**进程是资源分配的最小单位；线程是CPU调度的最小单位**

**什么是进程：**进程是资源分配的最小单位

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

**什么是线程：**线程是CPU调度的最小单位

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的**堆**和**方法区**资源，但每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

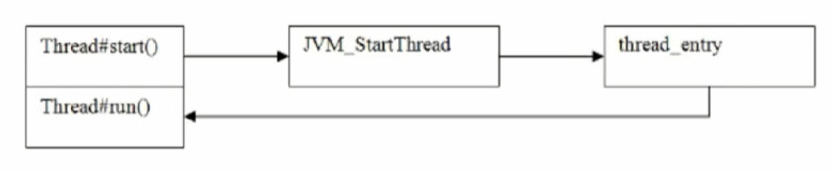
**进程与线程的区别**

* 线程不能独立应用，而进程可以看错独立应用
* 进程有独立的地址空间，相互不影响，线程只是进程的不同执行路径
* 线程没有独立的地址空间，多线程共享进程的堆内存，但是独有自己的程序计数器、虚拟机栈、本地方法栈等
* 进程的切换比线程开销大

**进程与线程的联系**

* Java对操作系统提供的功能进行封装，包括进程和线程
* 运行一个程序会产生一个进程，进程包含至少一个线程
* 每个进程对应一个JVM实例，多个线程共享JVM的堆
* Java采用单线程编程模型，程序会自动创建主线程
* 主线程创建子线程，原则上要后于子线程完成执行
* JVM实例创建的时候会创建多个线程，包括mian线程、GC线程等

**Thread的run()方法和start()的区别**



Thread.start()🡪 native Thread.start0()🡪JVM\_StartThread🡪thread\_entry(run().getName())

* 调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。
* 直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

**Runnable和Thread 是什么关系**

* Thread是实现Runnable接口的类，使得run支持多线程，Runnable是接口
* 因类的单一继承原则，推荐使用Runnable

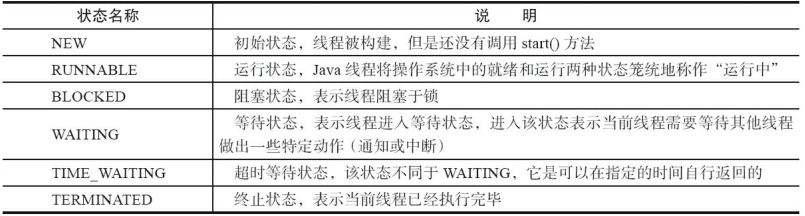
**Run方法传参**

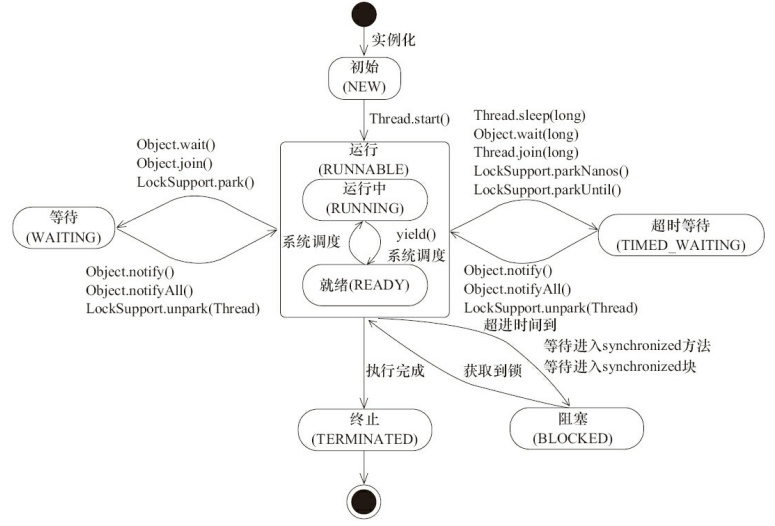
* 通过构造方法传递数据
* 通过变量和方法传递数据
* 通过回调函数传递数据

**如何实现-处理线程的返回值**

* 主线程等待法
* 使用Thread类的join()阻塞当前线程以等待子线程完成
* 通过Callable接口实现：FutureTask获取；通过线程池获取

**线程的生命周期和状态**





以下方法会让线程陷入无限期等待

* 没有设置TimeOut参数的Object.wait()方法
* 没有设置TimeOut参数的Thread.join()方法
* LockSupport.park()方法

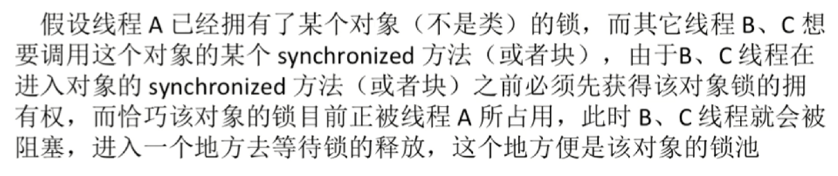
以下方法会让线程陷入限期等待

* 设置TimeOut参数的Object.wait()方法
* 设置TimeOut参数的Thread.join()方法
* Thread.sleep()方法
* LockSupport.parkNanos()方法
* LockSupport.parkUntil()方法

