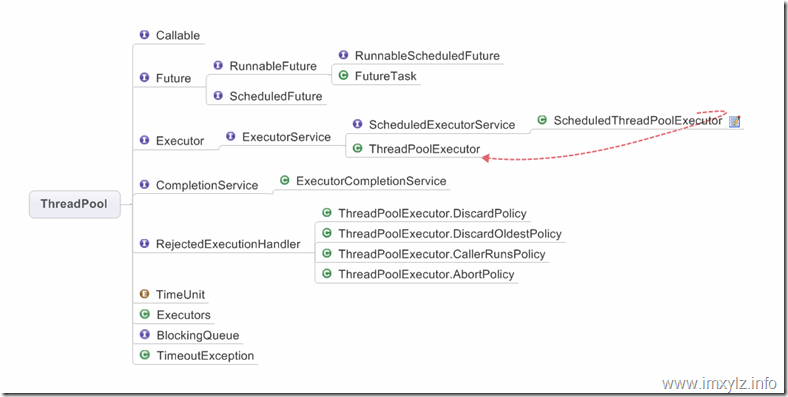
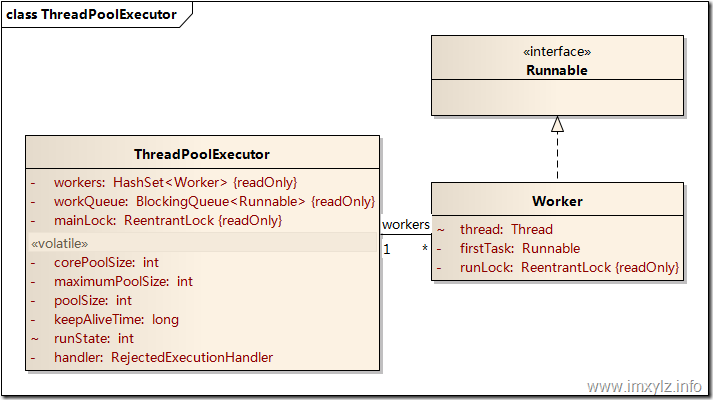
线程池的源码及原理[JDK1.6实现]

**1.线程池的包含的内容**

[](http://www.blogjava.net/images/blogjava_net/xylz/Windows-Live-Writer/-part-1-_8E6F/ThreadPool2_2.png)

**2.线程池的数据结构【核心类ThreadPoolExecutor】：**



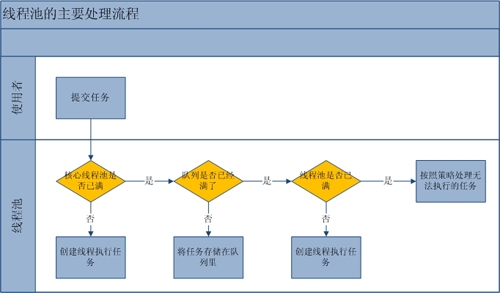
worker：工作类，一个worker代表启动了一个线程，它启动后会**循环执行workQueue**里面的所有任务

