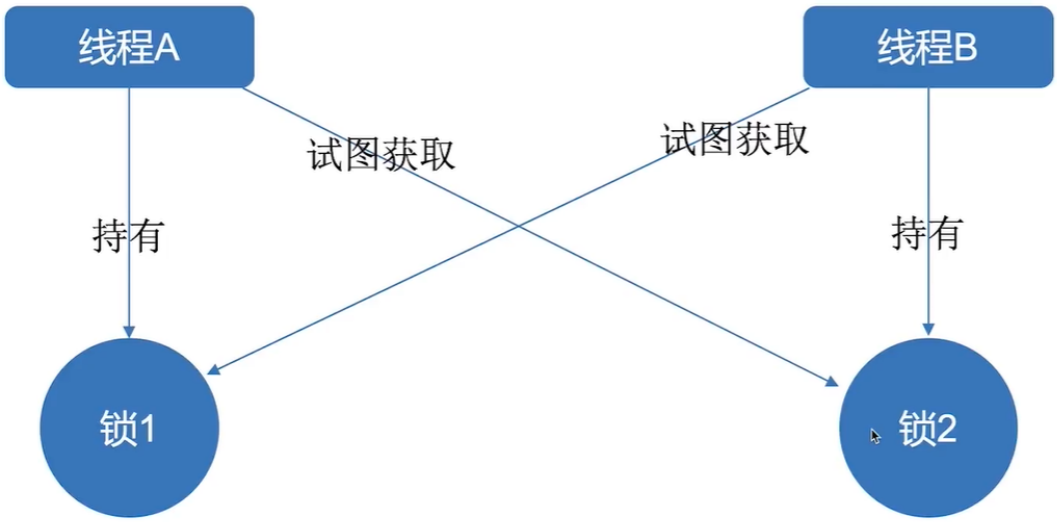
死锁

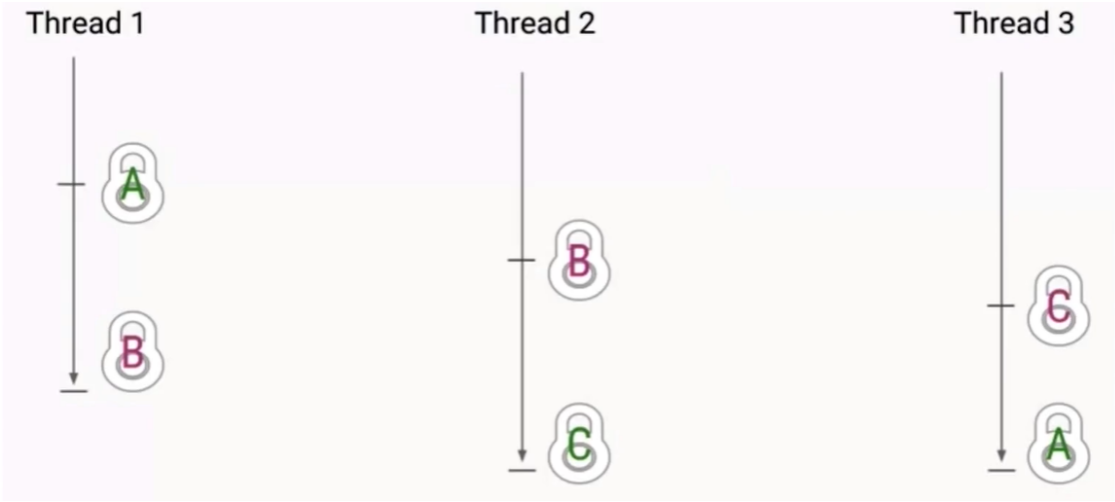
一、什么是死锁

- 发生在并发中
- 互不相让:当两个(或更多)线程(或进程)相互持有对方所需要的资源,又不主动释放,导致所有人都无法继续前进, 导致程序陷入无尽的阻塞,这就是死锁。
- 一图胜千言



