Thread.join() 的原理分析

作者：gitzzzf
原文链接：https://ld246.com/article/1577005411972
来源网站：链滴

一、Thread.join()方法的作用

首先，说到Thread.join()方法，如果我们了解过java并发知识的同学可能都知道，我们可以用Thread.oin()方法来控制线程的执行顺序，顾名思义也能知道，join嘛，加入的意思，也就是让当前正在执行线程等待，让加入的线程先执行完，然后唤醒当前主线程，再去执行当前主线程。举个例子如下。

```
public class JoinDemo extends Thread {
    Thread previousThread;

    public JoinDemo(Thread previousThread) {
        this.previousThread = previousThread;
    }

    @Override
    public void run() {
        try {
            // 调用线程的join方法
            previousThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("我是线程：" + Thread.currentThread().getName() + " ,我要等待线程：
+ previousThread.getName() + "执行完！");
    }

    public static void main(String[] args) throws Exception {
        Thread previousThread = Thread.currentThread();
        for (int i = 0; i < 10; i++) {
            JoinDemo joinDemo = new JoinDemo(previousThread);
            joinDemo.setName("子线程" + i);
            joinDemo.start();
```

```
            previousThread = joinDemo;
        }
        Thread.sleep(1000);
        System.out.println("我现在循环完了，你们可以输出了...");
    }
}
```

输出结果如下：

```
我现在循环完了，你们可以输出了...
我是线程：子线程0 ,我要等待线程：main执行完!
我是线程：子线程1 ,我要等待线程：子线程0执行完!
我是线程：子线程2 ,我要等待线程：子线程1执行完!
我是线程：子线程3 ,我要等待线程：子线程2执行完!
我是线程：子线程4 ,我要等待线程：子线程3执行完!
我是线程：子线程5 ,我要等待线程：子线程4执行完!
我是线程：子线程6 ,我要等待线程：子线程5执行完!
我是线程：子线程7 ,我要等待线程：子线程6执行完!
我是线程：子线程8 ,我要等待线程：子线程7执行完!
我是线程：子线程9 ,我要等待线程：子线程8执行完!
```

很明显，Thread.join()能很好的控制线程的执行顺序。

二、Thread.join()方法的原理与深入分析

既然我们知道了join()方法的作用，那么它到底是如何控制线程的执行顺序的呢，下面一一给你揭晓。

首先，调用完join()方法，它是如何让当前线程进入等待状态的?

结合上面实例的输出结果，我们来分析这个问题。我们看终端输出的日志中这一行：

```
我是线程：子线程0 ,我要等待线程：main执行完!
```

我们在for第一层中，首先构建对象JoinDemo joinDemo = new JoinDemo(previousThread);传递去的也就是主线程 -- main线程。然后调用start()方法，启动子线程执行。

这个时候，子线程执行run()方法体的时候，执行到 previousThread.join()，根据我们对join()方法的解，它会让当前线程阻塞，当前线程也就是线程名为”子线程0“的线程。那么join()是如何阻塞当前程（子线程0）的呢？我们跟进去join()方法的源码看一看，看看到底有何蹊跷。

```
public final synchronized void join(long millis)
throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
```

```
            while (isAlive()) {
                long delay = millis - now;
                if (delay <= 0) {
                    break;
                }
                wait(delay);
                now = System.currentTimeMillis() - base;
            }
        }
    }
```

找到重点，如下几行代码：

```
            while (isAlive()) {
                wait(0);
            }
```

也就是说在子线程0中执行previousThread.join()时候，previousThread代表的是main线程，那么joi()方法的含义也就变成了判断main线程是不是活着的，如果是活着的，则调用wait()方法，来阻塞当线程，所以当前”子线程0“也就被阻塞了。wait()方法是如何阻塞”子线程0“的呢？

