**《多线程相关面试题》**

*20181009版*

目录

[1、为什么用线程池? 1](#_Toc526876120)

[2、线程池参数什么意思？ 1](#_Toc526876121)

[3、讲一讲线程池中的threadpoolexecutor，每个参数干什么用的？ 1](#_Toc526876122)

[4、说一下线程池内部使用规则 4](#_Toc526876123)

[5、用过atomicinteger吗？怎么用的？ 5](#_Toc526876124)

[6、用过threadlocal吗？怎么用的？ 5](#_Toc526876125)

[7、程序、进程、线程的区别是什么? 举个现实的例子说明。 9](#_Toc526876126)

[8、【上机】Java中通过哪些方式创建多线程类? 分别使用代码说明。并调用之。 9](#_Toc526876127)

[9、Thread类有没有实现Runnable接口? 10](#_Toc526876128)

[10、当调用一个线程对象的start方法后，线程马上进入运行状态吗? 10](#_Toc526876129)

[11、下面的代码，实际上有几个线程在运行： 10](#_Toc526876130)

[12、线程的几种状态 10](#_Toc526876131)

[13、说说：sleep、yield、join、wait方法的区别。 11](#_Toc526876132)

[14、为什么不推荐使用stop和destroy方法来结束线程的运行? 13](#_Toc526876133)

[15、写个代码说明，终止线程的典型方式。 13](#_Toc526876134)

[16、A线程的优先级是10，B线程的优先级是1，那么当进行调度时一定会调用A吗? 14](#_Toc526876135)

[17、synchronize修饰在方法前是什么意思? 14](#_Toc526876136)

[18、使用Timer和TimerTask实现定时执行，定时在每天下午17：00执行。 14](#_Toc526876137)

[19、wait方法被调用时，所在线程是否会释放所持有的锁资源? sleep方法呢? 16](#_Toc526876138)

[20、wait、notify、notifyAll是在Thread类中定义的方法吗？作用分别是什么？ 16](#_Toc526876139)

[21、notify是唤醒所在对象wait pool中的第一个线程吗？ 17](#_Toc526876140)

## 1、为什么用线程池?

有时候，系统需要处理非常多的执行时间很短的请求，如果每一个请求都开启一个新线程的话，系统就要不断的进行线程的创建和销毁，有时花在创建和销毁线程上的时间会比线程真正执行的时间还长。

而且当线程数量太多时，系统不一定能受得了。

使用线程池主要为了解决一下几个问题：

通过重用线程池中的线程，来减少每个线程创建和销毁的性能开销。

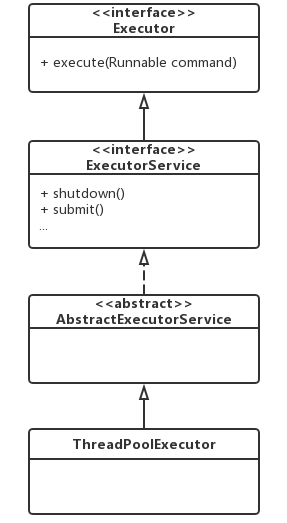
对线程进行一些维护和管理，比如定时开始，周期执行，并发数控制等等。

## 2、线程池参数什么意思？

比如去火车站买票, 有10个售票窗口, 但只有5个窗口对外开放. 那么对外开放的5个窗口称为核心线程数, 而最大线程数是10个窗口.如果5个窗口都被占用, 那么后来的人就必须在后面排队, 但后来售票厅人越来越多, 已经人满为患, 就类似于线程队列已满.这时候火车站站长下令, 把剩下的5个窗口也打开, 也就是目前已经有10个窗口同时运行. 后来又来了一批人,10个窗口也处理不过来了, 而且售票厅人已经满了, 这时候站长就下令封锁入口,不允许其他人再进来, 这就是线程异常处理策略.而线程存活时间指的是, 允许售票员休息的最长时间, 以此限制售票员偷懒的行为.

