二

06 一共有哪 3 类线程安全问题?

本课时我们学习 3 类线程安全问题。

什么是线程安全

要想弄清楚有哪 3 类线程安全问题，首先需要了解什么是线程安全，线程安全经常在工作中被提到，比如：你的对象不是线程安全的，你的线程发生了安全错误，虽然线程安全经常被提到，但我们可能对线程安全并没有一个明确的定义。

《Java Concurrency In Practice》的作者 Brian Goetz 对线程安全是这样理解的，当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行问题，也不需要进行额外的同步，而调用这个对象的行为都可以获得正确的结果，那这个对象便是线程安全的。

事实上，Brian Goetz 想表达的意思是，如果某个对象是线程安全的，那么对于使用者而言，在使用时就不需要考虑方法间的协调问题，比如不需要考虑不能同时写入或读写不能并行的问题，也不需要考虑任何额外的同步问题，比如不需要额外自己加 synchronized 锁，那么它才是线程安全的，可以看出对线程安全的定义还是非常苛刻的。

而我们在实际开发中经常会遇到线程不安全的情况，那么一共有哪 3 种典型的线程安全问题呢?

1. 运行结果错误;
2. 发布和初始化导致线程安全问题;
3. 活跃性问题。

运行结果错误

首先，来看多线程同时操作一个变量导致的运行结果错误。

```
public class WrongResult {

   volatile static int i;
```

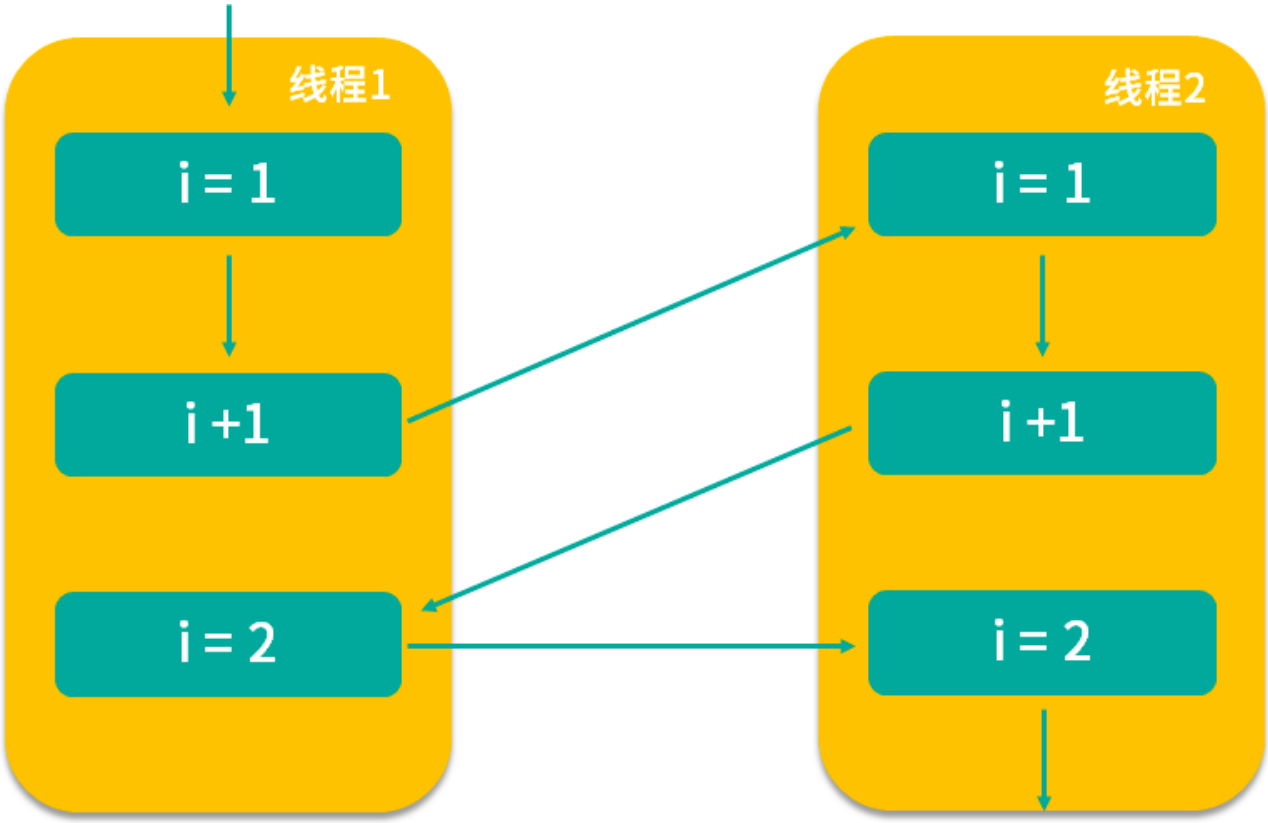
```
    public static void main(String[] args) throws InterruptedException {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                for (int j = 0; j < 10000; j++) {
                    i++;
                }
            }
        };
        Thread thread1 = new Thread(r);
        thread1.start();
        Thread thread2 = new Thread(r);
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(i);
    }
}
```

