辅助资料

|  |
| --- |
| [6] Monitor Ctrl-Break （跟JVM 关系不大，他是 IDEA 通过反射的方式，开启一个随着我们运行的jvm进程开启与关闭的一个监听线程。）  [5] Attach Listener （附加监听器。 简单来说，他是jdk里边一个工具类提供的**jvm 进程之间通信**的工具。 cmd -- java -version; jvm -- jstack、jmap、dump） 进程间的通信。  开启我们这个线程的两个方式： 1. 通过jvm参数开启。-XX: StartAttachListener   1. 延迟开启： cmd -- java -version --> JVM 适时开启A L 线程   [4] Signal Dispatcher （信号分发器。 我们通过cmd 发送jstack，传到了jvm进程，这时候信号分发器就要发挥作用了。）  [3] Finalizer （JVM 垃圾回收相关的内容。此处只做简单的介绍。 1. 只有当开始一轮垃圾收集的时候，才会开始调用finalize方法。 2. daemon prio=10 高优先级的守护线程。 3. jvm在垃圾收集的时候，会将失去引用的对象封装到我们的 Fianlizer 对象（Reference）， 放入我们的 F-queue 队列中。由 Finalizer 线程执行inalize方法）  [2] Reference Handler （引用处理的线程。强，软，弱，虚。 -GC 有不同表现 - JVM深入分析）  [1] main 主线程 |

|  |
| --- |
| "Monitor Ctrl-Break" #6 daemon prio=5 os\_prio=0 tid=0x000000001a932800 nid=0x3bf8 runnable [0x000000001b96e000]  java.lang.Thread.State: RUNNABLE  at java.net.SocketInputStream.socketRead0(Native Method)  at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)  at java.net.SocketInputStream.read(SocketInputStream.java:170)  at java.net.SocketInputStream.read(SocketInputStream.java:141)  at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)  at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)  at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)  - locked <0x00000000d67070b8> (a java.io.InputStreamReader)  at java.io.InputStreamReader.read(InputStreamReader.java:184)  at java.io.BufferedReader.fill(BufferedReader.java:161)  at java.io.BufferedReader.readLine(BufferedReader.java:324)  - locked <0x00000000d67070b8> (a java.io.InputStreamReader)  at java.io.BufferedReader.readLine(BufferedReader.java:389)  at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)  "Attach Listener" #5 daemon prio=5 os\_prio=2 tid=0x0000000019a39000 nid=0x283c waiting on condition [0x0000000000000000] prio=5 延迟开启的问题。  java.lang.Thread.State: RUNNABLE  "Signal Dispatcher" #4 daemon prio=9 os\_prio=2 tid=0x0000000019a38800 nid=0x1dd4 runnable [0x0000000000000000]  java.lang.Thread.State: RUNNABLE  "Finalizer" #3 daemon prio=8 os\_prio=1 tid=0x0000000017ae8800 nid=0x3708 in Object.wait() [0x0000000019e9f000] （Finalizer 专注垃圾收集，垃圾收集 -- 并行收集，不阻碍用户线程，低优先级线程。 prio=8 他是一个守护线程啊。而且这个线程目前并没有真正的开启，不足以发生minorgc或者是 full gc、）  java.lang.Thread.State: WAITING (on object monitor)  at java.lang.Object.wait(Native Method)  - waiting on <0x00000000d6108e98> (a java.lang.ref.ReferenceQueue$Lock)  at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)  - locked <0x00000000d6108e98> (a java.lang.ref.ReferenceQueue$Lock)  at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)  at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)  "Reference Handler" #2 daemon prio=10 os\_prio=2 tid=0x0000000017ae1800 nid=0x1ee0 in Object.wait() [0x000000001999f000] （引用处理线程-GC相关线程：GC 很重要啊，优先级还挺高）  java.lang.Thread.State: WAITING (on object monitor)  at java.lang.Object.wait(Native Method)  - waiting on <0x00000000d6106b40> (a java.lang.ref.Reference$Lock)  at java.lang.Object.wait(Object.java:502)  at java.lang.ref.Reference.tryHandlePending(Reference.java:191)  - locked <0x00000000d6106b40> (a java.lang.ref.Reference$Lock)  at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)  "main" #1 prio=5 os\_prio=0 tid=0x00000000028f4800 nid=0x25ac waiting on condition [0x00000000028ef000] （操作系统面向的是JVM 进程，JVM 进程里面向的是 我们的main函数，。所以对于我们的操作系统如何看待我们的main函数优先级，无所谓。 只要os 给我们jvm进程足够公平的优先级就行。）  java.lang.Thread.State: TIMED\_WAITING (sleeping)  at java.lang.Thread.sleep(Native Method)  at com.boot.jdk.ThreadPrint.main(ThreadPrint.java:20) |