## 3、讲一讲线程池中的threadpoolexecutor，每个参数干什么用的？

Executor是一个接口，跟线程池有关的基本都要跟他打交道。ThreadPoolExecutor的关系



Executor接口很简单，只有一个execute方法。

ExecutorService是Executor的子接口，增加了一些常用的对线程的控制方法，之后使用线程池主要也是使用这些方法。

AbstractExecutorService是一个抽象类。ThreadPoolExecutor就是实现了这个类。

ThreadPoolExecutor的参数：

ThreadPoolExecutor mExecutor = new ThreadPoolExecutor(corePoolSize,// 核心线程数

maximumPoolSize, // 最大线程数

keepAliveTime, // 闲置线程存活时间

TimeUnit.MILLISECONDS,// 时间单位

new LinkedBlockingDeque<Runnable>(),// 线程队列

Executors.defaultThreadFactory(),// 线程工厂

new AbortPolicy()// 队列已满,而且当前线程数已经超过最大线程数时的异常处理策略

);

corePoolSize

核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。

maximumPoolSize

线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的LinkedBlockingDeque时，这个值无效。

keepAliveTime

非核心线程的闲置超时时间，超过这个时间就会被回收。

unit

指定keepAliveTime的单位，如TimeUnit.SECONDS。当将allowCoreThreadTimeOut设置为true时对corePoolSize生效。

workQueue

线程池中的任务队列.

常用的有三种队列，SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue。

threadFactory

线程工厂，提供创建新线程的功能。ThreadFactory是一个接口，只有一个方法

public interface ThreadFactory {

Thread newThread(Runnable r);

}

通过线程工厂可以对线程的一些属性进行定制。

默认的工厂：

static class DefaultThreadFactory implements ThreadFactory {

private static final AtomicInteger poolNumber = new AtomicInteger(1);

private final ThreadGroup group;

private final AtomicInteger threadNumber = new AtomicInteger(1);

private final String namePrefix;

DefaultThreadFactory() {

SecurityManager var1 = System.getSecurityManager();

this.group = var1 != null?var1.getThreadGroup():Thread.currentThread().getThreadGroup();

this.namePrefix = "pool-" + poolNumber.getAndIncrement() + "-thread-";

}

public Thread newThread(Runnable var1) {

Thread var2 = new Thread(this.group, var1, this.namePrefix + this.threadNumber.getAndIncrement(), 0L);

if(var2.isDaemon()) {

var2.setDaemon(false);

}

if(var2.getPriority() != 5) {

var2.setPriority(5);

}

return var2;

}

}

RejectedExecutionHandler

RejectedExecutionHandler也是一个接口，只有一个方法

public interface RejectedExecutionHandler {

void rejectedExecution(Runnable var1, ThreadPoolExecutor var2);

}

当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution方法。

## 4、说一下线程池内部使用规则

线程池的线程执行规则跟任务队列有很大的关系。

下面都假设任务队列没有大小限制：

如果线程数量<=核心线程数量，那么直接启动一个核心线程来执行任务，不会放入队列中。

如果线程数量>核心线程数，但<=最大线程数，并且任务队列是LinkedBlockingDeque的时候，超过核心线程数量的任务会放在任务队列中排队。

如果线程数量>核心线程数，但<=最大线程数，并且任务队列是SynchronousQueue的时候，线程池会创建新线程执行任务，这些任务也不会被放在任务队列中。这些线程属于非核心线程，在任务完成后，闲置时间达到了超时时间就会被清除。

如果线程数量>核心线程数，并且>最大线程数，当任务队列是LinkedBlockingDeque，会将超过核心线程的任务放在任务队列中排队。也就是当任务队列是LinkedBlockingDeque并且没有大小限制时，线程池的最大线程数设置是无效的，他的线程数最多不会超过核心线程数。

