# 1、并发和并行

并发是一个时间段里面同时发生几个事件，当然这个时间段是很短的。

并行是同一个时间点同时发生几个事件。

因为同一个时间点发生几个事件只能是多核计算机才能够处理的，所以这两个概念是很容易区分的，所谓的高并发就是在一个很短的时间段内，然后我们的cpu可以通过快速切换执行的事件来实现高并发，而对于并行只能是同一时间点我们的计算机每个cpu处理一个事件。

# 2、进程和线程

进程是程序执行和资源分配的基本单位，进程在执行过程中具有独立的内存单元，进程中的多个线程共享这些内存资源，这些个线程进行切换执行。

线程是cpu调度和分派的基本单位

# 3、创建线程的方法

创建线程的方法总共有4种：

**①、**继承Thread类，并且重写里面的run方法，把我们这个线程的业务逻辑放在这个run方法中然后启动这个线程（调用start方法）就可以了

**②、**实现Runnable接口，重写里面的run方法，.......。

**③、**实现Callable接口

注意：Callable接口是实现了Runnable接口的，Callable接口的执行是依赖了FutureTask类的，而FutureTask的依赖关系如下：

FutureTask实现RunnableFuture，而RunnableFuture实现了Future接口和Runnable接口。

其中Future接口的作用是获取Runnable接口执行过程中的返回值，或者说是操控Runnable接口线程的执行过程，常用的方法（Future接口中的方法）有：

|  |
| --- |
| isDone（）；用来判断任务是否完成。。。 |
| Concel（）；能够中断任务。。。 |
| get（）；能够获取任务执行结果。。。，也就是获取call方法的返回值。。。 |

实现Callable接口的使用：

创建一个类实现Callable接口，并且重写里面的call方法。然后把我们的业务放到call方法中就可以了。

注意：call方法的功能和Runnable中的run方法的功能一样，但是这两个方法的区别是run方法没有返回值，而call方法有返回值。

注意：前三种种方法中启动线程的方式有所不同

|  |
| --- |
| Thread类：  假设我们创建的继承Thread类的实现类的名称是MyThread，那么启动这个线程就是：  MyThread xxx = new MyThread（）；  xxx.start（）； |
| Runnable接口：  使用类Thread的构造方法（这个类有很多构造方法）  MyRunnable implements Runnable{ }  Thread xxx = new Thread（new MyRunnable（））；  xxx.start（）； |
| Callable接口：  先创建FutureTask对象，使用它的构造方法创建MyCallable（这个类实现了Callable接口）  FutureTask<String> task = new FutureTask<String>（new Callable（））；  创建Thread实例，用来执行FutureTask  Thread xxx = new Thread（task）； //FutureTask实现Runnable，所以构造方法一样  启动线程  xxx.start（）； |

**④、**前面三种方法都需要我们new一个线程，这个非常耗时，也不利于开发，所以我们在开发过程中经常使用线程池的方法来创建线程，如下：

Java中为我们提供了一个线程池的接口：Excutor，下面有实现类，同时还为我们提供了一个工具类Excutors可以直接获取四大线程池。

使用如下：

|  |
| --- |
| 先根据工具类Excutors获得线程池对象（注意可以获得四种线程池对象，我们下面只以一种为例） //下面这种类型的线程池可以定义最多的线程个数  ExcutorService yyy = Excutors.newFiredThreadPool（10）；  通过线程池对象获得线程并执行MyRunnable实例  yyy.excute（new MyRunnable（））； |

# 4、实现接口和继承类的区别

一般我们都是使用实现接口的方法来创建线程的，优点如下：

①、接口更适合多个相同的程序代码的线程去共享同一个资源。

也就是说我们可以在接口实现类中写我们的逻辑代码，在真正使用的地方定义一个Thread类，然后传递进去这个实现类的对象就可以直接启动这个线程，执行里面的逻辑代码了，也就是在一个地方书写代码，可以在多个地方使用。

②、接口可以避免java中的单继承的局限性

③、接口代码可以被多个线程贡献，代码和线程独立

④、线程池只能放入实现了Runnable和Callable接口的线程，不能放入继承Thread的类

注意：在java中，每个程序运行至少启动两个线程，一个是主线程（main线程），一个是垃圾回收线程。

但是对于两个接口来说还是有很大区别的：

Runnable接口和Callable接口比较

|  |
| --- |
| 相同点：  ①、两者都是接口  ②、两者都可以用来辩词额多线程程序  ③、两者都需要调用Thread.start（）启动 |
| 不同点：  ①、实现Callable接口的线程能返回执行接口，而实现Runnable接口的线程不能返回接口  ②、Callable接口的call方法允许抛出异常，而Runnable接口中的run方法不允许抛出异常  ③、实现Callable接口的线程可以调用Future中定义的操作线程的方法，而Runnable的线程不可以  注意：  Callable接口支持返回执行接口，但是此时需要调用FutureTask中的get（）方法，此方法会阻塞主线程的执行，直到获取”将来”接口，当不调用此方法是，主线程不会阻塞。 |

synchronized与lock的区别

|  |
| --- |
| ①、synchronized是关键字（执行层面是JVM），而Lock是一个接口  ②、synchronized会自动释放锁，而Lock必须手动释放锁  ③、synchronized是不可中断的，而Lock可以中断也可以不中断  ④、通过Lock可以知道线程有没有拿到锁，但是synchronized不可以（也就是说synchronized的执行过程被封装了，我们看不到中间过程的信息）  ⑤、synchronized能锁住方法和代码块，而Lock只能锁住代码块（Lock的使用方法就是在需要同步的代码的前面执行lock.lock()方法，在同步代码执行完成之后再执行lock.unlock()方法）  ⑥、Lock可以使用读锁提高多线程的读效率  ⑦、synchronized是非公平锁，Lock的其中一个我们经常用到的实现类ReentrantLock可以控制是否是公平锁。 |

# 5、线程安全

造成线程安全的原因有：

①、多个线程在操作共享的数据

②、操作共享数据的线程代码有多条（就是某个run方法中有连续好几行代码都是对共享数据的操作）

③、多个线程对共享数据有写操作

解决线程安全的问题：

要解决线程的以上问题就是在某个线程修改共享资源的时候，其他线程不能修改改资源等待修改完毕同步之后，才能去抢夺CPU资源，然后完成对应的操作，保证了数据的同步性，解决了线程的不安全问题。

解决线程安全问题的方法：

①、同步代码块

②、同步方法

③、同步锁

④、特殊域变量

⑤、局部变量

⑥、阻塞队列

⑦、原子变量

Synchronized具有不可中断性，也就是说当一个线程进入到Synchronized的周期范围的时候是不能够被中断这个线程的。

# 6、Synchronized的原理

首先synchronized可以用来修改同步代码块，也可以用来修饰方法：

当修饰同步代码块的时候格式如下：

Object obj = new Object（）；

Synchronized（obj）{ 业务逻辑 }

其中obj在此起到了线程锁的作用。

下面我们重点说一下这个线程锁的知识：

首先在此其中真正起到锁的东西并不是obj这个对象，而是与这个obj对象相对应的monitor对象，这个对象不是java对象，而是一个C++的对象，这个对象是在obj对象的对象头中存在着一个指向monitor的引用。而在monitor这个C++对象中存在着几个很重要的属性，其中一个是owner，这个属性保存着进入到这个同步代码块中的线程的线程名称，还有一个属性是。。。，这个属性保存着一个计数器，这个计数器的值就是我们这个线程进入到的是第几层同步代码块。还有一个object属性，这个属性保存着指向我们obj对象的引用，也就是obj这个java对象的对象头中保存着执行monitor这个C++对象的引用，同时monitor这个C++对象中也保存着一个指向obj对象的引用。

|  |
| --- |
| **monitorenter：**  当JVM执行某个线程的某个方法内部的monitorenter时，会尝试去获取当前对象对应的monitor的所有权，其过程如下：  ①、若monitor的进入数recursions为0，则线程就可以进入monitor，并将monitor的进入数设置为1，当前线程成为monitor的owner（所有者）  ②、若线程已拥有monitor，那么允许它重入monitor，同时monitor的进入数recursions就会加1  ③、若其他线程已经占有了monitor的所有权，那么当前尝试获取monitor的所有权的线程会被阻塞，直到monitor的进入数变为0，才能重新尝试获取monitor的所有权。 |
|  |
| 当我们的线程执行到同步代码块的时候JVM会先去看同步代码块中放置的对象有没有关联的monitor对象，如果没有那么JVM就会创建一个monitor对象（注意这个对象是由JVM创建的C++对象，并不是我们自己创建的）， |
|  |
| **monitorexit：**  ①、能执行monitorexit指令的线程一定是拥有当前对象的monitor的所有权的线程  ②、执行monitorexit时会将monitor的进入线程数减1，当monitor的进入数减为0时，当前线程退出monitor，不再拥有monitor的所有权，此时其他被这个monitor阻塞的线程可以尝试去获取这个monitor的所有权  ③、monitorexit释放锁：monitorexit插入在方法结束处和异常处，JVM保证每个monitorenter必须有对应的monitorexit  ④、synchronized出现异常之后仍然会释放锁，因为在javap的反汇编中可以看到，专门有一个monitorexit指令对应于发生同步代码块中发生异常的时候执行的。 |
|  |
| 小结：  通过javap反汇编我们可以知道java代码中同步代码块中对synchronized的使用变成了调用monitorenter和monitorexit两个指令，并且每个锁对象都会关联一个monitor（监视器，这才是真正的锁对象），它内部有两个重要的成员变量owner会保存获得锁的线程，recursions会保存线程获得锁的次数，当执行到monitorexit时，这个计数器recursions会减1，当这个计数器减到0时这个线程会释放锁。 |
|  |
| **关于同步方法的介绍：**  通过反汇编javap之后我们可以知道，在反汇编的代码中同步方法会在方法的修饰符上加上ACC\_SYNCHRONIZED这个修饰词，而这个修饰词会隐式的调用monitorenter和monitorexit |

|  |
| --- |
| Synchronized关键字的可见性体现：  JMM（java内存模型）关于synchronized的两条可见性规定：  ①、线程解锁前，必须把共享变量的最新值刷新到主内存中  ②、线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。 |
| Synchronized关键字的原子性体现：  它的原子性很明显，因为synchronized能够保证一段代码的原子性我们可以很直观的想明白，因为synchronized最终是通过指令monitorexit和monitorenter来操作的，而当我们进入到一个同步代码块或者是同步方法中之后，我们就获得了这个同步代码块或者是同步方法的锁，只要我们不释放锁别的线程就不能对这段代码进行操作，保证在同一时间只能有一个线程操作这段代码，所以我们可以认为这是保证了synchronized范围内的原子性的，而正真的原理应该是以为我们java内存模型中定义的8种操作都是原子性的，不可再分割的，而这8条原子操作，其中有6条可以满足基本数据类型的访问读写具备原子性，还剩下lock和unlock两条原子操作，而这两条指令和monitorenter和monitorexit指令具有相同的作用，而这两条指令反应到java代码中就是synchronized关键字了，所以我们可以说synchronized是可以保证原子性的。 |
|  |

