(7)线程池

1.线程池简介

Java中的线程池是运用场景嘴都的并发框架，几乎手游需要异步或者并发执行任务的程序都可以使用线程池。开发中使用线程池的三个优点：

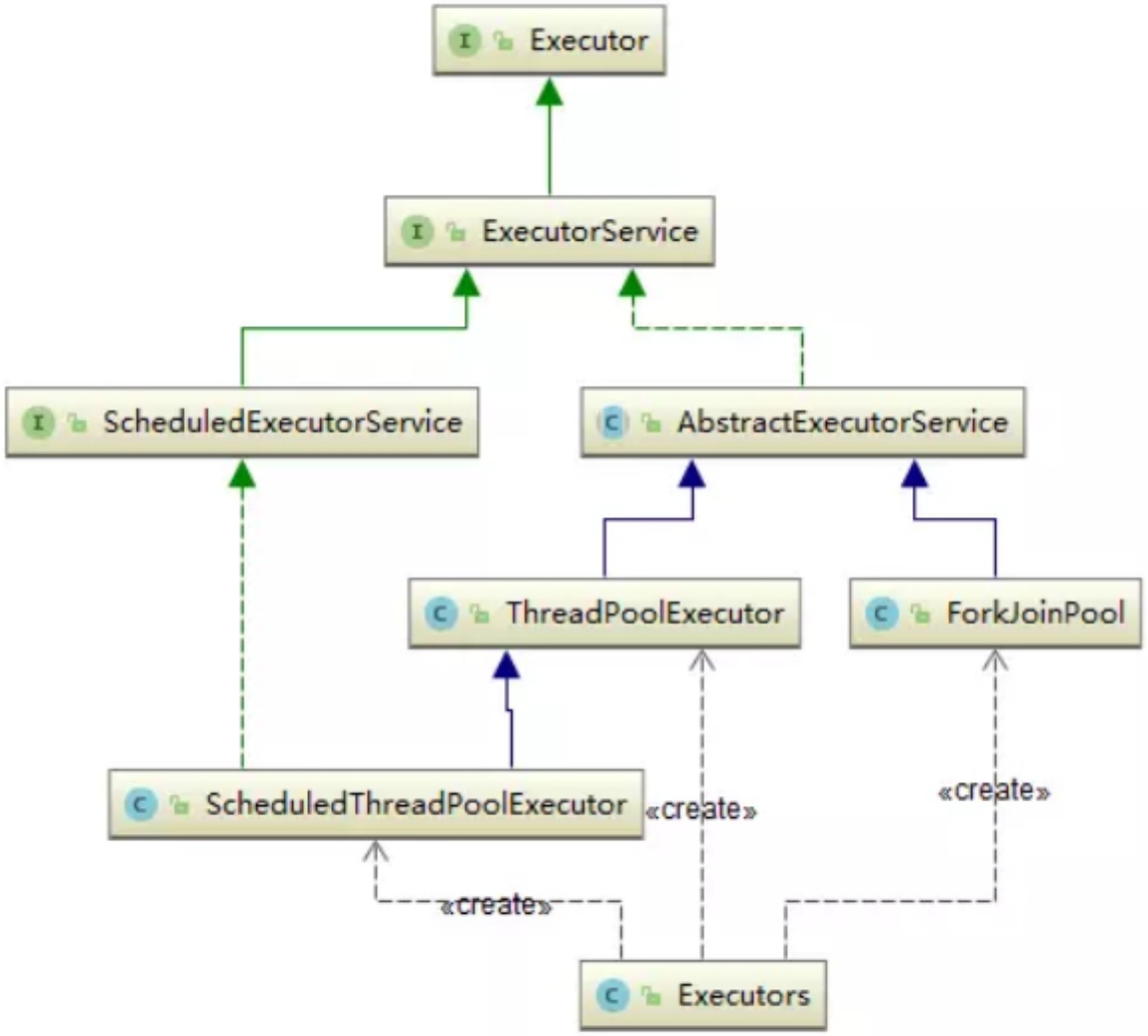
- Executor :线程池的**最上层接口**，提供了任务提交的基础方法。
- ExecutorService:提供了线程池管理的**上层接口**，如池的销毁，异步任务的提交
-
- ScheduleExecutorService:提供任务定时或周期执行方法的**ExectorService**。
- AbstractExecutorSerivice:为ExectorSerive的任务提交方法提供了默认实现
- ThreadPoolExecutor：大名鼎鼎的**线程类**，提供线程和任务调度的策略。
- ScheduledthreadPoolExecutor：属于**线程池的一种**，它可以允许任务延迟或周期执行，类似java的Timer。
- ForkJoinPool:JDK1.7假如的成员，也是**线程池的一种**。只允许执行ForkJoinTask任务，它是为那些能够被帝国地拆解成子任务地工作类型量身设计的，其目的在于能够使用所有可用的运算资源来提升应用性能。
- Executors:**创建各种线程池的工具类**

线程池分为三种：**基础线程池ThreadPoolExecutor**、**延时任务线程池ScheduledThreadPoolExecutor** 和**分治线程池ForkJoinPool**。每种线程池中都有其支持的任务类型

1.2线程池优点

1.**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带来的消耗。
2.**提高响应速度**：当任务到达时，任务可以不需要等待线程的创建就能立即执行。
3.**提高线程的客观理性**：使用线程池可以统一分配，调度和监控。