如果线程数量>核心线程数，并且>最大线程数，当任务队列是SynchronousQueue的时候，会因为线程池拒绝添加任务而抛出异常。

任务队列大小有限时

当LinkedBlockingDeque塞满时，新增的任务会直接创建新线程来执行，当创建的线程数量超过最大线程数量时会抛异常。

SynchronousQueue没有数量限制。因为他根本不保持这些任务，而是直接交给线程池去执行。当任务数量超过最大线程数时会直接抛异常。

看代码 ThreadPoolExecutorTest1-7

## 5、用过AtomicInteger吗？怎么用的？

AtomicInteger是int类型的原子操作类，对于全局变量的数值类型操作 num++，若没有加synchronized关键字则是线程不安全的，num++解析为num=num+1，明显，这个操作不具备原子性，多线程时必然会出现问题

看代码

要是换成volatile修饰count变量呢？

volatile修饰的变量能够在线程间保持可见性，能被多个线程同时读但是又能保证只被单线程写，并且不会读取到过期值（由java内存模型中的happen-before原则决定的）volatile修饰字段的写入操作总是优先于读操作，即使多个线程同时修改volatile变量字段，总能保证获取到最新的值

看代码

atomicinteger常用方法：

public final int getAndSet(int newValue) //给AtomicInteger设置newValue并返回加oldValue

public final boolean compareAndSet(int expect, int update) //如果输入的值和期望值相等就set并返回true/false

public final int getAndIncrement() //对AtomicInteger原子的加1并返回当前自增前的value

public final int getAndDecrement() //对AtomicInteger原子的减1并返回自减之前的的value

public final int getAndAdd(int delta) //对AtomicInteger原子的加上delta值并返加之前的value

public final int incrementAndGet() //对AtomicInteger原子的加1并返回加1后的值

public final int decrementAndGet() //对AtomicInteger原子的减1并返回减1后的值

public final int addAndGet(int delta) //给AtomicInteger原子的加上指定的delta值并返回加后的值

## 6、用过threadlocal吗？怎么用的？

早在JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal很容易让人望文生义，想当然地认为是一个“本地线程”。

其实，ThreadLocal并不是一个Thread，而是Thread的局部变量，也许把它命名为ThreadLocalVariable更容易让人理解一些。

ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。

class ConnectionManager {

private static Connection connect = null;

public static Connection openConnection() {

if(connect == null){

connect = DriverManager.getConnection();

}

return connect;

}

public static void closeConnection() {

if(connect!=null)

connect.close();

}

}

　　假设有这样一个数据库链接管理类，这段代码在单线程中使用是没有任何问题的，但是如果在多线程中使用呢？很显然，在多线程中使用会存在线程安全问题：第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

　　所以出于线程安全的考虑，必须将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理。

　　这样将会大大影响程序执行效率，因为一个线程在使用connect进行数据库操作的时候，其他线程只有等待。

　　那么大家来仔细分析一下这个问题，这地方到底需不需要将connect变量进行共享？事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。

　　到这里，可能会有朋友想到，既然不需要在线程之间共享这个变量，可以直接这样处理，在每个需要使用数据库连接的方法中具体使用时才创建数据库链接，然后在方法调用完毕再释放这个连接。比如下面这样：

class ConnectionManager {

private Connection connect = null;

public Connection openConnection() {

if(connect == null){

connect = DriverManager.getConnection();

}

return connect;

}

public void closeConnection() {

if(connect!=null)

connect.close();

}

}

class Dao{

public void insert() {

ConnectionManager connectionManager = new ConnectionManager();

Connection connection = connectionManager.openConnection();

//使用connection进行操作

connectionManager.closeConnection();

