Executor框架介绍：ExecutorService与ScheduledExecutorService接口

1. **Executor框架概述**

顶级接口Executor只定义一个execute方法；一般不使用该接口。

**如果操作一般线程池，调用ExecutorService接口；**

**如果操作定时线程池，调用ScheduledExecutorService接口。**

真正的顶级接口**ExecutorService**，定义了大部分方法如submit等；

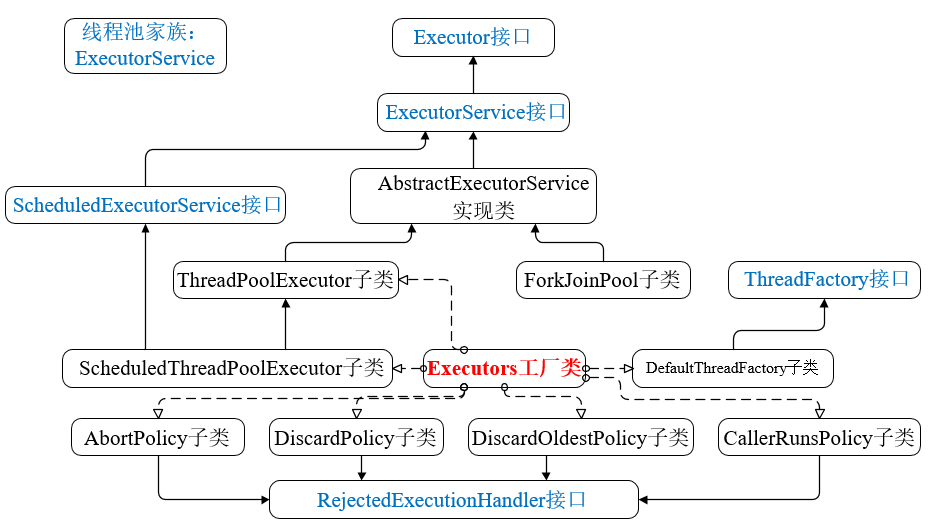
针对**ExecutorService接口**有一个抽象实现类AbstractExecutorService；

关键的线程池类**ThreadPoolExecutor**继承AbstractExecutorService，创建线程池就是通过该类实现的。

对于ScheduledThreadPoolExecutor类继承了ThreadPoolExecutor，利用DelayQueue实现了定时的功能。

1. **线程池接口及其实现类**

**线程池接口及其实现类框图：**



## Executor、ExecutorService 、ScheduledExecutorService关系

**Executor接口**是线程池的顶级接口，但是严格意义上讲，Executor并不是一个线程池，而只是执行线程的工具。**Executor没有实现类**。Executor接口有一个直接**子接口**ExecutorService，**ExecutorService接口是真正的线程池接口**。

ExecutorService接口下有一个子接口**ScheduledExecutorService定时线程池接口，与Timer/TimerTask类似，用于解决需要定时周期执行的问题**。

因此，**线程池接口主要是ExecutorService和ScheduledExecutorService。**

**Executor ：An object that executes submitted Runnable tasks.(执行线程任务)**

**ExecutorService ：An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.(提供方法管理线程)**

## 实现类：

Executor接口没有直接**实现类**；

ExecutorService的直接实现类有3个：**AbstractExecutorService**, **ForkJoinPool**, **ThreadPoolExecutor。**

ScheduledExecutorService的实现类有1个：**ScheduledThreadPoolExecutor(继承于ThreadPoolExecutor)**。

**AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法。ThreadPoolExecutor是ExecutorService接口的默认实现，继承了AbstractExecutorService类。ScheduledThreadPoolExecutor继承于ThreadPoolExecutor，用于定时周期性任务调度的类实现。**

1. **Executor接口—线程池的顶级接口<ExecutorService是主要接口>**
2. **package**

public interface Executor

Executor接口存在于java.util.concurrent包中，**具有2个子接口**，All Known Subinterfaces:**ExecutorService**, **ScheduledExecutorService；**

**所有的实现类：**All Known Implementing Classes:**AbstractExecutorService**, **ForkJoinPool**, **ScheduledThreadPoolExecutor**, **ThreadPoolExecutor。**

**Executor接口只规定了一个方法: execute(Runnable command)。**

**纠正：**之前Runnable的实现类喜欢命名为Target，不过命名为Task或Command比较好。Target、Task、Command都可以。

1. **功能介绍**

**Executor接口**是线程池的顶级接口，但是严格意义上讲**Executor**并不是一个线程池，而只是一个执行线程的工具。其子接口**ExecutorService**才是真正的线程池接口。

**An object that executes submitted Runnable tasks.** This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used **instead of explicitly(明确地) creating threads**. For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:

Executor executor = anExecutor;

executor.execute(**new RunnableTask1());**

executor.execute(**new RunnableTask2());**…

1. **唯一的接口方法：execute(Runnable command)**

**void execute(Runnable command)**

Executes the given command at some time in the future. The command may execute **in a new thread, in a pooled thread, or in the calling thread**, at the discretion of the Executor implementation. ( at the discretion of … 由…来决定)

Parameters: command - the runnable task

Throws: RejectedExecutionException - if this task cannot be accepted for execution.

