原文链接：http://arashmd.blogspot.com/#trpro,需翻墙.

介绍

在编程中,有时候你需要同时做一些任务(**同时地**),比如,拷贝一个文件并显示进度,或者当展示一些图片作为幻灯片的时候听音乐,或者一个监听器为中断一个文件拷贝操作处理取消事件.如果一个程序需要在同一时刻运行(做)多个任务(**线程**),这种程序被称为**并发**(**多线程**)程序.

使用一个多线程程序的主要问题是运行中的线程之间的 **同步(synchronization)**,因为每个线程想尽快去完成它的任务,比如,一个有三个线程的程序,每个线程内部有一个 `for` 循环,每个迭代打印 A,B,C.默认情况下,没有任何保证他们将在相同的时间完成它们的任务,因为它们是在同一时间开始,可能第三个线程在第一个和第二个线程之前完成它的工作.

同步线程 不是一个如此复杂和困难的工作,它可以很容易完成.在某些情况下,它可能需要大量的代码,并使得应用程序流程复杂化.

目录

- 为什么要并发编程?
  - 如何将一个单线程程序转换为并行的?
- 创建线程
  - Java
    - 示例
    - 练习
  - 将参数传递给线程
    - 示例
    - 练习
- 管理线程
  - 线程属性
    - 示例
  - 线程的行为
    - 示例
  - 线程的行为
- 同步线程
  - 如何让两个线程彼此同步?
    - 示例
  - 同步块
  - 同步方法
    - 示例
    - 练习
  - volatile 关键字
    - 如何将一个变量设置为volatile?

        - 示例

- 说明
    - 主线程
    - 创建线程
    - 后台线程
    - 终止一个线程
    - wait(1000) vs. sleep(1000)
    - 两个线程之间的切换(不真实的 yield())
- 高级主题
    - 管理你的内存
    - 信号量,限制运行中的线程的最大数目
    - 可回收的线程(线程池)

为什么需要并发编程?

让我们使用另一个问题回答这个问题,为什么要用单线程程序?在一个应用程序中有时候某些任务需要同时运行,特别是在一个GUI中,哪一个游戏作为示例,你需要在同一时间追踪时间,管理对象,播放声音,这个可以通过一个多线程的方式完成.即使一个简单的程序比如调用一个网络地址(资源),用户应该能够取消操作,它可以通过在两个独立的线程中完成.事实上,我们需要将一个程序分离为两个单元,那么就可以并发的工作,但是有时候,对于某种类型的应用程序是不行的(比如连续的数学方程式(sequential math formula),或连接到一个数据库).

如何将一个应用程序转换为一个多线程应用程序?是否有可能?

对于程序的每个阶段完全依赖于之前的阶段是不可能的,通常只是一个任务总体,比如一个定时器程序.

我们需要在一个应用程序中去找到需要或与另一个线程一起运行的任务,比如在一个游戏中.我们需要运行加载屏幕作为一个线程去显示用户的游戏加载的进行,它是通过几个模块一起运行的,或者我们可能运行一个文件加载任务作为一个线程去避免屏幕被GUI线程冻结.

- 如果一个程序执行多个任务(通常是GUI apps),可能需要将每个任务作为一个线程运行,当且仅当不存在所属任务的严格的一步一步(step-by-step)的依赖关系
- 如果一个任务可被分隔为几个小的单元,当且仅当在任务中没有逐行(line-by-line)依赖关系,比如,初始化一个大的数组

  注意,在并发应用程序中,其中的一个目的是获得处理器的最大功率.无论是 GPU 还是CPU,如果一个文件转换是利用率为CPU的10%,为什么我们不运行10个并发的线程使得利用率达到100%,减少10倍的执行时间?!

创建线程

首先,我们需要知道如何在我们的应用程序中创建一个线程.在Java语言中,使用一个 `Runnable` 接口或一个 `Thread` 类指定一个线程.在C/C++(11之前)没有特定的线程实现,不过有某些库是可行的.我建议使用 `pthread` ,它比较容易,有一个程序员需要的一起.这里我们使用Java,但是你可以使用其它语言做这个,任何库,你只需要知道逻辑.

Java

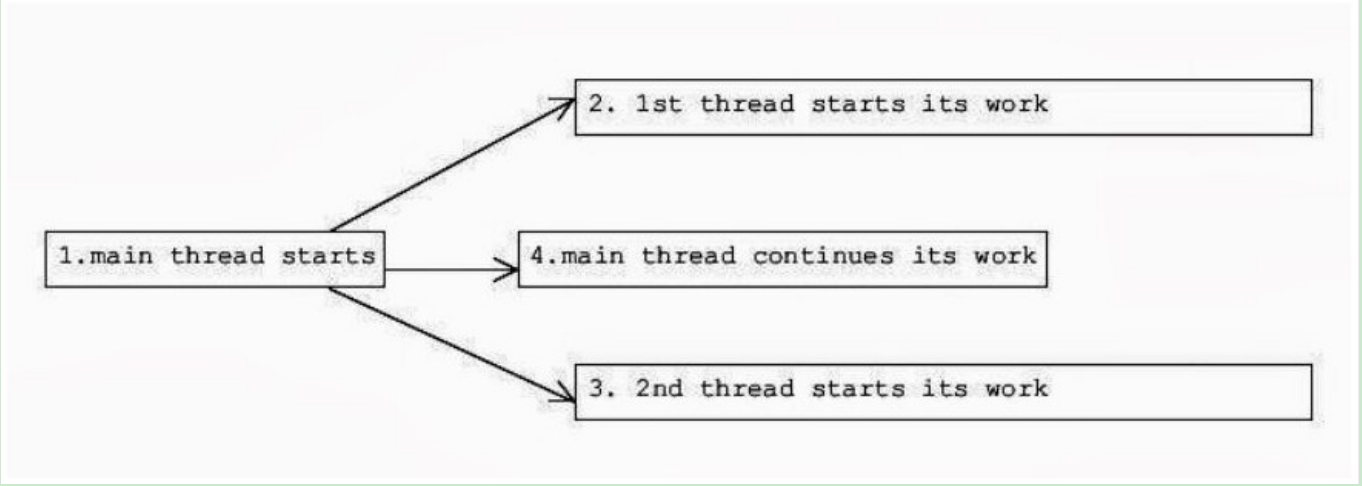
如之前提到的,在Java语言中, `Runnable` 接口或 `Thread` 类都可用来指定一个线程. `Thread` 类启动线程,它获得 `Runnable` 的一个实现,然后在一个独立的线程或当前线程中启动线程. `Runnable` 接口只有一个称为 `run` 的 `void` 方法,不接受任何参数,所以对于任意的输入,你应该通过实现它的类传递.

示例

下面的示例只是两个线程从主线程中启动(第一个线程).

```
1.  package arash.blogger.example.thread;
2.
3.  /**
4.   * by Arash M. Dehghani arashmd.blogspot.com
5.   */
6.  public class Core {
7.      public static void main(String[] args) {
8.          Runnable r0, r1;// pointers to thread methods
9.          r0 = new FirstIteration(); // init Runnable
10.         r1 = new SecondIteration();
11.         Thread t0, t1;// Thread class is used for starting a thread (runnable
12.                         // instance)
13.         t0 = new Thread(r0);// init thread object, but haven't started yet
14.         t1 = new Thread(r1);
15.         t0.start();// start the thread simultaneously
16.         t1.start();
17.         System.out.print("Threads started, no surprise here!\n");
18.     }
19. }
20.
21. class FirstIteration implements Runnable {
22.     @Override
23.     public void run() {// thread starts from here
24.         for (int i = 0; i < 20; i++) {
25.             System.out.print("Hello from 1st. thread, iteration=" + i + "\n");
26.         }
27.     }
28. }
29.
30. class SecondIteration implements Runnable {
31.     @Override
32.     public void run() {
33.         for (int i = 0; i < 20; i++) {
34.             System.out.print("who just called 2st. thread? iteration=" + i + "\n");
35.         }
36.     }
37. }
```

多次尝试上面的代码,然后你将会面对不同的响应,这是因为线程有竞态条件(Race conditions),这意味着每个线程想尽快的完成它的任务.下面的图解展示了在上面的例子中发生了什么.



练习

现在,轮到你了,只要使用 Ctrl + C和Ctrl + V停止,然后尝试下面的练习(有任何问题随时联系).

- 一个创建了100个线程的程序,并发地随机命名目录.

同时在单线程中开发上面的示例.

传递参数给线程

把一个值传递给一个线程真的很容易,但是隐式使用 `Runnable` 是不行的,它可以通过宿主类,构造函数,setter,等完成.

示例

检出下面的示例用于更好的理解.

```
1.  package arash.blogger.example.thread;
2.
3.  /**
4.   * by Arash M. Dehghani arashmd.blogspot.com
5.   */
6.  public class Core {
7.      public static void main(String[] args) {
8.          Runnable r0, r1;// pointers to a thread method
9.          r0 = new FirstIteration("Danial"); // init Runnable, and pass arg to thread 1 by co
    nstructor
10.         SecondIteration si = new SecondIteration();
11.         si.setArg("Pedram");// pass arg to thread 2 by setter
12.         r1 = si;
13.         Thread t0, t1;
14.         t0 = new Thread(r0);
15.         t1 = new Thread(r1);
16.         t0.start();
17.         t1.start();
18.         System.out.print("Threads started with args, nothing more!\n");
19.     }
20. }
21.
22. class FirstIteration implements Runnable {
23.     public FirstIteration(String arg) {
24.         // input arg for this class, but in fact input arg for this thread.
25.         this.arg = arg;
```

```
26.       }
27.
28.       private String arg;
29.
30.       @Override
31.       public void run() {
32.           for (int i = 0; i < 20; i++) {
33.               System.out.print("Hello from 1st. thread, my name is " + arg + "\n");// using t
      he passed(arg) value
34.           }
35.       }
36.   }
37.
38.   class SecondIteration implements Runnable {
39.       public void setArg(String arg) {// pass arg either by constructors or methods.
40.           this.arg = arg;
41.       }
42.
43.       String arg;
44.
45.       @Override
46.       public void run() {
47.           for (int i = 0; i < 20; i++) {
48.               System.out.print("2)my arg is=" + arg + ", " + i + "\n");
49.           }
50.       }
51.   }
```

在上面的例子中我们将一个 `String` 值作为参数传递给了线程,但是记住它应该通过宿主类(实现类)完成.