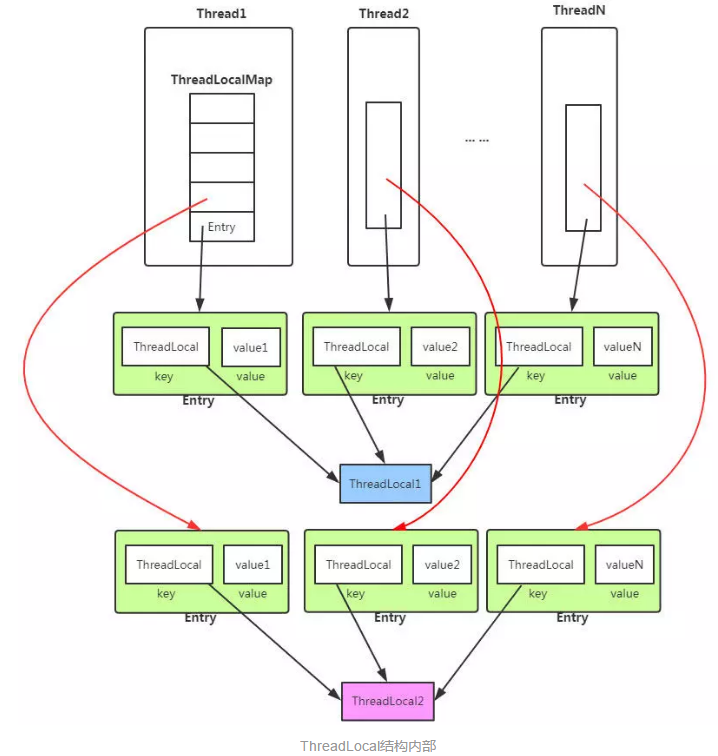
}

}

　　这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。但是这样会有一个致命的影响：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不尽严重影响程序执行效率，还可能导致服务器压力巨大。

　　那么这种情况下使用ThreadLocal是再适合不过的了，因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。



从上面的结构图，我们已经窥见ThreadLocal的核心机制：

每个Thread线程内部都有一个Map。

Map里面存储线程本地对象（key）和线程的变量副本（value）

但是，Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。

所以对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，互不干扰。

**ThreadLocal类提供如下几个核心方法：**

public T get()

public void set(T value)

public void remove()

* get()方法用于获取当前线程的副本变量值。
* set()方法用于保存当前线程的副本变量值。
* initialValue()为当前线程初始副本变量值。
* remove()方法移除当前前程的副本变量值。

用法 ： 见代码 ThreadLocalTest1

## 7、程序、进程、线程的区别是什么? 举个现实的例子说明。

程序（Program）：是一个指令的集合。程序不能独立执行，只有被加载到内存中，系统为它分配资源后才能执行。

进程（Process）：如上所述，一个执行中的程序称为进程。

进程是系统分配资源的独立单位，每个进程占有特定的地址空间。

程序是进程的静态文本描述，进程是程序在系统内顺序执行的动态活动。

线程（Thread）：是进程的“单一的连续控制流程“。

线程是CPU调度和分配的基本单位，是比进程更小的能独立运行的基本单位，也被称为轻量级的进程。

线程不能独立存在，必须依附于某个进程。一个进程可以包括多个并行的线程，一个线程肯定属于一个进程。Java虚拟机允许应用程序并发地执行多个线程。

举例：如一个车间是一个程序，一个正在进行生产任务的车间是一个进程，车间内每个从事不同工作的工人是一个线程。

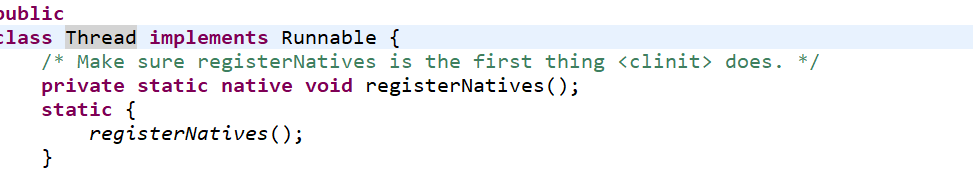
## 8、【上机】Java中通过哪些方式创建多线程类? 分别使用代码说明。并调用之。

* **继承Thread类创建线程**
* **实现Runnable接口创建线程**
* **实现Callable接口通过FutureTask包装器来创建Thread线程**
* **使用ExecutorService、Callable、Future实现有返回结果的线程**

****见代码ThreadCreateTest1-4****

## 9、Thread类有没有实现Runnable接口?

有实现。



## 10、当调用一个线程对象的start方法后，线程马上进入运行状态吗?

不是，只是进入就绪（可运行）状态，等待分配CPU时间片。一旦得到CPU时间片，即进入运行状态。

## 11、下面的代码，实际上有几个线程在运行：

public static void main(String[] argc) throws Exception {

Runnable r = new Thread6();

