# 四. 并发框架

## Doug Lea

如果IT的历史，是以人为主体串接起来的话，那么肯定少不了Doug Lea。这个鼻梁挂着眼镜，留着德王威廉二世的胡子，脸上永远挂着谦逊腼腆笑容，服务于纽约州立大学Oswego分校计算机科学系的老大爷。  
说他是这个世界上对Java影响力最大的个人，一点也不为过。因为两次Java历史上的大变革，他都间接或直接的扮演了举足轻重的角色。2004年所推出的Tiger。Tiger广纳了15项JSRs(Java Specification Requests)的语法及标准，其中一项便是JSR-166。JSR-166是来自于Doug编写的util.concurrent包。  
值得一提的是: Doug Lea也是JCP (Java社区项目)中的一员。  
Doug是一个无私的人，他深知分享知识和分享苹果是不一样的，苹果会越分越少，而自己的知识并不会因为给了别人就减少了，知识的分享更能激荡出不一样的火花。《Effective JAVA》这本Java经典之作的作者Joshua Bloch便在书中特别感谢Doug Lea是此书中许多构想的共鸣板，感谢Doug Lea大方分享丰富而又宝贵的知识。

# 一.线程

## 0.关于线程你需要搞懂这些：

* 线程的状态
* 线程的几种实现方式
* 三个线程轮流打印ABC十次
* 判断线程是否销毁
* yield功能
* 给定三个线程t1,t2,t3，如何保证他们依次执行

## 1. 基本概念

![fig:](data:text/html; charset=utf-8;base64,)  
![fig:](data:text/html; charset=utf-8;base64,)

## 2. 线程的启动

### 2.1 实现Runnable接口

* 1.自定义一个线程，实现Runnable接口的run方法  
  run方法就是要执行的内容，会在另一个分支上进行  
  Thread类本身也实现了Runnable接口
* 2.主方法中new一个自定义线程对象，然后new一个Thread类对象，其构造方法的参数是自定义线程对象
* 3.执行Thread类的start方法，线程开始执行  
  自此产生了分支，一个分支会执行run方法，在主方法中不会等待run方法调用完毕返回才继续执行，而是直接继续执行，是第二个分支。这两个分支并行运行

这里运用了静态代理模式：  
Thread类和自定义线程类都实现了Runnable接口  
Thread类是代理Proxy，自定义线程类是被代理类  
通过调用Thread的start方法，实际上调用了自定义线程类的start方法（当然除此之外还有其他的代码）

### 2.2 继承Thread类

* 自定义一个类MyThread，继承Thread类，重写run方法
* 在main方法中new一个自定义类，然后直接调用start方法  
  两个方法比较而言第二个方法代码量较少  
  但是第一个方法比较灵活，自定义线程类还可以继承其他的类，而不限于Thread类

### 2.3 实现Callable接口

![](data:text/html; charset=utf-8;base64,)

## 3. 线程的状态

### 初始态：NEW

创建一个Thread对象，但还未调用start()启动线程时，线程处于初始态。

### 运行态：RUNNABLE

在Java中，运行态包括就绪态 和 运行态。

### 就绪态 READY

该状态下的线程已经获得执行所需的所有资源，只要CPU分配执行权就能运行。  
所有就绪态的线程存放在就绪队列中。

### 运行态 RUNNING

获得CPU执行权，正在执行的线程。  
由于一个CPU同一时刻只能执行一条线程，因此每个CPU每个时刻只有一条运行态的线程。

### 阻塞态 BLOCKED

阻塞态专指请求排它锁失败时进入的状态。

### 等待态 WAITING

当前线程中调用wait、join、park函数时，当前线程就会进入等待态。  
进入等待态的线程会释放CPU执行权，并释放资源（如：锁），它们要等待被其他线程显式地唤醒。

### 超时等待态 TIME\_WAITING

当运行中的线程调用sleep(time)、wait、join、parkNanos、parkUntil时，就会进入该状态；  
进入该状态后释放CPU执行权 和 占有的资源。  
与等待态的区别：无需等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。

### 终止态

线程执行结束后的状态。

![](data:text/html; charset=utf-8;base64,)

## 4. 线程的方法

![](data:text/html; charset=utf-8;base64,)

### getName

Thread类的构造方法1  
![fig:](data:text/html; charset=utf-8;base64,)  
Thread类的构造方法2  
![fig:](data:text/html; charset=utf-8;base64,)

* new 一个子类对象的同时也new了其父类的对象，只是如果不显式调用父类的构造方法super(),那么会自动调用无参数的父类的构造方法。  
  可以在自定义类MyThread中（继承自Thread类）中写一个构造方法，显式调用父类的构造方法，其参数为一个字符串,表示创建一个以该字符串为名字的Thread对象。
* 效果是创建了一个MyThread对象，并且其父类Thread对象的名字是给定的字符串。
* 如果不显式调用父类的构造方法super(参数)，那么默认父类Thread是没有名字的。

### isAlive

**isAlive活着的定义是就绪、运行、阻塞状态**  
线程是有优先级的，优先级高的获得Cpu执行时间长，并不代表优先级低的就得不到执行

### sleep（当前线程.sleep）

sleep时持有的锁不会自动释放，sleep时可能会抛出InterruptedException。  
Thread.sleep(long millis)  
一定是当前线程调用此方法，当前线程进入TIME\_WAIT状态，但不释放对象锁，millis后线程自动苏醒进入READY状态。作用：给其它线程执行机会的最佳方式。

### join（其他线程.join）

t.join()/t.join(long millis)  
当前线程里调用线程1的join方法，当前线程进入WAIT状态，但不释放对象锁，直到线程1执行完毕或者millis时间到，当前线程进入可运行状态。  
join方法的作用是将分出来的线程合并回去，等待分出来的线程执行完毕后继续执行原有线程。类似于方法调用。（相当于调用thead.run()）

### yield（当前线程.yield）

Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的cpu时间片，由运行状态变会可运行状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield()不会导致阻塞。

### interrupt（其他线程.interrupt）

* 调用Interrupt方法时，线程的中断状态将被置位。这是每一个线程都具有的boolean标志；  
  中断可以理解为线程的一个标志位属性，表示一个运行中的线程是否被其他线程进行了中断操作。这里提到了其他线程，所以可以认为中断是线程之间进行通信的一种方式，简单来说就是由其他线程通过执行interrupt方法对该线程打个招呼，让起中断标志位为true，从而实现中断线程执行的目的。
* 其他线程调用了interrupt方法后，该线程通过检查自身是否被中断进行响应，具体就是该线程需要调用Thread.currentThread().isInterrupted方法进行判断是否被中断或者调用Thread类的静态方法interrupted对当前线程的中断标志位进行复位（变为false）。需要注意的是，如果该线程已经处于终结状态，即使该线程被中断过，那么调用isInterrupted方法返回仍然是false，表示没有被中断。
* 那么是不是线程调用了interrupt方法对该线程进行中断，该线程就会被中断呢？答案是否定的。因为**Java虚拟机对会抛出InterruptedException异常的方法进行了特别处理：Java虚拟机会将该线程的中断标志位清除，然后抛出InterruptedException，这个时候调用isInterrupted方法返回的也是false**。

**interrupt一个其他线程t时**

* 1）如果线程t中调用了可以抛出InterruptedException的方法，那么会在t中抛出InterruptedException并清除中断标志位。
* 2）如果t没有调用此类方法，那么会正常地将设置中断标志位。

**如何停止线程？**

* 1）在catch InterruptedException异常时可以关闭当前线程；
* 2）循环调用isInterrupted方法检测是否被中断，如果被中断，要么调用interrupted方法清除中断标志位，要么就关闭当前线程。
* 3）无论1）还是2），都可以通过一个volatile的自定义标志位来控制循环是否继续执行

**但是注意！**  
**如果线程中有阻塞操作，在阻塞时是无法去检测中断标志位或自定义标志位的，只能使用1）的interrupt方法才能中断线程，并且在线程停止前关闭引起阻塞的资源（比如Socket）。**

### wait（对象.wait）

* **调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。**
* obj.wait()，当前线程调用对象的wait()方法，当前线程释放对象锁，进入等待队列。依靠notify()/notifyAll()唤醒或者wait(long timeout)timeout时间到自动唤醒。
* 调用wait()方法的线程，如果其他线程调用该线程的interrupt()方法，则会重新尝试获取对象锁。只有当获取到对象锁，才开始抛出相应的InterruptedException异常，从wait中返回。

### notify（对象.notify）

obj.notify()唤醒在此对象监视器上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象监视器上等待的所有线程。

### wait&notify 最佳实践

等待方（消费者）和通知方（生产者）

等待方：  
synchronized(obj){  
 while(条件不满足){  
 obj.wait();  
}  
消费;  
}  
  
通知方：  
synchonized(obj){  
 改变条件;  
 obj.notifyAll();  
}

* 1）条件谓词：
* 将与条件队列相关联的条件谓词以及在这些条件谓词上等待的操作都写入文档。
* 在条件等待中存在一种重要的三元关系，包括**加锁、wait方法和一个条件谓词。在条件谓词中包含多个状态变量，而状态变量由一个锁来保护，因此在测试条件谓词之前必须先持有这个锁。锁对象和条件队列对象（即调用wait和notify等方法所在的对象）必须是同一个对象。**
* 当线程从wait方法中被唤醒时，它在重新请求锁时不具有任何特殊的优先性，而要去其他尝试进入同步代码块的线程一起正常地在锁上进行竞争。
* 每一次wait调用都会隐式地与特定的条件谓词关联起来。当调用某个特定条件谓词的wait时，调用者必须已经持有与条件队列相关的锁，并且这个锁必须保护着构成条件谓词的状态变量。
* 2）过早唤醒：  
  虽然在锁、条件谓词和条件队列之间的三元关系并不复杂，但wait方法的返回并不一定意味着线程正在等待的条件谓词已经变成真了。  
  当执行控制重新进入调用wait的代码时，它已经重新获取了与条件队列相关联的锁。现在条件谓词是不是已经变为真了？或许。在发出通知的线程调用notifyAll时，条件谓词可能已经变成真，但在重新获取锁时将再次变成假。在线程被唤醒到wait重新获取锁的这段时间里，可能有其他线程已经获取了这个锁，并修改了对象的状态。或者，条件谓词从调用wait起根本就没有变成真。你并不知道另一个线程为什么调用notify或notifyAll，也许是因为与同一条件队列相关的另一个条件谓词变成了真。一个条件队列与多个条件谓词相关是一种很常见的情况。  
  基于所有这些原因，每当线程从wait中唤醒时，都必须再次测试条件谓词。
* 3）notify与notifyAll：  
  由于多个线程可以基于不同的条件谓词在同一个条件队列上等待，因此如果使用notify而不是notifyAll，那么将是一种危险的操作，因为单一的通知很容易导致类似于信号地址（线程必须等待一个已经为真的条件，但在开始等待之前没有检查条件谓词）的问题。

只有同时满足以下两个条件时，才能用单一的notify而不是notifyAll：

* 1）所有等待线程的类型都相同。只有一个条件谓词与条件队列相关，并且每个线程在从wait返回后将执行相同的操作。
* 2）单进单出：在对象状态上的每次改变，最多只能唤醒一个线程来执行。

### suspend resume stop destroy（废弃方法）

* 线程的暂停、恢复、停止对应的就是suspend、resume和stop/destroy。
* suspend会使当前线程进入阻塞状态并不释放占有的资源，容易引起死锁；
* stop在结束一个线程时不会去释放占用的资源。它会直接终止run方法的调用，并且会抛出一个ThreadDeath错误。
* destroy只是抛出一个NoSuchMethodError。
* suspend和resume已被wait、notify取代。

### 线程的优先级

![fig:](data:text/html; charset=utf-8;base64,)  
判断当前线程是否正在执行  
注意优先级是概率而非先后顺序（优先级高可能会执行时间长，但也不一定）

线程优先级特性：

* 继承性  
  比如A线程启动B线程，则B线程的优先级与A是一样的。
* 规则性  
  高优先级的线程总是大部分先执行完，但不代表高优先级线程全部先执行完。
* 随机性  
  优先级较高的线程不一定每一次都先执行完。  
  **注意，在不同的JVM以及OS上，线程规划会存在差异，有些OS会忽略对线程优先级的设定。**

### 守护线程

![](data:text/html; charset=utf-8;base64,)

* 将线程转换为守护线程
* 守护线程的唯一用途是为其他线程提供服务。比如计时线程，它定时发送信号给其他线程；
* 当只剩下守护线程时，JVM就退出了。
* 守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至在一个操作的中间发生中断。
* 注意！Java虚拟机退出时Daemon线程中的finally块并不一定会被执行。

### 未捕获异常处理器

在Runnable的run方法中不能抛出异常，如果某个异常没有被捕获，则会导致线程终止。

要求异常处理器实现Thread.UncaughtExceptionHandler接口。  
可以使用setUncaughtExceptionHandler方法为任何一个线程安装一个处理器，  
也可以使用Thread.setDefaultUncaughtExceptionHandler方法为所有线程安装一个默认的处理器；

如果不安装默认的处理器，那么默认的处理器为空。如果不为独立的线程安装处理器，此时的处理器就是该线程的ThreadGroup对象  
ThreadGroup类实现了Thread.UncaughtExceptionHandler接口，它的uncaughtException方法做如下操作：

* 1）如果该线程组有父线程组，那么父线程组的uncaughtException方法被调用。
* 2）否则，如果Thread.getDefaultExceptionHandler方法返回一个非空的处理器，则调用该处理器。
* 3）否则，如果Throwable是ThreadDeath的一个实例（ThreadDeath对象由stop方法产生，而该方法已过时），什么都不做。
* 4）否则，线程的名字以及Throwable的栈踪迹被输出到System.err上。

如果是由线程池ThreadPoolExecutor执行任务，只有通过execute提交的任务，才能将它抛出的异常交给UncaughtExceptionHandler，而通过submit提交的任务，无论是抛出的未检测异常还是已检查异常，都将被认为是任务返回状态的一部分。如果一个由submit提交的任务由于抛出了异常而结束，那么这个异常将被Future.get封装在ExecutionException中重新抛出。

# 二.并发编程的问题

## 线程引入开销：上下文切换与内存同步

使用多线程编程时影响性能的首先是线程的上下文切换。每个线程占有一个CPU的时间片，然后会保存该线程的状态，切换到下一个线程执行。线程的状态的保存与加载就是上下文切换。  
减少上下文切换的方法有：无锁并发编程、CAS、使用最少线程、协程。

* 1）无锁并发：通过某种策略（比如hash分隔任务）使得每个线程不共享资源，避免锁的使用。
* 2）CAS：是比锁更轻量级的线程同步方式
* 3）避免创建不需要的线程，避免线程一直处于等待状态
* 4）协程：单线程实现多任务调度，单线程维持多任务切换

**vmstat可以查看上下文切换次数**  
**jstack 可以dump 线程信息，查看一个进程中各个线程的状态**

* 内存同步：在synchronized和volatile提供的可见性保证中可能会使用一些特殊指令，即内存栅栏。内存栅栏可以刷新缓存，使缓存失效，刷新硬件的写缓冲，以及停止执行管道。
* 内存栅栏可能同样会对性能带来间接的影响，因为它们将抑制一些编译器优化操作。在内存栅栏中，大多数操作都是不能被重排序。  
  不要担心非竞争同步带来的开销，这个基本的机制已经非常快了，并且JVM还能进行额外的优化以进一步降低或消除开销。因此，我们应该将优化重点放在那些发生锁竞争的地方。

## 死锁

死锁后会陷入循环等待中。  
如何避免死锁？

* 1）避免一个线程同时获取多个锁
* 2）避免一个线程在锁内占用多个资源，尽量保证每个锁只占用一个资源
* 3）尝试使用定时锁tryLock替代阻塞式的锁
* 4）对于数据库锁，加锁和解锁必须在一个数据库连接中，否则会解锁失败

# 线程安全性（原子性+可见性）

* 1、对象的状态：对象的状态是指存储在状态变量中的数据，对象的状态可能包括其他依赖对象的域。在对象的状态中包含了任何可能影响其外部可见行为的数据。
* 2、一个对象是否是线程安全的，取决于它是否被多个线程访问。这指的是在程序中访问对象的方式，而不是对象要实现的功能。当多个线程访问某个状态变量并且其中有一个线程执行写入操作时，必须采用同步机制来协同这些线程对变量的访问。同步机制包括synchronized、volatile变量、显式锁、原子变量。
* 3、有三种方式可以修复线程安全问题：
  + 1）不在线程之间共享该状态变量
  + 2）将状态变量修改为不可变的变量
  + 3）在访问状态变量时使用同步
* 4、线程安全性的定义：当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步，这个类都能表现出正确的行为，那么就称这个类是线程安全的。
* 5、无状态变量一定是线程安全的，比如局部变量。
* 6、读取-修改-写入操作序列，如果是后续操作是依赖于之前读取的值，那么这个序列必须是串行执行的。在并发编程中，由于不恰当的执行时序而出现不正确的结果是一种非常重要的情况，它称为竞态条件（Race Condition）。最常见的竞态条件类型就是先检查后执行的操作，通过一个可能失效的观测结果来决定下一步的操作。
* 7、复合操作：要避免竞态条件问题，就必须在某个线程修改该变量时，通过某种方式防止其他线程使用这个变量，从而确保其他线程只能在修改操作完成之前或之后读取和修改状态，而不是在修改状态的过程中。假定有两个操作A和B，如果从执行A的线程看，当另一个线程执行B时，要么将B全部执行完，要么完全不执行B，那么A和B对彼此来说就是原子的。原子操作是指，对于访问同一个状态的所有操作来说，这个操作是一个以原子方式执行的操作。  
  为了确保线程安全性，读取-修改-写入序列必须是原子的，将其称为复合操作。复合操作包含了一组必须以原子方式执行的接口以确保线程安全性。
* 8、在无状态的类中添加一个状态时，如果这个状态完全由线程安全的对象来管理，那么这个类仍然是线程安全的。（比如原子变量）
* 9、如果多个状态是相关的，需要同时被修改，那么对多个状态的操作必须是串行的，需要进行同步。要保持状态的一致性，就需要在单个原子操作中更新所有相关的状态变量。
* 10、内置锁：synchronized(object){同步块}  
  Java的内置锁相当于一种互斥体，这意味着最多只有一个线程能持有这种锁，当线程A尝试获取一个由线程B持有的锁时，线程A必须等待或阻塞，直到线程B释放这个锁。如果B永远不释放锁，那么A也将永远地等待下去。
* 11、重入：当某个线程请求一个由其他线程持有的锁时，发出请求的线程就会阻塞。然而，由于内置锁是可重入的，因此如果某个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。重入意味着获取锁的操作的粒度是线程，而不是调用。重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。当计数值为0时，这个锁就被认为是没有被任何线程持有。当线程请求一个未被持有的锁时，JVM将记下锁的持有者，并且将获取计数值置1。如果一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数值会相应递减。当计数值为0时，这个锁将被释放。
* 12、对于可能被多个线程同时访问的可变状态变量，在访问它时都需要持有同一个锁，在这种情况下，我们称状态变量是由这个锁保护的。  
  每个共享的和可变的变量都应该只由一个锁来保护，从而使维护人员知道是哪一个锁。  
  一种常见的加锁约定是，将所有的可变状态都封装在对象内部，并提供对象的内置锁（this）对所有访问可变状态的代码路径进行同步。在这种情况下，对象状态中的所有变量都由对象的内置锁保护起来。
* 13、不良并发：要保证同步代码块不能过小，并且不要将本应是原子的操作拆分到多个同步代码块中。应该尽量将不影响共享状态且执行时间较长的操作从同步代码块中分离出去，从而在这些操作的执行过程中，其他线程可以访问共享状态。
* 14、可见性：为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。
* 15、加锁与可见性：当线程B执行由锁保护的同步代码块时，可以看到线程A之前在同一个同步代码块中的所有操作结果。如果没有同步，那么就无法实现上述保证。加锁的含义不仅仅局限于互斥行为，还包括内存可见性。为了确保所有线程都能看到共享变量的最新值，所有执行读操作或写操作的线程都必须在同一个锁上同步。
* 16、volatile变量：当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或其他对处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。volatile的语义不足以确保递增操作的原子性，除非你能确保只有一个线程对变量执行写操作。原子变量提供了“读-改-写”的原子操作，并且常常用做一种更好的volatile变量。
* 17、加锁机制既可以确保可见性，又可以确保原子性，而volatile变量只能确保可见性。
* 18、当且仅当满足以下的所有条件时，才应该使用volatile变量：
  + 1）对变量的写入操作不依赖变量的当前值（不存在读取-判断-写入序列），或者你能确保只有单个线程更新变量的值。
  + 2）该变量不会与其他状态变量一起纳入不可变条件中
  + 3）在访问变量时不需要加锁
* 19、栈封闭：在栈封闭中，只能通过局部变量才能访问对象。维护线程封闭性的一种更规范的方法是使用ThreadLocal，这个类能使线程的某个值与保存值的对象关联起来，ThreadLocal通过了get和set等访问接口或方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此get总是返回由当前执行线程在调用set时设置的最新值。
* 20、在并发程序中使用和共享对象时，可以使用一些使用的策略，包括：
  + 1）线程封闭：线程封闭的对象只能由一个线程拥有，对象被封闭在该线程中，并且只能由这个线程修改。
  + 2）只读共享：在没有额外同步的情况下，共享的只读对象可以由多个线程并发访问，但任何线程都不能修改它。共享的只读对象包括不可变对象和事实不可变对象（从技术上来说是可变的，但其状态在发布之后不会再改变）。
  + 3）线程安全共享。线程安全的对象在其内部实现同步，因此多个线程可以通过对象的公有接口来进行访问而不需要进一步的同步。
  + 4）保护对象。被保护的对象只能通过持有对象的锁来访问。保护对象包括封装在其他线程安全对象中的对象，以及已发布并且由某个特定锁保护的对象。
* 21、饥饿：当线程由于无法访问它所需要的资源而不能继续执行时，就发生了饥饿（某线程永远等待）。引发饥饿的最常见资源就是CPU时钟周期。比如线程的优先级问题。在Thread API中定义的线程优先级只是作为线程调度的参考。在Thread API中定义了10个优先级，JVM根据需要将它们映射到操作系统的调度优先级。这种映射是与特定平台相关的，因此在某个操作系统中两个不同的Java优先级可能被映射到同一优先级，而在另一个操作系统中则可能被映射到另一个不同的优先级。  
  当提高某个线程的优先级时，可能不会起到任何作用，或者也可能使得某个线程的调度优先级高于其他线程，从而导致饥饿。  
  通常，我们尽量不要改变线程的优先级，只要改变了线程的优先级，程序的行为就将与平台相关，并且会导致发生饥饿问题的风险。  
  事务T1封锁了数据R,事务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待。然后T4又请求封锁R，当T3释放了R上的封锁之后，系统又批准了T的请求......T2可能永远等待
* 22、活锁  
  活锁是另一种形式的活跃性问题，该问题尽管不会阻塞线程，但也不能继续执行，因为线程将不断重复执行相同的操作，而且总会失败。活锁通常发生在处理事务消息的应用程序中。如果不能成功处理某个消息，那么消息处理机制将回滚整个事务，并将它重新放到队列的开头。虽然处理消息的线程并没有阻塞，但也无法继续执行下去。这种形式的活锁通常是由过度的错误恢复代码造成的，因为它错误地将不可修复的错误作为可修复的错误。  
  当多个相互协作的线程都对彼此进行响从而修改各自的状态，并使得任何一个线程都无法继续执行时，就发生了活锁。要解决这种活锁问题，需要在重试机制中引入随机性。在并发应用程序中，通过等待随机长度的时间和回退可以有效地避免活锁的发生。
* 23、当在锁上发生竞争时，竞争失败的线程肯定会阻塞。JVM在实现阻塞行为时，可以采用自旋等待（Spin-Waiting,指通过循环不断地尝试获取锁，直到成功），或者通过操作系统挂起被阻塞的线程。这两种方式的效率高低，取决于上下文切换的开销以及在成功获取锁之前需要等待的时间。如果等待时间较短，则适合采用自旋等待的方式，而如果等待时间较长，则适合采用线程挂起方式。
* 24、有两个因素将影响在锁上发生竞争的可能性：锁的请求频率，以及每次持有该锁的时间。如果二者的乘积很小，那么大多数获取锁的操作都不会发生竞争，会因此在该锁上的竞争不会对可伸缩性造成严重影响。然而，如果在锁上的请求量很高，那么需要获取该锁的线程将被阻塞并等待。在极端情况下，即使仍有大量工作等待完成，处理器也会被闲置。  
  有3种方式可以降低锁的竞争程度：
  + 1）减少锁的持有时间：  
    缩小锁的范围，将与锁无关的代码移出同步代码块，尤其是开销较大的操作以及可能被阻塞的操作（IO操作）。  
    当把一个同步代码块分解为多个同步代码块时，反而会对性能提升产生负面影响。在分解同步代码块时，理想的平衡点将与平台相关，但在实际情况中，仅可以将一些大量的计算或阻塞操作从同步代码块移出时，才应该考虑同步代码块的大小。  
    减小锁的粒度：锁分解和锁分段  
    锁分解是采用多个相互独立的锁来保护独立的状态变量，从而改变这些变量在之前由单个锁来保护的情况。这些技术能减小锁操作的粒度，并能实现更高的可伸缩性，然而，使用的锁越多，那么发生死锁的风险也就越高。  
    锁分段：比如JDK1.7及之前的ConcurrentHashMap采用的方式就是分段锁的方式。
  + 2）降低锁的请求频率
  + 3）使用带有协调机制的独占锁，这些机制允许更高的并发性比如读写锁，并发容器等

# 四.线程间通信/线程同步 工具使用

## synchronized

synchronized锁定的是对象而非代码，所处的位置是代码块或方法

### 一种使用方法是对代码块使用synchronized关键字

public void fun(){  
 synchronized (this){ }  
}

* 括号中锁定的是普通对象或Class对象
* 如果是this，表示在执行该代码块时锁定当前对象，其他线程不能调用该对象的其他锁定代码块，但可以调用其他对象的所有方法(包括锁定的代码块)，也可以调用该对象的未锁定的代码块或方法。
* 如果是Object o1，表示执行该代码块的时候锁定该对象，其他线程不能访问该对象（该对象是空的，没有方法，自然不能调用）
* 如果是类.class，那么锁住了该类的Class对象，只对静态方法生效。

### 另一种写法是将synchronized作为方法的修饰符

* public synchronized void fun() {} //这个方法执行的时候锁定该当前对象
* 每个类的对象对应一把锁，每个 synchronized 方法都必须获得调用该方法的一个对象的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。
* 如果synchronized修饰的是静态方法，那么锁住的是这个类的Class对象，没有其他线程可以调用该类的这个方法或其他的同步静态方法。
* 实际上，synchronized(this)以及非static的synchronized方法，只能防止多个线程同时执行同一个对象的这个代码段。 synchronized锁住的是括号里的对象，而不是代码。对于非静态的synchronized方法，锁的就是对象本身也就是this。
* 获取锁的线程释放锁只会有两种情况：
  + 1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；
  + 2）线程执行发生异常，此时JVM会让线程自动释放锁。

## Lock

锁是可重入的(reentrant),因为线程可以重复获得已经持有的锁。锁保持一个持有计数（hold count）来跟踪对lock方法的嵌套调用。线程在每一次调用lock都要调用unlock来释放锁。由于这一特性，被一个锁保护的代码可以调用另一个使用相同的锁的方法。

public class TestReentrantLock {  
 public static void main(String[] args) {  
 Ticket ticket = new Ticket();  
 new Thread(ticket, "一号窗口").start();  
 new Thread(ticket, "二号窗口").start();  
 new Thread(ticket, "三号窗口").start();  
 }  
}  
  
class Ticket implements Runnable {  
 private int tickets = 100;  
 private Lock lock = new ReentrantLock();  
  
 @Override  
 public void run() {  
 while (true) {  
 lock.lock();  
 try {  
 Thread.sleep(50);  
 if(tickets > 0){  
 System.out.println(Thread.currentThread().getName() + "正在售票，余票为:" + (--tickets));  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 }  
}

## volatile

（作用是为成员变量的同步访问提供了一种免锁机制，如果声明一个成员变量是volatile的，那么会通知编译器和虚拟机这个成员变量是可能其他线程并发更新的  
对于volatile修饰的变量，jvm虚拟机只是保证从主内存加载到线程工作内存的值是最新的。

Java内存模型简要介绍（后面会详细介绍）：

* 多线程环境下，会共享同一份数据（线程公共的内存空间）。为了提高效率，JVM会为每一个线程设置一个线程私有的内存空间（线程工作内存），并将共享数据拷贝过来。写操作实际上写的是线程私有的数据。当写操作完毕后，将线程私有的数据写回到线程公共的内存空间。
* 如果在写回之前其他线程读取该数据，那么返回的可能是修改前的数据，视读取线程的执行效率而定。
* jvm运行时刻内存的分配：其中有一个内存区域是jvm虚拟机栈，每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值load到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，（从线程内存中读值）
* 在修改完之后的某一个时刻（线程退出之前），把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。

final修饰的变量是线程安全的

内存可见性问题是，当多个线程操作共享数据时，彼此不可见。  
解决这个问题有两种方法：

* 1、加锁：加锁会保证读取的数据一定是写回之后的，内存刷新。但是效率较低
* 2、volatile：会保证数据在读操作之前，上一次写操作必须生效，即写回。
  + 1）修改volatile变量时会强制将修改后的值刷新到主内存中。
  + 2）修改volatile变量后会导致其他线程工作内存中对应的变量值失效。因此，再读取该变量值的时候就需要重新从读取主内存中的值。相较于synchronized是一种较为轻量级的同步策略，但是volatile不具备互斥性；不能保证修改变量时的原子性。

public class TestVolatile {  
 public static void main(String[] args) {  
 MyThread myThread = new MyThread();  
 new Thread(myThread).start();  
 while(true){  
 synchronized (myThread) {  
 if(myThread.isFlag()){  
 System.out.println("flag被设置为true");  
 break;  
 }  
 }  
 }  
 }  
}  
  
class MyThread implements Runnable{  
 private volatile boolean flag = false;  
 public boolean isFlag() {  
 return flag;  
 }  
 public void setFlag(boolean flag) {  
 this.flag = flag;  
 }  
 @Override  
 public void run() {  
 try {  
 Thread.sleep(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 flag = true;  
 System.out.println("flag="+flag);  
 }  
}

## Atomic

原子变量  
可以实现原子性+可见性

# 五.Lock使用 深入

## 可重入锁 ReentrantLock

在一些内置锁无法满足需求的情况下，ReentrantLock可以作为一种高级工具。当需要一些高级功能时才应该使用ReentrantLock，这些功能包括：可定时的、可轮询的与可中断的锁获取操作，公平队列，绑定多个条件以及非块结构的锁。否则，还是应该优先使用synchronized。

* 1）可中断：lock()适用于锁获取操作不受中断影响的情况，此时可以忽略中断请求正常执行加锁操作，因为该操作仅仅记录了中断状态（通过Thread.currentThread().interrupt()操作，只是恢复了中断状态为true，并没有对中断进行响应)。如果要求被中断线程不能参与锁的竞争操作，则此时应该使用lockInterruptibly方法，一旦检测到中断请求，立即返回不再参与锁的竞争并且取消锁获取操作（即finally中的cancelAcquire操作）。
* 2）可定时：tryLock(time)
* 3）可轮询：tryLock()
* 4）可公平：公平锁与非公平锁
* 5）绑定多个条件：一个锁可以对应多个条件，而Object锁只能对应一个条件
* 6）非块结构：加锁与解锁不是块结构的

## Condition（与wait&notify区别）

BlockingQueue就是基于Condition实现的。

一个Condition对象和一个Lock关联在一起，就像一个条件队列和一个内置锁相关联一样。要创建一个Condition，可以在相关联的Lock上调用Lock.newCondition方法。

### Condition与wait&notify区别

![](data:text/html; charset=utf-8;base64,)

* 1）Condition比内置条件等待队列提供了更丰富的功能：在每个锁上可存在 可不响应中断、可等待至某个时间点、可公平的队列操作。  
  wait&notify一定响应中断并抛出遗产；Condition可以响应中断也可以不响应中断
* 2）与内置条件队列不同的是，对于每个Lock，可以有任意数量的Condition对象。
* await() awaitUninterruptibly() await(time) Condition对象继承了相关的Lock对象的公平性，对于公平的锁，线程会按照FIFO顺序从Condition.await中释放。

### await&signal

![fig:](data:text/html; charset=utf-8;base64,)  
await被中断会抛出InterruptedException。

Condition区分开了不同的条件谓词，更容易满足单次通知的需求。signal比signalAll更高效，它能极大地减少在每次缓存操作中发生的上下文切换与锁请求的次数。

线程进入临界区（同步块）时，发现必须要满足一定条件才能执行。要使用一个条件对象来管理那些已经获得一个锁但是不能做有用工作的线程  
条件对象也称为条件变量  
一个锁对象可以有多个相关的条件对象，newCondition方法可以获得一个条件对象。习惯上给每一个条件对象命名为可以反映它所表达的条件的名字。  
当发现条件不满足时，调用Condition对象的await方法  
此时线程被阻塞，并放弃了锁。等待其他线程的相关操作使得条件达到满足

