# 概述

## 简介

本文档主要记录学习JDK1.8中有关线程、线程池的学习笔记。

# 线程相关

## 创建线程方式

### 实现Runnable接口

具体实现参见Demo项目中的ThreadDemo，其调用方式如下：



图- 1实现Runnable接口创建线程

### 继承Thread类

具体实现参见Demo项目中的ThreadDemo，其调用方式如下：



图- 2继承Thread类创建线程

### 实现Callable接口

参考ThreadDemo例子中的代码。如果想要获取其他线程的执行结果，使用这种方式是最好的，但是如果主线程要获取子线程的结果，主线程会被get方法阻塞，因此会影响主线程的性能。需要掌握Future、RunnableFuture、FutureTask等相关类。

### 线程池获取

这一种方式需要理清楚线程池相关的类，线程执行逻辑。详见线程池章节。

## 相关类

### Runnable

一个接口，只包含一个方法run，所有实现了该接口的类都必须实现run方法来定义自己的 业务逻辑。

重点关注FunctionalInterface（函数式接口）注解，如果一个Interface被FunctionalInterface注解标注，表明该接口中只能有一个抽象方法。如果有多个抽象方法（可以有多个方法，但只能有一个抽象方法），编译器直接检测到错误。如果接口中存在Default标注的默认实现方法，该方法不算作抽象方法，因此不会报错；或者接口中有static方法，因为static方法不算做抽象方法，也不会报错。

接口定义如下，接口只包含一个方法，并且无返回值：

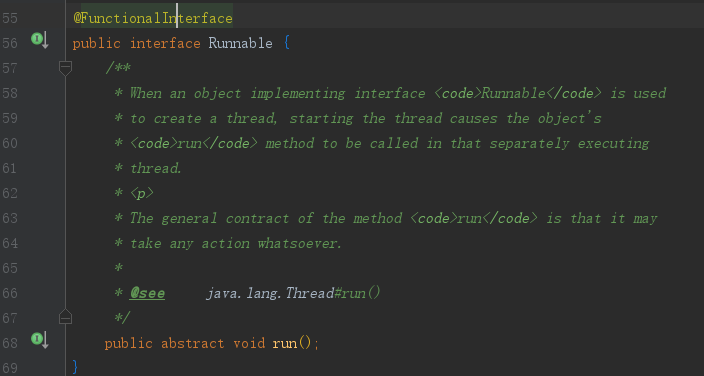


图- 3 Runnable接口定义

### Thread

#### 属性列表

表格 1列举了部分Thread类中属性，包括对象属性和类变量。

表格 1 Thread重要属性

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 类型 | 类变量/对象属性 | 描述 |
| name | String | 对象属性 | 指定线程的名字，该属性被volatile修饰 |
| priority | int | 对象变量 | 描述线程的优先级，取值为[1,10]，CPU调度时会用到，优先级越高，获得时间片的概率越大。默认值为5 |
| daemon | boolean | 对象变量 | 标记线程是否为守护线程 |
| tid | long | 对象变量 | 用于表示线程的ID，每一个线程都有一个唯一的ID，其值通过threadSeqNumber赋值所得 |
| threadInitNumber | int | 类变量 | 如果在创建Thread对象时没有传递name属性，则通过该变量设置name；如果传递了name属性，该变量值不会改变 |
| threadSeqNumber | long | 类变量 | 每创建一个Thread对象将自增1，并且只有在创建对象时该变量才会自增，其他任何时候都不会自增，因此threadSeqNumber的值表明当前虚拟机已经创建过的线程数（并不是存活的线程数） |
| threadStatus | int | 对象变量 | java线程的状态，详见1.3。初始为0，并且该变量被volatile修饰 |

