线程池【治理线程的最大法宝】

1. 线程池的自我介绍

什么是“池”

软件中的池可以理解为计划经济

如果不使用线程池，每个任务都新开一个线程处理

一个线程

线程多了：for循环创建线程

当任务数量上升到1000

这样开销太大,创建和销毁线程都有很大的开销，并且会增加垃圾回收器的负担。我们希望有固定数量的线程,来执行这1000个线程,这样就避免了反复创建并销毁线程所带来的开销问题。

为什么要使用线程池

- 问题一：反复创建线程开销大
- 问题二：过多的线程会占用太多内存

实际生产中任务是不受我们控制的，如果每个任务都创建一个线程，而系统和程序存在创建线程的上限（可能会OOM异常），这样我们系统很快会达到性能瓶颈。

- 解决以上两个问题的思路
    - 用少量的线程一避免内存占用过多
    - 让这部分线程都保持工作,且可以反复执行任务避免生命周期的损耗

线程池好处

- 加快响应速度
- 合理利用CPU和内存
- 统一管理资源，比如有3000个任务执行到一半，我们不可能再把这些线程一一停止，可以利用线程池统一去管理。

线程池适合应用的场合

- **服务器**接受到大量请求时,使用线程池技术是非常合适的,它可以大大减少线程的创建和销毁次数,提高服务器的工作效率，Tomcat的NIO，利用线程池和IO多路复用
- 实际上,在开发中,如果需要创建5个以上的线程,那么就可以使用线程池来管理

2. 创建和停止线程池

2.1 线程池构造函数的参数

2.1.1 每个参数的含义概览（6个）

参数名	类型	含义
corePoolSize	int	核心线程数，详解见下文
maxPoolSize	int	最大线程数，详解见下文
keepAliveTime	long	保持存活时间
workQueue	BlockingQueue	任务存储队列
threadFactory	ThreadFactory	当线程池需要新的线程的时候，会使用threadFactory来生成新的线程
Handler	RejectedExecutionHandler	由于线程池无法接受你所提交的任务的拒绝策略

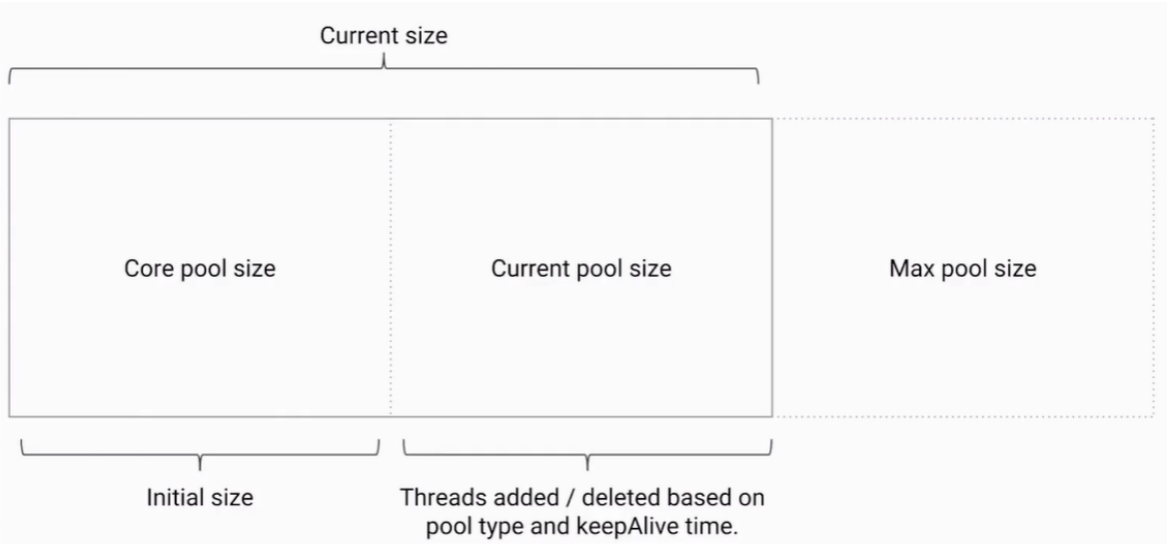
> maximumPoolSize的说明 在课程中, maximumPoolSize?和maxPoolSize的含义相同,不做额外区分。实际上,在 ThreadpoolExecutor类的参数中,变量名是 maximumPoolSize;不过在 org.springframework.scheduling.concurrent包的ThreadPoolExecutorFactoryBean类等其他类中,也有使用 maxPoolSize作为参数名的情况,我们直接理解为 maximumPoolSize和 maxPoolSize是相同的就可以了

