# 07 | 死锁：为什么流计算应用突然卡住，不处理数据了？

今天，我们来讨论一个非常有趣的话题，也就是流计算系统中的死锁问题。

在第 06 课时，我们讲解了 CompletableFuture 这个异步编程类的工作原理，并用它实现了一个流计算应用。为了流计算应用不会出现 OOM 问题，我们还专门使用 BackPressureExecutor 执行器，实现了反向压力的功能。

另外，我们在 05 课时已经讲过，描述一个流计算过程使用的是 DAG，也就是“有向无环图”。对于“有向”，我们知道这是代表着流数据的流向。而“无环”又是指什么呢？为什么一定要是“无环”？

其实之所以要强调“无环”，是因为在流计算系统中，当“有环”和“反向压力”一起出现时，流计算系统将会出现“死锁”问题。而程序一旦出现“死锁”，那除非人为干预，否则程序将一直停止执行，也就是我们常说的“卡死”。这在生产环境是绝对不能容忍的。

所以，我们今天将重点分析流计算系统中的“死锁”问题。

## 为什么流计算过程不能有环

我们从一个简单的流计算过程开始，这个流计算过程的 DAG 如下图 1 所示。

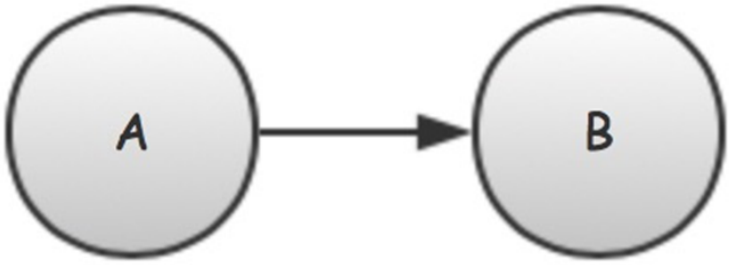



图 1 只包含两个步骤的流计算过程



DAG 描述了一个最简单的流计算过程，步骤 A 的输出给步骤 B 进行处理。

这个流计算过程用 CompletableFuture 实现非常简单。如下所示（请参考完整代码）：

```
ExecutorService AExecutor = new BackPressureExecutor(
        "AExecutor", 1, 1, 1, 10, 1);
ExecutorService BExecutor = new BackPressureExecutor(
        "BExecutor", 1, 1, 1, 10, 1);
AtomicLong itemCounter = new AtomicLong(0L);
String stepA() {
    String item = String.format("item%d", itemCounter.getAndDecrement());
    logger.info(String.format("stepA item[%s]", item));
    return item;
}
void stepB(String item) {
    logger.info(String.format("stepB item[%s]", item));
    sleep(10); // 睡眠一会，故意让 stepB 处理得比 stepA 慢
}
void demo1() {
    while (!Thread.currentThread().isInterrupted()) {
        CompletableFuture
                .supplyAsync(this::stepA, this.AExecutor)
                .thenAcceptAsync(this::stepB, this.BExecutor);
    }
}
```

上面的代码中，步骤 A 和步骤 B 使用了两个不同的执行器，即 AExecutor 和 BExecutor 。并且为了避免 OOM 问题，我们使用的执行器都是带反向压力功能的 BackPressureExecutor。

上面的程序运行起来没有任何问题。即便我们明确通过 sleep 函数，让 stepB 的处理速度只有 stepA 的十分之一，上面的程序都能够长时间的稳定运行（stepA 和 stepB 会不断打印出各自的处理结果，并且绝不会出现 OOM 问题）。

到此为止，一切都非常符合我们的预期。

但是现在，我们需要对图 1 的 DAG 稍微做点变化，让 B 在处理完后，将其结果重新输入给自己再处理一次。这种处理逻辑，在实际开发中也会经常遇到，比如 B 在处理失败时，就将处理失败的任务，重新添加到自己的输入队列，从而实现失败重试的功能。