|  |
| --- |
| setPriority 这个方法，他是 jvm 提供的一个方法，并且能够调用 本地方法 setPriority0. 我们发现优先级貌似没有起作用，为什么？ 1. 我们现在的计算机都是多核的，t1，t2 会让哪个cpu处理不好说。由不同的cpu同时提供资源执行。 2. 优先级不代表先后顺序。哪怕你的优先级低，也是有可能先拿到我们的cpu时间片的，只不过这个时间片比高优先级的线程的时间片短。 优先级针对的是 cpu时间片的长短问题。 3. 目前工作中，实际项目里，不必要使用setPriority方法。我们现在都是用 hystrix， sential也好，一些开源的信号量控制工具，都能够实现线程资源的合理调度。这个 setPriority方法，很难控制。实际的运行环境太复杂。 |

|  |
| --- |
| **public final void** setDaemon(**boolean** on) {  checkAccess();  **if** (isAlive()) {  // 告诉我们，必须要先设置线程是否为守护线程，然后再调用start方法。如果你先调用start。 isAlive = true.  **throw new** IllegalThreadStateException();  }  **daemon** = on; } |

|  |
| --- |
| **"BlockedThread-2" #15 prio=5 os\_prio=0 tid=0x000000001b956000 nid=0x22d8 waiting for monitor entry [0x000000001d0be000] (发现死锁，一直不会释放的话)**  **java.lang.Thread.State: BLOCKED (on object monitor)**  **at com.boot.jdk.Blocked.run(ReadStackLog.java:45)**  **- waiting to lock <0x00000000d67bca20> (a java.lang.Class for com.boot.jdk.Blocked)**  **at java.lang.Thread.run(Thread.java:745)**  **"BlockedThread-1" #14 prio=5 os\_prio=0 tid=0x000000001b955000 nid=0x4c4c waiting on condition [0x000000001cfbf000]**  **java.lang.Thread.State: TIMED\_WAITING (sleeping)**  **at java.lang.Thread.sleep(Native Method)**  **at com.boot.jdk.Blocked.run(ReadStackLog.java:45)**  **- locked <0x00000000d67bca20> (a java.lang.Class for com.boot.jdk.Blocked)**  **at java.lang.Thread.run(Thread.java:745)**  "WaitingThread" #13 prio=5 os\_prio=0 tid=0x000000001b954800 nid=0x39cc in Object.wait() [0x000000001cebf000]  java.lang.Thread.State: WAITING (on object monitor)  at java.lang.Object.wait(Native Method)  - waiting on <0x00000000d67ba680> (a java.lang.Class for com.boot.jdk.Waiting)  at java.lang.Object.wait(Object.java:502)  at com.boot.jdk.Waiting.run(ReadStackLog.java:31)  - locked <0x00000000d67ba680> (a java.lang.Class for com.boot.jdk.Waiting)  at java.lang.Thread.run(Thread.java:745)  "TimeWaitingThread" #12 prio=5 os\_prio=0 tid=0x000000001b951800 nid=0x3820 waiting on condition [0x000000001cdbe000]  java.lang.Thread.State: TIMED\_WAITING (sleeping)  at java.lang.Thread.sleep(Native Method)  at com.boot.jdk.TimeWaiting.run(ReadStackLog.java:20)  at java.lang.Thread.run(Thread.java:745)  "Monitor Ctrl-Break" #6 daemon prio=5 os\_prio=0 tid=0x000000001b6c9000 nid=0x1210 runnable [0x000000001c6be000]  java.lang.Thread.State: RUNNABLE  at java.net.SocketInputStream.socketRead0(Native Method)  at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)  at java.net.SocketInputStream.read(SocketInputStream.java:170)  at java.net.SocketInputStream.read(SocketInputStream.java:141)  at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)  at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)  at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)  - locked <0x00000000d67070b8> (a java.io.InputStreamReader)  at java.io.InputStreamReader.read(InputStreamReader.java:184)  at java.io.BufferedReader.fill(BufferedReader.java:161)  