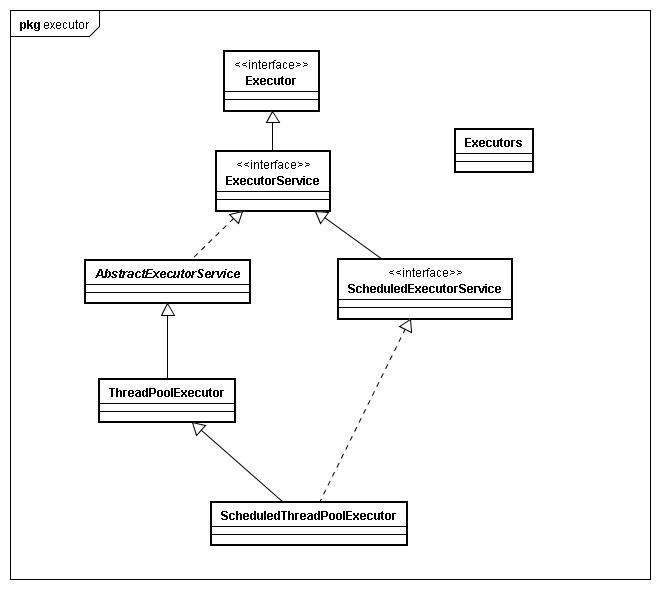
**1. ExecutorService**  
  
Java从1.5开始正式提供了并发包,而这个并发包里面除了原子变量,synchronizer,并发容器,另外一个非常重要的特性就是线程池.对于线程池的意义,我们这边不再多说.

上图是线程池的主体类图,ThreadPoolExecutor是应用最为广泛的一个线程池实现(我也将在接下来的文字中详细描述我对这个类的理解和执行机制),ScheduledThreadPoolExecutor则在ThreadPoolExecutor上提供了定时执行的等附加功能,这个可以从ScheduledExecutorService接口的定义中看出来.Executors则类似工厂方法,提供了几个非常常用的线程池初始化方法.

**ThreadPoolExecutor**

这个类继承了AbstractExecutorService抽象类, AbstractExecutorService主要的职责有2部分,一部分定义和实现提交任务的方法(3个submit方法的实现) ,实例化FutureTask并且交给子类执行,另外一部分实现invokeAny,invokeAll方法.留给子类的方法为execute方法,也就是Executor接口定义的方法.

|  |  |
| --- | --- |
| //实例化一个FutureTask,交给子类的execute方法执行.这种设计能够保证callable和runnable的执行接口方法的一致性(FutureTask包装了这个差别) | |
| public <T> Future<T> submit(Runnable task, T result) { |

|  |
| --- |
| if (task == null) throw new NullPointerException(); |
| RunnableFuture<T> ftask = newTaskFor(task, result); |

|  |  |
| --- | --- |
| execute(ftask); | |
| return ftask; |

|  |
| --- |
| } |
|  |

|  |  |
| --- | --- |
| protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) { | |
| return new FutureTask<T>(runnable, value); |

|  |
| --- |
| } |

关于FutureTask这个类的实现,我在前面的JAVA LOCK代码浅析有讲过其实现原理,主要的思想就是关注任务完成与未完成的状态,任务提交线程get()结果时被park住,等待任务执行完成被唤醒,任务执行线程在任务执行完毕后设置结果,并且unpark对应线程并且让其得到执行结果.

回到ThreadPoolExecutor类.ThreadPoolExecutor需要实现除了我们刚才说的execute(Runnable command)方法外,还得实现ExecutorService接口定义的部分方法.但ThreadPoolExecutor所提供的不光是这些,以下根据我的理解来列一下它所具有的特性  
1. execute流程  
2. 池  
3. 工作队列  
4. 饱和拒绝策略  
5. 线程工厂  
6. beforeExecute和afterExecute扩展

execute方法的实现有个机制非常重要,当当前线程池线程数量小于corePoolSize,那么生成一个新的worker并把提交的任务置为这个工作线程的头一个执行任务,如果大于corePoolSize,那么会试着将提交的任务塞到workQueue里面供线程池里面的worker稍后执行,并不是直接再起一个worker,但是当workQueue也满,并且当前线程池小于maxPoolSize,那么起一个新的worker并将该任务设为该worker执行的第一个任务执行,大于maxPoolSize,workQueue也满负荷,那么调用饱和策略里面的行为.

