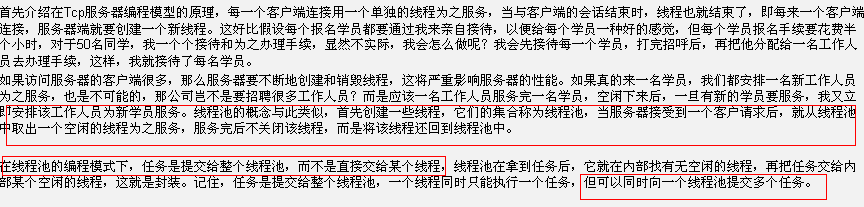
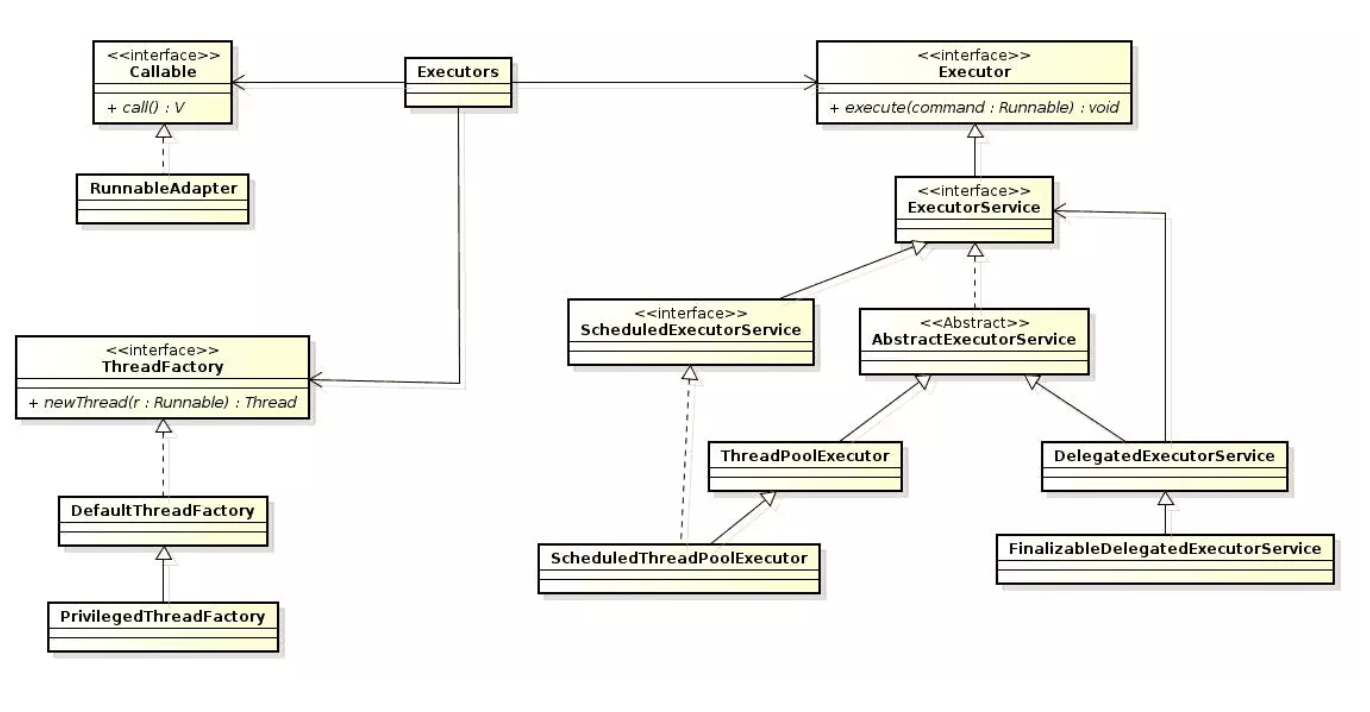
1. 线程池



