

---

# Table of Contents

简介	1.1
1.线程基础	1.2
1.1什么是线程	1.2.1
1.2线程的分类	1.2.2
1.3线程的信息	1.2.3
1.4线程的生命周期	1.2.4
1.5线程的状态转换	1.2.5
1.6线程优先级	1.2.6
1.7线程创建并启动	1.2.7
1.8线程中断	1.2.8
1.9优雅停止线程	1.2.9
2.线程池	1.3
2.1什么是线程池	1.3.1
2.2为什么需要线程池	1.3.2
2.3线程池生命周期	1.3.3
2.4创建线程池	1.3.4
2.5为什么不建议使用Executors创建线程池	1.3.5

个人学习笔记.工作经验总结.学习总结等等内容.

## **环境:**

Java 1.8

MySQL 5.7

maven 3

IDEA

# 介绍

线程基础

线程是一个独立执行的调用序列，同一个进程的线程在同一时刻共享一些系统资源（比如文件句柄等）也能访问同一个进程所创建的对象资源（内存资源）。java.lang.Thread对象负责统计和控制这种行为。

每个程序都至少拥有一个线程-即作为Java虚拟机(JVM)启动参数运行在主类main方法的线程。在Java虚拟机初始化过程中也可能启动其他的后台线程。这种线程的数目和种类因JVM的实现而异。然而所有用户级线程都是显式被构造并在主线程或者是其他用户线程中被启动。

只要当前JVM实例中尚存任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作，Daemon作用是为其他线程提供便利服务，守护线程最典型的应用就是GC(垃圾回收器)，他就是一个很称职的守护者。

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

#### User Thread(用户线程)

粗略理解，非守护线程约等于用户线程

```
Thread one = new Thread();
one.setDaemon(false);
```

#### DaemonThread(守护线程)

```
Thread one = new Thread();
one.setDaemon(true);
```

Thread类的对象中保存了一些属性信息能够帮助我们来辨别每一个线程，知道它的状态，调整控制其优先级。

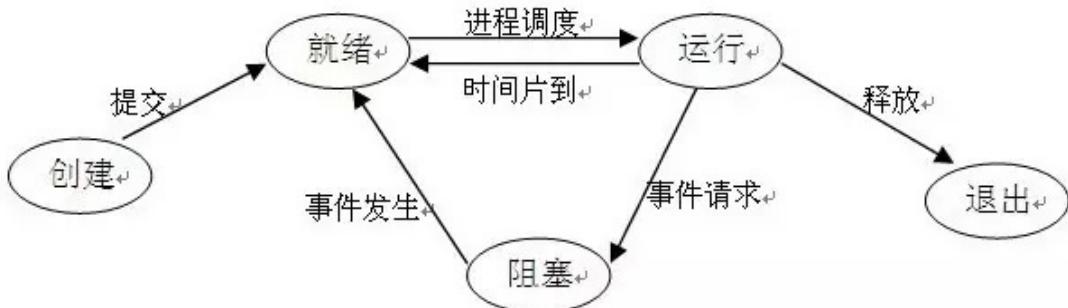
部分Thread类的属性：

- ID: 每个线程的独特标识。
- Name: 线程的名称。
- Priority: 线程对象的优先级。优先级别在1-10之间，1是最低级，10是最高级。不建议改变它们的优先级，但是你想的话也是可以的。
- Status: 线程的状态。在Java中，线程只能有这6种中的一种状态： new, runnable, blocked, waiting, time waiting, 或 terminated.

```
/*
 * 打印线程信息
 */
private static void printThreadInfo(Thread thread){
    System.out.println("线程ID: "+thread.getId());
    System.out.println("线程名字: "+thread.getName());
    System.out.println("线程优先级: "+thread.getPriority());
    System.out.println("线程状态: "+thread.getState());
    System.out.println("线程只能有这6种中的一种状态: new, runnable, blocked, waiting, time waiting, 或 terminated.");
};

public static void main(String[] args) {
    System.out.println("建议单个测试");
    printThreadInfo(Thread.currentThread());
}
```

## 进程三态状态转换图



- (1) 就绪→执行 处于就绪状态的进程，当进程调度程序为之分配了处理机后，该进程便由就绪状态转变成执行状态。
- (2) 执行→就绪 处于执行状态的进程在其执行过程中，因分配给它的一个时间片已用完或更高优先级的进程抢占而不得不让出处理器，于是进程从执行状态转变成就绪状态。
- (3) 执行→阻塞 正在执行的进程因等待某种事件发生而无法继续执行时，便从执行状态变成阻塞状态。
- (4) 阻塞→就绪 处于阻塞状态的进程，若其等待的事件已经发生，于是进程由阻塞状态转变为就绪状态。
- (5) 运行→终止 程序执行完毕，撤销而终止