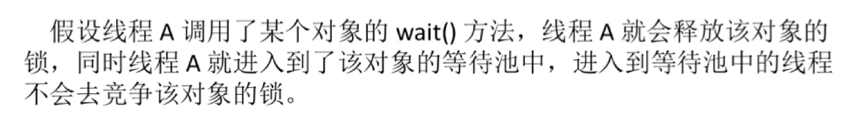
**Sleep()和wait()方法的区别**

* Sleep()方法是Thread类的，wait()方法是Object类中定义的
* **Thread.sleep()只会让出CPU，不会导致锁行为的改变；Object.wait()不仅能让出CPU，还会释放已经占用的同步资源锁**
* Sleep()方法可以在任何地方使用；Wait()方法只能在synchronized方法或者synchronized修饰的代码块中使用

**锁池EntryList**



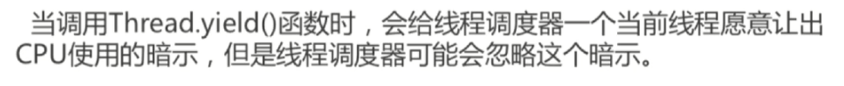
**等待池WaitSet**



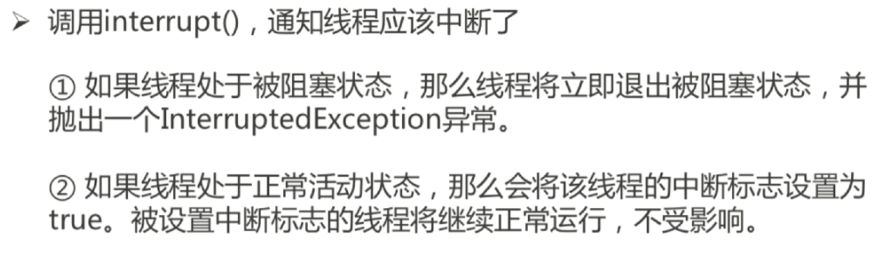
**Notify和notifyAll的区别**

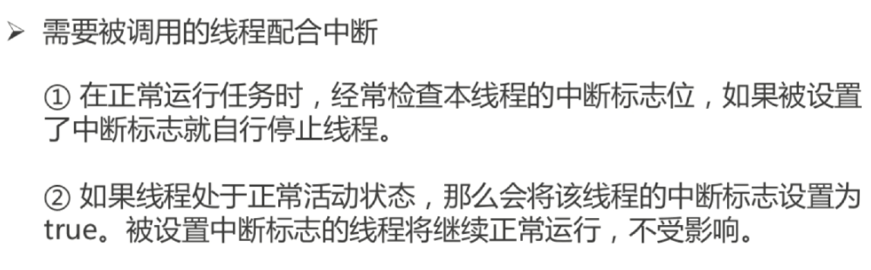
* NotifyAll会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会，已经进入等待池没有获取到锁的线程不会再回到等待池中
* Notify只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会

**yield()**



**interrupt()**





Linux的内核态和用户态，为什么内核态与用户态的要转换，怎么转换？什么是系统中断？

内核态的多线程是如何用轻量级线程实现的

**互斥锁的特性**

**互斥性**：即在同一时间只允许一个线程持有某个对象锁，通过这种特性来实现多线程的协调机制，这样在同一时间只有一个线程对需要同步的代码块（复合操作）进行访问。互斥性也称为操作的原子性

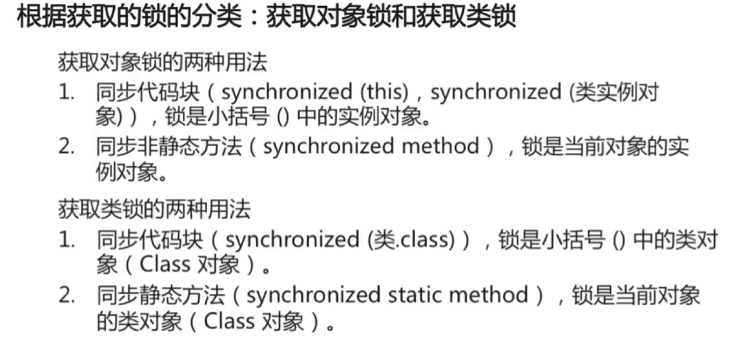
**可见性**：必须确保在锁被释放之前，对共享变量所做的修改，对随后获得该锁的另一个线程是可见的，即在获得锁时应获得最新共享变量的值，否则另一个线程可能是在本地缓存的某个副本上继续操作，从而引起不一致

**Synchronized**

**synchronized关键字最主要的三种使用方式**

* **修饰实例方法:** 给当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
* **修饰静态方法:** 给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
* **修饰代码块:** 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

