Java线程

PART ONE：线程的创建

线程的创建可以分为两种方式:A）继承Tread类；B）实现Runnable接口

两种创建线程的方式区别和联系主要有哪些?

1）、Java单继承机制，限制了Thread类的使用；然后可以通过实现Runnable接口实现多线程，同时也可继承其他类来实现其他功能；

2）、Runable可以使用多线程去处理同一资源，同时也增加了程序的健壮性，相同的代码可以被多个线程共享，这是Thread类所不具备的;

3）、如买票的经典例子:

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  ThreadTest thread1 = **new** Test.ThreadTest();  ThreadTest thread2 = **new** Test.ThreadTest();  thread1.start();;  thread2.start();  RunableTest runable = **new** Test.RunableTest();  **new** Thread(runable).start();;  **new** Thread(runable).start();;  }  **static** **class** ThreadTest **extends** Thread{  **private** **int** ticket = 10;  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 20; i++) {  **if**(**this**.ticket > 0){  System.***out***.println("[" + Thread.*currentThread*().getName() + "] -- " + **this**.ticket--);  }  }  }  }  **static** **class** RunableTest **implements** Runnable {  **private** **int** ticket = 10;  **public** **void** run() {  **for** (**int** i = 0; i < 20; i++) {  **if**(**this**.ticket > 0){  System.***out***.println("[" + Thread.*currentThread*().getName() + "] -- " + **this**.ticket--);  }  }  }  }  } |

PS:

1、JDK中Thread类也是Runnable接口的子类

2、Thread的run()和start()的关系(JDK源码可知)：

start()方法使用native关键字修饰，该关键字表示调用操作系统的底层函数(JNI);

Thread的start()一旦被调用，JVM则会去调用run()方法;

PART TWO：线程的同步

1、线程同步的两个特性：可见性和有序性

简单来说，多线程的交互方式往往是通过共享变量的方式来实现。如多线程共享内存中的一个对象，如果某一个线程修改了该对象的某个属性值，对于其他线程而言是可以看到被修改的属性值，即:可见性；再如多线程共享内存中的一个对象，如果某一个线程想修改该对象的某个属性值，而同时另外一个线程也想修改这个属性值，为了避免修改冲突，一定要保证修改的先后顺序，即：有序性。

2、Java线程同步的实现：volatile、synchronized、Lock

2-1、volatile

1）、确切的说该关键字并不是真正意义上的加锁同步（个人理解）；它本质是告诉jvm当前变量在寄存器中的值是不确定的,需要从主存中读取；

2）、只作用于变量级别；

2-2、synchronized

1）、并发线程访问同一个对象实例中synchronized(this) 或 synchronized method()时，一个时间内只能有一个线程得到执行，其他线程必须等待当前线程执行完才能继续执行；

2）、当一个线程访问一个对象实例中synchronized(this) 或 synchronized method()时，其他线程仍然可以访问对象实例中的非synchronized(this) 或 非synchronized method()；

3）、当一个线程访问一个对象实例中synchronized(this) 或 synchronized method()时，其他线程对该对象实例中的其他synchronized(this) 或 synchronized method()的访问仍将被阻塞；

4）、不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

5）、synchronized static是某个类的范围，主要防止多个线程同时访问这个类中的synchronized static method()，它可以对类的所有对象实例起作用；synchronized 是某实例的范围，主要防止多个线程同时访问这个实例中的synchronized 方法；

2-3、Lock（JDK1.5以上的锁框架）

1）、ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（即：当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

2）、Lock 和 synchronized 有一点明显的区别：Lock必须在finally块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放，This is very terrible!

2-x、锁的概念：

每个锁都有两个队列，一个是就绪队列，一个是阻塞队列，就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程，当一个线程被唤醒(notify)后，才会进入到就绪队列，等待CPU的调度，反之，当一个线程被wait后，就会进入阻塞队列，等待下一次被唤醒，一个线程执行互斥代码过程如下：

1）. 获得同步锁；

2）. 清空工作内存；

3）. 从主内存拷贝对象副本到工作内存；

4）. 执行代码(计算或者输出等)；

5）. 刷新主内存数据；

6）. 释放同步锁。

3、Java线程同步的实现

3-1、volatile(略)