- 多个线程发生死锁

  如果多个线程之间的依赖关系是环形,存在环路的锁的依赖关系,那么也可能会发生死锁



二、哲学化就餐问题

代码演示哲学家就餐死锁问题

```
1  package deadlock;
2
3
4  /**
```

```
 * 描述:       演示哲学家就餐问题导致的死锁，所有哲学家吃饭都先拿左边筷子
 *             synchronized (leftChopstick) {
 *                  doAction("Picked up left chopstick");
 *                  synchronized (rightChopstick) {、
 *                          // eating
 *                  }
 *             }
 */
public class DiningPhilosophers {

    public static class Philosopher implements Runnable {

        private Object leftChopstick;

        public Philosopher(Object leftChopstick, Object rightChopstick) {
            this.leftChopstick = leftChopstick;
            this.rightChopstick = rightChopstick;
        }

        private Object rightChopstick;

        @Override
        public void run() {
            try {
                while (true) {
                    doAction("Thinking");
                    synchronized (leftChopstick) {
                        doAction("Picked up left chopstick");
                        synchronized (rightChopstick) {
                            doAction("Picked up right chopstick - eating");
                            doAction("Put down right chopstick");
                        }
                        doAction("Put down left chopstick");
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        private void doAction(String action) throws InterruptedException {
            System.out.println(Thread.currentThread().getName() + " " +
action);
            Thread.sleep((long) (Math.random() * 10));
        }
    }

    public static void main(String[] args) {
        Philosopher[] philosophers = new Philosopher[5];
        Object[] chopsticks = new Object[philosophers.length];
        for (int i = 0; i < chopsticks.length; i++) {
            chopsticks[i] = new Object();
        }
        for (int i = 0; i < philosophers.length; i++) {
            Object leftChopstick = chopsticks[i];
            Object rightChopstick = chopsticks[(i + 1) % chopsticks.length];
            philosophers[i] = new Philosopher(leftChopstick,
rightChopstick);
```

```
            new Thread(philosophers[i], "哲学家" + (i + 1) + "号").start();
        }
    }
}
```

解决方案:

1. 服务员检查(避免策略)
2. 改变一个哲学家拿叉子的顺序(避免策略)
3. 餐票（避免策略）5个哲学家只有四张餐票
4. 领导调节（检测与恢复策略）： 破坏不可剥夺条件

检测与恢复策略

> 一段时间检测是否有死锁,如果有就剥夺某一个资源,来打开死锁

- 允许发生死锁
- 每次调用锁都记录
- 定期检查“锁的调用链路图”中是否存在环路
- 一旦发生死锁,就用死锁恢复机制进行恢复

恢复方法

1. 进程终止

   逐个终止线程,直到死锁消除。

   终止顺序

   优先级(是前台交互还是后台处理)，应尽量不影响前台的交互 已占用资源、还需要的资源 已经运行时间

2. 资源抢占

   把已经分发出去的锁给收回来让线程回退几步,这样就不用结束整个线程,成本比较低

   缺点:可能同一个线程一直被抢占,那就造成饥饿

检测算法:

锁的调用链路图，用一个有向图来记录锁的调用，如果发生了环路说明有死锁的发生。

改变一个哲学家拿叉子的顺序(避免策略)

```
package deadlock;


/**
 * 描述:       演示哲学家就餐问题导致的死锁 -- 解决方案一：
 *             if (i == philosophers.length - 1) {
 *                   philosophers[i] = new Philosopher(rightChopstick, leftChopstick);
 *             }
 *             如果是最后一个哲学家就餐，就拿右边筷子，再拿左边筷子，颠倒顺序，如果右边没有筷子了，就没有必要再拿左边筷子了。
 *
 */
public class DiningPhilosophersSolution1 {

    public static class Philosopher implements Runnable {
```

```
15
16          private Object leftChopstick;
17
18          public Philosopher(Object leftChopstick, Object rightChopstick) {
19              this.leftChopstick = leftChopstick;
20              this.rightChopstick = rightChopstick;
21          }
22
23          private Object rightChopstick;
24
25          @Override
26          public void run() {
27              try {
28                  while (true) {
29                      doAction("Thinking");
30                      synchronized (leftChopstick) {
31                          doAction("Picked up left chopstick");
32                          synchronized (rightChopstick) {
33                              doAction("Picked up right chopstick - eating");
34                              doAction("Put down right chopstick");
35                          }
36                          doAction("Put down left chopstick");
37                      }
38                  }
39              } catch (InterruptedException e) {
40                  e.printStackTrace();
41              }
42          }
43
44          private void doAction(String action) throws InterruptedException {
45              System.out.println(Thread.currentThread().getName() + " " +
action);
46              Thread.sleep((long) (Math.random() * 10));
47          }
48      }
49
50      public static void main(String[] args) {
51          Philosopher[] philosophers = new Philosopher[5];
52          Object[] chopsticks = new Object[philosophers.length];
53          for (int i = 0; i < chopsticks.length; i++) {
54              chopsticks[i] = new Object();
55          }
56          for (int i = 0; i < philosophers.length; i++) {
57              Object leftChopstick = chopsticks[i];
58              Object rightChopstick = chopsticks[(i + 1) % chopsticks.length];
59              if (i == philosophers.length - 1) {
60                  philosophers[i] = new Philosopher(rightChopstick,
leftChopstick);
61              } else {
62                  philosophers[i] = new Philosopher(leftChopstick,
rightChopstick);
63              }
64              new Thread(philosophers[i], "哲学家" + (i + 1) + "号").start();
65          }
66      }
67  }
```