worker线程在执行完一个任务之后并不会立刻关闭,而是尝试着去workQueue里面取任务,如果取不到,根据策略关闭或者保持空闲状态.所以submit任务的时候,提交的顺序为**核心线程池——工作队列——扩展线程池**.

池包括核心池,扩展池(**2者的线程在同一个hashset中,这里只是为了方便才这么称呼,并不是分离的**),核心池在池内worker没有用完的情况下,只要有任务提交都会创建新的线程,其代表线程池正常处理任务的能力.扩展池,是在核心线程池用完,并且工作队列也已排满任务的情况下才会开始初始化线程,其代表的是线程池超出正常负载时的解决方案,一旦任务完成,并且试图从workQueue取不到任务,那么会比较当前线程池与核心线程池的大小,大于核心线程池数的worker将被销毁.

|  |  |
| --- | --- |
| Runnable getTask() { | |
| for (;;) { |

|  |
| --- |
| try { |
| int state = runState; | |

|  |  |
| --- | --- |
| //>SHUTDOWN就是STOP或者TERMINATED | |
| //直接返回 |

|  |  |
| --- | --- |
| if (state > SHUTDOWN) | |
| return null; |

|  |
| --- |
| Runnable r; |
| //如果是SHUTDOWN状态,那么取任务,如果有 | |

|  |
| --- |
| //将剩余任务执行完毕,否则就结束了 |
| if (state == SHUTDOWN)  // Help drain queue | |

|  |
| --- |
| r = workQueue.poll(); |
| //如果不是以上状态的(也就是RUNNING状态的),那么如果当前池大于核心池数量, | |

|  |
| --- |
| //或者允许核心线程池取任务超时就可以关闭,那么从任务队列取任务, |
| //如果超出keepAliveTime,那么就返回null了,也就意味着这个worker结束了 |

|  |
| --- |
| else if (poolSize > corePoolSize || allowCoreThreadTimeOut) |
| r = workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS); |

|  |  |
| --- | --- |
| //如果当前池小于核心池,并且不允许核心线程池取任务超时就关闭,那么take(),直到拿到任务或者被interrupt | |
| else |

|  |
| --- |
| r = workQueue.take(); |
| //如果经过以上判定,任务不为空,那么返回任务 | |

|  |  |
| --- | --- |
| if (r != null) | |
| return r; |

|  |  |
| --- | --- |
| //如果取到任务为空,那么判定是否可以退出 | |
| if (workerCanExit()) { |

|  |  |
| --- | --- |
| //如果整个线程池状态变为SHUTDOWN或者TERMINATED,那么将所有worker interrupt (如果正在执行,那继续让其执行) | |
| if (runState >= SHUTDOWN) // Wake up others |

|  |  |
| --- | --- |
| interruptIdleWorkers(); | |
| return null; |

|  |
| --- |
| } |
| // Else retry | |

|  |
| --- |
| } catch (InterruptedException ie) { |
| // On interruption, re-check runState | |

|  |  |
| --- | --- |
| } | |
| } |

|  |  |
| --- | --- |
| } | |
|  |

|  |  |
| --- | --- |
| //worker从workQueue中取不到数据的时候调用此方法,以决定自己是否跳出取任务的无限循环,从而结束此worker的运行 | |
| private boolean workerCanExit() { |

|  |  |
| --- | --- |
| final ReentrantLock mainLock = this.mainLock; | |
| mainLock.lock(); |

|  |  |
| --- | --- |
| boolean canExit; | |
| try { |

|  |
| --- |
| /\*\*//\* |
| \*线程池状态为stop或者terminated, | |

|  |
| --- |
| \*或者任务队列里面任务已经为空, |
| \*或者允许线程池线程空闲超时(实现方式是从工作队列拿最多keepAliveTime的任务,超过这个时间就返回null了)并且 | |