等待获得锁的线程和调用await方法的线程有本质区别。一旦一个线程调用await方法，它进入该条件的等待集。当锁可用时，该线程不能马上解除阻塞，相反，它处于阻塞状态，直到另一个线程调用同一个条件的signalAll方法为止  
await方法和signalAll方法是配套使用的  
await进入等待，signalAll解除等待

signalAll方法会重新激活因为这一条件而等待的所有线程。当这些线程从等待集中移出时，它们再次成为可运行的，线程调度器将再次激活它们。同时它们将试图重新进入该对象。一旦锁可用，它们中的某个将从await调用返回，获得该锁并从被阻塞的地方继续执行

线程应该再次测试该条件。由于无法确保该条件被满足，signalAll方法仅仅是通知正在等待的线程，此时有可能满足条件，值得再次去检测该条件  
对于await方法的调用应该用在这种形式：

while(!(ok to continue)){  
 condition.await();  
}

最重要的是需要其他某个线程调用signalAll方法。当一个线程调用await方法，它没有办法去激活自身，只能寄希望于其他线程。如果没有其他线程来激活等待的线程，那么就会一直等待，出现死锁。  
如果所有其他线程都被阻塞，且最后一个线程也调用了await方法，那么它也被阻塞，此时程序挂起。

signalAll方法不会立刻激活一个等待的线程，仅仅是解除等待线程的阻塞，以便这些线程可以在当前线程（调用signalAll方法的线程）退出时，通过竞争来实现对对象的方法  
这个await和signalAll方法的组合类似于Object对象的wait和notifyAll方法的组合

public class ConditionBoundedBuffer<T> {  
 private static final int BUFFER\_SIZE = 20;  
 protected final Lock lock = new ReentrantLock();  
 private final Condition notFull = lock.newCondition();  
   
 private final Condition notEmpty = lock.newCondition();  
   
 private final T[] items = (T[]) new Object[BUFFER\_SIZE];  
 private int tail;  
 private int head;  
 private int count;  
   
 public void put(T t) throws InterruptedException {  
 lock.lock();  
 try {  
 while(count == items.length){  
 notFull.await();  
 }  
 items[tail] = t;  
 if(++tail == items.length){  
 tail = 0;  
 }  
 ++count;  
 notEmpty.signal();  
 } finally {  
 lock.unlock();  
 }  
 }  
   
 public T take() throws InterruptedException {  
 lock.lock();  
 try{  
 while(count == 0){  
 notEmpty.await();  
 }  
 T t = items[head];  
 items[head] = null;  
 if(++head == items.length){  
 head = 0;  
 }  
 --count;  
 notFull.signal();  
 return t;  
 }finally {  
 lock.unlock();  
 }  
 }  
}

## 公平锁

public ReentrantLock(boolean fair) {  
 sync = fair ? new FairSync() : new NonfairSync();  
}

在公平的锁上，线程按照他们发出请求的顺序获取锁，但在非公平锁上，则允许‘插队’：当一个线程请求非公平锁时，如果在发出请求的同时该锁变成可用状态，那么这个线程会跳过队列中所有的等待线程而获得锁。   
非公平的ReentrantLock 并不提倡插队行为，但是无法防止某个线程在合适的时候进行插队。

非公平锁性能高于公平锁性能的原因：  
在恢复一个被挂起的线程与该线程真正运行之间存在着严重的延迟。

假设线程A持有一个锁，并且线程B请求这个锁。由于锁被A持有，因此B将被挂起。当A释放锁时，B将被唤醒，因此B会再次尝试获取这个锁。与此同时，如果线程C也请求这个锁，那么C很可能会在B被完全唤醒之前获得、使用以及释放这个锁。这样就是一种双赢的局面：B获得锁的时刻并没有推迟，C更早的获得了锁，并且吞吐量也提高了。

当持有锁的时间相对较长或者请求锁的平均时间间隔较长，应该使用公平锁。在这些情况下，插队带来的吞吐量提升（当锁处于可用状态时，线程却还处于被唤醒的过程中）可能不会出现。

非公平锁可能会引起线程饥饿，但是线程切换更少，吞吐量更大

## 读写锁 ReentrantReadWriteLock

读写锁是一种性能优化措施，在一些特定的情况下能实现更高的并发性。在实际情况中，对于在多处理器系统上被频繁读取的数据结构，读写锁能够提高性能。而在其他情况下，读写锁的性能比独占锁的性能要略差一些，这是因为它们的复杂性更高。如果要判断在某种情况下使用读写锁是否会带来性能提升，最好对程序进行分析。由于ReadWriteLock使用Lock来实现锁的读写部分，因此如果分析结果表明读写锁没有提高性能，那么可以很容易地将读写锁换为独占锁。

ReentrantReadWriteLock 如果很多线程从一个数据结构中读取数据而很少线程修改其中数据，那么允许对读的线程共享访问是合适的。

读写锁：分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由jvm自己控制的，你只要上好相应的锁即可。如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！

特性：

* (a).重入方面其内部的写锁可以获取读锁，但是反过来读锁想要获得写锁则永远都不要想。
* (b).写锁可以降级为读锁，顺序是：先获得写锁再获得读锁，然后释放写锁，这时候线程将保持读锁的持有。反过来读锁想要升级为写锁则不可能，为什么？参看(a)
* (c) 读锁不能升级为写锁，目的是保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。
* (d).读锁可以被多个线程持有并且在作用时排斥任何的写锁，而写锁则是完全的互斥。这一特性最为重要，因为对于高读取频率而相对较低写入的数据结构，使用此类锁同步机制则可以提高并发量。
* (e).不管是读锁还是写锁都支持Interrupt，语义与ReentrantLock一致。
* (f).写锁支持Condition并且与ReentrantLock语义一致，而读锁则不能使用Condition，否则抛出UnsupportedOperationException异常。

![](data:text/html; charset=utf-8;base64,)

public class TestReadWriteLock {  
 public static void main(String[] args) {  
 ReadWriteLockDemo demo = new ReadWriteLockDemo();  
 for (int i = 0; i < 100; ++i) {  
 new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 demo.get();  
 }  
 }, "Read" + i).start();  
 }  
 new Thread(new Runnable() {  
  
 @Override  
 public void run() {  
 demo.set(222);  
 }  
 }, "Write").start();  
  
 }  
}  
  
class ReadWriteLockDemo {  
 private ReadWriteLock lock = new ReentrantReadWriteLock();  
 private int data = 2;  
  
 public void get() {  
 lock.readLock().lock();  
 try {  
 System.out.println("读操作 " + Thread.currentThread().getName() + ":" + data);  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
  
 public void set(int data) {  
 lock.writeLock().lock();  
 try {  
 System.out.println("写操作 " + Thread.currentThread().getName() + ":" + data);  
 this.data = data;  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
}

## LockSupport（锁住的是线程，synchronized锁住的是对象）

当需要阻塞或唤醒一个线程的时候，都会使用LockSupport工具类来完成相应工作。  
LockSupport定义了一组公共静态方法，这些方法提供了最基本的线程阻塞和唤醒功能，而LockSupport也成为构建同步组件的基础工具。  
LockSupport定义了一组以park开头的方法用来阻塞当前线程，以及unpark(thread)方法来唤醒一个被阻塞的线程。  
![fig:](data:text/html; charset=utf-8;base64,)  
park等方法还可以传入阻塞对象，有阻塞对象的park方法在dump线程时可以给开发人员更多的现场信息。

park对于中断只会设置中断标志位，不会抛出InterruptedException。  
LockSupport是可不重入的，如果一个线程连续2次调用 LockSupport .park()，那么该线程一定会一直阻塞下去  
unpark函数可以先于park调用。比如线程B调用unpark函数，给线程A发了一个“许可”，那么当线程A调用park时，它发现已经有“许可”了，那么它会马上再继续运行。

LockSupport.park()和unpark()，与object.wait()和notify()的区别？   
主要的区别应该说是它们面向的对象不同。阻塞和唤醒是对于线程来说的，LockSupport的park/unpark更符合这个语义，以“线程”作为方法的参数， 语义更清晰，使用起来也更方便。而wait/notify的实现使得“阻塞/唤醒对线程本身来说是被动的，要准确的控制哪个线程、什么时候阻塞/唤醒很困难， 要不随机唤醒一个线程（notify）要不唤醒所有的（notifyAll）。  
park/unpark模型真正解耦了线程之间的同步。线程之间不再须要一个Object或者其他变量来存储状态。不再须要关心对方的状态。

## synchronized与Lock的区别

* 1）层次：前者是JVM实现，后者是JDK实现
* 2）功能：前者仅能实现互斥与重入，后者可以实现 可中断、可轮询、可定时、可公平、绑定多个条件、非块结构  
  synchronized在阻塞时不会响应中断，Lock会响应中断，并抛出InterruptedException异常。
* 3）异常：前者线程中抛出异常时JVM会自动释放锁，后者必须手工释放
* 4）性能：synchronized性能已经大幅优化，如果synchronized能够满足需求，则尽量使用synchronized

# 六.原子操作类使用

* 1、近年来，在并发算法领域的大多数研究都侧重于非阻塞算法，这种算法用底层的原子机器指令（compareAndSwap）代替锁来确保数据在并发访问中的一致性。非阻塞算法被广泛地用于在操作系统和JVM中实现进程/线程调度机制、垃圾回收机制以及锁和其他并发数据结构。  
  非阻塞算法可以使多个线程在竞争相同的数据时不会发生阻塞，因此它能在粒度更细的层次上进行协调，并且极大地减少调度开销。而且，在非阻塞算法中不存在死锁和其他活跃性问题。在基于锁的算法中，如果一个线程在休眠或自旋的同时持有一个锁，那么其他线程都无法执行下去，而非阻塞算法不会受到单个线程失败的影响。非阻塞算法常见应用是原子变量类。  
  即使原子变量没有用于非阻塞算法的开发，它们也可以用作一个更好的volatile类型变量。原子变量提供了与volatile类型变量相同的内存语义，此外还支持原子的更新操作，从而使它们更加适用于实现计数器、序列发生器和统计数据收集等，同时还能比基于锁的方法提供更高的可伸缩性。
* 2、锁的缺点：
  + 1）在挂起和恢复线程等过程中存在着很大的开销，并且通常存在着较长时间的中断。
  + 2）volatile变量同样存在一些局限：虽然它们提供了相似的可见性保证，但不能用于构建原子的操作。
  + 3）当一个线程正在等待锁时，它不能做任何其他事情。如果一个线程在持有锁的情况下被延迟执行，那么所有需要这个锁的线程都无法执行下去。
  + 4）总之，锁定方式对于细粒度的操作（比如递增计数器）来说仍然是一种高开销的机制。在管理线程之间的竞争应该有一种粒度更细的技术，比如CAS。
* 3、独占锁是一种悲观技术。对于细粒度的操作，还有另外一个更高效的办法，也是一种乐观的办法，通过这种方法可以在不发生干扰的情况下完成更新操作。这种方法需要借助冲突检查机制来判断在更新过程中是否存在来自其他线程的干扰，如果存在，这个操作将失败，并且可以重试。在针对多处理器操作而设计的处理器中提供了一些特殊的指令，用于管理对共享数据的并发访问。在早期的处理器中支持原子的测试并设置（Test-and-Set），获取并递增（Fetch-and-increment）以及交换（swap）指令。现在几乎所有的现代处理器都包含了某种形式的原子读-改-写指令，比如比较并交换（compare-and-swap）等。
* 4、CAS包含了三个操作数——需要读写的内存位置V，进行比较的值A和拟写入的新值B。当且仅当V的值等于A时，CAS才会以原子方式用新值B来更新V的值，否则不会执行任何操作。无论位置V的值是否等于A，都将返回V原有的值。  
  CAS的含义是：我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少。  
  ![fig:](data:text/html; charset=utf-8;base64,)  
  上面这段代码模拟了CAS操作（但实际上不是基于synchronized实现的原子操作，而是由操作系统支持的）。  
  当多个线程尝试使用CAS同时更新一个变量时，只有其中一个线程能更新变量的值，而其他线程都将失败。然而，失败的线程并不会被挂起，而是被告知在这次竞争中失败，并可以再次尝试。由于一个线程在竞争CAS时失败不会被阻塞，因此它可以决定是否重新尝试，或者执行一些恢复操作，也或者不执行任何操作。这种灵活性就大大减少了与锁相关的活跃性风险。
* 5、基于CAS实现的非阻塞计数器  
  ![fig:](data:text/html; charset=utf-8;base64,)
* 6、初看起来，基于CAS的计数器似乎比基于锁的计数器在性能上更差一些，因为它需要执行更多的操作和更复杂的控制流，并且还依赖看似复杂的CAS操作。但实际上，当竞争程序不高时，基于CAS的计数器在性能上远远超过了基于锁的计数器，而在没有竞争时甚至更高。如果要快速获取无竞争的锁，那么至少需要一次CAS操作再加上与其他锁相关操作，因此基于锁的计数器即使在更好的情况下也会比基于CAS的计数器在一般情况下能执行更多的操作。  
  CAS的主要缺点是，它将使调用者处理竞争问题，而在锁中能自动处理竞争问题。
* 7、原子变量比锁的粒度更细，量级更轻，并且对于在多处理器系统上实现高性能的并发代码来说是非常关键的。原子变量将发生竞争的范围缩小到单个变量上，这是你获得的粒度最细的情况。更新原子变量的快速路径不会比获取锁的路径慢，并且通常会更快，而它的慢速路径肯定比锁的慢速路径快，因为它不需要挂起或重新调度线程。  
  原子变量类相当于一种泛化的volatile变量，能够支持原子的和有条件的读-改-写操作。  
  共用12个原子变量类，可分为4组：标量类、更新器类、数组类以及复合变量类。原子数组类中的元素可以实现原子更新。  
  原子数组类中的元素可以实现原子更新。原子数组类为数组的元素提供了volatile类型的访问语义，这是普通数组锁不具备的特性——volatile类型的数组仅在数组引用上具有volatile语义，而在其元素上则没有。  
  ![fig:](data:text/html; charset=utf-8;base64,)
* 8、ABA问题  
  有时候还需要知道“自从上次看到V的值为A以来，这个值是否发生了变化”，在某些算法中，如果V的值首先由A变为B，再由B变为A，那么仍然被认为是发生了变化，并需要重新执行算法中的某些步骤。  
  有一个相对简单的解决方案：不是更新某个引用的值，而是更新两个值，包括一个引用和一个版本号。即使这个值由A变为B，然后又变为A，版本号也将是不同的。AtomicStampedReference支持在两个变量上执行原子的条件更新。  
  AtomicStampedReference将更新一个“对象-引用”二元组，通过在引用上加上：“版本号”，从而避免ABA问题。类似地，AtomicMarkableRefernce将更新一个“对象引用-布尔值”二元组，在某些算法中将通过这种二元组使节点保存在链表中同时又将其标记为“已删除的节点”。CAS存在ABA，循环时间长开销大，以及只能保证一个共享变量的原子操作（变量合并，AtomicReference）三个问题。

# 七.Java内存模型 线程同步工具原理

## JMM Java Memory Model

### JMM抽象结构

在Java中，堆内存在线程之间共享，线程之间的通信由Java内存模型JMM控制。线程之间的共享变量存储在主内存中，每个线程都有一个私有的本地内存（并不真实存在），本地内存中存储了线程读写共享变量的副本。  
![fig:](data:text/html; charset=utf-8;base64,)

如果线程A与线程B之间要通信的话,必须要经历下面2个步骤：

* 1)线程A把本地内存A中更新过的共享变量刷新到主内存中去。
* 2)线程B到主内存中去读取线程A之前已更新过的共享变量。

### 指令重排序

* 在执行程序时，为了提高性能，编译器和CPU常常会对指令进行重排序，分为以下3种类型：
* 1、编译优化重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句执行顺序。
* 2、指令级并行的重排序。CPU采用了指令级并行技术将多条指令重叠执行。
* 3、内存系统的重排序。由于CPU使用cache和读/写缓冲区，因此加载和存储操作可能在乱序执行。  
  ![fig:](data:text/html; charset=utf-8;base64,)
* 1属于编译器重排序,2和3属于处理器重排序。这些重排序可能会导致多线程程序出现内存可见性问题。
* 对于编译器,JMM的编译器重排序规则会禁止特定类型的编译器重排序(不是所有的编译器重排序都要禁止)。
* 对于处理器重排序,JMM的处理器重排序规则会要求Java编译器在生成指令序列时,插入特定类型的内存屏障(Memory Barriers,Intel称之为Memory Fence)指令,通过内存屏障指令来禁止特定类型的处理器重排序。

### 内存屏障

JMM把内存屏障分为四类：

* LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
* StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
* LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
* StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

### happens-before（抽象概念，基于内存屏障）

JDK1.5后，Java采用JSR133内存模型，通过happens-before概念来阐述操作之间的内存可见性。在JMM中，**如果一个操作执行的结果要对另一个操作可见，那么这两个操作之间必须要有happens-before关系。**

定义：

* 1)如果一个操作happens-before另一个操作,那么第一个操作的执行结果将对第二个操作可见,而且第一个操作的执行顺序排在第二个操作之前。
* 2)两个操作之间存在happens-before关系,并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果,与按happens-before关系来执行的结果一致,那么这种重排序并不非法(也就是说,JMM允许这种重排序)。
* 上面的1)是JMM对程序员的承诺。从程序员的角度来说,可以这样理解happens-before关系:如果A happens-before B,那么Java内存模型将向程序员保证——A操作的结果将对B可见,且A的执行顺序排在B之前。注意,这只是Java内存模型向程序员做出的保证!
* 上面的2)是JMM对编译器和处理器重排序的约束原则。正如前面所言,JMM其实是在遵循一个基本原则:只要不改变程序的执行结果(指的是单线程程序和正确同步的多线程程序)编译器和处理器怎么优化都行。JMM这么做的原因是:程序员对于这两个操作是否真的被重排序并不关心,程序员关心的是程序执行时的语义不能被改变(即执行结果不能被改变)。因此,happens-before关系本质上和as-if-serial语义是一回事。

与程序员密切相关的happens-before规则如下。

* 1）程序顺序规则:一个线程中的每个操作,happens-before于该线程中的任意后续操作。（单线程顺序执行）
* 2）监视器锁规则:对一个锁的解锁,happens-before于随后对这个锁的加锁。（先解锁后加锁）比如:
  + lock.unlock();
  + lock.lock();
  + 那么不会重排序，因为重排序后肯定会发生死锁
* 3）volatile变量规则:对一个volatile域的写,happens-before于任意后续对这个volatile域的读。（先写后读）
* 4）传递性:如果A happens-before B,且B happens-before C,那么A happens-before C。
* 5）start规则:如果线程A执行操作ThreadB.start()，那么A线程的ThreadB.start() happens-before于线程B的任意操作。
* 6）join规则:如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

**happens-before与JMM的关系如下：**

## 指令重排序

重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

### 数据依赖性

* 对于单个CPU和单个线程中所执行的操作而言，如果两个操作都访问了同一个变量，且两个操作中有写操作，那么这两个操作就具有数据依赖性。
* **（RW,WW,WR）这三种操作只要重排序对操作的执行顺序，程序的执行结果就会被改变，因此，编译器和处理器在进行重排序的时候会遵守数据依赖性，不会改变存在数据依赖关系的两个操作的执行顺序。**

### as-if-serial

* as-if-serial：无论如何重排序，（单线程）程序的执行结果不能被改变。  
  编译器，runtime，CPU都必须遵守as-if-serial语义，因此，编译器和CPU不会对存在数据依赖关系的操作进行重排序。
* 在单线程中，对存在控制依赖性的操作进行重排序，不会改变执行结果，而在多线程中则可能会改变结果。

## 顺序一致性

程序未正确同步的时候，就可能存在数据竞争。  
数据竞争：

* 1）在一个线程中写一个变量
* 2）在另一个线程中读同一个变量
* 3）而且写和读没有通过同步来排序。

JMM对正确同步的多线程程序的内存一致性做了如下保证：  
如果程序是正确同步的，程序的执行将具有顺序一致性——即程序的执行结果与该程序的顺序一致性内存模型的执行结果相同。  
顺序一致性模型有两大特性：

* 1）一个线程中的所有操作必须按照程序的顺序来执行
* 2）不管程序是否同步，所有线程都只能看到单一的操作执行顺序。每个操作都必须原子执行且立即对所有线程可见。

JMM中，临界区内的代码可以重排序。

而对于未正确同步的多线程程序，JMM只提供最小的安全性：线程执行时所读取到的值，要么是之前某个线程所写入的值，要么是默认值。

## volatile原理

### 汇编上的实现

volatile修饰的共享变量在转换为汇编语言后，会出现Lock前缀指令，该指令在多核处理器下引发了两件事：

* 1、将当前处理器缓存行（CPU cache中可以分配的最小存储单位）的数据写回到系统内存。
* 2、这个写回内存的操作使得其他CPU里缓存了该内存地址的数据无效。  
  （当前CPU该变量的缓存回写；其他CPU该变量的缓存失效）

### 内存语义

volatile写的内存语义：

* 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存  
  volatile读的内存语义：
* 当读一个volatile变量时，JMM会把线程对应的本地内存置为无效，线程接下来将从主内存读取共享变量

一个volatile变量的单个读写操作，与一个普通变量的读写操作使用同一个锁来同步，它们的执行效果相同。锁的happens-before规则保证释放锁和获取锁的两个线程之间的内存可见性，这也意味着对一个volatile变量的读操作，总是能看到任意线程对该变量最后的写入。

对于volatile变量本身的单个读写操作具有原子性，但是与锁不同的是，多个对于volatile变量的复合操作不具有原子性。而锁的语义保证了临界区代码的执行具有原子性。

JAVA1.5后，JSR-133增强了volatile的内存语义，严格限制编译器和CPU对于volatile变量与普通变量的重排序，从而确保volatile变量的写-读操作可以实现线程之间的通信，提供了一种比锁更轻量级的线程通信机制。从内存语义的角度而言，volatile的写-读与锁的释放-获取有相同的内存效果：写操作=锁的释放；读操作=锁的获取。

A线程写一个volatile变量x后，B线程读取x以及其他共享变量。

* 1. 当A线程对x进行写操作时，JMM会把该线程A对应的cache中的共享变量值刷新到主存中.（实质上是线程A向接下来要读变量x的线程发出了其对共享变量修改的消息）
* 2.当B线程对x进行读取时，JMM会把该线程对应的cache值设置为无效，而从主存中读取x。（实质上是线程B接收了某个线程发出的对共享变量修改的消息）

两个步骤综合起来看，在线程B读取一个volatile变量x后，线程A本地cache中在写这个变量x之前所有其他可见的共享变量的值都立即变得对B可见。线程A写volatile变量x，B读x的过程实质上是线程A通过主存向B发送消息。

需要注意的是，由于volatile仅仅保证对单个volatile变量的读写操作具有原子性，而锁的互斥则可以确保整个临界区代码执行的原子性。

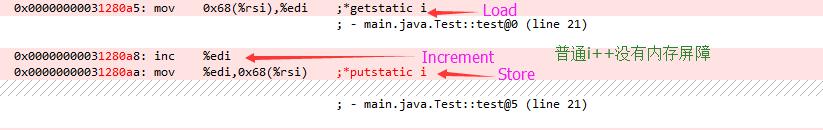
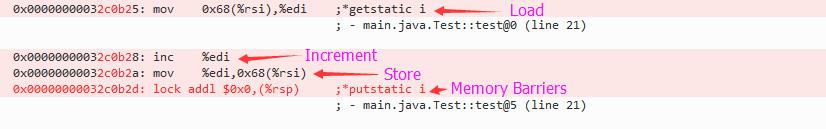
### 内存语义的实现（内存屏障）

* 在每个volatile写操作的前面插入一个StoreStore屏障
* 在每个volatile写操作的后面插入一个StoreLoad屏障
* 在每个volatile读操作的后面插入一个LoadLoad屏障
* 在每个volatile读操作的后面插入一个LoadStore屏障

StoreStore屏障；（禁止上面的普通写和下面的volatile写重排序，保证上面所有的普通写在volatile写之前刷新到主内存）  
volatile写；  
StoreLoad屏障；（禁止上面的volatile写和下面的volatile读/写重排序）

volatile读；  
LoadLoad屏障；  
LoadStore屏障；

**从汇编入手分析volatile多线程问题**

* 1、普通方式int i，执行i++：  
    
  普通方式没有任何与锁有关的指令；其他方式都出现了与锁相关的汇编指令lock。  
  解释指令：其中edi为32位寄存器。如果是long则为64位的rdi寄存器。
* 2、volatile方式volatile int i，执行i++：  
    
  指令“lock; addl $0,0(%%esp)”表示加锁，把0加到栈顶的内存单元，该指令操作本身无意义，但这些指令起到内存屏障的作用，让前面的指令执行完成。具有XMM2特征的CPU已有内存屏障指令，就直接使用该指令。  
  volatile字节码为：  
  

## synchronized原理

### monitor

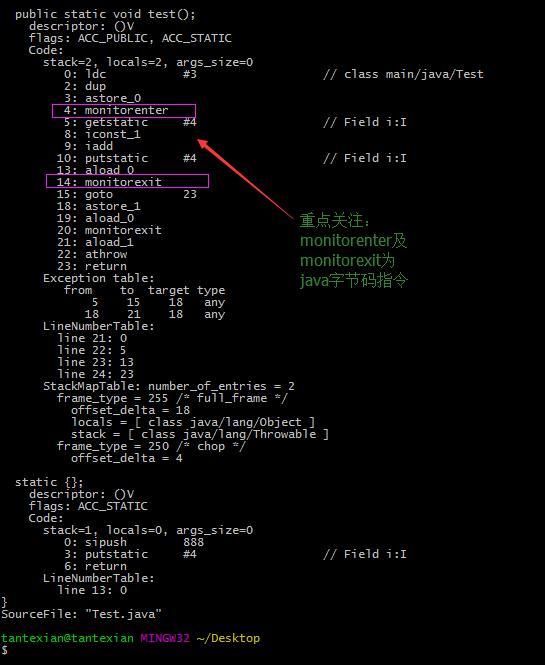
代码块同步是使用monitorenter和monitorexit指令实现。monitorenter和monitorexit指令是在编译后插入到同步代码块开始和结束的的位置。任何一个对象都有一个monitor与之关联，当一个monitor被某个线程持有之后，该对象将处于锁定状态。线程执行到monitorenter指令时，会尝试获取该对象对应的monitor所有权，也即获得对象的锁。

monitorenter ：每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

* 1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
* 2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
* 3、如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit：执行monitorexit的线程必须是对象所对应的monitor的所有者。  
指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。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。  
**其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因**  
在HotSpotJVM实现中，锁有个专门的名字：对象监视器。

### 汇编上的实现（cmpxchg）

synchronizied方式实现i++  
字节码：  
  
汇编  
  
monitorenter与monitorexit包裹了getstatic i及putstatic i，等相关代码执行指令。中间值的交换采用了原子操作lock cmpxchg %rsi,(%rdi)，如果交换成功，则执行goto直接退出当前函数return。如果失败，执行jne跳转指令，继续循环执行，直到成功为止。

cmpxchg指令：比较rsi和目的操作数rdi(第一个操作数)的值，如果相同，ZF标志被设置，同时源操作数(第二个操作)的值被写到目的操作数，否则，清ZF标志为0，并且把目的操作数的值写回rsi，则执行jne跳转指令。

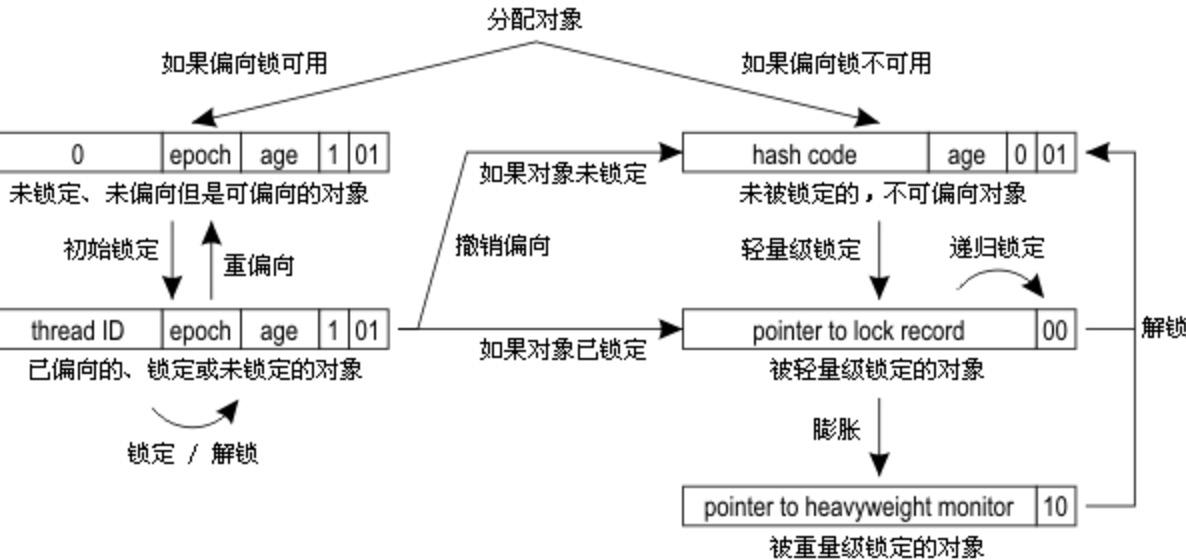
### Java对象头

synchronized用的锁放在java对象头里。  
有两种情况：  
数组对象，虚拟机使用3个字宽存储对象头。  
非数组对象，则使用2个字宽来存储对象头。32位虚拟机中，1个字宽等于4字节，即32字节。

|  |  |  |
| --- | --- | --- |
| 长度 | 内容 | 说明 |
| 32/64bit | mark word | 存储对象的hashCode或者锁信息 |
| 32/64bit | Class metadata address | 存储对象描述数据的指针 |
| 32/64bit | Array length | 数组的长度 |

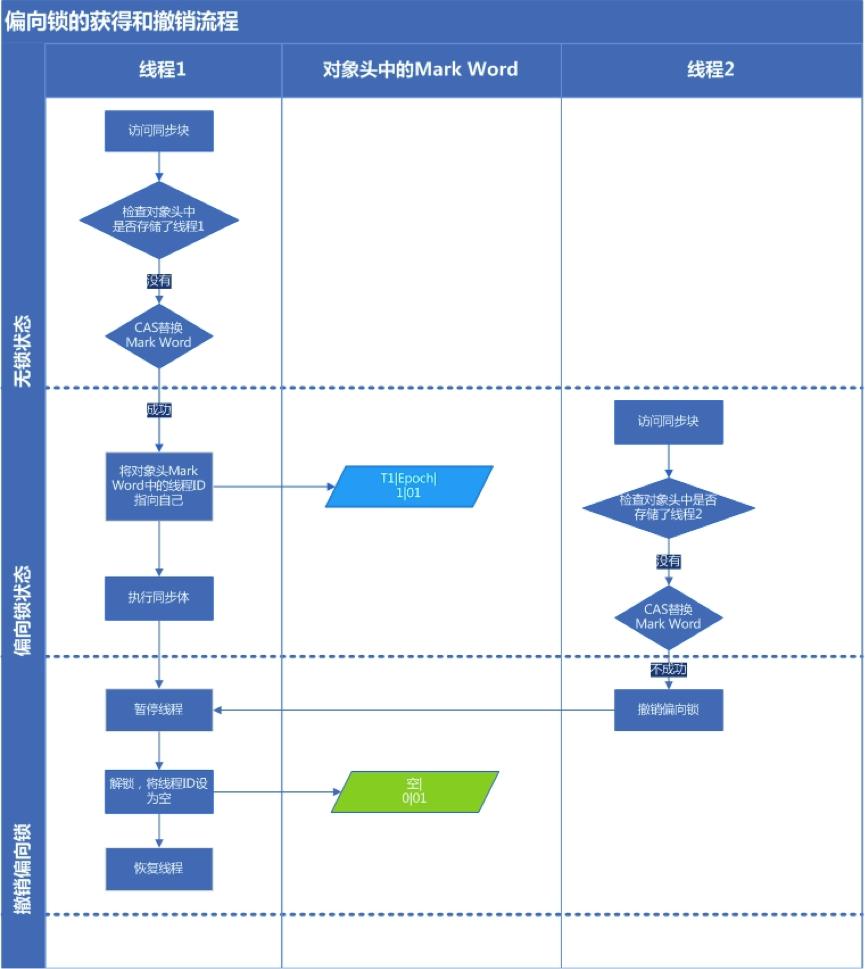
Mark Word 的存储结构：  


### 锁的分类

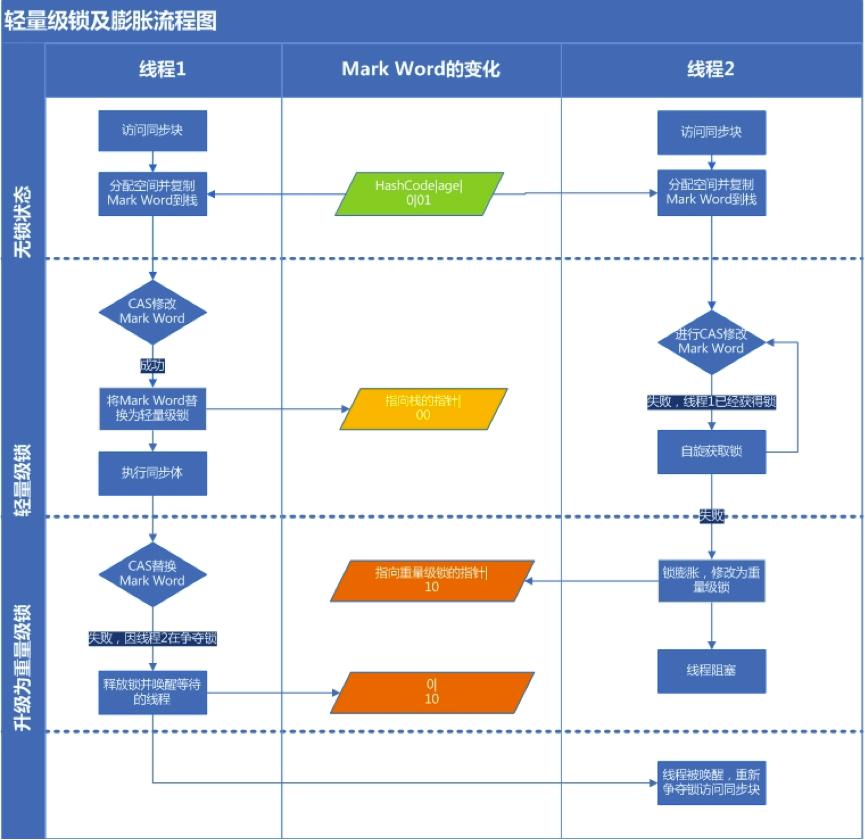
* synchronized是重量级锁，效率较低。
* synchronized所用到的锁是存在Java对象头中。在Java1.6中，锁一共有4种状态，由低到高依次是：无锁，偏向锁，轻量级锁，重量级锁，这几种状态会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。
* monitorenter和monitorexit是上层指令，底层实现可能是偏向锁、轻量级锁、重量级锁等。  
  

