二

69 如何用命令行和代码定位死锁?

本课时我们主要介绍“如何用命令和代码来定位死锁”。

在此之前，我们介绍了什么是死锁，以及死锁发生的必要条件。当然，即便我们很小心地编写代码，也必不可免地依然有可能会发生死锁，一旦死锁发生，**第一步要做的就是把它给找到**，因为在找到并定位到死锁之后，才能有接下来的补救措施，比如解除死锁、解除死锁之后恢复、对代码进行优化等；若找不到死锁的话，后面的步骤就无从谈起了。

下面就来看一下是如何用命令行的方式找到死锁的。

命令：jstack

这个命令叫作 jstack，它能看到我们 Java 线程的一些相关信息。如果是比较明显的死锁关系，那么这个工具就可以直接检测出来；如果死锁不明显，那么它无法直接检测出来，不过我们也可以**借此来分析线程状态，进而就可以发现锁的相互依赖关系**，所以这也是很有利于我们找到死锁的方式。

我们就来试一试，执行这个命令。

首先，我们运行一下第 67 讲的必然发生死锁的 **MustDeadLock** 类：

```
/**
 * 描述：        必定死锁的情况
 */
public class MustDeadLock implements Runnable {
    public int flag;
    static Object o1 = new Object();
    static Object o2 = new Object();
    public void run() {
        System.out.println("线程"+Thread.currentThread().getName() + "的flag为" + fl
```

```
        if (flag == 1) {
            synchronized (o1) {
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("线程1获得了两把锁");
                }
            }
        }
        if (flag == 2) {
            synchronized (o2) {
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println("线程2获得了两把锁");
                }
            }
        }
    }
    public static void main(String[] argv) {
        MustDeadLock r1 = new MustDeadLock();
        MustDeadLock r2 = new MustDeadLock();
```

```
            r1.flag = 1;
            r2.flag = 2;
            Thread t1 = new Thread(r1, "t1");
            Thread t2 = new Thread(r2, "t2");
            t1.start();
            t2.start();
        }
    }
```

由于它发生了死锁，在我们没有干预的情况下，程序在运行后就不会停止；然后打开我们的终端，执行 ${JAVA_HOME}/bin/jps 这个命令，就可以查看到当前 Java 程序的 pid，我的执行结果如下：

```
56402 MustDeadLock
56403 Launcher
56474 Jps
55051 KotlinCompileDaemon
```

有多行，可以看到第一行是 MustDeadLock 这类的 pid 56402；然后我们继续执行下一个命令，${JAVA_HOME}/bin/jstack 加空格，接着输入我们刚才所拿到的这个类的 pid，也就是 56402，所以完整的命令是 ${JAVA_HOME}/bin/jstack 56402；最后它会打印出很多信息，就包含了线程获取锁的信息，比如**哪个线程获取哪个锁，它获得的锁是在哪个语句中获得的，它正在等待或者持有的锁是什么**等，这些重要信息都会打印出来。我们截取一部分和死锁相关的有用信息，展示如下：

```
Found one Java-level deadlock:
=============================
"t2":
  waiting to lock monitor 0x00007fa06c004a18 (object 0x000000076adabaf0, a java.lan
  which is held by "t1"
"t1":
  waiting to lock monitor 0x00007fa06c007358 (object 0x000000076adabb00, a java.lan
```

```
  which is held by "t2"

Java stack information for the threads listed above:

===================================================

"t2":

        at lesson67.MustDeadLock.run(MustDeadLock.java:31)

        - waiting to lock <0x000000076adabaf0> (a java.lang.Object)

        - locked <0x000000076adabb00> (a java.lang.Object)

        at java.lang.Thread.run(Thread.java:748)

"t1":

        at lesson67.MustDeadLock.run(MustDeadLock.java:19)

        - waiting to lock <0x000000076adabb00> (a java.lang.Object)

        - locked <0x000000076adabaf0> (a java.lang.Object)

        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock
```

