Java多线程进阶（三九）—— J.U.C之executors框架：executors框架概述

 Ressmix 发布于 2018-10-03

> 本文首发于一世流云专栏：https://segmentfault.com/blog...

一、executors框架简介

`juc-executors`框架是整个J.U.C包中类/接口关系最复杂的框架，真正理解executors框架的前提是理清楚各个模块之间的关系，高屋建瓴，从整体到局部才能透彻理解其中各个模块的功能和背后的设计思路。

网上有太多文章讲executors框架，要么泛泛而谈，要么一叶障目不见泰山，缺乏整体视角，很多根本没有理解整个框架的设计思想和模块关系。本文将对整个executors框架做综述，介绍各个模块的功能和联系，后续再深入探讨每个模块，包括模块中的各个工具类。

从Executor谈起

`Executor`是JDK1.5时，随着J.U.C引入的一个接口，引入该接口的主要目的是**解耦任务本身和任务的执行。**我们之前通过线程执行一个任务时，往往需要先创建一个线程，然后调用线程的`start`方法来执行任务：

```
new Thread(new(RunnableTask())).start();
```

> 上述RunnableTask是实现了Runnable接口的任务类

而Executor接口解耦了任务和任务的执行，该接口只有一个方法，入参为待执行的任务：

```
public interface Executor {
    /**
     * 执行给定的Runnable任务.
     * 根据Executor的实现不同，具体执行方式也不相同.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be accepted for execution
     * @throws NullPointerException       if command is null
     */
    void execute(Runnable command);
}
```

我们可以像下面这样执行任务，而不必关心线程的创建：

```
Executor executor = someExecutor;       // 创建具体的Executor对象
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

由于Executor仅仅是一个接口，所以根据其实现的不同，执行任务的具体方式也不尽相同，比如：

①同步执行任务

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

DirectExecutor是一个同步任务执行器，对于传入的任务，只有执行完成后execute才会返回。

②异步执行任务

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

ThreadPerTaskExecutor是一个异步任务执行器，对于每个任务，执行器都会创建一个新的线程去执行任务。

> 注意：Java线程与本地操作系统的线程是一一映射的。Java线程启动时会创建一个本地操作系统线程；当该Java线程终止时，对应操作系统线程会被回收。由于CPU资源是有限的，所以线程数量有上限，所以一般由线程池来管理线程的创建/回收，而上面这种方式其实是线程池的雏形。

③对任务进行排队执行

```
class SerialExecutor implements Executor {
  final Queue<Runnable> tasks = new ArrayDeque<Runnable>();
  final Executor executor;
  Runnable active;

  SerialExecutor(Executor executor) {
    this.executor = executor; //传进一个执行框架
  }

  public synchronized void execute(final Runnable r) {
    tasks.offer(new Runnable() {
      public void run() {
       try {
         r.run();
       } finally {
         scheduleNext();
       }
      }
    });
    if (active == null) {
      scheduleNext();//执行
    }
  }

  protected synchronized void scheduleNext() {
    if ((active = tasks.poll()) != null) {
      executor.execute(active);
    }
  }
```

重构任务，执行完后执行下一个任务

SerialExecutor 会对传入的任务进行排队（FIFO顺序），然后从队首取出一个任务执行。

以上这些示例仅仅是给出了一些可能的Executor实现，J.U.C包中提供了很多Executor的具体实现类，我们以后会具体讲到，这里关键是理解Executor的设计思想——对任务和任务的执行解耦。

增强的Executor——ExecutorService

Executor接口提供的功能很简单，为了对它进行增强，J.U.C又提供了一个名为`ExecutorService`接口，ExecutorService也是在JDK1.5时，随着J.U.C引入的：

```
public interface ExecutorService
extends Executor
```

可以看到，ExecutorService继承了Executor，它在Executor的基础上增强了对任务的控制，同时包括对自身生命周期的管理，主要有四类：

1. 关闭执行器，禁止任务的提交；
2. 监视执行器的状态；
3. 提供对异步任务的支持；
4. 提供对批处理任务的支持。