我们知道wait()方法，是Object对象的方法，wait()方法的含义是，当前线程A中调用xxx.wait()，会当前线程A加入到等待xxx对象锁（JVM会给每一个对象都分配一把唯一的锁，这把锁是在对象中）的待队列中去，等待其他线程调用xxx.notify()或者xxx.notifyAll()来唤醒在等待队列中的线程。

这里我们顺带说一句，正是因为xxx.wait()方法，会把当前线程加入到xxx对象锁的等待队列中，当前程在调用xxx.wait()方法的时候一定是持有xxx对象锁的，所以wait()方法一般放在synchronized方法者synchronized代码块中执行。

至此，我们已经知道”子线程0“在调用完previousThread.join()方法后，为何进入等待状态了。同，我们也知道进入等待状态的线程，需要其他线程调用notify()或者notifyAll()来唤醒，我们看join()法源码只看到了调用wait()方法进行了阻塞，没有看到在哪儿调用notify()方法唤醒等待的线程啊，那到底是如何唤醒的呢？这里就需要翻看jdk的源码了。

在hotspot的源码中找到 thread.cpp，看看线程退出以后有没有做相关的事情。

```
void JavaThread::exit(bool destroy_vm, ExitType exit_type) {
  assert(this == JavaThread::current(),  "thread consistency check");
  ...
  // Notify waiters on thread object. This has to be done after exit() is called
  // on the thread (if the thread is the last thread in a daemon ThreadGroup the
  // group should have the destroyed bit set before waiters are notified).
  ensure_join(this);
  assert(!this->has_pending_exception(), "ensure_join should have cleared");
  ...
```

我们注意看ensure_join(this); 上面的注释，通知唤醒等待在这个线程对象锁的其他阻塞线程们，这个在线程终止之后做的事情，我们再跟进ensure_join(this) 这个方法看下。

```
static void ensure_join(JavaThread* thread) {
  // We do not need to grap the Threads_lock, since we are operating on ourself.
  Handle threadObj(thread, thread->threadObj());
  assert(threadObj.not_null(), "java thread object must exist");
  ObjectLocker lock(threadObj, thread);
  // Ignore pending exception (ThreadDeath), since we are exiting anyway
  thread->clear_pending_exception();
```

```
  // Thread is exiting. So set thread_status field in  java.lang.Thread class to TERMINATED.
  java_lang_Thread::set_thread_status(threadObj(), java_lang_Thread::TERMINATED);
  // Clear the native thread instance - this makes isAlive return false and allows the join()
  // to complete once we've done the notify_all below
  // 这里是清除native线程，这个操作会导致isAlive()方法返回false
  java_lang_Thread::set_thread(threadObj(), NULL);
  // 调用notifyAll方法
  lock.notify_all(thread);
  // Ignore pending exception (ThreadDeath), since we are exiting anyway
  thread->clear_pending_exception();
}
```

重点关注其中如下部分

```
  // 这里是清除native线程，这个操作会导致isAlive()方法返回false
  java_lang_Thread::set_thread(threadObj(), NULL);
  // 调用notifyAll方法
  lock.notify_all(thread);
```

也就是说，任何线程在退出时候，都会将isAlive()置为false，同时通知唤醒等待在这个线程对象锁上的其他线程们。

那么，回到我们上面提到的”子线程0“上面来，我们也就知道了，它在”main“线程的对象锁的等队列上，在”main“线程执行完退出的时候”子线程0“也就会被唤醒。

三、说点题外话

止于此，关于Thread.join()方法的分析完毕。下面说点题外话，写这篇文章的目的是告诉自己，任何自己学习的过程中，希望自己能深挖一步，知其然，更要知其所以然，才能掌握理解的更加透彻，学整个java的过程中，或者说各种框架、中间件的过程中，要了解的更加深入一些，不要永远只停留在用的层面。当你多尝试深挖几次之后，会发现很多问题的思路想法都是相通的。这个时候的自己分析题、解决问题的能力才能得到提升。

知易，行难。

以上，与君共勉。