### Callable接口

该接口和Runnable接口类似，都只有一个方法，但是该方法包含一个返回值，并且可以抛出异常。其定义如下：

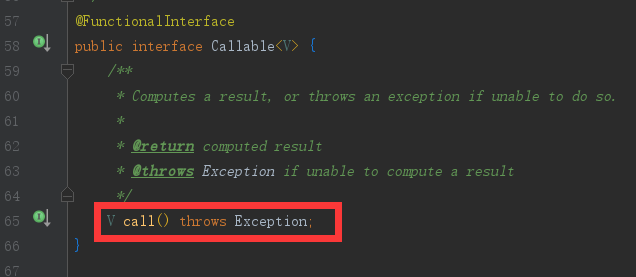


图- 4 Callable接口定义

### Future接口

这是一种异步线程模式，主线程可以通过get接口获取子线程的运行接口，也可以设置异步等待时间。

这一种方式适用于一个任务只需要被执行一次时，采用这种方式，只需要第一个想要执行该任务的线程执行即可，其他线程直接获取任务被执行的结果。

### RunnableFuture接口

实现了Runnable和Future接口的接口。

### FutureTask类

FutureTask类是一个泛型，可以提供cancel功能来停止线程的执行，理解该类需要理清楚线程执行状态。类的继承关系：

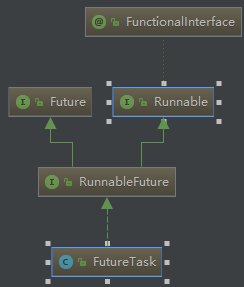


图- 5 FutureTask类继承关系

#### 构造方法

FutureTask有两种构造方式，一种是接收Runnable接口的子类对象，一种是接收Callable接口的子类对象。两种方式的初始状态都为NEW，其实现代码分别如图- 6和图- 7所示：

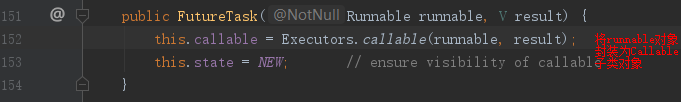


图- 6 接收Runnable子类对象的构造方式

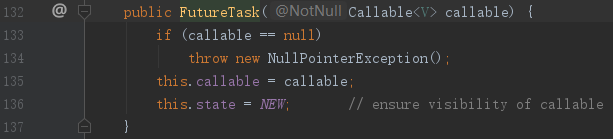


图- 7 接收Callable子类对象的构造方式

通过两种方式可以知道，在FutureTask类中最终存放的是一个Callable子类对象。

#### 状态转换

FutureTask类对象具有多种运行状态，分别如下所示：

|  |  |
| --- | --- |
| 状态 | 描述 |
| NEW |  |
| COMPLETING |  |
| NORMAL |  |
| EXCEPTIONAL |  |
| CANCELLED |  |
| INTERRUPTING |  |
| INTERRUPTED |  |

状态转换存在如下几种情景：

* new—completing—normal
* new—completing—exceptional
* new—canceled
* new—interrupting—interrupted

## java线程状态

参考：<https://blog.csdn.net/pange1991/article/details/53860651/>

参考：Thread类定义中的State枚举类

### 状态列表

Java线程一共有6个状态，分别如下表所示。这6中状态只针对Java线程，而不是操作系统线程的状态。各个状态之间的转换详见1.3.2。

|  |  |
| --- | --- |
| 状态 | 描述 |
| New | 当一个线程被创建当还未调用start方法时，此时线程处于New状态 |
| Runnable | 此状态包括两种情况，一种是Ready，一种是Running。当线程的start方法被调用后，线程将位于可运行线程池中等待被调度，获取CPU，此时处于Ready状态；当线程获取到CPU时，线程处于Running状态 |
| Blocked | 线程如果未获取到锁将被阻塞，此时线程将处于Blocked状态 |
| Waiting | 线程在执行过程中如果调用了Object.wait（）；Thread.join（）；LockSupport.park（）方法时，线程将进入Waiting状态。 |
| Timed\_Waiting | 线程在执行过程中如果调用了Thread.sleep()；Object.wait(long)；Thread.join（long）;LockSupport. parkNanos();LockSupport.parkUntil()方法时，线程将进入Timed\_Waiting状态 |
| Terminated | 线程完成执行之后将进入种终止状态 |