以上是最经典也是最基本的三种进程状态，但现在的操作系统都根据需要重新设计了一些新的状态。

- 运行状态（TASK\_RUNNING）：是运行态和就绪态的合并，表示进程正在运行或准备运行，Linux 中使用 TASK\_RUNNING 宏表示此状态
- 可中断睡眠状态（浅度睡眠）（TASK\_INTERRUPTIBLE）：进程正在睡眠（被阻塞），等待资源到来是唤醒，也可以通过其他进程信号或时钟中断唤醒，进入运行队列。Linux 使用 TASK\_INTERRUPTIBLE 宏表示此状态。
- 不可中断睡眠状态（深度睡眠状态）（TASK\_UNINTERRUPTIBLE）：
- 其和浅度睡眠基本类似，但有一点就是不可被其他进程信号或时钟中断唤醒。Linux 使用 TASK\_UNINTERRUPTIBLE 宏表示此状态。
- 暂停状态（TASK\_STOPPED）：进程暂停执行接受某种处理。如正在接受调试的进程处于这种状态，Linux 使用 TASK\_STOPPED 宏表示此状态。
- 僵死状态（TASK\_ZOMBIE）：进程已经结束但未释放PCB，Linux 使用 TASK\_ZOMBIE 宏表示此状态

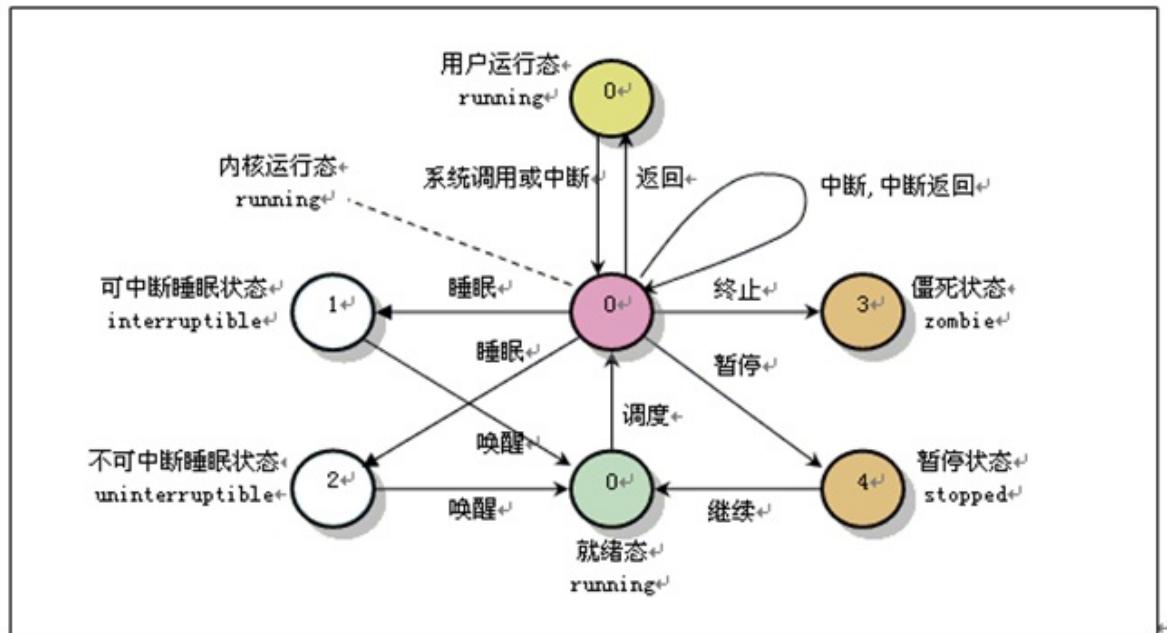


图2.6 进程状态及转换关系

在Java中，线程只能有这6种中的一种状态： new, runnable, blocked, waiting, time waiting, 或 terminated.

