多线程 / 高并发

1. stop() 和 suspend() 方法为何不推荐使用？

反对使用 stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态，那么其他线程能在那种状态下检查和修改它们。结果很难检查出真正的问题所在。

suspend() 方法容易发生死锁。调用 suspend() 的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被 "挂起" 的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。所以不应该使用 suspend()，而应在自己的 Thread 类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，便用 wait() 命其进入等待状态。若标志指出线程应当恢复，则用一个 notify() 重新启动线程。

2. sleep() 和 wait() 有什么区别?

sleep 就是正在执行的线程主动让出 cpu，cpu 去执行其他线程，在 sleep 指定的时间过后，cpu 才会回到这个线程上继续往下执行，如果当前线程进入了同步锁，sleep 方法并不会释放锁，即使当前线程使用 sleep 方法让出了 cpu，但其他被同步锁挡住了的线程也无法得到执行。wait 是指在一个已经进入了同步锁的线程内，让自己暂时让出同步锁，以便其他正在等待此锁的线程可以得到同步锁并运行，只有其他线程调用了 notify 方法（notify 并不释放锁，只是告诉调用过 wait 方法的线程可以去参与获得锁的竞争了，但不是马上得到锁，因为锁还在别人手里，别人还没释放。如果 notify

方法后面的代码还有很多，需要这些代码执行完后才会释放锁，可以在 notfiy 方法后增加一个等待和一些代码，看看效果），调用 wait 方法的线程就会解除 wait 状态和程序可以再次得到锁后继续向下运行。

3. 同步和异步有何异同，在什么情况下分别使用他们？

如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行同步存取。

当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。

4. 当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法?

- 其他方法前是否加了 synchronized 关键字，如果没加，则能。
- 如果这个方法内部调用了 wait，则可以进入其他 synchronized 方法。
- 如果其他个方法都加了 synchronized 关键字，并且内部没有调用 wait，则不能。
- 如果其他方法是 static，它用的同步锁是当前类的字节码，与非静态的方法不能同步，因为非静态的方法用的是 this。

5. 简述 synchronized 和 java.util.concurrent.locks.Lock 的异同？

主要相同点：Lock 能完成 synchronized 所实现的所有功能。

主要不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能。synchronized 会自动释放锁，而 Lock 一定要求程序员手工释放，并且必须在 finally

从句中释放。Lock 还有更强大的功能，例如，它的 tryLock 方法可以非阻塞方式去拿锁。

举例说明（对下面的题用 lock 进行了改写）

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ThreadTest {

  /**
   * @param args
   */

  private int j;
  private Lock lock = new ReentrantLock();
  public static void main(String[] args) {
      // TODO Auto-generated method stub
      ThreadTest tt = new ThreadTest();
      for(int i=0;i<2;i++)
      {
          new Thread(tt.new Adder()).start();
          new Thread(tt.new Subtractor()).start();
      }
  }
```

```
private class Subtractor implements Runnable
{

    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(true)
        {
            /*synchronized (ThreadTest.this) {
                System.out.println("j--=" + j--);
                //这里抛异常了，锁能释放吗？
            }*/
            lock.lock();
            try
            {
                System.out.println("j--=" + j--);
            }finally
            {
                lock.unlock();
            }
        }
    }
```

```
}

private class Adder implements Runnable
{

    @Override
    public void run() {
        // TODO Auto-generated method stub
        while(true)
        {
            /*synchronized (ThreadTest.this) {
            System.out.println("j++=" + j++);
            }*/
            lock.lock();
            try
            {
                System.out.println("j++=" + j++);
            }finally
            {
                lock.unlock();
            }
        }
    }
```

```
    }
}
```

6. 概括的解释下线程的几种可用状态。

- 新建 new。
- 就绪 放在可运行线程池中，等待被线程调度选中，获取 cpu。
- 运行 获得了 cpu。
- 阻塞
    - 等待阻塞 执行 wait() 。
    - 同步阻塞 获取对象的同步琐时，同步锁被别的线程占用。
    - 其他阻塞 执行了 sleep() 或 join() 方法)。
- 死亡。

7. 什么是 ThreadLocal?

ThreadLocal 用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择 ThreadLocal 变量。

每个线程都会拥有他们自己的 Thread 变量，它们可以使用 get()\set() 方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal 实例通常是希望它们同线程状态关联起来是 private static 属性。

8. run() 和 start() 区别。

run( )：只是调用普通 run 方法

start( )：启动了线程，由 Jvm 调用 run 方法

启动一个线程是调用 start() 方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行。这并不意味着线程就会立即运行。run() 方法可以产生必须退出的标志来停止一个线程。

9. 请说出你所知道的线程同步的方法。

wait()：使一个线程处于等待状态，并且释放所持有的对象的 lock。 **sleep()：**使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉 InterruptedException 异常。 **notify()：**唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且不是按优先级。 **notityAll()：**唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

10. 线程调度和线程控制。

线程调度（优先级）：

与线程休眠类似，线程的优先级仍然无法保障线程的执行次序。只不过，优先级高的线程获取 CPU 资源的概率较大，优先级低的并非没机会执行。线程的优先级用 1-10 之间的整数表示，数值越大优先级越高，默认的优先级为 5。 在一个线程中开启另外一个新线程，则新开线程称为该线程的子线程，子线程初始优先级与父线程相同。

线程控制

- **sleep( )** // 线程休眠 **join( )** // 线程加入 **yield( )** // 线程礼让 **setDaemon( )** // 线程守护

中断线程

- **stop( ) interrupt( )** ==(首先选用)==

11. 什么是线程饿死，什么是活锁？

当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI 中线程活锁可能发生在以下情形：

- 当所有线程在序中执行 Object.wait(0)，参数为 0 的 wait 方法。程序将发生活锁直到在相应的对象上有线程调用 Object.notify() 或者 Object.notifyAll()。
- 当所有线程卡在无限循环中。

12. 多线程中的忙循环是什么?

忙循环就是程序员用循环让一个线程等待，不像传统方法 wait(), sleep() 或 yield() 它们都放弃了 CPU 控制，而忙循环不会放弃 CPU，它就是在运行一个空循环。这么做的目的是为了保留 CPU 缓存。

在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

13. volatile 变量是什么？volatile 变量和 atomic 变量有什么不同？

volatile 则是保证了所修饰的变量的可见。因为 volatile 只是在保证了同一个变量在多线程中的可见性，所以它更多是用于修饰作为开关状态的变量，即 Boolean 类型的变量。

volatile 多用于修饰类似开关类型的变量、Atomic 多用于类似计数器相关的变量、其它多线程并发操作用 synchronized 关键字修饰。

volatile 有两个功用：

- 这个变量不会在多个线程中存在复本，直接从内存读取。
- 这个关键字会禁止指令重排序优化。也就是说，在 volatile 变量的赋值操作后面会有一个内存屏障（生成的汇编代码上），读操作不会被重排序到内存屏障之前。

14. volatile 类型变量提供什么保证？能使得一个非原子操作变成原子操作吗？

volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。

在 Java 中除了 long 和 double 之外的所有基本类型的读和赋值，都是原子性操作。而 64 位的 long 和 double 变量由于会被 JVM 当作两个分离的 32 位来进行操作，所以不具有原子性，会产生字撕裂问题。但是当你定义 long 或 double 变量时，如果使用 volatile 关键字，就会获到（简单的赋值与返回操作的）原子性。