链滴

Java 多线程并发第一步

作者：mnizht

原文链接：https://ld246.com/article/1579345041125

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

多线程-执行屏障

- 最近在看Java多线程的内容，看到个有意思的题库，看了多线程部分的一道简单的题。。。
- 嗯。。。果然是一点都不会。
- 看了别人的答案和解释，结合百度才算是明白答案。

1114.按序打印（这是原题）

难度：简单

我们提供了一个类：

```
public class Foo {
  public void one() { print("one"); }
  public void two() { print("two"); }
  public void three() { print("three"); }
}
```

三个不同的线程将会共用一个 Foo 实例。

- 线程 A 将会调用 one() 方法
- 线程 B 将会调用 two() 方法
- 线程 C 将会调用 three() 方法

请设计修改程序，以确保 two() 方法在 one() 方法之后被执行，three() 方法在 two() 方法之后被执
。

示例 1:

```
输入: [1,2,3]
输出: "onetwothree"
解释:
有三个线程会被异步启动。
输入 [1,2,3] 表示线程 A 将会调用 one() 方法，线程 B 将会调用 two() 方法，线程 C 将会调用 three()
方法。
正确的输出是 "onetwothree"。
```

示例 2:

```
输入: [1,3,2]
输出: "onetwothree"
解释:
输入 [1,3,2] 表示线程 A 将会调用 one() 方法，线程 B 将会调用 three() 方法，线程 C 将会调用 two()
方法。
正确的输出是 "onetwothree"。
```

注意:

尽管输入中的数字似乎暗示了顺序，但是我们并不保证线程在操作系统中的调度顺序。

你看到的输入格式主要是为了确保测试的全面性。

来源：力扣（LeetCode）

链接：https://leetcode-cn.com/problems/print-in-order

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

以下是题解

3个线程分别调用one,two,three方法，线程调度顺序不定，要求输出顺序始终是"onetwothree",这要求调用one方法的线程一定是第一个执行完的，two第二，three第三。但由于三个线程的启动顺序不定的，这就需要在one、two、three方法中增加限制线程执行的屏障。

下面是两个点赞最多的答案

Semaphore 信号量阻断

```
import java.util.concurrent.Semaphore;
class Foo {
    public Semaphore seam_first_two = new Semaphore(0);

    public Semaphore seam_two_second = new Semaphore(0);

    public Foo() {

    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        seam_first_two.release();
    }

    public void second(Runnable printSecond) throws InterruptedException {
        seam_first_two.acquire();
        printSecond.run();
        seam_two_second.release();
    }

    public void third(Runnable printThird) throws InterruptedException {
        seam_two_second.acquire();
        printThird.run();
    }
}
```

CountDownLatch 计数器锁（这里着重分析一下这种方法，他和上一种方法的底层现是一样的）

```
import java.util.concurrent.CountDownLatch;

class Foo {
     //定义两个计数器锁，计数都为1
    private CountDownLatch second = new CountDownLatch(1);
```

```
    private CountDownLatch third = new CountDownLatch(1);
    public Foo() {

    }

    public void first(Runnable printFirst) throws InterruptedException {

        // printFirst.run() outputs "first". Do not change or remove this line.
        printFirst.run();
        second.countDown(); //second计数器锁计数减一
    }

    public void second(Runnable printSecond) throws InterruptedException {

        second.await(); //将当前线程加入second的共享同步序列中，只有当计数为0时里面的线程才
被执行
        // printSecond.run() outputs "second". Do not change or remove this line.
        printSecond.run();
        third.countDown(); //third计数器锁计数减一
    }

    public void third(Runnable printThird) throws InterruptedException {

        third.await(); //将当前线程加入third的共享同步序列中，只有当计数为0时里面的线程才会被执

        // printThird.run() outputs "third". Do not change or remove this line.
        printThird.run();
    }
}
```

