java并发总结（二）

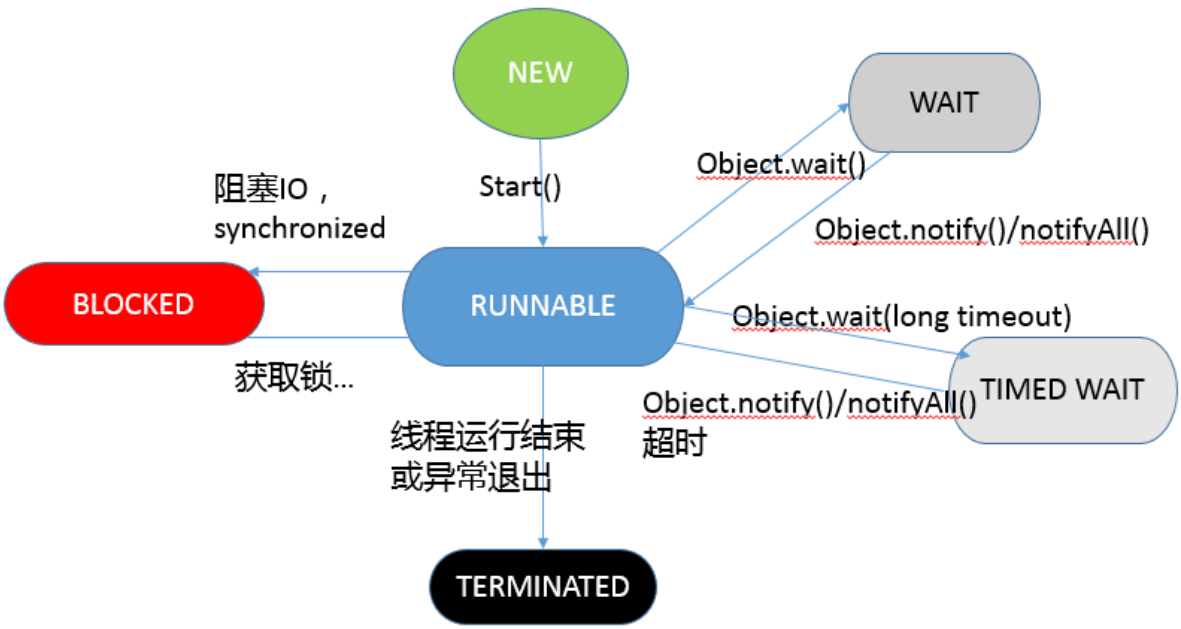
一.同一个线程能调用两次start()方法吗?

java的线程是不允许启动两次的，第二次启动会抛出异常，IllegalThreadStateException,这是一种运行时异常，多次调用start被认为是编程错误。

二.线程的生命周期

线程的生命周期存放在java.lang.Thread.State这个枚举里。

- 新建（NEW），表示线程被创建出来，但是没有启动，可以认为它是个java内部状态。
- 就绪（RUNNABLE），表示该线程已经在JVM中执行，当然由于执行需要计算cpu资源，所以它可能正在运行也有可能正在等待cpu分配给它的时间片。
- 阻塞（BLOCKED）,阻塞表示线程正在等待monitor lock，比如线程试图通过sychronized去获取某个锁，但是被其他线程所霸占，那么当前线程就处于阻塞状态。
- 等待（WAITING），表示正在等待其他线程采取某些操作。一个常见的场景就是生产者消费者线程，发现任务条件尚未满足，就让当前消费者线程等待，另外的生产者线程就会去准备数据，然后通过notify，通知消费者线程可以继续工作了。Thread.join()也会令线程进入等待状态。
- 计时等待（TIMED_WAIT）,其进入条件和等待状态类似，但是调用的是存在超时的条件的方法，比如wait,join等方法指定超时方法。`public final native void wait(long timeout) throws InterruptedException;`
- 终止（TERMINATED），不管是意外退出还是正常执行结束，线程已经完成使命，终止运行，也有人把这个状态叫做死亡。



三.什么是线程安全？如何保证线程安全?