修改后的 DAG 如下图所示。

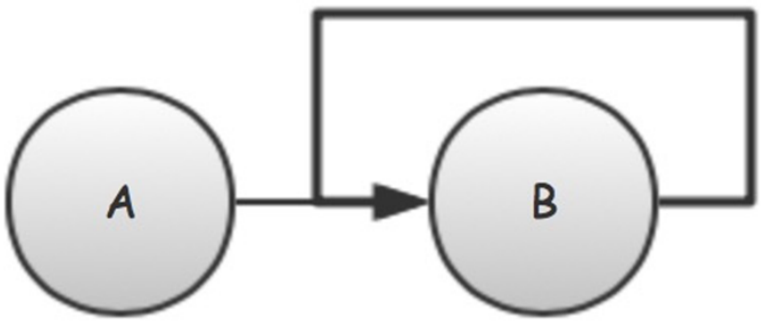



图2 步骤 B 的输出重新作为其输入



很明显，上面的 DAG 在步骤 B 上形成了一个“环”，因为有一条从 B 开始的有向线段，重新指向了 B 自己。相应的，前面的代码也需要稍微做点调整，改成下面的方式：

```
void demo1() {
    while (!Thread.currentThread().isInterrupted()) {
        CompletableFuture
                .supplyAsync(this::stepA, this.AExecutor)
                .thenApplyAsync(this::stepB, this.BExecutor)
                .thenApplyAsync(this::stepB, this.BExecutor);
    }
}
```

上面的代码中，我们增加了一次 thenApplyAsync 调用，用于将 stepB 的输出重新作为其输入。需要注意的是，由于第二次 stepB 调用后没有再设置后续步骤，所以，虽然 DAG 上“有环”，但 stepB 并不会形成死循环。

上面这段代码，初看起来并没什么问题，毕竟就是简单地新增了一个“重试”的效果嘛。但是，如果你实际运行上面这段代码就会发现，只需要运行不到 1 秒钟，上面这段程序就会“卡”住，之后控制台会一动不动，没有一条日志打印出来。

这是怎么回事呢？事实上这就是因为程序已经“死锁”了！

## 流计算过程死锁分析

说到“死锁”，你一定会想到“锁”的使用。一般情况下之所以会出现“死锁”，主要是因为我们使用锁的方式不对，比如使用了不可重入锁，或者使用多个锁时出现了交叉申请锁的情况。这种情况下出现的“死锁”问题，我们确确实实看到了“锁”的存在。

但当我们在使用流计算编程时，你会发现，“流”的编程方式已经非常自然地避免了“锁”的使用，也就是说我们并不会在“流”处理的过程中用到“锁”。这是因为，当使用“流”时，被处理的对象依次从上游流到下游。当对象在流到某个步骤时，它是被这个步骤的线程唯一持有，因此不存在对象竞争的问题。

但这是不是就说流计算过程中不会出现“死锁”问题呢？不是的。最直接的例子就是前面的代码，我们根本就没有用到“锁”，但它还是出现了“死锁”的问题。

所以，为什么会出现“死锁”呢？这里就需要我们仔细分析下了。下面的图 3 描绘了图 2 中的流计算过程之所以会发生死锁的原因。

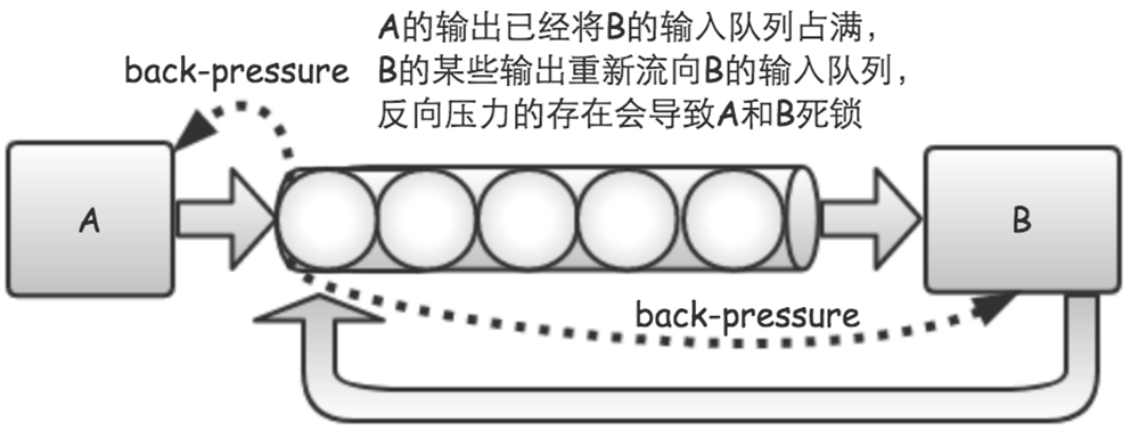



图3 流计算死锁之输出变输入



在图 3 中，整个流计算过程有 A 和 B 这两个步骤，并且具备“反向压力”能力。这时候，如果 A 的输出已经将 B 的输入队列占满，而 B 的输出又需要重新流向 B 的输入队列，那么由于“反向压力”的存在，B 会一直等到其输入队列有空间可用。而 B 的输入队列又因为 B 在等待，永远也不会有空间被释放，所以 B 会一直等待下去。同时，A 也会因为 B 的输入队列已满，由于反向压力的存在，它也只能不停地等待下去。