CountDownLatch 简单同步辅助程序

```
public class CountDownLatch{

  private static final class Sync extends AbstractQueuedSynchronizer{
    ...
    /**
    *尝试以共享模式获取。此方法应查询对象的状态是否允许在共享模式下获取该对象，如果允许则
取该对象。
    *具体哪种状态允许获取，由实现这个方法的人自己决定
    */
    protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
    }
  }
  private final Sync sync;
  ...
  /**
     * 使当前线程等待直到锁存器倒计时为零，除非线程是中断的
    * 如果当前计数是零，那么方法会立即返回
    * 如果当前计数大于零，则当前出于线程调度目的，线程将被禁用，并处于休眠状态，直到发生以
两种情况之一：
    *        由于其他线程调用使得计数为零
    *        一些其它线程中断了当前线程
```

```
     *    如果当前线程：
     *        在进入此方法时已经是中断状态；或者在循环等待时被中断了，那就会抛出 InterruptedExc
ption 异常，          *    且当前线程的中断状态会被清除。
     *
     */
    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    /**
     * 减少锁存器的计数，如果计数达到零，则释放所有等待的线程
     * 如果当前计数大于零，就减少计数
     * 如果新的计数等于零，那么所有等待的线程都会重新进入可执行状态以进行线程调度
     * 如果当前计数等于零，则什么都不做
     */
    public void countDown() {
        sync.releaseShared(1);
    }
```

AbstractQueuedSynchronizer 同步器基础类

```
/**
* 提供一个框架，用于实现依赖于先进先出(FIFO)等待队列的阻塞锁和相关同步器(信号量，事件等)。
类被设计为大多数
* 类型的同步器的基础，这些同步器依赖于单个原子int值来表示状态。
*/
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer
    implements java.io.Serializable {

    ...

    /**
     * 同步状态.子类自行定义该状态的值与对应的意义
     */
    private volatile int state;

    /**
    *    以共享模式获取，如果线程处于中断状态则终止。
    * 首先通过Thread.interrupted 方法校验线程是否是中断的，如果是就抛出异常。
    * 调用 tryAcquireShared（由CountDownLatch 内部类Sync实现） 校验线程状态是否支持获取
    */
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted()) //检查线程是否是中断状态
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0) //检查对象的状态是否允许在共享模式下获取对象
            doAcquireSharedInterruptibly(arg); //以共享可中断模式获取
    }


    /**
```

```
    * 以共享可中断模式获取
    * 每个线程进来都会创建一个自己的Node加入到SyncQueue(同步队列)，队列有
    */
    private void doAcquireSharedInterruptibly(int arg)
        throws InterruptedException {
        final Node node = addWaiter(Node.SHARED); //将对象以共享模式加入到等待队列
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    int r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        return;
                    }
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    throw new InterruptedException();
            }
        } catch (Throwable t) {
            cancelAcquire(node);
            throw t;
        }
    }


}

/**
* 这个抽象类为创建可能需要所有权概念的锁和相关的同步器提供了基础。
*/
public abstract class AbstractOwnableSynchronizer
    implements java.io.Serializable {

      ....
  /**
     * 独占模式同步的当前所有者
     */
    private transient Thread exclusiveOwnerThread;

  /**
     *设置当前拥有独占访问权限的线程。 一个null参数表示没有线程拥有访问权限。 此方法不会强加
何同步或volatile字段访问。
     */
    protected final void setExclusiveOwnerThread(Thread thread) {
        exclusiveOwnerThread = thread;
    }

    /**
     *返回setExclusiveOwnerThread最后设置的线程，或null如果从未设置。 这种方法不会强加任何
步或volatile字段访问。
     */
```

```
    protected final Thread getExclusiveOwnerThread() {
        return exclusiveOwnerThread;
    }

}
```

Thread检测线程是否是中断状态

```
public class Thread implements Runnable{
  /**
     * 检测当前线程是否是中断的。这个“中断状态”会被这个方法清除。换句话说，如果这个方法被功调用两次，第二        * 次调用就会返回false（除非线程在两次调用之间被再次中断）
     * 线程的中断会被忽略，由于线程处于非活动状态。这个方法会返回false。
     *
     */
    public static boolean interrupted() {
        return currentThread().isInterrupted(true);
    }

    /**
     * 检测线程是否是中断的。线程的“中断状态”不受这个方法影响
     * 处于非活动状态的线程，此方法会返回false
     */
    public boolean isInterrupted() {
        return isInterrupted(false);
    }

    /**
     * 检测线程是否是中断状态。线程的“中断状态”是否重置由 ClearInterrupted 参数决定。
     *    @HotSpotIntrinsicCandidate 中HotSpot是Java的一种虚拟机，这个方法的实现由虚拟机完

     */
    @HotSpotIntrinsicCandidate
    private native boolean isInterrupted(boolean ClearInterrupted);
}
```