实际工程中避免死锁的8个tips

1. 设置超时时间，使用tryLock

Lockeytrylock(long timeout, Timeunit unit) synchronized不具备尝试锁的能力

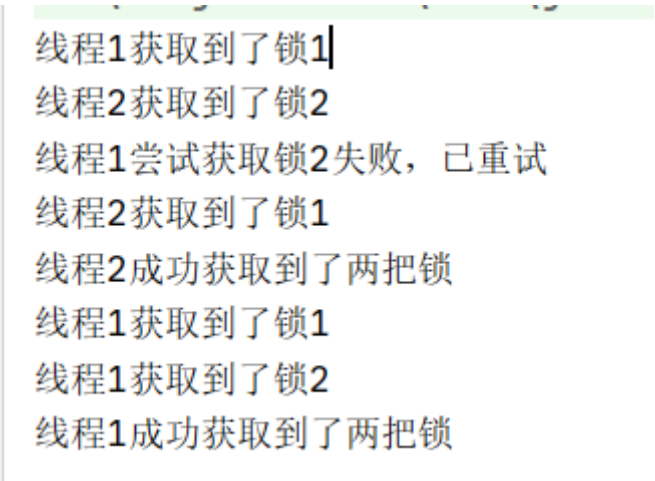
造成超时的可能性多:发生了死锁、线程陷入死循环、线程执行很慢。获取锁失败:打日志、发报警邮件、重启等

代码演示

```
package deadlock;

import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 描述：      用tryLock来避免死锁
 */
public class TryLockDeadlock implements Runnable {

    int flag = 1;
    static Lock lock1 = new ReentrantLock();
    static Lock lock2 = new ReentrantLock();

    public static void main(String[] args) {
        TryLockDeadlock r1 = new TryLockDeadlock();
        TryLockDeadlock r2 = new TryLockDeadlock();
        r1.flag = 1;
        r2.flag = 0;
        new Thread(r1).start();
        new Thread(r2).start();
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (flag == 1) {
                try {
                    if (lock1.tryLock(800, TimeUnit.MILLISECONDS)) {
                        System.out.println("线程1获取到了锁1");
                        Thread.sleep(new Random().nextInt(1000));
                        if (lock2.tryLock(800, TimeUnit.MILLISECONDS)) {
                            System.out.println("线程1获取到了锁2");
                            System.out.println("线程1成功获取到了两把锁");
                            lock2.unlock();
                            lock1.unlock();
                            break;
                        } else {
                            System.out.println("线程1尝试获取锁2失败，已重试");
                            lock1.unlock();
                            Thread.sleep(new Random().nextInt(1000));
                        }
                    } else {
                        System.out.println("线程1获取锁1失败，已重试");
                    }
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            if (flag == 0) {
                try {
                    if (lock2.tryLock(3000, TimeUnit.MILLISECONDS)) {
                        System.out.println("线程2获取到了锁2");

                        Thread.sleep(new Random().nextInt(1000));
                        if (lock1.tryLock(3000, TimeUnit.MILLISECONDS)) {
                            System.out.println("线程2获取到了锁1");
                            System.out.println("线程2成功获取到了两把锁");
                            lock1.unlock();
                            lock2.unlock();
                            break;
                        } else {
                            System.out.println("线程2尝试获取锁1失败，已重试");
                            lock2.unlock();
                            Thread.sleep(new Random().nextInt(1000));
                        }
                    } else {
                        System.out.println("线程2获取锁2失败，已重试");
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

输出结果



2. 多使用并发类而不是自己设计锁

- ConcurrentHashMap、 ConcurrentLinkedQueue、AtomicBoolean等
- 实际应用中java. util. concurrent. atomic 十分有用,简单方便，且效率比使用Lock更高
  - 多用并发集合少用同步集合,并发集合比同步集合的可扩展性更好
  - 并发场景需要用到map,首先想到用 ConcurrentHashMap
- 尽量降低锁的使用粒度:用不同的锁而不是一个锁
- 如果能使用同步代码块,就不使用同步方法:自己指定锁对象

- 给你的线程起个有意义的名字: debug和排查时事半功倍,框架和JDK都遵守这个最佳实践
- 避免锁的嵌套: Mustdead Lock类
- 分配资源前先看能不能收回来:银行家算法
- 尽量不要几个功能用同一把锁:专锁专用

三、活锁

什么是活锁

死锁:每个哲学家都拿着左手的餐叉,永远都在等右边的餐叉(或者相反) 活锁:在完全相同的时刻进入餐厅,并同时拿起左边的餐叉那么这些哲学家就会等待五分钟,同时放下手中的餐叉,再**都**等五分钟,又同时拿起这些餐叉。

活锁虽然线程并没有阻塞,也始终在运行(所以叫做"活"锁,线程是"活"的),但是程序却得不到进展,因为线程始终重复做同样的事 如果这里死锁,那么就是这里两个人都始终一动不动,直到对方先抬头,他们之间不再说话了,只是等待

活锁不阻塞，消耗cpu资源； 死锁阻塞，不消耗cpu资源

代码演示：牛郎织女没饭吃