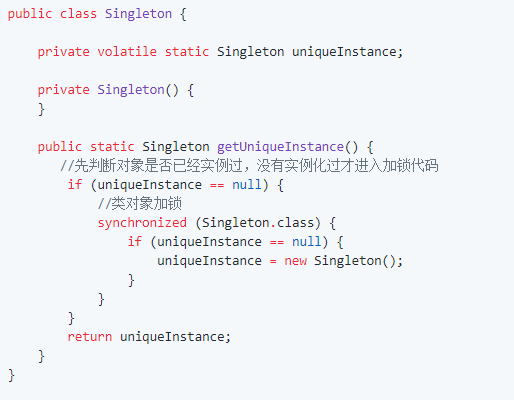
**总结：** synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓存功能



**Synchronized可重入性**

从互斥多的设计上来说，当一个线程试图操作一个有其他先吃持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有的对象锁的临界资源时，这种情况属于重入。

**双重校验锁实现对象单例（线程安全）**



另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

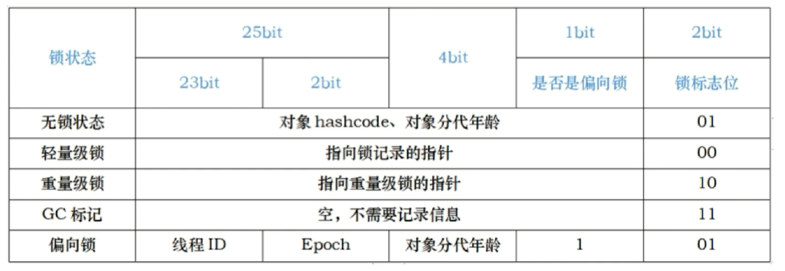
**使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行**。

**synchronized 关键字的底层原理**

**对象头**



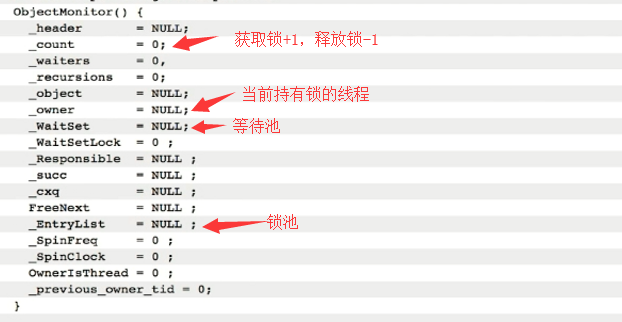
**MarkWord**



**Monitor：**

每个Java对象会自带一个锁，JVM中是通过ObjectMonitor实现的

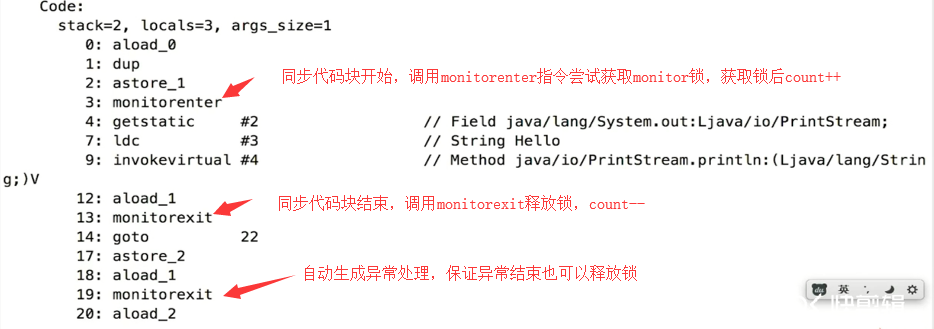
(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因)



**分别介绍同步代码块和同步方法锁机制的原理**

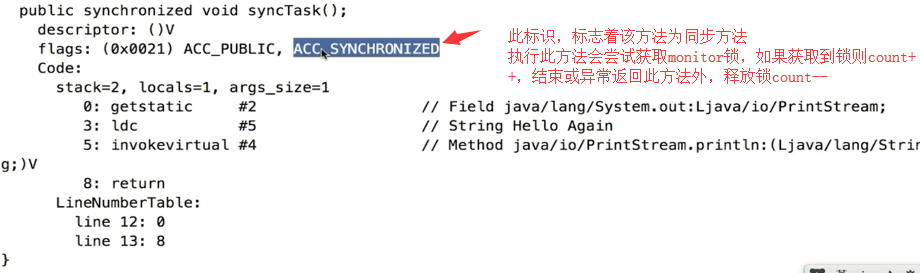


**同步代码块**



**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。** 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。如果同一个线程多次获取锁，则锁计数器将持续++1，释放锁也逐个—1

**同步方法**



synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。同样也是通过对monitor加锁实现的，同样需要锁计数器计数。

**synchronized 优化**

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

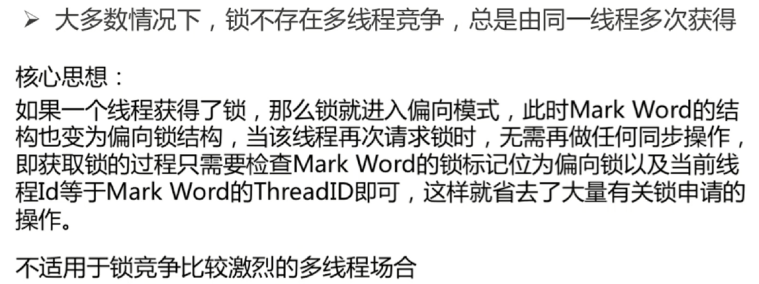
锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

