# 多线程与并发

## 线程

### wait/notify

wait()的作用是使当前执行代码的线程进行等待，将当前线程置入"预执行队列"中，并且wait()所在的代码处停止执行，直到接到通知或被中断。**在调用wait()之前，线程必须获得该对象的锁，因此只能在同步方法/同步代码块中调用wait()方法。**

notify()的作用是，如果有多个线程等待，那么线程规划器随机挑选出一个wait的线程，对其发出通知notify()，并使它等待获取该对象的对象锁。**注意"等待获取该对象的对象锁"，这意味着，即使收到了通知，wait的线程也不会马上获取对象锁，必须等待notify()方法的线程释放锁才可以。**和wait()一样，notify()也要在同步方法/同步代码块中调用。

总结起来就是，**wait()使线程停止运行，notify()使停止运行的线程继续运行。**

### wait和sleep区别

要深入了解这两个的区别，我们需要先了解线程的两个状态。一个是运行态，一个是冻结态。

1.什么是运行态呢？ 即同时具备cpu的执行资格和执行权。也就是说这个线程有权被cpu执行并且此时正在被cpu执行。

2.什么事冻结态？ 即同时释放了cpu的执行权和释放了cpu的执行资格，也就是说这个线程既不拥有被cpu的执行的权利，此时也不被cpu执行。

然后我们对执行资格和执行权进行深入的理解。当一个线程start的时候，就说明这个线程具备了执行资格，但是此时cpu不一定在执行这个线程，也就是说不一定拥有cpu的执行权。当线程拥有执行资格但不拥有执行权的时候，这种状态成为线程的临时阻塞状态。

稍微写多一些多线程程序和有基本理解的人现在应该都能理解上面几种概念了吧。

然后下面的cpu资源指的就是cpu的执行权和执行资格。

然后我们回到wait和sleep的区别上来。

很基本的区别：wait是object类的方法，sleep是thread类的方法。

1.wait释放cpu资源，同时释放锁。

2.sleep也释放cpu资源，但不释放锁。

先来简单科普一下wait和notify：我们在使用object类的wait,notify,notifyall的时候，我们都是要标识它们所属于的锁的，就是 锁对象.wait 这样的写法，这三个方法必须是在synchronized块里面使用，因为他们就是操作锁的，这里的锁也叫做监视器对象，离开了锁，操作这三个方法就没有意义。当出现synchronized块嵌套时，就必须要清楚地标明到底是哪个锁的wait和notify，不然就会引发混乱。当你直接写wait的时候，默认是调用this.wait.而synchronized默认的锁对象也是this，所以此时不会报错，但是当你synchronized使用的锁对象不是this时，这个时候就得好好注意啦，我的理解是一般都不要拿this当作锁（除非你方法都在一个类里面，而且结构比较简单），这个锁最好创建好以后放在一个接口里面，然后实现该接口然后调用就完事了。当你notifyall是也不会有线程安全问题，因为此时只有一个线程拥有锁对象。

此时我们再来理解一下wait的释放锁。也就是说在调用锁对象的wait方法时，此线程会释放cpu资源，释放锁，即线程不再运行，放到了线程池当中处于冻结状态。然后另一个线程可以获取这个锁，然后再调用这个锁的notify方法，把在线程池中处于冻结状态的对应线程唤醒。重新获取cpu的执行资格和执行权。

然后看sleep不释放锁是什么意思。

sleep就是把cpu的执行资格和执行权释放出去，不再运行此线程，当定时时间结束再取回cpu资源。而如果此时该线程有锁，那么sleep不会释放这个锁，而是把锁带着进入了冻结状态，也就是说其他需要这个锁的线程根本不可能获取到这个锁。也就是说无法执行程序。

### join

## 锁

### 锁和监视器区别

在JVM的规范中，有这么一些话：

“在JVM中，每个对象和类在逻辑上都是和一个监视器相关联的”

“为了实现监视器的排他性监视能力，JVM为每一个对象和类都关联一个锁”

“锁住了一个对象，就是获得对象相关联的监视器”

从这些话，看出监视器和对象锁好像是一回事，那为何要定义两个东西，若不一样，他们的关系如何？

### Monitor Object 模式

Monitor 其实是一种同步工具，或者说是同步机制，它通常被描述成一个对象，主要特点是：

同步。对象内的所有方法都互斥的执行。好比一个 Monitor 只有一个运行许可，任一个线程进入任何一个方法都需要获得这个许可，离开时把许可归还。

协作。通常提供signal机制：允许正持有许可的线程暂时放弃许可，等待某个监视条件成真，条件成立后，当前线程可以通知正在等待这个条件的线程，让它可以重新获得运行许可。