# 7、Volatile关键字

这个关键字有两个作用：第一个是保持线程的可见性，第二个是防止指令重排序（有序性）

第三个是仅能够保证对volatile变量读写操作的原子性

Volatile一般用来共享变量之前（类的成员变量、类的静态成员变量），被volatile修饰之后，就会具有以下两层语义：

①、保证了不同线程对这个变量进行操作是的可见性，即一个线程修改了某个变量的值，这个新值对其他线程来说是立即可见的（注意：不保证原子性）

②、禁止进行指令重排序（保证变量所在行的有序性）

Volatile关键字就是在执行被这个关键字修饰的属性的前后去加入了一些屏障指令，而我们常说的volatile关键字能够强制我们去刷新主内存或者是重新从主内存中读取数据都是因为在执行加载volatile修饰的属性前后去执行了我们所说的屏障指令造成的。

|  |
| --- |
| 针对可见性：  一般情况下：JVM为了性能的优化都是对工作内存中的变量数据就行修改，然后把修改后的新值刷新到主内存中，但是接下来再次执行代码逻辑的时候并不是重新的从主内存中读取变量的新值，而是直接使用已经加载到工作内存中的值进行修改，而volatile的作用就是强制每次对变量根据逻辑代码进行修改的都会先到主内存中读取这个变量的最新值到工作内存中，之后在修改，每次都是这样。所以保证了有volatile修饰的变量的最新值能够对其他线程可见。  了解一下JVM运行时的内存分配过程：  内存的运行时数据区域通常我们分为5个部分：堆、方法区、虚拟机栈、本地方法栈、程序计数器。其中前两个是多线程公用的部分，后三个是每个线程私有的部分。  每一个线程都有一个虚拟机栈，而这个线程所执行的方法都会以栈帧的形式从上到下依次排列在这个线程的栈中，当我们执行到这个线程的栈中的某个方法对象的栈帧的时候，根据这个栈帧内部的局部变量表中保存的引用变量可以找到这个引用变量在堆中对应的真实对象，然后我们就会把这个真实对象的数据加载到在cpu中的工作内存中（我们称这个数据为副本），然后对工作内存中的数据进行各种代码逻辑操作（在这个过程中和堆内存（主内存）中的数据没有任何关系），当我们在工作内存中执行完逻辑代码之后才会把已经根据代码逻辑修改后的工作内存中的值（副本）刷新到主内存（堆中的值）。 |
|  |
| 针对有序性：  有序性我们可以通过下面的一个例子做一个更深刻的理解：  **public** **class** Singleton{  **private** **volatile** **static** Singleton *singleton*;  **private** Singleton() {  }  **public** **static** Singleton getInstance() {  **if**(*singleton* == **null**) { //1  **synchronized**(Singleton.**class**) {  **if**(*singleton* == **null**) {  *singleton* = **new** Singleton(); //2  }  }  }  **return** *singleton*;  }  }  实际上当程序执行到2处的时候，如果我们没有使用volatile关键字修饰变量singleton，那么就可能会造成错误，这是因为使用new关键字初始化一个对象的过程并不是一个原子的操作，它分为下面三个步骤：  ①、给singleton分配内存  ②、调用Singleton的构造方法来初始化成员变量  ③、将singleton对象指向分配的内存空间（执行完这步singleton就为非null了，因为我们这里的singleton是一个引用变量，当执行完这一步之后就是将我们的这个引用变量指向了堆内存中的步骤①中分配的内存空间中的对象了，所以这一步执行完成之后singleton就不为空了）  所以如果虚拟机存在指令重排序优化则步骤②和步骤③的顺序是无法确定的，如果A线程率先进入同步代码块并先执行了步骤③而没有执行②，此时因为singleton已经非null，这时候线程B到了1处，判断singleton非null并将其返回使用，因为此时Singleton实际上还未初始化，自然就会出错，synchronized可以解决内存可见性，但是不能解决重排序问题。 |

# 锁的种类

* Synchronized，它就是一个：非公平，悲观，独享，互斥，可重入的重量级锁
* 对于锁的分类以及概念描述参考下面这个连接文章

|  |
| --- |
| <https://blog.csdn.net/w450093854/article/details/85062357> |

|  |
| --- |
| ①、偏向锁  经过研究发现在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一个线程多次获得，为了让线程获得锁的代价，引进了偏向锁。  偏向锁的意思是这个锁会偏向于第一个获得它的线程，会在对象头中存储锁翩跹的线程的id，以后该线程进入和退出同步代码块时只需要检查是否为偏向锁、锁标记位（如果是偏向锁则为1、不为偏向锁则为0）以及线程id即可。  偏向锁的撤销：当偏向锁出现竞争的时候，也就是出现了多于一条线程执行这段代码的时候，偏向锁就会立刻终止，转化为轻量级锁 |
|  |
| ②、轻量级锁  轻量级锁的出现并不是要代替重量级锁的，但是当轻量级锁中多个线程在同一时刻进入到临界区，会导致轻量级锁膨胀升级为重量级锁  使用场景：多个线程交替执行某段同步代码的时候，不存在竞争但是也存在多条线程执行，这样的时候就适合轻量级锁 |
|  |
| ③、自旋锁  在JVM虚拟机在由轻量级锁升级为重量级锁的时候还会挣扎一下，而这个挣扎的过程就是自旋锁。  因为线程的阻塞和唤醒需要CPU从用户态转化为和核心态（用户态和核心态可以类比磁盘和内存的关系），频繁的阻塞和唤醒对CPU来说是一件负担很重的工作。同时研究表明共享数据的锁定状态只会持续很短的一段时间，为了这很短的一段时间而阻塞和唤醒线程非常的不值当。所以才产生了自旋锁。  要想使用自旋锁必须要保证我们的电脑是一个多核的计算机，这样的话抢到锁的线程执行同步代码，没有抢到锁的线程可以同时在外面进行自旋准备再次抢锁。 |
|  |
| ④、重量级锁  我们平时一般说的线程的锁都是指的重量级锁，这个锁的特点有： |
| **偏向锁/轻量级锁/重量级锁** ****这三种锁是指锁的状态****，并且是针对Synchronized。在Java 5通过引入锁升级的机制来实现高效Synchronized。****这三种锁的状态是通过对象监视器在对象头中的字段来表明的****。  ****偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价****。  ****轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁****，不会阻塞，提高性能。  ****重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。****重量级锁会让其他申请的线程进入阻塞，性能降低。 |

在jdk1.5之前，synchronized中只包含一种锁，也就是我们分析的monitor，这是一种重量级的锁，在jdk1.6之后我们的锁就不仅仅包含重量级的monitor锁了，还包含偏向锁、轻量级锁，自旋锁、重量级锁，同时还包括锁消除和锁粗化

在jdk1.6之后，如果需要加锁，那么加锁的顺序是：

偏向锁--->轻量级锁--->（自旋锁）--->重量级锁

偏向锁：主要使用于只有一条线程的或者是多条线程执行，但是基本上都是其中一条线成执行，其他的线程基本不执行的情况。。。偏向锁主要是对第一个

# 9、线程通信方式

首先线程通信是先把cpu在一定的时间段内分片，然后每条线程执行这段时间内的其中某片时间，当这片时间结束后不管线程有没有执行完成都要停下来让占据下一个时间分片的线程执行，以此类推。

所谓的线程间的通信就是两个线程之间怎么建立联系。

线程间通信的方式如下：

|  |
| --- |
| ①、休眠唤醒方式  用到了Object类中定义的wait、notify、notifyAll方法  用到了Condition类中的await、singnal、singnalAll方法  **注意1：**在Object类中的wait和notify这两个方法的使用时依赖synchronized的，所以想要调用这两个方法必须是在synchronized修饰的同步方法中或者是synchronized修饰的同步代码块中的。  **注意2：**创建Condition对象：Condition condition = lock.newCondition（）；  在Condition中的await和singnal这两个方法的使用是依赖lock对象的，意思就是在使用await和singnal方法之前需要有lock.lock()，在这两个方法执行之后需要有lock.unlock()，  **注意3：**Object中的wait（）方法必须在synchronized（同步锁）下使用。  Object中的wait（）方法必须要通过nodify（）方法进行唤醒。  Condition中的await（）方法必须和lock（互斥锁/共享锁）配合使用。  Condition中的await（）必须通过singnal（）方法进行唤醒。 |
|  |
| ②、CountDownLatch类：这个类的作用是用于某个线程A等待若干个其他线程执行完之后他才执行  CountDownLatch的原理：设置一个计数器，这个计数器的初始值为线程的数量，每执行一个线程，这个计数器就减1，一直到这个计数器减为0之后等待的这个线程A就可以执行了。 |
|  |
| ③、CyclicBarrier类：这个类的作用是一组线程等待至某个状态之后再全部同时执行 |
|  |
| ④、Semaphore类：用于控制对某组资源的访问权限。 |

**sleep和wait方法的区别：**

|  |  |  |
| --- | --- | --- |
|  | wait方法 | sleep方法 |
| 同步 | 只能在同步上下文中调用wait方法，否则会抛出illegaMontorStateException异常 | sleep的调用没有同步上下文的限制 |
| 作用对象 | wait方法定义在Object类中，作用于对象本身 | sleep方法定义在java.lang.Thread中，作用于当前线程 |
| 释放锁资源 | 是 | 否 |
| 唤醒条件 | 其他线程调用对象的notify或者notifyAll方法唤醒 | 超时或者调用interrupt方法唤醒 |
| 方法属性 | wait是实体方法 | sleep是Thread类中的静态方法 |

sleep()方法是Thread类的静态方法 等到计时时间一到，此线程会自动苏醒

而wait()方法是object类的方法

**wait和notify方法的区别：**

①、wait和notify都是Object中的方法

②、wait和notify执行当前线程都必须获得对象锁

③、wait的作用是使当前线程进行等待

④、notify的作用是通知其他等待当前线程的对象锁的线程

# 10、java的内存模型

java程序执行流程：

首先是我们自己编写的代码文件（.java）---> 通过java编译器把java文件编译成为class文件（.class）文件-----> 通过类加载器把class文件加载到内存中（可以理解为运行时数据区）----->执行引擎就会根据运行时数据区的内容去执行代码。

而java的内存模型就是对运行时数据区的分析：

首先根据JVM规范，内存共分为虚拟机栈、堆、方法区（里面包含常量池）、程序计数器、本地方法栈五个部分

方法区其实也是堆的一种只是它是永久性的，也就是垃圾回收一般不会光顾这里（也会光顾但是光顾的少而已），方法区中存放了要加载的类的信息（名称、修饰符等）、类中的静态常量、类中定义为final类型的常量、类中的Field信息、类中的方法信息。

虚拟机栈中的栈帧中的结构包括：局部变量表、操作数栈、动态链接方法、返回地址，其中局部变量表中主要存放了这个方法中包含的局部变量，而这个局部变量指的是基本类型的局部变量，而引用类型的局部变量的引用是在局部变量表中的，但是具体的对象是在堆中存放的。

