多线程学习笔记

### 线程实现4种方式

#### 继承Thread类创建线程

Thread类本质上是实现了Runnable接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。代码简单就不写示例了。

最佳实践：创建线程最好给线程指定一个线程名称，这样方便管理。在出现问题的时候，打印线程栈 (jstack -pid) 一眼就可以看出是哪个线程出的问题，这个线程是干什么的。

#### 实现Runnable接口创建线程

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，可以实现一个Runnable接口，并且实现run方法。

#### 实现Callable接口通过FutureTask包装器来创建Thread线程

Callable接口（也只有一个方法）定义如下：

|  |
| --- |
| public interface Callable<V>{  V call() throws Exception;  } |

创建线程代码如下：

|  |
| --- |
| public class SomeCallable<V> extends OtherClass implements Callable<V> {  @Override  public V call() throws Exception {  // TODO Auto-generated method stub  return null;  }  } |

|  |
| --- |
| Callable<V> oneCallable = new SomeCallable<V>();  //由Callable<Integer>创建一个FutureTask<Integer>对象：  FutureTask<V> oneTask = new FutureTask<V>(oneCallable);  //注释：FutureTask<Integer>是一个包装器，它通过接受Callable<Integer>来创建，它同时实现了Future和Runnable接口。  //由FutureTask<Integer>创建一个Thread对象：  Thread oneThread = new Thread(oneTask);  oneThread.start(); |

#### 使用ExecutorService、Callable、Future实现有返回结果的线程

ExecutorService、Callable、Future三个接口实际上都是属于Executor框架。返回结果的线程是在JDK1.5中引入的新特征，有了这种特征就不需要再为了得到返回值而大费周折了。而且自己实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口。类似的，无返回值的任务必须实现Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

注意：get方法是阻塞的，即：线程无返回结果，get方法会一直等待。

再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

下面提供了一个完整的有返回结果的多线程测试例：

|  |
| --- |
| /\*\*  \* 有返回值的线程  \*/  @SuppressWarnings("unchecked")  public class Test {  public static void main(String[] args) throws ExecutionException,  InterruptedException {  System.out.println("----程序开始运行----");  Date date1 = new Date();  int taskSize = 5;  // 创建一个线程池  ExecutorService pool = Executors.newFixedThreadPool(taskSize);  // 创建多个有返回值的任务  List<Future> list = new ArrayList<Future>();  for (int i = 0; i < taskSize; i++) {  Callable c = new MyCallable(i + " ");  // 执行任务并获取Future对象  Future f = pool.submit(c);  // System.out.println(">>>" + f.get().toString());  list.add(f);  }  // 关闭线程池  pool.shutdown();  // 获取所有并发任务的运行结果  for (Future f : list) {  // 从Future对象上获取任务的返回值，并输出到控制台  System.out.println(">>>" + f.get().toString());  }  Date date2 = new Date();  System.out.println("----程序结束运行----，程序运行时间【"  + (date2.getTime() - date1.getTime()) + "毫秒】");  }  }    public class MyCallable implements Callable<Object> {  private String taskNum;    MyCallable(String taskNum) {  this.taskNum = taskNum;  }    public Object call() throws Exception {  System.out.println(">>>" + taskNum + "任务启动");  Date dateTmp1 = new Date();  Thread.sleep(1000);  Date dateTmp2 = new Date();  long time = dateTmp2.getTime() - dateTmp1.getTime();  System.out.println(">>>" + taskNum + "任务终止");  return taskNum + "任务返回运行结果,当前时间【" + time + "毫秒】";  }  } |

#### 总结

从上面4个实例可以看出实现线程的方式也可以说是3种方式，分别是继承Threa类，实现Runnable接口、实现Callable（一种使用FutureTask，一种使用J.U.C的线程池）。

### 终止线程三种方式

有三种方法可以使终止线程：

* 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
* 使用stop方法强行终止线程（这个方法不推荐使用，因为stop会导致线程立马停止，这样可能会导致一些锁或数据、资源等释放操作无法得到正确执行，这样就可能出现不可预料的结果）。
* 使用interrupt方法中断线程。

#### 使用退出标志终止线程

当run方法执行完后，线程就会退出。但有时run方法是永远不会结束的。如在服务端程序中使用线程进行监听客户端请求，或是其他的需要循环处理的任务。在这种情况下，一般是将这些任务放在一个循环中，如while循环。如果想让循环永远运行下去，可以使用while（true）{……}来处理。但要想使while循环在某一特定条件下退出，最直接的方法就是设一个boolean类型的标志，并通过设置这个标志为true或false来控制while循环是否退出。下面给出了一个利用退出标志终止线程的例子。

|  |
| --- |
| public class ThreadFlag extends Thread{  public volatile boolean exit = false;  public void run(){  while (!exit){//执行逻辑};  }  public static void main(String[] args) throws Exception{  ThreadFlag thread = new ThreadFlag();  thread.start();  sleep(5000); // 主线程延迟5秒  thread.exit = true; // 终止线程thread  thread.join();  System.out.println("线程退出!");  }  } |

上述代码中，join() 的作用：让“主线程”等待“子线程”结束之后才能继续运行。也就是说执行到【thread.join()】代码时，主线程（main）会等待thread线程执行结束后，主线程才继续执行。

#### 使用stop方法终止线程

使用stop方法可以强行终止正在运行或挂起的线程。我们可以使用如下的代码来终止线程：thread.stop();

虽然使用上面的代码可以终止线程，但使用stop方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，因此，并不推荐使用stop方法来终止线程。

**Thread类扩展知识点**：

* suspend()方法就是将一个线程挂起(暂停)，resume()方法就是将一个挂起线程复活继续执行。suspend被弃用的原因是因为它会造成死锁。suspend方法和stop方法不一样，它不会破换对象和强制释放锁，相反它会一直保持对锁的占有，一直到其他的线程调用resume方法，它才能继续向下执行。假如有A，B两个线程，A线程在获得某个锁之后被suspend阻塞，这时A不能继续执行，线程B在或者相同的锁之后才能调用resume方法将A唤醒，但是此时的锁被A占有，B不能继续执行，也就不能及时的唤醒A，此时A，B两个线程都不能继续向下执行而形成了死锁。这就是suspend被弃用的原因。
* sleep：sleep相当于让线程睡眠，交出CPU，让CPU去执行其他的任务。但是有一点要非常注意，sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。
* yield：调用yield方法会让当前线程交出CPU权限，让CPU去执行其他的线程。它跟sleep方法类似，同样不会释放锁。但是yield不能控制具体的交出CPU的时间，另外yield方法只能让拥有相同优先级的线程有获取CPU执行时间的机会。注意，调用yield方法并不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间，这一点是和sleep方法不一样的。
* join：让当前线程等待调用join方法的线程执行完才继续执行。join()实际是利用了wait()，只不过它不用等待notify()/notifyAll()，且不受其影响。它结束的条件是：1）等待时间到；2）目标线程已经run完（通过isAlive()来判断）。

#### 使用interrupt方法终止线程

使用interrupt方法来终端线程可分为两种情况：

* 线程处于阻塞状态，如使用了sleep方法。
* 使用while（！isInterrupted（））{……}来判断线程是否被中断。

在第一种情况下使用interrupt方法，sleep方法将抛出一个InterruptedException例外，而在第二种情况下线程将直接退出。下面的代码演示了在第一种情况下使用interrupt方法。

|  |
| --- |
| public class ThreadInterrupt extends Thread{  public void run(){  try{  sleep(50000); // 延迟50秒  } catch (InterruptedException e){  System.out.println(e.getMessage());  }  }  public static void main(String[] args) throws Exception{  Thread thread = new ThreadInterrupt();  thread.start();  System.out.println("在50秒之内按任意键中断线程!");  System.in.read();  thread.interrupt();  thread.join();  System.out.println("线程已经退出!");  }  } |

上面代码的运行结果如下：

|  |
| --- |
| 在50秒之内按任意键中断线程!  sleep interrupted  线程已经退出! |

在调用interrupt方法后， sleep方法抛出异常，然后输出错误信息：sleep interrupted.

注意：在Thread类中有两个方法可以判断线程是否通过interrupt方法被终止。一个是静态的方法interrupted（），一个是非静态的方法isInterrupted（），这两个方法的区别是interrupted用来判断当前线是否被中断，而isInterrupted可以用来判断其他线程是否被中断。因此，while （！isInterrupted（））也可以换成while （！Thread.interrupted（））。

### 线程状态

Java中的线程的生命周期大体可分为5种状态：

* 新建(NEW)：新创建了一个线程对象。
* 可运行(RUNNABLE)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。
* 运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。
* 阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
* 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
* 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
* 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
* 死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

线程的状态图如下：



### 内存模型

一个现代计算机通常由两个或者多个CPU。其中一些CPU还有多核。从这一点可以看出，在一个有两个或者多个CPU的现代计算机上同时运行多个线程是可能的。每个CPU在某一时刻运行一个线程是没有问题的。这意味着，如果你的Java程序是多线程的，在你的Java程序中每个CPU上一个线程可能同时（并发）执行。

每个CPU都包含一系列的寄存器，它们是CPU内内存的基础。CPU在寄存器上执行操作的速度远大于在主存上执行的速度。这是因为CPU访问寄存器的速度远大于主存。

每个CPU可能还有一个CPU缓存层。实际上，绝大多数的现代CPU都有一定大小的缓存层。CPU访问缓存层的速度快于访问主存的速度，但通常比访问内部寄存器的速度还要慢一点。一些CPU还有多层缓存，但这些对理解Java内存模型如何和内存交互不是那么重要。只要知道CPU中可以有一个缓存层就可以了。

一个计算机还包含一个主存。所有的CPU都可以访问主存。主存通常比CPU中的缓存大得多。

通常情况下，当一个CPU需要读取主存时，它会将主存的部分读到CPU缓存中。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当CPU需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在某个时间点将值刷新回主存。

当CPU需要在缓存层存放一些东西的时候，存放在缓存中的内容通常会被刷新回主存。CPU缓存可以在某一时刻将数据局部写到它的内存中，和在某一时刻局部刷新它的内存。它不会再某一时刻读/写整个缓存。通常，在一个被称作“cache lines”的更小的内存块中缓存被更新。一个或者多个缓存行可能被读到缓存，一个或者多个缓存行可能再被刷新回主存。

想象一下，共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到CPU缓存中。然后修改了这个对象。只要CPU缓存没有被刷新会主存，对象修改后的版本对跑在其它CPU上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

使用Java中的volatile关键字。volatile关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。同时会通知其他线程该变量cpu的缓存行无效，其他线程将直接从主存中读取该变量的值到cpu缓存中。

如果想了解指令重排序和happens-before原则的知识点或更多关于内存模型的知识，可以阅读：<https://www.cnblogs.com/dolphin0520/p/3920373.html>

以下为代码示例：

|  |
| --- |
| public class VolatileTest implements Runnable{  private volatile boolean tag = false;  private static CountDownLatch latch = new CountDownLatch(10);  @Override  public void run() {  if(tag){  System.out.println("tag为："+tag);  }else{  System.out.println("tag111111为："+tag);  tag = true;  System.out.println("tag2222为："+tag);  }  latch.countDown();  }  public static void main(String[] args) throws InterruptedException {  ExecutorService executorService = Executors.newFixedThreadPool(10);  VolatileTest test = new VolatileTest();  for(int i = 0; i < 10; i++){  executorService.submit(test);  }  latch.await();  executorService.shutdown();  System.out.println("测试完毕");  }  } |

上面代码示例其中的一次执行结果如下：

|  |
| --- |
| tag111111为：false  tag111111为：false  tag111111为：false  tag2222为：true  tag2222为：true  tag2222为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  tag为：true  测试完毕 |

Volatile修饰的变量能保证变量原子操作下的线程安全，非原子操作无法保证线程安全，如下面代码：

|  |
| --- |
| volatile int count;  public void add(){  count++;  } |

上述代码的add方法在多线程环境下存在线程安全的问题，比如有2个线程（A和B），A将count的值（0）读到cpu缓存中，并进行加1，但还没赋值给count的节点前任意时刻，线程B读到cpu缓存的count值也为0，并且进行++操作，所以最终获取的值为1，而不是2。这里++操作就不是原子操作，所以哪怕使用volatile修饰也存在线程安全问题。

### Synchronized

线程安全是并发编程中的重要关注点，应该注意到的是，造成线程安全问题的主要诱因有两点，一是存在共享数据(也称临界资源)，二是存在多条线程共同操作共享数据。因此为了解决这个问题，我们可能需要这样一个方案，当存在多个线程操作共享数据时，需要保证同一时刻有且只有一个线程在操作共享数据，其他线程必须等到该线程处理完数据后再进行，这种方式有个高尚的名称叫互斥锁，即能达到互斥访问目的的锁，也就是说当一个共享数据被当前正在访问的线程加上互斥锁后，在同一个时刻，其他线程只能处于等待的状态，直到当前线程处理完毕释放该锁。在 Java 中，关键字 synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)，同时我们还应该注意到synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到（保证可见性，完全可以替代Volatile功能），这点确实也是很重要的。

synchronized关键字最主要有以下3种应用方式：

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

#### synchronized的可重入性

从互斥锁的设计上来说，当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁，请求将会成功。在java中synchronized是基于原子性的内部锁机制，是可重入的，因此在一个线程调用synchronized方法的同时在其方法体内部调用该对象另一个synchronized方法，也就是说一个线程得到一个对象锁后再次请求该对象锁，是允许的，这就是synchronized的可重入性。简单代码如下：

|  |
| --- |
| public class SynchronziedTest implements Runnable{  public synchronized void test1(){  System.out.println("开始执行test1");  test2();  System.out.println("执行test1结束");  }  public synchronized void test2(){  System.out.println("执行test2");  }  @Override  public void run() {  test1();  }  public static void main(String[] args) {  new Thread(new SynchronziedTest()).start();  }  } |