#### 偏向锁（只有一个线程进入临界区）

引入背景：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，为了让线程获得锁的代价更低而引入了偏向锁，减少不必要的CAS操作。

* 加锁：当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要花费CAS操作来加锁和解锁，而只需简单的测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁，如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁），如果没有设置，则使用CAS竞争锁，如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程（此时会引发竞争，偏向锁会升级为轻量级锁）。
* 膨胀过程：当前线程执行CAS获取偏向锁失败（这一步是偏向锁的关键），表示在该锁对象上存在竞争并且这个时候另外一个线程获得偏向锁所有权。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，并从偏向锁所有者的私有Monitor Record列表中获取一个空闲的记录，并将Object设置LightWeight Lock状态并且Mark Word中的LockRecord指向刚才持有偏向锁线程的Monitor record，最后被阻塞在安全点的线程被释放，进入到轻量级锁的执行路径中，同时被撤销偏向锁的线程继续往下执行同步代码。
* 偏向锁，顾名思义，它会偏向于第一个获取锁的线程，如果在接下来的运行过程中，该锁没有被其他的线程尝试获取，则持有偏向锁的线程将永远不需要触发同步。  
  在锁对象的对象头中有个偏向锁的线程ID字段，这个字段如果是空的，第一次获取锁的时候，就CAS将自身的线程ID写入到MarkWord的偏向锁线程ID字段内，将MarkWord中的偏向锁的标识置1。这样下次获取锁的时候，直接检查偏向锁线程ID是否和自身线程ID一致，如果一致，则认为当前线程已经获取了锁，因此不需再次获取锁；
* 如果不一致，则表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活，如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程，如果原来的线程依然存活，则偏向锁升级为轻量级锁，（偏向锁就是这个时候升级为轻量级锁的）  
  

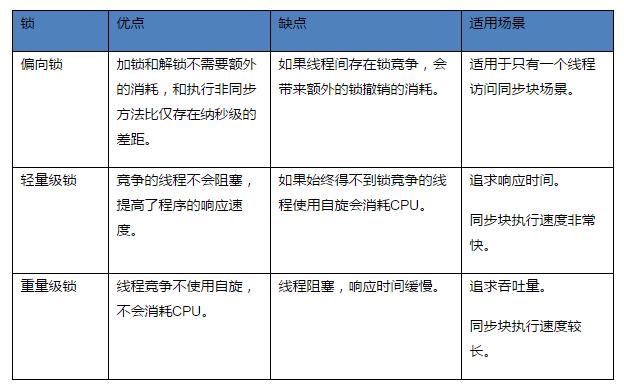
#### 轻量级锁（多个线程交替进入临界区）

* 引入背景：轻量级锁认为竞争存在，但是竞争的程度很轻，一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（自旋），另一个线程就会释放锁。 但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁，重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。
* 轻量级锁加锁：线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋重试。重试一定次数后膨胀为重量级锁（修改MarkWord，改为指向重量级锁的指针），阻塞当前线程。
* 轻量级锁解锁：轻量级解锁时，会使用原子的CAS操作来将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示有其他线程尝试获得锁，则释放锁，并唤醒被阻塞的线程。  
  

#### 重量级锁（多个线程同时进入临界区）

在JDK 1.6之前，监视器锁可以认为直接对应底层操作系统中的互斥量（mutex）。这种同步方式的成本非常高，包括系统调用引起的内核态与用户态切换、线程阻塞造成的线程切换等。因此，后来称这种锁为“重量级锁”。

### 锁的比较



### 锁的优化

Java1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

#### 自旋锁

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。所以引入自旋锁。

**何谓自旋锁？**

* 所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。怎么等待呢？执行一段无意义的循环即可（自旋）。
* 自旋等待不能替代阻塞，先不说对处理器数量的要求（多核，貌似现在没有单核的处理器了），虽然它可以避免线程切换带来的开销，但是它占用了处理器的时间。如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好，反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，反而会带来性能上的浪费。所以说，自旋等待的时间（自旋的次数）必须要有一个限度，如果自旋超过了规定的时间仍然没有获取到锁，则应该被挂起。

#### 适应性自旋锁

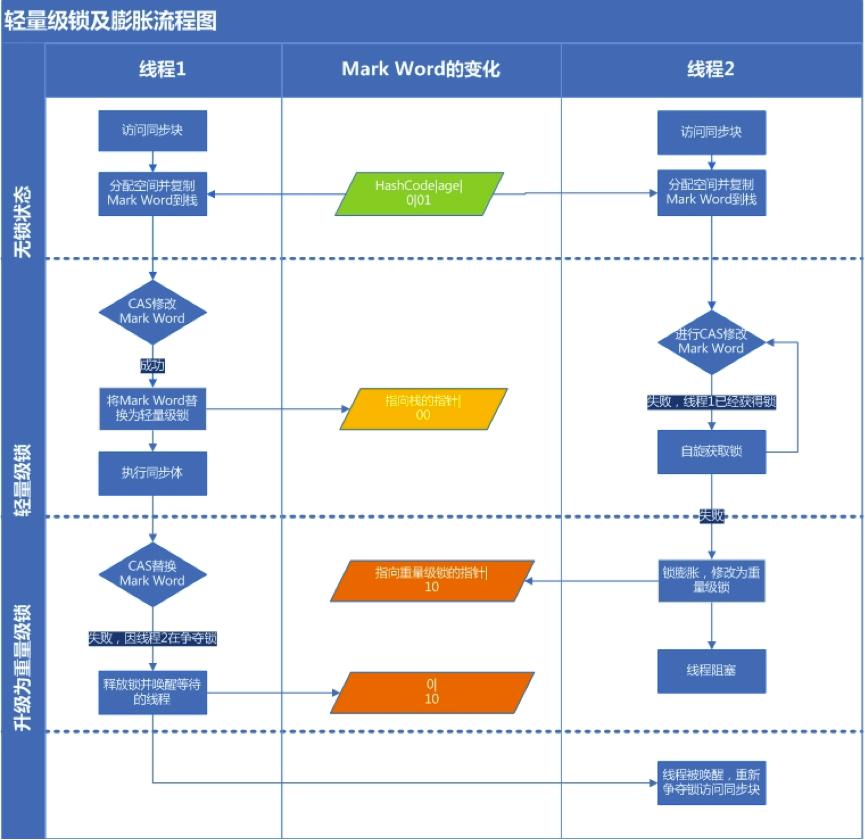
* JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。所谓自适应就意味着自旋的次数不再是固定的，它是由上一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。它怎么做呢？线程如果自旋成功了，那么下次自旋的次数会更加多，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。
* 反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要获得这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源。   
  有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明。

#### 锁消除

为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持。

#### 锁粗化

public void vectorTest(){  
 Vector<String> vector = new Vector<String>();  
 for(int i = 0 ; i < 10 ; i++){  
 vector.add(i + "");  
 }  
  
 System.out.println(vector);  
 }

* 我们知道在使用同步锁的时候，需要让同步块的作用范围尽可能小—仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。
* 在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗化的概念。
* 锁粗化是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。如上面实例：vector每次add的时候都需要加锁操作，JVM检测到对同一个对象（vector）连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到for循环之外。  
  

## 原子操作原理

CAS操作的意思是比较并交换，它需要两个数值，一个旧值（期望操作前的值）和新值。操作之前比较两个旧值是否变化，如无变化才交换为新值。

### CPU如何实现原子操作

* **1）在硬件层面，CPU依靠总线加锁和缓存锁定机制来实现原子操作。**
  + 使用总线锁保证原子性。如果多个CPU同时对共享变量进行写操作（i++），通常无法得到期望的值。CPU使用总线锁来保证对共享变量写操作的原子性，当CPU在总线上输出LOCK信号时，其他CPU的请求将被阻塞住，于是该CPU可以独占共享内存。
  + 使用缓存锁保证原子性。频繁使用的内存地址的数据会缓存于CPU的cache中，那么原子操作只需在CPU内部执行即可，不需要锁住整个总线。缓存锁是指在内存中的数据如果被缓存于CPU的cache中，并且在LOCK操作期间被锁定，那么当它执行锁操作写回到内存时，CPU修改内部的内存地址，并允许它的缓存一致性来保证操作的原子性，因为缓存一致性机制会阻止同时修改由两个以上处理器 缓存的 内存区域数据。当其他CPU回写被锁定的cache行数据时候，会使cache行无效。

### Java如何实现原子操作

* **2）Java使用了锁和循环CAS的方式来实现原子操作。**
  + 使用循环CAS实现原子操作。JVM的CAS操作使用了CPU提供的CMPXCHG指令来实现，自旋式CAS操作的基本思路是循环进行CAS操作直到成功为止。1.5之后的并发包中提供了诸如AtomicBoolean, AtomicInteger等包装类来支持原子操作。CAS存在ABA，循环时间长开销大，以及只能保证一个共享变量的原子操作三个问题。
  + cmpxchg(void\* ptr, int old, int new)，如果ptr和old的值一样，则把new写到ptr内存，否则返回ptr的值，整个操作是原子的。
  + 使用锁机制实现原子操作。锁机制保证了只有获得锁的线程才能给操作锁定的区域。JVM的内部实现了多种锁机制。除了偏向锁，其他锁的方式都使用了循环CAS，也就是当一个线程想进入同步块的时候，使用循环CAS方式来获取锁，退出时使用CAS来释放锁。

### CAS在OpenJDK中的实现

以compareAndSwapInt为例：

UNSAFE\_ENTRY(jboolean, Unsafe\_CompareAndSwapInt(JNIEnv \*env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))  
 UnsafeWrapper("Unsafe\_CompareAndSwapInt");  
 oop p = JNIHandles::resolve(obj);  
 jint\* addr = (jint \*) index\_oop\_from\_field\_offset\_long(p, offset);  
 return (jint)( (x, addr, e)) == e;  
UNSAFE\_END

linux下

inline jint Atomic::cmpxchg (jint exchange\_value, volatile jint\* dest, jint compare\_value) {  
 int mp = os::is\_MP();  
 \_\_asm\_\_ volatile (LOCK\_IF\_MP(%4) "cmpxchgl %1,(%3)"  
 : "=a" (exchange\_value)  
 : "r" (exchange\_value), "a" (compare\_value), "r" (dest), "r" (mp)  
 : "cc", "memory");  
 return exchange\_value;  
}

程序会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。如果程序是在多处理器上运行，就为cmpchg指令加上lock前缀；如果是在单处理器上运行，就省略lock前缀。  
Intel对lock前缀的说明如下：

* 1）确保对内存的读-改-写操作原子执行（基于总线锁或缓存锁）
* 2）禁止该指令，与 之前 和 之后 的读写指令重排序
* 3）把写缓冲区中的所有数据刷新到内存中。

# 同步容器

同步容器类都是线程安全的，但在某些情况下可能需要额外的客户端加锁来保护复合操作。容器上常见的复合操作包括：迭代，跳跃，以及条件运算。

## ConcurrentHashMap

它们提供的迭代器不会抛出ConcurrentModificationException，因此不需要再迭代过程中对容器加锁。ConcurrentHasMap返回的迭代器具有弱一致性，而并非及时失败。弱一致性的迭代器可以容忍并发的修改，当修改迭代器会遍历已有的元素，并可以在迭代器被构造后将修改操作反映给容器。

## CopyOnWriteArrayList

* 用于替代同步List，在某些情况下它提供了更好的并发性能，并且在迭代期间不需要对容器进行加锁或复制。在每次修改时，都会创建并重新发布一个新的容器副本，从而实现可变性。写入时复制容器返回的迭代器不会抛出
* ConcurrentModificationException，并且返回的元素与迭代器创建时的元素完全一致，而不必考虑之后修改操作所带来的影响。
* 显然，每当修改容器时都会复制底层数组，这需要一定的开销，特别是当容器的规模较大时，仅当迭代操作远远多于修改操作时，才应该使用写入时复制容器。

## BlockingQueue

* 阻塞队列提供了可阻塞的put和take方法，以及支持定时的offer和poll方法。如果队列已经满了，那么put方法将阻塞直到有空间可用；如果队列为空，那么take方法将会阻塞直到有元素可用。队列可以是有界的也可以是无界的，无界队列永远都不会充满，因此无界队列上的put方法也永远不会阻塞。
* 在构建高可靠的应用程序时，有界队列ArrayBlockingQueue是一种强大的资源管理工具：它们能抑制并防止产生过多的工作项，使应用程序在负荷过载的情况下变得更加健壮。

## ThreadLocal

在线程之间共享变量是存在风险的，有时可能要避免共享变量，使用ThreadLocal辅助类为各个线程提供各自的实例。  
例如有一个静态变量

public static final SimpleDateFormat sdf = new SimpleDateFormat(“yyyy-MM-dd”);

如果两个线程同时调用sdf.format(…)  
那么可能会很混乱，因为sdf使用的内部数据结构可能会被并发的访问所破坏。当然可以使用线程同步，但是开销很大；或者也可以在需要时构造一个局部SImpleDateFormat对象。但这很浪费

# 同步工具使用

## Semaphore（信号量）

* 信号量用来控制同时访问某个特定资源的操作数量，或者同时执行某个指定操作的数量。还可以用来实现某种资源池，或者对容器施加边界。
* Semaphore中管理着一组虚拟的许可permit，许可的初始数量可通过构造函数来指定。在执行操作时可以首先获得许可（只要还有剩余的许可），并在使用以后释放许可。如果没有许可，那么acquire将阻塞直到有许可（或者直到被中断或者操作超时）。release方法将返回一个许可给信号量。计算信号量的一种简化形式是二值信号量，即初始值为1的Semaphore。二值信号量可以用作互斥体，并具备不可重入的加锁语义：谁拥有这个唯一的许可，谁就拥有了互斥锁。
* 可以用于实现资源池，当池为空时，请求资源将会阻塞，直至存在资源。将资源返回给池之后会调用release释放许可。

public class BoundedHashSet<T> {  
 private final Set<T> set;  
 private final Semaphore semaphore;  
  
 public BoundedHashSet(int bound) {  
 this.set = Collections.synchronizedSet(new HashSet<T>());  
 this.semaphore = new Semaphore(bound);  
 }  
 public boolean add(T t) throws InterruptedException {  
 semaphore.acquire();  
 boolean wasAdded = false;  
 try {  
 wasAdded = set.add(t);  
 return wasAdded;  
 } finally {  
 if(!wasAdded){  
 semaphore.release();   
 }  
 }  
 }  
   
 public boolean remove(Object o){  
 boolean wasRemoved = set.remove(o);  
 if(wasRemoved){  
 semaphore.release();  
 }  
 return wasRemoved;  
 }  
   
}

## CyclicBarrier（可循环使用的屏障/栅栏）

|  |  |
| --- | --- |
| CountDownLatch | CyclicBarrier |
| 减计数方式 | 加计数方式 |
| 计算为0时释放所有等待的线 | 计数达到指定值时释放所有等待线程 |
| 计算为0时释放所有等待的线程 | 计数达到指定值时释放所有等待线程 |
| 计数为0时，无法重置 | 计数达到指定值时，计数置为0重新开始 |
| 调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响 | 调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞 |
| 不可重复利用 | 可重复利用 |

* 线程在countDown()之后，会继续执行自己的任务，而CyclicBarrier会在所有线程任务结束之后，才会进行后续任务。
* Barrier类似于闭锁，它能阻塞一组线程直到某个线程发生。栅栏与闭锁的关键区别在于，前者未达到条件时每个线程都会阻塞在await上，直至条件满足所有线程解除阻塞，后者未达到条件时countDown不会阻塞，条件满足时会解除await线程的阻塞。
* CyclicBarrier可以使一定数量的参与方反复地在栅栏位置汇集，它在并行迭代算法中非常有用；这种算法通常将一个问题拆分成一系列相互独立的子问题。当线程到达栅栏位置时将调用await方法，这个方法将阻塞直到所有线程都达到栅栏位置。如果所有线程都到达了栅栏位置，那么栅栏将打开，此时所有线程都被释放，而栅栏将被重置以便下次使用。
* 如果对await方法的调用超时，或者await阻塞的线程被中断，那么栅栏就被认为是打破了，所有阻塞的await调用都被终止并抛出BrokenBarrierException。如果成功通过栅栏，那么await将为每个线程返回一个唯一的到达索引号，我们可以利用这些索引来选举产生一个领导线程，并在下一次迭代中由该领导线程执行一些特殊的工作。CyclicBarrier还可以使你将一个栅栏操作传递给构造函数，这是一个Runnable，当成功通过栅栏时会在一个子任务线程中执行它。
* 可以用于多线程计算数据，最后合并计算结果的场景。CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset方法重置。

/\*\*  
 \* 通过CyclicBarrier协调细胞自动衍生系统中的计算  
 \*/  
public class CellularAutomata {  
 private final Board mainBoard;  
 private final CyclicBarrier cyclicBarrier;  
 private final Worker[] workers;  
   
 public CellularAutomata(Board board){  
 this.mainBoard = board;  
 int count = Runtime.getRuntime().availableProcessors();  
 this.cyclicBarrier = new CyclicBarrier(count, new Runnable() {  
 public void run() {  
 mainBoard.commitNewValues();  
 }  
 });  
 this.workers = new Worker[count];  
 for (int i = 0; i < count; i++) {  
 workers[i] = new Worker(mainBoard.getSubBoard(count,i));  
 }  
 }  
   
 private class Worker implements Runnable{  
 private final Board board;  
   
 public Worker(Board board){  
 this.board = board;  
 }  
  
 public void run() {  
 while (!board.hasConverged()) {  
 for (int x = 0; x < board.getMaxX(); x++) {  
 for (int y = 0; y < board.getMaxY(); y++) {  
 board.setNewValue(x, y, computeValue(x, y));  
 }  
 }  
 try {  
 cyclicBarrier.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 return;  
 }  
 }  
 }  
 }  
  
 private int computeValue(int x, int y) {  
 return x+y;  
 }  
  
  
 public void start(){  
 for (int i = 0; i < workers.length; i++) {  
 new Thread(workers[i]).start();  
 }  
 mainBoard.waitForConvergence();  
 }  
}

## Exchanger（两个线程交换数据）

* 另一种形式的栅栏是Exchanger，它是一种两方栅栏，各方在栅栏位置上交换数据。当两方执行不对称的操作时，Exchanger会非常有用。
* Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据， 如果第一个线程先执行exchange方法，它会一直等待第二个线程也执行exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

public class TestExchanger {  
  
 private Exchanger<String> exchanger = new Exchanger<String>();  
  
 private ExecutorService threadPool = Executors.newFixedThreadPool(2);  
  
 public void start() {  
 threadPool.execute(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 String A = "银行流水A";// A录入银行流水数据  
 exchanger.exchange(A);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
   
 threadPool.execute(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 String B = "银行流水B";// B录入银行流水数据  
 String A = exchanger.exchange("B");  
 System.out.println("A和B数据是否一致：" + A.equals(B) + ",A录入的是："  
 + A + ",B录入是：" + B);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
 threadPool.shutdown();  
 }  
  
 public static void main(String[] args) {  
 new TestExchanger().start();  
 }  
}

## CountDownLatch（闭锁）

* 闭锁可以延迟线程的进度直到其达到终止状态。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭的，并且没有任何线程能通过，当到达结束状态时，这扇门会打开并允许所有的线程通过。当闭锁达到结束状态后，将不会再改变状态，因此这扇门将永远保持打开状态。闭锁可以用来确保某些活动指导其他活动都完成后才继续执行。
* 闭锁状态包括一个计数器，该计数器被初始化为一个正数，表示需要等待的事件数量。countDown方法递减计数器，表示有一个事件发生了，而await方法等待计数器达到0，这表示所有需要等待的事件都已经发生。如果计数器的值非零，那么await会一直阻塞直到计数器为0，或者等待中的线程中断，或者等待超时。

public class TestCountDownLatch {  
 public static void main(String[] args) {  
 CountDownLatch latch = new CountDownLatch(5);  
 LatchDemo latchDemo = new LatchDemo(latch);  
 long begin = System.currentTimeMillis();  
 for (int i = 0; i < 5; ++i) {  
 new Thread(latchDemo).start();  
 }  
 try {  
 latch.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 long end = System.currentTimeMillis();  
 System.out.println("总计耗时：" + (end - begin));  
 }  
}  
  
class LatchDemo implements Runnable {  
 private CountDownLatch latch;  
  
 public LatchDemo(CountDownLatch latch) {  
 this.latch = latch;  
 }  
  
 @Override  
 public void run() {  
 try {  
 for (int i = 0; i < 50000; i++) {  
 if (i % 2 == 0) {  
 System.out.println(i);  
 }  
 }  
 } finally {  
 latch.countDown();  
 }  
 }  
}

## FutureTask（Future实现类）

* FurureTask是Future接口的唯一实现类。
* FutureTask表示的计算是通过Callable来实现的，相当于一种可生成结果的Runnable，并且可以处于以下3种状态：等待运行、正在运行和运行完成。
* Future.get方法的行为取决于任务的状态。如果任务已经完成，那么get会立即返回结果，否则会阻塞直到任务进入完成状态，然后返回结果或者抛出异常。- --- -- FutureTask将计算结果从执行计算的线程传递到获取这个结果的线程，而FutureTask的规范确保了这种传递过程能实现结果的安全发布。
* Callable表示的任务可以抛出受检查的或未受检查的异常，并且任何代码都可能抛出一个Error。无论任务代码抛出什么异常，都会被封装到一个ExecutionException中，并在future.get中被重新抛出。
* 当get方法抛出ExecutionException，可能是以下三种情况之一：Callable抛出的受检查异常，RuntimeException，以及Error。

### Future

Future接口设计初衷是对将来某个时刻会发生的结果进行建模。它建模了一种异步计算，返回一个执行运算结果的引用，当运算结束后，这个引用被返回给调用方。 在Future中触发那些潜在耗时的操作把调用线程解放出来，让它能继续执行其他有价值的工作，不再需要等待耗时的操作完成。

**示例**

public void future() {  
 ExecutorService executor = Executors.newCachedThreadPool();  
 Future<Double> future = executor.submit(new Callable<Double>() {  
 @Override  
 public Double call() throws Exception {  
 return doSomethingComputation();  
 }  
 });  
 // 在另一个线程执行耗时操作的同时，去执行一些其他的任务。  
 // 这些任务不依赖于future的结果，可以与future并发执行。  
 // 如果下面的任务马上依赖于future的结果，那异步操作是没有意义的。  
 doSomethingElse();  
 try {  
 // 如果不设置超时时间，那么线程会阻塞在这里。  
 Double result = future.get(1, TimeUnit.SECONDS);  
 System.out.println("result is " + result);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 } catch (TimeoutException e) {  
 e.printStackTrace();  
 }  
}  
  
private void doSomethingElse() {  
 System.out.println("doSomethingElse");  
}  
  
private double doSomethingComputation() {  
 System.out.println("doSomethingComputation");  
 return 0.1;  
}

**局限性**

Future无法实现以下的功能。

* 1) 将两个异步操作计算合并为一个——这两个异步计算之间相互独立，同时第二个又依赖于第一个的记过
* 2）等待Future集合中的所有任务都完成
* 3）仅等待Future集合中最快结束的任务完成，并返回它的结果
* 4）通过编程方式完成一个Future任务的执行（以手工设定异步操作结果）
* 5）应对Future的完成事件（完成回调）

## CompletableFuture

**实现异步API（将任务交给另一线程完成，该线程与调用方异步，通过回调函数或阻塞的方式取得任务结果）**

**1）Shop**

public class Shop {  
 private ThreadLocalRandom random;  
 private ExecutorService executorService = Executors.newCachedThreadPool();  
   
 public Future<Double> getPriceAsync(String product){  
 CompletableFuture<Double> future = new CompletableFuture<>();  
 // 另一个线程计算  
 executorService.submit(() -> {  
 try {  
 double price = calculatePrice(product);  
 future.complete(price);  
 } catch (Exception e) {  
 // 处理异常  
 future.completeExceptionally(e);  
 e.printStackTrace();  
 }  
 });  
 return future;  
 }  
   
 private double calculatePrice(String product){  
 random = ThreadLocalRandom.current();  
 // 模拟耗时操作  
 delay();  
 // 随机  
 return random.nextDouble() \* product.charAt(0) + product.charAt(1);  
 }  
   
 public static void delay(){  
 try {  
 Thread.sleep(1000);  
 throw new RuntimeException("product is not available");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void main(String[] args) {  
 Shop shop = new Shop();  
  
 Future<Double> price = shop.getPriceAsync("my favorite product");  
 // 计算price和doSomethingElse是并发执行的  
 doSomethingElse();  
 try {  
 // 如果此时已经计算完毕，则立即返回；如果没有计算完毕，则会阻塞  
 Double result = price.get();  
 System.out.println("result is " + result);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
   
 }  
  
 private static void doSomethingElse() {  
 System.out.println("doSomethingElse");  
 }  
}

**2) GracefulShop**

工厂方法创建的Future自己内部维护了一个线程池。  
public class GracefulShop {  
 private ThreadLocalRandom random;  
 public Future<Double> getPriceAsync(String product){  
 // 接收一个Supplier，该Supplier会交由ForkJoinPool池中的某个执行线程执行  
 return CompletableFuture.supplyAsync(() -> calculatePrice(product));  
 }  
   
 private double calculatePrice(String product){  
 random = ThreadLocalRandom.current();  
 // 模拟耗时操作  
 delay();  
 // 随机  
 return random.nextDouble() \* product.charAt(0) + product.charAt(1);  
 }  
   
 public static void delay(){  
 try {  
 Thread.sleep(1000);  
 throw new RuntimeException("product is not available");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void main(String[] args) {  
 GracefulShop shop = new GracefulShop();  
  
 Future<Double> price = shop.getPriceAsync("my favorite product");  
 // 计算price和doSomethingElse是并发执行的  
 doSomethingElse();  
 try {  
 // 如果此时已经计算完毕，则立即返回；如果没有计算完毕，则会阻塞  
 Double result = price.get();  
 System.out.println("result is " + result);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
   
 }  
  
 private static void doSomethingElse() {  
 System.out.println("doSomethingElse");  
 }  
}

### 将批量同步操作转为异步操作（并行流/CompletableFuture）

如果原本的getPrice是同步方法的话，那么如果想批量调用getPrice，提高效率的方法要么使用并行流，要么使用CompletableFuture。

public class SyncShop {  
 private String name;  
  
 public SyncShop(String name) {  
 this.name = name;  
 }  
  
 public static void delay() {  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
   
 public double getPrice(String product) {  
 ThreadLocalRandom random = ThreadLocalRandom.current();  
 delay();  
 return random.nextDouble() \* product.charAt(0) + product.charAt(1);  
 }  
  
 public String getName() {  
 return name;  
 }  
}  
  
public class BestProductPriceCalculator {  
 private List<SyncShop> shops = Arrays.asList(  
 new SyncShop("BestPrice"),  
 new SyncShop("LetsSaveBig"),  
 new SyncShop("MyFavoriteShop"),  
 new SyncShop("BuyItAll")  
 );  
  
 public List<String> findPricesWithParallelStream(String product) {  
 return shops  
 .parallelStream()  
 .map(shop -> shop.getName() + ":" + shop.getPrice(product))  
 .collect(Collectors.toList());  
 }  
  
 public List<String> findPricesWithCompletableFuture(String product) {  
 List<CompletableFuture<String>> futures = shops  
 .stream()  
 .map(shop -> CompletableFuture.supplyAsync(() -> shop.getName() + ":" + shop.getPrice(product)))  
 .collect(Collectors.toList());  
 // join方法和Future的get方法有相同的含义，并且也声明在Future接口中，它们唯一的不同就是join不会抛出任何检测到的异常。  
 return futures.stream().map(CompletableFuture::join).collect(Collectors.toList());  
 }  
}  
  
  
public class FutureTest {  
 private BestProductPriceCalculator calculator = new BestProductPriceCalculator();  
 // 1s   
 @Test  
 public void testParallelStream(){  
 calculator.findPricesWithParallelStream("my favorite product");  
 }  
 //2s   
 @Test  
 public void testCompletableFuture(){  
 calculator.findPricesWithCompletableFuture("my favorite product");   
 }  
}

使用并行流还是CompletableFuture？  
前者是无法调整线程池的大小的（处理器个数），而后者可以。  
如果是计算密集型应用，且没有IO，那么推荐使用并行流  
如果是IO密集型，需要等待IO，那么使用CompletableFuture灵活性更高，比如根据《Java并发编程实战》中给出的公式计算线程池合适的大小。

### 多个异步任务合并

逻辑如下：  
从每个商店获取price，price以某种格式返回。拿到price后解析price，然后调用远程API根据折扣计算最终price。  
可以分为三个任务，每个商店都要执行这三个任务。

public class PipelineShop {  
 private String name;  
  
 public PipelineShop(String name) {  
 this.name = name;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getPrice(String product) {  
 ThreadLocalRandom random = ThreadLocalRandom.current();  
 double price = calculatePrice(product);  
 Discount.DiscountCode code = Discount.DiscountCode.values()[random.nextInt(Discount.DiscountCode.values().length)];  
 return String.format("%s:%.2f:%s", name, price, code);  
 }  
   
 private double calculatePrice(String product) {  
 ThreadLocalRandom random = ThreadLocalRandom.current();  
 // 模拟耗时操作  
 delay();  
 // 随机  
 return random.nextDouble() \* product.charAt(0) + product.charAt(1);  
 }  
  
 public static void delay() {  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
  
public class Discount {  
 public enum DiscountCode{  
 NONE(0),SILVER(5),GOLD(10),PLATINUM(15),DIAMOND(20);  
 private int percentage;  
 DiscountCode(int percentage){  
 this.percentage = percentage;   
 }  
 }  
   
 public static String applyDiscount(Quote quote){  
 return quote.getShopName()+ " price is " + apply(quote.getPrice(), quote.getDiscountCode());  
 }  
  
 private static double apply(double price, DiscountCode discountCode) {  
 // 模拟调用远程服务的延迟  
 delay();  
 return price \* ( 100 - discountCode.percentage ) / 100 ;   
 }  
  
