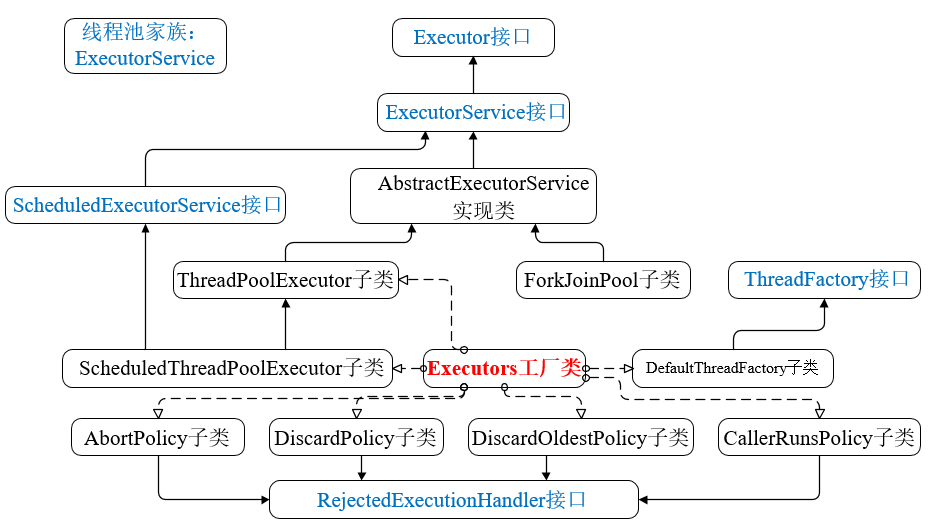
线程池实现类

# 线程池家族框图



# 线程池的工作机制及其原理

## 线程池的两个队列：

### **工作任务等待池**即工作队列BlockingQueue<Runnable> workQueue;

**作用：**存放待处理的任务；

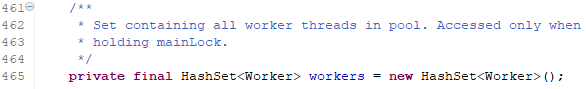
private final BlockingQueue<Runnable> workQueue;

### **任务处理池(PoolWorker)**：即存放线程池的所有工作线程的Set集合(HashSet<**Worker**>);

**作用:存放线程池中的所有的工作线程。 一个Worker对应一个工作的线程。**

ThreadPoolExecutor内部定义的Worker，是对Thread、任务Task进行了封装，并利用**completedTasks**记录该Worker已经处理的任务数，便于线程池遍历所有的worker，相加即可得到该线程池处理的任务总数。

private final **HashSet<Worker> workers = new HashSet<Worker>();**

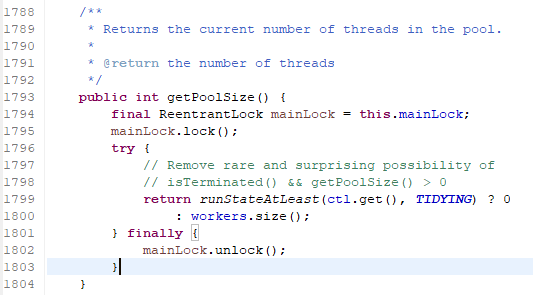


**Worker是ThreadPoolExecutor中的内部类；**

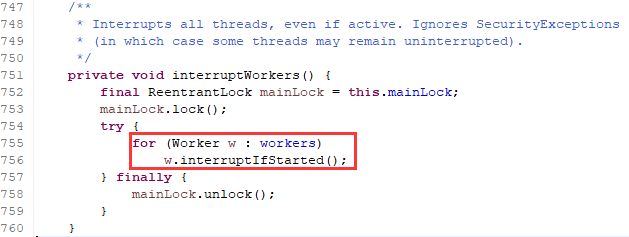
private final class **Worker** extends **AbstractQueuedSynchronizer** implements **Runnable**