上面示例执行结果如下：

|  |
| --- |
| 开始执行test1  执行test2  执行test1结束 |

线程获取SynchronziedTest对象锁进入test1方法后，如果在test1方法中调用test2方法（该方法也被synchronized）是被允许的，我们就说该锁是可重入锁（或该锁有可重入的特性）。

#### 文章推荐

<http://cmsblogs.com/?p=2071>

### wait、notify和notifyAll

wait、notify和notifyAll主要注意的有以下方面：

* wait、notify和notifyAll方法必须在同步代码块中使用
* sleep和wait的区别：sleep不会释放对象锁，wait会释放对象锁
* notify和notifyAll方法调用后，尽量立即退出临界区，就是说尽量不要在notify和notifyAll方法后面写一些耗时的操作
* 多线程中校验某个条件的变化用if还是用while？如：

while(条件不满足){this.wait();}还是if(条件不满足){this.wait();}，答案是使用while。

当一个线程正在某一个对象的同步方法中运行时调用了这个对象的wait()方法，那么这个线程将释放该对象的独占锁并被放入这个对象的等待队列。注意，wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。在调用对象的notify()和notifyAll()方法之前,调用线程必须已经得到该对象的锁。因此,必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

wait、notify和notifyAll方法只是实现线程间通信的一种方式，更多内容可以查看：<http://ifeve.com/thread-signaling/>

### 死锁

死锁是两个或更多线程阻塞着等待其它处于死锁状态的线程所持有的锁。死锁通常发生在多个线程同时但以不同的顺序请求同一组锁的时候。

例如，如果线程1锁住了A，然后尝试对B进行加锁，同时线程2已经锁住了B，接着尝试对A进行加锁，这时死锁就发生了。线程1永远得不到B，线程2也永远得不到A，并且它们永远也不会知道发生了这样的事情。为了得到彼此的对象（A和B），它们将永远阻塞下去。这种情况就是一个死锁。

一般造成死锁必须同时满足如下4个条件：

* 互斥条件：线程使用的资源必须至少有一个是不能共享的；
* 请求与保持条件：至少有一个线程必须持有一个资源并且正在等待获取一个当前被其它线程持有的资源；
* 非剥夺条件：分配资源不能从相应的线程中被强制剥夺；
* 循环等待条件：第一个线程等待其它线程，后者又在等待第一个线程。

因为要产生死锁，这4个条件必须同时满足，所以要防止死锁的话，只需要破坏其中一个条件即可。一些推荐做法如下：

* 尽量使用tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。
* 尽量使用java.util.concurrent(jdk 1.5以上)包的并发类代替手写控制并发，比较常用的是ConcurrentHashMap、ConcurrentLinkedQueue、AtomicBoolean等等，实际应用中java.util.concurrent.atomic十分有用，简单方便且效率比使用Lock更高
* 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁
* 尽量减少同步的代码块

比较简单的方法：所有线程都按顺序获取资源，比如资源ABCD ，那所有线程必须获取了A资源再去 申请B资源 ， 依次类推。

其实我们在操作资源的时候，可以使用到队列。让线程按照顺序操作资源，这样的话，就不会造成死锁现象。

按顺序获取资源，是破坏循环等待的条件，算是比较好的方法但是也是有缺点的：

* 是限制的进程对资源的要求，而已给系统中所有的资源合理编号也是件难事，同时增加了系统开销。
* 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

对于这个问题，单一的临界资源可以使用资源分配图算法，多类的临界资源可以使用“银行家算法”。

### ThreadLocal

#### 作用

ThreadLocal的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。也可以归纳为以下两点：

1。每个线程中都有一个自己的ThreadLocalMap类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。

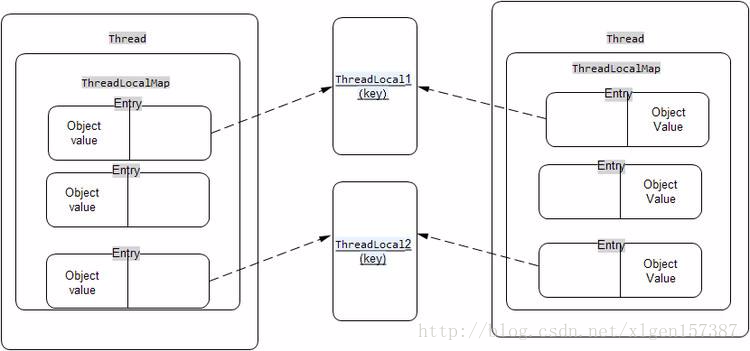
2。将一个共用的ThreadLocal静态实例作为key，将不同对象的引用保存到不同线程的ThreadLocalMap中，然后在线程执行的各处通过这个静态ThreadLocal实例的get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

#### 使用场景

最常见的ThreadLocal使用场景为用来解决数据库连接、Session管理等。我们平常开发常用于传递登录的用户信息，不用每次从session中去取，同时减少user信息到处传递的麻烦。比如拦截器中校验用户信息后，将用户信息放到ThreadLocalMap中，这样无论是controoler层还是service层（或其他层）需要获取用户信息，只需要从ThreadLocal中获取即可。当然ThreadLocal也不能乱用，如果存储的内容过多，并且释放不及时（后面会继续介绍），可能会导致内存泄漏的问题。

#### 源码分析

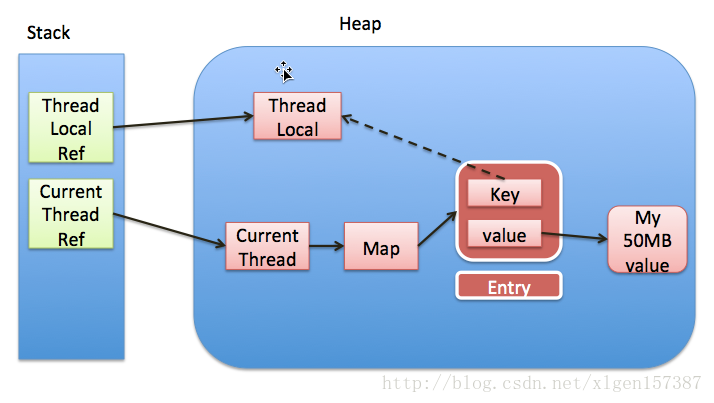
##### Thread、ThreadLocal、ThreadLocalMap、Entry之间的关系



上图中描述了：一个Thread中只有一个ThreadLocalMap，一个ThreadLocalMap中可以有多个ThreadLocal对象，其中一个ThreadLocal对象对应一个ThreadLocalMap中一个的Entry实体（也就是说：一个Thread可以依附有多个ThreadLocal对象）。

##### ThreadLocal各类引用关系

在ThreadLocal的生命周期中，都存在这些引用。（ 实线代表强引用，虚线代表弱引用）。



ThreadLocal到Entry对象key的引用断裂，而不及时的清理Entry对象，可能会造成OOM内存溢出！

##### 引用的类型

我们对引用的理解也许很简单，就是：如果 reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。但是书上说的这种方式过于狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态，对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存之中；如果内存在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

一般的引用类型分为：强引用（ Strong Reference）、软引用（ Soft Reference）、弱引用（ Weak Reference）、虚引用（ Phantom Reference）四种，这四种引用强度依次逐渐减弱。下边是四中类型的介绍：

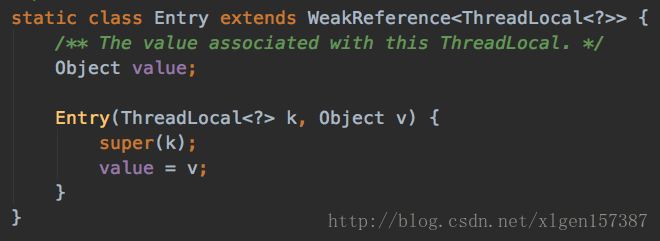
* 强引用：就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象，也就是说即使Java虚拟机内存空间不足时，GC收集器也绝不会回收该对象，如果内存空间不够就会导致内存溢出。
* 软引用：用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行回收，以免出现内存溢出。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。软引用适合引用那些可以通过其他方式恢复的对象，例如：数据库缓存中的对象就可以从数据库中恢复，所以软引用可以用来实现缓存。等会会介绍MyBatis中的使用软引用实现缓存的案例。
* 弱引用：也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 WeakReference 类来实现弱引用。ThreadLocal使用到的就有弱引用。
* 虚引用：也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了PhantomReference 类来实现虚引用。

各引用类型的生命周期及作用：



##### ThreadLocal中的弱引用

上述我们知道了当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。我们的ThreadLocal中ThreadLocalMap中的Entry类的key就是弱引用的，如下：



而弱引用会在垃圾收集器工作的时候进行回收，也就是说，只要执行垃圾回收，这些对象就会被回收，也就是上述图B中的虚线连接的地方断开了，就成了一个没有key的Entry。这里暂时不做过多的介绍，建议多看写得不错的一篇文章：<https://blog.csdn.net/xlgen157387/article/details/78513735?ref=myread>

##### 内存泄漏

synchronized是用时间换空间、ThreadLocal是用空间换时间，为什么这么说？

因为synchronized操作数据，只需要在主存存一个变量即可，就阻塞等共享变量，而ThreadLocal是每个线程都创建一块小的堆工作内存。一个线程对应一块工作内存，线程可以存储多个ThreadLocal。那么假设，开启1万个线程，每个线程创建1万个ThreadLocal，也就是每个线程维护1万个ThreadLocal小内存空间，而且当线程执行结束以后，假设这些ThreadLocal里的Entry还不会被回收，那么将很容易导致堆内存溢出。

怎么办？难道JVM就没有提供什么解决方案吗？

ThreadLocal当然有想到，所以他们把ThreadLocal里的Entry设置为弱引用，当垃圾回收的时候，回收ThreadLocal。

什么是弱引用？

Key使用强引用：也就是上述说的情况，引用的ThreadLocal的对象被回收了，ThreadLocal的引用ThreadLocalMap的Key为强引用并没有被回收，如果不手动回收的话，ThreadLocal将不会回收那么将导致内存泄漏。

Key使用弱引用：引用的ThreadLocal的对象被回收了，ThreadLocal的引用ThreadLocalMap的Key为弱引用，如果内存回收，那么将ThreadLocalMap的Key将会被回收，ThreadLocal也将被回收。value在ThreadLocalMap调用get、set、remove的时候就会被清除。

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

那按你这么说，既然JVM有保障了，还有什么内存泄漏可言？

ThreadLocalMap使用ThreadLocal对象作为弱引用，当垃圾回收的时候，ThreadLocalMap中Key将会被回收，也就是将Key设置为null的Entry。如果线程迟迟无法结束，也就是ThreadLocal对象将一直不会回收，回顾到上面存在很多线程+TheradLocal，那么也将导致内存泄漏。

其实，在ThreadLocal中，当调用remove、get、set方法的时候，会清除为null的弱引用，也就是回收ThreadLocal。

总结：

JVM利用设置ThreadLocalMap的Key为弱引用，来避免内存泄露。

JVM利用调用remove、get、set方法的时候，回收弱引用。

当ThreadLocal存储很多Key为null的Entry的时候，而不再去调用remove、get、set方法，那么将导致内存泄漏。

当使用static ThreadLocal的时候，延长ThreadLocal的生命周期，那也可能导致内存泄漏。因为，static变量在类未加载的时候，它就已经加载，当线程结束的时候，static变量不一定会回收。那么，比起普通成员变量使用的时候才加载，static的生命周期加长将更容易导致内存泄漏危机。

##### 我的总结

我们使用tomcat为容器，用户发送请求，到请求结束这整个过程结束后，处理请求的线程未必会被销毁，有可能会放到tomcat的线程池中一直在重复被利用，所以这种情况下，ThreadLocal会可能出现内存泄漏的问题，或者说ThreadLocalMap会一直占用内存资源，下面我们分别分析ThreadLocal2种不同的使用情况。

ThreadLocal对象不使用static修饰的情况：

* 由于ThreadLocalMap的key是弱引用，所以每次gc后，key会被回收，而线程未必会被销毁（如容器为tomcat），那么ThreadLocal中会出现key为null，value为存储的对象，这样会造成资源浪费（还有其他问题）。虽然ThreadLocal中的get、set方法会在每次调用的时候清除这类key为null的数据，但是如果线程一直不调用get和set方法呢？那内存就会一直被占用着，最后就可能出现内存泄漏的问题。
* 如果线程被销毁，那ThreadLocalMap会被回收，这里主要是有2个问题：
* 使用过程中发生gc（ThreadLocal的get方法调用前），当前线程获取副本数据将为空(这样就容易出现莫名其妙的bug)；所以jdk推荐使用static修饰ThreadLocal；这里关于获取副本数据为空的情况是不会出现的，这点做保留只是为了自己不会再次出现这种错误的理解（因为只有将ThreadLocal对象设置为null且当前线程销毁后，gc才回收ThreadLocalMap对象），关于推荐使用static修饰ThreadLocal，只是为了延长ThreadLocal的生命周期。
* 在线程被销毁前（线程可能会多次使用后才销毁），内存浪费（可能出现内存泄漏问题），除非调用了ThreadLocal的get、set、remove方法，不然key为null，value有内容的数据一直不会被回收，并占据着内存。