}  
  
public class Quote {  
 private String shopName;  
 private double price;  
 private Discount.DiscountCode discountCode;  
  
 public Quote(String shopName, double price, Discount.DiscountCode discountCode) {  
 this.shopName = shopName;  
 this.price = price;  
 this.discountCode = discountCode;  
 }  
   
 public static Quote parse(String str){  
 String [] slices = str.split(":");  
 return new Quote(slices[0],Double.parseDouble(slices[1]),Discount.DiscountCode.valueOf(slices[2]));  
 }  
  
 public String getShopName() {  
 return shopName;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public Discount.DiscountCode getDiscountCode() {  
 return discountCode;  
 }  
}  
  
  
public class BestProductPriceWithDiscountCalculator {  
 private List<PipelineShop> shops = Arrays.asList(  
 new PipelineShop("BestPrice"),  
 new PipelineShop("LetsSaveBig"),  
 new PipelineShop("MyFavoriteShop"),  
 new PipelineShop("BuyItAll")  
 );  
  
 public List<String> findPricesWithPipeline(String product) {  
 List<CompletableFuture<String>> futures = shops  
 .stream()  
 .map(shop -> CompletableFuture.supplyAsync(() -> shop.getPrice(product)))  
 .map(future -> future.thenApply(Quote::parse))  
 .map(future -> future.thenCompose(  
 quote -> CompletableFuture.supplyAsync(  
 () -> Discount.applyDiscount(quote)  
 )  
 ))  
 .collect(Collectors.toList());  
 return futures.stream().map(CompletableFuture::join).collect(Collectors.toList());  
 }  
}

### 回调

public class CallbackBestProductPriceCalculator {  
 private List<PipelineShop> shops = Arrays.asList(  
 new PipelineShop("BestPrice"),  
 new PipelineShop("LetsSaveBig"),  
 new PipelineShop("MyFavoriteShop"),  
 new PipelineShop("BuyItAll")  
 );  
  
 public Stream<CompletableFuture<String>> findPricesStream(String product) {  
 return shops  
 .stream()  
 .map(shop -> CompletableFuture.supplyAsync(() -> shop.getPrice(product)))  
 .map(future -> future.thenApply(Quote::parse))  
 .map(future -> future.thenCompose(  
 quote -> CompletableFuture.supplyAsync(  
 () -> Discount.applyDiscount(quote)  
 )  
 ));  
 }  
}  
  
  
@Test  
public void testCallback(){  
 CompletableFuture[] futures = callbackBestProductPriceCalculator.findPricesStream("my favorite product").map(  
 future -> future.thenAccept(System.out::println)  
 ).toArray(size -> new CompletableFuture[size]);  
 CompletableFuture.allOf(futures).join();  
}

### API

CompletableFuture类实现了CompletionStage和Future接口。Future是Java 5添加的类，用来描述一个异步计算的结果，但是获取一个结果时方法较少,要么通过轮询isDone，确认完成后，调用get()获取值，要么调用get()设置一个超时时间。但是这个get()方法会阻塞住调用线程，这种阻塞的方式显然和我们的异步编程的初衷相违背。  
为了解决这个问题，JDK吸收了guava的设计思想，加入了Future的诸多扩展功能形成了CompletableFuture。

CompletionStage是一个接口，从命名上看得知是一个完成的阶段，它里面的方法也标明是在某个运行阶段得到了结果之后要做的事情。、

#### supplyAsync 提交任务

public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier);  
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor);

#### thenApply 变换（等待前一个任务返回后执行，处于同一个CompletableFuture）

public <U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn);  
public <U> CompletionStage<U> thenApplyAsync(Function<? super T,? extends U> fn);  
public <U> CompletionStage<U> thenApplyAsync(Function<? super T,? extends U> fn,Executor executor);

首先说明一下以Async结尾的方法都是可以异步执行的，如果指定了线程池，会在指定的线程池中执行，如果没有指定，默认会在ForkJoinPool.commonPool()中执行，下文中将会有好多类似的，都不详细解释了。关键的入参只有一个Function，它是函数式接口，所以使用Lambda表示起来会更加优雅。它的入参是上一个阶段计算后的结果，返回值是经过转化后结果。  
不带Async的方法会在和前一个任务相同的线程中处理；  
以Async的方法会将任务提交到一个线程池，所有每个任务是由不同的线程处理的。

public void thenApply() {  
 String result = CompletableFuture.supplyAsync(() -> "hello").thenApply(s -> s + " world").join();  
 System.out.println(result);  
}

#### thenAccept 消耗

public CompletionStage<Void> thenAccept(Consumer<? super T> action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action,Executor executor);  
  
public void thenAccept() {  
 CompletableFuture.supplyAsync(() -> "hello").thenAccept(s -> System.out.println(s + " world"));  
}

#### thenRun 执行下一步操作，不关心上一步结果

public CompletionStage<Void> thenRun(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action,Executor executor);  
  
thenRun它的入参是一个Runnable的实例，表示当得到上一步的结果时的操作。  
public void thenRun() {  
 CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "hello";  
 }).thenRun(() -> System.out.println("hello world"));  
}

#### thenCombine 结合两个CompletionStage的结果，进行转化后返回

public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn);  
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn);  
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn,Executor executor);

它需要原来的处理返回值，并且other代表的CompletionStage也要返回值之后，利用这两个返回值，进行转换后返回指定类型的值。

public void thenCombine() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "hello";  
 }).thenCombine(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "world";  
 }), (s1, s2) -> s1 + " " + s2).join();  
 System.out.println(result);  
}

#### thenCompose（合并多个CompletableFuture，流水线执行，在调用外部接口返回CompletableFuture类型时更方便）

<U> CompletableFuture<U> thenCompose(Function<? super T,CompletableFuture<U>> fn);

* thenCompose方法允许对两个异步操作（supplyAsync）进行流水线，第一个操作完成时，将其结果作为参数传递给第二个操作。
* 创建两个CompletableFuture，对第一个CompletableFuture对象调用thenCompose，并向其传递一个函数。当第一个CompletableFuture执行完毕后，它的结果将作为该函数的参数，这个函数的返回值是以第一个CompletableFuture的返回做输入计算出的第二个CompletableFuture对象。

#### thenAccptBoth 结合两个CompletionStage的结果，进行消耗

public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U> other,BiConsumer<? super T, ? super U> action);  
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends U> other,BiConsumer<? super T, ? super U> action);  
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends U> other,BiConsumer<? super T, ? super U> action, Executor executor);

它需要原来的处理返回值，并且other代表的CompletionStage也要返回值之后，利用这两个返回值，进行消耗。

public void thenAcceptBoth() {  
 CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "hello";  
 }).thenAcceptBoth(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "world";  
 }), (s1, s2) -> System.out.println(s1 + " " + s2));  
}

#### runAfterBoth 在两个CompletionStage都运行完执行，不关心上一步结果

public CompletionStage<Void> runAfterBoth(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,Runnable action,Executor executor);

不关心这两个CompletionStage的结果，只关心这两个CompletionStage执行完毕，之后在进行操作（Runnable）。

public void runAfterBoth() {  
 CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s1";  
 }).runAfterBothAsync(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s2";  
 }), () -> System.out.println("hello world"));  
}

#### applyToEither 两个CompletionStage，谁计算的快，我就用那个CompletionStage的结果进行下一步的转化操作

public <U> CompletionStage<U> applyToEither(CompletionStage<? extends T> other,Function<? super T, U> fn);  
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T> other,Function<? super T, U> fn);  
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T> other,Function<? super T, U> fn,Executor executor);

我们现实开发场景中，总会碰到有两种渠道完成同一个事情，所以就可以调用这个方法，找一个最快的结果进行处理。

public void applyToEither() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s1";  
 }).applyToEither(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "hello world";  
 }), s -> s).join();  
 System.out.println(result);  
}

#### acceptEither 两个CompletionStage，谁计算的快，我就用那个CompletionStage的结果进行下一步的消耗操作

public CompletionStage<Void> acceptEither(CompletionStage<? extends T> other,Consumer<? super T> action);  
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T> other,Consumer<? super T> action);  
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T> other,Consumer<? super T> action,Executor executor);  
  
public void acceptEither() {  
 CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s1";  
 }).acceptEither(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "hello world";  
 }), System.out::println);  
 while (true) {  
 }  
}

#### runAfterEither 两个CompletionStage，任何一个完成了都会执行下一步的操作，不关心上一步结果

public CompletionStage<Void> runAfterEither(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other,Runnable action,Executor executor);  
public void runAfterEither() {  
 CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s1";  
 }).runAfterEither(CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s2";  
 }), () -> System.out.println("hello world"));  
}

#### exceptionally 当运行时出现了异常，可以进行补偿

public CompletionStage<T> exceptionally(Function<Throwable, ? extends T> fn);  
public void exceptionally() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 if (1 == 1) {  
 throw new RuntimeException("测试一下异常情况");  
 }  
 return "s1";  
 }).exceptionally(e -> {  
 System.out.println(e.getMessage());  
 return "hello world";  
 }).join();  
 System.out.println(result);  
}

#### whenComplete 当运行完成时，若有异常则改变返回值，否则返回原值

public CompletionStage<T> whenComplete(BiConsumer<? super T, ? super Throwable> action);  
public CompletionStage<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action);  
public CompletionStage<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action,Executor executor);  
  
public void whenComplete() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 if (1 == 1) {  
 throw new RuntimeException("测试一下异常情况");  
 }  
 return "s1";  
 }).whenComplete((s, t) -> {  
 System.out.println(s);  
 System.out.println(t.getMessage());  
 }).exceptionally(e -> {  
 System.out.println(e.getMessage());  
 return "hello world";  
 }).join();  
 System.out.println(result);  
}

null  
java.lang.RuntimeException: 测试一下异常情况  
java.lang.RuntimeException: 测试一下异常情况  
hello world  
这里也可以看出，如果使用了exceptionally，就会对最终的结果产生影响，它无法影响如果没有异常时返回的正确的值，这也就引出下面我们要介绍的handle。

#### handle 当运行完成时，无论有无异常均可转换

public <U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends U> fn);  
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn);  
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn,Executor executor);  
  
public void handle() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //出现异常  
 if (1 == 1) {  
 throw new RuntimeException("测试一下异常情况");  
 }  
 return "s1";  
 }).handle((s, t) -> {  
 if (t != null) {  
 return "hello world";  
 }  
 return s;  
 }).join();  
 System.out.println(result);  
}

hello world

public void handle() {  
 String result = CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "s1";  
 }).handle((s, t) -> {  
 if (t != null) {  
 return "hello world";  
 }  
 return s;  
 }).join();  
 System.out.println(result);  
}

s1

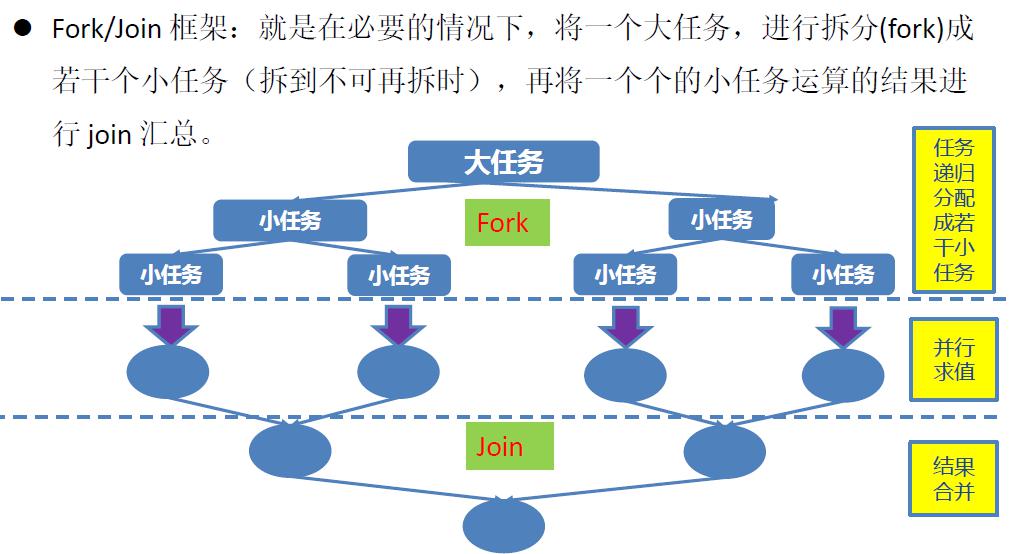
#### allOf

allOf工厂方法接收一个由CompletableFuture构成的数组，数组中的所有CompletableFuture对象执行完成之后，它返回一个CompletableFuture对象。这意味着，如果你需要等待最初Stream中的所有CompletableFuture对象执行完毕，对allOf方法返回的CompletableFuture执行join操作是个不错的注意。

#### anyOf

只要CompletableFuture对象数组中有一个执行完毕，便不再等待。

## ForkJoin

双端队列LinkedBlockingDeque适用于另一种相关模式，即工作密取（work stealing）。在生产者——消费者设计中，所有消费者有一个共享的工作队列，而在工作密取设计中，每个消费者都有各自的双端队列。如果一个消费者完成了自己双端队列中的全部工作，那么它可以从其他消费者双端队列头部秘密地获取工作。密取工作模式比传统的生产者——消费者模式具有更高的可伸缩性，这是因为工作者线程不会在单个共享的任务队列上发生竞争。在大多数时候，它们都只是访问自己的双端队列，从而极大地减少了竞争。当工作者线程需要访问另一个队列时，它会从队列的头部而不是从尾部获取工作，因此进一步降低了队列上的竞争程度。  
  


* 第一步分割任务。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，所以还需要不停的分割，直到分割出的子任务足够小。
* 第二步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。

Fork/Join使用两个类来完成以上两件事情：

* ForkJoinTask：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承- ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类：
  + oRecursiveAction：用于没有返回结果的任务。
  + oRecursiveTask ：用于有返回结果的任务。
* ForkJoinPool ：ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

threshold 临界值

RecursiveTask有两个方法：fork和join  
fork是执行子任务，join是取得子任务的结果，用于合并

public class TestForkJoin {  
 public static void main(String[] args) throws InterruptedException, ExecutionException {  
 ForkJoinPool pool = new ForkJoinPool();  
 ForkJoinCalculator calculator = new ForkJoinCalculator(0, 10000000L);  
 Long result = pool.invoke(calculator);  
 System.out.println(result);  
 pool.shutdown();  
 }  
}  
  
class ForkJoinCalculator extends RecursiveTask<Long> {  
  
 private static final long serialVersionUID = -6682191224530210391L;  
  
 private long start;  
 private long end;  
 private static final long THRESHOLD = 10000L;  
  
 public ForkJoinCalculator(long start, long end) {  
 this.start = start;  
 this.end = end;  
 }  
  
 @Override  
 protected Long compute() {  
 long length = end - start;  
 if (length < THRESHOLD) {  
 long sum = 0L;  
 for (long i = start; i < end; ++i) {  
 sum += i;  
 }  
 return sum;  
 } else {  
 long middle = (start + end) / 2;  
 ForkJoinCalculator left = new ForkJoinCalculator(start, middle);  
 left.fork();  
 ForkJoinCalculator right = new ForkJoinCalculator(middle, end);  
 right.fork();  
 return left.join() + right.join();  
 }  
 }  
}

### 原理浅析

* 1. 每个Worker线程都维护一个任务队列，即ForkJoinWorkerThread中的任务队列。
* . 任务队列是双向队列，这样可以同时实现LIFO和FIFO。
* . 子任务会被加入到原先任务所在Worker线程的任务队列。
  1. Worker线程用LIFO的方法取出任务，也就后进队列的任务先取出来（子任务总是后加入队列，但是需要先执行）。
  2. Worker线程的任务队列为空，会随机从其他的线程的任务队列中拿走一个任务执行（所谓偷任务：steal work，FIFO的方式）。
  3. 如果一个Worker线程遇到了join操作，而这时候正在处理其他任务，会等到这个任务结束。否则直接返回。
  4. 如果一个Worker线程偷任务失败，它会用yield或者sleep之类的方法休息一会儿，再尝试偷任务（如果所有线程都是空闲状态，即没有任务运行，那么该线程也会进入阻塞状态等待新任务的到来）。

### 与MapReduce的区别

MapReduce是把大数据集切分成小数据集，并行分布计算后再合并。

ForkJoin是将一个问题递归分解成子问题，再将子问题并行运算后合并结果。

二者共同点：都是用于执行并行任务的。基本思想都是把问题分解为一个个子问题分别计算，再合并结果。应该说并行计算都是这种思想，彼此独立的或可分解的。从名字上看Fork和Map都有切分的意思，Join和Reduce都有合并的意思，比较类似。

区别：

* 1）环境差异，分布式 vs 单机多核：ForkJoin设计初衷针对单机多核（处理器数量很多的情况）。MapReduce一开始就明确是针对很多机器组成的集群环境的。也就是说一个是想充分利用多处理器，而另一个是想充分利用很多机器做分布式计算。这是两种不同的的应用场景，有很多差异，因此在细的编程模式方面有很多不同。
* 2）编程差异：MapReduce一般是：做较大粒度的切分，一开始就先切分好任务然后再执行，并且彼此间在最后合并之前不需要通信。这样可伸缩性更好，适合解决巨大的问题，但限制也更多。ForkJoin可以是较小粒度的切分，任务自己知道该如何切分自己，递归地切分到一组合适大小的子任务来执行，因为是一个JVM内，所以彼此间通信是很容易的，更像是传统编程方式。

# 线程池使用

## 引入原因

* 1）任务处理过程从主线程中分离出来，使得主循环能够更快地重新等待下一个到来的连接，使得任务在完成前面的请求之前可以接受新的请求，从而提高响应性。
* 2）任务可以并行处理，从而能同时服务多个请求。如果有多个处理器，或者任务由于某种原因被阻塞，程序的吞吐量将得到提高。
* 3）任务处理代码必须是线程安全的，因为当有多个任务时会并发地调用这段代码。

**无限制创建线程的不足：**

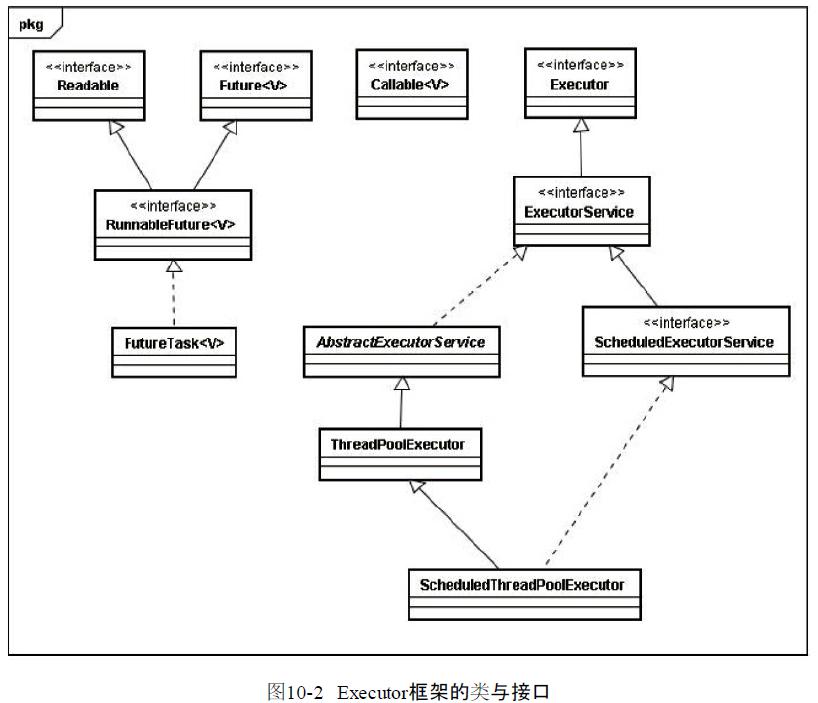
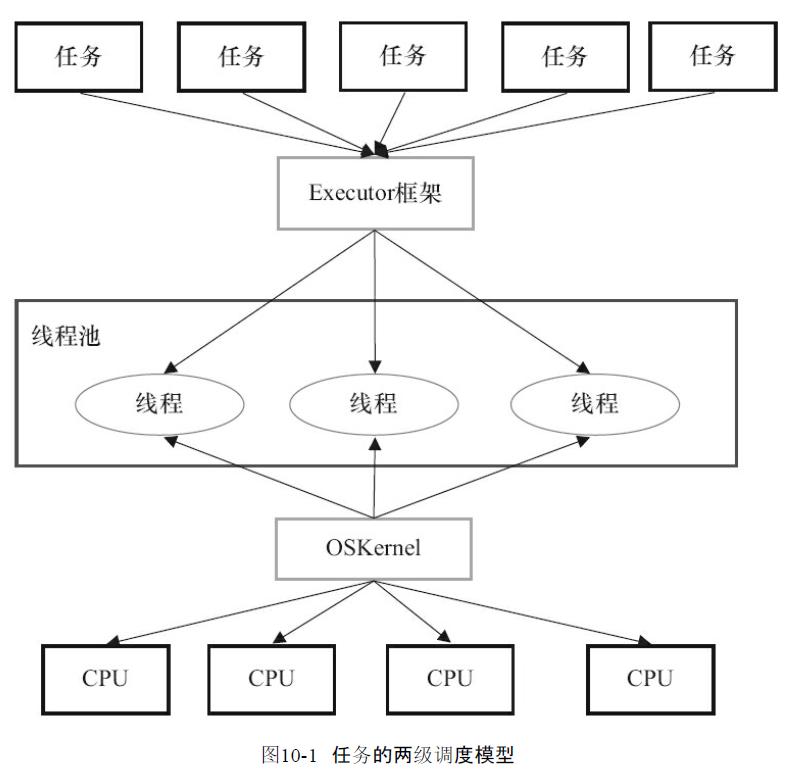
* 1）线程生命周期的开销非常高  
  2）资源消耗
* 3）稳定性

解决方式：线程池 Executor框架

使用线程池的好处：

* 1）降低资源消耗
* 2）提高响应速度
* 3）提高线程的可管理性

## Executor ExecutorService ScheduledExecutorService

继承体系  
  
  


### ExecutorService



### ScheduledExecutorService



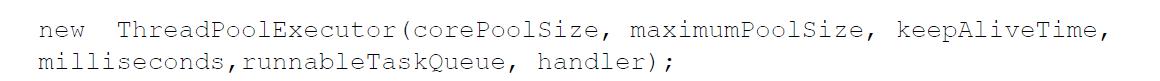
返回值



示例

public class QuoteTask implements Callable<TravelQuote> {  
 private final TravelCompany company;  
 private final TravelInfo travelInfo;  
 private ExecutorService exec;  
 public QuoteTask(TravelCompany company, TravelInfo travelInfo) {  
 this.company = company;  
 this.travelInfo = travelInfo;  
 }  
   
 public TravelQuote call() throws Exception {  
 return company.solicitQuote(travelInfo);  
 }  
   
 public List<TravelQuote> getRankedTravelQuotes(TravelInfo travelInfo, Set<TravelCompany> companies, Comparator<TravelQuote> ranking, long time, TimeUnit unit) throws InterruptedException {  
 //任务  
 List<QuoteTask> tasks = new ArrayList<QuoteTask>();  
 for (TravelCompany company : companies) {  
 tasks.add(new QuoteTask(company,travelInfo));  
 }  
 //执行  
 List<Future<TravelQuote>> futures = exec.invokeAll(tasks,time,unit);  
 List<TravelQuote> quotes = new ArrayList<TravelQuote>(tasks.size());  
 Iterator<QuoteTask> taskIterator = tasks.iterator();  
 //取出结果  
 for(Future<TravelQuote> future:futures){  
 QuoteTask task = taskIterator.next();  
 try {  
 quotes.add(future.get());  
 } catch (ExecutionException e) {  
 quotes.add(task.getFailureQuote(e.getCause()));  
 e.printStackTrace();  
 }catch(CancellationException e){  
 quotes.add(task.getTimeOutQuote(e));  
 }  
 }  
 Collections.sort(quotes,ranking);  
 return quotes;  
 }  
}

## ThreadPoolExecutor

创建线程池  


线程动态变化

* 1.当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。
* 2.当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行
* 3.当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务
* 4.当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理
* 5.当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程
* 6.当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

创建一个线程池时需要以下几个参数：

* 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。如果调用线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。
* 2）runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列：
  + a）ArrayBlockingQueue：基于数组的有界阻塞队列，FIFO
  + b) LinkedBlockingQueue：基于链表的无界阻塞队列，FIFO，吞吐量高于ArrayBlockingQueue，Executors.newFixedThreadPoll()使用了这个队列
  + c）SynchronousQueue：一个只存储一个元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入一直处于阻塞状态，吞吐量高于LinkedBlockingQueue，Executors#newCachedThreadPoll()使用了这个队列
  + d）PriorityBlockingQueue：具有优先级的无界阻塞队列
* 3）maximumPoolSize（线程池的最大数量）：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。如果使用了无界队列该参数就没有意义了。
* 4）ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
* 5）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，或者当线程池已关闭时，会采用一种策略处理提交的新任务。这个策略默认是AbortPolicy，表示无法处理新任务时抛出异常。有以下四种饱和策略：
  + a）AbortPolicy：直接抛出异常
  + b）CallerRunsPolicy：使用调用者所在线程来运行任务
  + c）DiscardOldestPolicy：丢弃队列中最近的一个任务，并执行当前任务
  + d）DiscardPolicy：不处理，直接丢弃  
    也可以自定义饱和策略。
* 6）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。出现timeout情况下，而且线程数超过了核心线程数，会销毁销毁线程。保持在corePoolSize数。除非设置了allowCoreThreadTimeOut和超时时间，这种情况线程数可能减少到0，最大可能是Integer.MAX\_VALUE。  
  如果任务很多，每个任务执行的时间比较短，可以调大时间，提高线程的利用率。
  + allowCoreThreadTimeOut为true该值为true，则线程池数量最后销毁到0个。
  + allowCoreThreadTimeOut为false销毁机制：超过核心线程数时，而且（超过最大值或者timeout过），就会销毁。
* 7）TimeUnit（线程活动保持时间的单位）

### 使用注意

* 1、只有当任务都是同类型并且相互独立时，线程池的性能才能达到最佳。如果将运行时间较长的与运行时间较短的任务混合在一起，那么除非线程池很大，否则将可能造成拥塞。如果提交的任务依赖于其他任务，那么除非线程池无限大，否则将可能造成死锁。幸运的是，在基于网络的典型服务器应用程序中——web服务器、邮件服务器、文件服务器等，它们的请求通常都是同类型的并且相互独立的。
* 2、设置线程池的大小：  
  基于Runtime.getRuntime().avialableprocessors() 进行动态计算  
  对于计算密集型的任务，在N个处理器的系统上，当线程池为N+1时，通过能实现最优的利用率（缺页故障等暂停时额外的线程也能确保CPU时钟周期不被浪费）。  
  对于包含IO操作或者其他阻塞操作的任务，由于线程并不会一直执行，因此线程池的规模应该更大，比如2*N。要正确地设置线程池的大小，你必须估算出任务的等待时间与计算时间的比值。线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。这种估算不需要很精确，而且可以通过一些分析或监控工具来获得。你还可以通过另一种方法来调节线程池的大小：在某个基准负载下，分别设置不同大小的线程池来运行应用程序，并观察CPU利用率。*  
  *最佳线程数目 = （线程等待时间与线程计算时间之比 + 1）* CPU数目
* 3、线程的创建与销毁  
  基本大小也就是线程池的目标大小，即在没有任务执行时线程池的大小，并且只有在工作队列满了的情况下才会创建超出这个数量的线程。线程池的最大大小表示可同时活动的线程数量的上限。如果某个线程的空闲时间超过了存活时间，那么将被标记为可回收的，并且当线程池的当前大小超过了基本大小时，这个线程将被终止。
* 4、管理队列任务  
  ThreadPoolExecutor允许提供一个BlockingQueue来保存等待执行的任务。基本的任务排队方法有3种：无界队列、有界队列和同步移交。  
  一种稳妥的资源管理策略是使用有界队列，有界队列有助于避免资源耗尽的情况发生，但又带来了新的问题：当队列填满后，新的任务该怎么办？
* 5、饱和策略  
  当有界队列被填满后，饱和策略开始发挥作用。ThreadPoolExecutor的饱和策略可以通过setRejectedExecutionHandler来修改。JDK提供了几种不同的RejectedExecutionHandler的实现，每种实现都包含有不同的饱和策略：AbortPolicy、CallerRunsPolicy、DiscardPolicy、DiscardOldestPolicy。
  + 1）中止策略是默认的饱和策略，该策略将抛出未检查的RejectedExecutionException。调用者可以捕获这个异常，然后根据需求编写自己的处理代码。
  + 2）当新提交的任务无法保存到队列中执行时，抛弃策略会悄悄抛弃该任务。
  + 3）抛弃最旧的策略则会抛弃下一个将被执行的任务，然后尝试重新提交下一个将被执行的任务（如果工作队列是一个优先级队列，那么抛弃最旧的将抛弃优先级最高的任务）
  + 4）调用者运行策略实现了一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将某些任务回退给调用者，从而降低新任务的流量。它不会在线程池的某个线程中执行新提交的任务，而是在一个调用了execute的线程中执行该任务。为什么好？因为当服务器过载时，这种过载情况会逐渐向外蔓延开来——从线程池到工作队列到应用程序再到TCP层，最终达到客户端，导致服务器在高负载下实现一种平缓的性能降低。
* 6、线程工厂  
  在许多情况下都需要使用定制的线程工厂方法。例如，你希望为线程池中的线程指定一个UncaughtExceptionHandler，或者实例化一个定制的Thread类用于执行调试信息的记录，你还可能希望修改线程的优先级（虽然不提倡这样做），或者只是给线程取一个更有意义的名字，用来解释线程的转储信息和错误日志。
* 7、在调用构造函数后再定制ThreadPoolExecutor

### 扩展ThreadPoolExecutor

public class TimingThreadPool extends ThreadPoolExecutor {  
 public TimingThreadPool(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {  
 super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);  
 }  
 private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();  
 private final Logger log = Logger.getLogger("TimingThreadPool");  
 private final AtomicLong numTasks = new AtomicLong();  
 private final AtomicLong totalTime = new AtomicLong();  
  
 @Override  
 protected void beforeExecute(Thread t, Runnable r) {  
 super.beforeExecute(t,r);  
 log.fine(String.format("Thread %s :start %s",t,r));  
 startTime.set(System.nanoTime());  
 }  
  
 @Override  
 protected void afterExecute(Runnable r, Throwable t) {  
 try{  
 long endTime = System.nanoTime();  
 long taskTime = endTime - startTime.get();  
 numTasks.incrementAndGet();  
 totalTime.addAndGet(taskTime);  
 log.fine(String.format("Thread %s : end %s ,time = %dns",t,r,taskTime));  
 }finally {  
 super.afterExecute(r, t);  
 }  
 }  
  
 @Override  
 protected void terminated() {  
 try {  
 log.info(String.format("Terminated : avg time = %dns",totalTime.get() / numTasks.get()));  
 } finally {  
 super.terminated();  
 }  
 }  
}

### 任务时限

Future的get方法可以限时，如果超时会抛出TimeOutException，那么此时可以通过cancel方法来取消任务。如果编写的任务是可取消的，那么可以提前中止它，以免消耗过多的资源。  
创建n个任务，将其提交到一个线程池，保留n个Future，并使用限时的get方法通过Future串行地获取每一个结果，这一切都很简单。但还有一个更简单的实现：invokeAll。  
将多个任务提交到一个ExecutorService并获得结果。invokeAll方法的参数是一组任务，并返回一组Future。这两个集合有着相同的结构。invokeAll按照任务集合中迭代器的顺序将所有的Future添加到返回的集合中，从而使调用者能将各个Future与其表示的Callable关联起来。当所有任务执行完毕时，或者调用线程被中断时，又或者超时，invokeAll将返回。当超时时，任何还未完成的任务都会取消。当invokeAll返回后，每个任务要么正常地完成，要么被取消，而客户端代码可以调用get或isCancelled来判断究竟是何种情况。

public class QuoteTask implements Callable<TravelQuote> {  
 private final TravelCompany company;  
 private final TravelInfo travelInfo;  
 private ExecutorService exec;  
 public QuoteTask(TravelCompany company, TravelInfo travelInfo) {  
 this.company = company;  
 this.travelInfo = travelInfo;  
 }  
   
 public TravelQuote call() throws Exception {  
 return company.solicitQuote(travelInfo);  
 }  
   
 public List<TravelQuote> getRankedTravelQuotes(TravelInfo travelInfo, Set<TravelCompany> companies, Comparator<TravelQuote> ranking, long time, TimeUnit unit) throws InterruptedException {  
 //任务  
 List<QuoteTask> tasks = new ArrayList<QuoteTask>();  
 for (TravelCompany company : companies) {  
 tasks.add(new QuoteTask(company,travelInfo));  
 }  
 //执行  
 List<Future<TravelQuote>> futures = exec.invokeAll(tasks,time,unit);  
 List<TravelQuote> quotes = new ArrayList<TravelQuote>(tasks.size());  
 Iterator<QuoteTask> taskIterator = tasks.iterator();  
 //取出结果  
 for(Future<TravelQuote> future:futures){  
 QuoteTask task = taskIterator.next();  
 try {  
 quotes.add(future.get());  
 } catch (ExecutionException e) {  
 quotes.add(task.getFailureQuote(e.getCause()));  
 e.printStackTrace();  
 }catch(CancellationException e){  
 quotes.add(task.getTimeOutQuote(e));  
 }  
 }  
 Collections.sort(quotes,ranking);  
 return quotes;  
 }  
}

### 任务关闭

线程有一个相应的所有者，即创建该线程的类，因此线程池是工作者线程的所有者，如果要中断这些线程，那么应该使用线程池。  
ExecutorService中提供了shutdown和shutdownNow方法。  
前者是正常关闭，后者是强行关闭。