**通过ThreadPoolExecutor类的getPoolSize理解workers的作用：**

**workers的大小就是线程池中线程的数量。**



**通过ThreadPoolExecutor类的interruptWorkers理解workers的作用：**



## 线程池线程大小参数

核心池大小：corePoolSize，默认情况下，线程池线程数在corePoolSize之下的线程是不会释放的；

最大处理线程数maximumPoolSize:线程池允许的最大的线程数，当线程池中的线程数大于corePoolSize小于maximumPoolSize时，线程池会动态创建与回收线程池里面线程的资源。

## 线程池的运行机制

核心线程池大小为corePoolSize，meximumPoolSize是线程池的一种补救措施，即当任务量突然过大时的一种补救措施。

举个例子最好说明。(很好的比喻)

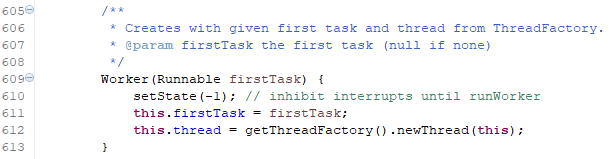
假设有一个工厂，工厂有10名正式员工，每个人工人同时只能做一件任务，因此只要当10个工人中有工人是空闲的，来了任务就分配给空闲的工人做，当10个工人都有任务在做时，如果还来了任务，就把任务进行排队等待，也就是把任务放入等待队列中。如果说新任务数目的增长速度远远大于工人做任务的速度，那么此时工人主管可能会想补救措施，比如招4个临时工人进来，然后就将任务也分配给这4个临时工人做。如果这14个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。(至于怎么抛弃就是对应的任务拒绝策略)。当14个人有人空闲下来，新任务的增长又比较缓慢，工人主管可能就考虑逐渐一个一个地辞掉这4个临时工人，只保持原来的10个工人，毕竟请额外的工人是要花钱的。

# Worker的源码理解

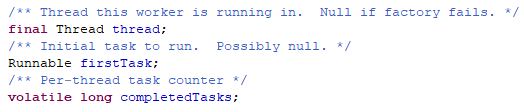
**Worker是ThreadPoolExecutor中的内部类；**

private final class **Worker** extends **AbstractQueuedSynchronizer**

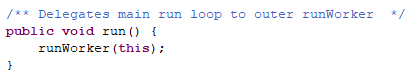
implements **Runnable**



Worker对Thread、任务Task进行了进一步的封装，并利用completedTasks记录该Worker已经处理的任务数。

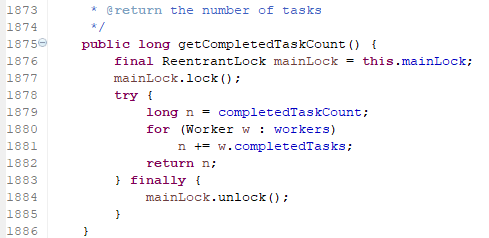


实现了Runnable的run方法：



获取线程池已经处理的任务总数：

遍历workers，把每一个Worker的完成的任务数**completedTasks**相加起来。



# AbstractExecutorService

## 继承关系

public abstract class **AbstractExecutorService** extends Object implements **ExecutorService**

**ExecutorService**接口的抽象实现类。

## 功能

AbstractExecutorService实现了ExecutorService接口中的大部分方法。

Provides **default implementations** of **ExecutorService** execution methods. This class implements the **submit**, **invokeAny** and **invokeAll** methods using a **RunnableFuture** returned by **newTaskFor**, which defaults to the **FutureTask** class provided in this package. For example, the implementation of submit(Runnable) creates an associated RunnableFuture that is executed and returned. Subclasses may override the **newTaskFor** methods to return **RunnableFuture** implementations other than FutureTask.

示例：

Extension example. Here is a sketch of a class that customizes ThreadPoolExecutor to use a CustomTask class instead of the default FutureTask:

public class CustomThreadPoolExecutor extends ThreadPoolExecutor {

static class CustomTask<V> implements RunnableFuture<V> {...}

protected <V> RunnableFuture<V> newTaskFor(Callable<V> c) {

return new CustomTask<V>(c);

}

protected <V> RunnableFuture<V> newTaskFor(Runnable r, V v) {

return new CustomTask<V>(r, v);

}

// ... add constructors, etc.

