Java并行(2): Monitor

1. 什么是Monitor?

Monitor其实是一种同步工具，也可以说是一种同步机制，它通常被描述为一个对象，主要特点是:

- 对象的所有方法都被“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。
- 通常提供singal机制：允许正持有“许可”的线程暂时放弃“许可”，等待某个谓词成真（条件变量），而条件成立后，当前进程可以“通知”正在等待这个条件变量的线程，让他可以重新去获得运行许可。

Monitor对象可以被多线程安全地访问。关于“互斥”与“为什么要互斥”，我就不傻X兮兮解释了；而关于Monitor的singal机制，历史上曾经出现过两大门派，分别是Hoare派和Mesa派（上过海波老师OS课的SS同学应该对这个有印象），我还是用我的理解通俗地庸俗地解释一下:

- Hoare派的singal机制江湖又称“Blocking condition variable”，特点是，当“发通知”的线程发出通知后，立即失去许可，并“亲手”交给等待者，等待者运行完毕后再将许可交还通知者。在这种机制里，可以等待者拿到许可后，谓词肯定为真——也就是说等待者不必再次检查条件成立与否，所以对条件的判断可以使用“if”，不必“while”
- Mesa派的signal机制又称“Non-Blocking condition variable”， 与Hoare不同，通知者发出通知后，并不立即失去许可，而是把闻风前来等待者安排在ready queue里，等到schedule时有机会去拿到“许可”。这种机制里，等待者拿到许可后不能确定在这个时间差里是否有别的等待者进入过Monitor，因此不能保证谓词一定为真，所以对条件的判断必须使用“while”

这两种方案可以说各有利弊，但Mesa派在后来的盟主争夺中渐渐占了上风，被大多数实现所采用，有人给这种signal另外起了个别名叫“notify”，想必你也知道，Java采取的就是这个机制。

2. Monitor与Java不得不说的故事

子曰：“Java对象是天生的Monitor。”每一个Java对象都有成为Monitor的“潜质”。这是为什么？因为在Java的设计中，每一个对象自打娘胎里出来，就带了一把看不见的锁，通常我们叫“内部锁”，或者“Monitor锁”，或者“Intrinsic lock”。为了装逼起见，我们就叫它Intrinsic lock吧。有了这个锁的帮助，只要把类的所有对象方法都用synchronized关键字修饰，并且所有域都为私有（也就是只能通过方法访问对象状态），就是一个货真价实的Monitor了。比如，我们举一个大俗例吧：

```
public class Account {
        private int balance;

        public Account(int balance) {
                this.balance = balance;
        }

        synchronized public boolean withdraw(int amount){
                if(balance<amount)
                        return false;
                balance -= amount;
                return true;
        }

        synchronized public void deposit(int amount){
                balance +=amount;
        }

}
```

3. synchronized关键字

上面我们已经看到synchronized的一种用法，用来修饰方法，表示进入该方法需要对Intrinsic lock加锁，离开时放锁。synchronized可以用在程序块中，显示说明对“哪个对象的Intrinsic lock加锁”，比如

```
synchronized public void deposit(int amount){
        balance +=amount;
}

// 等价于
public void deposit(int amount){
        synchronized(this){
                balance +=amount;
```

```
        }
}
```

这时，你可能就要问了，你不是说任何对象都有intrinsic lock么？而synchronized关键字又可以显示指定去锁谁，那我们是不是可以这样做：

```
public class Account {
        private int balance;
        private Object lock = new Object();

        public Account(int balance) {
                this.balance = balance;
        }

        public boolean withdraw(int amount){
                synchronized (lock) {
                        if(balance<amount)
                                return false;
                        balance -= amount;
                        return true;
                }
        }

        public void deposit(int amount){
                synchronized (lock) {
                        balance +=amount;
                }
        }
}
```

不用this的内部锁，而是用另外任意一个对象的内部锁来完成完全相同的任务？没错，完全可以。不过，需要注意的是，这时候，你实际上禁止了“客户代码加锁”的行为。前几天BBS上简哥有一贴提到的bug其实就是这个，这个时候使用这份代码的客户程序如果想当然地认为Account的同步是基于其内部锁的，并且傻X兮兮地写了类似下面的代码：

```
        public static void main(String[] args) {
                Account account =new Account(1000);

                //some threads modifying account through Account's methods.

                synchronized (account) {
                        ;//blabla
```

```
                }
        }
```

自认为后面的同步快对account加了锁，期间的操作不会被其余通过Account方法操作account对象的线程所干扰，那就太悲剧了。因为他们并不相干，锁住了不同的锁。