* 1）它们都会阻止新任务的提交
* 2）正常关闭是停止空闲线程，正在执行的任务继续执行并完成所有未执行的任务
* 3）强行关闭是停止所有（空闲+工作）线程，关闭当前正在执行的任务，然后返回所有尚未执行的任务。

通常调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用  
shutdownNow方法。

但是我们无法通过常规方法来找出哪些任务已经开始但尚未结束，这意味着我们无法在关闭过程中知道正在执行的任务的状态，除非任务本身会执行某种检查。要知道哪些任务还没有完成，你不仅需要知道哪些任务还没有开始，而且还需要知道当Executor关闭时哪些任务正在执行。

处理非正常的线程终止（只对execute提交的任务有效，submit提交的话会在future.get时将受检异常直接抛出）

要为线程池中的所有线程设置一个UncaughtExceptionHandler，需要为ThreadPoolExecutor的构造函数提供一个ThreadFactory。标准线程池允许当发生未捕获异常时结束线程，但由于使用了一个try-finally块来接收通知，因此当线程结束时，将有新的线程来代替它。如果没有提供捕获异常处理器或者其他的故障通知机制，那么任务会悄悄失败，从而导致很大的混乱。如果你希望在任务由于发生异常而失败时获得通知，并且执行一些特定于任务的恢复操作，那么可以将任务封装在能捕获异常的Runnable或Callable中，或者改写ThreadPoolExecutor的afterExecute方法。

只有通过execute提交的任务，才能将它抛出的异常交给未捕获异常处理器。如果一个由submit提交的任务由于抛出了异常而结束，那么这个异常将被Future.get封装在ExecutionException中重新抛出。

public class QuoteTask implements Callable<TravelQuote> {  
 private final TravelCompany company;  
 private final TravelInfo travelInfo;  
 private ExecutorService exec;  
 public QuoteTask(TravelCompany company, TravelInfo travelInfo) {  
 this.company = company;  
 this.travelInfo = travelInfo;  
 }  
   
 public TravelQuote call() throws Exception {  
 return company.solicitQuote(travelInfo);  
 }  
   
 public List<TravelQuote> getRankedTravelQuotes(TravelInfo travelInfo, Set<TravelCompany> companies, Comparator<TravelQuote> ranking, long time, TimeUnit unit) throws InterruptedException {  
 //任务  
 List<QuoteTask> tasks = new ArrayList<QuoteTask>();  
 for (TravelCompany company : companies) {  
 tasks.add(new QuoteTask(company,travelInfo));  
 }  
 //执行  
 List<Future<TravelQuote>> futures = exec.invokeAll(tasks,time,unit);  
 List<TravelQuote> quotes = new ArrayList<TravelQuote>(tasks.size());  
 Iterator<QuoteTask> taskIterator = tasks.iterator();  
 //取出结果  
 for(Future<TravelQuote> future:futures){  
 QuoteTask task = taskIterator.next();  
 try {  
 quotes.add(future.get());  
 } catch (ExecutionException e) {  
 quotes.add(task.getFailureQuote(e.getCause()));  
 e.printStackTrace();  
 }catch(CancellationException e){  
 quotes.add(task.getTimeOutQuote(e));  
 }  
 }  
 Collections.sort(quotes,ranking);  
 return quotes;  
 }  
}

## ScheduledThreadPoolExecutor

它继承自ThreadPoolExecutor，主要用来在给定的延迟之后运行任务，或者定期执行任务。Timer是单个后台线程，而ScheduledThreadPoolExecutor可以在构造函数中指定多个对应的后台线程数。

public ScheduledThreadPoolExecutor(int corePoolSize,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,  
 new DelayedWorkQueue(), threadFactory, handler);  
}

内部工作队列是DelayedWorkQueue，它是一个无界队列，maxPoolSize这个参数没有意义。

static class DelayedWorkQueue extends AbstractQueue<Runnable>  
 implements BlockingQueue<Runnable>   
  
public class TestScheduledThreadPool {  
 public static void main(String[] args) throws InterruptedException, ExecutionException {  
 ScheduledExecutorService pool = Executors.newScheduledThreadPool(5);  
 for(int i = 0; i < 10 ;++i){  
 Future<Integer> result = pool.schedule(new ThreadPoolDemo2(), 800, TimeUnit.MILLISECONDS);  
 System.out.println(result.get());  
 }  
 pool.shutdown();  
 }  
}  
  
class ThreadPoolDemo2 implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
 int sum = 0;  
 for (int i = 0; i < 100; ++i) {  
 sum += i;  
 System.out.println(Thread.currentThread().getName() + "\t" + i);  
 }  
 return sum;  
 }  
}

## Executors

Executors是一个工厂类，可以创建3种类型的ThreadPoolExecutor和2种类型的ScheduledThreadPool。  


### FixedThreadPool

创建固定线程数的FixedThreadPool，适用于负载比较重的服务器。

public static ExecutorService newFixedThreadPool(int nThreads) {  
 return new ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>());  
}

corePoolSize和maxPoolSize都被设置为创建FixedThreadPoolExecutor时指定的参数nThreads。

keepAliveTime为0表示多余的空闲线程将会被立即终止。

使用无界队列LinkedBlockingQueue来作为线程池的工作队列，并且默认容量为Integer.MAX\_VALUE。使用无界队列会带来以下影响：

* 1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize
* 2）maximumPoolSize是一个无效的参数
* 3）keepAliveTime是一个无效参数
* 4）运行中的FixedThreadPool（未执行shutdown或shutdownNow）不会拒绝任务。

### SingleThreadExecutor

适用于需要保证顺序地执行各个任务，并且在任意时间点不会有多个线程活动的应用场景。

public static ExecutorService newSingleThreadExecutor() {  
 return new FinalizableDelegatedExecutorService  
 (new ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>()));  
}

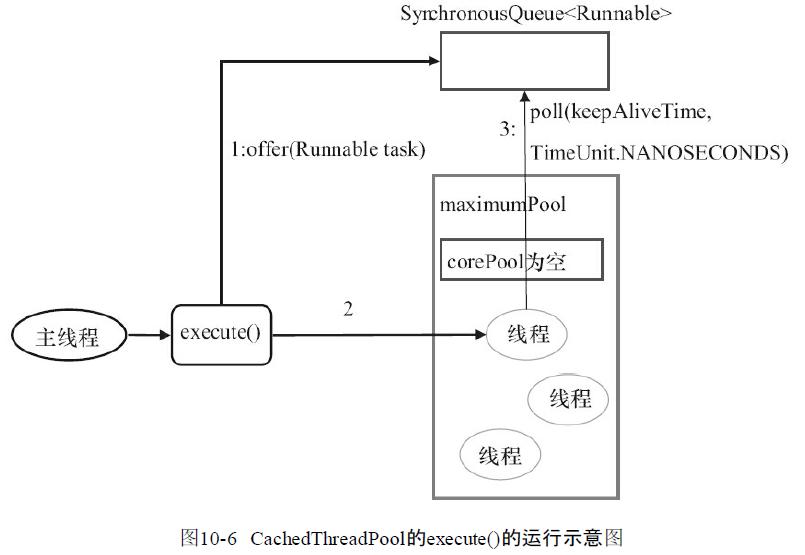
它也是使用无界队列，corePoolSize和maxPoolSize都为1。

### CachedThreadPool

大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。

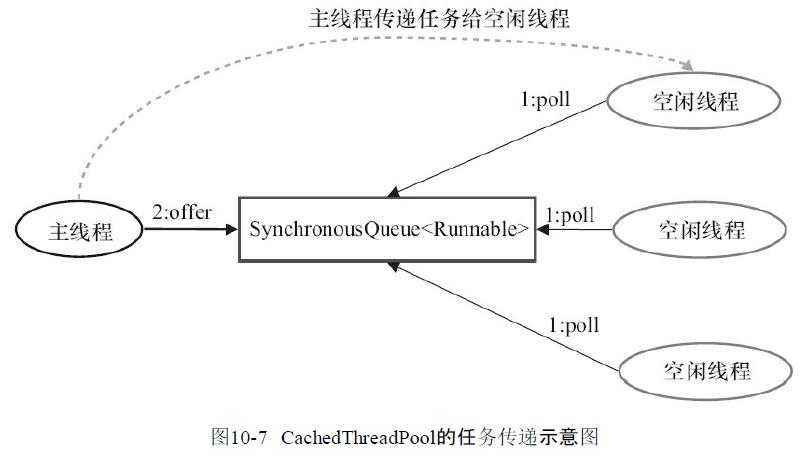
public static ExecutorService newCachedThreadPool() {  
 return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,  
 60L, TimeUnit.SECONDS,  
 new SynchronousQueue<Runnable>());  
}

使用没有缓冲区、只能存储一个元素的SynchronousQueue作为工作队列。

maxPoolSize是无界的，如果主线程提交任务的速度高于maxPool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。  


任务执行过程：

* 1）首先执行SynchronousQueue#offer(Runnable) 。如果当前maxPool中有空闲线程正在执行SynchronousQueue#poll，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；否则执行2）
* 2）当初始maxPool为空，或者maxPool中没有空闲线程时，此时CachedThreadPool会创建一个新线程执行任务
* 3）在2）中新创建的线程执行任务完毕后，会执行SynchronousQueue#poll，这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒。如果60秒内主线程提交了一个新任务，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。



### ScheduledThreadPoolExecutor

固定线程个数，适用于多个后台线程执行周期任务，同时为了满足资源管理的需求而需要限制后台线程的梳理的应用场景。

### SingleThreadScheduledExecutor

public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory) {  
 return new DelegatedScheduledExecutorService  
 (new ScheduledThreadPoolExecutor(1, threadFactory));  
}

适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。

## CompletionService

CompletionService将Executor和BlockingQueue的概念融合在一起，你可以将Callable任务提交给它来执行，然后使用类似于队列操作的take和poll等方法来获得已完成的结果，而这些结果会在完成时封装为Future。ExecutorCompletionService实现了CompletionService并将计算任务委托给一个Executor。

ExecutorCompletionService的实现非常简单，在构造函数中创建一个BlockingQueue来保存计算完成的结果。当计算完成时，调用FutureTask的done方法。当提交某个任务时，该任务将首先包装为一个QueueingFuture，这是FutureTask的一个子类，然后再改写子类的done方法，并将结果放入BlockingQueue中。take和poll方法委托给了BlockingQueue，这些方法会在得出结果之前阻塞。

多个ExecutorCompletionService可以共享一个Executor，因此可以创建一个对于特定计算私有，又能共享一个公共Executor的ExecutorCompletionService。

public class CompletionServiceTest {  
 public void test() throws InterruptedException, ExecutionException {  
 ExecutorService exec = Executors.newCachedThreadPool();  
 CompletionService<Integer> completionService = new ExecutorCompletionService<Integer>(exec);  
 for (int i = 0; i < 10; i++) {  
 completionService.submit(new Task());  
 }  
 int sum = 0;  
 for (int i = 0; i < 10; i++) {  
 //检索并移除表示下一个已完成任务的 Future，如果目前不存在这样的任务，则等待。   
 Future<Integer> future = completionService.take();  
 sum += future.get();  
 }  
 System.out.println("总数为：" + sum);  
 exec.shutdown();  
 }  
}

# J.U.C 源码解析

实现整个并发体系的真正底层是CPU提供的lock前缀+cmpxchg指令和POSIX的同步原语（mutex&condition）

synchronized和wait&notify基于JVM的monitor，monitor底层又是基于POSIX同步原语。

volatile基于CPU的lock前缀指令实现内存屏障。

而J.U.C是基于LockSupport，底层基于POSIX同步原语。

## AbstractQueuedSynchronizer（AQS）

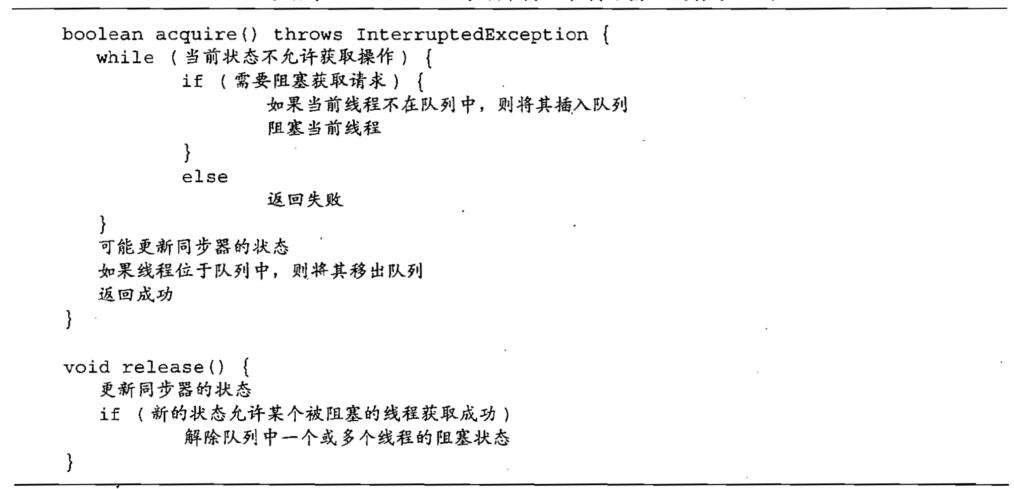
在ReentrantLock和Semaphore这两个接口之间存在许多共同点，这两个类都可以用作一个阀门，即每次只允许一定数量的线程通过，并当线程到达阀门时，可以通过（在调用lock或acquire时成功返回），也可以等待（在调用lock或acquire时阻塞），还可以取消（在调用tryLock或tryAcquire时返回假，表示在指定的时间内锁是不可用的或无法得到许可）。

可以通过锁来实现计数信号量。

事实上，它们在实现时都使用了一个共同的基类，即AbstractQueuedSynchronizer（AQS），这个类也是其他许多同步类的基类。AQS是一个用于构建锁和同步器的框架，许多同步器都可以通过AQS很容易并且高效地构造出来。不仅ReentrantLock和Semaphore，还包括CountDownLatch、ReentrantReadWriteLock、SynchronousQueue和FutureTask，都是基于AQS构造的。

在基于AQS构建的同步器中，只可能在一个时刻发生阻塞，从而降低上下文切换的开销，并提高吞吐量。在设计AQS时充分考虑了可伸缩性，因此java.util.concurrent中所有基于AQS构建的同步器都能获得这个优势。

在基于AQS构建的同步器类中，最基本的操作包括各种形式的获取操作和释放操作。获取操作是一种依赖状态的操作，并且通常会阻塞。当使用锁或信号量时，获取操作的含义就很直观，即获取的是锁或许可，并且调用者可能会一直等待直到同步器类处于可被获取的状态。

AQS负责管理同步器类中的状态，它管理了一个整数类型的状态信息，可以通过getState、setState以及compareAndSetState等protected类型方法来进行操作。这个整数可以用于表示任意状态。  


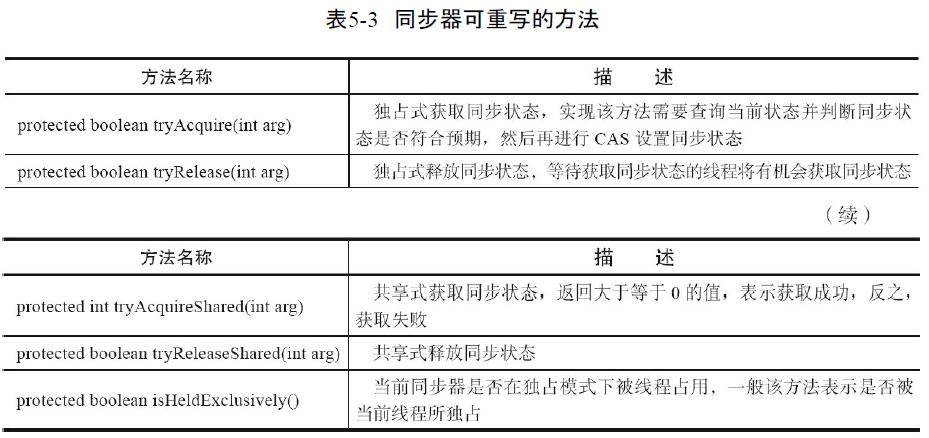
它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。

子类通过继承AQS并实现它的抽象方法来管理同步状态，修改同步状态依赖于AQS的getState、setState、compareAndSetState来进行操作，它们能够保证状态的改变是安全的。

子类推荐被定义为自定义同步组件的静态内部类，AQS自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用。AQS既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态。

### AQS的接口

AQS的设计是基于模板方法模式的，使用者需要继承同步器并重写指定的方法，随后将AQS组合在自定义同步组件的实现中，并调用AQS提供的模板方法，而这些模板方法将会调用使用者重写的方法。

同步器可重写的方法：  


同步器提供的模板方法：  


### AQS使用实例（互斥锁，tryAcquire只需一次CAS）

public class Mutex implements Lock {  
 private final Sync sync = new Sync();  
  
 @Override  
 public void lock() {  
 sync.acquire(1);  
 }  
  
 @Override  
 public void lockInterruptibly() throws InterruptedException {  
 sync.acquireInterruptibly(1);  
 }  
  
 @Override  
 public boolean tryLock() {  
 return sync.tryAcquire(1);  
 }  
  
 @Override  
 public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {  
 return sync.tryAcquireNanos(1, unit.toNanos(time));  
 }  
  
 @Override  
 public void unlock() {  
 sync.release(1);  
 }  
  
 @Override  
 public Condition newCondition() {  
 return sync.newCondition();  
 }  
  
 private static class Sync extends AbstractQueuedSynchronizer {  
 @Override  
 protected boolean tryAcquire(int arg) {  
 if (super.compareAndSetState(0, 1)) {  
 super.setExclusiveOwnerThread(Thread.currentThread());  
 return true;  
 }  
 return false;  
 }  
  
 @Override  
 protected boolean tryRelease(int arg) {  
 if (super.getState() == 0) {  
 throw new IllegalMonitorStateException();  
 }  
 super.setExclusiveOwnerThread(null);  
 super.setState(0);  
 return true;  
 }  
  
 @Override  
 protected boolean isHeldExclusively() {  
 return super.getState() == 1;  
 }  
  
 Condition newCondition() {  
 return new ConditionObject();  
 }  
 }  
}

### AQS实现

主要工作基于CLH队列，voliate关键字修饰的状态state，线程去修改状态成功了就是获取成功，失败了就进队列等待，等待唤醒。在等待唤醒的时候，很多时候会使用自旋（while（!cas()））的方式，不停的尝试获取锁，直到被其他线程获取成功。

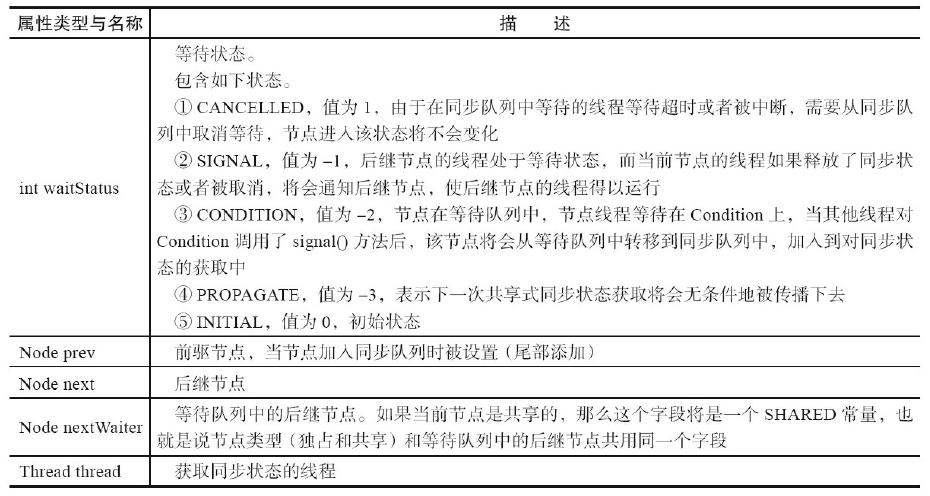
#### AQS#state getState setState

/\*\*  
 \* The synchronization state.  
 \*/  
private volatile int state;  
  
/\*\*  
 \* Returns the current value of synchronization state.  
 \* This operation has memory semantics of a {@code volatile} read.  
 \* @return current state value  
 \*/  
protected final int getState() {  
 return state;  
}  
  
/\*\*  
 \* Sets the value of synchronization state.  
 \* This operation has memory semantics of a {@code volatile} write.  
 \* @param newState the new state value  
 \*/  
protected final void setState(int newState) {  
 state = newState;  
}

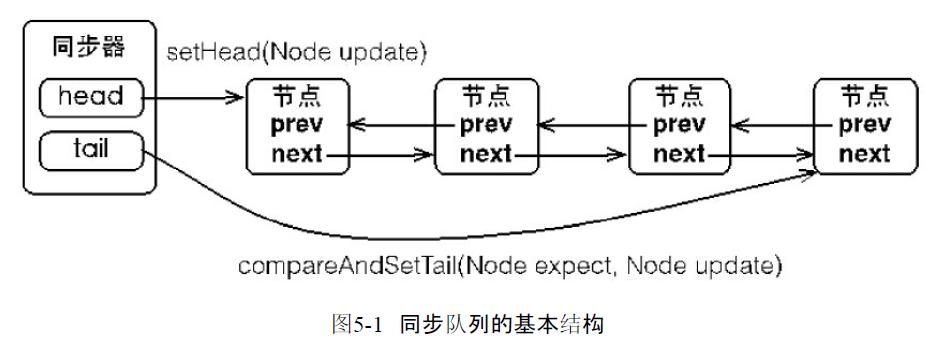
#### 同步队列

AQS依赖内部的CLH同步队列（一个FIFO双向队列）来完成同步状态的管理。当前线程获取同步状态失败时，AQS会将当前线程以及等待状态等信息构造为一个Node并将其加入同步队列，并阻塞当前线程。当同步状态释放时，会把后继节点线程唤醒，使其再次尝试获取同步状态。后继节点将会在获取同步状态成功时将自己设置为头节点。

#### AQS#Node



static final class Node {  
 /\*\* Marker to indicate a node is waiting in shared mode \*/  
 static final Node SHARED = new Node();  
 /\*\* Marker to indicate a node is waiting in exclusive mode \*/  
 static final Node EXCLUSIVE = null;  
  
 /\*\* waitStatus value to indicate thread has cancelled \*/  
 static final int CANCELLED = 1;  
 /\*\* waitStatus value to indicate successor's thread needs unparking \*/  
 static final int SIGNAL = -1;  
 /\*\* waitStatus value to indicate thread is waiting on condition \*/  
 static final int CONDITION = -2;  
 /\*\*  
 \* waitStatus value to indicate the next acquireShared should  
 \* unconditionally propagate  
 \*/  
 static final int PROPAGATE = -3;  
  
 /\*\*  
 \* Status field, taking on only the values:  
 \* SIGNAL: The successor of this node is (or will soon be)  
 \* blocked (via park), so the current node must  
 \* unpark its successor when it releases or  
 \* cancels. To avoid races, acquire methods must  
 \* first indicate they need a signal,  
 \* then retry the atomic acquire, and then,  
 \* on failure, block.  
 \* CANCELLED: This node is cancelled due to timeout or interrupt.  
 \* Nodes never leave this state. In particular,  
 \* a thread with cancelled node never again blocks.  
 \* CONDITION: This node is currently on a condition queue.  
 \* It will not be used as a sync queue node  
 \* until transferred, at which time the status  
 \* will be set to 0. (Use of this value here has  
 \* nothing to do with the other uses of the  
 \* field, but simplifies mechanics.)  
 \* PROPAGATE: A releaseShared should be propagated to other  
 \* nodes. This is set (for head node only) in  
 \* doReleaseShared to ensure propagation  
 \* continues, even if other operations have  
 \* since intervened.  
 \* 0: None of the above  
 \*  
 \* The values are arranged numerically to simplify use.  
 \* Non-negative values mean that a node doesn't need to  
 \* signal. So, most code doesn't need to check for particular  
 \* values, just for sign.  
 \*  
 \* The field is initialized to 0 for normal sync nodes, and  
 \* CONDITION for condition nodes. It is modified using CAS  
 \* (or when possible, unconditional volatile writes).  
 \*/  
 volatile int waitStatus;  
  
 /\*\*  
 \* Link to predecessor node that current node/thread relies on  
 \* for checking waitStatus. Assigned during enqueuing, and nulled  
 \* out (for sake of GC) only upon dequeuing. Also, upon  
 \* cancellation of a predecessor, we short-circuit while  
 \* finding a non-cancelled one, which will always exist  
 \* because the head node is never cancelled: A node becomes  
 \* head only as a result of successful acquire. A  
 \* cancelled thread never succeeds in acquiring, and a thread only  
 \* cancels itself, not any other node.  
 \*/  
 volatile Node prev;  
  
 /\*\*  
 \* Link to the successor node that the current node/thread  
 \* unparks upon release. Assigned during enqueuing, adjusted  
 \* when bypassing cancelled predecessors, and nulled out (for  
 \* sake of GC) when dequeued. The enq operation does not  
 \* assign next field of a predecessor until after attachment,  
 \* so seeing a null next field does not necessarily mean that  
 \* node is at end of queue. However, if a next field appears  
 \* to be null, we can scan prev's from the tail to  
 \* double-check. The next field of cancelled nodes is set to  
 \* point to the node itself instead of null, to make life  
 \* easier for isOnSyncQueue.  
 \*/  
 volatile Node next;  
  
 /\*\*  
 \* The thread that enqueued this node. Initialized on  
 \* construction and nulled out after use.  
 \*/  
 volatile Thread thread;  
  
 /\*\*  
 \* Link to next node waiting on condition, or the special  
 \* value SHARED. Because condition queues are accessed only  
 \* when holding in exclusive mode, we just need a simple  
 \* linked queue to hold nodes while they are waiting on  
 \* conditions. They are then transferred to the queue to  
 \* re-acquire. And because conditions can only be exclusive,  
 \* we save a field by using special value to indicate shared  
 \* mode.  
 \*/  
 Node nextWaiter;  
  
 /\*\*  
 \* Returns true if node is waiting in shared mode.  
 \*/  
 final boolean isShared() {  
 return nextWaiter == SHARED;  
 }  
  
 /\*\*  
 \* Returns previous node, or throws NullPointerException if null.  
 \* Use when predecessor cannot be null. The null check could  
 \* be elided, but is present to help the VM.  
 \*  
 \* @return the predecessor of this node  
 \*/  
 final Node predecessor() throws NullPointerException {  
 Node p = prev;  
 if (p == null)  
 throw new NullPointerException();  
 else  
 return p;  
 }  
  
 Node() { // Used to establish initial head or SHARED marker  
 }  
  
 Node(Thread thread, Node mode) { // Used by addWaiter  
 this.nextWaiter = mode;  
 this.thread = thread;  
 }  
  
 Node(Thread thread, int waitStatus) { // Used by Condition  
 this.waitStatus = waitStatus;  
 this.thread = thread;  
 }  
}



#### 独占式同步状态

在获取同步状态时，AQS调用tryAcquire获取同步状态。AQS维护一个同步队列，获取同步状态失败的线程都会被加入到队列中并在队列进行自旋（等待）；移出队列的条件是前驱节点是头结点且成功获取了同步状态；

在释放同步状态时，AQS调用tryRelease释放同步状态，然后唤醒头节点的后继节点，使其尝试获取同步状态。

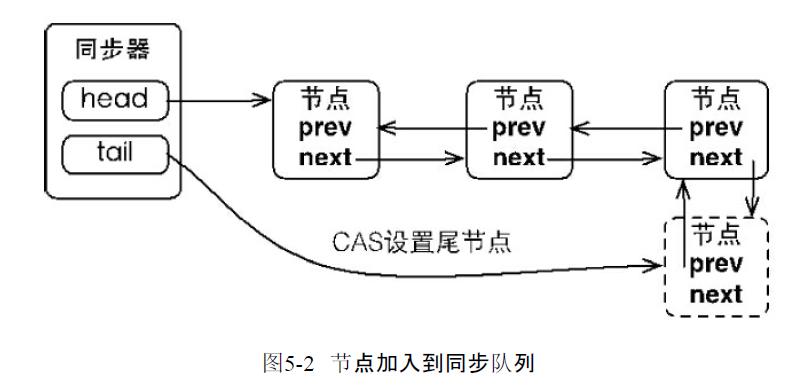
##### AQS#acquire

acquire(int)可以获取同步状态，对中断不敏感。

* 1）调用自定义同步器实现的tryAcquire
* 2）如果成功，那么结束
* 3）如果失败，那么调用addWaiter加入同步队列尾部，并调用acquireQueued获取同步状态（前提是前驱节点为head）
  + 3.1）如果获取到了，那么将自己设置为头节点，返回
  + 3,2）如果前驱节点不是head或者没有获取到，那么判断前驱节点状态是否为SIGNAL，
  + 3.2.1) 如果是，那么阻塞当前线程，阻塞解除后仍自旋获取同步状态
  + 3.2.2) 如果不是，那么删除状态为CANCELLED的前驱节点，将前驱节点状态设置为SIGNAL，继续自旋尝试获取同步状态。

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

###### addWaiter（新Node添加到同步队列尾部，初始状态下head是一个空节点）

  
获取同步状态失败的线程会被构造成Node加入到同步队列尾部，这个过程必须是线程安全的，AQS基于CAS来设置同步队列的尾节点compareAndSetTail。

private Node addWaiter(Node mode) {  
 Node node = new Node(Thread.currentThread(), mode);  
 // Try the fast path of enq; backup to full enq on failure  
 Node pred = tail;  
 if (pred != null) {  
 node.prev = pred;  
 if (compareAndSetTail(pred, node)) {  
 pred.next = node;  
 return node;  
 }  
 }  
 enq(node);  
 return node;  
}

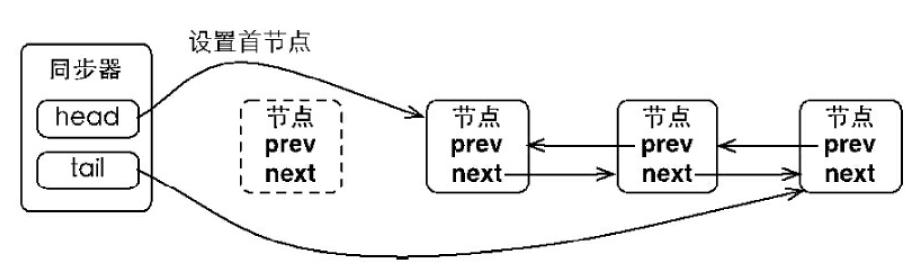
可能tail为null，或者tail不为null，但CAS添加node至尾部失败，此时会enq

如果tail为null，则设置head和tail都指向一个空节点

然后循环CAS添加node至尾部，直至成功。

private Node enq(final Node node) {  
 for (;;) {  
 Node t = tail;  
 if (t == null) { // Must initialize  
 if (compareAndSetHead(new Node()))  
 tail = head;  
 } else {  
 node.prev = t;  
 if (compareAndSetTail(t, node)) {  
 t.next = node;  
 return t;  
 }  
 }  
 }  
}

###### acquireQueued



* 设置首节点是由获取同步状态成功的线程来完成的，因为只有一个线程能够成功获取同步状态，因此设置头节点的方法并不需要CAS的包装。
* 如果自己是第二个结点，那么尝试获取同步状态，如果成功，那么将自己设置为头节点，并返回。
* 如果自己不是第二个结点或者CAS获取失败，那么判断是否应该阻塞，如果应该，那么阻塞，否则自旋重新尝试获取同步状态。

final boolean acquireQueued(final Node node, int arg) {  
 boolean failed = true;  
 try {  
 boolean interrupted = false;  
 for (;;) {  
 // 如果前驱是head，即该结点是第二个结点，那么便有资格去尝试获取资源（可能是head释放完资源唤醒自己的，当然也可能被interrupt了）  
 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);  
 }  
}  
  
/\*\*  
 \* Sets head of queue to be node, thus dequeuing. Called only by  
 \* acquire methods. Also nulls out unused fields for sake of GC  
 \* and to suppress unnecessary signals and traversals.  
 \*  
 \* @param node the node  
 \*/  
private void setHead(Node node) {  
 head = node;  
 node.thread = null;  
 node.prev = null;  
}

###### shouldParkAfterFailedAcquire

* 1）如果前一个节点状态是SIGNAL，那么表示已经设置了前驱节点在获取到同步状态时会唤醒自己，就可以放心的去阻塞了。
* 2）否则会检查前一个节点状态是否是Cancelled
* 2.1）如果是，那么就删除前一个节点，直至状态不是Cancelled。
* 2.2）如果不是，那么将其状态设置为SIGNAL。