}

## 方法

### submit方法

### newTaskFor方法

### invokeAny方法

# ThreadPoolExecutor

## 继承关系

public class **ThreadPoolExecutor** extends **AbstractExecutorService**

Direct Known Subclasses ： **ScheduledThreadPoolExecutor**



## 内部类

static class **ThreadPoolExecutor.AbortPolicy**

A handler for rejected tasks that throws a RejectedExecutionException.

static class **ThreadPoolExecutor.CallerRunsPolicy**

A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded.

static class **ThreadPoolExecutor.DiscardOldestPolicy**

A handler for rejected tasks that discards the oldest unhandled request and then retries execute, unless the executor is shut down, in which case the task is discarded.

static class **ThreadPoolExecutor.DiscardPolicy**

A handler for rejected tasks that silently discards the rejected task.

## 构造方法

**共享的参数**：**int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue。**

**可选的参数：ThreadFactory threadFactory, 或RejectedExecutionHandler handler。**

**Executors工厂类中就是直接利用这些构造方法创建线程池的，其中使用newXxxThreadPool()若有参数ThreadFactory，调用的就是带有ThreadFactory的构造方法。带有与不带有ThreadFactory的区别在于：带有ThreadFactory的在创建新的线程的时候，利用指定的ThreadFactory创建定的线程。**

### **ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)**

Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory and rejected execution handler.

参数介绍：Parameters:

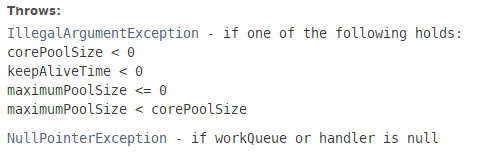
**corePoolSize** - the number of threads to keep in the pool, even if they are idle, unless **allowCoreThreadTimeOut** is set.(在线程池中维持线程的数目)

**maximumPoolSize** - the maximum number of threads to allow in the pool(最大线程数)

**keepAliveTime** - when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait for new tasks before terminating.(大于corePoolSize时存活时间)

**unit** - the time unit for the keepAliveTime argument

**workQueue** - the queue to use for holding tasks before they are executed. This queue will hold only the **Runnable** tasks submitted by the **execute** method.



### **ThreadPoolExecutor(**int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,BlockingQueue<Runnable> workQueue**, RejectedExecutionHandler handler)**

参数：**RejectedExecutionHandler** **handler** - the handler to use when execution is blocked because the thread bounds and queue capacities are reached.

Creates a new ThreadPoolExecutor with the given initial parameters and default thread factory.

### **ThreadPoolExecutor(**int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, **ThreadFactory threadFactory)**

Creates a new ThreadPoolExecutor with the given initial parameters and default rejected execution handler.

参数: **threadFactory** - the factory to use when the executor creates a new thread

### **ThreadPoolExecutor(**int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue**, ThreadFactory threadFactory, RejectedExecutionHandler handler)**

Creates a new ThreadPoolExecutor with the given initial parameters.

## 构造方法参数解析

**想要自定义线程池，就必须从以下7个参数入手**。

### corePoolSize：核心池大小。(核心线程数大小)

创建线程池之后，默认情况下，线程池中没有任何线程，只有等待任务到来后才创建线程去执行任务，除非调用了**prestartAllCoreThreads()或prestartCoreThread()**方法，从这2个方法从名字上就可以看出，是**预创建线程**的意思，即**在任务到来之前就创建corePoolSize个线程或1个线程**。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目到达corePoolSize后，就会把到达的任务放到缓存队列当中。

### maximumPoolSize：线程池最大线程数。

这个参数是一个非常重要的参数，表示**在线程池中最多能创建多少个线程**。在corePoolSize和maximumPoolSize之间的线程数会被自动释放，小于corePoolSize的不会。

### keepAliveTime：表示**线程没有任务执行时最多保持多久时间会终止**。

默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime参数才会起作用，直到线程池中的线程数不大于corePoolSize。(也就是说，当线程池的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中线程数不超过corePoolSize)。

但是，如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池数不大于corePoolSize的时候，keepAliveTime参数也会起作用，直到线程池中的线程数为0。