NullPointerException - if command is null

1. **Executor接口的实现方式介绍**
2. 最简单的方式：直接实现

However, the Executor interface does not strictly require that execution be **asynchronous(异步的)**. In the simplest case, an executor can run the submitted task immediately in the caller's thread:

class **DirectExecutor** implements **Executor** {

public void execute(Runnable r) {

r.run();

}

}

1. 典型的：**开启新线程**

**More typically**, tasks are executed in some thread other than the caller's thread. The executor below spawns a new thread for each task.

class **ThreadPerTaskExecutor** implements **Executor** {

public void execute(Runnable r) {

new Thread(r).start();

}

}

1. 施加限制条件

Many Executor implementations **impose**(施加,利用) **some sort of limitation** on how and when tasks are scheduled. The executor below serializes the submission of tasks to a second executor, illustrating a **composite**(混合的) executor.

class **SerialExecutor** implements **Executor** {

final Queue<Runnable> tasks = new **ArrayDeque**<Runnable>();

final Executor executor;

Runnable active

SerialExecutor(Executor executor) {

this.executor = executor;

}

public **synchronized** void execute(final Runnable r) {

tasks.offer(new Runnable() {

public void run() {

try {

r.run();

} finally {

scheduleNext();

}

}

});

if (active == null) {

scheduleNext();

}

}

protected **synchronized** void scheduleNext() {

**if ((active = tasks.poll()) != null) {**

**executor.execute(active);**

**}**

}

}

The **Executor** implementations provided in this package implement **ExecutorService**, which is a more extensive interface. The **ThreadPoolExecutor** class provides an extensible thread pool implementation. The **Executors** class provides convenient factory methods for these **Executors**.

## 源代码：

\* @since 1.5

\* **@author Doug Lea**

public interface Executor {

void execute(Runnable command);

}

# ExecutorService—真正的线程池接口

## package

public interface **ExecutorService** extends **Executor**

**ExecutorService接口**存在于**java.util.concurrent**包中，继承了Executor接口，还有一个子接口**ScheduledExecutorService；**

4个实现类：**AbstractExecutorService**, ForkJoinPool, **ThreadPoolExecutor，***ScheduledThreadPoolExecutor***(继承了ThreadPoolExecutor，实现ScheduledExecutorService)。**

## 功能

An **Executor** that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

An **ExecutorService** can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an **ExecutorService**. The shutdown() method will allow previously submitted tasks to execute before terminating, while the **shutdownNow()** method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted. An unused ExecutorService should be shut down to allow reclamation of its resources.

shutdown方法：允许当前执行的线程执行完毕；

shutdownNow方法：试图停止当前正在进行和等待执行的线程。

Method **submit** extends base method **Executor.execute(java.lang.Runnable)** by creating and returning a **Future** that can be used to cancel execution and/or wait for completion. Methods **invokeAny** and **invokeAll** perform the most commonly useful forms of bulk execution, executing a collection of tasks and then waiting for at least one, or all, to complete. (Class ExecutorCompletionService can be used to write customized variants of these methods.)

The **Executors** class provides factory methods for the executor services provided in this package.(见Executors)

## 定义的方法

### 继承Executor接口的execute方法

void execute(Runnable command)

Executes the given command at some time in the future.

### invokeAll方法

**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)**

Executes the given tasks, returning a list of **Futures** holding their status and results when all complete.

**<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)**

Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.

### invokeAny方法

**<T> T invokeAny(Collection<? extends Callable<T>> tasks)**

Executes the given tasks**, returning the result of one** that has completed successfully (i.e., without throwing an exception), if any do.

**<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)**

Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.

### submit

<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.

### awaitTermination方法

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.

### isXxx判断方法

boolean **isShutdown()**

Returns true if this executor has been shut down.

boolean **isTerminated()**

Returns true if all tasks have completed following shut down.