Thread t = new Thread(r, "Name test");

t.start();

}

两个：线程t和main()方法（主线程）。

## 12、线程的几种状态

1.线程通常有五种状态，创建，就绪，运行、阻塞和死亡状态。

2.阻塞的情况又分为三种：

(1)、等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，wait是object类的方法

(2)、同步阻塞：运行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”中。

(3)、其他阻塞：运行的线程执行sleep()或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，线程重新转入就绪状态。sleep是Thread类的方法

1.新建状态（New）：新创建了一个线程对象。

2.就绪状态（Runnable）：线程对象创建后，其他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，变得可运行，等待获取CPU的使用权。

3. 运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4.阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

5.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。

## 13、说说：sleep、yield、join、wait方法的区别。

sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程仍然不能访问共享数据。 作用于线程

 Thread.sleep()方法用来暂停线程的执行，将CPU放给线程调度器。

 Thread.sleep()方法是一个静态方法，它暂停的是当前执行的线程。

 Java有两种sleep方法，一个只有一个毫秒参数，另一个有毫秒和纳秒两个参数。

 与wait方法不同，sleep方法不会释放锁

 如果其他的线程中断了一个休眠的线程，sleep方法会抛出Interrupted Exception。

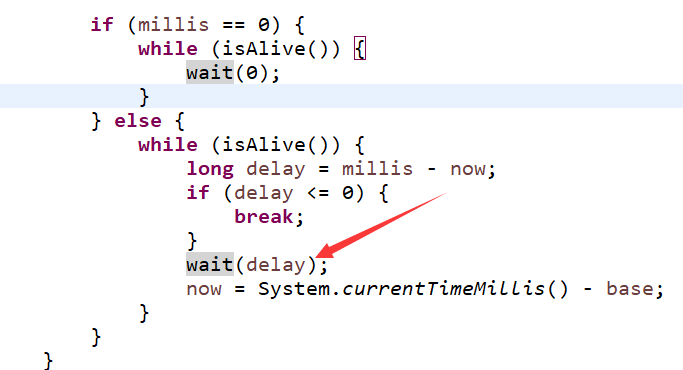
 休眠的线程在唤醒之后不保证能获取到CPU，它会先进入就绪态，与其他线程竞争CPU。

 有一个易错的地方，当调用t.sleep()的时候，会暂停线程t。这是不对的，因为Thread.sleep是一个静态方法，它会使当前线程而不是线程t进入休眠状态。

join(): 当前线程等待，调用此方法的线程执行结束再继续执行。如：在main方法中调用t.join(),那main方法在此时进入阻塞状态，一直等t线程执行完，main方法再恢复到就绪状态，准备继续执行。

