14 僵持不下—死锁详解

更新时间：2019-10-15 10:17:58

> 天才就是这样，终身努力，便成天才。
>
> ——门捷列夫

前面几节讲解了并发的三大特性 - 原子性、可见性、有序性。解决这些问题的关键就是同步，而一种重要的同步方式就是加锁，所谓的加锁就是某个线程声明某个资源暂时由我独享，等其用完后，此线程解锁，也就是放弃对该资源的占有。如果有其它线程等待使用该资源，那么会再次对此资源加锁。所谓的死锁，其实就是因为某种原因，达不到解锁的条件，导致某线程对资源的占有无法释放，其他线程会一直等待其解锁，而被一直 block 住。

1. 死锁产生原因

产生死锁的原因很多，我们逐个来看一下：

交叉死锁

A 线程持有资源 R1 的锁时，想要获取 R2 的锁。而线程 B 此时持有 R2 的锁，想要获取 R1 的锁。结果就是两个线程互相等待对方释放，并且一直等待下去。这就像两个小孩打架搂抱在一起，A 说：你放手！B 说：你先放手！A 说：你先放手我就放手！B 说：凭什么我先放手，你先放手我就放手！瞧，是不是死锁了，最后估计还得继续打下去。

内存不足

某系统内存 20M，两个线程正在分别执行任务，各自已经使用了 10M 内存。但是执行到一半时需要更大的内存，但是系统已经没有内存可供使用。那么两个线程都会等待对方执行完毕 时释放内存。这就造成了两个线程互相等待，从而形成死锁。

一问一答式的数据交换

所谓的一问一答式数据交换就是客户端发送请求，服务端返回响应。如果在交互过程中出现了数据的丢失，双方产生误解，以为对方没有收到消息，陷入等待之中。如果此时没有设置 timeout，就会造成互相的等待一直持续下去，从而形成死锁。

数据库锁

如果某个线程对数据库表或者行加锁，但是意外导致没能正确释放锁，而其他线程则会等待数据库锁的释放，从而陷入死锁。

文件锁

某个线程获取文件锁后开始执行。但是执行过程中意外退出，而没能释放锁。那么其他等待该文件锁的线程将会一直等待，直到系统释放文件句柄的资源。

死循环

假如某个线程，由于编码问题，在对资源加锁后，陷入死循环，导致一致无法释放锁。

2 . 死锁举例

下面我们看一个交叉死锁的例子，来切身感受下死锁是如何炼成的。例子很简单，DeadLock 类有一个读方法和一个写方法，读方法获取读锁后，又尝试获取写锁。而写方法获取写锁后，又尝试获取读锁。这种情况下，两个线程会互相等待对方的锁释放，从而形成了死锁。我们看下面的代码：

```
public class DeadLock {
    private final String write_lock = new String();
    private final String read_lock = new String();

    public void read() {
        synchronized (read_lock) {
            System.out.println(Thread.currentThread().getName() + " got read lock and then i want to write");
            synchronized (write_lock) {
                System.out.println(Thread.currentThread().getName() + " got read lock and write lock");
            }
        }
    }

    public void write() {
        synchronized (write_lock) {
            System.out.println(Thread.currentThread().getName() + " got write lock and then i want to read");
            synchronized (read_lock) {
                System.out.println(Thread.currentThread().getName() + " got write lock and read lock");
            }
        }
    }

    public static void main(String[] args) {
        DeadLock deadLock = new DeadLock();
        new Thread(() -> {
            while (true) {
                deadLock.read();
            }
        },"read-first-thread").start();

        new Thread(() -> {
            while (true) {
                deadLock.write();
            }
        },"write-first-thread").start();
    }
}
```

注意 mian 方法中使用了 lambda 表达式为 thread 提供了 run 方法的现实。免去了我们编写两个实现 run 方法类的麻烦。这段程序运行后，控制台输出如下：

