请谈谈你对 volatile 的理解

volatile 是 Java 虚拟机提供的轻量级的同步机制

- 保证可见性
- 禁止指令排序
- 不保证原子性

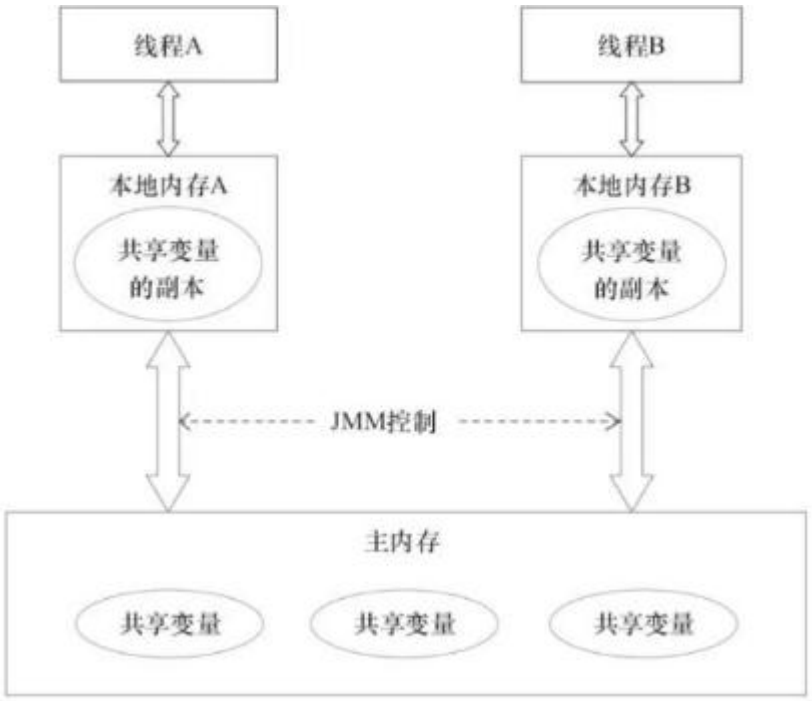
JMM（Java 内存模型） 你谈谈

基本概念

- JMM 本身是一种抽象的概念并不是真实存在，它描述的是一组规定或则规范，通过这组规范定义了程序中的访问方式。
- JMM 同步规定
    - 线程解锁前，必须把共享变量的值刷新回主内存
    - 线程加锁前，必须读取主内存的最新值到自己的工作内存
    - 加锁解锁是同一把锁
- 由于 JVM 运行程序的实体是线程，而每个线程创建时 JVM 都会为其创建一个工作内存，工作内存是每个线程的私有数据区域，而 Java 内存模型中规定所有变量的储存在主内存，主内存是共享内存区域，所有的线程都可以访问，但线程对变量的操作（读取赋值等）必须都工作内存进行看。
- 首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内

存中的变量副本拷贝，前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成。

- 内存模型图



三大特性

- 可见性

```
1/**
2 * @Author: cuzz
3 * @Date: 2019/4/16 21:29
4 * @Description: 可见性代码实例
5 */
6public class VolatileDemo {
7    public static void main(String[] args) {
8        Data data = new Data();
9        new Thread(() -> {
10            System.out.println(Thread.currentThread().getName() + "
11coming...");
12            try {
13                Thread.sleep(3000);
14            } catch (InterruptedException e) {
15                e.printStackTrace();
```

```
            }
            data.addOne();
            System.out.println(Thread.currentThread().getName() + "
updated...");
        }).start();

        while (data.a == 0) {
            // looping
        }
        System.out.println(Thread.currentThread().getName() + " job
is done...");
    }
}

class Data {
    // int a = 0;
    volatile int a = 0;
    void addOne() {
        this.a += 1;
    }
}
```

- 如果不加 volatile 关键字，则主线程会进入死循环，加 volatile 则主线程能够退出，说明加了 volatile 关键字变量，当有一个线程修改了值，会马上被另一个线程感知到，当前值作废，从新从主内存中获取值。对其他线程可见，这就叫可见性。

- 原子性

```
public class VolatileDemo {
    public static void main(String[] args) {
       // test01();
       test02();
    }

    // 测试原子性
    private static void test02() {
        Data data = new Data();
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    data.addOne();
```

```
                }
            }).start();
        }
        // 默认有 main 线程和 gc 线程
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        System.out.println(data.a);
    }
}

class Data {
    volatile int a = 0;
    void addOne() {
        this.a += 1;
    }
}
```

- 发现并不能输入 20000

- 有序性

    - 计算机在执行程序时，为了提高性能，编译器个处理器常常会对指令做重排，一般分为以下 3 种
        - 编译器优化的重排
        - 指令并行的重排
        - 内存系统的重排
    - 单线程环境里面确保程序最终执行的结果和代码执行的结果一致
    - 处理器在进行重排序时必须考虑指令之间的数据依赖性
    - 多线程环境中线程交替执行，由于编译器优化重排的存在，两个线程中使用的变量能否保证用的变量能否一致性是无法确定的，结果无法预测
    - 代码示例

```java
public class ReSortSeqDemo {
    int a = 0;
    boolean flag = false;

    public void method01() {
        a = 1;            // flag = true;
                          // ----线程切换----
        flag = true;      // a = 1;
    }

    public void method02() {
        if (flag) {
            a = a + 3;
            System.out.println("a = " + a);
        }
    }

}
```

- 如果两个线程同时执行，method01 和 method02 如果线程 1 执行 method01 重排序了，然后切换的线程 2 执行 method02 就会出现不一样的结果。

