线程

创建线程的几种方式

1. 继承Thread

Class ThreadTest extends Thread{

Public void run(){

//实现

}

Public static void main(String[]args){

ThreadTest tt = new ThreadTest ();

tt.start();

}

}

1. 实现Runnable

Class RunnableTest implements Runnable{

Public void run(){

//实现

}

Public static void main(String[]args){

RunnableTest rt = new RunnableTest ();

Thread t = new Thread(rt);

t.start();

}

}

1. 使用Callabe与FutureTask

Class CallableTest implements Callable<Integer>{

public Integer call(){

//处理

return i;

}

Public static void main(String[]args){

CallableTest ct = new CallableTest();

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

Thread thread = new Thread(ft);

thread.start();

//子线程执行结束后的返回值

ft.get();

}

}

1. 线程池

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程；  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待；  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行；  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

1. **newCachedThreadPool**

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

**public static void main(String[] args) {**

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

**for (int i = 0; i < 10; i++) {**

**final int index = i;**

**try {**

**Thread.sleep(index \* 1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**cachedThreadPool.execute(new Runnable() {**

**public void run() {**

**System.out.println(index);**

**}**

**});**

**}**

**}**

1. **newFixedThreadPool**

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。  
定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()

public static void main(String[] args) {

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

final int index = i;

fixedThreadPool.execute(new Runnable() {

public void run() {

try {

System.out.println(index);

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

}

}

1. newScheduledThreadPool

public static void main(String[] args) {

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

// 延迟3秒执行

scheduledThreadPool.schedule(new Runnable() {

public void run() {

System.out.println("delay 3 seconds");

}

}, 3, TimeUnit.SECONDS);

// 表示延迟1秒后每3秒执行一次

scheduledThreadPool.scheduleAtFixedRate(new Runnable() {

public void run() {

System.out.println("delay 1 seconds, and excute every 3 seconds");

}

}, 1, 3, TimeUnit.SECONDS);

}

1. newSingleThreadExecutor

1）基本定义和资源拥有方面：线程是指进程内的一个执行单元，也是进程内可调度的实体，是CPU调度和分派的基本单位。进程是具有一定独立功能的程序，是系统进行资源分配和调度的一个独立单元。线程自己基本上不拥有系统资源、地址空间,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

2）一个程序至少有一个进程，一个进程至少有一个线程

3）二者皆可并发执行，多线程程序的并发性高

4）由逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行，但操作系统并没有将多个线程看做多个独立的应用来实现进程的调度和管理，这就是线程和进程的重要区别。

5）一个线程可以创建和撤消另一个线程，但是进程一个进程无法直接访问另一个进程的变量和数据结构，必须通过进程间的通信

6）创建进程的开销大，包括创建虚拟地址空间等需要大量系统资源；线程创建和切换的开销更小。创建线程开销小，基本上只有一个内核对象和一个堆栈。

7）线程在执行过程中与进程还是有区别的。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，必须依存在应用程序中，由应用程序提供多个线程执行控制

广义上讲，进程间通信(Inter-Process Communication, IPC)是指运行在不同进程(不论是否在同一台机器)中的若干线程间的数据交换  
共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。  
管道：生命周期随着进程结束而完结，管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系  
为了克服局限性，可以使用命名管道，也是半双工，它允许无亲缘关系进程间的通信。  
套接字：其实网络通信中所使用的API与跨进程使用的是完全一样的；注意，如果服务器与客户端在同一台主机上，则是跨进程通信。如果在不同的主机上，则变成了网络通信。  
消息队列：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点  
信号: 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。  
信号量: 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

1. 线程状态

1、这两个方法来自不同的类分别是，sleep来自Thread类，和wait来自Object类。

sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。

2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

sleep不出让系统资源；wait是进入线程等待池等待，出让系统资源，其他线程可以占用CPU。一般wait不会加时间限制，因为如果wait线程的运行资源不够，再出来也没用，要等待其他线程调用notify/notifyAll唤醒等待池中的所有线程，才会进入就绪队列等待OS分配系统资源。sleep(milliseconds)可以用时间指定使它自动唤醒过来，如果时间不到只能调用interrupt()强行打断。

Thread.Sleep(0)的作用是“触发操作系统立刻重新进行一次CPU竞争”。

3、使用范围：wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用   
   synchronized(x){   
      x.notify()   
     //或者wait()   
   }  
4、sleep必须捕获异常InterruptedException，而wait，notify和notifyAll不需要捕获异常