```
     *
     * @param tasks 任务集合
     * @param <T>   任务的返回结果类型
     * @return 任务的Future对象列表，列表顺序与集合中的迭代器所生成的顺序相同，
     * @throws InterruptedException       如果等待时发生中断，会将所有未完成的任务取消.
     * @throws NullPointerException       任一任务为 null
     * @throws RejectedExecutionException 如果任一任务无法安排执行
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException;

    /**
     * 执行给定集合中的所有任务，当所有任务都执行完成后或超时期满时（无论哪个首先发生），返回保持任务状态和结果的 Future 列表.
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) throws InterruptedException;

    /**
     * 执行给定集合中的任务，只有其中某个任务率先成功完成（未抛出异常），则返回其结果.
     * 一旦正常或异常返回后，则取消尚未完成的任务.
     */
    <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException;

    /**
     * 执行给定集合中的任务，如果在给定的超时期满前，某个任务已成功完成（未抛出异常），则返回其结果.
     * 一旦正常或异常返回后，则取消尚未完成的任务.
     */
```

> 关于Future，其实就是Java多线程设计模式中**Future模式**，读者可以先参考下我的这篇博文（https://segmentfault.com/a/11...），后面我们会专门讲解J.U.C中的Future框架。
> `Future`对象提供了对任务异步执行的支持，也就是说调用线程无需等待任务执行完成，提交待执行的任务后，就会立即返回往下执行。然后，可以在需要时检查Future是否有结果了，如果任务已执行完毕，通过`Future.get()`方法可以获取到执行结果——Future.get()是阻塞方法。

周期任务的调度——ScheduledExecutorService

在工业环境中，我们可能希望提交给执行器的某些任务能够定时执行或周期性地执行，这时我们可以自己实现Executor接口来创建符合我们需要的类，Doug Lea已经考虑到了这类需求，所以在ExecutorService的基础上，又提供了一个接口——`ScheduledExecutorService`，该接口也是在JDK1.5时，随着J.U.C引入的：

```
public interface ScheduledExecutorService
extends ExecutorService
```

ScheduledExecutorService提供了一系列schedule方法，可以在给定的延迟后执行提交的任务，或者每个指定的周期执行一次提交的任务，我们来看下面这个示例：

```
public class ScheduleExecutorTest {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = someScheduler;      // 创建一个ScheduledExecutorService实例

        final ScheduledFuture<?> scheduledFuture = scheduler.scheduleAtFixedRate(new BeepTask(), 10, 10,
                TimeUnit.SECONDS);                                // 每隔10s蜂鸣一次

        scheduler.schedule(new Runnable() {
            @Override
            public void run() {
                scheduledFuture.cancel(true);
            }
        }, 1, TimeUnit.HOURS)       // 1小时后，取消蜂鸣任务
    }

