# 多线程

## 线程安全

线程安全，当多个线程对象访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其它的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象就是线程安全的

### 1.1如何实现线程安全

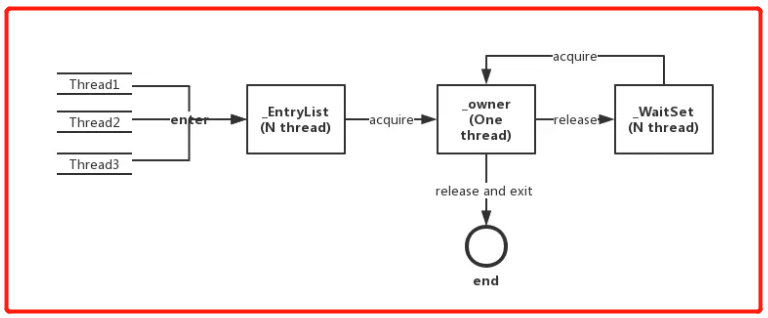
* 互斥同步是最常见的一种线程安全保障手段，同步是指在多个线程并发访问共享数据时，保证共享数据在同一时刻只被同一条线程使用。而互斥是实现同步的一种手段，临界区，互斥量和信号量都是主要的互斥实现方式。

在 Java 中，最基本的互斥同步手段就是 synchronized 关键字，synchronized 有两种使用形式，分别为同步方法和同步代码块：

那么对象是如何与监视器关联的呢?在java中，对象包含三块：对象头，实例数据，填充数据。其中对象头就包含了Mark Word,Mark Word一般存储对象的hashCode，GC分代年龄以及锁信息。锁信息就是包含了只想互斥量（重量级锁）的指针，指向一个监视器；监视器是通过ObjectMonirot实现的

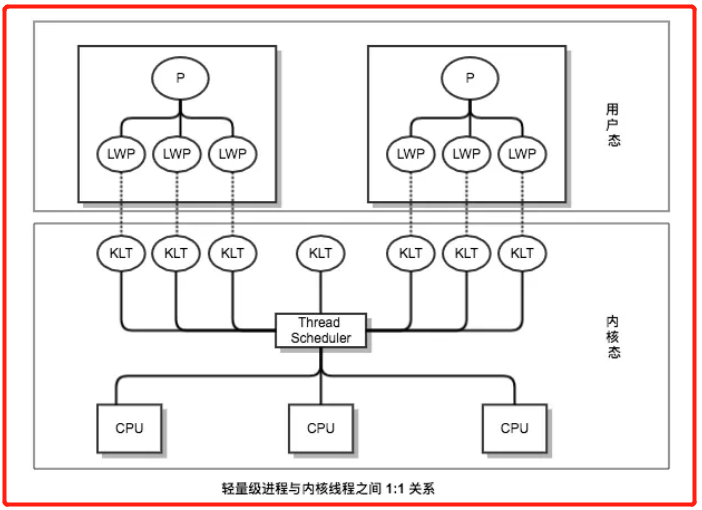


从上面代码可以看到有 ObjectMonitor 两个队列,分别是 \_WaitSet 和 \_EntryList，，\_owner 指向持有 ObjectMonitor 对象的线程，当多个线程获取到对象 monitor 后进入 \_owner 区域，并把 \_owner 设置为指向当前线程，并把 \_count 数量加1；当调用 wait() 方法后，将释放当前持有的 monitor，\_owner 置为空，\_count 减 1 操作，同时，将该线程进入 \_WaitSet 集合中等待唤醒，总结如下图：



### 上下文切换

现代操作系统中，运行一个程序，系统会为它创建一个进程。一个进程中可以有多个线程，线程是最小的调度单元，也叫轻量级进程（Light Weight Process）,这些线程都有各自的计数器、堆栈和局部变量等属性，并且能否访问共享的内存变量



由于有内核线程的支持，每个轻量级进程都成为一个独立的调度单元，及时有一个轻量级的进程阻塞了，也不会影响整个进程的工作，但是轻量级进程有它的局限，就是每个轻量级进程都需要一个内核线程的支持，因此轻量级进程需要消耗内核资源，因此一个系统支持轻量级进程的数量是有限的，其次，系统调用的代价相对较高，需要在用户态（user Mode）和内核态（Kernal Mode）中切换，线程上下文切换直接损耗CPU寄存器保存和加载, 系统调度器的代码需要执行, TLB实例需要重新加载, CPU 的pipeline需要刷掉。对于抢占式操作系统来说：

