(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：

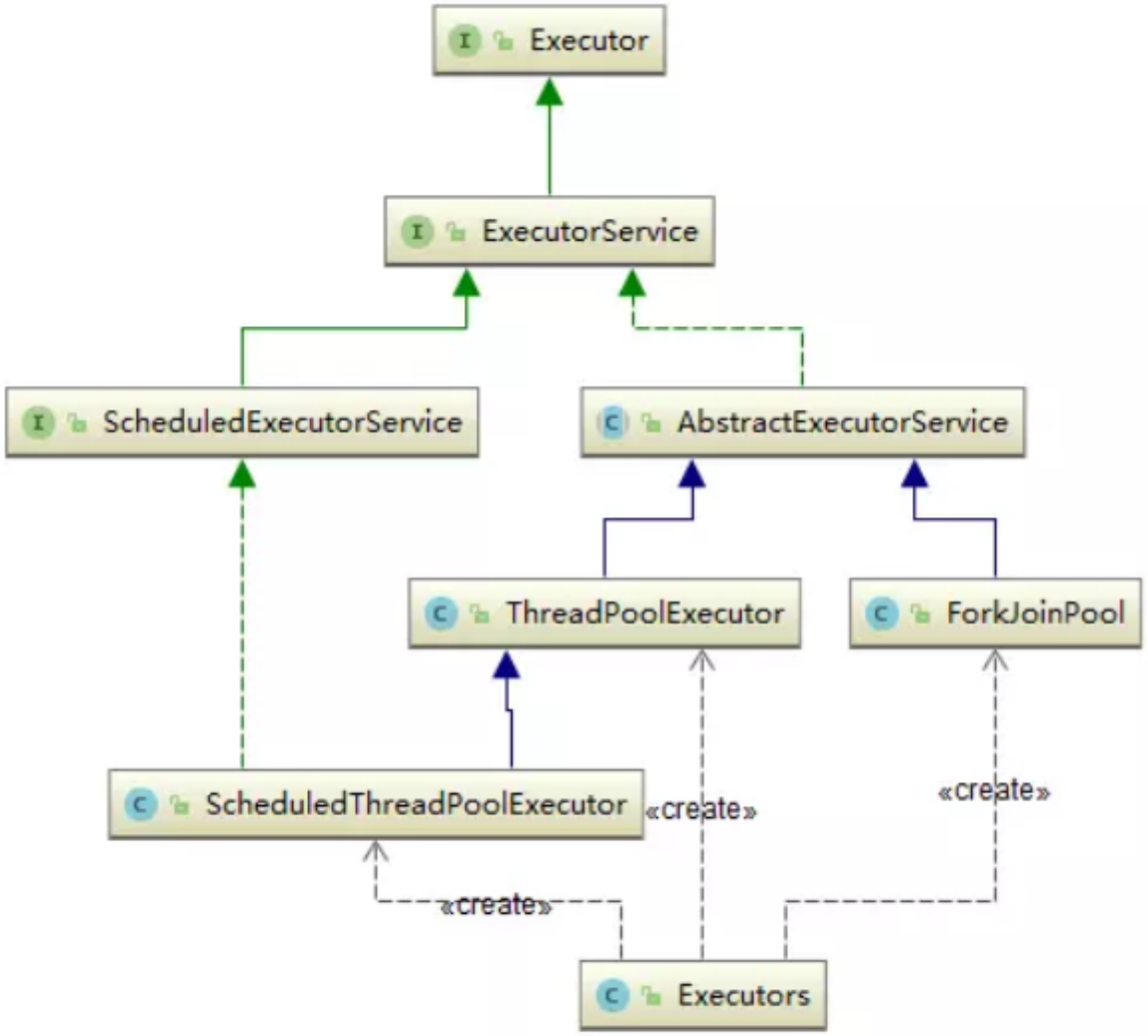
- Executor :线程池的**最上层接口**，提供了任务提交的基础方法。
- ExecutorService:提供了线程池管理的**上层接口**，如池的销毁，异步任务的提交
-
- ScheduleExecutorService:提供任务定时或周期执行方法的**ExectorService**。
- AbstractExecutorSerivice:为ExectorSerive的任务提交方法提供了默认实现
- ThreadPoolExecutor：大名鼎鼎的**线程类**，提供线程和任务调度的策略。
- ScheduledthreadPoolExecutor：属于**线程池的一种**，它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。

