

JUC核心知识点

- [JMM](#)
- [volatile关键字](#)
 - [可见性](#)
 - [原子性](#)
 - [有序性](#)
 - [哪些地方用到过volatile?](#)
 - [单例模式的安全问题](#)
- [CAS](#)
 - [CAS底层原理](#)
 - [CAS缺点](#)
- [ABA问题](#)
 - [AtomicReference](#)
 - [AtomicStampedReference和ABA问题的解决](#)
- [集合类不安全问题](#)
 - [List](#)
 - [CopyOnWriteArrayList](#)
 - [Set](#)
 - [HashSet和HashMap](#)
 - [Map](#)
- [Java锁](#)
 - [公平锁/非公平锁](#)
 - [可重入锁/递归锁](#)
 - [锁的配对](#)
 - [自旋锁](#)
 - [读写锁/独占/共享锁](#)
 - [Synchronized和Lock的区别](#)
- [CountDownLatch/CyclicBarrier/Semaphore](#)
 - [CountDownLatch](#)
 - [枚举类的使用](#)
 - [CyclicBarrier](#)
 - [Semaphore](#)
- [阻塞队列](#)
 - [SynchronousQueue](#)
- [Callable接口](#)
- [阻塞队列的应用——生产者消费者](#)
 - [传统模式](#)
 - [阻塞队列模式](#)
- [阻塞队列的应用——线程池](#)
 - [线程池基本概念](#)

- [线程池三种常用创建方式](#)
- [线程池创建的七个参数](#)
- [线程池底层原理](#)
- [线程池的拒绝策略](#)
- [实际生产使用哪一个线程池？](#)
 - [自定义线程池参数选择](#)
- [死锁编码和定位](#)

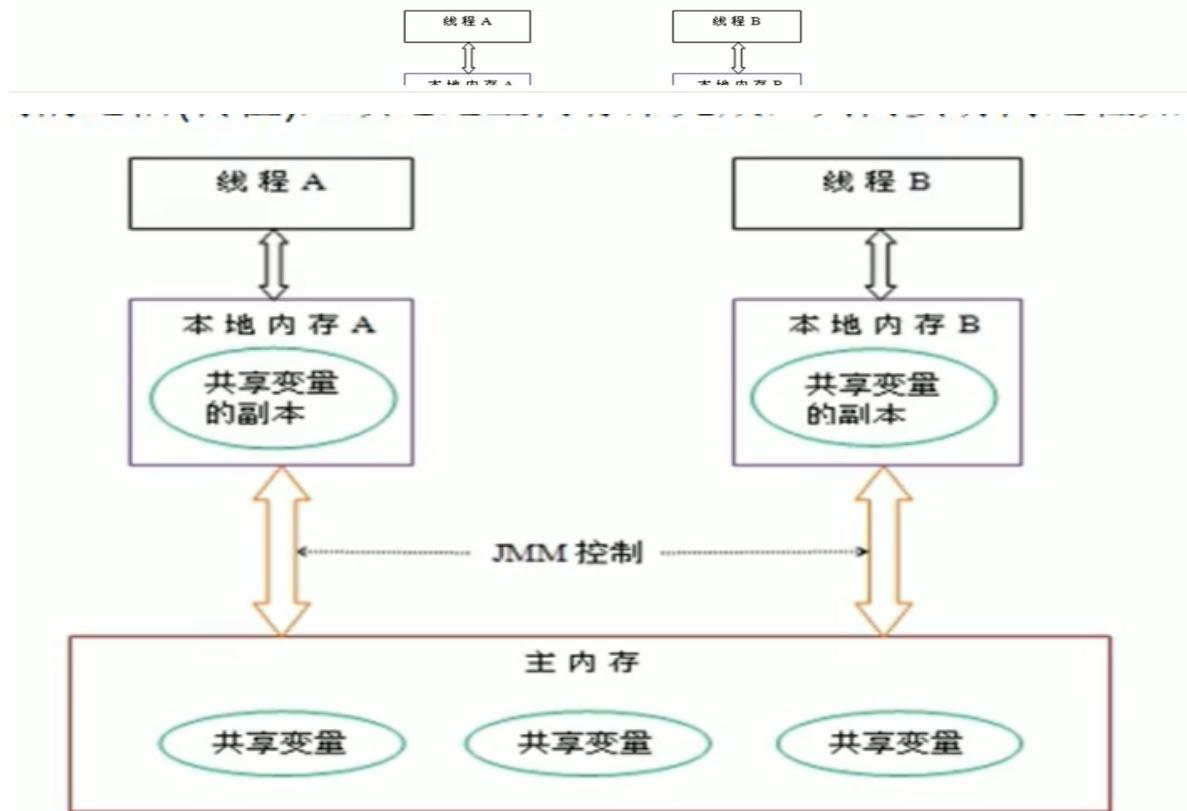
JMM

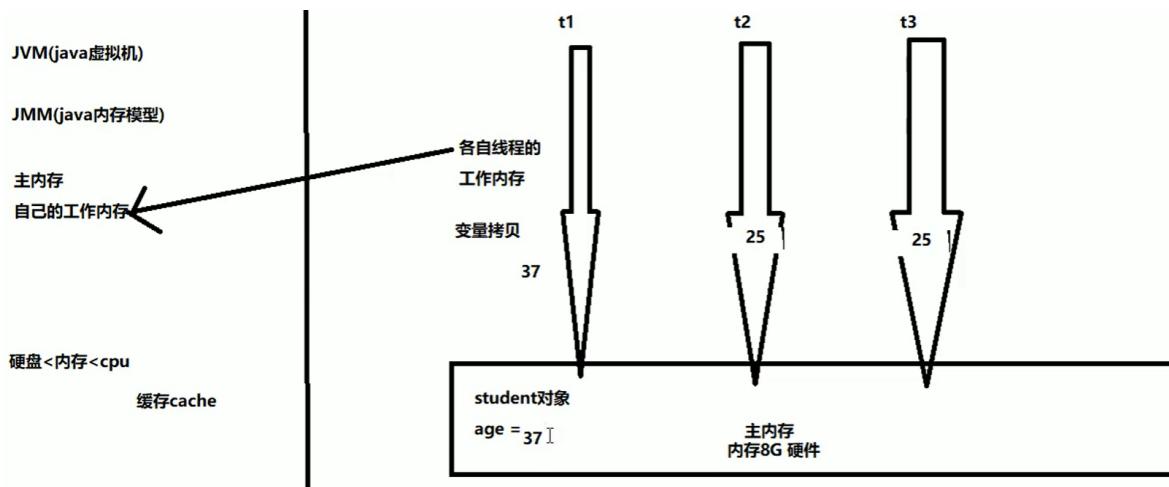
JMM(Java内存模型Java Memory Model, 简称JMM)本身是一种抽象的概念并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。