* 当前执行任务的时间片用完之后，系统CPU正常调度下一个任务
* 当前执行任务碰到IO阻塞，调度器将此任务挂起，继续下一任务
* 多个任务抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续下一任务
* 用户代码挂起当前任务，让出CPU时间
* 硬件中断

综上所述，在 JDK 1.5 之前，通过 synchronized 关键字是保证线程安全的一种重要手段，但是 synchronized 是一个重量级锁，为什么说 synchronized 是一个重量级锁呢？因为Synchronized依赖于系统的MutexLock(互斥锁)来实现的，且等待获取锁的线程会阻塞，被阻塞的线程不会消耗CPU资源，但是阻塞和唤醒一个线程都需要涉及到上下文切换，涉及到用户态和内核态切换，所以比较耗时。

除了Synchronized还有其他方法吗？当然有，java.util.concurrent包ReentrantLock

### Synchronized 关键字

synchronized有两种使用形式，同步方法和同步代码块



* 当作用于方法的时候：通过反编译得知:是通过ACC\_SYNCHRONIZED 标记符来实现同步。当某个线程要访问该方法时：首先会检查时候有ACC\_SYNCHRONIZED,如果有设置，则需要先获得监视器锁，然后开始执行方法，方法执行完后才释放监视器锁，其它线程又来执行该方法时，会因没有获得监视器的锁，而被阻塞住，当方法执行出了异常要往外抛的时候，在抛给外部方法前，就会释放锁
* 当作用于代码块的时候，JVM采用monitorenter、monitorexit两个指令来实现同步。采用计数器的方式。未被锁定的对象计数为0，当有一个线程获得该对象的锁的时候（执行monitorenter命令），计数器加1，可以重复获得锁，计数器累加1，当线程释放锁的时候（执行monitorexit命令）计数器减1，当计数器为0时，其它线程就可以获得该对象的锁

## 2.同步器框架

在 JDK1.5 的 java.util.concurrent 包中，大部分并发类都是基于AbstractQueuedSynchronized（简称AQS）这个简单的同步器框架构建的。这个框架为原子性管理同步状态、阻塞和唤醒线程、排队等提供了一种通用的机制。

提供了一个框架，用于实现依赖先进先出（FIFO）等待队列的阻塞锁和相关同步器(semaphores、events 等），这些同步器依赖于单个原子 int 值来表示同步状态，提供了原子方法更新状态 getState()、 setState(int)、compareAndSetState(int, int)。

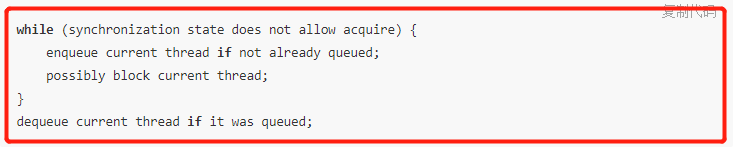
AQS是一个抽象类，并没有对并发类提供了一个统一的接口定义，而是由子类自身的情况实现响应的方法，AQS一般包含两个方法acquire(int)和release(int)表示获取同步状态和释放同步状态，AQS根据起状态是否独占分为**独占模式**和**共享模式**。

* 独占模式：同一时刻最多只有一个线程获取同步状态，处于该模式下，其他线程试图获取该锁将无法获取成功。
* 共享模式：同一时刻会有多个线程获取共享同步状态，处于该模式下，其他线程试图获取该锁可能会获取成功。

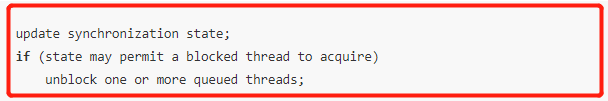
这个类还定义了 AbstractQueuedSynchronizer.ConditionObject，实现了 Condition 接口，用于支持管程形式的 await、signal 操作，这些操作与独占模式的 Lock 类有关，且 Condition 的实现天生就和与其关联的Lock 类紧密相关。

### 设计与实现

同步器的背后的基本思想非常简单，acquire 操作如下：



release 操作如下：



为了实现上述 acquire、release 操作，需要完成以下三个功能：

* 同步状态的原子性管理；
* 线程的阻塞与唤醒；
* 排队机制；

### 同步状态

AQS类通过单个int类型保存同步变量，并提供了getState(),setState(int), compareAndSetState(int, int) 三个方法来读取和更新状态，并且此同步状态通过volatile关键字修饰，保证了多线程环境下的可见性，compareAndSetState(int,int)是通过CAS(compare and swap，比较并交换)实现的，当多个线程同时对某个资源进行CAS操作的时候，只有一个线程操作成功，但并不会阻塞其它线程，其它线程会收到失败的信号，CAS是一个轻量级的乐观锁。CAS的底层是通过Unsafe类实现的，利用处理器的CMPXCHG指令实现其原子性，使得仅当同步器状态为一个期望值的时候，才会被原子的更新成目标值，相比Synchronized不会导致过多的上下文切换和挂起线程。在java.util.concurrent包中，大量地使用了CAS来实现原子性