常量池中存储两类数据：字面量和引用量，

字面量：字符串、final变量

引用量：类/接口/方法/属性（字段）的名称和描述符

常量池在编译期就已经确定了，并保存在已经编译的class文件中。

|  |
| --- |
| 针对于某一个方法我们的java内存模型的工作模式如下 |
| **public** **class** MyDemo{  **public** **static** **void** main(String[] args) {  Object obj = **new** Object();  System.***out***.println(obj);  }  } |
| 针对上面的这段代码，首先会先通过编译器将这段java代码编译成为class文件，然后通过类加载器把编译之后的class文件加载到内存中。  首先方法区会存储这段代码中的类的名字、方法的名字、类的修饰符、方法的修饰符、关键字、返回值类型。  并且会将我们new Object（）对象存储到堆内存中，然后将obj这个引用存储到方法区中的main方法对应的栈帧中的局部变量表中。  然后再为main方法创建一个程序计数器记录方法执行的位置  当执行引擎执行的时候就会去方法区找第一个需要执行的方法，找到之后就会为这个main方法创建虚拟机栈，然后到方法区中找到main方法对应的栈帧，然后请求CPU资源请求执行，这时候CPU就会从当前要执行的这个栈中的处于栈顶的栈帧中的方法的信息加载到CPU中的寄存器和高速缓存中（工作内存），我们所说的运行时区（五个部分）是主内存。 |
| 如果是单线程的话那么上面的执行是没有问题的，但是如果是多线程的话就会出现一个线程已经把堆中保存的信息读取到CPU工作内存中进行修改，并且已经修改完成但是没有重新写入到堆内存中的时候，另一个线程就从堆内存中读取到了同样的信息，这样就会出现脏读等线程安全问题。 |

# 11、多线程的三个特性

原子性：即一个操作或者多个操作要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行

可见性：可见性是指多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值，显然对于单线程来说，可见性问题是不存在的。

有序性：即程序执行的顺序按照代码的先后顺序执行

# 12、关于线程池

**①、为什么使用线程池**

因为在处理任务的线程的创建和销毁都非常耗时并消耗资源，并且多线程之间的切换也会非常耗时同时也非常消耗资源，为了解决这个问题，我们就引入了线程池

**②、常用的线程池有哪些**

因为ThreadPoolExecutor构造函数的参数很多，使用起来很麻烦，为了方法便的创建线程池，javaSE中又定义了Executors类，Executors类提供了四个创建线程池的方法，分别如下：

* newCacheThreadPool

该方法创建的是一个可缓存线程，如果线程池长度超过处理需要（也就是我们线程池中的线程数大于我们真正使用的线程数，也就是有一部分是空闲着的），可灵活回收空闲线程，若无可回收，则新建线程

* newFixedThreadPool

该方法创建一个指定工作线程数量的线程池，每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池出事的最大数，则将提交的任务存入到池队列中

* newSingleThreadExecutor

该方法创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序执行，如果这个线程异常结束会有另一个取代它，保证顺序执行

单工作线程最大的特点是可保证顺序第执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

* newScheduledThreadPool

该方法创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行

**③、注意**

线程池只支持Runnable和Callable接口创建的线程，不支持一个线程类直接创建的对象

上面四个都是Executors类的静态方法，当使用Executors调用对应的方法的时候就能够获得与之对应的线程池对象，然后再使用创建的这个线程池对象调用executor方法，并且传入接口类型的线程对象就可以在run方法中编写我们的线程执行的逻辑代码了。

线程中真正为我们提供的创建线程的类是ThreadPoolExecutor这个类，这个类为我们提供了很多种构造方法供我们去创建线程池，但是因为这个类的构造方法中都提供了很多个参数，所以我们自己创建线程池会非常麻烦，所以线程类又为我们提供了Executors类，这个类里面有四个我们上面描述的静态方法，由Executors类调用对应的静态方法就能创建出对应的线程池供我们使用了。

**④、线程池的原理**

下面我们总结一下线程池的工作流程（也算是原理）

　1）首先，要清楚corePoolSize和maximumPoolSize的含义；

　2）其次，要知道Worker是用来起到什么作用的；

　3）要知道任务提交给线程池之后的处理策略，这里总结一下主要有4点：

* 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
* 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；
* 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；
* 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

**⑤、线程池的初始化过程**

* 默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。
* 在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

prestartCoreThread()：初始化一个核心线程；

prestartAllCoreThreads()：初始化所有核心线程

**⑥、线程池中的依赖关系**

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。

AbstractExecutorService是一个抽象类，它实现了ExecutorService接口。

而ExecutorService又是继承了Executor接口，里面就只有一个方法Execute(Runable r)，这个方法就是提交任务让线程池执行的方法。

Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

　　然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

　　抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

　　然后ThreadPoolExecutor继承了类AbstractExecutorService。

　　在ThreadPoolExecutor类中有几个非常重要的方法：

execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

　　submit()方法是在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果

**⑦、几个构造方法中的参数分析**

ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器

下面是这些构造器中的参数，我们逐个进行分析：

* corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
* maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
* keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
* unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

|  |
| --- |
| TimeUnit.DAYS; //天 TimeUnit.HOURS; //小时 TimeUnit.MINUTES; //分钟 TimeUnit.SECONDS; //秒 TimeUnit.MILLISECONDS; //毫秒 TimeUnit.MICROSECONDS; //微妙 TimeUnit.NANOSECONDS; //纳秒 |

* workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

|  |
| --- |
| ArrayBlockingQueue; LinkedBlockingQueue; SynchronousQueue;  ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。 |

* threadFactory：线程工厂，主要用来创建线程；
* handler：表示当拒绝处理任务时的策略，有以下四种取值：

|  |
| --- |
| ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。 ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。 ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程） ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务 |

⑧、线程池的状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

|  |
| --- |
| volatile int runState;  static final int *RUNNING* = 0; static final int *SHUTDOWN* = 1; static final int *STOP* = 2; static final int *TERMINATED* = 3; |

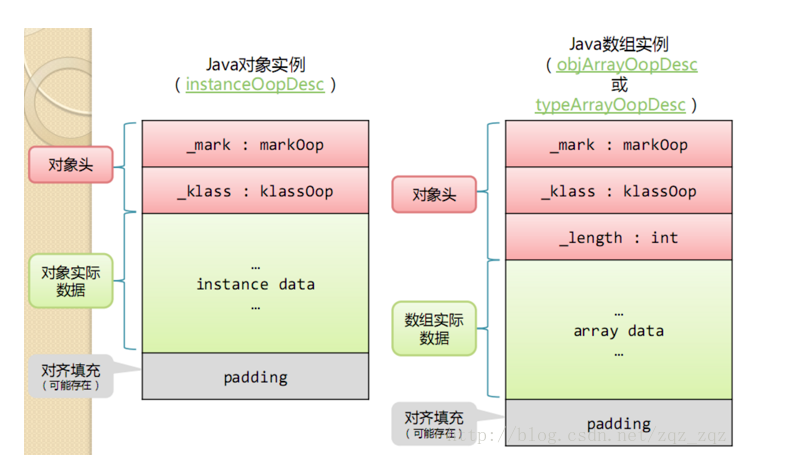
runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性；

下面的几个static final变量表示runState可能的几个取值。

* 当创建线程池后，初始时，线程池处于RUNNING状态；
* 如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；
* 如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；
* 当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

# 对象的布局

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。下图是普通对象实例与数组对象实例的数据结构：



1. 对象头

HotSpot虚拟机的对象头包括两部分信息：

1. markword   
   第一部分markword,用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32bit和64bit，官方称它为“MarkWord”。

2、klass   
对象头的另外一部分是klass类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例.

3、数组长度（只有数组对象有）   
如果对象是一个数组, 那在对象头中还必须有一块数据用于记录数组长度.

②、实例数据

实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的，还是在子类中定义的，都需要记录起来。

③对其填充

在32位系统下，存放Class指针的空间大小是4字节，MarkWord是4字节，对象头为8字节。

在64位系统下，存放Class指针的空间大小是8字节，MarkWord是8字节，对象头为16字节。

64位开启指针压缩的情况下，存放Class指针的空间大小是4字节，MarkWord是8字节，对象头为12字节。

数组长度4字节+数组对象头8字节(对象引用4字节（未开启指针压缩的64位为8字节）+数组markword为4字节（64位未开启指针压缩的为8字节）)+对齐4=16字节。

静态属性不算在对象大小内。

# 14、synchronized和Volatile

## （1）、synchronized的三种特性实现

**可见性：**

JMM对于synchronized有两条规定：

线程解锁前，必须把共享变量的最新之刷新到主内存中

线程加锁时，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值（加锁与解锁需要是同一把锁）

这样就会导致synchronized代码块是按照下面的顺序执行的：

①、获得互斥锁，清空工作内存并从主内存拷贝变量的最新副本到工作内存

②、执行代码

③、将更改后的共享变量的值刷新到主内存

④、释放互斥锁

正是上面的执行顺序使得synchronized具备内存可见性。显示锁（Lock）和synchronized有相同的内存可见性语义,其实原理跟synchronized类似。

**原子性：**

synchronized关键字是排它锁可以保证原子性的实现

它能够锁住一块方法、代码块等使得某一个时间点只能由一个线程执行这个方法或者是代码块，其他线程不能执行，必须由这个线程执行完成或者是发生异常退出的时候其他线程才能够执行这个方法或者代码块，从而保证了原子性。

**有序性：**synchronized不能保证有序性

## （2）、Volatile的三种特性实现

**可见性**

为了实现 volatile 的可见性内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

对于volatile是如何实现内存可见性，深入来说：是通过加入内存屏障和禁止重排序优化来实现的。（重排序指单线程中在保证执行结果不变的前提下java虚拟机为了提升处理速度可能会将指令重排，达到最合理化）

对volatile变量执行写操作时，会在写操作后加入一条store屏障指令，执行效果：

改变线程工作内存中的volatile变量副本的值

将改变后的副本的值从工作内存刷新到主内存

对volatile变量执行读操作时，会在读操作前加入一条load屏障指令，执行效果：

从主内存中读取volatile变量的最新值到线程的工作内存中

从工作内存中读取volatile变量的副本

简单来说：**volatile变量在每次被线程访问时，都强迫从主内存中重读变量的值，而当该变量发生变化时，又会强迫线程将最新的值刷新到主内存。这样在任何时刻，不同的线程总能看到该变量的最新值。从而保证了变量的内存可见性。**

**有序性**

volatile关键字将会直接禁止JVM和处理器对关键字修饰的指令重排序，但是对于volatile关键字修饰的前后的、无依赖的指令，可以进行重排序

被volatile修饰的变量，可以认为插入了一个内存屏障，他会进行如下保障：

确保指令重排序时不会将其后面的代码排到内存屏障之前

确保指令重排序时不会将其前面的代码排到内存屏障之后

确保在执行到内存屏障修饰的指令时前面的代码全部执行完成

强制将线程工作内存中值的修改刷新至主内存中

如果是写操作，则会导致其他线程工作内存(CPU Cache)中的缓存数据失效

|  |
| --- |
| int x = 0;  int y = 1;  volatile int z=20；  x++;  y--； |
| 在上述语句volatile int z=20之前，先执行x的定义还是先执行y的定义，我们并不关心，只要能够百分之百地保证在执行到z=20的时候x=0, y=1，同理关于x的自增以及y的自减操作都必须在z=20以后才能发生。这个结果就是上面的逻辑处理后的结果。 |
|  |

**原子性：**volatile没有实现原子性

|  |
| --- |
| 比如x=1；  他可以简单的理解为三个步骤：  读取count的值；  操作count的值；  回写count的值；  volatile可以保障在第一步的时候，读取到了正确的值，但是由于不是原子的，在接下来的操作过程中，count的值，可能已经被更新过了，也就是读取到了旧值  继续使用这个旧值很显然就把别人的更新抹掉了，你读取的1，可能此时应该是2了，但是你操作后还是2，无故的擦除了别人的增加 |