## 使用示例

Here is a sketch of a network service in which threads in a thread pool service incoming requests. It uses the preconfigured Executors.newFixedThreadPool(int) factory method:

class **NetworkService** implements Runnable {

private final ServerSocket serverSocket;

private final ExecutorService pool;

public NetworkService(int port, int poolSize)

throws IOException {

serverSocket = new ServerSocket(**port**);

pool = Executors.newFixedThreadPool(**poolSize**);

}

public void run() { // run the service

try {

for (;;) {

pool.execute(new Handler(serverSocket.accept()));

}

} catch (IOException ex) {

pool.shutdown();

}

}

}

class Handler implements Runnable {

private final Socket socket;

Handler(Socket socket) { this.socket = socket; }

public void run() {

// read and service request on socket

}

}

The following method shuts down an **ExecutorService** in two phases, **first** by calling shutdown to reject incoming tasks, and **then** calling shutdownNow, if necessary, to cancel any lingering tasks:

void shutdownAndAwaitTermination(ExecutorService pool) {

pool.shutdown(); // Disable new tasks from being submitted

try {

// Wait a while for existing tasks to terminate

if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {

**pool.shutdownNow();** // Cancel currently executing tasks

// Wait a while for tasks to respond to being cancelled

if (!pool.awaitTermination(60, TimeUnit.SECONDS))

System.err.println("Pool did not terminate");

}

} catch (InterruptedException ie) {

// (Re-)Cancel if current thread also interrupted

pool.shutdownNow();

// Preserve interrupt status

Thread.currentThread().interrupt();

}

}

# ScheduledExecutorService定时线程池接口<继承ExecutorService>

## package

public interface **ScheduledExecutorService** extends **ExecutorService**

**ScheduledExecutorService接口**存在于java.util.concurrent包中，直接继承于ExecutorService接口，间接继承于**Executor**；

有1个实现类：**ScheduledThreadPoolExecutor**。

## 功能：

**定时执行；定时且周期执行；定时且指定时间间隔执行。**

An **ExecutorService** that can schedule commands to run after a given delay, or to execute periodically.

The schedule methods create tasks with various delays and return a task object that can be used to cancel or check execution. The **scheduleAtFixedRate** and **scheduleWithFixedDelay** methods create and execute tasks that run periodically until cancelled.

Commands submitted using the Executor.execute(java.lang.Runnable) and ExecutorService submit methods are scheduled with a requested delay of zero. Zero and negative delays (but not periods) are also allowed in schedule methods, and are treated as requests for immediate execution.

All schedule methods accept relative delays and periods as arguments, not absolute times or dates. It is a simple matter to transform an absolute time represented as a Date to the required form. For example, to schedule at a certain future date, you can use: schedule(task, date.getTime() - System.currentTimeMillis(), TimeUnit.MILLISECONDS). Beware however that expiration of a relative delay need not coincide with the current Date at which the task is enabled due to network time synchronization protocols, clock drift, or other factors. The **Executors** class provides convenient factory methods for the **ScheduledExecutorService** implementations provided in this package.

## 方法介绍

### 继承Executor、ExecutorService的方法

### **schedule方法：定时执行任务**

<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：首次定时，后序周期执行**

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：首次定时，后序指定时间间隔**

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.

## scheduleAtFixedRate与scheduleWithFixedDelay的区别

**初始initialDelay延迟都一样，不同的是delay**，scheduleAtFixedRate指的是前一个线程开始到后1个线程开始之间的时间为delay，周期执行；而scheduleWithFixedDelay指的是前一个线程结束到后1个线程开始之间的时间为delay，并不一定是周期执行的。

schedule**AtFixedRate**：线程开始-->开始，时间间隔Delay；

schedule**WithFixedDelay**：线程结束-->下一个线程开始，时间间隔Delay。

## 使用示例

Here is a class with a method that sets up a **ScheduledExecutorService** to beep every ten seconds for an hour:

import static java.util.concurrent.TimeUnit.\*;

class BeeperControl {

private final ScheduledExecutorService **scheduler** =

**Executors.newScheduledThreadPool(1);**

public void beepForAnHour() {

final Runnable beeper = new Runnable() {

public void run() { System.out.println("beep"); }

};

final ScheduledFuture<?> beeperHandle =

**scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);**

scheduler.**schedule**(new Runnable() {

public void run() { beeperHandle.cancel(true); }

}, 60 \* 60, SECONDS);

}

}

# ThreadPool 与Executor的区别

现有理解：Thread Pool中存在多个线程，这些线程可以循环接收Runnable实现任务；而Executor只是一个执行器，用来执行Runnable实现任务。对于单线程的线程池其等价与一个Executor，因为接收任务直接run即可。