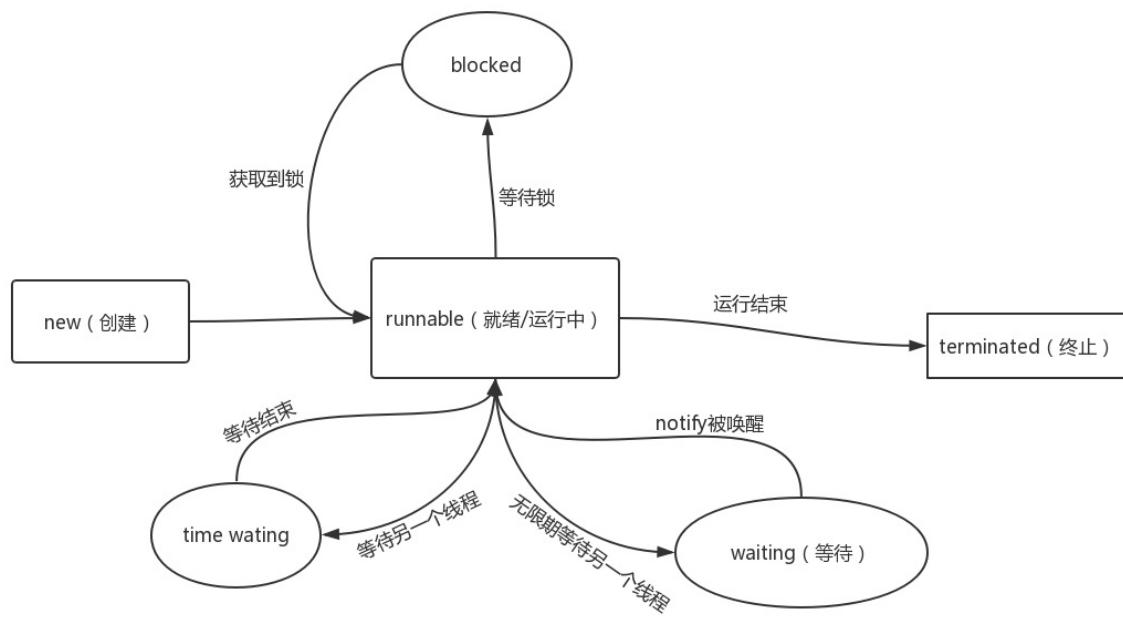
在JDK源码中有个内部枚举类，如下：

```

/*
 * 线程状态。 线程可以处于以下状态之一：
 * #NEW
 * 尚未启动的线程处于此状态。
 * #RUNNABLE
 * 在Java虚拟机中执行的线程处于此状态
 * #BLOCKED
 * 被阻塞等待监视器锁定的线程
 * 处于这种状态
 * #WAITING
 * 一个无限期等待另一个线程的线程
 * 执行特定操作处于此状态。
 * #TIMED_WAITING
 * 正在等待另一个线程执行操作的线程
 * 在指定的等待时间内心处于此状态。
 * #TERMINATED
 * 已退出的线程处于此状态。
 * 线程在给定时间点只能处于一种状态。
 * 这些状态是虚拟机状态，不反映
 * 任何操作系统线程状态。
 * @since 1.5
 * @see #getState
 */
public enum State {
    /**
     * 线程创建之后，但是还没有启动(not yet started)。这时候它的状态就是NEW
     */
    NEW,
    /**
     * RUNNABLE: 正在Java虚拟机下跑任务的线程的状态。在RUNNABLE状态下的线程可能会处于等待状态， 因为 * 它正在等待一些系统资源的释放，比如IO
     */
    RUNNABLE,
    /**
     * BLOCKED: 阻塞状态，等待锁的释放，比如线程A进入了一个synchronized方法， * 线程B也想进入这个方法，但是这个方法的锁已经被线程A获取了，这个时候线程B就处于BLOCKED状态
     */
    BLOCKED,
    /**
     * WAITING: 等待状态，处于等待状态的线程是由于执行了3个方法中的任意方法。
     * 1. Object的wait方法，并且没有使用timeout参数;
     * 2. Thread的join方法，没有使用timeout参数
     * 3. LockSupport的park方法。 处于waiting状态的线程会等待另外一个线程处理特殊的行为。
     * 再举个例子，如果一个线程调用了一个对象的wait方法，那么这个线程就会处于waiting状态
     * 直到另外一个线程调用这个对象的notify或者notifyAll方法后才会解除这个状态
     */
    WAITING,
    /**
     * TIMED_WAITING: 有等待时间的等待状态
     * 比如调用了以下几个方法中的任意方法，并且指定了等待时间，线程就会处于这个状态。
     * 1. Thread.sleep方法
     * 2. Object的wait方法，带有时间
     * 3. Thread.join方法，带有时间
     * 4. LockSupport的parkNanos方法，带有时间
     * 5. LockSupport的parkUntil方法，带有时间
     */
    TIMED_WAITING,
    /**
     * 线程中止的状态，这个线程已经完整地执行了它的任务
     */
}

