# ExecutorService概述

* 1. ExecutorService本质上是一个线程池。意义：减少服务器端的线程的创建和销毁，来提高线程资源的利用率
  2. 线程池刚创建的时候是空的
  3. 每过来一个请求，就会在线程池中创建一个**核心线程**来处理这个请求。核心线程的数量在定义线程池的时候需要指定
  4. 核心线程用完之后不会被销毁而是继续等待下一个请求
  5. 只要核心线程没有达到指定的数量，那么每一个请求都会触发创建一个新的核心线程处理
  6. 如果核心线程被全部占用，那么新来的请求会放到**工作队列**中进行排队等待。工作队列本质上是一个阻塞式队列 （BlockingQueue），定义线程时指定。
  7. 如果工作队列被全部占用，那么新来的请求会交给**临时线程**来处理。临时线程的数量在定义线程池的时候需要指定
  8. 临时线程用完之后会存活一段时间（定义线程池时可指定），如果在这段时间内没有接收到新的任务那么就会被销毁
  9. 工作队列中的任务不会被临时线程执行：尽量缩短临时线程的存活时间，尽量提高核心线程的利用率
  10. 如果临时线程被全部占用，那么新来的请求会交给**拒绝执行处理器**来处理（可以在定义线程池时指定）
  11. 示意图

|  |
| --- |
| 计算机生成了可选文字: c 0 r e t h r e a d 5  temporary  hread 5  work queue 7  RejectedExecutionHand1er |

|  |
| --- |
|  |

# 使用

|  |
| --- |
| */\*\*  \* @param* ***corePoolSize*** *\* 保留在池中的线程数，即使它们是空闲的，除非设置了allowCoreThreadTimeOut  \* @param* ***maximumPoolSize*** *\* 允许在线程池中的线程的最大线程数 包括核心线程和临时线程  \* @param* ***keepAliveTime*** *\* 当临时线程在终止之前等待新任务的最大时间。（临时线程等待时间）  \* @param* ***unit*** *the time unit for the {@code keepAliveTime} argument  \* keepAliveTime 的单位  \* @param* ***workQueue*** *t  \* 在执行任务之前使用队列来保存任务。此队列将只保存由{@code execute}方法提交的{@code Runnable}任务。  \* @param* ***handler*** *\* 当执行被阻塞时使用的处理程序，因为达到了线程边界和队列容量  \* @throws IllegalArgumentException if one of the following holds:<br>  \* {@code corePoolSize < 0}<br>  \* {@code keepAliveTime < 0}<br>  \* {@code maximumPoolSize <= 0}<br>  \* {@code maximumPoolSize < corePoolSize}  \* @throws NullPointerException if {@code workQueue}  \* or {@code handler} is null  \*/*  public static void main(String[] args) {   *ExecutorService* es =  new ThreadPoolExecutor(  5*//核心线程数量5*  , 12*//核心线程数量和临时线程总数量12 临时线程 12-5等于7*  , 20*//临时线程空闲时等待销毁的时间*  , TimeUnit.*SECONDS//临时线程空闲时等待销毁的时间单位秒* , new ArrayBlockingQueue<*Runnable*>(5) *//工作队列5*   , new RejectedExecutionHandler() { *// 拒绝执行处理器*  @Override  public void rejectedExecution(*Runnable* r, ThreadPoolExecutor executor) {  System.*out*.println("拒绝");  }  });  for (int i = 0; i < 20; i++) {  es.execute(new Runnable() {  @Override  public void run() {  System.*out*.println("start");  try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  });  }  // 如果线程池用完，可以关闭线程池 // 实际开发中，线程池开启之后一般是不关的 es.shutdown();  } |

上述案例中会输出12个start，3个拒绝：

首先创建20个线程，5个核心线程开始执行，5个加入工作队列,7个加入临时线程执行。因为所有线程等待3秒，所以在未执行完前，其它3个线程会被拒绝。

待5个核心线程依次执行完成，工作队列中5个的也会依次加入执行，所以先输出12个start和3个拒绝，等会会再输出5个start

遇到的坑：如果没用main方法测试而是用@Test注解，会出问题，这里就出现了后面5个start未输出