线程安全是在多线程环境下正确性的概念，也就是保证多线程环境下**共享的，可修改的**数据或者状态的正确性。

按照定义来看，如何保证线程安全的方法，就不要让数据可共享，可修改，也就是**封装，不可变**的概念。

封装：我们可以把对象内部的状态隐藏起来，保护起来。

不可变：也就是final和immutable。

线程安全的几个基本特性需要满足：

1. **原子性**：其中一个线程已经开始对这个对象的内部状态操作了，其他线程不能干扰这个线程的操作，一般就是通过同步机制实现。
2. **可见性**：其中一个线程已经对这个对象的内部状态操作了，这个对象内部的状态要立刻通知其他线程，通常解释为将线程的本地状态反映到主内存上，一般使用volatile解决。
3. **有序性**：保证线程内串行语义，避免指令重排。

代码分析：

```
public class ThreadSafeSample {
    public int sharedState;
    public void nonSafeAction() {
        while (sharedState < 100000) {
            int former = sharedState++;
            int latter = sharedState;
            if (former != latter - 1) {
                System.out.printf("Observed data race, former is " +
                        former + ", " + "latter is " + latter);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadSafeSample sample = new ThreadSafeSample();
        Thread threadA = new Thread(){
            public void run(){
                sample.nonSafeAction();
            }
        };
        Thread threadB = new Thread(){
            public void run(){
                sample.nonSafeAction();
            }
       };
        threadA.start();
        threadB.start();
        threadA.join();
        threadB.join();
    }
}
```

这样的代码，运行结果肯定不会跟预料的一样，因为sharedState是被共享的，可能是threadA先运行了，这样可能改变了sharedState的初始值，然后threadB开始运行，导致sharedState的初始值不是0，程序不是线程安全的。

需要加上synchronized保证同步：

```
synchronized (this) {
    int former = sharedState ++;
    int latter = sharedState;
    // …
}
```

如果我们使用反编译工具,javap来看

```
11: astore_1
12: monitorenter
13: aload_0
14: dup
15: getfield    #2                  // Field sharedState:I
18: dup_x1
…
56: monitorexit
```

它有monitorenter和monitorexit。Monitor对象是同步实现的基本单元。

四.如何停止线程?

1. stop()

   这个方法天生就是不安全的，并且已经被废弃了。

   ```
   `@Deprecatedpublic final void stop() {    stop(new ThreadDeath());}`
   ```

   会造成死锁。抛出ThreadDeath异常。

2. interrupt()

   ```
   public void Thread.interrupt() // 无返回值
   public boolean Thread.isInterrupted() // 有返回值
   public static boolean Thread.interrupted() // 静态，有返回值
   ```

   ```
   public static boolean interrupted() {
       return currentThread().isInterrupted(true);
   }
   ```

   检测当前线程是否已经中断，是则返回true，否则false，**并清除中断状态**。换言之，如果该方法被连续调用两次，第二次必将返回false，除非在第一次与第二次的瞬间线程再次被中断。如果中断调用时线程已经不处于活动状态，则返回false。

   ```
   public boolean isInterrupted() {
       return isInterrupted(false);
   }
   ```