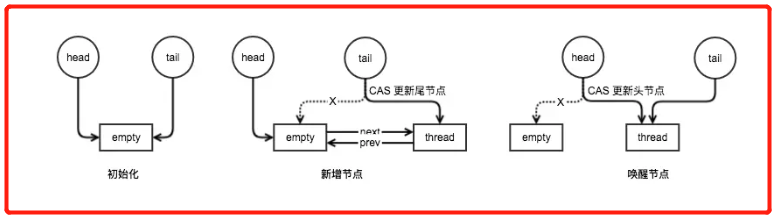
### 线程的阻塞和唤醒

LockSupport是一个非常方便的线程阻塞工具类，它可以在线程任意位置让线程组设。和Thread.suspend(),它弥补了由于 resume() 在前发生，导致线程无法继续执行的情况。和Object.wait()相比，它不需要先获得某个对象的锁，也不会抛出 InterruptedException 异常。LockSupport.park()方法阻塞当前线程，这是因为LockSupport类使用类似信号量的机制，它为每个线程准备了一个许可，如果许可可用，那么park() 会立即返回，并且消费这个许可(设置许可不可用),就会阻塞，而unpark()则使得一个许可可用（但是与信号量不用的是，许可不可以叠加可用，你不可能拥有一个超过一个许可，它永远只有一个）.

### 排队机制

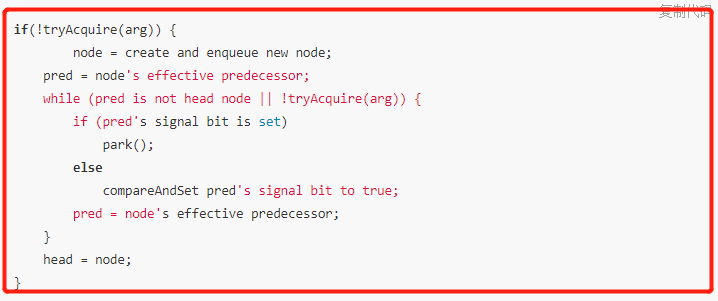
**同步队列** AQS 整个框架的关键都是如何管理被阻塞线程的队列,在AQS中，运用带了CLH锁的思想，CLH锁被用于自旋锁，可以确保没有饥饿感，提供先到先得的公平服务，CLH锁是基于列表的可伸缩性，高能性，公平和自旋锁，应用程序线程仅在局部变量上旋转，它不断轮询前驱状态，如果发现预释放锁定结束旋转。

AQS同步队列是一个FIFO队列。在此队列中，一个节点表示一个线程，它保存着线程的引用、状态、前驱节点、后继节点。同步队列通过两个节点tail和head来存取，初始化时，tail、head初始化为一个空节点，线程要加入到同步队列中，通过CAS原子地拼接为新的tail节点，线程要退出队列，只需要设置head节点指向当前线程节点。

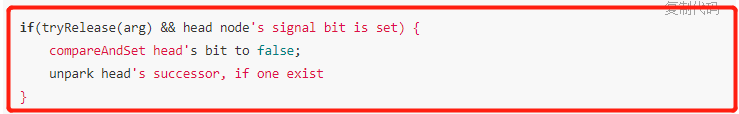


同步队列的优点在于其出队和入队都是无锁的、快速的。为了将CLH锁队列用于阻塞同步器，该同步队列需要做些额外的修改以提供一种高效的方式定位某个节点的后继节点，在自旋锁中，一个节点只需要改变其状态，下一次自旋中其后继节点就能注意到这个改变。但是在同步阻塞器中，一个节点需要显示地唤醒器后继节点。同步队列包含一个next链接到它的后继节点。第二个对 CLH 锁队列主要的修改是将每个节点都有的状态字段用于控制阻塞而非自旋。