join方法必须在线程start方法调用之后调用才有意义。这个也很容易理解：如果一个线程都没有start，那它也就无法同步了。作用于线程

实现原理

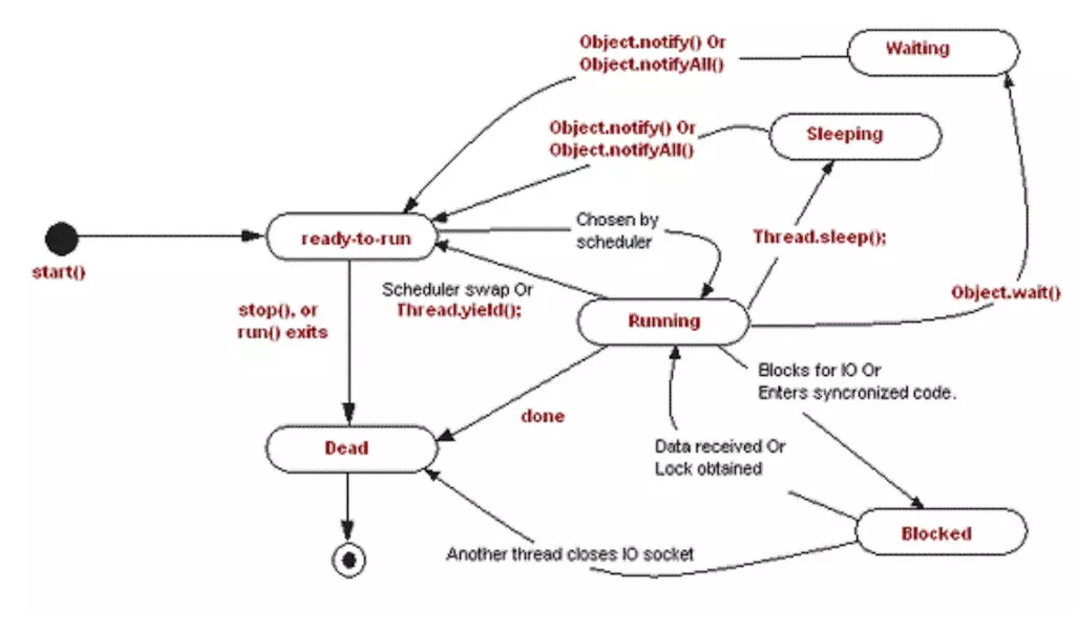


yield(): 它仅仅释放线程所占有的CPU资源，从而让其他线程有机会运行，但是并不能保证某个特定的线程能够获得CPU资源。谁能获得CPU完全取决于调度器，在有些情况下调用yield方法的线程甚至会再次得到CPU资源。所以，依赖于yield方法是不可靠的，它只能尽力而为。作用于线程

wait():

* + wait只能在同步（synchronize）环境中被调用，而sleep不需要。
  + 进入wait状态的线程能够被notify和notifyAll线程唤醒，但是进入sleeping状态的线程不能被notify方法唤醒。
  + wait通常有条件地执行，线程会一直处于wait状态，直到某个条件变为真。但是sleep仅仅让你的线程进入睡眠状态。
  + wait方法在进入wait状态的时候会释放对象的锁，但是sleep方法不会。

wait方法是针对一个被同步代码块加锁的对象



见代码ThreadBlockTest1-5

## 14、为什么不推荐使用stop和destroy方法来结束线程的运行?

stop():此方法可以强行中止一个正在运行或挂起的线程。但stop方法不安全，就像强行切断计算机电源，而不是按正常程序关机。可能会产生不可预料的结果。

举例来说：

当在一个线程对象上调用stop()方法时，这个线程对象所运行的线程就会立即停止，并抛出特殊的ThreadDeath()异常。这里的“立即”因为太“立即”了，

假如一个线程正在执行：

synchronized void {

x = 3;

y = 4;

}

由于方法是同步的，多个线程访问时总能保证x,y被同时赋值，而如果一个线程正在执行到x = 3;时，被调用了 stop()方法，即使在同步块中，它也干脆地stop了，这样就产生了不完整的残废数据。而多线程编程中最最基础的条件要保证数据的完整性，所以请忘记 线程的stop方法，以后我们再也不要说“停止线程”了。

destroy():该方法最初用于破坏该线程，但不作任何资源释放。它所保持的任何监视器都会保持锁定状态。不过，该方法决不会被实现。即使要实现，它也极有可能以 suspend() 方式被死锁。如果目标线程被破坏时保持一个保护关键系统资源的锁，则任何线程在任何时候都无法再次访问该资源。如果另一个线程曾试图锁定该资源，则会出现死锁。

## 15、写个代码说明，终止线程的典型方式。

(1) 当run()方法执行完后，线程就自动终止了。

(2) 但有些时候run()方法不会结束（如服务器端监听程序），或者其它需要用循环来处理的任务。在这种情况下，一般是将这些任务放在一个循环中，如while循环。如果想让循环永远运行下去，可以使用while（true）{……}来处理。但要想使while循环在某一特定条件下退出，最直接的方法就是设一个boolean类型的标志，并通过设置这个标志为true或false来控制while循环是否退出。