    private static class BeepTask implements Runnable {
        @Override
        public void run() {
            System.out.println("beep!");
        }
    }
}
```

上述示例先创建一个ScheduledExecutorService类型的执行器，然后利用scheduleAtFixedRate方法提交了一个“蜂鸣”任务，每隔10s该任务会执行一次。

注意：`scheduleAtFixedRate`方法返回一个ScheduledFuture对象，ScheduledFuture其实就是在Future的基础上增加了延迟的功能。通过ScheduledFuture，可以取消一个任务的执行，本例中我们利用schedule方法，设定在1小时后，执行任务的取消。

ScheduledExecutorService完整的接口声明如下：

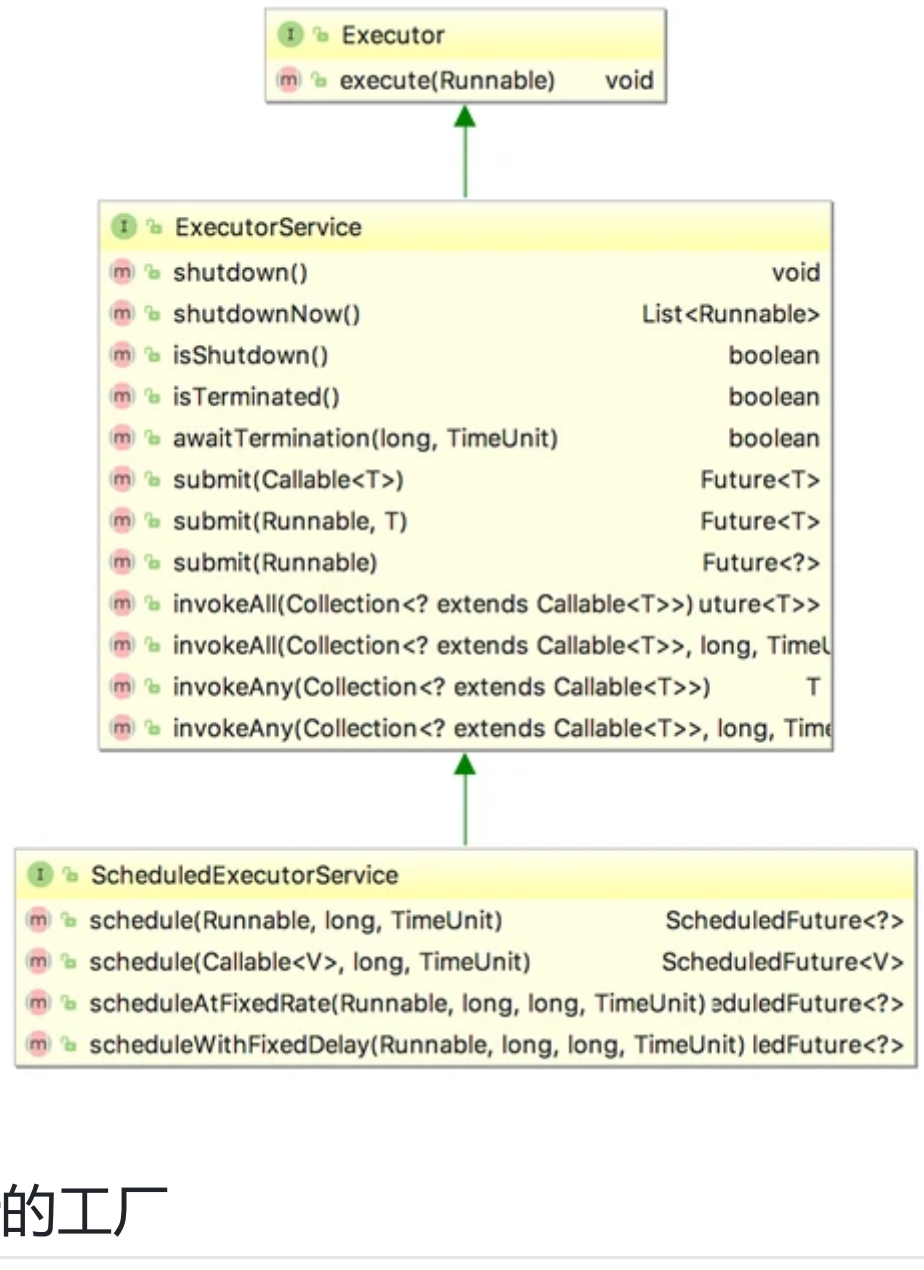
```
public interface ScheduledExecutorService extends ExecutorService {

    /**
     * 提交一个待执行的任务，并在给定的延迟后执行该任务.
     *
     * @param command 待执行的任务
     * @param delay   延迟时间
     * @param unit    延迟时间的单位
     */
    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

    /**
     * 提交一个待执行的任务（具有返回值），并在给定的延迟后执行该任务.
     *
     * @param command 待执行的任务
     * @param delay   延迟时间
     * @param unit    延迟时间的单位
     * @param <V>     返回值类型
     */
    public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

