多线程学习笔记

### 线程实现4种方式

#### 继承Thread类创建线程

Thread类本质上是实现了Runnable接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。代码简单就不写示例了。

最佳实践：创建线程最好给线程指定一个线程名称，这样方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

#### 实现Runnable接口创建线程

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，可以实现一个Runnable接口，并且实现run方法。

#### 实现Callable接口通过FutureTask包装器来创建Thread线程

Callable接口（也只有一个方法）定义如下：

|  |
| --- |
| public interface Callable<V>{  V call() throws Exception;  } |

创建线程代码如下：

|  |
| --- |
| public class SomeCallable<V> extends OtherClass implements Callable<V> {  @Override  public V call() throws Exception {  // TODO Auto-generated method stub  return null;  }  } |

|  |
| --- |
| Callable<V> oneCallable = new SomeCallable<V>();  //由Callable<Integer>创建一个FutureTask<Integer>对象：  FutureTask<V> oneTask = new FutureTask<V>(oneCallable);  //注释：FutureTask<Integer>是一个包装器，它通过接受Callable<Integer>来创建，它同时实现了Future和Runnable接口。  //由FutureTask<Integer>创建一个Thread对象：  Thread oneThread = new Thread(oneTask);  oneThread.start(); |

#### 使用ExecutorService、Callable、Future实现有返回结果的线程

ExecutorService、Callable、Future三个接口实际上都是属于Executor框架。返回结果的线程是在JDK1.5中引入的新特征，有了这种特征就不需要再为了得到返回值而大费周折了。而且自己实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口。类似的，无返回值的任务必须实现Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

注意：get方法是阻塞的，即：线程无返回结果，get方法会一直等待。

再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

下面提供了一个完整的有返回结果的多线程测试例：

|  |
| --- |
| /\*\*  \* 有返回值的线程  \*/  @SuppressWarnings("unchecked")  public class Test {  public static void main(String[] args) throws ExecutionException,  InterruptedException {  System.out.println("----程序开始运行----");  Date date1 = new Date();  int taskSize = 5;  // 创建一个线程池  ExecutorService pool = Executors.newFixedThreadPool(taskSize);  // 创建多个有返回值的任务  List<Future> list = new ArrayList<Future>();  for (int i = 0; i < taskSize; i++) {  Callable c = new MyCallable(i + " ");  // 执行任务并获取Future对象  Future f = pool.submit(c);  // System.out.println(">>>" + f.get().toString());  list.add(f);  }  // 关闭线程池  pool.shutdown();  // 获取所有并发任务的运行结果  for (Future f : list) {  // 从Future对象上获取任务的返回值，并输出到控制台  System.out.println(">>>" + f.get().toString());  }  Date date2 = new Date();  System.out.println("----程序结束运行----，程序运行时间【"  + (date2.getTime() - date1.getTime()) + "毫秒】");  }  }    public class MyCallable implements Callable<Object> {  private String taskNum;    MyCallable(String taskNum) {  this.taskNum = taskNum;  }    public Object call() throws Exception {  System.out.println(">>>" + taskNum + "任务启动");  Date dateTmp1 = new Date();  Thread.sleep(1000);  Date dateTmp2 = new Date();  long time = dateTmp2.getTime() - dateTmp1.getTime();  System.out.println(">>>" + taskNum + "任务终止");  return taskNum + "任务返回运行结果,当前时间【" + time + "毫秒】";  }  } |

#### 总结

从上面4个实例可以看出实现线程的方式也可以说是3种方式，分别是继承Threa类，实现Runnable接口、实现Callable（一种使用FutureTask，一种使用J.U.C的线程池）。

### 终止线程三种方式

有三种方法可以使终止线程：

* 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
* 使用stop方法强行终止线程（这个方法不推荐使用，因为stop会导致线程立马停止，这样可能会导致一些锁或数据、资源等释放操作无法得到正确执行，这样就可能出现不可预料的结果）。
* 使用interrupt方法中断线程。

#### 使用退出标志终止线程

当run方法执行完后，线程就会退出。但有时run方法是永远不会结束的。如在服务端程序中使用线程进行监听客户端请求，或是其他的需要循环处理的任务。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。下面给出了一个利用退出标志终止线程的例子。

|  |
| --- |
| public class ThreadFlag extends Thread{  public volatile boolean exit = false;  public void run(){  while (!exit){//执行逻辑};  }  public static void main(String[] args) throws Exception{  ThreadFlag thread = new ThreadFlag();  thread.start();  sleep(5000); // 主线程延迟5秒  thread.exit = true; // 终止线程thread  thread.join();  System.out.println("线程退出!");  }  } |

上述代码中，join() 的作用：让“主线程”等待“子线程”结束之后才能继续运行。也就是说执行到【thread.join()】代码时，主线程（main）会等待thread线程执行结束后，主线程才继续执行。

#### 使用stop方法终止线程

使用stop方法可以强行终止正在运行或挂起的线程。我们可以使用如下的代码来终止线程：thread.stop();