```

```
*/
TERMINATED;
}
```



## 线程优先级的介绍

java 中的线程优先级的范围是1~10， 默认的优先级是5。“高优先级线程”会优先于“低优先级线程”执行。

java 中有两种线程：用户线程和守护线程。可以通过isDaemon()方法来区别它们：如果返回false，则说明该线程是“用户线程”；否则就是“守护线程”。用户线程一般用户执行用户级任务，而守护线程也就是“后台线程”，一般用来执行后台任务。需要注意的是：Java虚拟机在“用户线程”都结束后会后退出。

每个线程都有一个优先级。“高优先级线程”会优先于“低优先级线程”执行。每个线程都可以被标记为一个守护进程或非守护进程。在一些运行的主线程中创建新的子线程时，子线程的优先级被设置为等于“创建它的主线程的优先级”，当且仅当“创建它的主线程是守护线程”时“子线程才会是守护线程”。

当Java虚拟机启动时，通常有一个单一的非守护线程（该线程通过main()方法启动）。JVM会一直运行直到下面的任意一个条件发生，JVM就会终止运行：(01) 调用了exit()方法，并且exit()有权限被正常执行。(02) 所有的“非守护线程”都死了(即JVM中仅仅只有“守护线程”)。

每一个线程都被标记为“守护线程”或“用户线程”。当只有守护线程运行时，JVM会自动退出。

## 示例

```
class MyThread extends Thread{
    public MyThread(String name) {
        super(name);
    }
    public void run(){
        for (int i=0; i<5; i++) {
            System.out.println(Thread.currentThread().getName()
                +"("+Thread.currentThread().getPriority()+ ")"
                +", loop "+i);
        }
    }
};

public class Demo {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()
            +"("+Thread.currentThread().getPriority()+ ")");
        Thread t1=new MyThread("t1"); // 新建t1
        Thread t2=new MyThread("t2"); // 新建t2
        t1.setPriority(1); // 设置t1的优先级为1
        t2.setPriority(10); // 设置t2的优先级为10
        t1.start(); // 启动t1
        t2.start(); // 启动t2
    }
}
```

## 来源：

[Java多线程系列--“基础篇”10之 线程优先级和守护线程](#)

## 1. 创建线程

创建线程有两种方法，一种是继承Thread类重写run方法，另一种是实现Runnable接口重写run方法。

继承Thread类重写run方法：

```
public class NewThread extends Thread {
    /**
     * 重写run方法
     */
    @Override
    public void run() {
        System.out.println("我是新线程要执行的方法！继承方式");
    }
}
```

实现Runnable接口重写run方法：

```
public class ThreadByRunnable implements Runnable {
    /**
     * 实现run方法
     */
    @Override
    public void run() {
        System.out.println("我是新线程要执行的方法！实现接口方式");
    }
}
```

## 2. 启动线程

在某一线程内启动线程，调用 `start()` 方法。一般在主线程内开启新的线程。如下例子：

```
public class MainTest {
    public static void main(String[] args) {
        //继承Thread启动的方法
        NewThread t1 = new NewThread();
        t1.start(); //启动线程

        //实现Runnable启动线程的方法
        ThreadByRunnable r = new ThreadByRunnable();
        Thread t2 = new Thread(r);
        t2.start(); //启动线程
    }
}
```

调用start()方法后并不是立即的执行多线程的代码，而是使该线程变为可运行态，什么时候运行多线程代码是由操作系统决定的。

## 使用 interrupt()中断线程

当一个线程运行时，另一个线程可以调用对应的 Thread 对象的 interrupt()方法来中断它，该方法只是在目标线程中设置一个标志，表示它已经被中断，并立即返回。这里需要注意的是，如果只是单纯的调用 interrupt()方法，线程并没有实际被中断，会继续往下执行。

