocked状态，比如调用了sleep, wait 方法被加入 waitSet 中；
* 被转换成Blocked状态，如进行 IO 阻塞操作，如查询数据库进入阻塞状态；
* 被转换成Blocked状态，比如获取某个锁的释放，而被加入该锁的阻塞队列中；
* 该线程的时间片用完，CPU 再次调度，进入Runnable状态；
* 线程主动调用 yield 方法，让出 CPU 资源，进入Runnable状态

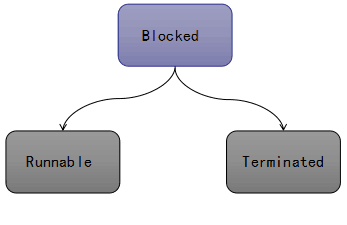
#### 阻塞状态

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。

**阻塞可以分为三种**

* 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态
* 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
* 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。

**Blocked状态的线程能够发生哪些状态改变？**



* 被转换成Terminated状态，比如调用 stop() 方法，或者是 JVM 意外 Crash;
* 被转换成Runnable状态，阻塞时间结束，比如读取到了数据库的数据后；
* 完成了指定时间的休眠，进入到Runnable状态；
* 正在wait中的线程，被其他线程调用notify/notifyAll方法唤醒，进入到Runnable状态；
* 线程获取到了想要的锁资源，进入Runnable状态；
* 线程在阻塞状态下被打断，如其他线程调用了interrupt方法，进入到Runnable状态；

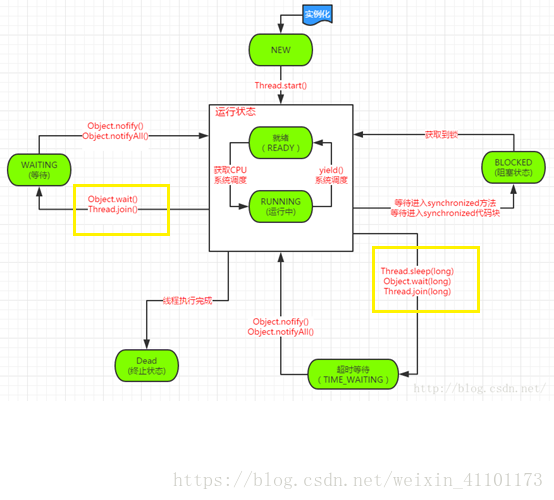
#### 终止状态

一旦线程进入了Terminated状态，就意味着这个线程生命的终结，哪些情况下，线程会进入到Terminated状态呢？

* 线程正常运行结束，生命周期结束；
* 线程运行过程中出现意外错误；

JVM 异常结束，所有的线程生命周期均被结束。

#### Sleep和wait方法的区别



**Sleep和wait的阻塞场景图**

①sleep()实现线程阻塞的方法，我们称之为“**线程睡眠**”，方式是超时等待，怎么理解？就是sleep()通过传入“**睡眠时间**”作为方法的参数，时间一到就从“睡眠”中“醒来”；

②**wait()方法实现线程阻塞的方法**，我们称之为“**线程等待**”，方式有两种：

1）和sleep()方法一样，通过传入“睡眠时间”作为参数，时间到了就“醒了”；

1. 不传入时间，进行一次“无限期的等待”，只用通过notify()方法来“唤醒”。

通过上面图的展示和笔者的相关描述，相信读者你已经有几分清楚了，sleep()和wait()方法的区别之一，就是实现线程阻塞的方式不一样。

那么sleep()和wait()二者还有一个很大的区别就是，二者“**是否释放同步锁**”不一样。我们都知道，多线程开发中，为了实现不同线程间的同步会采用同步锁的方式——synchronized即在线程使用一个资源时为其加锁，这样其他的线程便不能访问那个资源了，直到解锁后才可以访问。感兴趣的读者可以参考笔者之前文章《利用synchronized实现线程同步》里面详细介绍了多线程开发中利用synchronized实现线程同步的方式。

而使用sleep()和wait()两种方法对于“CPU执行权”和“同步锁”的方式不同：

**①sleep()释放CPU执行权，但不释放同步锁；**

**②wait()释放CPU执行权，也释放同步锁，使得其他线程可以使用同步控制块或**者方法。

以上，就是sleep()和wait()方法的**两个关键性区别**。

总结：综上我们利用下表展示sleep()和wait()的所有区别：