ThreadLocal对象使用static修饰的情况：

* 由于ThreadLocal使用static修饰，那么ThreadLocalMap的key不再是弱引用，所以每次gc后，key不会被回收。同样因为线程未必会被销毁，所以ThreadLocalMap中的内容永远不会被垃圾回收器回收。这里更为严重的是，就算线程在某一时间被销毁了，但是因为key为强引用（static修饰不会被回收），所以ThreadLocalMap中的内容也不会被回收，这样内存就一直被浪费着，最后可能出现内存泄漏的问题。这里有点注意的是，key是否回收不重要，因为他用static修饰，只有一份，最重要的是value内的数据回收是关键。

##### 最佳实现

* 每次使用完ThreadLocal，都调用它的remove()方法，清除数据。
* 如果需要延长ThreadLocal生命周期，推荐使用static修饰。

##### 如何学习

我觉得多阅读源代码，配合以下文章进行学习，这样关于ThreadLocal的知识应该是能尽可能掌握好了，下面是文章链接：

<https://blog.csdn.net/xlgen157387/article/details/78513735?ref=myread>

<https://www.cnblogs.com/qiuyong/p/7091689.html>

<https://www.zhihu.com/question/23089780>

<http://www.importnew.com/22039.html>

### AQS

#### 概述

Java的内置锁一直都是备受争议的，在JDK 1.6之前，synchronized这个重量级锁其性能一直都是较为低下，虽然在1.6后，进行大量的锁优化策略,但是与Lock相比synchronized还是存在一些缺陷的：虽然synchronized提供了便捷性的隐式获取锁释放锁机制（基于JVM机制），但是它却缺少了获取锁与释放锁的可操作性，可中断、超时获取锁，且它为独占式在高并发场景下性能大打折扣。

在介绍Lock之前，我们需要先熟悉一个非常重要的组件，掌握了该组件JUC包下面很多问题都不在是问题了。该组件就是AQS。

AQS，AbstractQueuedSynchronizer，即队列同步器。它是构建锁或者其他同步组件的基础框架（如ReentrantLock、ReentrantReadWriteLock、Semaphore等），JUC并发包的作者（Doug Lea）期望它能够成为实现大部分同步需求的基础。它是JUC并发包中的核心基础组件。

AQS解决了实现同步器时涉及当的大量细节问题，例如获取同步状态、FIFO同步队列。基于AQS来构建同步器可以带来很多好处。它不仅能够极大地减少实现工作，而且也不必处理在多个位置上发生的竞争问题。

在基于AQS构建的同步器中，只能在一个时刻发生阻塞，从而降低上下文切换的开销，提高了吞吐量。同时在设计AQS时充分考虑了可伸缩行，因此J.U.C中所有基于AQS构建的同步器均可以获得这个优势。

#### 基本功能

同步器至少要有以下两种类型的方法acquire和release

* acquire：至少要有一个操作能实现对调用线程的阻塞，直到同步器允许它进行操作。
* release：至少要有一个操作能用一种方式解锁一个或者更多个已经阻塞的线程改变同步状态。

同时，同步器还需要支持以下几种功能：

* 非阻塞式的同步过程尝试(tryLock)
* 可选的超时机制，可以允许程序放弃等待
* 可以通过中断执行取消

而为了适应不同的同步器，同步器要支持两种模式:

* 独占式 exclusive。要保证一次只有一个线程可以经过阻塞点
* 共享式 shared。可以允许多个线程阻塞点

#### 框架



它维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里volatile是核心关键词，具体volatile的语义，在此不述。state的访问方式有三种:

* getState()
* setState()
* compareAndSetState()

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch为例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

#### 源码详解

本节开始讲解AQS的源码实现。依照acquire-release、acquireShared-releaseShared的次序来。

##### acquire(int)

此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了。下面是acquire()的源码：

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

函数流程如下：

* tryAcquire()尝试直接去获取资源，如果成功则直接返回；
* addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
* acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
* 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

这时单凭这4个抽象的函数来看流程还有点朦胧，不要紧，看完接下来的分析后，你就会明白了。就像《大话西游》里唐僧说的：等你明白了舍生取义的道理，你自然会回来和我唱这首歌的。

###### tryAcquire(int)

此方法尝试去获取独占资源。如果获取成功，则直接返回true，否则直接返回false。这也正是tryLock()的语义，还是那句话，当然不仅仅只限于tryLock()。如下是tryAcquire()的源码：

|  |
| --- |
| protected boolean tryAcquire(int arg) {  throw new UnsupportedOperationException();  } |

什么？直接throw异常？说好的功能呢？好吧，还记得概述里讲的AQS只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现吗？就是这里了！！！AQS这里只定义了一个接口，具体资源的获取交由自定义同步器去实现了（通过state的get/set/CAS）！！！至于能不能重入，能不能加塞，那就看具体的自定义同步器怎么去设计了！！！当然，自定义同步器在进行资源访问时要考虑线程安全的影响。

这里之所以没有定义成abstract，是因为独占模式下只用实现tryAcquire-tryRelease，而共享模式下只用实现tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口。说到底，Doug Lea还是站在咱们开发者的角度，尽量减少不必要的工作量。

###### addWaiter(Node)

此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点。还是上源码吧：

|  |
| --- |
| private Node addWaiter(Node mode) {  //以给定模式构造结点。mode有两种：EXCLUSIVE（独占）和SHARED（共享）  Node node = new Node(Thread.currentThread(), mode);    //尝试快速方式直接放到队尾。  Node pred = tail;  if (pred != null) {  node.prev = pred;  if (compareAndSetTail(pred, node)) {  pred.next = node;  return node;  }  }    //上一步失败则通过enq入队。  enq(node);  return node;  } |

不用再说了，直接看注释吧。这里我们说下Node。Node结点是对每一个访问同步代码的线程的封装，其包含了需要同步的线程本身以及线程的状态，如是否被阻塞，是否等待唤醒，是否已经被取消等。变量waitStatus则表示当前被封装成Node结点的等待状态，共有4种取值CANCELLED、SIGNAL、CONDITION、PROPAGATE。

* CANCELLED：值为1，在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消该Node的结点，其结点的waitStatus为CANCELLED，即结束状态，进入该状态后的结点将不会再变化。
* SIGNAL：值为-1，被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。说白了，就是处于唤醒状态，只要前继结点释放锁，就会通知标识为SIGNAL状态的后继结点的线程执行。
* CONDITION：值为-2，与Condition相关，该标识的结点处于等待队列中，结点的线程等待在Condition上，当其他线程调用了Condition的signal()方法后，CONDITION状态的结点将从等待队列转移到同步队列中，等待获取同步锁。
* PROPAGATE：值为-3，与共享模式相关，在共享模式中，该状态标识结点的线程处于可运行状态。
* 0状态：值为0，代表初始化状态。

AQS在判断状态时，通过用waitStatus>0表示取消状态，而waitStatus<0表示有效状态。

###### enq(Node)

此方法用于将node加入队尾。源码如下：

|  |
| --- |
| private Node enq(final Node node) {  //CAS"自旋"，直到成功加入队尾  for (;;) {  Node t = tail;  if (t == null) { // 队列为空，创建一个空的标志结点作为head结点，并将tail也指向它。  if (compareAndSetHead(new Node()))  tail = head;  } else {//正常流程，放入队尾  node.prev = t;  if (compareAndSetTail(t, node)) {  t.next = node;  return t;  }  }  }  } |

如果你看过AtomicInteger.getAndIncrement()函数源码，那么相信你一眼便看出这段代码的精华。CAS自旋volatile变量，是一种很经典的用法。还不太了解的，自己去百度一下吧。

###### acquireQueued(Node, int)

OK，通过tryAcquire()和addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了。聪明的你立刻应该能想到该线程下一部该干什么了吧：进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了。没错，就是这样！是不是跟医院排队拿号有点相似~~acquireQueued()就是干这件事：在等待队列中排队拿号（中间没其它事干可以休息），直到拿到号后再返回。这个函数非常关键，还是上源码吧：

|  |
| --- |
| final boolean acquireQueued(final Node node, int arg) {  boolean failed = true;//标记是否成功拿到资源  try {  boolean interrupted = false;//标记等待过程中是否被中断过    //又是一个“自旋”！  for (;;) {  final Node p = node.predecessor();//拿到前驱  //如果前驱是head，即该结点已成老二，那么便有资格去尝试获取资源（可能是老大释放完资源唤醒自己的，当然也可能被interrupt了）。  if (p == head && tryAcquire(arg)) {  setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。  p.next = null; // setHead中node.prev已置为null，此处再将head.next置为null，就是为了方便GC回收以前的head结点。也就意味着之前拿完资源的结点出队了！  failed = false;  return interrupted;//返回等待过程中是否被中断过  }    //如果自己可以休息了，就进入waiting状态，直到被unpark()  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;//如果等待过程中被中断过，哪怕只有那么一次，就将interrupted标记为true  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

到这里了，我们先不急着总结acquireQueued()的函数流程，先看看shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()具体干些什么。

###### shouldParkAfterFailedAcquire(Node, Node)

此方法主要用于检查状态，看看自己是否真的可以去休息了（进入waiting状态，如果线程状态转换不熟，可以参考本人上一篇写的Thread详解），万一队列前边的线程都放弃了只是瞎站着，那也说不定，对吧！

|  |
| --- |
| private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  int ws = pred.waitStatus;//拿到前驱的状态  if (ws == Node.SIGNAL)  //如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了  return true;  if (ws > 0) {  /\*  \* 如果前驱放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。  \* 注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，稍后就会被保安大叔赶走了(GC回  \*/  do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  //如果前驱正常，那就把前驱的状态设置成SIGNAL，告诉它拿完号后通知自己一下。有可能失败，人家说不定刚刚释放完呢！  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  }  return false;  } |

整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心去休息，需要去找个安心的休息点，同时可以再尝试下看有没有机会轮到自己拿号。

###### parkAndCheckInterrupt()

如果线程找好安全休息点后，那就可以安心去休息了。此方法就是让线程去休息，真正进入等待状态。

|  |
| --- |
| private final boolean parkAndCheckInterrupt() {  LockSupport.park(this);//调用park()使线程进入waiting状态  return Thread.interrupted();//如果被唤醒，查看自己是不是被中断的。  } |

park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt()。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

###### 小结

OK，看了shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()，现在让我们再回到acquireQueued()，总结下该函数的具体流程：

* 结点进入队尾后，检查状态，找到安全休息点；
* 调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；
* 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

##### 小结

OK，acquireQueued()分析完之后，我们接下来再回到acquire()！再贴上它的源码吧：

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

再来总结下它的流程吧：

* 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回；
* 没成功，则addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
* acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
* 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

由于此函数是重中之重，我再用流程图总结一下：



至此，acquire()的流程终于算是告一段落了。这也就是ReentrantLock.lock()的流程，不信你去看其lock()源码吧，整个函数就是一条acquire(1)！

##### release(int)

上一小节已经把acquire()说完了，这一小节就来讲讲它的反操作release()吧。此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

|  |
| --- |
| public final boolean release(int arg) {  if (tryRelease(arg)) {  Node h = head;//找到头结点  if (h != null && h.waitStatus != 0)  unparkSuccessor(h);//唤醒等待队列里的下一个线程  return true;  }  return false;  } |

逻辑并不复杂。它调用tryRelease()来释放资源。有一点需要注意的是，它是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自定义同步器在设计tryRelease()的时候要明确这一点！

###### tryRelease(int)

此方法尝试去释放指定量的资源。下面是tryRelease()的源码：

|  |
| --- |
| protected boolean tryRelease(int arg) {  throw new UnsupportedOperationException();  } |

跟tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，上面已经提到了，release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false。

###### unparkSuccessor(Node)

此方法用于唤醒等待队列中下一个线程。下面是源码：

|  |
| --- |
| private void unparkSuccessor(Node node) {  //这里，node一般为当前线程所在的结点。  int ws = node.waitStatus;  if (ws < 0)//置零当前线程所在的结点状态，允许失败。  compareAndSetWaitStatus(node, ws, 0);  Node s = node.next;//找到下一个需要唤醒的结点s  if (s == null || s.waitStatus > 0) {//如果为空或已取消  s = null;  for (Node t = tail; t != null && t != node; t = t.prev)  if (t.waitStatus <= 0)//从这里可以看出，<=0的结点，都是还有效的结点。  s = t;  }  if (s != null)  LockSupport.unpark(s.thread);//唤醒  } |

此方法用于唤醒等待队列中下一个线程。下面是源码：这个函数并不复杂。一句话概括：用unpark()唤醒等待队列中最前边的那个未放弃线程，这里我们也用s来表示吧。此时，再和acquireQueued()联系起来，s被唤醒后，进入if (p == head && tryAcquire(arg))的判断（即使p!=head也没关系，它会再进入shouldParkAfterFailedAcquire()寻找一个安全点。这里既然s已经是等待队列中最前边的那个未放弃线程了，那么通过shouldParkAfterFailedAcquire()的调整，s也必然会跑到head的next结点，下一次自旋p==head就成立啦），然后s把自己设置成head标杆结点，表示自己已经获取到资源了，acquire()也返回了！！And then, DO what you WANT!