禁止指令排序

volatile 实现禁止指令重排序的优化，从而避免了多线程环境下程序出现乱序的现象

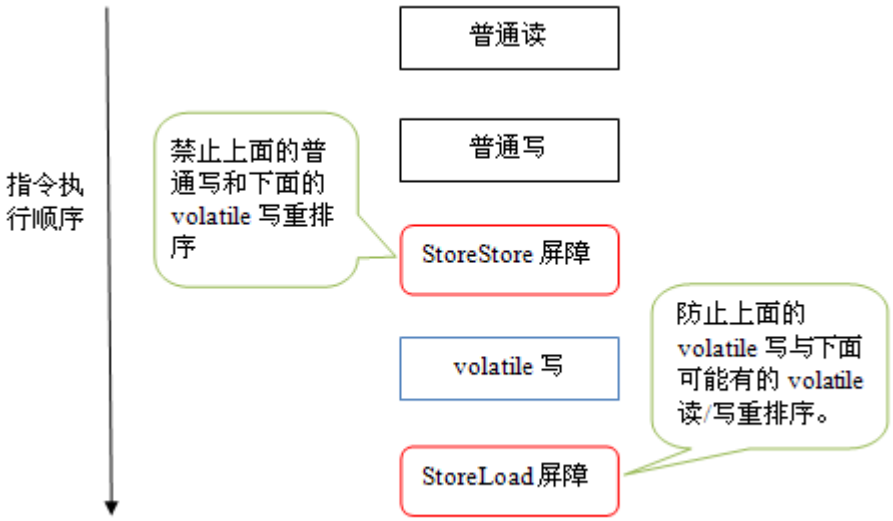
先了解一个概念，内存屏障（Memory Barrier）又称内存栅栏，是一个 CPU 指令，他的作用有两个：

- 保证特定操作的执行顺序
- 保证某些变量的内存可见性（利用该特性实现 volatile 的内存可见性）

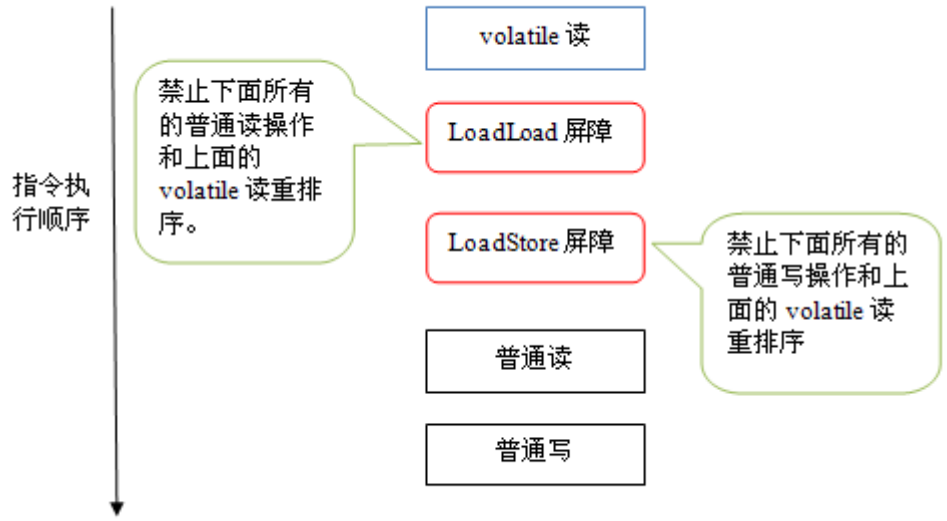
由于编译器个处理器都能执行指令重排序优化，如果在指令间插入一条 Memory Barrier 则会告诉编译器和 CPU，不管什么指令都不能个这条 Memory Barrier 指令重排序，也就

是说通过插入内存屏障禁止在内存屏障前后执行重排序优化。内存屏障另一个作用是强制刷出各种 CPU 缓存数据，因此任何 CPU 上的线程都能读取到这些数据的最新版本。

下面是保守策略下，volatile 写插入内存屏障后生成的指令序列示意图：



下面是在保守策略下，volatile 读插入内存屏障后生成的指令序列示意图：



线程安全性保证

- 工作内存与主内存同步延迟现象导致可见性问题
  - 可以使用 synchronzied 或 volatile 关键字解决，它们可以使用一个线程修改后的变量立即对其他线程可见
- 对于指令重排导致可见性问题和有序性问题
  - 可以利用 volatile 关键字解决，因为 volatile 的另一个作用就是禁止指令重排序优化

你在哪些地方用到过 volatile

单例

- 多线程环境下可能存在的安全问题

```
@NotThreadSafe
public class Singleton01 {
    private static Singleton01 instance = null;
    private Singleton01() {
        System.out.println(Thread.currentThread().getName() + "
construction...");
    }
    public static Singleton01 getInstance() {
        if (instance == null) {
            instance = new Singleton01();
        }
        return instance;
    }

    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newFixedThreadPool(10);
        for (int i = 0; i < 10; i++) {
            executorService.execute(()-> Singleton01.getInstance());
        }
```

```
        executorService.shutdown();
    }
}
```

- 发现构造器里的内容会多次输出

- 双重锁单例

    - 代码

```
public class Singleton02 {
    private static volatile Singleton02 instance = null;
    private Singleton02() {
        System.out.println(Thread.currentThread().getName() + "
construction...");
    }
    public static Singleton02 getInstance() {
        if (instance == null) {
            synchronized (Singleton01.class) {
                if (instance == null) {
                    instance = new Singleton02();
                }
            }
        }
        return instance;
    }

    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newFixedThreadPool(10);
        for (int i = 0; i < 10; i++) {
            executorService.execute(()-> Singleton02.getInstance());
        }
        executorService.shutdown();
    }
}
```

    - 如果没有加 volatile 就不一定是线程安全的，原因是指令重排序的存在，加入 volatile 可以禁止指令重排。

- 原因是在于某一个线程执行到第一次检测，读取到的 instance 不为 null 时，**instance 的引用对象可能还没有完成初始化。**

- `instance = new Singleton()` 可以分为以下三步完成

```
memory = allocate();  // 1.分配对象空间
instance(memory);     // 2.初始化对象
instance = memory;    // 3.设置 instance 指向刚分配的内存地址，此时 instance != null
```

- 步骤 2 和步骤 3 不存在依赖关系，而且无论重排前还是重排后程序的执行结果在单线程中并没有改变，因此这种优化是允许的。

- 发生重排

```
memory = allocate();  // 1.分配对象空间
instance = memory;    // 3.设置 instance 指向刚分配的内存地址，此时 instance != null，但对象还没有初始化完成
instance(memory);     // 2.初始化对象
```

