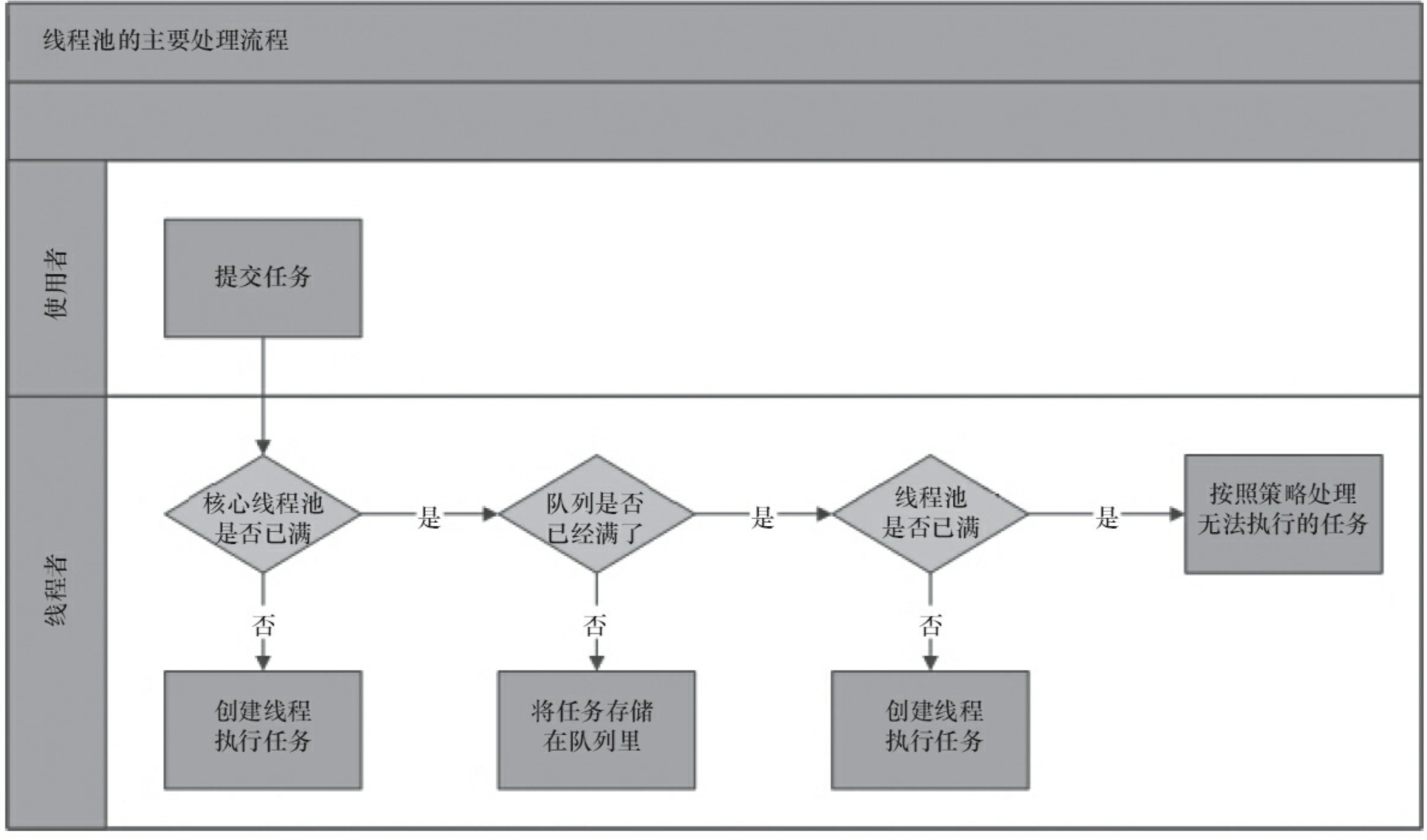
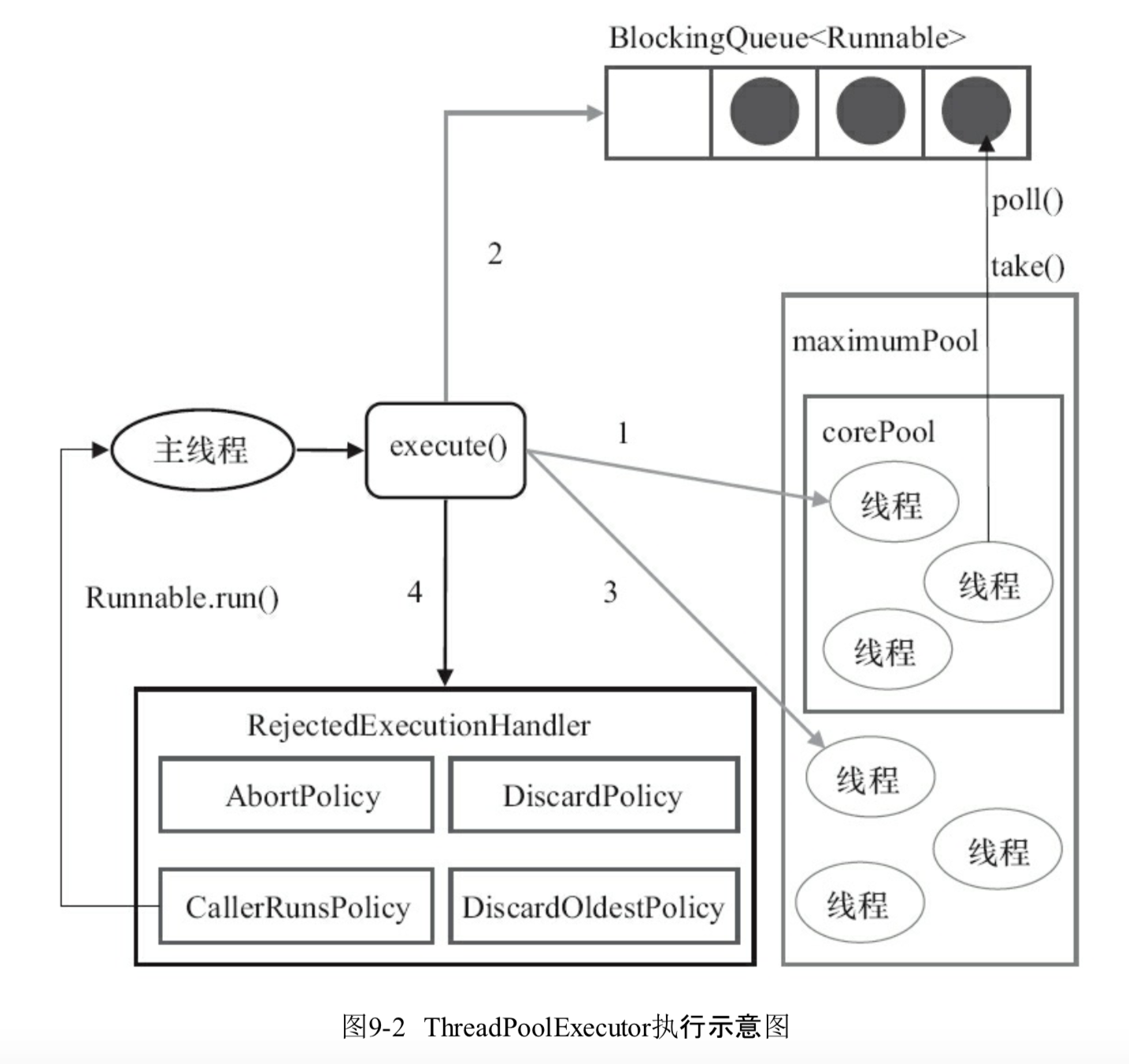
# Java并发编程的艺术

# Java中的线程池

## 9.1 实现原理





ThreadPoolExecutor执行execute方法：

1. 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（需要获取全局锁）
2. 如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue；
3. 如果无法将任务加入BlockingQueue(对列已满)，则创建新的线程来处理任务(需要获取全局锁)
4. 如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法

ThreadPoolExecutor的几个字段和状态

|  |
| --- |
| // ctl的二级制的高位3位是状态值，低位29位是池中运行的数量  private final AtomicInteger ctl = new AtomicInteger(*ctlOf*(*RUNNING*, 0));  // *COUNT\_BITS = 32 - 3 ，即位数* private static final int *COUNT\_BITS* = Integer.*SIZE* - 3;  // 最大容量是2^29-1 private static final int *CAPACITY* = (1 << *COUNT\_BITS*) - 1;  // runState is stored in the high-order bits  // RUNNING 运行状态是个负数 private static final int *RUNNING* = -1 << *COUNT\_BITS*;  // 始终是0值,不接受新任务，但是处理对列中的任务 private static final int *SHUTDOWN* = 0 << *COUNT\_BITS*;  // 不接受新任务，不处理对列中的任务，且对进行重的任务进行中断 private static final int *STOP* = 1 << *COUNT\_BITS*;  // 所有任务已经停止，工作线程数量为0，这是个过度状态，在这个状态时，将要执行钩子方法*terminated()* private static final int *TIDYING* = 2 << *COUNT\_BITS*;  // 执行完*terminated()方法后的状态* private static final int *TERMINATED* = 3 << *COUNT\_BITS*;  // Packing and unpacking ctl  // 从ctl中解包出状态 private static int runStateOf(int c) { return c & ~*CAPACITY*; }  // 从ctl中解包出工作线程数量 private static int workerCountOf(int c) { return c & *CAPACITY*; }  // 将运行状态和工作线程数量包装在ctl这个原子类中 private static int ctlOf(int rs, int wc) { return rs | wc; } |

ThreadPoolExecutor.execute

|  |
| --- |
| public void execute(Runnable command) {  if (command == null)  throw new NullPointerException();  /\*  \* Proceed in 3 steps:  \*  \* 1. If fewer than corePoolSize threads are running, try to  \* start a new thread with the given command as its first  \* task. The call to addWorker atomically checks runState and  \* workerCount, and so prevents false alarms that would add  \* threads when it shouldn't, by returning false.  \* 1.正在执行的线程数少于corePoolSize，尝试创建新的线程(Worker)执行command(作为Worker的firstTask)，  \* 调用addWorker方法检测runState(运行状态)和workerCount(工作线程数量)，  \*  \*  \* 2. If a task can be successfully queued, then we still need  \* to double-check whether we should have added a thread  \* (because existing ones died since last checking) or that  \* the pool shut down since entry into this method. So we  \* recheck state and if necessary roll back the enqueuing if  \* stopped, or start a new thread if there are none.  \* 2. 如果任务成功加入对列中，还需要双重检测是否已经加入线程或者线程池是否关闭，  \* 我们重检测状态，而且若是有必要需要将刚加入对列的任务回滚出列  \* 3. If we cannot queue task, then we try to add a new  \* thread. If it fails, we know we are shut down or saturated  \* and so reject the task.  \* 若是不能如对列，尝试创建新的线程，若是创建线程也失败了，就表名线程池应该是关闭了，  \* 需要执行拒绝  \*/  int c = ctl.get();  if (workerCountOf(c) < corePoolSize) {  if (addWorker(command, true))  return;  c = ctl.get();  }  if (isRunning(c) && workQueue.offer(command)) {  int recheck = ctl.get();  if (! isRunning(recheck) && remove(command))  reject(command);  else if (workerCountOf(recheck) == 0)  addWorker(null, false);  }  else if (!addWorker(command, false))  reject(command); } |

