线程核心基础

8、常见面试问题

笔记来源：慕课网悟空老师视频《Java并发核心知识体系精讲》

1. 线程启动模式

1. 方法

- 方法一：实现Runnable接口
- 方法二：继承Thread类

2. 两种方法的对比

- 结论：方法一（实现Runnable接口）更好
- 问题：为什么继承Thread类是不推荐的?

```
1. 从代码架构角度：具体的任务（run方法）应该和“创建和运行线程的机制（Thread类）”解耦，用runnable对象可以实现解耦。
2. 使用继承Thread的方式的话，那么每次想创建一个新任务，只能新建一个独立的线程，而这样做的损耗会比较大（比如重头开始创建一个线程，执行完毕以后再销毁等。如果线程的实际工作内容，就是run方法里面只是简单的打印一行字的话，那么可能线程的实际工作内容还不如损耗来的大）。如果使用Runnable和线程池，就可以大大减少这样的损耗。
3. 继承Thread类后，由于Java语言不支持双继承，这样就无法再继承其他的类，限制了可扩展性。
```

3. 两种方法的本质对比

方法一和方法二，也就是“实现Runnable接口并实现run方法”和“继承Thread类然后重写run方法”在实现多线程的本质上并没有区别，最终都是调用了start()方法来启动线程。这两个方法的最主要区别在于run方法的内容来源：

- 方法一：最终调用target.run()
- 方法二：run()整个都被重写了

4. 同时用两种方法会怎么样?

实现了Runnable的run方法，最终会被Thread override的run覆盖，所以走Thread的逻辑

5. 总结

准确的说，创建线程只有1种方式，那就是构造Thread类，而实现线程的执行单元有2种方式：

- 方法一：实现Runnable接口，重写run方法，并把Runnable实例传给Thread类
- 方法二：重写Thread的run方法（继承Thread类）

6. 典型错误

1. 线程池创建线程也算是一种新建线程的方式（本质也是通过Thread的方式）

2. 通过Callable和FutureTask创建线程，也算是一种新建线程的方式（本质实现了Runnable接口）
3. “无返回值”是实现Runnable接口，“有返回值”是实现callable接口，所以callable是新的实现线程的方式（同上，本质都是实现了Runnable）
4. 定时器（TimerTask implements Runnable）
5. 匿名内部类和Lambda表达式

总结：多线程的实现方式，在代码中写法千变万化，但其本质万变不离其宗。就是通过2种方式：继承Thread类；实现Runnable接口

2. 怎样才是正确的线程启动方式?

- start()和run()的比较：runnable.run()方法是由main线程执行的，而要子线程执行就一定要先调用start()启动新线程去执行。
- start()方法含义：启动新线程、准备工作、不能重复调用start(重复会抛出非法的线程异常)
- start()源码解析：启动新线程检查线程状态 --> 加入线程组 --> 调用start0
- run()方法原理解读：直接调用runnable对象run方法本质跟普通方法一样的执行方式(主线程)，通过start()来间接地调用run()子线程运行。
- 启动线程——常见面试问题

```
1. 一个线程两次调用start()方法会出现什么情况？为什么？
    情况：抛出一个IllegalThreadStateException异常。
    原因：Thread的start方法一开始会对线程状态threadStatus进行判断。线程未启动时，threadStatus=0，当线程执行了start后，threadStatus就被改变了，第二次再执行start方法的时候，start方法检测到threadStatus状态不对，就抛出了异常。
2. 既然start()方法会调用run()方法，为什么我们选择调用start()方法，而不是直接调用run()方法呢？
    原因：start方法才是真正意义上的启动一个线程，它会去经历线程的各个生命周期。而如果直接用run方法，等效就只是执行一个普通方法，也不会用子线程去调用
```

3. 线程停止、中断之最佳实践

原理介绍

- 使用interrupt来通知，而不是强制：interrupt是中断，A线程通知B线程去中断，而B线程是具有主动权的。B线程何时停止是B线程自己决定的，可能根据当前业务逻辑完成情况，所以说是通知，而不是强制

Java中停止线程的原则是什么

- 在Java中，最好的停止线程的方式是使用中断interrupt，但是`这仅仅是会通知到被终止的线程“你该停止运行了”，被终止的线程自身拥有决定权（决定是否、以及何时停止）`，这依赖于请求停止方和被停

止方都遵守一种约定好的编码规范。

- 任务和线程的启动很容易。在大多数时候，我们都会让它们运行直到结束，或者让它们自行停止。然而有时候我们希望提前结束任务或线程或许是因为用户取消了操作或者服务需要被快速关闭，或者是运行超时或出错了。
- 要使任务和线程能安全、快速、可靠地停止下来，并不是一件容易的事。Java没有提供任何机制来安全地终止线程。但它提供了中断（Interruption），这是一种协作机制，能够使一个线程终止另一个线程的当前工作。
- 这种协作式的方法是必要的，我们很少希望某个任务、线程或服务立即停止，因为这种立即停止会使共享的数据结构处于不一致的状态。相反，在编写任务和服务时可以使用种协作的方式当需要停止时它们首先会清除当前正在执行的工作然后再结束。这提供了更好的灵活性，因为任务本身的代码比发出取消请求的代码更清楚如何执行清除工作。
- 生命周期结束(End- of-Lifecycle)的问题会使任务、服务以及程序的设计和实现等过程变得复杂，而这个在程序设计中非常重要的要素却经常被忽略。一个在行为良好的软件与勉强运的软件之间的最主要区别就是，行为良好的软件能很完善地处理失败、关闭和取消等过程。
- 接下来将给出各种实现取消和中断的机制，以及如何编写任务和服务，使它们能对取消请求做出响应。

正确的停止方法：interrupt

我们分三种情况讨论interrupt如何停止线程。

- 通常线程会在什么情况下停止普通情况？run方法内没有sleep或wait方法时，停止线程。
- 线程被阻塞的情况（sleep），while循环后sleep(1000)，整个while和sleep都try/catch。**可停止，抛异常**
- 如果线程在每次while迭代(for/while循环体内)都sleep(10)，整个while都try/catch。**可停止，抛异常**
- while内sleep(10)，并且每次循环都进行是否中断判断，只在while内的sleep(10)外try/catch的问题。**不可停止，中途抛异常。**为什么加了Thread.currentThread().isInterrupted()判断后，线程没有停止，仍然继续执行？：其实是因为sleep响应中断后，会把interrupt状态清除，所以中断信号无效。
- **实际开发中的2种最佳实践**

  a. 优先选择：传递中断（将中断向上抛出，由顶层run方法来处理中断）

```
1  public class RightWayStopThreadInProd implements Runnable {
2      @Override
3      public void run() {
4          while (true && !Thread.currentThread().isInterrupted()) {
5              System.out.println("go");
6              try {
7                  throwInMethod();
8              } catch (InterruptedException e) {
9                  System.out.println("保存日志逻辑");
10                 e.printStackTrace();
11             }
12         }
```

```
    }

    private void throwInMethod() throws InterruptedException {
        //在方法签名中抛出异常
        Thread.sleep(2000);
    }

    public static void main(String[] args) throws InterruptedExce
ption {
        Thread thread = new Thread(new RightWayStopThreadInProd
());
        thread.start();
        Thread.sleep(1000);
        thread.interrupt();
    }
}
//输出结果
go
保存日志逻辑
go //中断后，interrupt标志位被清除，所以会不断循环打印go
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at ConcurrenceFolder.mooc.threadConcurrencyCore.stopthreads.R
ightWayStopThreadInProd.throwInMethod(RightWayStopThreadInProd.ja
va:26)
    at ConcurrenceFolder.mooc.threadConcurrencyCore.stopthreads.R
ightWayStopThreadInProd.run(RightWayStopThreadInProd.java:17)
    at java.lang.Thread.run(Thread.java:748)
go
```

b. 不想或无法传递：恢复中断

```
public class RightWayStopThreadInProd2 implements Runnable {
    @Override
    public void run() {
        while (true) {
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Interrupted，程序运行结束");
                break;
            }
```

```
            reInterrupt();
        }
    }

    private void reInterrupt() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            //在这里恢复中断状态
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws InterruptedExce
ption {
        Thread thread = new Thread(new RightWayStopThreadInProd2
());
        thread.start();
        Thread.sleep(1000);
        thread.interrupt();
    }
}
//输出结果
java.lang.InterruptedException: sleep interrupted
Interrupted, 程序运行结束
    at java.lang.Thread.sleep(Native Method)
    at ConcurrenceFolder.mooc.threadConcurrencyCore.stopthreads.R
ightWayStopThreadInProd2.reInterrupt(RightWayStopThreadInProd2.ja
va:26)
    at ConcurrenceFolder.mooc.threadConcurrencyCore.stopthreads.R
ightWayStopThreadInProd2.run(RightWayStopThreadInProd2.java:20)
    at java.lang.Thread.run(Thread.java:748)
```

- 响应中断的方法总结列表
  - Object.wait()/wait(long)/wait(long,int)
  - Thread.sleep(long)/Thread.sleep(long,int)
  - Thread.join()/join(long)/join(long,int)
  - java.util.concurrent.BlockingQueue.take()/put(E)

- java.util.concurrent.locks.Lock.lockInterrruptibly()
- java.util.concurrent.CountDownLatch.await()
- java.util.concurrent.CyclicBarrier.await()
- java.util.concurrent.Exchanger.exchange(V)
- java.nio.channels.InterruptibleChannel相关方法
- java.nio.channels.Selector的相关方法

停止线程的错误方法

被弃用的stop、suspend和resume方法

- 用stop()来停止线程，会导致线程运行一半突然停止：用stop()来停止线程，会导致线程运行一半突然停止，没办法完成一个基本单位的操作（一个连队），会造成脏数据（有的连队多领取少领取装备）
- suspend将线程挂起，运行->阻塞；调用后并不释放所占用的锁（不释放锁，可能会导致死锁）
- resume将线程解挂，阻塞->就绪（不释放锁，可能会导致死锁）

用volatile设置boolean标记位

```
// 示例1：可行
public class WrongWayVolatile implements Runnable {
    private volatile boolean canceled = false;

    @Override
    public void run() {
        int num = 0;
        try {
            while (num < 10000 && !canceled) {
                if (num % 100 == 0) {
                    System.out.println(num + "是100的倍数");
                }
                num++;
                Thread.sleep(1);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws InterruptedExc
eption {
        WrongWayVolatile r = new WrongWayVolatile();
```

```
        Thread thread = new Thread(r);
        thread.start();
        Thread.sleep(5000);
        r.canceled = true;

    }
}
//输出结果
0是100的倍数
100是100的倍数
200是100的倍数
...

-----------------------------------------------------------------------

// 示例2：不可行(陷入阻塞时，volatile是无法停止线程的)
// 此例中，生产者的生产速度很快，消费者消费速度慢，所以阻塞队列满了以后，生产者会阻塞，等待消费者进一步消费
// 错误原因：线程阻塞在storage.put(num)，无法继续执行
public class WrongWayVolatileCantStop {
    public static void main(String[] args) throws InterruptedException {
        //生产者产生数据
        ArrayBlockingQueue storage = new ArrayBlockingQueue(10);
        Producer producer = new Producer(storage);
        Thread producerThread = new Thread(producer);
        producerThread.start();
        Thread.sleep(1000);

        //消费者消费数据
        Consumer consumer = new Consumer(storage);
        while (consumer.needMoreNums()) {
            System.out.println(consumer.storage.take() + "被消费了");
            Thread.sleep(100);
        }
        System.out.println("消费者不需要更多数据了。");

        //一旦消费不需要更多数据了，我们应该让生产者也停下来，但实际情况是没
```

```
有停下来
59          producer.canceled = true;
60      }
61  }
62
63  /**
64   * 生产者
65   */
66  class Producer implements Runnable {
67      public volatile boolean canceled = false;
68      BlockingQueue storage;
69
70      public Producer(BlockingQueue storage) {
71          this.storage = storage;
72      }
73
74      @Override
75      public void run() {
76          int num = 0;
77          try {
78              while (num < 10000 && !canceled) {
79                  if (num % 100 == 0) {
80                      storage.put(num);
81                      System.out.println(num + "是100的倍数，被放到仓
库中了。");
82                  }
83                  num++;
84              }
85          } catch (InterruptedException e) {
86              e.printStackTrace();
87          } finally {
88              System.out.println("生产者结束运行");
89          }
90      }
91  }
92
93  /**
94   * 消费者
95   */
96  class Consumer {
```

```
    BlockingQueue storage;

    public Consumer(BlockingQueue storage) {
        this.storage = storage;
    }

    public boolean needMoreNums() {
        return Math.random() > 0.95 ? false : true;
    }
}

//输出结果
0是100的倍数，被放到仓库中了。
100是100的倍数，被放到仓库中了。
200是100的倍数，被放到仓库中了。
300是100的倍数，被放到仓库中了。
400是100的倍数，被放到仓库中了。
500是100的倍数，被放到仓库中了。
600是100的倍数，被放到仓库中了。
700是100的倍数，被放到仓库中了。
800是100的倍数，被放到仓库中了。
900是100的倍数，被放到仓库中了。
1000是100的倍数，被放到仓库中了。
0被消费了
100被消费了
1100是100的倍数，被放到仓库中了。
200被消费了
1200是100的倍数，被放到仓库中了。
1300是100的倍数，被放到仓库中了。
300被消费了
1400是100的倍数，被放到仓库中了。
400被消费了
1500是100的倍数，被放到仓库中了。
500被消费了
600被消费了
1600是100的倍数，被放到仓库中了。
700被消费了
1700是100的倍数，被放到仓库中了。
1800是100的倍数，被放到仓库中了。
800被消费了
```

```
900被消费了
1900是100的倍数，被放到仓库中了。
1000被消费了
2000是100的倍数，被放到仓库中了。
消费者不需要更多数据了。
//线程没有停止

------------------------------------------------------------------
------------------

// 示例三：修正方案
ublic class WrongWayVolatileFixed {
    public static void main(String[] args) throws InterruptedExc
eption {
        WrongWayVolatileFixed body = new WrongWayVolatileFixed
();
        //生产者产生数据
        ArrayBlockingQueue storage = new ArrayBlockingQueue(10);
        Producer producer = body.new Producer(storage);
        Thread producerThread = new Thread(producer);
        producerThread.start();
        Thread.sleep(1000);

        //消费者消费数据
        Consumer consumer = body.new Consumer(storage);
        while (consumer.needMoreNums()) {
            System.out.println(consumer.storage.take() + "被消费
了");
            Thread.sleep(100);
        }
        System.out.println("消费者不需要更多数据了。");
        producerThread.interrupt();
    }

    /**
     * 生产者
     */
    class Producer implements Runnable {
        BlockingQueue storage;