at java.io.BufferedReader.readLine(BufferedReader.java:324)  - locked <0x00000000d67070b8> (a java.io.InputStreamReader)  at java.io.BufferedReader.readLine(BufferedReader.java:389)  at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)  "Attach Listener" #5 daemon prio=5 os\_prio=2 tid=0x000000001a72b000 nid=0x4ea8 waiting on condition [0x0000000000000000]  java.lang.Thread.State: RUNNABLE  "Signal Dispatcher" #4 daemon prio=9 os\_prio=2 tid=0x000000001a6d2800 nid=0x3d94 runnable [0x0000000000000000]  java.lang.Thread.State: RUNNABLE  "Finalizer" #3 daemon prio=8 os\_prio=1 tid=0x000000001a6b1800 nid=0x4254 in Object.wait() [0x000000001ab8f000] （只有进行垃圾收集的时候，才会被notify。 用到我们的 signal Dispatcher）  java.lang.Thread.State: WAITING (on object monitor)  at java.lang.Object.wait(Native Method)  - waiting on <0x00000000d6108e98> (a java.lang.ref.ReferenceQueue$Lock)  at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)  - locked <0x00000000d6108e98> (a java.lang.ref.ReferenceQueue$Lock)  at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)  at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)  "Reference Handler" #2 daemon prio=10 os\_prio=2 tid=0x00000000187c1000 nid=0x48a8 in Object.wait() [0x000000001a68f000] （引用处理线程。）  java.lang.Thread.State: WAITING (on object monitor)  at java.lang.Object.wait(Native Method)  - waiting on <0x00000000d6106b40> (a java.lang.ref.Reference$Lock)  at java.lang.Object.wait(Object.java:502)  at java.lang.ref.Reference.tryHandlePending(Reference.java:191)  - locked <0x00000000d6106b40> (a java.lang.ref.Reference$Lock)  at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)  Thread.start 之后，他会进入一个就绪状态，还没有分配到 cpu的执行权。 当cpu的时间片切换到他的时候，他才会开始执行，进入running状态 |

|  |
| --- |
| ***BLOCKED ： 只针对我们的 sync 锁。*** |

|  |
| --- |
| Thread.join 方法，他底层代码调用的是 Object的 wait方法（后边会带大家看 join方法的jdk源码）。那么想要唤醒join方法，就需要使用 object的notify以及 notifyall |

|  |
| --- |
| Thread.init方法代码片段：  **if** (g == **null**) {  **if** (security != **null**) {  g = security.getThreadGroup();  }   **if** (g == **null**) {  g = parent.getThreadGroup();  } }  尊重线程初始化穿入的threadgroup；次选System security mananger 的 tg；再次选 parent的 tg。  g.addUnstarted(); // NEW状态的线程，会添加到tg。  **this**.**daemon** = parent.isDaemon(); **this**.**priority** = parent.getPriority();  新的线程的属性依赖于 父类线程。  **private static synchronized long** nextThreadID() {  **return** ++*threadSeqNumber*; }  保证我们的tid的唯一性。  **public synchronized void** start() {}  避免多线程同时启动一个线程。IllegalThreadStateException  **try** {  // start0 完全执行完之前，线程处于 Ready  start0();  // 完成后，只要cpu分配执行权，我们的线程就进入了Running状态。  started = **true**; }  **try** {  **if** (!started) {  **group**.threadStartFailed(**this**);  }  } **catch** (Throwable ignore) {  */\* do nothing. If start0 threw a Throwable then  it will be passed up the call stack \*/*  *Start0 这个异常，会直接反馈给我们的调用线程。*  *Main函数里边的 thread.start方法。 防止我们的 thread.start方法感知不到异常，导致程序的错误的继续执行。* } |