## （3）、synchronized和volatile的比较

①、volatile不需要加锁，比synchronized更加轻量级，不会阻塞线程

②、从内存可见性讲，volatile读相当于加锁，volatile写相当于解锁

③、synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性



# 15、线程的状态

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

**1、新建状态(New)**：新创建了一个线程对象。

**2、就绪状态(Runnable)**：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权，

   即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。

**3、运行状态(Running)**：就绪状态的线程获取了CPU，执行程序代码。

**4、阻塞状态(Blocked)**：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

   阻塞的情况分三种：

①.等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，

   必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，

②.同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。

③.其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时，

   或者I/O处理完毕时，线程重新转入就绪状态。

**5、死亡状态(Dead)**：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

# 16、线程在工作中的使用场景

# 锁级别的转换以及详细说明

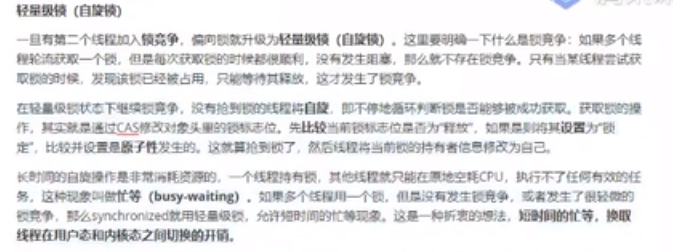
①偏向锁：当我们线程A进入到一个synchronize块的时候，我们的线程A首先回去检查这个同步代码快中的对象锁中的对象头中的标记位，这个标记位标记的就是现在是哪个线程进来这个同步代码块了，如果这个标记位上面没有值，说明现在还没有线程执行这个同步代码块，这时候线程A会把这个标记位修改为我们这个线程的信息（线程id，锁的状态等信息，注意猜测这个标记位应该是不是一个位置上的值，而是好几个位置上的好几个值），修改标记位之后就会进入到同步代码块去执行逻辑代码，当我们的代码执行完之后并且线程A已经走出了同步代码块，但是我们在标记位上的修改内容不会去清除，也就是代码执行完之后就不管了，线程A就直接走了。这就是偏向锁，比轻量级锁还轻量级的锁。

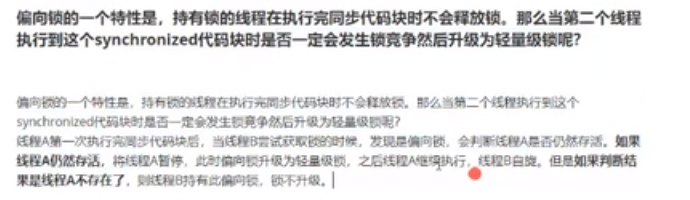
这时候我们的线程B进来了，进来之后也是同样的回去检查标记位上面有值没有，如果没有说明和上面分析的线程A的情况是一样的，都是这个同步代码块被第一次访问，如果标记位有值了，这时候我们的线程B就会去继续检查这个锁的标记位上面的线程id是否还存活，如果存活说明这个线程此时还在执行同步代码块，还没有出来，**1**此时只能等待（此时说明有竞争力了）**1**，如果标记位上面的线程id此时已经不存活了，那么我们的线程B就会去修改这个锁对象的标记位，把这个标记位修改成线程B的信息，并且进而进入同步代码块中取执行代码逻辑。并且线程B执行完之后也直接撤了，并不去清除标记位上面的值。

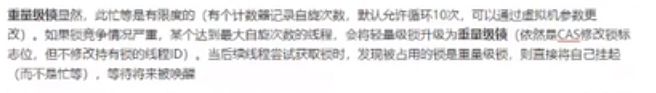
注意：偏向锁没有一点竞争。

1. 在上面的①中我们标记的1的地方，当我们去检查线程A的id是否还存活的时候得到的答案是线程A还存活，这个时候我们的锁就会升级成为轻量级锁，所谓的轻量级的锁就是锁之间存在竞争了，在我们的对象头中是否是偏向锁的地方的值也会改变，并且轻量级锁之间就会存在竞争了。当我们的竞争越来越激烈的时候，按道理来说我们的轻量级锁就会升级成为重量级的锁，也就是我们的线程就会出现阻塞，没有抢到锁的线程就会被加入到缓存队列中等等情况，但是当我们的从阻塞状态转化为运行状态这个过程是需要消耗CPU的，并且一般情况下一个线程执行一段同步代码块的时间也是极其短的，所以我们刚开始的时候会让
2. 此时就进化成了自旋锁，所谓的自旋锁我们可以认为是while（true）循环，但是它并不是一味的就去自旋，当达到我们所要求的的临界值的时候（比如自旋的次数，时间等），当达临界值的时候我们的锁会升级为重量级的锁，因为此时线程的竞争太激烈了，依靠自旋已经解决不了这个问题了，所以就会让我们等待的线程的状态变成为阻塞状态。准确的来说重量级的锁是让我们的线程去暂停抢占锁，然后去等待唤醒，然后我们此时会有一个问题，那么是谁唤醒了进入暂停状态的线程B呢，此时就用到了我们的观察者模式，我们会去观察锁的标记位中的标记的线程id对应的线程是否还存活，如果线程不存活了就会进入到去唤醒我们的线程B或者更多的处于暂停状态中的线程去抢占这个同步代码块中的锁。

**下面这两张截图仔细看看：**





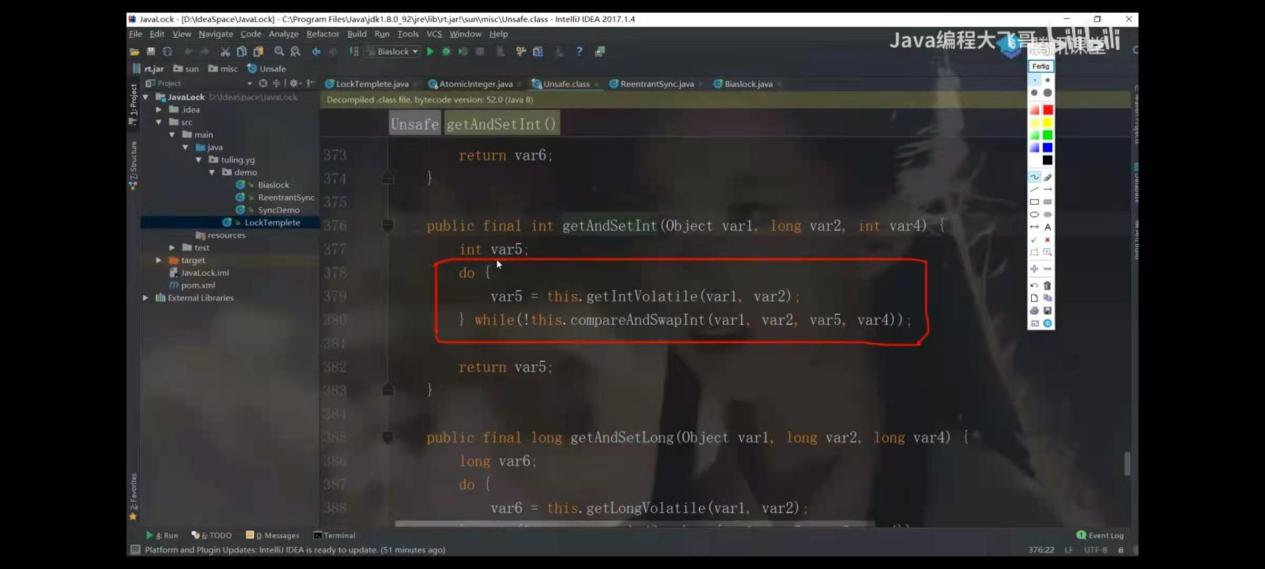


# 分布式锁

①、为什么要有分布式锁呢

因为当我们的系统是分布式的系统的时候我们以前使用的synchronize或者是lock等这种锁已经不能使用了，因为这些锁都是在一台主机上对于同一个JVM来说数据还是共享的，但是在分布式中我们的系统是部署在多台主机上面的，此时对我们多台JVM上的数据是不共享的，也就是说你用synchronize或者是lock这些锁锁住对象之后，其他的服务器是不知道的，所以才有了我们的分布式锁。而我们的分布式锁说直白一点就是找一台服务器，搭建一个分布式锁，也就是谁使用了这个资源，并且对这个资源上了锁就去这台服务器上等级一下，使用完了之后释放了资源也要去这台服务器上说一声。

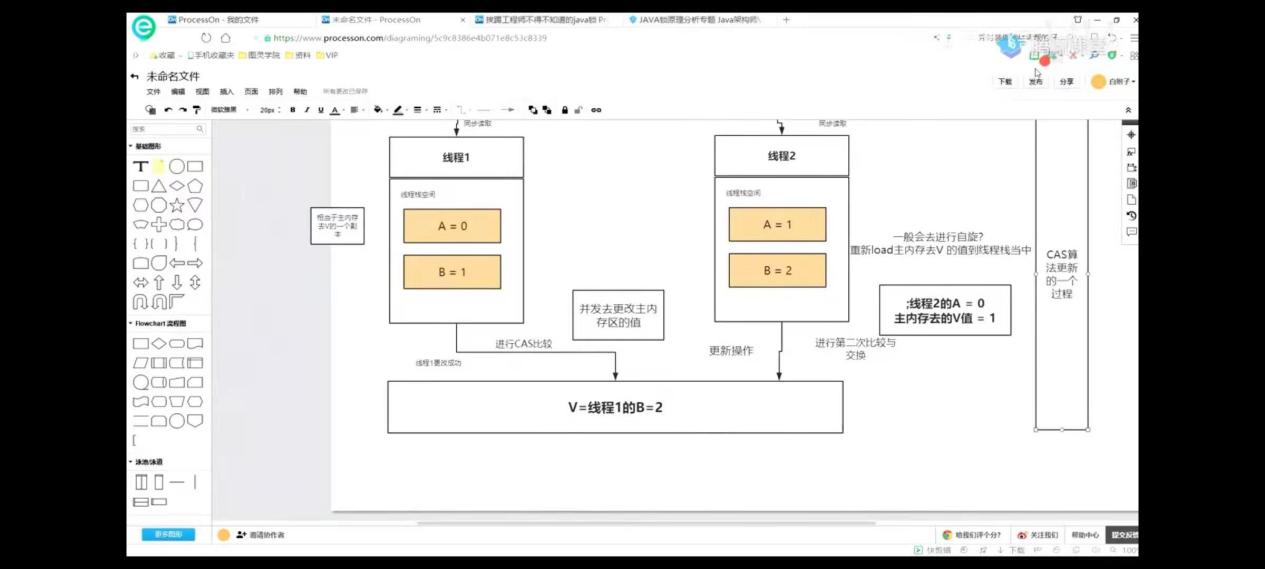
# 关于锁的自旋、CAS



标记的地方就是自旋，其中var5就相当于A值，var4就相当于B值

A值就是我们从主内存读取到工作内存中的副本，B值就是我们要修改成的值，V值就是存在于主内存中用于和A值进行对比的值。

所谓的自旋就是不停的执行do大括号里面的方法去重新获取（load）主内存中的值(V值)到我们的栈内存中（也就是我们的工作内存中，也就是把工作内存中的A值修改成现在主内存中的新值，然后再次去第二次的比较与交换）直到循环到我们规定的次数