###### 小结

release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

##### acquireShared(int)

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。下面是acquireShared()的源码：

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

这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取。所以这里acquireShared()的流程就是：

* tryAcquireShared()尝试获取资源，成功则直接返回；
* 失败则通过doAcquireShared()进入等待队列，直到获取到资源为止才返回。

###### doAcquireShared(int)

此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。下面是doAcquireShared()的源码：

|  |
| --- |
| private void doAcquireShared(int arg) {  final Node node = addWaiter(Node.SHARED);//加入队列尾部  boolean failed = true;//是否成功标志  try {  boolean interrupted = false;//等待过程中是否被中断过的标志  for (;;) {  final Node p = node.predecessor();//前驱  if (p == head) {//如果到head的下一个，因为head是拿到资源的线程，此时node被唤醒，很可能是head用完资源来唤醒自己的  int r = tryAcquireShared(arg);//尝试获取资源  if (r >= 0) {//成功  setHeadAndPropagate(node, r);//将head指向自己，还有剩余资源可以再唤醒之后的线程  p.next = null; // help GC  if (interrupted)//如果等待过程中被打断过，此时将中断补上。  selfInterrupt();  failed = false;  return;  }  }    //判断状态，寻找安全点，进入waiting状态，等着被unpark()或interrupt()  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

有木有觉得跟acquireQueued()很相似？对，其实流程并没有太大区别。只不过这里将补中断的selfInterrupt()放到doAcquireShared()里了，而独占模式是放到acquireQueued()之外，其实都一样，不知道Doug Lea是怎么想的。

跟独占模式比，还有一点需要注意的是，这里只有线程是head.next时（“老二”），才会去尝试获取资源，有剩余的话还会唤醒之后的队友。那么问题就来了，假如老大用完后释放了5个资源，而老二需要6个，老三需要1个，老四需要2个。老大先唤醒老二，老二一看资源不够，他是把资源让给老三呢，还是不让？答案是否定的！老二会继续park()等待其他线程释放资源，也更不会去唤醒老三和老四了。独占模式，同一时刻只有一个线程去执行，这样做未尝不可；但共享模式下，多个线程是可以同时执行的，现在因为老二的资源需求量大，而把后面量小的老三和老四也都卡住了。当然，这并不是问题，只是AQS保证严格按照入队顺序唤醒罢了（保证公平，但降低了并发）。

###### setHeadAndPropagate(Node, int)

|  |
| --- |
| private void setHeadAndPropagate(Node node, int propagate) {  Node h = head;  setHead(node);//head指向自己  //如果还有剩余量，继续唤醒下一个邻居线程  if (propagate > 0 || h == null || h.waitStatus < 0) {  Node s = node.next;  if (s == null || s.isShared())  doReleaseShared();  }  } |

此方法在setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合（比如还有剩余资源），还会去唤醒后继结点，毕竟是共享模式！

doReleaseShared()我们留着下一小节的releaseShared()里来讲。

###### 小结

OK，至此，acquireShared()也要告一段落了。让我们再梳理一下它的流程：

* tryAcquireShared()尝试获取资源，成功则直接返回；
* 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

其实跟acquire()的流程大同小异，只不过多了个自己拿到资源后，还会去唤醒后继队友的操作（这才是共享嘛）。

##### releaseShared()

上一小节已经把acquireShared()说完了，这一小节就来讲讲它的反操作releaseShared()吧。此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。下面是releaseShared()的源码：

|  |
| --- |
| public final boolean releaseShared(int arg) {  if (tryReleaseShared(arg)) {//尝试释放资源  doReleaseShared();//唤醒后继结点  return true;  }  return false;  } |

此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于独占下可重入的考量；而共享模式下的releaseShared()则没有这种要求，共享模式实质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等待结点。例如，资源总量是13，A（5）和B（7）分别获取到资源并发运行，C（4）来时只剩1个资源就需要等待。A在运行过程中释放掉2个资源量，然后tryReleaseShared(2)返回true唤醒C，C一看只有3个仍不够继续等待；随后B又释放2个，tryReleaseShared(2)返回true唤醒C，C一看有5个够自己用了，然后C就可以跟A和B一起运行。而ReentrantReadWriteLock读锁的tryReleaseShared()只有在完全释放掉资源（state=0）才返回true，所以自定义同步器可以根据需要决定tryReleaseShared()的返回值。

###### doReleaseShared()

此方法主要用于唤醒后继。下面是它的源码：

|  |
| --- |
| private void doReleaseShared() {  for (;;) {  Node h = head;  if (h != null && h != tail) {  int ws = h.waitStatus;  if (ws == Node.SIGNAL) {  if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  continue;  unparkSuccessor(h);//唤醒后继  }  else if (ws == 0 &&  !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  continue;  }  if (h == head)// head发生变化  break;  }  } |

##### 小结

本节我们详解了独占和共享两种模式下获取-释放资源(acquire-release、acquireShared-releaseShared)的源码，相信大家都有一定认识了。值得注意的是，acquire()和acquireSahred()两种方法下，线程在等待队列中都是忽略中断的。AQS也支持响应中断的，acquireInterruptibly()/acquireSharedInterruptibly()即是，这里相应的源码跟acquire()和acquireSahred()差不多，这里就不再详解了。

#### 简单应用

通过前边几个章节的学习，相信大家已经基本理解AQS的原理了。这里再将“框架”一节中的一段话复制过来：

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

OK，下面我们就以AQS源码里的Mutex为例，讲一下AQS的简单应用。

##### Mutex（互斥锁）

Mutex是一个不可重入的互斥锁实现。锁资源（AQS里的state）只有两种状态：0表示未锁定，1表示锁定。下边是Mutex的核心源码：

|  |
| --- |
| class Mutex implements Lock, java.io.Serializable {  // 自定义同步器  private static class Sync extends AbstractQueuedSynchronizer {  // 判断是否锁定状态  protected boolean isHeldExclusively() {  return getState() == 1;  }  // 尝试获取资源，立即返回。成功则返回true，否则false。  public boolean tryAcquire(int acquires) {  // assert如果[boolean表达式]为true，则程序继续执行。  // assert如果为false，则程序抛出AssertionError，并终止执行。  assert acquires == 1; // 这里限定只能为1个量  if (compareAndSetState(0, 1)) {//state为0才设置为1，不可重入！  setExclusiveOwnerThread(Thread.currentThread());//设置为当前线程独占资源  return true;  }  return false;  }  // 尝试释放资源，立即返回。成功则为true，否则false。  protected boolean tryRelease(int releases) {  assert releases == 1; // 限定为1个量  if (getState() == 0)//既然来释放，那肯定就是已占有状态了。只是为了保险，多层判断！  throw new IllegalMonitorStateException();  setExclusiveOwnerThread(null);  setState(0);//释放资源，放弃占有状态  return true;  }  }  // 真正同步类的实现都依赖继承于AQS的自定义同步器！  private final Sync sync = new Sync();  //lock<-->acquire。两者语义一样：获取资源，即便等待，直到成功才返回。  public void lock() {  sync.acquire(1);  }  //tryLock<-->tryAcquire。两者语义一样：尝试获取资源，要求立即返回。成功则为true，失败则为false。  public boolean tryLock() {  return sync.tryAcquire(1);  }  //unlock<-->release。两者语文一样：释放资源。  public void unlock() {  sync.release(1);  }  //锁是否占有状态  public boolean isLocked() {  return sync.isHeldExclusively();  }  } |

同步类在实现时一般都将自定义同步器（sync）定义为内部类，供自己使用；而同步类自己（Mutex）则实现某个接口，对外服务。当然，接口的实现要直接依赖sync，它们在语义上也存在某种对应关系！！而sync只用实现资源state的获取-释放方式tryAcquire-tryRelelase，至于线程的排队、等待、唤醒等，上层的AQS都已经实现好了，我们不用关心。

除了Mutex，ReentrantLock/CountDownLatch/Semphore这些同步类的实现方式都差不多，不同的地方就在获取-释放资源的方式tryAcquire-tryRelelase。掌握了这点，AQS的核心便被攻破了！

#### 文章推荐

<https://www.cnblogs.com/waterystone/p/4920797.html>

### Condition

#### 概念

### CAS

#### 悲观锁与乐观锁

悲观锁：悲观锁思想认为如果多个线程中使用共享资源，则它们肯定会同时进行修改从而引起冲突，悲观锁的解决方式是共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：乐观锁思想认为如果多个线程使用共享资源，它们修改应该是有先后顺序的，不会同时进行修改，如果真的有冲突则后面修改失败。乐观锁的解决方式是共享资源可以由多个线程同时访问修改，对于冲突失败，让其重试直到成功即可。

#### 概念

CAS全程是compareAndSwap，维基百科上的中文称为“比较并交换”，是乐观锁的一种实现方式，涉及有三个操作数，内存位置（V）、预期值（A）和新值（B），如语句CAS(V,A,B)，当\*V等于A时，则将值替换为新值B。虽然从语言描述上来说是分为多个操作的，但实际上CAS操作是一个原子操作，是基于CPU提供的原子操作指令实现的。如下CAS实现更新的伪代码：

|  |
| --- |
| do {  A=current();//\*V赋予A，A为就是预期值  B=update();//获取B作为新值  } while(!compareAndSwap(V, A, B)); |

\*V：使用了C语言描述，表示V内存位置的值，便于描述

更新：我们所说的更新是指更新内存位置V的值，也就是改变\*V

上面伪代码将\*V赋予A，然后获取新值赋予B，最后进行交换判断直到成功，例如：

| **-** | **1** | **2** | **3** | **4** | **5** | **6** |
| --- | --- | --- | --- | --- | --- | --- |
| 线程1 | A=5 | B=6 |  |  |  | CAS:false |
| 线程2 |  |  | A=5 | B=7 | \*V=7 |  |

上面两个线程同时进入循环进行进行更新操作，第一轮循环中只有线程2更新成功，线程1因为\*V的值被线程2改变导致和预期值不一致从而失败，只能重新进入下一轮循环直到成功。

#### 优势

其实这个问题前面介绍悲观锁和乐观锁时其实已经回答，当多个线程操作时，它解决了悲观锁使用了独占锁，一次只能有一个线程进入临界区的问题，在竞争状态比较低的情况下提高了并发性能。为何说是竞争低的情况下，如果上面有很多个线程同时进入循环，那么每个线程都在占用资源执行，但每次只有一个线程能更新成功。

#### 劣势

* 当竞争很强烈时，每个线程都占用资源执行但是只有一个成功
* ABA问题，上面的例子中，如果还有一个线程3在修改，并将线程2修改后的值又变成了5，那么线程1此时是察觉不到的，它还能进行成功的执行！例如：

| * **-** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 线程1 | A=5 | B=6 |  |  |  |  |  |  | \*A=6 |
| 线程2 |  |  | A=5 | B=7 | \*V=7 |  |  |  |  |
| 线程3 |  |  |  |  |  | A=7 | B=5 | \*A=5 |  |

ABA问题解决方式：引入一个版本号，在每次更新后都加1，在上面的例子中，即时线程3将\*V修改回来原值，因为版本号不一致也会导致失败重试。

CAS在java.util.concurrent有着广泛的使用，如AtomicInteger。且对于ABA问题，Java也提供了AtomicStampedReference来处理。

#### 来源

<https://blog.csdn.net/eejron/article/details/53189090>

<http://ifeve.com/atomic-operation/>

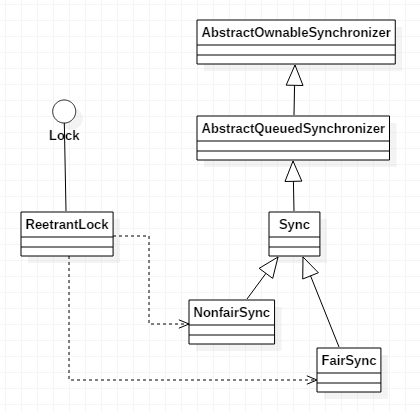
### 同步类

#### ReentrantLock

##### 概念

ReentrantLock是java concurrent包提供的一种锁实现。不同于synchronized，ReentrantLock是从代码层面实现同步的。

以下为reentrantLock的类层次结构图：



##### synchronized的局限性

synchronized是java内置的关键字，它提供了一种独占的加锁方式。synchronized的获取和释放锁由JVM实现，用户不需要显示的释放锁，非常方便。然而synchronized也有一定的局限性，例如：

* 当线程尝试获取锁的时候，如果获取不到锁会一直阻塞。
* 如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待。

JDK1.5之后发布，加入了Doug Lea实现的concurrent包。包内提供了Lock类，用来提供更多扩展的加锁功能。Lock弥补了synchronized的局限，提供了更加细粒度的加锁功能。

##### Lock简介

Lock api如下:

|  |
| --- |
| void lock();  void lockInterruptibly() throws InterruptedException;  boolean tryLock();  boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  void unlock();  Condition newCondition(); |