|  |
| --- |
| Thread的 join方法（普通方法。）  **if** (millis == 0) {  **while** (isAlive()) {  wait(0);  //这个wait是调用的Object的，但是这是父类。其实这个wait方法前边有一个隐含的意义： this.wait(不是很准确)--》 当前的线程类（Thread类--有一个当前的线程）。 其实目前来看，这个是当前线程释放了cpu，而且是当前线程（Thread类）这个对象释放了锁。  } } |

|  |
| --- |
| Thread的join方法，释放的是当前调用 join方法的那个对象的锁。 |

|  |
| --- |
| Classfile /E:/IdeaWorkspace/demo/target/classes/com/boot/jdk/SyncUsingWay.class  Last modified 2022-5-8; size 917 bytes  MD5 checksum 9a53c6cd6851b0895ead00ce639fde81  Compiled from "SyncUsingWay.java"  public class com.boot.jdk.SyncUsingWay  minor version: 0  major version: 52  flags: ACC\_PUBLIC, ACC\_SUPER  Constant pool:  #1 = Methodref #8.#30 // java/lang/Object."<init>":()V  #2 = Fieldref #31.#32 // java/lang/System.out:Ljava/io/PrintStream;  #3 = String #16 // SyncMethod  #4 = Methodref #33.#34 // java/io/PrintStream.println:(Ljava/lang/String;)V  #5 = String #17 // StaticSyncMethod  #6 = String #18 // method  #7 = Class #35 // com/boot/jdk/SyncUsingWay  #8 = Class #36 // java/lang/Object  #9 = Utf8 <init>  #10 = Utf8 ()V  #11 = Utf8 Code  #12 = Utf8 LineNumberTable  #13 = Utf8 LocalVariableTable  #14 = Utf8 this  #15 = Utf8 Lcom/boot/jdk/SyncUsingWay;  #16 = Utf8 SyncMethod  #17 = Utf8 StaticSyncMethod  #18 = Utf8 method  #19 = Utf8 StackMapTable  #20 = Class #35 // com/boot/jdk/SyncUsingWay  #21 = Class #36 // java/lang/Object  #22 = Class #37 // java/lang/Throwable  #23 = Utf8 main  #24 = Utf8 ([Ljava/lang/String;)V  #25 = Utf8 args  #26 = Utf8 [Ljava/lang/String;  #27 = Utf8 MethodParameters  #28 = Utf8 SourceFile  #29 = Utf8 SyncUsingWay.java  #30 = NameAndType #9:#10 // "<init>":()V  #31 = Class #38 // java/lang/System  #32 = NameAndType #39:#40 // out:Ljava/io/PrintStream;  #33 = Class #41 // java/io/PrintStream  #34 = NameAndType #42:#43 // println:(Ljava/lang/String;)V  #35 = Utf8 com/boot/jdk/SyncUsingWay  #36 = Utf8 java/lang/Object  #37 = Utf8 java/lang/Throwable  #38 = Utf8 java/lang/System  #39 = Utf8 out  #40 = Utf8 Ljava/io/PrintStream;  #41 = Utf8 java/io/PrintStream  #42 = Utf8 println  #43 = Utf8 (Ljava/lang/String;)V  {  public com.boot.jdk.SyncUsingWay();  descriptor: ()V  flags: ACC\_PUBLIC  Code:  stack=1, locals=1, args\_size=1  0: aload\_0  1: invokespecial #1 // Method java/lang/Object."