07 深入Thread类—线程API精讲

更新时间：2019-09-25 15:53:08

> 每个人的生命都是一只小船，理想是小船的风帆。
>
> ——张海迪

前面几节我们都在围绕着如何创建 Thread 和 启动 Thread 做分析。上节我们讲解了 Thread 的几种状态，以及状态间的变化。有些状态的变化是被动发生的，比如 run 方法执行完后进入 TERMINATED 状态。不过更多时候，状态的变化是由于主动调用了某些方法。而这些方法大多数是 Thread 类的 API。本小结，我们就来重点学习下 Thread 类暴露出来的 API。

sleep 方法

顾名思义，线程的 sleep 方法会使线程休眠指定的时间长度。休眠的意思是，当前逻辑执行到此不再继续执行，而是等待指定的时间。但在这段时间内，该线程持有的 monitor 锁（锁在后面会讲解，这里可以认为对共享资源的独占标志）并不会被放弃。我们可以认为线程只是工作到一半休息了一会，但它所占有的资源并不会交还。这样设计很好理解，因为线程在 sleep 的时候可能是处于同步代码块的中间位置，如果此时把锁放弃，就违背了同步的语义。所以 sleep 时并不会放弃锁，等过了 sleep 时长后，可以确保后面的逻辑还在同步执行。

sleep 方法有两个重载，分别是：

```
public static native void sleep(long millis) throws InterruptedException;
public static void sleep(long millis, int nanos) throws InterruptedException
```

两者的区别只是一个支持休眠时间到毫秒级，另外一个到纳秒级。但其实第二个并不能真的精确到纳秒级别，我们来看第二个重载方法代码：

```
public static void sleep(long millis, int nanos)
throws InterruptedException {
    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }
    if (nanos < 0 || nanos > 999999) {
        throw new IllegalArgumentException(
                    "nanosecond timeout value out of range");
    }
    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
        millis++;
    }
    sleep(millis);
}
```

可以清楚的看到，最终调用的还是第一个毫秒级别的 sleep 方法。而传入的纳秒会被四舍五入。如果大于 50 万，毫秒 ++，否则纳秒被省略。

yield 方法

yield 方法我们平时并不常用。yield 单词的意思是让路，在多线程中意味着本线程愿意放弃 CPU 资源，也就是可以让出 CPU 资源。不过这只是给 CPU 一个提示，当 CPU 资源并不紧张时，则会无视 yield 提醒。如果 CPU 没有无视 yield 提醒，那么当前 CPU 会从 RUNNING 变为 RUNNABLE 状态，此时其它等待 CPU 的 RUNNABLE 线程，会去竞争 CPU 资源。讲到这里有个问题，刚刚 yield 的线程同为 RUNNABLE 状态，是否也会参与竞争再次获得 CPU 资源呢？经过我大量测试，刚刚 yield 的线程是不会马上参与竞争获得 CPU 资源的。

我们看下面测试代码：

```
public class YieldExampleClient {

    public static void main(String[] args) {
        Thread xiaoming = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("小明--" + i);
//                if (i == 2) {
//                    Thread.yield();
//                }
            }
        });

        Thread jianguo = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("建国--" + i);
            }
        });

        xiaoming.start();
        jianguo.start();
    }
}
```

可以看到启动两个线程打印，控制台输出如下：

```
小明--0
小明--1
小明--2
小明--3
小明--4
小明--5
小明--6
小明--7
小明--8
小明--9
建国--0
建国--1
建国--2
建国--3
建国--4
建国--5
建国--6
建国--7
建国--8
建国--9
```

每次结果有所区别，但是一般都是小明输出到 5 以后，建国才开始输出。这一是因为线程启动需要时间，另外也是因为 CPU 紧张， jianguo 线程在排队。

我们放开小明线程注解部分，让输出到 xiaoming 线程输出到 2 的时候 yield ，看看会怎么样。输出如下：

```
小明--0
小明--1
小明--2
建国--0
建国--1
......
```

我们看前四行，xiaoming 先获得了 CPU 的使用权，不过在打印到 2 的时候调用了 yield 方法，提示可以让出 CPU 的使用权，而此时 CPU 接受了提示，从而让建国获得了 CPU 的使用权。我尝试建立更多的线程，多次尝试，发现小明打印到 2 的时候，肯定会切换为其它线程打印。不过如果 CPU 资源丰富，那么会无视 yield 方法，xiaoming 也无需让出 CPU 资源。

yield 方法为了提升线程间的交互，避免某个线程长时间过渡霸占 CPU 资源。但 yield 在实际开发中用的比较少，源码的注解也提到这一点：“*It is rarely appropriate to use this method.*”。