AbstractQueueSynchronizer.Node

机翻加查资料，我理解的Node大概就是一个个线程的载体。每一个Node实体都对应一个线程，里面储着线程的id，使用 `waitStatus` 字段标识线程的等待状态。只有某些特定状态的线程才是可执行的允许争抢锁从而被调用）。
想要控制线程的执行顺序，就通过AbstractQueueSynchronizer 去创建一个SyncQueue（同步队列，这个队列可以有两种模式
    1.SHARED 共享模式：队列中允许有多个可执行的线程，具体谁能够获取锁看实际调度情况。
    2.EXCLUSIVE 独占模式：队列中仅允许有一个可执行的线程。

再通过 Semaphore 信号量阻断 或 CountDownLatch 计数器锁 的方式创造获取所得时间节点（比计数器锁中的计数为0时，队列中的线程会根据等待状态被唤醒和调度）。

```
/**
     * Wait queue node class.
     *
     * <p>The wait queue is a variant of a "CLH" (Craig, Landin, and
```

```
* Hagersten) lock queue. CLH locks are normally used for
* spinlocks.  We instead use them for blocking synchronizers, but
* use the same basic tactic of holding some of the control
* information about a thread in the predecessor of its node.  A
* "status" field in each node keeps track of whether a thread
* should block.  A node is signalled when its predecessor
* releases.  Each node of the queue otherwise serves as a
* specific-notification-style monitor holding a single waiting
* thread. The status field does NOT control whether threads are
* granted locks etc though.  A thread may try to acquire if it is
* first in the queue. But being first does not guarantee success;
* it only gives the right to contend.  So the currently released
* contender thread may need to rewait.
*
* <p>To enqueue into a CLH lock, you atomically splice it in as new
* tail. To dequeue, you just set the head field.
* <pre>
*      +------+  prev +-----+       +-----+
* head |      | <---- |     | <---- |     |  tail
*      +------+       +-----+       +-----+
* </pre>
*
* <p>Insertion into a CLH queue requires only a single atomic
* operation on "tail", so there is a simple atomic point of
* demarcation from unqueued to queued. Similarly, dequeuing
* involves only updating the "head". However, it takes a bit
* more work for nodes to determine who their successors are,
* in part to deal with possible cancellation due to timeouts
* and interrupts.
*
* <p>The "prev" links (not used in original CLH locks), are mainly
* needed to handle cancellation. If a node is cancelled, its
* successor is (normally) relinked to a non-cancelled
* predecessor. For explanation of similar mechanics in the case
* of spin locks, see the papers by Scott and Scherer at
* http://www.cs.rochester.edu/u/scott/synchronization/
*
* <p>We also use "next" links to implement blocking mechanics.
* The thread id for each node is kept in its own node, so a
* predecessor signals the next node to wake up by traversing
* next link to determine which thread it is.  Determination of
* successor must avoid races with newly queued nodes to set
* the "next" fields of their predecessors.  This is solved
* when necessary by checking backwards from the atomically
* updated "tail" when a node's successor appears to be null.
* (Or, said differently, the next-links are an optimization
* so that we don't usually need a backward scan.)
*
* <p>Cancellation introduces some conservatism to the basic
* algorithms.  Since we must poll for cancellation of other
* nodes, we can miss noticing whether a cancelled node is
* ahead or behind us. This is dealt with by always unparking
* successors upon cancellation, allowing them to stabilize on
* a new predecessor, unless we can identify an uncancelled
```

```
     * predecessor who will carry this responsibility.
     *
     * <p>CLH queues need a dummy header node to get started. But
     * we don't create them on construction, because it would be wasted
     * effort if there is never contention. Instead, the node
     * is constructed and head and tail pointers are set upon first
     * contention.
     *
     * <p>Threads waiting on Conditions use the same nodes, but
     * use an additional link. Conditions only need to link nodes
     * in simple (non-concurrent) linked queues because they are
     * only accessed when exclusively held.  Upon await, a node is
     * inserted into a condition queue.  Upon signal, the node is
     * transferred to the main queue.  A special value of status
     * field is used to mark which queue a node is on.
     *
     * <p>Thanks go to Dave Dice, Mark Moir, Victor Luchangco, Bill
     * Scherer and Michael Scott, along with members of JSR-166
     * expert group, for helpful ideas, discussions, and critiques
     * on the design of this class.
     */
    static final class Node {

    }
```

水平有限，如发现有错误的地方，欢迎指正。若有不同见解，也欢迎讨论。

关于线程是如何获取锁的部分，如doAcquireSharedInterruptibly的方法是如何实现的，暂时还没搞，之后明白了再补一篇。

参考：

https://blog.csdn.net/u010577768/article/details/79929346