JMM关于同步的规定：

- 1 线程解锁前，必须把共享变量的值刷新回主内存
- 2 线程加锁前，必须读取主内存的最新值到自己的工作内存
- 3 加锁解锁是同一把锁

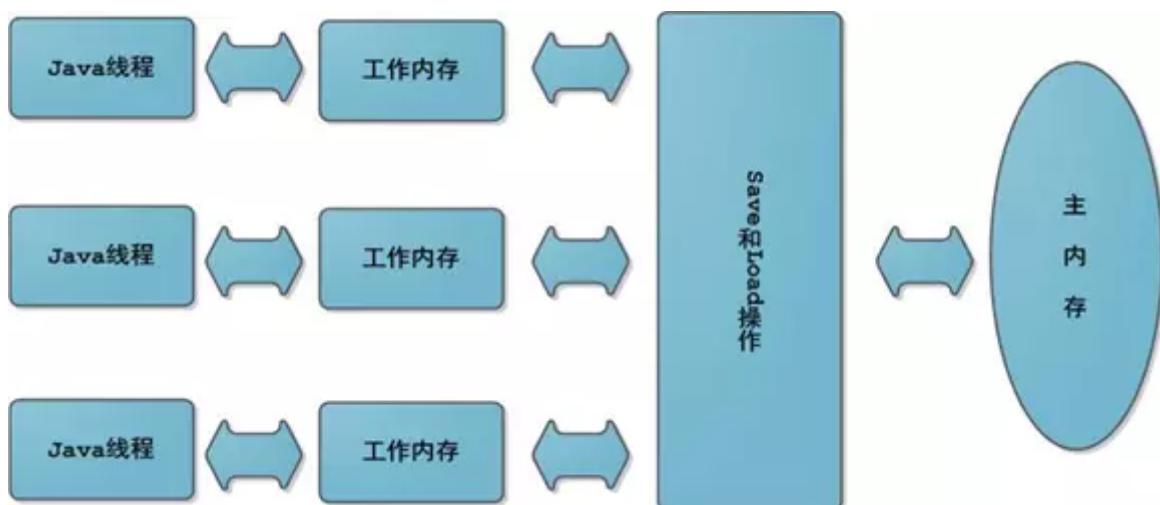
由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，工作内存是每个线程的私有数据区域，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后将变量写回主内存，不能直接操作主内存中的变量，各个线程中的工作内存中存储着主内存中的变量副本拷贝，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图：





JMM是指Java**内存模型**, 不是Java**内存布局**, 不是所谓的栈、堆、方法区。

每个Java线程都有自己的**工作内存**。操作数据, 首先从主内存中读, 得到一份拷贝, 操作完毕后再写回到主内存。



JMM可能带来**可见性**、**原子性和有序性**问题。所谓**可见性**, 就是某个线程对主内存内容的更改, 应该立刻通知到其它线程。**原子性**是指一个操作是不可分割的, 不能执行到一半, 就不执行了。**所谓有序性**, 就是指令是有序的, 不会被重排。

volatile关键字

`volatile`关键字是Java提供的一种**轻量级**同步机制。它能够保证**可见性和有序性**, 但是不能保证**原子性**。

可见性

可见性测试

```

1 class MyData{
2     int number=0;
3     //volatile int number=0;
4
5     AtomicInteger atomicInteger=new AtomicInteger();
6     public void setTo60(){
7         this.number=60;
8     }
9
10    //此时number前面已经加了volatile, 但是不保证原子性

```

```

11     public void addPlusPlus(){
12         number++;
13     }
14
15     public void addAtomic(){
16         atomicInteger.getAndIncrement();
17     }
18 }
19
20 //volatile可以保证可见性，及时通知其它线程主物理内存的值已被修改
21 private static void volatileVisibilityDemo() {
22     System.out.println("可见性测试");
23     MyData myData=new MyData(); //资源类
24     //启动一个线程操作共享数据
25     new Thread(() ->{
26         System.out.println(Thread.currentThread().getName()+"\t come in");
27         try {TimeUnit.SECONDS.sleep(3);myData.setTo60();
28             System.out.println(Thread.currentThread().getName()+"\t update
number value: "+myData.number);}catch (InterruptedException e)
29             {e.printStackTrace();}
30         }, "AAA").start();
31         while (myData.number==0){
32             //main线程持有共享数据的拷贝，一直为0
33         }
34         System.out.println(Thread.currentThread().getName()+"\t mission is over.
main get number value: "+myData.number);
35     }

```

MyData 类是资源类，一开始number变量没有用volatile修饰，所以程序运行的结果是：

```

1 可见性测试
2 AAA come in
3 AAA update number value: 60

```

虽然一个线程把number修改成了60，但是main线程持有的仍然是最开始的0，所以一直循环，程序不会结束。

如果对number添加了volatile修饰，运行结果是：

```

1 AAA come in
2 AAA update number value: 60
3 main      mission is over. main get number value: 60

```

可见某个线程对number的修改，会立刻反映到主内存上。

原子性

volatile并不能保证操作的原子性。这是因为，比如一条number++的操作，会形成3条指令。

```

1 getfield      //读
2 iconst_1      //++常量1
3 iadd          //加操作
4 putfield      //写操作

```

假设有3个线程，分别执行number++, 都先从主内存中拿到最开始的值，number=0，然后三个线程分别进行操作。假设线程0执行完毕，number=1，也立刻通知到了其它线程，但是此时线程1、2已经拿到了number=0，所以结果就是写覆盖，线程1、2将number变成1。