ThreadPoolExecutor.addWorker

|  |
| --- |
| private boolean addWorker(Runnable firstTask, boolean core) {  retry:  for (;;) {  // ctl是表示正在运行线程数量的原子类，包含了状态和数量  // 二进制的高三位表示状态，剩下的29位表示运行的数量  int c = ctl.get();  // 运行状态  int rs = *runStateOf*(c);   // Check if queue empty only if necessary.  // 对列是空的，状态不是RUNNING  if (rs >= *SHUTDOWN* &&  ! (rs == *SHUTDOWN* &&  firstTask == null &&  ! workQueue.isEmpty()))  return false;  // 主要是自旋通过CAS修改ctl的值  for (;;) {  int wc = *workerCountOf*(c);  if (wc >= *CAPACITY* ||  wc >= (core ? corePoolSize : maximumPoolSize))  return false;  // 这里是cas修改操作，是加了一次，即运行数量类加了一次  if (compareAndIncrementWorkerCount(c))  break retry;  // 再次检测状态是否正确  c = ctl.get(); // Re-read ctl  if (*runStateOf*(c) != rs)  continue retry;  // else CAS failed due to workerCount change; retry inner loop  }  }   boolean workerStarted = false;  boolean workerAdded = false;  Worker w = null;  try {  // 创建工作线程  w = new Worker(firstTask);  final Thread t = w.thread;  if (t != null) {  final ReentrantLock mainLock = this.mainLock;  mainLock.lock();  try {  // Recheck while holding lock.  // Back out on ThreadFactory failure or if  // shut down before lock acquired.  int rs = *runStateOf*(ctl.get());  // 将创建的工作线程加入线程池中workers(是个HashSet)  if (rs < *SHUTDOWN* ||  (rs == *SHUTDOWN* && firstTask == null)) {  if (t.isAlive()) // precheck that t is startable  throw new IllegalThreadStateException();  workers.add(w);  int s = workers.size();  // 更新线程池大小  if (s > largestPoolSize)  largestPoolSize = s;  workerAdded = true;  }  } finally {  mainLock.unlock();  }  if (workerAdded) {  // 若是新工作线程加入成功，需要启动工作线程  t.start();  workerStarted = true;  }  }  } finally {  // 启动失败，这进行失败的处理  if (! workerStarted)  addWorkerFailed(w);  }  return workerStarted; } |

ThreadPoolExecutor.addWorkerFailed

|  |
| --- |
| private void addWorkerFailed(Worker w) {  final ReentrantLock mainLock = this.mainLock;  mainLock.lock();  try {  if (w != null)  // 从线程池中移除  workers.remove(w);  // 并且将工作线程数量减去1  decrementWorkerCount();  // 尝试进行终止  tryTerminate();  } finally {  mainLock.unlock();  } } |

|  |
| --- |
| final void tryTerminate() {  for (;;) {  int c = ctl.get();  // 正在运行，状态是*TIDYING或者状态是SHUTDOWN 但对列不是空的都不进行终止*  if (*isRunning*(c) ||  *runStateAtLeast*(c, *TIDYING*) ||  (*runStateOf*(c) == *SHUTDOWN* && ! workQueue.isEmpty()))  return;  if (*workerCountOf*(c) != 0) { // Eligible to terminate  // 随机终止一个线程  interruptIdleWorkers(*ONLY\_ONE*);  return;  }   final ReentrantLock mainLock = this.mainLock;  mainLock.lock();  try {  // 尝试将ctl修改为运行线程数量是0，运行状态是*TIDYING的值*  if (ctl.compareAndSet(c, *ctlOf*(*TIDYING*, 0))) {  try {  terminated();  } finally {  // 最后将ctl状态修改为*TERMINATED*  ctl.set(*ctlOf*(*TERMINATED*, 0));  // 唤醒wait的线程  termination.signalAll();  }  return;  }  } finally {  mainLock.unlock();  }  // else retry on failed CAS  } } |

Worker线程

worker是继承AQS同步器，实现Runnable的线程，

|  |
| --- |
| private final class Worker  extends AbstractQueuedSynchronizer  implements Runnable {  */\*\*  \* This class will never be serialized, but we provide a  \* serialVersionUID to suppress a javac warning.  \*/* private static final long *serialVersionUID* = 6138294804551838833L;   */\*\* Thread this worker is running in. Null if factory fails. \*/  // 工作线程*  final Thread thread;  */\*\* Initial task to run. Possibly null. \*/*  *// 任务线程，新建Worker时，传入的任务就是赋给这个值，当新建的任务执行完后*  *// worker回去对列中取去任务执行* Runnable firstTask;  */\*\* Per-thread task counter \*/*  *// 每个worker完成的任务数量* volatile long completedTasks;   */\*\*  \* Creates with given first task and thread from ThreadFactory.  \** ***@param*** *firstTask the first task (null if none)  \*/* Worker(Runnable firstTask) {  setState(-1); // inhibit interrupts until runWorker  this.firstTask = firstTask;  // 通过工厂新生成一个worker线程,Worker是Runnable，这里是将Worker  // 转成能启动的Thread  this.thread = getThreadFactory().newThread(this);  }   */\*\* Delegates main run loop to outer runWorker \*/* public void run() {  // 这个是worker线程处理任务的方法  runWorker(this);  }   // Lock methods  //  // The value 0 represents the unlocked state.  // The value 1 represents the locked state.   protected boolean isHeldExclusively() {  return getState() != 0;  }   protected boolean tryAcquire(int unused) {  if (compareAndSetState(0, 1)) {  setExclusiveOwnerThread(Thread.*currentThread*());  return true;  }  return false;  }   protected boolean tryRelease(int unused) {  setExclusiveOwnerThread(null);  setState(0);  return true;  }   public void lock() { acquire(1); }  public boolean tryLock() { return tryAcquire(1); }  public void unlock() { release(1); }  public boolean isLocked() { return isHeldExclusively(); }   void interruptIfStarted() {  Thread t;  if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {  try {  t.interrupt();  } catch (SecurityException ignore) {  }  }  } } |

ThreadPoolExecutor.runWorker

|  |
| --- |
| final void runWorker(Worker w) {  // 当前运行的线程  Thread wt = Thread.*currentThread*();  // 任务线程  Runnable task = w.firstTask;  // 将worker的任务置空  w.firstTask = null;  w.unlock(); // allow interrupts  boolean completedAbruptly = true;  try {  while (task != null || (task = getTask()) != null) {  w.lock();  // If pool is stopping, ensure thread is interrupted;  // if not, ensure thread is not interrupted. This  // requires a recheck in second case to deal with  // shutdownNow race while clearing interrupt  // 如果线程次正在停止，确保线程已经中断  // 如果不是，确保线程没有被中断，  if ((*runStateAtLeast*(ctl.get(), *STOP*) ||  (Thread.*interrupted*() &&  *runStateAtLeast*(ctl.get(), *STOP*))) &&  !wt.isInterrupted())  wt.interrupt();  try {  // 钩子方法，空方法  beforeExecute(wt, task);  Throwable thrown = null;  try {  task.run();  } catch (RuntimeException x) {  thrown = x; throw x;  } catch (Error x) {  thrown = x; throw x;  } catch (Throwable x) {  thrown = x; throw new Error(x);  } finally {  // 钩子方法，空方法  afterExecute(task, thrown);  }  } finally {  task = null;  w.completedTasks++;  w.unlock();  }  }  completedAbruptly = false;  } finally {  // worker线程处理完之后的处理  processWorkerExit(w, completedAbruptly);  } } |

ThreadPoolExecutor.processWorkerExit

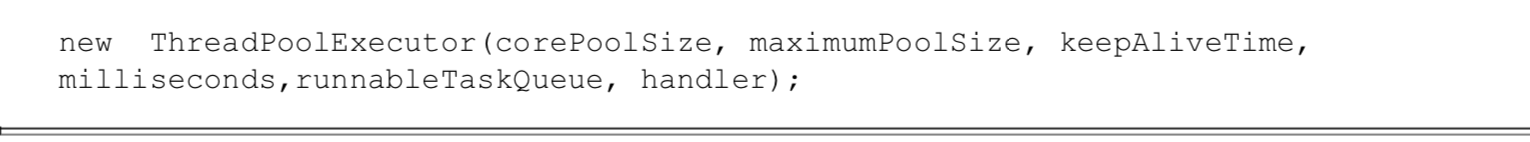
|  |
| --- |
| private void processWorkerExit(Worker w, boolean completedAbruptly) {  if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted  decrementWorkerCount();   final ReentrantLock mainLock = this.mainLock;  mainLock.lock();  try {  completedTaskCount += w.completedTasks;  workers.remove(w);  } finally {  mainLock.unlock();  }   tryTerminate();   int c = ctl.get();  if (*runStateLessThan*(c, *STOP*)) {  if (!completedAbruptly) {  int min = allowCoreThreadTimeOut ? 0 : corePoolSize;  if (min == 0 && ! workQueue.isEmpty())  min = 1;  if (*workerCountOf*(c) >= min)  return; // replacement not needed  }  addWorker(null, false);  } } |