    /**
     * 提交一个待执行的任务.
     * 该任务在 initialDelay 后开始执行，然后在 initialDelay+period 后执行，接着在 initialDelay + 2 * period 后执行，
依此类推.
     *
```

至此，Executors框架中的三个最核心的接口介绍完毕，这三个接口的关系如下图：



二、生产executor的工厂

通过第一部分的学习，读者应该对Executors框架有了一个初步的认识，Executors框架就是用来解耦任务本身与任务的执行，并提供了三个核心接口来满足使用者的需求：

1. ***Executor***：提交普通的可执行任务
2. ***ExecutorService***：提供对线程池生命周期的管理、异步任务的支持
3. ***ScheduledExecutorService***：提供对任务的周期性执行支持

既然上面三种执行器只是接口，那么就一定存在具体的实现类，J.U.C提供了许多默认的接口实现，如果要用户自己去创建这些类的实例，就需要了解这些类的细节，有没有一种直接的方式，仅仅根据一些需要的特性（参数）就创建这些实例呢？因为对于用户来说，其实使用的只是这三个接口。

JDK1.5时，J.U.C中还提供了一个`Executors`类，专门用于创建上述接口的实现类对象。Executors其实就是一个简单工厂，它的所有方法都是static的，用户可以根据需要，选择需要创建的执行器实例，Executors一共提供了**五类**可供创建的Executor执行器实例。

固定线程数的线程池

Executors提供了两种创建具有固定线程数的Executor的方法，固定线程池在初始化时确定其中的线程总数，运行过程中会始终维持线程数量不变。

可以看到下面的两种创建方法其实都返回了一个`ThreadPoolExecutor`实例。ThreadPoolExecutor是一个ExecutorService接口的实现类，我们会在后面用专门章节讲解，现在只需要了解这是一种Executor，用来调度其中的线程的执行即可。

```
/**
 * 创建一个具有固定线程数的Executor.
 */
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>());
}

/**
 * 创建一个具有固定线程数的Executor.
 * 在需要时使用提供的 ThreadFactory 创建新线程.
 */
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(), threadFactory);

}
```

上面需要注意的是ThreadFactory这个接口：

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

设置一些线程的属性（比如名称等）

既然返回的是一个线程池，那么就涉及线程的创建，一般我们需要通过 new Thread ()这种方法创建一个新线程，但是我们可能希望设置一些线程属性，比如
名称、守护程序状态、ThreadGroup 等等，线程池中的线程非常多，如果每个线程都这样手动配置势必非常繁琐，而ThreadFactory作为一个线程工厂可以让我们从这些繁琐的线程状态设置的工作中解放出来，还可以由外部指定ThreadFactory实例，以决定线程的具体创建方式。

Executors提供了静态内部类，实现了ThreadFactory接口，最简单且常用的就是下面这个**DefaultThreadFactory**：

```
/**
 * 默认的线程工厂.
 */
static class DefaultThreadFactory implements ThreadFactory {
    private static final AtomicInteger poolNumber = new AtomicInteger(1);
    private final ThreadGroup group;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    private final String namePrefix;

    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() : Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" + poolNumber.getAndIncrement() + "-thread-";
    }

    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r, namePrefix + threadNumber.getAndIncrement(), 0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

可以看到，DefaultThreadFactory 初始化的时候定义了线程组、线程名称等信息，每创建一个线程，都给线程统一分配这些信息，避免了一个个手工通过new的方式创建线程，又可进行工厂的复用。

单个线程的线程池

除了固定线程数的线程池，Executors还提供了两种创建只有单个线程Executor的方法：

```
/**
 * 创建一个使用单个 worker 线程的 Executor.
 */
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>()));
}

/**
 * 创建一个使用单个 worker 线程的 Executor.
 * 在需要时使用提供的 ThreadFactory 创建新线程.
 */
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
    return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>(), threadFactory));
}
```

可以看到，只有单个线程的线程池其实就是指定线程数为1的固定线程池，主要区别就是，返回的Executor实例用了一个`FinalizableDelegatedExecutorService`对象进行包装。