在这里它首先会打印“Found one Java-level deadlock”，表明“找到了一个死锁”，然后是更详细的信息，从中间这部分的信息中可以看出，t2 线程想要去获取这个尾号为 af0 的锁对象，但是它被 t1 线程持有，同时 t2 持有尾号为 b00 的锁对象；相反，t1 想要获取尾号为 b00 的锁对象，但是它被 t2 线程持有，同时 t1 持有的却是尾号为 af0 的锁对象，这就形成了一个依赖环路，发生了死锁。最后它还打印出了“Found 1 deadlock.”，可以看出，jstack 工具不但帮我们找到了死锁，甚至还把**哪个线程、想要获取哪个锁、形成什么样的环路**都告诉我们了，当我们有了这样的信息之后，死锁就非常容易定位了，所以接下来我们就可以进一步修改代码，来避免死锁了。

以上就是利用 jstack 来定位死锁的方法，jstack 可以用来帮助我们分析线程持有的锁和需要的锁，然后分析出是否有循环依赖形成死锁的情况。

代码：ThreadMXBean

下面我们再看一下用代码来定位死锁的方式。

我们会用到 ThreadMXBean 工具类，代码示例如下：

```
public class DetectDeadLock implements Runnable {
```

```
public int flag;
static Object o1 = new Object();
static Object o2 = new Object();
public void run() {
    System.out.println(Thread.currentThread().getName()+" flag = " + flag);
    if (flag == 1) {
        synchronized (o1) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            synchronized (o2) {
                System.out.println("线程1获得了两把锁");
            }
        }
    }
    if (flag == 2) {
        synchronized (o2) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
            synchronized (o1) {
                System.out.println("线程2获得了两把锁");
            }
        }
```

```
        }
    }
    public static void main(String[] argv) throws InterruptedException {
        DetectDeadLock r1 = new DetectDeadLock();
        DetectDeadLock r2 = new DetectDeadLock();
        r1.flag = 1;
        r2.flag = 2;
        Thread t1 = new Thread(r1,"t1");
        Thread t2 = new Thread(r2,"t2");
        t1.start();
        t2.start();
        Thread.sleep(1000);
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        long[] deadlockedThreads = threadMXBean.findDeadlockedThreads();
        if (deadlockedThreads != null && deadlockedThreads.length > 0) {
            for (int i = 0; i < deadlockedThreads.length; i++) {
                ThreadInfo threadInfo = threadMXBean.getThreadInfo(deadlockedThread
                System.out.println("线程id为"+threadInfo.getThreadId()+",线程名为" +
            }
        }
    }
}
```

这个类是在前面 MustDeadLock 类的基础上做了升级，MustDeadLock 类的主要作用就是让线程 1 和线程 2 分别以不同的顺序来获取到 o1 和 o2 这两把锁，并且形成死锁。在 main 函数中，在启动 t1 和 t2 之后的代码，是我们本次新加入的代码，我们用 Thread.sleep(1000) 来确保已经形成死锁，然后利用 ThreadMXBean 来检查死锁。

通过 ThreadMXBean 的 findDeadlockedThreads 方法，可以获取到一个 deadlockedThreads 的数组，然后进行判断，当这个数组不为空且长度大于 0 的时候，我

们逐个打印出对应的线程信息。比如我们打印出了**线程 id，也打印出了线程名，同时打印出了它所需要的那把锁正被哪个线程所持有**，那么这一部分代码的运行结果如下。

```
t1 flag = 1

t2 flag = 2

线程 id 为 12，线程名为 t2 的线程已经发生死锁，需要的锁正被线程 t1 持有。

线程 id 为 11，线程名为 t1 的线程已经发生死锁，需要的锁正被线程 t2 持有。
```

一共有四行语句，前两行是“t1 flag = 1“、“t2 flag = 2”，这是发生死锁之前所打印出来的内容；然后的两行语句就是我们检测到的死锁的结果，可以看到，它打印出来的是“线程 id 为 12，线程名为 t2 的线程已经发生了死锁，需要的锁正被线程 t1 持有。”同样的，它也会打印出“线程 id 为 11，线程名为 t1 的线程已经发生死锁，需要的锁正被线程 t2 持有。”

可以看出，ThreadMXBean 也可以帮我们找到并定位死锁，如果我们在业务代码中加入这样的检测，那我们就可以在发生死锁的时候及时地定位，**同时进行报警等其他处理**，也就增强了我们程序的健壮性。

总结

下面进行总结。本课时我们介绍了两种方式来定位代码中的死锁，在发生死锁的时候，我们可以用 jstack 命令，或者在代码中利用 ThreadMXBean 来帮我们去找死锁。

上一页 下一页