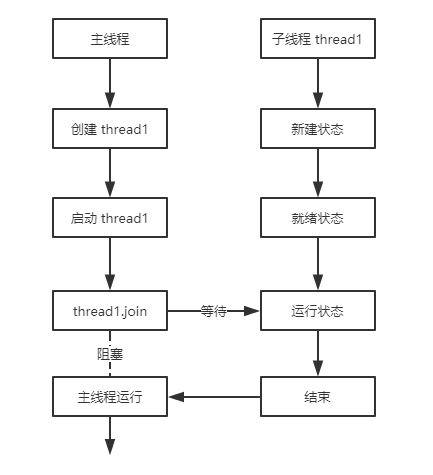
**如何控制线程打印顺序**

**使用 join 方法**

如下面代码所示：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | public class ThreadDemo {   public static void main(String[] args) throws InterruptedException {  final Thread thread1 = new Thread(() -> System.out.println("thread1"));  final Thread thread2 = new Thread(() -> System.out.println("thread2"));  final Thread thread3 = new Thread(() -> System.out.println("thread3"));   thread1.start();  thread1.join();  thread2.start();  thread2.join();  thread3.start();  thread3.join();  } } |

通过 join 方法去保证多线程的顺序的特性，join 主要让主线程等待子线程结束后才能继续运行。

[](http://blog.cuzz.site/2019/07/23/%E9%9D%A2%E8%AF%95%E9%AB%98%E9%A2%91%E9%A2%98/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20190717155358.png)

**使用 Executors.new**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | public class ThreadDemo {   public static void main(String[] args) throws InterruptedException {  final Thread thread1 = new Thread(() -> System.out.println("thread1"));  final Thread thread2 = new Thread(() -> System.out.println("thread2"));  final Thread thread3 = new Thread(() -> System.out.println("thread3"));   ExecutorService executorService = Executors.newSingleThreadExecutor();  executorService.submit(thread1);  executorService.submit(thread2);  executorService.submit(thread3);  executorService.shutdown();  } } |

看看源码就知道了，使用阻塞队列来维护任务，是任务有序进行。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | /\*\*  \* Creates an Executor that uses a single worker thread operating  \* off an unbounded queue. (Note however that if this single  \* thread terminates due to a failure during execution prior to  \* shutdown, a new one will take its place if needed to execute  \* subsequent tasks.) Tasks are guaranteed to execute  \* sequentially, and no more than one task will be active at any  \* given time. Unlike the otherwise equivalent  \* {@code newFixedThreadPool(1)} the returned executor is  \* guaranteed not to be reconfigurable to use additional threads.  \*  \* @return the newly created single-threaded Executor  \*/ public static ExecutorService newSingleThreadExecutor() {  return new FinalizableDelegatedExecutorService  (new ThreadPoolExecutor(1, 1,  0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>())); } |

**Java 中的 volatile 和 synchronized**

* JMM（Java Memory Mode）：解决并发过程中可见性、原子性和有序性的问题。
* 并发过程中的两个关键问题
  + 线程之间如何通信
    - 共享内存：隐身通信
    - 消息传递：显示通信，如 wait() / notify() / notifyall()
  + 线程之间如何同步
    - 线程同步指的是：程序用于控制不同线程先后顺序的机制。
    - 在消息传递的并发模型中，由于消息的发送必需在消息接受之前，所以是隐式同步。
* 定位内存可见行问题
  + 什么是内存共享
* synchronized：可重入锁、互斥性和可见性
* volatile：可以做到原子性、可见性不能做到复合操作的原子性
* volatile 保证可见性原理
  + 对于 volatile 修饰的变量进行写操作的时候，JVM 会向处理器发送一条 Lock 前缀的指令，会把这个变量所在缓存行的数据写回系统内存。
  + 在多处理器的情况下，各个处理器会通过总线传过的的值来发现自己保存的值是不是过期了，实现缓存一致性。当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。
* synchronized 是关键字属于 JVM 层面，反应在字节码上是 monitorenter 和 monitorexit，其底层是通过 monitor 对象来完成，其实 wait/notify 等方法也是依赖 monitor 对象只有在同步快或方法中才能调用 wait/notify 等方法。

**新建一个线程的几种方式**

* Thread
* Rannable
* Callable/Future
* ExectutorService

[**分布式锁的实现**](http://www.hollischuang.com/archives/1716)

* 通过数据库实现
* 使用 zookeeper 实现
* 基于 redis 实现

**如果通过一个 swap 交换两个数**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | public class SwapDemo {   public static void main(String[] args) {  Integer a = 1;  Integer b = 2;  System.out.println("before : a = " + a + ", " + "b = " + b);  swap(a, b);  System.out.println("after : a = " + a + ", " + "b = " + b);  }   private static void swap(Integer a, Integer b) {  int temp = a;  a = b;  b = temp;  } } // 输出 // before : a = 1, b = 2 // after : a = 1, b = 2 |

发现并没有交换，因为 Java 使用的值传递，传递的只是一个副本，如果是基本类型就基本类型副本，如果是引用类型就是引用类型的副本。

查看源码，发现 Integer 这个类有一个 final 成员变量。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | /\*\*  \* The value of the {@code Integer}.  \*  \* @serial  \*/ private final int value; |

我们知道 final 修饰的变量不能修改，但是我们可以通过反射修改。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | public class SwapDemo {   public static void main(String[] args) throws Exception{  Integer a = 1; // 同 Integer.valueOf(1)  Integer b = 2;  System.out.println("before : a = " + a + ", " + "b = " + b);  swap(a, b);  System.out.println("after : a = " + a + ", " + "b = " + b);  }   private static void swap(Integer a, Integer b) throws NoSuchFieldException, IllegalAccessException {  Field field = Integer.class.getDeclaredField("value");  field.setAccessible(true); // 访问私有变量  Integer temp = new Integer(a.intValue()); // 因为 -128 ~ 127 会缓冲，所以要 new 出来  field.set(a, b.intValue());  field.set(b, temp);  } } // 输出 // before : a = 1, b = 2 // after : a = 2, b = 1 |