①偏向锁

引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步

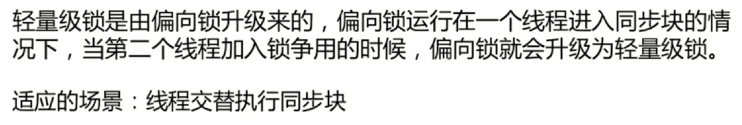
但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。



② 轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)。轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了CAS操作。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢，如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁



③ 自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。

何谓自旋锁，通过让线程执行忙循环等待锁的释放，此时并不让出CPU

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过--XX:+UseSpinning参数来开启。JDK1.6及1.6之后，就改为默认开启的了。

需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。自旋次数的默认值是10次，用户可以修改--XX:PreBlockSpin来更改。

另外,在 JDK1.6 中引入了自适应的自旋锁。自旋的次数不再固定，由前一次在同一个锁上的自旋时间及锁拥有者的状态来决定

④ 锁消除

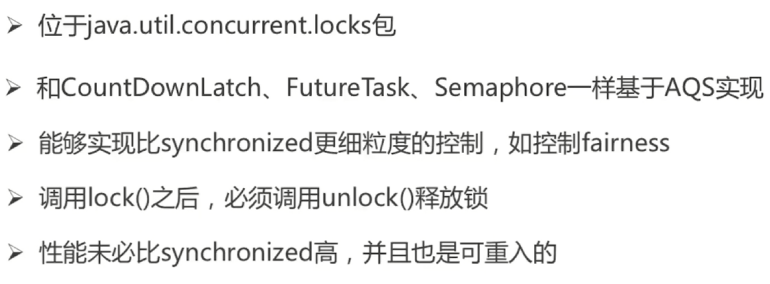
锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

⑤ 锁粗化

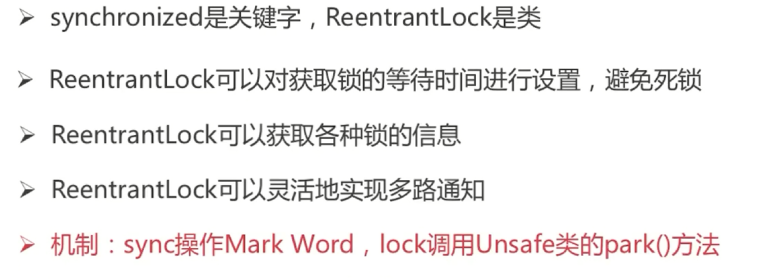
通过扩大锁的范围，避免反复加锁和解锁

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小，只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。但是如果多次反复多次加锁和解锁，性能也会影响很大。

**ReenTrantLock**



**ReenTrantLock 与synchronized区别**



**① 两者都是可重入锁**

**② synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API**

synchronized 是依赖于 JVM 实现的，ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

**③ ReenTrantLock 比 synchronized 增加了一些高级功能**

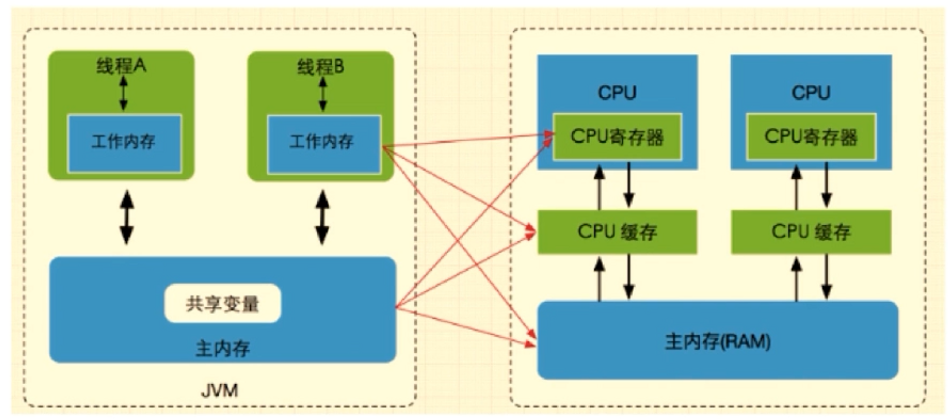
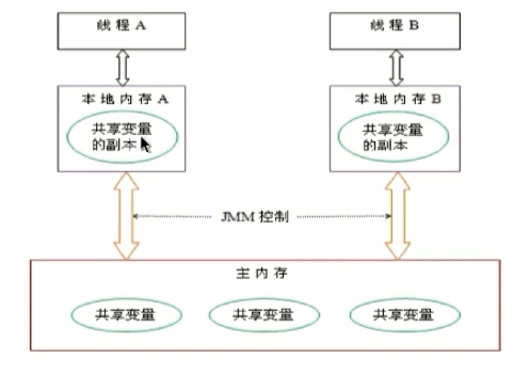
相比synchronized，ReenTrantLock增加了一些高级功能。主要来说主要有三点：**①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**