**<init>**":()V  4: return  LineNumberTable:  line 3: 0  LocalVariableTable:  Start Length Slot Name Signature  0 5 0 this Lcom/boot/jdk/SyncUsingWay;  public synchronized void SyncMethod();  descriptor: ()V  flags: ACC\_PUBLIC, **ACC\_SYNCHRONIZED**  Code:  stack=2, locals=1, args\_size=1  0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;  3: ldc #3 // String SyncMethod  5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V  8: return  LineNumberTable:  line 5: 0  line 6: 8  LocalVariableTable:  Start Length Slot Name Signature  0 9 0 this Lcom/boot/jdk/SyncUsingWay;  public static synchronized void StaticSyncMethod();  descriptor: ()V  flags: ACC\_PUBLIC, **ACC\_STATIC, ACC\_SYNCHRONIZED**  Code:  stack=2, locals=0, args\_size=0  0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;  3: ldc #5 // String StaticSyncMethod  5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V  8: return  LineNumberTable:  line 9: 0  line 10: 8  public void method();  descriptor: ()V  flags: ACC\_PUBLIC  Code:  stack=2, locals=3, args\_size=1  0: aload\_0  1: dup  2: astore\_1  **3: monitorenter // 进入同步代码块（进入临界范围内，锁的原子内部）**  4: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;  7: ldc #6 // String method  9: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V  12: aload\_1  **13: monitorexit // 正常退出同步代码块**  14: goto 22  17: astore\_2  18: aload\_1  **19: monitorexit //防止任何异常情况下，退出同步代码块。JVM 仍然可以释放锁**  20: aload\_2  21: athrow  22: return  **Exception table: //配合来了异常退出 monitorexit**  from to target type  4 14 17 any  17 20 17 any  LineNumberTable:  line 13: 0  line 14: 4  line 15: 12  line 16: 22  LocalVariableTable:  Start Length Slot Name Signature  0 23 0 this Lcom/boot/jdk/SyncUsingWay;  StackMapTable: number\_of\_entries = 2  frame\_type = 255 /\* full\_frame \*/  offset\_delta = 17  locals = [ class com/boot/jdk/SyncUsingWay, class java/lang/Object ]  stack = [ class java/lang/Throwable ]  frame\_type = 250 /\* chop \*/  offset\_delta = 4    SourceFile: "SyncUsingWay.java" |