解决的方式就是：

1. 对 addPlusPlus() 方法加锁。
2. 使用 `java.util.concurrent.AtomicInteger` 类。

```
1 private static void atomicDemo() {  
2     System.out.println("原子性测试");  
3     MyData myData=new MyData();  
4     for (int i = 1; i <= 20; i++) {  
5         new Thread(()->{  
6             for (int j = 0; j <1000 ; j++) {  
7                 myData.addPlusPlus();  
8                 myData.addAtomic();  
9             }  
10        },String.valueOf(i)).start();  
11    }  
12    while (Thread.activeCount()>2){  
13        Thread.yield();  
14    }  
15    System.out.println(Thread.currentThread().getName()+"\t int type finally  
16    number value: "+myData.number);  
17    System.out.println(Thread.currentThread().getName()+"\t AtomicInteger  
18    type finally number value: "+myData.atomicInteger);  
19}
```

结果：可见，由于 `volatile` 不能保证原子性，出现了线程重复写的问题，最终结果比20000小。而 `AtomicInteger` 可以保证原子性。

```
1 原子性测试  
2 main      int type finally number value: 17542  
3 main      AtomicInteger type finally number value: 20000
```

有序性

有序性案例

`volatile`可以保证**有序性**，也就是防止**指令重排序**。所谓指令重排序，就是出于优化考虑，CPU执行指令的顺序跟程序员自己编写的顺序不一致。就好比一份试卷，题号是老师规定的，是程序员规定的，但是考生（CPU）可以先做选择，也可以先做填空。

```
1 int x = 11; //语句1  
2 int y = 12; //语句2  
3 x = x + 5; //语句3  
4 y = x * x; //语句4
```

以上例子，可能出现的执行顺序有1234、2134、1342，这三个都没有问题，最终结果都是x = 16，y=256。但是如果4开头，就有问题了，y=0。这个时候就**不需要指令重排序**。

`volatile`底层是用CPU的**内存屏障**（Memory Barrier）指令来实现的，有两个作用，一个是保证特定操作的顺序性，二是保证变量的可见性。在指令之间插入一条Memory Barrier指令，告诉编译器和CPU，在Memory Barrier指令之间的指令不能被重排序。

`volatile`实现禁止指令重排优化，从而避免多线程环境下程序出现乱序执行的现象

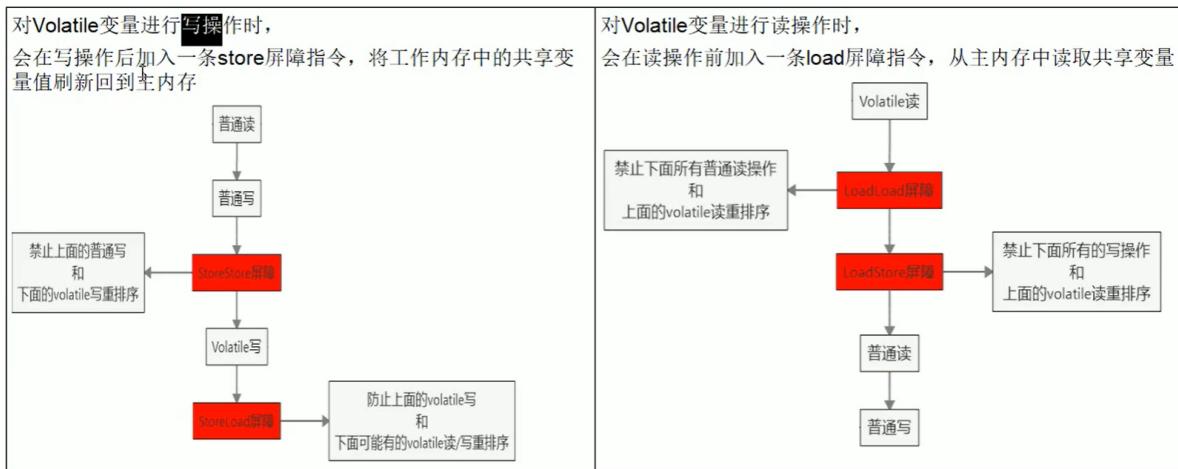


先了解一个概念，内存屏障(Memory Barrier) 又称内存栅栏，是一个CPU指令，它的作用有两个：

一是保证特定操作的执行顺序，

二是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）。

由于编译器和处理器都能执行指令重排优化。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和这条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的指令执行重排序优化。内存屏障另外一个作用是强制刷出各种CPU的缓存数据，因此任何CPU上的线程都能读取到这些数据的最新版本。



哪些地方用到过`volatile`？

单例模式的安全问题

常见的DCL (Double Check Lock) 模式虽然加了同步，但是在多线程下依然会有线程安全问题。

```

1 public class SingletonDemo {
2     private static SingletonDemo singletonDemo=null;
3     private SingletonDemo(){
4         System.out.println(Thread.currentThread().getName()+"\t 我是构造方
法");
5     }
6     //DCL模式 Double Check Lock 双端检索机制：在加锁前后都进行判断
7     public static SingletonDemo getInstance(){
8         if (singletonDemo==null){
9             synchronized (SingletonDemo.class){
10                 if (singletonDemo==null){
11                     singletonDemo=new SingletonDemo();
12                 }
13             }
14         }
15         return singletonDemo;
16     }
17
18     public static void main(String[] args) {
19         for (int i = 0; i < 10; i++) {
20             new Thread(()->{
21                 SingletonDemo.getInstance();
22                 ,String.valueOf(i+1)).start();
23             }
24         }
25     }
}
  
```

这个漏洞比较tricky，很难捕捉，但是是存在的。`instance=new SingletonDemo();`可以大致分为三步

```
1 | memory = allocate();      //1.分配内存
2 | instance(memory);        //2.初始化对象
3 | instance = memory;       //3.设置引用地址
```

其中2、3没有数据依赖关系，**可能发生重排**。如果发生，此时内存已经分配，那么 `instance=memory` 不为null。如果此时线程挂起，`instance(memory)` 还未执行，对象还未初始化。由于 `instance!=null`，所以两次判断都跳过，最后返回的 `instance` 没有任何内容，还没初始化。

解决的方法就是对 `singletondemo` 对象添加上 `volatile` 关键字，禁止指令重排。

CAS

CAS是指**Compare And Swap**，**比较并交换**，是一种很重要的同步思想。如果主内存的值跟期望值一样，那么就进行修改，否则一直重试，直到一致为止。