基本的 acquire 操作的最终实现一般形式（互斥、非中断、无超时）：



release 操作如下：



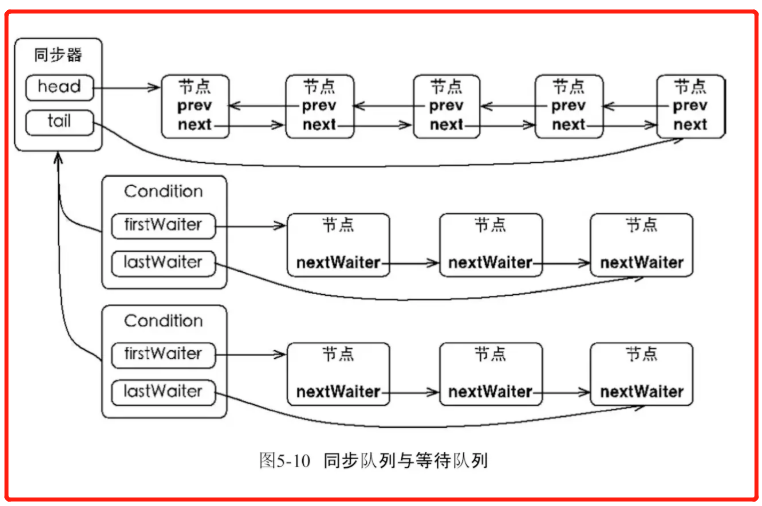
acquire 的操作的主循环次数依赖于具体实现类 tryAcquire 的实现方式。支持取消或超时的操作主要是在 acquire 循环里的 park() 返回时检查中断或超时，由超时或中断而被取消等待的线程会设置其节点状态，然后 unpark 其后继节点。由于“取消”操作，该线程再也不会被阻塞，节点的链接和状态字段可以被快速重建。

尽管同步队列是基于FIFO的，但它们不一定公平的，可以注意到基础的acquire算法中，tryAcquire是在入队前被执行的因此一个新的 acquire 线程能够”窃取“本该属于队列头部的第一个线程获取通过同步器的机会，可闯入FIFO 策略通常会提供比其他技术更高的吞吐率。当然，需要严格的公平性需求时，可以改写 tryAcquire 方法定义，可以通过框架提供的 getFirstQueuedThread() 方法检查是否是头节点，如果是则获取同步状态，如果不是则返回失败。

**等待队列** AQS 框架提供了一个 ConditionObject 类，给维护独占同步的类及实现 Lock 接口的类使用。一个锁对象可以关联任意数目的条件对象，可以提高类似管程风格的 await、signal 和 signalAll 操作，包括带有超时的以及一些检测、监控的方法。

等待队列是一个 FIFO 队列，在队列中的每个节点都包含一个线程引用，该线程就是在 Condition 对象上等待的线程，如果一个线程调用了 Condition.await() 方法，那么线程将会释放锁，构造成节点将加入等待队列进入等待状态。事实上，节点的定义复用了同步器中节点的定义，也就是说同步队列和等待队列中节点类型都是同步器的静态内部类 java.util.concurrent.locks.AbstractQueuedSynchronizer.Node。  


如图所示，Condition 拥有首尾节点的引用，而新增节点只要将原有的尾节点 nextWaiter 指向它，并且更新尾节点即可。上述节点的更新并不需要使用 CAS 保证，原因在于调用 await() 方法的线程必定获取了锁的线程，也就是该过程是由锁来保证线程安全的。调用 sigal() 方法将会唤醒在等待队列中等待时间最长的节点（首节点），主要将 CONDITION 状态节点从等待队列中移到同步队列中。 AQS 的图结构如下：



### 用法





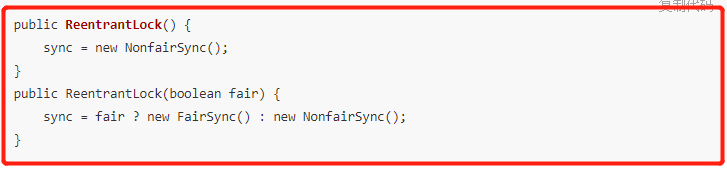
## 3.ReentrantLock

### 概述

一个可重入互斥 Lock 具有与使用Synchronized方法和语句访问的隐式监视锁的基本行为和语义，但具有扩展功能。ReentrantLock支持可重入性，并提供了公平锁和非公平锁（默认非公平锁）。因为独占模式，重写了tryAcquire、tryReplease、isHeldExclusively 三个方法，支持响应中断、超时机制，下面详细解读下 JDK 1.8 中的 ReentrantLock 的源代码实现。