|  |  |
| --- | --- |
| \*当前线程池大于corePoolSize(>1) | |
| \*那么允许线程结束 |

|  |
| --- |
| \*static final int RUNNING    = 0; |
| \*static final int SHUTDOWN   = 1; |

|  |
| --- |
| \*static final int STOP       = 2; |
| \*static final int TERMINATED = 3; |

|  |
| --- |
| \*/ |
| canExit = runState >= STOP || | |

|  |
| --- |
| workQueue.isEmpty() || |
| (allowCoreThreadTimeOut && | |

|  |  |
| --- | --- |
| poolSize > Math.max(1,corePoolSize)); | |
| } finally { |

|  |  |
| --- | --- |
| mainLock.unlock(); | |
| } |

|  |  |
| --- | --- |
| return canExit; | |
| } |

当提交任务是,线程池都已满,并且工作队列也无空闲位置的情况下,ThreadPoolExecutor会执行reject操作,JDK提供了四种reject策略,包括AbortPolicy(直接抛RejectedException Exception),CallerRunsPolicy(提交任务线程自己执行,当然这时剩余任务也将无法提交),DiscardOldestPolicy(将线程池的workQueue任务队列里面最老的任务剔除,将新任务丢入),DiscardPolicy(无视,忽略此任务,并且立即返回).实例化ThreadPoolExecutor时,如果不指定任何饱和策略,默认将使用AbortPolicy.

个人认为这些饱和策略并不十分理想,特别是在应用既要保证快速,又要高可用的情况下,我的想法是能够加入超时等待策略,也就是提交线程时线程池满,能够park住提交任务的线程,一旦有空闲,能在第一时间通知到等待线程. 这个实际上和主线程执行相似,但是主线程执行期间即使线程池有大量空闲也不会立即可以提交任务,效率上后者可能会比较低,特别是执行慢速任务.

实例化Worker的时候会调用ThreadFactory的addThread(Runnable r)方法返回一个Thread,这个线程工厂是可以在ThreadPoolExecutor实例化的时候指定的,如果不指定,那么将会使用DefaultThreadFactory, 这个也就是提供给使用者命名线程,线程归组,是否是demon等线程相关属性设置的机会.

beforeExecute和afterExecute是提供给使用者扩展的,这两个方法会在worker runTask之前和run完毕之后分别调用.JDK注释里 Doug Lea(concurrent包作者)展示了beforeExecute一个很有趣的示例.代码如下.

|  |  |
| --- | --- |
| class PausableThreadPoolExecutor extends ThreadPoolExecutor { | |
| private boolean isPaused; |

|  |
| --- |
| private ReentrantLock pauseLock = new ReentrantLock(); |
| private Condition unpaused = pauseLock.newCondition(); |

|  |
| --- |
|  |
| public PausableThreadPoolExecutor() { super(); } | |

|  |
| --- |
|  |
| protected void beforeExecute(Thread t, Runnable r) { | |

|  |  |
| --- | --- |
| super.beforeExecute(t, r); | |
| pauseLock.lock(); |

|  |
| --- |
| try { |
| while (isPaused) unpaused.await(); | |

|  |  |
| --- | --- |
| } catch (InterruptedException ie) { | |
| t.interrupt(); |

|  |
| --- |
| } finally { |
| pauseLock.unlock(); | |

|  |  |
| --- | --- |
| } | |
| } |

|  |
| --- |
|  |
| public void pause() { | |

|  |  |
| --- | --- |
| pauseLock.lock(); | |
| try { |

|  |  |
| --- | --- |
| isPaused = true; | |
| } finally { |

|  |  |
| --- | --- |
| pauseLock.unlock(); | |
| } |

|  |
| --- |
| } |
|  |

|  |  |
| --- | --- |
| public void resume() { | |
| pauseLock.lock(); |

|  |
| --- |
| try { |
| isPaused = false; | |

|  |  |
| --- | --- |
| unpaused.signalAll(); | |
| } finally { |

|  |  |
| --- | --- |
| pauseLock.unlock(); | |
| } |

|  |
| --- |
| } |
| } | |

使用这个线程池,用户可以随时调用pause中止剩余任务执行,当然也可以使用resume重新开始执行剩余任务.