|  |
| --- |
| 偏向锁使用的前提：  1. 至少JDK1.6 版本且开启了偏向锁配置。  偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟：-XX:BiasedLockingStartupDelay=0。如果你确定应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁：-XX:-UseBiasedLocking=false，那么程序默认会进入轻量级锁状态。  **2被加锁的对象，没有真正、或者隐式的调用父类 Object 里边的hashcode方法。（如果一旦调用了object的hashcode方法，那么我们的对象头里边就有真正的hashcode值了，如果偏向锁来进行markword的替换，至少要提供一个保存hashcode的地方吧？可惜的是，偏向锁并没有地方进行markword的保存，只有轻量级锁才会有“displace mark word”）**  **为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在对象头：存储线程id**  **和栈帧中的锁记录里： 线程有自己的栈帧，LOCK RECORD: 存储当前线程id**  **存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需简单地测试一下对象头的Mark Word里是否存储着指向**  **当前线程的偏向锁。：id的匹配**  **如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁）：如果没有设置，则使用CAS竞争锁；如果设置了，则尝试使用CAS将对象头的**  **偏向锁指向当前线程： 其实是cas竞争替换 线程id。** |

|  |
| --- |
| 偏向锁：   1. A线程获取偏向锁，并且A线程死亡退出。B线程争抢偏向锁，会直接升级当前对象的锁为轻量级锁。**这只是针对我们争抢了一次。** 2. A线程获取偏向锁，并且A线程没有释放偏向锁（），还在syhnc的代码块里边。B线程此时过来争抢偏向锁，会直接升级为重量级锁。 3. A线程获取偏向锁，并且A线程释放了锁，但是A线程并没有死亡还在活跃状态。B线程过来争抢，会直接升级为轻量级锁。   综上所述，当我们尝试第一次竞争偏向锁时，如果A线程已经死亡，升级为轻量级锁；如果A线程未死亡，并且未释放锁，直接升级为重量级锁；如果A线程未死亡，并且已经释放了锁，直接升级为轻量级锁。   1. A线程获取偏向锁，并且A线程没有释放偏向锁（），还在syhnc的代码块里边。B线程多次争抢锁，会在加锁过程中采用重量级锁；但是，一旦锁被释放，当前对象还是会以轻量级锁的初始状态执行。这块算是锁降级吗？不算。这个示例就是我们一些博客论坛里边的一些认为可以锁降级的示例。--- 锁升级是在线程运行过程中和争抢过程中的一种升级。这句话里一定要注意 中 这个字儿，很重要。我想请问，刚才我们演示的是在竞争中的锁降级吗？ |

|  |
| --- |
| 1. A线程获取偏向锁，并且A线程释放了锁，但是A线程并没有死亡还在活跃状态。B线程过来争抢。部分争抢会升级为轻量级锁；部分争抢会依旧保持偏向锁。 |

|  |
| --- |
| 批量重偏向： 当我们的一个对象，Object 类，在经过默认 20次的争抢的情况下，会将后边的所有争抢从新偏向争抢的线程。1. 当B线程争抢第 18 次的时候，触发了批量重偏向的阈值；在第十八次以及以后的争抢里，jvm会将线程偏向线程b，因为jvm认为，这个对象更加适合线程B |

|  |
| --- |
| 批量撤销：如果基于批量重偏向的基础上，还在继续进行争抢达到40次，并且有第三条线程C加入了，这个时候会触发批量撤销。JVM会标记该对象不能使用偏向锁，以后新创建的对象，直接以轻量级锁开始。 这个时候，才是真正的完成了锁升级。  真正的锁升级，是依赖于 class 的，而并不是依赖于 某一个 new出来的对象（偏向锁升级为轻量级锁）。  真正的锁升级，是依赖于 当前new出来的对象的（轻量级锁升级为重量级锁）  轻量级锁升级为重量级锁：这个时候，只要我们的线程发生了竞争，并且CAS替换失败，就会发起锁膨胀，升级为重量级锁（针对的是一个对象实例）。 |

|  |
| --- |
| 轻量级锁---重量级锁： 释放锁（前四步）并唤醒等待线程   1. 线程1 初始化monitor 对象； 2. 将状态设置为膨胀中（inflating）； 3. 将monitor里边的header属性，set称为对象的markword；（将自己lock record里边的存放的mark word的hashcode，分代年龄，是否为偏向锁 set 到 objectmonitor对象的header属性里） 4. 设置对象头为重量级锁状态（标记为改为00）；然后将前30位指向第1不他初始化的monitor 对象；（真正的锁升级是由线程1操控的） 5. 唤醒线程2； 6. 线程2 开始争抢重量级锁。（线程2就干了一件事儿，就是弄了一个临时的重量级锁指针吧？还不是最后的重量级锁指针。因为最后的重量级锁指针是线程1初始化的并且是线程1修改的。 而且，线程2被唤醒之后，还不一定能够抢到这个重量级锁。Sync是非公平锁。 线程2费力不讨好，但是线程2做了一件伟大的事情：他是锁升级的奠基者。） |

|  |
| --- |
| 同步队列Aqs 为什么在设置尾结点的时候需要使用CAS:  三个线程abc，a持有锁，bc竞争失败，需要添加到 AQS的同步队列的尾端。此时bc同时竞争tail节点，这个时候就是要保证线程安全性，正确的添加尾节点，需要使用cas操作。  同步队列设置 头节点，需要使用cas吗？  不需要，因为设置头节点的线程是已经获取锁成功的线程，这个时候只有一条线程获取锁成功了，所以直接普通**setHead**节点就可以了。没有竞争，就无需保证安全性。 |