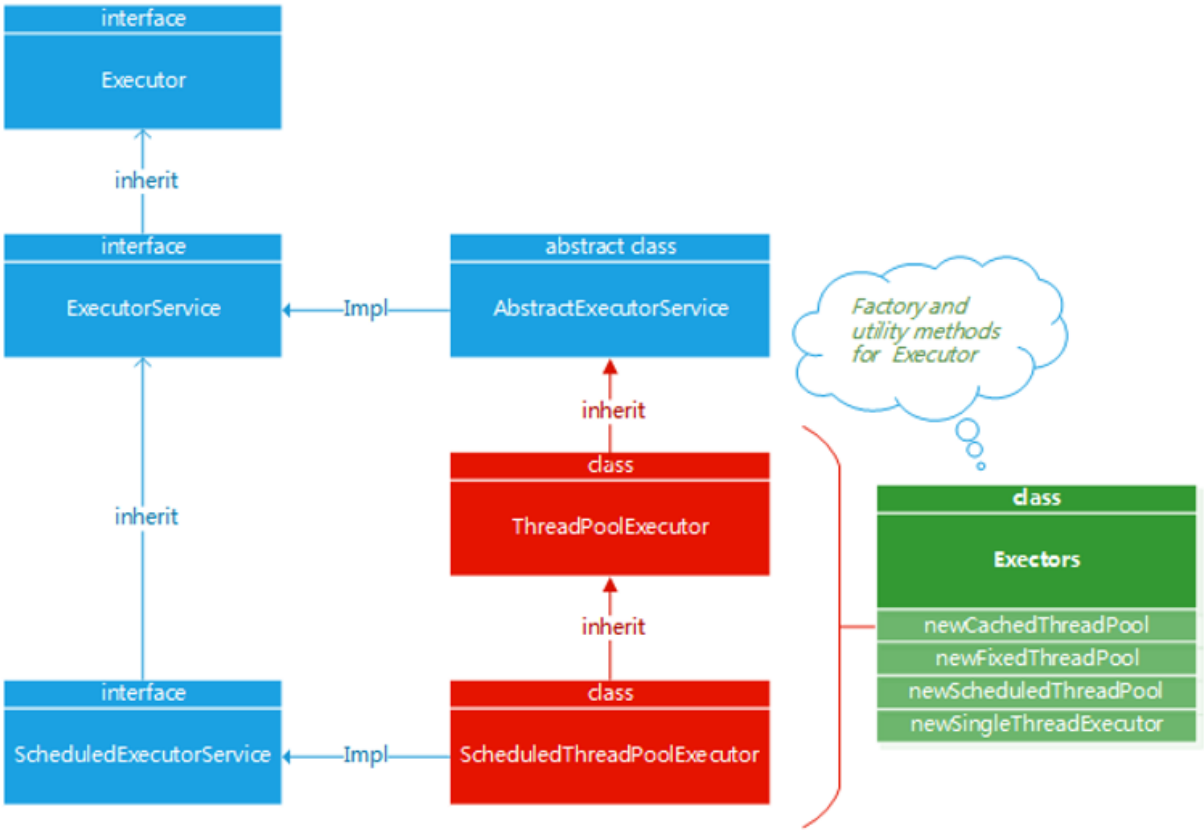
1.3线程池核心框图



线程池框架结构

inherit：继承

Impl:实现



2.线程池框架分析

2.1三大接口分析

2.1.1顶层父接口Executor

线程池的最上层接口，提供了任务提交的基础方法。

核心方法： void **execute**(Runnable command);

```
1  public interface Executor {
2
3   /**
4   * Executes the given command at some time in the future. The command
5   * may execute in a new thread, in a pooled thread, or in the calling
6   * thread, at the discretion of the {@code Executor} implementation.
7   *
8   * @param command the runnable task
9   * @throws RejectedExecutionException if this task cannot be
10   * accepted for execution
```

```
 * @throws NullPointerException if command is null
 */
 void execute(Runnable command);
}
```

总结：

> void execute(Runnable command);：**提交任务的基础方法**
> execute：**只接受Runnable和对象实例**

2.1.2普通调度池的核心接口 ExecutorService（继承Exector）

ExecutorService:提供了线程池管理的**上层接口**，如池的销毁，异步任务的提交

核心方法：**submit**

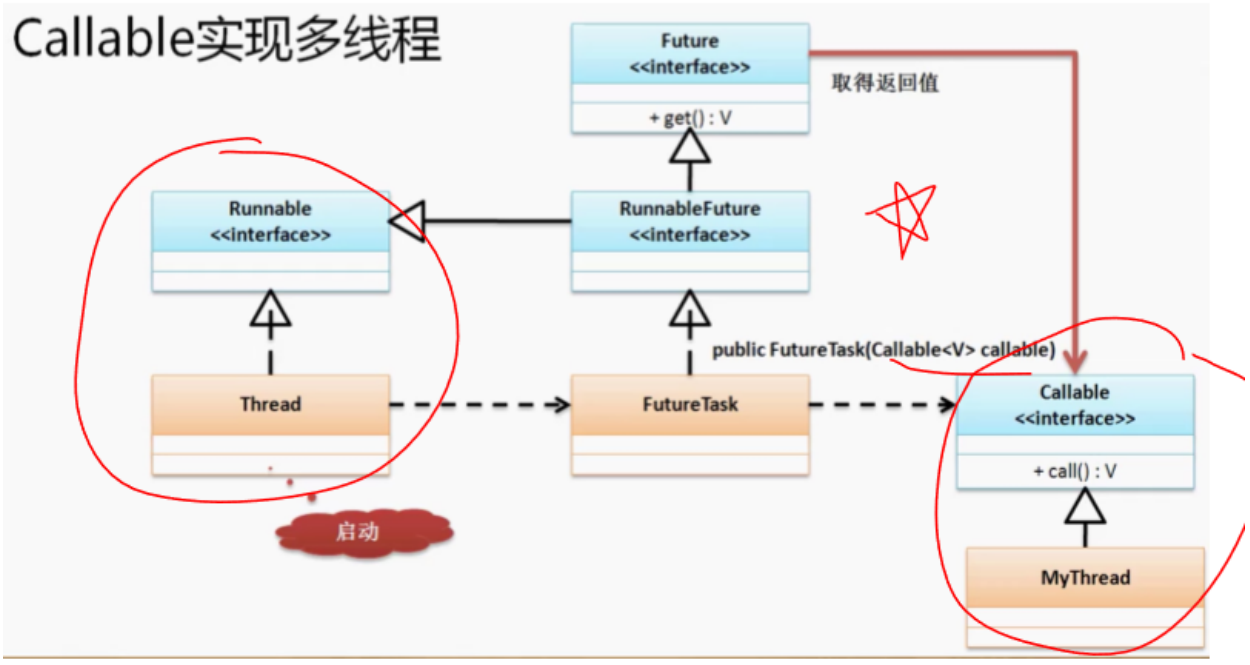
```
public interface ExecutorService extends Executor {
 ....................................
 <T> Future<T> submit(Callable<T> task/Runnable task, T result);
}
```

总结：<T> Future<T> submit(Callable<T> task/Runnable task, T result):

> 返回值：Future<T> 类型
> 可以接受Runnable和Callable的实例对象。

我们来回忆一下实创建3个线程的3个种方式。

即利用返回的Future接口可以调用其get（）方法获得Callable创建的线程地返回值。



call方法

```
V call() throws Exception :线程执行带返回值V
```

java.util.Future<V>：**取得call方法得返回值**

```
V get( )throws InterruptedException, ExecutionException
```

应用场景:当线程需要返回值时,只能用callable接口实现多线程.