练习

现在轮到你了,尝试以下的操作:

- 一个使两个float数组A,B相乘的应用程序,然后把结果存储在数组C(C[0] = a[0] * b[0])中(提示: 传递数组的引用和索引)
- 从用户那里获得字符串行(s)的一个程序,将它们保存到10个文件中
- 并发创建100个命名的目录(从0到99)的程序
- 一个并发杀死所有其它OS进程的程序(危险!不要实际那么做)

管理线程

嗯,我们已经创建和运行过线程了,但是如何管理它们?如何睡眠(空闲)一个线程(停止一个指定的时间)?我们如何暂停(停止工作直到有一个恢复信号)一个线程?我们如何停止一个运行中的线程?或者我们如何得到当前运行中的线程?

线程的属性

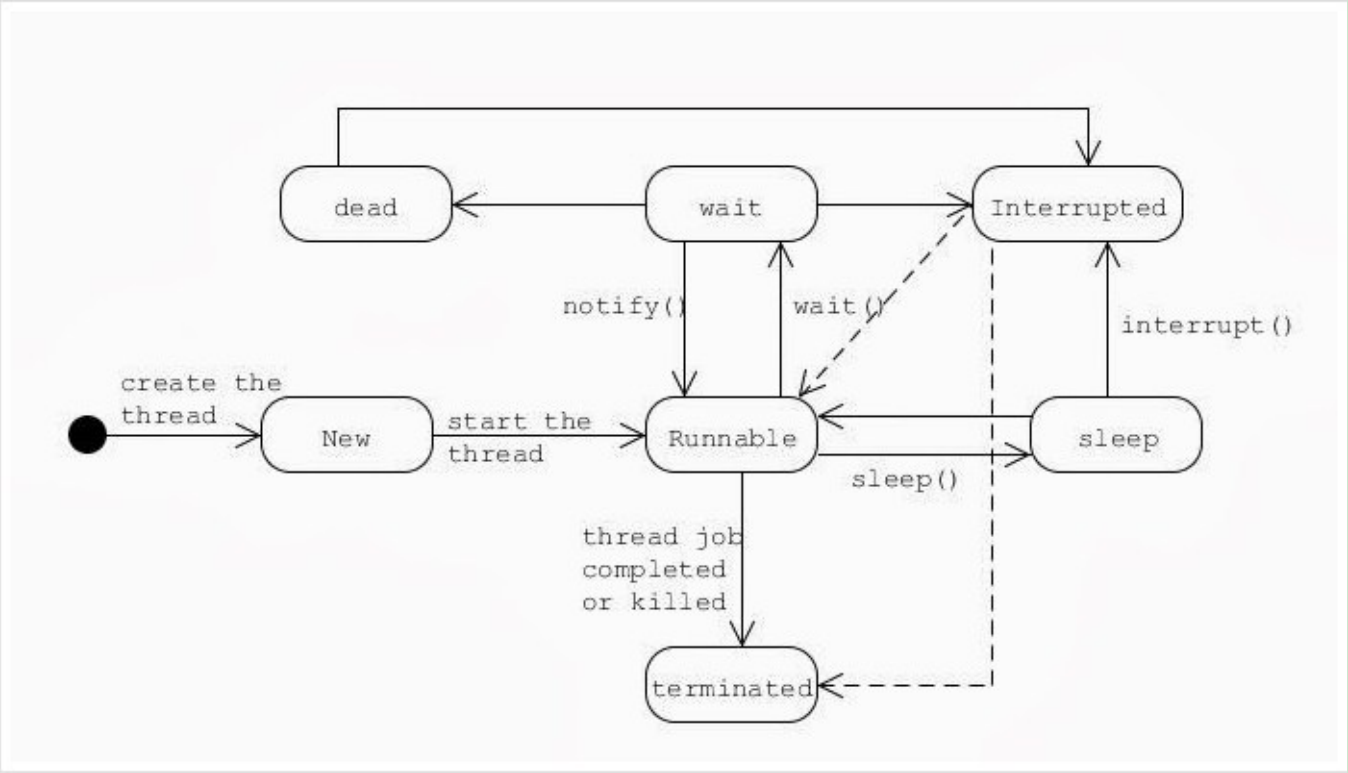
通常在每种语言和平台中,线程有一些共同的属性,比如,线程的id或私有内存(栈).让我们深入地研究下.

属性

- 线程 id
  每个线程有一个唯一的ID(int值),由运行期间或OS提供.这个是一个只读的并且唯一的值.
- 线程名称

用于线程的名称,在许多线程中它可能是相同的.根据一个特定的模块或任务分类线程是非常有用的.

- 线程状态
  线程的当前状态.线程的状态是由 `State` 枚举决定的,是下面中的其中一个:
  - Runnable: 表明线程已经启动,正在运行中.
  - New: 表明线程已经被创建,但是还没有被调用(启动)-当你只是创建了线程的一个实例,还没有尝试去调用 `start()` .
  - Blocked: 当一个线程在处于运行中的状态,然后被阻塞,因为它等待锁定一个已经被另一个线程锁定的资源.
  - Waiting: 当线程空闲,并等待一个信号(通知)的时候.
  - Timed_Waiting: 当线程睡眠一段指定的时间或从另一线程中等待一个会超时的信号.
  - Terminated: 当一个线程已经完成了它的任务,或被杀死.注意一个终止的线程不能再次启动,它已经结束了.
  - Interrupted: 这个表明如果一个中断信号被发送给线程或没有,注意这个不是一个实际的状态,一个线程可以在它运行中或等待中的时候中断(在New或者Terminated状态中不会).你不能通过 `State` 枚举得到中断状态.
  - Dead: 当线程A等待线程B的一个信号,同时线程B也等待线程A的一个信号,或者可能线程B已经发送了信号只是在线程A尝试等待它之前(这个实际上是死锁),here there is nothing left behind to signal the waiting thread.



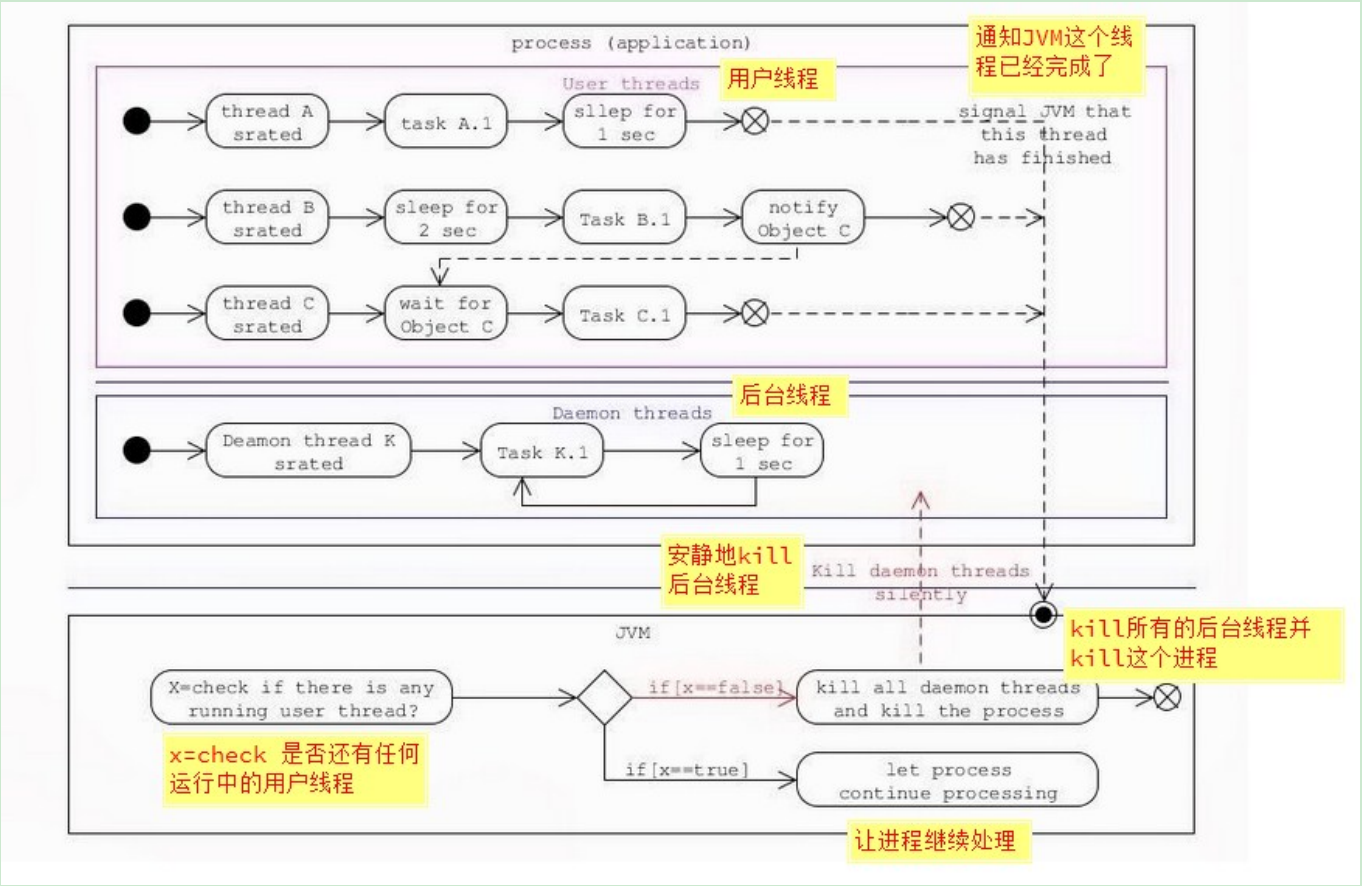
你只需要以一种你保证不会有死锁状态的方式编程你的应用程序.注意没有任何属性或方法可以发现是否一个线程处于死锁状态,所以如果你遇到死锁,这是你不好的编程方式.

注意: 在一些教程中,它们将死锁作为终止状态解释,这是可以的,死锁状态也被称为终止状态,但是根据我的认知,死锁和终止是两个独立的主题.

- 线程的优先级
  它表明线程的优先级,一个高的优先级会给线程引起更多的过程注意(process attention)(上下文切换),可能是以下中的其中一个:
  - MAX_PRIORITY
  - MIN_PRIORITY

- NORM_PRIORITY(默认值)

- 后台进程模式:
  这个一个用于设置线程是否作为一个守护(后台)模式.当一个线程是后台模式,这意味着它运行直到任何其他的用户线程(非后台)在应用程序中是活的.无论何时没有用户线程存活,JVM强制(安静地)杀死后台线程.你不能得到关于退出信号的任何异常.这种类型的线程非常有用当你的应用程序需要有一个后台服务线程或一个事件处理器.这些线程通常以一个低优先级运行.再次说明,有一些线程运行在你的进程中,但是JVM不会把它们作为用户线程用于检查进程的生命周期(查看图解).