```

```
173         public Producer(BlockingQueue storage) {
174             this.storage = storage;
175         }
176 
177         @Override
178         public void run() {
179             int num = 0;
180             try {
181                 while (num < 10000 && !Thread.currentThread().is
    Interrupted()) {
182                     if (num % 100 == 0) {
183                         storage.put(num);
184                         System.out.println(num + "是100的倍数，被放
    到仓库中了。");
185                     }
186                     num++;
187                 }
188             } catch (InterruptedException e) {
189                 e.printStackTrace();
190             } finally {
191                 System.out.println("生产者结束运行");
192             }
193         }
194     }
195 
196     /**
197      * 消费者
198      */
199     class Consumer {
200         BlockingQueue storage;
201 
202         public Consumer(BlockingQueue storage) {
203             this.storage = storage;
204         }
205 
206         public boolean needMoreNums() {
207             return Math.random() > 0.95 ? false : true;
208         }
209     }
210 }
```

```
//输出结果
...
3800是100的倍数，被放到仓库中了。
2900被消费了
3900是100的倍数，被放到仓库中了。
4000是100的倍数，被放到仓库中了。
3000被消费了
java.lang.InterruptedException
消费者不需要更多数据了。
生产者结束运行
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.reportInterruptAfterWait(AbstractQueuedSynchronizer.java:2014)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2048)
    at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:353)
    at ConcurrenceFolder.mooc.threadConcurrencyCore.stopthreads.volatiledemo.WrongWayVolatileFixed$Producer.run(WrongWayVolatileFixed.java:51)
    at java.lang.Thread.run(Thread.java:748)
```

重要函数的源码解析

判断是否已被中断相关方法

- static boolean interrupted()：判断是否中断，同时清除中断状态

```
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}

private native boolean isInterrupted(boolean ClearInterrupted);
```

- boolean isInterrupted()
- Thread.interrupted() 的 目的对象：静态interrupted只跟当前线程有关，与对象无关

停止线程——常见面试问题

如何停止线程

- 1、原理：用interrupt来请求，好处是可以保证数据安全，应该把主动权交给被中断的线程

- 2、想停止线程，要请求方、被停止方（在线程run中检测interrupt状态）、子方法被调用方（内部调用方法将中断向上抛出，由顶层run来做处理，而不是吞掉中断异常）相互配合
- 3、最后再说错误的方法：stop/suspend已废弃（不释放锁，可能导致死锁），volatile的boolean无法处理长时间阻塞的情况

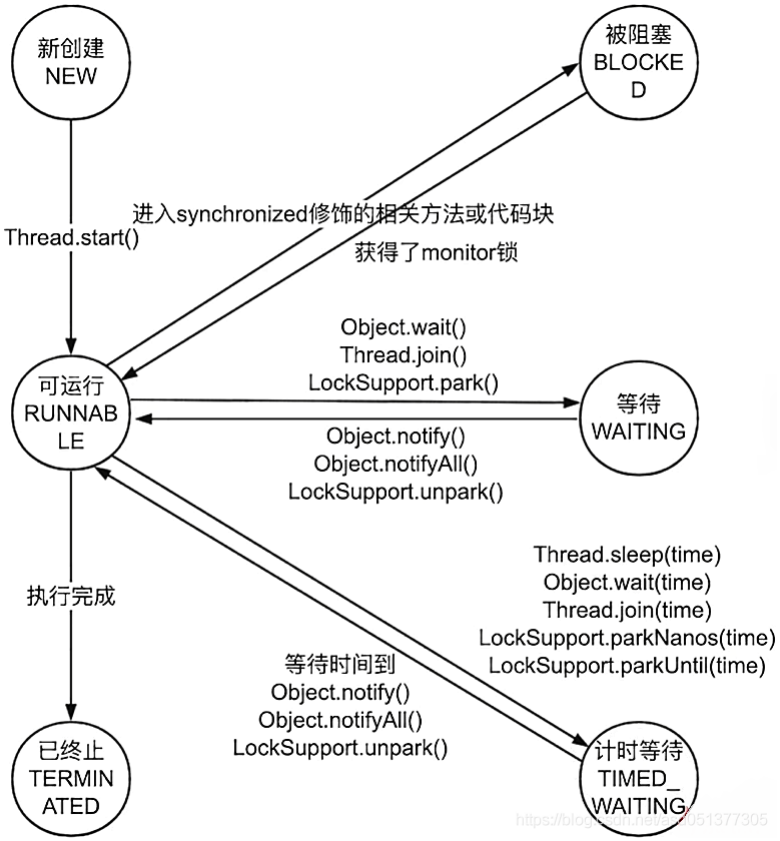
如何处理不可中断的阻塞

- 针对特定的场景，用特定的方法来处理

4. 图解线程生命周期

6种状态状态是什么含义?

- New：新建还未执行（start()）
- Runnable(可运行的)：调用了start方法后，就会变为Runnable状态
- Blocked：进入synchronized修饰的区域，同时锁被其他线程拿走
- Waiting：只能手工唤醒
- Timed Waiting：计时等待。等到固定time时间后，就可以被唤醒；或者通过手工唤醒2种方式都可以
- Terminated：程序正常执行完毕；或者出现没有被捕获的异常，中止了run方法



5. Thread和Object类中线程相关方法(wait/notify、sleep、join、yield)

1、方法概览

类	方法名	简介
Thread	sleep相关	本表格的“相关”，指的是重载方法，如sleep有多个重载方法，但实际作用大同小异
.	join	主线程等待ThreaA执行完毕（ThreadA.join()）。 主要是让调用该方法的thread完成run方法里面的东西后，再执行join()方法后面的代码。
.	yield相关	放弃已经获取到的CPU资源
.	currentThread	获取当前执行线程的引用
.	start，run相关	启动线程相关
.	interrupt相关	中断线程
.	stop(),suspend(),resuem()相关	已废弃
Object	wait/notify/notifyAll相关	让线程暂时休息和唤醒

2、wait、notify、notifyAll方法详解

1. 作用、用法：阻塞阶段、唤醒阶段、遇到中断
    - 阻塞阶段：线程调用wait()方法，则该线程进入到阻塞状态，直到以下4种情况之一发生时，才会被唤醒
        - 另一个线程调用这个对象的notify()方法且刚好被唤醒的是本线程
        - 另一个线程调用这个对象的notifyAll()方法且刚好被唤醒的是本线程
        - 过了wait(long timeout)规定的超时时间，如果传入0就是永久等待
        - 线程自身调用了interrupt
    - 唤醒阶段
        - notify会唤起单个在等待某对象monitor的线程，如果有多个线程在等待，则只会唤起其中随机的一个
        - notifyAll会将所有等待的线程都唤起，而唤起后具体哪个线程会获得monitor，则看操作系统的调度
        - notify必须在synchronized中调用，否则会抛出 IllegalMonitorStateException 异常
    - 遇到中断
        - 假设线程执行了wait()，在此期间被中断，则会抛出interruptException，同时释放已经获取到的monitor
2. 特点、性质
    - 使用的时候必须先拥有monitor(synchronized锁)
    - notify只能唤醒其中一个

- 属于Object类

3. 常见面试问题
   - 两个线程交替打印0~100的奇偶数
     - 基本方式：用synchronized关键字实现

```
要点：
    创建两个线程，一个线程处理偶数，一个线程处理奇数
    两个线程之间通过synchronized进行同步，保证count++每次只有一个线程进行操作
    为什么两个线程能交替执行，这里很巧的是count++ 从0~100自增过程就是一个奇偶数交替的过程，
  实际上两个线程都是在不停的尝试（while循环）进入synchronized代码块，如果满足相对应的条件（偶数或是奇数）就打印输出。
```

     - 更好的方法：wait/notify
   - 为什么wait()需要在同步代码块内使用，而sleep()不需要
     - 正常逻辑是先执行wait，后续在执行notify唤醒。如果wait/notify不放同步代码块，执行wait的时候，线程切换去执行其他任务如notify，导致notify先于wait，就会导致后续切回wait的时候，一直阻塞着，无法释放，导致死锁。
     - 而sleep是针对本身的当前线程的，不影响
   - 为什么线程通信的方法wait()，notify()和notifyAll被定义在Object类里？而sleep定义在Thread类里？
     - wait、notify、notifyAll是锁级别的操作，属于Object对象的，而线程实际上是可以持有多把锁的，如果把wait定义到Thread里面，就无法做到这么灵活的控制了
   - wait方法是属于Object对象的，那调用Thread.wait会怎么样？
     - Thread线程退出的时候，会自动调用notify，这可能不是我们所期望的，所以最好不要用Thread.wait
   - 如何选择notify还是notifyAll？
     - notify是唤起一个线程，选择哪个是随机的。而notifyAll是唤起所有线程，然后这些线程再次抢去夺锁
   - notifyAll之后所有的线程都会再次抢夺锁，如果某线程抢夺失败怎么办？
     - 实质就跟初始状态一样，多个线程抢夺锁，抢不到的线程就等待，等待上一个线程释放锁
   - 用suspend()和resume()来阻塞线程可以吗？为什么？
     - 这2个方法由于不安全，已经被弃用了。最好还是使用wait和notify

3、sleep方法详解

1. 作用：我只想让线程在预期的时间执行，其他时候不要占用CPU资源
2. 不释放锁
   - 包括synchronized和lock
   - 和wait不同
3. sleep方法响应中断

- 抛出InterruptedException
- 清除中断状态

4. sleep总结
   - sleep方法可以让线程进入Waiting状态，并且不占用CPU资源
   - 但是不释放锁，直到规定时间后再执行
   - 休眠期间如果被中断，会抛出异常并清除中断状态
5. sleep常见面试问题
   - wait/notify、sleep异同（方法属于哪个对象？线程状态怎么切换？）
     - 相同：都会阻塞，都可以响应中断
     - 不同
       - wait/notify需要在synchronized方法中，而sleep不需要
       - 释放锁：wait会释放锁，而sleep不释放锁
       - 指定时间：sleep必须传参时间，而wait有多个构造方法，不传时间则直到自己被唤醒
       - 所属类：wait/notify是Object方法，sleep是Thread类的方法

4、join方法

1. 作用：因为新的线程加入了“我们”，所以“我们”要等他执行完再出发
2. 用法：（在main方法中thread1.join）main等待thread1执行完毕，注意谁等谁（父等待子）
3. 可以使用封装工具类：CountDownLatch或CyclicBarrier
4. join原理
   - 源码

```
(1)thread.join();
(2)public final void join() throws InterruptedException {
      join(0);
}
(3)public final synchronized void join(long millis)throws Interru
ptedException {
      ...
      if (millis == 0) {
          while (isAlive()) {
              wait(0);
          }
      }
  }

 分析：线程在run执行完成后，JVM底层会自动调用一个notifyAll唤醒，所以即使在jo
in()内没有notify显示调用，执行完run()后，也会唤醒

等价
```

```
// thread.join();   等价于下面synchronized的代码
synchronized (thread) {
    thread.wait();
}
```

5. 常见面试问题
    - 在join期间，线程处于哪种线程状态？Waiting

5、yield方法

- 作用：释放我的CPU时间片。线程状态仍然是RUNNABLE，不释放锁，也不阻塞
- 定位：JVM不保证遵循yield逻辑
- yield和sleep区别：yield随时可能再次被调度

6、获取当前执行线程的引用：Thread.currentThread()方法

- 同一个方法，不同线程会打印出各自线程的名称

6. 线程各属性

1、线程各属性纵览

属性名称	用户
编号(ID)	每个线程有自己的ID，用于标识不同的线程。 getId内部调用是nextThreadID --> ++threadSeqNumber
名称(Name)	作用让用户或程序员在开发、调试或运行过程中，更容易区分每个不同的线程、定位问题等。 "Thread-" + nextThreadNum()
是否是守护线程(isDaemon)	true代表该线程是【守护线程】，false代表线程是非守护线程，也就是【用户线程】
优先级(Priority)	优先级这个属性的目的是告诉线程调度器，用户希望哪些线程相对多运行、哪些少运行

2. 守护线程

1. 作用：给用户线程提供服务
2. 三个特性
    - 线程类型默认继承自父线程（守护线程的子线程也是守护线程）
    - 通常守护线程都是由JVM自动启动的
    - 不影响JVM退出：JVM退出只会考虑是否还有用户线程
3. 守护线程的常见面试问题
    - 守护线程和普通线程的区别
        - 整体无区别

        - 唯一区别在于JVM的离开：用户线程会影响JVM的停止，而守护线程不影响
        - 作用不同：用户线程是执行逻辑的，而守护线程是给用户线程提供服务的
    - 我们是否需要给线程设置为守护线程？thread.setDaemon(true);
        - 不应该把自己的用户线程设置为守护线程。
        - 例如：如果设置了用户线程为守护线程，JVM发现只有一个守护线程，就中止退出了，导致程序逻辑没有走完。
        - 其实JVM本身提供的守护线程就已经足够了

4、线程优先级

10个级别，默认5

- 引申面试题：我们应该如何应用线程优先级来帮助程序运行？有哪些禁忌？
- 不同的操作系统如何处理优先级问题？

程序设计不应依赖于优先级

- 不同操作系统不一样
- 优先级会被操作系统改变

5、各属性总结

属性名称	用途	注意事项
编号(ID)	标识不同的线程	线程回收后，id被后续创建的线程使用；无法保证id的唯一性（之前线程id，跟后续线程id不一定是同一个线程，可能是回收后后续创建的）；不允许修改id
名称(Name)	定位问题	可以设置一个清晰有意义的名字（方便跟踪定位）；默认的名称是Thread-0/1/2/3
是否是守护线程(isDaemon)	守护线程/用户线程	二选一；继承父线程；setDaemon
优先级(Priority)	告诉线程调度器，哪些线程相对多运行、哪些少运行	默认和父线程的优先级相等，共有10个等级，默认5；不应依赖优先级

7. 线程异常处理（全局异常处理UncaughtExceptionHandler）

- 线程的未捕获异常UncaughtException应该如何处理？

1、为什么需要UncaughtExceptionHandler?

