二

02 如何正确停止线程？为什么 volatile 标记位的停止方法是错误的？

在本课时我们主要学习如何正确停止一个线程？以及为什么用 volatile 标记位的停止方法是错误的？

首先，我们来复习如何启动一个线程，想要启动线程需要调用 Thread 类的 start() 方法，并在 run() 方法中定义需要执行的任务。启动一个线程非常简单，但如果想要正确停止它就没那么容易了。

原理介绍

通常情况下，我们不会手动停止一个线程，而是允许线程运行到结束，然后让它自然停止。但是依然会有许多特殊的情况需要我们提前停止线程，比如：用户突然关闭程序，或程序运行出错重启等。

在这种情况下，即将停止的线程在很多业务场景下仍然很有价值。尤其是我们想写一个健壮性很好，能够安全应对各种场景的程序时，正确停止线程就显得格外重要。但是Java 并没有提供简单易用，能够直接安全停止线程的能力。

为什么不强制停止？而是通知、协作

对于 Java 而言，最正确的停止线程的方式是使用 interrupt。但 interrupt 仅仅起到通知被停止线程的作用。而对于被停止的线程而言，它拥有完全的自主权，它既可以选择立即停止，也可以选择一段时间后停止，也可以选择压根不停止。那么为什么 Java 不提供强制停止线程的能力呢？

事实上，Java 希望程序间能够相互通知、相互协作地管理线程，因为如果不了解对方正在做的工作，贸然强制停止线程就可能会造成一些安全的问题，为了避免造成问题就需要给对方一定的时间来整理收尾工作。比如：线程正在写入一个文件，这时收到终止信号，它就需要根据自身业务判断，是选择立即停止，还是将整个文件写入成功后停止，而如果选择立即停止就可能造成数据不完整，不管是中断命令发起者，还是接收者都不希望数据出现问题。

如何用 interrupt 停止线程

```
while (!Thread.currentThread().isInterrupted() && more work to do) {
    do more work
}
```

明白 Java 停止线程的设计原则之后，我们看看如何用代码实现停止线程的逻辑。我们一旦调用某个线程的 interrupt() 之后，这个线程的中断标记位就会被设置成 true。每个线程都有这样的标记位，当线程执行时，应该定期检查这个标记位，如果标记位被设置成 true，就说明有程序想终止该线程。回到源码，可以看到在 while 循环体判断语句中，首先通过 Thread.currentThread().isInterrupt() 判断线程是否被中断，随后检查是否还有工作要做。&& 逻辑表示只有当两个判断条件同时满足的情况下，才会去执行下面的工作。

我们再看看具体例子。

```
public class StopThread implements Runnable {
    @Override
    public void run() {
        int count = 0;
        while (!Thread.currentThread().isInterrupted() && count < 1000) {
            System.out.println("count = " + count++);
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new StopThread());
        thread.start();
        Thread.sleep(5);
        thread.interrupt();
    }
}
```

在 StopThread 类的 run() 方法中，首先判断线程是否被中断，然后判断 count 值是否小于 1000。这个线程的工作内容很简单，就是打印 0~999 的数字，每打印一个数字 count 值加

1，可以看到，线程会在每次循环开始之前，检查是否被中断了。接下来在 main 函数中会启动该线程，然后休眠 5 毫秒后立刻中断线程，该线程会检测到中断信号，于是在还没打印完1000个数的时候就会停下来，这种就属于通过 interrupt 正确停止线程的情况。

sleep 期间能否感受到中断

```
Runnable runnable = () -> {
    int num = 0;
    try {
        while (!Thread.currentThread().isInterrupted() &&
        num <= 1000) {
            System.out.println(num);
            num++;
            Thread.sleep(1000000);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};
```