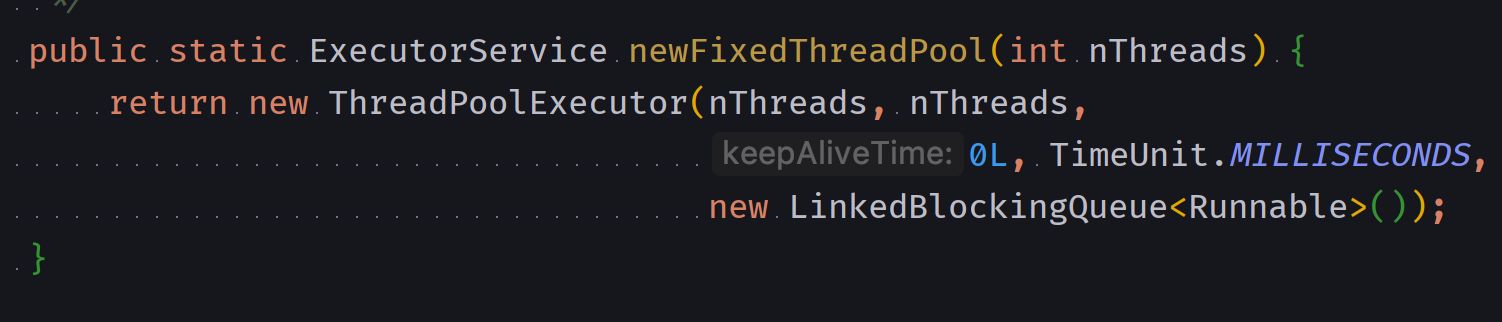
1. 为什么要用线程池
   1. 减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
   2. 可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。
2. 线程池体系，要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池



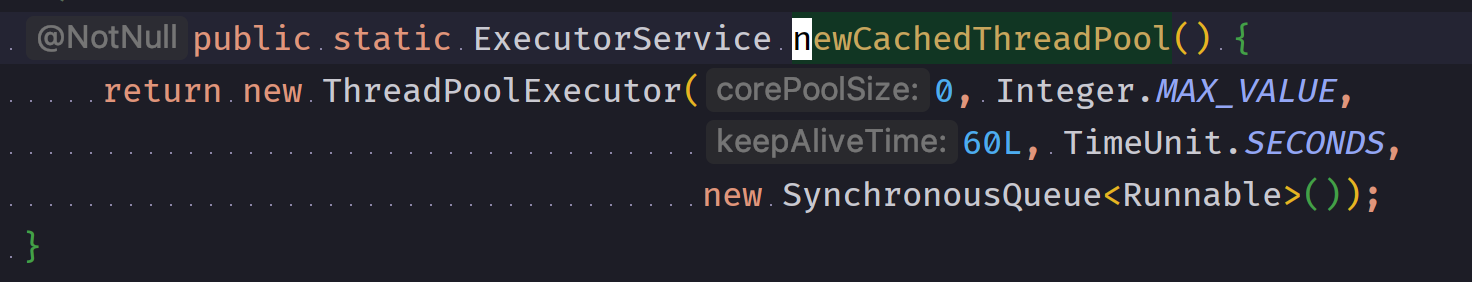
1. newSingleThreadExecutor，创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行



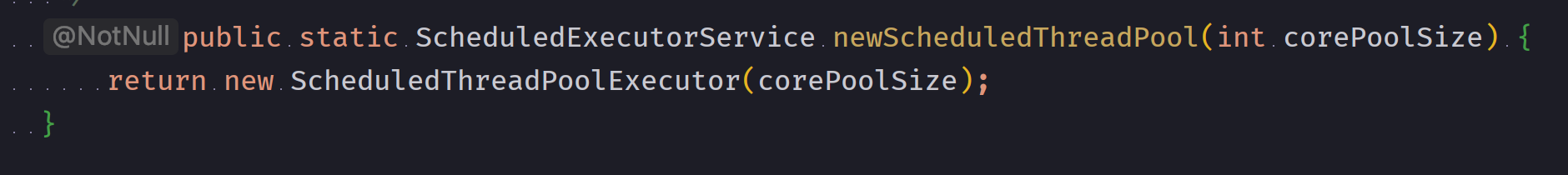
1. newFixedThreadPool，创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