**ScheduledThreadPoolExecutor**

ScheduledThreadPoolExecutor是一个很实用的类,它的实现核心是基于DelayedWorkQueue.从ScheduledThreadPoolExecutor的继承结构上来看,各位应该能够看出些端倪来,就是ScheduledThreadPoolExecutor将ThreadPoolExecutor中的任务队列设置成了DelayedWorkQueue,这也就是说,线程池Worker从任务队列中取的一个任务,需要等待这个队列中最短超时任务的超时,也就是实现定时的效果.所以ScheduledThreadPoolExecutor所做的工作其实是比较少的.主要就是实现任务的实例化并加入工作队列,以及支持scheduleAtFixedRate和scheduleAtFixedDelay这种周期性任务执行.

|  |
| --- |
| public ScheduledThreadPoolExecutor(int corePoolSize,ThreadFactory threadFactory) { |
| super(corePoolSize, Integer.MAX\_VALUE, 0, TimeUnit.NANOSECONDS,newDelayedWorkQueue(), threadFactory); |

|  |
| --- |
| } |

对于scheduleAfFixedRate和scheduleAtFiexedDelay这种周期性任务支持,是由ScheduledThreadPoolExecutor内部封装任务的ScheduledFutureTask来实现的.这个类在执行任务后,对于周期性任务,它会处理周期时间,并将自己再次丢入线程池的工作队列,从而达到周期执行的目的.

|  |
| --- |
| private void runPeriodic() { |
| boolean ok = ScheduledFutureTask.super.runAndReset(); | |

|  |
| --- |
| boolean down = isShutdown(); |
| // Reschedule if not cancelled and not shutdown or policy allows | |

|  |  |
| --- | --- |
| if (ok && (!down ||(getContinueExistingPeriodicTasksAfterShutdownPolicy() && !isStopped()))) { | |
| long p = period; |

|  |
| --- |
| if (p > 0) |
| time += p; | |

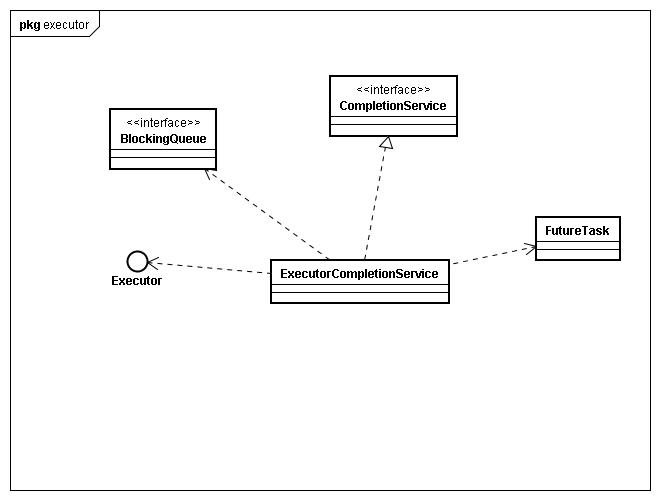
|  |
| --- |
| else |
| time = triggerTime(-p); | |

|  |
| --- |
|  |
| ScheduledThreadPoolExecutor.super.getQueue().add(this); | |

|  |
| --- |
| } |
| // This might have been the final executed delayed | |

|  |  |
| --- | --- |
| // task.  Wake up threads to check. | |
| else if (down) |

|  |  |
| --- | --- |
| interruptIdleWorkers(); | |
| } |

**2. CompletionService**  
  
**ExecutorCompletionService**

CompletionService定义了线程池执行任务集,可以依次拿到任务执行完毕的Future,ExecutorCompletionService是其实现类,先举个例子,如下代码,这个例子中,需要注意ThreadPoolExecutor核心池一定保证能够让任务提交并且马上执行,而不是放到等待队列中去,那样次序将会无法控制,CompletionService也将失去效果(其实核心池中的任务完成顺序还是准确的).

