线程池

一 使用线程池的好处

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个**线程池**还维护一些基本统计信息，例如已完成任务的数量。**使用线程池的好处**：

- **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度**。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

二 Executor 框架

2.1 简介

Executor 框架是 Java5 之后引进的，在 Java 5 之后，通过 Executor 来启动线程比使用 Thread 的 start 方法更好，除了更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免 this 逃逸问题。补充：this 逃逸是指在构造函数返回之前其他线程就持有该对象的引用. 调用尚未构造完全的对象的方法可能引发令人疑惑的错误。

2.2 Executor 框架结构(主要由三大部分组成)

1 任务。

执行任务需要实现的 **Runnable 接口**或 **Callable 接口**。**Runnable 接口**或 **Callable 接口**实现类都可以被 **ThreadPoolExecutor** 或 **ScheduledThreadPoolExecutor** 执行。

两者的区别：

Runnable 接口不会返回结果但是 **Callable 接口**可以返回结果。后面介绍 **Executors**

2 任务的执行

如下图所示，包括任务执行机制的**核心接口 Executor** ，以及继承自 Executor 接口的 **ExecutorService 接口**。**ScheduledThreadPoolExecutor** 和 **ThreadPoolExecutor** 这两个关键类实现了 **ExecutorService 接口**。

注意： 通过查看 ScheduledThreadPoolExecutor 源代码我们发现 ScheduledThreadPoolExecutor 实际上是继承了 ThreadPoolExecutor 并实现了 ScheduledExecutorService ，而 ScheduledExecutorService 又实现了 ExecutorService，正如我们下面给出的类关系图显示的一样。

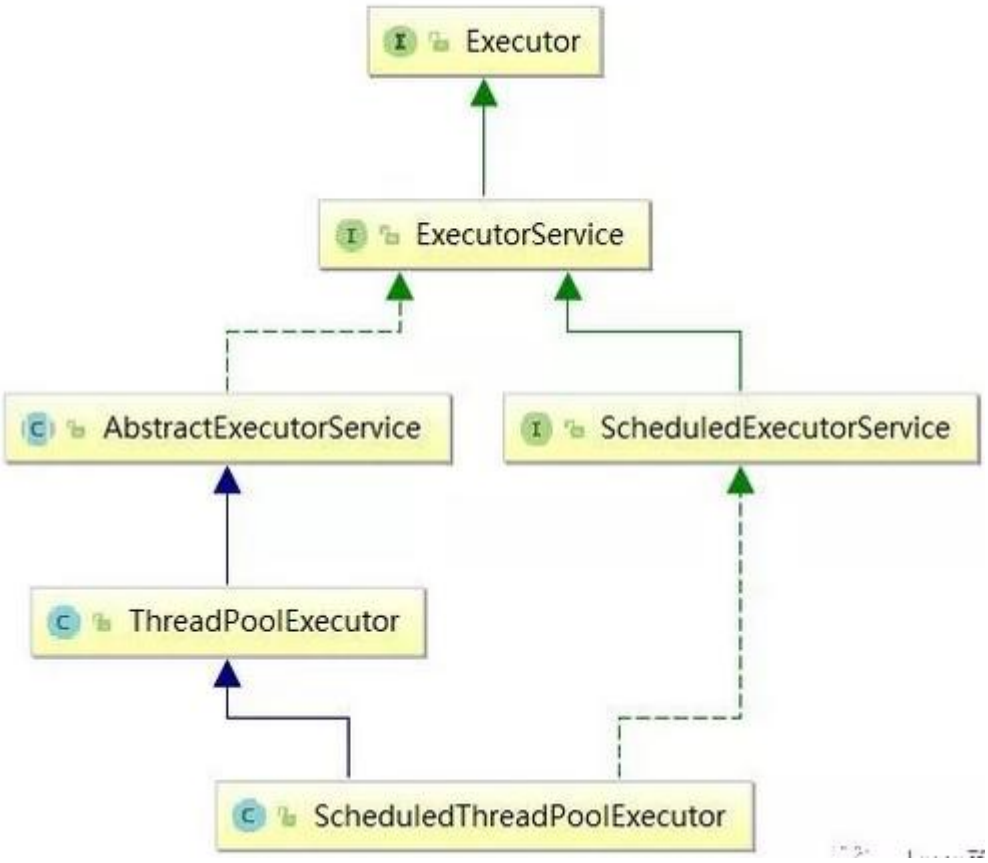
类的一些方法的时候会介绍到两者的相互转换。

ThreadPoolExecutor 类描述:

```
//AbstractExecutorService实现了ExecutorService接口
public class ThreadPoolExecutor extends AbstractExecutorService
```

ScheduledThreadPoolExecutor 类描述:

```
//ScheduledExecutorService实现了ExecutorService接口
public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService
```



3 异步计算的结果

Future 接口以及 Future 接口的实现类 **FutureTask 类**。 当我们把 **Runnable 接口**或 **Callable 接口**的实现类提交（调用 submit 方法）给 **ThreadPoolExecutor** 或 **ScheduledThreadPoolExecutor** 时，会返回一个 **FutureTask 对象**。