### 构造方法

提供两种构造方法：（1）无参构造方法，构造非公平锁（2）有参构造方法，根据参数值来构造公平、非公平锁。



### lock()

从上面通过构造方法创建了一个可重入锁对象，接下来我们开始看下 ReentrantLock.lock() 方法的源代码实现：

/\*\*

\* 获得锁的方法，如果锁没有被另一个线程占用并且立即返回，则将锁定计数设置为1。

\* 如果当前线程已经保持锁定，则保持计数增加1，该方法立即返回。

\* 如果锁被另一个线程保持，则当前线程将被禁用以进行线程调度，并且在锁定已被获取之前处于休眠状态，此时锁定保持计数被设置为1。

\*\*/

public void lock() {

sync.lock();

}

 ReentrantLock在上锁时，会根据实例化时指定的策略去获取锁，默认为非公平锁。如果上锁成功，锁状态值+1(重入，最大次数为 Integer.MAX\_VALUE)，并将锁持有者设置为当前线程实例。在 Sync 内部维护了一个队列，存放了所有上锁失败的线程。公平锁在上锁前，会检查在自己前面是否还有其他线程等待，如果有就放弃竞争，继续等待。而非公平锁会抓住每个机会，不管是否前面是否还有其它线程等待，只顾上锁

 ReetrantLock在释放锁时，将状态计数器减一(重入)，当状态计数器为0时，锁可用。此时再从等待队列中寻找合适的线程唤醒，默认从队首开始，如果队列正在更新中，且未找到合适的线程，那么从队尾开始寻找

## 4.线程池

多线程的设计方法确实可以最大限度的发挥多核处理器的计算能力，提高吞吐量和性能。但是如果不加控制随意使用线程，对系统的性能反而会产生不利。

和进程相比，线程虽然是一种轻量级的，但是创建和关闭依然需要花费时间，如果每一个小任务都创建一个线程，则会很有可能会出现创建和销毁线程占用的时间大于该线程任务的时间。其次线程本身也是需要占用空间的，大量的线程会抢占宝贵的内存资源。

因此线程的使用需要掌握一个度，再有限的范围内增加线程的数量可以提高系统的性能，一旦超过这个范围，大量的线程只会拖垮整个系统。

### 4.1什么是线程池

为了避免系统频繁的创建和销毁线程，我们可以让创建的线程复用。我们可以使用一个线程池维护一些线程，当你需要使用线程的时候，可以从池子中拿一个空闲的线程，当工作完成时，并不急着关闭线程，而是将线程退回池中，方便下次使用

简而言之，再使用线程池后，创建线程编程了从线程池中获得空闲线程，关闭线程变为想线程池归还线程。

### 4.2线程池的创建

线程池的成员都在java.util.concurrent包中，是JDK并发包的核心。其中ThreadPoolExecutor表示一个线程池。Executors类则是一个线程工厂的角色，通过Executors可以取得一个拥有特定功能的线程池，通过Executors可以取得一个特定功能的线程池。

#### 4.2.1 newFixedThreadPool()方法

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是固定不变的。当有一个新的任务提交时，线程池中如有空闲的线程，则立即执行。若没有则新的任务会暂存在一个队列中，待有线程空闲时，便会处理队列中的任务，**所以适用于任务量比较固定但耗时长的任务**

#### 4.2.2newSingleThraedExecutor()方法

该方法返回一个只有一个线程的线程池。若有多余的任务则放到队列中，待线程空闲时，则按照先进先出的原则处理任务，**适用于串行执行任务的场景，一个任务一个任务地执行。**

#### 4.2.3newCachedThreadPool()方法

该方法会返回一个根据实际调整线程池的大小。线程池的线程数量不确定，当有新任务来的时候，若有空闲线程，则调用该空闲线程，若所有线程都在工作，则新建线程处理任务。所有线程在完成，**这个队列是无法插入任务的，一有任务立即执行，所以CachedThreadPool比较适合任务量大但耗时少的任务。**

#### 4.2.4 newSingleThreadScheduledExecutor()方法

该方法返回一个ScheduledExecutorService对象，线程池大小为1。ScheduledExecutorService接口在ExecutorService接口上扩展了在给定时间执行某任务的功能，如在某个固定的延时后执行，**或者周期性执行某个任务。**

#### 4.2.5 newScheduledThreadPool()方法

该方法会返回一个ScheduledExecutorService对象，但该线程池可以执行线程数量。

##### 使用场景

周期性执行任务的场景，需要限制线程数量的场景

##### ScheduledExecutorService

newScheduledThreadPool()方法返回一个ScheduledExecutorService对象，可以根据时间需要对线程进行调度。主要方法如下

public <V> ScheduledFuture<V> schedule(Callable<V> callable,long delay, TimeUnit unit);