2.1.3定时调度池核心接口ScheduledExecutorService（继承ExecutorService）

作用：提供任务定时或周期执行方法的ExectorService。因为其所有方法返回值都是**ExectorService接口对象**

核心方法：**schedule 、scheduleAtFixedRate**

```
public interface ScheduledExecutorService extends ExecutorService {
  ..........................
  public ScheduledFuture<?> schedule(Runnable command/Callable<V>
callable,
  long delay, TimeUnit unit);
  延迟delay 个时间单位后开始执行

  public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
  long initialDelay,
  long period,
   TimeUnit unit);
   延迟initialDelay个时间个单位后开始执行，
   并且每隔period个时间单位就执行一次（一直执行）
 }
```

2.2四大实现子类

2.2.1普通调度核心子类： ExecutorService接口的子类 --AbstractExecutorService--ThreadPoolExecutor

- **AbstractExecutorSerivice:** 为ExectorSerive的任务提交方法提供了默认实现
- **ThreadPoolExecutor**：大名鼎鼎的**线程类**，**提供线程和任务调度的策略。**

核心方法：submit的实现

```
public abstract class AbstractExecutorService implements ExecutorService
{}

public class ThreadPoolExecutor extends AbstractExecutorService {}
```

2.2.2定时调度池地核心子类：eduledExecutorService接口的子类 -- ScheduledThreadPoolExecutor

- **ScheduledthreadPoolExecutor**：属于**线程池的一种**，它可以允许任务延迟或周期执行，类似java的Timer。

核心方法：schedule，scheduleAtFixedRate的实现

```
public class ScheduledThreadPoolExecutor
 extends ThreadPoolExecutor
 implements ScheduledExecutorService {
 ..................................
 }
```

2.2.3（执行ForkJoinTask任务）ForkJoinPool--AbstractExecutorService--ForkJoinPool

- **ForkJoinPool**:JDK1.7加如的成员，也是**线程池的一种。只允许执行ForkJoinTask任务**，它是为那些能够**递归拆解成子任务**的工作类型量身设计的，其目的在于**能够使用所有可用的运算资源来提升应用性能。**

```
public class ForkJoinPool extends AbstractExecutorService {
 ....................................................
 }
```

2.2.4特殊类 工具类:(创建各种线程池的工具类）Executors（不继承任何类）

- **Executors:创建各种线程池的工具类**

核心方法：new+各种线程池（）

eg:

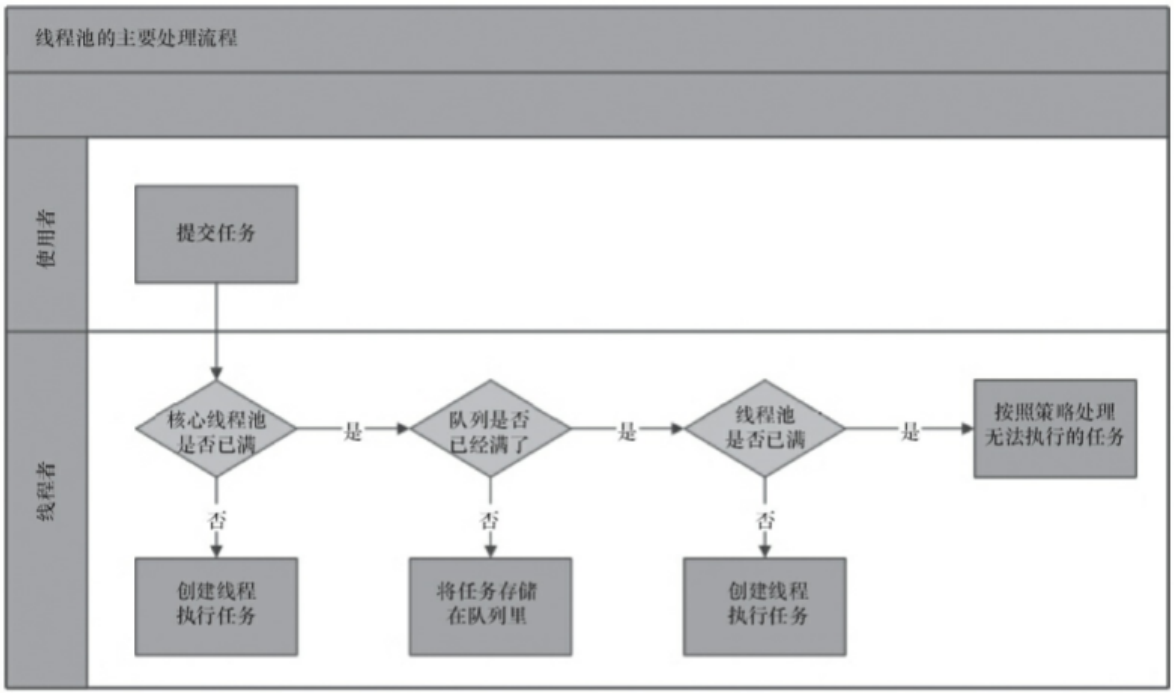
```
public static ScheduledExecutorService newScheduledThreadPool(int corePoo
lSize) {
```

```
public class Executors
```

3.线程池实现

3.1线程池实现原理

当线程池提交一个任务之后，线程池是如何处理这个任务的，主要处理流程如下



主处理流程：**核心4步法**

第一步：判断核心线程池是否已满，**如果未满，创建一个新的线程来执行任务。（创建线程需要全局锁）**
如果已满，判断是否有空闲线程，有的话将任务分配给空闲线程。（核心线程池已满）否则，执行步骤2

第二步：判断（阻塞队列）工作队列是否已满，**如果工作队列未满，将任务存储在工作队列中，等待空闲的线程调度。如果工作队列已满，**（阻塞队列/**工作线程池已满**）执行步骤3

第三步：**判断当前线程池的线程数量是否已达到最大值，如果已达到最大值，就创建新的线程执行任务。（创建线程需要全局锁）线程池已满**）否则，执行步骤4
此时线程不在核心线程

第四部：调用**饱和策略**来处理任务--四个策略

tips:

> 第一步，第三步：创建线程需要全局锁

范例：ThreadPoolExector执行execute()方法的流程如下：

1）如果当前运行的线程少于corePoolSize，则创建新的线程来执行任务（注意：执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多余corePoolSize，则将任务加入BlockingQueue。

3)如果无法将任务加入BlockingQueue(队列已满)，则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。
4）如果创建新的线程将是当前运行的线程超过maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。（**饱和策略)**

总节:
ThreadPoolExecutor采用上述步骤的总体设计思路，是为了在执行execute()方法时，尽可能的避免获取全局锁（那将会是一个严重的可伸缩瓶颈）。在ThreadPoolExecutor完成预热之后（当前运行线程数大于等于corePoolSize）,几乎所有的execute()方法调用都是执行步骤2，而步骤2不需要获取全局锁。