- 主线程可以轻松发现异常，子线程却不行
- 子线程异常无法用传统方法(try-catch)捕获（类似main方法中执行thread.start,抛出异常是在子线程的run中，而try-catch的是主线程main，所以捕获不到）

- 不能直接捕获会导致一些后果（无法捕获到异常，做相应的重试操作逻辑）

2、两种解决方案

方案一（不推荐）：手动在每个run方法里进行try catch

方案二（推荐）：利用UncaughtExceptionHandler

- UncaughtExceptionHandler接口
- void uncaughtException(Thread t, Throwable e);

```
// Thread.java
        @FunctionalInterface
    public interface UncaughtExceptionHandler {
        /**
         * Method invoked when the given thread terminates due to the
         * given uncaught exception.
         * <p>Any exception thrown by this method will be ignored by the
         * Java Virtual Machine.
         * @param t the thread
         * @param e the exception
         */
        void uncaughtException(Thread t, Throwable e);
    }
```

- 异常处理器的调用策略
- 自己实现
  - 给程序统一设置
  - 给每个线程单独设置
  - 给线程池设置

```
// 1、自己的UncaughtExceptionHandler
public class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler {
    private String name;

    public MyUncaughtExceptionHandler(String name) {
        this.name = name;
    }

    @Override
    public void uncaughtException(Thread t, Throwable e) {
```

```
        Logger logger = Logger.getAnonymousLogger();
        logger.log(Level.WARNING, "线程异常，终止了" + t.getName(), e);
        System.out.println(name + "捕获了异常" + t.getName() + "异常" + e);
    }
}

// 2、使用自己的UncaughtExceptionHandler，触发
public class UseOwnUncaughtExceptionHandler implements Runnable {
    public static void main(String[] args) throws InterruptedException {
        Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandler("捕获器1"));
        new Thread(new UseOwnUncaughtExceptionHandler(), "MyThread-1").start();
        Thread.sleep(300);
        new Thread(new UseOwnUncaughtExceptionHandler(), "MyThread-2").start();
        Thread.sleep(300);
        new Thread(new UseOwnUncaughtExceptionHandler(), "MyThread-3").start();
        Thread.sleep(300);
        new Thread(new UseOwnUncaughtExceptionHandler(), "MyThread-4").start();
    }
    @Override
    public void run() {
        throw new RuntimeException();
    }
}
```

3、线程的未捕获异常-常见面试问题

1. 为什么要全局处理？如何全局处理异常？不处理行不行？
    - 为什么要全局处理：统一处理，方便，不用每个地方都处理
    - 如何全局处理：定义一个自己的UncaughtExceptionHandler，然后Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandler("捕获器1"));
    - 必须得处理

2. run方法是否可以抛出异常？如果抛出异常，线程的状态会怎么样？
    - run没有声明throws exception，所以只能try-catch
    - 如果在run中try-catch没有捕获，漏了，则会抛出异常，线程会中止运行，打印出堆栈
3. 线程中如何处理某个未处理异常？
    - 使用全局处理ExceptionHandler

8. 线程安全-多线程会导致的问题

1、线程安全

1.1 什么是线程安全

- 当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的————《Java并发编程实战》

1.2 线程不安全：get同时set

- 全都线程安全？：运行速度、设计成本、trade off
- 完全不用于多线程的代码：不过度设计

1.3 什么情况下会出现线程安全问题，怎么避免？

1.3.1 运行结果错误：a++多线程下出现消失的请求现象

1.3.2 活跃性问题：死锁、活锁、饥饿

1.3.3 对象发布和初始化时的安全问题

什么是发布？

- public、return都算是获得对象，发布了该对象出去

什么是溢出？

- 1.方法返回一个private对象（定义了private对象的getXX()方法）（private的本意是不让外部访问）
- 2.还未完成初始化（构造函数没完全执行完毕）就把对象提供给外界，比如

（1）在构造函数中未初始化完毕就把this赋值出去了

（2）隐式逸出————注册监听事件

```
public class MultiThreadsError {
    int count;

    public MultiThreadsError(MySource source) {
        source.registerListener(new EventListener() {
            @Override
            //这里EventListener是一个匿名内部类，实际上也用了count这个外部引用变量，当count未初始化完成，拿到的值就还是0
            public void onEvent(Event e) {
                System.out.println("\n我得到的数字是：" + count);
            }
        });
```

```
        for (int i = 0; i < 10000; i++) {
            System.out.print(i);
        }
        count = 100;
    }

    public static void main(String[] args) {
        MySource mySource = new MySource();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                mySource.eventCome(new Event() {
                });
            }
        }).start();
        MultiThreadsError multiThreadsError = new MultiThreadsErr
or(mySource);
    }

    static class MySource {
        private EventListener listener;

        void registerListener(EventListener eventListener) {
            this.listener = eventListener;
        }

        void eventCome(Event e) {
            if (listener != null) {
                listener.onEvent(e);
            } else {
                System.out.println("还未初始化完毕");
            }
        }
    }

```

```
    interface EventListener {
        void onEvent(Event e);
    }

    interface Event {

    }
}
//输出结果
012345678910...
我得到的数字是：0
```

（3）构造函数中运行线程

1.4 如何解决逸出

- 返回“副本”（返回对象的deepCopy）--对应解决（1.方法返回了private对象）
- 工厂模式--对应解决（2.还没初始化就吧对象提供给外界）

2、各种需要考虑线程安全的情况

- 访问共享的变量或资源，会有并发风险，比如对象的属性、静态变量、共享缓存、数据库等
- 所有依赖时序的操作，即使每一步操作都是线程安全的，还是存在并发问题：read-modify-write、check-then-act（a++问题）
- 不同的数据之间存在捆绑关系的时候（原子操作：要么全部执行，要么全部不执行）
- 我们使用其他类的时候，如果对方没有声明自己是线程安全的，则我们需要做相应的处理逻辑

3、双刃剑：多线程会导致的问题

3.1 性能问题有哪些体现、什么是性能问题

- 服务响应慢、吞吐量低、资源消耗（例如内存）过高等
- 虽然不是结果错误，但仍然危害巨大
- 引入多线程不能本末倒置

3.2 为什么多线程会带来性能问题

（1）调度：上下文切换

- 什么是上下文？：线程A执行到某个地方，然后要切换到另一个线程B的时候，CPU会保存当前的线程A在CPU中的状态（上下文）到内存中的某处，等线程B执行完成后，回到线程A需要还原线程A之前保存的状态（这种切换需要耗时）
- 缓存开销（考虑缓存失效）：多线程切换，从线程A切换到线程B，线程A的缓存就失效了，需要重新加载
- 何时会导致密集的上下文切换：抢锁、IO

（2）协作：内存同步

- 为了数据的正确性，同步手段往往会使用禁止编译器优化、使CPU内的缓存失效（java内存模型）

4、常见面试问题

（1）你知道有哪些线程不安全的情况

- 运行结果错误：a++多线程下出现消失的请求现象
- 活跃性问题：死锁、活锁、饥饿
- 对象发布和初始化时的安全问题

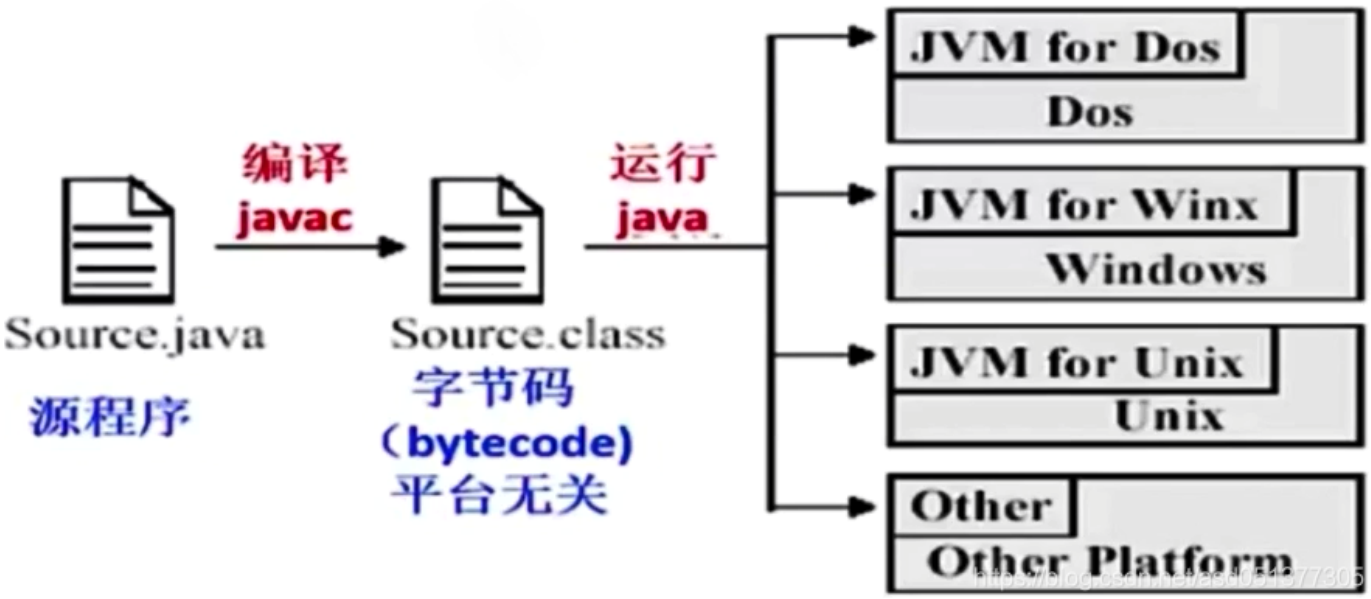
（2）平时哪些情况下需要额外注意线程安全问题?

（3）什么是多线程的上下文切换?

9. Java内存模型JMM——底层原理

1、到底什么叫“底层原理”？本章研究的内容是什么?

1.1 从Java代码到CPU指令




```
1. ①最开始，我们编写的Java代码，是*.java文件
2. ②在编译（javac命令）后，从刚才的*.java文件会变出一个新的Java字节码文件（*.class）
3. ③JVM会执行刚才生成的字节码文件（*.class），并把字节码文件转化为机器指令
4. ④机器指令可以直接在CPU上运行，也就是最终的程序执行
```

1.2 JVM实现会带来不同的“翻译”，不同的CPU平台的机器指令又千差万别，无法保证并发安全的效果一致

1.3 因此引入内存模型：转换过程的规范、原则

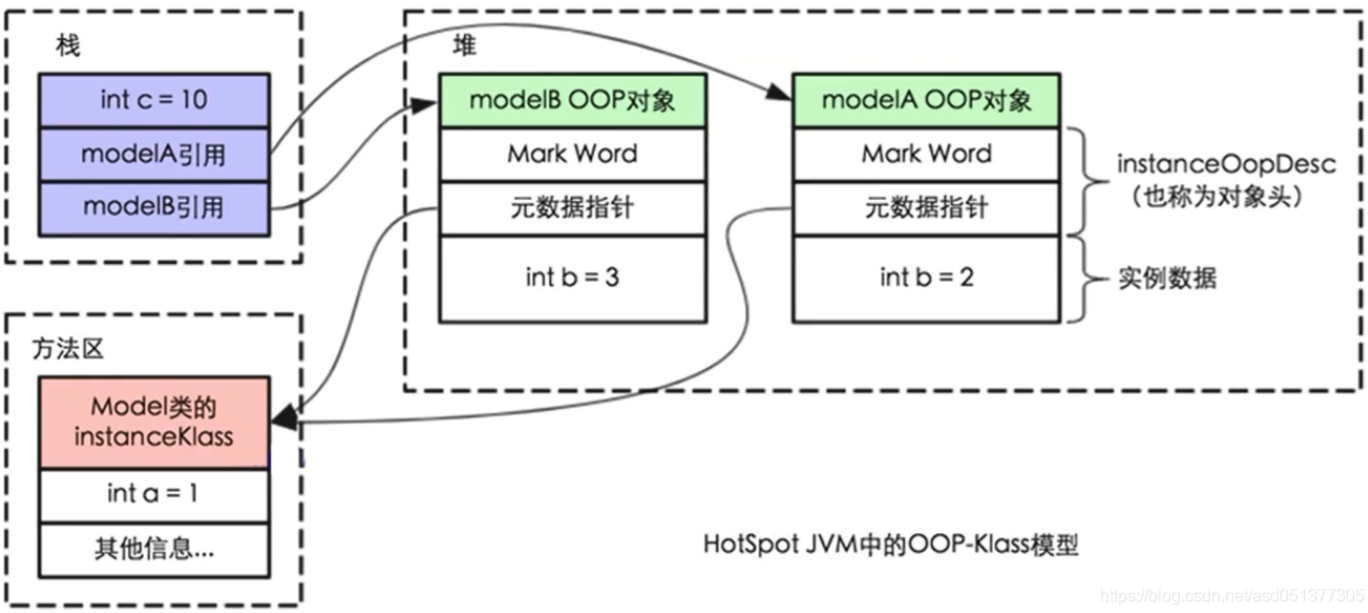
2、三兄弟：JVM内存结构 VS Java内存模型 VS Java对象模型

整体方向：

- **JVM内存结构，和Java虚拟机的运行时区域有关。如堆和栈**
- **Java内存模型，和Java的并发编程有关**
- **Java对象模型，和Java对象在虚拟机中的表现形式有关**

- 堆（heap）：是运行时数据区中占用最大的。存储对象的实例
- 虚拟机栈/Java栈（VM stack）：保存各个基本类型、对象引用
- 方法区（method）：存放static的静态变量/类/常量，以及永久引用
- 本地方法栈：存放与本地方法(native)相关的
- 程序计数器

2.2 Java对象模型



HotSpot JVM中的OOP-Klass模型

- Java对象自身的存储模型
- JVM会给这个类创建一个instanceKlass保存在方法区，用来在JVM层表示该Java类。
- 当我们在Java代码中，使用new创建一个对象的时候，JVM会创建一个instanceOopDesc对象，这个对象中包含了对象头以及实例数据。

3、JMM是什么

3.1 为什么需要JMM

- C语言不存在内存模型的概念
- 依赖处理器，不同处理器结果不一样
- 无法保证并发安全
- 需要一个标准，让多线程运行的结果可预期

3.2 JMM是规范

- Java Memory Model
- JMM是一组规范，需要各个JVM的实现来遵守JMM规范，以便于开发者可以利用这些规范，更方便地开发多线程程序
- 如果没有这样的一个JMM内存模型来规范，那么很可能经过了不同JVM的不同规则的重排序之后，导致不同的虚拟机上运行的结果不一样，那是很大的问题

3.3 JMM是工具类和关键字的原理

- volatile、synchronized、Lock等的原理都是JMM
- 如果没有JMM，那就需要我们自己指定什么时候用内存栅栏等，那是相当麻烦的，幸好有了JMM，让我们只需要用同步工具和关键字就可以开发并发程序

3.4 最重要的3点内容：重排序、可见性、原子性为什么需要JMM

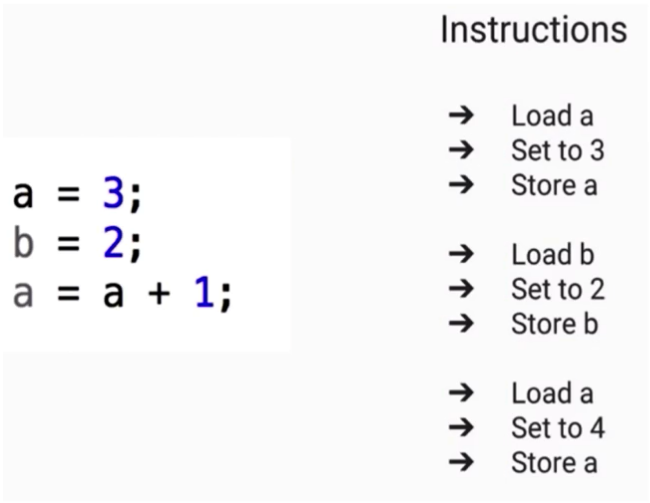
4、重排序

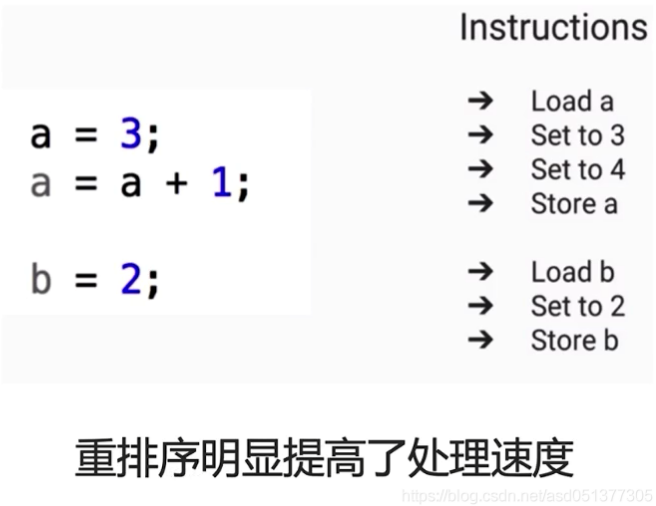
4.1 什么是重排序：

在线程内部的两行代码的【实际执行顺序】和代码在【Java文件中的顺序】不一致，代码指令并不是严格按照代码语句顺序执行的，它们的顺序被改变了，这就是重排序。

4.2 重排序的好处：提高处理速度

- 对比重排序前后的指令优化





4.3 重排序的3种情况：编译器优化、CPU指令重排、内存的“重排序”