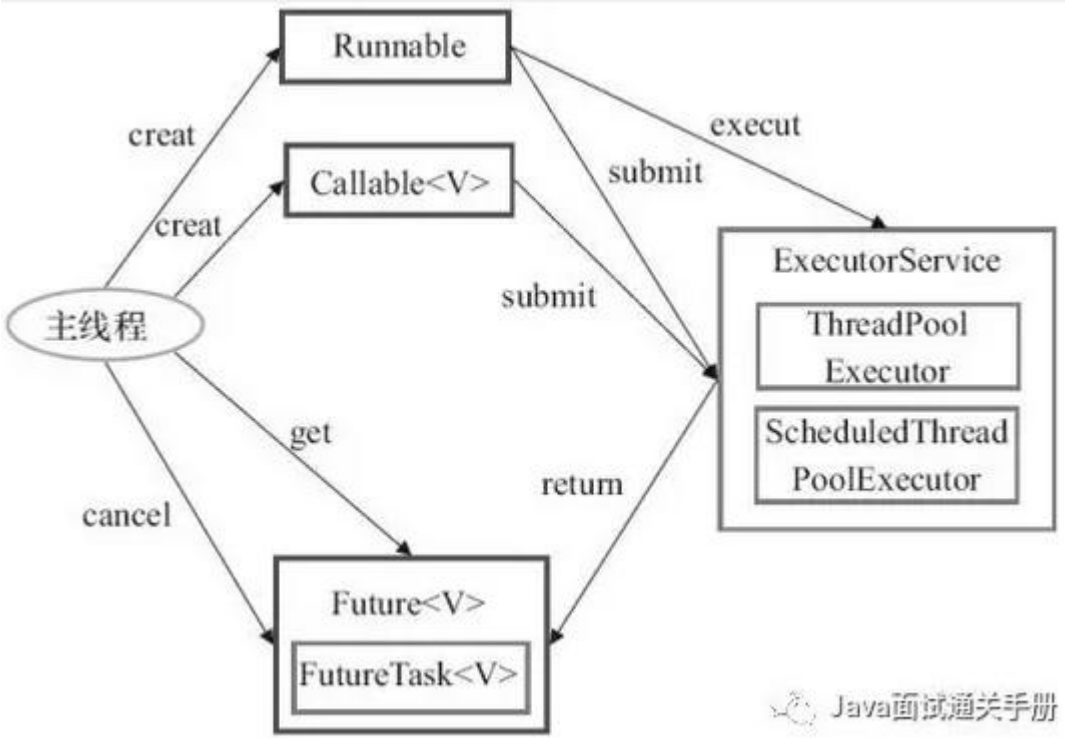
我们以 **AbstractExecutorService** 接口中的一个 submit 方法为例子来看看源代码：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

上面方法调用的 newTaskFor 方法返回了一个 FutureTask 对象。

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}
```

2.3 Executor 框架的使用示意图



1. 主线程首先要创建实现 Runnable 或者 Callable 接口的任务对象。 备注： 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。（Executors.callable（Runnable task）或 Executors.callable（Runnable task，Object resule））。

2. 然后可以把创建完成的 Runnable 对象直接交给 ExecutorService 执行（ExecutorService.execute（Runnable command））；或者也可以把 Runnable 对象或 Callable 对象提交给 ExecutorService 执行（ExecutorService.submit（Runnable task）或 ExecutorService.submit（Callable task））。

执行 execute()方法和 submit()方法的区别是什么呢？ 1)**execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；** 2)**submit()方法用于提交需要返回值的任务。线程池会返回一个 future 类型的对象，通过这个 future 对象可以判断任务是否执行成功**，并且可以通过 future 的 get()方法来获取

返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

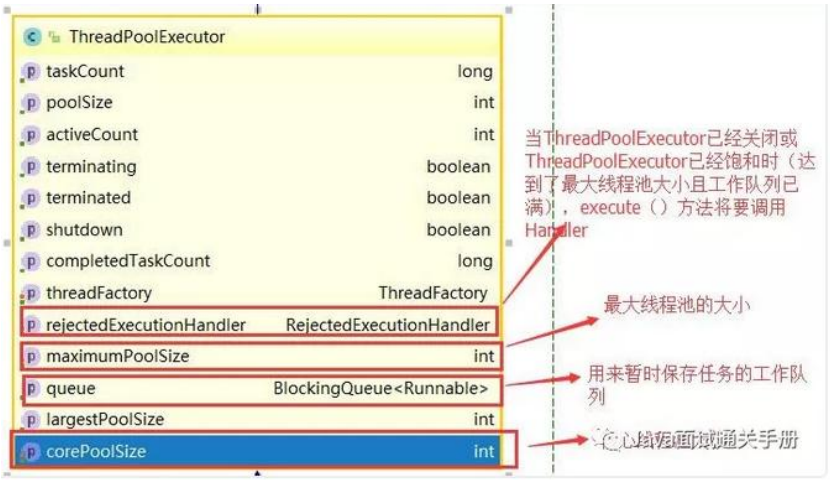
3. 如果执行 ExecutorService.submit（...），ExecutorService 将返回一个实现 Future 接口的对象（我们刚刚也提到过了执行 execute()方法和 submit()方法的区别，到目前为止的 JDK 中，返回的是 FutureTask 对象）。由于 FutureTask 实现了 Runnable，程序员也可以创建 FutureTask，然后直接交给 ExecutorService 执行。

4. 最后，主线程可以执行 FutureTask.get()方法来等待任务执行完成。主线程也可以执行 FutureTask.cancel（boolean mayInterruptIfRunning）来取消此任务的执行。

三 ThreadPoolExecutor 详解

线程池实现类 ThreadPoolExecutor 是 Executor 框架最核心的类，先来看一下这个类中比较重要的四个属性

3.1 ThreadPoolExecutor 类的四个比较重要的属性



3.2 ThreadPoolExecutor 类中提供的四个构造方法

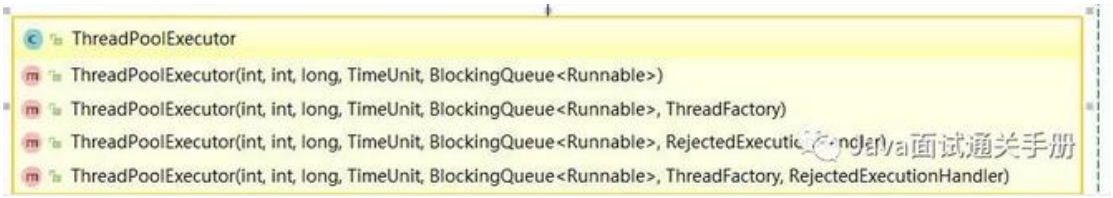
我们看最长的那个，其余三个都是在这个构造方法的基础上产生（给定某些默认参数的构造方法）