4.线程池的使用

4.1****手工创建线程池****

我们可以通过ThreadPoolExecutor来创建一个线程池

```
public ThreadPoolExecutor(int corePoolSize,
 int maximumPoolSize,
 long keepAliveTime,
 TimeUnit unit,
 BlockingQueue<Runnable> workQueue,
```

```
6   RejectedExecutionHandler handler){}
```

1）corePooolSize:(核心线程池大小)

当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新的任务也会创建线程，等到需要执行的任务数大于(**核心线程池大小**)时就不再创建。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。

2）BlockingQueue<Runnable> workQueu(任务队列)

保存等待执行任务的阻塞队列。可选择以下几个阻塞队列

a.ArrayBlockingQueue:是一个基于数组结构的**有界阻塞队列**，此队列按FIFO（先进先出）原则对元素进行排序。

b.LinkedBlockingQueue：一个基于链表结构的**无界阻塞队列**，此队列按**FIFO排序元素**，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法**Executors.newFixedThreadPool**（）使用了这个队列。（**固定大小线程池**就采用此队列）

c.SynchronousQueue：一个**不存储元素的阻塞队列（无界队列）**。每个插入操作需要等待另一个线程的移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkBlockingQueue。内置线程池，**Executors。newCachedThreadPool-缓存线程池**就采用此队列。

d.ProiorityBlockingQueue:具有优先级的无界阻塞队列。

- [ ] 优先队列

3）maximumPoolSize (线程池最大数量)

线程池允许创建的最大线程池数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。指的注意的是，如果使用了无界的任务队列这参数就没有什么效果。

4）keepAliveTime(线程活动保持时间)：

线程池的工作线程空闲后，保持存活的时间。所以，**如果任务很多，并且每个任务执行的时间比较短，可以调大时间，提高线程的利用率。**

5）TimeUnit(线程活动保持时间的单位):

可以选的单位有天（DAYS）、小时（HOURS）、分钟（MINUTES）、毫秒（MILLISECONDS）、微秒（MICROSECONDS，千分之一秒）和纳秒（NANOSECONDS、千分之一微秒）。

6） RejectedExecutionHandler（饱和策略）：

当队列和线程池都满了，说明线程处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AborPolicy,表示无法处理新任务时抛出异常。在JDK1.5中的Java线程池框架提供了以下
4种策略。

-AborPolicy：直接抛出异常。（默认采用此种策略）
-CAllerRunsPolicy:只用调用所在线程来运行任务。
-DiscardOldsetPolicy:丢弃队列里最近的一个任务，并执行当前任务/
-DiscardPolicy:不处理，丢弃掉。

- [ ] FutureTask Callable实验
- [ ] 查看线程状态：未关闭线程池时线程。

tips:

1.当调用Future的get()方法，会阻塞其他线程。直到取得当前线程执行完毕后的返回值。
FutureTask中的任务，只会执行一次。只会执行一次

2.如果采用LinkedBlockingQueue时：三个参数都被废了 饱和策略、线程池最大数量、任务队列
线程执行完后关闭线程池
shutdown( );

范例手工创建一个线程池

```
1
2 ThreadPoolExecutor threadPoolExecutor=
3  new ThreadPoolExecutor(3,5,2000,TimeUnit.MILLISECONDS,
4  new LinkBlockingDeque<Runnable>());
5
```

```
6
7
8  参数说明：
9   1.核心线程池大小：3个线程
10   2.线程池最大数量：5个线程
11   3..线程活动保持时间：2000
12   4.线程活动保持时间单位：毫秒
13   5.任务队列：链式无界阻塞队列 FIFO排序元素
14   6.饱和策略：不填，默认的AborPolicy：直接抛出异常。（默认采用此种策略）
```

4.2向线程池提交任务

可以使用两种方法向线程池提交任务，分别为execute()和submit()方法。

4.2.1execute()方法

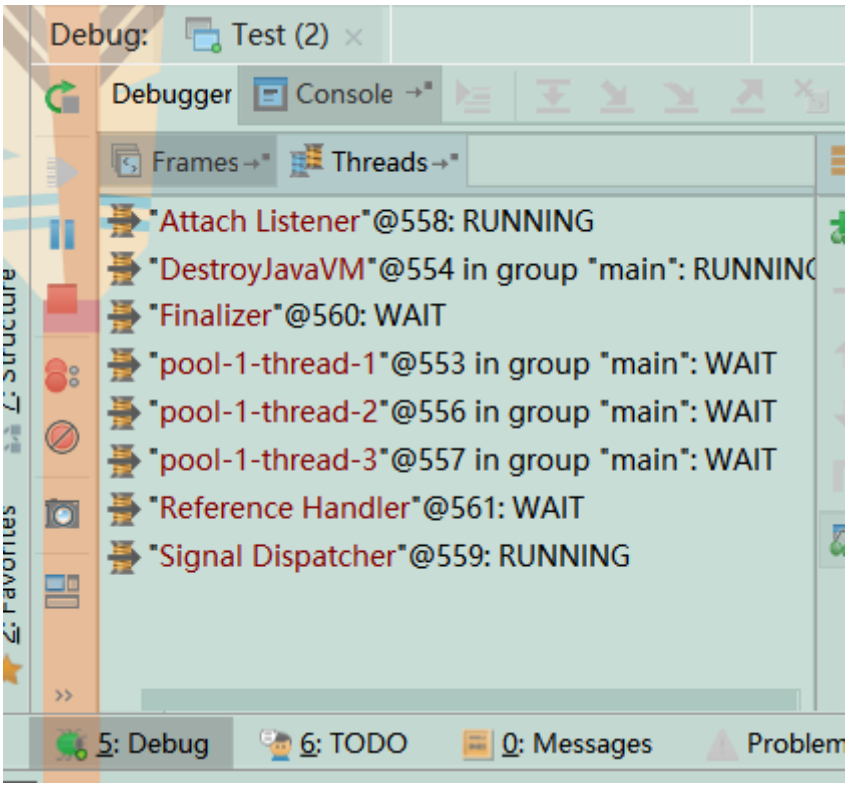
execute()方法用于**提交不需要返回值的任务**，所以无法判断任务是否被线程池执行成功。

范例：使用execut()方法

```
1  class Runnable implements Runnable{
2   @Override
3   public void run(){
4   for(int i = 0;i<50;i++){
5   System.out.printl(Thread.currentThread().getName()+","+i);
6   }
7   }
8  }
9
10  public class Test {
11   public static void main(String[] args) {
12   //execute（）方法
13   RunnableThread runnableThread = new RunnableThread();
14   ThreadPoolExecutor threadPoolExecutor =
15   new ThreadPoolExecutor(3, 5, 2000, TimeUnit.MILLISECONDS,
16   new LinkedBlockingQueue<Runnable>());
17   for (int i = 0; i < 5; i++) {
```

```
18    threadPoolExecutor.execute(runnableThread);
19    }
20    threadPoolExecutor.shutdown();//关闭线程池
21  }
```