2.1.2参数中的corePoolSize和maxPoolSize有什么不同

corePoolSize指的是核心线程数:线程池在完成初始化后,**默认情况下,线程池中并没有任何线程**,线程池会**等待有任务**到来时,**再创建corePoolSize个线程**去执行任务（随着任务一个个创建），且核心线程数不会再改变，即使此时没有任务执行，也不会销毁核心线程，除非发生异常。

线程池有可能会在核心线程数的基础上,额外增加一些线程,但是这些新增加的线程数有一个上限,这就是最大量 maxpoolsize

切记： 创建线程的时候不是会指定一个阻塞队列吗，一般不会轻易突破核心线程数，只有当达到核心线程数，且阻塞队列中的任务满了的时候，maxPoolSize才会起作用，去创建新的线程。



2.1.3 线程增加和减少以及task进入队列排队的规则

线程添加规则

1. 如果线程数小于 corepoolSize,即使其他工作线程处于空闲状态,也会创建一个新线程来运行新任务。
2. 如果线程数等于(或大于) corepoolsize但少于maximumpoolsize,则将**任务**放入队列。
3. 如果队列已满,并且线程数小于 maxpoolsize,则创建一个新线程来**运行队列头的任务**，新的任务放到阻塞队列队尾。

4. 如果队列已满,并且线程数大于或等于maxpoolsize,则拒绝新任务。

是否需要增加线程的判断顺序是

1. corePoolSize
2. workQueue
3. maxPoolSize

增减线程的特点

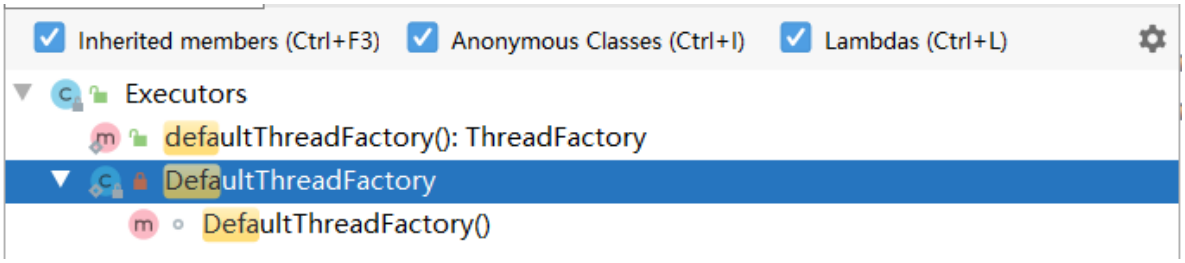
1. 通过设置 `corePoolSize` 和 `maximumPoolsize` 相同,就可以创建固定大小的线程池。
2. 线程池希望保持较少的线程数,并且只有在负载变得很大时才增加它。
3. 通过设置 `maximumPoolSize` 为很高的值,例如 `Integer.MAXVALUE`,可以允许线程池容纳任意数量的并发任务。
4. 是只有在队列填满时才创建多于 `corePoolSize` 的线程,所以如果你使用的是无界队列(例如 `linkedBlockingQueue`),那么线程数就不会超过 `corePoolSize`。

2.1.4 keepAlivedTime

如果线程池当前的线程数多于 corePoolSize,那么非核心线程的空闲时间如果超过 keepalivetime,它们就会被终止，如果核心线程设置了 `allowCoreThreadTimeOut = true` 那么核心线程也会被终止，通常 `allowCoreThreadTimeOut = false`

2.1.5 ThreadFactory（用来创建线程）

新的线程是由 Threadfactory创建的,默认使用 `Executors.defaultThreadFactory()` ,创建出来的线程都在同个线程组,拥有同样的NORM_ PRIORITY优先级并且都不是守护线程。如果自己指定 Thread Factory,那么就可以改变线程名、线程组、优先级、是否是守护线程等。一般就使用默认的就可以。