虽然使用上面的代码可以终止线程，但使用stop方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，因此，并不推荐使用stop方法来终止线程。

#### 使用interrupt方法终止线程

使用interrupt方法来终端线程可分为两种情况：

* 线程处于阻塞状态，如使用了sleep方法。
* 使用while（！isInterrupted（））{……}来判断线程是否被中断。

在第一种情况下使用interrupt方法，sleep方法将抛出一个InterruptedException例外，而在第二种情况下线程将直接退出。下面的代码演示了在第一种情况下使用interrupt方法。

|  |
| --- |
| public class ThreadInterrupt extends Thread{  public void run(){  try{  sleep(50000); // 延迟50秒  } catch (InterruptedException e){  System.out.println(e.getMessage());  }  }  public static void main(String[] args) throws Exception{  Thread thread = new ThreadInterrupt();  thread.start();  System.out.println("在50秒之内按任意键中断线程!");  System.in.read();  thread.interrupt();  thread.join();  System.out.println("线程已经退出!");  }  } |

上面代码的运行结果如下：

|  |
| --- |
| 在50秒之内按任意键中断线程!  sleep interrupted  线程已经退出! |

在调用interrupt方法后， sleep方法抛出异常，然后输出错误信息：sleep interrupted.

注意：在Thread类中有两个方法可以判断线程是否通过interrupt方法被终止。一个是静态的方法interrupted（），一个是非静态的方法isInterrupted（），这两个方法的区别是interrupted用来判断当前线是否被中断，而isInterrupted可以用来判断其他线程是否被中断。因此，while （！isInterrupted（））也可以换成while （！Thread.interrupted（））。

### 线程状态

Java中的线程的生命周期大体可分为5种状态：

* 新建(NEW)：新创建了一个线程对象。
* 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 。
* 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。
* 阻塞(BLOCKED)：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：
* 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。
* 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。
* 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。
* 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。

线程的状态图如下：



### 内存模型

一个现代计算机通常由两个或者多个CPU。其中一些CPU还有多核。从这一点可以看出，在一个有两个或者多个CPU的现代计算机上同时运行多个线程是可能的。每个CPU在某一时刻运行一个线程是没有问题的。这意味着，如果你的Java程序是多线程的，在你的Java程序中每个CPU上一个线程可能同时（并发）执行。

每个CPU都包含一系列的寄存器，它们是CPU内内存的基础。CPU在寄存器上执行操作的速度远大于在主存上执行的速度。这是因为CPU访问寄存器的速度远大于主存。

每个CPU可能还有一个CPU缓存层。实际上，绝大多数的现代CPU都有一定大小的缓存层。CPU访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。一些CPU还有多层缓存，但这些对理解Java内存模型如何和内存交互不是那么重要。只要知道CPU中可以有一个缓存层就可以了。

一个计算机还包含一个主存。所有的CPU都可以访问主存。主存通常比CPU中的缓存大得多。

通常情况下，当一个CPU需要读取主存时，它会将主存的部分读到CPU缓存中。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当CPU需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在某个时间点将值刷新回主存。

当CPU需要在缓存层存放一些东西的时候，存放在缓存中的内容通常会被刷新回主存。CPU缓存可以在某一时刻将数据局部写到它的内存中，和在某一时刻局部刷新它的内存。它不会再某一时刻读/写整个缓存。通常，在一个被称作“cache lines”的更小的内存块中缓存被更新。一个或者多个缓存行可能被读到缓存，一个或者多个缓存行可能再被刷新回主存。

想象一下，共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到CPU缓存中。然后修改了这个对象。只要CPU缓存没有被刷新会主存，对象修改后的版本对跑在其它CPU上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

使用Java中的volatile关键字。volatile关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。同时会通知其他线程该变量cpu的缓存行无效，其他线程将直接从主存中读取该变量的值到cpu缓存中。

如果想了解指令重排序和happens-before原则的知识点或更多关于内存模型的知识，可以阅读：<https://www.cnblogs.com/dolphin0520/p/3920373.html>

以下为代码示例：

|  |
| --- |
| public class VolatileTest implements Runnable{  private volatile boolean tag = false;  private static CountDownLatch latch = new CountDownLatch(10);  @Override  public void run() {  if(tag){  System.out.println("tag为："+tag);  }else{  System.out.println("tag111111为："+tag);  tag = true;  System.out.println("tag2222为："+tag);  }  latch.countDown();  }  public static void main(String[] args) throws InterruptedException {  ExecutorService executorService = Executors.newFixedThreadPool(10);  VolatileTest test = new VolatileTest();  for(int i = 0; i < 10; i++){  executorService.submit(test);  }  latch.await();  executorService.shutdown();  System.out.println("测试完毕");  }  } |

上面代码示例其中的一次执行结果如下：

|  |
| --- |
| tag111111为：false  tag111111为：false  tag111111为：false  tag2222为：true  tag2222为：true  tag2222为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  测试完毕 |