```
    /**
     * 用给定的初始参数创建一个新的ThreadPoolExecutor。
     * @param keepAliveTime 当线程池中的线程数量大于corePoolSize的时候，如果这时没有新的任务提交，
     *核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime；
     * @param unit  keepAliveTime参数的时间单位
     * @param workQueue 等待队列，当任务提交时，如果线程池中的线程数量大于等于corePoolSize的时候，把该任务封装成一个Worker对象放入等待队列；
     *
     * @param threadFactory 执行者创建新线程时使用的工厂
     * @param handler RejectedExecutionHandler类型的变量，表示线程池的饱和策略。
     * 如果阻塞队列满了并且没有空闲的线程，这时如果继续提交任务，就需要采取一种策略处理该任务。
     * 线程池提供了4种策略：
        1.AbortPolicy：直接抛出异常，这是默认策略；
        2.CallerRunsPolicy：用调用者所在的线程来执行任务；
        3.DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
        4.DiscardPolicy：直接丢弃任务；
     */
```

```
  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;
  }
```

3.3 如何创建 ThreadPoolExecutor

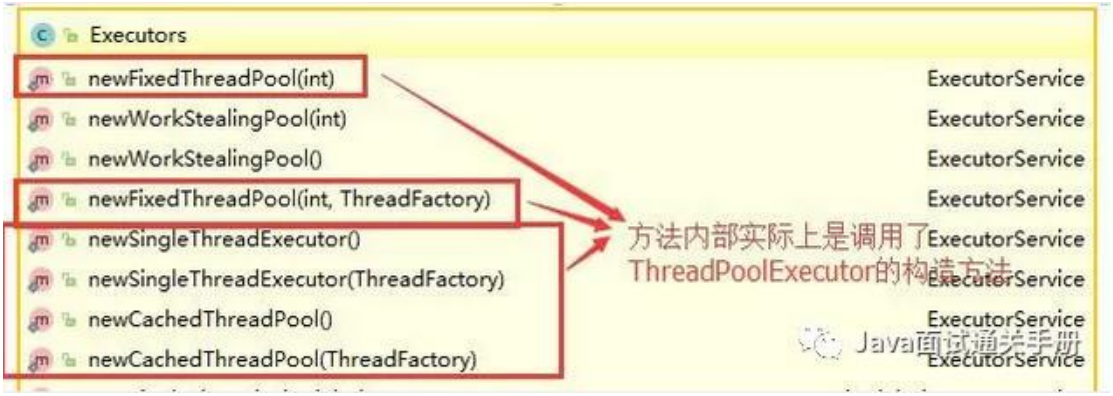
方式一：通过构造方法实现（官方 API 文档并不推荐，所以建议使用第二种方式）



方式二：通过 Executor 框架的工具类 Executors 来实现 我们可以创建三种类型的 ThreadPoolExecutor：

- **FixedThreadPool**
- **SingleThreadExecutor**
- **CachedThreadPool**

对应 Executors 工具类中的方法如图所示：



3.4 FixedThreadPool 详解

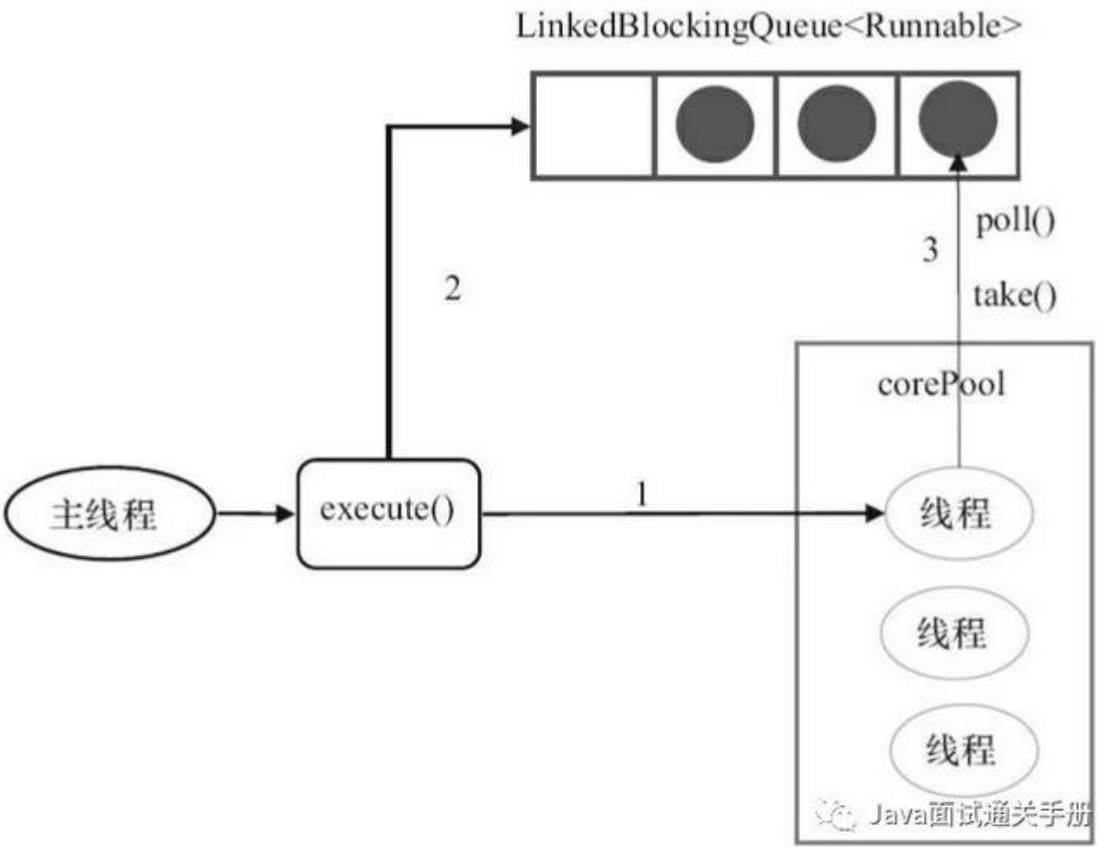
FixedThreadPool 被称为可重用固定线程数的线程池。通过 Executors 类中的相关源代码来看一下相关实现：

