第二章 线程（中）

2.1 线程同步的思想

1.多线程的同步控制

有时候线程之间彼此不独立，需要同步：

（1）线程间的互斥：同时运行的几个线程需要共享一个（些）数据；共享的数据，在某一时刻只允许一个线程对其进行操作。

如“生产者/消费者”问题：假设有一个线程负责往数据区写数据，另一个线程从同一数据区中读数据，两个线程可以并行执行，如果数据区已经满了，生产者需要消费者取走一些数据后才能再写入；当数据区空时，消费者要等生产者写入一些数据后再读取。

例：用两个线程模拟存票和售票过程：

假定开始售票处并没有票，一个线程往里存票，另一个往外买票。我们新建一个票类对象，让存票和售票线程都访问它。本例采用两个线程共享同一数据对象来实现对同一份数据的操作。

代码：

```
public class ProducerAndConsumer {

    public static void main(String[] args) {
        Tickets t = new Tickets(20);// 建立票对象，票总数20
        //new Consumer(t).start();// 开始卖票
        //new Producer(t).start();// 开始存票
        new Producer(t).start();// 开始存票
        new Consumer(t).start();// 开始卖票
    }
}

class Tickets {
    int number = 0;// 票号
    int size;// 票总数
    boolean ava = false;// 是否有票可售

    public Tickets(int size) {
        // 构造方法，传入总票数参数
        this.size = size;
    }
}
//存票线程
class Producer extends Thread {
    Tickets t = null;
```

```
    public Producer(Tickets t) {
        this.t = t;
    }

    public void run() {
        while (t.number < t.size) {
            System.out.println("Producer puts ticket " +
(++t.number));
            t.ava = true;
        }
    }
}
//售票线程
class Consumer extends Thread{
    Tickets t = null;
    int i =0;
    public Consumer(Tickets t) {
        this.t =t;
    }
    public void run() {
        while(i<t.size)
        {
            if(t.ava == true && i<t.number)
                System.out.println("Consumer buys ticket " + (++i));
            if(i == t.number)
                t.ava = false;
        }
    }

}
```

结果：

```
Producer puts ticket 1
Producer puts ticket 2
Producer puts ticket 3
……
Consumer buys ticket 1
Consumer buys ticket 2
Consumer buys ticket 3
```

修改示例：

将 Consumer 里面代码一部分代码修改为：

```
            if(i == t.number)
                try {
                    Thread.sleep(1);
                }catch(InterruptedException exception) {}
```

```
                t.ava = false;
```

则会出现错误。原因是票线程运行到 t.ava=false 之前，休眠 1ms：导致 CPU 切换到存票线程，将 ava 设置为 true，直到整个存票线程结束，这个时候售票又醒过来了，执行 t.ava=false，此时售票号小于村票数，且存票线程已经结束，不能将 t.ava 设置为 true，则售票线程陷入了死循环。

2.2 线程同步的实现方式——Synchronization

1.线程同步

（1）互斥：许多线程在同一个共享数据上操作而互不干扰，同一时刻只能有一个线程访问该共享数据。因此有些方法或者程序段在同一时刻只能被一个线程执行，称之为监视区。

（2）协作：多个线程可以有条件的同时操作共享数据。执行监视去代码的线程在条件满足的情况下可以允许其它线程进入监视区。

2.synchronized——线程同步关键字，实现互斥

用于指定需要同步的代码段或者方法，也就是监视区。可实现与一个锁的交互，例如：synchronized(对象){代码段}

功能是首先判断对象的锁是否存在，如果存在就获得锁，然后执行后面的代码段；如果锁不存在（已经被其它线程拿走），就进入等待状态，直到获得锁。

当被 synchronized 限定的代码执行完，就释放锁。

Java 使用监视器机制：每个对象只有一个“锁”，利用多线程对“锁”的争夺实现线程间的互斥。当线程 A 获得一个对象的锁之后，线程 B 必须等待线程 A 完成规定的操作、并释放出锁后，才能获得该对象的锁，并执行线程 B 中的操作。

例：用 synchronized 关键字解决上例问题——设置为互斥关系

存票线程和售票线程应保持互斥关系，即售票线程执行时不进入存票线程，存票线程执行时不进入售票线程。

代码：

Producer 类中 run()方法修改为：

```
    public void run() {
        while (t.number < t.size) {
            synchronized (t) {
                // 申请对象t的锁
                System.out.println("Producer puts ticket " +
(++t.number));
                t.ava = true;
            }//释放对象t的锁

        }
        System.out.println("Producer ends!");
    }
```