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,long initialDelay, long period,TimeUnit unit);

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit);

和其他线程不同，ScheduledExecutorService不一定会立即安排执行任务。他其实是起到了计划任务的作用，会在指定的时间对任务进行调度。

schedule()会在给定时间对任务进行一次调度。scheduleAtFixedRate()和scheduleWithFixedDelay()方法会对任务进行周期性调度，但是二者还是有区别的。scheduleAtFixedRate()方法的任务调度频率是一定的，它是以上一个任务开始执行的时间为起点，再在规定的时间调度下一次任务。而scheduleWithFixedDelay()方法是以上一个任务的结束后再经过规定时间进行任务调度。

### 4.3线程池的内部实现

对于几个核心的线程池，虽然看着创建的线程池有着不同的功能特点，但是其内部都是使用了ThreadPoolExecutor类。

 corePoolSize：指定了线程池中的线程数量。

 maximumPoolSize：指定了线程池中的最大线程数。

 keepAliveTime：当线程池中的线程数量超过corePoolSize，多余的空闲线程的存活时间，即超过corePoolSize的空闲线程在多长时间内被销毁。

 unit：keepAliveTime的单位。

 workQueue：任务队列，被提交但尚未被执行的任务的存放队列。

 threadFacotry：线程工厂，用于创建线程。

 handler：拒绝策略。当任务太多来不及处理时如何拒绝任务。

#### 4.3.1 workQueue-任务队列

参数workQueue指被提交但未执行的任务队列，它是一个BlockingQueue接口的对象,仅用于存放Runnable对象。在ThreadPoolExecutor构造中可以使用以下几种BlockingQueue接口：

* 直接提交队列：有SynchronousQueue对象提供。SynchronousQueue没有容量，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。
* 有界的任务队列：有界任务队列使用ArrayBlockingQueue类实现，ArrayBlockingQueue类的构造函数必须带一个容量参数，表示该队列的最大容量。有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。
* 无界的任务队列：无界的任务队列使用LinkedBlockingQueue类实现。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

#### 4.3.2 拒绝策略

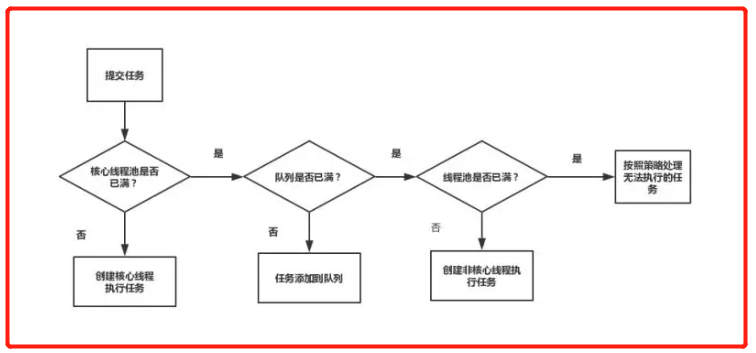
拒绝策略是当任务数量超过系统实际承载能力时执行的策略。拒绝策略可以说时系统超负荷运行时的补救措施。

JDK内置了四种拒绝策略：

* AbortPolicy策略：该策略会直接抛出异常，阻止正常工作。
* CallerRunsPolicy策略：只要线程池未关闭，该策略直接在当前调用者线程中运行当前被丢弃的任务。这样不会真的丢弃线程，但是会使任务提交线程性能下降。
* DiscardOldestPolicy策略：该策略将丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
* DiscardPolicy策略：该策略丢弃无法处理的任务，不进行任何处理。

#### 4.3.3任务执行

线程池执行流程，即对应execute()方法：



* 当有新的任务来的时候，首先判断核心线程池是否有空闲的线程，没有则创建新的线程，前提是核心线程池没有满
* 当核心线程池满了，队列是linkblockingQueue则会加入队列，因为这个队列最大的数量Integer.MaxValue所以很大，线程池的最大线程数则不会起效，如果是SynchronousQueue则不会加入队列，因为要创建新的线程，这个时候线程池的最大线程数就会起效，如果超出了最大线程数，则按拒绝策略执行，默认抛出异常