下面一段代码演示了休眠线程的中断：

```
public class SleepInterrupt extends Object implements Runnable{
    public void run(){
        try{
            System.out.println("in run() - about to sleep for 20 seconds");
            Thread.sleep(20000);
            System.out.println("in run() - woke up");
        }catch(InterruptedException e){
            System.out.println("in run() - interrupted while sleeping");
            //处理完中断异常后，返回到run()方法入口,
            //如果没有return，线程不会实际被中断，它会继续打印下面的信息
            return;
        }
        System.out.println("in run() - leaving normally");
    }

    public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();
        //主线程休眠2秒，从而确保刚才启动的线程有机会执行一段时间
        try {
            Thread.sleep(2000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        System.out.println("in main() - interrupting other thread");
        //中断线程t
        t.interrupt();
        System.out.println("in main() - leaving");
    }
}
```

## Thread.currentThread().isInterrupted()方法判断中断状态

判断某个线程是否已被发送过中断请求，请使用 Thread.currentThread().isInterrupted() 方法（因为它将线程中断标示位设置为 true 后，不会立刻清除中断标示位，即不会将中断标设置为 false），而不要使用 thread.interrupted()（该方法调用后会将中断标示位清除，即重新设置为 false）方法来判断。

## 如何中断线程

如果一个线程处于了阻塞状态（如线程调用了 thread.sleep、thread.join、thread.wait、1.5 中的 condition.await、以及可中断的通道上的 I/O 操作方法后可进入阻塞状态），则在线程在检查中断标示时如果发现中断标示为 true，则会在这些阻塞方法（sleep、join、wait、1.5 中的 condition.await 及可中断的通道上的 I/O 操作方法）调用处抛出 InterruptedException 异常，并且在抛出异常后立即将线程的中断标示位清除，即重新设置为 false。抛出异常是为了线程从阻塞状态醒过来，并在结束线程前让程序员有足够的时间来处理中断请求。

注:synchronized在获锁的过程中是不能被中断的，意思是说如果产生了死锁，则不可能被中断。与synchronized功能相似的reentrantLock.lock()方法也是一样，它也不可中断的，即如果发生死锁，那么reentrantLock.lock()方法无法终止，如果调用时被阻塞，则它一直阻塞到它获取到锁为止。但是如果调用带超时的tryLock方法reentrantLock.tryLock(long timeout, TimeUnit unit)，那么如果线程在等待时被中断，将抛出一个InterruptedException异常，这是一个非常有用特性，因为它允许程序打破死锁。你也可以调用reentrantLock.lockInterruptibly()方法，它就相当于一个超时设为无限的tryLock方法。

Thread.interrupt()方法不会中断一个正在运行的线程。这一方法实际上完成的是，设置线程的中断标志位，在线程受到阻塞的地方（如调用sleep、wait、join等地方）抛出一个异常InterruptedException，并且中断状态也将被清除，这样线程就得以退出阻塞的状态

## 中断总结

一、没有任何语言方面的需求一个被中断的线程应该终止。中断一个线程只是为了引起该线程的注意，被中断线程可以决定如何应对中断。

二、对于处于sleep、join等操作的线程，如果被调用interrupt()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。

三、不可中断的操作，包括进入synchronized段以及Lock.lock()，inputSteam.read()等，调用interrupt()对于这几个问题无效，因为它们都不抛出中断异常。如果拿不到资源，它们会无限期阻塞下去。

对于Lock.lock()，可以改用Lock.lockInterruptibly()，可被中断的加锁操作，它可以抛出中断异常。等同于等待时间无限长的Lock.tryLock(long time, TimeUnit unit)。

对于inputStream等资源，有些(实现了interruptibleChannel接口)可以通过close()方法将资源关闭，对应的阻塞也会被放开。

一般说来，如果一个方法声明抛出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。

**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。**

## 来源：

[Thread的中断机制\(interrupt\)](#)

## 通过stop()停止

我们的系统肯定有些线程为了保证业务需要是要常驻后台的，一般它们不会自己终止，需要我们通过手动来终止它们。我们知道启动一个线程是start方法，自然有一个对应的终止线程的stop方法，通过stop方法可以很快速、方便地终止一个线程。

通过注解@Deprecated看出stop方法被标为废弃的方法，jdk在以后的版本中可能被移除，不建议大家使用这种API。

那为什么这么好的一个方法为什么不推荐使用，还要标注为废弃呢？

假设有这样的一个业务场景，一个线程正在处理一个复杂的业务流程，突然间线程被调用stop而意外终止，这个业务数据还有可能是一致的吗？这样是肯定会出问题的，stop会释放锁并强制终止线程，造成执行一半的线程终止，带来的后果也是可想而知的，这就是为什么jdk不推荐使用stop终止线程的方法的原因，因为它很暴力会带来数据不一致性的問題。