我们来看下**FinalizableDelegatedExecutorService**，该类 只定义了一个finalize方法：

```
static class FinalizableDelegatedExecutorService extends DelegatedExecutorService {
    FinalizableDelegatedExecutorService(ExecutorService executor) {
        super(executor);
    }
    protected void finalize() {
        super.shutdown();
    }
}
```

核心是其继承的**DelegatedExecutorService** ， 这是一个包装类，实现了ExecutorService的所有方法，但是内部实现其实都委托给了传入的ExecutorService 实例：

```
/**
 * ExecutorService实现类的包装类.
 */
static class DelegatedExecutorService extends AbstractExecutorService {
    private final ExecutorService e;

    DelegatedExecutorService(ExecutorService executor) {
        e = executor;
    }

    public void execute(Runnable command) {
        e.execute(command);
    }

    public void shutdown() {
        e.shutdown();
    }

    public List<Runnable> shutdownNow() {
        return e.shutdownNow();
    }

    public boolean isShutdown() {
        return e.isShutdown();
    }
```

> **为什么要多此一举，加上这样一个委托层?** 因为返回的ThreadPoolExecutor包含一些设置线程池大小的方法——比如setCorePoolSize，对于只有单个线程的线程池来说，我们是不希望用户通过强转的方式使用这些方法的，所以需要一个包装类，只暴露ExecutorService本身的方法。

可缓存的线程池

有些情况下，我们虽然创建了具有一定线程数的线程池，但出于资源利用率的考虑，可能希望在特定的时候对线程进行回收（比如线程超过指定时间没有被使用），Executors就提供了这种类型的线程池：

```
/**
 * 创建一个可缓存线程的Execotor.
 * 如果线程池中没有线程可用，则创建一个新线程并添加到池中;
 * 如果有线程长时间未被使用(默认60s，可通过threadFactory配置)，则从缓存中移除.
 */
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>());
}

/**
 * 创建一个可缓存线程的Execotor.
 * 如果线程池中没有线程可用，则创建一个新线程并添加到池中;
 * 如果有线程长时间未被使用(默认60s，可通过threadFactory配置)，则从缓存中移除.
 * 在需要时使用提供的 ThreadFactory 创建新线程.
 */
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(), threadFactory);
}
```

可以看到，返回的还是ThreadPoolExecutor对象，只是指定了超时时间，另外线程池中线程的数量在`[0, Integer.MAX_VALUE]`之间。

可延时/周期调度的线程池

如果有任务需要延迟/周期调用，就需要返回ScheduledExecutorService接口的实例，`ScheduledThreadPoolExecutor`就是实现了ScheduledExecutorService接口的一种Executor，和ThreadPoolExecutor一样，这个我们后面会专门讲解。

```
/**
 * 创建一个具有固定线程数的 可调度Executor.
 * 它可安排任务在指定延迟后或周期性地执行.
 */
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

/**
 * 创建一个具有固定线程数的 可调度Executor.
 * 它可安排任务在指定延迟后或周期性地执行.
 * 在需要时使用提供的 ThreadFactory 创建新线程.
 */
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
```

