# Java多线程

## Java线程状态



**新建状态（New）：**当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();

**就绪状态（Runnable）：**当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

**运行状态（Running）：**当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

**阻塞状态（Blocked）：**处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，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()或发出了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、或者I/O处理完毕时，线程重新转入就绪状态。

**死亡状态（Dead）：**线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## Java线程创建和启动

1.继承Thread类，重写该类的run()方法

2.实现Runnable接口，并重写该接口的run()方法，该run()方法同样是线程执行体，创建Runnable实现类的实例，并以此实例作为Thread类的target来创建Thread对象，该Thread对象才是真正的线程对象。

Ps: Thread和Runnable之间到底是什么关系?

Thread thread = new MyThread(myRunnable);

那么这种方式可以顺利创建出一个新的线程么？答案是肯定的。至于此时的线程执行体到底是MyRunnable接口中的run()方法还是MyThread类中的run()方法呢？通过输出我们知道线程执行体是MyThread类中的run()方法。其实原因很简单，因为Thread类本身也是实现了Runnable接口，而run()方法最先是在Runnable接口中定义的方法。由于多态的存在，根本就没有执行到Thread类中的run()方法，而是直接先执行了运行时类型即MyThread类中的run()方法。

3.使用Callable和Future接口创建线程。具体是创建Callable接口的实现类，并实现call()方法。并使用FutureTask类来包装Callable实现类的对象，且以此FutureTask对象作为Thread对象的target来创建线程。

Ps:FutureTask类实际上是同时实现了Runnable和Future接口，由此才使得其具有Future和Runnable双重特性。通过Runnable特性，可以作为Thread对象的target，而Future特性，使得其可以取得新创建线程中的call()方法的返回值。

## 同步及死锁

### 同步方式

1.同步方法：即有synchronized关键字修饰的方法。  由于java的每个对象都有一个内置锁，当用此关键字修饰方法时， 内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。代码如： public synchronized void save(){} 注： synchronized关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类

2.同步代码块

即有synchronized关键字修饰的语句块。

被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

代码如：

synchronized(object){

}

注：同步是一种高开销的操作，因此应该尽量减少同步的内容。

通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。

1. 使用特殊域变量(volatile)实现线程同步

a.volatile关键字为域变量的访问提供了一种免锁机制，

b.使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新，

c.因此每次使用该域就要重新计算，而不是使用寄存器中的值

d.volatile不会提供任何原子操作，它也不能用来修饰final类型的变量

ps：自增操作不是原子性操作

4.使用重入锁

ReentrantLock() : 创建一个ReentrantLock实例

lock() : 获得锁   
unlock() : 释放锁 ，通常在finally代码释放锁

5. ThreadLocal

如果使用ThreadLocal管理变量，则每一个使用该变量的线程都获得该变量的副本，  
    副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

**6.使用阻塞队列实现线程同步**

**LinkedBlockingQueue 类常用方法**   
    LinkedBlockingQueue() : 创建一个容量为Integer.MAX\_VALUE的LinkedBlockingQueue   
    put(E e) : 在队尾添加一个元素，如果队列满则阻塞   
    size() : 返回队列中的元素个数   
    take() : 移除并返回队头元素，如果队列空则阻塞

7. **使用原子变量实现线程同步**

原子操作就是指将读取变量值、修改变量值、保存变量值看成一个整体来操作  
即-这几种行为要么同时完成，要么都不完成。

**原子更新基本类型**

**原子更新数组**

**原子更新引用**

**原子更新属性**

### 死锁

## 线程池

### ThreadPoolExecutor类

public class ThreadPoolExecutor extends AbstractExecutorService {

.....

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

...

}

corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

　　ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

ThreadPoolExecutor、AbstractExecutorService、ExecutorService和Executor几个之间的关系了。

Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

然后ThreadPoolExecutor继承了类AbstractExecutorService。

在ThreadPoolExecutor类中有几个非常重要的方法：

execute()

submit()

shutdown()

shutdownNow()

execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果

### 线程池状态

volatile int runState;

static final int RUNNING = 0;

static final int SHUTDOWN = 1;

static final int STOP = 2;

static final int TERMINATED = 3;

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。