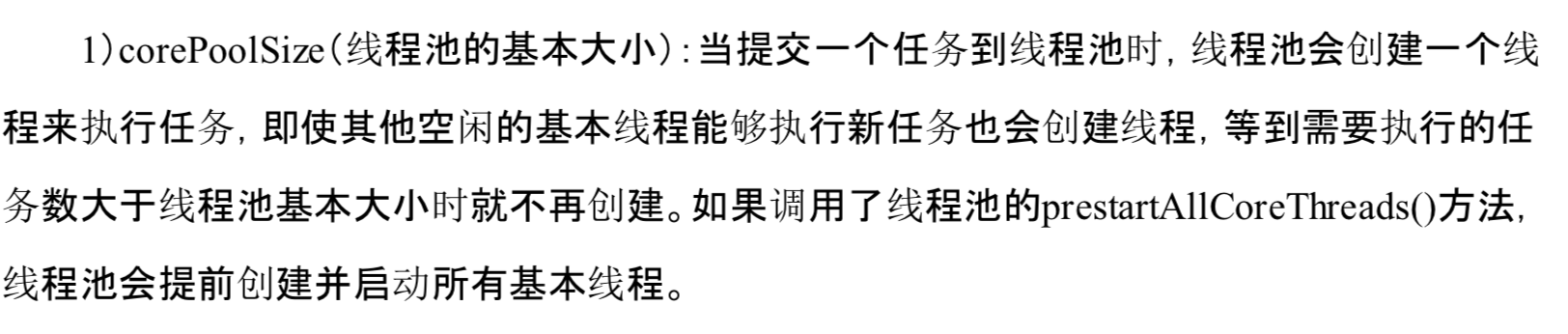
DefaultThreadFactory:默认的线程工厂实现

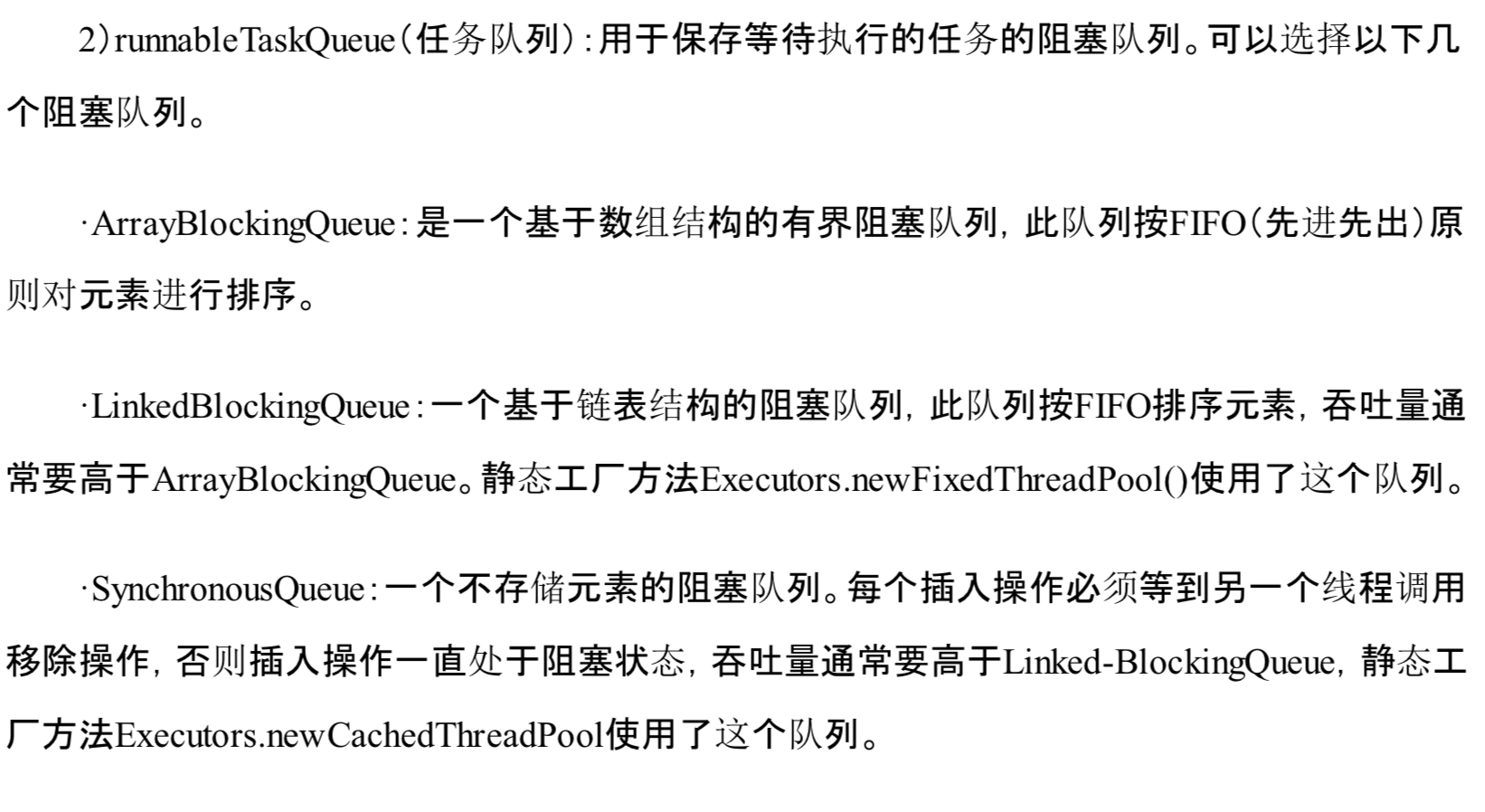
其实就是维护了个线程组

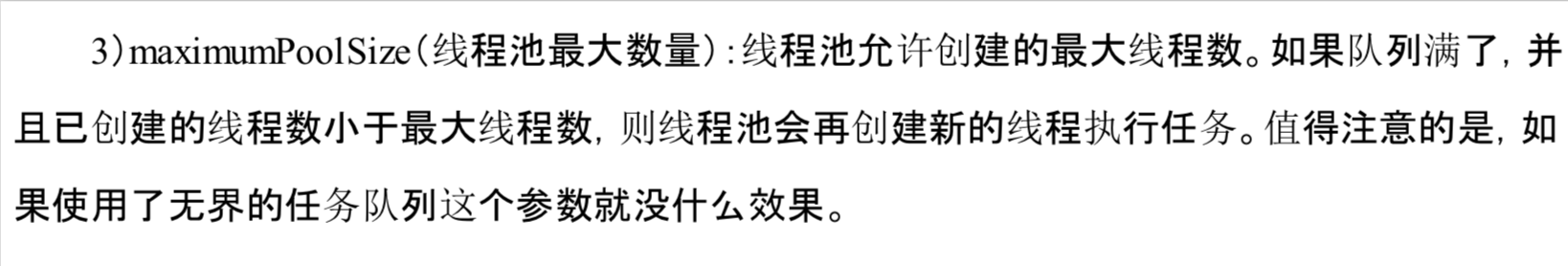
|  |
| --- |
| static class DefaultThreadFactory implements ThreadFactory {  private static final AtomicInteger *poolNumber* = new AtomicInteger(1);  private final ThreadGroup group;  private final AtomicInteger threadNumber = new AtomicInteger(1);  private final String namePrefix;   DefaultThreadFactory() {  SecurityManager s = System.*getSecurityManager*();  group = (s != null) ? s.getThreadGroup() :  Thread.*currentThread*().getThreadGroup();  namePrefix = "pool-" +  *poolNumber*.getAndIncrement() +  "-thread-";  }   public Thread newThread(Runnable r) {  Thread t = new Thread(group, r,  namePrefix + threadNumber.getAndIncrement(),  0);  if (t.isDaemon())  t.setDaemon(false);  if (t.getPriority() != Thread.*NORM\_PRIORITY*)  t.setPriority(Thread.*NORM\_PRIORITY*);  return t;  } } |