```
1  package deadlock;
2
3  import java.util.Random;
4  import jdk.management.resource.internal.inst.RandomAccessFileRMHooks;
5
6  /**
7   * 描述:       演示活锁问题
8   */
9  public class LiveLock {
10
11     static class Spoon {
12
13         private Diner owner;
14
15         public Spoon(Diner owner) {
16             this.owner = owner;
17         }
18
19         public Diner getOwner() {
20             return owner;
21         }
22
23         public void setOwner(Diner owner) {
24             this.owner = owner;
25         }
26
27         public synchronized void use() {
28             System.out.printf("%s吃完了!", owner.name);
29         }
30     }
31
32     static class Diner {
33
34         private String name;
```

```
35          private boolean isHungry;
36
37          public Diner(String name) {
38              this.name = name;
39              isHungry = true;
40          }
41
42          public void eatWith(Spoon spoon, Diner spouse) {
43              while (isHungry) {
44                  if (spoon.owner != this) {
45                      try {
46                          Thread.sleep(1);
47                      } catch (InterruptedException e) {
48                          e.printStackTrace();
49                      }
50                      continue;
51                  }
52  //                Random random = new Random();
53                  if (spouse.isHungry /*&& random.nextInt(10) < 9*/) {
54                      System.out.println(name + ": 亲爱的" + spouse.name + "你
    先吃吧");
55                      spoon.setOwner(spouse);
56                      continue;
57                  }
58
59                  spoon.use();
60                  isHungry = false;
61                  System.out.println(name + ": 我吃完了");
62                  spoon.setOwner(spouse);
63              }
64          }
65      }
66
67
68      public static void main(String[] args) {
69          Diner husband = new Diner("牛郎");
70          Diner wife = new Diner("织女");
71
72          Spoon spoon = new Spoon(husband);
73
74          new Thread(new Runnable() {
75              @Override
76              public void run() {
77                  husband.eatWith(spoon, wife);
78              }
79          }).start();
80
81          new Thread(new Runnable() {
82              @Override
83              public void run() {
84                  wife.eatWith(spoon, husband);
85              }
86          }).start();
87      }
88  }
```

解决的办法就是，打开上面代码中的注释，定义一个随机数，增加一个随即条件。

工程中的活锁实例:消息队列

策略:消息如果处理失败,就放在队列**开头**重试

由于依赖服务出了问题,处理该消息一直失败

没阻塞,但程序无法继续

解决方法：放到队列尾部、重试限制

如如何解决活锁问题

以太网的指数退避算法

加入随机因素

饥饿

当线程需要某些资源(例如CPU),但是却始终得不到

线程的**优先级设置得过于低**，或者有某线程持有锁同时又无限循环从而不释放锁，或者某程序始终占用某文件的写锁

饥饿可能会导致响应性差:比如,我们的浏览器有一个线程负责处理前台响应(打开收藏夹等动作),另外的后台线程负责下载图片和文件、计算渲染等。在这种情况下,如果后台线程把CPU资源都占用了,那么前台线程将无法得到很好地执行,这会导致用户的体验很差

回顾线程优先级

- 10个级别,默认5
- 程序设计不应依赖于优先级
    - 不同OS对应的优先级是不一样的。
    - 优先级会被操作系统改变

如何定位死锁

jstack

通过堆栈分析发现死锁

ThreadMXBean