## 变量控制

只需要添加一个变量，判断这个变量在某个值的时候就退出循环，这时候每个循环为一个整合不被强行终止就不会影响单个业务的执行结果。

例子：

```
public class TaskThread extends Thread{
    private static volatile boolean stop = false;

    @Override
    public void run(){
        while(!stop){
            //...
            //...
        }
    }
}
```

## 异常控制

这点 Thread 也想到了，提供了一个「**异常**」来达到这个打断的目的。这个异常在其他线程要打断某个特定线程时执行，如果是符合条件，会抛出来。此时这个特定线程自行根据这次打断来判断后续是不是要再执行线程内的逻辑，还是直接跳出处理。

这个异常就是 `InterruptedException`。一般使用方式类似这样：在主线程调用 `interrupt()` 方法使得目标线程抛出 `InterruptedException`

```
public class TaskThread extends Thread{
    private static volatile boolean stop = false;

    @Override
    public void run(){
        try{
            //...
            //...
        }catch(InterruptedException e){
            //do something
        }
    }
}
```



## 介绍

线程池

## 线程池

我们在工作中或多或少都使用过线程池，但是为什么要使用线程池呢？从他的名字中我们就应该知道，线程池使用了一种池化技术，和很多其他池化技术一样，都是为了更高效的利用资源，例如链接池，内存池等等。

数据库链接是一种很昂贵的资源，创建和销毁都需要付出高昂的代价，为了避免频繁的创建数据库链接，所以产生了链接池技术。优先在池子中创建一批数据库链接，有需要访问数据库时，直接到池子中去获取一个可用的链接，使用完了之后再归还到链接池中去。

同样的，线程也是一种宝贵的资源，并且也是一种有限的资源，创建和销毁线程也同样需要付出不菲的代价。我们所有的代码都是由一个一个的线程支撑起来的，如今的芯片架构也决定了我们必须编写多线程执行的程序，以获取最高的程序性能。

那么怎样高效的管理多线程之间的分工与协作就成了一个关键问题，Doug Lea 大神为我们设计并实现了一款线程池工具，通过该工具就可以实现多线程的能力，并实现任务的高效执行与调度。