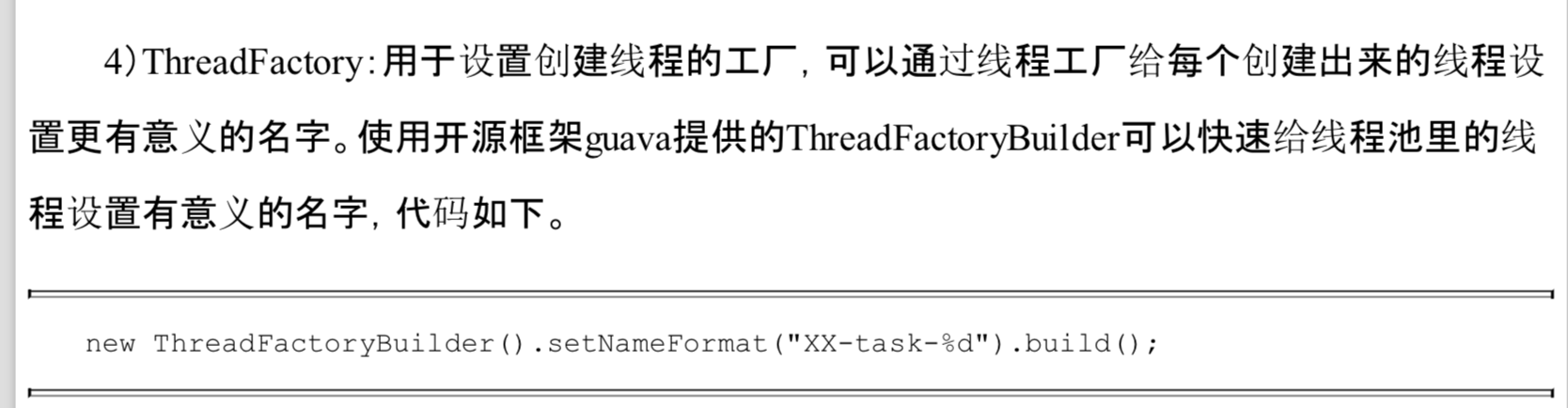
## 9.2 线程池使用

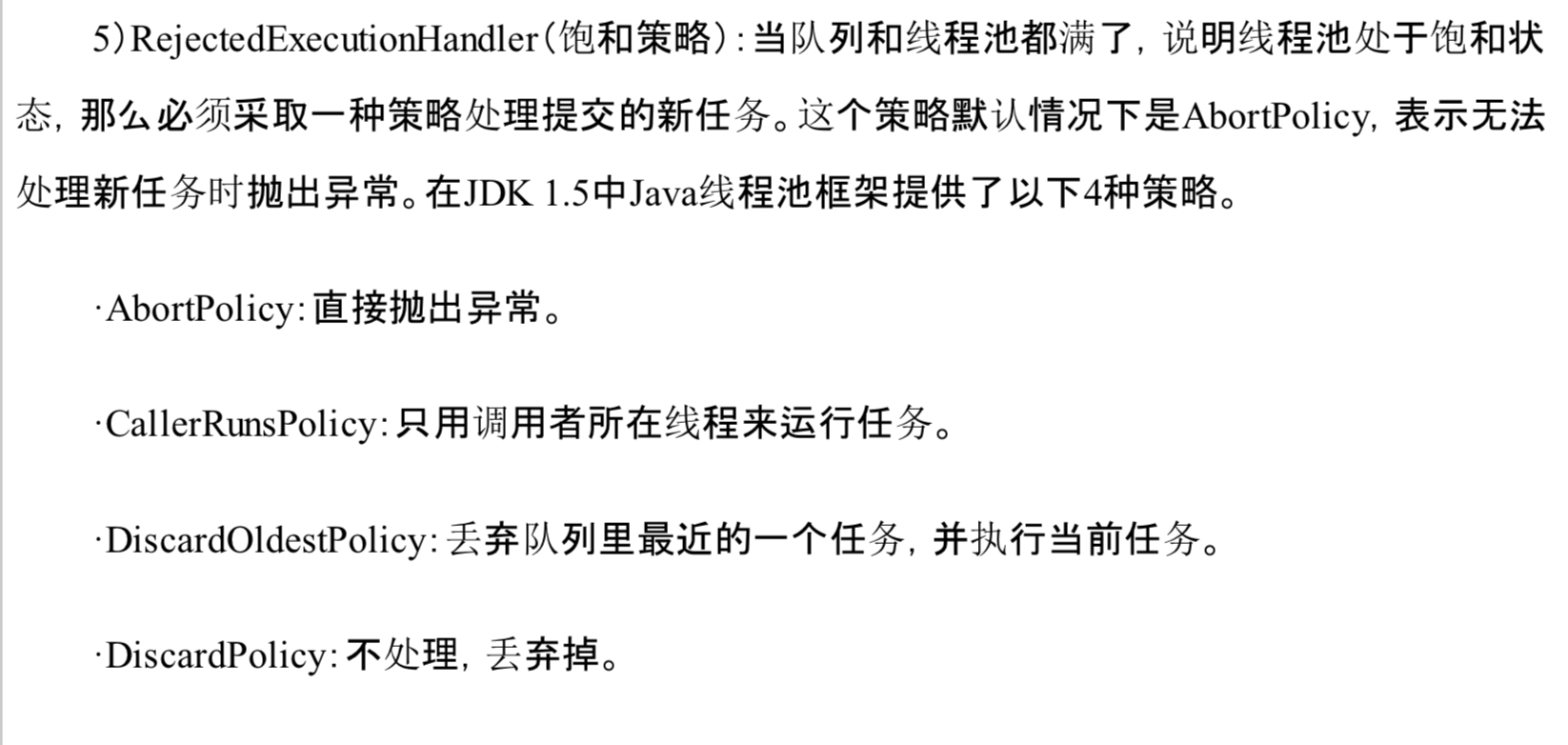


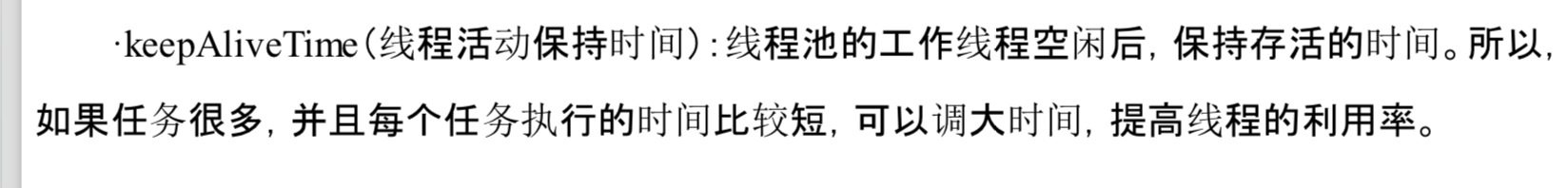


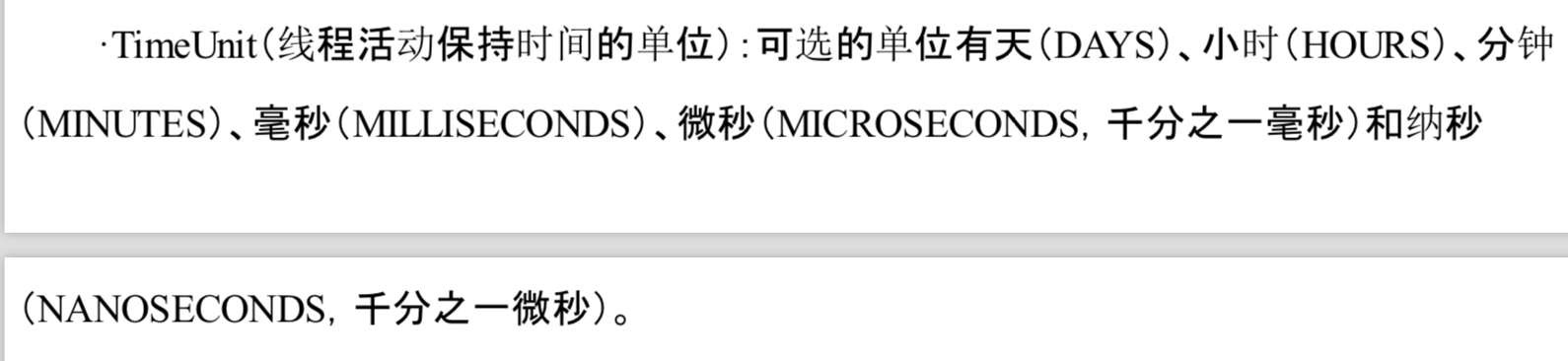


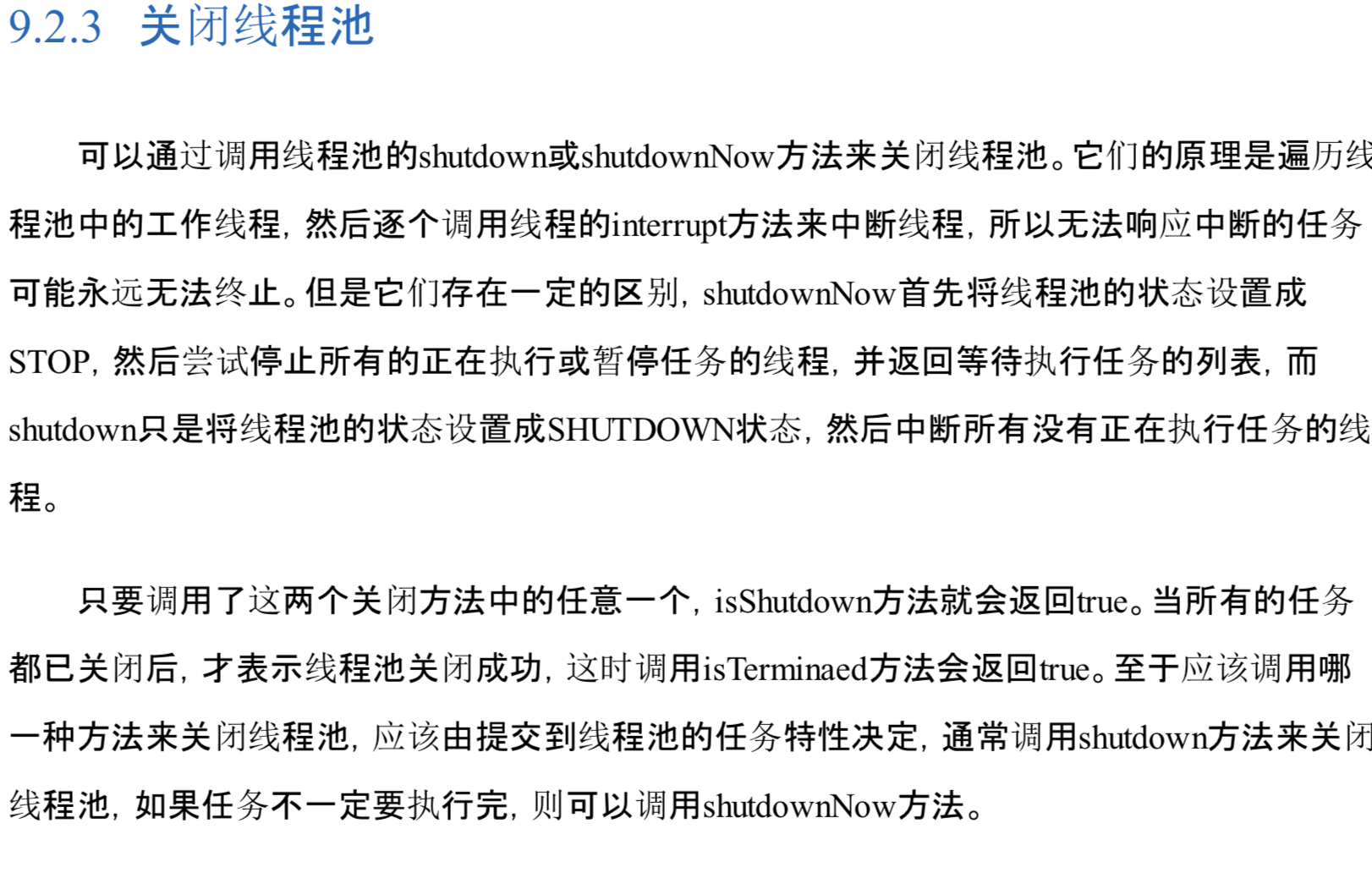










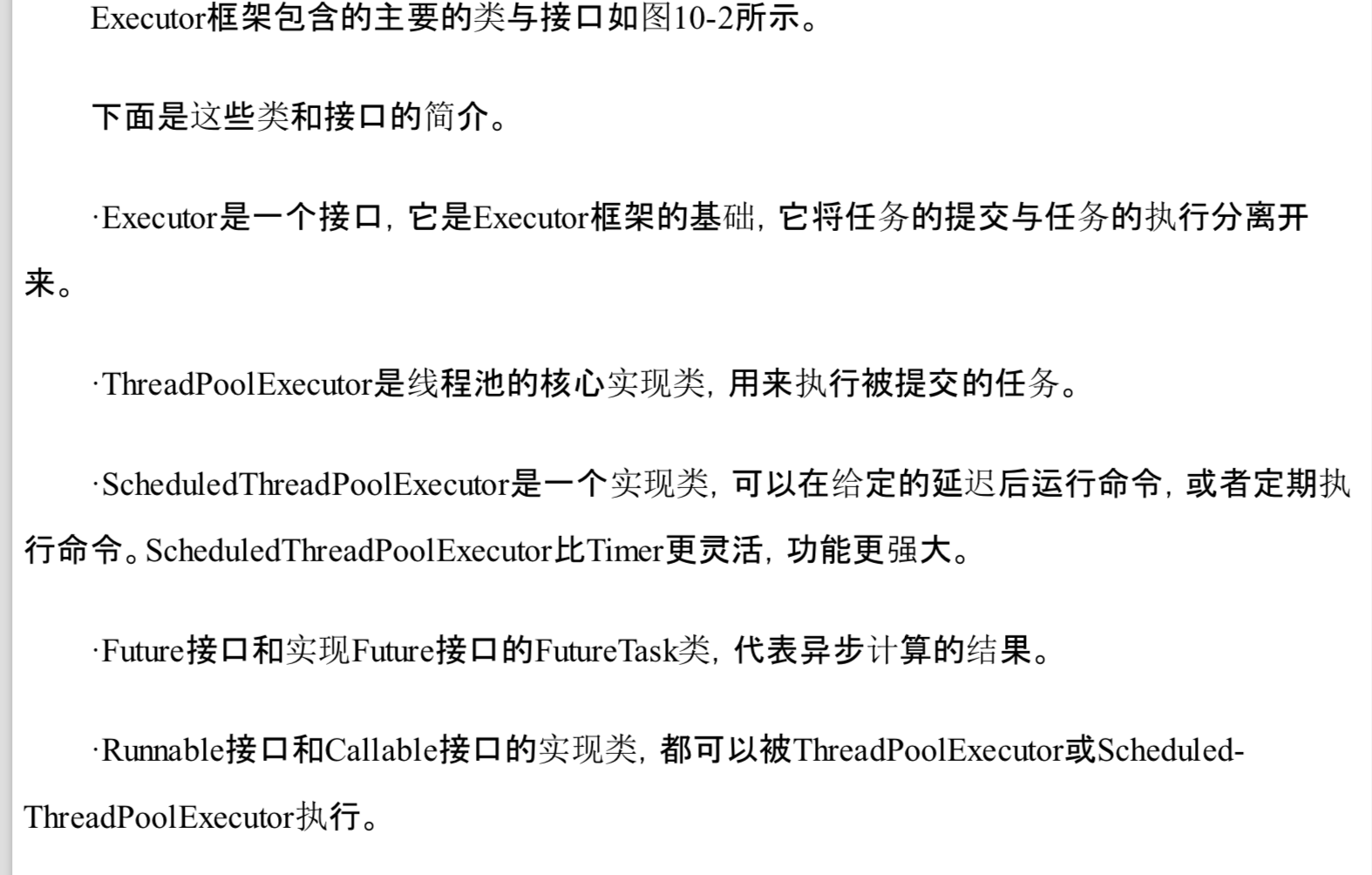


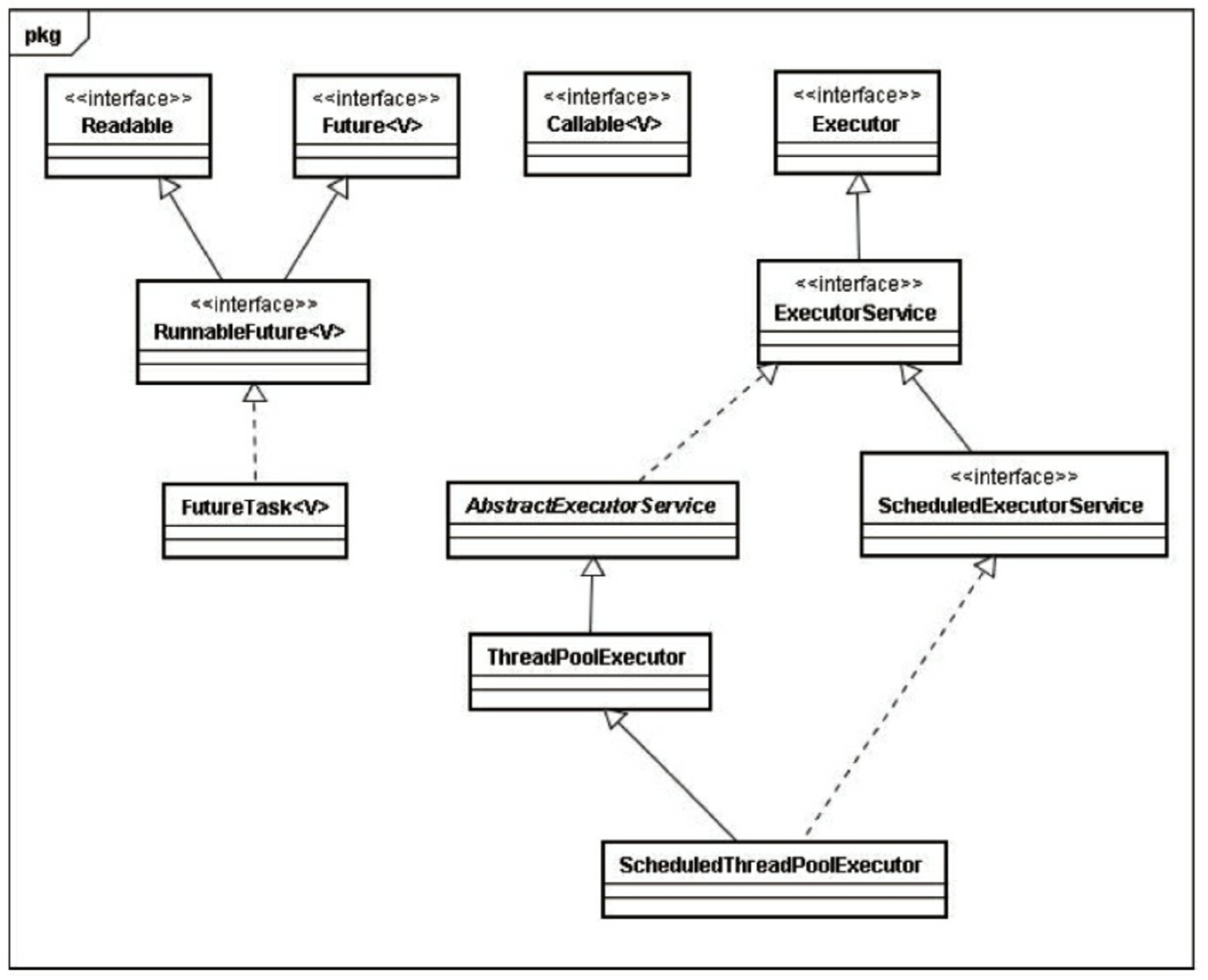
