二

04 waitnotifynotifyAll 方法的使用注意事项?

本课时我们主要学习 wait/notify/notifyAll 方法的使用注意事项。

我们主要从三个问题入手：

1. 为什么 wait 方法必须在 synchronized 保护的同步代码中使用?
2. 为什么 wait/notify/notifyAll 被定义在 Object 类中，而 sleep 定义在 Thread 类中?
3. wait/notify 和 sleep 方法的异同?

为什么 wait 必须在 synchronized 保护的同步代码中使用?

首先，我们来看第一个问题，为什么 wait 方法必须在 synchronized 保护的同步代码中使用?

我们先来看看 wait 方法的源码注释是怎么写的。

“wait method should always be used in a loop:

```
synchronized (obj) {
    while (condition does not hold)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

This method should only be called by a thread that is the owner of this object's monitor.”

英文部分的意思是说，在使用 wait 方法时，必须把 wait 方法写在 synchronized 保护的 while 代码块中，并始终判断执行条件是否满足，如果满足就往下继续执行，如果不满足就执行 wait 方法，而在执行 wait 方法之前，必须先持有对象的 monitor 锁，也就是通常所说的 synchronized 锁。那么设计成这样有什么好处呢?

我们逆向思考这个问题，如果不要求 wait 方法放在 synchronized 保护的同步代码中使用，而是可以随意调用，那么就有可能写出这样的代码。

```
class BlockingQueue {
    Queue<String> buffer = new LinkedList<String>();
    public void give(String data) {
        buffer.add(data);
        notify();  // Since someone may be waiting in take
    }
    public String take() throws InterruptedException {
        while (buffer.isEmpty()) {
            wait();
        }
        return buffer.remove();
    }
}
```

在代码中可以看到有两个方法，give 方法负责往 buffer 中添加数据，添加完之后执行 notify 方法来唤醒之前等待的线程，而 take 方法负责检查整个 buffer 是否为空，如果为空就进入等待，如果不为空就取出一个数据，这是典型的生产者消费者的思想。

但是这段代码并没有受 synchronized 保护，于是便有可能发生以下场景：

1. 首先，消费者线程调用 take 方法并判断 buffer.isEmpty 方法是否返回 true，若为 true 代表buffer是空的，则线程希望进入等待，但是在线程调用 wait 方法之前，就被调度器暂停了，所以此时还没来得及执行 wait 方法。
2. 此时生产者开始运行，执行了整个 give 方法，它往 buffer 中添加了数据，并执行了 notify 方法，但 notify 并没有任何效果，因为消费者线程的 wait 方法没来得及执行，所以没有线程在等待被唤醒。
3. 此时，刚才被调度器暂停的消费者线程回来继续执行 wait 方法并进入了等待。

虽然刚才消费者判断了 buffer.isEmpty 条件，但真正执行 wait 方法时，之前的 buffer.isEmpty 的结果已经过期了，不再符合最新的场景了，因为这里的“判断-执行”不是一个原子操作，它在中间被打断了，是线程不安全的。

假设这时没有更多的生产者进行生产，消费者便有可能陷入无穷无尽的等待，因为它错过了刚才 give 方法内的 notify 的唤醒。

我们看到正是因为 wait 方法所在的 take 方法没有被 synchronized 保护，所以它的 while 判断和 wait 方法无法构成原子操作，那么此时整个程序就很容易出错。

我们把代码改写成源码注释所要求的被 synchronized 保护的同步代码块的形式，代码如下。

```
public void give(String data) {
   synchronized (this) {
      buffer.add(data);
      notify();
  }
}
public String take() throws InterruptedException {
   synchronized (this) {
    while (buffer.isEmpty()) {
         wait();
       }
     return buffer.remove();
  }
}
```

这样就可以确保 notify 方法永远不会在 buffer.isEmpty 和 wait 方法之间被调用，提升了程序的安全性。

另外，wait 方法会释放 monitor 锁，这也要求我们必须首先进入到 synchronized 内持有这把锁。

这里还存在一个“虚假唤醒”（spurious wakeup）的问题，线程可能在既没有被 notify/notifyAll，也没有被中断或者超时的情况下被唤醒，这种唤醒是我们不希望看到的。虽然在实际生产中，虚假唤醒发生的概率很小，但是程序依然需要保证在发生虚假唤醒的时候的正确性，所以就需要采用while循环的结构。

```
while (condition does not hold)
    obj.wait();
```

这样即便被虚假唤醒了，也会再次检查while里面的条件，如果不满足条件，就会继续wait，也就消除了虚假唤醒的风险。

为什么 wait/notify/notifyAll 被定义在 Object 类中，而 sleep 定义在 Thread 类中?

我们来看第二个问题，为什么 wait/notify/notifyAll 方法被定义在 Object 类中？而 sleep 方法定义在 Thread 类中？主要有两点原因：

1. 因为 Java 中每个对象都有一把称之为 monitor 监视器的锁，由于每个对象都可以上锁，这就要求在对象头中有一个用来保存锁信息的位置。这个锁是对象级别的，而非线程级别的，wait/notify/notifyAll 也都是锁级别的操作，它们的锁属于对象，所以把它们定义在 Object 类中是最合适，因为 Object 类是所有对象的父类。
2. 因为如果把 wait/notify/notifyAll 方法定义在 Thread 类中，会带来很大的局限性，比如一个线程可能持有多把锁，以便实现相互配合的复杂逻辑，假设此时 wait 方法定义在 Thread 类中，如何实现让一个线程持有多把锁呢？又如何明确线程等待的是哪把锁呢？既然我们是让当前线程去等待某个对象的锁，自然应该通过操作对象来实现，而不是操作线程。

wait/notify 和 sleep 方法的异同?

第三个问题是对比 wait/notify 和 sleep 方法的异同，主要对比 wait 和 sleep 方法，我们先说相同点：

1. 它们都可以让线程阻塞。
2. 它们都可以响应 interrupt 中断：在等待的过程中如果收到中断信号，都可以进行响应，并抛出 InterruptedException 异常。

但是它们也有很多的不同点：

1. wait 方法必须在 synchronized 保护的代码中使用，而 sleep 方法并没有这个要求。
2. 在同步代码中执行 sleep 方法时，并不会释放 monitor 锁，但执行 wait 方法时会主动释放 monitor 锁。
3. sleep 方法中会要求必须定义一个时间，时间到期后会主动恢复，而对于没有参数的 wait 方法而言，意味着永久等待，直到被中断或被唤醒才能恢复，它并不会主动恢复。

4. wait/notify 是 Object 类的方法，而 sleep 是 Thread 类的方法。

以上就是关于 wait/notify 与 sleep 的异同点。

上一页 下一页