# ExecutorService流程概述

* 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（抽象概念）**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. 工作队列中的任务会被临时线程执行，在临时线程执行完后继续取工作队列中的任务。
  10. 工作队列中的任务会被核心线程执行，在核心线程执行完后继续取工作队列中的任务。
  11. 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）。
  12. 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 |

|  |
| --- |
|  |

# ThreadPoolExecutors参数详解

## corePoolSize(核心线程)

保留在池中的线程数，即使它们是空闲的，除非设置了allowCoreThreadTimeOut

## maximumPoolSize（最大线程）

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，该值减去corePoolSize的值就是抽象的临时线程的数量

## keepAliveTime（临时线程存活时间）

当临时线程执行完未接到新任务销毁前等待的时间（未接到新任务等待该时间后将会销毁）

## unit（临时线程存活时间单位）

keepAliveTime熟悉的单位，例：毫秒、秒

## workQueue（工作队列）

在执行任务之前使用队列来保存任务。此队列将只保存由线程池的{@code execute}方法提交的{@code Runnable}任务。

通常有三种选择。

### SynchronousQueue（默认）

\* TODO **直接交付**。一个好的默认的工作队列的选择是直接交付任务给线程而不是保留任务的的{@link SynchronousQueue}队列。  
\* TODO 这里，一个尝试放入队列的任务将会失败如果没有立即可用的线程去执行它，所以一个新的线程将被创建。  
\* TODO 该策略在处理可能具有内部依赖项的请求集时避免锁定，直接交付通常请求无限的{@code maximumPoolSizes}去避免拒绝新提交的任务。  
\* TODO 反过来承认当命令以平均速度高于他们能处理的速度持续到达时，线程存在无限增长的可能性。

总结：任务来了后立即交给队列，队列立即交给线程处理，没有则创建新线程，没有线程将会失败。

### LinkedBlockigQueue

\* TODO **无限队列**。当所有的核心线程正在忙时，如果使用一个无边界的队列（例如一个没有预定于容量的队列：{@link LinkedBlockingQueue}）将会导致新任务在队列中等待。  
\* TODO 因此，不会创建超过corePoolSize数量的线程（此时maximumPoolSize的值无效）。  
\* TODO 当每个任务完全独立于其它任务时这也许是恰当的，所以任务不会影响其他任务的执行;例如，在一个网页服务器。  
\* TODO 这种类型的排队在一个平滑的瞬间爆发的请求方面是有用的，当命令以平均速度高于他们能处理的速度持续到达时，它允许线程存在无限增长的可能性。

总结：当核心线程都在忙时，新来的任务都交给队列，由于该队列无限，所以maximumPoolSize不起作用。也就是说整个线程池永远只有定义的核心线程在执行任务。

*无限队列理论上始终有容量容纳新的任务，当核心线程设置为0时，如果没有新的请求到来，此时只会创建一个线程来依次取出队列的任务执行。如果核心线程不为0，则无需另外创建，直接使用核心线程来执行任务。*

*如果有新的请求到来且此时线程数量小于最大线程数量，那么会创建新线程执行新任务，新任务执行完后同时取队列中的任务。*

### ArrayBlockingQueue

\* TODO **有限队列**：当我们使用一个无穷的maximumPoolSizes事，一个有限队列（例如，{@link ArrayBlockingQueue}）帮助预防资源耗尽，但是可能更难调整和控制。  
\* TODO 队列大小和线程池最大大小可以互相交换：使用大队列和小池子可以最大限度减少CPU的使用，操作系统资源，和上下文切换的开销，但是也能人工导致低吞吐量。  
\* TODO 如果一个任务频繁的阻塞（例如他们是I/O），一个系统可能有能力去安排时间给比你允许的更多的线程。  
\* TODO 使用小队列通常请求大池子，保持CPU更忙，但是也会造成无法接受的调度开销，也会降低吞吐量。

*如果核心线程为0，当使用的有界队列满时，如果没有新的请求到来，此时只会创建一个线程来依次取出队列的任务执行（如果核心线程不为0，则无需另外创建，直接使用核心线程来执行任务）。如果有新的请求到来且此时线程数量小于最大线程数量，那么会创建新线程执行新任务，新任务执行完后同时也会取队列中的任务。*

## threadFactory（线程创建类）

源码翻译：

\* TODO:使用一个{@link ThreadFactory}创建新线程。如果没有指定其他的创建类，一个 {@link Executors#defaultThreadFactory}默认值将被使用，  
\* TODO:通过这种方式创建的线程都在相同的ThreadGroup并且有相同的优先级（NORM\_PRIORITY）和 non-daemon（非守护线程）状态。  
\* TODO:通过提供一个不同的ThreadFactory，你可以修改线程名称，线程组，优先级，守护状态，等等。  
\* TODO:如果一个{@code ThreadFactory}失败的创建了一个线程，此时被要求通过newThread方法返回null，这个执行器将继续，但可能没有能力去执行任何任务。  
\* TODO:线程应该拥有 "modifyThread" {@code RuntimePermission}，如果工作者线程或其他线程没有拥有权限正在在使用池子，服务可能被降级：  
\* TODO:配置改变可能不会造成效果在一个及时的方式，一个关闭的池子可能维持在一个一个终止但未完成的一个状态