currentThread 方法

我们前几节中已经使用过该方法，这是一个静态方法，用于获取当前线程的实例。用法很简单，如下：

```
Thread.currentThread();
```

拿到线程的实例后，我们还可以获取 Thread 的 名称：

```
Thread.currentThread().getName();
```

这两个方法在之前例子中我们都使用过，也比较简单，就不再赘述。

此外我们还可以获取线程 ID ：

```
Thread.currentThread().getId();
```

setPriority 方法

此方法用于设置线程的优先级。每个线程都有自己的优先级数值，当 CPU 资源紧张的时候，优先级高的线程获得 CPU 资源的概率会更大。请注意仅仅是概率会更大，并不意味着就一定能够先于优先级低的获取。这和摇车牌号一个道理，我现在中签概率是标准的 9 倍，但摇中依然摇摇无期。而身边却时不时的出现第一次摇号就中的朋友。如果在 CPU 比较空闲的时候，那么优先级就没有用了，人人都有肉吃，不需要摇号了。

优先级别高可以在大量的执行中有所体现。在大量数据的样本中，优先级高的线程会被选中执行的次数更多。

最后我们看下 setPriority 的源码：

```
public final void setPriority(int newPriority) {
    ThreadGroup g;
    checkAccess();
    if (newPriority > MAX_PRIORITY || newPriority < MIN_PRIORITY) {
        throw new IllegalArgumentException();
    }
    if((g = getThreadGroup()) != null) {
        if (newPriority > g.getMaxPriority()) {
            newPriority = g.getMaxPriority();
        }
        setPriority0(priority = newPriority);
    }
}
```

Thread 有自己的最小和最大优先级数值，范围在 1-10。如果不在此范围内，则会报错。另外如果设置的 priority 超过了线程所在组的 priority ，那么只能被设置为组的最高 priority 。最后通过调用 native 方法 setPriority0 进行设置。

interrupt 相关方法

interrupt 的意思是打断。调用了 interrupt 方法后，线程会怎么样？不知道你的答案是什么。我在第一次学习 interrupt 的时候，第一感觉是让线程中断。其实，并不是这样。inerrupt 方法的作用是让可中断方法，比如让 sleep 中断。也就是说其中断的并不是线程的逻辑，中断的是线程的阻塞。这一点在本小结一开始就要彻底搞清池，否则带着错误的想法会影响学习的效果。

那么 interrupt 方法调用后，对未使用可中断方法的线程有影响吗？我们做个简单的实验，代码如下：

```
public class InterruptClient {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(()->{
            for(int i=0; i<100 ;i++){
                System.out.println("I'm doing my work");
                System.out.println("I'm interrupted?"+Thread.currentThread().isInterrupted());
            }
        });
        thread.start();
        Thread.sleep(1);
        thread.interrupt();
    }
}
```

线程 run 方法中没有调用可中断方法，只是输出 **I'm doing my work**，另外还会输出自己的中断状态。而主线程会 sleep 一毫秒，留时间给 thread 线程启动，然后调用 thread 线程的 interrupt 方法。我截取其中关键一段输出如下：

```
I'm doing my work
I'm interrupted?false
I'm doing my work
I'm interrupted?true
I'm doing my work
I'm interrupted?true
```

这段后面的输出一直到结束，都在重复 “I'm doing my work I'm interrupted?true“，这说明两个问题：

1. 调用 interrupt 方法，并不会影响可中断方法之外的逻辑。线程不会中断，会继续执行。这里的中断概念并不是指中断线程；
2. 一旦调用了 interrupt 方法，那么线程的 interrupted 状态会一直为 ture（没有通过调用可中断方法或者其他方式主动清除标识的情况下）；

通过上面实现我们了解了 interrupt 方法中断的不是线程。它中断的其实是可中断方法，如 sleep 。可中断方法被中断后，会把 interrupted 状态归位，改回 false 。

我们还是做个实验，代码如下：

```
public class InterruptSleepClient {
    public static void main(String[] args) throws InterruptedException {
        Thread xiaopang = new Thread(()->{
            for(int i=0; i<100 ;i++){
                System.out.println("I'm doing my work");
                try {
                    System.out.println("I will sleep");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("My sleeping was interrupted");
                }
                System.out.println("I'm interrupted?"+Thread.currentThread().isInterrupted());
            }
        });
        xiaopang.start();
        Thread.sleep(1);
        xiaopang.interrupt();
    }
}
```

这次干活的小胖比较懒，每次干完活都要休息一秒钟。有一次被让他干活的老师发现，把他叫醒了。但是后来看他照睡不误，也就随他去了。这段代码执行结果如下：

```
I'm doing my work
I will sleep
My sleeping was interrupted
I'm interrupted? false
I'm doing my work
I will sleep
I'm interrupted? false
I'm doing my work
I will sleep
I'm interrupted? false
```