那么我们考虑一种特殊情况，改写上面的代码，如果线程在执行任务期间有休眠需求，也就是每打印一个数字，就进入一次 sleep ， 而此时将 Thread.sleep() 的休眠时间设置为 1000 秒钟。

```
public class StopDuringSleep {
    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = () -> {
            int num = 0;
            try {
                while (!Thread.currentThread().isInterrupted() && num <= 1000) {
                    System.out.println(num);
```

```
                    num++;
                    Thread.sleep(1000000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();
        Thread.sleep(5);
        thread.interrupt();
    }
}
```

主线程休眠 5 毫秒后，通知子线程中断，此时子线程仍在执行 sleep 语句，处于休眠中。那么就需要考虑一点，在休眠中的线程是否能够感受到中断通知呢？是否需要等到休眠结束后才能中断线程呢？如果是这样，就会带来严重的问题，因为响应中断太不及时了。正因为如此，Java 设计者在设计之初就考虑到了这一点。

如果 sleep、wait 等可以让线程进入阻塞的方法使线程休眠了，而处于休眠中的线程被中断，那么线程是可以感受到中断信号的，并且会抛出一个 InterruptedException 异常，同时清除中断信号，将中断标记位设置成 false。这样一来就不用担心长时间休眠中线程感受不到中断了，因为即便线程还在休眠，仍然能够响应中断通知，并抛出异常。

两种最佳处理方式

在实际开发中肯定是团队协作的，不同的人负责编写不同的方法，然后相互调用来实现整个业务的逻辑。那么如果我们负责编写的方法需要被别人调用，同时我们的方法内调用了 sleep 或者 wait 等能响应中断的方法时，仅仅 catch 住异常是不够的。

```
void subTas() {
    try {
        Thread.sleep(1000);
```

```
    } catch (InterruptedException e) {
        // 在这里不处理该异常是非常不好的
    }
}
```

我们可以在方法中使用 try/catch 或在方法签名中声明 throws InterruptedException。

方法签名抛异常，run() 强制 try/catch

我们先来看下 try/catch 的处理逻辑。如上面的代码所示，catch 语句块里代码是空的，它并没有进行任何处理。假设线程执行到这个方法，并且正在 sleep，此时有线程发送 interrupt 通知试图中断线程，就会立即抛出异常，并清除中断信号。抛出的异常被 catch 语句块捕捉。

但是，捕捉到异常的 catch 没有进行任何处理逻辑，相当于把中断信号给隐藏了，这样做是非常不合理的，那么究竟应该怎么处理呢？首先，可以选择在方法签名中抛出异常。

```
void subTask2() throws InterruptedException {
    Thread.sleep(1000);
}
```

正如代码所示，要求每一个方法的调用方有义务去处理异常。调用方要不使用 try/catch 并在 catch 中正确处理异常，要不将异常声明到方法签名中。如果每层逻辑都遵守规范，便可以将中断信号层层传递到顶层，最终让 run() 方法可以捕获到异常。而对于 run() 方法而言，它本身没有抛出 checkedException 的能力，只能通过 try/catch 来处理异常。层层传递异常的逻辑保障了异常不会被遗漏，而对 run() 方法而言，就可以根据不同的业务逻辑来进行相应的处理。

再次中断

```
private void reInterrupt() {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
```

```
            e.printStackTrace();
        }
    }
```

除了刚才推荐的将异常声明到方法签名中的方式外，还可以在 catch 语句中再次中断线程。如代码所示，需要在 catch 语句块中调用 Thread.currentThread().interrupt() 函数。因为如果线程在休眠期间被中断，那么会自动清除中断信号。如果这时手动添加中断信号，中断信号依然可以被捕捉到。这样后续执行的方法依然可以检测到这里发生过中断，可以做出相应的处理，整个线程可以正常退出。

我们需要注意，我们在实际开发中不能盲目吞掉中断，如果不在方法签名中声明，也不在 catch 语句块中再次恢复中断，而是在 catch 中不作处理，我们称这种行为是“屏蔽了中断请求”。如果我们盲目地屏蔽了中断请求，会导致中断信号被完全忽略，最终导致线程无法正确停止。

为什么用 volatile 标记位的停止方法是错误的

下面我们来看一看本课时的第二个问题，为什么用 volatile 标记位的停止方法是错误的?