* **ReenTrantLock提供了一种能够中断等待锁的线程的机制**，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
* **ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。** ReenTrantLock默认情况是非公平的，可以通过 ReenTrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
* synchronized关键字与wait()和notify/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），**线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”** ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

**JMM(Java Memory Model)**

Java内存模型(Java Memory Model)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。其模型如下所示：

**JMM如何解决可见性问题**



**JMM中的主内存**

* 存储Java实例对象
* 包括成员变量、类信息、常量、静态变量等
* 属于数据共享的区域，多线程并发操作时会引发线程安全问题

**JMM中的工作内存**

* 存储当前方法的所有本地变量信息，本地变量对其他线程不可见
* 字节码行号指示器、Native方法信息
* 属于线程私有数据区域，不存在线程安全问题

**JMM与Java内存区域划分是不同的概念层次**

* JMM描述的是一组规则，围绕原子性、有序性、可见性展开
* 相似点：存在共享区域和私有区域

**主内存和工作内存的数据存储类型及操作方式归纳**

* 方法里的基本数据类型本地变量将直接存储在工作内存的栈帧结构中
* 引用类型的本地变量：引用存储在工作内存中，实例存储在主内存中
* 成员变量、static变量、类信息均会被存储在住内存中
* 主内存共享的方式是线程各拷贝一份数据到工作内存，操作完成后刷新回主内存

**指令重排序需要满足的条件**

* 在单线程环境下不能改变程序的运行结果
* 存在数据依赖关系的不允许重排序
* 无法通过happens-before原则推导出来的，才能重进行指令排序

**Happens-before规则**

* 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先于写在后面的操作
* 锁定规则：unlock操作先于后面对同一个锁的lock操作
* Volatile变量规则：对一个变量的写操作先于后面对这个变量的读操作
* 传递规则：操作A先于操作B，操作B先于操作C，则操作A一定先于操作C
* 线程启动规则：Thread对象的start()方法先于此线程的每一个动作发生
* 线程中断规则：对线程interrupt()方法的调用，先于中断线程的代码检测到中断事件
* 线程终结规则：线程中所有的操作都发生在线程的终止检测前，可以通过Thread.jion()方法结束，Thread.isAlive()方法检测线程释放终止
* 对象终结规则：一个对象的初始化完成先于它的finalize()方法的开始

**Happens-before的概念**

* 如果两个操作不满足上述的任何一个happens-before规则，那么这两个操作就没有顺序保障，JVM可以对这两个操作进行重排序；
* 如果操作A happe-before操作B，那么操作A在内存上所做的操作对操作B都是可见的

**Volatile**

* JVM提供的轻量级别的同步机制
* 保证被volatile修饰的共享变量对所有线程总是可见的
* 禁止指令重排序优化
* Volatile保证可见性，但是不保证原子性，例如 自加自减操作不是原子性操作，所有对自加自减操作的变量，volatile保证不了可见性

**Volatile如何禁止重排序优化**

* 通过插入内存屏障指令禁止在内存屏障前后的指令执行重排序优化
  + 内存屏障(Memory Barrier)
    - 保证特点操作的执行顺序
    - 保证某些变量的内存可见性
* 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

**Volatile和synchronized的区别**

* Volatile本质是告诉JVM当前变量在寄存器（工作内存）中的值是不确定的，需要从主内存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞，直到该线程完成变量操作为止
* Volatile仅能使用在变量级别，synchronized则可以使用在变量、方法和类级别
* Volatile仅能实现变量的可见性，不能保证原子性；synchronized可以保证可见性和原子性
* Volatile不会造成线程阻塞，synchronized可能会造成线程阻塞
* Volatile标记的变量不会被编译器优化执行数据线；synchronized标记的变量可以被优化

源码：

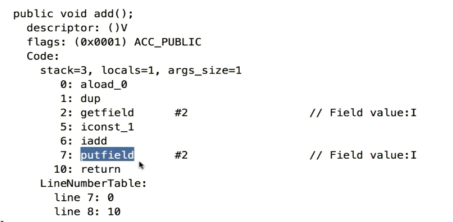
public volatile int value;

public void add() {

value++;