### 状态机

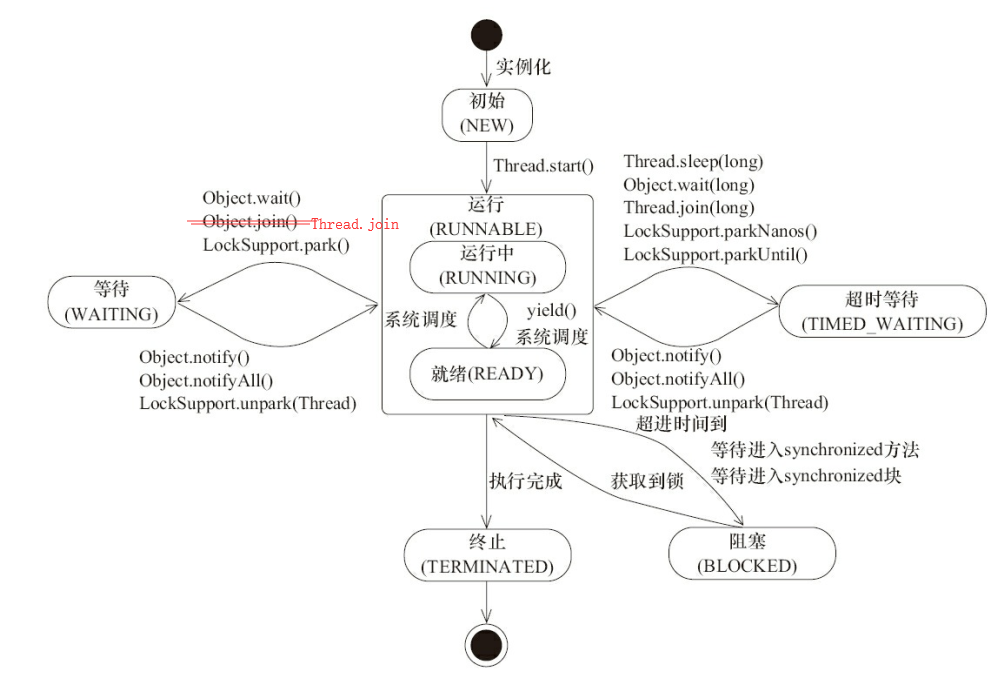


图- 8 Java线程状态机

# 线程池

学习线程池主要掌握各种线程池的实现以及工具类Executors。

## 工具类Executors

工具类Executors提供各种线程池的静态创建入口（其作用类似于Collections类），Executors提供的创建的线程池包括如几类：

|  |  |
| --- | --- |
| 线程池种类 | 描述 |
| FixedThreadPool | 线程数固定的线程池，提供LinkedBlockingQueue缓存任务 |
| CachedThreadPool | 线程数不限的线程池，提供SynchronousQueue缓存任务 |
| WorkStealingPool | JDK1.8之后引入的线程池 |
| SingleThreadExecutor | 只包含一个线程的线程池，提供LinkedBlockingQueue缓存任务 |
| ScheduledThreadPool | 支持定时或者周期性执行的线程池，其中无最大线程数限制 |
| SingleThreadScheduledExecutor | 该线程池其实就是coreSize=1的ScheduledThreadPool |

通过对比各个线程池的创建代码可以发现，FixedThreadPool、SingleThreadExecutor、CachedThreadPool、ScheduledThreadPool、SingleThreadScheduledExecutor几个线程池的实质是ThreadPoolExecutor对象；WorkStealingPool的实质是ForkJoinPool。

因此要重点理解ThreadPoolExecutor、ForkJoinPool 。在理解各个线程池之前，需要熟悉线程池对应的任务缓存队列。

### ThreadPoolExecutor

参考：<https://www.jianshu.com/p/d2729853c4da>

#### 池中元素

线程池从数据结构方面来说也可算作一个集合，因此其中必定存在集合的元素。线程池中的元素便是“线程”，但是此处的线程并不直接是Thread类的对象，而是一个内部类Worker，其中Worker类包含一个Thread对象，以及一个Runnable的任务task。Worker类实现了AbstractQueueSynchronizer（AQS，有关这部分内容要单独整理学习笔记）。

了解线程池最主要的核心就是当线程池接收到一个任务task时，线程是如何执行的。也就是当调用execute方法时的具体过程。

* 重要变量
* ctl：原子变量，该变量高三位用于存放线程池的状态，低29位保存当前线程池的工作线程数。

#### 线程池运行状态

参考Demo项目中的ClearAllCommets类。

* Running：既可以接收处理新的任务，又可以处理队列中的任务；
* Shutdown：不能处理新的任务，但是能处理队列中的任务（和Tcp四次挥手类似，线程池可以不再接受处理请求，但可以继续执行已有的请求）；
* Stop：既不能处理新任务，也不能处理队列中的任务
* Tidying：最后的整理阶段，也就是线程池中所有任务都执行完成，队列中已无其他任务
* Terminated：