workQueue：任务队列，用于存放待执行的任务

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。

**线程池原理：**预先启动一些线程，线程无限循环从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已。如此反复。

**3.线程池任务submit及执行流程**



a.一个任务提交，如果线程池大小没达到corePoolSize，则每次都启动一个worker也就是一个线程来立即执行

b.如果来不及执行，则把多余的线程放到workQueue，等待已启动的worker来循环执行

c.如果队列workQueue都放满了还没有执行，则在maximumPoolSize下面启动新的worker来循环执行workQueue

d.如果启动到maximumPoolSize还有任务进来，线程池已达到满负载，此时就执行任务拒绝RejectedExecutionHandler

 Java Code **线程池核心的代码**

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 |  | // 流程就是：没达到corePoolSize，创建worker执行，达到corePoolSize加入workQueue //           workQueue满了且在maximumPoolSize下，创建新worker，达到maximumPoolSize，执行reject public void execute(Runnable command) {     if (command == null)         throw new NullPointerException();      // 1：poolSize达到corePoolSize，执行3把任务加入workQueue     // 2：poolSize没达到，执行addIfUnderCorePoolSize()在corePoolSize内创建新worker立即执行任务     //    如果达到corePoolSize，则同上执行3     if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {         // 3:workQueue满了，执行5         if (runState == RUNNING && workQueue.offer(command)) {             if (runState != RUNNING || poolSize == 0) {                 // 4:如果线程池关闭，执行拒绝策略                 //   如果poolSize==0，新启动一个线程执行队列内任务                 ensureQueuedTaskHandled(command);             }             // 5:在maximumPoolSize内创建新worker立即执行任务             //   如果达到maximumPoolSize，执行6拒绝策略         } else if (!addIfUnderMaximumPoolSize(command))             // 6:拒绝策略             reject(command); // is shutdown or saturated     } } |

从上面代码可以看出，一个任务提交什么时候**立即执行**呢？

runState == RUNNING && ( poolSize < corePoolSize || (workQueue.isFull() && poolSize < maxnumPoolSize))

**4.工作Worker的原理**

上面讲过线程池创建线程其实是委托给Worker这个对象完成的。worker会循环获取工作队列的任务来完成

 Java Code **工作线程Worker执行**

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 |  | public void run() {     try {         Runnable task = firstTask;         firstTask = null;         // getTask()是从workQueue里面阻塞获取任务，**如果getTask()返回null则终结本线程**         while (task != null || (task = getTask()) != null) {             runTask(task);             task = null;         }     } finally {         // 走到这里代表这个worker或者说这个线程由于线程池关闭或超过aliveTime需要关闭了         workerDone(this);     } } |

**5.线程的销毁**

keepAliveTime：代表的就是线程空闲后多久后销毁，线程的销毁是通过worker的getTask()来实现的。

一般来说，Worker会循环获取getTask()，**如果getTask()返回null则工作线程worker终结**，那我们再看看什么时候getTask()返回null

 Java Code Worker的getTask方法

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 |  | Runnable getTask() {     for (;;) {         try {             int state = runState;             if (state > SHUTDOWN)                 return null;             Runnable r;             if (state == SHUTDOWN)  // Help drain queue                 r = workQueue.poll();             else if (poolSize > corePoolSize || allowCoreThreadTimeOut)                 // **在poolSize大于corePoolSize**或允许核心线程超时时                 // 阻塞超时获取有可能获取到null，此时worker线程销毁                 **r = workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS);**             else                 r = workQueue.take();             if (r != null)                 return r;             // 这里是是否运行worker线程销毁的判断             if (**workerCanExit()**) {                 if (runState >= SHUTDOWN)                     // STOP或TERMINATED状态，终止空闲worker                     interruptIdleWorkers();                 **return null;**// 这里返回null，代表工作线程worker销毁             }             // 其他：retry，继续循环         } catch (InterruptedException ie) {             // On interruption, re-check runState         }     } } |

**6.线程池关闭**

平缓关闭 shutdown：已经启动的任务全部执行完毕，同时不再接受新的任务

立即关闭 shutdownNow：取消所有正在执行和未执行的任务

具体参考源码

**7.线程池的监控**

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

*taskCount：线程池需要执行的任务数量。*

*completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于taskCount。*

*largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。*

*getPoolSize:线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不+ getActiveCount：获取活动的线程数。*

通过扩展线程池进行监控。通过继承线程池并重写线程池的beforeExecute，afterExecute和terminated方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

**8.线程池调优[更多可参考：线程池与工作队列]**

调整线程池的大小 - 线程池的最佳大小取决于可用处理器的数目以及工作队列中的任务的性质。

调整线程池的大小基本上就是**避免两类错误：线程太少或线程太多。**

**a.CPU限制的任务，提高CPU利用率。**

    在运行于具有 N 个处理器机器上的计算限制的应用程序中，在线程数目接近 N 时添加额外的线程可能会改善总处理能力，而在线程数目超过 N 时添加额外的线程将不起作用。事实上，太多的线程甚至会降低性能，因为它会导致额外的环境切换开销。