}

字节码：



Volatile能保证可见性，但是不能保证上述三个指令的原子性，解决方法就是用synchronized或者用AtomicInteger

**JAVA线程池**

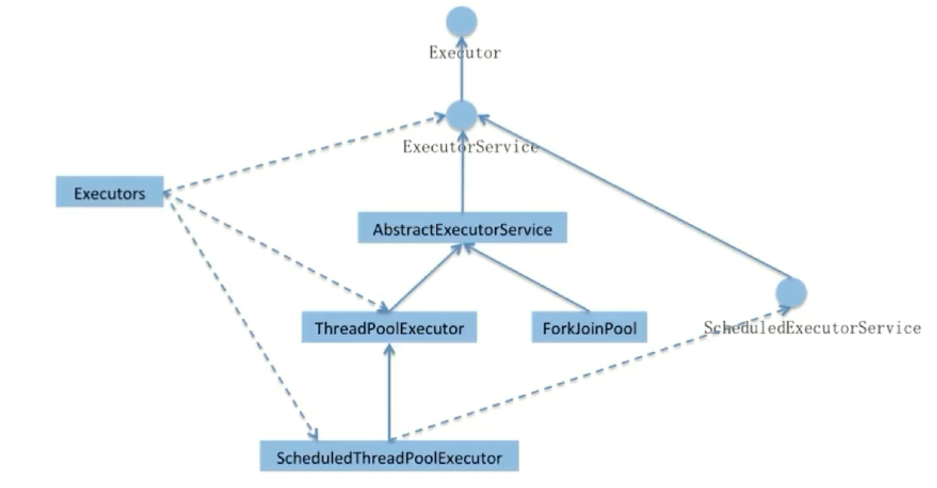
**为什么要使用线程池**

* 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
* 提高响应速度。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，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

**JUC的三个Executor接口**

1. Executor：运行新任务的简单接口，将任务提交和任务执行细节解耦
2. ExecutorService：具备管理执行器和任务生命周期的方法，提交任务机制更完善
3. ScheduledExecutorService：支持Future和定期执行任务

**Executor框架**



**Executor创建不同的线程池**

1. newFixedThreadPool（int nThreads）

* 指定工作线程数量的线程池
* 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

1. newCachedThreadPool（）

* 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。
* 处理大量短时间工作任务的线程池
  + 1. 试图缓存线程并重用，当无缓存线程可用是，就会创建新的工作线程
    2. 如果线程闲置的时间超过阈值，则会被终止并移出缓存
    3. 系统长时间闲置的时候，不会消耗什么资源

1. newSingleThreadExecutor（）

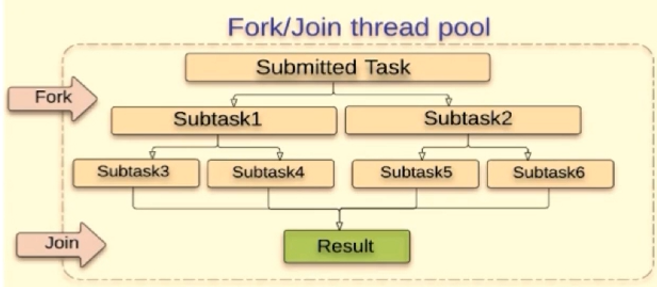
* 创建唯一的工作线程来执行任务，如果线程异常结束，会有另一个线程取代它
* 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。

1. newSingleThreadScheduledExecutor（）和newScheduledThreadPool(int corePoolSize)

定时或周期性的工作调度，两者之间的区别是单一工作线程还是多线程

1. newWorkStealingPool()

* 内部会构建ForkJoinPool，利用working-stealing算法，并行处理任务，不保证处理顺序
* Fork/Join框架
  + 把大任务分割成若干个小任务并行执行，最终汇总每个小人物结果后得到大任务结果的框架



**Executors 返回线程池对象的弊端**

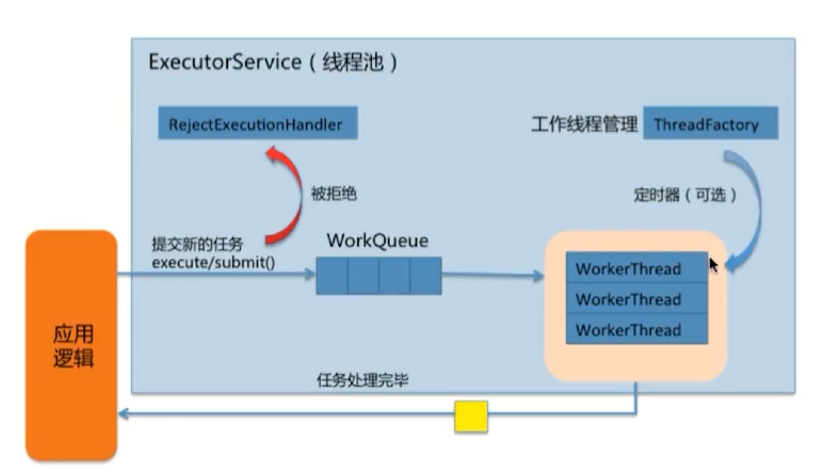
* FixedThreadPool和 SingleThreadExecutor

允许请求的队列长度为 Integer.MAX\_VALUE，可能堆积大量的请求，从而导致OOM。

* CachedThreadPool 和 ScheduledThreadPool

允许创建的线程数量为 Integer.MAX\_VALUE ，可能会创建大量线程，从而导致OOM

**ThreadPoolExecutor**



为什么设计为队列满+核心线程数满了才创建新线程？

[**https://blog.csdn.net/weixin\_35422230/article/details/114540632**](https://blog.csdn.net/weixin_35422230/article/details/114540632)

**workqueue工作原理**

[**https://blog.csdn.net/tugangkai/article/details/82849172**](https://blog.csdn.net/tugangkai/article/details/82849172)

**ThreadPoolExecutor的构造函数**

* corePoolSize : 核心线程数线程数定义了最小可以同时运行的线程数量。
* maximumPoolSize : 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
* workQueue: 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。
* keepAliveTime:当线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁；
* unit : keepAliveTime 参数的时间单位。
* threadFactory :executor 创建新线程的时候会用到。
* handler :饱和策略

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，ThreadPoolTaskExecutor 定义一些策略:

* 1. ThreadPoolExecutor.AbortPolicy

抛出 RejectedExecutionException来拒绝新任务的处理。

* 1. ThreadPoolExecutor.CallerRunsPolicy

用调用者所在的线程执行任务。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。

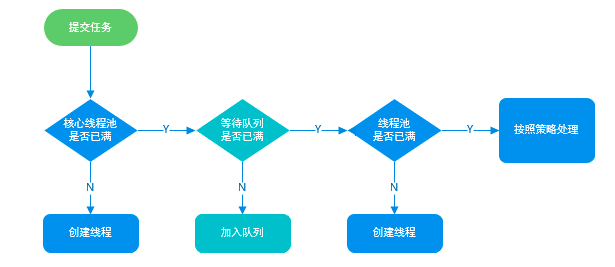
* 1. ThreadPoolExecutor.DiscardPolicy

不处理新任务，直接丢弃掉。

* 1. ThreadPoolExecutor.DiscardOldestPolicy

此策略将丢弃最早的未处理的任务请求，即丢弃队列中最靠前的任务

**线程池原理分析-新任务提交execute执行过程**

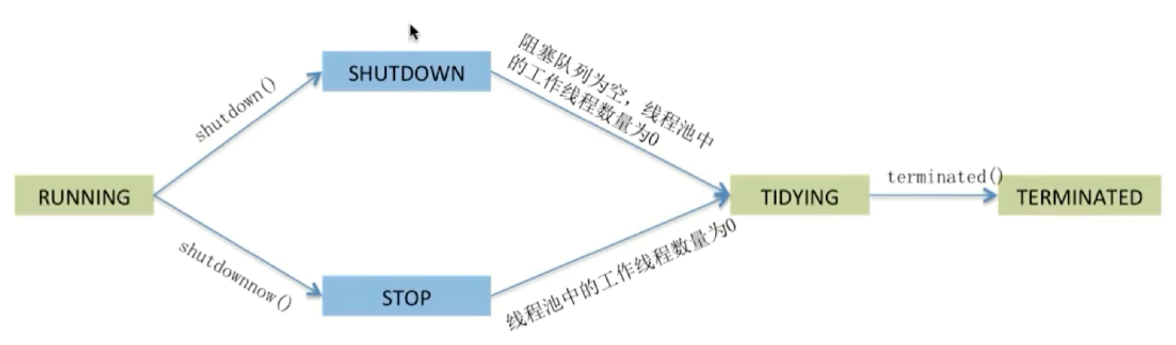


* 如果运行线程少于corePoolSize，则创建新线程来处理任务，即使线程池中的其他线程是空闲的
* 如果线程池中的线程数量大于corePoolSize且小于maxmumPoolSize，则只有当workQueue满的时候才能创建新的线程
* 如果设置的corePoolSize和maxmumPoolSize相同，则创建的线程池的大小是固定的，这时如果有新的任务提交，若workQueue未满，则将请求放入workQueue中，等待有空闲线程去从workQueue中取出任务并处理
* 如果运行的线程数据大于等于maxmumPoolSize，此时若workQueue已经满了，则通过handler所指定的策略来处理任务

**线程池的状态**

* Running：能接受新提交的任务，并且也能处理阻塞队列中的任务
* Shutdown：不再接受新提交的任务，但可以处理存量任务
* Stop：不再接受新提交的任务，也不处理存量任务
* Tidying：所有的任务都已终止
* Terminated：terminated()方法执行完后进入该状态

**线程池的状态状态流转**



线程池的大小选定（推荐值或经验值）

CPU密集型：线程数=按照核数或核数+1设置

I/O密集型：线程数=CPU核数\*(1+平均等待时间/平均工作时间)

### 程序计数器为什么是私有的?

程序计数器主要有下面两个作用：

* 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
* 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了**线程切换后能恢复到正确的执行位置**

### 虚拟机栈和本地方法栈为什么是私有的?

* **虚拟机栈：** 每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
* **本地方法栈：** 和虚拟机栈所发挥的作用非常相似，区别是： **虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。** 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了**保证线程中的局部变量不被别的线程访问到**，虚拟机栈和本地方法栈是线程私有的。

### 说说并发与并行的区别

* **并发：** 同一时间段，多个任务都在执行 (单位时间内不一定同时执行)；
* **并行：** 单位时间内，多个任务同时执行。

### 为什么要使用多线程呢

* 现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。
* 也为了提高 CPU 利用率，目前系统基本上都是多核的。举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU 只会一个 CPU 核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

### 什么是上下文切换

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

**当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换**

### 什么是线程死锁?如何避免死锁?

线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

**产生死锁必须具备以下四个条件**

* 互斥条件：该资源任意一个时刻只由一个线程占用。
* 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
* 不剥夺条件:线程已获得的资源在末使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
* 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

**如何避免死锁**

我们只要破坏产生死锁的四个条件中的其中一个就可以了

* 破坏互斥条件 ：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
* 破坏请求与保持条件 ：一次性申请所有的资源。
* 破坏不剥夺条件 ：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
* 破坏循环等待条件 ：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

### ThreadLocal

**ThreadLocal简介**

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的专属本地变量该如何解决呢？ JDK中提供的ThreadLocal类正是为了解决这样的问题。 ThreadLocal类主要解决的就是让每个线程绑定自己的值，可以将ThreadLocal类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是ThreadLocal变量名的由来。他们可以使用 get（） 和 set（） 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

**ThreadLocal原理**

public class Thread implements Runnable {

......

//与此线程有关的ThreadLocal值。由ThreadLocal类维护

ThreadLocal.ThreadLocalMap threadLocals = null;

//与此线程有关的InheritableThreadLocal值。由InheritableThreadLocal类维护

ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;

......

}

从上面Thread类源代码可以看出Thread 类中有一个 threadLocals 和 一个 inheritableThreadLocals 变量，它们都是 ThreadLocalMap 类型的变量,我们可以把 ThreadLocalMap 理解为ThreadLocal 类实现的定制化的 HashMap。默认情况下这两个变量都是null，只有当前线程调用 ThreadLocal 类的 set或get方法时才创建它们，实际上调用这两个方法的时候，我们调用的是ThreadLocalMap类对应的 get()、set() 方法。

ThreadLocal类的set()方法

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

}

最终的变量是放在了当前线程的 ThreadLocalMap 中，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是ThreadLocalMap的封装，传递了变量值。 ThrealLocal 类中可以通过Thread.currentThread()获取到当前线程对象后，直接通过getMap(Thread t)可以访问到该线程的ThreadLocalMap对象。

每个Thread中都具备一个ThreadLocalMap，而ThreadLocalMap可以存储以ThreadLocal为key ，Object 对象为 value的键值对。

**ThreadLocal 内存泄露问题**

ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用,而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，ThreadLocalMap 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。使用完 ThreadLocal方法后 最好手动调用remove()方法

### 实现Runnable接口和Callable接口的区别

Runnable 接口不会返回结果或抛出检查异常，但是\*\*Callable 接口\*\*可以

### 执行execute()方法和submit()方法的区别是什么呢？

* execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；
* submit()方法用于提交需要返回值的任务。线程池会返回一个 Future 类型的对象，通过这个 Future 对象可以判断任务是否执行成功，并且可以通过 Future 的 get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

### Atomic 原子类

**介绍一下Atomic 原子类**

Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

**JUC 包中的原子类是哪4类**

**基本类型**：使用原子的方式更新基本类型

AtomicInteger：整形原子类

AtomicLong：长整型原子类

AtomicBoolean：布尔型原子类

**数组类型**：使用原子的方式更新数组里的某个元素

AtomicIntegerArray：整形数组原子类

AtomicLongArray：长整形数组原子类

AtomicReferenceArray：引用类型数组原子类

**引用类型**

AtomicReference：引用类型原子类

AtomicStampedReference：原子更新引用类型里的字段原子类

AtomicMarkableReference ：原子更新带有标记位的引用类型

**对象的属性修改类型**

AtomicIntegerFieldUpdater：原子更新整形字段的更新器

AtomicLongFieldUpdater：原子更新长整形字段的更新器

AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

**能不能给我简单介绍一下 AtomicInteger 类的原理**

AtomicInteger 类的部分源码：

// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

private volatile int value;

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

### AQS

**AQS简介**

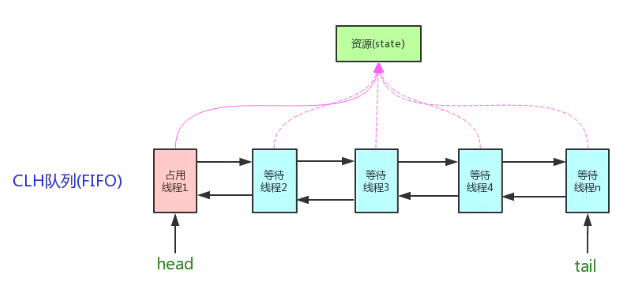
AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。

AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

**AQS 原理**

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

* CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。



AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

private volatile int state;//共享变量，使用volatile修饰保证线程可见性

状态信息通过procted类型的getState，setState，compareAndSetState进行操作

//返回同步状态的当前值

protected final int getState() {

return state;

}

// 设置同步状态的值

protected final void setState(int newState) {

state = newState;

}

//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）

protected final boolean compareAndSetState(int expect, int update) {

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}