可以看到当 xiaopang.interrupt () 执行后，睡眠中的 xiaopang 被唤醒了。这里额外需要注意的是，此时 xiaopang 线程的 interrupted 状态还是 false 。因为可中断线程会捕获中断的信号，并且会清除掉 interrupted 标识。因此输出的 “I’m interrupted ?” 全部是 false 。

最后我们再看一下静态方法 interrupted 。这个方法其实和成员方法 isInterrupted 方法类似，都是返回了 interrupted 状态。不同就是 interrupted 方法返回状态后，如果为 true 则会清除掉状态。而 isInterrupted 则不会。上面第一段测试代码已经验证了这一点，被打断后，调用 isInterrupted 一直返回 true。

下面我们来验证下 interrupted 是否会清除标识位。把第一段代码稍微改一下：

```
public class InterruptedClient {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(()->{
            for(int i=0; i<100 ;i++){
                System.out.println("I'm doing my work");
//原代码  System.out.println("I'm interrupted?"+Thread.currentThread().isInterrupted());
                System.out.println("I'm interrupted?"+Thread.interrupted());
            }
        });

        thread.start();
        Thread.sleep(1);
        thread.interrupt();
    }
}
```

改动已经在注解中说明，仅仅是改了获取 interrupted 状态的方法。但输出结果却是不一样的：

```
I'm doing my work
I'm interrupted?false
I'm doing my work
I'm interrupted?false
I'm doing my work
I'm interrupted?true
I'm doing my work
I'm interrupted?false
I'm doing my work
I'm interrupted?false
```

可以看到在输出 “I’m interrupted?true” 后，中断状态又变回了 false。

通过以上讲解，可以看出 interrupt 方法只是设置了中断标识位，这个标识位只对可中断方法会产生作用。不过我们还可以利用它做更多的事情，比如说如果线程的 run 方法中这么写：

```
while(!isInterrupted()){
//do somenting
}
```

这样主线程中可以通过调用此线程的 interrupt 方法，让其推出运行。此时 interrupted 的含义就真的是线程退出了。不过假如你的 while 循环中调用了可中断方法，那么就会有干扰。

join 方法

最后我们再讲解一个重要的方法 join。这个方法功能强大，也很实用。我们用它能够实现并行化处理。比如主线程需要做两件没有相互依赖的事情，那么可以起 A、B 两个线程分别去做。通过调用 A、B 的 join 方法，让主线程 block 住，直到 A、B 线程的工作全部完成，才继续走下去。我们来看下面这段代码：

```
public class JoinClient {
    public static void main(String[] args) throws InterruptedException {
        Thread backendDev = createWorker("backed dev", "backend coding");
        Thread frontendDev = createWorker("frontend dev", "frontend coding");
        Thread tester = createWorker("tester", "testing");

        backendDev.start();
        frontendDev.start();

//        backendDev.join();
//        frontendDev.join();

        tester.start();
    }

    public static Thread createWorker(String role, String work) {
        return new Thread(() -> {
            System.out.println("I finished " + work + " as a " + role);
        });
    }
}
```

这段代码中，我们把 join 方法去掉。执行结果如下：

```
I finished backend coding as a backed dev
I finished testing as a tester
I finished backend coding as a frontend dev
```

我们期望的是前端和后端开发完成工作后，测试才开始测试。但从输出结果看并非如此。要想实现这个需求，我们只需把注释打开，让 **backendDev** 和 **frontendDev** 先做 join 操作，此时主线程会被 block 住。直到 **backendDev** 和 **frontendDev** 线程都执行结束，才会继续往下执行。输出如下：

```
I finished backend coding as a backed dev
I finished frontend coding as a frontend dev
I finished testing as a tester
```

可以看到现在的输出完全符合我们的期望。可见调用 join 方法后 block 的并不是被调用的 **backendDev** 或 **frontendDev 线程**，而是调用方线程，这个需要牢记。

总结

本小结讲解了 Thread 的几个常用的方法，这些方法在我们实际开发中会经常用到的，需要我们认真学习和理解。有些已经被弃用的方法没有再讲解，比如 stop 方法。关于更多的方法，其实读者可以直接阅读 Thread 源代码，Thread 类的注解写得相当详细。其实很多时候我们自己动手直接阅读源代码和注解，是更为快捷的学习方式，而且也更为权威。

下一节我们继续讲解线程的相关操作 wait ()、notify ()、notifyAll ()。这些方法也会改变线程的状态，但并不是 Thread 的 API 。

}

06 线程什么时候开始真正执行？—线程的状态详解

08 集体协作，什么最重要？沟通！—线程的等待和通知