# Executor框架

## 10.1 Executor简介

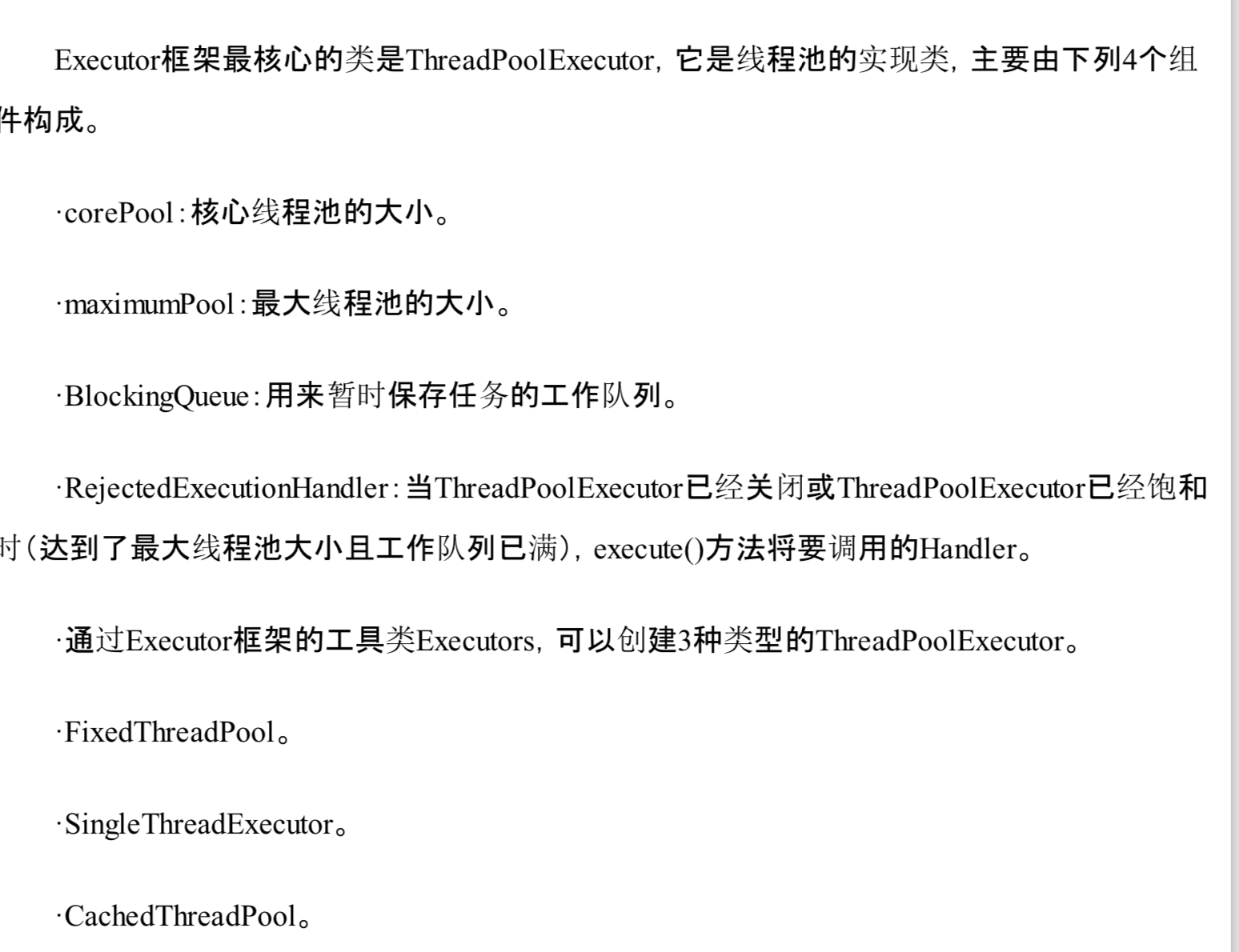
Executor框架主要由3大部分组成：

1. 任务。包括被执行的任务，需要实现的接口：Runnable接口或Callable接口
2. 任务的执行：包括任务执行机制的核心接口Executor，以及继承自Executor的ExecutorService接口。Executor框架有两个关键类实现了ExecutorService接口(ThreadPoolExecutor和ScheduledThreadPoolExecutor)
3. 异步计算结果：包括接口Future和实现Future接口的FutureTask类。