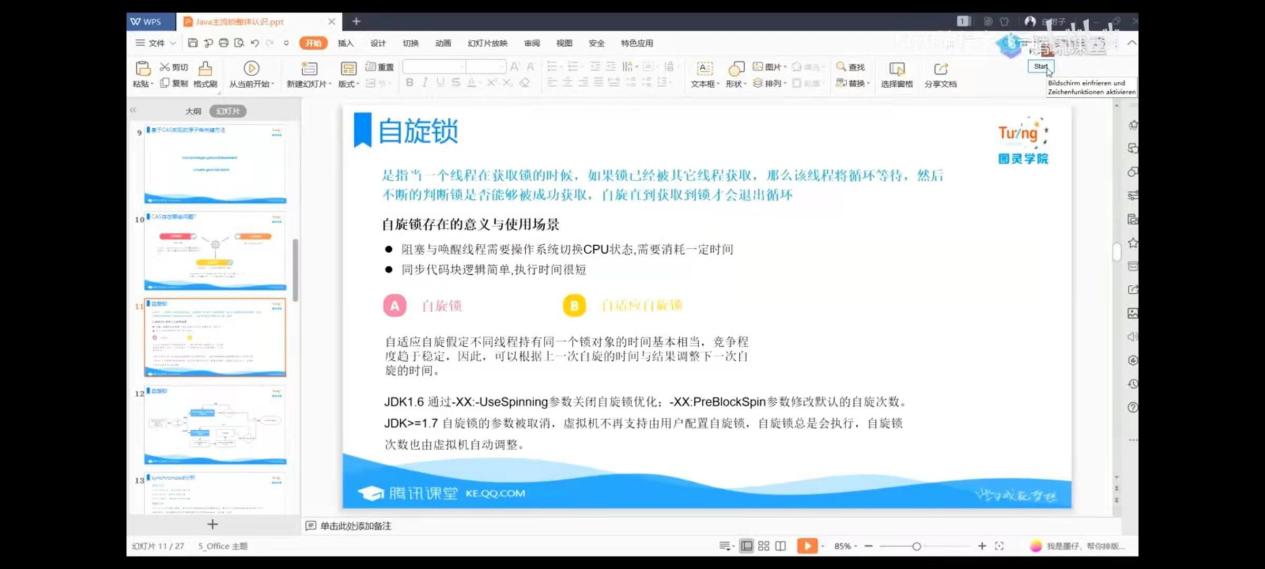


上面这张图片也是CAS的流程图

自旋的一个缺点：

①、假如我们的自旋线程个数过于多就会占用我们cpu的内核（现在的cpu都是多核的，也就是说自旋必须是多核的，真正发生并发的计算机中发生的），这样就会发生什么事都没干却消耗我们大量的资源，因为自旋其实什么事都没干，只是在等待

②、cas中存在一个问题就是ABA问题，当第一个线程刷新主内存的值为一个新值，第二个线程又把主内存中值又改成了原来的值，这时候第三个线程过来一看主内存中的V值和它的A值是相同的，然后他就直接修改V的值为它的B值，也就是说第三个线程没有感知到主内存中的V值的变化，为了解决这个问题我们可以在主内存中的V值上面加上一个版本号，每被修改一次版本号就加一，过来的线程除了对比主内存中的V值和工作内存中的A值是否相同，还要对比版本号是否相同。



③、CAS原子性操作原理

举例说明：

比如AtomicInteger原子类中的incrementAndGet()方法调用CAS操作自增1：

|  |
| --- |
| public final int incrementAndGet() {  for (;;) {  int current = get();  int next = current + 1;  if (compareAndSet(current, next))  return next;  } }  public final boolean compareAndSet(int expect, int update) {  return unsafe.compareAndSwapInt(this, valueOffset, expect, update); } |

上述栗子可见，最终java调用的是Unsafe类的CAS方法，该方法是native方法，它的实现是C++写的（unsafe.cpp）

CAS操作调用的是unsafe类的CAS方法，该方法是native方法，底层实现通过cmpxchgl这个指令在CPU级完成CAS操作的

在多核下还需添加一条lock指令；即lock cmpxchg 来保障原子性；lock指令会锁定CPU的缓存行或者锁总线

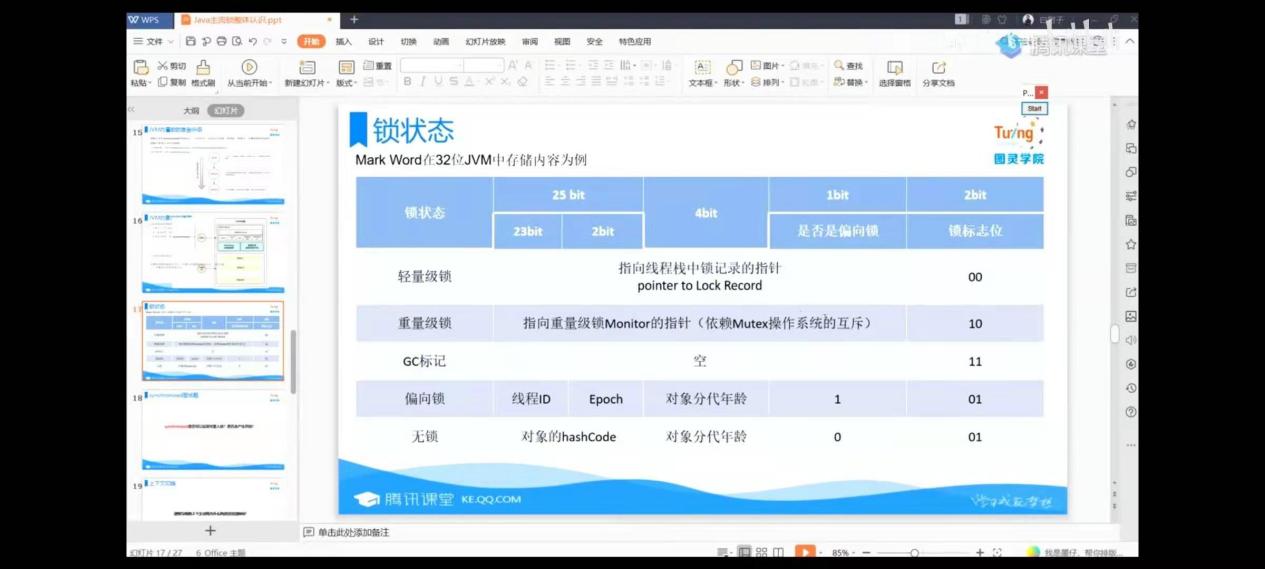
# 锁中的AQS

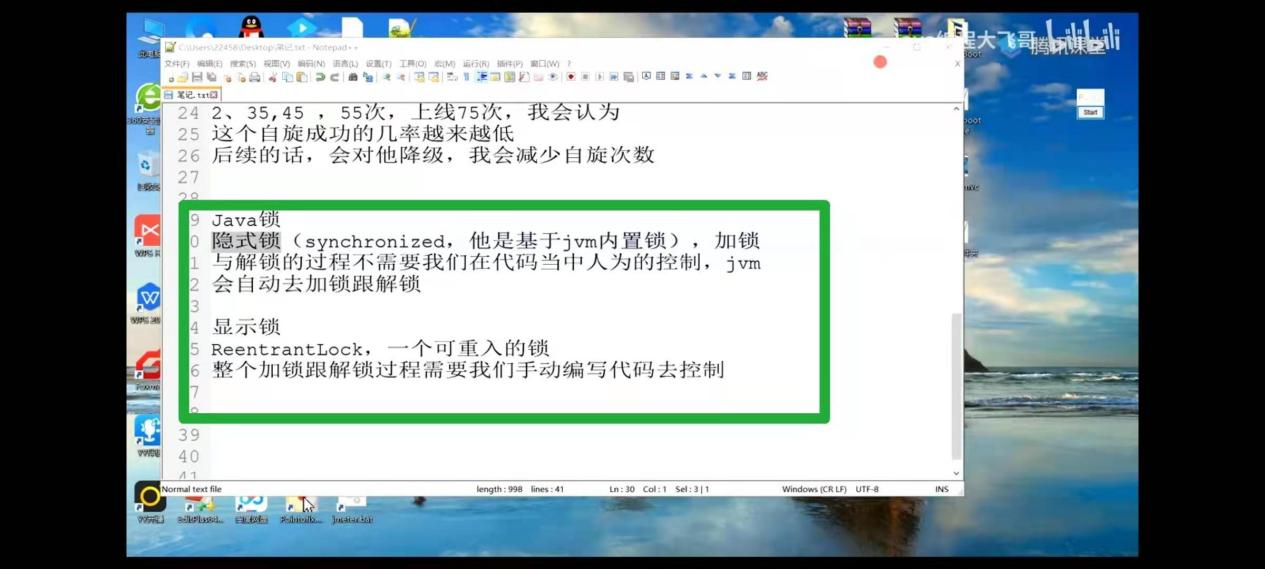
①、为什么会出现AQS的说明

我们的jvm是运行在我们的操作系统上面的，所以对于操作系统它是属于一个用户的，并且我们的jvm是没有那么大的权利去分配内存，分配cpu执行时间的，所以相对于我们的操作系统来说，我们的jvm属于用户态空间，我们的操作系统数组内核态空间，我们的线程必须通过我们的jvm用户态传递命令到内核态（也就是线程由jvm把命令传递给操作系统，然后再由操作系统去分配内存或者是分配cpu的执行时间），所以在这个过程中就会有用户态转变为内核态，这样的话非常耗费时间。

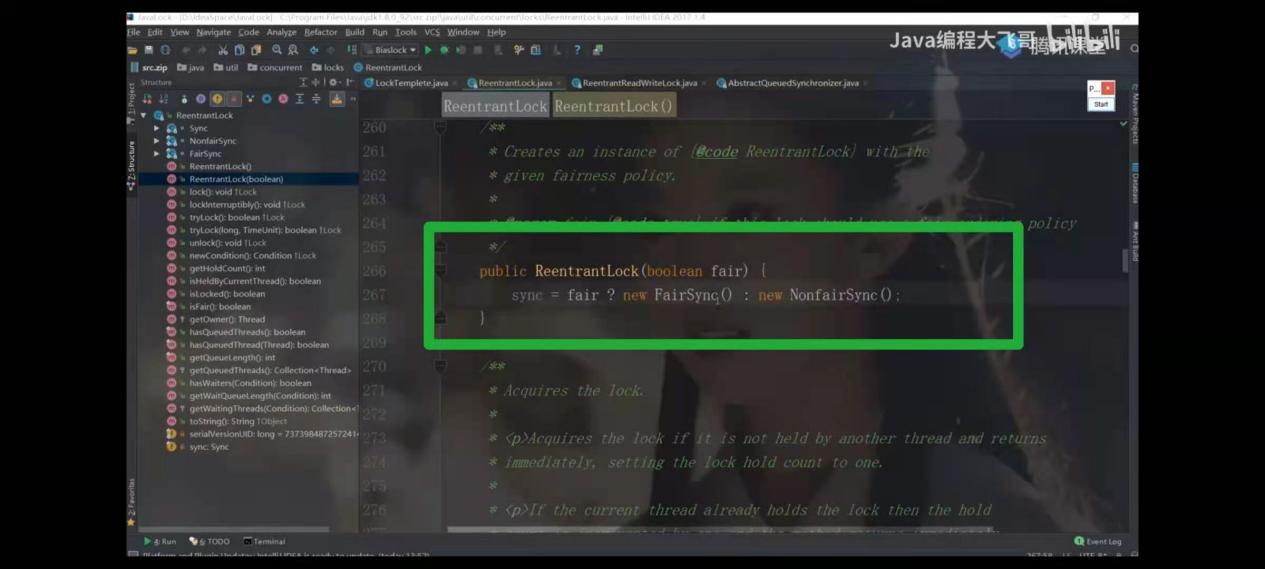
为什么我们的用户态转化为内核态会消耗资源和时间，因为我们的cpu在线程1的执行时间段中假如执行到了线程1中的某段代码a处，线程1的执行时间结束了，这时候cpu需要去切换到线程2去执行线程2的代码，这时候cpu首先需要把执行线程1中得到的变量存储到电脑内存条中的一处叫做PCB的地方以及执行到的位置存储到虚拟机栈中的程序计数器中，然后再切换到线程2中执行线程2的内容，假如一下子把线程2处的代码执行完了，然后再唤醒线程1去接着执行踏的代码，这时候cpu会先去内存条的PCB把原本存储的变量数据读取过来并且到虚拟机栈中的程序计数器中代码执行的位置读取过来，之后才开始接着执行线程1的代码。

