链滴

并发与活跃性危险问题

作者：tianxin

原文链接：https://ld246.com/article/1488710788448

来源网站：链滴

并发与活跃性危险问题

最近几天重新整理了下Java并发编程实践这本书的知识点，在此把并发活跃性相关问题列举出来，对编程1~2年且有过多线程编程经验却没有相关的知识树的同学（统称）可以花点时间去读下这本经典作。

作者当时写这本书的时候应该正是JDK1.6 到JDK1.7过渡的阶段，所以关于JDK1.8并发的内容并未涉，但作者谈了他的猜想（本书的作者正是Java并发包的开发人员之一）

综述

活跃性常见问题：

- 死锁：线程A占有资源1，想访问资源2,线程B占有资源2，想访问资源1；二者互不谦让
- 活锁：二者总是互相谦让
- 饥饿：线程想访问某个资源（比如CPU时钟周期作为资源），可惜优先级过低，或者该资源被他资源占用（while无限循环），导致饥饿
- 信号丢失：生产者与消费者背景下（Object的wait与notify方法），wait方法还未调用，notify已被调用
- 糟糕的响应性

主要关注于如何死锁的问题上

死锁的产生

典型错误案例A

```
public class DeadSynchronizedTest {
    private Object resourceA = new Object();
    private Object resourceB = new Object();

    // 路径A
    public void routeA(){
        synchronized(resourceA) {
            synchronized(reourceB) {
                // 某些操作
            }
        }
    }

    // 路径B
    public void routeB() {
        synchronized(resourceB) {
            synchronized(resourceA) {
                // 某些操作
            }
        }
    }
}
```

当两个线程分别执行A路径与B路径时，便会产生死锁的可能

典型错误案例B

```
public class DeadSynchronizedDemo {

    public void deadSynchronized(Object resourceA, Object resourceB) {
        synchronized(resourceA) {
            synchronized(resourceB) {
                // 某些操作
            }
        }
    }
}
```

上述deadSynchronized方法并未直接体现隐含的死锁情况，可是当我们这样调用该方法时：

```
Object resourceA = new Object();
Object resourceB = new Object();
// 线程A
deadSynchronized(resourceA, resourceB);
// 线程B
deadSynchronized(resourceB, resourceA);
```

同样的死锁问题便产生（我们永远无法保证用户传递参数一定会符合我们API接口想要的顺序，所以避免这种发生的可能性）

经典错误案例C

在ResourceA和ResourceB中确实每个方法只锁住了自己的对象，但是依然可能造成死锁问题 —— 为ResourceA对ResourceB进行了访问

```
class ResourceA {
    private ResourceB resrouceB = new ResourceB();
    public synchronized void deadSynchronizedA() {
        // 某些操作

        resourceB.deadSynchronizedB();
    }
}

class ResourceB {
    public synchronized void deadSynchronizedB() {
        // 某些操作
    }
}
```

改进点：缩小锁的范围，只锁自己的方法

```
class ResourceA {
    private ResourceB resrouceB = new ResourceB();
    public void deadSynchronizedA() {
        synchronized(this) {
```

```
            // 某些操作
        }
        resourceB.deadSynchronizedB();
    }
}

class ResourceB {
    public void deadSynchronizedB() {
        synchronized (this) {
            // 某些操作
        }
    }
}
```

总结

以上三个例子其实本质上都是双方互相握住了对方想要的资源，只是我们在编码的过程当中有些死锁例子可能不是那么明显，所以需要多加小心。

同样，最后的改进方案并不是唯一的答案。

死锁的避免

将加锁的顺序统一起来 （避免单行道却出现两辆车相向而行的情况）

```
public class DeadSynchronizedTest {
    private Object resourceA = new Object();
    private Object resourceB = new Object();

    // 路径A
    public void routeA(){
        synchronized(resourceA) {
            synchronized(reourceB) {
                // 某些操作
            }
        }
    }

    // 路径B
    public void routeB() {
        synchronized(resourceA) {
            synchronized(resourceB) {
                // 某些操作
            }
        }
    }
}
```

然而，如果项目中的锁比较多，那么顺序性就很难满足，而且也很难设计了

定时的锁

1. 思路介绍

前提背景：有两个锁，分别设为代号1和2.