CAS是什么

CAS的全称为**Compare-And-Swap**，它是一条CPU并发原语。

它的功能是判断内存某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的。

CAS并发原语体现在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**。这是一种完全依赖于**硬件**的功能，通过它实现了原子操作。再次强调，由于CAS是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说**CAS**是一条CPU的原子指令，不会造成所谓的数据不一致问题。

```
1 | public class CASDemo {
2 |     public static void main(String[] args) {
3 |         AtomicInteger atomicInteger=new AtomicInteger(5);
4 |         System.out.println(atomicInteger.compareAndSet(5, 2019)+"\t current
5 | data : "+ atomicInteger.get());
6 |         //修改失败
7 |         System.out.println(atomicInteger.compareAndSet(5, 1024)+"\t current
8 | data : "+ atomicInteger.get());
    }
```

第一次修改，期望值为5，主内存也为5，修改成功，为2019。第二次修改，期望值为5，主内存为2019，修改失败。

查看 `AtomicInteger.getAndIncrement()` 方法，发现其没有加 `synchronized` 也实现了同步。这是为什么？

CAS底层原理

`AtomicInteger` 内部维护了 `volatile int value` 和 `private static final unsafe` 两个比较重要的参数。

```
1 | public final int getAndIncrement(){
2 |     return unsafe.getAndAddInt(this,valueOffset,1);
3 | }
```

`AtomicInteger.getAndIncrement()` 调用了 `Unsafe.getAndAddInt()` 方法。`Unsafe` 类的大部分方法都是 native 的，用来像C语言一样从底层操作内存。

```

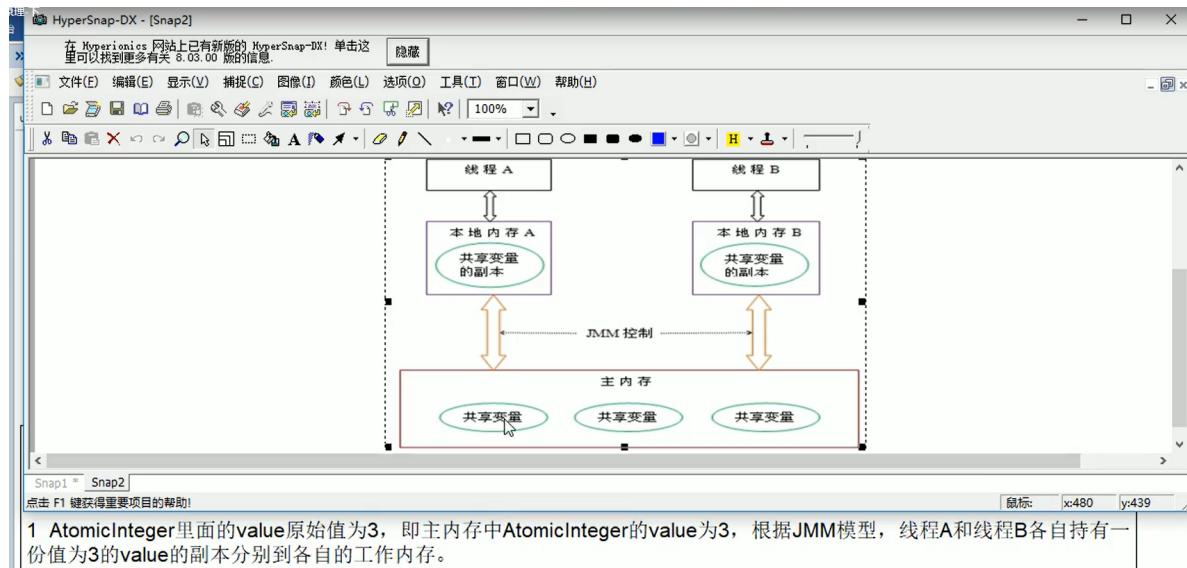
1 public final int getAddAndGet(Object var1, long var2, int var4) {
2     int var5;
3     do{
4         //这里的this就是unsafe对象，它能读到主内存的值
5         var5 = this.getIntVolatile(var1, var2);
6     } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
7     return var5;
8 }

```

var1 AtomicInteger对象本身。
 var2 该对象值得引用地址。
 var4 需要变动的数量。
 var5 是用过var1 var2 找出的主内存中真实的值。
 用该对象当前的值与var5比较：
 如果相同，更新var5+var4并且返回true，
 如果不同，继续取值然后再比较，直到更新完成。

这个方法的var1和var2，就是根据**对象和内存偏移量**得到**在主内存的快照值**var5。然后**compareAndSwapInt**方法通过var1和var2得到**当前主内存的实际值**。如果这个**实际值跟快照值相等**，那么就更新主内存的值为var5+var4。如果不等，那么就一直循环，一直获取快照，一直对比，直到实际值和快照值相等为止。

比如有A、B两个线程，一开始都从主内存中拷贝了原值为3，A线程执行到**var5=this.getIntVolatile**，即var5=3。此时A线程挂起，B修改原值为4，B线程执行完毕，由于加了volatile，所以这个修改是立即可见的。A线程被唤醒，执行**this.compareAndSwapInt()**方法，发现这个时候主内存的值不等于快照值3，所以继续循环，重新从主内存获取。



```

1 //valueoffset 内存偏移量，就是内存地址，执行完静态代码块后，这个valueOffset就是变量
2 //value在主内存的地址值，可以以此来定位是哪个变量
3
4 private static final Unsafe unsafe = Unsafe.getUnsafe();
5     private static final long valueOffset;
6
7     static {
8         try {
9             valueOffset = unsafe.objectFieldOffset
10            (AtomicInteger.class.getDeclaredField("value"));
11        } catch (Exception ex) { throw new Error(ex); }
12    }
13
14     private volatile int value;

```

CAS缺点

CAS实际上是一种自旋锁,

1. 一直循环，开销比较大（主内存的值一直被其他线程改动，那么当前线程要一直去循环取值比较）。
2. 只能保证一个变量的原子操作，多个变量依然要加锁。
3. 引出了ABA问题。

ABA问题

CAS会导致“ABA问题”。

CAS算法实现一个重要前提需要取出内存中某时刻的数据并在当下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程one从内存位置V中取出A，这时候另一个线程two也从内存中取出A，并且线程two进行了一些操作将值变成了B，然后线程two又将V位置的数据变成A，这时候线程one进行CAS操作发现内存中仍然是A，然后线程one操作成功。