问题：线程池不关闭，则线程存在哪里（显示WAIT状态）



4.2.2submit()

使用**submit()**方法用于提交需要返回的**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**获取反回值**，**get()**方法会**阻塞当前现线程直到任务完成**，而使用**get(long tomeout ,TimeUnit unit)**方法则会阻塞当前线程一段时间后立即返回，**这时侯可能任务没有执行完。**

小结：

1.当调用Future的get()方法，会阻塞其他线程。直到取得当前线程执行完毕后的返回值。FutureTask中的任务，只会执行一次。只会执行一次

2.get(long tomeout ,TimeUnit unit)方法则会阻塞当前线程一段时间后立即返回，**这时侯可能任务没有执行完。**

范例：使用submit（）方法

```
class CallableThread implements Callable<String> {
 @Override
 public String call() throws Exception {
 for (int i = 0; i < 10; i++) {
 System.out.println(Thread.currentThread().getName() + "," + i);
 }
 return Thread.currentThread().getName() + "任务执行完毕";
 }
}

public class Test {
 public static void main(String[] args) {
 ////sbumit方法
 CallableThread callableThread = new CallableThread();
 ThreadPoolExecutor threadPoolExecutor1 =
 new ThreadPoolExecutor(3, 5, 2000, TimeUnit.MILLISECONDS,
 new LinkedBlockingQueue<Runnable>());
 for (int i = 0; i < 5; i++) {
 Future<String> future = threadPoolExecutor1.submit(callableThr
ead);
 String str = null;
 try {
 str = future.get();
 System.out.println(str);
 } catch (InterruptedException e) {
 e.printStackTrace();
 } catch (ExecutionException e) {
 e.printStackTrace();
 }
 }
 threadPoolExecutor1.shutdown();//线程池的关闭
 }
}
```

4.3关闭线程池

可以通过调用线程的shutdown或shutdownNow方法来关闭线程池。它们的**原理是遍历线程池中的工作线程，然后 逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止**。但是它们存在一定的区别。

shutdownNow首先**将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表。**

shutdown只是将线程池的状态设置成**SHUTDOWN状态**，然后**中断所有没有正在执行任务的线程。**

只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都已关闭后，才表示线程 池关闭成功，这时调用iserminaed方法会返回true。

至于应该调用哪一种方法来关闭线程池，应该由提交到线程池 的任务特性决定，通常调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。

```
1 //1.
2 public void shutdown()
3 threadPoolExecutor.shutdown();
4
5 //2.
6 public List<Runnable> shutdownNow() {
7  //返回等待执行任务的列表。
8
```

4.4合理配置线程池

要想合理地配置线程池，就必须首先分析**任务特性**，可以从以下几个角度来分析。

> **任务的性质：CPU密集型任务、IO密集型任务和混合型任务。**
> **任务的优先级：高、中和低。**
> **任务的执行时间：长、中和短。**
> **任务的依赖性：是否依赖其他系统资源，如数据库连接。**

性质不同的任务可以用**不同规模的线程池**分开处理。

4.4.1 CPU密集型任务

CPU密集型任务应配置尽**可能小的线程**，如配置Ncpu+1个线程 的线程池。

4.4.2 IO密集型任务

由于**IO密集型任务**线程并不是一直在执行任务，则应**配置尽可能多的线程**，如2*Ncpu。

4.4.3 混合型的任务

混合型的任务， 如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大， 那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。**可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。**

4.4.4优先级不同的任务

优先级不同的任务可以使用**优先级队列PriorityBlockingQueue来**处理。它可以让优先级高的任务先执行。

注意：如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列，让执行时间短的任务先执行。

4.4.5依赖数据库连接池的任务

依赖数据库连接池的任务，**因为线程提交SQL后需要等待数据库返回结果**，等待的时间越长，**则CPU空闲时间就越长**，那么线程数应该设置得越大，这样才能**更好地利用CPU。**

如何理解cpu空闲时间和线程数之间的关系

5. Executor框架

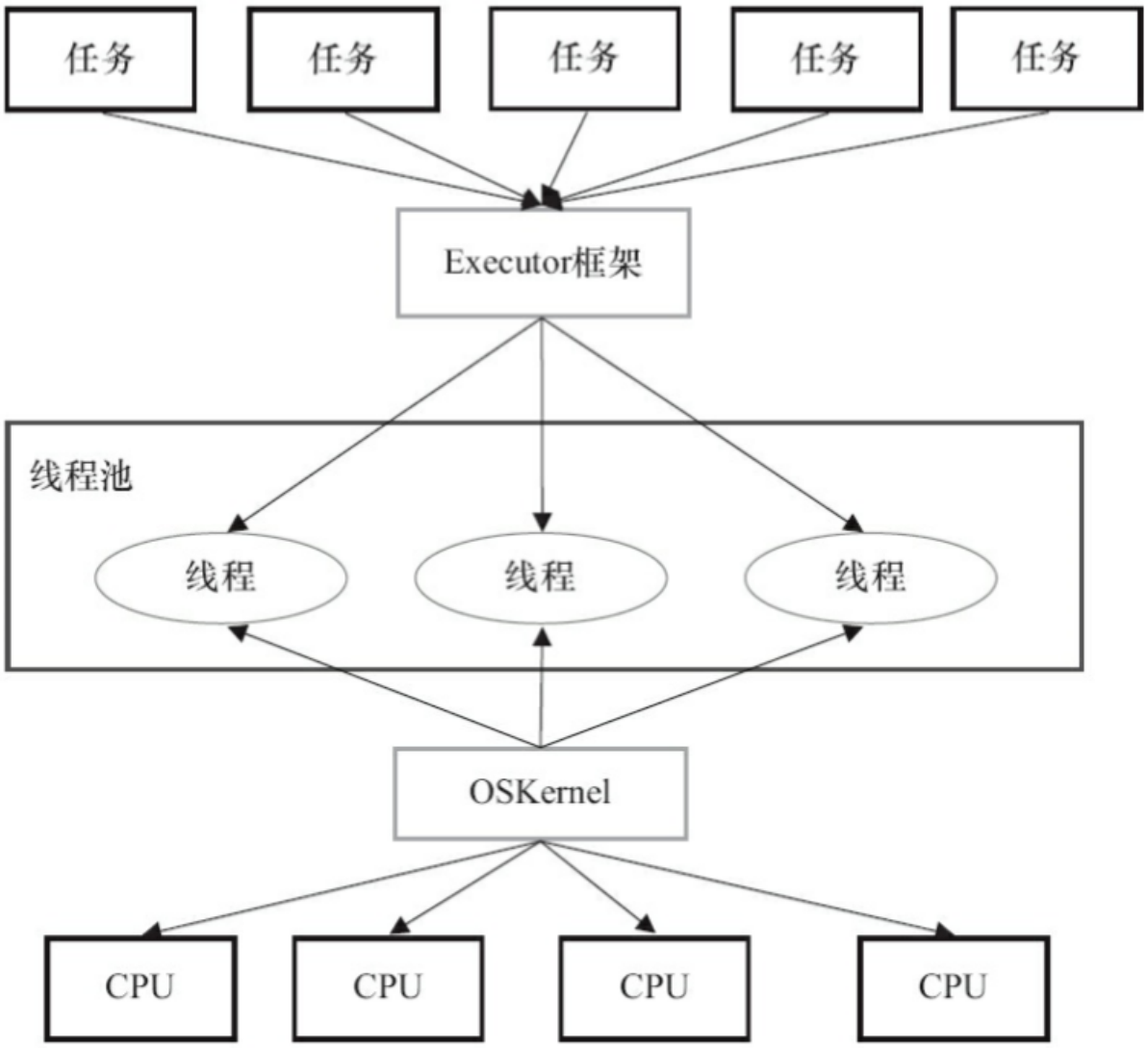
在java中，使用线程来异步执行任务时。java线程得到创建和销毁需要一定的开销，如果我们为每个任务创建一个新的线程来执行，这些线程的创建与开销将销毁大量的计算资源。同时，为每一个任务创建一个线程来执行，这种策略可能会使处于高负载状态得到应用最终崩溃。