- 编译器优化：包括JVM，JIT编译器等
- CPU指令重排：就算编译器不发生重排，CPU也可能对指令进行重排
- 内存的“重排序”：线程A的修改线程B却看不到，引出可见性问题

5、可见性

5.1 可见性问题

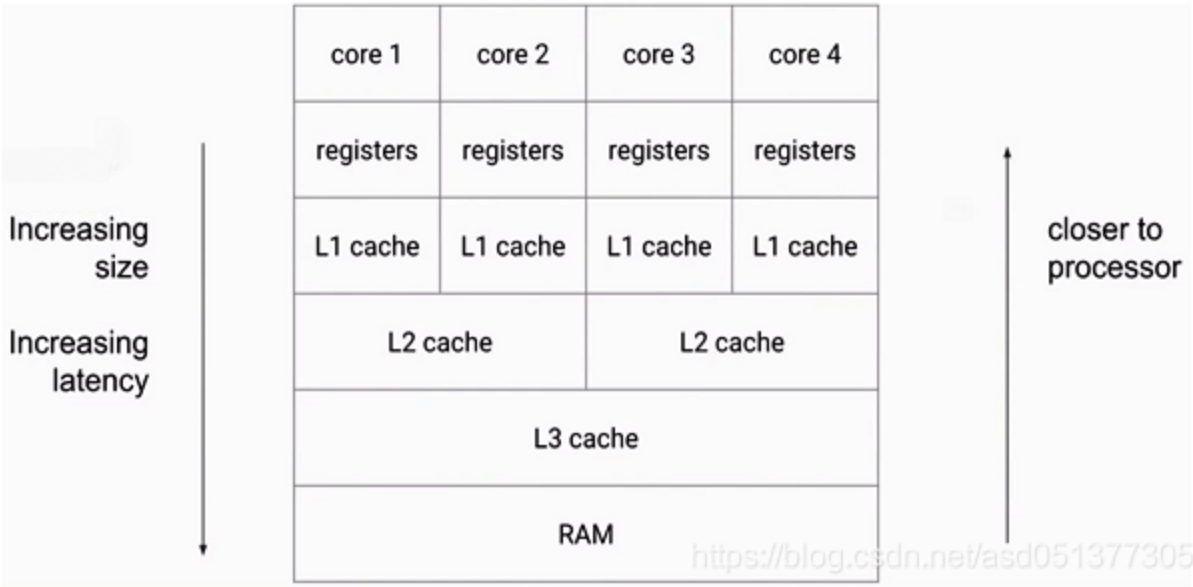
- 可见性问题出现问题原因：

```
①主内存中原x=0，线程1和线程2分别读取了x=0
②线程1在工作内存中赋值x=1，但还没有写入到主内存
③此时线程2的本地内存中x还是0，所以导致了可见性问题
```

- 用volatile解决问题

```
①使用了volatile，线程1在工作内存中修改了x=1后，会强制flush到主内存
②当线程2要读取x/使用旧的x的时候，会判断x为失效，同时重新从主内存中读取进来，则x为新的1
```

5.2、为什么会有可见性问题




```
1 RAM是主内存
2 registers是寄存器
3 core假设是多核CPU
```

- CPU读取寄存器registers中缓存——>registers读取L1 cache级缓存——>L2——>L3——>主内存RAM

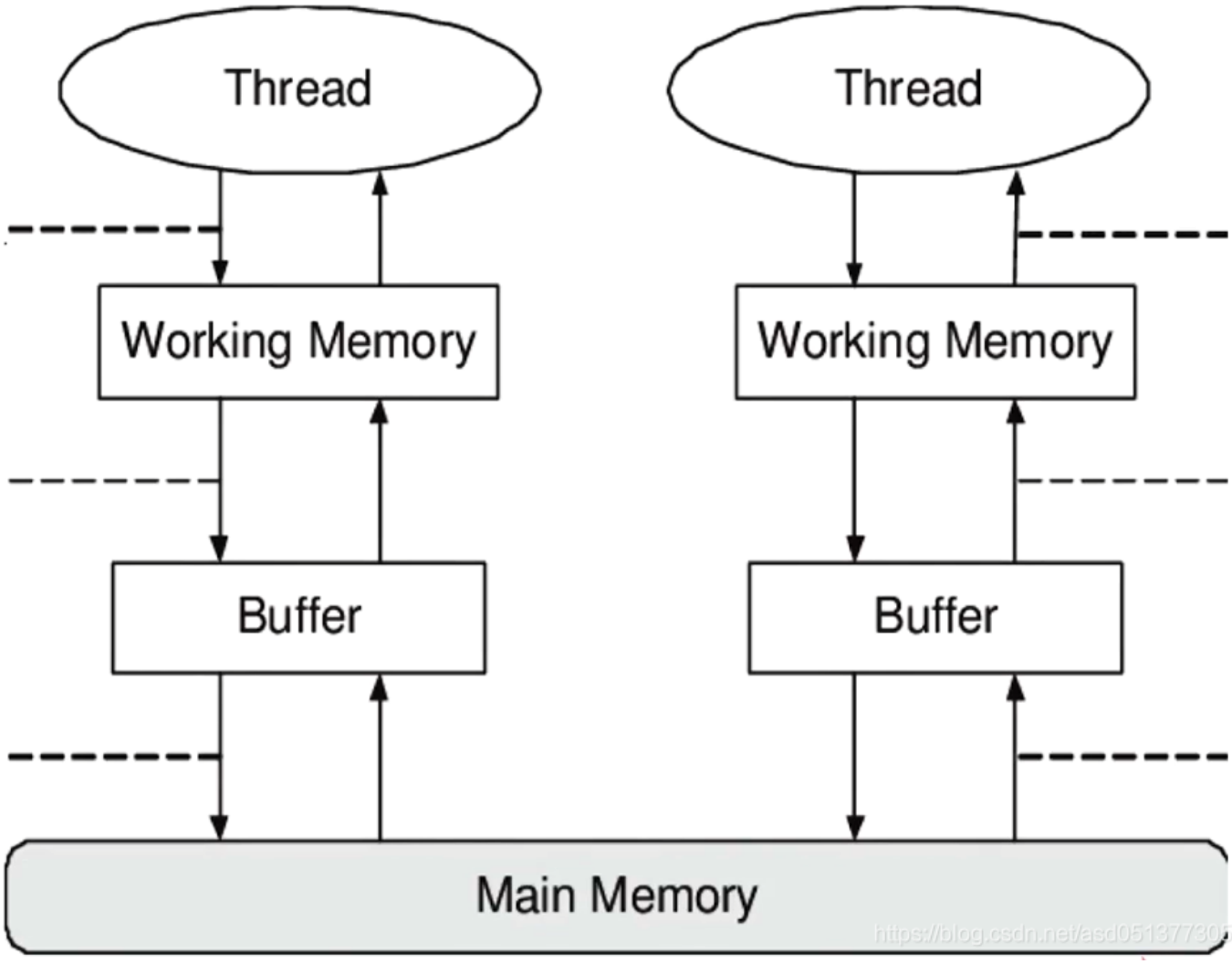
CPU有多级缓存，导致读的数据过期

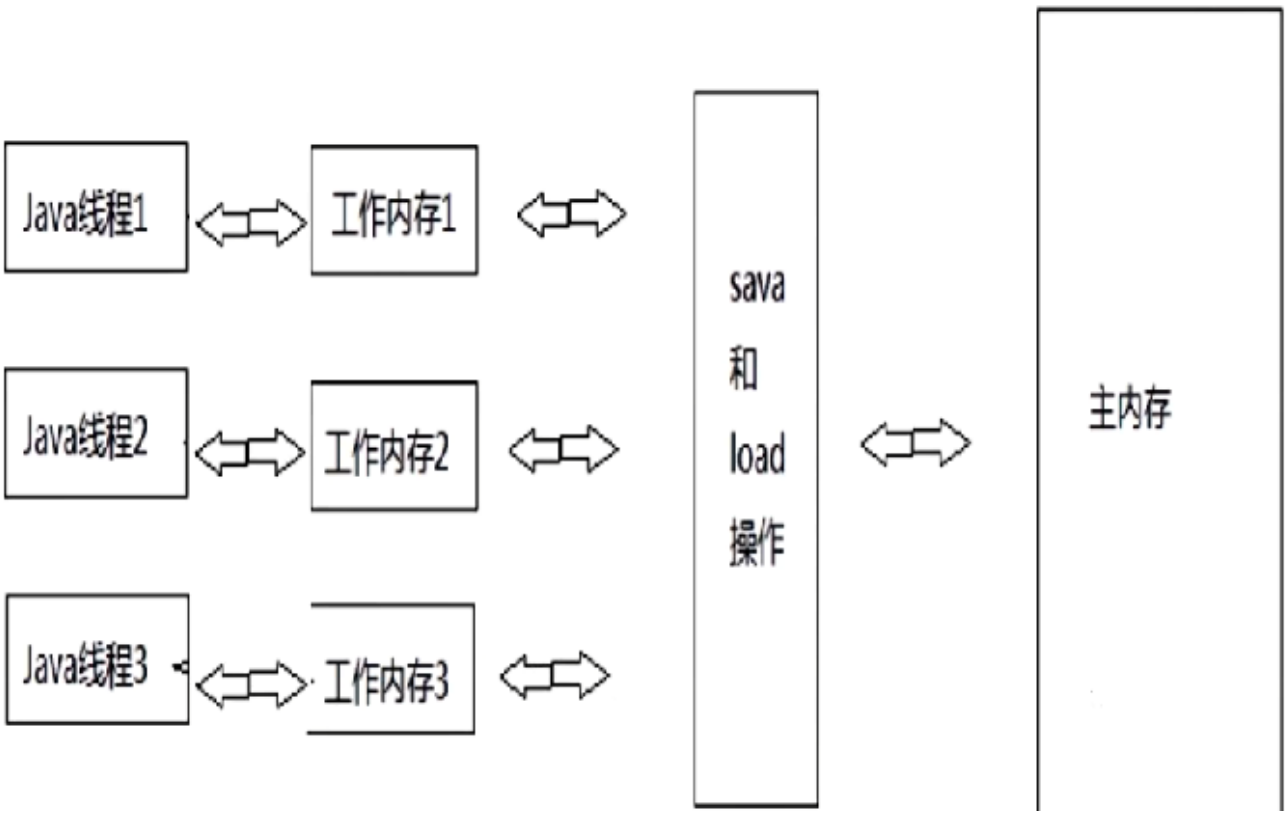
- 高速缓存的容量比主内存小，但是速度仅次于寄存器，所以在CPU和主内存之间就多了Cache层
- 线程间的对于共享变量的可见性问题不是直接由多核引起的，而是由多缓存引起的
- 如果所有的核心(core)都只用一个缓存，那么也就不存在内存可见性问题了
- 每个核心都会将自己需要的数据读到独占缓存(工作内存)中，数据修改后也是写入到独占缓存中，然后等待刷入到主存中。所以会导致有些核心读取的值是一个过期的值

5.3、JMM的抽象：主内存和本地内存

5.3.1 什么是主内存和本地内存

- Java作为高级语言，屏蔽了这些底层细节，用JMM定义了一套读写内存数据的【规范】，虽然我们【不再需要关心一级缓存和二级缓存】的问题，但是，JMM抽象了主内存和本地内存的概念
- 这里说的本地内存【并不是真的是一块给每个线程分配的内存】，而是JMM的一个抽象，是对于寄存器、一级缓存、二级缓存等的【抽象】
- 以上面core、regisiters的那张图来说：（registers、L1、L2是线程的本地内存）（L3、RAM是线程共享的）


Thread
Thread
Working Memory
Working Memory
Buffer
Buffer
Main Memory


Java线程1
工作内存1
Java线程2
工作内存2
Java线程3
工作内存3
sava
和
load
操作
主内存

5.3.2 主内存和本地内存的关系

JMM有以下规定：

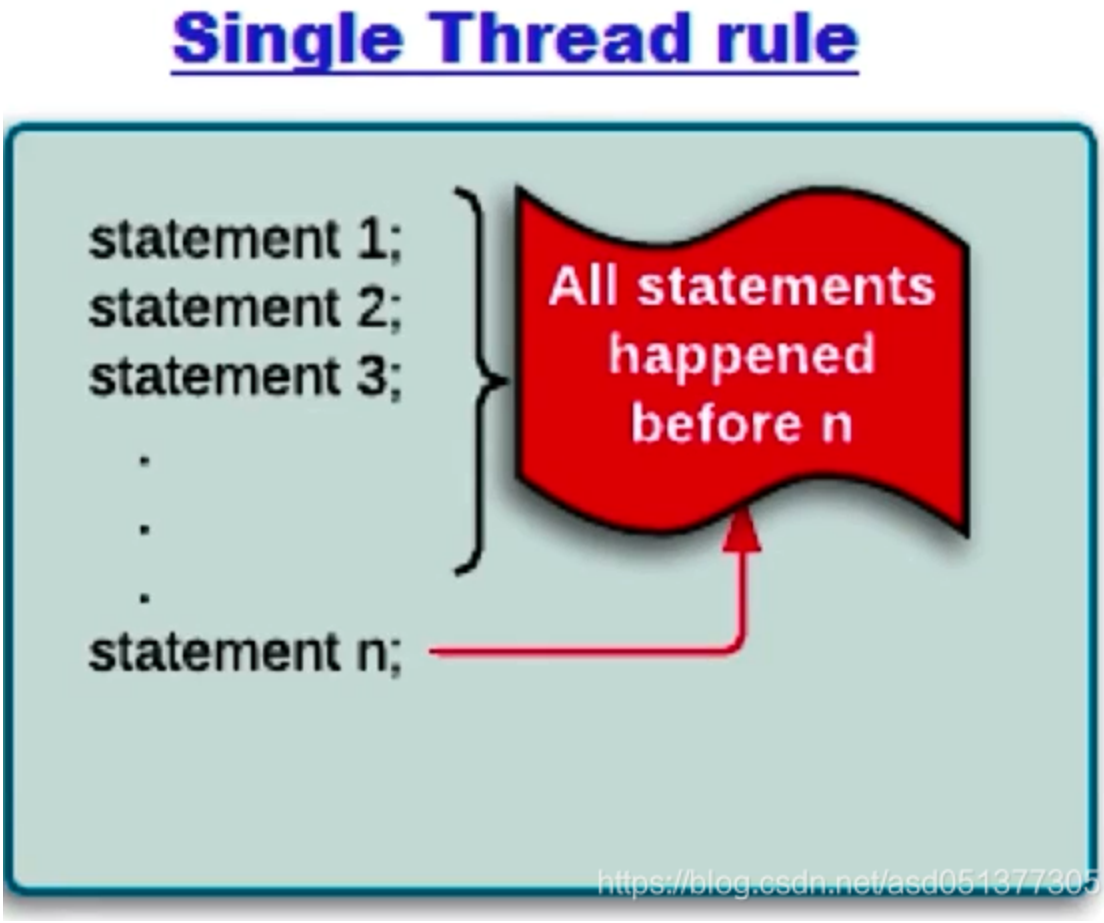
- 【所有的变量】都存储在【主】内存中，同时【每个线程】也有自己【独立】的【工作内存】，工作内存中的变量内容是主内存中的【拷贝】
- 线程【不能直接读写主内存中】的变量，而是只能【操作自己工作内存】中的变量，然后再【同步】到主内存中
- 【主内存】是【多个线程共享】的，但【线程间不共享工作内存】，如果线程间需要【通信】，必须借助【主内存中转】来完成

所有的【共享变量存在于主内存】中，每个【线程有自己的本地内存】，而且【线程读写共享数据也是通过本地内存交换】的，所以才导致了【可见性问题】

5.4、Happens-Before原则

- 什么是happens-before：（解决可见性问题）在时间上，动作A发生在动作B之前，B保证能看见A，这就是happens-before
- Happens-Before原则有哪里？

1. 单线程规则



- 同个线程（同个工作内存）内，前面修改的变量对后面的操作是可见的。但不影响重排序

2. 锁操作（synchronized和Lock）

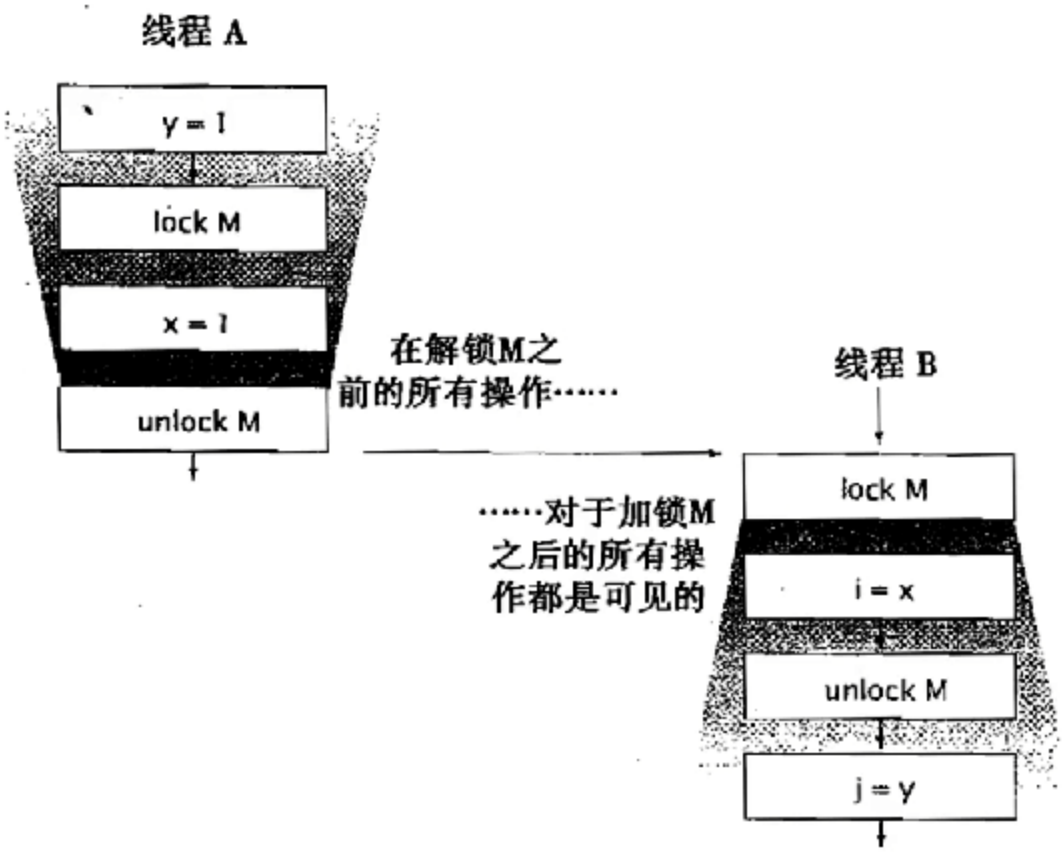
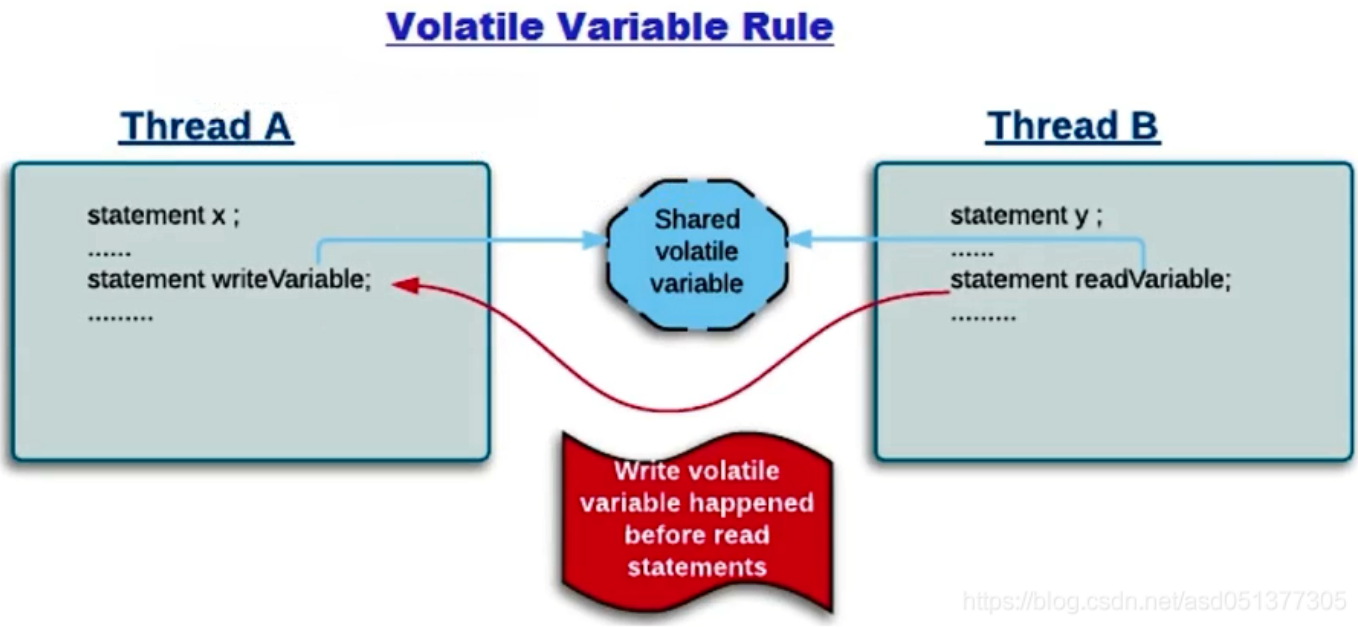


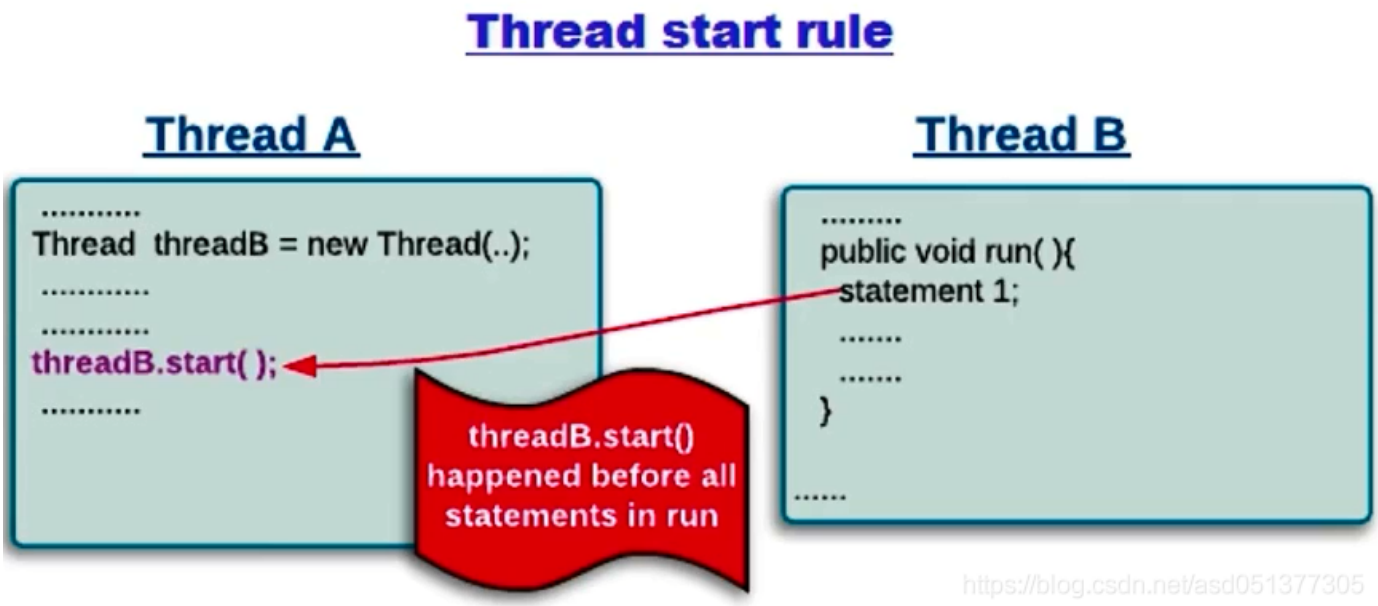
图 16-2　在 Java 内存模型中说明 Happens-Before 关系

3. volatile变量



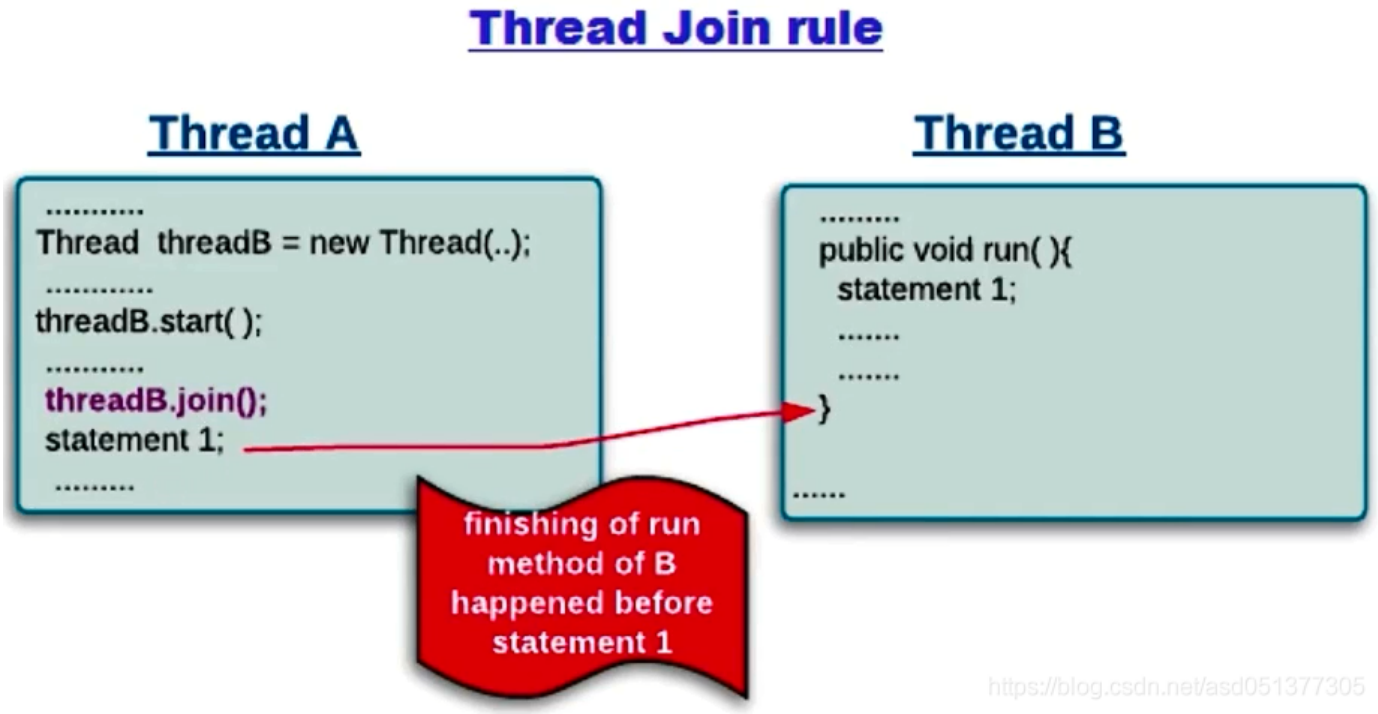
- 理解：只要TheadA volatile变量是已经写入了，那么ThreadB读取就肯定可以读取到最新的结果

4. 线程启动



- ThreadB执行的时候，可以看到ThreadA之前的操作

5. 线程join



- 在ThreadA（例如主线程为main()方法）中执行了ThreadB.join，则ThreadA会等待ThreadB执行完毕后，才执行下面的statement1的操作逻辑
- 当ThreadB执行完毕后，下面的statement1也可以看到statement1的变化

6. 传递性：如果hb(A,B)而且hb(B,C)，那么可以推出hb(A,C)

- 假设背景为main主线程中有ThreadA、ThreadB、ThreadC的执行，如果ThreadA和ThreadB遵循happen-before原则，ThreadB和ThreadC也遵循happens-before，则可以推出hb(A,C)

7. 中断：一个线程被其他线程interrupt时，那么检测中断(isInterrupted)或者抛出InterruptedException一定能看到

8. 构造方法：对象构造方法的最后一行指令happens-before于finalize()方法的第一行指令

- finalize()已不推荐使用

9. 工具类的Happens-Before原则

- （1）线程安全的容器get一定能看到在此之前的put等存入动作

```
如线程安全的ConcurrentHashMap的get和put
```

- （2）CountDownLatch

```
CountDownLatch latch = new CountDownLatch(1);
Thread one = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            latch.countDown();
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        a = 1;
        x = b;
    }
});
one.start();
latch.countDown();

当执行CountDownLatch的countDown()，Thread one才能从await中唤醒，继续执行下面的a=1；x=b
```

- （3）Semaphore：类似CountDownLatch
- （4）Future：可以去后台执行，并拿到一个线程执行结果的类。Future的get是拿到Future的执行结果，get对于之前的执行结果是可见的（不用过多关注，默认保证的）
- （5）线程池：我们会向线程池提交许多任务，然后在提交的任务中，每个任务都可以看到在提交之前的所有的执行结果（不用过多关注，默认保证的）
- （6）CyclicBarrier：CountDownLatch

```
①CyclicBarrier cyclicBarrier1 = new CyclicBarrier(1);
②cyclicBarrier1.await();
③xxx
④cyclicBarrier1.reset(); //当执行了reset后，才能从②await中唤起，继续执行③的代码
```

案例：happens-before演示

- happens-before有一个原则是：如果A是对volatile变量的写操作，B是对同一个变量的读操作，那么hb(A,B)
- 改进：之前是对a、b都加了volatile，实际上在该场景，只要对b加volatile就可以了

```
int a = 1;
volatile int b = 2;

private void change() {
    a = 3;
    b = a;
}

private void print() {
    System.out.println("b=" + b + ";a=" + a);
}
```

- 近朱者赤：给b加了volatile，不仅b被影响，也可以实现轻量级同步
- b之前的写入（对应代码b=a）对读取b后的代码（print b）都可见，所以在writerThread里对a的赋值，一定会对readerThread里的读取可见，所以这里的**【a即使不加volatile，只要b读到的是3，就可以由happens-before原则保证了print a读到的也都是3而不可能读到1】**

5.5、volatile关键字

5.5.1 volatile是什么

- volatile是一种【同步机制】，比synchronized或者Lock相关类【更轻量】，因为使用volatile并不会发生【上下文切换】等开销很大的行为
- 如果一个变量被修饰成volatile，那么JVM就知道了这个变量可能【会被并发修改】（JVM就会做一些相关逻辑，如禁止重排序）
- 开销小，相应的能力也小，虽然说volatile是用来同步地保证线程安全的，但volatile无法保证synchronized那样的【原子保护】，volatile仅在【很有限的场景】下才能发挥作用

5.5.2 volatile的适用场合

（1）不适用a++

（2）适用场景：volatile的变量，不依赖之前的值。如果依赖之前的值如a++（先读a，再+），就会有问题。如果只是对变量进行覆盖赋值（不依赖之前的值），则适用

（3）适用场景1：boolean flag，如果一个共享变量自始自终只【被各个线程赋值】，而没有其他的操作（对比、取值），那么就可以用volatile来代替synchronized或者代替原子变量，因为赋值自身是有原子性的，而volatile又保证了可见性，所以就足以保证线程安全1

（4）适用场景2：作为刷新之前变量的触发器

```
Map configOptions;
char[] configText;
```

```
volatile boolean initialized = false;

//Thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

//Thread B
## 当在ThreadA中initialized设置为true，则在ThreadB中就跳过while，同时因为volatile的happens-before，则在ThreadA的initialized赋值操作前的configOptions肯定已经初始化完毕了
while (!initialized) {
    sleep();
}

//use configOptions
```

5.5.3 volatile的作用：可见性、禁止重排序

（1）可见性：读volatile变量时会去【主内存读取最新值】，写一个volatile属性会【立即刷入到主内存】

（2）禁止指令【重排序】优化：解决单例双重锁乱序问题

5.5.4 volatile和synchronized的关系?

- volatile可看做是【轻量版的synchronized】：如果一个共享变量自始至终【只被各个线程赋值】，而没有其他的操作（读值），那么就可以用volatile来代替synchronized或者代替原子变量，因为【赋值自身是有原子性的，而volatile又保证了可见性】，所以就足以保证线程安全

5.5.5 volatile小结

- 1、volatile修饰符【适用于以下场景】：某个属性被多个线程共享，其中一个线程修改了此属性，其他线程可以立即得到修改后的值，比如【boolean flag】;或者作为【触发器】，实现轻量级同步
- 2、volatile属性的书写操作都是【无锁】的，它不能替代synchronized，因为它没有提供【原子性】和【互斥性】。因为无锁，不需要花费时间在获取锁和释放锁上，所以说它是【低成本】的
- 3、volatile只能作用于【属性】。使用volatile修饰属性，该属性就不会被指令重排序
- 4、volatile提供了【可见性】，任何一个线程对其的修改将立马对其他线程可见。volatile属性不会被线程缓存，始终【从主存中读取】
- 5、volatile提供了【happens-before】保证，对volatile变量v的写入操作-【happens-before】于-所有其他线程后续对v的读操作
- 6、volatile可以【使得long和double的赋值是原子】的

5.6、能保证可见性的措施

- 除了volatile可以让变量保证可见性外，【synchronized、Lock、并发集合、Thread.join()和Thread.start()】等都可以保证可见性
- 具体看happens-before原则的规定

5.7、升华：对synchronized可见性的正确理解

- synchronized不仅保证了原子性，还保证了【可见性】
- synchronized不仅让被保护的代码安全，还让其之前的代码执行结果可见

6、原子性

6.1、什么是原子性

- 一系列操作，要么全部执行成功，要么全部不执行，不会出现执行一半的情况，是不可分割的
- 银行转账问题（A转账给B）：A先减100，B再加100
- i++不是原子性的
- 用synchronized实现原子性

6.2、Java中的原子操作有哪些?

- 除long和double之外的【基本类型】（int,byte,boolean,short,char,float）的赋值操作
- 所有引用【reference的赋值操作】，不管是32位的机器还是64位的机器
- java.concurrent.Atomic.* 包中所有类的原子操作

6.3、long和double的原子性

- 问题描述：官方文档、对于64位的值的写入，可以分为两个32位的操作进行写入、读取错误、使用volatile解决 https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7
- 结论：在32位上的JVM上，long和double的操作不是原子的，但在64位的JVM上是原子的
- 实际开发中：商用Java虚拟机中不会出现

6.4、原子操作+原子操作!=原子操作

- 简单地把原子操作组合在一起，并不能保证整体依然具有原子性
- 全同步的HashMap也不能完全安全（多个synchronized方法操作组合在一起，就不是原子的了）

7、面试常见问题

7.1 JMM应用实例：单例模式8种写法、单例和并发的关系（真实面试超高频考点）

（1）单例模式的作用和使用场景

单例模式的作用

- 为什么需要单例模式：节省内存和计算、保证结果正确、方便管理

单例模式适用场景

- 无状态的工具类：比如日志工具类，不管是在哪里适用，我们需要的只是它帮我们记录日志信息，除此之外，并不需要在它的实例对象上存储任何状态，这个时候我们就只需要一个实例对象即可
- 全局信息类：比如我们在一个类上记录网站的访问次数，我们不希望有的访问被记录在对象A上，有的却记录在对象B上，这时候我们就让这个类成为单例

（2）单例模式的8种写法

1、饿汉式（静态常量）[可用]

```
1 public class Singleton1 {
2     //类加载时就完成了初始化
3     private final static Singleton1 INSTANCE = new Singleton1();
4     private Singleton1() {
```

```
5
6       }
7       public static Singleton1 getInstance() {
8           return INSTANCE;
9       }
10  }
```

2、饿汉式（静态代码块）[可用]

```
1   public class Singleton2 {
2       //类加载时就完成了初始化
3       private final static Singleton2 INSTANCE;
4       static {
5           INSTANCE = new Singleton2();
6       }
7       private Singleton2() {
8
9       }
10      public static Singleton2 getInstance() {
11          return INSTANCE;
12      }
13  }
```

3、懒汉式（线程不安全）[不可用]

```
1   public class Singleton3 {
2       private static Singleton3 instance;
3
4       private Singleton3() {
5
6       }
7
8       public static Singleton3 getInstance() {
9           //如果2个线程同时执行到这一行，则会执行2次new Singleton3()，创建了
    多个实例
10          if (instance == null) {
11              instance = new Singleton3();
12          }
13          return instance;
14      }
15  }
```

4、懒汉式（线程安全，同步方法）[不推荐用，性能差]

```
public class Singleton4 {
    private static Singleton4 instance;

    private Singleton4() {

    }

    //synchronized，多个线程执行会阻塞等待
    public synchronized static Singleton4 getInstance() {
        if (instance == null) {
            instance = new Singleton4();
        }
        return instance;
    }
}
```

5、懒汉式（线程不安全，同步代码块）[不可用]

```
public class Singleton5 {
    private static Singleton5 instance;

    private Singleton5() {

    }

    public static Singleton5 getInstance() {
        //2个线程同时进入这里，则A线程synchronized执行完后，B线程又执行一次synchronized初始化
        //本质还是执行了2次初始化，线程不安全
        if (instance == null) {
            synchronized (Singleton5.class) {
                instance = new Singleton5();
            }
        }
        return instance;
    }
}
```

6、双重检查[推荐用]

```
public class Singleton6 {
    private volatile static Singleton6 instance;

    private Singleton6() {

    }

    public static Singleton6 getInstance() {
        if (instance == null) {
            synchronized (Singleton6.class) {
                if (instance == null) {
                    instance = new Singleton6();
                }
            }
        }
        return instance;
    }
}
```

- 优点：线程安全；延迟加载；效率较高
- 为什么要double-check

```
①线程安全
②单check行不行？：不行多个线程同时执行到instance==null，虽然有synchronized，但也会执行了2次初始化
③直接在方法加synchronized呢？：性能问题，多个线程排队等待
```

- 为什么要用volatile

```
①新建对象实际上有3个步骤：创建空对象、空对象内初始化、赋值
②重排序会带来nullpointexception(重排序后可能顺序：创建空对象、赋值、空对象内初始化，若赋值后就切到另一个线程，此时该单例对象构造方法中有成员初始化，然后此线程还使用到了单例中的未初始化的成员就会空指针)
③防止重排序
```

7、静态内部类[推荐使用]

```
public class Singleton7 {

    private Singleton7() {
```

```
4
5       }
6       //JVM加载Singleton7类的时候，不会初始化内部类变量，达到了懒加载
7       private static class SingletonInstance {
8           private static final Singleton7 instance = new Singleton7
    ();
9       }
10
11      public static Singleton7 getInstance() {
12          //只有当调用到的是，才会进行加载
13          return SingletonInstance.instance;
14      }
15  }
```

8、枚举[推荐用]

```
1  public enum  Singleton8 {
2      INSTANCE;
3      public void whatever() {
4
5      }
6  }
7
8  //调用
9  Singleton8.INSTANCE.whatever();
```

（3）不同写法对比

- 饿汉：简单，但是没有lazy loading，直接就初始化创建了一些对象，而这些对象可能是不需要的
- 懒汉：写法复杂，同时有线程安全问题
- 静态内部类：可用
- 双重检查：同时做到了线程安全和懒加载
- 枚举：最好

（4）用哪种单例的实现方案最好?

《Effective Java》中表明：使用枚举实现单例的方法虽然还没有广泛采用，但单元素的枚举类型已经成为实现Singleton的最佳方法

- 写法简单
- 线程安全有保障
- 避免反序列化破坏单例

（5）各种写法的适用场景

- 最好的方法是利用【枚举】，因为还可以防止反序列化重新创建新的对象

- 非线程同步的方法不能使用
- 如果程序一开始要加载的资源太多，那么就应该使用【懒加载】
- 饿汉式如果是对象的创建需要配置文件就不适用（假设对象的创建需要调用一个前置方法去获取配置，但因为饿汉式，对象被提前创建，而没有将对应的前置方法数据赋值进去，造成创建的对象是一个空对象）
- 懒加载虽然好，但是静态内部类这种方式会引入编程复杂性

7.2 讲一讲什么是Java内存模型

- JMM是什么？：一组规范
- 最重要的3点内容：重排序、可见性、原子性
- 可见性内容从主内存和本地内存、Happens-before原则、volatile
- 原子性：实现原子性的方法、单例模式

7.3 volatile和synchronized的异同?

- volatile可以算是轻量版的synchronized，开销小，适用场合相对就少一点：如果一个共享变量至始至终只被各个线程赋值，而没有其他的操作，那么就可以用volatile来代替

7.4 什么是原子操作？Java中有哪些原子操作？生成对象的过程是不是原子操作?

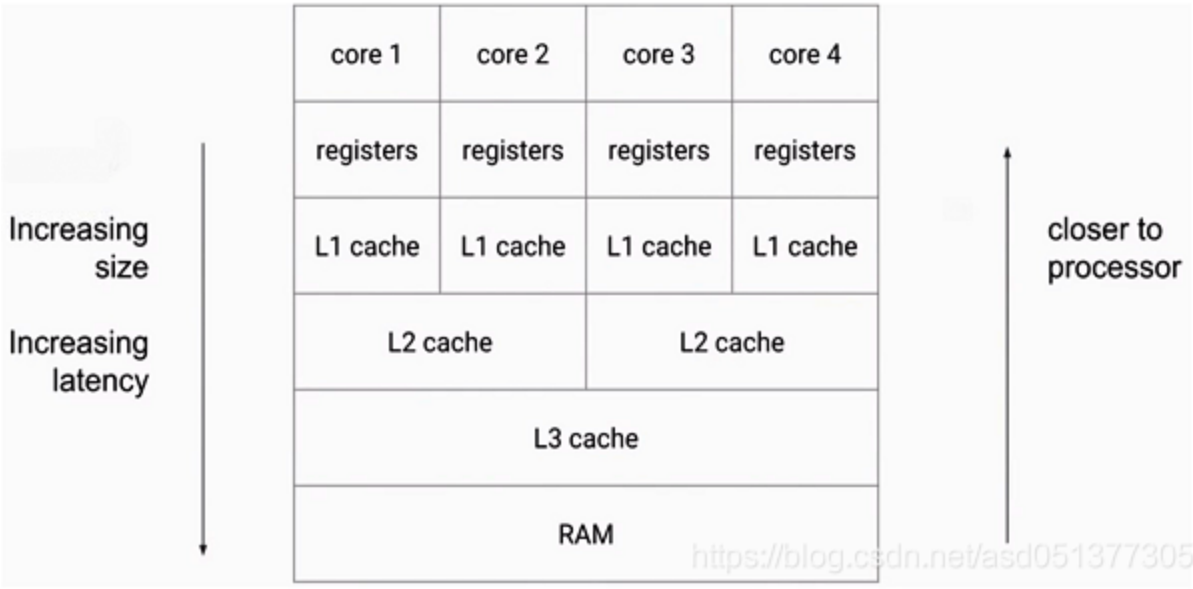
- 什么是原子操作：要么全部执行，要么全部不执行
- Java中有哪里原子操作

```
除long和double之外的【基本类型】（int,byte,boolean,short,char,float）的赋值操作
所有引用【reference的赋值操作】，不管是32位的机器还是64位的机器
java.concurrent.Atomic.* 包中所有类的原子操作
```

- 生成对象的过程是不是原子操作：是多步操作，无法保证原子操作

```
①新建一个空的Person对象
②执行Person的构造函数
③把这个对象的地址指向p
```

7.5 什么是内存可见性?



7.6 64位的double和long写入的时候是原子的吗

- 32位上不是原子的，64位上是原子的，一般不需要我们考虑

8、总结：Java内存模型————底层原理

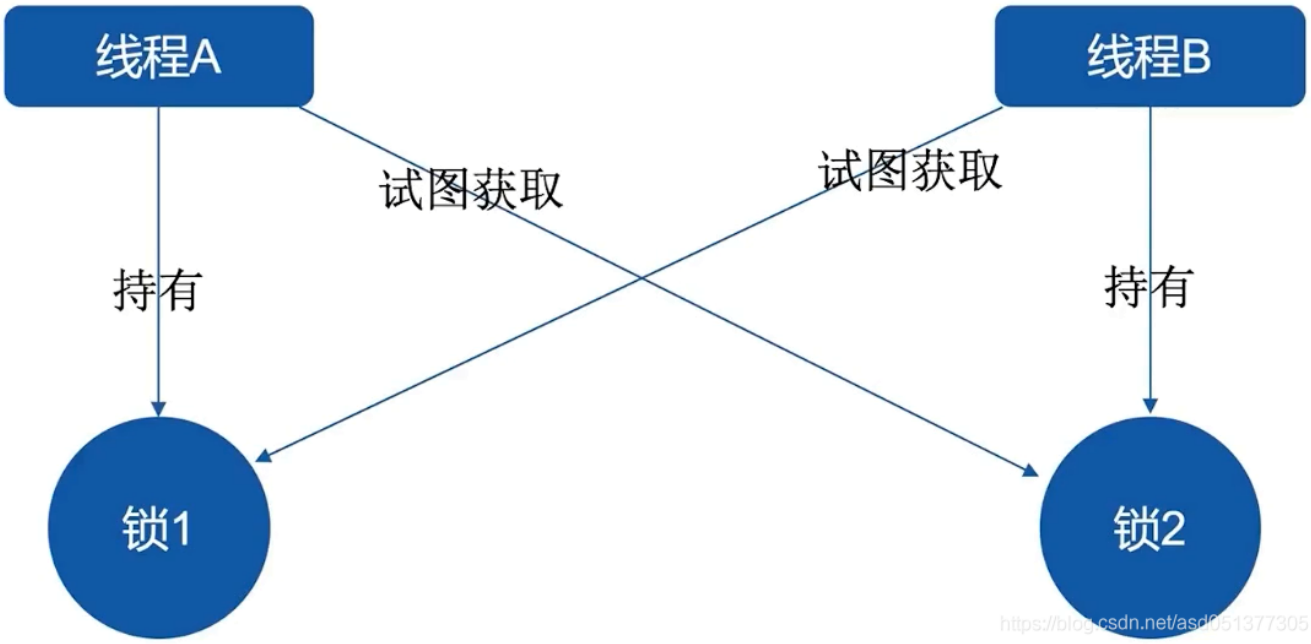
- 什么叫“底层原理”
- 三兄弟：JVM内存结构 VS Java内存模型 VS Java对象模型
- JMM是什么
- 重排序
- 可见性
- 原子性

10. 并发死锁问题与企业级解决方案（死锁、活锁、饥饿）

1、死锁是什么？有什么危害？

1.1 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。



- 多个线程造成死锁的情况（A->B->C->A）

1.2 死锁的影响

死锁的影响在不同系统中是不一样的，这取决于系统对死锁的处理能力

- 数据库中：检测到死锁，（两个事务AB相互竞争），会放弃其中一个事务A，让B先执行，然后再执行A
- JVM中：无法自动处理

1.3 几率不高但危害大

- 不一定发生，但遵循“墨菲定律”（如果事情有变坏的可能，不管这种可能性有多小，它总会发生）
- 一旦发生，多是【高并发】场景，影响用户多
- 整个系统崩溃、子系统崩溃、性能降低
- 压力测试无法找出所有潜在的死锁

2、发生死锁的例子

2.1 最简单的情况

代码

```
public class MustDeadLock implements Runnable {
    int flag = 1;
    static Object o1 = new Object();
    static Object o2 = new Object();

    public static void main(String[] args) {
        MustDeadLock r1 = new MustDeadLock();
        MustDeadLock r2 = new MustDeadLock();
        r1.flag = 1;
        r2.flag = 0;
        Thread t1 = new Thread(r1);
```

```
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
    @Override
    public void run() {
        System.out.println("flag = " + flag);
        if (flag == 1) {
            synchronized (o1) {
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("线程1成功拿到两把锁");
                }
            }
        }
        if (flag == 0) {
            synchronized (o2) {
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println("线程2成功拿到两把锁");
                }
            }
        }
    }
}

//输出结果
flag = 1
flag = 0
//线程一直不解释，处于死锁状态
```

分析

- T1和T2【互相等待】，都需要对方锁定的资源才能继续执行，从而死锁
- 强制中止程序，IDEA会打印多一行（code -1）

```
flag = 1
flag = 0

Process finished with exit code -1
```

- 非0是不正常退出的信号，正常结束的程序的【结束信号是0】

2.2 实际生产中的例子：转账

- 需要两把锁
- 获取两把锁成功，且余额大于0，则扣除转出人，增加收款人的余额，是原子操作
- 顺序相反导致死锁

```
public class TransferMoney implements Runnable {
    int flag = 1;
    static Account a = new Account(500);
    static Account b = new Account(500);

    public static void main(String[] args) throws InterruptedExce
ption {
        TransferMoney r1 = new TransferMoney();
        TransferMoney r2 = new TransferMoney();
        r1.flag = 1;
        r2.flag = 0;
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("a的余额" + a.balance);
        System.out.println("b的余额" + b.balance);
    }

    @Override
    public void run() {
        if (flag == 1) {
            transferMoney(a, b, 200);
        }
```

```
26          if (flag == 0) {
27              transferMoney(b, a, 200);
28          }
29      }
30
31      public static void transferMoney(Account from, Account to, in
    t amount) {
32          synchronized (from) {
33          // 备注代码，可开启备注演示转账死锁
34  //              try {
35  //                  Thread.sleep(500);
36  //              } catch (InterruptedException e) {
37  //                  e.printStackTrace();
38  //              }
39              synchronized (to) {
40                  if (from.balance - amount < 0) {
41                      System.out.println("余额不足，转账失败");
42                  }
43                  from.balance -= amount;
44                  to.balance += amount;
45                  System.out.println("成功转账" + amount + "元");
46              }
47          }
48      }
49
50      static class Account {
51          public Account(int balance) {
52              this.balance = balance;
53          }
54
55          int balance;
56      }
57  }
58
59  //输出结果（没有死锁的情况）
60  成功转账200元
61  成功转账200元
62  a的余额500
63  b的余额500
64
```

```
65 //开启备注的Thread.sleep则a和b线程死锁，没有输出，都在相互等待
```

2.3 模拟多人随机转账

- 5W人很多，但是依然会发生死锁，墨菲定律
- 发生死锁几率不高但危害大

```
1  public class MultiTransferMoney {
2      private static final int NUM_ACCOUNTS = 500;
3      private static final int NUM_MONEY = 1000;
4      private static final int NUM_THREADS = 20;
5      private static int NUM_ITERATIONS = 1000000;
6
7      public static void main(String[] args) {
8          Random rnd = new Random();
9          Account[] accounts = new Account[NUM_ACCOUNTS];
10         for (int i = 0; i < accounts.length; i++) {
11             accounts[i] = new Account(NUM_MONEY);
12         }
13         class TransferThread extends Thread {
14             @Override
15             public void run() {
16                 for (int i = 0; i < NUM_ITERATIONS; i++) {
17                     int fromAcct = rnd.nextInt(NUM_ACCOUNTS);
18                     int toAcct = rnd.nextInt(NUM_ACCOUNTS);
19                     int amount = rnd.nextInt(NUM_MONEY);
20                     TransferMoney.transferMoney(accounts[fromAcc
   t], accounts[toAcct], amount);
21                 }
22                 System.out.println("运行结束");
23             }
24         }
25         for (int i = 0; i < NUM_THREADS; i++) {
26             new TransferThread().start();
27         }
28     }
29 }
30 //输出结果（输出到一定时间后，20个线程都卡住了，死锁）
31 成功转账568元
32 成功转账129元
33 成功转账225元
```

```
成功转账623元
...
成功转账889元
余额不足，转账失败
成功转账451元
余额不足，转账失败
成功转账138元
//所有线程卡住
```

3、死锁的4个必要条件（缺一不可）

- **互斥条件**（线程A拿到了锁lock-a，则其他线程要获取lock-a时只能等待）
- **请求与保持条件**（线程A在请求lock-b的时候，同时保持着lock-a锁）
- **不剥夺条件**（线程A持有lock-a，外界不能剥夺A对lock-a的持有）
- **循环等待条件**（多个线程形成环路，A等待B，B等待C，C等待A）

4、如何定位死锁

4.1 使用java命令jstack（${JAVA_HOME}/bin/jstack pid）

- 查到上面执行的程序的进程pid，执行：D:\Program Files\Java\jdk1.8.0_172\bin>jstack.exe 108352

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x000000001be53948 (object 0x0000000780caf9a0, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x000000001a93cd18 (object 0x0000000780caf9b0, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at ConcurrenceFolder.mooc.threadConcurrencyCore.deadlock.MustDeadLock.run(MustDeadLock.java:48)
        - waiting to lock <0x0000000780caf9a0> (a java.lang.Object)
        - locked <0x0000000780caf9b0> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)
```

```
"Thread-0":
        at ConcurrenceFolder.mooc.threadConcurrencyCore.deadlock.MustDeadLock.run(MustDeadLock.java:36)
        - waiting to lock <0x0000000780caf9b0> (a java.lang.Object)
        - locked <0x0000000780caf9a0> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

- Thread1 lock f9b0,waiting f9a0
- Thread0 lock f9a0,waiting f9b0
- 同时也显示了死锁的位置MustDeadLock.java:48和MustDeadLock.java:36

4.2 ThreadMXBean代码检测

```
public class ThreadMXBeanDetection implements Runnable {
    int flag = 1;
    static Object o1 = new Object();
    static Object o2 = new Object();

    public static void main(String[] args) throws InterruptedException {
        ThreadMXBeanDetection r1 = new ThreadMXBeanDetection();
        ThreadMXBeanDetection r2 = new ThreadMXBeanDetection();
        r1.flag = 1;
        r2.flag = 0;
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        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(deadLockedThreads[i]);
```

```
                System.out.println("发现死锁：" + threadInfo.getThr
eadName());
            }
        }
    }
    @Override
    public void run() {
        System.out.println("flag = " + flag);
        if (flag == 1) {
            synchronized (o1) {
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("线程1成功拿到两把锁");
                }
            }
        }
        if (flag == 0) {
            synchronized (o2) {
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println("线程2成功拿到两把锁");
                }
            }
        }
    }
}

//输出结果
flag = 1
flag = 0
发现死锁：Thread-1
发现死锁：Thread-0
```

5、修复死锁的策略

5.1 线上发生死锁应该怎么办?

- 线上问题都需要防患于未然，不造成损失地扑灭几乎已经是不可能
- 保存案发现场然后立刻重启服务器
- 暂时保证线上服务的安全，然后再利用刚才保存的信息，排查死锁，修改代码，重新发版

5.2 常见修复策略

- 避免策略：【哲学家就餐】的换手方案、转账换序方案（思路：避免相反的获取锁的顺序）
- 检测与恢复策略：一段时间检测是否有死锁，如果有就剥夺某个资源，来打开死锁

5.2.1 转账时避免死锁（转账换序方案）

- 实际上不在乎获取锁的顺序
- 代码演示
- 通过【hashcode】来决定获取锁的顺序、冲突时需要“加时赛”

```
public class TransferMoney implements Runnable {
    int flag = 1;
    static Account a = new Account(500);
    static Account b = new Account(500);
    static Object lock = new Object();

    public static void main(String[] args) throws InterruptedExce
ption {
        TransferMoney r1 = new TransferMoney();
        TransferMoney r2 = new TransferMoney();
        r1.flag = 1;
        r2.flag = 0;
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("a的余额" + a.balance);
        System.out.println("b的余额" + b.balance);
    }

    @Override
    public void run() {
        if (flag == 1) {
```

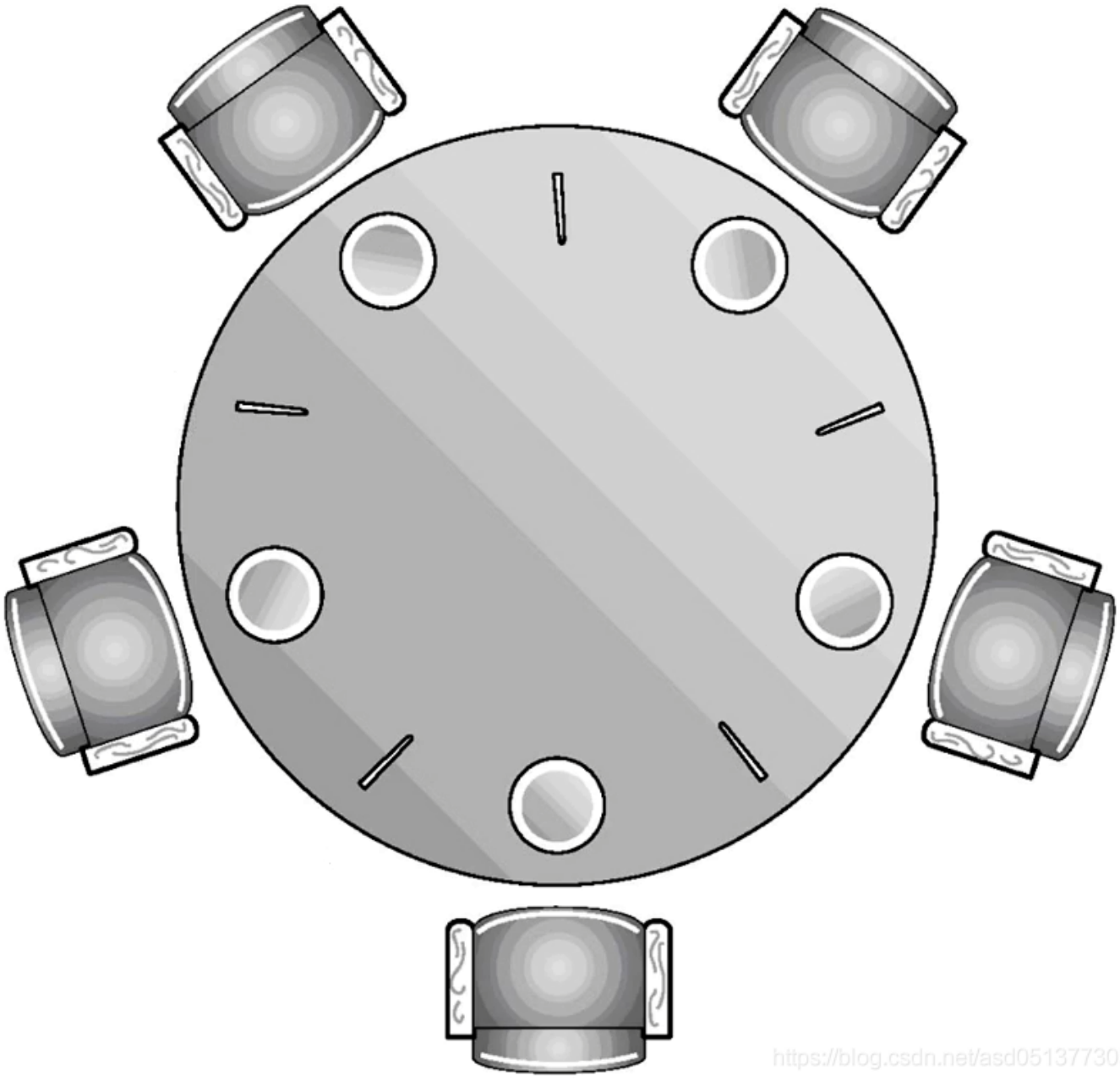
```
25              transferMoney(a, b, 200);
26          }
27          if (flag == 0) {
28              transferMoney(b, a, 200);
29          }
30      }
31
32      public static void transferMoney(Account from, Account to, in
    t amount) {
33          //增加内部类
34          class Helper {
35              public void transfer() {
36                  if (from.balance - amount < 0) {
37                      System.out.println("余额不足，转账失败");
38                  }
39                  from.balance -= amount;
40                  to.balance += amount;
41                  System.out.println("成功转账" + amount + "元");
42              }
43          }
44          int fromHash = System.identityHashCode(from);
45          int toHash = System.identityHashCode(to);
46          //通过通过【hashcode】来决定获取锁的顺序
47          if (fromHash < toHash) {
48              synchronized (from) {
49                  synchronized (to) {
50                      new Helper().transfer();
51                  }
52              }
53          } else if (fromHash > toHash) {
54              synchronized (to) {
55                  synchronized (from) {
56                      new Helper().transfer();
57                  }
58              }
59          } else {
60              //当hashcode相同的时候，冲突时需要“加时赛”，用额外的lock锁
61              synchronized (lock) {
62                  synchronized (to) {
63                      synchronized (from) {
```

```
64                          new Helper().transfer();
65                      }
66                  }
67              }
68          }
69      }
70
71      static class Account {
72          public Account(int balance) {
73              this.balance = balance;
74          }
75
76          int balance;
77      }
78  }
```

- 如果实体有【主键】就更方便

5.2.2 哲学家就餐问题

（1）问题描述



- 流程

```
①先拿起左手的筷子
②然后拿起右手的筷子
③如果筷子被人使用了，那就等别人用完
④吃完后，把筷子放回原位
```

（2）有【死锁】和【资源耗尽】的风险

- 死锁：每个哲学家都拿着左手的筷子，【永远都在等右边】的筷子（或相反）

（3）代码演示：哲学家进入死锁

```
public class DiningPhilosophers {
    public static class Philosopher implements Runnable {
        private Object leftChopstick;
```

```
4         private Object rightChopstick;
5
6         public Philosopher(Object leftChopstick, Object rightChop
  stick) {
7             this.leftChopstick = leftChopstick;
8             this.rightChopstick = rightChopstick;
9         }
10
11        @Override
12        public void run() {
13            try {
14                while (true) {
15                    doAction("Thinking");
16                    synchronized (leftChopstick) {
17                        doAction("Picked up left chopstick");
18                        synchronized (rightChopstick) {
19                            doAction("Pick up right chopstick - e
  ating");
20                            doAction("Put down right chopstick");
21                        }
22                        doAction("Put down left chopstick");
23                    }
24                }
25            } catch (InterruptedException e) {
26                e.printStackTrace();
27            }
28        }
29
30        private void doAction(String action) throws InterruptedEx
  ception {
31            System.out.println(Thread.currentThread().getName() +
  " " + action);
32            Thread.sleep((long) (Math.random() * 10));
33        }
34    }
35
36    public static void main(String[] args) {
37        Philosopher[] philosophers = new Philosopher[5];
38        Object[] chopsticks = new Object[philosophers.length];
39        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) % philosophers.length];
            philosophers[i] = new Philosopher(leftChopstick, rightChopstick);
            new Thread(philosophers[i], "哲学家" + (i + 1) + "号").start();

        }
    }
}
//输出结果
哲学家1号 Thinking
哲学家2号 Thinking
哲学家3号 Thinking
哲学家4号 Thinking
哲学家5号 Thinking
哲学家2号 Picked up left chopstick
哲学家1号 Picked up left chopstick
哲学家5号 Picked up left chopstick
哲学家3号 Picked up left chopstick
哲学家4号 Picked up left chopstick
//程序卡住，死锁
```

（4）多种解决方案

- 服务员检查（避免策略）：服务员检查是否会陷入死锁，如果检查可能存在，则让你先停止请求吃饭
- 【改变一个哲学家拿筷子的顺序（避免策略）】
- 餐票（避免策略）：事先提供允许吃饭的餐票，只有拿到餐票的才可以执行吃饭
- 领导调节（检测与恢复策略）；让程序正常执行，当发现死锁的时候，有一个外部指令进来中止其中一个线程，相当于破坏掉死锁的“不剥夺条件”（线程A持有lock-a，外界不能剥夺A对lock-a的持有）

（5）代码演示：解决死锁

- 【改变一个哲学家拿筷子的顺序（避免策略）】

```
public class DiningPhilosophers {
    public static class Philosopher implements Runnable {
        private Object leftChopstick;
        private Object rightChopstick;
```

```
5
6           public Philosopher(Object leftChopstick, Object rightChop
  stick) {
7               this.leftChopstick = leftChopstick;
8               this.rightChopstick = rightChopstick;
9           }
10
11          @Override
12          public void run() {
13              try {
14                  while (true) {
15                      doAction("Thinking");
16                      synchronized (leftChopstick) {
17                          doAction("Picked up left chopstick");
18                          synchronized (rightChopstick) {
19                              doAction("Pick up right chopstick - e
  ating");
20                              doAction("Put down right chopstick");
21                          }
22                          doAction("Put down left chopstick");
23                      }
24                  }
25              } catch (InterruptedException e) {
26                  e.printStackTrace();
27              }
28          }
29
30          private void doAction(String action) throws InterruptedEx
  ception {
31              System.out.println(Thread.currentThread().getName() +
  " " + action);
32              Thread.sleep((long) (Math.random() * 10));
33          }
34      }
35
36      public static void main(String[] args) {
37          Philosopher[] philosophers = new Philosopher[5];
38          Object[] chopsticks = new Object[philosophers.length];
39          for (int i = 0; i < chopsticks.length; i++) {
40              chopsticks[i] = new Object();
```

```
        }
        for (int i = 0; i < philosophers.length; i++) {
            Object leftChopstick = chopsticks[i];
            Object rightChopstick = chopsticks[(i + 1) % philosophers.length];
            //改进：当是最后一个哲学家，则反过来，先取右边的筷子。这样最多让4个人同时先拿左边筷子，一个人先拿右边筷子。
            // 若此时有同时4个人先拿起了左边的筷子，由于第五个人A需要先拿右筷子，而A的右筷子其实已经被拿起来啦，此时A就需要等待，而A左边的人B就可以拿A左边，但是对于B来说是右边的筷子吃饭，然后再放下筷子换人，此时就不会产生死锁。
            if (i == philosophers.length - 1) {
                philosophers[i] = new Philosopher(rightChopstick, leftChopstick);
            } else {
                philosophers[i] = new Philosopher(leftChopstick, rightChopstick);
            }
            new Thread(philosophers[i], "哲学家" + (i + 1) + "号").start();
        }
    }
}
//输出结果
//长时间打印，没有处于死锁状态
```

5.2.3 死锁检测与恢复策略

（1）检测算法：锁的调用链路图

- 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

（2）恢复方法1：【进程中止】

- 【逐个终止】线程，直到死锁消除
- 终止顺序

```
①优先级（是前台交互还是后台处理）
②已占用资源、还需要的资源（还需要一点资源就可以完成任务的，则优先执行，终止其他的）
③已运行时间（已运行较长时间，快要完成任务的，则优先执行，终止其他的）
```

（3）恢复方法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】

6、实际工程中如何避免死锁

6.1 设置【超时】时间

- Lock的tryLock(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
- 获取锁失败时：打印错误日志、发报警邮件、重启等

```
1  public class TryLockDeadlock implements Runnable {
2      int flag = 1;
3      static Lock lock1 = new ReentrantLock();
4      static Lock lock2 = new ReentrantLock();
5
6      public static void main(String[] args) {
7          TryLockDeadlock r1 = new TryLockDeadlock();
8          TryLockDeadlock r2 = new TryLockDeadlock();
9          r1.flag = 1;
10         r2.flag = 0;
11         new Thread(r1).start();
12         new Thread(r2).start();
13     }
14
15     @Override
16     public void run() {
17         for (int i = 0; i < 100; i++) {
18             if (flag ==1) {
19                 try {
20                     if (lock1.tryLock(800, TimeUnit.MILLISECOND
   S)) {
21                         System.out.println("线程1获取到了锁1");
22                         Thread.sleep(new Random().nextInt(1000));
23                         if (lock2.tryLock(800, TimeUnit.MILLISECO
   NDS)) {
24                             System.out.println("线程1获取到了锁2");
25                             System.out.println("线程1成功获取到了两把
   锁，释放全部锁");
```

```
                            lock2.unlock();
                            lock1.unlock();
                            break;
                        } else {
                            System.out.println("线程1尝试获取锁2失
败，已重试，释放锁1");
                            lock1.unlock();
                            Thread.sleep(new Random().nextInt(100
0));
                        }
                    } 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.MILLISECOND
S)) {
                        System.out.println("线程2获取到了锁2");
                        Thread.sleep(new Random().nextInt(1000));
                        if (lock1.tryLock(3000, TimeUnit.MILLISEC
ONDS)) {
                            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，释放全部锁");
                            lock1.unlock();
                            lock2.unlock();
                            break;
                        } else {
                            System.out.println("线程2尝试获取锁1失
败，已重试，释放锁2");
                            lock2.unlock();
                            Thread.sleep(new Random().nextInt(100
0));
                        }
                    } 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
线程1获取到了锁1
线程2获取到了锁2
线程1尝试获取锁2失败，已重试，释放锁1
线程2获取到了锁1
线程2成功获取到了两把锁，释放全部锁
线程1获取到了锁1
线程1获取到了锁2
线程1成功获取到了两把锁，释放全部锁
```

6.2 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(Collections.synchronizedMap()和Collections.synchronizedList())，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】

6.3 尽量降低锁的使用【粒度】：用不同的锁而不是一个锁

6.4 如果能使用【同步代码块】，就不使用同步方法：方便自己指定锁对象，而不是直接整个方法

6.5 给线程起一个有意义的名字：debug和排查时事半功倍，框架和JDK都遵循这个最佳实践

6.6 避免锁的【嵌套】：MustDeadLock类

```
synchronized(lock1) {
    synchronized(lock2) {
          //xxx
    }
}
```

6.7 分配资源前先看下能不能收回来：银行家算法

6.8 尽量不要几个功能用同一把锁：【专锁专用】

7、其他活性故障

- 死锁是最常见的活跃性问题，不过除了刚才的死锁之外，还有一些类似的问题，会导致程序无法顺利执行，统称为活跃性问题
- 【活锁(LiveLock)】
- 【饥饿】

7.1 活锁

7.1.1 什么是活锁

- 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），但程序却【得不到进展】，因为线程始终重复做同样的事（一直询问请求对方的锁）（同时占用着CPU）
- 如果是死锁，那么就是阻塞，相互等待（不占用CPU）
- 死锁和活锁的【结果是一样的】，就是相互等待着

7.1.2 代码演示

```
public class LiveLock {
    static class Spoon {
        private Diner owner;

        public Spoon(Diner owner) {
            this.owner = owner;
        }

        public Diner getOwner() {
            return owner;
        }

        public void setOwner(Diner owner) {
            this.owner = owner;
        }

        public synchronized void use() {
            System.out.printf("%s吃完了!", owner.name);
        }
    }

    static class Diner {
        private String name;
        private boolean isHunger;

        public Diner(String name) {
            this.name = name;
            isHunger = true;
```

```
        }

        public void eatWith(Spoon spoon, Diner spouse) {
            while (isHunger) {
                if (spoon.owner != this) {
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    continue;
                }
                if (spouse.isHunger) {
                    System.out.println(name + " : 亲爱的" + spous
e.name + "你先吃吧");
                    spoon.setOwner(spouse);
                    continue;
                }
                spoon.use();
                isHunger = false;
                System.out.println(name + " : " + "我吃完了");
                spoon.setOwner(spouse);
            }
        }
    }

    public static void main(String[] args) {
        Diner husband = new Diner("牛郎");
        Diner wife = new Diner("织女");
        Spoon spoon = new Spoon(husband);
        new Thread(() -> husband.eatWith(spoon, wife)).start();
        new Thread(() -> wife.eatWith(spoon, husband)).start();
    }
}
//输出结果
牛郎 : 亲爱的织女你先吃吧
织女 : 亲爱的牛郎你先吃吧
牛郎 : 亲爱的织女你先吃吧
织女 : 亲爱的牛郎你先吃吧
...//一直循环交替输出，不停止
```

```
68 牛郎 ： 亲爱的织女你先吃吧
69 织女 ： 亲爱的牛郎你先吃吧
70 牛郎 ： 亲爱的织女你先吃吧
71 织女 ： 亲爱的牛郎你先吃吧
```

7.1.3 如何解决活锁问题

- 原因：重试机制不变，消息队列始终重试，【吃饭始终谦让】
- 以太网的指数【退避】算法：双方以随机时间等待后再重试，不会因为再次同时碰撞
- 加入【随机】因素

```
1  public class LiveLock {
2      static class Spoon {
3          private Diner owner;
4
5          public Spoon(Diner owner) {
6              this.owner = owner;
7          }
8
9          public Diner getOwner() {
10             return owner;
11         }
12
13         public void setOwner(Diner owner) {
14             this.owner = owner;
15         }
16
17         public synchronized void use() {
18             System.out.printf("%s吃完了!", owner.name);
19         }
20     }
21
22     static class Diner {
23         private String name;
24         private boolean isHunger;
25
26         public Diner(String name) {
27             this.name = name;
28             isHunger = true;
29         }
30
```

```
        public void eatWith(Spoon spoon, Diner spouse) {
            while (isHunger) {
                if (spoon.owner != this) {
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    continue;
                }
                Random random = new Random();
                //加入随机因素
                if (spouse.isHunger && random.nextInt(10) < 9) {
                    System.out.println(name + " : 亲爱的" + spouse.name + "你先吃吧");
                    spoon.setOwner(spouse);
                    continue;
                }
                spoon.use();
                isHunger = false;
                System.out.println(name + " : " + "我吃完了");
                spoon.setOwner(spouse);
            }
        }
    }

    public static void main(String[] args) {
        Diner husband = new Diner("牛郎");
        Diner wife = new Diner("织女");
        Spoon spoon = new Spoon(husband);
        new Thread(() -> husband.eatWith(spoon, wife)).start();
        new Thread(() -> wife.eatWith(spoon, husband)).start();
    }
}
//输出结果
牛郎 : 亲爱的织女你先吃吧
织女 : 亲爱的牛郎你先吃吧
牛郎 : 亲爱的织女你先吃吧
织女吃完了!织女 : 我吃完了
牛郎吃完了!牛郎 : 我吃完了
```

7.1.4 工程中的活锁实例：【消息队列】

- 错误方法：消息处理失败时，如果放到队列开头重试，当服务出了问题，处理该消息一直失败，则会导致程序一直卡着
- 解决：【将失败的消息放到队列尾部】、重试限制（比如限制重连3次，超过3次就做其他的逻辑）

7.2 饥饿

- 当线程需要某些资源（例如CPU），但却【始终得不到】
- 线程的【优先级】设置得过于低（如设置为1），或者有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，浏览器有A线程负责前台响应（打开收藏夹等动作），B线程负责后台下载图片和文件、计算渲染等。如果后台线程B把CPU资源都占用了，那么前台线程A将无法得到很好地执行，这会导致用户体验很差

8、常见面试问题

（1）写一个【必然死锁】的例子，生产中什么场景下会发生死锁?

- 例子：线程设置flag区分启动，相互调用对方的锁（AB，BA）
- 什么场景下会发生死锁：相互调用锁

2、发生死锁必须满足【哪些条件】？

- **互斥条件**（线程A拿到了锁lock-a，则其他线程要获取lock-a时则只能等待）
- **请求与保持条件**（线程A在请求lock-b的时候，同时保持着lock-a锁）
- **不剥夺条件**（线程A持有lock-a，外界不能剥夺A对lock-a的持有）
- **循环等待条件**（多个线程形成环路，A等待B，B等待C，C等待A）

3、如何【定位】死锁

- jstack：发生死锁后，通过pid dump出线程详情
- ThreadMXBean：代码中检测

4、有哪些【解决】死锁问题的【策略】？

- 避免策略：【哲学家就餐】的换手方案（最后一个人切换方向）、转账换序方案（通过【hashcode】来决定获取锁的顺序）
- 检测与恢复策略：一段时间【检测】是否有死锁，如果有就【剥夺】某个资源，来打开死锁
- 鸵鸟策略：不推荐

5、讲一讲经典的【哲学家就餐】问题

- 解决方案

6、实际工程中如何【避免死锁】？

- ①设置【超时】时间
- ②多使用【并发类】而不是自己设计锁
- ③尽量降低锁的使用【粒度】：用不同的锁而不是一个锁
- ④如果能使用【同步代码块】，就不使用同步方法：方便自己指定锁对象，而不是直接整个方法
- ⑤给线程起一个有意义的名字：debug和排查时事半功倍，框架和JDK都遵循这个最佳实践
- ⑥避免锁的【嵌套】：MustDeadLock类
- ⑦分配资源钱先看下能不能收回来：银行家算法

- ⑧尽量不要几个功能用同一把锁：【专锁专用】

7、什么是活跃性问题？活锁、饥饿和死锁有什么区别？