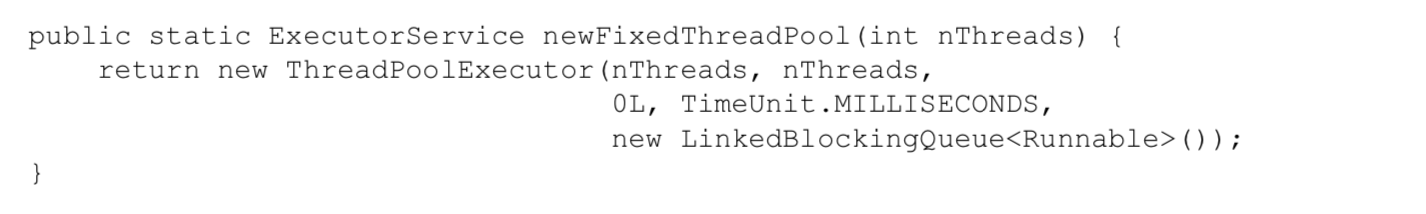


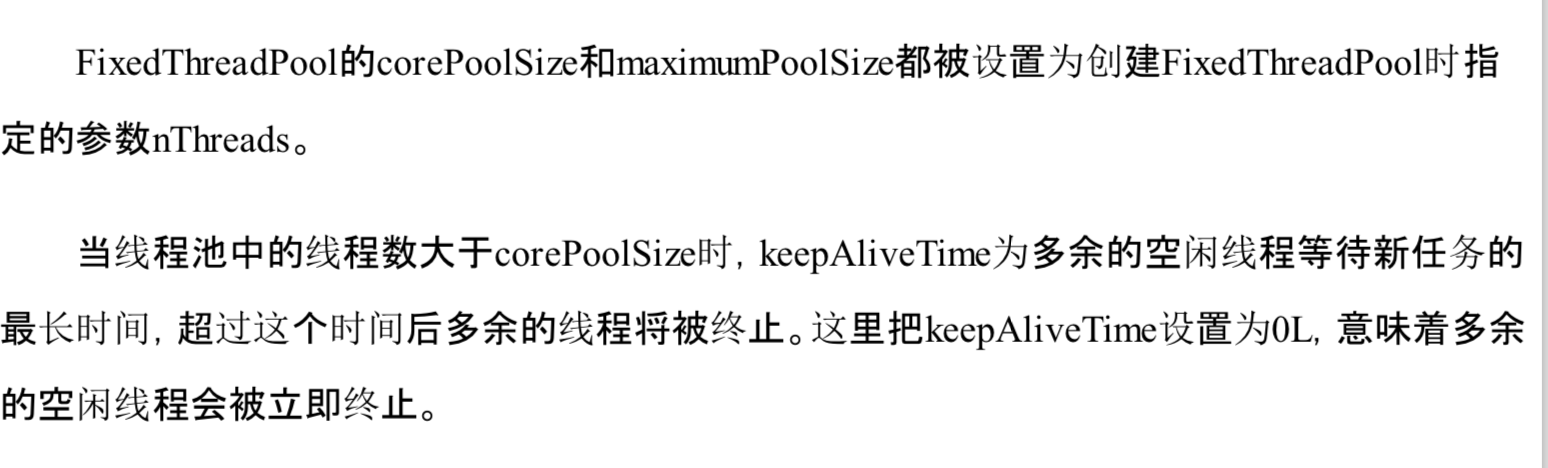
## 10.2 ThreadPoolExecutor详解

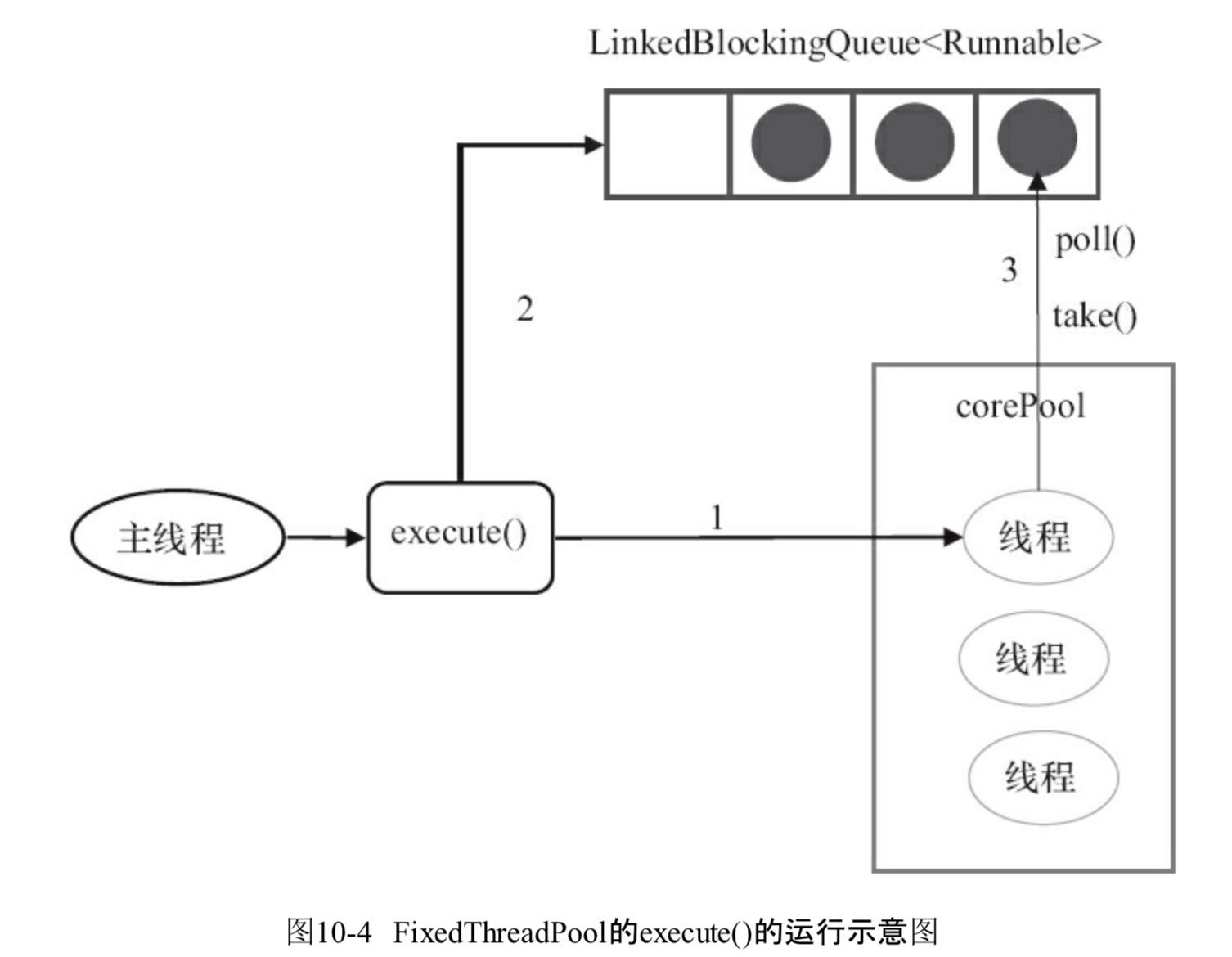


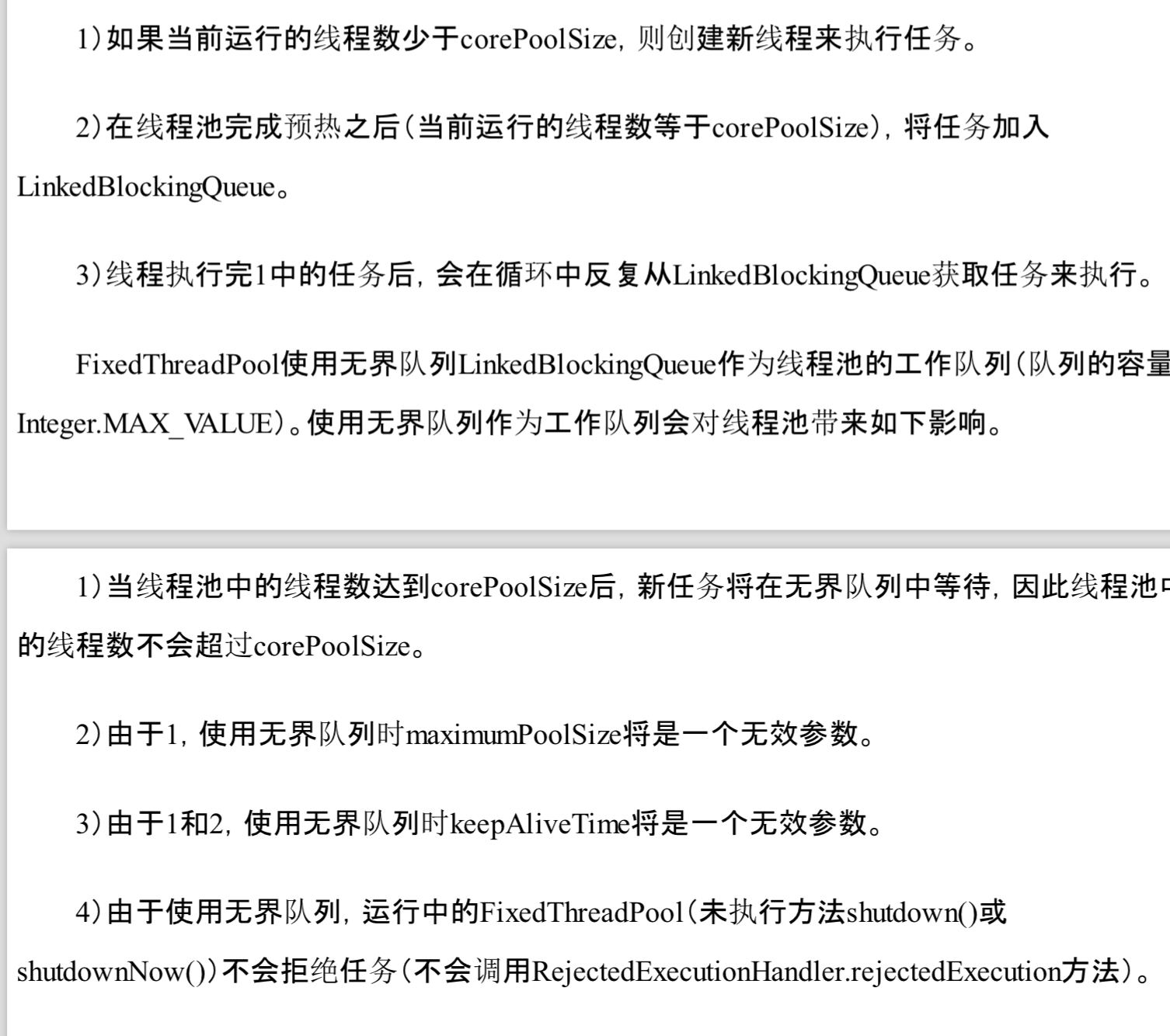
### 10.2.1 FixedThreadPool详解

FixedThreadPool可重用固定线程数的线程池。



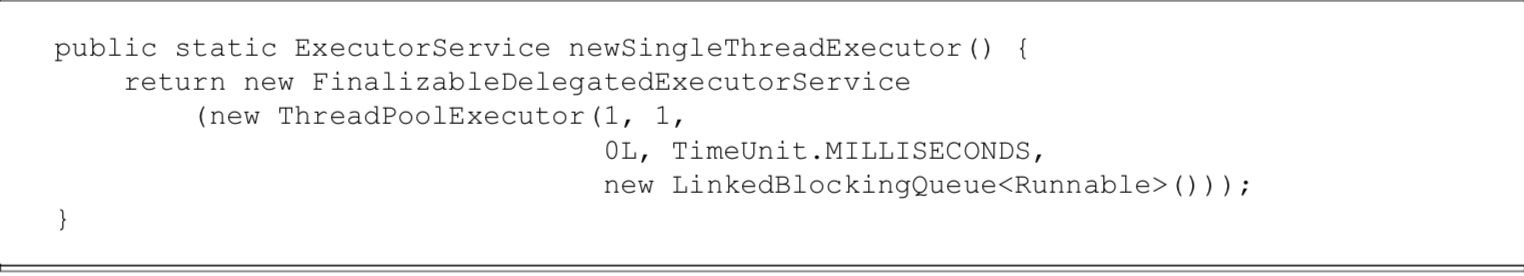