Executors中有静态的 `defaultThreadFactory()` 方法,会对ThreadFactory初始化

```
public static ThreadFactory defaultThreadFactory() {
    return new DefaultThreadFactory();
}
```

DefaultThreadFactory作为Executors的静态内部类

从源码中可以看出，我们的线程工厂给线程设置了默认名字（pool-线程池自增编号-thread-线程的自增编号），非守护线程，默认优先级、默认线程组。

```
 private 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;
        }
    }
```

2.1.6 BlockingQueue

直接交接：SynchronousQueue，此队列没有容量，如果使用该队列，maxPoolSize可能需要设置的大一些。

无界队列：LinkedBlockingQueue，可以防止流量突增，但是如果处理的速度跟不上任务提交的速度，会造成队列中任务越来越多，会造成内存浪费，甚至OOM异常。

有界队列：ArrayBlockingQueue

2.2 线程池应该手动创建还是自动创建（阿里巴巴规约）

手动创建更好,因为这样可以让我们更加明确线程池的运行规则,避免资源耗尽的风险。

让我们来看看自动创建线程池(也就是直接调用JDK封装好的构造函数)可能带来哪些问题

2.2.1 Executors.newFixedThreadPool(int nThreads)

进入Executors看源码

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

发现还是调用了 `new ThreadPoolExecutor` , `nThreads` 为传进来的要创建的线程数。

点击 `ThreadPoolExecutor` 进入 `ThreadPoolExecutor` 源码

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
}
```

可以发现前两个参数`int corePoolSize, int maximumPoolSize`都设置为了相同的值，`keepAliveTime`过期时间为 0，`BlockingQueue`使用的`LinkedBlockingQueue`无界队列。

> 由于传进去的`LinkedBlockingQueue`是没有容量上限的所以当请求数越来越多,并且无法及时处理完毕的时候也就是请求堆积的时候,会容易造成占用大量的内存,**可能会导致OOM**。

OOM代码演示`class FixedThreadPoolOOM`

```
package threadpool;

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

/**
 * 描述：    演示newFixedThreadPool出错的情况 ，为了更快的看到OOM，JVM参数设置为 -Xmx8m -Xms8m
 * 可以看到Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main"
 */
public class FixedThreadPoolOOM {

    private static ExecutorService executorService = Executors.newFixedThreadPool(1);
    public static void main(String[] args) {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            executorService.execute(new SubThread());
        }
    }
}

class SubThread implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(1000000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

2.2.2 Executors.newSingleThreadExecutor()

只生产一个线程，同样使用`LinkedBlockingQueue`，会导致OOM

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

2.2.3 Executors.newCachedThreadPool()

可缓存线程池 特点:无界线程池,具有自动回收多余线程的功能

最大线程数为`Integer.MAX_VALUE`，核心线程数为0，使用`SynchronousQueue`直接交接，队列没有容量。所以每来一个任务，都会创建一个线程。每个线程超过60s没有使用了，就会被回收。

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

> 这里的弊端在于第二个参数 maximumpoolsize被设置为了 Intege. MAX VALUE,这**可能会创建数量非常多的线程**，甚至导致OOM。

2.2.4 Executors.newScheduledThreadPool(10)

支持定时及周期性任务执行的线程池，使用了延迟队列`DelayedWorkQueue`，也是可以创建`Integer.MAX_VALUE`多个线程，可能会导致OOM。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE,
          DEFAULT_KEEPALIVE_MILLIS, MILLISECONDS,
          new DelayedWorkQueue());
}
```

代码演示:

```
package threadpool;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * 描述：     TODO
 */
public class ScheduledThreadPoolTest {