尽管线程one的CAS操作成功，但是不代表这个过程就是没有问题的。

所谓ABA问题，就是比较并交换的循环，存在一个**时间差**，而这个时间差可能带来意想不到的问题。比如线程T1将值从A改为B，然后又从B改为A。线程T2看到的就是A，但是**却不知道这个A发生了更改**。尽管线程T2 CAS操作成功，但不代表就没有问题。

有的需求，比如CAS，**只注重头和尾**，只要首尾一致就接受。但是有的需求，还看重过程（挪用公款去炒股有还回来，有腐败），中间不能发生任何修改，这就引出了AtomicReference原子引用。

AtomicReference

AtomicInteger对整数进行原子操作，如果是一个POJO呢？可以用AtomicReference来包装这个POJO，使其操作原子化。

```

1 User user1 = new User("Jack",25);
2 User user2 = new User("Lucy",21);
3 AtomicReference<User> atomicReference = new AtomicReference<>();
4 atomicReference.set(user1);
5 System.out.println(atomicReference.compareAndSet(user1,user2)); // true
6 System.out.println(atomicReference.compareAndSet(user1,user2)); //false

```

AtomicStampedReference和ABA问题的解决

使用 `AtomicStampedReference` 类可以解决ABA问题。这个类维护了一个“**版本号**”Stamp，在进行CAS操作的时候，不仅要比较当前值，还要比较**版本号**。只有两者都相等，才执行更新操作。

```
1 | AtomicStampedReference.compareAndSet(expectedReference,newReference,oldStamp,  
newStamp);
```

详见[ABADemo](#)。

集合类不安全问题

List

`ArrayList` 不是线程安全类，在多线程同时写的情况下，会抛出 `java.util.ConcurrentModificationException` 异常。

```
1 | private static void listNotSafe() {  
2 |     List<String> list=new ArrayList<>();  
3 |     for (int i = 1; i <= 30; i++) {  
4 |         new Thread(() -> {  
5 |             list.add(UUID.randomUUID().toString().substring(0, 8));  
6 |             System.out.println(Thread.currentThread().getName() + "\t" +  
7 |             list);  
8 |         }, string.valueOf(i)).start();  
9 |     }  
10 | }
```

解决方法：

1. 使用 `vector` (`ArrayList` 所有方法加 `synchronized`，太重）。
2. 使用 `collections.synchronizedList()` 转换成线程安全类。
3. 使用 `java.concurrent.CopyOnWriteArrayList` (推荐)。

CopyOnWriteArrayList

这是JUC的类，通过**写时复制**来实现**读写分离**。比如其 `add()` 方法，就是先**复制**一个新数组，长度为原数组长度+1，然后将新数组最后一个元素设为添加的元素。

```
1 | public boolean add(E e) {  
2 |     final ReentrantLock lock = this.lock;  
3 |     lock.lock();  
4 |     try {  
5 |         //得到旧数组  
6 |         Object[] elements = getArray();  
7 |         int len = elements.length;  
8 |         //复制新数组  
9 |         Object[] newElements = Arrays.copyOf(elements, len + 1);  
10 |        //设置新元素  
11 |        newElements[len] = e;  
12 |        //设置新数组  
13 |        setArray(newElements);  
14 |        return true;  
15 |    } finally {
```

```
16     lock.unlock();
17 }
18 }
```

Set

跟List类似，`HashSet` 和 `TreeSet` 都不是线程安全的，与之对应的有 `CopyOnWriteSet` 这个线程安全类。这个类底层维护了一个 `CopyOnWriteArrayList` 数组。

```
1 private final CopyOnWriteArrayList<E> al;
2 public CopyOnWriteSet() {
3     al = new CopyOnWriteArrayList<E>();
4 }
```

HashSet和HashMap

`HashSet` 底层是用 `HashMap` 实现的。既然是用 `HashMap` 实现的，那 `HashMap.put()` 需要传**两个参数**，而 `HashSet.add()` 只传**一个参数**，这是为什么？实际上 `HashSet.add()` 就是调用的 `HashMap.put()`，只不过 `Value` 被写死了，是一个 `private static final Object` 对象。

Map

`HashMap` 不是线程安全的，`Hashtable` 是线程安全的，但是跟 `Vector` 类似，太重量级。所以也有类似 `CopyOnWriteMap`，只不过叫 `ConcurrentHashMap`。

关于集合不安全类请看[ContainerNotSafeDemo](#)。

Java锁

公平锁/非公平锁

概念：所谓**公平锁**，就是多个线程按照**申请锁的顺序**来获取锁，类似排队，先到先得。而**非公平锁**，则是多个线程抢夺锁，会导致**优先级反转或饥饿现象**。

区别：公平锁在获取锁时先查看此锁维护的**等待队列**，**为空**或者当前线程是等待队列的**队首**，则直接占有锁，否则插入到等待队列，FIFO原则。非公平锁比较粗鲁，上来直接**先尝试占有锁**，失败则采用公平锁方式。非公平锁的优点是**吞吐量**比公平锁更大。

`synchronized` 和 `juc.ReentrantLock` 默认都是**非公平锁**。`ReentrantLock` 在构造的时候传入 `true` 则是**公平锁**。

可重入锁/递归锁

可重入锁又叫递归锁，指的同一个线程在外层方法获得锁时，进入**内层方法**会自动获取锁。也就是说，线程可以进入任何一个它已经拥有锁的代码块。比如 `get` 方法里面有 `set` 方法，两个方法都有同一把锁，得到了 `get` 的锁，就自动得到了 `set` 的锁。

就像有了家门的锁，厕所、书房、厨房就为你敞开了一样。可重入锁可以**避免死锁**的问题。

详见[ReentrantLockDemo](#)。

锁的配对

锁之间要配对，加了几把锁，最后就得解开几把锁，下面的代码编译和运行都没有任何问题。但锁的数量不匹配会导致死循环。

```
1 lock.lock();
2 lock.lock();
3 try{
4     someAction();
5 }finally{
6     lock.unlock();
7 }
```

自旋锁

所谓自旋锁，就是尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取。自己在那儿一直循环获取，就像“**自旋**”一样。这样的好处是减少线程切换的上下文开销，缺点是会消耗CPU。CAS底层的 `getAndAddInt` 就是**自旋锁**思想。

```
1 //跟CAS类似，一直循环比较。
2 while (!atomicReference.compareAndSet(null, thread)) { }
```