```
/**
  * 创建一个可重用固定数量线程的线程池
  *在任何时候至多有n个线程处于活动状态
  *如果在所有线程处于活动状态时提交其他任务，则它们将在队列中等待，
  *直到线程可用。 如果任何线程在关闭之前的执行期间由于失败而终止，
  *如果需要执行后续任务，则一个新的线程将取代它。池中的线程将一直存在
  *知道调用shutdown方法
  * @param nThreads 线程池中的线程数
  * @param threadFactory 创建新线程时使用的factory
  * @return 新创建的线程池
  * @throws NullPointerException 如果threadFactory为null
  * @throws IllegalArgumentException if {@code nThreads <= 0}
  */
 publ  stat  ExecutorSer  newFixedThread in  nThread ThreadFact  threadFactor
 ic    ic    vice        Pool(           t  s,       ory        y) {
      return new ThreadPoolExecutor(nThreads, nThreads,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>(),
                                    threadFactory);
  }
```

另外还有一个 FixedThreadPool 的实现方法，和上面的类似，所以这里不多做阐述：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

从上面源代码可以看出新创建的 FixedThreadPool 的 corePoolSize 和 maximumPoolSize 都被设置为 nThreads。 **FixedThreadPool 的 execute()方法运行示意图（该图片来源：《Java 并发编程的艺术》）：**



上图说明：

1. 如果当前运行的线程数小于 corePoolSize，则创建新的线程来执行任务；
2. 当前运行的线程数等于 corePoolSize 后，将任务加入 LinkedBlockingQueue；
3. 线程执行完 1 中的任务后，会在循环中反复从 LinkedBlockingQueue 中获取任务来执行；

FixedThreadPool 使用无界队列 LinkedBlockingQueue（队列的容量为 Intger.MAX_VALUE）作为线程池的工作队列会对线程池带来如下影响：

1. 当线程池中的线程数达到 corePoolSize 后，新任务将在无界队列中等待，因此线程池中的线程数不会超过 corePoolSize；
2. 由于 1，使用无界队列时 maximumPoolSize 将是一个无效参数；
3. 由于 1 和 2，使用无界队列时 keepAliveTime 将是一个无效参数；

4. 运行中的 FixedThreadPool（未执行 shutdown()或 shutdownNow()方法）不会拒绝任务

3.5 SingleThreadExecutor 详解

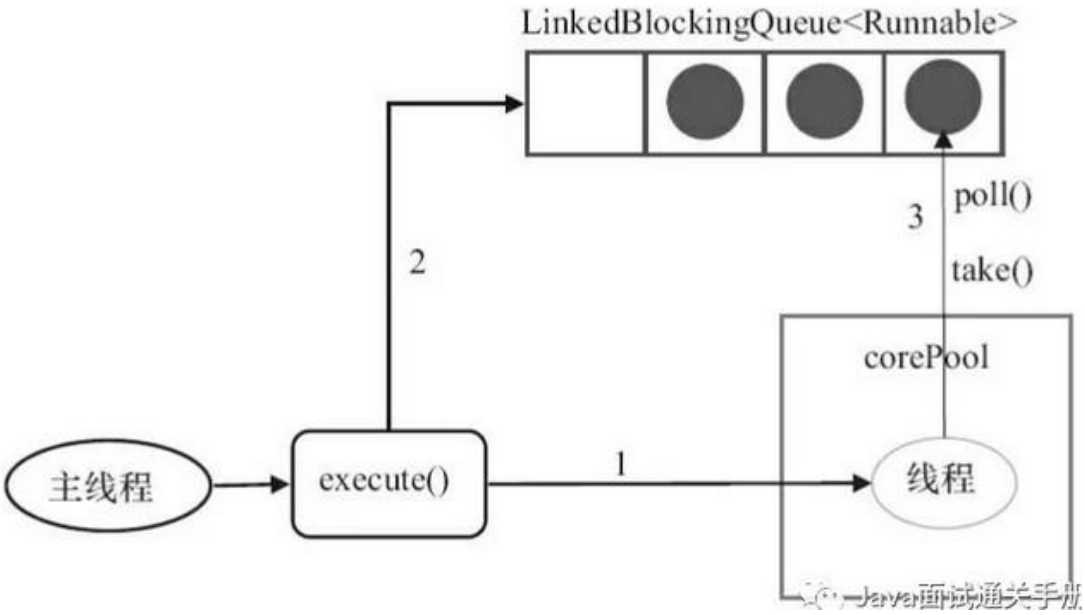
SingleThreadExecutor 是使用单个 worker 线程的 Executor。下面看看 **SingleThreadExecutor 的实现：**