    public static void main(String[] args) {
        ScheduledExecutorService threadPool =
Executors.newScheduledThreadPool(10);
//        threadPool.schedule(new Task(), 5, TimeUnit.SECONDS);
        threadPool.scheduleAtFixedRate(new Task(), 1, 3, TimeUnit.SECONDS);
    }
}
```

`threadPool.schedule(new Task(), 5, TimeUnit.SECONDS);` 可以为传进来的任务设置时间，任务来了以后，5秒后才执行。

`threadPool.scheduleAtFixedRate(new Task(), 1, 3, TimeUnit.SECONDS);` 1:任务第一次执行是1s后，后面每3秒执行一次

正确的创建线程池的方法

根据不同的业务场景,自己设置线程池参数,比如我们的内存有多大,我们想给线程取什么名字等等。

2.3 线程池里的线程数量设定为多少比较合适?

- CPU密集型(加密、计算hash等):最佳线程数为CPU核心 数的1-2倍左右。
- 耗时IO型(读写数据库、文件、网络读写等):最佳线程数般会大于cpu核心数很多倍,以VM线程控显示繁忙情况为依据,保证线程空可以衔接上。参考 Brain Goetz推荐的计算方法:**线程数=CPU核心数*(1+平均等待时间/平均工作时间)** 。如果想做到更加精准的话，应该是根据不同的程序做压测。

2.4 停止线程池的正确方法

shutdown()

> 关闭线程池的方法，将线程池状态置为SHUTDOWN,线程并不会立即停止，此方法只是初始化整个关闭过程。因为线程池在执行到一半的时候，1. 包括线程中有正在执行的任务，2. 还包括队列中等待被执行任务，所以我们不能说让它停止就停止，只是做到一个通知的作用，执行此方法，存量的任务直到会执行完毕，后续来的任务会拒绝。是一种优雅关闭的方式。

- 将线程池状态置为**SHUTDOWN**
- 不能接受新的**submit**
- 并没有任何的**interrupt**操作，会等待线程池中所有线程（执行中的以及排队的）执行完毕

isShutdown()

如果执行了shutdown方法，则会打印为true，返回true不代表线程池结束，只是意味着收到了结束的通知。

isTerminated()

判断线程池是否已经完全终止（所有任务已经执行完毕）。

awaitTermination(n, TimeUnit)

> 不是用来停止线程的，只是等待一段时间，在等待的这一段时间内，如果所有线程都执行完毕了，返回true否则放回false，在等待的时间被打断，抛出InterruptException。该方法在返回之前是阻塞的。

- 该方法返回值为boolean类型
- 方法的两个参数规定了方法的阻塞时间，在**阻塞时间内**除非所有线程**都执行完毕**才会提前返回 `true`
- 如果到了规定的时间，线程池中的线程并没有全部结束返回false
- InterruptedException 这个异常也会导致方法的终止

利用这个阻塞方法的特性，我们可以优雅的关闭线程池中的任务。