详见[SpinLockDemo](#)。

读写锁/独占/共享锁

读锁是共享的，写锁是独占的。 `juc.ReentrantLock` 和 `synchronized` 都是**独占锁**，独占锁就是一个锁只能被一个线程所持有。有的时候，需要**读写分离**，那么就要引入读写锁，即 `juc.ReentrantReadWriteLock`。

比如缓存，就需要读写锁来控制。缓存就是一个键值对，以下Demo模拟了缓存的读写操作，读的 `get` 方法使用了 `ReentrantReadWriteLock.ReadLock()`，写的 `put` 方法使用了 `ReentrantReadWriteLock.writeLock()`。这样避免了写被打断，实现了多个线程同时读。

[ReadWriteLockDemo](#)

Synchronized和Lock的区别

`synchronized` 关键字和 `java.util.concurrent.locks.Lock` 都能加锁，两者有什么区别呢？

- 原始构成：** `sync` 是JVM层面的，底层通过 `monitorenter` 和 `monitorexit` 来实现的。`Lock` 是JDK API层面的。（`sync` 一个enter会有两个exit，一个是正常退出，一个是异常退出）
- 使用方法：** `sync` 不需要手动释放锁，而 `Lock` 需要手动释放。
- 是否可中断：** `sync` 不可中断，除非抛出异常或者正常运行完成。`Lock` 是可中断的，通过调用 `interrupt()` 方法。
- 是否为公平锁：** `sync` 只能是非公平锁，而 `Lock` 既能是公平锁，又能是非公平锁。
- 绑定多个条件：** `sync` 不能，只能随机唤醒。而 `Lock` 可以通过 `Condition` 来绑定多个条件，精确唤醒。

CountDownLatch/CyclicBarrier/Semaphore

CountDownLatch

CountDownLatch 内部维护了一个计数器，只有当计数器==0时，某些线程才会停止阻塞，开始执行。

CountDownLatch 主要有两个方法，`countDown()` 来让计数器-1，`await()` 来让线程阻塞。当 `count==0` 时，阻塞线程自动唤醒。

案例一班长关门： main线程是班长，6个线程是学生。只有6个线程运行完毕，都离开教室后，main线程班长才会关教室门。

案例二秦灭六国： 只有6国都被灭亡后（执行完毕），main线程才会显示“秦国一统天下”。

枚举类的使用

在**案例二**中会使用到枚举类，因为灭六国，循环6次，想根据 `i` 的值来确定输出什么国，比如1代表楚国，2代表赵国。如果用判断则十分繁杂，而枚举类可以简化操作。

枚举类就像一个**简化的数据库**，枚举类名就像数据库名，枚举的项目就像数据表，枚举的属性就像表的字段。

关于 CountDownLatch 和枚举类的使用，请看[CountDownLatchDemo](#)。

CyclicBarrier

CountDownLatch 是减，而 CyclicBarrier 是加，理解了 CountDownLatch，CyclicBarrier 就很容易。比如召集7颗龙珠才能召唤神龙，详见[CyclicBarrierDemo](#)。

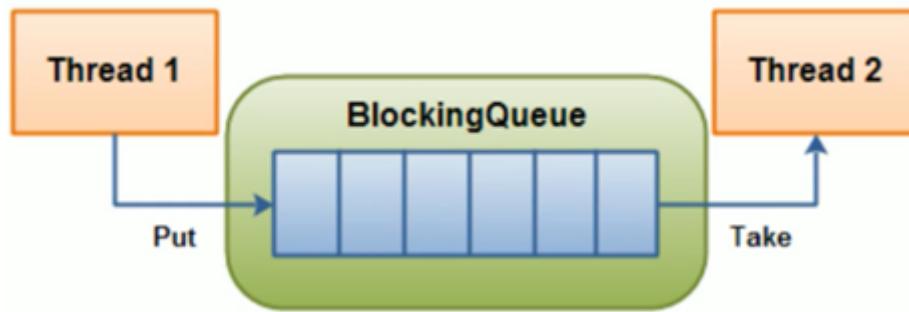
Semaphore

CountDownLatch 的问题是**不能复用**。比如 `count=3`，那么加到3，就不能继续操作了。而 Semaphore 可以解决这个问题，比如6辆车3个停车位，对于 CountDownLatch 只能停3辆车，而 Semaphore 可以停6辆车，车位空出来后，其它车可以占有，这就涉及到了 `Semaphore.acquire()` 和 `Semaphore.release()` 方法。

```
1  Semaphore semaphore=new Semaphore(3);
2  for (int i = 1; i <=6 ; i++) {
3      new Thread()->{
4          try {
5              //占有资源
6              semaphore.acquire();
7              System.out.println(Thread.currentThread().getName()+"\t抢到车位");
8              try{ TimeUnit.SECONDS.sleep(3);} catch (Exception e)
9 {e.printStackTrace(); }
10             System.out.println(Thread.currentThread().getName()+"\t停车3秒后离开车位");
11         }
12         catch (InterruptedException e) {e.printStackTrace();}
13         //释放资源
14         finally {semaphore.release();}
15     },String.valueOf(i)).start();
16 }
```

阻塞队列

概念：当阻塞队列为空时，获取（take）操作是阻塞的；当阻塞队列为满时，添加（put）操作是阻塞的。



好处：阻塞队列不用手动控制什么时候该被阻塞，什么时候该被唤醒，简化了操作。

体系： collection → Queue → BlockingQueue → 七个阻塞队列实现类。

类名	作用
ArrayBlockingQueue	由数组构成的 有界 阻塞队列
LinkedBlockingQueue	由链表构成的 有界 阻塞队列
PriorityBlockingQueue	支持优先级排序的 无界 阻塞队列
DelayQueue	支持优先级的延迟 无界 阻塞队列
SynchronousQueue	单个元素的阻塞队列
LinkedTransferQueue	由链表构成的 无界 阻塞队列
LinkedBlockingDeque	由链表构成的 双向 阻塞队列

粗体标记的三个用得比较多，许多消息中间件底层就是用它们实现的。

需要注意的是 `LinkedBlockingQueue` 虽然是有界的，但有个巨坑，其默认大小是 `Integer.MAX_VALUE`，高达21亿，一般情况下内存早爆了（在线程池的 `ThreadPoolExecutor` 有体现）。