可以理解为用户态和内核态中切换的过程中发生了cpu和内存条中间的一次io，并且须知内存的速度小于寄存器的速度，但是大于硬盘的速度





②、AQS详解



synchronized关键字和lock接口都是悲观锁，ReentrantLock实现了Lock接口，并且他里面有一个Sync的属性，而Sync有继承了AbstractQueuedSynchronizer类，我们常说的AQS就是这里的AbstractQueuedSynchronizer抽象队列同步器

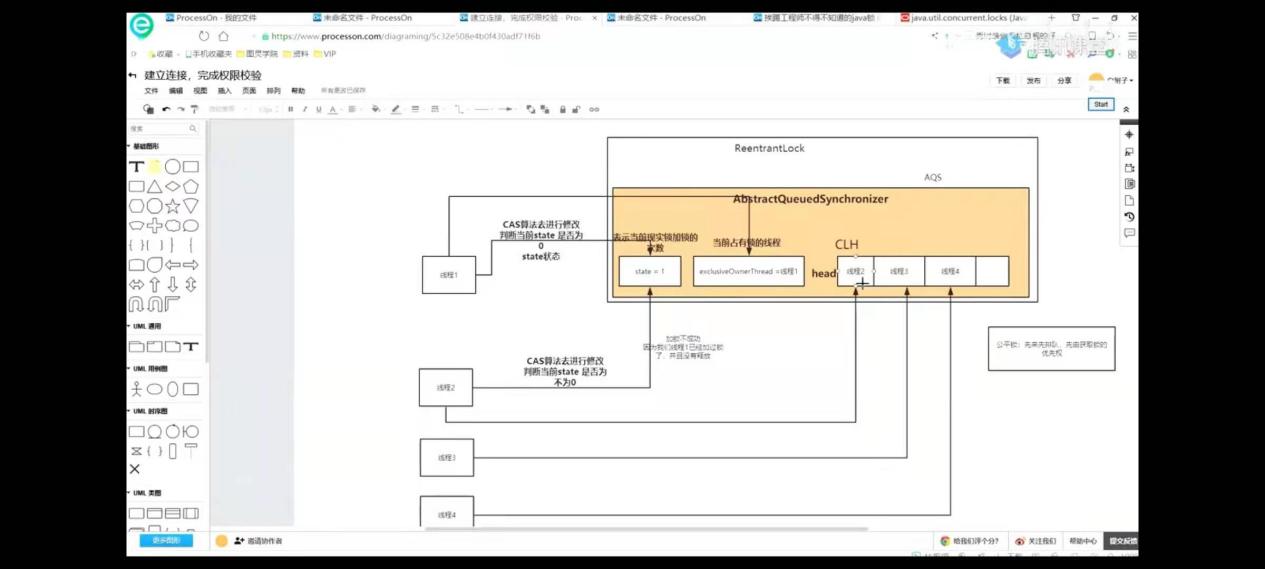
同时由上图中的构造方法我们知道ReentractLock类的构造方法通过传入的参数不同可以构造出一个公平锁也可以构造出一个非公平锁。

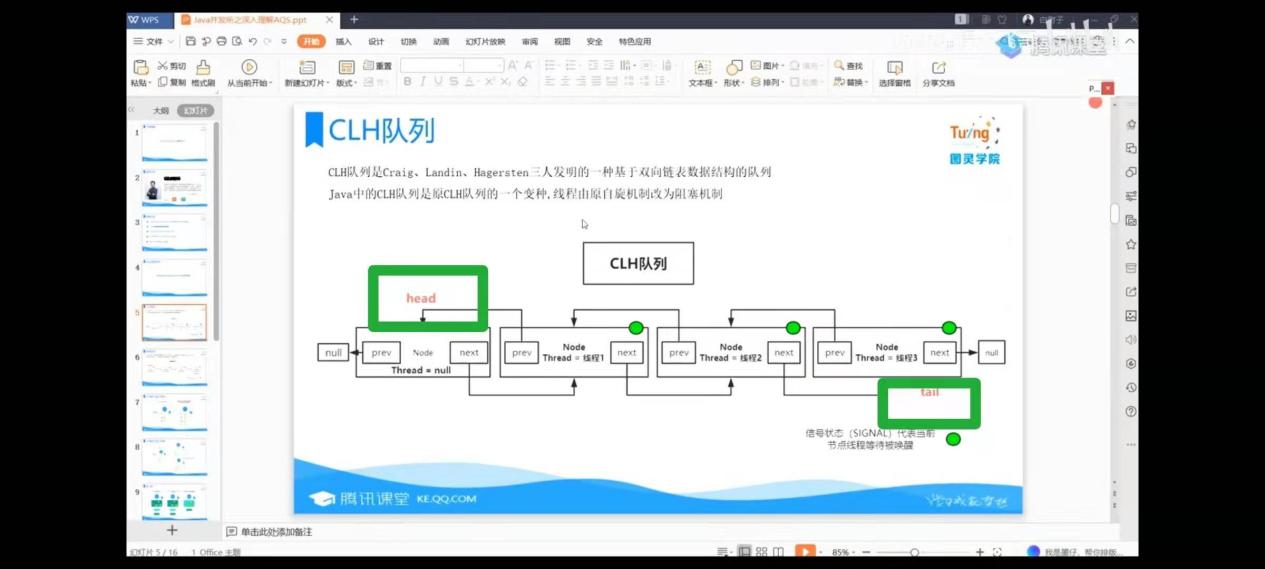
对于公平锁来说FairSync类是ReentractLock的内部静态类，并且有final修饰，并且也实现了Sync类（Sync类是ReentractLock的内部静态抽象类）

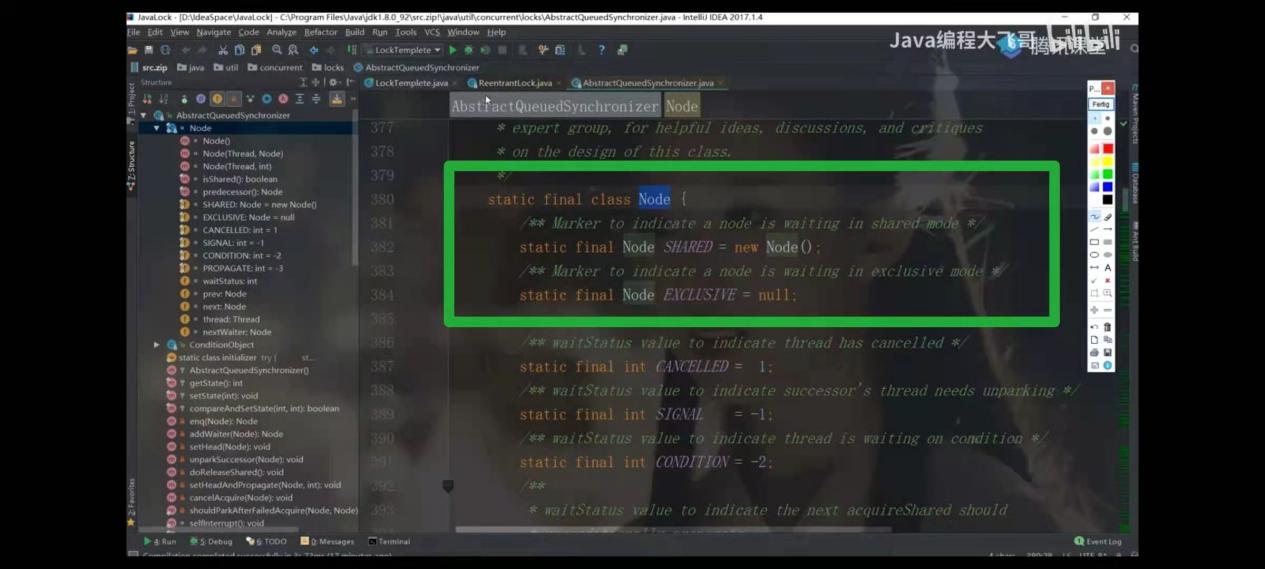
在Sync类当然也是在抽象队列同步器中有四个重要的属性第一个是state，它表示我们的这把锁被锁的次数，也就是说我们的ReentractLock是可重入锁，通过lock. lock加锁，每加一次锁state就会加1，通过lock. unlock释放锁，每执行一次state就减一，直到减完为止。第二个参数叫做exclusiveOwnerThread，它叫做独占线程，记录了当前抢占锁的线程的名称，它的初始值为null，当锁被释放完全之后也是null，

要解释后两个参数就需要我们使用的场景是多线程并发的场景了，当是多线程的时候，线程1过来先通过我们前面讲的CAS算法获取当前锁的状态，判断是否为0，如果为0，也把他设置为1，并且把第二个参数的值设置为线程1的名称，当线程2过来的时候也是通过CAS算法获取当前锁的状态，当为0的时候和上面一样，当不为0的时候就说明其他线程（线程1）还占着锁没有释放，此时就把线程2放到我们的CLH队列（这里的CLH队列是有头节点和尾节点的，当线程2没有获取锁的时候就进入到CLH队列的头节点中，如果还有其他的线程就会根据请求的先后顺序依次进入的这个队列的后面几个节点，也正是这个队列的有序性才使得我们的ReentractLock是一把公平锁，因为这时候当线程1执行完自己的逻辑代码之后释放锁把state设置为0，把第二个参数设置为null的时候我们头节点中的线程2会被唤醒去获取锁…）中去，