如此一来，整个流计算过程就形成了一个死锁，A 和 B 两个步骤都会永远等待下去，这样就出现了我们前边看到的程序“卡”住现象。

## 形成“环”的原因

在图 2 所示的 DAG 中，我们是因为需要让 stepB 失败重试，所以“随手”就让 stepB 将其输出重新作为输入重新执行一次。这姑且算是一种比较特殊的需求吧。

但在实际开发过程中，我们的业务逻辑明显是可以分为多个依次执行的步骤，用 DAG 画出来时，也是“无环”的。但在写代码时，有时候一不小心，也会无意识地将一个本来无环的 DAG，实现成了有环的过程。下面图 4 就说明了这种情况。

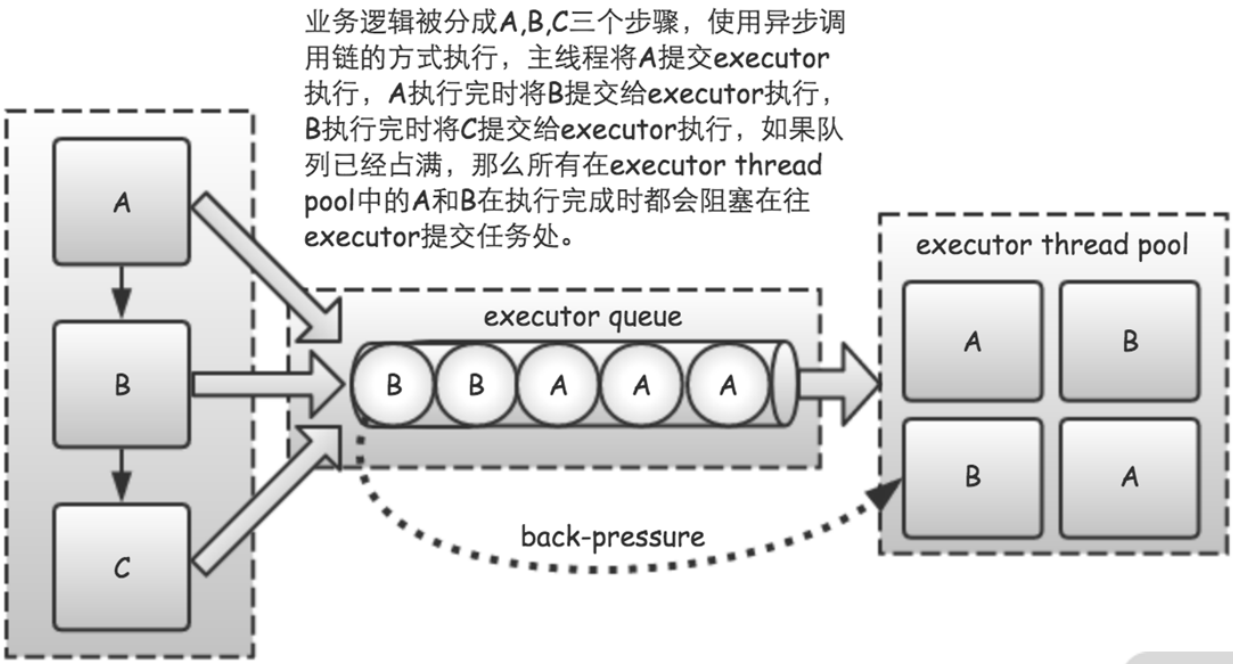



图4 流计算死锁之多个步骤使用同一执行器



在图 4 中，业务逻辑本来是 A 到 B 到 C 这样的“无环”图，**结果由于我们给这三个不同的步骤，分配了同一个执行器 executor，实际实现的流计算过程就成了一个“有环”的过程**。