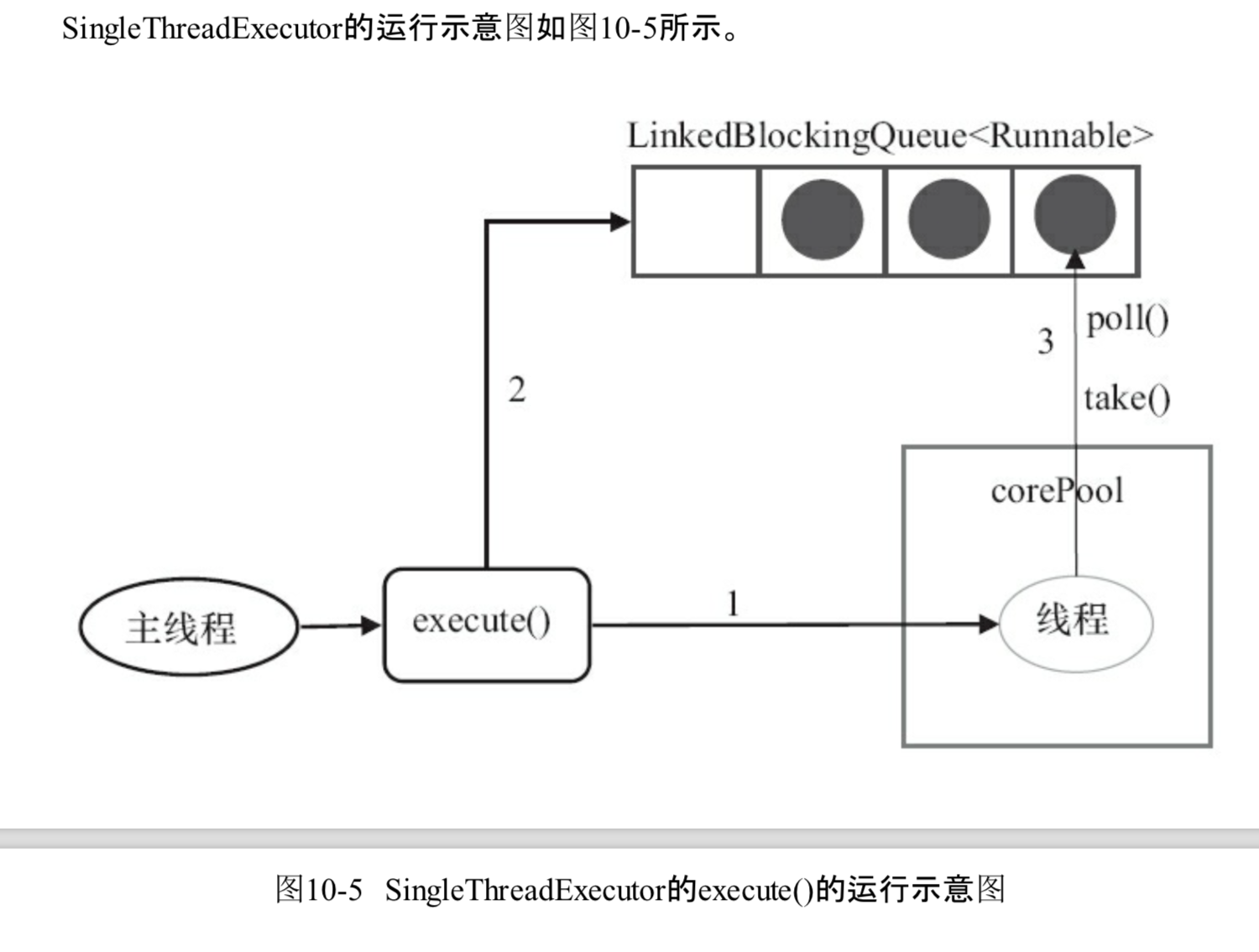


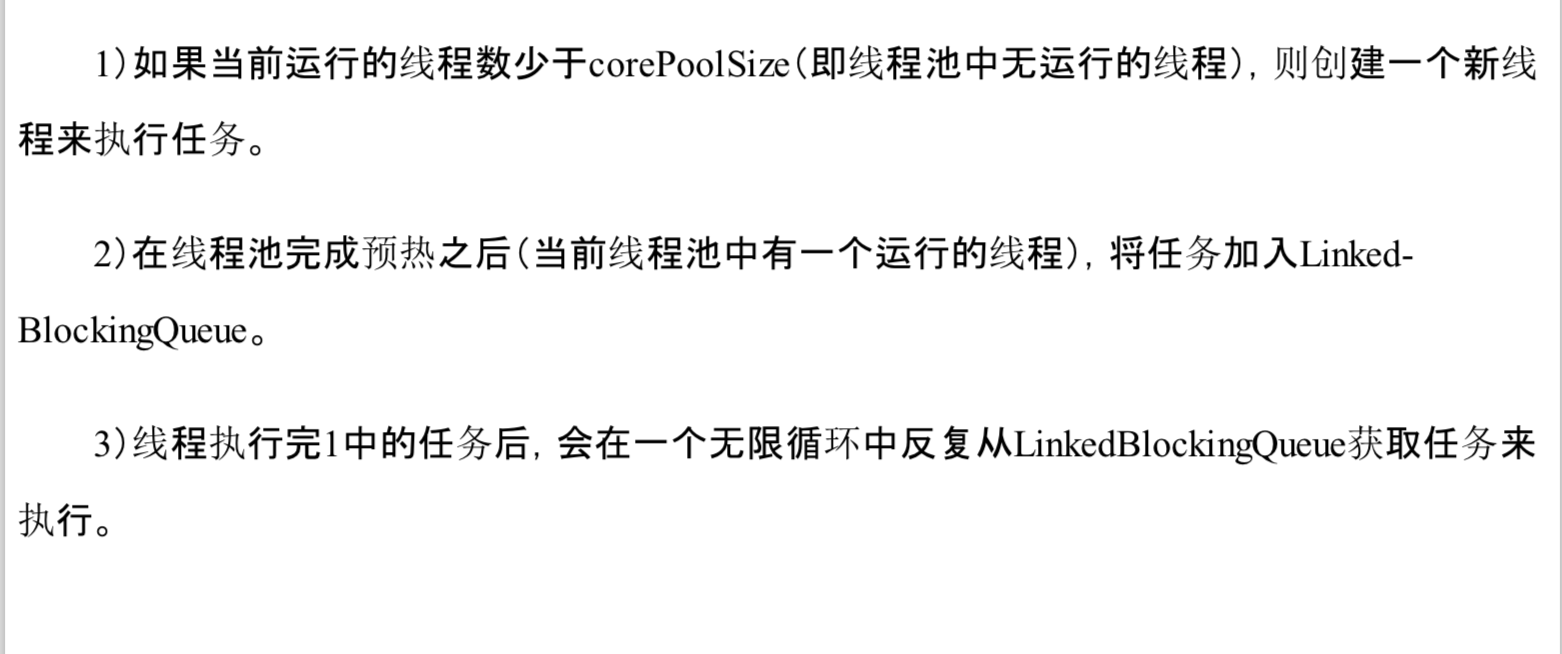


### 10.2.2 SingleThreadPool详解

SingleThreadExecutor是使用单个worker线程的Executor。

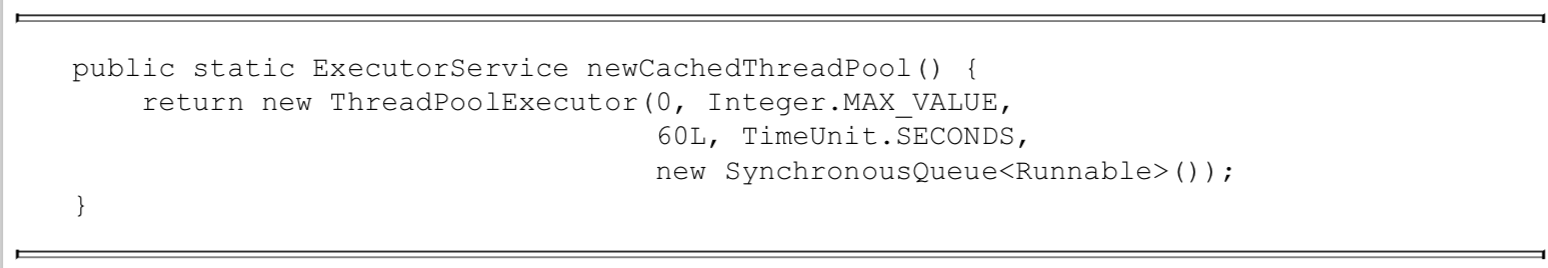


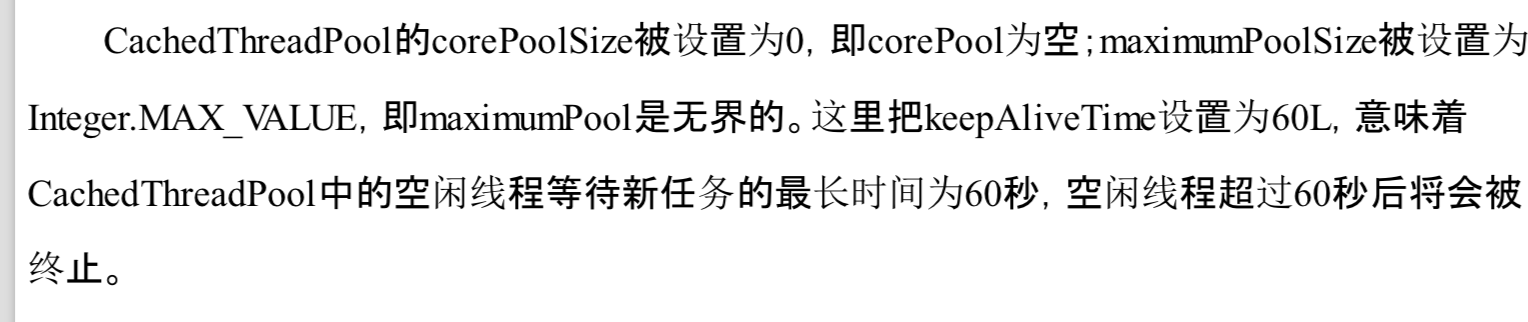


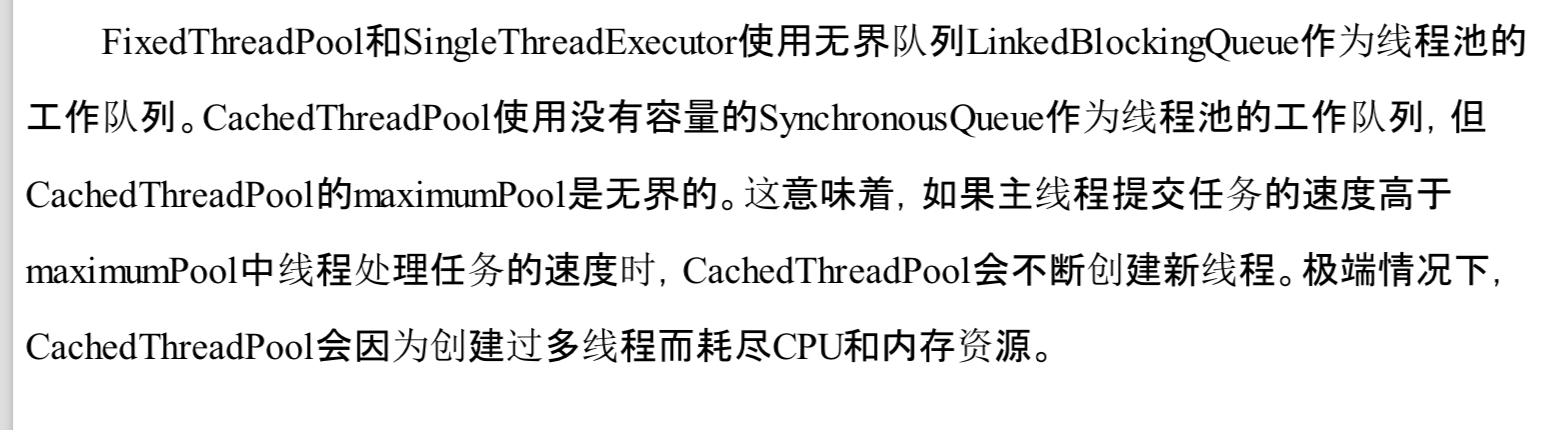


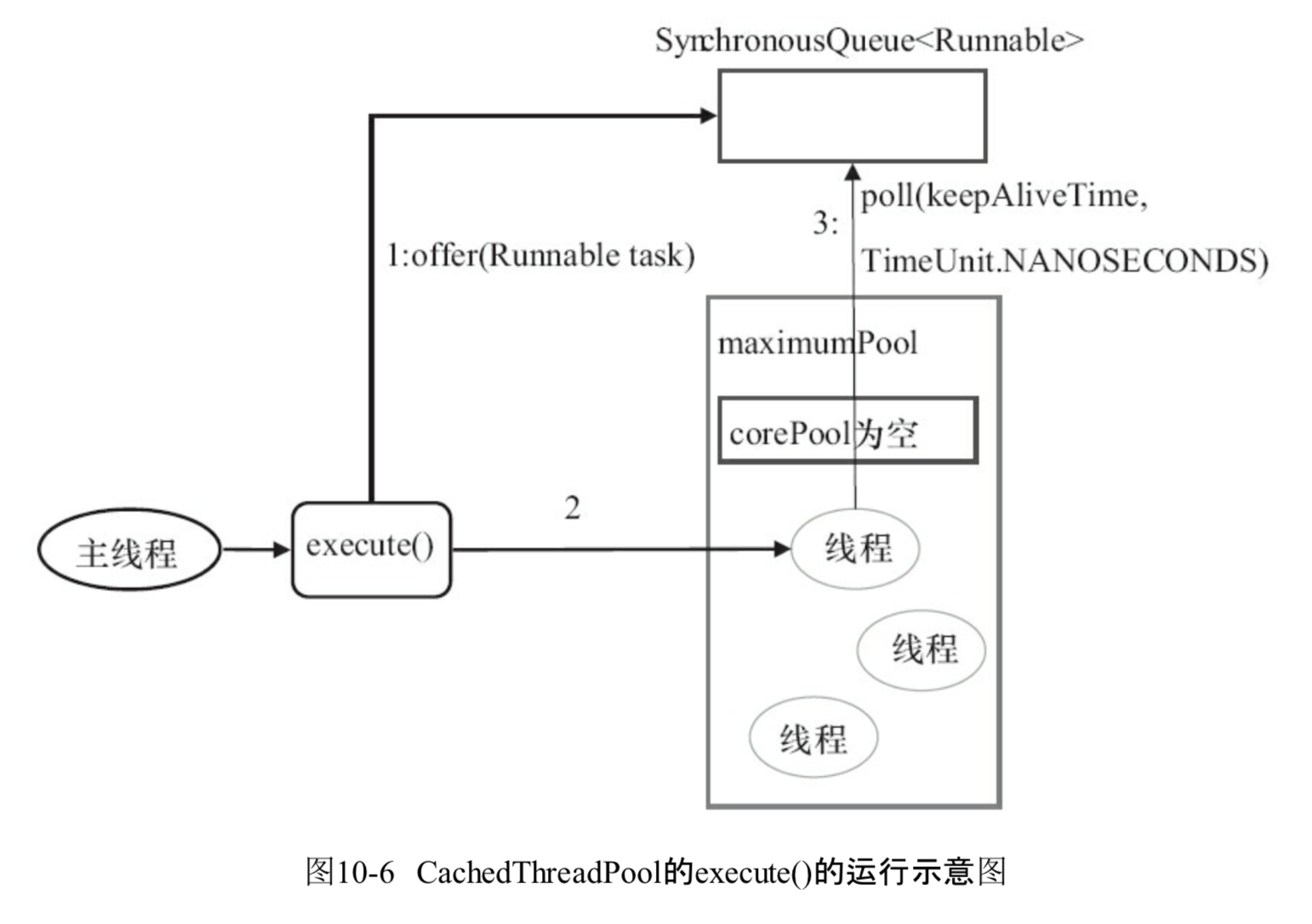