故事主线：线程A在已经获取锁1，且在等待获取锁2。线程B已经获取锁2，且在等待获取锁1。

解决方案：

线程A获取锁2超出某个时间后，便主动释放当前掌握的锁1，随后再获取锁1，再尝试获取锁2。

应用场景：只有两个锁的场景下可以使用。

2. 额外补充一点

synchronized关键字代表的内置锁不支持中断、时间设定等功能。所以我们需要使用JDK1.5之后（入了Lock接口 —— 显式锁）实现了Lock接口的类（比如ReentrantLock）

死锁诊断 —— 通过线程转储信息分析死锁

线程转储

初步看到这个名词是一脸懵逼的

- 线程的栈的调用信息
- 线程拥有哪些锁
- 线程在哪个 栈帧获取这些锁
- 线程被阻塞时在等待哪一个锁

JVM 通过等待关系图（什么是等待关系图？）进行搜索循环找出死锁，并得到是哪个锁以及是哪个线程

个人对等待关系图的理解（并未看过具体实现，所以只是猜测）

等待关系图：暂时不清楚实现原理，目前个人理解是锁资源与线程可以构成一幅有向图，然后利用深遍历去寻找这个图是否存在环，如果存在环，那么就有死锁，环上的节点（包括锁资源节点与线程节）就是发生死锁的相关线程和具体锁

Lock显式锁与synchronized内置锁

注意JDK1.5之前是不支持Lock显式锁 的 以上信息。JDK6增加了对Lock显式锁转储信息的支持，但法精确到栈帧相关

其他活跃性危险

饥饿

可能造成原因

1. 线程想得到资源，却得不到（资源可能被占用）
2. 线程优先级过低，得不到CPU资源

但是Java线程优先级不能作为一个标准：优先级高则一定比低优先级占用CPU时间更长。（可能会将同优先级映射到OS下的同一个优先级上）

活锁

不再讨论活锁的问题

信号丢失

最后再讨论一下信号丢失的例子

错误示范

在这里省略了其他代码，只看等待与唤醒两个操作

```
public class SignalMissDemo {

    public synchroinzed void entryWaitState() {
        wait();
        // 等待被唤醒，然后做某些操作
    }

    public synchronized void notfiyObject() {
        notifyAll();
    }
}
```

信号的丢失：关键在于对象还未进入wait状态，notifyAll方法就已经被调用，所以会错失这个信号

改进

```
public class SignalMissDemo {
    /**
     * 用signal来记录是否有信号量
     */
    private volatile boolean signal = false;

    public synchroinzed void entryWaitState() {
        while(!signal){
            wait();
        }
        // 等待被唤醒，然后做某些操作
    }

    public synchronized void notfiyObject() {
        notifyAll();
        signal= true;
    }
}
```

最后送给作为学生党的自己

关于死锁方面，任何一本操作系统上经典书籍一定会讲

- 产生死锁的四个必要条件

- 死锁的检测与恢复
  - 单类资源：有向图的环检测
  - 多类资源：向量判断法
  - 以及各种不靠谱恢复法
- 死锁的避免与银行家算法
  - 资源轨迹图（其实就是访问顺序不一致问题）
  - 单个银行家算法
- 死锁的预防（杜绝那个四个必要条件中的任何一个即可）
  - 资源可以被多个进程共有
  - 资源可被抢占（资源是分为抢占式资源——内存和不可抢占式资源——锁）
  - 线程一次性获得所有资源，而不是一个一个获取
  - 我们上述讲到的讲各个线程对资源的请求顺序一致