# JAVA多线程

## 线程与进程

1. 线程：

线程是进程中的一个执行流程，一个进程可以运行多个线程。比如java.exe进程中可以运行很多线程。线程总属于某个进程，没有自己的虚拟地址空间，与进程内的其他线程一起共享分配给该进程的所有资源。

线程本身依靠程序进行运行。

每个独立的线程有一个程序运行的入口，顺序执行序列和程序的出口。但是线程不能够独立执行，必须存在应用程序中，由应用程序提供多个线程执行控制。

线程是cpu调度和分派的基本单位。只拥有一点在运行中必不可少的资源(程序计数器，一组寄存器和栈)，但是他可与同属一个进程的其他的线程共享进程所拥有的的全部资源。

线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程包括以下内容：

1. 一个指向当前被执行指令的指令指针；
2. 一个栈；
3. 一个寄存器值的集合，定义了一部分描述正在执行线程的处 理器状态的值；
4. 一个私有的数据区。

在Java中，每次程序运行至少启动2个线程：一个是main线程，一个是垃圾收集线程。因为每当使用java命令执行一个类的时候，实际上都会启动一个JVM，每一个JVM实际上就是在操作系统中启动了一个进程。

1. 进程：

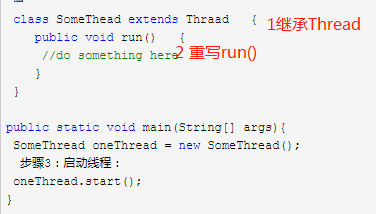
一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间，即进程空间。进程不依赖与线程而独立存在，一个进程中可以启动多个线程。比如在windows中，一个exe就是一个进程。

进程是系统进行资源分配和调度的一个独立单位。

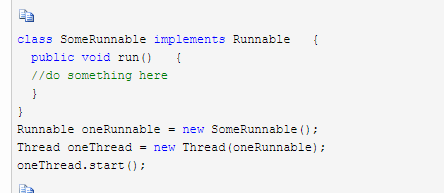
1. 线程比进程的优势有：
2. 进程之间不能共享数据，线程可以。
3. 系统创建进程需要为该进程重新分配系统资源，故创建线程代价比较 小。

## 创建线程和启动

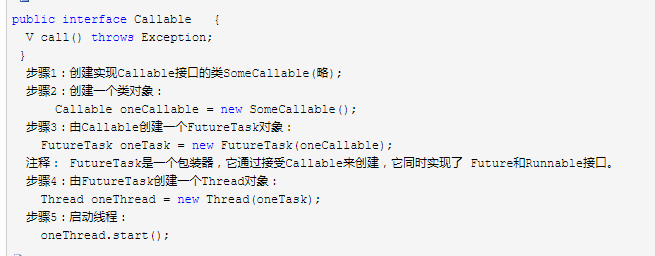
1. 继承Thread类创建线程类



1. 实现Runnable接口创建线程类



1. 通过Callable和Future创建线程
2. 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线 程执行体，并且有返回值。
3. 创建Callable实现类的实例，使用FutureTask类来包装Callable对象， 该FutureTask对象封装了该Callable对象的call()方法的返回值。
4. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
5. 调用FutureTask对象的get方法来获得子线程执行结束后的返回值。



## 线程的生命周期

1. 新建状态

用new关键字和Thread类或其子类建立一个线程对象后，该线程对象就处于新生状态。处于新生状态的线程有自己的内存空间，通过调用start方法进入就绪状态。

注意：不能对已经启动的线程再次调用start()方法，否则会出现异常。

1. 就绪状态

处于就绪状态的线程已经具备了运行条件，但还没与分配到CPU，处于线程就绪队列(尽管是采用队列形式，事实上，把它称为可运行池而不是可运行队列。因为cpu的调度不一定是按照先进先出的顺序来调度的)，一旦获得cpu，线程就进入运行状态并自动调用自己的run()方法。

1. 运行状态

处于运行状态的线程最为复杂，他可以变为阻塞状态，就绪状态和死亡状态。

失去cpu资源后，会从运行状态变为就绪状态。重新等待系统分配资源。也可以运行中的线程调用yield()方法，他就会让出cpu资源，再次变为就绪状态。让运行状态变为阻塞状态的方法：

1. 线程调用sleep方法主动放弃所占用的系统资源。
2. 线程调用一个阻塞式IO方法，在该方法返回之前，该线程被阻塞。
3. 线程在等待某个通知(notify)

线程run()方法执行完，或者被强制性终止例如抛出异常、stop()、destory()会让线程从运行状态变为死亡状态。

1. 阻塞状态

处于运行状态的线程在某些情况下，如执行了sleep()方法，或等待I/O设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。

在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时便 会转入就绪状态，重新导就绪队列中等待。被系统选中后从原来停止的 位置继续运行。

1. 死亡状态

当run()执行完，或被强制性中指。就认为他死去，也许该线程对象是活的，但是，他已经不是一个单独执行的线程。如果在一个死去的线程上调用start()方法，会抛出异常。