我们上面所说的就是AQS的原理分析，视图如下所示：

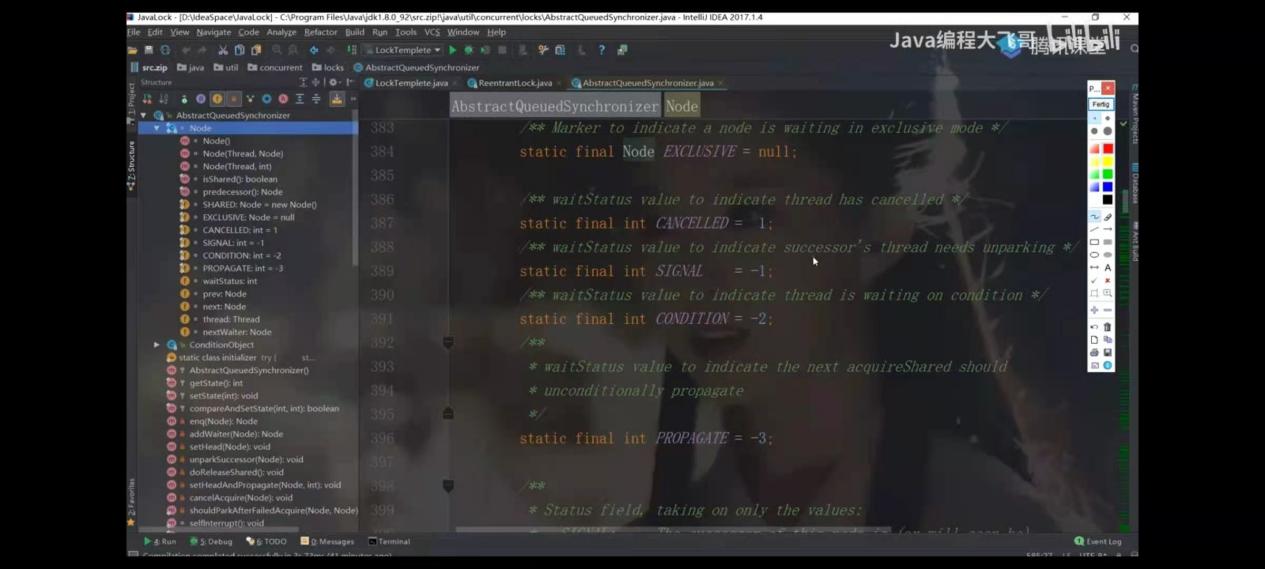






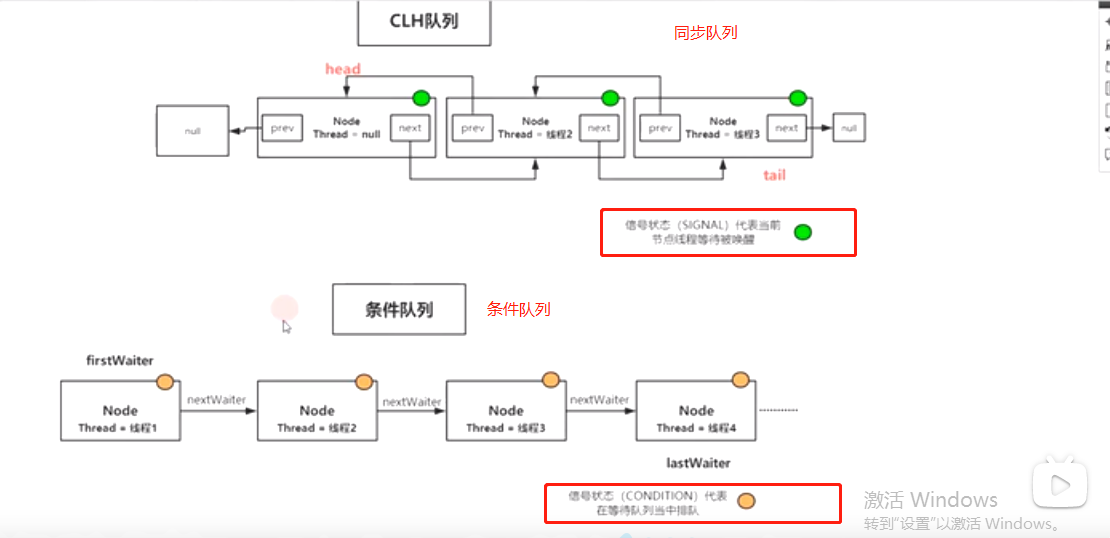
第二张图片是我们上面所说的CLH队列的结构，里面我们标记出来的两个参数第一个表示共享，第二个表示独占，因为在ReentractLock中有一个读锁的实现类是一种共享的锁机制，其他的都是独占的锁机制。这个Node类中还有两个属性都是private Node pre和private Node next分别表示指向前一个节点和指向后一个节点，并且还有一个属性是Thread是用来存储我们放在队列中的线程2，线程3的，

针对第三个参数head就表示我们的头节点和第四个参数tail就表示我们的尾节点，我们如上第一张图所示，



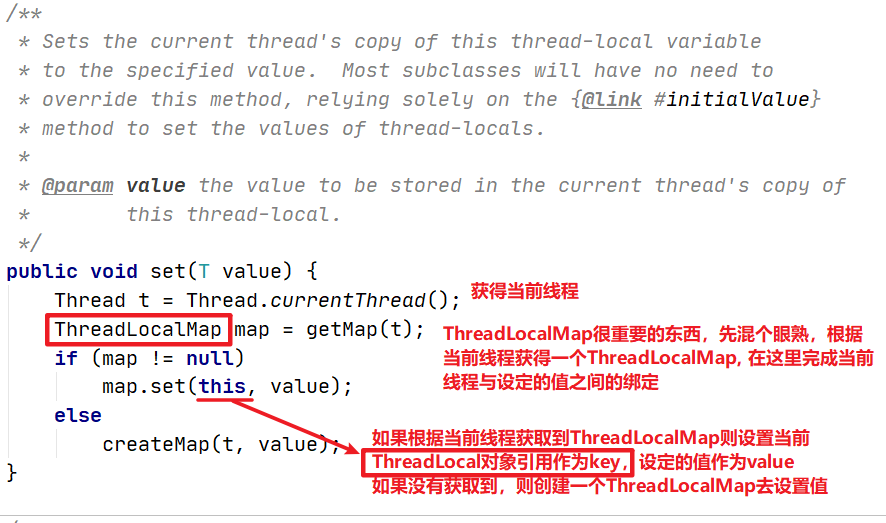
上面这张图中的属性表示我们在CLH队列中的线程的几种状态，如果为1表示线程已经死亡了，我们可以把这个节点从我们的队列中剔除掉，如果是-1的话说明当前节点的线程还是正常状态，还可以被唤醒去抢占锁

当为-2的时候说明当前线程存在于条件队列中并且也是属于正常的等待状态（这里需要注意一下我们这里的条件队列必须是在前面的第二个参数表示独占锁的时候才有条件队列，如果是共享锁的时候就没有条件队列）

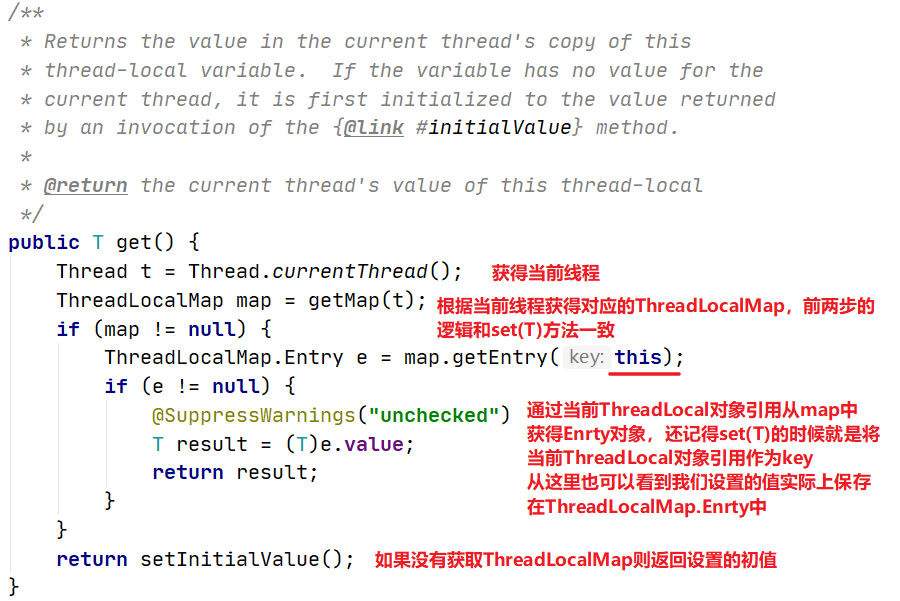


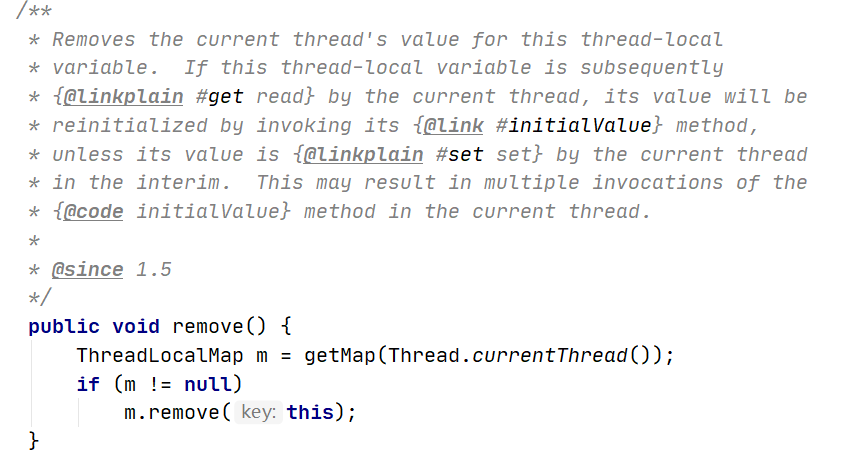
说一下非公平锁，所谓的非公平锁其实就是当我们的线程1释放了锁，但是在我们的LCH队列中还有其他的线程正在排队，当我们使用非公平锁的时候我们新来的线程就不会直接去LCH队列中排队，而是直接去抢占锁。

# 21、关于ThreadLocal类



此处也就是我们ThreadLocalMap中的Entry中的key值就是指向我们在堆中的ThreadLocal的对象的引用，不过这个引用是一个弱引用





②、为什么会产生内存泄漏

|  |
| --- |
| @GetMapping(value = "/install/war") public String getInstallWar(){  return "war包测试"; }  public void set(T value) {  Thread t = Thread.*currentThread*();  ThreadLocalMap map = getMap(t);  if (map != null)  map.set(this, value);  else  createMap(t, value); } |

set方法里面，先调用到当前线程thread，每个线程里都会有一个threadlocals成员变量，指向对应的ThreadLocalMap ，然后以new出来的引用作为key，和给定的value一块保存起来。

当外部引用解除以后，对应的ThreadLocal对象由于被内部ThreadLocalMap 引用，不会GC，可能会导致内存泄露。

而我们的JVM的解决办法就是使用弱引用：

|  |
| --- |
| static class Entry extends WeakReference<ThreadLocal<?>> {  */\*\* The value associated with this ThreadLocal. \*/* Object value;   Entry(ThreadLocal<?> k, Object v) {  super(k);  value = v;  } } |

继承了一个软引用，在系统进行gc的时候就可以回收

但是回收以后，key变成null，value也无法被访问到，还是可能存在内存泄露。 因此一旦不用了，必须对里面的keyvalue对remove掉，否则就会有内存泄露；而且在threadlocal源码里面，在每次get或者set的时候会清楚里面key为value的记录

# 22、关于ReentrantLock

**①、ReentrantLock实现了Lock接口，内部有三个内部类，**Sync、NonfairSync、FairSync，Sync是一个抽象类型，它继承AbstractQueuedSynchronizer，这个AbstractQueuedSynchronizer是一个模板类，它实现了许多和锁相关的功能，并提供了钩子方法供用户实现，比如tryAcquire，tryRelease等。Sync实现了AbstractQueuedSynchronizer的tryRelease方法。NonfairSync和FairSync两个类继承自Sync，实现了lock方法，然后分别公平抢占和非公平抢占针对tryAcquire有不同的实现。

**②、首先，AbstractQueuedSynchronizer继承自**AbstractOwnableSynchronizer，AbstractOwnableSynchronizer的实现很简单，它表示独占的同步器，内部使用变量exclusiveOwnerThread表示独占的线程。