**1.sleep()方法**

在指定时间内让当前正在执行的线程暂停执行，但不会释放“锁标志”。不推荐使用。

sleep()使当前线程进入阻塞状态，在指定时间内不会执行。

**2.wait()方法**

在其他线程调用对象的notify或notifyAll方法前，导致当前线程等待。线程会释放掉它所占有的“锁标志”，从而使别的线程有机会抢占该锁。

当前线程必须拥有当前对象锁。如果当前线程不是此锁的拥有者，会抛出IllegalMonitorStateException异常。

唤醒当前对象锁的等待线程使用notify或notifyAll方法，也必须拥有相同的对象锁，否则也会抛出IllegalMonitorStateException异常。

waite()和notify()必须在synchronized函数或synchronized　block中进行调用。如果在non-synchronized函数或non-synchronized　block中进行调用，虽然能编译通过，但在运行时会发生IllegalMonitorStateException的异常。

**3.yield方法**

暂停当前正在执行的线程对象。

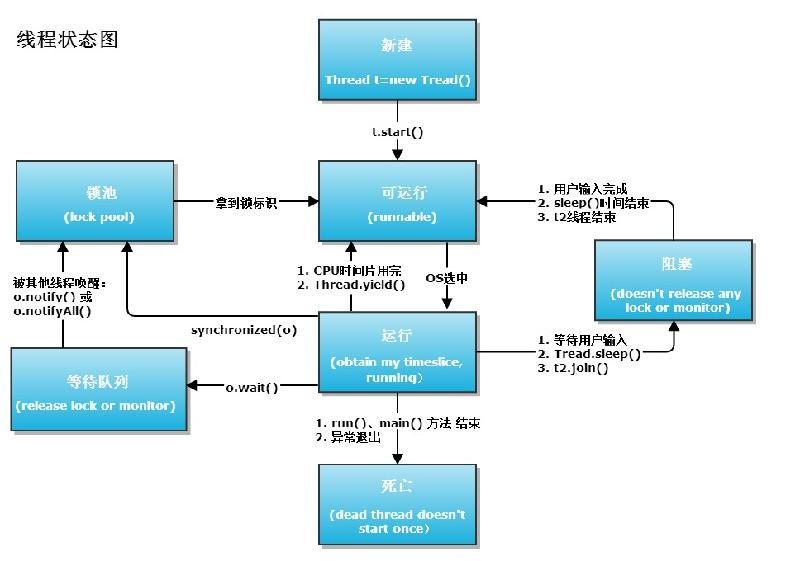
yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

yield()只能使同优先级或更高优先级的线程有执行的机会。

**4.join方法**

等待该线程终止。

等待调用join方法的线程结束，再继续执行。如：t.join();//主要用于等待t线程运行结束，若无此句，main则会执行完毕，导致结果不可预测。



1. 新建(new)：新创建了一个线程对象。

2. 可运行(runnable)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。

3. 运行(running)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。

4. 阻塞(block)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

(一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。

(二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。

(三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。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，线程重新转入可运行(runnable)状态。

5. 死亡(dead)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

1. 线程锁

公平锁

1、为什么有公平锁

　　CPU在调度线程的时候是在等待队列里随机挑选一个线程，由于这种随机性所以是无法保证线程先到先得的（synchronized控制的锁就是这种非公平锁）。但这样就会产生饥饿现象，即有些线程（优先级较低的线程）可能永远也无法获取cpu的执行权，优先级高的线程会不断的强制它的资源。那么如何解决饥饿问题呢，这就需要公平锁了。

　　产生饥饿的另一个原因是：某个线程占据资源不释放，那其他需要该资源的线程只能处于无限等待中。在这里我们主要解决第一种饥饿问题。

2、什么是公平锁

　　公平锁可以保证线程按照时间的先后顺序执行，避免饥饿现象的产生。但公平锁的效率比较低，因为要实现顺序执行，需要维护一个有序队列。

二、公平锁的使用

　　JDK1.5为我们提供了实习公平锁的方式，创建公平锁的构造函数是：

java.util.concurrent.locks.ReentrantLock

  public ReentrantLock(boolean fair) {  
        sync = fair ? new FairSync() : new NonfairSync();  
  }

通过判断fair的值来决定重入锁（ReentrantLock）是使用公平锁 FairSync 还是非公平锁（默认） NonfairSync。

1. 线程安全
2. 线程死锁

产生死锁的必要条件

1〉互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。

2〉不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。

3〉占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。

4〉循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，......，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。

解决死锁的方法分为死锁的预防，避免，检测与恢复三种，在实际的操作系统中往往采用死锁的检测与恢复方法来排除死锁。

死锁的预防：

〈1〉打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

   〈2〉打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

    〈3〉打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有如下缺点：

（1）在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；

（2）资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；

（3）降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。

〈4〉打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在以下缺点：

（1）限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；

（2）为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

1. 线程死锁、活锁、饥饿

<http://f.dataguru.cn/thread-521009-1-1.html>