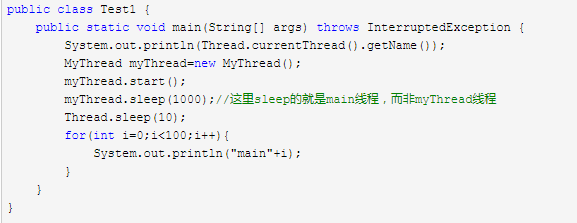
## 线程管理

1. 线程睡眠 sleep

如果我们需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过Thread的sleep方法。

注：

1. sleep是静态方法，只用Thread.sleep()调用就好了。



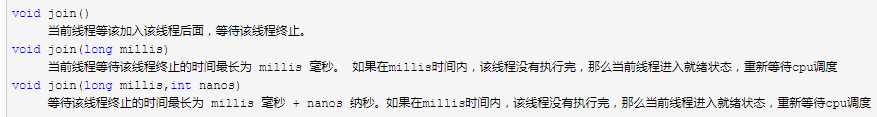
1. sleep()无法实现精准调度。
2. 线程让步 yield

yield()方法和sleep()方法有点相似，它也是Thread类提供的一个静态的方法，也可以让当前执行的线程暂停，让出cpu资源给其他线程。但该方法执行不会使线程进入阻塞状态，而是进入就绪状态。

sleep()和yield()区别：

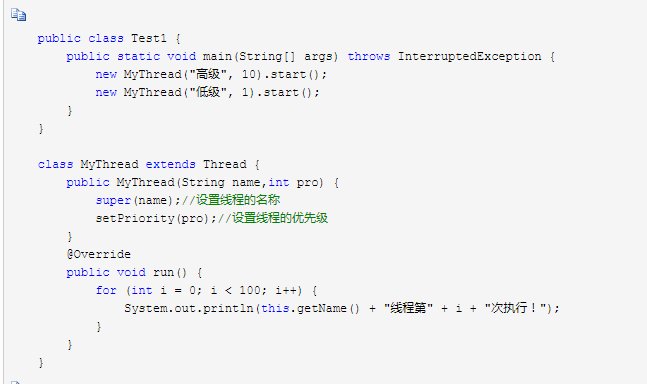
1. Sleep方法暂停当前线程后，会进入阻塞状态，只有当睡眠时间到了， 才会转入就绪状态。而yield方法调用后，是直接进入就绪状态，所以有 可能刚进入就绪状态，又被调度到运行状态。
2. Sleep方法声明抛出了IneruptedException，所以调用该方法要捕获该 异常。而yield方法没有异常抛出。
3. Sleep方法比yield方法有更好的可移植性，通常不要依靠yield方法 来控制并发线程的执行。
4. 线程合并 join

应用场景是当一个线程必须等待另一个线程执行完毕才能执行时，Thread类提供了join方法来完成这个功能，注意，它不是静态方法。



1. 设置线程的优先级

优先级高的线程获取cpu资源概率较大，优先级低的也并非没有机会执行。每个线程的优先级都与创建它的父线程具有相同的优先级，在默认情况下，main线程具有普通优先级。



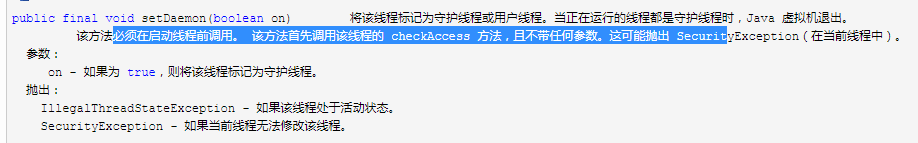
最好使用MAX\_PRIORITY，MIN\_PRIORITY和NORM\_PRIORITY三个静态常量来设定优先级。可以保证可移植性。

1. 后台(守护)线程

JVM垃圾回收，内存管理等线程都是守护线程，或做数据库应用时，使用的数据库连接池本身包含很多后台线程，监控连接个数，超时时间，状态等等。调用线程对象 setDaemon(true)可以将其设置为守护线程。

作用：

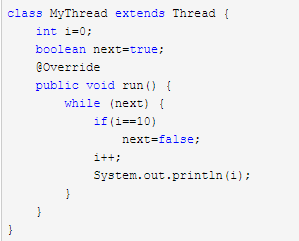
1. 执行后台作业。
2. 不需要关心它的结束问题。



1. 正确结束线程

正常执行完run()方法，然后结束掉；

控制循环条件和判断条件的标识符来结束掉线程。



## 线程同步

当多个线程同时操作一个可共享的资源变量时，会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程调用，从而保证了该变量唯一性和准确性。

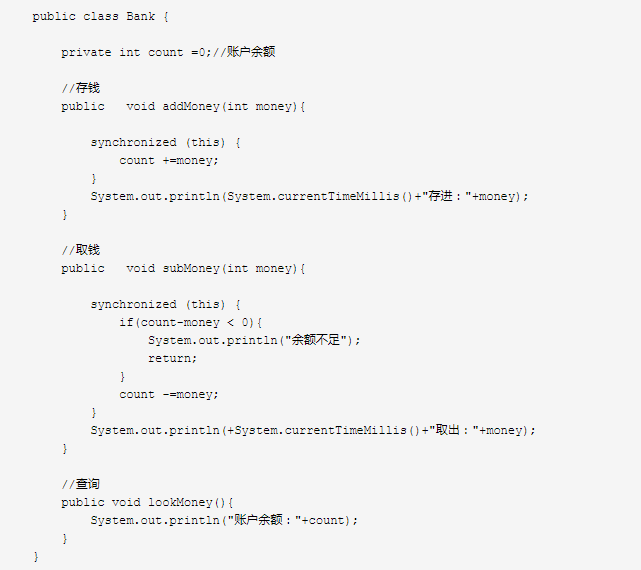
1. 同步方法

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。

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。

1. 同步代码块

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。



尽量减少同步的内容。通常没有必要同步整个方法。

1. 使用特殊域变量(volatile)实现线程同步
2. volatile关键字为域变量的访问提供了一种免锁机制；
3. 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；
4. 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；
5. 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。

注：多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。用final域，有锁保护的域和volatile域可以避免非同步的问题。