检测当前线程是否已经中断，是则返回true，否则false。中断状态不受该方法的影响。如果中断调用时线程已经不处于活动状态，则返回false。

> interrupted()与isInterrupted()的唯一区别是，**前者会读取并清除中断状态，后者仅读取状态。**

3. **停止一个线程的最佳方法是让它执行完毕，没有办法立即停止一个线程，但你可以控制何时或什么条件下让他执行完毕**

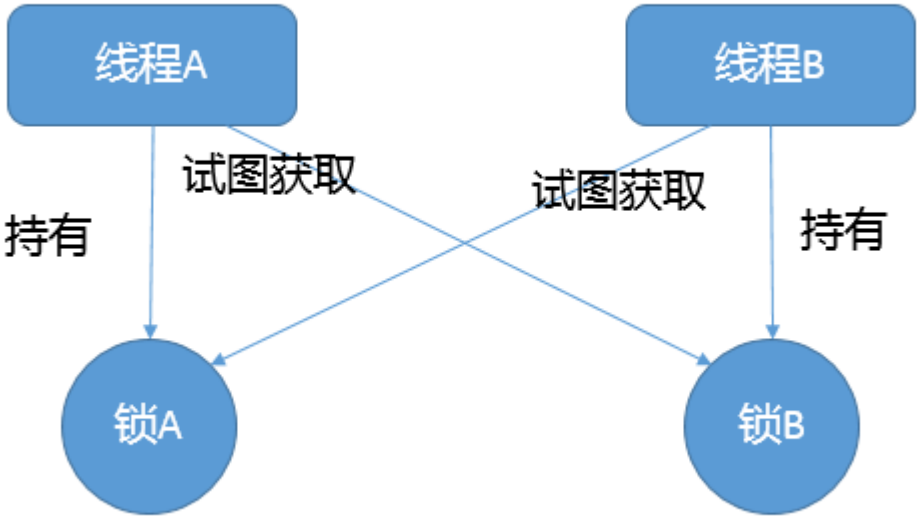
```
    public class BestPractice extends Thread {
    private volatile boolean finished = false;   // ① volatile条件变量
    public void stopMe() {
        finished = true;    // ② 发出停止信号
    }

    @Override
    public void run() {
        while (!finished) {    // ③ 检测条件变量
            // do dirty work   // ④业务代码
        }
    }
}`
```

五.Volatile

1. （适用于Java所有版本）读和写一个volatile变量有全局的排序。也就是说每个线程访问一个volatile作用域时会在继续执行之前读取它的当前值，而不是（可能）使用一个缓存的值。(但是并不保证经常读写volatile作用域时读和写的相对顺序，也就是说通常这并不是有用的线程构建)。
2. （适用于Java5及其之后的版本）volatile的读和写建立了一个happens-before关系，类似于申请和释放一个互斥锁

六.死锁

死锁是一种特定的程序状态，在实体之间，由于循环依赖导致彼此一直处于等待之中，没有任何个体可以继续前进。死锁不仅仅是在线程之间发生，存在资源独占的进程之间同样也可能出现死锁。



定位死锁最常用的方式就是利用jstack等工具获取栈。

1. 互斥条件，任务使用的资源至少有一个是不能共享的。
2. 至少有一个任务，它必须持有一个资源且正在等待获取一个当前被别的任务持有的资源。
3. 资源不能被任务抢占，任务必须把资源释放当做普通事件。
4. 必须有循环等待，这时，一个任务等待其他任务所持有的资源，后者有在等待另一个任务所持有的资源，这样一直下去，直到有一个任务在等待第一个任务所持有的资源，使得大家都被锁住。

```
public class DeadLock implements Runnable{
    A a = new A();
    B b = new B();

    public void init(){
        a.foo(b);
    }

    @Override
    public void run() {
        b.bar(a);
    }

    public static void main(String[] args) {
        DeadLock lock = new DeadLock();
        Thread t = new Thread(lock," test ");
        t.start();
        lock.init();
    }
}

class A{
    public synchronized void foo(B b){
        System.out.println("concurrent Threads'name is " +
Thread.currentThread().getName() + " start to execute foo() ");
```

```
        try {
            TimeUnit.SECONDS.sleep(1);
        }catch (InterruptedException e){
            e.printStackTrace();
        }

        b.b();
        System.out.println("concurrent Threads'name is " +
Thread.currentThread().getName() + " start to execute b() ");
    }

    public synchronized void a(){
        System.out.println("a.a() method");
    }
}

class B{
    public synchronized void bar(A a){
        System.out.println("concurrent Threads'name is " +
Thread.currentThread().getName() + " start to execute bar() ");
        try {
            TimeUnit.SECONDS.sleep(1);
        }catch (InterruptedException e){
            e.printStackTrace();
        }

        a.a();
        System.out.println("concurrent Threads'name is " +
Thread.currentThread().getName() + " start to execute a() ");

    }

    public synchronized void b(){
        System.out.println("b.b() method");
    }
}
```