如上面的图解所示,无论何时一个用户线程完成了,JVM测试是否还有任何用户线程正在运行中?如果没有,它将安静地杀死每个运行中后台线程(强制地),所以要注意一个后台线程中的变化(操纵),如果你知道它马上就会被杀死,最好在JVM做之前通知它.我们在后面还将关注后台线程,继续阅读.

示例

下面的示例展示了如何设置和获取线程的属性,相当简单.

```
1.  package arash.blogger.example.thread;
2.
3.  public class Core {
4.      public static void main(String[] args) throws InterruptedException {
5.          Runnable r0 = new MyThread();
6.          Thread t0 = new Thread(r0);
7.          System.out.print("T0 thread state before thread start is " + t0.getState() + "\n");
8.          t0.setName("my lucky thread");
9.          t0.setPriority(Thread.MAX_PRIORITY);
10.         t0.start();
11.         Thread.sleep(1000);// wait for 1 sec here
12.         System.out.print("T0 thread state is " + t0.getState() + " right away\n");
13.         t0.interrupt(); // this method will throw a Interrupt exception at the target threa
    d.
```

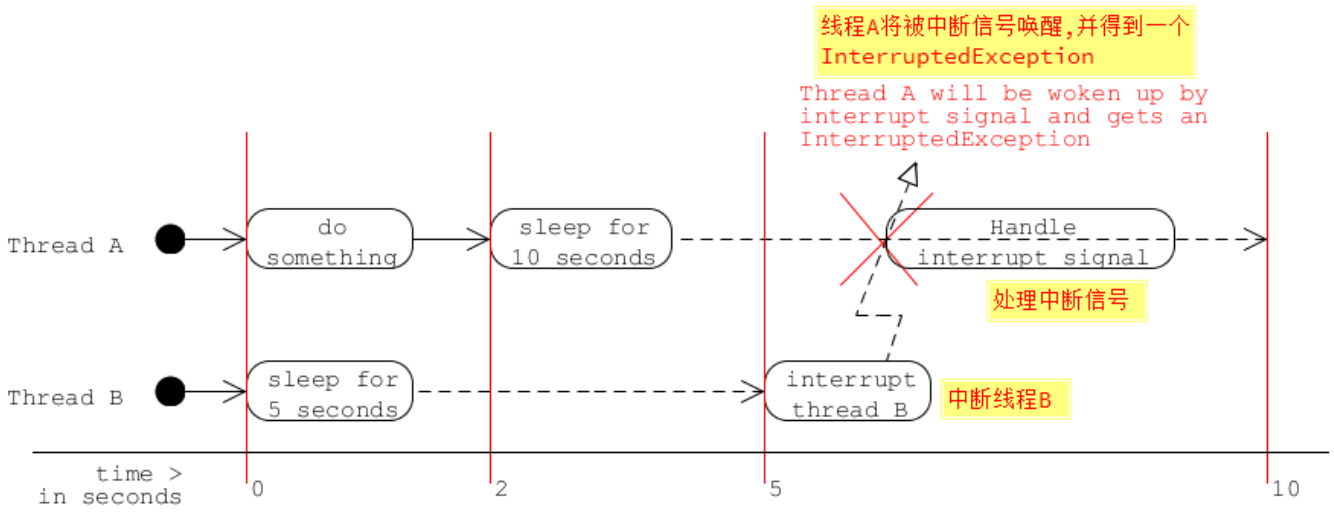
```
            Thread.sleep(5000);
            System.out.print("T0 thread state is " + t0.getState() + " right away\n");
        }
    }

    class MyThread implements Runnable {
        @Override
        public void run() {
            Thread t = Thread.currentThread();// returns the running thread, this thread.
            System.out.print("HI, my thread name is \"" + t.getName() + "\" and my priority is
    \"" + t.getPriority()
                    + "\" and my ID is \"" + t.getId() + "\"\n");
            try {
                Thread.sleep(5000);// wait for 5 seconds here
                System.out.print("Hallelujah, thread has terminated successfully!\n");
            } catch (InterruptedException e) {
                /*
                 * if there is an interrupt signal it defines by
                 * InterruptedException so this catch determine whenever this thread
                 * got interrupt ex.
                 */
                System.out.print("Oops, I've got interrupt message!\n");
            }
        }
    }
```

线程的行为

好了,除了属性外,一个线程有一些特殊的方法,其中一些属于 `Thread` 类,有一个属性 `Thread` 实例.线程的基本行为(方法)列在下面.

- Start(同时): 当线程需要专门的栈同时运行,这个操作通过 `start()` 方法完成.
- Sleep: 无论何时一个线程需要睡眠一段指定的时间, `Thread.sleep(<<time in ms>>)` 将起到作用,注意不要担心准确的时间.休眠的处理(提醒时间)可以跳过如果另一个线程中断了睡眠的线程.
- Interrupt: 这个方法用在当另一个线程需要去中断另一个处理 `wait(和join()` , `sleep` , 或者可运行状态的线程的时候.比如,线程A正在等待一个通知,线程B中断线程A,这里是中断的信息:

    1. 唤醒目标线程如果它处于 `wait(wait/join)` 或 `sleep` 状态.
    2. 抛出 `InterruptedException` 给目标线程,为了提醒它有中断信号除非被中断的线程处于运行中的状态.
    3. 设置中断标志位true,如果它处于运行中状态的时候得到一个中断信号



这里是代码：

```
1.  /**
2.     * by Arash M. Dehghani
3.     * arashmd.blogspot.com
4.  */
5.  package arash.blogger.example.thread;
6.
7.  public class Core {
8.      public static void main(String[] args) {
9.          Runnable r0, r1;
10.         r0 = new ThreadA();
11.         Thread threadA, threadB;
12.         threadA = new Thread(r0);
13.         r1 = new ThreadB(threadA);// pass the thread A ref to thread B for interruption
14.         threadB = new Thread(r1);
15.         threadA.start();
16.         threadB.start();
17.     }
18. }
19.
20. class ThreadA implements Runnable {
21.     @Override
22.     public void run() {
23.         System.out.print("A:I'm doing some operations\n");
24.         // .....some work
25.         try {// try to sleep
26.             System.out.print("A:let's sleep for 10 sec\n");
27.             Thread.sleep(10000); // sleep for 10 sec
28.         } catch (InterruptedException e)// while in sleep mode, interruption
29.                                         // would happened
30.         {// handle if interrupted, do alternative tasks
31.             System.out.print("A:Oops, someone sent the interruption siganl and I will finis
    h myself.");
32.             return;// terminate the thread
33.         }
34.         System.out.print("A:I have to be a lucky guy, no any interruption message! thanks \
    n");
35.     }
36. }
37.
```

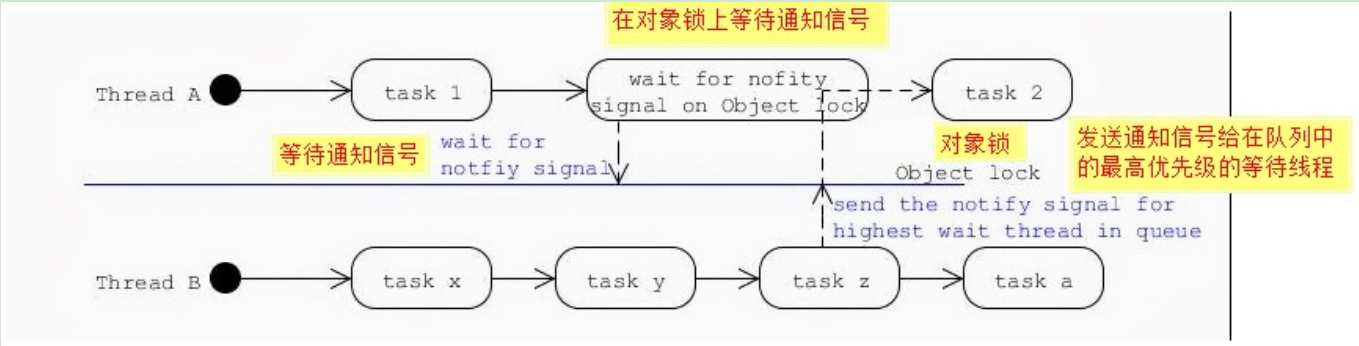
```
38.   class ThreadB implements Runnable {
39.       public ThreadB(Thread t) {
40.           this.threadAref = t;
41.       }
42.
43.       Thread threadAref;
44.
45.       @Override
46.       public void run() {
47.           try {
48.               Thread.sleep(5000);// sleeps for 5 sec
49.               threadAref.interrupt();// comment it and see the differences
50.           } catch (InterruptedException e) {
51.               System.out.print("B: I've got an interruption signal, but I will continue my ta
      sk!\n");
52.           }
53.           System.out.print("B: nothing interesting here!\n");
54.       }
55.   }
```

如上面的示例所示,两个线程A和B同时启动它们的任务,线程A尝试做一些任务,这将花费2秒,然后睡眠10秒,同时线程A睡眠3秒然后唤醒和中断线程A.