java的线程即是**工作单元**，也是**执行机制**。从JDK5开始，把**工作单元与执行机制分离**开来。**工作单元包括Runnbale和Callable**，而**执行机制由Executor框架提供。**

5.1Executor框架的两级调度模型

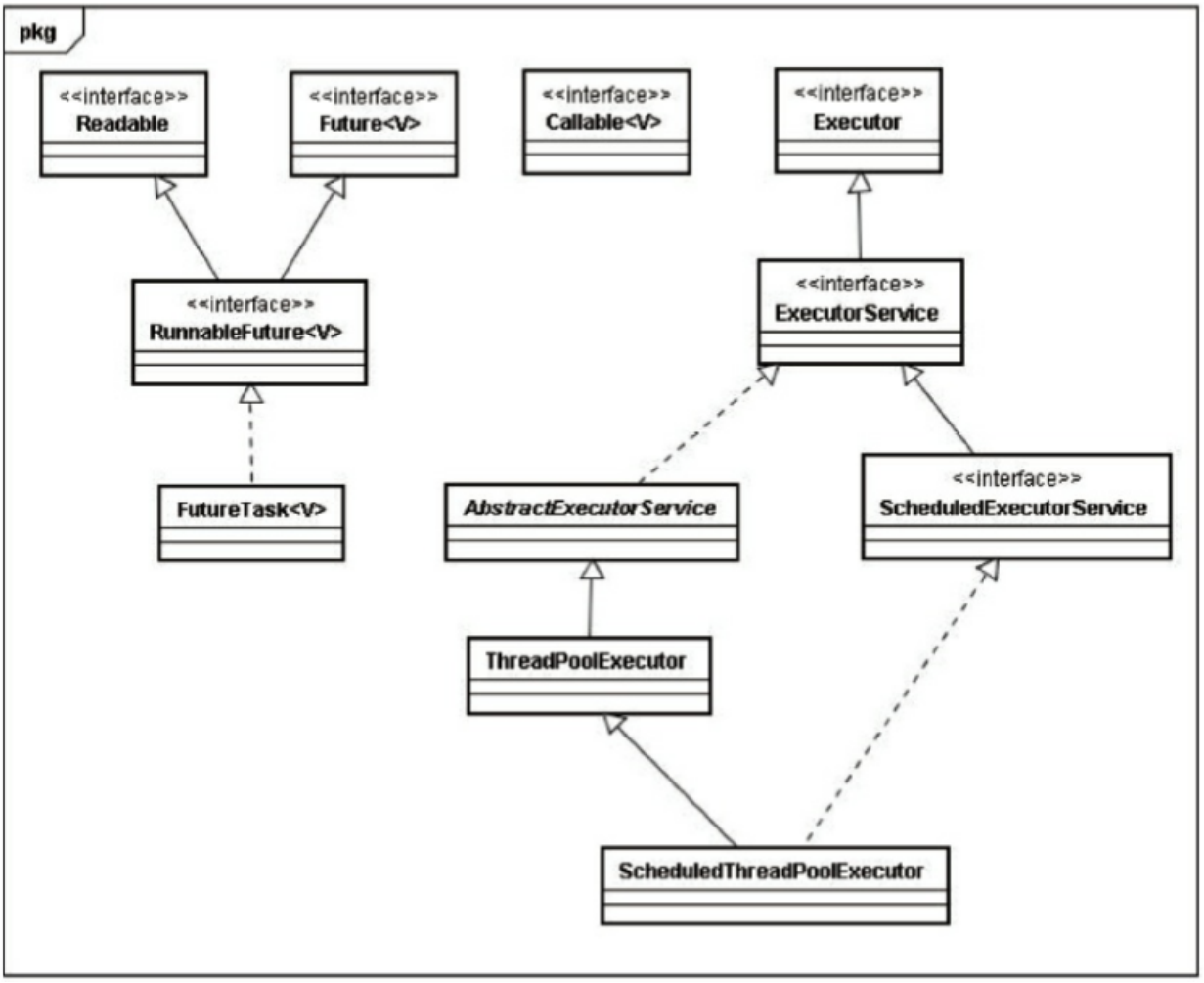
java线程**（java.lang.Thread）**被一对一映射为**本地操作系统线程。**java线程启动时会**创建一个本地操作系统**线程；当该线程终止的时候，这个操作系统线程机会被回收。操作系统会调度所有线程并将它们分配给可用的**CPU。**

在上层，java多线程程序通常把**应用分解为若干个任务**，然会使用**用户级的调度器（Executor框架）**将这些任务映射为固定数量得线程；在底侧过，操作系统内核将这些线程映射到硬件处理器上。这种两级调度模型得示意图如下所示：



5.2Executor框架的结构与成员

Executor框架的主要成员如下图：



Readable

Future

RunnableFuture

FutureTask

Callable

Executor

ExecutorService

AbstractExecutor

ThreadPoolExecutor

ScheduleExecutor Service

ScheduleThreadPoolExecutor

5.3ThreadPoolExecutor详情

Executor框架最核心的类是ThreadPoolExecutor，他是线程池的实现类。通过Executor，可以创建3种类型的ThreadPoolExecutor。