3-2、synchronized

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  **final** Samp1 samp1\_0 = **new** Samp1();  Samp1 samp1\_1 = **new** Samp1();  /\*  \* 1. new Thread(samp1\_0, "thread-A-0").start();  \* 2. new Thread(samp1\_0, "thread-B-0").start();  \* 3. new Thread(samp1\_1, "thread-A-1").start();  \* 4. new Thread(new Runnable() { public void run() { samp1\_0.base\_ticket();} }, "thread-C-0").start();  \* 5. new Thread(new Runnable() { public void run() { samp1\_0.ticket();} }, "thread-D-0").start();  \* 6. new Thread(new Runnable() { public void run() { samp1\_0.syn\_ticket();} }, "thread-E-0").start();  \* 7. new Thread(new Runnable() { public void run() { Samp1.syn\_stat\_ticket();} }, "thread-F-0").start();  \* 8. new Thread(new Runnable() { public void run() { Samp1.syn\_stat\_ticket2();} }, "thread-G-0").start();  \*  \* 1 and 2 == > synchronized 性质1  \* 1 and 3 == > synchronized 性质4  \* 4 and 5 == > synchronized 性质2  \* 4 and 6 == > synchronized 性质3  \* 7 and 8 == > synchronized 性质5  \*/  **new** Thread(samp1\_0, "thread-A-0").start();  **new** Thread(samp1\_0, "thread-B-0").start();  **new** Thread(samp1\_1, "thread-A-1").start();    **new** Thread(**new** Runnable() { **public** **void** run() { samp1\_0.base\_ticket();} }, "thread-C-0").start();  **new** Thread(**new** Runnable() { **public** **void** run() { samp1\_0.ticket();} }, "thread-D-0").start();  **new** Thread(**new** Runnable() { **public** **void** run() { samp1\_0.syn\_ticket();} }, "thread-E-0").start();  **new** Thread(**new** Runnable() { **public** **void** run() { Samp1.*syn\_stat\_ticket*();} }, "thread-F-0").start();  **new** Thread(**new** Runnable() { **public** **void** run() { Samp1.*syn\_stat\_ticket2*();} }, "thread-G-0").start();  }  }  **public** **class** Samp1 **implements** Runnable{  **private** **int** ticketNum = 10;  **private** **static** **int** *ticketNum\_1* = 10;  **public** **void** base\_ticket() {  **synchronized** (**this**) {  **for** (**int** i = 0; i < 20; i++) {  **if**(ticketNum > 0){  System.*out*.println("[" + Thread.*currentThread*().getName()+"] -- " + ticketNum--);  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  **public** **void** ticket(){  **for** (**int** i = 0; i < 20; i++) {  **if**(ticketNum > 0){  System.*out*.println("[" + Thread.*currentThread*().getName()+"] -- " + ticketNum--);  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **public** **synchronized** **void** syn\_ticket(){  **for** (**int** i = 0; i < 20; i++) {  **if**(ticketNum > 0){  System.*out*.println("[" + Thread.*currentThread*().getName()+"] -- " + ticketNum--);  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **public** **void** run() {  }  **public** **synchronized** **static** **void** syn\_stat\_ticket(){  **for** (**int** i = 0; i < 20; i++) {  **if**(*ticketNum\_1* > 0){  System.*out*.println("[" + Thread.*currentThread*().getName()+"] -- " + *ticketNum\_1*--);  }  }  }  **public** **synchronized** **static** **void** syn\_stat\_ticket2(){  **for** (**int** i = 0; i < 20; i++) {  **if**(*ticketNum\_1* > 0){  System.*out*.println("[" + Thread.*currentThread*().getName()+"] -- " + *ticketNum\_1*--);  }  }  }  } |

3-3、Lock

|  |
| --- |
| **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** ReentrantLockTest {  **private** Lock lock = **new** ReentrantLock();  **private** **int** count = 0;  **public** **void** sampLock() {  **try** {  lock.lock();  **for** (**int** i = 0; i < 100000; i++) {  count += 1;  }  System.*out*.println(count);  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String[] args) {  **final** ReentrantLockTest locktest = **new** ReentrantLockTest();  **new** Thread(**new** Runnable() {**public** **void** run() { locktest.sampLock();}}).start();  **new** Thread(**new** Runnable() {**public** **void** run() { locktest.sampLock();}}).start();  }  } |

PART THREE：线程池

JDK具体的接口以及类的说明可以参看http://automaticthoughts.iteye.com/blog/1612388

简单线程池的代码实现

|  |
| --- |
| **public** **class** ThreadPoolExecutors {  //THANKS FOR code4crafer@gmail.com  **private** **int** threadNum;  //thread safe integer class, flag of current pool's alive thread  **private** AtomicInteger threadAlive = **new** AtomicInteger();  /\*\*  \* 1、ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。  \* 2、它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）  \*/  **private** ReentrantLock reentrantLock = **new** ReentrantLock();    **private** Condition condition = reentrantLock.newCondition();  **private** ExecutorService executorService;    **public** ThreadPoolExecutors(**int** threadNum) {  **this**.threadNum = threadNum;  **this**.executorService = Executors.*newFixedThreadPool*(threadNum);  }    **public** **int** getThreadAlive() {  **return** threadAlive.get();  }  **public** **int** getThreadNum() {  **return** threadNum;  }  **public** **void** execute(**final** Runnable runnable) {  // 判断当前线程池活动线程数是否大于线程池最大线程数  **if** (threadAlive.get() >= threadNum) {  **try** {  // 加锁  reentrantLock.lock();  **while** (threadAlive.get() >= threadNum) {  **try** {  // 如果当前线程池依然没有空闲线程，则等待，直到线程池有空闲线程，之后释放锁  condition.await();  } **catch** (InterruptedException e) {  }  }  } **finally** {  // 释放锁  reentrantLock.unlock();  }  }  // 增加降低线程池中活动线程数标记  threadAlive.incrementAndGet();  // 执行任务  executorService.execute(**new** Runnable() {  **public** **void** run() {  **try** {  //执行任务  runnable.run();  } **finally** {  **try** {  reentrantLock.lock();// 加锁  threadAlive.decrementAndGet();// 降低线程池中活动线程数标记  condition.signal();// 唤醒线程  } **finally** {  // 释放锁  reentrantLock.unlock();  }  }  }  });  }  **public** **boolean** isShutdown() {  **return** executorService.isShutdown();  }  **public** **void** shutdown() {  executorService.shutdown();  }  } |