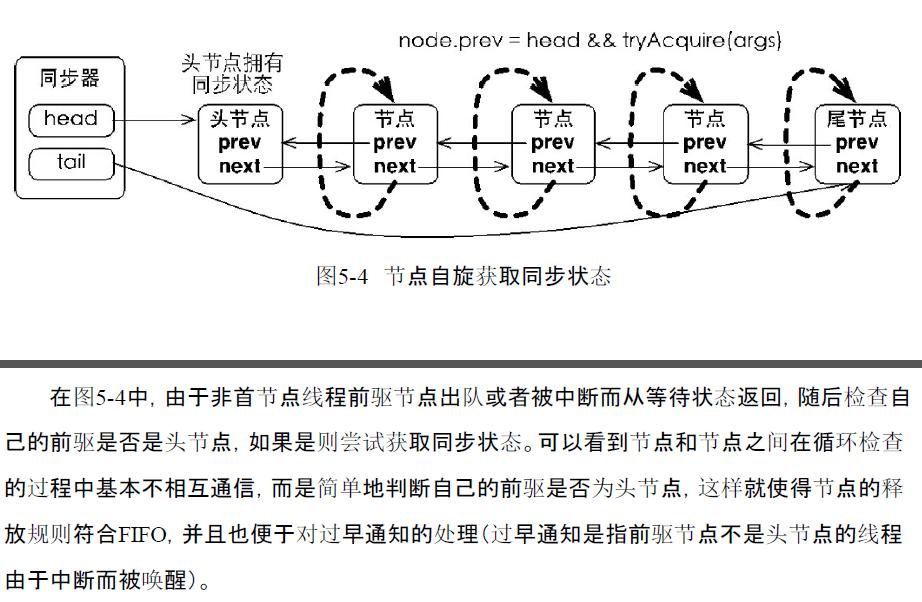
/\*\*  
 \* Checks and updates status for a node that failed to acquire.  
 \* Returns true if thread should block. This is the main signal  
 \* control in all acquire loops. Requires that pred == node.prev.  
 \*  
 \* @param pred node's predecessor holding status  
 \* @param node the node  
 \* @return {@code true} if thread should block  
 \*/  
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
 int ws = pred.waitStatus;  
 if (ws == Node.SIGNAL)  
 /\*  
 \* This node has already set status asking a release  
 \* to signal it, so it can safely park.  
 \*/  
 return true;  
 if (ws > 0) {  
 /\*  
 \* Predecessor was cancelled. Skip over predecessors and  
 \* indicate retry.  
 \*/  
 do {  
 node.prev = pred = pred.prev;  
 } while (pred.waitStatus > 0);  
 pred.next = node;  
 } else {  
 /\*  
 \* waitStatus must be 0 or PROPAGATE. Indicate that we  
 \* need a signal, but don't park yet. Caller will need to  
 \* retry to make sure it cannot acquire before parking.  
 \*/  
 compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  
 }  
 return false;  
}

###### parkAndCheckInterrupt

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

为什么只有前驱节点是头节点才能尝试获取同步状态？

* 1）头节点是成功获取到同步状态的节点，头节点的线程释放了同步状态之后，将会唤醒其后继节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头结点
* 2）维护同步队列的FIFO原则。



##### AQS#release

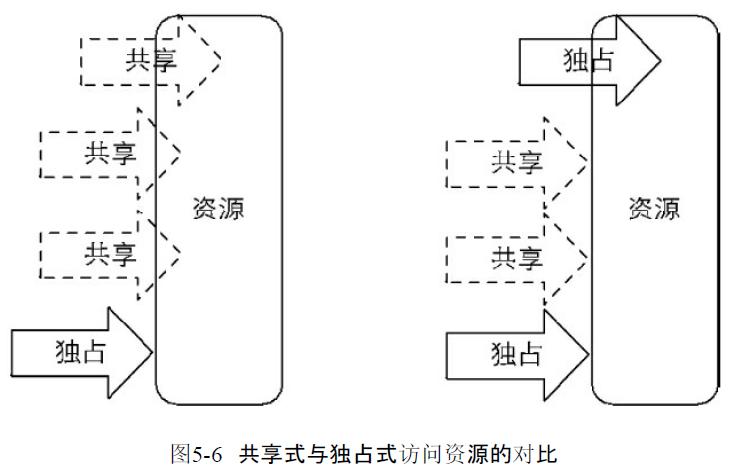
在释放同步状态之后，会唤醒其后继节点，使后继节点继续尝试获取同步状态。

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

###### unparkSuccessor

private void unparkSuccessor(Node node) {  
 /\*  
 \* If status is negative (i.e., possibly needing signal) try  
 \* to clear in anticipation of signalling. It is OK if this  
 \* fails or if status is changed by waiting thread.  
 \*/  
 int ws = node.waitStatus;  
 if (ws < 0)  
 compareAndSetWaitStatus(node, ws, 0);  
  
 /\*  
 \* Thread to unpark is held in successor, which is normally  
 \* just the next node. But if cancelled or apparently null,  
 \* traverse backwards from tail to find the actual  
 \* non-cancelled successor.  
 \*/  
 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);  
}

#### 共享式同步状态

共享式获取与独占式获取最主要的区别在于同一时刻能否有多个线程同时获取到同步状态。  


* 左半部分：共享式访问资源时，其他共享式的访问均被允许，而独占式访问被阻塞
* 右半部分：独占式访问资源时，同一时刻其他访问均被阻塞。

##### AQS#acquireShared

AQS会调用tryAcquireShared方法尝试获取同步状态，该方法返回值为int，当返回值大于等于0时，表示能够获取到同步状态。

如果返回值等于0表示当前线程获取共享锁成功，但它后续的线程是无法继续获取的，也就是不需要把它后面等待的节点唤醒。如果返回值大于0，表示当前线程获取共享锁成功且它后续等待的节点也有可能继续获取共享锁成功，也就是说此时需要把后续节点唤醒让它们去尝试获取共享锁。

* 1）调用自定义同步器实现的tryAcquireShared
* 2）如果成功，那么结束
* 3）如果失败，那么调用addWaiter加入SHARED节点至同步队列尾部，并调用再次尝试获取同步状态（前提是前驱节点为head）
* 3.1）如果获取到了，那么将自己设置为头节点，并向后唤醒共享节点（如果还有剩余acquire），返回
* 3.2） 如果前驱节点不是head或者没有获取到，那么判断前驱节点状态是否为SIGNAL
  + 3.2.1) 如果是，那么阻塞当前线程，阻塞解除后仍自旋获取同步状态
  + 3.2.2) 如果不是，那么删除状态为CANCELLED的前驱节点，将前驱节点状态设置为SIGNAL，继续自旋尝试获取同步状态。

public final void acquireShared(int arg) {  
 if (tryAcquireShared(arg) < 0)  
 doAcquireShared(arg);  
}

##### doAcquiredShared

构造一个当前线程对应的共享节点，如果前驱节点是head并且尝试获取同步状态成功，那么将当前节点设置为head

private void doAcquireShared(int arg) {  
 final Node node = addWaiter(Node.SHARED);  
 boolean failed = true;  
 try {  
 boolean interrupted = false;  
 for (;;) {  
 final Node p = node.predecessor();  
 if (p == head) {  
 int r = tryAcquireShared(arg);  
 if (r >= 0) {  
 setHeadAndPropagate(node, r);  
 p.next = null; // help GC  
 if (interrupted)  
 selfInterrupt();  
 failed = false;  
 return;  
 }  
 }  
 if (shouldParkAfterFailedAcquire(p, node) &&  
 parkAndCheckInterrupt())  
 interrupted = true;  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

##### setHeadAndPropagate

如果获取了同步状态，仍有剩余的acquire，那么继续向后唤醒

/\*\*  
 \* Sets head of queue, and checks if successor may be waiting  
 \* in shared mode, if so propagating if either propagate > 0 or  
 \* PROPAGATE status was set.  
 \*  
 \* @param node the node  
 \* @param propagate the return value from a tryAcquireShared  
 \*/  
private void setHeadAndPropagate(Node node, long propagate) {  
 Node h = head; // Record old head for check below  
 setHead(node);  
 /\*  
 \* Try to signal next queued node if:  
 \* Propagation was indicated by caller,  
 \* or was recorded (as h.waitStatus either before  
 \* or after setHead) by a previous operation  
 \* (note: this uses sign-check of waitStatus because  
 \* PROPAGATE status may transition to SIGNAL.)  
 \* and  
 \* The next node is waiting in shared mode,  
 \* or we don't know, because it appears null  
 \*  
 \* The conservatism in both of these checks may cause  
 \* unnecessary wake-ups, but only when there are multiple  
 \* racing acquires/releases, so most need signals now or soon  
 \* anyway.  
 \*/  
 if (propagate > 0 || h == null || h.waitStatus < 0 ||  
 (h = head) == null || h.waitStatus < 0) {  
 Node s = node.next;  
// 如果当前节点的后继节点是共享类型或者没有后继节点，则进行唤醒  
// 这里可以理解为除非明确指明不需要唤醒（后继等待节点是独占类型），否则都要唤醒  
 if (s == null || s.isShared())  
 doReleaseShared();  
 }  
}

#### AQS#releaseShared

public final boolean releaseShared(int arg) {  
 if (tryReleaseShared(arg)) {  
 doReleaseShared();  
 return true;  
 }  
 return false;  
}

##### doReleaseShared

private void doReleaseShared() {  
 /\*  
 \* Ensure that a release propagates, even if there are other  
 \* in-progress acquires/releases. This proceeds in the usual  
 \* way of trying to unparkSuccessor of head if it needs  
 \* signal. But if it does not, status is set to PROPAGATE to  
 \* ensure that upon release, propagation continues.  
 \* Additionally, we must loop in case a new node is added  
 \* while we are doing this. Also, unlike other uses of  
 \* unparkSuccessor, we need to know if CAS to reset status  
 \* fails, if so rechecking.  
 \*/  
 for (;;) {  
 Node h = head;  
 if (h != null && h != tail) {  
 int ws = h.waitStatus;  
// 表示后继节点需要被唤醒  
 if (ws == Node.SIGNAL) {  
 if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  
 continue; // loop to recheck cases  
 unparkSuccessor(h);  
 }  
//如果后继节点暂时不需要唤醒，则把当前节点状态设置为PROPAGATE确保以后可以传递下去  
 else if (ws == 0 &&  
 !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  
 continue; // loop on failed CAS  
 }  
//如果头结点没有发生变化，表示设置完成，退出循环  
//如果头结点发生变化，比如说其他线程获取到了锁，将自己设置为了头节点。为了使自己的唤醒动作可以传递给之后的节点，就需要重新进入循环  
 if (h == head) // loop if head changed  
 break;  
 }  
}

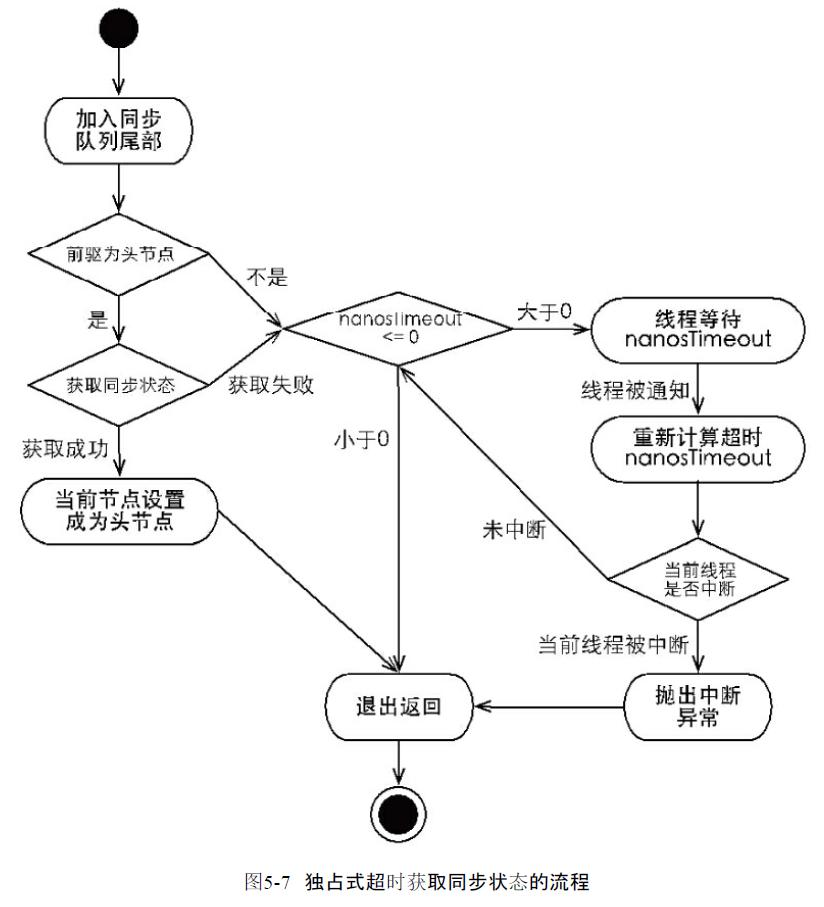
#### 独占式超时获取同步状态

##### AQS#tryAcquireNanos

public final boolean tryAcquireNanos(int arg, long nanosTimeout)  
 throws InterruptedException {  
 if (Thread.interrupted())  
 throw new InterruptedException();  
 return tryAcquire(arg) ||  
 doAcquireNanos(arg, nanosTimeout);  
}

* 该方法可以超时获取同步状态，即在指定上的时间段内获取同步状态，如果成功返回true，失败则返回false。
* 对比另一个获取同步状态的方法acquireInterruptibly，该方法等待时如果被中断，那么会立即返回并抛出InterruptedException；而synchronized即使被中断也仅仅是设置中断标志位，并不会立即返回。
* 而tryAcquireNanos不仅支持响应中断，还增加了超时获取的特性。
* 针对超时获取，主要需要计算出需要等待的时间间隔nanosTImeout，为了防止过早通知，nanosTimeout的计算公式为:nanosTimeout -= now – lastTime。now是当前唤醒时间，lastTime为上次唤醒时间。
* 如果nanosTimeout大于0，则表示超时时间未到，需要继续等待nanosTimeout纳秒；反之已经超时。

##### AQS#doAcquireNanos



private boolean doAcquireNanos(int arg, long nanosTimeout)  
 throws InterruptedException {  
 if (nanosTimeout <= 0L)  
 return false;  
 final long deadline = System.nanoTime() + nanosTimeout;  
 final Node node = addWaiter(Node.EXCLUSIVE);  
 boolean failed = true;  
 try {  
 for (;;) {  
 final Node p = node.predecessor();  
 if (p == head && tryAcquire(arg)) {  
 setHead(node);  
 p.next = null; // help GC  
 failed = false;  
 return true;  
 }  
 nanosTimeout = deadline - System.nanoTime();  
 if (nanosTimeout <= 0L)  
 return false;  
 if (shouldParkAfterFailedAcquire(p, node) &&  
 nanosTimeout > spinForTimeoutThreshold)  
 LockSupport.parkNanos(this, nanosTimeout);  
 if (Thread.interrupted())  
 throw new InterruptedException();  
 }  
 } finally {  
 if (failed)  
 cancelAcquire(node);  
 }  
}

ReentrantLock  
锁是Java编程中最重要的同步机制，除了让临界区互斥执行之外，还可以让释放锁的线程向获取锁的线程发送消息。当线程释放锁时，JMM会把该线程对应的本地cache中的共享变量刷新到主存中。当线程获取锁时，JMM会把该线程对应的本地内存置为无效，从而使得临界区的代码必须从主存中读取共享变量。

对比锁和volatile的内存语义可以看出：锁的释放与volatile的写操作有相同的内存语义，锁的获取与volatile的读操作有相同的内存语义。  
公平锁加锁  
ReentrantLock#lock  
public void lock() {  
 sync.lock();  
}

FairSync#lock  
final void lock() {  
 acquire(1);  
}

FairSync#tryAcquire（重入）  
状态值在没有线程持有锁时为0，有线程持有锁时大于0

获取状态  
1）如果为0，表示是首次获取，判断同步队列中当前节点是否有前驱节点  
 1.1）如果有前驱节点，那么说明锁已被其他线程占有，返回失败  
 1.2) 如果没有前驱节点，那么说明当前节点为head，CAS将状态设置为1  
 1.2.1) 如果设置成功，那么获取锁成功，将独占锁持有者设置为当前线程  
 1.2.2) 如果设置失败，那么说明锁竞争失败，返回失败  
2）如果不为0，判断独占锁持有者是否是当前线程  
 2.1）如果是，那么说明出现了重入，则将状态++  
 2.2）如果不是，那么说明锁已被其他线程占有，返回失败

与非公平的tryAcquire相比，多了一个方法调用hasQueuedPredecessors，即加入了同步队列中当前节点是否有前驱节点的判断。如果有前驱节点，那么有线程比当前线程更早地请求锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.currentThread();  
 int c = getState();  
// 首次获取  
 if (c == 0) {  
 if (!hasQueuedPredecessors() &&  
 compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
// 重入  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
// 已被其他线程占有  
 return false;  
}  
hasQueuedPredecessors  
public final boolean hasQueuedPredecessors() {  
 // The correctness of this depends on head being initialized  
 // before tail and on head.next being accurate if the current  
 // thread is first in queue.  
 Node t = tail; // Read fields in reverse initialization order  
 Node h = head;  
 Node s;  
 return h != t &&  
 ((s = h.next) == null || s.thread != Thread.currentThread());  
}

公平锁解锁  
ReentrantLock#unlock  
public void unlock() {  
 sync.release(1);  
}

Sync#tryRelease  
1）如果当前线程不是独占锁持有者，则抛出异常。  
2）获取状态，将状态值减一，如果减为0，则将独占锁持有者设置为null，表示当前线程不再持有这个锁  
3）更新状态值

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;  
}

公平锁总结  
在释放锁的最后写volatile变量state，在获取锁时首先读这个volatile变量。根据volatile的happens-before规则，释放锁的线程在写volatile变量之前可见的共享内存，在获取锁的线程读取同一个volatile变量后将立即变得对获取锁的线程可见。

非公平锁加锁  
NonfairSync#lock  
CAS同时具有volatile读和volatile写的内存语义。  
底层是基于CPU的cmpxchg指令实现的，Intel会规定该指令：禁止该指令与 之前 和 之后 的读写指令重排序；把写缓冲区中的所有数据刷新到内存中。  
这一点就足以同时实现volatile读和volatile写的内存语义了。

final void lock() {  
 if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.currentThread());  
 else  
 acquire(1);  
}

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

NonfairLock#tryAcquire  
protected final boolean tryAcquire(int acquires) {  
 return nonfairTryAcquire(acquires);  
}  
NonfairLock#nonfairTryAcquire（重入）  
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;  
}

ReentrantReadWriteLock  
读写状态的设计  
读写锁的自定义同步器需要在同步状态上维护多个读线程和一个写线程的状态，使得该状态的设计成为读写锁实现的关键。  
如果在一个整型变量上维护多种状态就需要按位切割使用该变量，读写锁把变量切成了两个部分，高16位表示读，低16位表示写。

当前同步状态为S，则写状态=S&0x0000FFFF，读状态=S>>>16。  
写状态加一，就是S+1；读状态加一，就是S+（1<<16）。  
S不为0时，若写状态为0，则读状态大于0，读锁已被获取。  
写锁的获取与释放  
写锁是一个支持重入的排它锁。如果当前线程已经获取了写锁，则增加写状态，如果当前线程在获取写锁时，读锁已经被获取，或者该线程不是已经获取写锁的线程，则当前线程进入等待状态。  
WriteLock#lock  
public void lock() {  
 sync.acquire(1);  
}  
Sync#tryAcquire  
如果存在读锁，那么写锁不能被获取。因为读写锁要确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。因此，只有等待其他读线程都是释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。

获取状态  
1）如果状态不为0，计算写状态  
 如果写状态为0（存在读锁）或者独占锁持有者不是当前线程，则返回失败  
 否则是写线程重入的情况，更新写状态++，返回成功  
2）如果状态为0，如果是公平锁，那么判断当前节点是否有后继节点  
 2.1）如果有，则返回失败  
 2.2）如果没有，或者是非公平锁，则CAS更新写状态++  
 2.2.1）如果更新失败，则返回失败  
 2.2.2) 如果更新成功，则将独占锁持有者设置为当前线程，返回成功  
protected final boolean tryAcquire(int acquires) {  
 Thread current = Thread.currentThread();  
 int c = getState();  
 int w = exclusiveCount(c);  
 if (c != 0) {  
 // (Note: if c != 0 and w == 0 then shared count != 0)  
 if (w == 0 || current != getExclusiveOwnerThread())  
 return false;  
 if (w + exclusiveCount(acquires) > MAX\_COUNT)  
 throw new Error("Maximum lock count exceeded");  
 // Reentrant acquire  
 setState(c + acquires);  
 return true;  
 }  
 if (writerShouldBlock() ||  
 !compareAndSetState(c, c + acquires))  
 return false;  
 setExclusiveOwnerThread(current);  
 return true;  
}

writerShouldBlock  
如果是公平，那么返回当前节点是否有后继节点，即hasQueuedPredecessors；如果是非公平，则直接返回false  
WriteLock#unlock  
public void unlock() {  
 sync.release(1);  
}

Sync#tryRelease  
protected final boolean tryRelease(int releases) {  
 if (!isHeldExclusively())  
 throw new IllegalMonitorStateException();  
 int nextc = getState() - releases;  
 boolean free = exclusiveCount(nextc) == 0;  
 if (free)  
 setExclusiveOwnerThread(null);  
 setState(nextc);  
 return free;  
}  
读锁的获取与释放（放弃）  
读锁是一个支持重入的共享锁，它能够被多个线程同时获取，在写状态为0时，读锁总会成功地获取，而所做的也只是线程安全地增加读状态。如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已经被其他线程获取，则进入等待状态。  
在Java6中除了保存所有线程获取读锁次数的总和（state的高16位），也保存了每个线程各自获取读锁的次数（ThreadLocal）。

ReadLock#lock  
public void lock() {  
 sync.acquireShared(1);  
}

Sync#tryAcquireShared

1）如果有线程持有写锁并且不是当前线程，直接返回失败；  
2）获取读状态  
2.1）如果是公平锁，那么判断当前节点是否有后继节点  
 2.1.1）如果有，则执行fullTryAcquireShared  
 2.1.2) 如果没有，继续执行  
2.2）如果是非公平锁，那么判断  
CAS设置state成功，则设置读锁count的值。这一步并没有检查读锁重入的情况，被延迟到fullTryAcquireShared里了，因为大多数情况下不是重入的；  
3.如果步骤2失败了，或许是队列策略返回false或许是CAS设置失败了等，则执行fullTryAcquireShared。

protected final int tryAcquireShared(int unused) {  
 Thread current = Thread.currentThread();  
 int c = getState();  
// 有其他写线程，则失败  
 if (exclusiveCount(c) != 0 &&  
 getExclusiveOwnerThread() != current)  
 return -1;  
 int r = sharedCount(c);  
 if (!readerShouldBlock() &&  
 r < MAX*COUNT &&*  
 *compareAndSetState(c, c + SHARED*UNIT)) {  
 if (r == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
 HoldCounter rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 cachedHoldCounter = rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 }  
 return 1;  
 }  
 return fullTryAcquireShared(current);  
}  
readerShouldBlock  
如果是公平，那么返回当前节点是否有后继节点，即hasQueuedPredecessors；如果是非公平，则调用apparentlyFirstQueuedIsExclusive  
/\*\*

* Returns {@code true} if the apparent first queued thread, if one
* exists, is waiting in exclusive mode. If this method returns
* {@code true}, and the current thread is attempting to acquire in
* shared mode (that is, this method is invoked from {@link
* #tryAcquireShared}) then it is guaranteed that the current thread
* is not the first queued thread. Used only as a heuristic in
* ReentrantReadWriteLock.  
  \*/  
  final boolean apparentlyFirstQueuedIsExclusive() {  
  Node h, s;  
  return (h = head) != null &&  
   (s = h.next) != null &&  
   !s.isShared() &&  
   s.thread != null;  
  }

fullTryAcquireShared  
final int fullTryAcquireShared(Thread current) {  
 /\*  
 \* This code is in part redundant with that in  
 \* tryAcquireShared but is simpler overall by not  
 \* complicating tryAcquireShared with interactions between  
 \* retries and lazily reading hold counts.  
 \*/  
 HoldCounter rh = null;  
 for (;;) {  
 int c = getState();  
 if (exclusiveCount(c) != 0) {  
// 有线程持有写锁且不是当前线程，直接失败  
 if (getExclusiveOwnerThread() != current)  
 return -1;  
// 如果队列策略不允许，需要检查是否是读锁重入的情况。队列策略是否允许，分两种情况:  
// 1.公平模式：如果当前AQS队列前面有等待的结点，返回false；2.非公平模式：如果  
// AQS前面有线程在等待写锁，返回false(这样做的原因是为了防止写饥饿)。  
 } else if (readerShouldBlock()) {  
 // 如果当前线程是第一个获取读锁的线程，则有资格获取读锁  
 if (firstReader == current) {  
 // assert firstReaderHoldCount > 0;  
 } else {  
// 优先赋值成上一次获取读锁成功的cache，如果发现线程tid和当前线程不相等，再从ThreadLocal里获取  
 if (rh == null) {  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current)) {  
 rh = readHolds.get();  
 if (rh.count == 0)  
 readHolds.remove();  
 }  
 }  
// 说明不是读锁重入的情况，直接返回失败了  
 if (rh.count == 0)  
 return -1;  
 }  
 }  
 if (sharedCount(c) == MAX*COUNT)*  
 *throw new Error("Maximum lock count exceeded");*  
 *if (compareAndSetState(c, c + SHARED*UNIT)) {  
// 设置当前线程为第一个获取读锁的线程  
 if (sharedCount(c) == 0) {  
 firstReader = current;  
 firstReaderHoldCount = 1;  
// 读锁重入  
 } else if (firstReader == current) {  
 firstReaderHoldCount++;  
 } else {  
// 其他获取读锁成功的情况  
 if (rh == null)  
 rh = cachedHoldCounter;  
 if (rh == null || rh.tid != getThreadId(current))  
 rh = readHolds.get();  
 else if (rh.count == 0)  
 readHolds.set(rh);  
 rh.count++;  
 cachedHoldCounter = rh; // cache for release  
 }  
 return 1;  
 }  
 }  
}

锁降级  
锁降级是指把持有写锁，再获取到读锁，随后释放写锁的过程。

锁降级中读锁的获取是否必要？主要是为了保证数据的可见性。如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程T获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞。

LockSupport  
public static void park() {  
 UNSAFE.park(false, 0L);  
}

/\*\*

* Block current thread, returning when a balancing
* unpark occurs, or a balancing unpark has
* already occurred, or the thread is interrupted, or, if not
* absolute and time is not zero, the given time nanoseconds have
* elapsed, or if absolute, the given deadline in milliseconds
* since Epoch has passed, or spuriously (i.e., returning for no
* "reason"). Note: This operation is in the Unsafe class only
* because unpark is, so it would be strange to place it
* elsewhere.  
  \*/  
  public native void park(boolean isAbsolute, long time);

Unsafe*Park*  
*UNSAFE*ENTRY(void, Unsafe*Park(JNIEnv \*env, jobject unsafe, jboolean isAbsolute, jlong time))*  
 *UnsafeWrapper("Unsafe*Park");  
 EventThreadPark event;  
#ifndef USDT2  
 HS*DTRACE*PROBE3(hotspot, thread**park**begin, thread->parker(), (int) isAbsolute, time);  
#else /\* USDT2 */*  
 *HOTSPOTTHREADPARKBEGIN(*  
 *(uintptrt) thread->parker(), (int) isAbsolute, time);*  
*#endif /* USDT2 */*  
 *JavaThreadParkedState jtps(thread, time != 0);*  
 *thread->parker()->park(isAbsolute != 0, time);*  
*#ifndef USDT2*  
 *HSDTRACEPROBE1(hotspot, thread****park****end, thread->parker());*  
*#else /* USDT2 */*  
 *HOTSPOTTHREADPARKEND(*  
 *(uintptrt) thread->parker());*  
*#endif /* USDT2 \*/  
 if (event.should*commit()) {*  
 *oop obj = thread->current*park*blocker();*  
 *event.set*klass((obj != NULL) ? obj->klass() : NULL);  
 event.set*timeout(time);*  
 *event.set*address((obj != NULL) ? (TYPE*ADDRESS) cast*from*oop<uintptr*t>(obj) : 0);  
 event.commit();  
 }  
UNSAFE*END*  
*Parker*  
*定义私有属性*counter：可以理解为是否可以调用park的一个许可证，只有\_count > 0的时候才能调用；  
提供public方法park和unpark支撑阻塞/唤醒线程；  
Parker继承PlatformParker  
class Parker : public os::PlatformParker {  
private:  
 volatile int \_counter ;  
 Parker \* FreeNext ;  
 JavaThread \* AssociatedWith ; // Current association

public:  
 Parker() : PlatformParker() {  
 \_counter = 0 ;  
 FreeNext = NULL ;  
 AssociatedWith = NULL ;  
 }  
protected:  
 ~Parker() { ShouldNotReachHere(); }  
public:  
 // For simplicity of interface with Java, all forms of park (indefinite,  
 // relative, and absolute) are multiplexed into one call.  
 void park(bool isAbsolute, jlong time);  
 void unpark();

// Lifecycle operators  
 static Parker \* Allocate (JavaThread \* t) ;  
 static void Release (Parker \* e) ;  
private:  
 static Parker \* volatile FreeList ;  
 static volatile int ListLock ;

};

Linux#PlatformParker  
linux下的PlatformParker，基于POSIX的线程编写的。  
POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。在类Unix操作系统（Unix、Linux、Mac OS X等）中，都使用Pthreads作为操作系统的线程。Windows操作系统也有其移植版pthreads-win32   
class PlatformParker : public CHeapObj {  
 protected:  
 enum {  
 REL*INDEX = 0,*  
 *ABS*INDEX = 1  
 };  
 int *cur*index; // which cond is in use: -1, 0, 1  
 pthread*mutex*t *mutex [1] ;*  
 *pthread*cond\_t \_cond [2] ; // one for relative times and one for abs.

public: // TODO-FIXME: make dtor private  
 ~PlatformParker() { guarantee (0, "invariant") ; }

public:  
 PlatformParker() {  
 int status;  
 status = pthread*cond*init (&*cond[REL*INDEX], os::Linux::condAttr());  
 assert*status(status == 0, status, "cond*init rel");  
 status = pthread*cond*init (&*cond[ABS*INDEX], NULL);  
 assert*status(status == 0, status, "cond*init abs");  
 status = pthread*mutex*init (*mutex, NULL);*  
 *assert*status(status == 0, status, "mutex*init");*  
 *\_cur*index = -1; // mark as unused  
 }  
};  
Parker#park  
用mutex和condition保护了一个\_counter的变量，当park时，这个变量置为了0，当unpark时，这个变量置为1。

1、先尝试使用Atomic的xchg，CAS查看counter是否大于0，如果是，那么更新为0，返回  
2、构造一个ThreadBlockInVM，判断如果*counter > 0，可以调用，将*counter置为0，，unlock mutex，返回  
3、根据等待时间调用不同的等待函数等待，如果等待返回正确，将\_counter置为0，unlock mutex，返回，park调用成功。  
void Parker::park(bool isAbsolute, jlong time) {  
 // Ideally we'd do something useful while spinning, such  
 // as calling unpackTime().