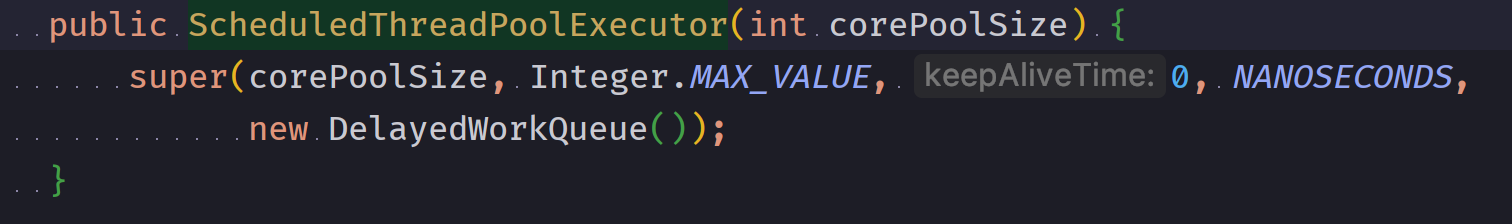


1. newCachedThreadPool，创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以[智能](https://link.jianshu.com/?t=http://lib.csdn.net/base/aiplanning)的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。



1. newScheduledThreadPool，创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求

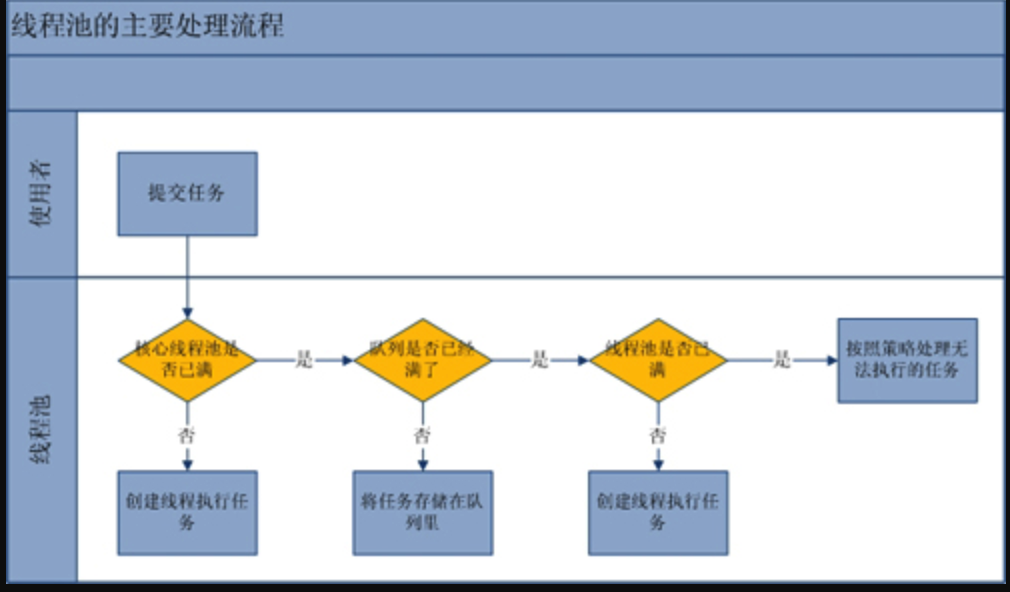




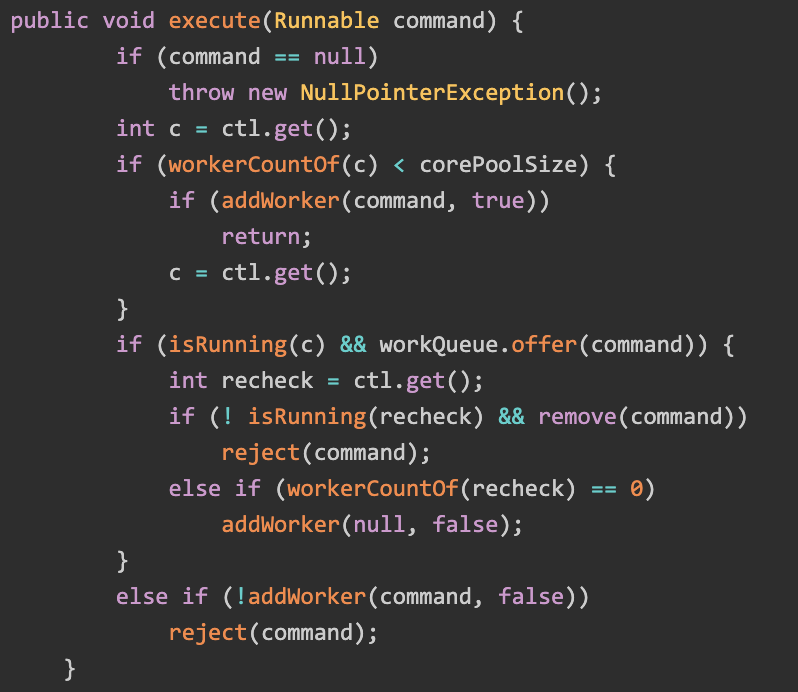
1. ThreadPoolExecutor，底层核心实现
   1. 构造方法参数
      1. **corePoolSize**: 当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使有其他空闲的核心线程能够执行新任务也会创建线程， 等到需要执行的任务数大于线程池核心大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有核心线程
      2. **maximumPoolSize**（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
      3. **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。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率
      4. **TimeUnit**（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)