### TimeUnit：参数keepAliveTime的时间单位。

### BlockingQueue workQueue：工作队列(任务队列)。

一个阻塞队列，用来存储等待执行的任务。这个参数的选择很重要，对线程池的运行过程产生重大影响。选择合适的BlockingQueue根据场景的需求：一般是ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue等。

### ThreadFactory：线程工厂，主要用来创建线程。默认使用的是Executors.defaultThreadFactory()，也可以自定义一个线程工厂来创建自己需要的线程，设置线程的优先级、安全管理、名称等。

### RejectedExecutionHandler handler：表示任务拒绝处理策略，Executors中定义了4种，默认的是Executors.AbortPolicy策略。

## 添加新方法

### setXxx

Xxx与构造函数中的参数保持一致。

### getXxx

Xxx除了构造函数之外的，还有些线程池在执行过程中记录的数据。

int getCorePoolSize()

int getMaximumPoolSize()  
**int getActiveCount()**

**int getLargestPoolSize()**

**int getPoolSize() Returns the current number of threads in the pool.**

long getKeepAliveTime(TimeUnit unit)

ThreadFactory getThreadFactory()

**long getCompletedTaskCount()**

BlockingQueue<Runnable> getQueue() Returns the task queue used by this executor.

RejectedExecutionHandler getRejectedExecutionHandler()

Returns the current handler for unexecutable tasks.

### 控制线程池

shutdown

shutdownNow

boolean remove(Runnable task)

对应的判断方法

boolean isTerminating()

boolean isTerminated()

boolean isShutdown()

### execute方法

void **execute**(Runnable command)

protected void **afterExecute**(Runnable r, Throwable t)

protected void **beforeExecute**(Thread t, Runnable r)

### allowsCoreThreadTimeOut

void allowCoreThreadTimeOut(boolean value)

Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.

boolean allowsCoreThreadTimeOut()

Returns true if this pool allows core threads to time out and terminate if no tasks arrive within the keepAlive time, being replaced if needed when new tasks arrive.

### awaitTermination

boolean awaitTermination(long timeout, TimeUnit unit)

Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

### prestartAllCoreThreads

prestartCoreThread()

在创建线程池之后，预创建corePoolSize或1个线程。注意：在创建线程池之后，默认情况下，线程的数量为0，只有当任务来了之后，才会创建一个线程去执行任务。

# ThreadPoolExecutor的源码分析

## 构造方法源码4个

虽然是4个构造方法，但是都是调用第4个构造方法，这里有两个参数是有默认值的。

ThreadFactory和RejectedExecutionHandler；Executors.defaultThreadFactory()；和

private static final RejectedExecutionHandler defaultHandler =new **AbortPolicy**();

第一个：

public **ThreadPoolExecutor**(**int corePoolSize,**

**int maximumPoolSize,**

**long keepAliveTime,**

**TimeUnit unit,**

**BlockingQueue<Runnable> workQueue**) {

**this**(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

Executors.defaultThreadFactory(), defaultHandler);

}

第二个：

public **ThreadPoolExecutor**(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory) {

**this**(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

**threadFactory**, **defaultHandler**);

}

第三个：

public **ThreadPoolExecutor**(**int corePoolSize,**

**int maximumPoolSize,**

**long keepAliveTime,**

**TimeUnit unit,**

**BlockingQueue<Runnable> workQueue,**

RejectedExecutionHandler handler) {

**this**(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

Executors.defaultThreadFactory(), handler);

}

**第四个：上面三个都是调用第4个构造函数。**

public **ThreadPoolExecutor**(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

if (corePoolSize < 0 || maximumPoolSize <= 0 || maximumPoolSize < corePoolSize || keepAliveTime < 0)

throw new IllegalArgumentException();

if (workQueue == null || threadFactory == null || handler == null)

throw **new NullPointerException();**

**this.corePoolSize = corePoolSize;**

**this.maximumPoolSize = maximumPoolSize;**

**this.workQueue = workQueue;**

**this.keepAliveTime = unit.toNanos(keepAliveTime);**

**this.threadFactory = threadFactory;**

**this.handler = handler;**

}

## 属性

private **volatile** ThreadFactory threadFactory;

private **volatile** long keepAliveTime;

private **volatile** RejectedExecutionHandler handler;

private **volatile** boolean allowCoreThreadTimeOut;

private **volatile** int corePoolSize;

private **volatile** int maximumPoolSize;

private **static** **final** RejectedExecutionHandler defaultHandler = new AbortPolicy();