错误的停止方法

首先，我们来看几种停止线程的错误方法。比如 stop()，suspend() 和 resume()，这些方法已经被 Java 直接标记为 @Deprecated。如果再调用这些方法，IDE 会友好地提示，我们不应该再使用它们了。但为什么它们不能使用了呢？是因为 stop() 会直接把线程停止，这样就没有给线程足够的时间来处理想要在停止前保存数据的逻辑，任务戛然而止，会导致出现数据完整性等问题。

而对于 suspend() 和 resume() 而言，它们的问题在于如果线程调用 suspend()，它并不会释放锁，就开始进入休眠，但此时有可能仍持有锁，这样就容易导致死锁问题，因为这把锁在线程被 resume() 之前，是不会被释放的。

假设线程 A 调用了 suspend() 方法让线程 B 挂起，线程 B 进入休眠，而线程 B 又刚好持有一把锁，此时假设线程 A 想访问线程 B 持有的锁，但由于线程 B 并没有释放锁就进入休眠了，所以对于线程 A 而言，此时拿不到锁，也会陷入阻塞，那么线程 A 和线程 B 就都无法继续向下执行。

正是因为有这样的风险，所以 suspend() 和 resume() 组合使用的方法也被废弃了。那么接下来我们来看看，为什么用 volatile 标记位的停止方法也是错误的?

volatile 修饰标记位适用的场景

```
public class VolatileCanStop implements Runnable {
    private volatile boolean canceled = false;
    @Override
    public void run() {
        int num = 0;
        try {
            while (!canceled && num <= 1000000) {
                if (num % 10 == 0) {
                    System.out.println(num + "是10的倍数。");
                }
                num++;
                Thread.sleep(1);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) throws InterruptedException {
        VolatileCanStop r = new VolatileCanStop();
        Thread thread = new Thread(r);
        thread.start();
        Thread.sleep(3000);
        r.canceled = true;
    }
}
```

什么场景下 volatile 修饰标记位可以让线程正常停止呢？如代码所示，声明了一个叫作

VolatileStopThread 的类， 它实现了 Runnable 接口，然后在 run() 中进行 while 循环，在循环体中又进行了两层判断，首先判断 canceled 变量的值，canceled 变量是一个被 volatile 修饰的初始值为 false 的布尔值，当该值变为 true 时，while 跳出循环，while 的第二个判断条件是 num 值小于1000000（一百万），在while 循环体里，只要是 10 的倍数就打印出来，然后 num++。

接下来，首先启动线程，然后经过 3 秒钟的时间，把用 volatile 修饰的布尔值的标记位设置成 true，这样，正在运行的线程就会在下一次 while 循环中判断出 canceled 的值已经变成 true 了，这样就不再满足 while 的判断条件，跳出整个 while 循环，线程就停止了，这种情况是演示 volatile 修饰的标记位可以正常工作的情况，但是如果我们说某个方法是正确的，那么它应该不仅仅是在一种情况下适用，而在其他情况下也应该是适用的。

volatile 修饰标记位不适用的场景

接下来我们就用一个生产者/消费者模式的案例来演示为什么说 volatile 标记位的停止方法是不完美的。

```
class Producer implements Runnable {
    public volatile boolean canceled = false;
    BlockingQueue storage;
    public Producer(BlockingQueue storage) {
        this.storage = storage;
    }
    @Override
    public void run() {
        int num = 0;
        try {
            while (num <= 100000 && !canceled) {
                if (num % 50 == 0) {
                    storage.put(num);
                    System.out.println(num + "是50的倍数,被放到仓库中了。");
                }
                num++;
```

```
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("生产者结束运行");
        }
    }
}
```