在这个“有环”的实现中，只要任意一个步骤的处理速度比其他步骤慢，就会造成执行器的输入队列占满。一旦输入队列占满，由于反向压力的存在，各个步骤的输出就不能再输入到队列中。最终，所有执行步骤将会阻塞，也就形成了死锁，整个系统也被“卡”死。

## 如何避免死锁

所以，我们在流计算过程中，应该怎样避免死锁呢？其实很简单，有三种方法。

一是**不使用反向压力功能**。只需要我们不使用反向压力功能，即使业务形成“环”了，也不会死锁，因为每个步骤只需要将其输出放到输入队列中，不会发生阻塞等待，所以就不会死锁了。但很显然，这种方法禁止使用。毕竟，没有反向压力功能，就又回到 OOM 问题了，这是万万不可的！

二是**避免业务流程形成“环”**。这个方法最主要的作用，是指导我们在设计业务流程时，不要将业务流程设计成“有环”的了。否则如果系统有反向压力功能的话，容易出现类似于图 3 的死锁问题。

三是**千万不要让多个步骤使用相同的队列或执行器**。这个是最容易忽略的问题，特别是一些对异步编程和流计算理解不深的开发人员，最容易给不同的业务步骤分配相同的队列或执行器，在不知不觉中就埋下了死锁的隐患。

总的来说，在流计算过程中，反向压力功能是必不可少的，**为了避免“死锁”的问题，流计算过程中的任何一个步骤，它的输出绝不能再重新流回作为它的输入**。

只需要注意以上几点，你就可以放心大胆地使用“流”式编程了，而且不用考虑“锁”的问题。由于没有了竞态问题，这既可以简化你编程的过程，也可以给程序带来显著的性能提升。

## 小结

今天，我们分析了流计算过程中的死锁问题。这是除 OOM 问题外，另一个需要尤其注意的问题。

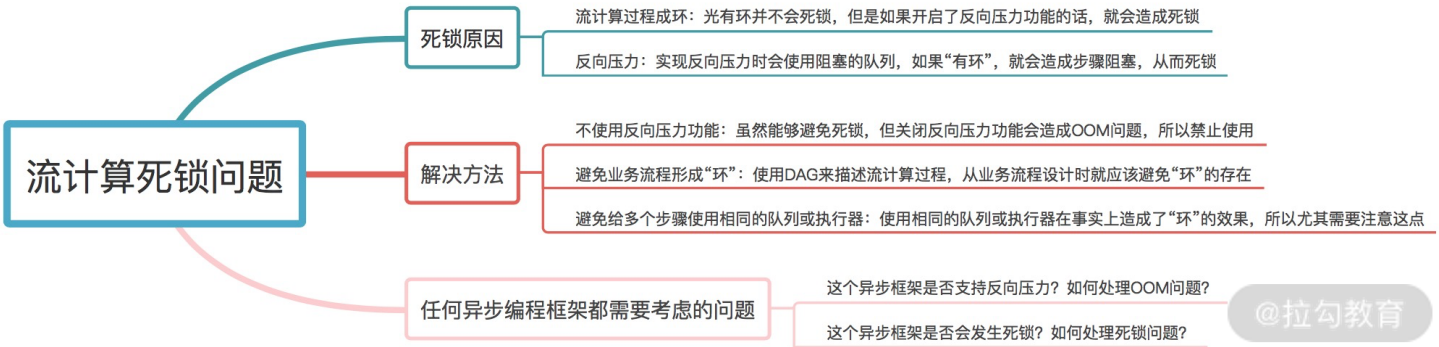
我们之前说过，“流”的本质是“异步”的，并且你可以看到，我们今天实现描述流计算过程的 DAG 时，用的就是 CompletableFuture 这个异步编程框架。所以，其实流计算的这种死锁问题，在其他“异步”场景下也会出现。

如果你需要使用其他编程语言或其他异步编程框架（比如 Node.js 中的 async 和 await）进行程序开发的话，一定要注意以下问题：

- 这个异步框架支持反向压力？没有不支持的话，是如何处理 OOM 问题的？
- 这个异步框架会发生死锁吗？如果会死锁的话，是如何处理死锁问题的？

那么，你在以往的异步编程过程中，有没有遇到过死锁的问题呢？你可以将你遇到的问题，写在留言区！

> 本课时精华：




点击此链接查看本课程所有课时的源码


PB 级企业大数据项目实战 + 拉勾硬核内推，5 个月全面掌握大数据核心技能。点击链接，全面赋能！