API： 抛出异常是指当队列满时，再次插入会抛出异常；返回布尔是指当队列满时，再次插入会返回 false；阻塞是指当队列满时，再次插入会被阻塞，直到队列取出一个元素，才能插入。超时是指当一个时限过后，才会插入或者取出。API使用见[BlockingQueueDemo](#)。

方法类型	抛出异常	返回布尔	阻塞	超时
插入	<code>add(E e)</code>	<code>offer(E e)</code>	<code>put(E e)</code>	<code>offer(E e,Time,TimeUnit)</code>
取出	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(Time,TimeUnit)</code>
队首	<code>element()</code>	<code>peek()</code>	无	无

SynchronousQueue

队列只有一个元素，如果想插入多个，必须等队列元素取出后，才能插入，只能有一个“坑位”，用一个插一个，详见[SynchronousQueueDemo](#)。

Callable接口

与Runnable的区别：

1. Callable带返回值。
2. 会抛出异常。
3. 覆写 call() 方法，而不是 run() 方法。

Callable接口的使用：

```
1 public class CallableDemo {  
2     //实现callable接口  
3     class MyThread implements Callable<Integer> {  
4         @Override  
5         public Integer call() throws Exception {  
6             System.out.println("callable come in ...");  
7             return 1024;  
8         }  
9     }  
10    public static void main(String[] args) throws ExecutionException,  
11        InterruptedException {  
12        //创建FutureTask类, 接受MyThread。  
13        FutureTask<Integer> futureTask = new FutureTask<>(new MyThread());  
14        //将FutureTask对象放到Thread类的构造器里面。  
15        new Thread(futureTask, "AA").start();  
16        int result01 = 100;  
17        //用FutureTask的get方法得到返回值。  
18        int result02 = futureTask.get();  
19        System.out.println("result=" + (result01 + result02));  
20    }  
21 }
```

阻塞队列的应用——生产者消费者

传统模式

传统模式使用 Lock 来进行操作，需要手动加锁、解锁。详见[ProdConsTradiDemo](#)。

```
1 public void increment() throws InterruptedException {  
2     lock.lock();  
3     try {  
4         //1 判断 如果number=1, 那么就等待, 停止生产  
5         while (number != 0) {  
6             //等待, 不能生产  
7             condition.await();  
8         }  
9         //2 干活 否则, 进行生产  
10        number++;  
11        System.out.println(Thread.currentThread().getName() + "\t" +  
12            number);  
13        //3 通知唤醒 然后唤醒消费线程  
14        condition.signalAll();  
15    } catch (Exception e) {  
16        e.printStackTrace();  
17    } finally {  
18        //最后解锁  
19        lock.unlock();  
20    }  
21 }
```

```
19     }
20 }
```

阻塞队列模式

使用阻塞队列就不需要手动加锁了，详见[ProdConsBlockQueueDemo](#)。

```
1 public void myProd() throws Exception {
2     String data = null;
3     boolean retvalue;
4     while (FLAG) {
5         data = atomicInteger.incrementAndGet() + "";//++i
6         retvalue = blockingQueue.offer(data, 2L, TimeUnit.SECONDS);
7         if (retvalue) {
8             System.out.println(Thread.currentThread().getName() + "\t" + "插入队列" + data + "成功");
9         } else {
10             System.out.println(Thread.currentThread().getName() + "\t" + "插入队列" + data + "失败");
11         }
12         TimeUnit.SECONDS.sleep(1);
13     }
14     System.out.println(Thread.currentThread().getName() + "\tFLAG==false, 停止生产");
15 }
```

阻塞队列的应用——线程池

线程池基本概念

概念：线程池主要是控制运行线程的数量，将待处理任务放到等待队列，然后创建线程执行这些任务。如果超过了最大线程数，则等待。

优点：

1. 线程复用：不用一直new新线程，重复利用已经创建的线程来降低线程的创建和销毁开销，节省系统资源。
2. 提高响应速度：当任务达到时，不用创建新的线程，直接利用线程池的线程。
3. 管理线程：可以控制最大并发数，控制线程的创建等。

体系： Executor → ExecutorService → AbstractExecutorService → ThreadPoolExecutor。

ThreadPoolExecutor 是线程池创建的核心类。类似 Arrays、Collections 工具类，Executor 也有自己的工具类 Executors。

线程池三种常用创建方式

newFixedThreadPool： 使用 LinkedBlockingQueue 实现，定长线程池。

```
1 public static ExecutorService newFixedThreadPool(int nThreads) {
2     return new ThreadPoolExecutor(nThreads, nThreads,
3                                   0L, TimeUnit.MILLISECONDS,
4                                   new LinkedBlockingQueue<Runnable>());
5 }
```

newSingleThreadExecutor： 使用 LinkedBlockingQueue 实现，一池只有一个线程。

```
1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1,
3             1,
4             0L, TimeUnit.MILLISECONDS,
5             new LinkedBlockingQueue<Runnable>()));
}
```

newCachedThreadPool: 使用 `SynchronousQueue` 实现，变长线程池。

```
1 public static ExecutorService newCachedThreadPool() {
2     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
3             60L, TimeUnit.SECONDS,
4             new SynchronousQueue<Runnable>());
5 }
```

线程池创建的七个参数

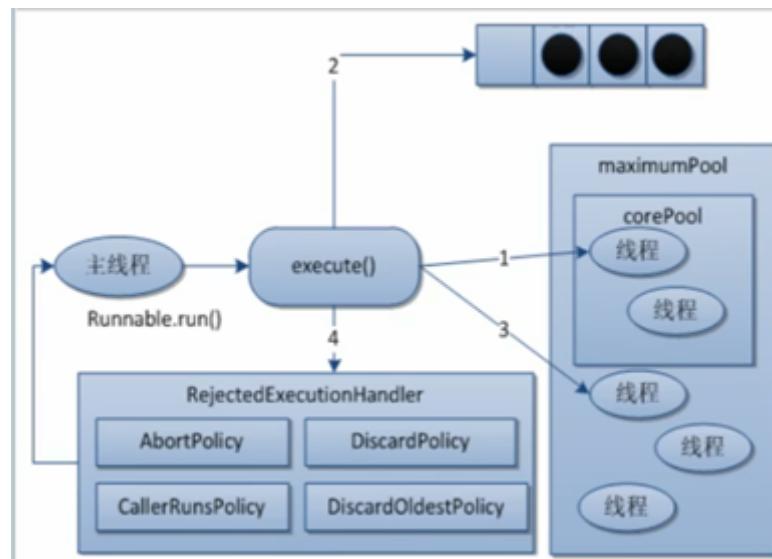
参数	意义
corePoolSize	线程池常驻核心线程数
maximumPoolSize	能够容纳的最大线程数
keepAliveTime	空闲线程存活时间
unit	存活时间单位
workQueue	存放提交但未执行任务的队列
threadFactory	创建线程的工厂类
handler	等待队列满后的拒绝策略