```
/**
  *创建使用单个worker线程运行无界队列的Executor
  *并使用提供的ThreadFactory在需要时创建新线程
  *
  * @param threadFactory 创建新线程时使用的factory
  *
  * @return 新创建的单线程Executor
  * @throws NullPointerException 如果ThreadFactory为空
  */
 publi stati ExecutorServi newSingleThreadExecu ThreadFacto threadFactory
 c     c     ce           tor(                  ry         ) {
      return new FinalizableDelegatedExecutorService
          (new ThreadPoolExecutor(1, 1,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory));
  }

public static ExecutorService newSingleThreadExecutor() {
      return new FinalizableDelegatedExecutorService
          (new ThreadPoolExecutor(1, 1,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>()));
  }
```

从上面源代码可以看出新创建的 SingleThreadExecutor 的 corePoolSize 和 maximumPoolSize 都被设置为 1.其他参数和 FixedThreadPool 相同。SingleThreadExecutor 使用无界队列 LinkedBlockingQueue 作为线程池的工作队列（队列的容量为 Intger.MAX_VALUE）。SingleThreadExecutor 使用无界队列作为线程池的工作队列会对线程池带来的影响与 FixedThreadPool 相同。

SingleThreadExecutor 的运行示意图（该图片来源：《Java 并发编程的艺术》）：



上图说明;

1. 如果当前运行的线程数少于 corePoolSize，则创建一个新的线程执行任务；
2. 当前线程池中有一个运行的线程后，将任务加入 LinkedBlockingQueue
3. 线程执行完 1 中的任务后，会在循环中反复从 LinkedBlockingQueue 中获取任务来执行；

3.6 CachedThreadPool 详解

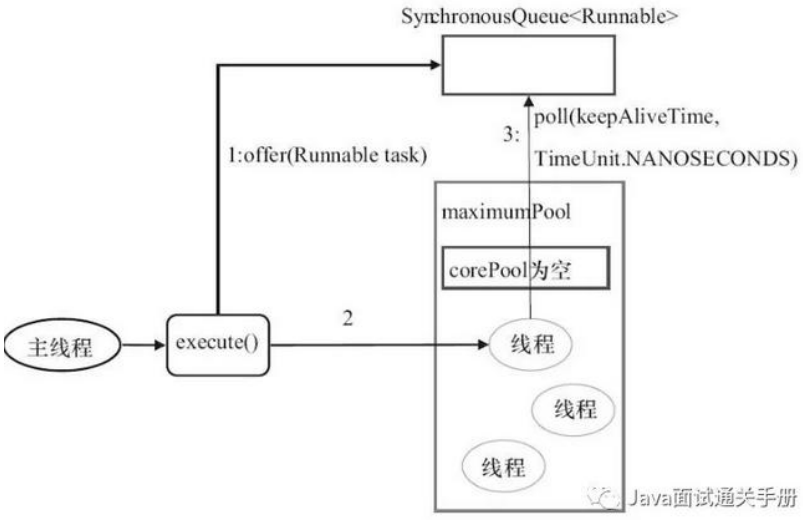
CachedThreadPool 是一个会根据需要创建新线程的线程池。下面通过源码来看看 CachedThreadPool 的实现：

```
/**
 * 创建一个线程池，根据需要创建新线程，但会在先前构建的线程可用时重用它，
 *并在需要时使用提供的ThreadFactory创建新线程。
 * @param threadFactory 创建新线程使用的factory
 * @return 新创建的线程池
 * @throws NullPointerException 如果threadFactory为空
 */
publi  stati  ExecutorServi  newCachedThreadPooThreadFacto  threadFactory)
c      c      ce            l(                  ry           {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>(),
                                  threadFactory);
}

public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

CachedThreadPool 的 corePoolSize 被设置为空（0），maximumPoolSize 被设置为 Integer.MAX.VALUE，即它是无界的，这也就意味着如果主线程提交任务的速度高于 maximumPool 中线程处理任务的速度时，CachedThreadPool 会不断创建新的线程。极端情况下，这样会导致耗尽 cpu 和内存资源。

CachedThreadPool 的 execute()方法的执行示意图（该图片来源：《Java 并发编程的艺术》）：



上图说明：

1. 首先执行 SynchronousQueue.offer(Runnable task)。如果当前 maximumPool 中有闲线程正在执行 SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)，那么主线程执行 offer 操作与空闲线程执行的 poll 操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成，否则执行下面的步骤 2；
2. 当初始 maximumPool 为空，或者 maximumPool 中没有空闲线程时，将没有线程执行 SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)。这种情况下，步骤 1 将失败，此时 CachedThreadPool 会创建新线程执行任务，execute 方法执行完成；

3.7 ThreadPoolExecutor 使用示例

3.7.1 示例代码

首先创建一个 Runnable 接口的实现类（当然也可以是 Callable 接口，我们上面也说了两者的区别是：Runnable 接口不会返回结果但是 Callable 接口可以返回结果。后面介绍 Executors 类的一些方法的时候会介绍到两者的相互转换。）

```java
import java.util.Date;

/**
 * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。
 */
public class WorkerThread implements Runnable {

    private String command;

    public WorkerThread(String s) {
        this.command = s;
    }

    @Override
    public void run() {
       Syst .out.print Thre .currentThread().getNam " Start. Time  +ne  Dat ())
       em   ln(        ad   e() +                    = "            w   e   ;
        processCommand();
       Syste.out.print Threa.currentThread().getName" End. Time   +ne  Dat ())
       m    ln(        d    () +                    = "            w   e   ;
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString() {
        return this.command;
    }
}
```

编写测试程序，我们这里以 FixedThreadPool 为例子

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExecutorDemo {

    public static void main(String[] args) {
        //创建一个FixedThreadPool对象
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            //创建WorkerThread对象（WorkerThread类实现了Runnable 接口）
            Runnable worker = new WorkerThread("" + i);
            //执行Runnable
            executor.execute(worker);
        }
        //终止线程池
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}
```

输出示例：