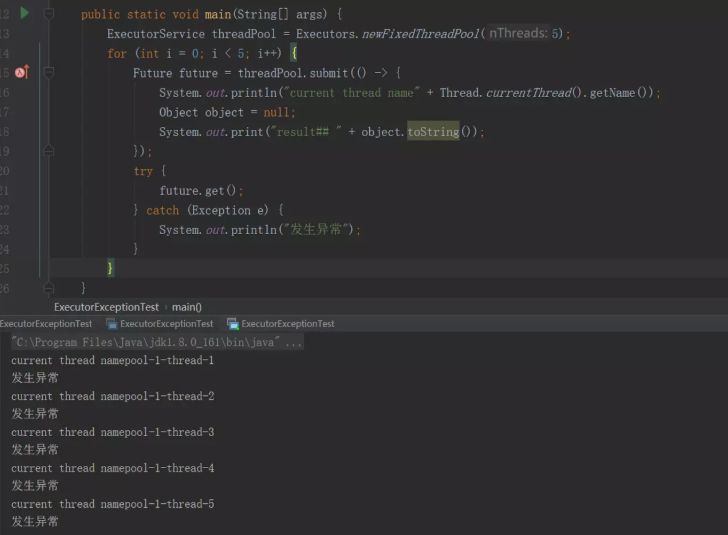
#### 4.3.4线程池异常处理

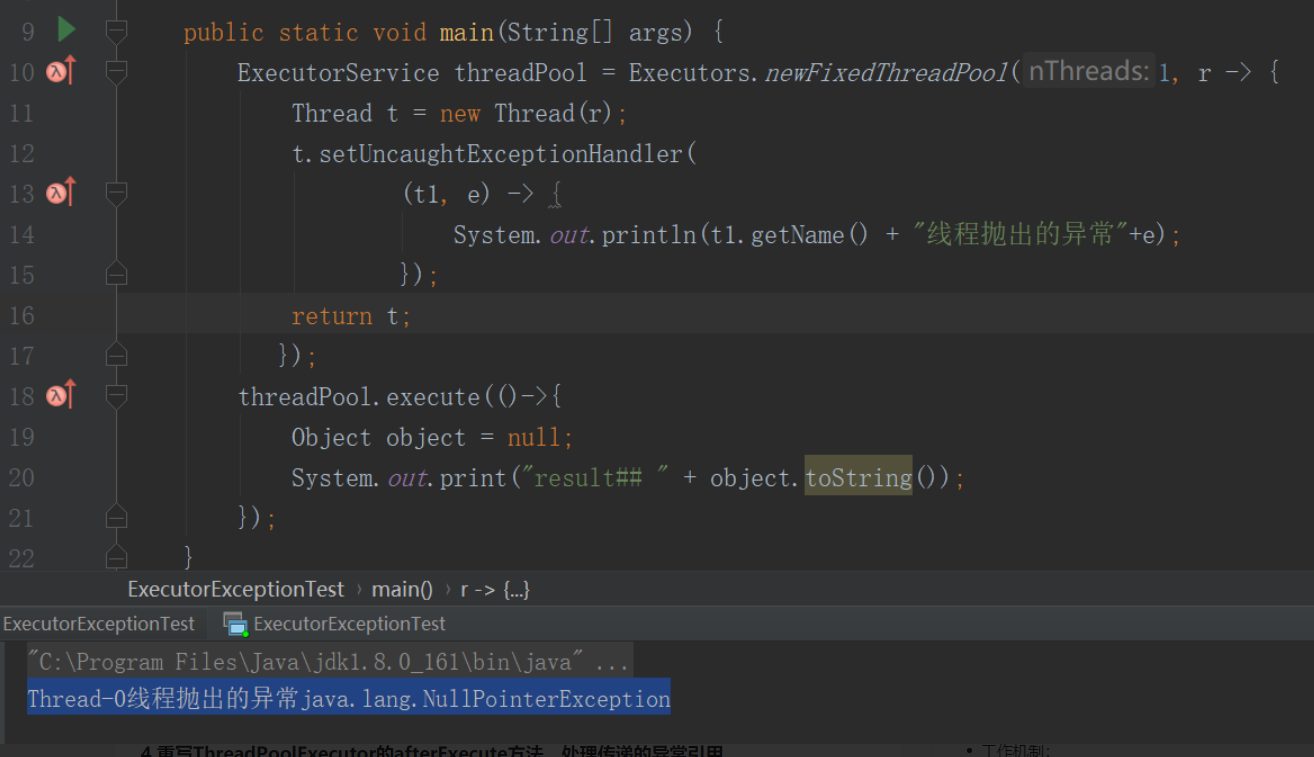
在使用线程池处理任务的时候，任务代码可能抛出RuntimeException，抛出异常后，线程池可能捕获它，也可能创建一个新的线程来代替异常的线程，我们可能无法感知任务出现了异常，因此我们需要考虑线程池的异常情况