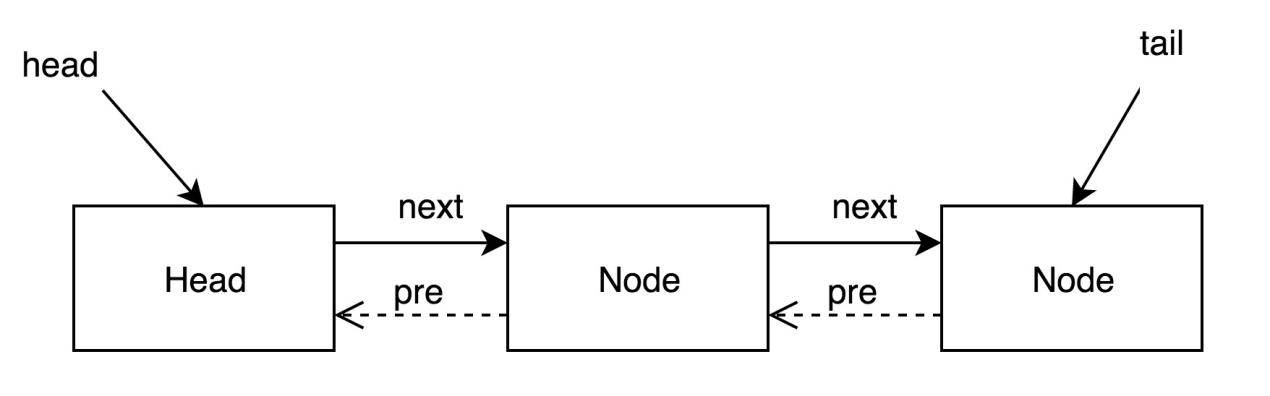
其中最常用的就是lock和unlock操作了。因为使用lock时，需要手动的释放锁，所以需要使用try..catch来包住业务代码，并且在finally中释放锁。典型使用如下：

|  |
| --- |
| private Lock lock = new ReentrantLock();  public void test(){  lock.lock();  try{  doSomeThing();  }catch (Exception e){  // ignored  }finally {  lock.unlock();  }  } |

##### AQS

AbstractQueuedSynchronizer简称AQS，是一个用于构建锁和同步容器的框架。事实上concurrent包内许多类都是基于AQS构建，例如ReentrantLock，Semaphore，CountDownLatch，ReentrantReadWriteLock，FutureTask等。AQS解决了在实现同步容器时设计的大量细节问题。

AQS使用一个FIFO的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态waitStatus。如图:



AQS中还有一个表示状态的字段state，例如ReentrantLocky用它表示线程重入锁的次数，Semaphore用它表示剩余的许可数量，FutureTask用它表示任务的状态。对state变量值的更新都采用CAS操作保证更新操作的原子性。

AbstractQueuedSynchronizer继承了AbstractOwnableSynchronizer，这个类只有一个变量：exclusiveOwnerThread，表示当前占用该锁的线程，并且提供了相应的get，set方法。

理解AQS可以帮助我们更好的理解JCU包中的同步容器。

##### 基础知识

ReentrantLock是Lock的默认实现之一。那么lock()和unlock()是怎么实现的呢？首先我们要弄清楚几个概念:

* 可重入锁。可重入锁是指同一个线程可以多次获取同一把锁。ReentrantLock和synchronized都是可重入锁。
* 可中断锁。可中断锁是指线程尝试获取锁的过程中，是否可以响应中断。synchronized是不可中断锁，而ReentrantLock则提供了中断功能。
* 公平锁与非公平锁。公平锁是指多个线程同时尝试获取同一把锁时，获取锁的顺序按照线程达到的顺序，而非公平锁则允许线程“插队”。synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁。
* CAS操作(CompareAndSwap)。CAS操作简单的说就是比较并交换。CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。” Java并发包(java.util.concurrent)中大量使用了CAS操作,涉及到并发的地方都调用了sun.misc.Unsafe类方法进行CAS操作。

##### 内部结构

ReentrantLock提供了两个构造器，分别是:

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

默认构造器初始化为NonfairSync对象，即非公平锁，而带参数的构造器可以指定使用公平锁和非公平锁。由lock()和unlock的源码可以看到，它们只是分别调用了sync对象的lock()和release(1)方法。

Sync是ReentrantLock的内部类，它的结构如下:



可以看到Sync扩展了AbstractQueuedSynchronizer。

##### NonfairSync

我们从源代码出发，分析非公平锁获取锁和释放锁的过程。

###### lock()

lock()源码如下:

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

首先用一个CAS操作，判断state是否是0（表示当前锁未被占用），如果是0则把它置为1，并且设置当前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS操作只能保证一个线程操作成功，剩下的只能乖乖的去排队啦。

“非公平”即体现在这里，如果占用锁的线程刚释放锁，state置为0，而排队等待锁的线程还未唤醒时，新来的线程就直接抢占了该锁，那么就“插队”了。

若当前有三个线程去竞争锁，假设线程A的CAS操作成功了，拿到了锁开开心心的返回了，那么线程B和C则设置state失败，走到了else里面。我们往下看acquire。

acquire(arg)

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

代码非常简洁，但是背后的逻辑却非常复杂，可见Doug Lea大神的编程功力。

1. 第一步。尝试去获取锁。如果尝试获取锁成功，方法直接返回。

tryAcquire(arg)

|  |
| --- |
| final boolean nonfairTryAcquire(int acquires) {  //获取当前线程  final Thread current = Thread.currentThread();  //获取state变量值  int c = getState();  if (c == 0) { //没有线程占用锁  if (compareAndSetState(0, acquires)) {  //占用锁成功,设置独占线程为当前线程  setExclusiveOwnerThread(current);  return true;  }  } else if (current == getExclusiveOwnerThread()) { //当前线程已经占用该锁  int nextc = c + acquires;  if (nextc < 0) // overflow  throw new Error("Maximum lock count exceeded");  // 更新state值为新的重入次数  setState(nextc);  return true;  }  //获取锁失败  return false;  } |

非公平锁tryAcquire的流程是：检查state字段，若为0，表示锁未被占用，那么尝试占用，若不为0，检查当前锁是否被自己占用，若被自己占用，则更新state字段，表示重入锁的次数。如果以上两点都没有成功，则获取锁失败，返回false。

1. 第二步，入队。由于上文中提到线程A已经占用了锁，所以B和C执行tryAcquire失败，并且入等待队列。如果线程A拿着锁死死不放，那么B和C就会被挂起。

先看下入队的过程。

先看addWaiter(Node.EXCLUSIVE)

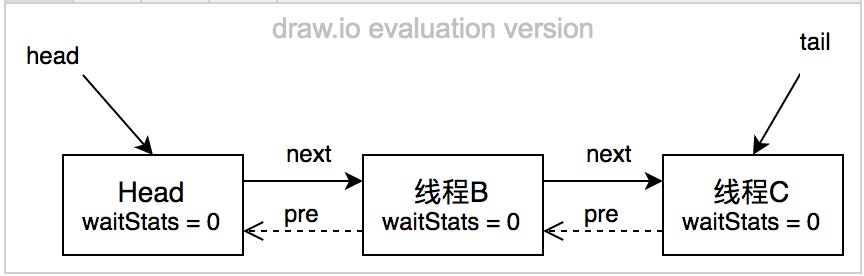
|  |
| --- |
| /\*\*  \* 将新节点和当前线程关联并且入队列  \* @param mode 独占/共享  \* @return 新节点  \*/  private Node addWaiter(Node mode) {  //初始化节点,设置关联线程和模式(独占 or 共享)  Node node = new Node(Thread.currentThread(), mode);  // 获取尾节点引用  Node pred = tail;  // 尾节点不为空,说明队列已经初始化过  if (pred != null) {  node.prev = pred;  // 设置新节点为尾节点  if (compareAndSetTail(pred, node)) {  pred.next = node;  return node;  }  }  // 尾节点为空,说明队列还未初始化,需要初始化head节点并入队新节点  enq(node);  return node;  } |

1. C线程同时尝试入队列，由于队列尚未初始化，tail==null，故至少会有一个线程会走到enq(node)。我们假设同时走到了enq(node)里。

|  |
| --- |
| /\*\*  \* 初始化队列并且入队新节点  \*/  private Node enq(final Node node) {  //开始自旋  for (;;) {  Node t = tail;  if (t == null) { // Must initialize  // 如果tail为空,则新建一个head节点,并且tail指向head  if (compareAndSetHead(new Node()))  tail = head;  } else {  node.prev = t;  // tail不为空,将新节点入队  if (compareAndSetTail(t, node)) {  t.next = node;  return t;  }  }  }  } |

这里体现了经典的自旋+CAS组合来实现非阻塞的原子操作。由于compareAndSetHead的实现使用了unsafe类提供的CAS操作，所以只有一个线程会创建head节点成功。假设线程B成功，之后B、C开始第二轮循环，此时tail已经不为空，两个线程都走到else里面。假设B线程compareAndSetTail成功，那么B就可以返回了，C由于入队失败还需要第三轮循环。最终所有线程都可以成功入队。

当B、C入等待队列后，此时AQS队列如下：



1. 第三步，挂起。B和C相继执行acquireQueued(final Node node, int arg)。这个方法让已经入队的线程尝试获取锁，若失败则会被挂起。

|  |
| --- |
| /\*\*  \* 已经入队的线程尝试获取锁  \*/  final boolean acquireQueued(final Node node, int arg) {  boolean failed = true; //标记是否成功获取锁  try {  boolean interrupted = false; //标记线程是否被中断过  for (;;) {  final Node p = node.predecessor(); //获取前驱节点  //如果前驱是head,即该结点已成老二，那么便有资格去尝试获取锁  if (p == head && tryAcquire(arg)) {  setHead(node); // 获取成功,将当前节点设置为head节点  p.next = null; // 原head节点出队,在某个时间点被GC回收  failed = false; //获取成功  return interrupted; //返回是否被中断过  }  // 判断获取失败后是否可以挂起,若可以则挂起  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  // 线程若被中断,设置interrupted为true  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

code里的注释已经很清晰的说明了acquireQueued的执行流程。假设B和C在竞争锁的过程中A一直持有锁，那么它们的tryAcquire操作都会失败，因此会走到第2个if语句中。我们再看下shouldParkAfterFailedAcquire和parkAndCheckInterrupt都做了哪些事吧。

|  |
| --- |
| /\*\*  \* 判断当前线程获取锁失败之后是否需要挂起.  \*/  private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  //前驱节点的状态  int ws = pred.waitStatus;  if (ws == Node.SIGNAL)  // 前驱节点状态为signal,返回true  return true;  // 前驱节点状态为CANCELLED  if (ws > 0) {  // 从队尾向前寻找第一个状态不为CANCELLED的节点  do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  // 将前驱节点的状态设置为SIGNAL  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  }  return false;  }    /\*\*  \* 挂起当前线程,返回线程中断状态并重置  \*/  private final boolean parkAndCheckInterrupt() {  LockSupport.park(this);  return Thread.interrupted();  } |

线程入队后能够挂起的前提是，它的前驱节点的状态为SIGNAL，它的含义是“Hi，前面的兄弟，如果你获取锁并且出队后，记得把我唤醒！”。所以shouldParkAfterFailedAcquire会先判断当前节点的前驱是否状态符合要求，若符合则返回true，然后调用parkAndCheckInterrupt，将自己挂起。如果不符合，再看前驱节点是否>0(CANCELLED)，若是那么向前遍历直到找到第一个符合要求的前驱，若不是则将前驱节点的状态设置为SIGNAL。

整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心挂起，需要去找个安心的挂起点，同时可以再尝试下看有没有机会去尝试竞争锁。

最终队列可能会如下图所示:



线程B和C都已经入队，并且都被挂起。当线程A释放锁的时候，就会去唤醒线程B去获取锁啦。

###### unlock()

unlock相对于lock就简单很多。源码如下:

|  |
| --- |
| public void unlock() {  sync.release(1);  }  public final boolean release(int arg) {  if (tryRelease(arg)) {  Node h = head;  if (h != null && h.waitStatus != 0)  unparkSuccessor(h);  return true;  }  return false;  } |

如果理解了加锁的过程，那么解锁看起来就容易多了。流程大致为先尝试释放锁，若释放成功，那么查看头结点的状态是否为SIGNAL，如果是则唤醒头结点的下个节点关联的线程，如果释放失败那么返回false表示解锁失败。这里我们也发现了，每次都只唤起头结点的下一个节点关联的线程。

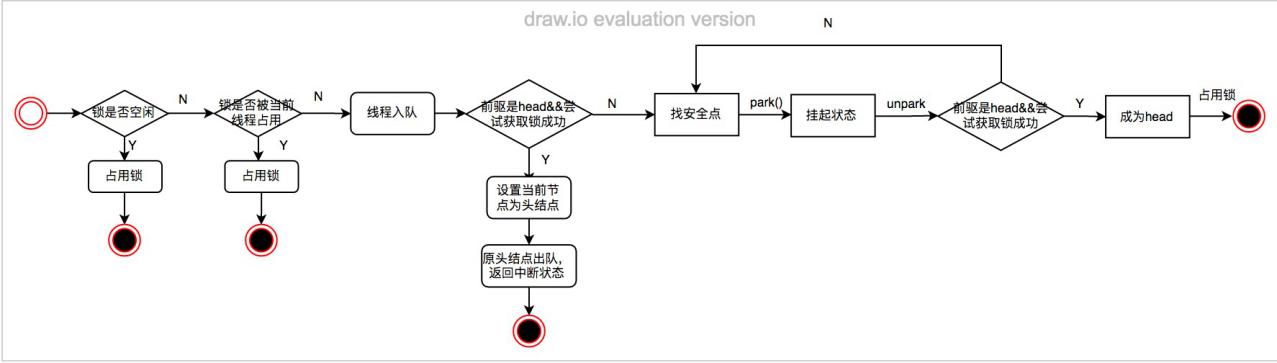
最后我们再看下tryRelease的执行过程:

|  |
| --- |
| /\*\*  \* 释放当前线程占用的锁  \* @param releases  \* @return 是否释放成功  \*/  protected final boolean tryRelease(int releases) {  // 计算释放后state值  int c = getState() - releases;  // 如果不是当前线程占用锁,那么抛出异常  if (Thread.currentThread() != getExclusiveOwnerThread())  throw new IllegalMonitorStateException();  boolean free = false;  if (c == 0) {  // 锁被重入次数为0,表示释放成功  free = true;  // 清空独占线程  setExclusiveOwnerThread(null);  }  // 更新state值  setState(c);  return free;  } |