      5. WorkQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列
         1. ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序
         2. LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue
         3. SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue
         4. PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
      6. **ThreadFactory**：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug和定位问题时非常有帮助
      7. **RejectedExecutionHandler**（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略
         1. AbortPolicy：直接抛出异常。
         2. CallerRunsPolicy：只用调用者所在线程来运行任务。
         3. DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
         4. DiscardPolicy：不处理，丢弃掉。
         5. 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务



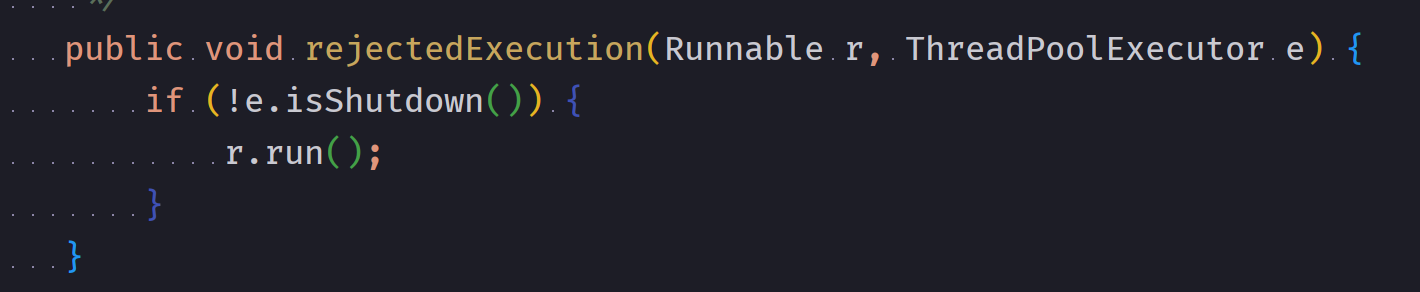
* 1. **execute**()方法，execute方法里面的代码逻辑对应着用户提交一个线程时的执行流程，如下流流程