如代码所示，首先定义了一个 int 类型的静态变量 i，然后启动两个线程，分别对变量 i 进行 10000 次 i++ 操作。理论上得到的结果应该是 20000，但实际结果却远小于理论结果，比如可能是12996，也可能是13323，每次的结果都还不一样，这是为什么呢？

是因为在多线程下，CPU 的调度是以时间片为单位进行分配的，每个线程都可以得到一定量的时间片。但如果线程拥有的时间片耗尽，它将会被暂停执行并让出 CPU 资源给其他线程，这样就有可能发生线程安全问题。比如 i++ 操作，表面上看只是一行代码，但实际上它并不是一个原子操作，它的执行步骤主要分为三步，而且在每步操作之间都有可能被打断。

- 第一个步骤是读取；
- 第二个步骤是增加；
- 第三个步骤是保存。

那么我们接下来看如何发生的线程不安全问题。



我们根据箭头指向依次看，线程 1 首先拿到 i=1 的结果，然后进行 i+1 操作，但此时 i+1 的结果并没有保存下来，线程 1 就被切换走了，于是 CPU 开始执行线程 2，它所做的事情和线程 1 是一样的 i++ 操作，但此时我们想一下，它拿到的 i 是多少？实际上和线程 1 拿到的 i 的结果一样都是 1，为什么呢？因为线程 1 虽然对 i 进行了 +1 操作，但结果没有保存，所以线程 2 看不到修改后的结果。

然后假设等线程 2 对 i 进行 +1 操作后，又切换到线程 1，让线程 1 完成未完成的操作，即将 i+1 的结果 2 保存下来，然后又切换到线程 2 完成 i=2 的保存操作，虽然两个线程都执行了对 i 进行 +1 的操作，但结果却最终保存了 i=2 的结果，而不是我们期望的 i=3，这样就发生了线程安全问题，导致了数据结果错误，这也是最典型的线程安全问题。

发布和初始化导致线程安全问题

第二种是对象发布和初始化时导致的线程安全问题，我们创建对象并进行发布和初始化供其他类或对象使用是常见的操作，但如果我们操作的时间或地点不对，就可能导致线程安全问题。如代码所示。

```
public class WrongInit {

    private Map<Integer, String> students;

    public WrongInit() {
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                students = new HashMap<>();
                students.put(1, "王小美");
                students.put(2, "钱二宝");
                students.put(3, "周三");
                students.put(4, "赵四");
            }
        }).start();
     }
    public Map<Integer, String> getStudents() {
        return students;
    }
    public static void main(String[] args) throws InterruptedException {
        WrongInit multiThreadsError6 = new WrongInit();
        System.out.println(multiThreadsError6.getStudents().get(1));
    }
}
```

在类中，定义一个类型为 Map 的成员变量 students，Integer 是学号，String 是姓名。然后在构造函数中启动一个新线程，并在线程中为 students 赋值。