注意当一个线程处于sleep,join和wait状态:

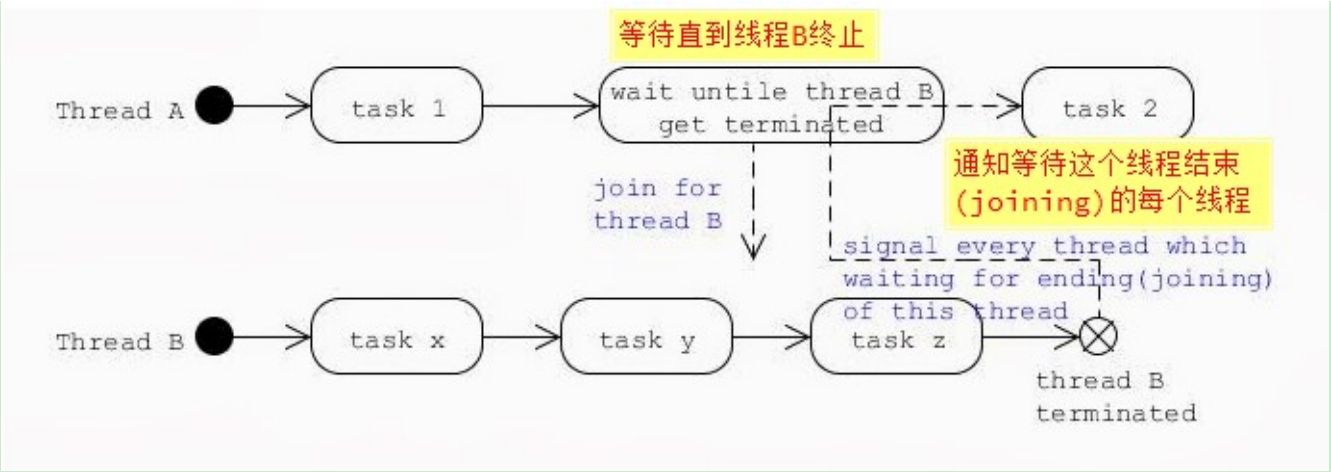
```
1.   - Sleep,join,或wait操作应该使用 `try/catch` 操作包围,因为有中断信号(异常)
2.   - 中断就像一个信号,它不能强制线程终止
```

- 得到当前的线程:你可以通过 `Thread.currentThread()` 访问当前的线程,它返回当前执行线程对象的引用,但是在线程之外你可能需要检查所有的线程或者有一个线程的引用.
- `wait` , `notify` , `notifyAll` :这些都是用来在线程间同步,我们将更详细地讨论它,但是如果我想解释一下这可能是:无论何时两个或n个线程需要互相同步,需要有一个共享的对象属于它们,现在假设线程A已经完成了操作1,然后它需要等待线程B去完成操作2,所以线程A等待通知对象锁的通知,然后只要线程B完成了操作B,它通过对象锁发送通知(脉冲-pulse)给线程A.通过文字很难解释,看下面的图解:



注意:有可能有多个线程正在等待一个对象的信号.当3个线程等待一个对象,它们站在通知队列中(不完全是排队,同样依赖于优先级),(it means for very next single notify signal),一个线程将从队列中移除被得到通知,但是 `notifyAll()` 释放在队列中的所有的线程,我们将在这节关注,继续阅读.

- join:它的意思和它所说的一样,当线程A join 线程B,线程A等待直到线程B结束(终止)




```java
/**
 * by Arash M. Dehghani
 * arashmd.blogspot.com
 */
package arash.blogger.example.thread;

public class Core {

    public static void main(String[] args) {
        Runnable r0, r1;
        r0 = new ThreadB();
        Thread threadA, threadB;
        threadA = new Thread(r0);
        r1 = new ThreadA(threadA);// pass the thread A ref to thread B for interruption
        threadB = new Thread(r1);
        threadA.start();
        threadB.start();
    }
}

class ThreadA implements Runnable {
    public ThreadA(Thread t) {
        this.threadBRef = t;
    }

    Thread threadBRef;

    @Override
    public void run() {
        try {
            System.out.print("A: Waiting for thread B get finished\n");
            threadBRef.join();// wait until threadB is alive
            System.out.print("A: thread B has completed almost");
        } catch (InterruptedException e) {// like the wait and sleep, join could get the in
terrupt signal too
            System.out.print("A: interrupt signal has received!\n");
        }
    }
}

class ThreadB implements Runnable {
    @Override
    public void run() {
```

```
            try {
                Thread.sleep(10);// wait a bit, let the ThreadA goes first (not recommended)
                System.out.print("B:doing some work");
                for (int i = 1; i < 10; i++) {
                    System.out.print('.');
                    Thread.sleep(1500);
                }
                System.out.print("\n B: Operation completed\n");
                Thread.sleep(2500);
            } catch (InterruptedException e) {
                System.out.print("B:interrupt signal has received!\n");
            }

        }
    }
```

- Yield：当一个线程礼让(yield),这意味着OS可以忽略这个线程,然后切换(上下文切换)到另一个线程.所以主机(OS)可以有机会切换到另一线程(如果有的话,并且如果必要,通常是相同的优先级),但是它和睡眠0秒不完全一样.比如,有两个运行中的线程有相同的任务和优先级和启动时间,然后你想尝试让它们一起完成(至少时间上的差异很小).在某个时刻线程A意识到它已经完成了工作的10%,同时线程B只完成了4%.睡眠一会儿即使是10ms也不是一个好的主意,因为它可能引起线程B完成任务的30%.所以我们可以告诉主机,它可以忽略这个线程(A),然后检查另一线程进行切换,然后再切换回来.

  对于下面的示例我很抱歉,因为这个对于我找到一个 yield 的完美的例子有点难,但是当你多次运行下面的例子,你将会面临不同的响应(因为yield没有一个特定的行为-because yield doesn't have a certain behavior),只是评论yield任务(just comment the yield task)和看下差异.

```
/**
 * by Arash M. Dehghani arashmd.blogspot.com
 */

package arash.blogger.example.thread;

import java.io.FileNotFoundException;
import java.io.PrintStream;

public class Core {
    public static ThreadA threadA;
    public static ThreadB threadB;

    public static void main(String[] args) throws InterruptedException, FileNotFoundExcepti
on {
        System.setOut(new PrintStream("./out.res"));
        Runnable r0 = null, r1 = null;
        Thread a, b;
        r0 = new ThreadA();
        r1 = new ThreadB();
        threadA = (ThreadA) r0;
        threadB = (ThreadB) r1;
        a = new Thread(r0);
        b = new Thread(r1);
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.flush();
    }
```

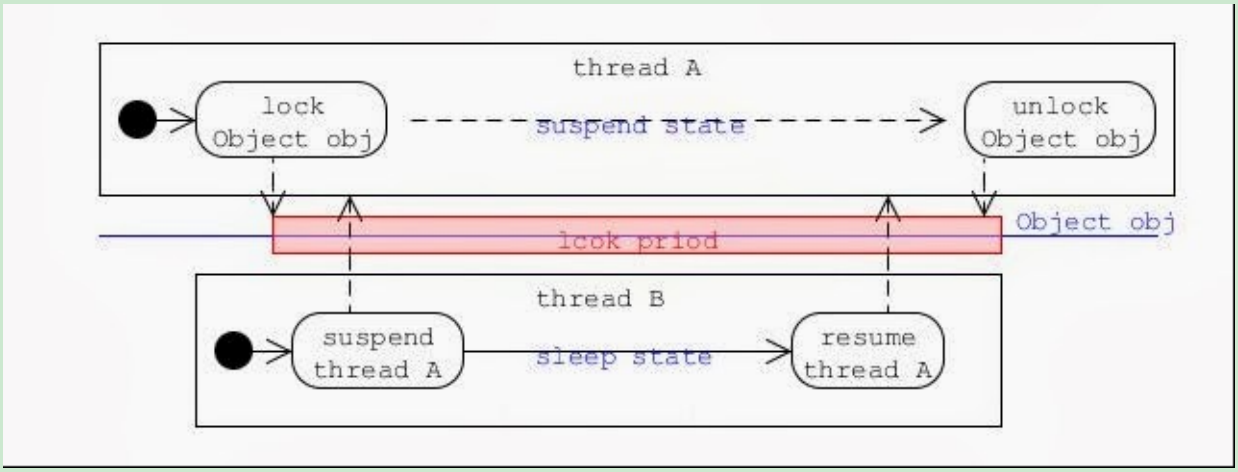
```
30.   }
31.
32.   class ThreadA implements Runnable {// imagine that two thread A and B cannot sync with othe
      r implicitly
33.       public int progress = 0, i = 0;
34.
35.       @Override
36.       public void run() {
37.           System.out.print("Hello from 1st. thread\r\n");
38.           for (; i < 10000; i++) {// this is possible this thread perform even 2000+ loop in
      a switch(turn)
39.               if (Core.threadB.progress + 5 < this.progress && Core.threadB.progress != 100)
      {
40.                   System.out.print("A: thread B is out of the date, try to switch [B(thread,
      loop):"
41.                           + Core.threadB.progress + " , " + Core.threadB.i + "  A(thread, loo
      p):" + Core.threadA.progress
42.                           + " , " + Core.threadA.i + "]\r\n");
43.                   Thread.yield();// tell host, ignore me now(a bit) and check the others
44.               }
45.               if (i % 100 == 0) {
46.                   System.out.print("Thread A, progress :" + (i / 100) + "\r\n");
47.                   progress++;
48.               }
49.           }
50.       }
51.   }
52.
53.   class ThreadB implements Runnable {
54.       public int progress = 0, i = 0;
55.
56.       @Override
57.       public void run() {
58.           System.out.print("Hello from 2nd. thread\r\n");
59.           for (; i < 10000; i++) {
60.               if (Core.threadA.progress + 5 < this.progress && Core.threadA.progress != 100)
      {
61.                   System.out.print("B: thread A is out of the date, try to switch [B(thread,
      loop):"
62.                           + Core.threadB.progress + " , " + Core.threadB.i + "  A(thread, loo
      p):" + Core.threadA.progress
63.                           + " , " + Core.threadA.i + "]\r\n");
64.                   Thread.yield();// tell host, ignore me now(a bit) and check the others
65.               }
66.               if (i % 100 == 0) {
67.                   System.out.print("Thread B, progress :" + (i / 100) + "\r\n");
68.                   progress++;
69.               }
70.           }
71.       }
72.   }
```

上面的代码不是一个很酷的示例,但是只要检查输出文件中的结果,并假想什么正在进行.记住如果有两个线程不能互相同步,使用 `yield()` 方法可能不是很好的选择.你需要使用另一种模式和方法替代,我们稍后讨论.

其它的线程行为

我们已经提到了最常用(新的形式)的线程的方法和行为,但是仍然有3个废弃(老的形式)的方法,你需要知道和新的方法的差异.

- Suspend和Resume: `wait` 和 `notify` (同步)旧的形式,但是有一个非常大的差异,首先 `suspend` 暂停线程(和 `wait` 所做的一样),但是不会解锁任何的同步(锁定的)资源,此外对于恢复线程而言,线程的引用需要去调用 `resume` 方法.而在 `wait` 和 `notify` 方法中你只需要锁定对象的引用.



同步线程

嗯,我可能会说这是这个教程中最重要的一个小节,同步两个线程相当重要当你开发一个多线程程序.对于同步两个线程,你需要一个非null的共享对象(不是基本类型).