```
pool-1-thread-5 Start. Time = Thu May 31 10:22:52 CST 2018
pool-1-thread-3 Start. Time = Thu May 31 10:22:52 CST 2018
pool-1-thread-2 Start. Time = Thu May 31 10:22:52 CST 2018
pool-1-thread-4 Start. Time = Thu May 31 10:22:52 CST 2018
pool-1-thread-1 Start. Time = Thu May 31 10:22:52 CST 2018
pool-1-thread-4 End. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-1 End. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-2 End. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-5 End. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-3 End. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-5 Start. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-2 Start. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-1 Start. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-4 Start. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-3 Start. Time = Thu May 31 10:22:57 CST 2018
pool-1-thread-5 End. Time = Thu May 31 10:23:02 CST 2018
pool-1-thread-1 End. Time = Thu May 31 10:23:02 CST 2018
pool-1-thread-2 End. Time = Thu May 31 10:23:02 CST 2018
pool-1-thread-3 End. Time = Thu May 31 10:23:02 CST 2018
pool-1-thread-4 End. Time = Thu May 31 10:23:02 CST 2018
Finished all threads
```

3.7.2 shutdown（ ）VS shutdownNow（ ）

shutdown（ ）方法表明关闭已在 Executor 上调用，因此不会再向 DelayedPool 添加任何其他任务（由 ScheduledThreadPoolExecutor 类在内部使用）。 但是，已经在队列中提交的任务将被允许完成。 另一方面，shutdownNow（ ）方法试图终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

3.7.3 isTerminated() Vs isShutdown()

isShutdown( )表示执行程序正在关闭，但并非所有任务都已完成执行。 另一方面，isShutdown（ ）表示所有线程都已完成执行。

四 ScheduledThreadPoolExecutor 详解

4.1 简介

ScheduledThreadPoolExecutor 主要用来在给定的延迟后运行任务，或者定期执行任务。