其次，AbstractQueuedSynchronizer内部使用CLH锁队列来将并发执行变成串行执行。整个队列是一个双向链表。每个CLH锁队列的节点，会保存前一个节点和后一个节点的引用，当前节点对应的线程，以及一个状态。这个状态用来表明该线程是否应该block。当节点的前一个节点被释放的时候，当前节点就被唤醒，成为头部。新加入的节点会放在队列尾部。

**③、在初始化ReentrantLock的时候，如果我们不传参数是否公平，**那么默认使用非公平锁，也就是NonfairSync。

当我们调用ReentrantLock的lock方法的时候，实际上是调用了NonfairSync的lock方法，这个方法先用CAS操作，去尝试抢占该锁。如果成功，就把当前线程设置在这个锁上，表示抢占成功。如果失败，则调用acquire模板方法，等待抢占。代码如下：

|  |
| --- |
| static final class NonfairSync extends Sync {  final void lock() {  if (compareAndSetState(0, 1))  setExclusiveOwnerThread(Thread.*currentThread*());  else  acquire(1);  }  protected final boolean tryAcquire(int acquires) {  return nonfairTryAcquire(acquires);  } } |

**④、调用acquire(1)实际上使用的是AbstractQueuedSynchronizer的acquire方**法，**它是一套锁抢占的模板，总体原理是先去尝试获取锁，如果没有获取成功，就在CLH队列中增加一个当前线程的节点，表示等待抢占。然后进入CLH队列的抢占模式，进入的时候也会去执行一次获取锁的操作，如果还是获取不到，就调用LockSupport.park将当前线程挂起。那么当前线程什么时候会被唤醒呢？当持有锁的那个线程调用unlock的时候，会将CLH队列的头节点的下一个节点上的线程唤醒，调用的是LockSupport.unpark方法。acquire代码比较简单**，具体如下：

|  |
| --- |
| public final void acquire(int arg) {  if (!tryAcquire(arg) &&  acquireQueued(addWaiter(Node.EXCLUSIVE), arg))  selfInterrupt(); } |

**⑤、acquire方法内部先使用tryAcquire这个钩子方法去尝试再次获取锁，这个方法在NonfairSync这个类中其实就是使用了nonfairTryAcquire，具体实现原理是先比较当前锁的状态是否是0，如果是0，则尝试去原子抢占这个锁（设置状态为1，然后把当前线程设置成独占线程），如果当前锁的状态不是0，就去比较当前线程和占用锁的线程是不是一个线程，如果是，会去增加状态变量的值，从这里看出可重入锁之所以可重入，就是同一个线程可以反复使用它占用的锁。如果以上两种情况都不通过，则返回失败false。**代码如下：

|  |
| --- |
| final boolean nonfairTryAcquire(int acquires) {  final Thread current = Thread.*currentThread*();  int c = getState();  if (c == 0) {  if (compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  return true;  }  }  else if (current == getExclusiveOwnerThread()) {  int nextc = c + acquires;  if (nextc < 0) // overflow  throw new Error("Maximum lock count exceeded");  setState(nextc);  return true;  }  return false; } boolean nonfairTryAcquire(int acquires) {  final Thread current = Thread.currentThread();  int c = getState();  if (c == 0) {  if (compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  return true;  }  }  else if (current == getExclusiveOwnerThread()) {  int nextc = c + acquires;  if (nextc < 0) // overflow  throw new Error("Maximum lock count exceeded");  setState(nextc);  return true;  }  return false; } |

**⑥、tryAcquire一旦返回false，就会则进入acquireQueued流程，也就是基于**CLH队列的抢占模式：

* 首先，在CLH锁队列尾部增加一个等待节点，这个节点保存了当前线程，通过调用addWaiter实现，这里需要考虑初始化的情况，在第一个等待节点进入的时候，需要初始化一个头节点然后把当前节点加入到尾部，后续则直接在尾部加入节点就行了。

|  |
| --- |
| private Node addWaiter(Node mode) {  // 初始化一个节点，这个节点保存当前线程  Node node = new Node(Thread.*currentThread*(), mode);  // 当CLH队列不为空的视乎，直接在队列尾部插入一个节点  Node pred = tail;  if (pred != null) {  node.prev = pred;  if (compareAndSetTail(pred, node)) {  pred.next = node;  return node;  }  }  // 当CLH队列为空的时候，调用enq方法初始化队列  enq(node);  return node; }  private Node enq(final Node node) {  for (;;) {  Node t = tail;  if (t == null) { // 初始化节点，头尾都指向一个空节点  if (compareAndSetHead(new Node()))  tail = head;  } else {// 考虑并发初始化  node.prev = t;  if (compareAndSetTail(t, node)) {  t.next = node;  return t;  }  }  } } Node addWaiter(Node mode) {  // 初始化一个节点，这个节点保存当前线程  Node node = new Node(Thread.currentThread(), mode);  // 当CLH队列不为空的视乎，直接在队列尾部插入一个节点  Node pred = tail;  if (pred != null) {  node.prev = pred;  if (compareAndSetTail(pred, node)) {  pred.next = node;  return node;  }  }  // 当CLH队列为空的时候，调用enq方法初始化队列  enq(node);  return node; }  private Node enq(final Node node) {  for (;;) {  Node t = tail;  if (t == null) { // 初始化节点，头尾都指向一个空节点  if (compareAndSetHead(new Node()))  tail = head;  } else {// 考虑并发初始化  node.prev = t;  if (compareAndSetTail(t, node)) {  t.next = node;  return t;  }  }  } } |

* 将节点增加到CLH队列后，进入acquireQueued方法。
* 首先，外层是一个无限for循环，如果当前节点是头节点的下个节点，并且通过tryAcquire获取到了锁，说明头节点已经释放了锁，当前线程是被头节点那个线程唤醒的，这时候就可以将当前节点设置成头节点，并且将failed标记设置成false，然后返回。至于上一个节点，它的next变量被设置为null，在下次GC的时候会清理掉。如果本次循环没有获取到锁，就进入线程挂起阶段，也就是shouldParkAfterFailedAcquire这个方法。代码如下：

|  |
| --- |
| final boolean acquireQueued(final Node node, int arg) {  boolean failed = true;  try {  boolean interrupted = false;  for (;;) {  final Node p = node.predecessor();  if (p == head && tryAcquire(arg)) {  setHead(node);  p.next = null; // help GC  failed = false;  return interrupted;  }  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  } } boolean acquireQueued(final Node node, int arg) {  boolean failed = true;  try {  boolean interrupted = false;  for (;;) {  final Node p = node.predecessor();  if (p == head && tryAcquire(arg)) {  setHead(node);  p.next = null; // help GC  failed = false;  return interrupted;  }  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  } } |

* 如果尝试获取锁失败，就会进入shouldParkAfterFailedAcquire方法，会判断当前线程是否挂起，如果前一个节点已经是SIGNAL状态，则当前线程需要挂起。如果前一个节点是取消状态，则需要将取消节点从队列移除。如果前一个节点状态是其他状态，则尝试设置成SIGNAL状态，并返回不需要挂起，从而进行第二次抢占。完成上面的事后进入挂起阶段。代码如下：

|  |
| --- |
| private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  int ws = pred.waitStatus;  if (ws == Node.SIGNAL)  //  return true;  if (ws > 0) {  //  do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  //  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  }  return false; } static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  int ws = pred.waitStatus;  if (ws == Node.SIGNAL)  //  return true;  if (ws > 0) {  //  do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  //  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  }  return false; } |

* 当进入挂起阶段，会进入parkAndCheckInterrupt方法，则会调用LockSupport.park(this)将当前线程挂起。代码：

|  |
| --- |
| private final boolean parkAndCheckInterrupt() {  LockSupport.park(this);  return Thread.*interrupted*(); } final boolean parkAndCheckInterrupt() {  LockSupport.park(this);  return Thread.*interrupted*(); } |

**⑦、非公平锁的unlock方法**

1、调用unlock方法，其实是直接调用AbstractQueuedSynchronizer的release操作。

2、进入release方法，内部先尝试tryRelease操作,主要是去除锁的独占线程，然后将状态减一，这里减一主要是考虑到可重入锁可能自身会多次占用锁，只有当状态变成0，才表示完全释放了锁。

3、一旦tryRelease成功，则将CHL队列的头节点的状态设置为0，然后唤醒下一个非取消的节点线程。

4、一旦下一个节点的线程被唤醒，被唤醒的线程就会进入acquireQueued代码流程中，去获取锁。

具体代码如下：

|  |
| --- |
| public void unlock() {  sync.release(1); } void unlock() {  sync.release(1); } |

release方法代码：

|  |
| --- |
| public final boolean release(int arg) {  if (tryRelease(arg)) {  Node h = head;  if (h != null && h.waitStatus != 0)  unparkSuccessor(h);  return true;  }  return false; } final boolean release(int arg) {  if (tryRelease(arg)) {  Node h = head;  if (h != null && h.waitStatus != 0)  unparkSuccessor(h);  return true;  }  return false; } |

Sync中通用的tryRelease方法代码：

|  |
| --- |
| protected final boolean tryRelease(int releases) {  int c = getState() - releases;  if (Thread.*currentThread*() != getExclusiveOwnerThread())  throw new IllegalMonitorStateException();  boolean free = false;  if (c == 0) {  free = true;  setExclusiveOwnerThread(null);  }  setState(c);  return free; } final boolean tryRelease(int releases) {  int c = getState() - releases;  if (Thread.currentThread() != getExclusiveOwnerThread())  throw new IllegalMonitorStateException();  boolean free = false;  if (c == 0) {  free = true;  setExclusiveOwnerThread(null);  }  setState(c);  return free; } |

unparkSuccessor代码：

|  |
| --- |
| private void unparkSuccessor(Node node) {  int ws = node.waitStatus;  if (ws < 0)  compareAndSetWaitStatus(node, ws, 0);  Node s = node.next;  if (s == null || s.waitStatus > 0) {  s = null;  for (Node t = tail; t != null && t != node; t = t.prev)  if (t.waitStatus <= 0)  s = t;  }  if (s != null)  LockSupport.unpark(s.thread); } void unparkSuccessor(Node node) {  int ws = node.waitStatus;  if (ws < 0)  compareAndSetWaitStatus(node, ws, 0);  Node s = node.next;  if (s == null || s.waitStatus > 0) {  s = null;  for (Node t = tail; t != null && t != node; t = t.prev)  if (t.waitStatus <= 0)  s = t;  }  if (s != null)  LockSupport.unpark(s.thread); } |

**⑧、公平锁和非公平锁的区别**

平锁和非公平锁，在CHL队列抢占模式上都是一致的，也就是在进入acquireQueued这个方法之后都一样，它们的区别在初次抢占上有区别，也就是tryAcquire上的区别，

真正的区别就是公平锁多了hasQueuePredecessors这个方法，这个方法用于判断CHL队列中是否有节点，对于公平锁，如果CHL队列有节点，则新进入竞争的线程一定要在CHL上排队，而非公平锁则是无视CHL队列中的节点，直接进行竞争抢占，这就有可能导致CHL队列上的节点永远获取不到锁，这就是非公平锁之所以不公平的原因。

**总结：**

线程使用ReentrantLock获取锁分为两个阶段，第一个阶段是初次竞争，第二个阶段是基于CHL队列的竞争。在初次竞争的时候是否考虑队列节点直接区分出了公平锁和非公平锁。在基于CHL队列的锁竞争中，依靠CAS操作保证原子操作，依靠LockSupport来做线程的挂起和唤醒，使用队列来保证并发执行变成了串行执行，从而消除了并发所带来的问题。