

**解决死锁的100种方法**

[](https://www.zhihu.com/people/douliyoulatiao)

[**兜里有辣条**](https://www.zhihu.com/people/douliyoulatiao)

资深Java后端开发工程师

19 人赞同了该文章

死锁是多线程编程或者说是并发编程中的一个经典问题，也是我们在实际工作中很可能会碰到的问题。相信大部分读者对“死锁”这个词都是略有耳闻的，但从我对后端开发岗位的面试情况来看很多同学往往对死锁都还没有系统的了解。虽然“死锁”听起来很高深，但是实际上已经被研究得比较透彻，大部分的解决方法都非常成熟和清晰，所以大家完全不用担心这篇文章的难度。

虽然本文是一篇介绍死锁及其解决方式的文章，但是对于多线程程序中的非死锁问题我们也应该有所了解，这样才能写出正确且高效的多线程程序。多线程程序中的非死锁问题主要分为两类：

1. 违反原子性问题

* 一些语句在底层会被分为多个底层指令运行，所以在多个线程之间这些指令就可能会存在穿插，这样程序的行为就可能会与预期不符造成bug。

2. 违反执行顺序问题

* + 一些程序语句可能会因为子线程立即启动早于父线程中的后续代码，或者是多个线程并发执行等情况，造成程序运行顺序和期望不符导致产生bug。

这两大非死锁多线程问题及其解决方案在之前的文章《[多线程中那些看不到的陷阱](https://link.zhihu.com/?target=https%3A//juejin.im/post/5cacb4c16fb9a068b36776ed)》里都有详细的介绍，感兴趣的读者可以了解一下。

接下来就让我们开始消灭死锁吧！

**初识死锁**

**什么是死锁？**

死锁，顾名思义就是导致线程卡死的锁冲突，例如下面的这种情况：



可以看出，上面的两个线程已经互相卡死了，线程t1在等待线程t2释放锁B，而线程t2在等待线程t1释放锁A。两个线程互不相让也就没有一个线程可以继续往下执行了。这种情况下就发生了**死锁**。

**死锁的四个必要条件**

上面的情况只是死锁的一个例子，我们可以用更精确的方式描述死锁出现的条件：

1. 互斥。资源被竞争性地访问，这里的资源可以理解为锁；
2. 持有并等待。线程持有已经分配给他们的资源，同时等待其他的资源；
3. 不抢占。线程已经获取到的资源不会被其他线程强制抢占；
4. 环路等待。线程之间存在资源的环形依赖链，每个线程都依赖于链条中的下一个线程释放必要的资源，而链条的末尾又依赖了链条头部的线程，进入了一个循环等待的状态。

上面这四个都是死锁出现的必要条件，如果其中任何一个条件不满足都不会出现死锁。虽然这四个条件的定义看起来非常的理论和官方，但是在实际的编程实践中，我们正是在死锁的这四个必要条件基础上构建出解决方案的。所以这里不妨思考一下这四个条件各自的含义，想一想如果去掉其中的一个条件死锁是否还能发生，或者为什么不能发生。

**阻止死锁的发生**

了解了死锁的概念和四个必要条件之后，我们下面就正式开始解决死锁问题了。对于死锁问题，我们最希望能够达到的当然是完全不发生死锁问题，也就是在死锁发生之前就阻止它。

那么想要阻止死锁的发生，我们自然是要让死锁无法成立，最直接的方法当然是破坏掉死锁出现的必要条件。只要有任何一个必要条件无法成立，那么死锁也就没办法发生了。

**破坏环路等待条件**

实践中最有效也是最常用的一种死锁阻止技术就是**锁排序**，通过对加锁的操作进行排序我们就能够破坏**环路等待条件**。例如当我们需要获取数组中某一个位置对应的锁来修改这个位置上保存的值时，如果需要同时获取多个位置对应的锁，那么我们就可以按位置在数组中的排列先后顺序统一从前往后加锁。

试想一下如果程序中所有需要加锁的代码都按照一个统一的固定顺序加锁，那么我们就可以想象锁被放在了一条不断向前延伸的直线上，而因为加锁的顺序一定是沿着这条线向下走的，所以每条线程都只能向前加锁，而不能再回头获取已经在后面的锁了。这样一来，线程只会向前单向等待锁释放，自然也就无法形成一个环路了。

其实大部分死锁解决方法不止可以用于多线程编程领域，还可以扩展到更多的并发场景下。比如在数据库操作中，如果我们要对某几行数据执行更新操作，那么就会获取这几行数据所对应的锁，我们同样可以通过对数据库更新语句进行排序来阻止在数据库层面发生的死锁。

但是这种方案也存在它的缺点，比如在大型系统当中，不同模块直接解耦和隔离得非常彻底，不同模块的研发同学之间都不清楚具体的实现细节，在这样的情况下就很难做到整个系统层面的全局锁排序了。在这种情况下，我们可以对方案进行扩充，例如Linux在内存映射代码就使用了一种锁分组排序的方式来解决这个问题。锁分组排序首先按模块将锁分为了不同的组，每个组之间定义了严格的加锁顺序，然后再在组内对具体的锁按规则进行排序，这样就保证了全局的加锁顺序一致。在Linux的对应的源码顶部，我们可以看到有非常详尽的注释定义了明确的锁排序规则。

这种解决方案如果规模过大的话即使可以实现也会非常的脆弱，只要有一个加锁操作没有遵守锁排序规则就有可能会引发死锁。不过在像微服务之类解耦比较充分的场景下，只要架构拆分合理，任务模块尽可能小且不会将加锁范围扩大到模块之外，那么锁排序将是一种非常实用和便捷的死锁阻止技术。

**破坏持有并等待条件**

想要破坏**持有并等待条件**，我们可以一次性原子性地获取所有需要的锁，比如通过一个专门的全局锁作为**加锁令牌**控制加锁操作，只有获取了这个锁才能对其他锁执行加锁操作。这样对于一个线程来说就相当于一次性获取到了所有需要的锁，且除非等待**加锁令牌**否则在获取其他锁的过程中不会发生锁等待。

这样的解决方案虽然简单粗暴，但这种简单粗暴也带来了一些问题：

1. 这种实现会降低系统的并发性，因为所有需要获取锁的线程都要去竞争同一个**加锁令牌**锁；
2. 并且因为要在程序的一开始就获取所有需要的锁，这就导致了线程持有锁的时间超出了实际需要，很多锁资源被长时间的持有所浪费，而其他线程只能等待之前的线程执行结束后统一释放所有锁；
3. 另一方面，现代程序设计理念要求我们提高程序的封装性，不同模块之间的细节要互相隐藏，这就使得在一个统一的位置一次性获取所有锁变得不再可能。

**破坏不抢占条件**

如果一个线程已经获取到了一些锁，那么在这个线程释放锁之前这些锁是不会被强制抢占的。但是为了防止死锁的发生，我们可以选择让线程在获取后续的锁失败时主动放弃自己已经持有的锁并在之后重试整个任务，这样其他等待这些锁的线程就可以继续执行了。

同样的，这个方案也会有自己的缺陷：

1. 虽然这种方式可以避免死锁，但是如果几个互相存在竞争的线程不断地放弃、重试、放弃，那么就会导致活锁问题（livelock）。在这种情况下，虽然线程没有因为锁冲突被卡死，但是仍然会被阻塞相当长的时间甚至一直处于重试当中。

* + 这个问题的一种解决方式是给任务重试添加一个随机的延迟时间，这样就能大大降低任务冲突的概率了。在一些接口请求框架中也使用了这种技巧来分散服务高峰期的请求重试操作，防止服务陷入阻塞、崩溃、阻塞的恶性循环。

2. 还是因为程序的封装性，在一个模块中难以释放其他模块中已经获取到的锁。

虽然每一个方案都有自己的缺陷，但是在适合它们的场景下，它们都能发挥出巨大的作用。

**破坏互斥条件**

在之前的文章中，我们已经了解了一种与锁完全不同的同步方式CAS。通过CAS提供的原子性支持，我们可以实现各种无锁数据结构，不仅避免了互斥锁所带来的开销和复杂性，也由此避开了我们一直在讨论的死锁问题。

AtomicInteger类中就大量使用了CAS操作来实现并发安全，例如incrementAndGet()方法就是用Unsafe类中基于CAS的原子累加方法getAndAddInt来实现的。下面是Unsafe类的getAndAddInt方法实现：

*/\*\**

*\* 增加指定字段值并返回原值*

*\**

*\* @param obj 目标对象*

*\* @param valueOffset 目标字段的内存偏移量*

*\* @param increment 增加值*

*\* @return 字段原值*

*\*/*

**public** **final** **int** **getAndAddInt(**Object obj**,** **long** valueOffset**,** **int** increment**)** **{**

*// 保存字段原值的变量*

**int** oldValue**;**

**do** **{**

*// 获取字段原值*

oldValue **=** **this.**getIntVolatile**(**obj**,** valueOffset**);**

*// obj和valueOffset唯一指定了目标字段所对应的内存区域*

*// while条件中不断调用CAS方法来对目标字段值进行增加，并保证字段的值没有被其他线程修改*

*// 如果在修改过程中其他线程修改了这个字段的值，那么CAS操作失败，循环语句会重试操作*

**}** **while(!this.**compareAndSwapInt**(**obj**,** valueOffset**,** oldValue**,** oldValue **+** increment**));**

*// 返回字段的原值*

**return** oldValue**;**

**}**

上面代码中的compareAndSwapInt方法就是我们说的CAS操作（Compare And Swap），我们可以看到，CAS在每次执行时不一定会成功。如果执行CAS操作时目标字段的值已经被别的线程修改了，那么这次CAS操作就会失败，循环语句将会在CAS操作失败的情况下不断重试同样的操作。这种不断重试的方式就被称为**自旋**，在jvm当中对互斥锁的等待也会通过少量的自旋操作来进行优化。

不过如果一个变量同时被多个线程以CAS方式修改，那么就有可能导致出现活锁，多个线程将会一直不断重试CAS操作。所以CAS操作的成本和数据竞争的激烈程度密切相关，在一些竞争非常激烈的情况下，CAS操作的成本甚至会超过互斥锁。

除了累加整型值这样的简单场景之外，还有更多更复杂的无锁(lock-free)数据结构，例如java.util.concurrent包中的ConcurrentLinkedDeque双端队列类就是一个无锁的并发安全链表实现，有兴趣的读者可以了解一下。

这种方法同样可以用在数据库操作上，当我们执行update语句时可以在where子句中添加上一些字段的旧值作为条件，比如update t\_xxxx set value = <newValue>, version = version + 1 where id = xxx and version = 10，这样我们就可以通过update语句返回的影响行数是不是0来判断更新操作有没有成功了，这是不是和CAS很相似？

**其他解决死锁的方法 —— 探测并恢复**

有时，我们并不需要完全阻止死锁的发生，而是可以通过其他的手段来控制死锁的影响。就像如果新的治疗手段可以使癌症病人继续活七八十年，那么癌症也就没有那么可怕了。

还有一种解决死锁的方法就是让死锁发生，之后再解决它，就像电脑死机以后直接重启一样。使用这种方法我们可以这么做：如果多个线程出现了死锁的情况，那么我们就杀死足够多的线程使系统恢复到可运行状态。在我们常用的关系型数据库中使用的就是这种方法，数据库会周期性地使用探测器创建**资源图**，然后检查其中是否存在循环。如果探测到了循环（死锁），那么数据库就会根据估算的执行成本高低杀死可以解决死锁问题的尽可能成本最小的线程。

数据库在被外部应用调用的过程中是没办法获知外部应用的逻辑细节的，所以自然也就没办法用之前说的种种方法来解决死锁问题，只能通过事后检测并恢复来对死锁问题做最低限度的保障。但是我们可以在我们的应用程序中应用更多的解决方案，从更上层解决死锁问题。

**总结**

在这篇文章中，我们从死锁的概念出发，首先介绍了死锁是什么和死锁发生的**四个必要条件**。然后通过破坏任意一个必要条件产生了四种不同的阻止死锁的解决方案，最后介绍了另外一种死锁解决方法——在死锁发生后再探测并恢复系统运行。相信大家可以在不同的场景中都能找到适合该场景的解决方案，但是锁本质上是容易引入问题的，所以如果不是确有必要，最好不要贸然用锁来进行处理。