这里入参为1。tryRelease的过程为：当前释放锁的线程若不持有锁，则抛出异常。若持有锁，计算释放后的state值是否为0，若为0表示锁已经被成功释放，并且则清空独占线程，最后更新state值，返回free。

###### 小结

用一张流程图总结一下非公平锁的获取锁的过程。



##### FairSync

公平锁和非公平锁不同之处在于，公平锁在获取锁的时候，不会先去检查state状态，而是直接执行aqcuire(1)，这里不再赘述。

##### 超时机制

在ReetrantLock的tryLock(long timeout, TimeUnit unit) 提供了超时获取锁的功能。它的语义是在指定的时间内如果获取到锁就返回true，获取不到则返回false。这种机制避免了线程无限期的等待锁释放。那么超时的功能是怎么实现的呢？我们还是用非公平锁为例来一探究竟。

|  |
| --- |
| public boolean tryLock(long timeout, TimeUnit unit)  throws InterruptedException {  return sync.tryAcquireNanos(1, unit.toNanos(timeout));  } |

还是调用了内部类里面的方法。我们继续向前探究

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

这里的语义是：如果线程被中断了，那么直接抛出InterruptedException。如果未中断，先尝试获取锁，获取成功就直接返回，获取失败则进入doAcquireNanos。tryAcquire我们已经看过，这里重点看一下doAcquireNanos做了什么。

|  |
| --- |
| /\*\*  \* 在有限的时间内去竞争锁  \* @return 是否获取成功  \*/  private boolean doAcquireNanos(int arg, long nanosTimeout)  throws InterruptedException {  // 起始时间  long lastTime = System.nanoTime();  // 线程入队  final Node node = addWaiter(Node.EXCLUSIVE);  boolean failed = true;  try {  // 又是自旋!  for (;;) {  // 获取前驱节点  final Node p = node.predecessor();  // 如果前驱是头节点并且占用锁成功,则将当前节点变成头结点  if (p == head && tryAcquire(arg)) {  setHead(node);  p.next = null; // help GC  failed = false;  return true;  }  // 如果已经超时,返回false  if (nanosTimeout <= 0)  return false;  // 超时时间未到,且需要挂起  if (shouldParkAfterFailedAcquire(p, node) &&  nanosTimeout > spinForTimeoutThreshold)  // 阻塞当前线程直到超时时间到期  LockSupport.parkNanos(this, nanosTimeout);  long now = System.nanoTime();  // 更新nanosTimeout  nanosTimeout -= now - lastTime;  lastTime = now;  if (Thread.interrupted())  //相应中断  throw new InterruptedException();  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

doAcquireNanos的流程简述为：线程先入等待队列，然后开始自旋，尝试获取锁，获取成功就返回，失败则在队列里找一个安全点把自己挂起直到超时时间过期。这里为什么还需要循环呢？因为当前线程节点的前驱状态可能不是SIGNAL，那么在当前这一轮循环中线程不会被挂起，然后更新超时时间，开始新一轮的尝试

##### 来源

<https://blog.csdn.net/u011202334/article/details/73188404>

#### ReentrantReadWriteLock

##### 概述

我们在介绍AbstractQueuedSynchronizer的时候介绍过，AQS支持独占式同步状态获取/释放、共享式同步状态获取/释放两种模式，对应的典型应用分别是ReentrantLock和Semaphore，AQS还可以混合两种模式使用，读写锁ReentrantReadWriteLock就是如此。

设想以下情景：我们在系统中有一个多线程访问的缓存，多个线程都可以对缓存进行读或写操作，但是读操作远远多于写操作，要求写操作要线程安全，且写操作执行完成要求对当前的所有读操作马上可见。

分析上面的需求：因为有多个线程可能会执行写操作，因此多个线程的写操作必须同步串行执行；而写操作执行完成要求对当前的所有读操作马上可见，这就意味着当有线程正在读的时候，要阻塞写操作，当正在执行写操作时，要阻塞读操作。一个简单的实现就是将数据直接加上互斥锁，同一时刻不管是读还是写线程，都只能有一个线程操作数据。但是这样的问题就是如果当前只有N个读线程，没有写线程，这N个读线程也要傻呵呵的排队读，尽管其实是可以安全并发提高效率的。因此理想的实现是：

当有写线程时，则写线程独占同步状态。

当没有写线程时只有读线程时，则多个读线程可以共享同步状态。

读写锁就是为了实现这种效果而生。

##### 使用示例

我们先来看一下读写锁怎么使用，这里我们基于hashmap（本身线程不安全）做一个多线程并发安全的缓存：

|  |
| --- |
| public class ReadWriteCache {  private static Map<String, Object> data = new HashMap<>();  private static ReadWriteLock lock = new ReentrantReadWriteLock(false);  private static Lock rlock = lock.readLock();  private static Lock wlock = lock.writeLock();  public static Object get(String key) {  rlock.lock();  try {  return data.get(key);  } finally {  rlock.unlock();  }  }  public static Object put(String key, Object value) {  wlock.lock();  try {  return data.put(key, value);  } finally {  wlock.unlock();  }  }  } |

限于篇幅我们只实现2个方法，get和put。从代码可以看出，我们先创建一个 ReentrantReadWriteLock 对象，构造函数 false 代表是非公平的（非公平的含义和ReentrantLock相同）。然后通过readLock、writeLock方法分别获取读锁和写锁。在做读操作的时候，也就是get方法，我们要先获取读锁；在做写操作的时候，即put方法，我们要先获取写锁。

通过以上代码，我们就构造了一个线程安全的缓存，达到我们之前说的：写线程独占同步状态，多个读线程可以共享同步状态。

##### 源码分析

我们先来看下 ReentrantReadWriteLock 类的整体结构：

|  |
| --- |
| public class ReentrantReadWriteLock implements ReadWriteLock,  java.io.Serializable {  private final ReentrantReadWriteLock.ReadLock readerLock;  /\*\* Inner class providing writelock \*/  private final ReentrantReadWriteLock.WriteLock writerLock;  /\*\* Performs all synchronization mechanics \*/  final Sync sync;  public ReentrantReadWriteLock(boolean fair) {  sync = fair ? new FairSync() : new NonfairSync();  readerLock = new ReadLock(this);  writerLock = new WriteLock(this);  }  public ReentrantReadWriteLock.WriteLock writeLock() { return writerLock; }  public ReentrantReadWriteLock.ReadLock readLock() { return readerLock; }  abstract static class Sync extends AbstractQueuedSynchronizer {}  static final class NonfairSync extends Sync {}  static final class FairSync extends Sync {}  public static class ReadLock implements Lock, java.io.Serializable {}  public static class WriteLock implements Lock, java.io.Serializable {}  } |

可以看到，在公平锁与非公平锁的实现上，与ReentrantLock一样，也是有一个继承AQS的内部类Sync，然后NonfairSync和FairSync都继承Sync，通过构造函数传入的布尔值决定要构造哪一种Sync实例。

读写锁比ReentrantLock多出了两个内部类：ReadLock和WriteLock， 用来定义读锁和写锁，然后在构造函数中，会构造一个读锁和一个写锁实例保存到成员变量 readerLock 和 writerLock。我们在上面的示例中使用到的 readLock() 和 writeLock() 方法就是返回这两个成员变量保存的锁实例。

我们在Sync类中可以看到下列代码：

|  |
| --- |
| static final int SHARED\_SHIFT = 16;  //每次要让共享锁+1，就应该让state加 1<<16  static final int SHARED\_UNIT = (1 << SHARED\_SHIFT);  //每种锁的最大重入数量  static final int MAX\_COUNT = (1 << SHARED\_SHIFT) - 1;  static final int EXCLUSIVE\_MASK = (1 << SHARED\_SHIFT) - 1;  /\*\* Returns the number of shared holds represented in count \*/  static int sharedCount(int c) { return c >>> SHARED\_SHIFT; }  /\*\* Returns the number of exclusive holds represented in count \*/  static int exclusiveCount(int c) { return c & EXCLUSIVE\_MASK; } |

可以看到主要是几个位移操作，通过上面的整体结构，我们知道了在读写锁内保存了读锁和写锁的两个实例。之前在ReentrantLock中，我们知道锁的状态是保存在Sync实例的state字段中的(继承自父类AQS)，现在有了读写两把锁，然而可以看到还是只有一个Sync实例，那么一个Sync实例的state是如何同时保存两把锁的状态的呢？答案就是用了位分隔：

state字段是32位的int，读写锁用state的低16位保存写锁(独占锁)的状态；高16位保存读锁(共享锁)的状态。

因此要获取独占锁当前的重入数量，就是 state & ((1 << 16) -1) (即 exclusiveCount 方法)

要获取共享锁当前的重入数量，就是 state >>> 16 (即 sharedCount 方法)

下面我们具体看写锁和读锁的实现。

###### 写锁

看下WriteLock类中的lock和unlock方法：

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

可以看到就是调用的独占式同步状态的获取与释放，因此真实的实现就是Sync的 tryAcquire和 tryRelease。

###### 写锁的获取

看下tryAcquire:

|  |
| --- |
| protected final boolean tryAcquire(int acquires) {  Thread current = Thread.currentThread();  int c = getState();  int w = exclusiveCount(c); //获取独占锁的重入数  if (c != 0) {  // 当前state不为0，此时：如果写锁状态为0说明读锁此时被占用返回false；如果写锁状态不为0且写锁没有被当前线程持有返回false  if (w == 0 || current != getExclusiveOwnerThread())  return false;  if (w + exclusiveCount(acquires) > MAX\_COUNT)  throw new Error("Maximum lock count exceeded"); //写锁重入数溢出  // Reentrant acquire  setState(c + acquires);  return true;  }  //到这里了说明state为0，尝试直接cas。writerShouldBlock是为了实现公平或非公平策略的  if (writerShouldBlock() ||  !compareAndSetState(c, c + acquires))  return false;  setExclusiveOwnerThread(current);  return true;  } |

逻辑很简单，直接看注释就能理解。

###### 写锁的释放

看下tryRelease:

|  |
| --- |
| protected final boolean tryRelease(int releases) {  if (!isHeldExclusively())  throw new IllegalMonitorStateException(); //非独占模式直接抛异常  int nextc = getState() - releases;  boolean free = exclusiveCount(nextc) == 0;  if (free)  setExclusiveOwnerThread(null); //如果独占模式重入数为0了，说明独占模式被释放  setState(nextc); //不管独占模式是否被释放，更新独占重入数  return free;  } |

逻辑很简单，直接看注释就能理解。

###### 读锁

类似于写锁，读锁的lock和unlock的实际实现对应Sync的 tryAcquireShared 和 tryReleaseShared方法。

###### 读锁的获取

|  |
| --- |
| protected final int tryAcquireShared(int unused) {  Thread current = Thread.currentThread();  int c = getState();  if (exclusiveCount(c) != 0 &&  getExclusiveOwnerThread() != current)  return -1; //如果独占模式被占且不是当前线程持有，则获取失败  int r = sharedCount(c);  //如果公平策略没有要求阻塞且重入数没有到达最大值，则直接尝试CAS更新state  if (!readerShouldBlock() &&  r < MAX\_COUNT &&  compareAndSetState(c, c + SHARED\_UNIT)) {  //更新成功后会在firstReaderHoldCount中或readHolds(ThreadLocal类型的)的本线程副本中记录当前线程重入数（浅蓝色代码），这是为了实现jdk1.6中加入的getReadHoldCount()方法的，这个方法能获取当前线程重入共享锁的次数(state中记录的是多个线程的总重入次数)，加入了这个方法让代码复杂了不少，但是其原理还是很简单的：如果当前只有一个线程的话，还不需要动用ThreadLocal，直接往firstReaderHoldCount这个成员变量里存重入数，当有第二个线程来的时候，就要动用ThreadLocal变量readHolds了，每个线程拥有自己的副本，用来保存自己的重入数。  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); //用来处理CAS没成功的情况，逻辑和上面的逻辑是类似的，就是加了无限循环  } |

下面这个方法就不用细说了，和上面的处理逻辑类似，加了无限循环用来处理CAS失败的情况。

|  |
| --- |
| final int fullTryAcquireShared(Thread current) {  HoldCounter rh = null;  for (;;) {  int c = getState();  if (exclusiveCount(c) != 0) {  if (getExclusiveOwnerThread() != current)  return -1;  // else we hold the exclusive lock; blocking here  // would cause deadlock.  } else if (readerShouldBlock()) {  // Make sure we're not acquiring read lock reentrantly  if (firstReader == current) {  // assert firstReaderHoldCount > 0;  } else {  if (rh == null) {  rh = cachedHoldCounter;  if (rh == null || rh.tid != getThreadId(current)) {  rh = readHolds.get();  if (rh.count == 0)  readHolds.remove();  }  }  if (rh.count == 0)  return -1;  }  }  if (sharedCount(c) == MAX\_COUNT)  throw new Error("Maximum lock count exceeded");  if (compareAndSetState(c, c + SHARED\_UNIT)) {  if (sharedCount(c) == 0) {  firstReader = current;  firstReaderHoldCount = 1;  } else if (firstReader == current) {  firstReaderHoldCount++;  } else {  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;  }  }  } |