#### 线程池处理任务过程

所有创建新线程时，新线程第一次执行的都是新添加的任务task。这种情况包括

* 当线程数小于corePoolSize，创建核心线程并执行新的任务task（即使其他线程处于空闲状态），这个过程通俗理解就是热身准备阶段；
* 当队列满，线程数小于maximumPoolSize时，创建新线程并执行新的任务task；

线程池中的所有线程都封装为了Worker类，并且会给该线程存放第一个执行任务task，如果任务执行完成后，后重复的去从workQueue中获取任务，详见Worker类的run方法，核心代码如下：

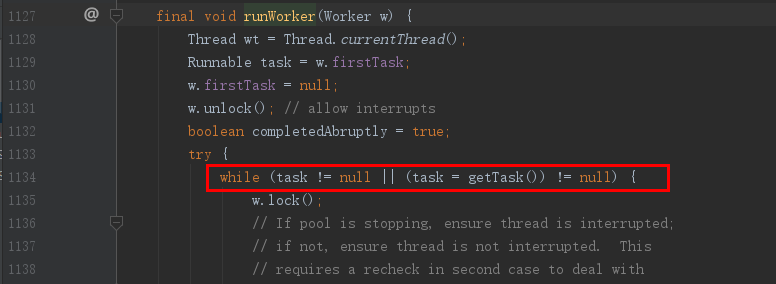


图- 9 worker执行任务过程

### ScheduledThreadPoolExecutor

参考<https://www.jianshu.com/p/4b8a257f1b90>

定时任务框架，可以设置任务定时执行或者周期执行。该类继承关系如下：

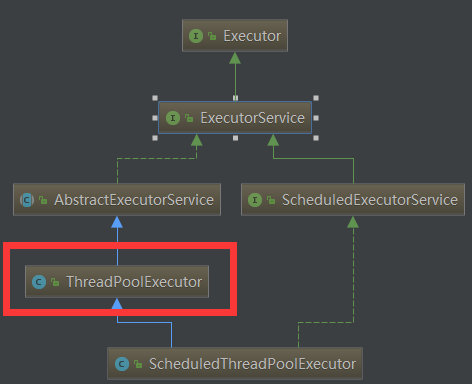


图- 10 ScheduledThreadPoolExecutor继承关系

通过继承关系可以看出ScheduledThreadPoolExecutor类继承了ThreadPoolExecutor类，因此ScheduledThreadPoolExecutor也算是一种线程池，只不过线程池中的线程可以定时执行或者周期执行。

ScheduledThreadPoolExecutor最大线程数无限制。如下图所示：

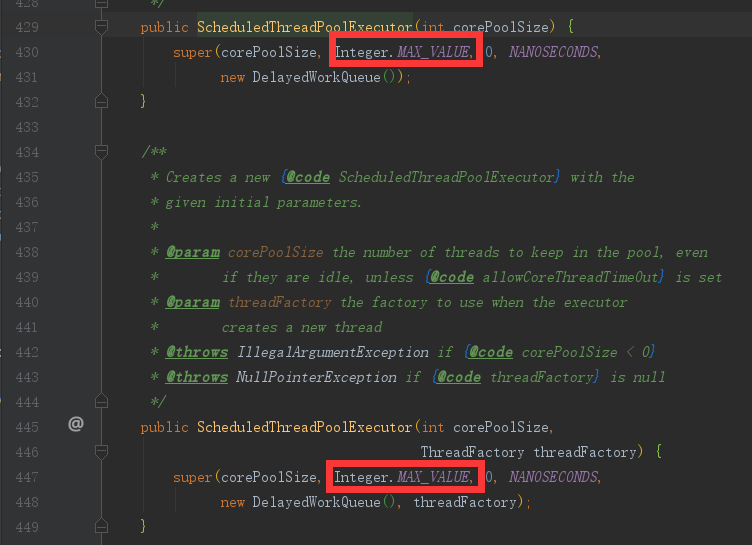


图- 11 ScheduledThreadPoolExecutor初始化

定时任务的核心就在其任务缓存队列DelayedWorkQueue，该队列是通过最小堆来实现的。

入口方法包括execute和submit，这两个入口都要调用schedule方法进行定时调度。schedule的核心思想就是创建一个对象并存放在DelayedWorkQueue队列，对象存放过程如下