|  |
| --- |
| ***SIGNAL：等待被通知状态，如果pre节点是这个状态，那么当前节点就会进行park操作***  *Cancelled ： 一个取消的线程状态。这个状态的线程会被移除* |

|  |
| --- |
| 原子性：无论你有多少个操作，只要我们将这部分操作通过加锁的形式进行锁定，那么就可以视为一个原子性操作。原子性操作 = 线程安全。 |

|  |
| --- |
| **对于原子更新数组类和原子更新对象引用类，他们实质上就是将我们的数组和对象的外层进行了一次 Atomic 的封装。所以，在进行替换的时候，其实是外层的这个封装保证了原子性。**  **想想我们的原子更新基本类型如：Integer。其实Integer不也是一个对象吗？也是在他的外层进行了一次封装。** |

|  |
| --- |
| DelayQueue s使用步骤：   1. 要放入queue的对象要实现 Delayed接口。   Type parameter 'java.lang.String' is not within its bound; should implement 'java.util.concurrent.Delayed'   1. getDelay： 判断是否达到了执行时间；换句话说，是否延时到期。 2. compareTo：比较优先级。时间优先级对比。 |

|  |
| --- |
| ThreadPoolExecutor采取上述步骤的总体设计思路，是为了在执行execute()方法时，尽可能地避免获取全局锁（那将会是一个严重的可伸缩瓶颈）  另外一种方式：就是：线程池为什么要有一个核心线程数和最大线程数的区分呢？   1. 核心线程数和最大线程数中间还有一个queue呢。如果核心线程数够用，不一定先要去创建其他的线程，因为需要看queue是否还能够继续容纳新的请求； 如果没有所谓的最大的线程数，只有一个核心线程数，那么当我们线程池的核心线程数配置小了的话，那么很多任务都会添加到queue里。如果queue满了，我们就没有其他的办法创建新的额外的线程了。 2. 如果仅仅依靠核心线程数，比如说我们将核心线程数的值设置的非常大，每次有新任务过来，都有可能在核心线程数里边创建新的线程，可能会造成全局锁的获取，导致性能一个瓶颈。   继续发问： 全局锁？在那块儿有这个逻辑？   1. 当我们的ThreadPoolExecutor进行execute方法执行的时候，如果当前的工作线程小于 Coresize，就会进行一个新的工作线程的添加，调用 addWorker 方法，这个方法里，当我们进行最终的worker.add的时候，是在一个 rentreenlock里执行的。也就是说，此处不允许并发添加新的worker。如果同时有多个线程进来，且都小于coresize，只能排队添加。 |

|  |
| --- |
| 1. LinkedBlockingQueue 到底是有界的还是无界的呀？   想解决这个问题，首先我们要知道无界和有界代表的意思是什么？ 有界队列是：创建队列的时候必须要传入一个指定的大小（构造函数上说）。 无界队列：创建的时候无需传入指定大小，就是无界队列。  那么LinkedBlockingQueue的默认大小是： Integer.***MAX\_VALUE。 说明他是一个有界队列吧？ 回答：不是。此处才能说明他是无界队列。*** |

|  |
| --- |
| 1. 对于fixed，single和 cache，我们在实际的工作中**最多使用到的（相对于fixed和cache）**是 single。一般情况下，对于并发量不高的场景，并且需要单条异步线程进行处理的时候，会使用一下 single。 真正的业务处理中，如果使用线程池，99%的情况下，需要自定义线程池（自己传入参数）。只有1%的情况会使用fixed，single和 cache。在这1%里，single稍微多一些。   Fixed和 single 容易造成我们的quque的消息无限积压，还会导致无法触发拒绝策略。所以一般没人用；  Cache，刚说的，可能高并发下，无法控制最高的线程创建数量，造成cpu和内存资源的消耗（消耗完）。 这种破线程池谁用？  这三个，仅仅是为了让我们创建线程池的时候方便一些，不代表他们很实用。 |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |