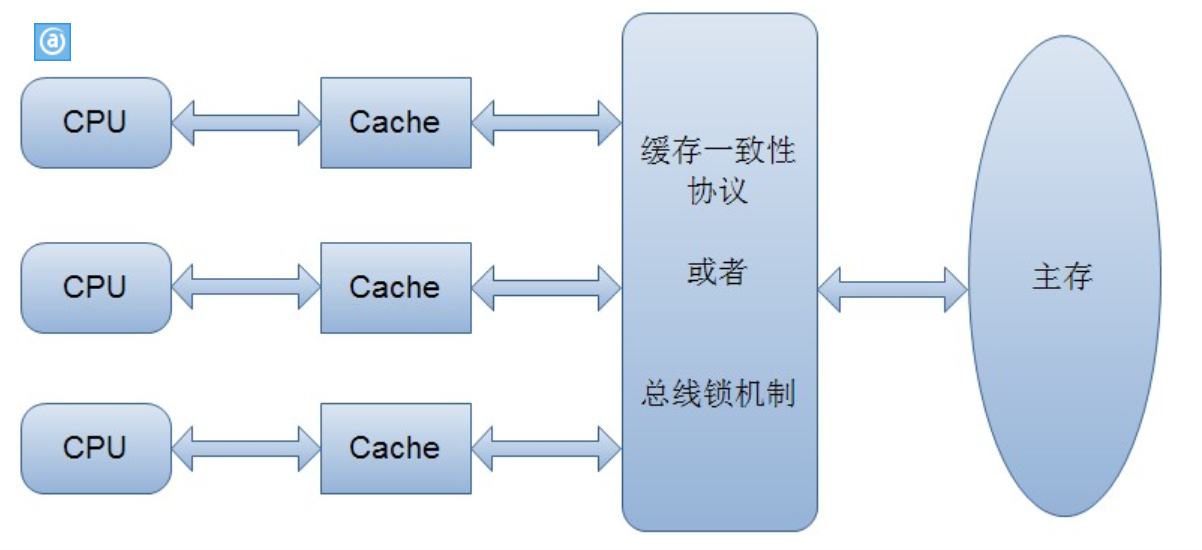
# Java并发

## 并发机制底层原理

### Volatile

#### java内存模型

**Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。**



缓存一致性协议。最出名的就是Intel 的MESI协议，MESI协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

#### 并发编程中的三个概念

**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。

i = 9;

假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：为低16位赋值，为高16位赋值。那么就可能发生一种情况：当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据。

**可见性：**可见性是指当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。

**有序性** ：在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。规则不具体介绍。

#### volatile关键字的两层语义

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。（可见性）

2）禁止进行指令重排序。（一定程度的有序性）

一定程度有序性：

1．当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

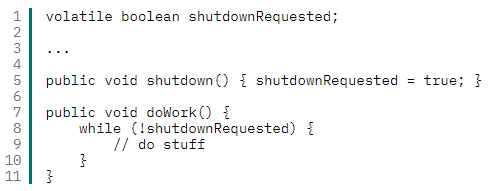
2．在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

在底层通过加lock锁实现内存屏障。

#### 应用场景

要始终牢记使用 volatile 的限制 —— 只有在状态真正独立于程序内其他内容时才能使用 volatile

##### 模式 #1：状态标志

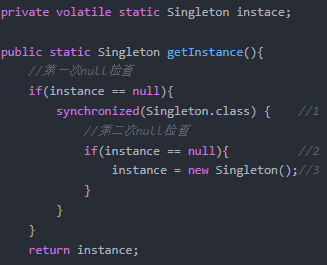


这种类型的状态标记的一个公共特性是：通常只有一种状态转换；shutdownRequested 标志从 false 转换为 true，然后程序停止。这种模式可以扩展到来回转换的状态标志，但是只有在转换周期不被察觉的情况下才能扩展（从 false 到 true，再转换到 false）。

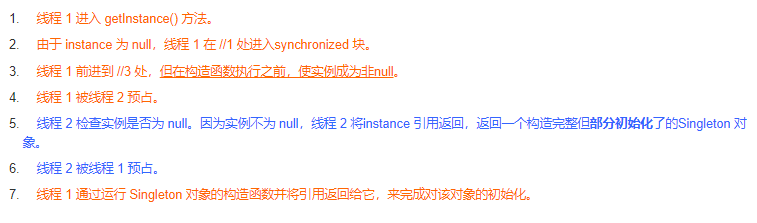
##### 模式 #2：一次性安全发布

在缺乏同步的情况下，可能会遇到某个对象引用的更新值（由另一个线程写入）和该对象状态的旧值同时存在。

这就是造成著名的双重检查锁定（double-checked-locking）问题的根源，其中对象引用在没有同步的情况下进行读操作，产生的问题是您可能会看到一个更新的引用，但是仍然会通过该引用看到不完全构造的对象。

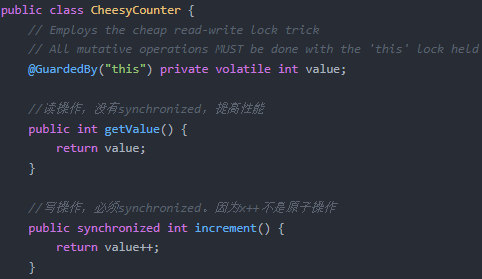


如果不用volatile，则因为内存模型允许所谓的“无序写入”，可能导致失败。**——某个线程可能会获得一个未完全初始化的实例。（禁止指令重排序）**



##### 模式 #3：开销较低的“读－写锁”策略

如果读操作远远超过写操作，您可以结合使用**内部锁**和 **volatile 变量**来减少公共代码路径的开销。如下显示的线程安全的计数器，使用 synchronized 确保增量操作是原子的，并使用 volatile 保证当前结果的可见性。如果更新不频繁的话，该方法可实现更好的性能，因为读路径的开销仅仅涉及 volatile 读操作，这通常要优于一个无竞争的锁获取的开销。



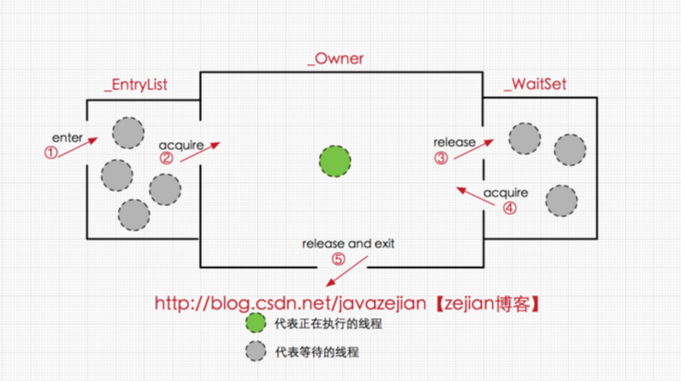
使用锁进行所有变化的操作，使用 volatile 进行只读操作。其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作。

### Synchronized

#### synchronized底层语义原理

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现， 无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法 并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的。

每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由ObjectMonitor实现的，ObjectMonitor中有两个队列，\_WaitSet 和 \_EntryList，用来保存ObjectWaiter对象列表( 每个等待锁的线程都会被封装成ObjectWaiter对象)，\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时，首先会进入 \_EntryList 集合，当线程获取到对象的monitor 后进入 \_Owner 区域并把monitor中的owner变量设置为当前线程同时monitor中的计数器count加1，若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)



补充：synchronized是可重入的

#### synchronized的优化

**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，而且总是由同一线程多次获得，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。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。下面我们接着了解轻量级锁。

**轻量级锁：**

轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

**自旋锁：**

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。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起。

**synchronized的执行过程：**   
1. 检测Mark Word里面是不是当前线程的ID，如果是，表示当前线程处于偏向锁   
2. 如果不是，则使用CAS将当前线程的ID替换Mard Word，如果成功则表示当前线程获得偏向锁，置偏向标志位1   
3. 如果失败，则说明发生竞争，撤销偏向锁，进而升级为轻量级锁。   
4. 当前线程使用CAS将对象头的Mark Word替换为锁记录指针，如果成功，当前线程获得锁   
5. 如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。   
6. 如果自旋成功则依然处于轻量级状态。   
7. 如果自旋失败，则升级为重量级锁。

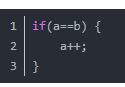
#### synchronized的三种应用方式

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 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。

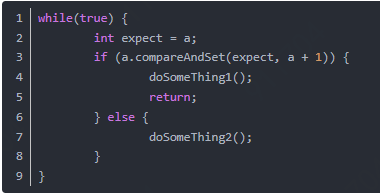
### CAS

CAS是单词compare and set的缩写，意思是指在set之前先比较该值有没有变化，只有在没变的情况下才对其赋值。其底层用了lock指令前缀来保证原子性。

正常有：



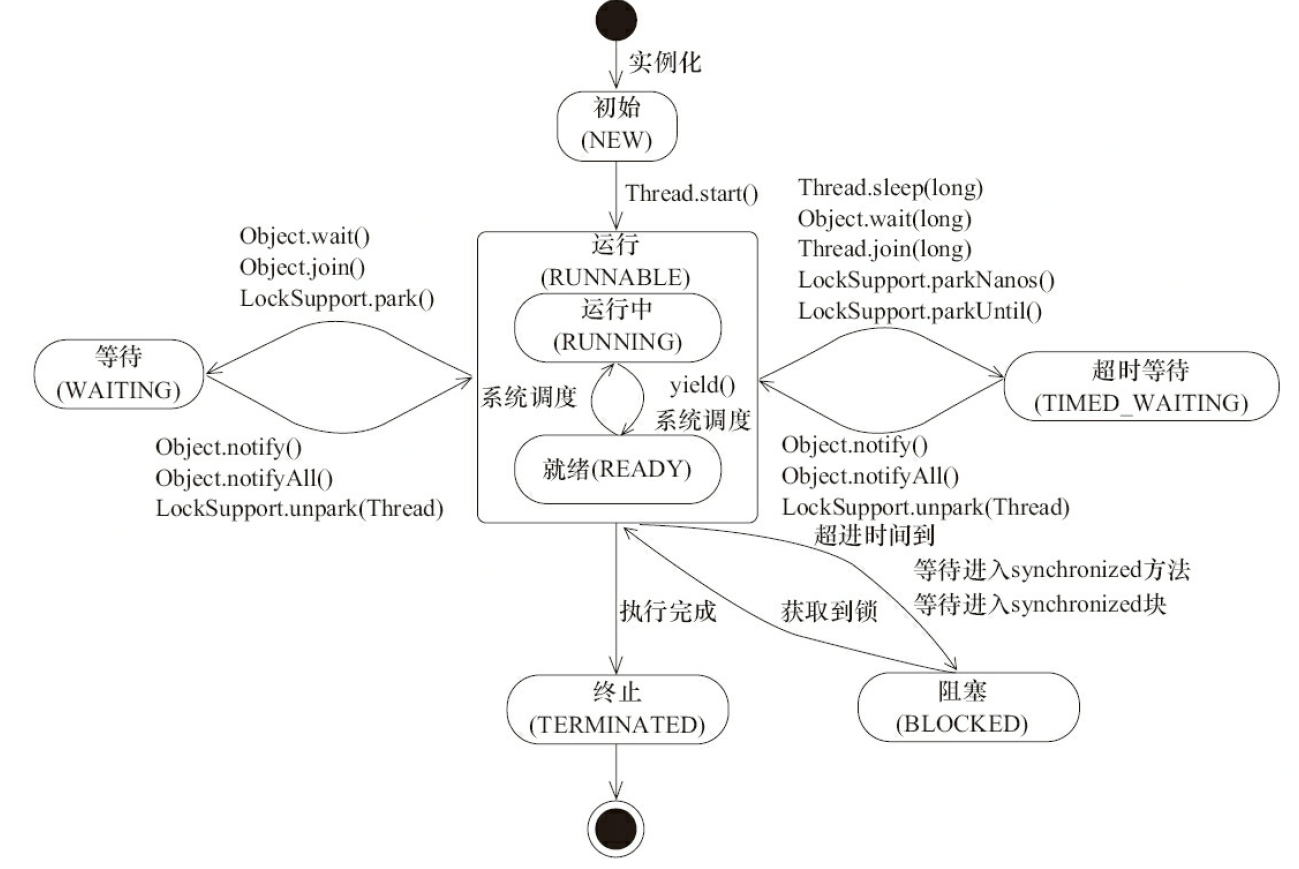
多线程下：



#### ABA问题

ABA问题。因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A→B→A就会变成1A→2B→3A。

## Java并发编程基础



### 线程生命周期

#### 构造线程

在运行线程之前首先要构造一个线程对象，线程对象在构造的时候需要提供线程所需要的属性，如线程所属的线程组、线程优先级、是否是Daemon线程等信息。

private void init(ThreadGroup g, Runnable target, String name,long stackSize, AccessControlContext acc)

{

if (name == null) throw new NullPointerException("name cannot be null")；

// 当前线程就是该线程的父线程

Thread parent = currentThread();

this.group = g;

// 将daemon、priority属性设置为父线程的对应属性

this.daemon = parent.isDaemon();

this.priority = parent.getPriority();

this.name = name.toCharArray();

this.target = target;

setPriority(priority);

// 将父线程的InheritableThreadLocal复制过来

if (parent.inheritableThreadLocals != null)

this.inheritableThreadLocals=ThreadLocal.createInheritedMap(parent. inheritableThreadLocals);

// 分配一个线程ID

tid = nextThreadID();

}

Daemon线程：Daemon线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着，当一个Java虚拟机中不存在非Daemon线程的时候，Java虚拟机将会退出。可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程。

Finally不执行的情况：在Java虚拟机退出时Daemon线程中的finally块并不一定会执行。

#### 启动线程

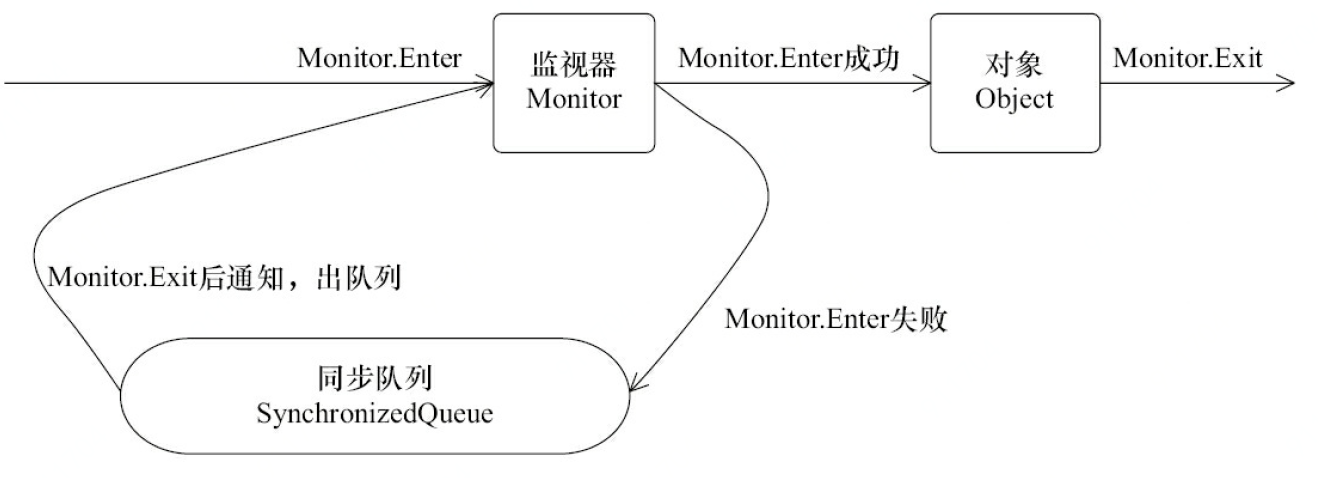
线程对象在初始化完成之后，调用start()方法就可以启动这个线程。

#### 中断

中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。其他线程通过调用该线程的interrupt()方法对其进行中断操作。

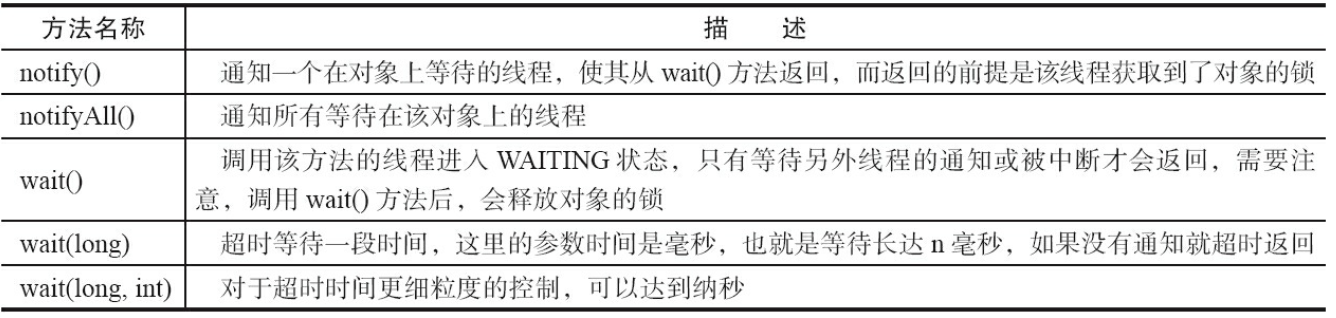
### 线程间通信

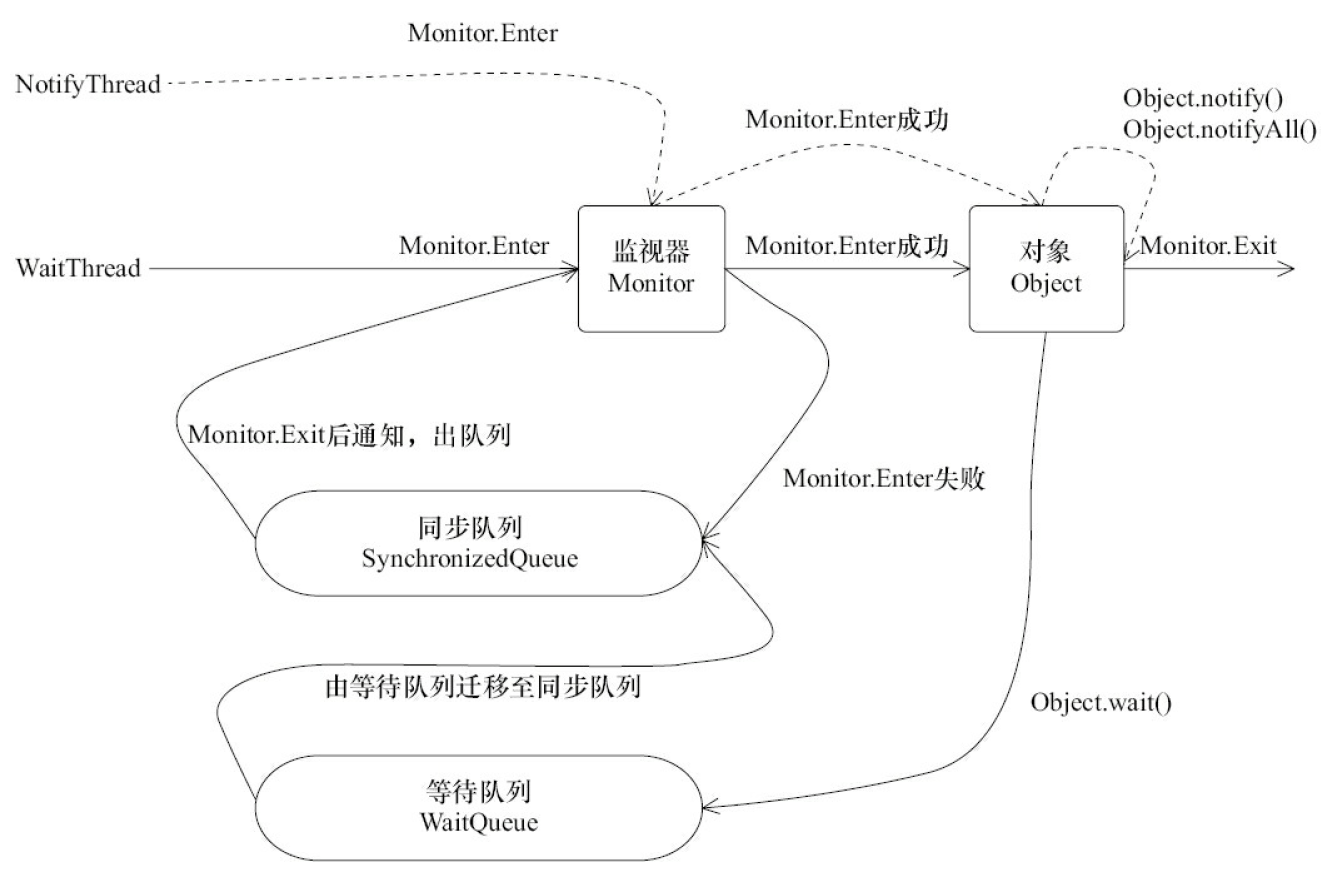
#### volatile和synchronized关键字



任意线程对Object（Object由synchronized保护）的访问，首先要获得Object的监视器。如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问Object的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

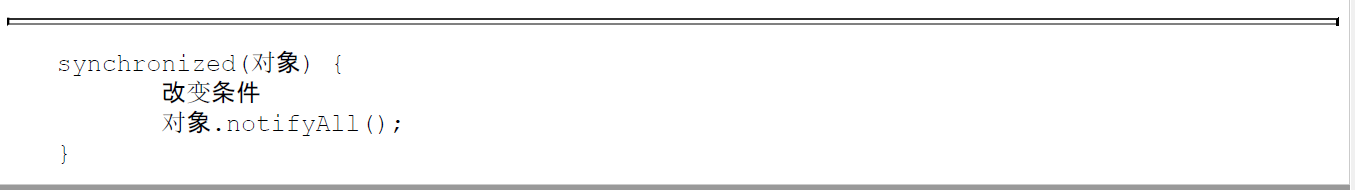
#### 等待/通知机制





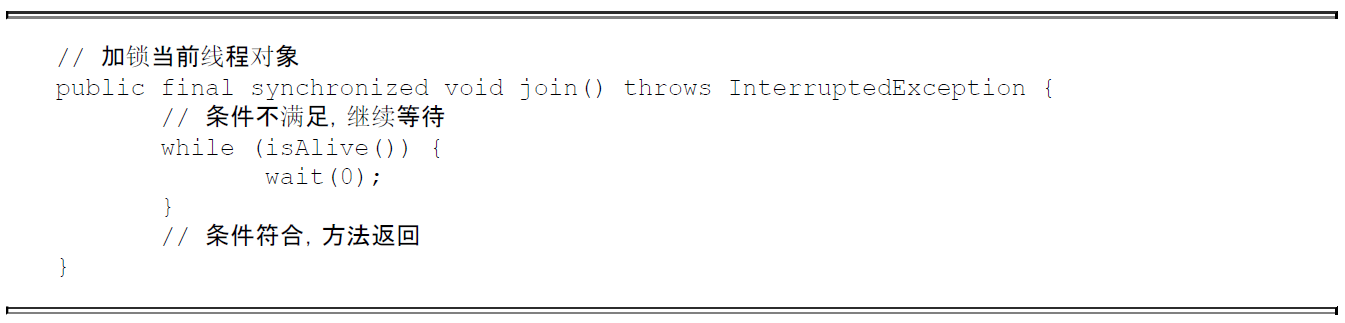
经典应用：





#### Thread.join()

如果一个线程A执行了thread.join()语句，其含义是：当前线程A等待thread线程终止之后才从thread.join()返回。



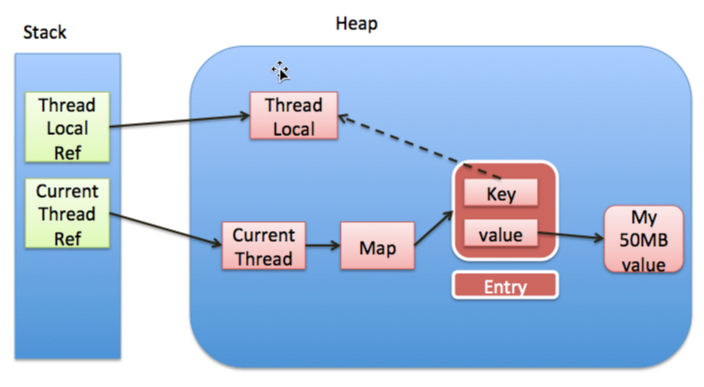
其中isAlive是判断调用join的线程是否存活，而wait是用来阻塞外部线程的。

#### ThreadLocal

ThreadLocal的作用是提供线程内的局部变量，不同的线程之间不会相互干扰，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或组件之间一些公共变量的传递的复杂度。

设计原理：

每个Thread维护一个ThreadLocalMap哈希表，这个哈希表的key是ThreadLocal实例本身，value才是真正要存储的值Object。也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。值得注意的是图中的虚线，表示 ThreadLocalMap 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。



**为什么会内存泄漏：**

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用。

Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏。

其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的value。

**使用弱引用的理由：**

key 使用强引用：引用的ThreadLocal的对象被回收了，但是ThreadLocalMap还持有ThreadLocal的强引用，如果没有手动删除，ThreadLocal不会被回收，导致Entry内存泄漏。

key 使用弱引用：引用的ThreadLocal的对象被回收了，由于ThreadLocalMap持有ThreadLocal的弱引用，即使没有手动删除，ThreadLocal也会被回收。value在下一次ThreadLocalMap调用set,get，remove的时候会被清除。

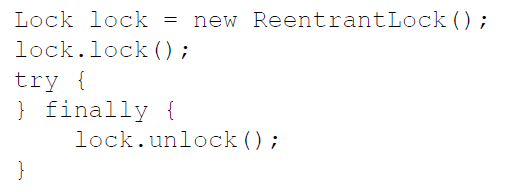
比较两种情况，我们可以发现：由于ThreadLocalMap的生命周期跟Thread一样长，如果都没有手动删除对应key，都会导致内存泄漏，但是使用弱引用可以多一层保障：**弱引用ThreadLocal不会内存泄漏，对应的value在下一次ThreadLocalMap调用set,get,remove的时候会被清除**。

因此，ThreadLocal内存泄漏的根源是：由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏，而不是因为弱引用。

ThreadLocal和synchronized的区别：

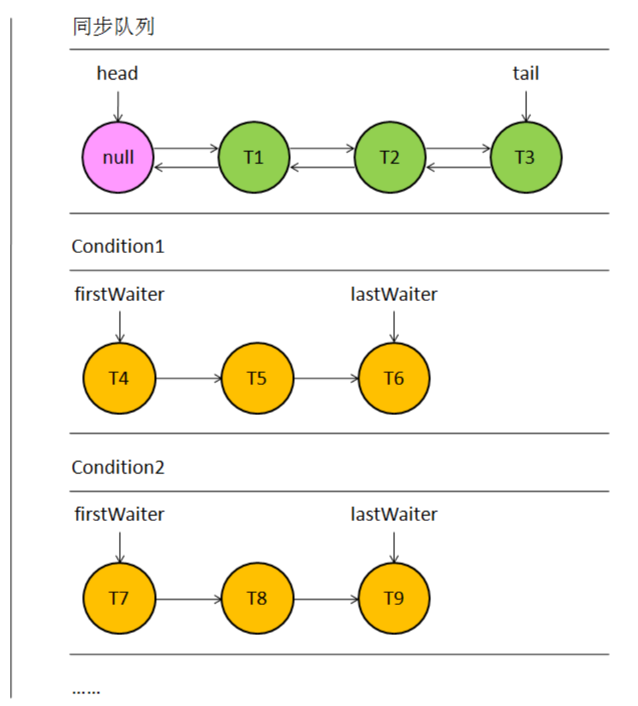
同步机制(synchronized关键字)采用了以“时间换空间”的方式，提供一份变量，让不同的线程排队访问。而ThreadLocal采用了“以空间换时间”的方式，为每一个线程都提供一份变量的副本，从而实现同时访问而互不影响。

## 锁（lock）



### 队列同步器（AQS）

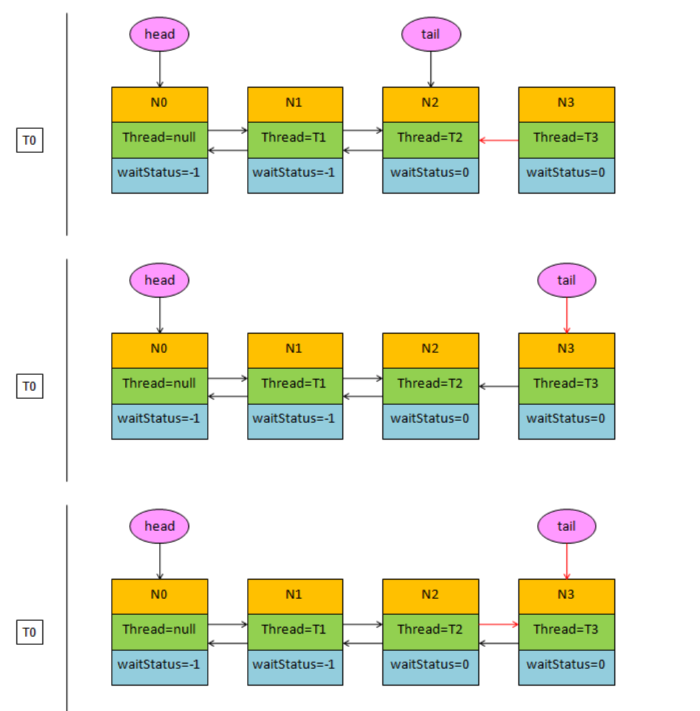
队列同步器AbstractQueuedSynchronizer（以下简称同步器），是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。



AQS内部其实有两个排队区，一个是同步队列，一个是条件队列。同步队列只有一条，而条件队列可以有多条。同步队列的结点分别持有前后结点的引用，而条件队列的结点只有一个指向后继结点的引用。T表示线程，每个结点包含一个线程，线程在获取锁失败后首先进入同步队列排队，而想要进入条件队列该线程必须持有锁才行。

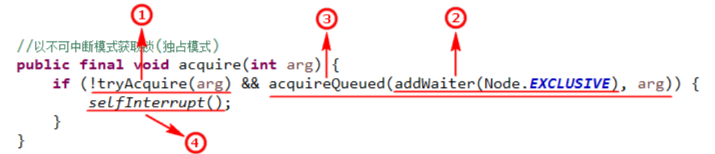


每个结点都有一个等待状态，这个等待状态分为CANCELLED，SIGNAL，CONDITION，PROPAGATE四种状态。可以将这个等待状态看作是挂在座位旁边的牌子，标识当前座位上的人的等待状态。这个牌子的状态不仅自己可以修改，其他人也可以修改。例如当这个线程在排队过程中已经打算放弃了，它就会将自己座位上的牌子设置为CANCELLED，这样其他人看到了就可以将它清理出队列。还有一种情况是，当线程在座位上要睡着之前，它怕自己睡过了头，就会将前面位置上的牌子改为SIGNAL，因为每个人在离开队列前都会回到自己座位上看一眼，如果看到牌子上状态为SIGNAL，它就会去唤醒下一个人。只有保证前面位置上的牌子为SIGNAL，当前线程才会安心的睡去。CONDITION状态表示该线程在条件队列中排队，PROPAGATE状态提醒后面来的线程可以直接获取锁，这个状态只在共享模式用到。



分为三步：指向尾结点，CAS更改尾结点，将旧尾结点的后继指向当前结点。在并发环境中这三步操作不一定能保证完成，所以在清空同步队列所有已取消的结点这一操作中，为了寻找非取消状态的结点，不是从前向后遍历而是从后向前遍历的。还有就是每个结点进入队列中时它的等待状态是为0，只有后继结点的线程需要挂起时才会将前面结点的等待状态改为SIGNAL。

#### 独占模式



**第一步：!tryAcquire(arg)**

这时候来了一个人，他首先尝试着去敲了敲门，如果发现门没锁(tryAcquire(arg)=true)，那就直接进去了。如果发现门锁了(tryAcquire(arg)=false)，就执行下一步。这个tryAcquire方法决定了什么时候锁是开着的，什么时候锁是关闭的。这个方法必须要让子类去覆盖，重写里面的判断逻辑。

**第二步：addWaiter(Node.EXCLUSIVE)**

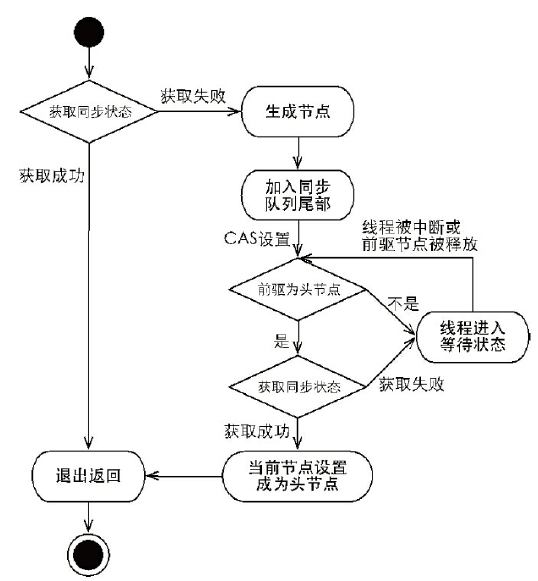
执行到这一步表明第一次获取锁失败，那么这个人就给自己领了块号码牌进入排队区去排队了，在领号码牌的时候会声明自己想要以什么样的方式来占用房间(独占模式or共享模式)。注意，这时候他并没有坐下来休息(将自己挂起)。

**第三步：acquireQueued(addWaiter(Node.EXCLUSIVE), arg)**

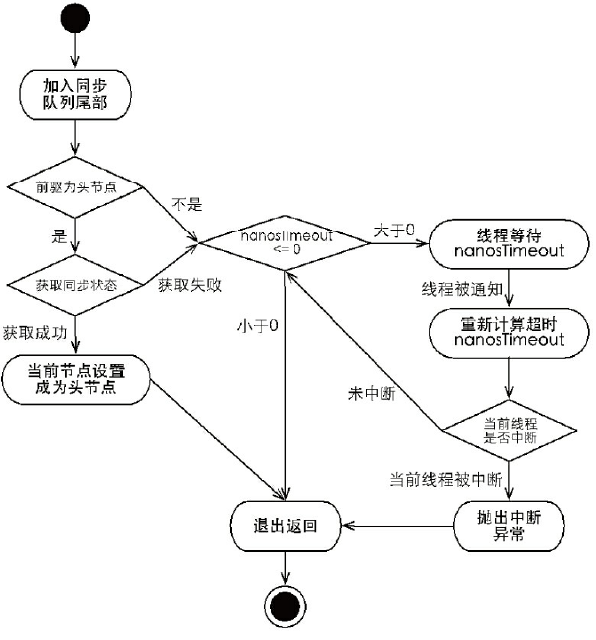
领完号码牌进入排队区后就会立马执行这个方法，当一个结点首次进入排队区后有两种情况，一种是发现他前面的那个人已经离开座位进入房间了，那他就不坐下来休息了，会再次去敲一敲门看看那小子有没有完事。如果里面的人刚好完事出来了，都不用他叫自己就直接冲进去了。否则，就要考虑坐下来休息一会儿了，但是他还是不放心，如果他坐下来睡着后没人提醒他怎么办？他就在前面那人的座位上留一个小纸条，好让从里面出来的人看到纸条后能够唤醒他。还有一种情况是，当他进入排队区后发现前面还有好几个人在座位上排队呢，那他就可以安心的坐下来咪一会儿了，但在此之前他还是会在前面那人(此时已经睡着了)的座位上留一个纸条，好让这个人在走之前能够去唤醒自己。当一切事情办妥了之后，他就安安心心的睡觉了。

**第四步：selfInterrupt()**

没有成功获取到锁之前不响应任何形式的线程中断，只有当线程成功获取到锁并从for循环出来后，他才会查看在这期间是否有人要求中断线程，如果是的话再去调用selfInterrupt()方法将自己挂起。



**超时计算与中断**



**线程释放锁并离开同步队列：**

线程持有锁进入房间后就会去办自己的事情，等事情办完后它就会释放锁并离开房间。通过tryRelease方法可以拨动密码锁进行解锁， tryRelease方法是需要让子类去覆盖的，不同的子类实现的规则不一样，也就是说不同的子类设置的密码不一样。像在ReentrantLock当中，房间里面的人每调用tryRelease方法一次，state就减1，直到state减到0的时候密码锁就开了。CountDownLatch和这个也有点类似，只不过它不是一个人在转，而是多个人每人都去转一下，集中大家的力量把锁给开了。线程出了房间后它会找到自己原先的座位，也就是找到head结点。看看座位上有没有人给它留了小纸条，如果有的话它就知道有人睡着了需要让它帮忙唤醒，那么它就会去唤醒那个线程。如果没有的话就表明同步队列中暂时还没有人在等待，也没有人需要它唤醒，所以它就可以安心的离去了。

#### 共享模式

**第一步：**

首先调用tryAcquireShared去尝试获取锁，tryAcquireShared方法返回一个获取锁的状态，这里AQS规定了返回状态若是负数代表当前结点获取锁失败，若是0代表当前结点获取锁成功，但后继结点不能再获取了，若是正数则代表当前结点获取锁成功，并且这个锁后续结点也同样可以获取成功。子类在实现tryAcquireShared方法获取锁的逻辑时，返回值需要遵守这个约定。如果调用tryAcquireShared的返回值小于0，就代表这次尝试获取锁失败了，接下来就调用doAcquireShared方法将当前线程添加进同步队列。

**第二步：**

进入doAcquireShared方法首先是将当前线程包装成结点放到同步队列尾部。结点进入同步队列后，如果它发现在它前面的结点就是head结点，因为head结点的线程已经获取锁进入房间里面了，那么下一个获取锁的结点就轮到自己了，所以当前结点先不会将自己挂起，而是再一次去尝试获取锁，如果前面那人刚好释放锁离开了，那么当前结点就能成功获得锁，如果前面那人还没有释放锁，那么就会将head结点的状态改为SIGNAL，只有保证前面结点的状态为SIGNAL，当前结点才能放心的将自己挂起。如果当前结点恰巧成功的获取了锁，那么接下来就会将自己设置为head结点，并且唤醒后面同样是共享模式的结点。

**第三步：**

首先将自己设置成head结点，然后再根据传入的tryAcquireShared方法的返回值来决定是否要去唤醒后继结点。前面已经讲到当返回值大于0就表明当前结点成功获取了锁，并且后面的结点也可以成功获取锁。这时当前结点就需要去唤醒后面同样是共享模式的结点，注意，每次唤醒仅仅只是唤醒后一个结点，如果后一个结点不是共享模式的话，当前结点就直接进入房间而不会再去唤醒更后面的结点了。共享模式下唤醒后继结点的操作是在doReleaseShared方法进行的，共享模式和独占模式的唤醒操作基本也是相同的，都是去找到自己座位上的牌子(等待状态)，如果牌子上为SIGNAL表明后面有人需要让它帮忙唤醒，如果牌子上为0则表明队列此时并没有人在排队。在独占模式下是如果发现没人在排队就直接离开队列了，而在共享模式下如果发现队列后面没人在排队，当前结点在离开前仍然会留个小纸条(将等待状态设置为PROPAGATE)告诉后来的人这个锁的可获取状态。那么后面来的人在尝试获取锁的时候可以根据这个状态来判断是否直接获取锁。

#### 条件队列

我们还是拿公共厕所做比喻，同步队列是主要的排队区，如果公共厕所没开放，所有想要进入厕所的人都得在这里排队。而条件队列主要是为条件等待设置的，我们想象一下如果一个人通过排队终于成功获取锁进入了厕所，但在方便之前发现自己没带手纸，碰到这种情况虽然很无奈，但是它也必须接受这个事实，这时它只好乖乖的出去先准备好手纸(进入条件队列等待)，当然在出去之前还得把锁给释放了好让其他人能够进来，在准备好了手纸(条件满足)之后它又得重新回到同步队列中去排队。当然进入房间的人并不都是因为没带手纸，可能还有其他一些原因必须中断操作先去条件队列中去排队，所以条件队列可以有多个，依不同的等待条件而设置不同的条件队列。条件队列是一条单向链表，Condition接口定义了条件队列中的所有操作。

**Await**

**第一步：**

当线程调用await方法的时候，首先会将当前线程包装成node结点放入条件队列尾部。注意，这个过程只是将结点添加到同步队列尾部而没有挂起线程。

**第二步：**

将当前线程包装成结点添加到条件队列尾部后，紧接着就调用fullyRelease方法释放锁。注意，方法名为fullyRelease也就这步操作会完全的释放锁，因为锁是可重入的，所以在进行条件等待前需要将锁全部释放了，不然的话别人就获取不了锁了。如果释放锁失败的话就会抛出一个运行时异常，如果成功释放了锁的话就返回之前的同步状态。

**第三步：**

在以上两个操作完成了之后就会进入循环，首先调用LockSupport.park(this)将线程挂起，所以线程就会一直在这里阻塞。在调用signal方法后仅仅只是将结点从条件队列转移到同步队列中去，至于会不会唤醒线程需要看情况。如果转移结点时发现同步队列中的前继结点已取消，或者是更新前继结点的状态为SIGNAL失败，这两种情况都会立即唤醒线程，否则的话在signal方法结束时就不会去唤醒已在同步队列中的线程，而是等到它的前继结点来唤醒。当然，线程阻塞在这里除了可以调用signal方法唤醒之外，线程还可以响应中断，如果线程在这里收到中断请求就会继续往下执行。可以看到线程醒来后会马上检查是否是由于中断唤醒的还是通过signal方法唤醒的，如果是因为中断唤醒的同样会将这个结点转移到同步队列中去。

**第四步：**

当线程终止了while循环也就是条件等待后，就会回到同步队列中。不管是因为调用signal方法回去的还是因为线程中断导致的，结点最终都会在同步队列中。这时就会调用acquireQueued方法执行在同步队列中获取锁的操作。

**Signal**

首先会用CAS操作将结点的状态从CONDITION设置为0，然后再将该结点添加到同步队列尾部。接下来的if判断语句，这个判断语句主要是用来判断什么时候会去唤醒线程，出现这两种情况就会立即唤醒线程，一种是当发现前继结点的状态是取消状态时，还有一种是更新前继结点的状态失败时。这两种情况都会马上去唤醒线程，否则的话就仅仅只是将结点从条件队列中转移到同步队列中就完了，而不会立马去唤醒结点中的线程。

### 重入锁



获取锁和释放锁的操作分别委托给Sync对象的lock方法和release方法。每个ReentrantLock对象都持有一个Sync类型的引用，这个Sync类是一个抽象内部类它继承自AbstractQueuedSynchronizer，它里面的lock方法是一个抽象方法。ReentrantLock的成员变量sync是在构造时赋值的,有公平和非公平两种锁。公平锁和非公平锁重写了lock方法。

如果是公平锁，线程将按照它们发出请求的顺序来获得锁，但在非公平锁上，则允许插队行为：当一个线程请求非公平的锁时，如果在发出请求的同时该锁的状态变为可用，那么这个线程将跳过队列中所有等待的线程直接获得这个锁。

#### 非公平锁

非公平锁的lock方法中，线程第一步就会以CAS方式将同步状态的值从0改为1。其实这步操作就等于去尝试获取锁，如果更改成功则表明线程刚来就获取了锁，而不必再去同步队列里面排队了。如果更改失败则表明线程刚来时锁还未被释放，所以接下来就调用acquire方法。我们知道这个acquire方法是继承自AbstractQueuedSynchronizer的方法。线程进入acquire方法后首先去调用tryAcquire方法尝试去获取锁。该锁的重写实现为：

首先去获取同步状态，如果同步状态为0就使用CAS操作更改同步状态。如果同步状态不为0表明锁被占用，此时会先去判断持有锁的线程是否是当前线程，如果是的话就将同步状态加1，否则的话这次尝试获取锁的操作宣告失败。于是会调用addWaiter方法将线程添加到同步队列。

#### 公平锁

调用公平锁的lock方法时会直接调用acquire方法。在该方法中也是首先获取同步状态的值，如果同步状态为0则表明此时锁刚好被释放，这时和非公平锁不同的是它会先去调用hasQueuedPredecessors方法查询同步队列中是否有人在排队，如果没人在排队才会去修改同步状态的值，可以看到公平锁在这里采取礼让的方式而不是自己马上去获取锁。

**重入锁的释放：**

首先会获取当前同步状态，并将当前同步状态减去传入的参数值得到新的同步状态，然后判断新的同步状态是否等于0，如果等于0则表明当前锁被释放，然后先将锁的释放状态置为真，再将当前占有锁的线程清空，最后调用setState方法设置新的同步状态并返回锁的释放状态。

## 并发工具类

### Semaphore

Semaphore(信号量)是JUC包中比较常用到的一个类，它是AQS共享模式的一个应用，可以允许多个线程同时对共享资源进行操作，并且可以有效的控制并发数，利用它可以很好的实现流量控制。Semaphore提供了一个许可证的概念，可以把这个许可证看作公共汽车车票，只有成功获取车票的人才能够上车，并且车票是有一定数量的，不可能毫无限制的发下去，这样就会导致公交车超载。所以当车票发完的时候(公交车以满载)，其他人就只能等下一趟车了。如果中途有人下车，那么他的位置将会空闲出来，因此如果这时其他人想要上车的话就又可以获得车票了

**Acquire**

Semaphore使用非公平锁tryAcquireShared，非公平获取锁的逻辑是先取出当前同步状态(同步状态表示许可证个数)，将当前同步状态减去参入的参数，如果结果不小于0的话证明还有可用的许可证，那么就直接使用CAS操作更新同步状态的值，最后不管结果是否小于0都会返回该结果值。这里我们要了解tryAcquireShared方法返回值的含义，返回负数表示获取失败，零表示当前线程获取成功但后续线程不能再获取，正数表示当前线程获取成功并且后续线程也能够获取。

当小于0时，会将当前线程包装成结点放入同步队列尾部，并且有可能挂起线程。

**release**

调用了AQS的releaseShared方法，首先获取同步状态，将同步状态加上传入的参数，然后以CAS方式更新同步状态，更新成功就返回true并跳出方法，否则就继续循环直到成功为止。

### CountDownLatch

CountDownLatch(闭锁)是一个很有用的工具类，它是AQS共享模式下的一个应用，利用它我们可以拦截一个或多个线程使其在某个条件成熟后再执行。它的内部提供了一个计数器，在构造闭锁时必须指定计数器的初始值，且计数器的初始值必须大于0。另外它还提供了一个countDown方法来操作计数器的值，每调用一次countDown方法计数器都会减1，直到计数器的值减为0时就代表条件已成熟，所有因调用await方法而阻塞的线程都会被唤醒。

**Await**

当线程调用await方法时其实是调用到了AQS的acquireSharedInterruptibly方法，该方法是以响应线程中断的方式来获取锁的，判断当前同步状态是否为0，如果为0则返回1表明可以获取锁，否则返回-1表示不能获取锁。如果tryAcquireShared方法返回1则线程能够不必等待而继续执行，如果返回-1那么后续就会让线程进入到同步队列里面等待。

**Countdown**

先是调用tryReleaseShared方法尝试释放锁，tryReleaseShared方法在AQS里面是一个抽象方法，它的具体实现逻辑在子类Sync类里面。tryReleaseShared方法如果返回true表示释放成功，返回false表示释放失败，只有当将同步状态减1后该同步状态恰好为0时才会返回true，其他情况都是返回false。那么当tryReleaseShared返回true之后就会马上调用doReleaseShared方法去唤醒同步队列的所有线程。

### CyclicBarrier

在CyclicBarrier类的内部有一个计数器，每个线程在到达屏障点的时候都会调用await方法将自己阻塞，此时计数器会减1，当计数器减为0的时候所有因调用await方法而被阻塞的线程将被唤醒。这就是实现一组线程相互等待的原理。



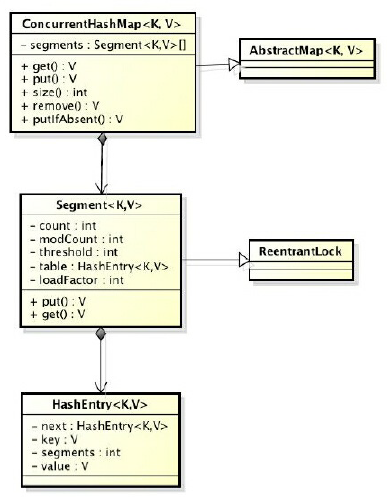
上面贴出了CyclicBarrier所有的成员变量，可以看到CyclicBarrier内部是通过条件队列trip来对线程进行阻塞的，并且其内部维护了两个int型的变量parties和count，parties表示每次拦截的线程数，该值在构造时进行赋值。count是内部计数器，它的初始值和parties相同，以后随着每次await方法的调用而减1，直到减为0就将所有线程唤醒。CyclicBarrier有一个静态内部类Generation，该类的对象代表栅栏的当前代，就像玩游戏时代表的本局游戏，利用它可以实现循环等待。barrierCommand表示换代前执行的任务，当count减为0时表示本局游戏结束，需要转到下一局。在转到下一局游戏之前会将所有阻塞的线程唤醒，在唤醒所有线程之前你可以通过指定barrierCommand来执行自己的任务。

**Await**

await方法中每次都将count减1，减完后立马进行判断看看是否等于0，如果等于0的话就会先去执行之前指定好的任务，执行完之后再调用nextGeneration方法将栅栏转到下一代，在该方法中会将所有线程唤醒，将计数器的值重新设为parties，最后会重新设置栅栏代次，在执行完nextGeneration方法之后就意味着游戏进入下一局。如果计数器此时还不等于0的话就进入for循环，根据参数来决定是调用trip.awaitNanos(nanos)还是trip.await()方法，这两方法对应着定时和非定时等待。如果在等待过程中当前线程被中断就会执行breakBarrier方法，该方法叫做打破栅栏，意味着游戏在中途被掐断，设置generation的broken状态为true并唤醒所有线程。同时这也说明在等待过程中有一个线程被中断整盘游戏就结束，所有之前被阻塞的线程都会被唤醒。线程醒来后会执行下面三个判断，看看是否因为调用breakBarrier方法而被唤醒，如果是则抛出异常；看看是否是正常的换代操作而被唤醒，如果是则返回计数器的值；看看是否因为超时而被唤醒，如果是的话就调用breakBarrier打破栅栏并抛出异常。这里还需要注意的是，如果其中有一个线程因为等待超时而退出，那么整盘游戏也会结束，其他线程都会被唤醒。

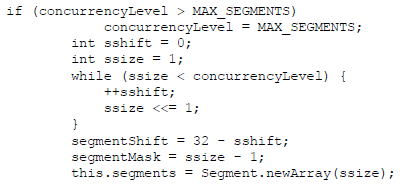
## 并发容器和框架

### ConcurrentHashMap



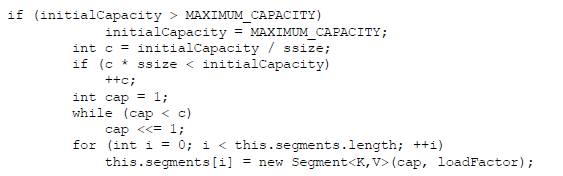
ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组。Segment的结构和HashMap类似，是一种数组和链表结构。一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁。

#### ConcurrentHashMap的初始化



segments数组的长度ssize是通过concurrencyLevel计算得出的。为了能通过按位与的散列算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14、15或16，ssize都会等于16，即容器里锁的个数也是16。

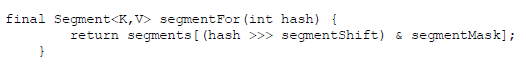
sshift等于ssize从1向左移位的次数，在默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。 segmentShift用于定位参与散列运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的。



cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以cap不是1，就是2的N次方。 segment的容量threshold＝（int）cap\*loadFactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过运算cap等于1，threshold等于零。

#### 定位Segment

首先进行再散列定位到segment。之所以进行再散列，目的是减少散列冲突，使元素能够均匀地分布在不同的Segment上，从而提高容器的存取效率。



默认情况下segmentShift为28，segmentMask为15，再散列后的数最大是32位二进制数据，向右无符号移动28位，意思是让高4位参与到散列运算中，（hash>>>segmentShift）。

最后通过哈希码计算元素在数组中的下标：(tab.length - 1) & h。

#### ConcurrentHashMap的操作

**Get**

get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空才会加锁重读。原因是它的get方法里将要使用的共享变量都定义成volatile类型，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写。在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是因为根据Java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值。

**Put**

由于put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须加锁。

put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置，然后将其放在HashEntry数组里。

**size**

如果要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是可能累加前使用的count发生了变化，那么统计结果就不准了。所以，最安全的做法是在统计size的时候把所有Segment的put、remove和clean方法全部锁住，但是这种做法显然非常低效。

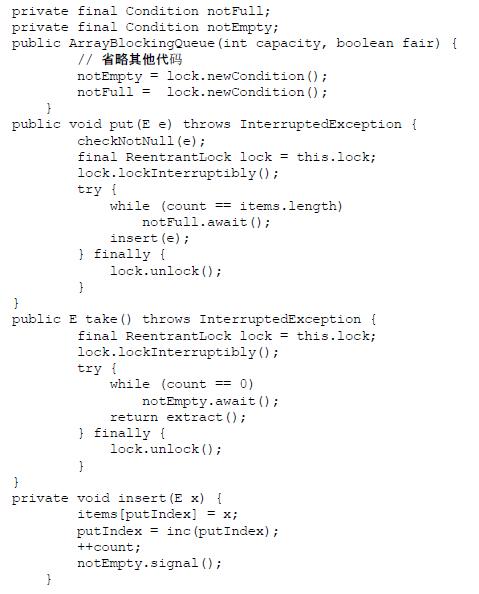
因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

**那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put、remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。**

### 阻塞队列

#### ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。使用通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。通过查看JDK源码发现ArrayBlockingQueue使用了Condition来实现。



#### LinkedBlockingQueue

LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序。

#### SynchronousQueue

SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。

### Fork/Join框架

Fork/Join框架是Java 7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

#### 工作窃取算法

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。假如我们需要做一个比较大的任务，可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应。比如A线程负责处理A队列里的任务。但是，有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

#### Fork/Join框架的设计

步骤1　分割任务。首先我们需要有一个fork类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停地分割，直到分割出的子任务足够小。

步骤2　执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

## 线程池

合理地使用线程池能够带来3个好处：

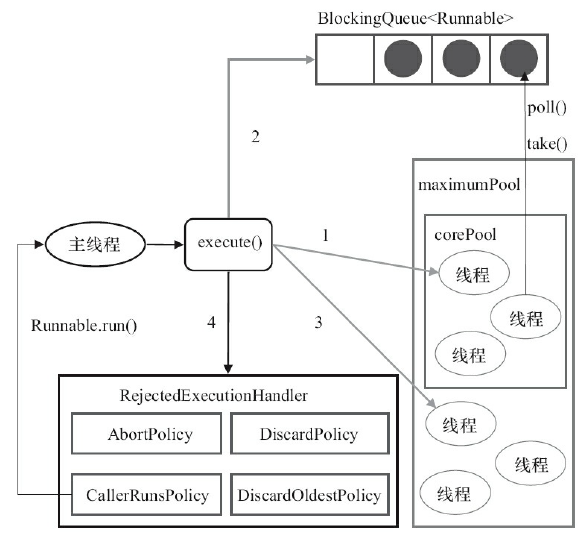
第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，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分配、调优和监控。但是，要做到合理利用线程池，必须对其实现原理了如指掌。

### 线程池实现原理

1. 线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。
2. 线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
3. 线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。



1. 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。
2. 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。
3. 如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。
4. 如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

### 线程池使用

创建一个线程池时需要输入几个参数：

1）corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任

务数大于线程池基本大小时就不再创建。

2）runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。

3）maximumPoolSize（线程池最大数量）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。

4）RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。

· AbortPolicy：直接抛出异常。

· CallerRunsPolicy：只用调用者所在线程来运行任务。

· DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

· DiscardPolicy：不处理，丢弃掉。

1. keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。

可以使用两个方法向线程池提交任务，分别为execute()和submit()方法。

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。

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。

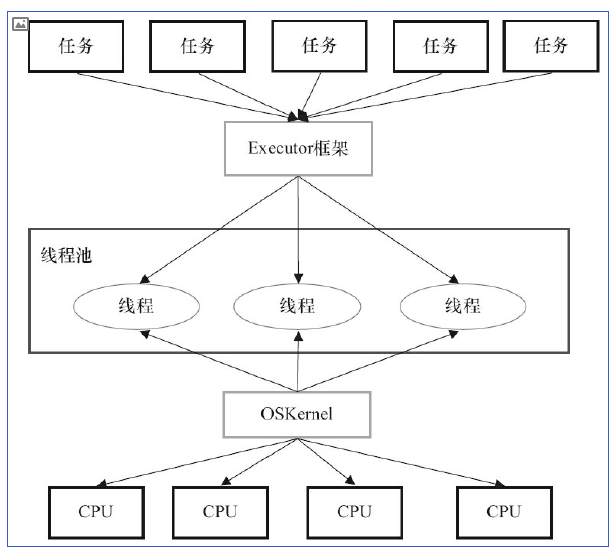
## Executor框架

### Executor框架的两级调度模型

在HotSpot VM的线程模型中，Java线程（java.lang.Thread）被一对一映射为本地操作系统线程。Java线程启动时会创建一个本地操作系统线程；当该Java线程终止时，这个操作系统线程也会被回收。操作系统会调度所有线程并将它们分配给可用的CPU。

在上层，Java多线程程序通常把应用分解为若干个任务，然后使用用户级的调度器

（Executor框架）将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。



Executor框架主要由3大部分组成如下。·

1. 任务。包括被执行任务需要实现的接口：Runnable接口或Callable接口。
2. 任务的执行。包括任务执行机制的核心接口Executor，以及继承自Executor的ExecutorService接口。Executor框架有两个关键类实现了ExecutorService接口。
3. 异步计算的结果。包括接口Future和实现Future接口的FutureTask类。



主线程首先要创建实现Runnable或者Callable接口的任务对象。然后可以把Runnable对象直接交给ExecutorService执行。如果执行ExecutorService.submit（…），ExecutorService将返回一个实现Future接口的对象（到目前为止的JDK中，返回的是FutureTask对象）。最后，主线程可以执行FutureTask.get()方法来等待任务执行完成。主线程也可以执行FutureTask.cancel（boolean mayInterruptIfRunning）来取消此任务的执行。

Callable与Runnable的区别：

(1)：Callcble是可以有返回值的，具体的返回值就是在Callable的接口方法call返回的，并且这个返回值具体是通过实现Future接口的对象的get方法获取的，这个方法是会造成线程阻塞的；而Runnable是没有返回值的，因为Runnable接口中的run方法是没有返回值的；

        (2)：Callable里面的call方法是可以抛出异常的，我们可以捕获异常进行处理；但是Runnable里面的run方法是不可以抛出异常的，异常要在run方法内部必须得到处理，不能向外界抛出

### ThreadPoolExecutor详解

Executor框架最核心的类是ThreadPoolExecutor，它是线程池的实现类. · 通过Executor框架的工具类Executors，可以创建3种类型的ThreadPoolExecutor。

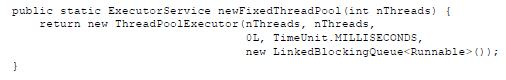
· FixedThreadPool。

· SingleThreadExecutor。

· CachedThreadPool。

#### FixedThreadPool详解

FixedThreadPool被称为可重用固定线程数的线程池。



FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。



1） 如果当前运行的线程数少于corePoolSize，则创建新线程来执行任务。

2） 在线程池完成预热之后（当前运行的线程数等于corePoolSize），将任务加入LinkedBlockingQueue。

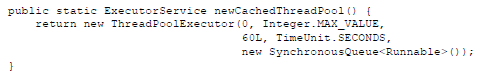
3）线程执行完1中的任务后，会在循环中反复从LinkedBlockingQueue获取任务来执行。

#### SingleThreadExecutor详解

SingleThreadExecutor是使用单个worker线程的Executor。

SingleThreadExecutor的corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。

#### CachedThreadPool详解



CachedThreadPool的corePoolSize被设置为0，即corePool为空；maximumPoolSize被设置为Integer.MAX\_VALUE，即maximumPool是无界的。这里把keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。

FixedThreadPool和SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列。CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下， CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。



1）首先执行SynchronousQueue.offer（Runnable task）。如果当前maximumPool中有空闲线程正在执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS），那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成；否则执行下面的步骤2）。

2）当初始maximumPool为空，或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这种情况下，步骤1）将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

3）在步骤2）中新创建的线程将任务执行完后，会执行

SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务（主线程执行步骤1）），那么这个空闲线程将执行主线程提交的新任务；否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用任何资源。

### ScheduledThreadPoolExecutor详解

它主要用来在给定的延迟之后运行任务，或者定期执行任务。

ScheduledThreadPoolExecutor会把待调度的任务（ScheduledFutureTask）放到一个DelayQueue中。ScheduledFutureTask主要包含3个成员变量，·

long型成员变量time，表示这个任务将要被执行的具体时间。

long型成员变量sequenceNumber，表示这个任务被添加到ScheduledThreadPoolExecutor中的序号。

long型成员变量period，表示任务执行的间隔周期。

DelayQueue封装了一个PriorityQueue，这个PriorityQueue会对队列中的Scheduled-FutureTask进行排序。排序时，time小的排在前面（时间早的任务将被先执行）。如果两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面（也就是说，如果两个任务的执行时间相同，那么先提交的任务将被先执行）。



1. 线程1从DelayQueue中获取已到期的ScheduledFutureTask（DelayQueue.take()）。到期任务是指ScheduledFutureTask的time大于等于当前时间。
2. 线程1执行这个ScheduledFutureTask。
3. 线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。
4. 线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中（Delay- Queue.add()）。

### FutureTask详解

Future接口和实现Future接口的FutureTask类，代表**异步计算**的结果。