理解: 线程池的创建参数，就像一个**银行**。

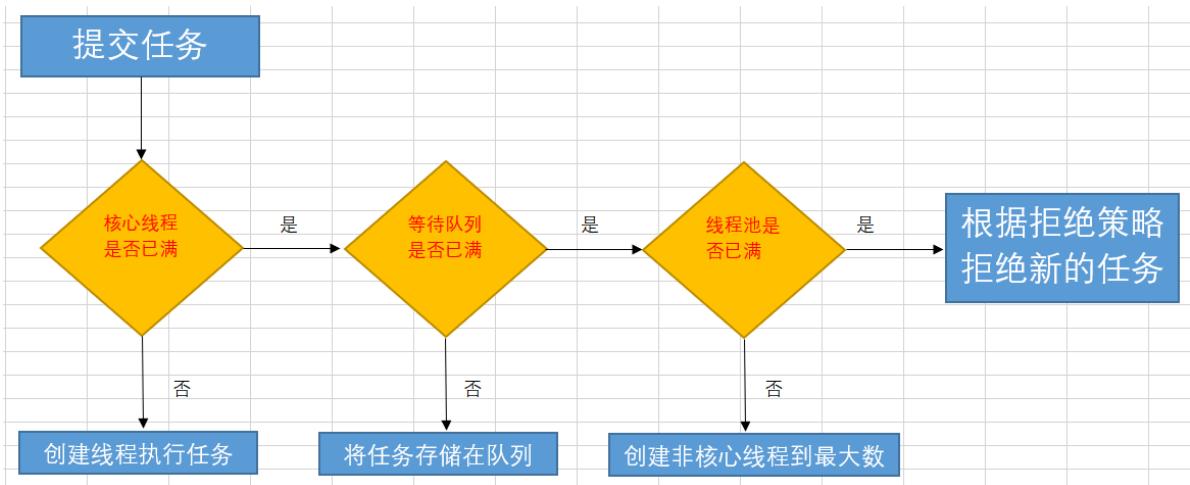
`corePoolSize` 就像银行的“**当值窗口**”，比如今天有**2位柜员**在受理**客户请求**（任务）。如果超过2个客户，那么新的客户就会在**等候区**（等待队列 `workQueue`）等待。当**等候区**也满了，这个时候就要开启“**加班窗口**”，让其它3位柜员来加班，此时达到**最大窗口** `maximumPoolSize`，为5个。如果开启了所有窗口，等候区依然满员，此时就应该启动“**拒绝策略**” `handler`，告诉不断涌入的客户，叫他们不要进入，已经爆满了。由于不再涌入新客户，办完事的客户增多，窗口开始空闲，这个时候就通过 `keepAliveTime` 将多余的3个“加班窗口”取消，恢复到2个“当值窗口”。

线程池底层原理

原理图: 上面银行的例子，实际上就是线程池的工作原理。



流程图：



新任务到达→

如果正在运行的线程数小于 `corePoolSize`，创建核心线程；大于等于 `corePoolSize`，放入等待队列。

如果等待队列已满，但正在运行的线程数小于 `maximumPoolSize`，创建非核心线程；大于等于 `maximumPoolSize`，启动拒绝策略。

当一个线程无事可做一段时间 `keepAliveTime` 后，如果正在运行的线程数大于 `corePoolSize`，则关闭非核心线程。

线程池的拒绝策略

当等待队列满时，且达到最大线程数，再有新任务到来，就需要启动拒绝策略。JDK提供了四种拒绝策略，分别是。

1. **AbortPolicy**: 默认的策略，直接抛出 `RejectedExecutionException` 异常，阻止系统正常运行。
2. **CallerRunsPolicy**: 既不会抛出异常，也不会终止任务，而是将任务返回给调用者。
3. **DiscardOldestPolicy**: 抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试再次提交任务。
4. **DiscardPolicy**: 直接丢弃任务，不做任何处理。

实际生产使用哪一个线程池？

单一、可变、定长都不用！原因就是 `FixedThreadPool` 和 `SingleThreadExecutor` 底层都是用 `LinkedBlockingQueue` 实现的，这个队列最大长度为 `Integer.MAX_VALUE`，显然会导致OOM。所以实际生产一般自己通过 `ThreadPoolExecutor` 的7个参数，自定义线程池。

```
1 ExecutorService threadPool=new ThreadPoolExecutor(2,5,
2                               1L,TimeUnit.SECONDS,
3                               new LinkedBlockingQueue<>(3),
4                               Executors.defaultThreadFactory(),
5                               new ThreadPoolExecutor.AbortPolicy());
```

自定义线程池参数选择

对于CPU密集型任务，最大线程数是CPU线程数+1。对于IO密集型任务，尽量多配点，可以是CPU线程数*2，或者CPU线程数/(1-阻塞系数)。阻塞系数在0.8~0.9，保守一点去0.9。（比如是4核心8线程，最大线程数为8/0.1=80）（看了其他帖子，说是核心线程数的配置，不是最大线程数：一般来说池中总线程数是核心池线程数量两倍，只要确保当核心池有线程停止时，核心池外能有线程进入核心池即可。我们所需要关心的主要是核心池线程的数量该如何设置。）

死锁编码和定位

主要是两个命令配合起来使用，定位死锁。

`jps`指令：`jps -l`可以查看运行的Java进程。

```
1 9688 thread.DeadLockDemo
2 12177 sun.tools.jps.Jps
```

`jstack`指令：`jstack pid`可以查看某个Java进程的堆栈信息，同时分析出死锁。

```
1 =====
2 "Thread AAA":
3     at xxxxx
4     - waiting to lock <0x000111>
5     - locked <0x000222>
6     at java.lang.Thread.run
7 "Thread BBB":
8     at xxxxx
9     - waiting to lock <0x000222>
10    - locked <0x000111>
11    at java.lang.Thread.run
12 Found 1 deadlock.
```