Volatile修饰的变量能保证变量原子操作下的线程安全，非原子操作无法保证线程安全，如下面代码：

|  |
| --- |
| volatile int count;  public void add(){  count++;  } |

上述代码的add方法在多线程环境下存在线程安全的问题，比如有2个线程（A和B），A将count的值（0）读到cpu缓存中，并进行加1，但还没赋值给count的节点前任意时刻，线程B读到cpu缓存的count值也为0，并且进行++操作，所以最终获取的值为1，而不是2。这里++操作就不是原子操作，所以哪怕使用volatile修饰也存在线程安全问题。

### Synchronized

线程安全是并发编程中的重要关注点，应该注意到的是，造成线程安全问题的主要诱因有两点，一是存在共享数据(也称临界资源)，二是存在多条线程共同操作共享数据。因此为了解决这个问题，我们可能需要这样一个方案，当存在多个线程操作共享数据时，需要保证同一时刻有且只有一个线程在操作共享数据，其他线程必须等到该线程处理完数据后再进行，这种方式有个高尚的名称叫互斥锁，即能达到互斥访问目的的锁，也就是说当一个共享数据被当前正在访问的线程加上互斥锁后，在同一个时刻，其他线程只能处于等待的状态，直到当前线程处理完毕释放该锁。在 Java 中，关键字 synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代Volatile功能），这点确实也是很重要的。

synchronized关键字最主要有以下3种应用方式：

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

#### synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。简单代码如下：

|  |
| --- |
| public class SynchronziedTest implements Runnable{  public synchronized void test1(){  System.out.println("开始执行test1");  test2();  System.out.println("执行test1结束");  }  public synchronized void test2(){  System.out.println("执行test2");  }  @Override  public void run() {  test1();  }  public static void main(String[] args) {  new Thread(new SynchronziedTest()).start();  }  } |

上面示例执行结果如下：

|  |
| --- |
| 开始执行test1  执行test2  执行test1结束 |

线程获取SynchronziedTest对象锁进入test1方法后，如果在test1方法中调用test2方法（该方法也被synchronized）是被允许的，我们就说该锁是可重入锁（或该锁有可重入的特性）。

### wait、notify和notifyAll

wait、notify和notifyAll主要注意的有以下方面：

* wait、notify和notifyAll方法必须在同步代码块中使用
* sleep和wait的区别：sleep不会释放对象锁，wait会释放对象锁
* notify和notifyAll方法调用后，尽量立即退出临界区，就是说尽量不要在notify和notifyAll方法后面写一些耗时的操作
* 多线程中校验某个条件的变化用if还是用while？如：

while(条件不满足){this.wait();}还是if(条件不满足){this.wait();}，答案是使用while。

当一个线程正在某一个对象的同步方法中运行时调用了这个对象的wait()方法，那么这个线程将释放该对象的独占锁并被放入这个对象的等待队列。注意，wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。在调用对象的notify()和notifyAll()方法之前,调用线程必须已经得到该对象的锁。因此,必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

wait、notify和notifyAll方法只是实现线程间通信的一种方式，更多内容可以查看：<http://ifeve.com/thread-signaling/>

### 死锁

死锁是两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程1锁住了A，然后尝试对B进行加锁，同时线程2已经锁住了B，接着尝试对A进行加锁，这时死锁就发生了。线程1永远得不到B，线程2也永远得不到A，并且它们永远也不会知道发生了这样的事情。为了得到彼此的对象（A和B），它们将永远阻塞下去。这种情况就是一个死锁。

一般造成死锁必须同时满足如下4个条件：

* 互斥条件：线程使用的资源必须至少有一个是不能共享的；
* 请求与保持条件：至少有一个线程必须持有一个资源并且正在等待获取一个当前被其它线程持有的资源；
* 非剥夺条件：分配资源不能从相应的线程中被强制剥夺；
* 循环等待条件：第一个线程等待其它线程，后者又在等待第一个线程。

因为要产生死锁，这4个条件必须同时满足，所以要防止死锁的话，只需要破坏其中一个条件即可。一些推荐做法如下：

* 尽量使用tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用java.util.concurrent(jdk 1.5以上)包的并发类代替手写控制并发，比较常用的是ConcurrentHashMap、ConcurrentLinkedQueue、AtomicBoolean等等，实际应用中java.util.concurrent.atomic十分有用，简单方便且效率比使用Lock更高
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁
* 尽量减少同步的代码块

比较简单的方法：所有线程都按顺序获取资源，比如资源ABCD ，那所有线程必须获取了A资源再去 申请B资源 ， 依次类推。

其实我们在操作资源的时候，可以使用到队列。让线程按照顺序操作资源，这样的话，就不会造成死锁现象。

按顺序获取资源，是破坏循环等待的条件，算是比较好的方法但是也是有缺点的：

* 是限制的进程对资源的要求，而已给系统中所有的资源合理编号也是件难事，同时增加了系统开销。
* 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。

对于这个问题，单一的临界资源可以使用资源分配图算法，多类的临界资源可以使用“银行家算法”。