锁对象通常是实际 `对象(Object)` 类,这个对象(或多个对象)应该对想要互相同步的线程可见(共享),所以不要忘记传递它们给目标线程.比如,线程A在锁对象上等待线程B的通知信号,这将引发线程A从等待状态退出然后继续运行.

注意!!!

- 多个线程可能在一个锁对象上等待,一个通知调用只会释放等待队列中的一个线程(依赖于优先级,和JVM选择)
- `notifyAll` 引起所有的等待线程从队列中释放

如何同步两个互相同步的线程?

Okay,时候到了,一个简单的示例,有助于开始.

示例

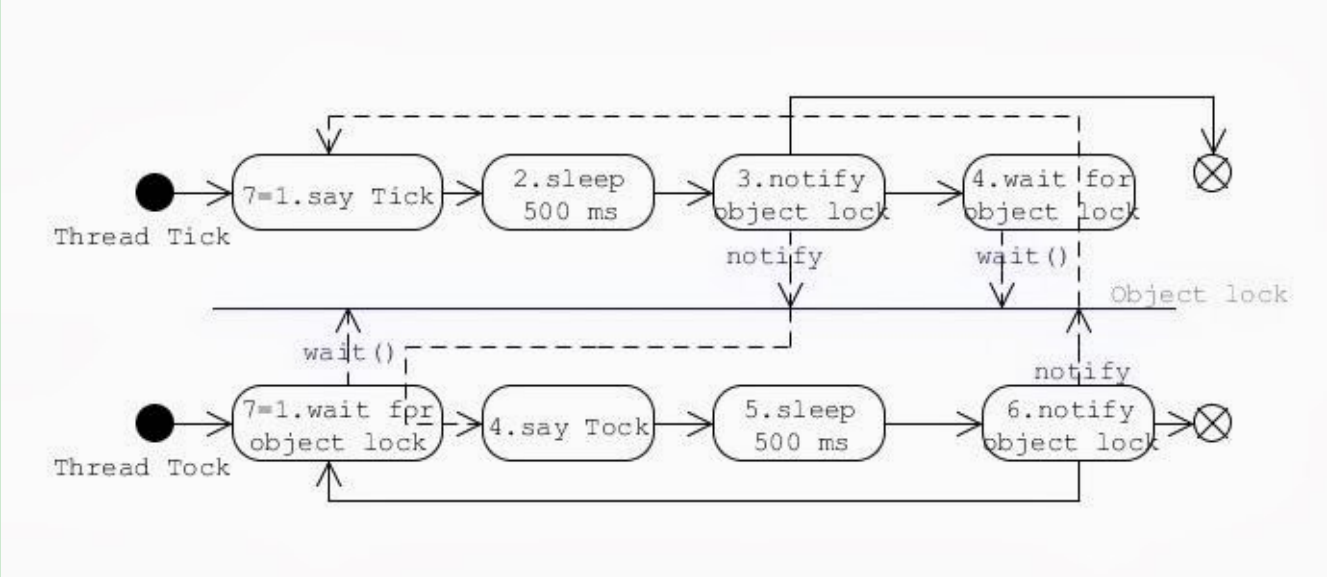
考虑一个 Tick Tock(滴答)应用程序.首先尝试通过一些简单的问题分析案例研究,比如,起始点在哪里?它是否需要线程?有多少线程?它是否需要同步?等等...

一个 Tick Tock 应用程序以发出 Tick(滴答声)开始,然后0.5秒等待,接着发出滴答声,再等待0.5秒,然后重新开始...,Tick, Tock, Tick, Tock,.....,然后让我们清理下流程;

1. 创建一个对象锁用于同步
2. 创建两个线程Tick,和Tock,然后传递锁对象给它们
3. 并行启动两个线程
4. (在一个 `for` 循环中,Tick发出滴答,同时,Tock在锁对象等待线程Tick的通知信号
5. Tick休眠0.5秒,而Tock仍然等待Tick信号
6. Tick发送通知信号给对象锁,Tock从锁中释放
7. 现在Tock发出滴答,同时Tick在锁对象等待线程Tock的通知信号
8. Tock休眠0.5秒,而Tick仍然等待Tock信号

9. Tock发送通知信号给对象锁,Tick从锁中释放
10. 回到步骤4!知道循环结束

图解可能是这样:



现在在我们看代码之前,我们需要知道Java中的锁,它可能比较容易.

同步块

在Java中有时候我们需要锁定一个资源,因为我们需要保当它锁定的时候没有任何线程在那个对象上读或写,所以这里一个锁定操作需求相关的线程.这是相当简单的,就像这样,使用同步块.

```
1.  synchronized(<<lock_Object>>){
2.  //here no any thread won't access (read, write) the lock_Object except the current thread.
3.  // 这里没有任何线程可以访问(读,写)锁对象,除了当前线程
4.  }
```

对于在一个锁对象上的等待和通知,它应该在 `wait` , `notify` , 或 `notifyAll` 之前锁定(同步).如下.

```
1.  //Object will be locked by this thread if it's free to lock, else has to wait until release
    d!
2.  synchronized(lock_Object){
3.      lock_Object.wait();//release the lock, and acquire it again after get notified
4.  }
5.  //lock_Object2.wait();// Error!, should get synchronized like above for either waiting or n
    otifying
```

重要提示:如我们所说的,当以线程锁定(同步)一个资源,没有任何线程可以访问那个资源.

如果一个线程调用一个对象(锁)的 `wait()` 方法,那个对象释放(释放锁)对象为了允许另一个线程去获取锁,和通知等待中的线程.这个引发不必要的 `joining` , `sleeping` , 和 `yield` 方法.

如果在一个同步块中,线程休眠了10年,这意味着同步对象被锁定了10年, `sleeping` , `joining` , 和 `suspending` 处理不会释放任何锁定的资源,好了,我们将用所有知道的去实现我们的例子.

`volatile` 关键字

提示

在编程中我们有两种方式更新一个变量,直接-修改(`direct-modify`),和读-修改(`read-modify`).唯一的差

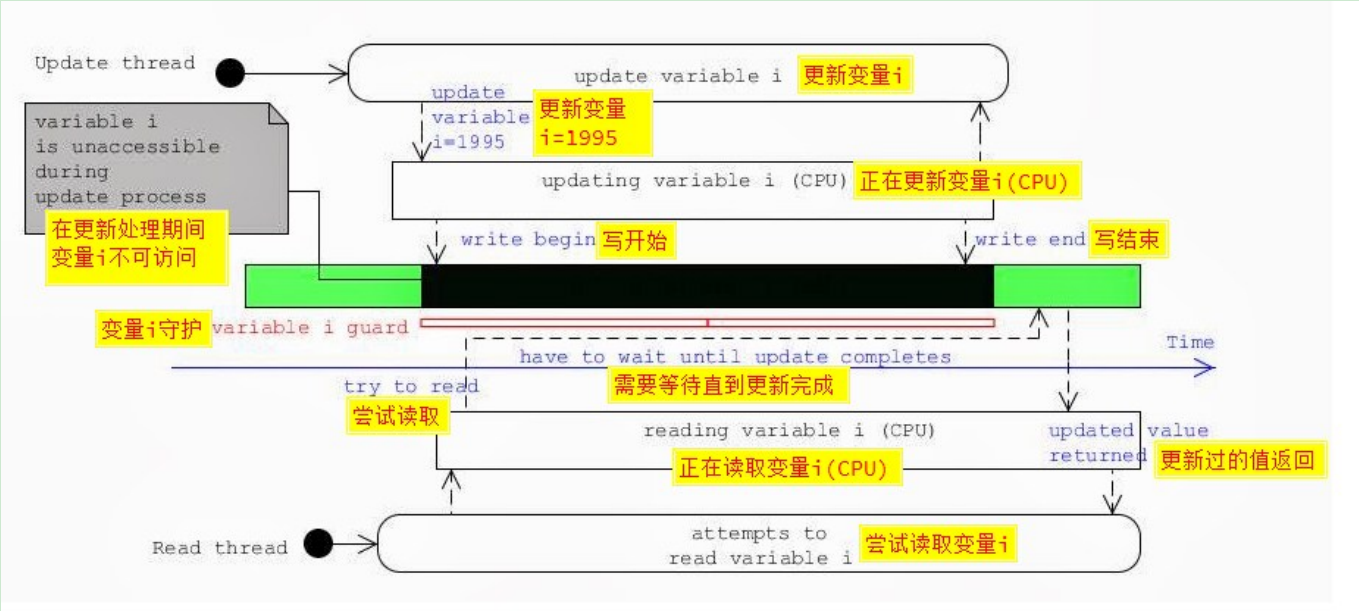
异是在 read-modify 模式中,数据被读取, 然后被更新(比如 `i = i + 10; a++; a +- 1990;`),但是在 direct-modify 模式中,数据被直接更新而不会依赖于之前的值(比如 `i=1990;a="FOLKS!"`).

案例研究

在并行程序中,有可能有两个线程想并发去访问(set 或 get)一个变量,比如两个线程,一个更新值 `i` ,同时另一个想去读取它,所有这里可能出现不一致,因为当变量开始被更新,另一个线程尝试去读取它,因为可能是就的值被读取.
所以(默认方式)不能保证强制第二个线程去等待更新中的线程结束更新,然后读(等待)新的值.

解决方案

解决这个问题,我们必须在每次更新的时候锁定变量,但是有另一种(更简单的)解决方案用于 direct-modify 更新去锁定变量,在更新期间通过设置变量为 `volatile` .
记住, `volatile` 不会锁定对象.它在当有一个线程(s)更新(direct-modify)一个变量,同时一个线程(s)需要读取它(比如一个拍卖系统)的时候被使用.这里 `volatile` 保证读取线程将得到最后更新的值,然后线程(s)将等待直接所有更新中的线程结束(如果有的话).



现在我们该如何使用 `volatile` ? 什么时候必须使用同步块替换?

- 如果一个变量被直接更新(比如 `i=1.59F` , `i='A'` , `i=0Xa19caa90e4d`), 使用 `volatile` .
- 如果一个变量基于当前值被更新(比如, `i = i+1` , `i %= 7` , `i = 1.9944D - i * 2.1`), 使用同步线程/块.

  替换所有的 `volatile` 方式为同步方式是可能的.一个 `volatile` 变量应该是基本类型或对象也可以是null,不过同步块我们只能锁定非null 的对象.

如何设置一个变量为 `volatile` ?