- 学号：1，姓名：王小美；
- 学号：2，姓名：钱二宝；
- 学号：3，姓名：周三；
- 学号：4，姓名：赵四。

只有当线程运行完 run() 方法中的全部赋值操作后，4 名同学的全部信息才算是初始化完毕，可是我们看在主函数 mian() 中，初始化 WrongInit 类之后并没有进行任何休息就直接打印 1 号同学的信息，试想这个时候程序会出现什么情况？实际上会发生空指针异常。

```
Exception in thread "main" java.lang.NullPointerException

at lesson6.WrongInit.main(WrongInit.java:32)
```

这又是为什么呢？因为 students 这个成员变量是在构造函数中新建的线程中进行的初始化和赋值操作，而线程的启动需要一定的时间，但是我们的 main 函数并没有进行等待就直接获取数据，导致 getStudents 获取的结果为 null，这就是在错误的时间或地点发布或初始化造成的线程安全问题。

活跃性问题

第三种线程安全问题统称为活跃性问题，最典型的有三种，分别为死锁、活锁和饥饿。

什么是活跃性问题呢，活跃性问题就是程序始终得不到运行的最终结果，相比于前面两种线程安全问题带来的数据错误或报错，活跃性问题带来的后果可能更严重，比如发生死锁会导致程序完全卡死，无法向下运行。

死锁

最常见的活跃性问题是死锁，死锁是指两个线程之间相互等待对方资源，但同时又互不相让，都想自己先执行，如代码所示。

```
public class MayDeadLock {
    Object o1 = new Object();
    Object o2 = new Object();
    public void thread1() throws InterruptedException {
        synchronized (o1) {
            Thread.sleep(500);
            synchronized (o2) {
                System.out.println("线程1成功拿到两把锁");
            }
        }
    }
    public void thread2() throws InterruptedException {
        synchronized (o2) {
```

```
            Thread.sleep(500);
            synchronized (o1) {
                System.out.println("线程2成功拿到两把锁");
            }
        }
    }
    public static void main(String[] args) {
        MayDeadLock mayDeadLock = new MayDeadLock();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    mayDeadLock.thread1();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    mayDeadLock.thread2();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
```

```
        }
    }
```

首先，代码中创建了两个 Object 作为 synchronized 锁的对象，线程 1 先获取 o1 锁，sleep(500) 之后，获取 o2 锁；线程 2 与线程 1 执行顺序相反，先获取 o2 锁，sleep(500) 之后，获取 o1 锁。 假设两个线程几乎同时进入休息，休息完后，线程 1 想获取 o2 锁，线程 2 想获取 o1 锁，这时便发生了死锁，两个线程不主动调和，也不主动退出，就这样死死地等待对方先释放资源，导致程序得不到任何结果也不能停止运行。

活锁

第二种活跃性问题是活锁，活锁与死锁非常相似，也是程序一直等不到结果，但对比于死锁，活锁是活的，什么意思呢？因为正在运行的线程并没有阻塞，它始终在运行中，却一直得不到结果。

举一个例子，假设有一个消息队列，队列里放着各种各样需要被处理的消息，而某个消息由于自身被写错了导致不能被正确处理，执行时会报错，可是队列的重试机制会重新把它放在队列头进行优先重试处理，但这个消息本身无论被执行多少次，都无法被正确处理，每次报错后又会被放到队列头进行重试，周而复始，最终导致线程一直处于忙碌状态，但程序始终得不到结果，便发生了活锁问题。

饥饿

第三个典型的活跃性问题是饥饿，饥饿是指线程需要某些资源时始终得不到，尤其是CPU资源，就会导致线程一直不能运行而产生的问题。在 Java 中有线程优先级的概念，Java 中优先级分为 1 到 10，1 最低，10 最高。如果我们把某个线程的优先级设置为 1，这是最低的优先级，在这种情况下，这个线程就有可能始终分配不到 CPU 资源，而导致长时间无法运行。或者是某个线程始终持有某个文件的锁，而其他线程想要修改文件就必须先获取锁，这样想要修改文件的线程就会陷入饥饿，长时间不能运行。

好了，今天的内容就全部讲完了，通过本课时的学习我们知道了线程安全问题主要有 3 种，i++ 等情况导致的运行结果错误，通常是因为并发读写导致的，第二种是对象没有在正确的时间、地点被发布或初始化，而第三种线程安全问题就是活跃性问题，包括死锁、活锁和饥饿。

上一页 下一页