- 所以不加 volatile 返回的实例不为空，但可能是未初始化的实例

CAS 你知道吗？

```
public class CASDemo {
    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(666);
        // 获取真实值，并替换为相应的值
        boolean b = atomicInteger.compareAndSet(666, 2019);
        System.out.println(b); // true
        boolean b1 = atomicInteger.compareAndSet(666, 2020);
        System.out.println(b1); // false
        atomicInteger.getAndIncrement();
    }
}
```

CAS 底层原理？谈谈对 UnSafe 的理解？

getAndIncrement();

```
/**
 * Atomically increments by one the current value.
 *
 * @return the previous value
 */
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

引出一个问题：UnSafe 类是什么？

UnSafe 类

```
public class AtomicInteger extends Number implements java.io.Serializable
{
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            // 获取下面 value 的地址偏移量
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
        // ...
}
```

- Unsafe 是 CAS 的核心类，由于 Java 方法无法直接访问底层系统，而需要通过本地（native）方法来访问， Unsafe 类相当一个后门，基于该类可以直接操作特定内存的数据。Unsafe 类存在于 sun.misc 包中，其内部方法操作可以像 C 指针一样直接操作内存，因为 Java 中 CAS 操作执行依赖于 Unsafe 类。

- 变量 vauleOffset，表示该变量值在内存中的偏移量，因为 Unsafe 就是根据内存偏移量来获取数据的。
- 变量 value 用 volatile 修饰，保证了多线程之间的内存可见性。

CAS 是什么

- CAS 的全称 Compare-And-Swap，它是一条 CPU 并发。
- 它的功能是判断内存某一个位置的值是否为预期，如果是则更改这个值，这个过程就是原子的。
- CAS 并发原体现在 JAVA 语言中就是 sun.misc.Unsafe 类中的各个方法。调用 UnSafe 类中的 CAS 方法，JVM 会帮我们实现出 CAS 汇编指令。这是一种完全依赖硬件的功能，通过它实现了原子操作。由于 CAS 是一种系统源语，源语属于操作系统用语范畴，是由若干条指令组成，用于完成某一个功能的过程，并且原语的执行必须是连续的，在执行的过程中不允许被中断，也就是说 CAS 是一条原子指令，不会造成所谓的数据不一致的问题。
- 分析一下 getAndAddInt 这个方法

```
 // unsafe.getAndAddInt
1public final int getAndAddInt(Object obj, long valueOffset, long
2expected, int val) {
3    int temp;
4    do {
5        temp = this.getIntVolatile(obj, valueOffset);  // 获取快照值
6    } while (!this.compareAndSwap(obj, valueOffset, temp, temp +
7val));  // 如果此时 temp 没有被修改，就能退出循环，否则重新获取
8    return temp;
 }
```

CAS 的缺点？

- 循环时间长开销很大
    - 如果 CAS 失败，会一直尝试，如果 CAS 长时间一直不成功，可能会给 CPU 带来很大的开销（比如线程数很多，每次比较都是失败，就会一直循环），所以希望是线程数比较小的场景。
- 只能保证一个共享变量的原子操作
    - 对于多个共享变量操作时，循环 CAS 就无法保证操作的原子性。
- 引出 ABA 问题

原子类 AtomicInteger 的 ABA 问题谈一谈？原子更新引用知道吗？

- 原子引用

```
public class AtomicReferenceDemo {
    public static void main(String[] args) {
        User cuzz = new User("cuzz", 18);
        User faker = new User("faker", 20);
        AtomicReference<User> atomicReference = new
AtomicReference<>();
        atomicReference.set(cuzz);
        System.out.println(atomicReference.compareAndSet(cuzz,
faker)); // true
        System.out.println(atomicReference.get()); //
User(userName=faker, age=20)
    }
}
```

- ABA 问题是怎么产生的

```
/**
 * @program: learn-demo
 * @description: ABA
```

```
 * @author: cuzz
 * @create: 2019-04-21 23:31
 **/
public class ABADemo {
    private static AtomicReference<Integer> atomicReference = new
AtomicReference<>(100);

    public static void main(String[] args) {
        new Thread(() -> {
            atomicReference.compareAndSet(100, 101);
            atomicReference.compareAndSet(101, 100);
        }).start();

        new Thread(() -> {
            // 保证上面线程先执行
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            atomicReference.compareAndSet(100, 2019);
            System.out.println(atomicReference.get()); // 2019
        }).start();
    }
}
```

- 当有一个值从 A 改为 B 又改为 A，这就是 ABA 问题。

- 时间戳原子引用

```
package com.cuzz.thread;

import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicStampedReference;

/**
 * @program: learn-demo
 * @description: ABA
 * @author: cuzz
 * @create: 2019-04-21 23:31
 **/

public class ABADemo2 {
```

```
    private static AtomicStampedReference<Integer>
atomicStampedReference = new AtomicStampedReference<>(100, 1);

    public static void main(String[] args) {
        new Thread(() -> {
            int stamp = atomicStampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "
的版本号为: " + stamp);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            atomicStampedReference.compareAndSet(100, 101,
atomicStampedReference.getStamp(),
atomicStampedReference.getStamp() + 1 );
            atomicStampedReference.compareAndSet(101, 100,
atomicStampedReference.getStamp(),
atomicStampedReference.getStamp() + 1 );
        }).start();

        new Thread(() -> {
            int stamp = atomicStampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "
的版本号为: " + stamp);
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            boolean b = atomicStampedReference.compareAndSet(100,
2019, stamp, stamp + 1);
            System.out.println(b); // false

System.out.println(atomicStampedReference.getReference()); // 100
        }).start();
    }
}
```

- 我们先保证两个线程的初始版本为一致 ,后面修改是由于版本不一样就会修改失败。

我们知道 ArrayList 是线程不安全，请编写一个不安全的案例并给出解决方案？

- 故障现象