Consumer 类中的 run()方法修改为：

```
    public void run() {
        while (i <t.size) {
            synchronized (t) {
                // 申请对象t的锁
```

```
                if (t.ava == true && i < t.number)
                    System.out.println("Consumer buys ticket " +
(++i));
                if (i == t.number) {
                    try {
                        Thread.sleep(1);
                    } catch (Exception e) {
                    }
                    t.ava = false;
                }
            } // 释放对象t的锁
        }
    }
```

实际上上面的这些代码，synchronized 都将其后面大括号里面的代码编程了原子操作，即里面的内容不会被别的代码打破运行顺序。

运行结果：

Producer puts ticket 1
Producer puts ticket 2
Producer puts ticket 3
……
Producer puts ticket 18
Producer puts ticket 19
Producer puts ticket 20
Producer ends!
Consumer buys ticket 1
Consumer buys ticket 2
Consumer buys ticket 3
……
Consumer buys ticket 18
Consumer buys ticket 19
Consumer buys ticket 20

例：改进上例功能，将互斥方法放在共享的资源类 Tickets 中

代码：

```
    public synchronized void put() {
        // 同步方法，实现存票功能
        System.out.println("Producer puts ticket " + (++number));
        ava = true;
    }

    public synchronized void sell() {
        //同步方法，实现售票功能
        if(ava = true && i<=number)
            System.out.println("Consumer buys ticket " + (++i));
        if(i == number)
```

```
                ava = false;
        }
Producer 类修改：
        public void run() {
                while (t.number < t.size) {
                        t.put();
                }
        }
Consumer 类修改：
public void run() {
                while (i < t.size) {
                        t.sell();
                }
        }
```

同步与锁的要点：

（1）只能同步方法，不能同步变量；

（2）每个对象只有一个锁，当提到同步，应该清楚在什么上同步；

（3）类可以同时拥有同步和非同步方法，非同步方法可以被多个线程自由访问而不受锁的限制；

（4）如果两个线程使用相同的实例来调用的 synchronized 方法，那么一次只能有一个线程执行方法，另一个需要等待锁；

（5）线程睡眠时，它所持有的任何锁都不会被释放；

（6）线程可以获得多个锁。比如说，一个对象的同步方法里面调用另外一个对象的同步方法，则获取了两个对象的同步锁；

（7）同步损害并发性，应该尽可能缩小同步范围。同步不但可以同步整个方法，还可以同步方法中一部分代码块；

（8）在使用同步代码块，应该制定在那个对象上同步，也就是说要获取哪个对象的锁。

2.3 线程的等待与唤醒

1.线程的等待

为了更有效地协调不同线程的工作，需要在线程间建立沟通渠道，通过线程间的“对话”来解决线程间的同步问题。

在 java.lang.Object 类的一些方法为线程间的通信提供了有效手段，例如：

（1）wait()方法：如果当前状态不适合本线程执行，正在执行同步代码的某个线程 A 调用该方法（在对象 x 上），该线程暂停执行而进入对象 x 的等待吃，并释放已获得的对象 x 的锁。线程 A 要一直等到其它线程在对象 x 上调用 notify 或 notifyAll 方法，才能够在重新获得对象 x 的锁后继续执行（从 wait 语句后继续执行）。

2.线程的唤醒——notify()和 notifyAll()方法

notify()随机唤醒一个等待的线程，本线程继续执行。线程被唤醒以后，还要等发出唤醒消息者释放监视器，这期间关键数据仍可能被改变。被唤醒的线程开始执行时，一定要判断当前状态是否适合自己运行。

notifyAll()唤醒所有等待的线程，本线程继续执行。

例：修改上例，要求，每存入一张票就售出一张，售出后再存入

代码：

修改 Tickets 类中的 sell 和 put 为：

```
    public synchronized void put() {
        // 同步方法，实现存票功能
        if (ava)// 如果还有票，则存票线程等待
            try {
                wait();
            } catch (Exception e) {
            }
        System.out.println("Producer puts ticket " + (++number));
        ava = true;
        notify();// 存票后唤醒售票线程开始售票
    }

    public synchronized void sell() {
        // 同步方法，实现售票功能
        if (!ava)// 如果没有存票则等待
            try {
                wait();
            } catch (Exception e) {
            }
        System.out.println("Consumer buys ticket " + (number));
        ava = false;
        notify();// 售票后唤醒存票线程
        if (number == size)
            number = size + 1;
    }
}
```

结果：

```
Producer puts ticket 1
Consumer buys ticket 1
Producer puts ticket 2
Consumer buys ticket 2
……
Producer puts ticket 19
Consumer buys ticket 19
Producer puts ticket 20
Consumer buys ticket 20
```