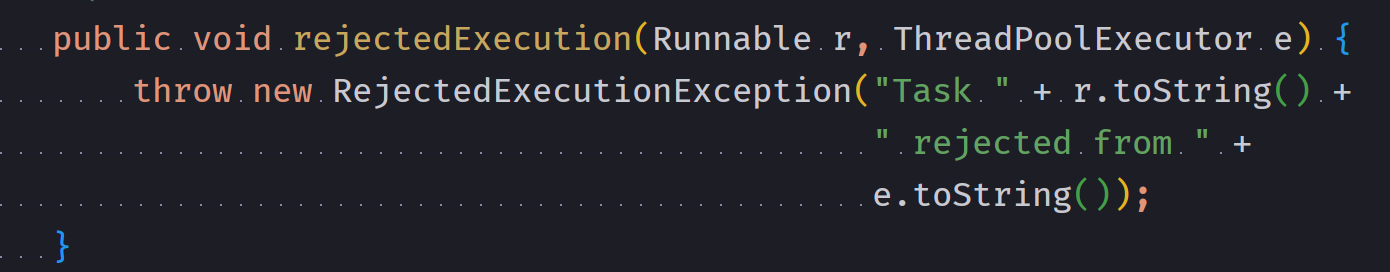
* + 1. 首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
    2. 其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
    3. 最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务
    4. 源码



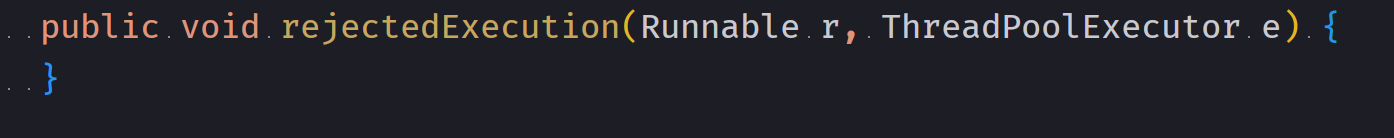
* 1. **BlockingQueue**<Runnable> 参数，线程池的实现，队列起着至关重要的作用。它也是ThreadPoolExecutor构造方法里面的参数之一。从上面的execute执行逻辑知道当核心线程已满，如果等待队列未满，我们就可以把线程任务放到队列中去，当线程池中有空闲线程就会执行该任务。排队的通用策略有以下三种
     1. 直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。直接提交通常要求无界线程数 maximumPoolSizes 以避免拒绝新提交的任务
     2. 无界队列。使用无界队列(例如不具有预定义容量的 LinkedBlockingQueue)将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize，因此，maximumPoolSize的值也就无效了。
     3. 有界队列。当使用有限的 maximumPoolSizes时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。
  2. **keepAliveTime** 参数。 ThreadPoolExecutor中额定的“工人”数量由corePoolSize决定，当任务数量超过额定工人数量时，将任务缓存在BlockingQueue之中，当发现如果连queue中也放不下时（可能是因为使用有界queue，也可能是使用SynchronousQueue），ThreadPoolExecutor会请求“老板”再派几个“工人”过来，接下来发生的事情有两种情况
     1. 任务不再过来了 – keepAliveTime，当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间
     2. 任务仍然继续过来 – RejectedExecutionHandler
  3. **RejectedExecutionHandler**，即拒绝策略，当即使向老板借了工人，但是任务还是继续过来，还是忙不过来，这时整个队伍只好拒绝接受了。JDK默认提供了四种策略
     1. CallerRunsPolicy，调用者负责运行该任务(直接调用run方法)