通过创建一个类实现{@link ThreadFactory}接口，重写newThread方法即可实现自定义线程创建工厂。

|  |
| --- |
|  |

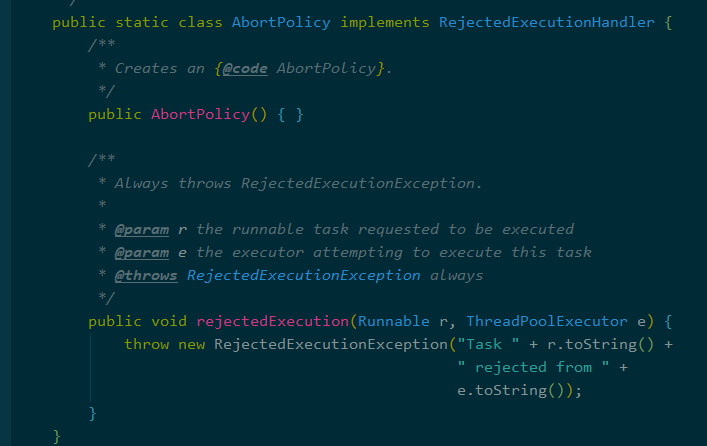
## handler（拒绝执行处理器）

*\* TODO 当执行器已经被关闭，或者当执行器使用有限的最大线程容量和工作队列数量，或者线程*池是饱和时，通过方法{@link #execute(Runnable)}提交的新任务将被拒绝。  
\* TODO 另一方面，{@code execute}方法调用{@link RejectedExecutionHandler#rejectedExecution(Runnable, ThreadPoolExecutor)}方法。  
\* TODO 提供四个预定义的handler策略。

也可以通过实现RejectedExecutionHandler接口自定义处理器。

### ThreadPoolExecutor.AbortPolicy(默认)

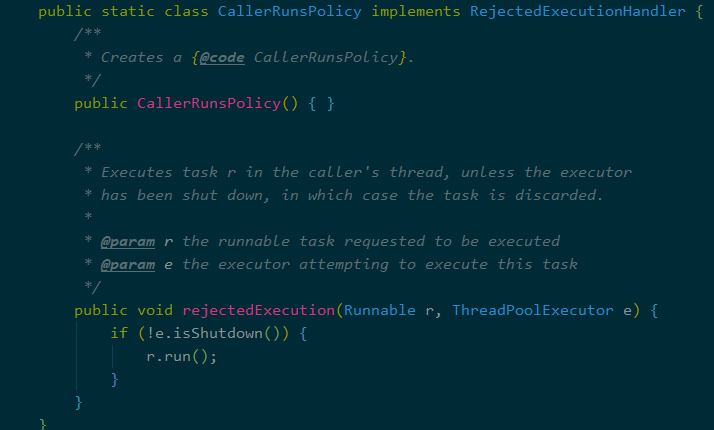
*the handler  
\* throws a runtime {****@link*** *RejectedExecutionException} upon rejection*



* 该处理器在拒绝任务后抛出*RejectedExecutionException异常。*

### ThreadPoolExecutor.CallerRunsPolicy

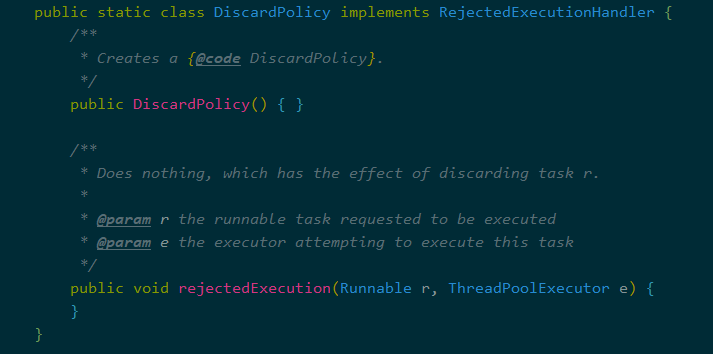
*In {****@link*** *ThreadPoolExecutor.CallerRunsPolicy}, the thread  
\* that invokes {****@code*** *execute} itself runs the task. This provides a  
\* simple feedback control mechanism that will slow down the rate that  
\* new tasks are submitted.*