为了正确合理的使用线程池工具，我们有必要对线程池的原理进行了解。

## 为什么需要线程池?

### 1. 创建/销毁线程伴随着系统开销，过于频繁的创建/销毁线程，会很大程度上影响处理效率，例如：

记创建线程消耗时间 T1，执行任务消耗时间 T2，销毁线程消耗时间 T3

如果  $T1+T3 > T2$ ，那么是不是说开启一个线程来执行这个任务太不划算了！

正好，线程池缓存线程，可用已有的闲置线程来执行新任务，避免了  $T1+T3$  带来的系统开销

### 2. 线程并发数量过多，抢占系统资源从而导致阻塞

我们知道线程能共享系统资源，如果同时执行的线程过多，就有可能导致系统资源不足而产生阻塞的情况

运用线程池能有效的控制线程最大并发数，避免以上的问题

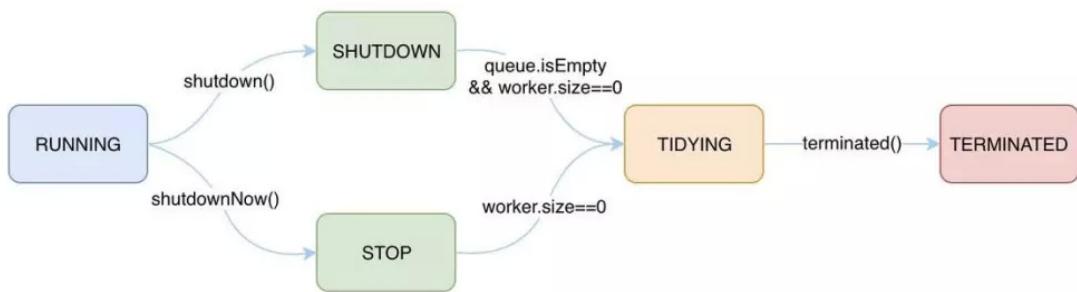
### 3. 对线程进行一些简单的管理

比如：延时执行、定时循环执行的策略等

运用线程池都能进行很好的实现

首先线程池是有状态的，这些状态标识这线程池内部的一些运行情况，线程池的开启到关闭的过程就是线程池状态的一个流转的过程。

线程池共有五种状态：



状态	含义
RUNNING	运行状态，该状态下线程池可以接受新的任务，也可以处理阻塞队列中的任务 执行 shutdown 方法可进入 SHUTDOWN 状态 执行 shutdownNow 方法可进入 STOP 状态
SHUTDOWN	待关闭状态，不再接受新的任务，继续处理阻塞队列中的任务 当阻塞队列中的任务为空，并且工作线程数为0时，进入 TIDYING 状态
STOP	停止状态，不接收新任务，也不处理阻塞队列中的任务，并且会尝试结束执行中的任务 当工作线程数为0时，进入 TIDYING 状态
TIDYING	整理状态，此时任务都已经执行完毕，并且也没有工作线程 执行 terminated 方法后进入 TERMINATED 状态
TERMINATED	终止状态，此时线程池完全终止了，并完成了所有资源的释放

## 来源：

[逅弈逐码](#)

## Executors创建线程池

java.util包下面提供了Executors工具类创建常见的四种线程池.分别为

1. CachedThreadPool():可缓存线程池
2. FixedThreadPool():定长线程池
3. ScheduledThreadPool():支持定时及周期性任务执行
4. SingleThreadExecutor():单线程化的线程池

### CachedThreadPool()

可缓存线程池:

1. 线程数无限制
2. 有空闲线程则复用空闲线程，若无空闲线程则新建线程
3. 一定程度减少频繁创建/销毁线程，减少系统开销

创建方法：

```
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
```

内部实现:

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

### FixedThreadPool()

定长线程池:

1. 可控制线程最大并发数（同时执行的线程数）
2. 超出的线程会在队列中等待

创建方法：

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(int nThreads);
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory);
```

内部实现:

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

### ScheduledThreadPool()

定长线程池：

- 支持定时及周期性任务执行。

创建方法：

```
ExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(int corePoolSize);
```

内部实现：

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

//ScheduledThreadPoolExecutor():
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE,
          DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS,
          new DelayedWorkQueue());
}
```

## SingleThreadExecutor()

单线程化的线程池：

1. 有且仅有一个工作线程执行任务
2. 所有任务按照指定顺序执行，即遵循队列的入队出队规则

创建方法：

```
ExecutorService singleThreadPool = Executors.newSingleThreadExecutor();
```

内部实现：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>()));
}
```

## ThreadPoolExecutor构造线程池

如上面Executors工具类的内部实现都是使用了ThreadPoolExecutor这个类来构造线程池,那么我可以直接使用这个ThreadPoolExecutor来构造线程池.

如下面所示,ThreadPoolExecutor有四个构造方法.总共有七个参数.

```

//五个参数的构造函数
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue)

//六个参数的构造函数-1
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory)

//六个参数的构造函数-2
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           RejectedExecutionHandler handler)