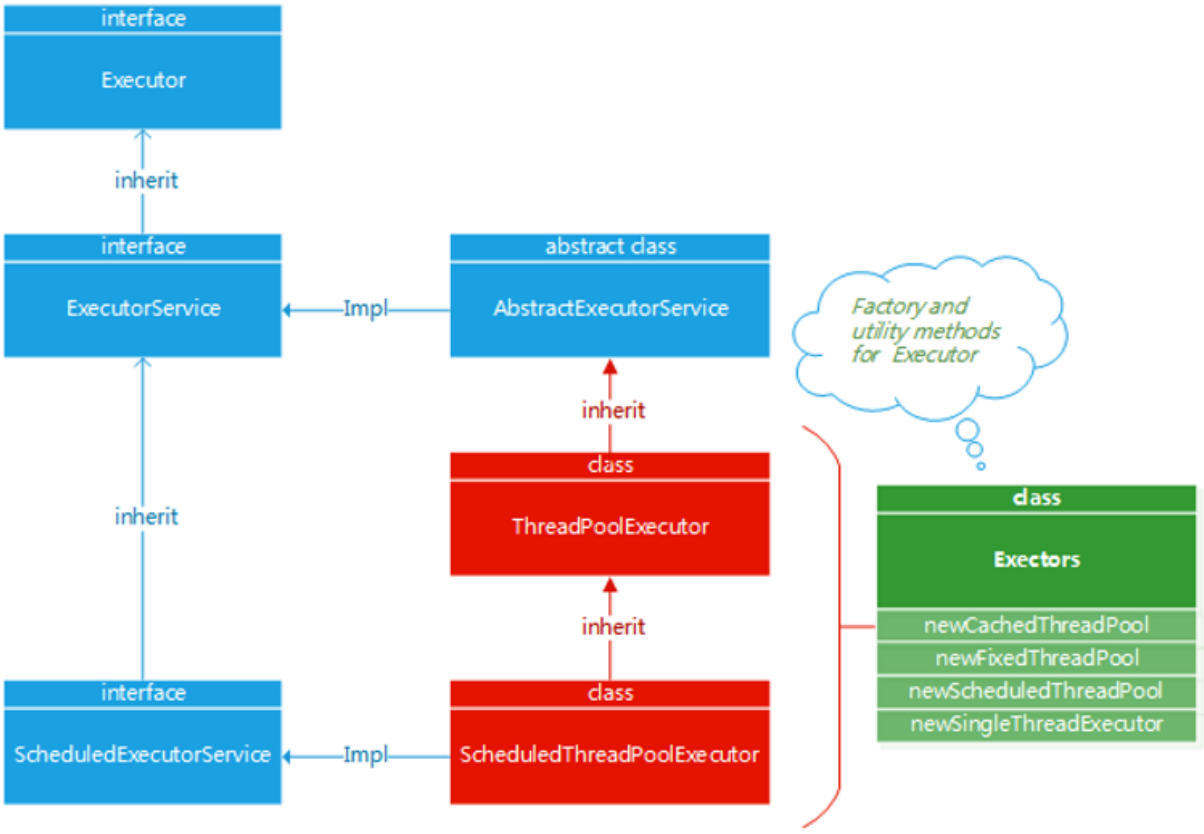
1.3线程池核心框图



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**

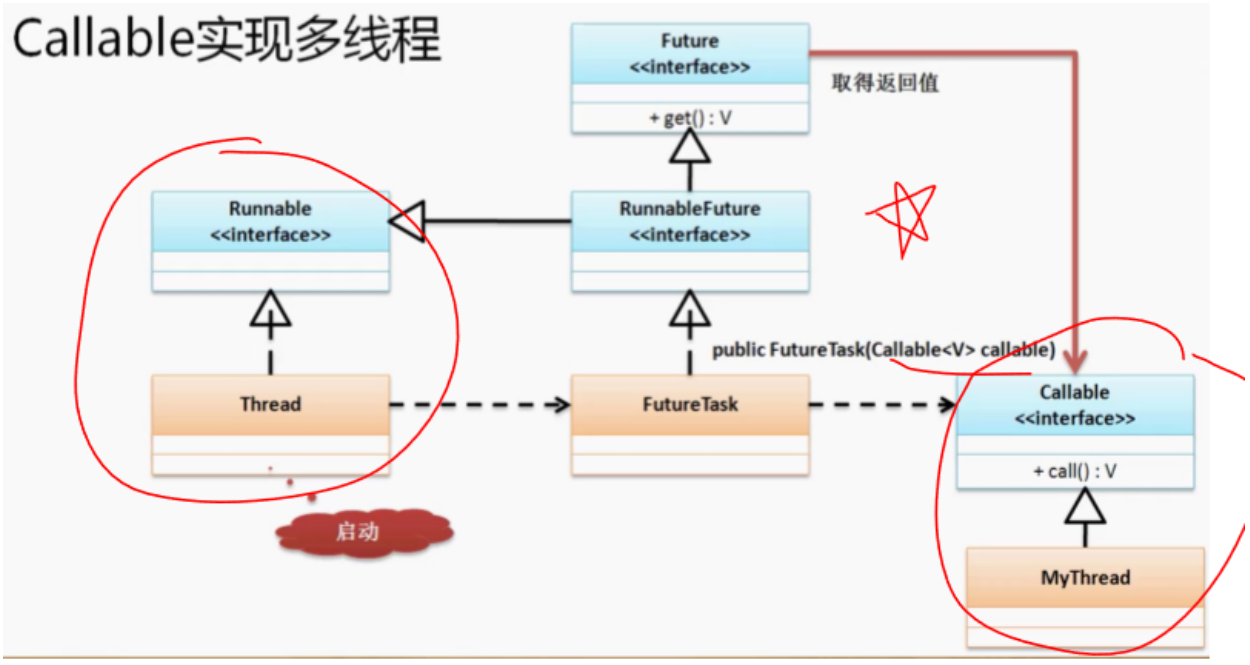
```
public interface ExecutorService extends Executor {
 ....................................
 <T> Future<T> submit(Callable<T> task/Runnable task, T result);
}
```

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:

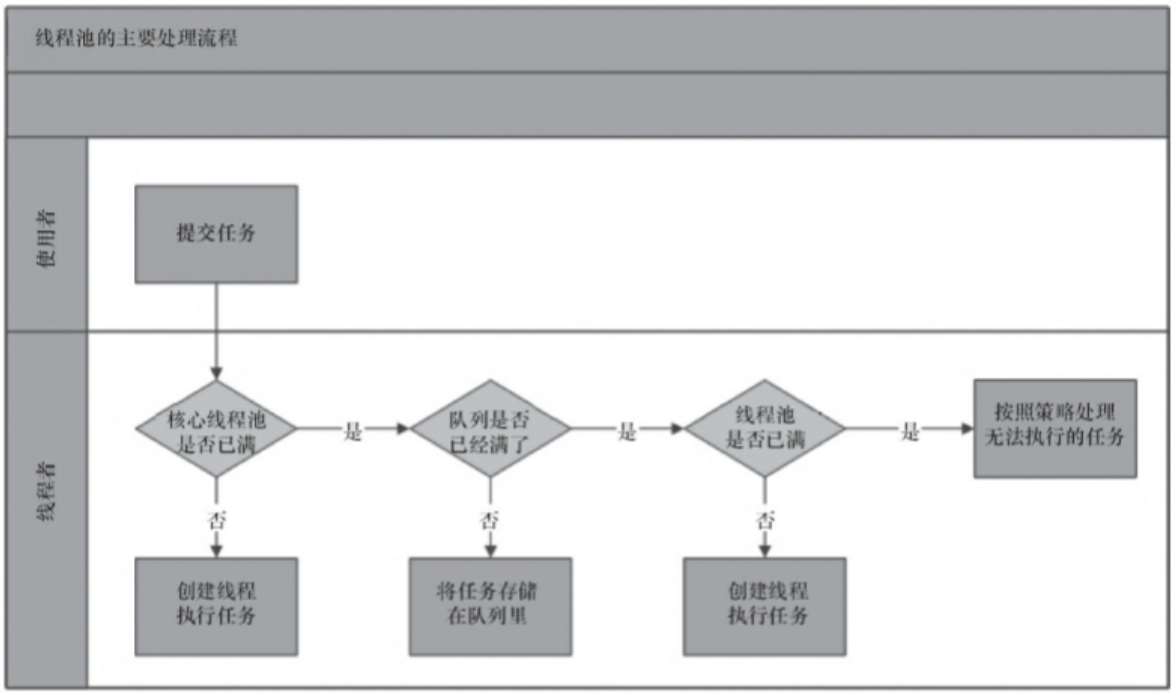
```
public static ScheduledExecutorService newScheduledThreadPool(int corePoo
lSize) {
```

```
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,
 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：
```

```
1.核心线程池大小：3个线程
 2.线程池最大数量：5个线程
 3..线程活动保持时间：2000
 4.线程活动保持时间单位：毫秒
 5.任务队列：链式无界阻塞队列 FIFO排序元素
 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。

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

```
```