```
// 创建线程池
// 执行业务逻辑
pool.shutdown();
if(!pool.awaitTermination(10, TimeUnit.SECONDS)) {
    pool.shutdownNow();
}
```

shutdownNow()

将线程池状态置为**STOP**。企图立即停止，事实上不一定：

- **会尝试interrupt**线程池中正在执行的线程
- 但是**并不能保证一定能**成功地interrupt线程池中的线程。
- 等待执行的线程并不会被执行（队列里等待的任务），会被返回`List<Runnable>`

shutdownNow()方法比shutdown()强硬了很多，不仅取消了排队的线程而且确实尝试终止当前正在执行的线程。它试图终止线程的方法是通过调用Thread.interrupt()方法来实现的，但是大家知道，这种方法的作用有限，如果线程中没有sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线程的。

3. 常见线程池的特点和用法

- FixedThreadPool
- CachedThreadPool
- ScheduledThreadPool
- SingleThreadExecutor

3.1 以上4种线程池的构造函数的参数

Parameter	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	constructor-arg	0	constructor-arg	1
maxPoolSize	same as corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0 seconds	60 seconds	0	0 seconds

3.2 以上4种线程池对应的阻塞队列分析

- FixedThreadPool和 SingleThreadExecutor的 Queue是Linked Blocking Queue?

  由于无法突破核心线程数，所以把阻塞队列的容量设置为无穷。

- CachedThreadPool使用的 Queue是 SynchronousQueue?

  任务过来直接让新的线程去执行。新的线程的数量是不受限制的。不需要队列来存储任务。使用SynchronousQueue不需要在队列中去中转，效率会高一些。

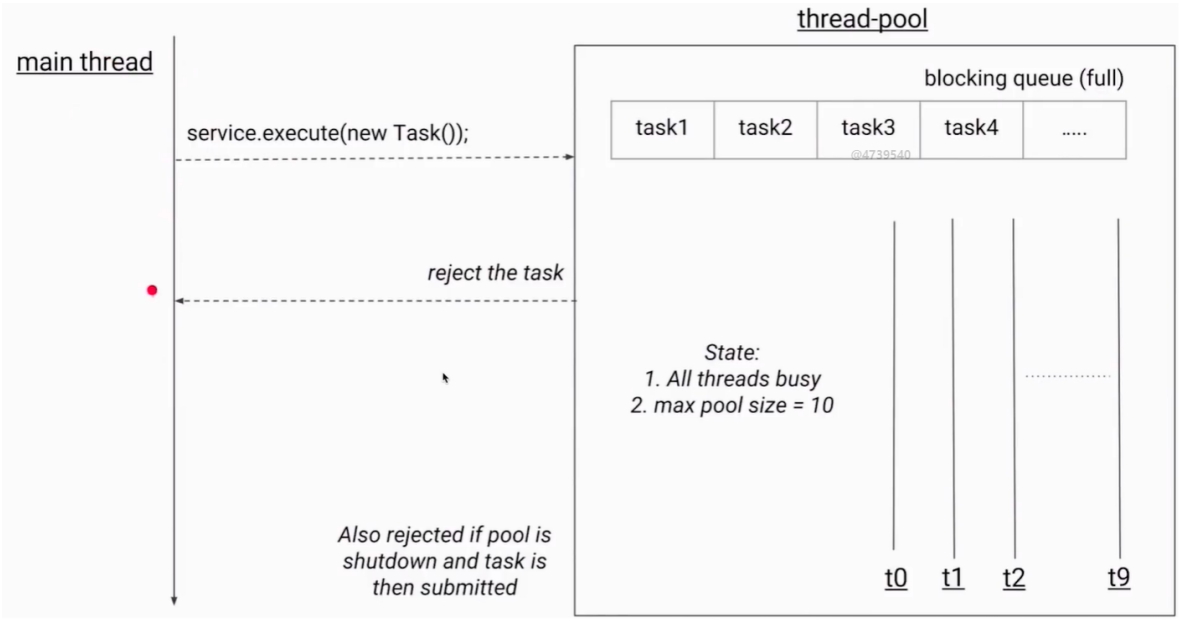
- ScheduledThreadpool来说,它使用的是延迟队列DelayedWorkQueue
- workStealingPool是JDK1.8加入的
  - 这里的任务不是普通的任务，如果这里的任务会产生子任务，才适合使用。比如说树的遍历，会层层往下。处理矩阵的时候也有可能产生子任务，比如把一个矩阵分为四个小矩阵，再可以依次分下去。
  - 用于一定的窃取能力：Stealing。这种线程池每一个线程之间可以合作。子任务会被放到每个线程独有的队列中去。但是如果有一些线程空闲，会帮助其他线程去把其他线程中队列中的任务给取出来，来帮助繁忙的线程执行任务。这样一来，相当于实现了并行执行的效果。但是要注意：1. 为了提高线程池的效率，我们的任务最好不要加锁，因为每个任务可能会被不

同的线程去执行。2. 由于线程直接的窃取，该线程池无法保证任务执行顺序。在之前的线程中，任务会按照队列中的顺序去执行。

4. 任务太多，怎么拒绝?

拒绝的时机

1. 当 Executors关闭时,提交新任务会被拒绝。
2. 以及当 Executor对最大线程和工作队列容量使用有限边界 并且已经饱和时



4种拒绝策略

> 线程池的默认拒绝策略为AbortPolicy，即丢弃任务并抛出RejectedExecutionException异常。我们可以通过代码来验证这一点，现有如下代码：

```
public class ThreadPoolTest {

    public static void main(String[] args) {

        BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);
        ThreadFactory factory = r -> new Thread(r, "test-thread-pool");
        ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 5,
                0L, TimeUnit.SECONDS, queue, factory);
        while (true) {
            executor.submit(() -> {
                try {
                    System.out.println(queue.size());
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
    }