//七个参数的构造函数
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,
                           RejectedExecutionHandler handler)

```

下面来对这七个参数来做说明.当解释完这七个参数,就可以清楚上面的Executors的构造方法,就可以清楚知道上面常用的四个线程池的区别和特点了.

## 重要参数

### 1.corePoolSize

#### 核心线程:

线程池新建线程的时候,如果当前线程总数小于 corePoolSize, 则新建的是核心线程, 如果超过 corePoolSize, 则新建的是非核心线程

核心线程默认情况下会一直存活在线程池中, 即使这个核心线程啥也不干(闲置状态)。

如果指定 ThreadPoolExecutor 的 allowCoreThreadTimeOut 这个属性为 **true**, 那么核心线程如果不干活(闲置状态)的话, 超过一定时间(时长下面参数决定), 就会被销毁掉

### 2.maximumPoolSize

#### 该线程池中线程总数最大值

线程总数 = 核心线程数 + 非核心线程数

### 3. keepAliveTime

该线程池中**非核心线程闲置超时时长**

一个非核心线程，如果不干活(闲置状态)的时长超过这个参数所设定的时长，就会被销毁掉

如果设置allowCoreThreadTimeOut = true，则会作用于核心线程

### 4. TimeUnit unit

keepAliveTime 的单位，TimeUnit 是一个枚举类型，其包括：

**NANOSECONDS**： 1微毫秒 = 1微秒 / 1000

**MICROSECONDS**： 1微秒 = 1毫秒 / 1000

**MILLISECONDS**： 1毫秒 = 1秒 /1000

**SECONDS**： 秒

**MINUTES**： 分

**HOURS**： 小时

**DAYS**： 天

### 5. BlockingQueue workQueue

该线程池中的任务队列：维护着等待执行的Runnable对象

当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果队列满了，则新建非核心线程执行任务

常用的 workQueue 类型：

**SynchronousQueue**：这个队列接收到任务的时候，会直接提交给线程处理，而不保留它，如果所有线程都在工作怎么办？那就新建一个线程来处理这个任务！所以为了保证不出现<线程数达到了 maximumPoolSize 而不能新建线程>的错误，使用这个类型队列的时候，maximumPoolSize 一般指定成 Integer.MAX\_VALUE，即无限大

**LinkedBlockingQueue**：这个队列接收到任务的时候，如果当前线程数小于核心线程数，则新建线程(核心线程)处理任务；如果当前线程数等于核心线程数，则进入队列等待。由于这个队列没有最大值限制，即所有超过核心线程数的任务都将被添加到队列中，这也就导致了 maximumPoolSize 的设定失效，因为总线程数永远不会超过 corePoolSize

**ArrayBlockingQueue**：可以限定队列的长度，接收到任务的时候，如果没有达到 corePoolSize 的值，则新建线程(核心线程)执行任务，如果达到了，则入队等候，如果队列已满，则新建线程(非核心线程)执行任务，又如果总线程数到了 maximumPoolSize，并且队列也满了，则发生交给拒绝策略handler决定

**DelayQueue**：队列内元素必须实现 Delayed 接口，这就意味着你传进去的任务必须先实现 Delayed 接口。这个队列接到任务时，首先先入队，只有达到了指定的延时时间，才会执行任务

### 6. ThreadFactory threadFactory

用于设置创建线程的工厂。不指定 则是默认

### 7. handler

线程池的拒绝策略。线程池中的线程已经饱和了，而且阻塞队列也已经满了，则线程池会选择一种拒绝策略来处理该任务

- `AbortPolicy`：默认策略 抛出异常
- `CallerRunsPolicy`：由当前调用者所在的线程来执行任务
- `DiscardOldestPolicy`：丢弃阻塞队列中靠最前的任务，并执行当前任务
- `DiscardPolicy`：直接丢弃多余的任务
- 我们还可以自定义拒绝策略，只需要实现`RejectedExecutionHandler`接口即可，友好的拒绝策略实现有如下：
  - 将数据保存到数据，待系统空闲时再进行处理。
  - 将数据用日志进行记录，后由人工处理。

JDK为我们提供了Executors线程池工具类，里面有默认的线程池创建策略，大概有以下几种：

1. FixedThreadPool：线程池线程数量固定，即corePoolSize和maximumPoolSize数量一样。
2. SingleThreadPool：单个线程的线程池。
3. CachedThreadPool：初始核心线程数量为0，最大线程数量为Integer.MAX\_VALUE，线程空闲时存活时间为60秒，并且它的阻塞队列为SynchronousQueue，它的初始长度为0，这会导致任务每次进来都会创建线程来执行，在线程空闲时，存活时间到了又会释放线程资源。
4. ScheduledThreadPool：创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

用Executors工具类虽然很方便，我依然不推荐大家使用以上默认的线程池创建策略，阿里巴巴开发手册也是强制不允许使用Executors来创建线程池，我们从JDK源码中寻找一波答案：

java.util.concurrent.Executors：

```
// FixedThreadPool
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}

// SingleThreadPool
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService(
        new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>()));
}

// CachedThreadPool
public static ExecutorService newCachedThreadPool() {
    // 允许创建线程数为Integer.MAX_VALUE
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}

// ScheduledThreadPool
public ScheduledThreadPoolExecutor(int corePoolSize) {
    // 允许创建线程数为Integer.MAX_VALUE
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
        new DelayedWorkQueue());
}
public LinkedBlockingQueue() {
    // 允许队列长度最大为Integer.MAX_VALUE
    this(Integer.MAX_VALUE);
}
```

## 结论：

从JDK源码可看出，Executors工具类无非是把一些特定参数进行了封装，并提供一些方法供我们调用而已，我们并不能灵活地填写参数，策略过于简单，不够友好。

**CachedThreadPool**和**ScheduledThreadPool**\*\*最大线程数为Integer.MAX\_VALUE，如果线程无限地创建，会造成OOM异常\*\*。

**LinkedBlockingQueue**基于链表的FIFO队列，是无界的，默认大小是Integer.MAX\_VALUE，因此**FixedThreadPool**和**SingleThreadPool**的阻塞队列长度为Integer.MAX\_VALUE，如果此时队列被无限地堆积任务，会造成**OOM异常**。