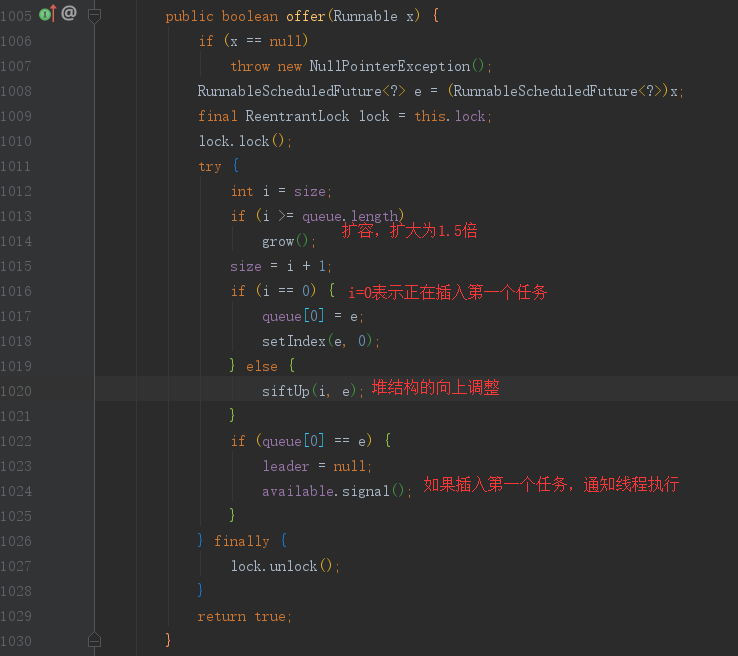


图- 12 DelayedWorkQueue添加元素

### ForkJoinPool

## 拒绝策略

线程池中的拒绝策略类全都实现了RejectedExecutionHandler接口，这个接口只提供一个方法rejectExecution，所有的具体类都要实行各自的拒绝策略，也就是通过rejectExecution来完成具体的拒绝行为。实现了该接口的类包括如下几个：

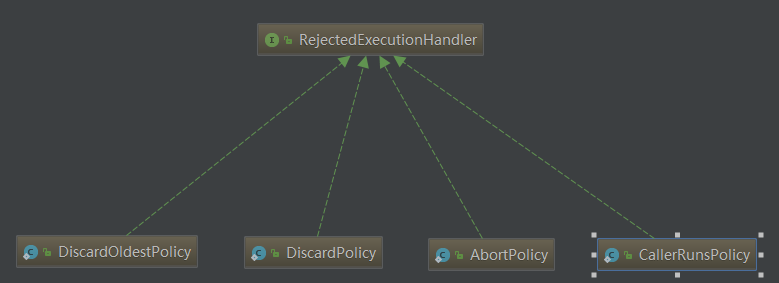


图- 13 RejectedExecutionHandler实现类

这四个拒绝策略类都是ThreadPoolExecutor的内部类。如果线程池中线程数达到最大MaximumSize，并且排队队列已满，这是将触发reject行为，触发地点在ThreadPoolExecutor的execute方法中，具体代码如下：