}
```

AbortPolicy

直接抛出异常：`java.util.concurrent.RejectedExecutionException`

`RejectedExecutionHandler handler = new ThreadPoolExecutor.AbortPolicy();`

`handler`是创建线程池的最后一个参数

DiscardPolicy

丢弃任务，但是不抛出异常。

DiscardOldestPolicy

丢弃队列最前面的任务，然后重新提交被拒绝的任务

CallerRunsPolicy

由调用线程（提交任务的线程）处理该任务

5. 钩子方法，给线程池加点料

我们可以在每个任务执行之前和之后调用`beforeExecute(Thread, Runnable)`和`afterExecute(Runnable, Throwable)`方法。这些可以用来操纵执行环境; 例如，重新初始化 ThreadLocal，收集统计信息或添加日志条目。此外，可以重写`terminated()`方法，以执行`Executor`完全终止后需要执行的任何特殊处理。

如果钩子或回调方法抛出异常，内部工作线程可能会失败并突然终止。

代码PausableThreadPoolExecutor类

```
package threadpool;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 描述：      演示每个任务执行前后放钩子函数
 */
public class PauseableThreadPool extends ThreadPoolExecutor {

    private final ReentrantLock lock = new ReentrantLock();
    private Condition unpaused = lock.newCondition();
    private boolean isPaused;


    public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
            TimeUnit unit,
            BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue);
    }
```

```
27
28      public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
29              TimeUnit unit, BlockingQueue<Runnable> workQueue,
30              ThreadFactory threadFactory) {
31          super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue, threadFactory);
32      }
33
34      public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
35              TimeUnit unit, BlockingQueue<Runnable> workQueue,
36              RejectedExecutionHandler handler) {
37          super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue, handler);
38      }
39
40      public PauseableThreadPool(int corePoolSize, int maximumPoolSize, long
keepAliveTime,
41              TimeUnit unit, BlockingQueue<Runnable> workQueue,
42              ThreadFactory threadFactory, RejectedExecutionHandler handler)
{
43          super(corePoolSize, maximumPoolSize, keepAliveTime, unit,
workQueue, threadFactory,
44                  handler);
45      }
46
47      @Override
48      protected void beforeExecute(Thread t, Runnable r) {
49          super.beforeExecute(t, r);
50          lock.lock();
51          try {
52              while (isPaused) {
53                  unpaused.await();
54              }
55          } catch (InterruptedException e) {
56              e.printStackTrace();
57          } finally {
58              lock.unlock();
59          }
60      }
61
62      private void pause() {
63          lock.lock();
64          try {
65              isPaused = true;
66          } finally {
67              lock.unlock();
68          }
69      }
70
71      public void resume() {
72          lock.lock();
73          try {
74              isPaused = false;
75              unpaused.signalAll();
76          } finally {
77              lock.unlock();
```

```
        }
    }

    public static void main(String[] args) throws InterruptedException {
        PauseableThreadPool pauseableThreadPool = new
PauseableThreadPool(10, 20, 10l,
                TimeUnit.SECONDS, new LinkedBlockingQueue<>());
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("我被执行");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10000; i++) {
            pauseableThreadPool.execute(runnable);
        }
        Thread.sleep(1500);
        pauseableThreadPool.pause();
        System.out.println("线程池被暂停了");
        Thread.sleep(1500);
        pauseableThreadPool.resume();
        System.out.println("线程池被恢复了");

    }
}
```

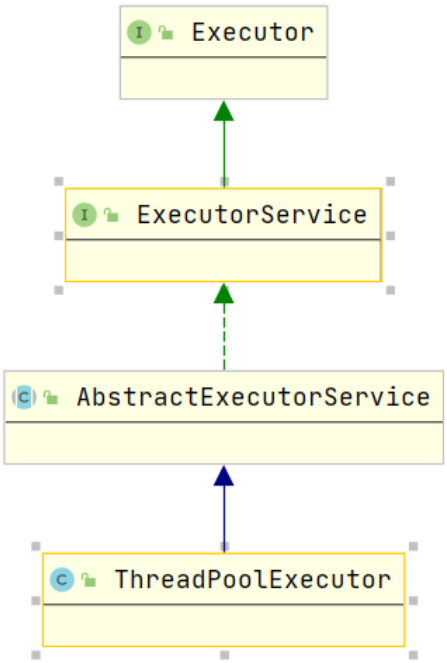
6. 实现原理、源码分析

6.1 线程池组成部分

- 线程池管理器
- 工作线程
- 任务列队任务接口（Task）

6.2 线程池、ThreadPoolExecutor、ExecutorService、Executor、Executors等这么多和线程池相关的类，大家都是什么关系?

####



Executors

`Executor` 是一个抽象层面的顶层接口，在它里面只声明了一个方法 `execute(Runnable)`，返回值为 `void`，参数为 `Runnable` 类型，从字面意思可以理解，就是用来执行传进去的任务的；

`Executor` 将任务本身和执行任务的过程解耦。