Fork/Join线程池

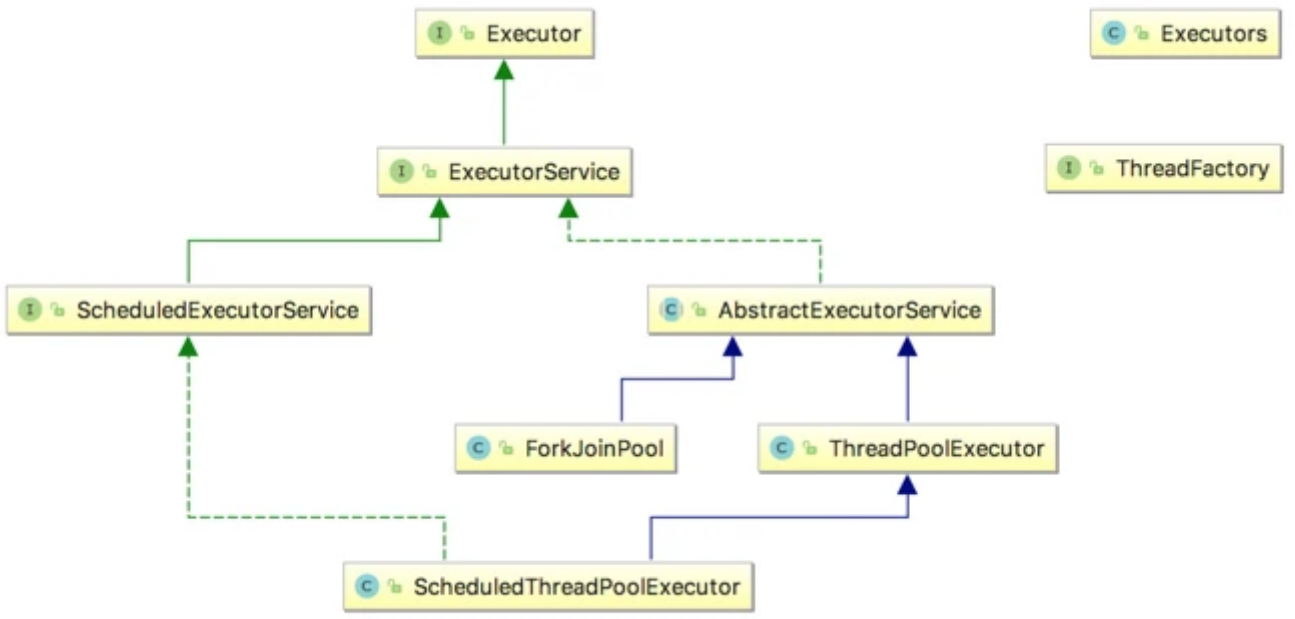
Fork/Join线程池是比较特殊的一类线程池，在JDK1.7时才引入，其核心实现就是`ForkJoinPool`类。关于Fork/Join框架，我们后面会专题讲解，现在只需要知道，Executors框架提供了一种创建该类线程池的便捷方法。

```
/**
 * 创建具有指定并行级别的ForkJoin线程池.
 */
public static ExecutorService newWorkStealingPool(int parallelism) {
    return new ForkJoinPool(parallelism, ForkJoinPool.defaultForkJoinWorkerThreadFactory, null, true);
}

/**
 * 创建并行级别等于CPU核心数的ForkJoin线程池.
 */
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),
ForkJoinPool.defaultForkJoinWorkerThreadFactory,
            null, true);
}
```

三、总结

至此，Executors框架的整体结构基本就讲解完了，此时我们的脑海中应有大致如下的一幅类继承图：



下面来回顾一下，上面的各个接口/类的关系和作用：

1. **Executor**
   执行器接口，也是最顶层的抽象核心接口， 分离了任务和任务的执行。
2. **ExecutorService**
   在Executor的基础上提供了执行器生命周期管理，任务异步执行等功能。
3. **ScheduledExecutorService**
   在ExecutorService基础上提供了任务的延迟执行/周期执行的功能。
4. **Executors**
   生产具体的执行器的静态工厂
5. **ThreadFactory**
   线程工厂，用于创建单个线程，减少手工创建线程的繁琐工作，同时能够复用工厂的特性。
6. **AbstractExecutorService**
   ExecutorService的抽象实现，为各类执行器类的实现提供基础。
7. **ThreadPoolExecutor**
   线程池Executor，也是最常用的Executor，可以以线程池的方式管理线程。
8. **ScheduledThreadPoolExecutor**
   在ThreadPoolExecutor基础上，增加了对周期任务调度的支持。
9. **ForkJoinPool**
   Fork/Join线程池，在JDK1.7时引入，时实现Fork/Join框架的核心类。

关于**ThreadPoolExecutor**和**ScheduledThreadPoolExecutor**，我们会在下一章详细讲解，帮助读者理解线程池的实现原理。至于**ForkJoinPool**，涉及Fork/Join这个并行框架的讲解，我们后面会专题介绍。

java　多线程　并发