|  |
| --- |
| public static void main(String[] args) throws InterruptedException, ExecutionException{ |
| ThreadPoolExecutor es=new ThreadPoolExecutor(10, 15, 2000, TimeUnit.MILLISECONDS,new ArrayBlockingQueue<Runnable>(10),new ThreadPoolExecutor.AbortPolicy()); |

|  |  |
| --- | --- |
| CompletionService<String> cs=new ExecutorCompletionService<String>(es); | |
| cs.submit(new Callable<String>() { |

|  |
| --- |
| @Override |
| public String call() throws Exception { | |

|  |
| --- |
| Thread.currentThread().sleep(1000); |
| return "i am sleeped 1000 milliseconds"; | |

|  |
| --- |
| } |
| }); | |

|  |
| --- |
|  |
| cs.submit(new Callable<String>() { | |

|  |
| --- |
| @Override |
| public String call() throws Exception { | |

|  |
| --- |
| Thread.currentThread().sleep(5000); |
| return "i am sleeped 5000 milliseconds"; | |

|  |
| --- |
| } |
| }); | |

|  |
| --- |
|  |
| cs.submit(new Callable<String>() { | |

|  |
| --- |
| @Override |
| public String call() throws Exception { | |

|  |
| --- |
| Thread.currentThread().sleep(4000); |
| return "i am sleeped 4000 milliseconds"; | |

|  |
| --- |
| } |
| }); | |

|  |
| --- |
|  |
| cs.submit(new Callable<String>() { | |

|  |
| --- |
| @Override |
| public String call() throws Exception { | |

|  |
| --- |
| Thread.currentThread().sleep(2000); |
| return "i am sleeped 2000 milliseconds"; | |

|  |  |
| --- | --- |
| } | |
| }); |

|  |
| --- |
|  |
| for(int i=0;i<4;i++){ | |

|  |
| --- |
| Future<String> fu=cs.take(); |
| System.out.println(fu.get()); | |

|  |  |
| --- | --- |
| } | |
| } |

执行结果:  
i am sleeped 1000 milliseconds  
i am sleeped 2000 milliseconds  
i am sleeped 4000 milliseconds  
i am sleeped 5000 milliseconds  
从执行结果看来,我们发现先完成的任务先被拿出来了,直到所有任务被执行完毕,也就是CompletionService的效果达到了.

ExecutorCompletionService并不复杂,关键的一个点就是它的内部类QueueingFuture继承了FutureTask类,并且实现了done()方法,done()方法是在线程池任务执行完毕,最后调用FutureTask的方法(这在 JAVA LOCK代码浅析(http://www.blogjava.net/BucketLi/archive/2010/09/30/333471.html)一文中对于FutureTask代码解析有提到)

QueueingFuture的done()方法实现是将执行完的任务(FutureTask)丢入全局的完成队列中(completionQueue),那么take是从这个blockingqueue中取元素.也就是任务完成就会有元素,即生产者消费者.

这种实现的思想是将原本在单个FutureTask上的等待转化为在BlockingQueue上的等待,即对全部FutureTask的等待,从而达到哪个先完成,哪个就可取执行结果的效果.

|  |  |
| --- | --- |
| private class QueueingFuture extends FutureTask<Void> { | |
| QueueingFuture(RunnableFuture<V> task) { |

|  |  |
| --- | --- |
| super(task, null); | |
| his.task = task; |

|  |
| --- |
| } |
| protected void done() { completionQueue.add(task); } | |

|  |  |
| --- | --- |
| private final Future<V> task; | |
| } |

总结:  
JUC提供的线程池体系核心是在ThreadPoolExecutor, 而ScheduledThreadPoolExecutor和ExecutorCompletionService只是对其扩展,这里没有去细讲Executors这个便捷类,这个类提供很多便捷的线程池构建方法.各位使用的时候不妨去看下.