###### 读锁的释放

|  |
| --- |
| protected final boolean tryReleaseShared(int unused) {  Thread current = Thread.currentThread();  //浅蓝色代码也是为了实现jdk1.6中加入的getReadHoldCount()方法，在更新当前线程的重入数。  if (firstReader == current) {  // assert firstReaderHoldCount > 0;  if (firstReaderHoldCount == 1)  firstReader = null;  else  firstReaderHoldCount--;  } else {  HoldCounter rh = cachedHoldCounter;  if (rh == null || rh.tid != getThreadId(current))  rh = readHolds.get();  int count = rh.count;  if (count <= 1) {  readHolds.remove();  if (count <= 0)  throw unmatchedUnlockException();  }  --rh.count;  }  //这里是真正的释放同步状态的逻辑，就是直接同步状态-SHARED\_UNIT，然后CAS更新，没啥好说的  for (;;) {  int c = getState();  int nextc = c - SHARED\_UNIT;  if (compareAndSetState(c, nextc))  // Releasing the read lock has no effect on readers,  // but it may allow waiting writers to proceed if  // both read and write locks are now free.  return nextc == 0;  }  } |

###### 补充内容

通过上面的源码分析，我们可以发现一个现象：

* 在线程持有读锁的情况下，该线程不能取得写锁(因为获取写锁的时候，如果发现当前的读锁被占用，就马上获取失败，不管读锁是不是被当前线程持有)
* 在线程持有写锁的情况下，该线程可以继续获取读锁（获取读锁时如果发现写锁被占用，只有写锁没有被当前线程占用的情况才会获取失败）

仔细想想，这个设计是合理的：因为当线程获取读锁的时候，可能有其他线程同时也在持有读锁，因此不能把获取读锁的线程“升级”为写锁；而对于获得写锁的线程，它一定独占了读写锁，因此可以继续让它获取读锁，当它同时获取了写锁和读锁后，还可以先释放写锁继续持有读锁，这样一个写锁就“降级”为了读锁。

综上：

* 一个线程要想同时持有写锁和读锁，必须先获取写锁再获取读锁；
* 写锁可以“降级”为读锁；
* 读锁不能“升级”为写锁。

###### 总结

读写锁还是很实用的，因为一般场景下，数据的并发操作都是读多于写，在这种情况下，读写锁能够提供比排它锁更好的并发性。

在读写锁的实现方面，本来以为会比较复杂，结果看完源码的感受也是快刀切西瓜，看来AQS的设计真的很棒，在AQS的基础上构建的组件实现都很简单。

###### 来源

<https://www.cnblogs.com/sheeva/p/6480116.html>

#### Atomic

java.util.concurrent中提供了atomic原子包，可以实现原子操作（atomic operation），以AtomicInteger为例：

|  |
| --- |
| public class AtomicExample {  public static void main(String[] args) {  ExecutorService ExecutorService = null;  try {  ExecutorService = Executors.newFixedThreadPool(100);  final AtomicInteger atomicInteger = new AtomicInteger();  final CountDownLatch latch = new CountDownLatch(1000);  for(int i = 0; i < 1000; i++){  ExecutorService.submit(new Runnable() {  @Override  public void run() {  atomicInteger.incrementAndGet();  latch.countDown();  }  });  }  latch.await();  System.out.println("atomicInteger结果：" + atomicInteger.get());  } catch (Exception e) {  e.printStackTrace();  }finally {  ExecutorService.shutdown();  }  }  } |

##### 源码分析

incrementAndGet函数：

|  |
| --- |
| private volatile int value;  public final int incrementAndGet() {  for (;;) {  //获取当前值value  int current = get();  int next = current + 1;  //循环执行到递增成功  if (compareAndSet(current, next))  return next;  }  } |

这里value是volatile类型，确保此线程能获取到最新值。increment操作是利用CAS来保证每次操作的原子性，同时利用for循环的自旋确保每次increment成功。

#### Semaphore

##### 概念

Semaphore也叫信号量，在JDK1.5被引入，可以用来控制同时访问特定资源的线程数量，通过协调各个线程，以保证合理的使用资源。

Semaphore内部维护了一组虚拟的许可，许可的数量可以通过构造函数的参数指定。

* 访问特定资源前，必须使用acquire方法获得许可，如果许可数量为0，该线程则一直阻塞，直到有可用许可。
* 访问资源后，使用release释放许可。

Semaphore和ReentrantLock类似，获取许可有公平策略和非公平许可策略，默认情况下使用非公平策略。

那么什么是公平Semaphore呢？？按照FIFO队列中的顺序分配Semaphore所管理的许可就是公平的。

非公平Semaphore信号量对于任何申请许可的线程来说，都是第一时间看是否有多余的许可，如果有则给此线程，如果没有则进队列排队等待，而不是此线程直接进AQS队列排队等待按顺序来拿到许可，利用此间隙来分配许可可以提高并发量。但是会引发一个问题：越活跃的线程越能够拿到许可，造成“饥渴死”现象。

##### 使用场景

Semaphore可以用于做流量控制，比如系统最大并发数限制，我们在请求进入系统时，请求信号量申请，如果申请成功，则进行业务处理，否则等待，业务处理完毕后，释放信号量。能用到信号量的地方很多，这里就不一一举例了。简单用法示例：

|  |
| --- |
| public class SemaphoreTest {  public static Semaphore semaphore = new Semaphore(3);  public void test(){  try {  semaphore.acquire();  System.out.println(Thread.currentThread().getName() + "，睡眠前时间：" + System.currentTimeMillis());  Thread.sleep(2000);  System.out.println(Thread.currentThread().getName() + "，睡眠后时间：" + System.currentTimeMillis());  semaphore.release();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  public static void main(String[] args) {  ExecutorService executorService = Executors.newFixedThreadPool(10);  try{  final CountDownLatch latch = new CountDownLatch(10);  final SemaphoreTest semaphoreTest = new SemaphoreTest();  for(int i = 0; i < 10; i++){  executorService.execute(new Runnable() {  public void run() {  semaphoreTest.test();  latch.countDown();  }  });  }  latch.await();  System.out.println("执行完毕");  }catch(Exception e){  e.printStackTrace();  }finally{  executorService.shutdown();  }  }  } |

##### 源码分析

Semaphore内部主要是重写AQS的tryAcquireShared和tryReleaseShared方法。

###### 构造函数

|  |
| --- |
| //非公平的构造函数  public Semaphore(int permits) {  sync = new NonfairSync(permits);  }  //通过fair参数决定公平性  public Semaphore(int permits, boolean fair) {  sync = fair ? new FairSync(permits) : new NonfairSync(permits);  } |

构造函数分为默认和指定公平策略2种方式，其中默认的为非公平策略。

###### acquire源代码（非公平策略）

|  |
| --- |
| // Semaphore类方法  public void acquire() throws InterruptedException {  sync.acquireSharedInterruptibly(1);  }  // Semaphore类方法  public void acquire(int permits) throws InterruptedException {  if (permits < 0) throw new IllegalArgumentException();  sync.acquireSharedInterruptibly(permits);  }  // AQS类方法  public final void acquireSharedInterruptibly(int arg)  throws InterruptedException {  if (Thread.interrupted())  throw new InterruptedException();  if (tryAcquireShared(arg) < 0) //小于0表示没有获得共享锁  //没有获取锁，则当前线程进入AQS中的doAcquireSharedInterruptibly  方法进入等待队列并挂起，直到被唤醒。  doAcquireSharedInterruptibly(arg);  }  // Semaphore类重写AQS的方法  protected int tryAcquireShared(int acquires) {  return nonfairTryAcquireShared(acquires);  }  //Semaphore类中同步器新增的方法  final int nonfairTryAcquireShared(int acquires) {  for (;;) {  int available = getState();//获取去中的信号量数  int remaining = available - acquires; //剩余信号量数  //1.信号量数大于0,获取共享锁,并设置执行compareAndSetState(available, remaining),返回剩余信号量数  //2.信号量数小于等于0,直接返回负数  if (remaining < 0 ||  compareAndSetState(available, remaining))  return remaining;  }  } |

公平策略其实就是在申请许可的时候，先查看队列中是否有等待的线程，如果有则进入等待队列。

|  |
| --- |
| protected int tryAcquireShared(int acquires) {  for (;;) {  if (hasQueuedPredecessors())  return -1;  int available = getState();  int remaining = available - acquires;  if (remaining < 0 ||  compareAndSetState(available, remaining))  return remaining;  }  } |

acquire方法总结如下：

* 尝试在非公平模式下获取一个许可（这里在公平和非公平策略下略有不同）。如果获取到则立即返回并将许可计数器减一，如果没有获取到，则进入AQS队列
* 进入到AQS队列自旋等待，当此节点的前驱是头结点后，又开始尝试获取锁。直至成功获取或中断取消。

###### release源代码

|  |
| --- |
| // Semaphore方法  public void release() {  sync.releaseShared(1);  }  //AQS提供的方法  public final boolean releaseShared(int arg) {  if (tryReleaseShared(arg)) {  doReleaseShared();  return true;  }  return false;  }  // Semaphore类重写了AQS的方法  protected final boolean tryReleaseShared(int releases) {  for (;;) {  int current = getState();  int next = current + releases;  if (next < current) // overflow  throw new Error("Maximum permit count exceeded");  if (compareAndSetState(current, next))  return true;  }  }  //AQS的方法  private void doReleaseShared() {  for (;;) {  Node h = head;  if (h != null && h != tail) {  int ws = h.waitStatus;  if (ws == Node.SIGNAL) {  if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  continue; // loop to recheck cases  unparkSuccessor(h);  }  else if (ws == 0 &&  !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  continue; // loop on failed CAS  }  if (h == head) // loop if head changed  break;  }  } |

如果熟悉AQS，这里释放许可（释放许可不涉及公平和非公平策略）的代码就容易看懂了（建议先看明白AQS的章节）。

#### CyclicBarrier

##### 概念

CyclicBarrier也叫同步屏障，在JDK1.5被引入，可以让一组线程达到一个屏障时被阻塞，直到最后一个线程达到屏障时，所以被阻塞的线程才能继续执行。同时CyclicBarrier可以被重用。

CyclicBarrier好比一扇门，默认情况下关闭状态，堵住了线程执行的道路，直到所有线程都就位，门才打开，让所有线程一起通过。

CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行barrierAction这个Runnable对象，方便处理更复杂的业务场景。

##### 应用场景

比如100米短跑，以下为简单代码示例：

|  |
| --- |
| public class CyclicBarrierTest {  private CyclicBarrier barrier;    public CyclicBarrierTest(int num){  this.barrier = new CyclicBarrier(num);  }    public void test(String name) throws InterruptedException, BrokenBarrierException{  System.out.println("运动员"+name+"已就位");  barrier.await();  System.out.println("所有运动员开始跑...");  Random random =new Random();  double completeTime = random.nextDouble() \* 3 + 9;  System.out.println(name + "完成100米短跑时间为：" + completeTime);  }    public static void main(String[] args) {  ExecutorService executorService = Executors.newFixedThreadPool(10);  final CyclicBarrierTest athlete = new CyclicBarrierTest(10);    System.out.println("各就各位");  for(int i = 0; i < 10; i++){  final String name = "运动员" + (i + 1);  executorService.execute(new Runnable() {  public void run() {  try {  athlete.test(name);  } catch (Exception e) {  e.printStackTrace();  }  }  });  }  }  } |

执行结果如下：

|  |
| --- |
| 各就各位  运动员运动员1已就位  运动员运动员3已就位  运动员运动员2已就位  运动员运动员4已就位  运动员运动员5已就位  运动员运动员6已就位  运动员运动员7已就位  运动员运动员8已就位  运动员运动员9已就位  运动员运动员10已就位  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  所有运动员开始跑...  运动员6完成100米短跑时间为：10.733005845314144  运动员5完成100米短跑时间为：9.497466169069922  运动员8完成100米短跑时间为：9.609492950433284  运动员1完成100米短跑时间为：9.46695047451573  运动员9完成100米短跑时间为：11.305097084882487  运动员10完成100米短跑时间为：11.533693301598763  运动员2完成100米短跑时间为：11.92620235135325  运动员4完成100米短跑时间为：10.025138543676464  运动员7完成100米短跑时间为：9.565148853188887  运动员3完成100米短跑时间为：10.238692476984678 |

##### 源码分析

CyclicBarrier实现主要基于ReentrantLock，在CyclicBarrier的内部定义了一个Lock对象，每当一个线程调用CyclicBarrier的await方法时，将剩余拦截的线程数减1，然后判断剩余拦截数是否为0，如果不是，进入Lock对象的条件队列等待。如果是，执行barrierAction对象的Runnable方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从await方法返回，再从CyclicBarrier的await方法中返回。具体分析如下：

|  |
| --- |
| //每一代的类，默认barrier是不损坏的，true：表示已损坏  private static class Generation {  boolean broken = false;  }  /\*\*锁对象 \*/  private final ReentrantLock lock = new ReentrantLock();  /\*\* Condition to wait on until tripped \*/  private final Condition trip = lock.newCondition();  /\*\* The number of parties，可以叫做初始化栅栏数量 \*/  private final int parties;  /\* The command to run when tripped，附加指令，等待结束后立即执行的线程 \*/  private final Runnable barrierCommand;  /\*\* The current generation \*/  private Generation generation = new Generation(); |