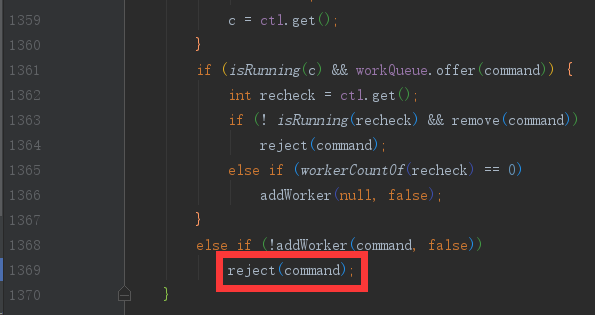


图- 14 线程池reject触发点

### AbortPolicy

如果线程池无法接收task，采用该策略将直接抛出异常。具体实现如下

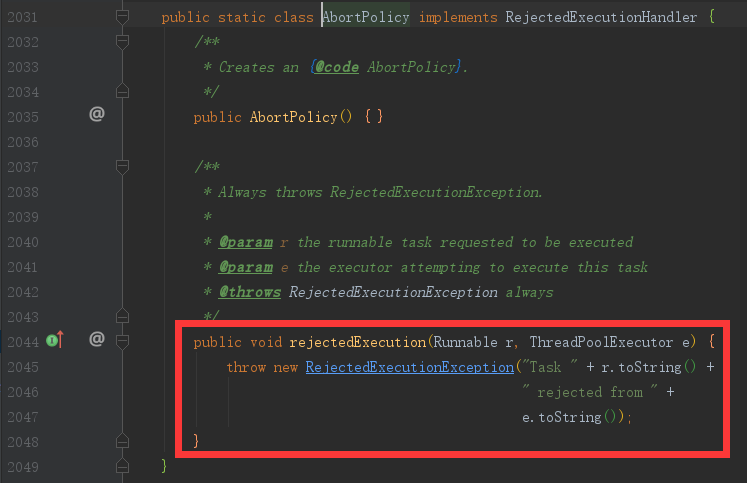


图- 15 AbortPolicy拒绝策略实现代码

### DiscardPolicy

如果线程池无法接收task，采用该策略将直接抛弃新的task。具体实现如下



图- 16 DiscardPolicy拒绝策略实现代码

### DiscardOldestPolicy

如果线程池无法接收task，采用该策略将直接抛弃队列中的第一个task。具体实现如下

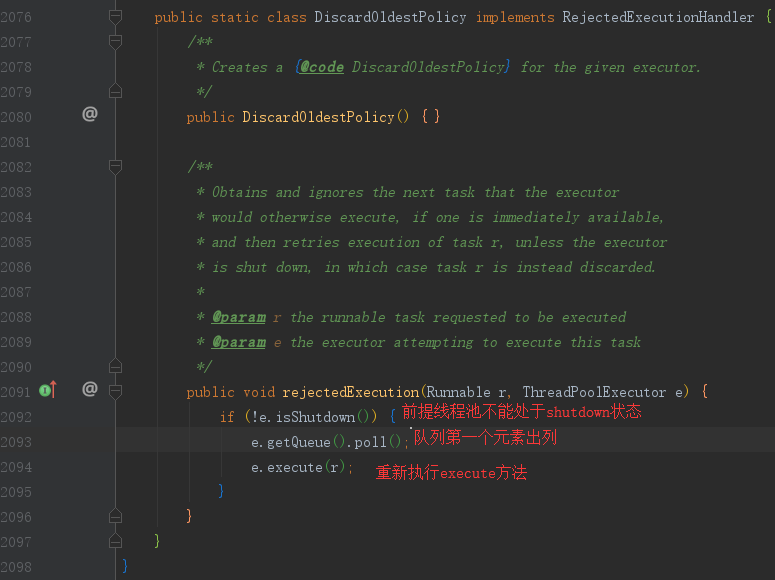


图- 17 DiscardOldestPolicy策略具体实现

### CallerRunsPolicy

如果线程池无法接收task，采用该策略将直接调用任务的run方法执行该任务。具体实现如下

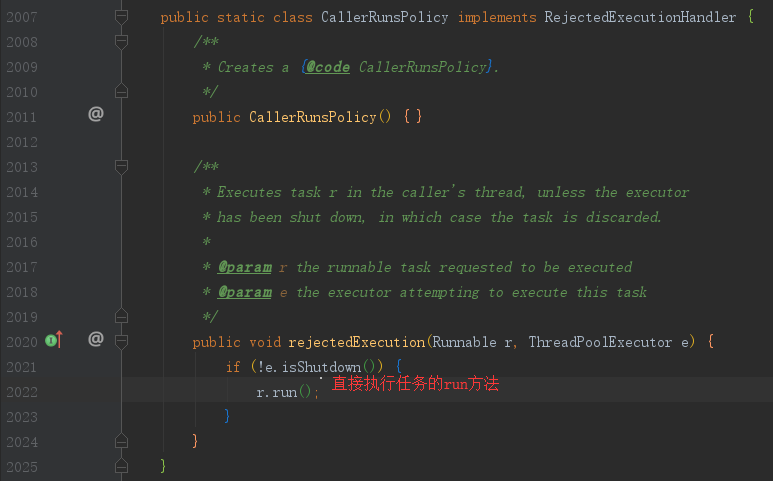


图- 18 CallerRunsPolicy策略具体实现

# 参考

《Java并发编程的艺术》