// Optional fast-path check:  
 // Return immediately if a permit is available.  
 // We depend on Atomic::xchg() having full barrier semantics  
 // since we are doing a lock-free update to *counter.*  
 *if (Atomic::xchg(0, &*counter) > 0) return;

Thread\* thread = Thread::current();  
 assert(thread->is*Java*thread(), "Must be JavaThread");  
 JavaThread \*jt = (JavaThread \*)thread;

// Optional optimization -- avoid state transitions if there's an interrupt pending.  
 // Check interrupt before trying to wait  
 if (Thread::is\_interrupted(thread, false)) {  
 return;  
 }

// Next, demultiplex/decode time arguments  
 timespec absTime;  
 if (time < 0 || (isAbsolute && time == 0) ) { // don't wait at all  
 return;  
 }  
 if (time > 0) {  
 unpackTime(&absTime, isAbsolute, time);  
 }

// Enter safepoint region  
 // Beware of deadlocks such as 6317397.  
 // The per-thread Parker:: mutex is a classic leaf-lock.  
 // In particular a thread must never block on the Threads*lock while*  
 *// holding the Parker:: mutex. If safepoints are pending both the*  
 *// the ThreadBlockInVM() CTOR and DTOR may grab Threads*lock.  
 ThreadBlockInVM tbivm(jt);

// Don't wait if cannot get lock since interference arises from  
 // unblocking. Also. check interrupt before trying wait  
 if (Thread::is*interrupted(thread, false) || pthread*mutex*trylock(*mutex) != 0) {  
 return;  
 }

int status ;  
 if (*counter > 0) { // no wait needed*  
 *\_counter = 0;*  
 *status = pthread*mutex*unlock(*mutex);  
 assert (status == 0, "invariant") ;  
 // Paranoia to ensure our locked and lock-free paths interact  
 // correctly with each other and Java-level accesses.  
 OrderAccess::fence();  
 return;  
 }

#ifdef ASSERT  
 // Don't catch signals while blocked; let the running threads have the signals.  
 // (This allows a debugger to break into the running thread.)  
 sigset*t oldsigs;*  
 *sigset*t\* allowdebug*blocked = os::Linux::allowdebug*blocked*signals();*  
 *pthread*sigmask(SIG*BLOCK, allowdebug*blocked, &oldsigs);  
#endif

OSThreadWaitState osts(thread->osthread(), false /\* not Object.wait() \*/);  
 jt->set*suspend*equivalent();  
 // cleared by handle*special*suspend*equivalent*condition() or java*suspend*self()

assert(*cur*index == -1, "invariant");  
 if (time == 0) {  
 *cur*index = REL*INDEX; // arbitrary choice when not timed*  
 *status = pthread*cond*wait (&*cond[*cur*index], *mutex) ;*  
 *} else {*  
 *\_cur*index = isAbsolute ? ABS*INDEX : REL*INDEX;  
 status = os::Linux::safe*cond*timedwait (&*cond[*cur*index], \_mutex, &absTime) ;*  
 *if (status != 0 && WorkAroundNPTLTimedWaitHang) {*  
 *pthread*cond*destroy (&*cond[*cur*index]) ;  
 pthread*cond*init (&*cond[*cur*index], isAbsolute ? NULL : os::Linux::condAttr());*  
 *}*  
 *}*  
 *\_cur*index = -1;  
 assert*status(status == 0 || status == EINTR ||*  
 *status == ETIME || status == ETIMEDOUT,*  
 *status, "cond*timedwait");

#ifdef ASSERT  
 pthread*sigmask(SIG*SETMASK, &oldsigs, NULL);  
#endif

*counter = 0 ;*  
 *status = pthread*mutex*unlock(*mutex) ;  
 assert\_status(status == 0, status, "invariant") ;  
 // Paranoia to ensure our locked and lock-free paths interact  
 // correctly with each other and Java-level accesses.  
 OrderAccess::fence();

// If externally suspended while waiting, re-suspend  
 if (jt->handle*special*suspend*equivalent*condition()) {  
 jt->java*suspend*self();  
 }  
}

Parker#unpark  
将*counter置为1；*  
*判断之前*counter的值：  
小于1时，调用pthread*cond*signal唤醒在park中等待的线程,unlock mutex；  
等于1时，unlock mutex，返回。  
void Parker::unpark() {  
 int s, status ;  
 status = pthread*mutex*lock(*mutex);*  
 *assert (status == 0, "invariant") ;*  
 *s = \_counter;*  
 *\_counter = 1;*  
 *if (s < 1) {*  
 *// thread might be parked*  
 *if (*cur*index != -1) {*  
 *// thread is definitely parked*  
 *if (WorkAroundNPTLTimedWaitHang) {*  
 *status = pthread*cond*signal (&*cond[*cur*index]);  
 assert (status == 0, "invariant");  
 status = pthread*mutex*unlock(*mutex);*  
 *assert (status == 0, "invariant");*  
 *} else {*  
 *status = pthread*mutex*unlock(*mutex);  
 assert (status == 0, "invariant");  
 status = pthread*cond*signal (&*cond[*cur*index]);*  
 *assert (status == 0, "invariant");*  
 *}*  
 *} else {*  
 *pthread*mutex*unlock(*mutex);  
 assert (status == 0, "invariant") ;  
 }  
 } else {  
 pthread*mutex*unlock(\_mutex);  
 assert (status == 0, "invariant") ;  
 }  
}

wait&notify（忽略）  
1）使用wait、notify、notifyAll时需要先对调用对象加锁  
2）调用wait方法后，会放弃对象的锁，线程状态由运行态转为等待态，并将当前线程放置到对象的等待队列  
3）notify或notifyAll方法调用后，等待线程不会从wait返回，需要调用notify或notifyAll的线程释放锁后，等待线程才有机会从wait返回  
4）notify方法是将等待队列中的一个等待线程能从等待队列移至同步队列中，notifyAll方法是将等待队列中的所有线程全部移至同步队列中，被移动的线程状态由等待态转为阻塞态  
5）从wait方法返回的前提是获得了调用对象的锁

wait&notify前提也是基于monitorenter、monitorexit指令实现的（对应1））。

WaitThread首先获取了对象的锁，然后调用对象的wait方法，从而放弃了锁，并进入了对象的等待队列WaitQueue中，进入等待状态。  
由于WaitThread释放了对象的锁，NotifyThread随后获取了对象的锁，并调用对象的notify方法，将WaitThrad从等待队列WaitQueue移到同步队列SynchronizedQueue中，此时WaitThread状态变为阻塞态。NotifyThread释放了锁之后，WaitThread再次获取到了锁，并从wait方法返回继续执行。  
ObjectMonitor#wait  
在HotSpot虚拟机中，monitor采用ObjectMonitor实现。  
void ObjectMonitor::wait(jlong millis, bool interruptible, TRAPS) {  
 Thread \* const Self = THREAD ;  
 assert(Self->is*Java*thread(), "Must be Java thread!");  
 JavaThread \*jt = (JavaThread \*)THREAD;

DeferredInitialize () ;

// Throw IMSX or IEX.  
 CHECK\_OWNER();

EventJavaMonitorWait event;

// check for a pending interrupt  
 if (interruptible && Thread::is*interrupted(Self, true) && !HAS*PENDING*EXCEPTION) {*  
 *// post monitor waited event. Note that this is past-tense, we are done waiting.*  
 *if (JvmtiExport::should*post*monitor*waited()) {  
 // Note: 'false' parameter is passed here because the  
 // wait was not timed out due to thread interrupt.  
 JvmtiExport::post*monitor*waited(jt, this, false);  
 }  
 if (event.should*commit()) {*  
 *post*monitor*wait*event(&event, 0, millis, false);  
 }  
 TEVENT (Wait - Throw IEX) ;  
 THROW(vmSymbols::java*lang*InterruptedException());  
 return ;  
 }

TEVENT (Wait) ;

assert (Self->*Stalled == 0, "invariant") ;*  
 *Self->*Stalled = intptr*t(this) ;*  
 *jt->set*current*waiting*monitor(this);

// create a node to be put into the queue  
 // Critically, after we reset() the event but prior to park(), we must check  
 // for a pending interrupt.  
 ObjectWaiter node(Self);  
 node.TState = ObjectWaiter::TS*WAIT ;*  
 *Self->*ParkEvent->reset() ;  
 OrderAccess::fence(); // ST into Event; membar ; LD interrupted-flag

// Enter the waiting queue, which is a circular doubly linked list in this case  
 // but it could be a priority queue or any data structure.  
 // \_WaitSetLock protects the wait queue. Normally the wait queue is accessed only  
 // by the the owner of the monitor *except* in the case where park()  
 // returns because of a timeout of interrupt. Contention is exceptionally rare  
 // so we use a simple spin-lock instead of a heavier-weight blocking lock.

Thread::SpinAcquire (&*WaitSetLock, "WaitSet - add") ;*  
 *AddWaiter (&node) ;*  
 *Thread::SpinRelease (&*WaitSetLock) ;

if ((SyncFlags & 4) == 0) {  
 *Responsible = NULL ;*  
 *}*  
 *intptr*t save = *recursions; // record the old recursion count*  
 *\_waiters++; // increment the number of waiters*  
 *\_recursions = 0; // set the recursion level to be 1*  
 *exit (true, Self) ; // exit the monitor*  
 *guarantee (*owner != Self, "invariant") ;

// As soon as the ObjectMonitor's ownership is dropped in the exit()  
 // call above, another thread can enter() the ObjectMonitor, do the  
 // notify(), and exit() the ObjectMonitor. If the other thread's  
 // exit() call chooses this thread as the successor and the unpark()  
 // call happens to occur while this thread is posting a  
 // MONITOR*CONTENDED*EXIT event, then we run the risk of the event  
 // handler using RawMonitors and consuming the unpark().  
 //  
 // To avoid the problem, we re-post the event. This does no harm  
 // even if the original unpark() was not consumed because we are the  
 // chosen successor for this monitor.  
 if (node.*notified != 0 && \_succ == Self) {*  
 *node.*event->unpark();  
 }

// The thread is on the WaitSet list - now park() it.  
 // On MP systems it's conceivable that a brief spin before we park  
 // could be profitable.  
 //  
 // TODO-FIXME: change the following logic to a loop of the form  
 // while (!timeout && !interrupted && \_notified == 0) park()

int ret = OS*OK ;*  
 *int WasNotified = 0 ;*  
 *{ // State transition wrappers*  
 *OSThread\* osthread = Self->osthread();*  
 *OSThreadWaitState osts(osthread, true);*  
 *{*  
 *ThreadBlockInVM tbivm(jt);*  
 *// Thread is in thread*blocked state and oop access is unsafe.  
 jt->set*suspend*equivalent();

if (interruptible && (Thread::is\_interrupted(THREAD, false) || HAS\_PENDING\_EXCEPTION)) {  
 // Intentionally empty  
 } else  
 if (node.\_notified == 0) {  
 if (millis <= 0) {  
 Self->\_ParkEvent->park () ;  
 } else {  
 ret = Self->\_ParkEvent->park (millis) ;  
 }  
 }  
  
 // were we externally suspended while we were waiting?  
 if (ExitSuspendEquivalent (jt)) {  
 // TODO-FIXME: add -- if succ == Self then succ = null.  
 jt->java\_suspend\_self();  
 }  
  
 } // Exit thread safepoint: transition \_thread\_blocked -> \_thread\_in\_vm

// Node may be on the WaitSet, the EntryList (or cxq), or in transition  
 // from the WaitSet to the EntryList.  
 // See if we need to remove Node from the WaitSet.  
 // We use double-checked locking to avoid grabbing \_WaitSetLock  
 // if the thread is not on the wait queue.  
 //  
 // Note that we don't need a fence before the fetch of TState.  
 // In the worst case we'll fetch a old-stale value of TS\_WAIT previously  
 // written by the is thread. (perhaps the fetch might even be satisfied  
 // by a look-aside into the processor's own store buffer, although given  
 // the length of the code path between the prior ST and this load that's  
 // highly unlikely). If the following LD fetches a stale TS\_WAIT value  
 // then we'll acquire the lock and then re-fetch a fresh TState value.  
 // That is, we fail toward safety.  
  
 if (node.TState == ObjectWaiter::TS\_WAIT) {  
 Thread::SpinAcquire (&\_WaitSetLock, "WaitSet - unlink") ;  
 if (node.TState == ObjectWaiter::TS\_WAIT) {  
 DequeueSpecificWaiter (&node) ; // unlink from WaitSet  
 assert(node.\_notified == 0, "invariant");  
 node.TState = ObjectWaiter::TS\_RUN ;  
 }  
 Thread::SpinRelease (&\_WaitSetLock) ;  
 }  
  
 // The thread is now either on off-list (TS\_RUN),  
 // on the EntryList (TS\_ENTER), or on the cxq (TS\_CXQ).  
 // The Node's TState variable is stable from the perspective of this thread.  
 // No other threads will asynchronously modify TState.  
 guarantee (node.TState != ObjectWaiter::TS\_WAIT, "invariant") ;  
 OrderAccess::loadload() ;  
 if (\_succ == Self) \_succ = NULL ;  
 WasNotified = node.\_notified ;  
  
 // Reentry phase -- reacquire the monitor.  
 // re-enter contended monitor after object.wait().  
 // retain OBJECT\_WAIT state until re-enter successfully completes  
 // Thread state is thread\_in\_vm and oop access is again safe,  
 // although the raw address of the object may have changed.  
 // (Don't cache naked oops over safepoints, of course).  
  
 // post monitor waited event. Note that this is past-tense, we are done waiting.  
 if (JvmtiExport::should\_post\_monitor\_waited()) {  
 JvmtiExport::post\_monitor\_waited(jt, this, ret == OS\_TIMEOUT);  
 }  
  
 if (event.should\_commit()) {  
 post\_monitor\_wait\_event(&event, node.\_notifier\_tid, millis, ret == OS\_TIMEOUT);  
 }  
  
 OrderAccess::fence() ;  
  
 assert (Self->\_Stalled != 0, "invariant") ;  
 Self->\_Stalled = 0 ;  
  
 assert (\_owner != Self, "invariant") ;  
 ObjectWaiter::TStates v = node.TState ;  
 if (v == ObjectWaiter::TS\_RUN) {  
 enter (Self) ;  
 } else {  
 guarantee (v == ObjectWaiter::TS\_ENTER || v == ObjectWaiter::TS\_CXQ, "invariant") ;  
 ReenterI (Self, &node) ;  
 node.wait\_reenter\_end(this);  
 }  
  
 // Self has reacquired the lock.  
 // Lifecycle - the node representing Self must not appear on any queues.  
 // Node is about to go out-of-scope, but even if it were immortal we wouldn't  
 // want residual elements associated with this thread left on any lists.  
 guarantee (node.TState == ObjectWaiter::TS\_RUN, "invariant") ;  
 assert (\_owner == Self, "invariant") ;  
 assert (\_succ != Self , "invariant") ;

} // OSThreadWaitState()

jt->set*current*waiting\_monitor(NULL);

guarantee (\_recursions == 0, "invariant") ;  
 \_recursions = save; // restore the old recursion count  
 \_waiters--; // decrement the number of waiters

// Verify a few postconditions  
 assert (*owner == Self , "invariant") ;*  
 *assert (*succ != Self , "invariant") ;  
 assert (((oop)(object()))->mark() == markOopDesc::encode(this), "invariant") ;

if (SyncFlags & 32) {  
 OrderAccess::fence() ;  
 }

// check if the notification happened  
 if (!WasNotified) {  
 // no, it could be timeout or Thread.interrupt() or both  
 // check for interrupt event, otherwise it is timeout  
 if (interruptible && Thread::is*interrupted(Self, true) && !HAS*PENDING*EXCEPTION) {*  
 *TEVENT (Wait - throw IEX from epilog) ;*  
 *THROW(vmSymbols::java*lang\_InterruptedException());  
 }  
 }

// NOTE: Spurious wake up will be consider as timeout.  
 // Monitor notify has precedence over thread interrupt.  
}

Condition  
每个Condition对象都包含着一个等待队列，该队列是Condition对象实现等待/通知功能的关键。  
Condition的实现类是AQS的内部类ConditionObject。  
ConditionObject  
/\*\* First node of condition queue. \*/  
private transient Node firstWaiter;  
/\*\* Last node of condition queue. \*/  
private transient Node lastWaiter;  
等待队列  
等待队列是一个FIFO的队列，在队列的每个节点上都包含了一个线程引用，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，那么该线程将会释放锁、构造成节点加入等待队列并进入等待状态。  
Condition拥有等待队列的首节点firstWaiter和尾节点lastWaiter。  
每个Node都持有同一个队列中下一个Node的引用。

Condition拥有首尾节点的引用，新增节点时仅需将原有的尾节点的nextWaiter指针指向它， 并更新尾节点即可。上述节点引用更新的过程并没有使用CAS保证，因为调用await方法的线程必定时获取了锁的线程。  
在Object的监视器模型上，一个对象拥有一个同步队列和一个等待队列，而并发包中的Lock拥有一个同步队列和多个等待队列。

Condition是AQS同步器的内部类，所以每个Conditions实例都能访问同步器提供的方法。

AQS维护了一个同步队列，一个AQS对应多个Condition，每个Condition维护了一个等待队列。

ConditionObject#await

public final void await() throws InterruptedException {  
 if (Thread.interrupted())  
 throw new InterruptedException();  
// 当前线程构造为Node，加入到等待队列  
 Node node = addConditionWaiter();  
// 释放锁,唤醒同步队列中的后继节点  
 long savedState = fullyRelease(node);  
 int interruptMode = 0;  
// 阻塞，直至被其他线程唤醒或中断  
 while (!isOnSyncQueue(node)) {  
 LockSupport.park(this);  
 if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)  
 break;  
 }  
// 重新获取锁  
 if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)  
 interruptMode = REINTERRUPT;  
// 移出等待队列  
if (node.nextWaiter != null) // clean up if cancelled  
 unlinkCancelledWaiters();  
 if (interruptMode != 0)  
// 被其他线程中断，抛出InterruptedException异常  
 reportInterruptAfterWait(interruptMode);  
}  
ConditionObject#addConditionWaiter  
private Node addConditionWaiter() {  
 Node t = lastWaiter;  
 // If lastWaiter is cancelled, clean out.  
 if (t != null && t.waitStatus != Node.CONDITION) {  
 unlinkCancelledWaiters();  
 t = lastWaiter;  
 }  
 Node node = new Node(Thread.currentThread(), Node.CONDITION);  
 if (t == null)  
 firstWaiter = node;  
 else  
 t.nextWaiter = node;  
 lastWaiter = node;  
 return node;  
}  
AQS#fullyRelease  
/\*\*

* Invokes release with current state value; returns saved state.
* Cancels node and throws exception on failure.
* @param node the condition node for this wait
* @return previous sync state  
  \*/  
  final long fullyRelease(Node node) {  
  boolean failed = true;  
  try {  
   long savedState = getState();  
   if (release(savedState)) {  
   failed = false;  
   return savedState;  
   } else {  
   throw new IllegalMonitorStateException();  
   }  
  } finally {  
   if (failed)  
   node.waitStatus = Node.CANCELLED;  
  }  
  }  
  AQS#isOnSyncQueue  
  /\*\*
* Returns true if a node, always one that was initially placed on
* a condition queue, is now waiting to reacquire on sync queue.
* @param node the node
* @return true if is reacquiring  
  */*  
  *final boolean isOnSyncQueue(Node node) {*  
  *if (node.waitStatus == Node.CONDITION || node.prev == null)*  
   *return false;*  
  *if (node.next != null) // If has successor, it must be on queue*  
   *return true;*  
  */*
  + node.prev can be non-null, but not yet on queue because
  + the CAS to place it on queue can fail. So we have to
  + traverse from tail to make sure it actually made it. It
  + will always be near the tail in calls to this method, and
  + unless the CAS failed (which is unlikely), it will be
  + there, so we hardly ever traverse much.  
    \*/  
    return findNodeFromTail(node);  
    }

/\*\*

* Returns true if node is on sync queue by searching backwards from tail.
* Called only when needed by isOnSyncQueue.
* @return true if present  
  \*/  
  private boolean findNodeFromTail(Node node) {  
  Node t = tail;  
  for (;;) {  
   if (t == node)  
   return true;  
   if (t == null)  
   return false;  
   t = t.prev;  
  }  
  }

ConditionObject#checkInterruptWhileWaiting  
/\*\*

* Checks for interrupt, returning THROW\_IE if interrupted
* before signalled, REINTERRUPT if after signalled, or
* 0 if not interrupted.  
  \*/  
  private int checkInterruptWhileWaiting(Node node) {  
  return Thread.interrupted() ?  
   (transferAfterCancelledWait(node) ? THROW\_IE : REINTERRUPT) :  
   0;  
  }

AQS#acquireQueued  
/\*\*

* Acquires in exclusive uninterruptible mode for thread already in
* queue. Used by condition wait methods as well as acquire.  
  \*
* @param node the node
* @param arg the acquire argument
* @return {@code true} if interrupted while waiting  
  \*/  
  final boolean acquireQueued(final Node node, long 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);  
  }  
  }

ConditionObject#unlinkCancelledWaiters  
/\*\*

* Unlinks cancelled waiter nodes from condition queue.
* Called only while holding lock. This is called when
* cancellation occurred during condition wait, and upon
* insertion of a new waiter when lastWaiter is seen to have
* been cancelled. This method is needed to avoid garbage
* retention in the absence of signals. So even though it may
* require a full traversal, it comes into play only when
* timeouts or cancellations occur in the absence of
* signals. It traverses all nodes rather than stopping at a
* particular target to unlink all pointers to garbage nodes
* without requiring many re-traversals during cancellation
* storms.  
  \*/  
  private void unlinkCancelledWaiters() {  
  Node t = firstWaiter;  
  Node trail = null;  
  while (t != null) {  
   Node next = t.nextWaiter;  
   if (t.waitStatus != Node.CONDITION) {  
   t.nextWaiter = null;  
   if (trail == null)  
   firstWaiter = next;  
   else  
   trail.nextWaiter = next;  
   if (next == null)  
   lastWaiter = trail;  
   }  
   else  
   trail = t;  
   t = next;  
  }  
  }

ConditionObject#reportInterruptAfterWait  
private void reportInterruptAfterWait(int interruptMode)  
 throws InterruptedException {  
 if (interruptMode == THROW\_IE)  
 throw new InterruptedException();  
 else if (interruptMode == REINTERRUPT)  
 selfInterrupt();  
}

ConditionObject#signal  
首先判断当前线程是否获取了锁，然后获取等待队列的首节点，将其移动到同步队列，并使用LockSupport.unpark唤醒节点中的线程。

public final void signal() {  
 if (!isHeldExclusively())  
 throw new IllegalMonitorStateException();  
 Node first = firstWaiter;  
 if (first != null)  
 doSignal(first);  
}

/\*\*

* Removes and transfers nodes until hit non-cancelled one or
* null. Split out from signal in part to encourage compilers
* to inline the case of no waiters.
* @param first (non-null) the first node on condition queue  
  \*/  
  private void doSignal(Node first) {  
  do {  
   if ( (firstWaiter = first.nextWaiter) == null)  
   lastWaiter = null;  
   first.nextWaiter = null;  
  } while (!transferForSignal(first) &&  
   (first = firstWaiter) != null);  
  }

AQS#transferForSignal  
/\*\*

* Transfers a node from a condition queue onto sync queue.
* Returns true if successful.
* @param node the node
* @return true if successfully transferred (else the node was
* cancelled before signal)  
  */*  
  *final boolean transferForSignal(Node node) {*  
  */*
  + If cannot change waitStatus, the node has been cancelled.  
    \*/  
    if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))  
    return false;
* /\*
  + Splice onto queue and try to set waitStatus of predecessor to
  + indicate that thread is (probably) waiting. If cancelled or
  + attempt to set waitStatus fails, wake up to resync (in which
  + case the waitStatus can be transiently and harmlessly wrong).  
    \*/  
    // 等待队列中的头节点移动到了同步队列  
    Node p = enq(node);  
    int ws = p.waitStatus;  
    if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))  
    // 唤醒该节点的线程  
    LockSupport.unpark(node.thread);  
    return true;  
    }

Semaphore（暂缓）  
CyclicBarrier（暂缓）  
CountDownLatch（暂缓）  
Exchanger（暂缓）  
AtomicInteger  
/\*\*

* Atomically increments by one the current value.  
  \*
* @return the updated value  
  \*/  
  public final int incrementAndGet() {  
  return unsafe.getAndAddInt(this, valueOffset, 1) + 1;  
  }

/\*\*

* Atomically adds the given value to the current value of a field
* or array element within the given object o
* at the given offset.  
  \*
* @param o object/array to update the field/element in
* @param offset field/element offset
* @param delta the value to add
* @return the previous value
* @since 1.8  
  \*/  
  public final int getAndAddInt(Object o, long offset, int delta) {  
  int v;  
  do {  
   v = getIntVolatile(o, offset);  
  } while (!compareAndSwapInt(o, offset, v, v + delta));  
  return v;  
  }

/\*\* Volatile version of {@link #getInt(Object, long)} \*/  
public native int getIntVolatile(Object o, long offset);

/\*\*

* Atomically update Java variable to x if it is currently
* holding expected.
* @return true if successful  
  \*/  
  public final native boolean compareAndSwapInt(Object o, long offset,  
   int expected,  
   int x);

ThreadPoolExeuctor  
线程池中持有一组Runnable，称为Worker，包装为Thread，调用Thread#start（作为一个线程去启动）。它们的run方法是一个循环，不断获取用户提交的Runnable并调用Runnable#run（不是启动线程，仅仅是方法调用）。  
状态转换

成员变量  
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));  
private static final int COUNT*BITS = Integer.SIZE - 3;*  
*private static final int CAPACITY = (1 << COUNT*BITS) - 1;

// runState is stored in the high-order bits  
private static final int RUNNING = -1 << COUNT*BITS;*  
*private static final int SHUTDOWN = 0 << COUNT*BITS;  
private static final int STOP = 1 << COUNT*BITS;*  
*private static final int TIDYING = 2 << COUNT*BITS;  
private static final int TERMINATED = 3 << COUNT\_BITS;  
private final BlockingQueue workQueue;

private final ReentrantLock mainLock = new ReentrantLock();

private final HashSet workers = new HashSet();

private final Condition termination = mainLock.newCondition();

private int largestPoolSize;

private long completedTaskCount;

private volatile ThreadFactory threadFactory;

private volatile RejectedExecutionHandler handler;

private volatile long keepAliveTime;

private volatile boolean allowCoreThreadTimeOut;

private volatile int corePoolSize;

private volatile int maximumPoolSize;

private static final RejectedExecutionHandler defaultHandler =  
 new AbortPolicy();  
private static final RuntimePermission shutdownPerm =  
 new RuntimePermission("modifyThread");

/\* The context to be used when executing the finalizer, or null. \*/  
private final AccessControlContext acc;

一个 ctl 变量可以包含两部分信息: 线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount). 由于 int 型的变量是由32位二进制的数构成, 所以用 ctl 的高3位来表示线程池的运行状态, 用低29位来表示线程池内有效线程的数量. 由于这两部分信息在该类中很多地方都会使用到, 所以我们也经常会涉及到要获取其中一个信息的操作, 通常来说, 代表这两个信息的变量的名称直接用他们各自英文单词首字母的组合来表示, 所以, 表示线程池运行状态的变量通常命名为 rs, 表示线程池中有效线程数量的变量通常命名为 wc, 另外, ctl 也通常会简写作 c。  
由于 ctl 变量是由线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)这两个信息组合而成, 所以, 如果知道了这两部分信息各自的数值, 就可以调用下面的 ctlOf() 方法来计算出 ctl 的数值：

// rs: 表示线程池的运行状态 (rs 是 runState中各单词首字母的简写组合)  
// wc: 表示线程池内有效线程的数量 (wc 是 workerCount中各单词首字母的简写组合)  
private static int ctlOf(int rs, int wc) { return rs | wc; }  
反过来, 如果知道了 ctl 的值, 那么也可以通过如下的 runStateOf() 和 workerCountOf() 两个方法来分别获取线程池的运行状态和线程池内有效线程的数量.  
private static int runStateOf(int c) { return c & ~CAPACITY; }  
private static int workerCountOf(int c) { return c & CAPACITY; }  
其中, CAPACITY 等于 (2^29)-1, 也就是高3位是0, 低29位是1的一个int型的数,  
private static final int COUNT*BITS = Integer.SIZE - 3; // 29*  
*private static final int CAPACITY = (1 << COUNT*BITS) - 1; // COUNT\_BITS == 29  
执行任务

当提交一个新任务到线程池时，线程池的处理流程如下：  
1）线程池判断当前运行的线程是否少于corePoolSize（需要获取全局锁），如果不是，则创建一个新的工作线程来执行任务（分配线程）。如果是，则进入2）  
2）线程池判断工作队列是否已经满了，如果没有满，则将新提交的任务存储到这个工作队列BlockingQueue里。如果满了，则进入3）  
3）线程池判断创建新的线程是否会使当前运行的线程超过maxPoolSize（需要获取全局锁），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。

ThreadPoolExecutor采取上述步骤的总体设计思路，是为了在执行execute方法时，尽可能地避免获取全局锁。在ThreadPoolExecutor完成预热之后（当前运行的线程数大于等于corePoolSize），几乎所有的execute方法都是执行步骤2，步骤2不需要获取全局锁。  
execute（Runnable不进行任何封装）  
public void execute(Runnable command) {  
 if (command == null)  
 throw new NullPointerException();  
 int c = ctl.get();  
// 如果当前运行线程数小于corePoolSize  
 if (workerCountOf(c) < corePoolSize) {  
// 创建线程并执行当前任务  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
 }  
// 如果线程池处于运行状态，且（当前运行线程数大于等于corePoolSize或线程创建失败），则将当前任务放入工作队列  
 if (isRunning(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();  
// 再次检查线程池的状态，如果线程池没有运行，且成功从工作队列中删除任务，则执行reject处理任务  
 if (!isRunning(recheck) && remove(command))  
 reject(command);  
 else if (workerCountOf(recheck) == 0)  
 addWorker(null, false);  
 }  
// 如果线程池不处于运行中或任务无法放进队列，并且当前线程数小于maxPoolSize，则创建一个线程执行任务  
 else if (!addWorker(command, false))  
// 创建线程失败，则执行reject处理任务  
 reject(command);  
}

1) addWorker  
addWorker主要负责创建新的线程并执行任务  
private boolean addWorker(Runnable firstTask, boolean core) {  
 retry:  
 for (;;) {  
 int c = ctl.get();  
 int rs = runStateOf(c);

// Check if queue empty only if necessary.  
 if (rs >= SHUTDOWN &&  
 ! (rs == SHUTDOWN &&  
 firstTask == null &&  
 ! workQueue.isEmpty()))  
 return false;  
  
 for (;;) {  
 int wc = workerCountOf(c);  
 if (wc >= CAPACITY ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;  
 if (compareAndIncrementWorkerCount(c))  
 break retry;  
 c = ctl.get(); // Re-read ctl  
 if (runStateOf(c) != rs)  
 continue retry;  
 // else CAS failed due to workerCount change; retry inner loop  
 }  
}  
  
boolean workerStarted = false;  
boolean workerAdded = false;  
Worker w = null;  
try {  
 w = new Worker(firstTask);  
 final Thread t = w.thread;  
 if (t != null) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 // Recheck while holding lock.  
 // Back out on ThreadFactory failure or if  
 // shut down before lock acquired.  
 int rs = runStateOf(ctl.get());  
  
 if (rs < SHUTDOWN ||  
 (rs == SHUTDOWN && firstTask == null)) {  
 if (t.isAlive()) // precheck that t is startable  
 throw new IllegalThreadStateException();  
 workers.add(w);  
 int s = workers.size();  
 if (s > largestPoolSize)  
 largestPoolSize = s;  
 workerAdded = true;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
 if (workerAdded) {  
 t.start();  
 workerStarted = true;  
 }  
 }  
} finally {  
 if (! workerStarted)  
 addWorkerFailed(w);  
}  
return workerStarted;

}

1.1) Worker（基于AQS）  
当addWorker中调用t.start()时，这个t是Worker构造方法中使用ThreadFactory创建出来的Thread，且将this作为Runnable传入，启动t时会调用Worker#run方法。

private final class Worker  
 extends AbstractQueuedSynchronizer  
 implements Runnable  
{  
 /\*\*  
 \* This class will never be serialized, but we provide a  
 \* serialVersionUID to suppress a javac warning.  
 \*/  
 private static final long serialVersionUID = 6138294804551838833L;

/\*\* Thread this worker is running in. Null if factory fails. \*/  
final Thread thread;  
/\*\* Initial task to run. Possibly null. \*/  
Runnable firstTask;  
/\*\* Per-thread task counter \*/  
volatile long completedTasks;  
  
/\*\*  
 \* Creates with given first task and thread from ThreadFactory.  
 \* @param firstTask the first task (null if none)  
 \*/  
Worker(Runnable firstTask) {  
 setState(-1); // inhibit interrupts until runWorker  
 this.firstTask = firstTask;  
 this.thread = getThreadFactory().newThread(this);  
}  
  
/\*\* Delegates main run loop to outer runWorker \*/  
public void run() {  
 runWorker(this);  
}  
// ...