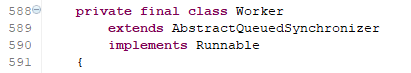
private **static final** RuntimePermission shutdownPerm = new RuntimePermission("modifyThread");

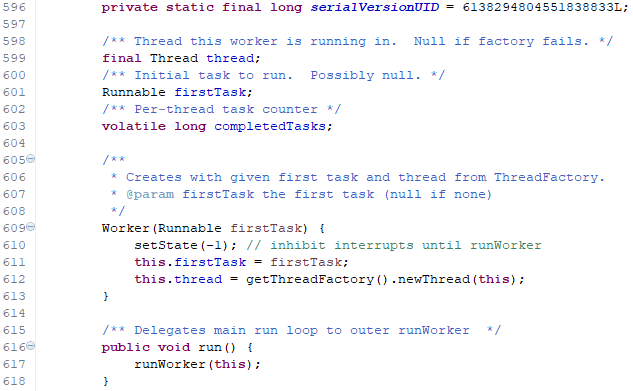
private **final** ReentrantLock mainLock = new ReentrantLock();

private **final** Condition **termination** = mainLock.newCondition();

private **final** BlockingQueue<Runnable> **workQueue**;

## ThreadPoolExecutor的内部类—Worker





# 自定义线程池

## 自定义线程池

就是通过**ThreadPoolExecutors**的构造函数来定义自己需要的线程池，所以需要弄清楚构造函数的7个参数的具体意义即可。

关键是BlockingQueue的选择与拒绝策略的选择。

## 自定义实现一个简单的Web服务的需求场景

需求场景说明

### 服务器可容纳的最小请求数是多少？

### 可以动态扩充的请求数大小是多少？

### 多久回收多余线程数即请求数；

### 用户访问量大了怎么处理；

### 线程队列机制采取有**优先级的排队的执行机制**。

## 定义线程池

**设置一个类不能被继承，两种方法：一是类用final修饰，二就是构造方法私有。**

**定义自己的线程池，简单的方法就是通过ThreadPoolExecutors的构造函数创建ThreadPool。**

**public class MyExecutors {**

**public static ExecutorService newMyWebThreadExecutors(int corePoolSize,int maximumPoolSize,int keepAliveTime){**

**return new ThreadPoolExecutor(corePoolSize,maximumPoolSize,keepAliveTime,**

**TimeUnit.MILLISECONDS,new PriorityBlockingQueue<Runnable>());**

**}**

**public static ScheduledExecutorService newMyWebScheduledEThreadxecutors(int corePoolSize){**

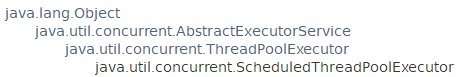
**return new ScheduledThreadPoolExecutor(corePoolSize);**

**}**

**}**

# ScheduledThreadPoolExecutor

## 继承关系介绍

public class **ScheduledThreadPoolExecutor** extends **ThreadPoolExecutor** implements **ScheduledExecutorService**

## 功能介绍

A **ThreadPoolExecutor** that can additionally **schedule commands** to run after a given delay, or to execute **periodically**. This class is preferable to **Timer** when multiple worker threads are needed, or when the additional flexibility or capabilities of **ThreadPoolExecutor** (which this class extends) are required.

**Delayed tasks** execute no sooner than they are enabled, but without any real-time guarantees about when, after they are enabled, they will commence. Tasks scheduled for exactly the same execution time are enabled in first-in-first-out (FIFO) order of submission.

When a submitted task is cancelled before it is run, execution is suppressed. By default, such a cancelled task is not automatically removed from the work queue until its delay elapses. While this enables further inspection and monitoring, it may also cause unbounded retention of cancelled tasks. To avoid this, set **setRemoveOnCancelPolicy(boolean)** to true, which causes tasks to be immediately removed from the work queue at time of cancellation.