ScheduledThreadPoolExecutor 使用的任务队列 DelayQueue 封装了一个 PriorityQueue，PriorityQueue 会对队列中的任务进行排序，执行所需时间短的放在前面先被执行(ScheduledFutureTask 的 time 变量小的先执行)，如果执行所需时间相同则先提交的任务将被先执行(ScheduledFutureTask 的 squenceNumber 变量小的先执行)。

ScheduledThreadPoolExecutor 和 Timer 的比较：

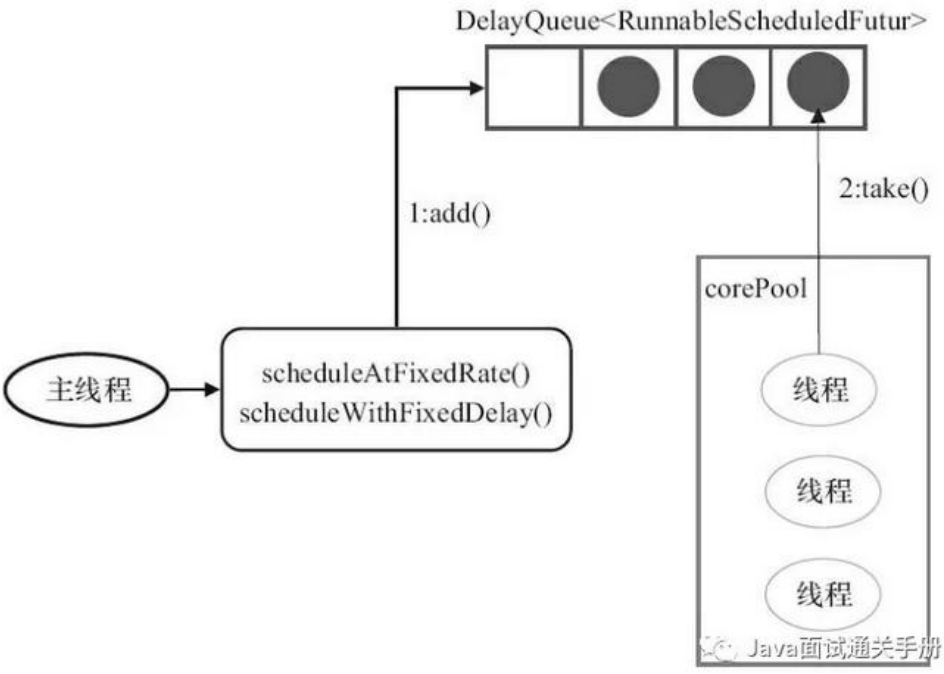
- Timer 对系统时钟的变化敏感，ScheduledThreadPoolExecutor 不是；

- Timer 只有一个执行线程，因此长时间运行的任务可以延迟其他任务。ScheduledThreadPoolExecutor 可以配置任意数量的线程。 此外，如果你想（通过提供 ThreadFactory），你可以完全控制创建的线程;
- 在 TimerTask 中抛出的运行时异常会杀死一个线程，从而导致 Timer 死机:-(...即计划任务将不再运行。ScheduledThreadExecutor 不仅捕获运行时异常，还允许您在需要时处理它们（通过重写 afterExecute 方法 ThreadPoolExecutor）。抛出异常的任务将被取消，但其他任务将继续运行。

综上，在 JDK1.5 之后，你没有理由再使用 Timer 进行任务调度了。

备注： Quartz 是一个由 java 编写的任务调度库，由 OpenSymphony 组织开源出来。在实际项目开发中使用 Quartz 的还是居多，比较推荐使用 Quartz。因为 Quartz 理论上能够同时对上万个任务进行调度，拥有丰富的功能特性，包括任务调度、任务持久化、可集群化、插件等等。

4.2 ScheduledThreadPoolExecutor 运行机制



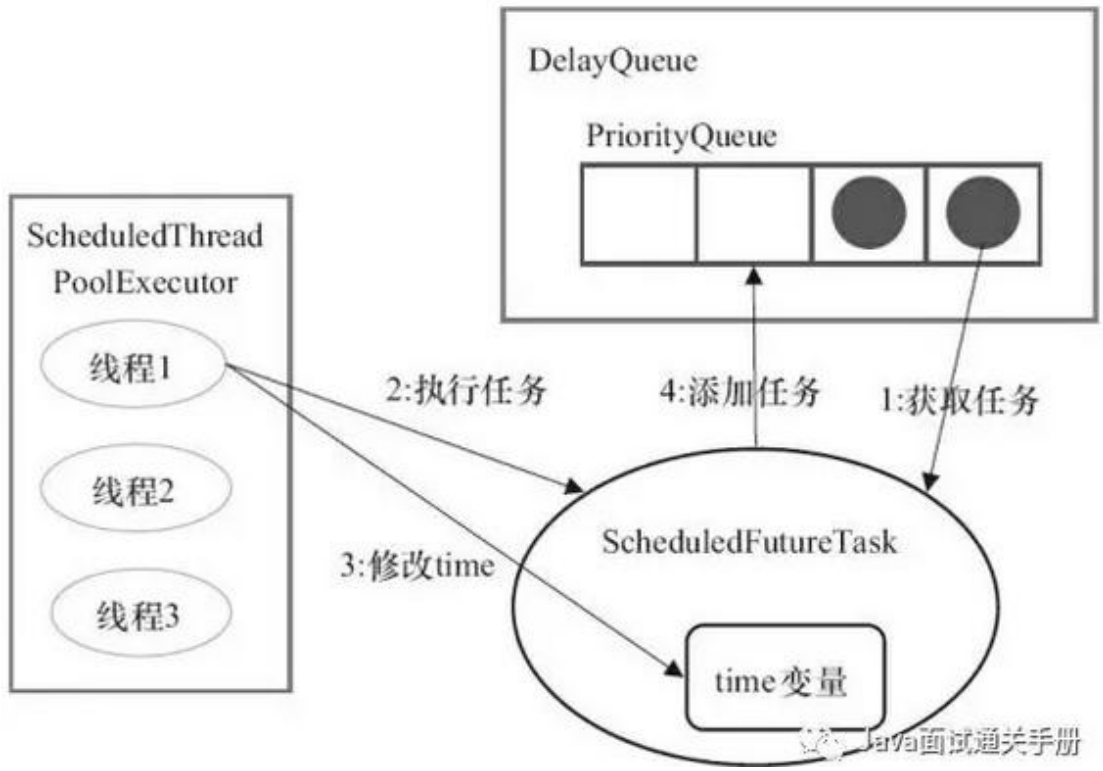
ScheduledThreadPoolExecutor 的执行主要分为两大部分：

1. 当调用 ScheduledThreadPoolExecutor 的 **scheduleAtFixedRate()** 方法或者 **scheduleWirhFixedDelay()** 方法时，会向 ScheduledThreadPoolExecutor 的 **DelayQueue** 添加一个实现了 **RunnableScheduledFutur** 接口的 **ScheduledFutureTask** 。
2. 线程池中的线程从 DelayQueue 中获取 ScheduledFutureTask，然后执行任务。

ScheduledThreadPoolExecutor 为了实现周期性的执行任务，对 ThreadPoolExecutor 做了如下修改：

- 使用 **DelayQueue** 作为任务队列；
- 获取任务的方不同
- 执行周期任务后，增加了额外的处理

4.3 ScheduledThreadPoolExecutor 执行周期任务的步骤



1. 线程 1 从 DelayQueue 中获取已到期的 ScheduledFutureTask（DelayQueue.take()）。到期任务是指 ScheduledFutureTask 的 time 大于等于当前系统的时间；
2. 线程 1 执行这个 ScheduledFutureTask；
3. 线程 1 修改 ScheduledFutureTask 的 time 变量为下次将要被执行的时间；
4. 线程 1 把这个修改 time 之后的 ScheduledFutureTask 放回 DelayQueue 中（DelayQueue.add()）。

4.4 ScheduledThreadPoolExecutor 使用示例

1. 创建一个简单的实现 Runnable 接口的类（我们上面的例子已经实现过）
2. 测试程序使用 ScheduledExecutorService 和 ScheduledThreadPoolExecutor 实现的 java 调度