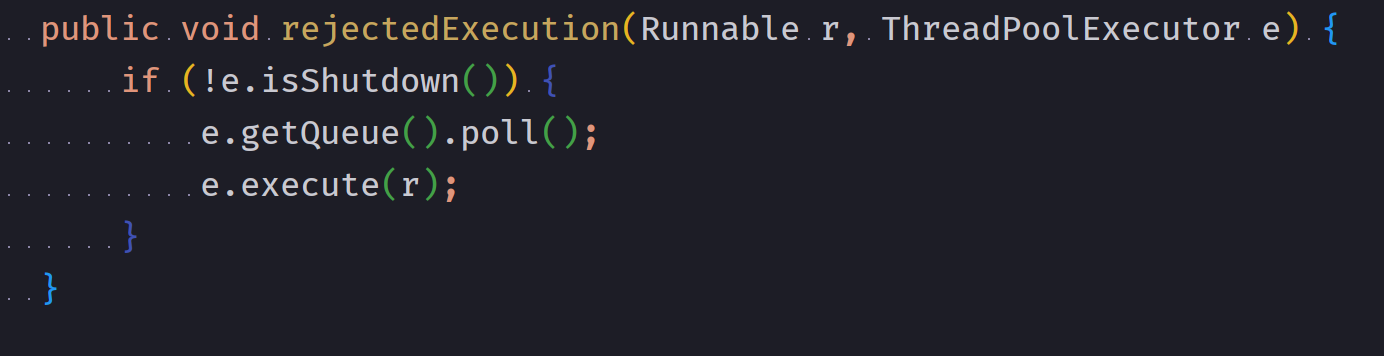
* + 1. AbortPolicy，抛出异常RejectedExecutionException，丢弃任务



* + 1. DiscardPolicy，丢弃任务，不抛异常



* + 1. DiscardOldestPolicy，新来任务踢掉最早的任务



1. **ForkJoinPool**，大任务可以拆成小任务，小任务还可以继续拆成更小的任务，最后把任务的结果汇总合并，得到最终结果，这种模型就是Fork/Join模型
   1. java.util.concurrent.RecursiveTask有返回值
   2. java.util.concurrent.RecursiveAction无返回值

举例，求和

|  |
| --- |
| // 任务  static class SumTask extends RecursiveTask<Long> {  final int THRESHOLD = 100**;** long[] array**;** int start**;** int end**;** SumTask(long[] arr**,** int start**,** int end) {  this.array = arr**;** this.start = start**;** this.end = end**;** }   @Override  protected Long compute() {  if (end - start <= THRESHOLD) {  // 如果任务足够小,直接计算:  long sum = 0**;** for (int i = start**;** i < end**;** i++) {  sum += array[i]**;** }  System.*out*.println(String.*format*("compute %d~%d = %d"**,** start**,** end**,** sum))**;** return sum**;** }  // 任务太大,一分为二:  int middle = (end + start) / 2**;** System.*out*.println(String.*format*("split %d~%d ==> %d~%d, %d~%d"**,** start**,** end**,** start**,** middle**,** middle**,** end))**;** SumTask subTask1 = new SumTask(this.array**,** start**,** middle)**;** SumTask subTask2 = new SumTask(this.array**,** middle**,** end)**;  *invokeAll***(subTask1**,** subTask2)**;** Long subResult1 = subTask1.join()**;** Long subResult2 = subTask2.join()**;** Long result = subResult1 + subResult2**;** System.*out*.println("result = " + subResult1 + " + " + subResult2 + " ==> " + result)**;** return result**;** } }  // 测试  public static void main(String[] args) {  // 创建随机数组成的数组:  long[] array = new long[400]**;** for (int i = 0**;** i < array.length**;** i++) {  array[i] = i**;** }  // fork/join task:  ForkJoinPool fjp = new ForkJoinPool(4)**;** // 最大并发数4  ForkJoinTask<Long> task = new SumTask(array**,** 0**,** array.length)**;** long startTime = System.*currentTimeMillis*()**;** Long result = fjp.invoke(task)**;** long endTime = System.*currentTimeMillis*()**;** System.*out*.println("Fork/join sum: " + result + " in " + (endTime - startTime) + " ms.")**;** } |

1. Java5之后的java.util.concurrent包用例