见代码ThreadEndTest

代码中定义了一个退出标志exit，当exit为true时，while循环退出，exit的默认值为false.在定义exit时，使用了一个Java关键字volatile，这个关键字的目的是使exit同步，也就是说在同一时刻只能由一个线程来修改exit的值。

## 16、A线程的优先级是10，B线程的优先级是1，那么当进行调度时一定会调用A吗?

不一定。线程优先级对于不同的线程调度器可能有不同的含义，可能并不是用户直观的推测。

见代码ThreadPriorityTest

## 17、synchronize修饰在方法前是什么意思?

一次只能有一个线程进入该方法,其他线程要想在此时调用该方法,只能排队等候,当前线程(就是在synchronized方法内部的线程)执行完该方法后,别的线程才能进入.

见代码SynchronizedTest1

## 18、使用Timer和TimerTask实现定时执行，定时在每天下午17：00执行。

见代码TimerTest1-3

知识点简介：

（1） Timer：定时器，实际上是个线程，定时调度所拥有的TimerTasks。

（2） TimerTask：一个拥有run方法的类，需要定时执行的代码放到run方法体内。 TimerTask一般是以匿名类的方式创建。

语法简介：

java.util.Timer timer = new java.util.Timer(true);

// true 说明这个timer以daemon方式运行（优先级低，程序结束timer也自动结束），注意，javax.swing 包中也有一个Timer类，如果import中用到swing包， 要注意名字的冲突。

TimerTask task = new TimerTask() {

public void run() {

... //每次需要执行的代码放到这里面。

}

};

用法简介：

//以下是几种调度task的方法：

timer.schedule(task, time);

// time为Date类型：在指定时间执行一次。

timer.schedule(task, firstTime, period);

// firstTime为Date类型,period为long

// 从firstTime时刻开始，每隔period毫秒执行一次。

timer.schedule(task, delay)

// delay 为long类型：从现在起过delay毫秒执行一次

timer.schedule(task, delay, period)

// delay为long,period为long：从现在起过delay毫秒以后，每隔period

// 毫秒执行一次。

举例：

import java.util.TimerTask;

public class TimePrintTask extends TimerTask {

int i=1;

publicvoid run(){

System.out.println(i);

i++;

}

}

public class Test {

public static void main(String[] args) {

Timer timer=new Timer();

//timer.schedule(new TimePrintTask(), 1000, 500);

DateFormat df=new SimpleDateFormat("yyyy/MM/dd HH:mm:ss SS");

Date firstDate=null;

try {

firstDate=df.parse("2016/12/09 17:00:00 00");

} catch (ParseException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

timer.schedule(new TimePrintTask(),firstDate , 1000)；

}

}

## 19、wait方法被调用时，所在线程是否会释放所持有的锁资源? sleep方法呢?

wait:释放CPU，释放锁；

sleep：释放CPU，不释放锁。

## 20、wait、notify、notifyAll是在Thread类中定义的方法吗？作用分别是什么？

wait(),notify(),notifyAll()不属于Thread类,而是属于Object类,也就是说每个对象都有wait(),notify(),notifyAll()的功能。

因为每个对像都有锁,锁是每个对像的基础,而wait(),notify(),notifyAll()都是跟锁有关的方法。

三个方法的作用分别是：

wait：导致当前线程等待，进入阻塞状态，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。当前线程必须拥有此对象监视器（对象锁）。该线程释放对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在此对象的监视器上等待的线程醒来。然后该线程将等到重新获得对监视器的所有权后才能继续执行.

notify：唤醒在此对象监视器（对象锁）上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程。此方法只应由作为此对象监视器的所有者的线程来调用.

"当前线程必须拥有此对象监视器"与"此方法只应由作为此对象监视器的所有者的线程来调用"说明wait方法与notify方法必须在同步块内执行,即synchronized(obj之内).

notifyAll: 唤醒在此对象监视器（对象锁）上等待的所有线程。

## 21、notify是唤醒所在对象wait pool中的第一个线程吗？

不是。

调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择。