其中Generation用来控制屏障的循环使用，如果generation.broken为true的话，说明这个屏障已经损坏，当某个线程await的时候，直接抛出异常。

await方法分析：

|  |
| --- |
| public int await() throws InterruptedException, BrokenBarrierException {  try {  return dowait(false, 0L);//直接调用方法进行处理  } catch (TimeoutException toe) {  throw new Error(toe); // cannot happen;  }  }  private int dowait(boolean timed, long nanos)  throws InterruptedException, BrokenBarrierException,  TimeoutException {  final ReentrantLock lock = this.lock;  lock.lock();//当前线程加锁  try {  final Generation g = generation;  if (g.broken) // 如果分代损坏，抛出屏障损坏异常  throw new BrokenBarrierException();  // 如果线程中断，中止barrier，唤醒（重置count计数器）所有线程并抛出异常  if (Thread.interrupted()) {  breakBarrier();  throw new InterruptedException();  }  int index = --count;  if (index == 0) { // 如果index为0，说明所有线程都已经到位  boolean ranAction = false;  try {  final Runnable command = barrierCommand;  // 附加指令如果不为null的话，就需要执行附加指令  if (command != null)//  command.run();  ranAction = true;  // 本代完成，设置下一代，让CyclicBarrier可以重复使用  nextGeneration();  return 0;  } finally {  // 这里如果执行附加命令出问题了，中止barrier，唤醒等待线程，重置count计数器。  if (!ranAction)  breakBarrier();  }  }  // index不为0的情况，说明当前线程不是要到位的最后一个线程，所以一直循环等待，直到收到唤醒信号  for (;;) {  try {  // 超时等待的timed默认为false，所以这里执行Condition.await方法  if (!timed)  trip.await();  //如果设置了超时等待，并且等待时间大于0，则执行awaitNanos  //awaitNanos：当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。  else if (nanos > 0L)  nanos = trip.awaitNanos(nanos);  } catch (InterruptedException ie) {//异常处理，看了上面这里就好理解了  if (g == generation && ! g.broken) {  breakBarrier();  throw ie;  } else {  // We're about to finish waiting even if we had not  // been interrupted, so this interrupt is deemed to  // "belong" to subsequent execution.  Thread.currentThread().interrupt();  }  }  //线程唤醒后需要再次判断障碍状态  if (g.broken)  throw new BrokenBarrierException();  //线程唤醒后判断  if (g != generation)  return index;  if (timed && nanos <= 0L) {  breakBarrier();  throw new TimeoutException();  }  }  } finally {  lock.unlock();  }  }  //重置下一代  private void nextGeneration() {  // signal completion of last generation  trip.signalAll();//唤醒所有线程  // set up next generation  count = parties;  generation = new Generation();//重置障碍状态  }  //终止方法  private void breakBarrier() {  generation.broken = true; //将障碍状态设置为true  count = parties;  trip.signalAll();//唤醒所有等待线程  } |

总结：

* 等待直到在这个barrier中有parties(构造函数中自己指定的)个线程调用了await方法。如果当前线程不是最后一个到达的，则由于线程调度的原因被禁用，睡眠直至以下事情发生：

|  |
| --- |
| 1、最后一个线程的到达。  2、其它的线程中断了当前线程  3、其它的线程中断了和一样正在等待的线程  4、其它线程等待barrier超时  5、其它线程在这个barrier处调用了reset方法 |

* 如果当前线程在进入这个方法时由中断标志或者是在等待的过程中被中断，则会抛中断异常，并且中断标志被清除。
* 当存在线程正在等待的时候，有线程调用了reset方法。或者是当调用await方法时barrier的isBroken已经被broken。则会抛BrokenBarrierException异常。
* 当存在线程在等待的时候被中断，则所有其他正在被等待的线程都将跑抛BrokenBarrierException异常，并且barrier就处于broken状态。
* 如果当前线程是最后一个到达，并且在构造方法中有一个非空的barrier action，则在允许其他线程继续运行之前 ，当前线程需要执行此Runnable的run方法。如果在执行此barrier action期间发生了异常，此异常就会在此线程中传播。并且barrier的状态将会设置为broken。
* 返回值：返回的是当前线程到达barrier的索引， 例如，第一个到达barrier处的线程返回的是patiers-1. 最后一个返回的是 0。

#### CountDownLatch

##### 概念

CountDownLatch是一个用来控制并发的很常见的工具，它允许一个或者多个线程（这里一个或多个线程我们在这里成为主线程）等待其他的线程（这里其他线程我们成为非主线程）完成某件事情之后才开始执行，比如我们需要处理一个大的文件夹内的文件（里面包含很多文件），所有文件处理完之后，需要做最后的处理（比如记录信息或者提示所有问题都已处理），这时候就可以使用CountDownLatch来实现。

##### 源码分析

CountDownLatch是基于AQS实现的，主要是实现AQS的tryAcquireShared和tryReleaseShared方法。以下是await方法分析：

|  |
| --- |
| public void await() throws InterruptedException {  sync.acquireSharedInterruptibly(1);  }  //1.获取共享锁  public final void acquireSharedInterruptibly(int arg)  throws InterruptedException {  //判断线程是否为中断状态，如果是抛出interruptedException  if (Thread.interrupted())  throw new InterruptedException();  //尝试获取共享锁，尝试成功就返回，否则加入队列等待。  if (tryAcquireShared(arg) < 0)  doAcquireSharedInterruptibly(arg);  }  //尝试获取共享锁，重写AQS里面的方法  protected int tryAcquireShared(int acquires) {  //锁状态 == 0，表示非主线程都执行完毕，即是可获取的状态，否则锁是不可获取的状态  return (getState() == 0) ? 1 : -1;  }  //AQS中的方法，线程获取锁不成功，则进入入队等操作，这里采用公平锁策略  private void doAcquireSharedInterruptibly(int arg)  throws InterruptedException {  final Node node = addWaiter(Node.SHARED);  boolean failed = true;  try {  for (;;) {  final Node p = node.predecessor();  if (p == head) {  int r = tryAcquireShared(arg);  if (r >= 0) {  setHeadAndPropagate(node, r);  p.next = null; // help GC  failed = false;  return;  }  }  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  throw new InterruptedException();  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

countDown分析如下：

|  |
| --- |
| public void countDown() {  sync.releaseShared(1);  }  //AQS的方法，释放队列中的线程  public final boolean releaseShared(int arg) {  if (tryReleaseShared(arg)) {//如果state=0，则调用doReleaseShared释放锁  doReleaseShared();  return true;  }  return false;  }  //确认是否达到释放锁的方法  protected boolean tryReleaseShared(int releases) {  for (;;) {  int c = getState();//获取state的值  //如果值为0，直接返回false，这种情况主线程会一直处于等待状态，当然这里作者应该是也不知道怎么处理为好（不等待也是有问题的）  if (c == 0)  return false;  int nextc = c-1;  if (compareAndSetState(c, nextc))  return nextc == 0;//非主线程都执行完后，返回true  }  }  //AQS的方法，释放等待队列中的线程  private void doReleaseShared() {  for (;;) {  Node h = head;  if (h != null && h != tail) {  int ws = h.waitStatus;  if (ws == Node.SIGNAL) {  if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  continue; // loop to recheck cases  unparkSuccessor(h);  }  else if (ws == 0 &&  !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))  continue; // loop on failed CAS  }  if (h == head) // loop if head changed  break;  }  } |

##### 比较

CountDownLatch类与Thread类的join方法比较：

* join是只有线程执行完毕后，主线程才执行，而CountDownLatch是所有线程都调用countDown方法后，就会执行，这里非主线程未必已经执行完毕；
* join只有一个线程等待其他线程完成任务，CountDownLatch可以做到1个或多个线程等待其他线程完成任务。
* CountDownLatch类提供更多功能，比如等待超时的功能。

CountDownLatch类与CyclicBarrier类比较：

* CountDownLatch是不可重用的，CyclicBarrier类是可重用的；
* 看起来他们作用类似，其实作用完全不同。CountDownLatch是一个或多个线程等待其他线程都调用countDown方法后才开始执行，而CyclicBarrier是多个线程先分别执行，直到所有的线程都到达await方法，才开始继续执行，这里先到达await方法的线程就暂时等待。CountDownLatch侧重一组线程等待另外一组线程的执行情况，而CyclicBarrier侧重一组线程在做某些“重要事情”前需要等组内所有线程可以开始做这些“重要事情”时才开始做。

#### Exchanger

##### 概念

当一个线程到达exchange调用点时，如果它的伙伴线程此前已经调用了此方法，那么它的伙伴会被调度唤醒并与之进行对象交换，然后各自返回。如果它的伙伴还没到达交换点，那么当前线程将会被挂起，直至伙伴线程到达——完成交换正常返回；或者当前线程被中断——抛出中断异常；又或者是等候超时——抛出超时异常。

##### 应用场景

Exchanger可以用于遗传算法，遗传算法里需要选出两个人作为交配对象，这时候会交换两人的数据，并使用交叉规则得出2个交配结果。

Exchanger也可以用于校对工作。比如我们需要将纸制银流通过人工的方式录入成电子银行流水，为了避免错误，采用AB岗两人进行录入，录入到Excel之后，系统需要加载这两个Excel，并对这两个Excel数据进行校对，看看是否录入的一致。

##### 源码分析

### 同步队列

#### 阻塞队列

##### 概念

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

常用的阻塞队列：

* ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
* LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
* PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
* DelayQueue：一个使用优先级队列实现的无界阻塞队列。
* SynchronousQueue：一个不存储元素的阻塞队列。
* LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
* LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

##### 应用场景

阻塞队列常用于生产者和消费者的场景，生产者是向队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器。

##### ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证访问者公平的访问队列，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常情况下为了保证公平性会降低吞吐量。

##### LinkedBlockingQueue

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

##### PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比较器comparator来指定元素的排序规则。元素按照升序排列。

##### DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。我们可以将DelayQueue运用在以下应用场景：

* 缓存系统的设计：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。
* 定时任务调度。使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行，从比如TimerQueue就是使用DelayQueue实现的。

关于延迟队列后面再做详细说明。

##### SynchronousQueue

SynchronousQueue是一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景,比如在一个线程中使用的数据，传递给另外一个线程使用，SynchronousQueue的吞吐量高于LinkedBlockingQueue 和 ArrayBlockingQueue。

##### LinkedTransferQueue

LinkedTransferQueue是一个由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列LinkedTransferQueue多了tryTransfer和transfer方法。

transfer方法：如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。

##### LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque多了addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast等方法，以First单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素。另外插入方法add等同于addLast，移除方法remove等效于removeFirst。但是take方法却等同于takeFirst，不知道是不是Jdk的bug，使用时还是用带有First和Last后缀的方法更清楚。在初始化LinkedBlockingDeque时可以初始化队列的容量，用来防止其再扩容时过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

##### 源码分析

这里以ArrayBlockingQueue源码来分析。

##### 来源

http://ifeve.com/java-blocking-queue/

#### 非阻塞队列

##### 概念

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。

#### 延迟队列

##### 概念

DelayQueue是一个无界的BlockingQueue，用于放置实现了Delayed接口的对象，其中的对象只能在其到期时才能从队列中取走。这种队列是有序的，即队头对象的延迟到期时间最长。

##### 应用场景

电商中订单支付，如果超过10分钟未支付，订单就会取消，还有考试系统中，考生必须在120分钟后全部交卷，延迟1分钟发送短信等。

##### 源码分析

### ConcurrentHashMap

### 线程池

#### 概念

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

#### 风险

#### 提交任务

我们可以使用execute提交的任务，但是execute方法没有返回值，所以无法判断任务知否被线程池执行成功。我们也可以使用submit 方法来提交任务，它会返回一个future,那么我们可以通过这个future来判断任务是否执行成功，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。这里需要注意的是使用submit方法性能会比execute性能差不少，如果不需要返回值的时候，建议选择execute方法来提交任务。

#### 线程池关闭

我们可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池，但是它们的实现原理不同，shutdown的原理是只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。shutdownNow的原理是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会首先将线程池的状态设置成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。

只要调用了这两个关闭方法的其中一个，isShutdown方法就会返回true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow。

#### ThreadPoolExecutor

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

##### 源码分析

#### 线程池种类

##### newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：

* 工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。
* 如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。
* 在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

##### newFixedThreadPool

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

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

##### newSingleThreadExecutor

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

##### newScheduleThreadPool

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

#### 来源

<http://ifeve.com/java-threadpool/>

https://www.cnblogs.com/aaron911/p/6213808.html

### Fork框架

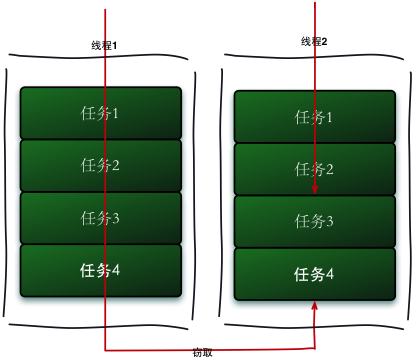
#### 概念

从JDK1.7开始，Java提供Fork/Join框架用于并行执行任务，它的思想就是讲一个大任务分割成若干小任务，最终汇总每个小任务的结果得到这个大任务的结果。

它的模型大致是这样的：线程池中的每个线程都有自己的工作队列（PS：这一点和ThreadPoolExecutor不同，ThreadPoolExecutor是所有线程公用一个工作队列，所有线程都从这个工作队列中取任务），当自己队列中的任务都完成以后，会从其它线程的工作队列中偷一个任务执行，这样可以充分利用资源。

#### 工作窃取

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



那么为什么需要使用工作窃取算法呢？

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

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

#### 源码分析

### 总结

建议大家多看看源码，同时多看下下面网站的内容：

http://ifeve.com/java-concurrency-thread-directory/