程序说明：

当 consumer 线程售票后，ava 变为 false，当 producer 线程放入票后，ava 变为 true；

只有 ava 为 true 时，consumer 线程才能售票，否则就必须等待 producer 线程放入新票后的通知；

只有 ava 为 false 时，producer 线程才能放票，否则必须等待 consumer 线程售出票后的通知。

2.4 后台进程

后台线程也叫守护线程，通常是为了辅助其它线程而运行的线程，它不妨碍程序终止。一个进程中只要有一个前台线程在运行，这个进程就不会结束；但是如果一个进程中所有前台线程都已经结束，那么无论后台线程是否结束，这个进程都会结束。例如，垃圾回收这个后台线程。

如果对某个线程对象在启动（调用 start 方法）之前调用了 setDaemon(true)方法，这个线程就变成了后台线程。

例：创建一个无限循环的后台线程，验证主线程结束后，程序即结束。

代码：

```
public class Ex8_10 {

    public static void main(String[] args) {
        ThreadTest t = new ThreadTest();
        t.setDaemon(true);
        t.start();
    }
}
class ThreadTest extends Thread{
    public void run()
    {
        while(true)
        {

        }
    }
}
```

运行结果：

整个程序在主线程结束时就随之中止运行了，在不是后台线程情况下，程序陷入死循环。

2.5 线程的生命周期与死锁

1.线程的生命周期

线程的生命周期指的是线程从产生到消亡的过程。一个线程在任何时刻都处于某种线程状态（thread state）。

线程的声明周期状态图如图 1 所示，从右下角开始看。

图 1

线程的几种基本状态：

①诞生状态：线程刚刚被创建。

②就绪状态：线程的 start 方法已经被执行，线程已经准备好运行。

③运行状态：CPU 分配给了线程，线程正在运行。

④阻塞状态（Blocked）：在线程发出输入/输出请求且必须等待其返回；遇到用

synchronized 标记的方法而未获得锁；为等候一个条件变量，调用 wait()方法。

⑤休眠状态（sleeping）：执行 sleep 方法而进入休眠。

⑥死亡状态：线程已完成或退出。

2.死锁（Deadlock）

线程在运行过程中，其中某个步骤往往需要满足一些条件才能继续进行下去，如果这个条件不能满足，线程将在这个步骤上出现阻塞。多个线程一起等待的时候，就会出现很大的麻烦，比如说，线程 A 可能陷入对 B 的等待，B 陷入对 C 的等待，依次类推，整个等待链最后又可能回到线程 A。如此一来便陷入一个彼此等待的轮回中，任何线程都动弹不得，此为死锁。

对于死锁问题，关键不在于出现问题后的调试，而是在于预防。

3.结束线程的生命

通常，可通过控制 run 方法中循环条件的方式来结束一个线程，也可以用 stop 方法来结束线程的生命。

但是，如果一个线程正在操作共享数据段，操作过程没有完成就使用 stop 结束的话，将会导致数据的不完整，因此并不提倡使用此方法。

2.6 线程的调度

1.线程的优先级

在单 CPU 的系统中，多个线程需要共享 CPU，在任何时间点上实际只能有一个线程在运行。控制多个线程在同一个 CPU 上以某种顺序运行被称为线程调度。

Java 虚拟机支持一种非常简单的、确定的调度算法，叫做固定优先级算法。这个算法基于线程的优先级对其进行调度。

每个 Java 线程都有一个优先级，其范围在 1 到 10 之间，某人情况下，每个线程优先级都设置为 5。

在线程 A 运行过程中创建 B，初始状态具有和 A 相同的优先级。若 A 是一个后台线程，则 B 也是后台线程。我们可以在线程创建以后的任何时候通过 setPriority(int priority)方法改变其原来的优先级。

说明：

（1）具有较高优先级的线程比优先级较低的线程优先执行。

（2）对具有同优先级，处理是随机的。

（3）底层操作系统支持的优先级可能要少于 10 个，这样会造成一些混乱。因此，只能讲优先级作为一种很粗略的工具使用。最后的控制可以通过明智地使用 yield()函数（**把自己的线程暂停下来让给同优先级的线程，若不存在同优先级的，则继续执行自己**）来完成。

（4）我们只能基于效率的考虑来使用线程优先级，而不能依靠线程优先级来保证算法的正确性。

假设某线程正在运行，则只有出现以下情况之一才会使其暂停运行：

（1）一个具有更高优先级的线程变为就绪状态；

（2）由于输入/输出（或其他原因）、调用 sleep/wait/yield 方法使其发生阻塞；

（3）对于支持时间分片的系统，时间片的时间期满。