* 在这个处理器中，这个线程会调用它自己的execute方法运行这个任务。这提供了一个简单的反馈控制机制将会降低新任务被提交的速率。

### ThreadPoolExecutor.DiscardPolicy

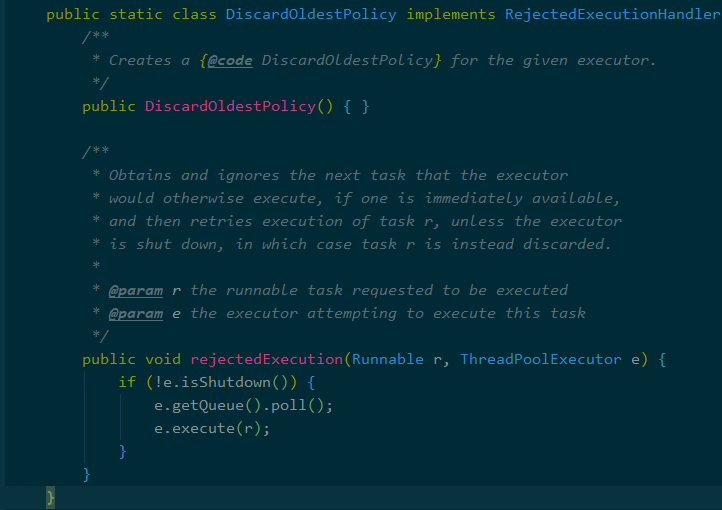
*\* <li>In {****@link*** *ThreadPoolExecutor.DiscardPolicy}, a task that  
\* cannot be executed is simply dropped.  
\**



* 若使用该策略，一个任务不能被执行，只是被简单的丢弃。

### ThreadPoolExecutor.DiscardOldestPolicy

*\* <li>In {****@link*** *ThreadPoolExecutor.DiscardOldestPolicy}, if the  
\* executor is not shut down, the task at the head of the work queue  
\* is dropped, and then execution is retried (which can fail again,  
\* causing this to be repeated.)*



* 如果线程池没有关闭，这个线程池中工作队列的头部任务将被丢弃，然后重新尝试（失败再试，导致重复执行）。

# ThreadPoolExecutors的使用

|  |
| --- |
| */\*\*  \** ***@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 *() {* 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

原因：JUnit的@Test里面，新启动的线程会随着@Test主线程的死亡而死亡，导致没有输出

# 其它封装(Executors工具类)

ThreadPoolExecutor构造方法参数太多，Executors提供了几个常用的创建线程池的方法。

|  |
| --- |
| */\* 特点：  1. 没有核心线程全部都是临时线程  2. 临时线程的数量是Integer.MAX\_VALUE，即2^31-1  考虑到单台服务器所能承载的线程数量远远小于21亿，  所以一般认为这个线程池能够处理无限多的请求  3. 临时线程用完之后最多存活60s  4. 工作队列是一个同步队列，实际生产过程中，  一般在测试阶段就会利用空请求将这个工作队列填充，  此时可以认为这个线程池没有工作队列 \*/ // 大池子小队列 // 适用于高并发的短任务的场景，例如即时通讯 // 不适用于长任务场景* ExecutorService es = Executors.*newCachedThreadPool()*;  */\*特点：  \* 1. 没有临时线程全部都是核心线程  \* 2. 工作队列是LinkedBlockingQueue，默认是Integer.MAX\_VALUE  \* 一般认为能够存储无限多的请求  \*/ // 小池子大队列 // 适用于并发低的长任务场景，例如网盘下载 // 不适用于高并发的短任务场景* ExecutorService es1 =Executors.*newFixedThreadPool(*5*)*; *//计划任务线程池* Executors.*newScheduledThreadPool(*10*)*; *//单个线程的线程池* Executors.*newSingleThreadExecutor()*; *//工作窃取线程池* Executors.*newWorkStealingPool()*; |

# ScheduledExecutorService

### 继承结构

|  |
| --- |
|  |

### 使用

|  |
| --- |
| public static void main*(*String*[]* args*) {* ScheduledExecutorService ses =Executors.*newScheduledThreadPool(*5*)*;  *// 推迟线程的启动时间  // ses.schedule(new ScheduleThread(), 5, TimeUnit.SECONDS);  // 从上一次启动，开始计算下一次的启动时间  // 每隔5s执行一次  // 间隔时间是取执行时间和指定时间的最大值* ses.scheduleAtFixedRate*(*new ScheduleThread*()*, 0, 5, TimeUnit.*SECONDS)*;  *// 从上一次结束，开始计算下一次的启动时间  // 每隔5s执行一次  // ses.scheduleWithFixedDelay(new ScheduleThread(), 0, 5, TimeUnit.SECONDS); }* |

# Java内存结构

Stack：计算/执行代码块。栈内存是线程独享的

Heap：存储对象。堆内存是线程共享的

Method Area：存储类信息。方法区是线程共享的

Native Stack：执行本地方法。用native修饰但是没有方法体的方法称之为本地方法，本地方法的方法体是用其他语言来实现的。本地方法栈是线程独享的

Program Counter:对线程来进行计数。当PC计数器对哪一行计数的时候，任务就会执行哪一行。PC计数器是线程独享的

如果要计算一台服务器的线程承载量，要考虑独享内存的数量：栈内存、本地方法栈、PC计数器。其中，PC计数器只占几个字节大小，所以可以忽略不计；线程执行过程中除非出现本地方法，不然不会占用本地方法栈；栈内存大小是128K~8192K之间

如果需要估计一台服务器的线程承载量，主要考虑栈内存

一般而言，一台服务器最多能允许将2/3的内存给栈内存使用

市面上主流的服务器内存是在64G~128G

以128G为准，考虑极端情况，栈内存以128K来计算

128G/128K\*2/3≈699050

实际开发过程中，一台服务器的线程数量大概是25W左右