Successive executions of a task scheduled via **scheduleAtFixedRate** or scheduleWithFixedDelay do not overlap. While different executions may be performed by different threads, the effects of prior executions happen-before those of subsequent ones.

While this class inherits from **ThreadPoolExecutor**, a few of the inherited tuning methods are not useful for it. In particular, because it acts as a fixed-sized pool using corePoolSize threads and an unbounded queue, adjustments to maximumPoolSize have no useful effect. Additionally, it is almost never a good idea to set corePoolSize to zero or use **allowCoreThreadTimeOut** because this may leave the pool without threads to handle tasks once they become eligible to run.

**Extension notes**: This class overrides the execute and submit methods to generate internal ScheduledFuture objects to control per-task delays and scheduling. To preserve functionality, any further overrides of these methods in subclasses must invoke superclass versions, which effectively disables additional task customization. However, this class provides alternative protected extension method decorateTask (one version each for Runnable and Callable) that can be used to customize the concrete task types used to execute commands entered via execute, submit, schedule, scheduleAtFixedRate, and scheduleWithFixedDelay. By default, a ScheduledThreadPoolExecutor uses a task type extending FutureTask. However, this may be modified or replaced using subclasses of the form:

public class **CustomScheduledExecutor** extends **ScheduledThreadPoolExecutor** {

static class CustomTask<V> implements RunnableScheduledFuture<V> { ... }

protected <V> RunnableScheduledFuture<V> decorateTask(

Runnable r, RunnableScheduledFuture<V> task) {

return new CustomTask<V>(r, task);

}

protected <V> RunnableScheduledFuture<V> decorateTask(

Callable<V> c, RunnableScheduledFuture<V> task) {

return new CustomTask<V>(c, task);

}

// ... add constructors, etc.

}

## 构造方法

参数：**int corePoolSize、ThreadFactory threadFactory 、RejectedExecutionHandler handler**。

ScheduledThreadPoolExecutor(int corePoolSize)

Creates a new ScheduledThreadPoolExecutor with the given core pool size.

ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler)

Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory)

Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler)

Creates a new ScheduledThreadPoolExecutor with the given initial parameters.

## 新方法

**最体现scheduled的3个方法：schedule、scheduleAtFixedRate、scheduleWithFixedDelay**

### **schedule**

<V> ScheduledFuture<V> **schedule**(Callable<V> callable, long delay, TimeUnit unit)

Creates and executes a **ScheduledFuture** that becomes enabled after the given delay.

ScheduledFuture<?> **schedule**(Runnable command, long delay, TimeUnit unit)

Creates and executes **a one-shot action** that becomes enabled after the given delay.

### **scheduleAtFixedRate**

ScheduledFuture<?> **scheduleAtFixedRate**(Runnable command, long initialDelay, long period, TimeUnit unit)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 \* period, and so on.

### **scheduleWithFixedDelay**

ScheduledFuture<?> **scheduleWithFixedDelay**(Runnable command, long initialDelay, long delay, TimeUnit unit)

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.

### submit方法

<T> Future<T> **submit**(Callable<T> task)

Submits a value-returning task for execution and returns a Future representing the pending results of the task.

Future<?> **submit**(Runnable task)

Submits a Runnable task for execution and returns a Future representing that task.

<T> Future<T> **submit**(Runnable task, T result)

Submits a Runnable task for execution and returns a Future representing that task.

### shutdown方法

void **shutdown**()

Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

List<Runnable> **shutdownNow**()

Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

### setXxx方法

void **setContinueExistingPeriodicTasksAfterShutdownPolicy**(boolean value)

Sets the policy on whether to continue executing existing periodic tasks even when this executor has been shutdown.

void **setExecuteExistingDelayedTasksAfterShutdownPolicy**(boolean value)

Sets the policy on whether to execute existing delayed tasks even when this executor has been shutdown.

void **setRemoveOnCancelPolicy**(boolean value)

Sets the policy on whether cancelled tasks should be immediately removed from the work queue at time of cancellation.