```
public interface Executor {

    /**
     * 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 {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute( @NotNull Runnable command);
}
```

ExecutorService

然后 `ExecutorService` 接口继承了 `Executor` 接口，并声明了一些方法：`invokeAll、invokeAny`

提供了返回 Future 对象的submit方法，解决了Runnable无返回值的问题；

提供了关闭线程池等方法，当调用 shutDown方法时，线程池会停止接受新的任务，但会继续执行完毕等待中的任务。

Executors

Executors是一个工具类，就和Collections类似，方便我们来创建常见类型的线程池，例如FixedThreadPool等。

AbstractExecutorService

抽象类`AbstractExecutorService`实现了`ExecutorService`接口，基本实现了`ExecutorService`中声明的所有方法；

然后`ThreadPoolExecutor`继承了类`AbstractExecutorService`，并提供了一些新功能，比如获取核心线程数、获取任务队列等。

线程池实现任务复用的原理

原因

线程重用的核心是，线程池对Thread做了包装，不重复调用thread.start()，而是自己有一个Runnable.run()，run方法里面循环在跑，跑的过程中不断检查我们是否有新加入的子Runnable对象，有新的Runnable进来的话就调一下我们的run()，其实就一个大run()把其它小run()#1,run()#2,...给串联起来了。同一个Thread可以执行不同的Runnable，主要原因是线程池把线程和Runnable通过BlockingQueue给解耦了，线程可以从BlockingQueue中不断获取新的任务。

7. 线程池状态

这是一个巧妙的设计，把同一个int变量，利用了2次，可以同时用高位和低位保存“线程状态”和“线程数”，节省了空间；但是每次取数的时候，要做“与操作”，属于用时间换空间，但是与操作速度是极快的，所以几乎不花费时间。

ctl 共32位，其中高3位表示”线程池状态”，低29位代表”线程池中的任务数量”，线程池状态枚举：

RUNNING：接受新任务并处理排队任务

SHUTDOWN：不接受新任务，但处理排队任务

STOP：不接受新任务，也不处理排队任务，并中断正在进行的任务

TIDYING：过渡状态，中文是整洁，理解了中文就容易理解这个状态了：所有任务都已终止，workerCount为零时，线程会转换到TIDYING状态，并将运行terminate（）钩子方法。

TERMINATED：terminate（）运行完成

runState单调增加，但就和线程状态一样，并不一定会经历到每一个状态。

execute方法

Execute方法这可以说是核心方法，因为这是层层继承过来的，最上面可以追溯到Executor类。

然后我们再来看看ThreadPoolExecutor对execute的实现：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    // 获取线程池的状态和数量，ctl 共32位，其中高3位表示“线程池状态”，低29位代表“线程池中的任务数量”
    int c = ctl.get();
    // 1. workerCountOf(c)从c中获取线程数，判断如果小于核心线程数量，去添加线程，添加成功直接return
    if (workerCountOf(c) < corePoolSize) {
        // addWorker方法第二个参数true表示使用基本大小作为临界值
        if (addWorker(command, true))
            return;
        // 重新获取
        c = ctl.get();
    }
```

```
    // 2. 判断是否有线程在运行且工作队列中可以继续放入任务
    if (isRunning(c) && workQueue.offer(command)) {
        // 因为线程可能中断，需要重新判断
        int recheck = ctl.get();
        // 如果没有运行的线程，才会执行后面的romove，把队列中的task移除
        if (! isRunning(recheck) && remove(command))
            // 执行拒绝策略
            reject(command);
        else if (workerCountOf(recheck) == 0)
            // 是为了避免：线程数为了0但是有task的情况，去添加一个没有task线程
            addWorker(null, false);
    }
    // 3. 直接使用线程池最大大小。addWorker方法第二个参数false表示使用最大大小
    else if (!addWorker(command, false))
        reject(command);
}
```

addWorker推荐源码阅读：https://fangjian0423.github.io/2016/03/22/java-threadpool-analysis/

7. 使用线程池的注意点

避免任务堆积

避免线程数过度增加

排查线程泄漏