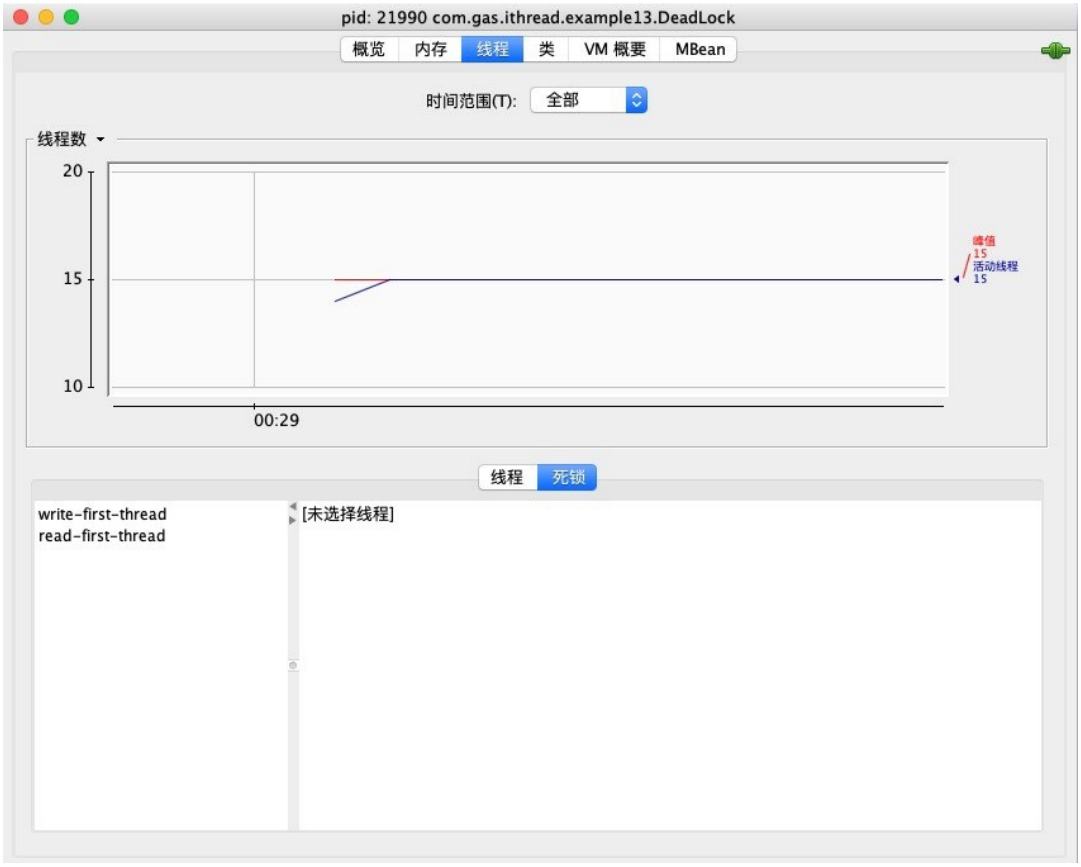
```
read-first-thread got read lock and then i want to write
read-first-thread got read lock and write lock
read-first-thread got read lock and then i want to write
read-first-thread got read lock and write lock
read-first-thread got read lock and then i want to write
read-first-thread got read lock and write lock
read-first-thread got read lock and then i want to write
write-first-thread got write lock and then i want to read
```

可以看到在 write 线程启动前，一切正常。read-first-thread 线程能够先后获得 read 锁和 write 锁。但是当 write 线程启动后，立刻出现了问题，日志不再打印，而是停留在 write 线程等待 read 锁这一步。这是因为已经死锁了。read 线程在等 write 线程释放写锁，而 write 线程在等 read 线程释放读锁。两个线程就会如此一直等下去了。

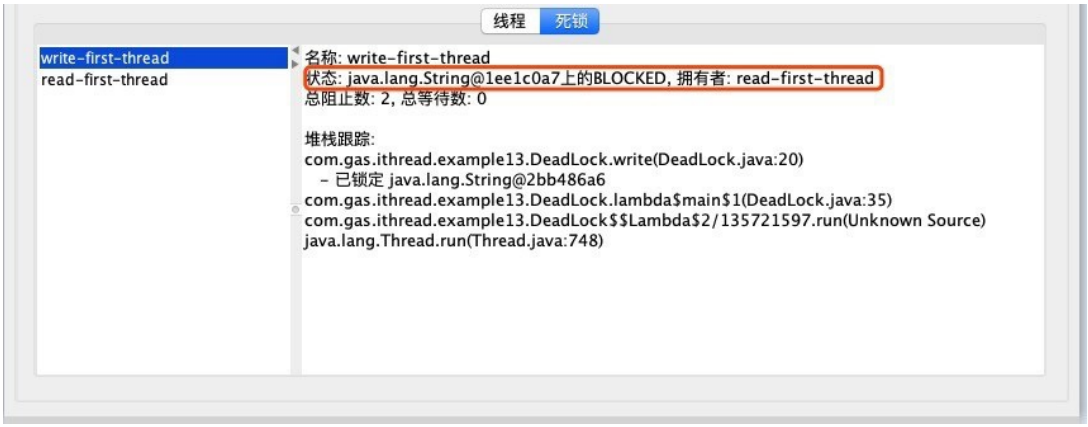
3. 死锁诊断

线程死锁可以通过 java 的监控工具来查看。此类工具很多，例如 jstack、jconsole、jprofile 等。下面我们看一下 Java 内置的 jconsole。如果你安装了 JDK，设置好了环境变量，那么可以直接在控制台输入 jconsole 来运行，界面如下：

在本地进程中我们可以看到刚才运行的 DeadLock。选中后点击连接。在下一个界面上面的菜单选择线程。在下方左侧框中可以看到 write 和 read 两个线程。然后我们点击检查死锁，显示如下图：



点击某个线程，如 write-first-thread，右侧框中出现此线程的状态，可以看到状态为 java.lang.String@1adb219c 上的 BLOCKED，拥有者：read-first-thread。意思是此线程在 java.lang.String@1adb219c 上被 block 住了，这个 String 对象其实就是 read_lock 对象，目前锁的拥有者是 read-first-thread。我们再查看另外一个线程 read-first-thread，可以看到正好是反过来的。这两个线程互相 block 住了。



4. 总结

本节中我们列举了多种引起死锁的原因，这对我们分析死锁的产生很有帮助，也有助于我们从代码层面找到可能导致死锁的风险。后面通过举例，更为形象的切身感受到死锁的产生过程。即使我们知道死锁产生的原理，但也还是很可能写出导致死锁的代码，那么出现死锁或者疑似死锁的时候应该怎么办呢？最后也给出了答案。

至此本章已经结束。本章主要讲解了多线程程序中会遇到的问题。并且针对如何解决这些问题，做了一些简单的讲解。在接下来的一章中，我们将会深入学习 Java 为我们提供的解决这些问题的工具。我们不但在遇到问题的时候要知道如何解决，还应该了解解决问题的原理是什么。

}

13 问题的根源—Java内存模型简介
15 原子性轻量级实现—深入理解Atomic与CAS