4. Java中的条件变量

正如我们前面所说，Java采取了wait/notify机制来作为intrinsic lock 相关的条件变量，表示为等待某一条件成立的条件队列——说到这里顺带插一段，条件队列必然与某个锁相关，并且语义上关联某个谓词（条件队列、锁、条件谓词就是吉祥的一家）。所以，在使用wait/notify方法时，必然是已经获得相关锁了的，在进一步说，一个推论就是“wait/notify 方法只能出现在相应的同步块中”。如果不呢？就像下面一段（notify表示的谓词是“帐户里有钱啦～”）：

```
        public void deposit(int amount){
                balance +=amount;
                notify();
        }
```

```
//或者这样：
```

```
        public void deposit(int amount){
                synchronized (lock) {
                        balance +=amount;
                        notify();
                }
        }
```

这两段都是错的，第一段没有在同步块里，而第二段拿到的是lock的内部锁，调用的却是this.notify()，让人遗憾。运行时他们都会抛IllegalMonitorStateException异常——唉，想前一阵我参加一次笔试的时候，有一道题就是这个，让你选所给代码会抛什么异常，我当时就傻了，想这考得也太偏了吧，现在看看，确实是很基本的概念，当初被虐是压根没有理解wait/notify机制的缘故。那怎么写是对的呢？

```
        public void deposit(int amount){
```

```
                synchronized (lock) {
                        balance +=amount;
                        lock.notify();
                }
        }
//或者（取决于你采用的锁）：
        synchronized public void deposit(int amount){
                balance +=amount;
                notify();
        }
```

5.这就够了吗?

看上去，Java的内部锁和wait/notify机制已经可以满足任何同步需求了，不是吗？em…可以这么说，但也可以说，不那么完美。有两个问题：

- 锁不够用

有时候，我们的类里不止有一个状态，这些状态是相互独立的，如果只用同一个内部锁来维护他们全部，未免显得过于笨拙，会严重影响吞吐量。你马上会说，你刚才不是演示了用任意一个Object来做锁吗？我们多整几个Object分别加锁不就行了吗？没错，是可行的。但这样可能显得有些丑陋，而且Object来做锁本身就有语义不明确的缺点。

- 条件变量不够用

Java用wait/notify机制实际上默认给一个内部锁绑定了一个条件队列，但是，有时候，针对一个状态（锁），我们的程序需要两个或以上的条件队列，比如，刚才的Account例子，如果某个2B银行有这样的规定“一个账户存款不得多于10000元”，这个时候，我们的存钱需要满足“余额+要存的数目不大于10000，否则等待，直到满足这个限制”，取钱需要满足“余额足够，否则等待，直到有钱为止”，这里需要两个条件队列，一个等待“存款不溢出”，一个等待“存款足够”，这时，一个默认的条件队列够用么？你可能又说，够用，我们可以模仿network里的“多路复用”，一个队列就能当多个来使，像这样：

```
public class Account {
        public static final int BOUND = 10000;
        private int balance;
```

```
        public Account(int balance) {
                this.balance = balance;
        }

        synchronized public boolean withdraw(int amount) throws Interrupted
                        while(balance<amount)
                                wait();// no money, wait
                        balance -= amount;
                        notifyAll();// not full, notify
                        return true;
        }

        synchronized public void deposit(int amount) throws InterruptedExce
                        while(balance+amount >BOUND)
                                wait();//full, wait
                        balance +=amount;
                        notifyAll();// has money, notify
        }
}
```

不是挺好吗？恩，没错，是可以。但是，仍然存在性能上的缺陷：每次都有多个线程被唤醒，而实际只有一个会运行，频繁的上下文切换和锁请求是件很废的事情。我们能不能不要notifyAll，而每次只用notify（只唤醒一个）呢？不好意思，想要“多路复用”，就必须notifyAll，否则会有丢失信号之虞（不解释了）。只有满足下面两个条件，才能使用notify：

一，只有一个条件谓词与条件队列相关，每个线程从wait返回执行相同的逻辑。

二，一进一出：一个对条件变量的通知，语义上至多只激活一个线程。

我又想插播一段：刚才写上面那段代码，IDE提示抛InterruptedException，我想提一下，这是因为wait是一个阻塞方法，几乎所有阻塞方法都会声明可能抛InterruptedException，这是和Java的interrupt机制有关的，以后我们有机会再说。

既然这么做不优雅不高效不亚克西，那如之奈何？Java提供了其他工具吗？是的。这就是传说中的java.util.concurrency包里的故事，今天也不说了，有机会在和大家讨论。

主要参考资料：

1. Wiki

2. Addison Wesley, Java Concurrency in Practice ,Brian Goetz