### 10.2.3 CachedThreadPool详解

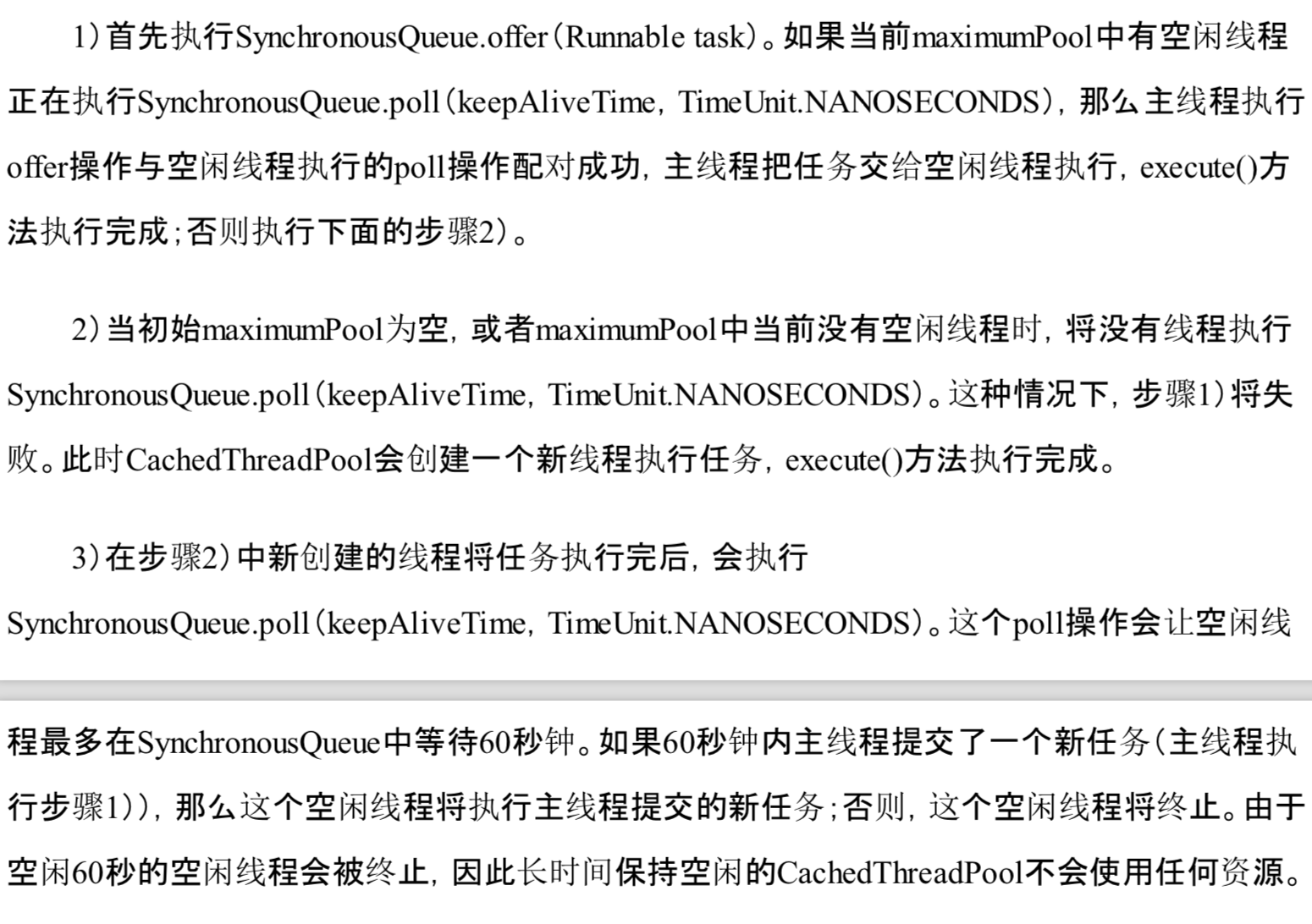
CacheThreeadPool是一个会根据需要创建新线程的线程池

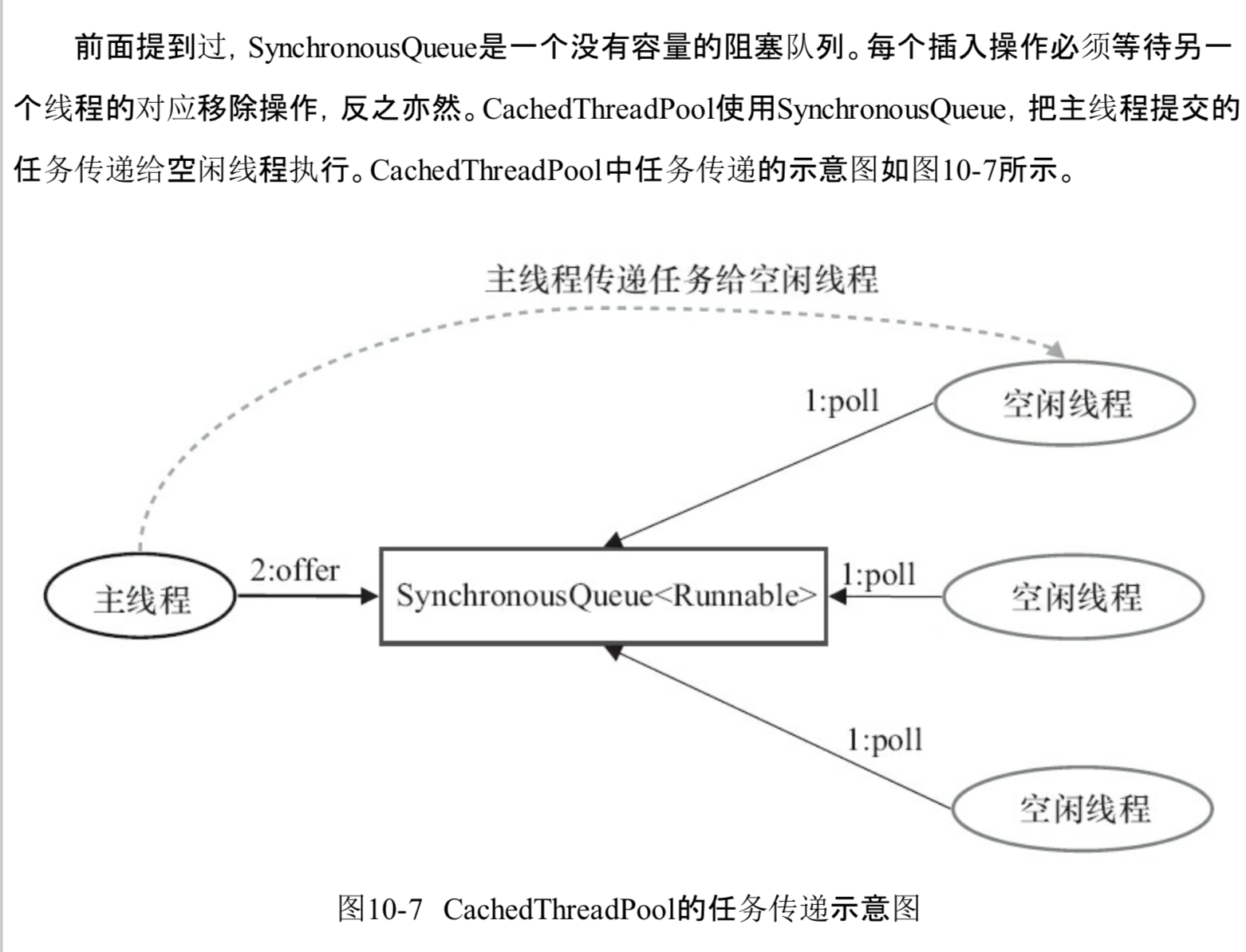




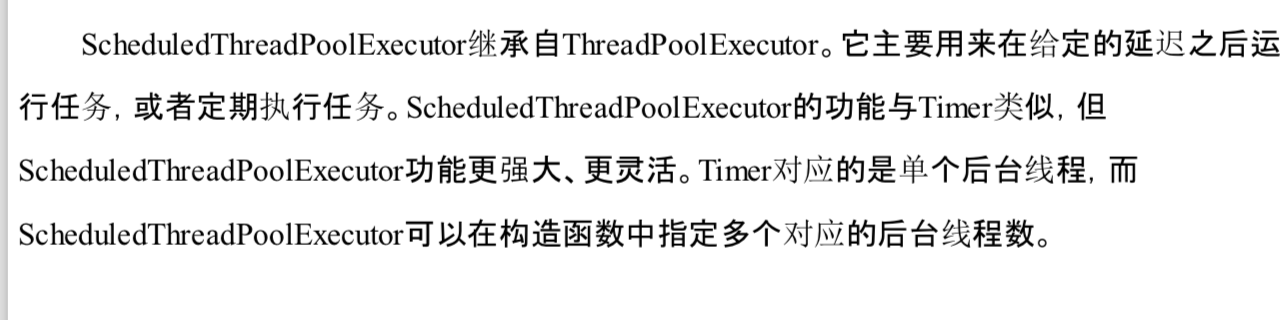








## 10.3 ScheduledThreadPoolExecutor详解



### 10.3.1 ScheduledThreadPoolExecutor的运行机制