很简单,在变量声明的时候使用 `volatile` 关键字.

```
1.   public class Invoker{
2.       public volatile Object obj;    //it could be null - 可以为 null
3.       private volatile int id;    //it could be primitive too - 也可以是基本类型
4.       public int getId(){
5.          return this.id;    // because id is volatile, so if there is an update in progress,
6.          // it(current thread) waits for the last update, then return the updated value.
```

```
7.
8.          // 因为 id 是 volatile,所以如果在进程会中有一个更新,
9.          // 它(当前线程)等待最后的更新,然后返回更新过的值
10.      }
11.  }
```

示例

现在让我们用一些示例,决定使用 `volatile` 还是 同步的方式.

最后登录的用户

考虑一个应用程序追踪(set 和 get) 最后登录到系统的用户 id.首先我们需要分析,它是否接收异步请求?当然,数据可以被并发更新或读取.Okay,下面我们需要理解用户 id 如何更新?变量以 direct-modify 模式更新,换句话说,它不依赖于当前的值,这里 `volatile` 方式可以有助于设置和获取值.

```
1.  public class SysMgr{
2.      private volatile long lastUserId=-1L;
3.
4.      public void set(long lui) {
5.          this.lastUserId=lui;
6.      }
7.
8.      public long get() {
9.          return this.lastUserId;
10.     }
11. }
```

如果一个线程(A)想获得 lastUserId 值,同时另一个线程(B)正在更新它,这样线程 A 将等待(阻塞) 直到线程 B 完成更新,然后返回新的值给线程 A,这些都是因为 `volatile` 关键字.

总共的登录次数

一个应用程序可以统计, 系统被登录了多少次.
这里例子和之前的有点像,除了一件事.对于统计你需要去更新一个当前值.也就是说,更新进程是 read-modify 模式.对于这种情况, `volatile` 没有什么帮助,因为 `volatile` 不会锁定变量当它在在处理器级别(寄存器/缓存)中更新的时候.但是它是十分方便的,如果需要一个统计重置功能.代码将会像这样,变量定义将是 volatile,对于读者和直接-设置值,同步方法保证只有一个线程正在更新值.

```
1.  public class Statistic{
2.      private volatile long logins = 0L;
3.
4.      public synchronized void newLogin() {
5.          logins++;
6.      }//jus one thread is allowed to access this method
7.      // 只有一个线程允许去访问这个方法
8.
9.      public long getValue() {
10.         return this.logins;
11.     }//the variable logins is volatile, so it returns the very last update value
12.     // 变量 logins 是 volatile, 所以它返回最后更新的值
13.
14.     public void reset() {
15.         this.logins=0;
16.     }//it could be without locking, because the update is direct-modify
```

```
17.       // 可以不需要锁定,因为更新的 direct-modify 的
18.
19.   /*
20.   * some members
21.   */
22.   }
```

`getValue()` 不是同步的,这不是必须的,因为对于任何读取,这个能保证你提交的(新的)值将被返回.注意,`getValue()` 方法必须是同步的,如果 logins 变量不是 `volatile` .

相对和绝对的更新

另一个例子

```
1.   public class Mgr {
2.       private volatile long l = 0L;
3.
4.       public synchronized void increment() {
5.           l++;
6.       }//this is read-modify update, so needs the lock
7.       // 这是 read-modity 更新,所以需要锁定
8.
9.       public void set(long l) {
10.          this.l=l;
11.      }//this is just modify update, so could be done with volatile
12.      // 这仅仅是修改更新,所以可以使用 volatile 完成
13.  }
```

说明

Okay Okay Okay, 我想说现在你可以自己动手编写任何的多线程应用程序了,但是仍然有一些重要的事情,你需要知道.比如模式,问题,管理等...

Main 线程

如果一个应用程序有一个 `main()` 方法,JVM创建一个名为 `main` 的线程,然后使用这个线程调用 `main()` 方法.

创建线程

在Java中有两种方式创建(声明)一个线程,实现 `Runnable` 接口或继承 `Thread` 类,第一个方式更常用.像下面这样:

```
1.   class MyThread implements Runnable{    //recommended - 推荐
2.       //@Override
3.       public void run() {
4.       /*Your business goes here*/
5.       /* 你的业务逻辑 */
6.       }
7.   }
```

第二种方式

```
1.   class MyThread2 extends Thread{
```

```
    //@Override
    public void run() {/*Your business goes here*/}
}
```

守护线程

如我提到的,在Java中有两种(运行方法)线程,用户-线程,和守护-线程.JVM 杀死和清理你的应用程序的内存当没有任何的用户-线程运行的时候,同样安静地杀死守护-线程(如果有的话).所以注意下守护-线程.
尽量不改变一些事情如果你的应用程序可以立即结束,因为守护-线程不会接受到任何的中断或线程死亡信号.但是如果你隐式地通知它们在你所有的用户线程终止之前.

```
package arash.blogger.example.thread;

/**
 * by Arash M. Dehghani arashmd.blogspot.com
 */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        Thread y = new Thread(t);
        y.setDaemon(true);
        // you have to set it before you start it!
        // set it as false and run the code again, see the differences
        // 你必须在启动它之前设置
        // 将它设置为 false, 然后再次运行代码, 看下差异

        y.start();
        Thread.sleep(5200);
        System.out.print("The very last result from our daemon-thread: " + t.res + "\n");
        System.out.print("Main thread has terminated, no any user thread available no more!
\n");
    }
}

class MyThread implements Runnable {
    long res = 0X0L;

    @Override
    public void run() {
        {
            System.out.print("Hello, I am a daemon thread, wohaha!\n");
            while (true) {
                try {
                    for (int i = 0; i < 10; i++) {// if I be a lucky thread, I will reach t
he 10 - 如果我是一个幸运地线程,我将达到10
                        Thread.sleep(1000);
                        res += i;// it's unsafe, but we don't care, not important. - 不安全
的,但是不关心,也不重要
                        System.out.print("Res variable value now -->" + res + "\n");
                    }
                } catch (Exception | ThreadDeath e) {// catch for any stop() or interrupt()
 signal - 捕获任何的 stop() 或 interrupt() 信号
                    System.out.print("I will never get any signal by JVM, for-ever daemon >
:D \n");
                }
            }
        }
```

```
42.         }
43.     }
```

你必须在一个线程启动之前将它设置为守护线程. `MyThread` 有一个循环,需要 10 秒完成,但是因为它是一个守护线程,它将被 JVM 杀死, 当 `main` 线程终止的时候.

终止一个线程

同样如我提到的,终止一个线程没有任何特定的方法,所以实际上你不能强制一个线程中.所以你需要设置一种方式去通知一个线程停止工作.一种方式是使用 `interrupt()` 方法和线程的 `interrupted` 标志状态,这是不推荐的,因为中断是被设计用来唤醒一个线程的,当它处于 sleep, join, 或 wait 状态的时候.
注意当我们调用 `interrupt` 方法,我可能有不同的行为.

- 如果一个线程处于 wait, join, 或 sleep 状态,然后剩下的睡眠/等待状态将被忽略,一个 `InterruptedException` 异常将发送给受害者线程(victim thread).
- 如果一个线程被某些 IO 操作阻塞,中断信号将被放入队列,一旦一个线程阻塞停止,该信号将受到影响.
- 如果一个线程被某些 NIO 操作阻塞,然后剩下的睡眠/等待状态将被忽略,IO 通道被关闭, `ClosedByInterruptException` 由 JVM 发送给线程.
- 如果一个线程处于 runnable 状态,只是将 `interrupted` 将被设置为 true.
- 如果一个线程处于 new 状态(还没有启动),什么都不会发生!.

所以我们可以使用 `interrupted` 标记作为终止标记通知受害者线程终止工作.

```
1.  package arash.blogger.example.thread;
2.
3.  /**
4.   * by Arash M. Dehghani arashmd.blogspot.com
5.   */
6.  public class Core {// Just run it for several times, you will faced with different results
    - 多运行几次,你将面临不同的结果
7.
8.      public static void main(String[] args) throws InterruptedException {
9.          MyThread t = new MyThread();
10.         Thread y = new Thread(t);
11.         y.start();
12.
13.         Thread.sleep(2);    // wait 2 ms, see how much your system strong is in 2 ms? - 等待
    2ms,看下你的系统在2ms内有多牢固?
14.
15.         y.interrupt();        // set the interrupted flag to true - 设置 interrupted 标志为
    true
16.
17.         System.out.print("Main thread: I've send the interrupt(shutdown) signal\n");
18.     }
19. }
20.
21. class MyThread implements Runnable {
22.     int i = 1;
23.
24.     @Override
25.     public void run() {
26.         while (Thread.currentThread().isInterrupted() == false) { //until this thread hasn'
    t got any interrupt signal - 直到这个线程还没有得到任何的中断信号
27.             double a = 0.0D;
28.             for (int k = 100; k > i; k--) {
29.                 a = k * k * 2 / 5 * Math.pow(2, k);// some operations.....
```

```
                a = Math.sqrt(a);
            }
            System.out.print("Res(" + i + "): " + a + "\n");
            i = i + 1;
        }
        System.out.print("I've got an interrupt signal :(\n");
    }
}
```

如你所见， `MyThread` 持续工作直到另一个线程设置它的 `interrupted` 状态为 true.注意在 `MyThread` 中,没有任何的 `try/catch` ，因为如我提到的,如果一个线程处于 running 状态， `interrupt()` 方法只是设置 `interrupted` 标记为 true.

虽然上面的代码工作没有任何问题，但是还是推荐使用一种隐式的方式去通知线程停止工作.

```
package arash.blogger.example.thread;

/**
 * by Arash M. Dehghani arashmd.blogspot.com
 */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        Thread y = new Thread(t);

        y.start();

        Thread.sleep(2);

        t.shutdown();
        System.out.print("Main thread: I've send the interrupt(shutdown) signal\n");
    }
}

class MyThread implements Runnable {
    private volatile boolean shutdown = false;// use a user shutdown flag - 使用用户 shutdown 标志

    public void shutdown() {
        this.shutdown = true;
    }// a method instead of interrupt method for sending the shutdown signal.
    // 替代的 interrupt 的方法用于发送 shutdown 信号

    int i = 1;

    @Override
    public void run() {
        while (this.shutdown == false) {// until shutdown signal hasn't sent - 直到 shutdown 信号还没有发送
            double a = 0.0D;
            // some big load!
            for (int k = 100; k > i; k--) {
                a = k * k * 2 / 5 * Math.pow(2, k);
                a = Math.sqrt(a);
            }
            System.out.print("Res(" + i + "): " + a + "\n");
            i = i + 1;
        }
```

```
            System.out.print("I've got an interrupt signal :(\n");
        }
    }
```

wait(1000) VS. sleep(1000)

我们已经提到过 `wait()` 和 `sleep()` 方法,让我们回顾下这两种方式的行为.

wait()

它释放 `synchronized(locked)` 对象,等待直到另一个线程通知(notify, stop 或 interrupt)它,然后在它得到通知后(如果被通知),它尝试去再次获取锁,然后继续工作.注意一个 waiting 的线程需要在它得到通知后在关联的对象上再次获取它的锁.

sleep()

它不会释放任何的锁定对象,休眠指定的时间,但是这个时间可以被跳过(唤醒),通过 stop 或 中断信号.