```
public class ContainerDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        Random random = new Random();
        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                list.add(random.nextInt(10));
                System.out.println(list);
            }).start();
        }
    }
}
```

- 发现报 `java.util.ConcurrentModificationException`
- 导致原因
    - 并发修改导致的异常
- 解决方案
    - `new Vector();`
    - `Collections.synchronizedList(new ArrayList<>());`
    - `new CopyOnWriteArrayList<>();`
- 优化建议
    - 在读多写少的时候推荐使用 CopeOnWriteArrayList 这个类

java 中锁你知道哪些？请手写一个自旋锁？

公平和非公平锁

- 是什么
  - **公平锁：**是指多个线程按照申请的顺序来获取值
  - **非公平锁：**是值多个线程获取值的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁，在高并发的情况下，可能会造成优先级翻转或者饥饿现象
- 两者区别
  - **公平锁：**在并发环境中，每一个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个就占有锁，否者就会加入到等待队列中，以后会按照 FIFO 的规则获取锁
  - **非公平锁：**一上来就尝试占有锁，如果失败在进行排队

可重入锁和不可重入锁

- 是什么
  - **可重入锁 :**指的是同一个线程外层函数获得锁之后 ,内层仍然能获取到该锁，在同一个线程在外层方法获取锁的时候 ,在进入内层方法或会自动获取该锁
  - **不可重入锁：** 所谓不可重入锁，即若当前线程执行某个方法已经获取了该锁，那么在方法中尝试再次获取锁时，就会获取不到被阻塞
- 代码实现
  - 可重入锁

```
public class ReentrantLock {
    boolean isLocked = false;
    Thread lockedBy = null;
    int lockedCount = 0;
    public synchronized void lock() throws
InterruptedException {
        Thread thread = Thread.currentThread();
        while (isLocked && lockedBy != thread) {
            wait();
        }
        isLocked = true;
        lockedCount++;
        lockedBy = thread;
    }

    public synchronized void unlock() {
        if (Thread.currentThread() == lockedBy) {
            lockedCount--;
            if (lockedCount == 0) {
                isLocked = false;
                notify();
            }
        }
    }
}
```

- 测试

```
public class Count {
//    NotReentrantLock lock = new NotReentrantLock();
    ReentrantLock lock = new ReentrantLock();
    public void print() throws InterruptedException{
        lock.lock();
        doAdd();
        lock.unlock();
    }

    private void doAdd() throws InterruptedException {
        lock.lock();
        // do something
        System.out.println("ReentrantLock");
        lock.unlock();
    }

```

```
    public static void main(String[] args) throws
InterruptedException {
        Count count = new Count();
        count.print();
    }
}
```

- 发现可以输出 ReentrantLock ,我们设计两个线程调用 print() 方法 ,第一个线程调用 print() 方法获取锁，进入 lock() 方法，由于初始 lockedBy 是 null ,所以不会进入 while 而挂起当前线程 ,而是是增量 lockedCount 并记录 lockBy 为第一个线程。接着第一个线程进入 doAdd() 方法 , 由于同一进程 , 所以不会进入 while 而挂起 , 接着增量 lockedCount , 当第二个线程尝试 lock，由于 isLocked=true，所以他不会获取该锁，直到第一个线程调用两次 unlock() 将 lockCount 递减为 0 ,才将标记为 isLocked 设置为 false。

- 不可重入锁

```
public class NotReentrantLock {
    private boolean isLocked = false;
    public synchronized void lock() throws
InterruptedException {
        while (isLocked) {
            wait();
        }
        isLocked = true;
    }
    public synchronized void unlock() {
        isLocked = false;
        notify();
    }
}
```

- 测试

```
public class Count {
    NotReentrantLock lock = new NotReentrantLock();
    public void print() throws InterruptedException{
        lock.lock();
        doAdd();
        lock.unlock();
    }

    private void doAdd() throws InterruptedException {
        lock.lock();
        // do something
        lock.unlock();
    }

    public static void main(String[] args) throws
InterruptedException {
        Count count = new Count();
        count.print();
    }
}
```

- 当前线程执行 print()方法首先获取 lock，接下来执行 doAdd()方法就无法执行 doAdd()中的逻辑 ,必须先释放锁。这个例子很好的说明了不可重入锁。

- synchronized 和 ReentrantLock 都是可重入锁

  - synchronzied

```
public class SynchronziedDemo {

    private synchronized void print() {
        doAdd();
    }
    private synchronized void doAdd() {
        System.out.println("doAdd...");
    }

    public static void main(String[] args) {
        SynchronziedDemo synchronziedDemo = new
SynchronziedDemo();
        synchronziedDemo.print(); // doAdd...
```

```
    }
}
```

- 上面可以说明 synchronized 是可重入锁。

- ReentrantLock

```
public class ReentrantLockDemo {
    private Lock lock = new ReentrantLock();

    private void print() {
        lock.lock();
        doAdd();
        lock.unlock();
    }

    private void doAdd() {
        lock.lock();
        lock.lock();
        System.out.println("doAdd...");
        lock.unlock();
        lock.unlock();
    }

    public static void main(String[] args) {
        ReentrantLockDemo reentrantLockDemo = new
ReentrantLockDemo();
        reentrantLockDemo.print();
    }
}
```

- 上面例子可以说明 ReentrantLock 是可重入锁，而且在 #doAdd 方法中加两次锁和解两次锁也可以。

自旋锁

- 是指定尝试获取锁的线程不会立即堵塞，而是**采用循环的方式去尝试获取锁**，这样的好处是减少线程上线文切换的消耗，缺点就是循环会消耗 CPU。

- 手动实现自旋锁

```
public class SpinLock {
    private AtomicReference<Thread> atomicReference = new
AtomicReference<>();
    private void lock () {
        System.out.println(Thread.currentThread() + " coming...");
        while (!atomicReference.compareAndSet(null,
Thread.currentThread())) {
            // loop
        }
    }

    private void unlock() {
        Thread thread = Thread.currentThread();
        atomicReference.compareAndSet(thread, null);
        System.out.println(thread + " unlock...");
    }

    public static void main(String[] args) throws
InterruptedException {
        SpinLock spinLock = new SpinLock();
        new Thread(() -> {
            spinLock.lock();
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("hahaha");
            spinLock.unlock();

        }).start();