在**Executors工具类**产生线程（内置四大线程池）

5.3.1.创建无大小限制的线程池：FixedThreadPool详解

FixedThreadPool被称为可重用固定线程数的线程池。
corePoolSize:是固定的

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

FixedThreadPool使用无界队列LinkBlockingQueue作为线程池的工作队列（队列的容易为Integer.MAX_VALUE）。使用无界队列作为工作队列对线程池带来如下影响。

1）当线程池中的线程数达到corePoolSize后，新任务在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）由于使用无界队列，运行中的FixedThreadPool（未执行方法 shutdown()或shutdownNow()）不会拒绝任务（不会调用**RejectedExecutionHandler.rejectedExecution方法**）。

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

```
package ExecutorTest;

```

```
3
4  import java.util.concurrent.*;
5
6
7  public class ThreadPoolTest {
8   public static void main(String[] args) {
9   ExecutorService executorService =
10   Executors.newFixedThreadPool(5);
11   for (int i = 0; i < 5; i++) {
12   executorService.execute(new Runnable() {
13   @Override
14   public void run() {
15   for (int j = 0; j < 10; j++) {
16   System.out.println(Thread.currentThread().getName() + "," +
 j);
17   }
18   }
19   });
20
21   }
22   executorService.shutdown();
23   }
24  }
```

5.3.2单线程池 SingleThreadPoolExecutor详解

SingleThreadExecutor是使用单个worker线程的Executor

```
1  public static ExecutorService newSingleThreadExecutor() {
2   return new FinalizableDelegatedExecutorService
3   (new ThreadPoolExecutor(1, 1,
4   0L, TimeUnit.MILLISECONDS,
5   new LinkedBlockingQueue<Runnable>()));
6  }
```

SingleThreadExecutor 的 corePoolSize 和 maximumPoolSize 被设置为 1。其他参数 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 Integer.MAX_VALUE)。

使用场景

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。

```
1  package ExecutorTest;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SingleThreadTest {
7   public static void main(String[] args) {
8   ExecutorService executorService =
9   Executors.newSingleThreadExecutor();
10   for (int i = 0; i < 5; i++) {
11   executorService.execute(new Runnable() {
12   @Override
13   public void run() {
14   for (int j = 0; j < 10; j++) {
15   System.out.println(Thread.currentThread().getName() + "," +
j);
16   }
17   }
18   });
19   }
20   executorService.shutdown();
21   }
22  }
```

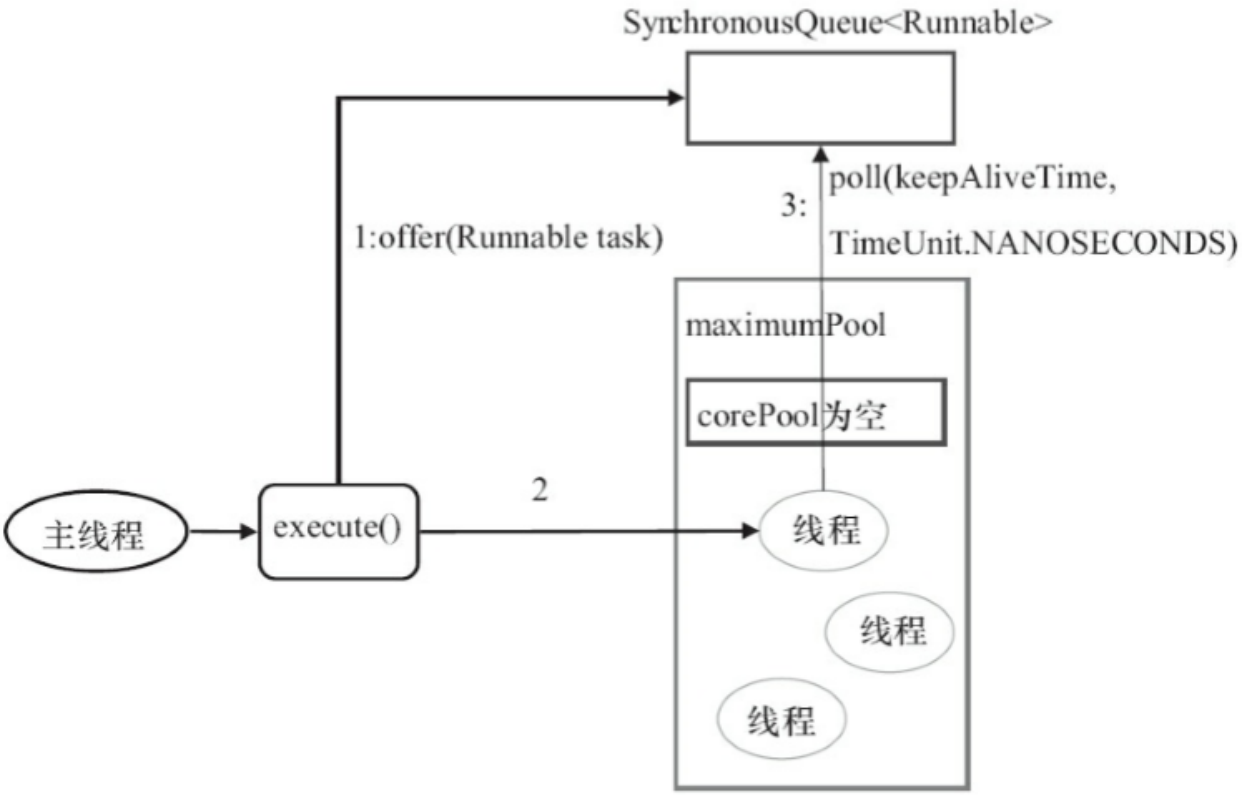
5.3.3 缓存线程池 CachedThreadPool详解

CachedThreadPool是一个会根据需要创建新线程的线程池。

```
1  public static ExecutorService newCachedThreadPool() {
```

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

1.CachedThreadPool的corePoolSize被设置为0，即corePool为空；
2.maximumPoolSize被设置为Integer.MAX_VALUE，即 maximumPool是无界的。
3.这里把keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的 长时间为60秒，空闲线程超过60秒后将会被终止。
4.FixedThreadPool 和 SingleThreadExecutor 使 用 无 界 队 列 LinkedBlockingQueue 作 为 线 程 池 的 工 作 队 列 。 CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列 但CachedThreadPool的maximumPool是无界的。
这意味着，**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 不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。**



步骤详解：
1）首先执行SynchronousQueue.offer（Runnable task）。如果当前maximumPool中有空闲线程正在执行 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）当初始maximumPool为空，或者maximumPool中当前没有空闲线程时，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会创建一个新线程执行任务，execute()方法执行完成。