```
/**
 * 使用ScheduledExecutorService和ScheduledThreadPoolExecutor实现的java调度程序示例程序。
 */
public class ScheduledThreadPoolDemo {

    public static void main(String[] args) throws InterruptedException {

        //创建一个ScheduledThreadPoolExecutor对象
       ScheduledExecutorSer scheduledThreadPooExecuto.newScheduledThreadP5)
       vice                 l =               rs     ool(                ;
        //计划在某段时间后运行
        System.out.println("Current Time = "+new Date());
        for(int i=0; i<3; i++){
            Thread.sleep(1000);
            WorkerThread worker = new WorkerThread("do heavy processing");
            //创建并执行在给定延迟后启用的单次操作。
            scheduledThreadPool.schedule(worker, 10, TimeUnit.SECONDS);
        }

        //添加一些延迟让调度程序产生一些线程
        Thread.sleep(30000);
        System.out.println("Current Time = "+new Date());
        //关闭线程池
        scheduledThreadPool.shutdown();
        while(!scheduledThreadPool.isTerminated()){
            //等待所有任务完成
        }
        System.out.println("Finished all threads");
    }

}
```

运行结果：

```
Current Time = Wed May 30 17:11:16 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:11:27 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:11:28 CST 2018
pool-1-thread-3 Start. Time = Wed May 30 17:11:29 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:11:32 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:11:33 CST 2018
pool-1-thread-3 End. Time = Wed May 30 17:11:34 CST 2018
Current Time = Wed May 30 17:11:49 CST 2018
Finished all threads
```

4.4.1 ScheduledExecutorService scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit)方法

我们可以使用 ScheduledExecutorService scheduleAtFixedRate 方法来安排任务在初始延迟后运行，然后在给定的时间段内运行。

时间段是从池中第一个线程的开始，因此如果您将 period 指定为 1 秒并且线程运行 5 秒，那么只要第一个工作线程完成执行，下一个线程就会开始执行。

```
for (int i = 0; i < 3; i++) {
    Thread.sleep(1000);
    WorkerThread worker = new WorkerThread("do heavy processing");
    // schedule task to execute at fixed rate
    scheduledThreadPool.scheduleAtFixedRate(worker, 0, 10,
    TimeUnit.SECONDS);
}
```

输出示例:

```
Current Time = Wed May 30 17:47:09 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:47:10 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:47:11 CST 2018
pool-1-thread-3 Start. Time = Wed May 30 17:47:12 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:47:15 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:47:16 CST 2018
pool-1-thread-3 End. Time = Wed May 30 17:47:17 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:47:20 CST 2018
pool-1-thread-4 Start. Time = Wed May 30 17:47:21 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:47:22 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:47:25 CST 2018
pool-1-thread-4 End. Time = Wed May 30 17:47:26 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:47:27 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:47:30 CST 2018
pool-1-thread-3 Start. Time = Wed May 30 17:47:31 CST 2018
pool-1-thread-5 Start. Time = Wed May 30 17:47:32 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:47:35 CST 2018
pool-1-thread-3 End. Time = Wed May 30 17:47:36 CST 2018
pool-1-thread-5 End. Time = Wed May 30 17:47:37 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:47:40 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:47:41 CST 2018
Current Time = Wed May 30 17:47:42 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:47:45 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:47:46 CST 2018
Finished all threads

Process finished with exit code 0
```

4.4.2 ScheduledExecutorService scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit)方法

ScheduledExecutorService scheduleWithFixedDelay 方法可用于以初始延迟启动周期性执行，然后以给定延迟执行。 延迟时间是线程完成执行的时间。

```
for (int i = 0; i < 3; i++) {
    Thread.sleep(1000);
    WorkerThread worker = new WorkerThread("do heavy processing");
    scheduledThreadPool.scheduleWithFixedDelay(worker, 0, 1,
    TimeUnit.SECONDS);
}
```

输出示例：

```
Current Time = Wed May 30 17:58:09 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:58:10 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:11 CST 2018
pool-1-thread-3 Start. Time = Wed May 30 17:58:12 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:58:15 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:16 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:58:16 CST 2018
pool-1-thread-3 End. Time = Wed May 30 17:58:17 CST 2018
pool-1-thread-4 Start. Time = Wed May 30 17:58:17 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:18 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:58:21 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:58:22 CST 2018
pool-1-thread-4 End. Time = Wed May 30 17:58:22 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:23 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:23 CST 2018
pool-1-thread-4 Start. Time = Wed May 30 17:58:24 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:58:27 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:28 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:58:28 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:29 CST 2018
pool-1-thread-4 End. Time = Wed May 30 17:58:29 CST 2018
pool-1-thread-4 Start. Time = Wed May 30 17:58:30 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:58:33 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:34 CST 2018
pool-1-thread-1 Start. Time = Wed May 30 17:58:34 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:35 CST 2018
pool-1-thread-4 End. Time = Wed May 30 17:58:35 CST 2018
pool-1-thread-4 Start. Time = Wed May 30 17:58:36 CST 2018
pool-1-thread-1 End. Time = Wed May 30 17:58:39 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:40 CST 2018
pool-1-thread-5 Start. Time = Wed May 30 17:58:40 CST 2018
pool-1-thread-4 End. Time = Wed May 30 17:58:41 CST 2018
pool-1-thread-2 Start. Time = Wed May 30 17:58:41 CST 2018
Current Time = Wed May 30 17:58:42 CST 2018
pool-1-thread-5 End. Time = Wed May 30 17:58:45 CST 2018
pool-1-thread-2 End. Time = Wed May 30 17:58:46 CST 2018
Finished all threads
```

4.4.3 scheduleWithFixedDelay() vs scheduleAtFixedRate()

scheduleAtFixedRate（...）将延迟视为两个任务开始之间的差异（即定期调用）scheduleWithFixedDelay(...)将延迟视为一个任务结束与下一个任务开始之间的差异

scheduleAtFixedRate(): 创建并执行在给定的初始延迟之后，随后以给定的时间段首先启用的周期性动作；那就是执行将在 initialDelay 之后开始，然后 initialDelay+period ，然后是 initialDelay + 2 * period ，等等。 如果任务的执行遇到异常，则后续的执行被抑制。 否则，任务将仅通过取消或终止执行人终止。如果任务执行时间比其周期长，则后续执行可能会迟到，但不会同时执行。 **scheduleWithFixedDelay()：**创建并执行在给定的初始延迟之后首先启用的定期动作，随后在一个执行的终止和下一个执行的开始之间给定的延迟。 如果任务的执行遇到异常，则后续的执行被抑制。 否则，任务将仅通过取消或终止执行终止。

五 各种线程池的适用场景介绍

FixedThreadPool： 适用于为了满足资源管理需求，而需要限制当前线程数量的应用场景。它适用于负载比较重的服务器；

SingleThreadExecutor： 适用于需要保证顺序地执行各个任务并且在任意时间点，不会有多个线程是活动的应用场景。

CachedThreadPool： 适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器；

ScheduledThreadPoolExecutor： 适用于需要多个后台执行周期任务，同时为了满足资源管理需求而需要限制后台线程的数量的应用场景，

SingleThreadScheduledExecutor：适用于需要单个后台线程执行周期任务，同时保证顺序地执行各个任务的应用场景。