}

1.2) Worker#runWorker  
final void runWorker(Worker w) {  
 Thread wt = Thread.currentThread();  
 Runnable task = w.firstTask;  
 w.firstTask = null;  
 w.unlock(); // allow interrupts  
 boolean completedAbruptly = true;  
 try {  
 while (task != null || (task = getTask()) != null) {  
 w.lock();  
 // If pool is stopping, ensure thread is interrupted;  
 // if not, ensure thread is not interrupted. This  
 // requires a recheck in second case to deal with  
 // shutdownNow race while clearing interrupt  
 if ((runStateAtLeast(ctl.get(), STOP) ||  
 (Thread.interrupted() &&  
 runStateAtLeast(ctl.get(), STOP))) &&  
 !wt.isInterrupted())  
 wt.interrupt();  
 try {  
 beforeExecute(wt, task);  
 Throwable thrown = null;  
 try {  
 task.run();  
 } catch (RuntimeException x) {  
 thrown = x; throw x;  
 } catch (Error x) {  
 thrown = x; throw x;  
 } catch (Throwable x) {  
 thrown = x; throw new Error(x);  
 } finally {  
 afterExecute(task, thrown);  
 }  
 } finally {  
 task = null;  
 w.completedTasks++;  
 w.unlock();  
 }  
 }  
 completedAbruptly = false;  
 } finally {  
 processWorkerExit(w, completedAbruptly);  
 }  
}

1.2.1) ThreadPoolExecutor#getTask  
1）判断线程池是否已经关闭，如果关闭，则退出循环  
2）根据当前Worker是否超时，对工作队列调用poll或take方法，进行阻塞。阻塞中的线程就是空闲线程。

当调用shutdown方法时，首先设置了线程池的状态为ShutDown，此时1阶段的worker进入到状态判断时会返回null，此时Worker退出。  
因为getTask的时候是不加锁的，所以在shutdown时可以调用worker.Interrupt.此时会中断退出，Loop到状态判断时，同时workQueue为empty。那么抛出中断异常，导致重新Loop，在检测线程池状态时，Worker退出。如果workQueue不为null就不会退出，此处有些疑问，因为没有看见中断标志位清除的逻辑，那么这里就会不停的循环直到workQueue为Empty退出。  
这里也能看出来SHUTDOWN只是清除一些空闲Worker，并且拒绝新Task加入，对于workQueue中的线程还是继续处理的。  
对于shutdown中获取mainLock而addWorker中也做了mainLock的获取，这么做主要是因为Works是HashSet类型的，是线程不安全的，我们也看到在addWorker后面也是对线程池状态做了判断，将Worker添加和中断逻辑分离开。

timed变量主要是标识着当前Worker超时是否要退出。wc > corePoolSize时需要减小空闲的Worker数，那么timed为true，但是wc <= corePoolSize时，不能减小核心线程数timed为false。  
timedOut初始为false，如果timed为true那么使用poll取线程。如果正常返回，那么返回取到的task。如果超时，证明worker空闲，同时worker超过了corePoolSize，需要删除。返回r=null。则 timedOut = true。此时循环到wc <= maximumPoolSize && ! (timedOut && timed)时，减小worker数，并返回null，导致worker退出。如果线程数<= corePoolSize，那么此时调用 workQueue.take()，没有线程获取到时将一直阻塞，直到获取到线程或者中断。  
private Runnable getTask() {  
 boolean timedOut = false; // Did the last poll() time out?

for (;;) {  
 int c = ctl.get();  
 int rs = runStateOf(c);  
  
 // Check if queue empty only if necessary.  
 if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {  
 decrementWorkerCount();  
 return null;  
 }  
  
 int wc = workerCountOf(c);  
 // Worker是否要减少  
 // Are workers subject to culling?  
 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;  
  
 if ((wc > maximumPoolSize || (timed && timedOut))  
 && (wc > 1 || workQueue.isEmpty())) {  
 if (compareAndDecrementWorkerCount(c))  
 return null;  
 continue;  
 }  
 // 如果要减少Worker的话，如果在keepAliveTime内没有拿到任务，那么设置为超时，下次循环被会移除；如果不需要减少Worker，那么阻塞获取任务  
 try {  
 Runnable r = timed ?  
 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :  
 workQueue.take();  
 if (r != null)  
 return r;  
 timedOut = true;  
 } catch (InterruptedException retry) {  
 timedOut = false;  
 }  
}

}

2) reject  
final void reject(Runnable command) {  
 handler.rejectedExecution(command, this);  
}

2.1) AbortPolicy#rejectedExecution  
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 throw new RejectedExecutionException("Task " + r.toString() +  
 " rejected from " +  
 e.toString());  
}

2.2) DiscardPolicy#rejectedExecution  
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
}

2.3) DiscardOldestPolicy#rejectedExecution  
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 if (!e.isShutdown()) {  
 e.getQueue().poll();  
 e.execute(r);  
 }  
}

2.4) CallerRunsPolicy#rejectedExecution  
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 if (!e.isShutdown()) {  
 r.run();  
 }  
}

submit（Callable包装为FutureTask）  
public Future submit(Callable task) {  
 if (task == null) throw new NullPointerException();  
 RunnableFuture ftask = newTaskFor(task);  
 execute(ftask);  
 return ftask;  
}  
1) newTaskFor（将Callable包装成Runnable+Future，Runnable可以放在ThreadPoolExecutor中执行）  
protected RunnableFuture newTaskFor(Callable callable) {  
 return new FutureTask(callable);  
}

public FutureTask(Callable callable) {  
 if (callable == null)  
 throw new NullPointerException();  
 this.callable = callable;  
 this.state = NEW; // ensure visibility of callable  
}

1.1) FutureTask#run  
public void run() {  
 if (state != NEW ||  
 !UNSAFE.compareAndSwapObject(this, runnerOffset,  
 null, Thread.currentThread()))  
 return;  
 try {  
 Callable c = callable;  
 if (c != null && state == NEW) {  
 V result;  
 boolean ran;  
 try {  
 result = c.call();  
 ran = true;  
 } catch (Throwable ex) {  
 result = null;  
 ran = false;  
 setException(ex);  
 }  
 if (ran)  
 set(result);  
 }  
 } finally {  
 // runner must be non-null until state is settled to  
 // prevent concurrent calls to run()  
 runner = null;  
 // state must be re-read after nulling runner to prevent  
 // leaked interrupts  
 int s = state;  
 if (s >= INTERRUPTING)  
 handlePossibleCancellationInterrupt(s);  
 }  
}  
1.1.1) FutureTask#set  
protected void set(V v) {  
 if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {  
 outcome = v;  
 UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state  
 finishCompletion();  
 }  
}  
1.1.2) FutureTask#setException  
protected void setException(Throwable t) {  
 if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {  
 outcome = t;  
 UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state  
 finishCompletion();  
 }  
}

1.1.1.1) FutureTask#finishCompletion（唤醒Waiter）  
private void finishCompletion() {  
 // assert state > COMPLETING;  
 for (WaitNode q; (q = waiters) != null;) {  
 if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {  
 for (;;) {  
 Thread t = q.thread;  
 if (t != null) {  
 q.thread = null;  
 LockSupport.unpark(t);  
 }  
 WaitNode next = q.next;  
 if (next == null)  
 break;  
 q.next = null; // unlink to help gc  
 q = next;  
 }  
 break;  
 }  
 }

done();  
  
callable = null; // to reduce footprint

}

2) FutureTask#get  
public V get() throws InterruptedException, ExecutionException {  
 int s = state;  
 if (s <= COMPLETING)  
// 如果尚未执行完毕，则等待  
 s = awaitDone(false, 0L);  
 return report(s);  
}

/\*\*

* Simple linked list nodes to record waiting threads in a Treiber
* stack. See other classes such as Phaser and SynchronousQueue
* for more detailed explanation.  
  \*/  
  static final class WaitNode {  
  volatile Thread thread;  
  volatile WaitNode next;  
  WaitNode() { thread = Thread.currentThread(); }  
  }

2.1) FutureTask#awaitDone（添加并阻塞Waiter）  
private int awaitDone(boolean timed, long nanos)  
 throws InterruptedException {  
 final long deadline = timed ? System.nanoTime() + nanos : 0L;  
 WaitNode q = null;  
 boolean queued = false;  
 for (;;) {  
 if (Thread.interrupted()) {  
 removeWaiter(q);  
 throw new InterruptedException();  
 }

int s = state;  
 if (s > COMPLETING) {  
 if (q != null)  
 q.thread = null;  
 return s;  
 }  
 else if (s == COMPLETING) // cannot time out yet  
 Thread.yield();  
 else if (q == null)  
 q = new WaitNode();  
 else if (!queued)

// 添加Waiter，以待被唤醒  
 queued = UNSAFE.compareAndSwapObject(this, waitersOffset,  
 q.next = waiters, q);  
 else if (timed) {  
 nanos = deadline - System.nanoTime();  
 if (nanos <= 0L) {  
 removeWaiter(q);  
 return state;  
 }  
 LockSupport.parkNanos(this, nanos);  
 }  
 else  
 LockSupport.park(this);  
 }  
}

2.2) FutureTask#report（抛出执行时的异常）  
private V report(int s) throws ExecutionException {  
 Object x = outcome;  
 if (s == NORMAL)  
 return (V)x;  
 if (s >= CANCELLED)  
 throw new CancellationException();  
 throw new ExecutionException((Throwable)x);  
}

3) FutureTask#cancel（实际上还是中断线程）  
public boolean cancel(boolean mayInterruptIfRunning) {  
 if (!(state == NEW &&  
 UNSAFE.compareAndSwapInt(this, stateOffset, NEW,  
 mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))  
 return false;  
 try { // in case call to interrupt throws exception  
 if (mayInterruptIfRunning) {  
 try {  
 Thread t = runner;  
 if (t != null)  
 t.interrupt();  
 } finally { // final state  
 UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);  
 }  
 }  
 } finally {  
 finishCompletion();  
 }  
 return true;  
}

关闭线程池  
shutdown方法会遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。  
shutdownNow方法会首先将线程池的状态设置为STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置为SHUTDOWN状态，然后中断所有没有在执行任务的线程。  
只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都关闭后，才表示线程池关闭成功，这时调用isTerminated方法会返回true。  
shutdown

public void shutdown() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
// 判断是否可以操作目标线程  
 checkShutdownAccess();  
// 设置线程池状态为SHUTDOWN，以后线程池不会执行新的任务  
 advanceRunState(SHUTDOWN);  
// 中断所有的空闲线程  
 interruptIdleWorkers();  
 onShutdown(); // hook for ScheduledThreadPoolExecutor  
 } finally {  
 mainLock.unlock();  
 }  
// 转到Terminate  
 tryTerminate();  
}

1) interruptIdleWorkers  
private void interruptIdleWorkers() {  
 interruptIdleWorkers(false);  
}  
中断worker，但是中断之前需要先获取锁，这就意味着正在运行的Worker不能中断。但是上面的代码有w.tryLock()，那么获取不到锁就不会中断，shutdown的Interrupt只是对所有的空闲Worker（正在从workQueue中取Task，此时Worker没有加锁）发送中断信号。  
private void interruptIdleWorkers(boolean onlyOne) {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 for (Worker w : workers) {  
 Thread t = w.thread;  
 if (!t.isInterrupted() && w.tryLock()) {  
//tryLock能获取到的Worker都是空闲的Worker，因为Worker在执行任务时是要拿到Worker的Lock的  
 try {  
// 让阻塞在工作队列中的Worker中断  
 t.interrupt();  
 } catch (SecurityException ignore) {  
 } finally {  
 w.unlock();  
 }  
 }  
 if (onlyOne)  
 break;  
 }  
 } finally {  
 mainLock.unlock();  
 }  
}

shutdownNow  
public List shutdownNow() {  
 List tasks;  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 checkShutdownAccess();  
 advanceRunState(STOP);  
 interruptWorkers();  
// 工作队列中没有执行的任务全部抛弃  
 tasks = drainQueue();  
 } finally {  
 mainLock.unlock();  
 }  
 tryTerminate();  
 return tasks;  
}  
1) interruptWorkers  
private void interruptWorkers() {  
 final ReentrantLock mainLock = this.mainLock;  
 mainLock.lock();  
 try {  
 for (Worker w : workers)  
 w.interruptIfStarted();  
 } finally {  
 mainLock.unlock();  
 }  
}

Worker#interruptIfStarted  
void interruptIfStarted() {  
 Thread t;  
 if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {  
 try {  
 t.interrupt();  
 } catch (SecurityException ignore) {  
 }  
 }  
}

1.1) drainQueue  
\*/  
private List drainQueue() {  
 BlockingQueue q = workQueue;  
 ArrayList taskList = new ArrayList();  
 q.drainTo(taskList);  
 if (!q.isEmpty()) {  
 for (Runnable r : q.toArray(new Runnable[0])) {  
 if (q.remove(r))  
 taskList.add(r);  
 }  
 }  
 return taskList;  
}

1.2) tryTerminate（两种关闭都会调用）TIDYING和TERMINATED的转化  
有几种状态是不能转化到TIDYING（整理中）的:  
RUNNING状态  
TIDYING或TERMINATED  
SHUTDOWN状态，但是workQueue不为空

也说明了两点：

1. SHUTDOWN想转化为TIDYING，需要workQueue为空，同时workerCount为0。
2. STOP转化为TIDYING，需要workerCount为0  
   如果满足上面的条件（一般一定时间后都会满足的），那么CAS成TIDYING，TIDYING也只是个过渡状态，最终会转化为TERMINATED。  
   final void tryTerminate() {  
    for (;;) {  
    int c = ctl.get();  
    if (isRunning(c) ||  
    runStateAtLeast(c, TIDYING) ||  
    (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))  
    return;  
    if (workerCountOf(c) != 0) { // Eligible to terminate  
    interruptIdleWorkers(ONLY\_ONE);  
    return;  
    }

* final ReentrantLock mainLock = this.mainLock;  
   mainLock.lock();  
   try {  
   if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {  
   try {  
   terminated();  
   } finally {  
   ctl.set(ctlOf(TERMINATED, 0));  
   termination.signalAll();  
   }  
   return;  
   }  
   } finally {  
   mainLock.unlock();  
   }  
   // else retry on failed CAS
* }  
  }

ScheduledThreadPoolExecutor（继承ThreadPoolExecutor）  
与ThreadPoolExecutor的区别：  
1）使用DelayedWorkQueue作为任务队列  
2）获取任务的方式不同  
3）执行周期任务后，增加了额外的处理  
成员变量  
/\*\*

* False if should cancel/suppress periodic tasks on shutdown.  
  \*/  
  private volatile boolean continueExistingPeriodicTasksAfterShutdown;

/\*\*

* False if should cancel non-periodic tasks on shutdown.  
  \*/  
  private volatile boolean executeExistingDelayedTasksAfterShutdown = true;

/\*\*

* True if ScheduledFutureTask.cancel should remove from queue  
  \*/  
  private volatile boolean removeOnCancel = false;

/\*\*

* Sequence number to break scheduling ties, and in turn to
* guarantee FIFO order among tied entries.  
  \*/  
  private static final AtomicLong sequencer = new AtomicLong();

构造方法  
/\*\*

* Creates a new {@code ScheduledThreadPoolExecutor} with the
* given core pool size.  
  \*
* @param corePoolSize the number of threads to keep in the pool, even
* if they are idle, unless {@code allowCoreThreadTimeOut} is set
* @throws IllegalArgumentException if {@code corePoolSize < 0}  
  \*/  
  public ScheduledThreadPoolExecutor(int corePoolSize) {  
  super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,  
   new DelayedWorkQueue());  
  }

/\*\*

* Creates a new {@code ScheduledThreadPoolExecutor} with the
* given initial parameters.  
  \*
* @param corePoolSize the number of threads to keep in the pool, even
* if they are idle, unless {@code allowCoreThreadTimeOut} is set
* @param threadFactory the factory to use when the executor
* creates a new thread
* @throws IllegalArgumentException if {@code corePoolSize < 0}
* @throws NullPointerException if {@code threadFactory} is null  
  \*/  
  public ScheduledThreadPoolExecutor(int corePoolSize,  
   ThreadFactory threadFactory) {  
  super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,  
   new DelayedWorkQueue(), threadFactory);  
  }

/\*\*

* Creates a new ScheduledThreadPoolExecutor with the given
* initial parameters.  
  \*
* @param corePoolSize the number of threads to keep in the pool, even
* if they are idle, unless {@code allowCoreThreadTimeOut} is set
* @param handler the handler to use when execution is blocked
* because the thread bounds and queue capacities are reached
* @throws IllegalArgumentException if {@code corePoolSize < 0}
* @throws NullPointerException if {@code handler} is null  
  \*/  
  public ScheduledThreadPoolExecutor(int corePoolSize,  
   RejectedExecutionHandler handler) {  
  super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,  
   new DelayedWorkQueue(), handler);  
  }

/\*\*

* Creates a new ScheduledThreadPoolExecutor with the given
* initial parameters.  
  \*
* @param corePoolSize the number of threads to keep in the pool, even
* if they are idle, unless {@code allowCoreThreadTimeOut} is set
* @param threadFactory the factory to use when the executor
* creates a new thread
* @param handler the handler to use when execution is blocked
* because the thread bounds and queue capacities are reached
* @throws IllegalArgumentException if {@code corePoolSize < 0}
* @throws NullPointerException if {@code threadFactory} or
* {@code handler} is null  
   \*/  
  public ScheduledThreadPoolExecutor(int corePoolSize,  
   ThreadFactory threadFactory,  
   RejectedExecutionHandler handler) {  
  super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,  
   new DelayedWorkQueue(), threadFactory, handler);  
  }

DelayedWorkQueue（底层是堆，无界阻塞队列，存放RunnableScheduledFuture）  
static class DelayedWorkQueue extends AbstractQueue  
 implements BlockingQueue {}

put  
public void put(Runnable e) {  
 offer(e);  
}

public boolean offer(Runnable x) {  
 if (x == null)  
 throw new NullPointerException();  
 RunnableScheduledFuture)x;  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 int i = size;  
 if (i >= queue.length)  
 grow();  
 size = i + 1;  
 if (i == 0) {  
 queue[0] = e;  
 setIndex(e, 0);  
 } else {  
 siftUp(i, e);  
 }  
 if (queue[0] == e) {  
 leader = null;  
 available.signal();  
 }  
 } finally {  
 lock.unlock();  
 }  
 return true;  
}

private void siftUp(int k, RunnableScheduledFuture<?> key) {  
 while (k > 0) {  
 int parent = (k - 1) >>> 1;  
 RunnableScheduledFuture<?> e = queue[parent];  
 if (key.compareTo(e) >= 0)  
 break;  
 queue[k] = e;  
 setIndex(e, k);  
 k = parent;  
 }  
 queue[k] = key;  
 setIndex(key, k);  
}

ScheduledFutureTask是RunnableScheduledFuture的唯一实现类，它实现了Comparable接口。先按照time（下一次运行的时间）比较，然后按seq比较，最后按delay比较。  
public int compareTo(Delayed other) {  
 if (other == this) // compare zero if same object  
 return 0;  
 if (other instanceof ScheduledFutureTask) {  
 ScheduledFutureTask)other;  
 long diff = time - x.time;  
 if (diff < 0)  
 return -1;  
 else if (diff > 0)  
 return 1;  
 else if (sequenceNumber < x.sequenceNumber)  
 return -1;  
 else  
 return 1;  
 }  
 long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECONDS);  
 return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;  
}

take（阻塞获取）  
public RunnableScheduledFuture<?> take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 for (;;) {  
 RunnableScheduledFuture<?> first = queue[0];  
 if (first == null)  
// 队列为空，则阻塞  
 available.await();  
 else {  
 long delay = first.getDelay(NANOSECONDS);  
// 已经过期，则移除  
 if (delay <= 0)  
 return finishPoll(first);  
 first = null; // don't retain ref while waiting  
 if (leader != null)  
 available.await();  
 else {  
 Thread thisThread = Thread.currentThread();  
 leader = thisThread;  
 try {  
// 未过期，则等待相应时间  
 available.awaitNanos(delay);  
 } finally {  
 if (leader == thisThread)  
 leader = null;  
 }  
 }  
 }  
 }  
 } finally {  
 if (leader == null && queue[0] != null)  
 available.signal();  
 lock.unlock();  
 }  
}

定期执行任务  
执行主要分为两大部分：  
1）当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或  
scheduleAtFixedDelay()方法时，会向DelayedWorkQueue添加一个实现了RunnableScheduledFuture接口的ScheduledFutureTask。  
2）线程池中的线程从DelayedWorkQueue中获取ScheduledFutureTask，然后执行任务。

scheduleAtFixedRate（Runnable包装为ScheduledFutureTask）  
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
 long initialDelay,  
 long period,  
 TimeUnit unit) {  
 if (command == null || unit == null)  
 throw new NullPointerException();  
 if (period <= 0)  
 throw new IllegalArgumentException();  
// 将Runnable包装成ScheduledFutureTask，它实现了RunnableScheduledFuture接口  
 ScheduledFutureTask sft =  
 new ScheduledFutureTask(command,  
 null,  
 triggerTime(initialDelay, unit),  
 unit.toNanos(period));  
 RunnableScheduledFuture t = decorateTask(command, sft);  
 sft.outerTask = t;  
 delayedExecute(t);  
 return t;  
}  
1) ScheduledFutureTask  
private class ScheduledFutureTask  
 extends FutureTask implements RunnableScheduledFuture {}  
成员变量  
/\*\* Sequence number to break ties FIFO \*/  
private final long sequenceNumber;

/\*\* The time the task is enabled to execute in nanoTime units \*/  
private long time;

/\*\*

* Period in nanoseconds for repeating tasks. A positive
* value indicates fixed-rate execution. A negative value
* indicates fixed-delay execution. A value of 0 indicates a
* non-repeating task.  
  \*/  
  private final long period;

/\*\* The actual task to be re-enqueued by reExecutePeriodic \*/  
RunnableScheduledFuture outerTask = this;

/\*\*

* Index into delay queue, to support faster cancellation.  
  \*/  
  int heapIndex;  
  构造方法  
  ScheduledFutureTask(Runnable r, V result, long ns, long period) {  
  super(r, result);  
  this.time = ns;  
  this.period = period;  
  this.sequenceNumber = sequencer.getAndIncrement();  
  }

public FutureTask(Runnable runnable, V result) {  
 this.callable = Executors.callable(runnable, result);  
 this.state = NEW; // ensure visibility of callable  
}

public static Callable callable(Runnable task, T result) {  
 if (task == null)  
 throw new NullPointerException();  
 return new RunnableAdapter(task, result);  
}

static final class RunnableAdapter implements Callable {  
 final Runnable task;  
 final T result;  
 RunnableAdapter(Runnable task, T result) {  
 this.task = task;  
 this.result = result;  
 }  
 public T call() {  
 task.run();  
 return result;  
 }  
}

2) delayedExecute（入队）  
private void delayedExecute(RunnableScheduledFuture<?> task) {  
 if (isShutdown())  
 reject(task);  
 else {  
// 加入到任务队列中  
 super.getQueue().add(task);  
 if (isShutdown() &&  
 !canRunInCurrentRunState(task.isPeriodic()) &&  
 remove(task))  
 task.cancel(false);  
 else  
// 确保至少一个线程在处理任务，即使核心线程数corePoolSize为0  
 ensurePrestart();  
 }  
}  
2.1) ensurePrestart（添加工作线程直至corePoolSize）  
void ensurePrestart() {  
 int wc = workerCountOf(ctl.get());  
 if (wc < corePoolSize)  
 addWorker(null, true);  
 else if (wc == 0)  
 addWorker(null, false);  
}

3) ScheduledFutureTask#run  
public void run() {  
// 判断是不是定时周期调度的  
 boolean periodic = isPeriodic();  
 if (!canRunInCurrentRunState(periodic))  
 cancel(false);  
 else if (!periodic)  
//调用FutureTask的run方法  
 ScheduledFutureTask.super.run();  
 else if (ScheduledFutureTask.super.runAndReset()) {  
//计算下一次执行时间  
 setNextRunTime();  
// 重新入队  
 reExecutePeriodic(outerTask);  
 }  
}

private void setNextRunTime() {  
 long p = period;  
 if (p > 0)  
 time += p;  
 else  
 time = triggerTime(-p);  
}

void reExecutePeriodic(RunnableScheduledFuture<?> task) {  
 if (canRunInCurrentRunState(true)) {  
 super.getQueue().add(task);  
 if (!canRunInCurrentRunState(true) && remove(task))  
 task.cancel(false);  
 else  
 ensurePrestart();  
 }  
}

练习题  
生产者消费者几种实现方式

wait&notify  
public class TestProducerConsumer {  
 public static void main(String[] args) {  
 SyncStack ss = new SyncStack();  
 Producer p = new Producer(ss);  
 Consumer c = new Consumer(ss);  
 new Thread(p, "A").start();  
 new Thread(p, "B").start();  
 new Thread(c).start();  
 }  
}

class Food {  
 private String id;

public Food(String id) {  
 this.id = id;  
}  
  
public String toString() {  
 return "产品" + id;  
}

}

class SyncStack {  
 private int index = 0;  
 private Food[] foods = new Food[6];

public SyncStack() {  
}  
  
public synchronized void push(Food f) {  
 while (index == foods.length) {  
 try {  
 System.out.println("容器已满");  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 foods[index] = f;  
 index++;  
 this.notifyAll();  
}  
  
public synchronized Food pop() {  
 while (index == 0) {  
 try {  
 System.out.println("容器已空");  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 index--;  
 this.notifyAll();  
 return foods[index];  
}

}

class Producer implements Runnable {  
 private SyncStack ss;

public Producer(SyncStack ss) {  
 this.ss = ss;  
}  
  
public void run() {  
 for (int i = 0; i < 10; i++) {  
 Food f = new Food(Thread.currentThread().getName() + i);  
 ss.push(f);  
 System.out.println("生产者"+Thread.currentThread().getName() + "生产了 " + f);  
 try {  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

class Consumer implements Runnable {  
 private SyncStack ss;

public Consumer(SyncStack ss) {  
 this.ss = ss;  
}  
  
public void run() {  
 for (int i = 0; i < 20; i++) {  
 Food f = ss.pop();  
 System.out.println("消费了 " + f);  
 try {  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

Lock&Condition  
public class TestProducerConsumer {  
 public static void main(String[] args) {  
 SyncStack ss = new SyncStack();  
Producer p = new Producer(ss);  
Consumer c = new Consumer(ss);  
 new Thread(p, "A").start();  
 new Thread(p, "B").start();  
 new Thread(c, "C").start();  
 new Thread(c, "D").start();  
 }  
}

class Food {  
 private String id;

public Food(String id) {  
 this.id = id;  
}  
  
public String toString() {  
 return "产品" + id;  
}

}

class SyncStack {  
 private int index = 0;  
 private Food[] foods = new Food[6];  
 private Lock lock = new ReentrantLock();  
 private Condition condition = lock.newCondition();

public SyncStack() {  
}  
  
public void push(Food f) {  
 lock.lock();  
 try {  
 while (index == foods.length) {  
 try {  
 System.out.println("容器已满");  
 condition.await();  
 //相当于this.wait()  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 foods[index] = f;  
 index++;  
 condition.signalAll();  
 //相当于this.notifyAll()  
 } finally {  
 lock.unlock();  
 }  
}  
  
public Food pop() {  
 lock.lock();  
 try {  
 while (index == 0) {  
 try {  
 System.out.println("容器已空");  
 condition.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 index--;  
 condition.signalAll();  
 } finally {  
 lock.unlock();  
 }  
 return foods[index];  
}

}

class Producer implements Runnable {  
 private SyncStack ss;

public Producer(SyncStack ss) {  
 this.ss = ss;  
}  
  
public void run() {  
 for (int i = 0; i < 10; i++) {  
 Food f = new Food(Thread.currentThread().getName() + i);  
 ss.push(f);  
 System.out.println("生产者" + Thread.currentThread().getName() + "生产了 " + f);  
 try {  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

class Consumer implements Runnable {  
 private SyncStack ss;

public Consumer(SyncStack ss) {  
 this.ss = ss;  
}  
  
public void run() {  
 for (int i = 0; i < 10; i++) {  
 Food f = ss.pop();  
 System.out.println("消费者" + Thread.currentThread().getName() + "消费了 " + f);  
 try {  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

BlockingQueue  
public class TestProducerConsumer {  
 public static void main(String[] args) {  
 BlockingQueue queue = new ArrayBlockingQueue(6);  
Producer p = new Producer(queue);  
Consumer c = new Consumer(queue);  
 new Thread(p, "A").start();  
 new Thread(p, "B").start();  
 new Thread(c, "C").start();  
 new Thread(c, "D").start();  
 }  
}

class Food {  
 private String id;

public Food(String id) {  
 this.id = id;  
}  
  
public String toString() {  
 return "产品" + id;  
}

}

class Producer implements Runnable {  
 private BlockingQueue foods;

public Producer(BlockingQueue<Food> foods) {  
 this.foods = foods;  
}  
  
public void run() {  
 for (int i = 0; i < 10; i++) {  
 Food f = new Food(Thread.currentThread().getName() + i);  
 try {  
 foods.put(f);  
 System.out.println("生产者" + Thread.currentThread().getName() + "生产了 " + f);  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

class Consumer implements Runnable {

private BlockingQueue<Food> foods;  
  
public Consumer(BlockingQueue<Food> foods) {  
 this.foods = foods;  
}  
  
public void run() {  
 for (int i = 0; i < 10; i++) {  
 try {  
 Food f = foods.take();  
 System.out.println("消费者" + Thread.currentThread().getName() + "消费了 " + f);  
 Thread.sleep((int) (Math.random() \* 1000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

}

线程按序交替执行  
设置3个线程，线程名分别为123，按照123的顺序打印，重复20遍。  
public class TestAlternate {  
 public static void main(String[] args) {  
 int threadNum = 3;  
 int loopTimes = 20;  
 AlternativeDemo atomicDemo = new AlternativeDemo(threadNum, loopTimes);  
 for (int i = 1; i <= threadNum; ++i) {  
 new Thread(atomicDemo, String.valueOf(i)).start();  
 }  
 }  
 // 所有线程共享lock和conditions  
 private static class AlternativeDemo implements Runnable {  
 private int nextThread = 1;  
 private Lock lock = new ReentrantLock();  
 private Condition[] conditions;  
 private int totalTimes;

public AlternativeDemo(int threadNum, int totalTimes) {  
 this.totalTimes = totalTimes;  
 this.conditions = new Condition[threadNum];  
 for (int i = 0; i < threadNum; ++i) {  
 conditions[i] = lock.newCondition();  
 }  
 }  
  
 public void run() {  
 for (int i = 1; i <= totalTimes; ++i) {  
 lock.lock();  
 // currentThread 取值为1,2,3   
 // currentThread-1为当前线程对应的Condition  
 int currentThread = Thread.currentThread().getName().charAt(0) - '0';  
 try {

// 下一个不是自己，则等待  
 if (currentThread != nextThread) {  
 conditions[currentThread - 1].await();  
 }  
 System.out.println("线程" + currentThread + ":" + currentThread);   
// 计算下一个要打印的线程  
 // 3 % 3 + 1 = 1 线程3后面的是线程1  
 nextThread = nextThread % conditions.length + 1;  
// 唤醒下一个要打印的线程  
 conditions[nextThread - 1].signal();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 }  
 }  
}

线程同步的基本使用习题  
public class Test8Questions {

public static void main(String[] args) {  
 Number number = new Number();  
 Number number2 = new Number();  
   
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 number.getOne();  
 }   
 }).start();  
   
 new Thread(new Runnable() {  
 @Override  
 public void run() {

// number.getTwo();  
 number2.getTwo();  
 }  
 }).start();

/\*new Thread(new Runnable() {  
 @Override  
 public void run() {  
 number.getThree();  
 }  
 }).start();\*/  
   
}

}

class Number{

public static synchronized void getOne(){//Number.class  
 try {  
 Thread.sleep(3000);  
 } catch (InterruptedException e) {  
 }  
   
 System.out.println("one");  
}  
  
public synchronized void getTwo(){//this  
 System.out.println("two");  
}  
  
public void getThree(){  
 System.out.println("three");  
}

}

1、持有同一个Number对象，并都加了锁，按调用顺序打印  
2、sleep方法不会释放锁  
3、普通方法不需要锁定对象，直接调用，最先调用  
4、两个Number对象，实际上并没有并发访问资源  
5、静态同步方法锁定的是类的Class对象，非静态同步方法锁定的是类的实例，同4，没有并发访问资源  
6、两个方法均为静态同步方法，此时构成并发访问，因为它们锁定的是类的同一个Class对象  
7、同4，没有并发访问资源  
8、同6，虽然是不同实例，但对应着同一个Class对象

直击灵魂的Interrupt七问  
1.Thread.interrupt()方法和InterruptedException异常的关系？是由interrupt触发产生了InterruptedException异常？  
2.Thread.interrupt()会中断线程什么状态的工作？ RUNNING or BLOCKING？  
3.一般Thread编程需要关注interrupt中断不？一般怎么处理？可以用来做什么？  
4.LockSupport.park()和unpark()，与object.wait()和notify()的区别？  
5.LockSupport.park(Object blocker)传递的blocker对象做什么用？  
6.LockSupport能响应Thread.interrupt()事件不？会抛出InterruptedException异常？  
7.Thread.interrupt()处理是否有对应的回调函数？类似于钩子调用？

1. Thread.interrupt()只是在Object.wait() .Object.join(), Object.sleep()几个方法会主动抛出InterruptedException异常。而在其他的block场景，只是通过设置了Thread的一个标志位信息，需要程序自己进行处理。  
   在J.U.C里面的ReentrantLock、Condition等源码都是自己去检测中断标志位，然后抛出InterruptedException。  
   if (Thread.interrupted()) // Clears interrupted status!  
   throw new InterruptedException();
2. Thread.interrupt设计的目的主要是用于处理线程处于block状态，比如wait(),sleep()状态就是个例子。但可以在程序设计时为支持task cancel，同样可以支持RUNNING状态。比如Object.join()和一些支持interrupt的一些nio channel设计。
3. interrupt用途： unBlock操作，支持任务cancel， 数据清理等。
4. 1) 面向的主体不一样。LockSuport主要是针对Thread进进行阻塞处理，可以指定阻塞队列的目标对象，每次可以指定具体的线程唤醒。Object.wait()是以对象为纬度，阻塞当前的线程和唤醒单个(随机)或者所有线程。  
   2) 实现机制不同。虽然LockSuport可以指定monitor的object对象，但和object.wait()，两者的阻塞队列并不交叉。可以看下测试例子。object.notifyAll()不能唤醒LockSupport的阻塞Thread.
5. 对应的blcoker会记录在Thread的一个parkBlocker属性中,通过jstack命令可以非常方便的监控具体的阻塞对象.
6. 能响应interrupt事件，但不会抛出InterruptedException异常