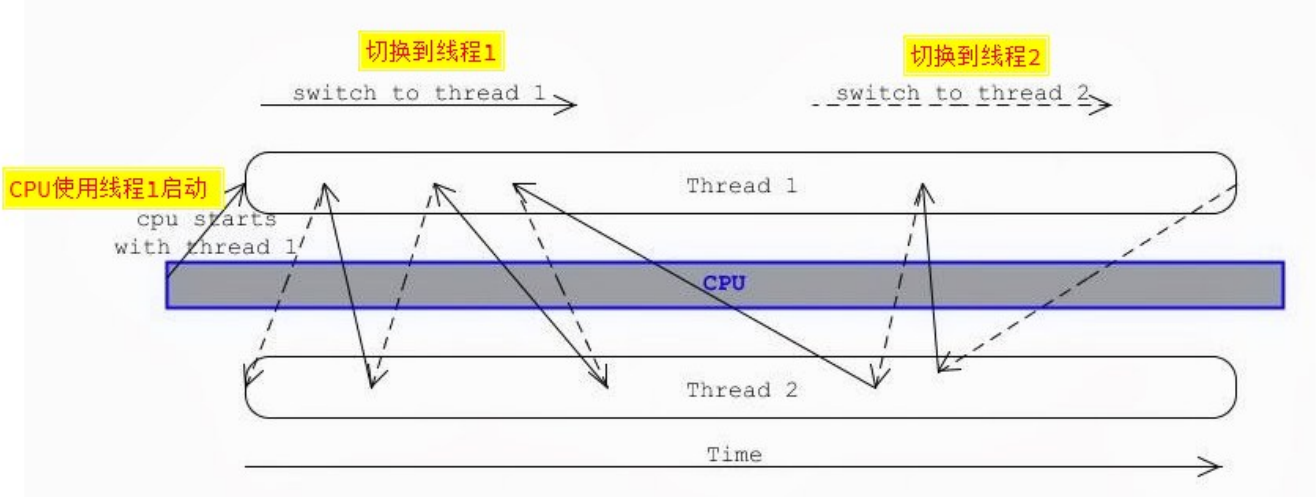
但是这里我们可以使用一个超时时间等待(`wait(max_of_wait_time)`)一个通知信号.这意味线程释放锁定的对象,然后等待通知信号,但是它将在超时的时候通知它自身如果没有任何线程通知它.

```
synchronized(lockObj){
   lockObj.wait();//releases the lockObj(just lockObj) and waits for notify signal, even if
 it could be 2 years
                  // 释放 lockObj(只有 lockObj),等待通知信号,即使它可能是2年
}

synchronized(lockObj){
   lockObj.wait(20000);//releases the lockObj and waits for notify signal,
   // but it will notify (and acquires lock again) itself after 20 sec

   // 释放 lockObj,等待通知信号
   // 但是它将通知(再次获取锁)它自身,在20秒后
}

synchronized(lockObj){
   Thread.sleep(20000);//just sleep for 20 sec
                       // 只是休眠20秒
}
```

在 `wait(1000)` 和 `sleep(1000)` 之间的唯一差异是 , `wait(1000)` 休眠1秒,然后释放锁定的对象,而 `sleep()` 不会释放锁.

在线程之间切换(不受信任的 yield())

切换是什么意思?这是一个很简单的主题,单个线程/单核CPU在一个时刻只能运行一个任务.所以对于运行一个并行应用程序或某些进程,它需要切换到另一个线程/进程,然后做一些其它线程的任务.但我们是如何看到一切都是顺畅的?因为实际上一个CPU很强大(cheers),在线程间切换可能甚至少于10纳秒!
当你想在你的线程之间切换,你需要通过 wait 和 nofity 隐式(直接)地这么做或明确(间接).
我们使用 notify/wait 的方式当运行中的线程的顺序对我们来说是重要的,比如,必须在步骤C之后运行步骤B.下面展示的图解是在两个未同步线程之间的切换,但是注意它甚至可以完全完成线程1,然后切换到线程2,或者线程1做了1^,切换到线程2,完成了100%,然后切换回到线程1,完成线程1,太多的可能性了.
如你看到的,不能保证你的两个(同一时间启动)并发的线程在同一时刻完成,除非你让它们互相同步.



但是当没有(没必要)任何 wait 和 notify 方法,如何切换(尝试切换)到另一个线程?我们提到过 `yield()` 方法,可以方便的完成,但是这种方式有点不值得信任, yield 调用一个底层OS的函数在线程之间切换,它说系统可以跳过当前(这个)线程,并切换到另一个.但是有时候你看到没有任何差异,OS简单的忽略调用.

```
1.  public class Core {
2.      public static void main(String[] args) throws InterruptedException {
3.          new Thread(new MyThread()).start();
4.          new Thread(new MyThread2()).start();
5.          // no one knows when this print will get invoked! after thread 1? or even 2? or eve
    n after them?
6.          // 没人直到这个打印何时被调用!在线程1之后?或甚至线程2之后?或在它们之后?
7.          System.out.print("Main Thread: I started the threads :)\n"); // 主线程语句 [S]
8.      }
9.  }
10.
11. class MyThread implements Runnable {
12.     @Override
13.     public void run() {
14.         for (int i = 0; i < 20; i++) {
15.             System.out.print("Thread 1: " + i + "\n");
16.         }
17.     }
18. }
19.
20. class MyThread2 implements Runnable {
21.     @Override
22.     public void run() {
23.         for (int i = 0; i < 20; i++) {
24.             System.out.print("Thread 2: " + i + "\n");
25.         }
26.     }
27. }
```

多次运行上面的代码,你将看到多种结果,甚至我得到而来这样的结果(完整的线程1,完整的线程2,然后main线程).Okay,现在让我们看个用例,我们想尝试等待线程1和线程2一会儿,让main线程先打印它的语句([S]处),然后切换回 它们,首先让我们使用 `yield` 方法,在线程1,和线程2中我们调用 `yield` 方法,这意味着它跳过这个(线程1和线程2),然后切换到另一个线程,检查新的线程方法.

```
1.  /**
2.  * by Arash M. Dehghani
3.  * arashmd.blogspot.com
```

```
*/
class MyThread implements Runnable {
    @Override
    public void run() {
        Thread.yield();//tells host you would skip me, switch to others. - 告诉主机你可以略过我,切换到其它的.

        for(int i=0;i<20;i++){
            System.out.print("Thread 1: "+i+"\n");
        }
    }
}
```

如果你问我,我将说没有任何差异!这是因为许多 geeks 尽量不要使用它,但是这里我们该怎么做?
一种解决方案是隐式休眠一会儿,但是记住当你休眠1ms,这并不意味着系统将在1m后恢复.

```
package arash.blogger.example.thread;

/**
 * by Arash M. Dehghani arashmd.blogspot.com
 */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        new Thread(new MyThread()).start();
        new Thread(new MyThread2()).start();
        // no one knows when this print will get invoked! after thread 1? or even 2? or even before them?
        System.out.print("Main Thread: I started the threads :)\n"); // main thread sentence [S]
    }
}

class MyThread implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(0, 50);// if system doesn't sleep me, I sleep myself for 50 nano seconds! , wohaha
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 20; i++) {
            System.out.print("Thread 1: " + i + "\n");
        }
    }
}

class MyThread2 implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(0, 50);// but still there is no warranty! - 但是仍然不能保证
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 20; i++) {
            System.out.print("Thread 2: " + i + "\n");
        }
```

```
40.         }
41.     }
```

在上面的代码中,我们只是替换了 yield 方法而休眠50纳秒!但是我仍然不能保证 main 线程的语句[S]先运行,这里 main 线程有更多的机会先运行,但是我再说一遍,没有任何保证强制 thread1 和 thread2 在 main 线程后运行,即使你休眠了2秒而不是50纳秒,除非你同步它们.
注意这同样依赖于你的CPU,如果你的CPU(FSB-RAM)足够强,50纳秒足够切换到另一个,但是另一个CPU可能需要更多时间,系统的繁忙的程度也有影响.

高级主题

对于低性能的apps,是没有什么生存空间的,所以让我们专业地讨论一下,这里我们将讨论高级主题,但是记住这篇文章是对初学者而言的.