     若在一个具有 N 个处理器的系统上只有一个工作队列，其中全部是计算性质的任务，在线程池具有**N 或 N+1**个线程时一般会获得最大的 CPU 利用率。

**b.I/O限制的任务（例如，从套接字读取 HTTP 请求的任务）**

    需要让池的大小超过可用处理器的数目，因为并不是所有线程都一直在工作。通过使用概要分析，您可以或得一些数据，并计算出大概的线程池大小。

    Amdahl 法则提供很好的近似公式。用 WT 表示每项任务的平均等待时间，ST 表示每项任务的平均服务时间（计算时间）。则 WT/ST 是每项任务等待所用时间的百分比。对于 N 处理器系统，池中可以近似有**N\*(1+WT/ST)** 个线程。

**c.综合考虑线程池性能瓶颈**

    a.处理器利用率

    b.随着线程池的增长，您可能会碰到调度程序、可用内存方面的限制，或者其它系统资源方面的限制，例如套接字、打开的文件句柄或数据库连接等的数目。

**9.线程池扩展 - 延时线程池 ScheduledThreadPoolExecutor**

    ScheduledThreadPoolExecutor是在普通线程池的基础上增加了两个功能，**一是延时执行+定时执行，二是重复执行**

    定时Executor的流程在大体上与普通线程池一致，因此它继承于ThreadPoolExecutor，**对于问题1，它采用了DelayedQueue来实现此功能。对于问题2，定时Executor每次执行完调用ThreadPoolExecutor.runAndReset()重置状态，然后重新把任务加入到Delayed队列中**

    定时Executor在外部Runnable的基础上套了一个ScheduledFutureTask，其核心源码如下：

 Java Code **普通任务的外部封装Future**

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 |  | // 加入的任务外部封装了ScheduledFutureTask，继承于FutureTask，因此也可以获取任务结果 private class ScheduledFutureTask<V> extends FutureTask<V> implements RunnableScheduledFuture<V> {     // 省略部分代码      // 周期性运行，执行完成就把任务加入到delay队列中     private void runPeriodic() {         // 这里重置线程池状态         boolean ok = ScheduledFutureTask.super.runAndReset();         boolean down = isShutdown();         // Reschedule if not cancelled and not shutdown or policy allows         if (ok && (!down ||                    (getContinueExistingPeriodicTasksAfterShutdownPolicy() &&                     !isTerminating()))) {             long p = period;             if (p > 0)                 time += p;             else                 time = now() - p;             // 重复把任务加入到线程池delay队列中             ScheduledThreadPoolExecutor.super.getQueue().add(this);         } else if (down)             interruptIdleWorkers();     }      // 线程池调用的run方法     public void run() {         if (isPeriodic())             runPeriodic();         else             ScheduledFutureTask.super.run();     } } |

**10.线程池扩展 - ExecutorCompletionService**

ExecutorCompletionService描述的是：我们提交一堆任务到线程池，**线程池在任务执行完成后把结果FutureTask放到FIFO队列中**，然后我们就可以来对结果进行操作，比如说汇总，累加等等。

其实现原理很简单，就是利用FutureTask的done()扩展方法，把此Future加入到queue，请看源码

 Java Code **ExecutorCompletionService核心代码**

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 |  | public class ExecutorCompletionService<V> implements CompletionService<V> {     // 部分代码省略      // 外部Future的封装类     private class QueueingFuture extends FutureTask<Void> {         QueueingFuture(RunnableFuture<V> task) {             super(task, null);             this.task = task;         }         // 这里把Future加入到 completionQueue         protected void done() {             completionQueue.add(task);         }         private final Future<V> task;     }      public Future<V> submit(Callable<V> task) {         if (task == null) throw new NullPointerException();         RunnableFuture<V> f = newTaskFor(task);         // 对f外层又包了一层 QueueingFuture         executor.execute(new QueueingFuture(f));         return f;     }      // 外部则可通过 completionQueue 来获取已完成的任务Future     public Future<V> take() throws InterruptedException {         return completionQueue.take();     } } |