首先，声明了一个生产者 Producer，通过 volatile 标记的初始值为 false 的布尔值 canceled 来停止线程。而在 run() 方法中，while 的判断语句是 num 是否小于 100000 及 canceled 是否被标记。while 循环体中判断 num 如果是 50 的倍数就放到 storage 仓库中，storage 是生产者与消费者之间进行通信的存储器，当 num 大于 100000 或被通知停止时，会跳出 while 循环并执行 finally 语句块，告诉大家“生产者结束运行”。

```
class Consumer {
    BlockingQueue storage;
    public Consumer(BlockingQueue storage) {
        this.storage = storage;
    }
    public boolean needMoreNums() {
        if (Math.random() > 0.97) {
            return false;
        }
        return true;
    }
}
```

而对于消费者 Consumer，它与生产者共用同一个仓库 storage，并且在方法内通过 needMoreNums() 方法判断是否需要继续使用更多的数字，刚才生产者生产了一些 50 的倍数供消费者使用，消费者是否继续使用数字的判断条件是产生一个随机数并与 0.97 进行比较，大于 0.97 就不再继续使用数字。

```
public static void main(String[] args) throws InterruptedException {
        ArrayBlockingQueue storage = new ArrayBlockingQueue(8);
        Producer producer = new Producer(storage);
        Thread producerThread = new Thread(producer);
        producerThread.start();
        Thread.sleep(500);
        Consumer consumer = new Consumer(storage);
        while (consumer.needMoreNums()) {
            System.out.println(consumer.storage.take() + "被消费了");
            Thread.sleep(100);
        }
        System.out.println("消费者不需要更多数据了。");
        //一旦消费不需要更多数据了，我们应该让生产者也停下来，但是实际情况却停不下来
        producer.canceled = true;
        System.out.println(producer.canceled);
    }
}
```

下面来看下 main 函数，首先创建了生产者/消费者共用的仓库 BlockingQueue storage，仓库容量是 8，并且建立生产者并将生产者放入线程后启动线程，启动后进行 500 毫秒的休眠，休眠时间保障生产者有足够的时间把仓库塞满，而仓库达到容量后就不会再继续往里塞，这时生产者会阻塞，500 毫秒后消费者也被创建出来，并判断是否需要使用更多的数字，然后每次消费后休眠 100 毫秒，这样的业务逻辑是有可能出现在实际生产中的。

当消费者不再需要数据，就会将 canceled 的标记位设置为 true，理论上此时生产者会跳出 while 循环，并打印输出“生产者运行结束”。

然而结果却不是我们想象的那样，尽管已经把 canceled 设置成 true，但生产者仍然没有停止，这是因为在这种情况下，生产者在执行 storage.put(num) 时发生阻塞，在它被叫醒之前是没有办法进入下一次循环判断 canceled 的值的，所以在这种情况下用 volatile 是没有办法让生产者停下来的，相反如果用 interrupt 语句来中断，即使生产者处于阻塞状态，仍然能够感受到中断信号，并做响应处理。

总结

好了，本课时的内容就全部讲完了，我们来总结下学到了什么，首先学习了如何正确停止线程，其次是掌握了为什么说 volatile 修饰标记位停止方法是错误的。

如果我们在面试中被问到“你知不知道如何正确停止线程”这样的问题，我想你一定可以完美地回答了，首先，从原理上讲应该用 interrupt 来请求中断，而不是强制停止，因为这样可以避免数据错乱，也可以让线程有时间结束收尾工作。

如果我们是子方法的编写者，遇到了 interruptedException，应该如何处理呢？

我们可以把异常声明在方法中，以便顶层方法可以感知捕获到异常，或者也可以在 catch 中再次声明中断，这样下次循环也可以感知中断，所以要想正确停止线程就要求我们停止方，被停止方，子方法的编写者相互配合，大家都按照一定的规范来编写代码，就可以正确地停止线程了。

最后我们再来看下有哪些方法是不够好的，比如说已经被舍弃的 stop()、suspend() 和 resume()，它们由于有很大的安全风险比如死锁风险而被舍弃，而 volatile 这种方法在某些特殊的情况下，比如线程被长时间阻塞的情况，就无法及时感受中断，所以 volatile 是不够全面的停止线程的方法。

上一页 下一页