**建议：**线程的异常处理，**我们可以直接try...catch捕获。**

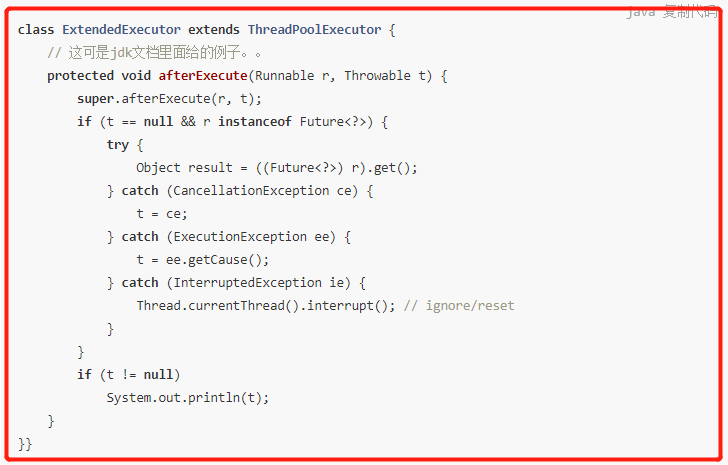
**还有一种就是利用future类处理**

通过以上分析，**submit执行的任务，可以通过Future对象的get方法接收抛出的异常，再进行处理。** 我们再通过一个demo，看一下Future对象的get方法处理异常的姿势，如下图：

**3.为工作者线程设置UncaughtExceptionHandler，在uncaughtException方法中处理异常**

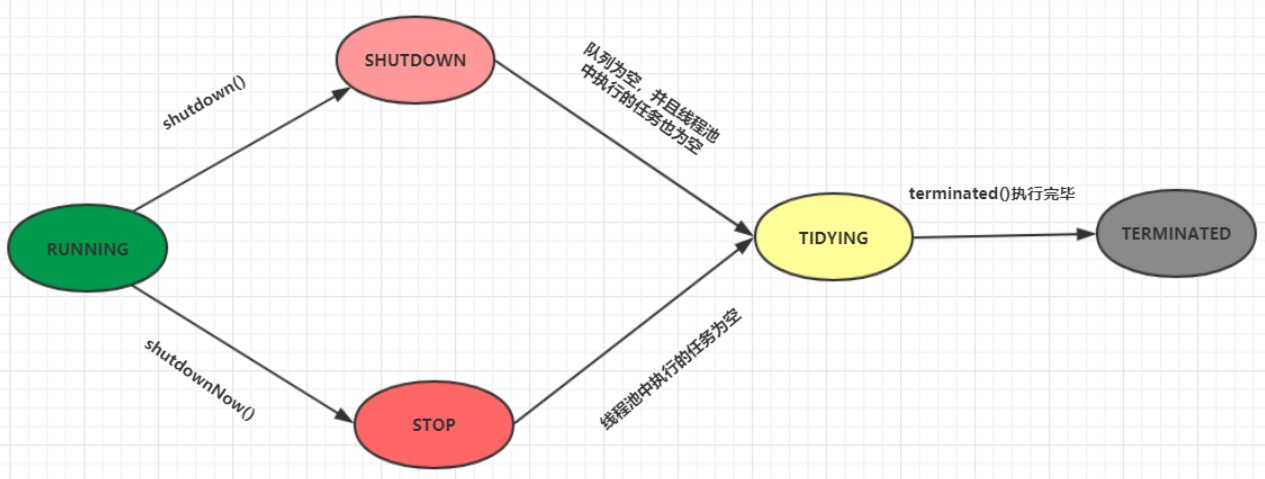


**4.重写ThreadPoolExecutor的afterExecute方法，处理传递的异常引用**



#### 4.3.5线程池状态

##### 线程池各个状态切换图：



**RUNNING**

* 该状态的线程池会接收新任务，并处理阻塞队列中的任务;
* 调用线程池的shutdown()方法，可以切换到SHUTDOWN状态;
* 调用线程池的shutdownNow()方法，可以切换到STOP状态;

**SHUTDOWN**

* 该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；
* 队列为空，并且线程池中执行的任务也为空,进入TIDYING状态;

**STOP**

* 该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；
* 线程池中执行的任务为空,进入TIDYING状态;

**TIDYING**

* 该状态表明所有的任务已经运行终止，记录的任务数量为0。
* terminated()执行完毕，进入TERMINATED状态

**TERMINATED**

* 该状态表示线程池彻底终止