管理你的内存

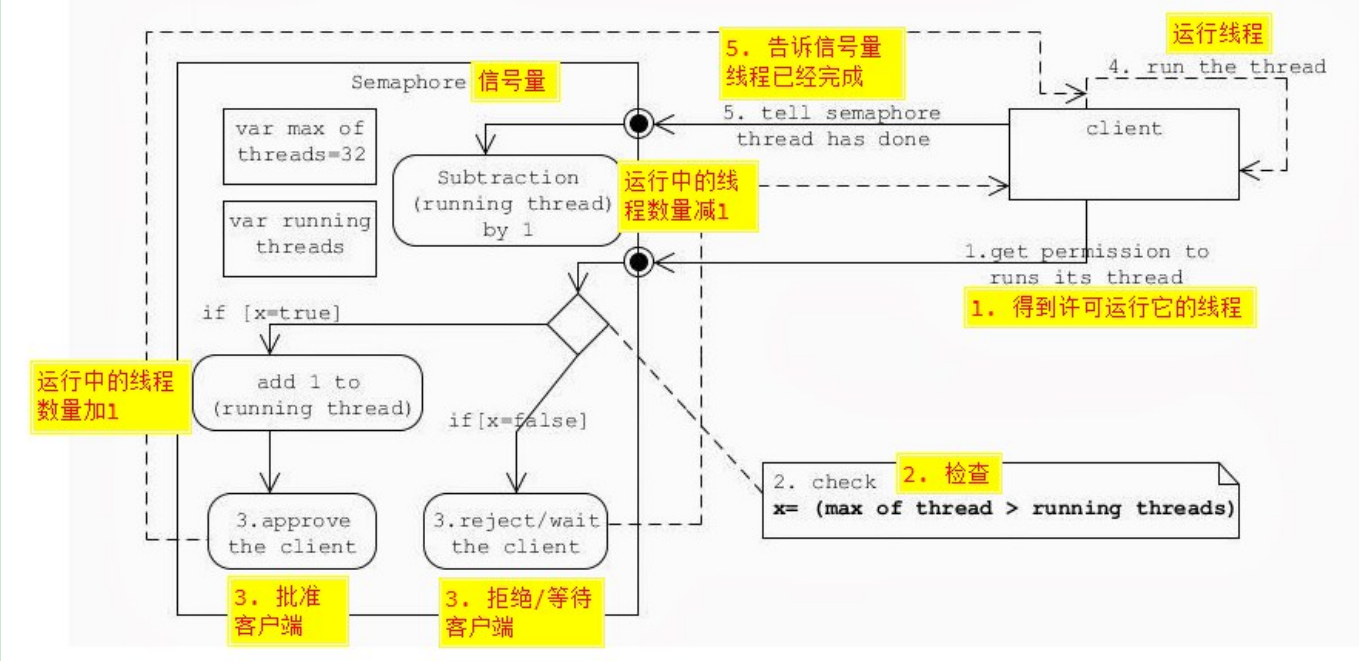
每个线程有一个私有的内存,对自身是可访问的(方法,类,等等...),一块共享的内存属于每个线程,注意这些资源(内存)被分成两种类型,其中一些由你(开发者)产生和管理,还有一些(比如线程id,栈,属性,等等...)属于主机(JVM或OS).
注意创建太多的线程,它将很容易吃掉你的整个内存/资源.对于CPU而言, 执行一个有序的(串行的,单线程)应用程序比一个多线程的应用程序更容易,我们不应该担心CPU的工作简单,(we need our process ASAP with minimum process would take).现在只要运行下面的代码去看下有多少线程干扰你的系统,观察你的系统的内存和性能.

```
1.  package arash.blogger.example.thread;
2.
3.  /**
4.   * by Arash M. Dehghani arashmd.blogspot.com
5.   */
6.  public class Core {
7.      public final static Object o = new Object();
8.
9.      public static void main(String[] args) throws InterruptedException {
10.         for (int i = 0; i < 5000; i++) {// what if running 5000 threads, watch your system!
     - 运行5000个线程将会怎样,观察你的系统
11.             new Thread(new MyThread()).start();
12.         }
13.     }
14. }
15.
16. class MyThread implements Runnable {
17.     // there are no any user variable - 没有任何用户变量
18.     @Override
19.     public void run() {
20.         // some work
21.         try {
22.             synchronized (Core.o) {
23.                 Core.o.wait();
24.             }
25.         } catch (Exception e) {
26.
27.         }
28.     }
29. }
```

有5000个线程在你的内存中将会怎样!?,一些正在等待垃圾收集,一些等待,一些休眠,一些空闲,它是一场灾难,但是可以使用模式解决,继续阅读.
注意有5000个线程在一个应用程序中不是一场灾难,因为你可以运行超过100000个线程在一个 GPU 上,5000个线程对于CPU(同样管理内存)有点困难,所以后面你应该将应用程序分离成并发和一系列的模块(执行流程),将代码分离到GPU和CPU之间.

信号量(Semaphore)-限制运行线程的最大数目(一个资源)

信号量是一种可用来你限制运行中的线程的最大数目的模式,这个对于阻止你的应用程序运行太多的线程引起内存和进程(执行-时间)问题非常有用.
这个在你的应用程序比较小(对于大的应用程序不推荐)的时候很有用.Java SE API 有一个好的 semaphore 类的实现,但是我建议你最少自己去做一次,这样可以理解一个信号量实际做了什么!?
对于有一个信号量,你需要知道它做了什么.信号量看起来像一个许可证制度,在时间 t0 的时候你想运行一个线程,所以首先你需要得到信号量的许可.信号量追踪(统计)每个运行的线程,然后决定批准,阻塞或拒绝你.



上面的信号量追踪所有运行中的线程(间接),通过统计它们.注意线程必须通知信号量当它完成的时候.这种类型的信号量也称为计数信号量,现在让我们看一个简单的实现.

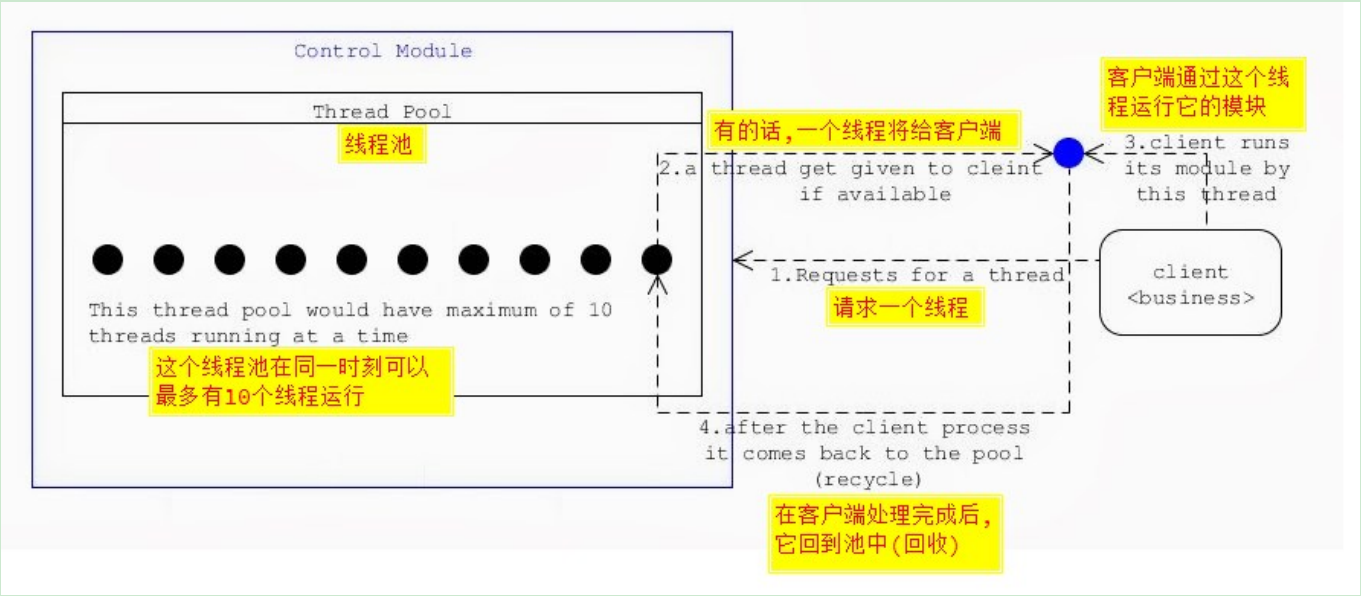
```
1.  package arash.blogger.example.thread;
2.
3.  /**
4.   * by Arash M. Dehghani arashmd.blogspot.com
5.   */
6.  public class Core {
7.      public static void main(String[] args) throws InterruptedException {
8.          System.out.print("trying to run 300 threads, but max 16 thread at a time\n");
9.          MySemaphore sem = new MySemaphore(16);// it could be 8, 32, or even 32, dependent t
    o your application business
10.                                             // 可以是8, 32,依赖于你的应用程序业务
11.
12.         for (int i = 0; i < 300; i++) {
13.             sem.acquire();                              // get a permission to run a thre
    ad [1] - 得到许可运行一个线程
14.             new Thread(new MyThread(sem, i)).start();     // run a thread [3]
15.         }
```

```
            Thread.sleep(100);
            System.out.print("Completed :v\n");
        }
    }

    class MyThread implements Runnable {
        MySemaphore sem;
        int val;

        public MyThread(MySemaphore sem, int val) {
            this.sem = sem;// pass the semaphore to client run, to release itself after it gets terminated
                           // 传递信号量给客户端运行,在它终止之后释放自身
            this.val = val;
        }

        @Override
        public void run() {
            try {
                System.out.print("Hello from client run\n client No: " + this.val + "\n");
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                sem.release(); // tells the semaphore this thread has finished [4]
                               // 告诉信号量这个线程完成了
            }
        }
    }

    class MySemaphore {
        int maxThreadSize;
        int runningThreadSize = 0;
        Object lock = new Object(), elock = new Object();

        public MySemaphore(int maxThreadSize) {
            this.maxThreadSize = maxThreadSize;
        }

        public synchronized void acquire() throws InterruptedException {
            if (runningThreadSize == maxThreadSize) {// checks is it possible to run a thread at this time? - 检查此时释放可以运行一个线程
                this.wait();// or should wait until another thread get finished [2] - 应该等待直到另一个线程结束
            }
            runningThreadSize++;
        }

        public synchronized void release() {
            runningThreadSize--;// decrease one thread from runningThreadSize - 从 runningThreadSize 减去一个线程
            this.notify();        // notify any waiting thread (if any) [5] - 通知任何等待中的线程(如果有的)
        }
    }
```

上面的例子运行300个线程,在同一时间运行300个线程不是一个好的主意,所以我们想使用信号量现在它(一个时刻).

main 线程为每个线程获得许可去运行,信号量需要决定等待还是批准请求,如果最大数目的线程正在运行,信号量阻塞线程直到一个线程(运行中的线程)终止,并通知信号量,这样信号量保证一个线程结束,它可以去运行一个新的.简单一点,你可以拒绝(抛出一个异常,错误,false,...),请求中的线程而不是阻塞它.

可回收的线程(线程池)



如之前提到的, 一个线程有一些属性(需要一些内存), 但是有一个重要的事情, 无论是 OS 还是 JVM 不会关心在一个线程中发生了什么事情(执行流程).它们知道的所有的事情就是运行一个线程.

这就像一个邮差,它把一个信封带到一个目的地,并不关心里面是什么.下一个信封包被销毁,然后一个新的包用于新的.所以它是耗时的,销毁和创建一个新的包.所以问题就是,为什么我们不使用老的包用于下一个信封?或者为什么我们不使用已存在的线程用于运行下一个业务?

但是当我们实现了一个在闪存中拷贝两个文件的一个 thread(runnable),然后这个线程只是永远在一个闪存中拷贝两个文件,这里我们需要以这样一种方式实现我们的线程,独立于任何的业务,一个线程应该是一个线程,而不是一个业务!

这个可以通过线程池很容易完成.