3）在步骤2）中新创建的线程将任务执行完后，会执行SynchronousQueue.poll（keepAliveTime， TimeUnit.NANOSECONDS）。这个poll操作会让空闲线程多在SynchronousQueue中等待60秒钟。如果60秒钟 内主线程提交了一个新任务（主线程执行步骤1）），那么这个空闲线程将执行主线程提交的新任务；否则， 这个空闲线程将终止。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用 任何资源。

使用场景

CachedThreadPool是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者负载较轻的服务器。

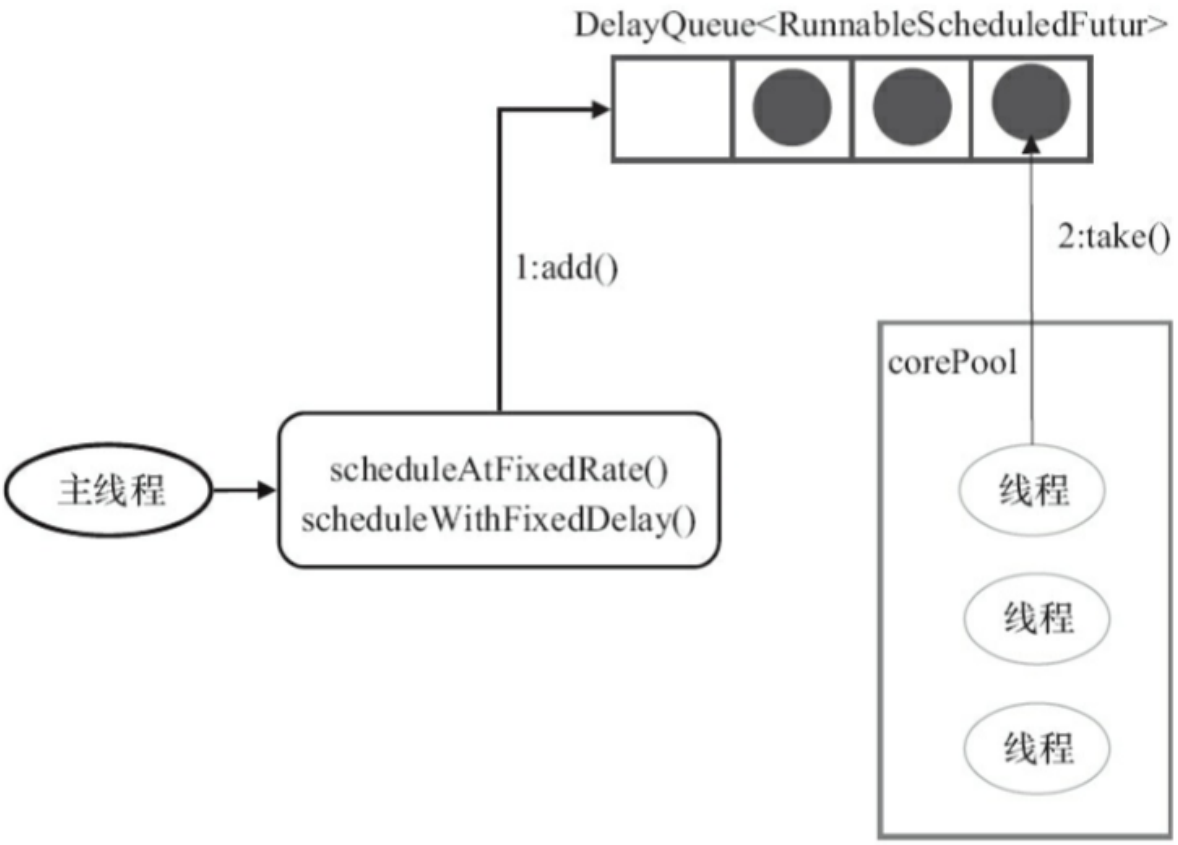
使用缓冲线程池

```
package ExecutorTest;

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

public class CachedThreadTest {
 public static void main(String[] args) {
 ExecutorService executorService =
 Executors.newCachedThreadPool();
  for (int i = 0; i < 5; i++) {
  try {
  Thread.sleep(0);
  } catch (InterruptedException e) {
  e.printStackTrace();
  }
```

```
16   executorService.submit(new Runnable() {
17   @Override
18   public void run() {
19   for (int j = 0; j < 10; j++) {
20   System.out.println(Thread.currentThread().getName() + "," +
j);
21   }
22   }
23   });
24   }
25   executorService.shutdown();
26   }
27  }
```

7.3.4 **定时调度池SchededThreadPool(int nThread)详解**

ScheduledThreadPoolExecutor继承自ThreadPoolExecutor。它主要用来在给定的延迟之后运行任务，或者定期执行任 务。ScheduledThreadPoolExecutor的功能与Timer类似，但ScheduledThreadPoolExecutor功能更强大、更灵活。Timer 对应的是单个后台线程，而ScheduledThreadPoolExecutor可以在构造函数中指定多个对应的后台线程数。ScheduledThreadPoolExecutor的执行流程图如下:



DelayQueue是一个无界队列，所以ThreadPoolExecutor的maximumPoolSize在ScheduledThreadPoolExecutor中没有什 么意义（设置maximumPoolSize的大小没有什么效果）。ScheduledThreadPoolExecutor的执行主要分为两大部分。

1）当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或者scheduleWithFixedDelay()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做了如下的修改。
1.使用DelayQueue作为任务队列。
2.获取任务的方式不同
3.执行周期任务后，增加了额外的处理

适用场景：
需要定时执行任务的场景。

```
1 package ExecutorTest;
```

```
2
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ScheduledExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  import static java.lang.Thread.sleep;
8
9  public class ScheduledThreadTest {
10   public static void main(String[] args) throws InterruptedExcep
tion {
11   ScheduledExecutorService executorService =
12   Executors.newScheduledThreadPool(5);
13   System.out.println(Thread.currentThread().getName());
14   for (int i = 0; i < 100; i++) {
15   executorService.schedule(new Runnable() {
16   @Override
17   public void run() {
18
19   for (int j = 0; j < 1000; j++) {
20   System.out.println(Thread.currentThread().getName() + "," +
j);
21   }
22   }
23   }, 8, TimeUnit.SECONDS);
24   }
25  // sleep(10002); // ?定时调度池在延迟期间主线程也在一直运行 为什么不
执行关闭线程池的方法
26   executorService.shutdown();
27   }
28  }
```

问题

- [ ] 问题：定时调度池在延迟期间主线程也在一直运行 为什么不执行关闭线程池的方法