并发多线程的第二版Anki

多线程中 synchronized 锁升级的原理是什么?

- **synchronized 锁升级原理**:
  - 在锁对象的对象头里面有一个 threadid 字段，**在第一次访问的时候threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id**
  - 再次进入的时候会先判断threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，**如果不一致，则升级偏向锁为轻量级锁**
  - **通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁**，此过程就构成了 synchronized 锁的升级。
  - > 一个对象刚开始实例化的时候，没有任何线程来访问它的时候。它是可偏向的，意味着，它现在认为只可能有一个线程来访问它，**所以当第一个线程来访问它的时候，它会偏向这个线程**，并且使用CAS进行标记这个线程.此时，对象持有偏向锁。偏向第一个线程，**之后再次访问这个对象时，只需要对比是否是原来的那个线程，不需要再使用CAS在进行操作。**
    >
    > **一旦有第二个线程访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象时偏向状态，这时表明在这个对象上已经存在竞争了**，检查原来持有该对象锁的线程是否依然存活，**如果挂了，则可以将对象变为无锁状态**，然后重新偏向新的线程，**如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁**，（偏向锁就是这个时候升级为轻量级锁的）。如果不存在使用了，则可以将对象回复成无锁状态，然后重新偏向。
    >
    > 轻量级锁认为竞争存在，但是竞争的程度很轻，**一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（自旋）**，另一个线程就会释放锁。 **但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁**，重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。
- **锁的升级的目的**：锁升级是**为了减低了锁带来的性能消耗。**在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。
- **偏向锁**：它会偏向于第一个访问锁的线程，如果在运行过程中，**同步锁只有一个线程访问，不存在多线程争用的情况**，则线程是不需要触发同步的，减少加锁 / 解锁的一些CAS操作（比如等待队列的一些CAS操作），这种情况下，**就会给线程加一个偏向锁。** 如果**在运行过程中，遇到了其他线程抢占锁，则持有偏锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。**
- **轻量级锁：**是由偏向锁升级来的，偏向锁运行在一个线程进入同步块的情况下，**当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁**；此时两个线程争用锁对象，就会采取自旋的方式，自旋次数超过一定次数就会升为重量级锁
- **重量级锁：**是synchronized ，是 Java 虚拟机中最为基础的锁实现。在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。
- **自旋锁：**内核态与用户态的切换上不容易优化。但**通过自旋锁，可以减少线程阻塞造成的线程切换**（包括挂起线程和恢复线程）。

什么是自旋

- 很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。
- 既然synchronized 里面的代码执行得非常快，**不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。**如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。
- **忙循环**：就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了

synchronized可重入的原理

重入锁是指一个线程获取到该锁之后，该线程可以继续获得该锁。**底层原理维护一个计数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器减一，当计数器值为0时，表明该锁未被任何线程所持有，其它线程可以竞争获取锁。**

说一下 synchronized 底层实现原理?

- Synchronized的语义底层是通过一个monitor（监视器锁）的对象来完成，
- 每个对象有一个监视器锁(monitor)。每个Synchronized修饰过的代码当它的monitor被占用时就会处于锁定状态并且尝试获取monitor的所有权 ，过程：
    - 1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
    - 2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
    - 3、如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

线程的 sleep()方法和 yield()方法有什么区别?

- sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；**yield()方法只会给相同优先级或更高优先级的线程以运行的机会；**
- **线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；**
- sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；
- sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

如果你提交任务时，线程池队列已满，这时会发生什么

有俩种可能：

- **如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话**，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以乎认为是一个无穷大的队列，可以无限存

放任务

- **如果使用的是有界队列比如 ArrayBlockingQueue**，任务首先会被添加到ArrayBlockingQueue中，**ArrayBlockingQueue 满了，会根据maximumPoolSize的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是 AbortPolicy**

GC ROOTS有哪些

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 本地方法栈(Native 方法)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 所有被同步锁持有的对象

synchronized、volatile、CAS 比较

- synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。
- volatile 提供多线程共享变量可见性和禁止指令重排序优化。
- CAS 是基于冲突检测的乐观锁（非阻塞）

synchronized 和 Lock 有什么区别?

- 首先synchronized是Java内置关键字，在JVM层面，Lock是个Java类；
- synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。
- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；
- 而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

线程池中 submit() 和 execute() 方法有什么区别?

- 相同点：
  - 相同点就是都可以开启线程执行池中的任务。
- 不同点：
  - 接收参数：**execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。**
  - 返回值：**submit()方法可以返回持有计算结果的 Future 对象，而execute()没有**
  - 异常处理：submit()方便Exception处理

阻塞队列和非阻塞队列区别

- **阻塞队列为空的时，从队列中获取元素的操作将会被阻塞**，也就是说获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素
- **阻塞队列满时，往队列里添加元素的操作也会被阻塞**

happens-before规则

1. 程序顺序规则：一个线程中的每一个操作，happens-before于该线程中的任意后续操作。
2. 监视器规则：**对一个锁的解锁，happens-before于随后对这个锁的加锁。**
3. volatile规则：对一个volatile变量的写，happens-before于任意后续对一个volatile变量的读。
4. 传递性：若果A happens-before B，B happens-before C，那么A happens-before C。
5. 线程启动规则：**Thread对象的start()方法，happens-before于这个线程的任意后续操作。**
6. 线程终止规则：线程中的任意操作，happens-before于该线程的终止监测。我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
7. 线程中断操作：对线程interrupt()方法的调用，happens-before于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到线程是否有中断发生。
8. 对象终结规则：一个对象的初始化完成，happens-before于这个对象的finalize()方法的开始。

同步方法和同步块，哪个是更好的选择?

- 同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。
- 同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

你是如何调用 wait() 方法的？使用 if 块还是循环？为什么?

- 处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。
- wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的

```
synchronized (monitor) {
 // 判断条件谓词是否得到满足
while(!locked) {
   // 等待唤醒
monitor.wait();
 }
 // 处理其他的业务逻辑
}
```

什么是 Callable 和 Future?

- Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。
- Future 接口表示异步任务，是一个可能还没有完成的异步任务的结果。所以说 Callable用于产生结果，Future 用于获取结果。