        Thread.sleep(1);

        new Thread(() -> {
            spinLock.lock();
            System.out.println("hehehe");
            spinLock.unlock();
        }).start();
    }
}
```

- 输出：

```
Thread[Thread-0,5,main] coming...
Thread[Thread-1,5,main] coming...
hahaha
Thread[Thread-0,5,main] unlock...
hehehe
Thread[Thread-1,5,main] unlock...
```

- 获取锁的时候，如果原子引用为空就获取锁，不为空表示有人获取了锁，就循环等待。

独占锁（写锁）/共享锁（读锁）

- 是什么
  - **独占锁**：指该锁一次只能被一个线程持有
  - **共享锁**：该锁可以被多个线程持有
- 对于 ReentrantLock 和 synchronized 都是独占锁；对与 ReentrantReadWriteLock 其读锁是共享锁而写锁是独占锁。读锁的共享可保证并发读是非常高效的，读写、写读和写写的过程是互斥的。
- 读写锁例子

```
public class MyCache {

    private volatile Map<String, Object> map = new HashMap<>();

    private ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
    WriteLock writeLock = lock.writeLock();
    ReadLock readLock = lock.readLock();

    public void put(String key, Object value) {
        try {
```

```
            writeLock.lock();
            System.out.println(Thread.currentThread().getName() + "
正在写入...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            map.put(key, value);
            System.out.println(Thread.currentThread().getName() + "
写入完成，写入结果是 " + value);
        } finally {
            writeLock.unlock();
        }
    }

    public void get(String key) {
        try {
            readLock.lock();
            System.out.println(Thread.currentThread().getName() + "
正在读...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Object res = map.get(key);
            System.out.println(Thread.currentThread().getName() + "
读取完成，读取结果是 " + res);
        } finally {
            readLock.unlock();
        }
    }
}
```

- 测试

```
public class ReadWriteLockDemo {
    public static void main(String[] args) {
        MyCache cache = new MyCache();

        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(() -> {
```

```
                cache.put(temp + "", temp + "");
            }).start();
        }

        for (int i = 0; i < 5; i++) {
            final int temp = i;
            new Thread(() -> {
                cache.get(temp + "");
            }).start();
        }
    }
}
```

- 输出结果

```
Thread-0 正在写入...
Thread-0 写入完成，写入结果是 0
Thread-1 正在写入...
Thread-1 写入完成，写入结果是 1
Thread-2 正在写入...
Thread-2 写入完成，写入结果是 2
Thread-3 正在写入...
Thread-3 写入完成，写入结果是 3
Thread-4 正在写入...
Thread-4 写入完成，写入结果是 4
Thread-5 正在读...
Thread-7 正在读...
Thread-8 正在读...
Thread-6 正在读...
Thread-9 正在读...
Thread-5 读取完成，读取结果是 0
Thread-7 读取完成，读取结果是 2
Thread-8 读取完成，读取结果是 3
Thread-6 读取完成，读取结果是 1
Thread-9 读取完成，读取结果是 4
```

- 能保证**读写**、**写读**和**写写**的过程是互斥的时候是独享的，**读读**的时候是共享的。

CountDownLatch/CyclicBarrier/Semaphore 使用过吗？

CountDownLatch

让一些线程堵塞直到另一个线程完成一系列操作后才被唤醒。CountDownLatch 主要有两个方法，当一个或多个线程调用 await 方法时，调用线程会被堵塞，其他线程调用 countDown 方法会将计数减一（调用 countDown 方法的线程不会堵塞），当计数其值变为零时，因调用 await 方法被堵塞的线程会被唤醒，继续执行。

假设我们有这么一个场景，教室里有班长和其他 6 个人在教室上自习，怎么保证班长等其他 6 个人都走出教室在把教室门给关掉。

```
public class CountDownLanchDemo {
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " 离开了教室...");
            }, String.valueOf(i)).start();
        }
        System.out.println("班长把门给关了，离开了教室...");
    }
}
```

此时输出

```
0 离开了教室...
1 离开了教室...
2 离开了教室...
3 离开了教室...
班长把门给关了，离开了教室...
5 离开了教室...
4 离开了教室...
```

发现班长都没有等其他人理他教室就把门给关了，此时我们就可以使用 CountDownLatch 来控制

```
public class CountDownLanchDemo {
    public static void main(String[] args) throws InterruptedException {
```

```
        CountDownLatch countDownLatch = new CountDownLatch(6);
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                countDownLatch.countDown();
                System.out.println(Thread.currentThread().getName() + " 离开了教室...");
            }, String.valueOf(i)).start();
        }
        countDownLatch.await();
        System.out.println("班长把门给关了，离开了教室...");
    }
}
```

此时输出

```
0 离开了教室...
1 离开了教室...
2 离开了教室...
3 离开了教室...
4 离开了教室...
5 离开了教室...
班长把门给关了，离开了教室...
```

CyclicBarrier

我们假设有这么一个场景，每辆车只能坐个人，当车满了，就发车。

```
public class CyclicBarrierDemo {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(4, () -> {
            System.out.println("车满了，开始出发...");
        });
        for (int i = 0; i < 8; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " 开始上车...");
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
```

```
            }).start();
        }
    }
}
```

输出结果

```
Thread-0 开始上车...
Thread-1 开始上车...
Thread-3 开始上车...
Thread-4 开始上车...
车满了，开始出发...
Thread-5 开始上车...
Thread-7 开始上车...
Thread-2 开始上车...
Thread-6 开始上车...
车满了，开始出发...
```

Semaphore

假设我们有 3 个停车位，6 辆车去抢

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3);
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                try {
                    semaphore.acquire(); // 获取一个许可
                    System.out.println(Thread.currentThread().getName() + "抢到车位...");
                    Thread.sleep(3000);
                    System.out.println(Thread.currentThread().getName() + "离开车位");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release(); // 释放一个许可
                }
            }).start();
        }
    }
}
```

输出

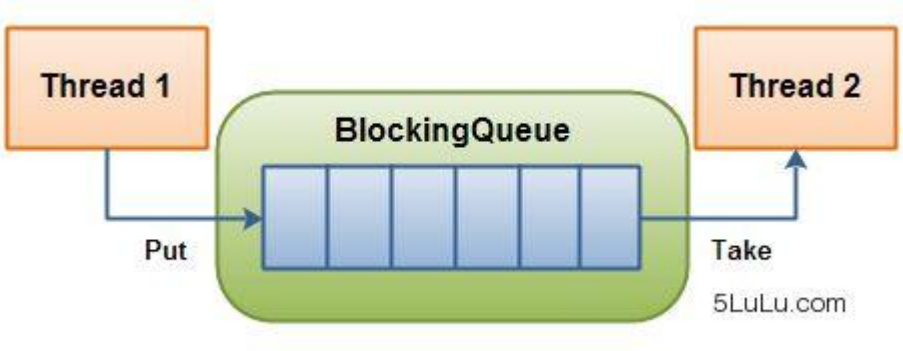
```
Thread-1 抢到车位...
Thread-2 抢到车位...
Thread-0 抢到车位...
Thread-2 离开车位
Thread-0 离开车位
Thread-3 抢到车位...
Thread-1 离开车位
Thread-4 抢到车位...
Thread-5 抢到车位...
Thread-3 离开车位
Thread-5 离开车位
Thread-4 离开车位
```

堵塞队列你知道吗？

阻塞队列有哪些

- ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）对元素进行排序。
- LinkedBlokcingQueue：是一个基于链表结构的阻塞队列，此队列按 FIFO（先进先出）对元素进行排序，吞吐量通常要高于 ArrayBlockingQueue。
- SynchronousQueue：是一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlokcingQueue。

什么是阻塞队列



- 阻塞队列，顾名思义，首先它是一个队列，而一个阻塞队列在数据结构中所起的作用大致如图所示：
- 当阻塞队列是空时，从队列中获取元素的操作将会被阻塞。
- 当阻塞队列是满时，往队列里添加元素的操作将会被阻塞。

- 核心方法

方法\行为	抛异常	特定的值	阻塞	超时
插入方法	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
移除方法		poll()、remove(o)	take()	poll(timeout, timeunit)
检查方法	element()	peek()		

- 行为解释：
    - 抛异常：如果操作不能马上进行，则抛出异常
    - 特定的值：如果操作不能马上进行，将会返回一个特殊的值，一般是 true 或者 false

        - 阻塞：如果操作不能马上进行，操作会被阻塞

        - 超时：如果操作不能马上进行，操作会被阻塞指定的时间，如果指定时间没执行，则返回一个特殊值，一般是 true 或者 false

    - 插入方法：

        - add(E e)：添加成功返回 true，失败抛 IllegalStateException 异常
        - offer(E e)：成功返回 true，如果此队列已满，则返回 false
        - put(E e)：将元素插入此队列的尾部，如果该队列已满，则一直阻塞

    - 删除方法：

        - remove(Object o) ：移除指定元素,成功返回 true，失败返回 false
        - poll()：获取并移除此队列的头元素，若队列为空，则返回 null
        - take()：获取并移除此队列头元素，若没有元素则一直阻塞

    - 检查方法：

        - element() ：获取但不移除此队列的头元素，没有元素则抛异常
        - peek() :获取但不移除此队列的头；若队列为空，则返回 null

SynchronousQueue

SynchronousQueue，实际上它不是一个真正的队列，因为它不会为队列中元素维护存储空间。与其他队列不同的是，它维护一组线程，这些线程在等待着把元素加入或移出队列。

```
1public class SynchronousQueueDemo {
2
3    public static void main(String[] args) {
```

```
        SynchronousQueue<Integer> synchronousQueue = new
SynchronousQueue<>();
        new Thread(() -> {
            try {
                synchronousQueue.put(1);
                Thread.sleep(3000);
                synchronousQueue.put(2);
                Thread.sleep(3000);
                synchronousQueue.put(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        new Thread(() -> {
            try {
                Integer val = synchronousQueue.take();
                System.out.println(val);
                Integer val2 = synchronousQueue.take();
                System.out.println(val2);
                Integer val3 = synchronousQueue.take();
                System.out.println(val3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

使用场景

- 生产者消费者模式
- 线程池
- 消息中间件

synchronized 和 Lock 有什么区别？

- 原始结构

    - synchronized 是关键字属于 JVM 层面，反应在字节码上是 monitorenter 和 monitorexit，其底层是通过 monitor 对象来完成，其实 wait/notify 等方法也是依赖 monitor 对象只有在同步快或方法中才能调用 wait/notify 等方法。
    - Lock 是具体类（java.util.concurrent.locks.Lock）是 api 层面的锁。
- 使用方法
    - synchronized 不需要用户手动去释放锁，当 synchronized 代码执行完后系统会自动让线程释放对锁的占用。
    - ReentrantLock 则需要用户手动的释放锁，若没有主动释放锁，可能导致出现死锁的现象，lock() 和 unlock() 方法需要配合 try/finally 语句来完成。
- 等待是否可中断
    - synchronized 不可中断，除非抛出异常或者正常运行完成。
    - ReentrantLock 可中断，设置超时方法 tryLock(long timeout, TimeUnit unit)，lockInterruptibly() 放代码块中，调用 interrupt() 方法可中断。
- 加锁是否公平
    - synchronized 非公平锁
    - ReentrantLock 默认非公平锁，构造方法中可以传入 boolean 值，true 为公平锁，false 为非公平锁。
- 锁可以绑定多个 Condition
    - synchronized 没有 Condition。
    - ReentrantLock 用来实现分组唤醒需要唤醒的线程们，可以精确唤醒，而不是像 synchronized 要么随机唤醒一个线程要么唤醒全部线程。

线程池使用过吗？谈谈对 ThreadPoolExector 的理解？

为什使用线程池，线程池的优势？

线程池用于多线程处理中，它可以根据系统的情况，可以有效控制线程执行的数量，优化运行效果。线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量，那么超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。

主要特点为：

- 线程复用
- 控制最大并发数量
- 管理线程

主要优点

- 降低资源消耗，通过重复利用已创建的线程来降低线程创建和销毁造成的消耗。
- 提高相应速度，当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性，线程是稀缺资源，如果无限制的创建，不仅仅会消耗系统资源，还会降低体统的稳定性，使用线程可以进行统一分配，调优和监控。

创建线程的几种方式

- 继承 Thread

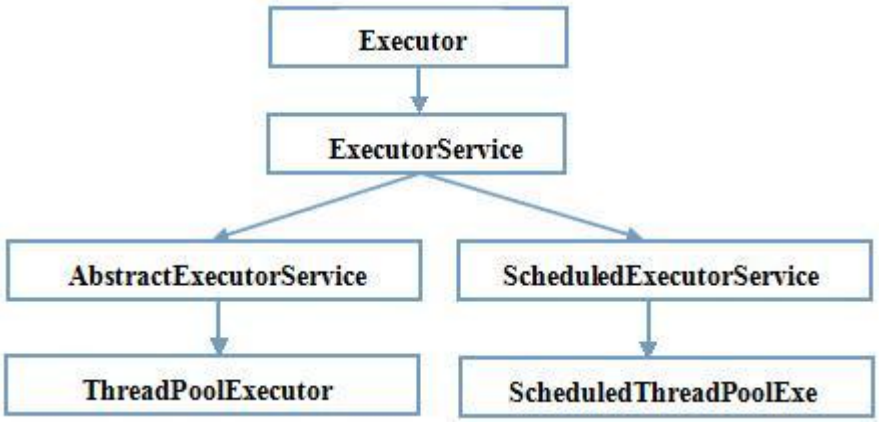
- 实现 Runnable 接口

- 实现 Callable

```
public class CallableDemo {
    public static void main(String[] args) throws
ExecutionException, InterruptedException {
        // 在 FutureTask 中传入 Callable 的实现类
        FutureTask<Integer> futureTask = new FutureTask<>(new
Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                return 666;
            }
        });
        // 把 futureTask 放入线程中
        new Thread(futureTask).start();
        // 获取结果
        Integer res = futureTask.get();
        System.out.println(res);
    }
}
```

线程池如果使用？

架构说明



编码实现

- Executors.newSingleThreadExecutor() :只有一个线程的线程池 ,因此所有提交的任务是顺序执行
- Executors.newCachedThreadPool() : 线程池里有很多线程需要同时执行 , 老的可用线程将被新的任务触发重新执行 , 如果线程超过 60 秒内没执行 , 那么将被终止并从池中删除
- Executors.newFixedThreadPool() :拥有固定线程数的线程池 ,如果没有任务执行，那么线程会一直等待
- Executors.newScheduledThreadPool()：用来调度即将执行的任务的线程池
- Executors.newWorkStealingPool()： newWorkStealingPool 适合使用在很耗时的操作，但是 newWorkStealingPool 不是 ThreadPoolExecutor 的扩展，它是新的线程池类 ForkJoinPool 的扩展，但是都是在统一的一个 Executors 类中实现，由于能够合理的使用 CPU 进行对任务操作（并行操作），所以适合使用在很耗时的任务中

ThreadPoolExecutor

ThreadPoolExecutor 作为 java.util.concurrent 包对外提供基础实现，以内部线程池的形式对外提供管理任务执行，线程调度，线程池管理等等服务。

线程池的几个重要参数介绍？

参数	作用
corePoolSize	核心线程池大小
maximumPoolSize	最大线程池大小
keepAliveTime	线程池中超过 corePoolSize 数目的空闲线程最大存活时间；可以 allowCoreThreadTimeOut(true) 使得核心线程有效时间
TimeUnit	keepAliveTime 时间单位
workQueue	阻塞任务队列
threadFactory	新建线程工厂
RejectedExecutionHandler	当提交任务数超过 maxmumPoolSize+workQueue 之和时，任务会交给 RejectedExecutionHandler 来处理

说说线程池的底层工作原理？

重点讲解：其中比较容易让人误解的是 :corePoolSize ,maximumPoolSize ,workQueue 之间关系。

1. 当线程池小于 corePoolSize 时 ,新提交任务将创建一个新线程执行任务 ,即使此时线程池中存在空闲线程。

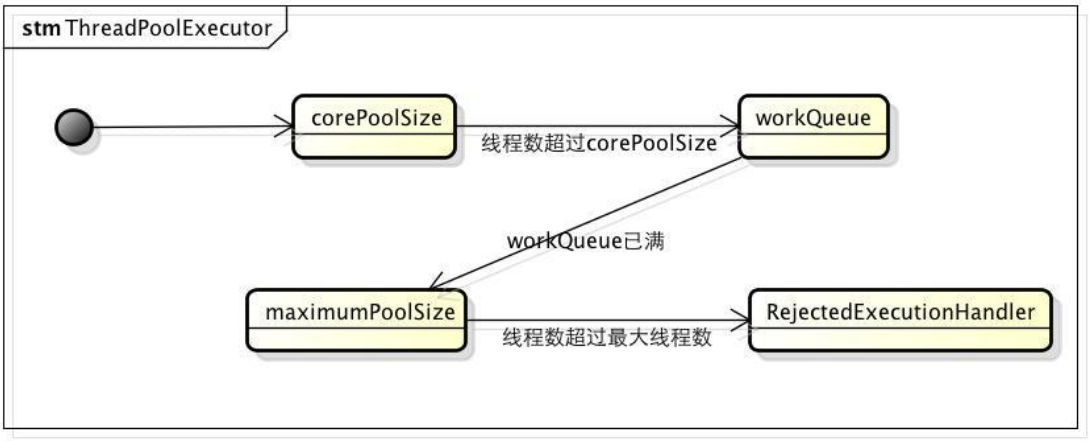
2. 当线程池达到 corePoolSize 时，新提交任务将被放入 workQueue 中，等待线程池中任务调度执行。

3. 当 workQueue 已满，且 maximumPoolSize 大于 corePoolSize 时，新提交任务会创建新线程执行任务。

4. 当提交任务数超过 maximumPoolSize 时，新提交任务由 RejectedExecutionHandler 处理。

5. 当线程池中超过 corePoolSize 线程，空闲时间达到 keepAliveTime 时，关闭空闲线程 。

6. 当设置 allowCoreThreadTimeOut(true) 时，线程池中 corePoolSize 线程空闲时间达到 keepAliveTime 也将关闭。



线程池用过吗？生产上你如何设置合理参数？

线程池的拒绝策略你谈谈？

- 是什么
  - 等待队列已经满了，再也塞不下新的任务，同时线程池中的线程数达到了最大线程数，无法继续为新任务服务。
- 拒绝策略
  - AbortPolicy：处理程序遭到拒绝将抛出运行时RejectedExecutionException
  - CallerRunsPolicy：线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。
  - DiscardPolicy：不能执行的任务将被删除
  - DiscardOldestPolicy：如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）

你在工作中单一的、固定数的和可变的三种创建线程池的方法，你用哪个多，超级大坑？

如果读者对 Java 中的阻塞队列有所了解的话，看到这里或许就能够明白原因了。

Java 中的 BlockingQueue 主要有两种实现，分别是 ArrayBlockingQueue 和 LinkedBlockingQueue。

ArrayBlockingQueue 是一个用数组实现的有界阻塞队列，必须设置容量。

LinkedBlockingQueue 是一个用链表实现的有界阻塞队列，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，最大长度为 Integer.MAX_VALUE。

这里的问题就出在：不设置的话，将是一个无边界的阻塞队列，最大长度为 Integer.MAX_VALUE。也就是说，如果我们不设置 LinkedBlockingQueue 的容量的话，其默认容量将会是 Integer.MAX_VALUE。

而 newFixedThreadPool 中创建 LinkedBlockingQueue 时，并未指定容量。此时，LinkedBlockingQueue 就是一个无边界队列，对于一个无边界队列来说，是可以不断的向队列中加入任务的，这种情况下就有可能因为任务过多而导致内存溢出问题。

上面提到的问题主要体现在 newFixedThreadPool 和 newSingleThreadExecutor 两个工厂方法上，并不是说 newCachedThreadPool 和 newScheduledThreadPool 这两个方法就安全了，这两种方式创建的最大线程数可能是 Integer.MAX_VALUE，而创建这么多线程，必然就有可能导致 OOM。

你在工作中是如何使用线程池的，是否自定义过线程池使用？

自定义线程池

```
public class ThreadPoolExecutorDemo {

    public static void main(String[] args) {
        Executor executor = new ThreadPoolExecutor(2, 3, 1L,
TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(5),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.DiscardPolicy());
    }
}
```

合理配置线程池你是如果考虑的？

- CPU 密集型
  - CPU 密集的意思是该任务需要大量的运算，而没有阻塞，CPU 一直全速运行。
  - CPU 密集型任务尽可能的少的线程数量，一般为 CPU 核数 + 1 个线程的线程池。
- IO 密集型
  - 由于 IO 密集型任务线程并不是一直在执行任务，可以多分配一点线程数，如 CPU * 2 。
  - 也可以使用公式 :CPU 核数 / (1 - 阻塞系数) ;其中阻塞系数在 0.8 ~ 0.9 之间。

死锁编码以及定位分析

- 产生死锁的原因
    - 死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种相互等待的现象，如果无外力的干涉那它们都将无法推进下去，如果系统的资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。
- 代码

```
public class DeadLockDemo {
    public static void main(String[] args) {
        String lockA = "lockA";
        String lockB = "lockB";

        DeadLockDemo deadLockDemo = new DeadLockDemo();
        Executor executor = Executors.newFixedThreadPool(2);
        executor.execute(() -> deadLockDemo.method(lockA, lockB));
        executor.execute(() -> deadLockDemo.method(lockB, lockA));

    }

    public void method(String lock1, String lock2) {
        synchronized (lock1) {
            System.out.println(Thread.currentThread().getName() + "-
-获取到：" + lock1 + "; 尝试获取：" + lock2);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (lock2) {
                System.out.println("获取到两把锁!");
            }
        }
    }
}
```

- 解决

- jps -l 命令查定位进程号

```
28519 org.jetbrains.jps.cmdline.Launcher
32376 com.intellij.idea.Main
28521 com.cuzz.thread.DeadLockDemo
27836 org.jetbrains.kotlin.daemon.KotlinCompileDaemon
28591 sun.tools.jps.Jps
```

- jstack 28521 找到死锁查看

```
2019-05-07 00:04:15
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.191-b12
mixed mode):

"Attach Listener" #13 daemon prio=9 os_prio=0
tid=0x00007f7acc001000 nid=0x702a waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
// ...
Found one Java-level deadlock:
=============================
"pool-1-thread-2":
  waiting to lock monitor 0x00007f7ad4006478 (object
0x00000000d71f60b0, a java.lang.String),
  which is held by "pool-1-thread-1"
"pool-1-thread-1":
  waiting to lock monitor 0x00007f7ad4003be8 (object
0x00000000d71f60e8, a java.lang.String),
  which is held by "pool-1-thread-2"

Java stack information for the threads listed above:
===================================================
"pool-1-thread-2":
        at
com.cuzz.thread.DeadLockDemo.method(DeadLockDemo.java:34)
        - waiting to lock <0x00000000d71f60b0> (a
java.lang.String)
        - locked <0x00000000d71f60e8> (a java.lang.String)
        at
com.cuzz.thread.DeadLockDemo.lambda$main$1(DeadLockDemo.java:21)
        at
com.cuzz.thread.DeadLockDemo$$Lambda$2/2074407503.run(Unknown
Source)
```

```
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExec
utor.java:1149)
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExe
cutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
"pool-1-thread-1":
        at
com.cuzz.thread.DeadLockDemo.method(DeadLockDemo.java:34)
        - waiting to lock <0x00000000d71f60e8> (a
java.lang.String)
        - locked <0x00000000d71f60b0> (a java.lang.String)
        at
com.cuzz.thread.DeadLockDemo.lambda$main$0(DeadLockDemo.java:20)
        at
com.cuzz.thread.DeadLockDemo$$Lambda$1/558638686.run(Unknown
Source)
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExec
utor.java:1149)
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExe
cutor.java:624)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

- 最后发现一个死锁。