线程池较Thread,Runnable的优点

我们有两种常见的创建线程的方法，一种是继承Thread类，一种是实现Runnable接口，但是我们创建这两种线程在运行结束后都会被虚拟机销毁，如果线程数量多的话，频繁的创建和销毁会大大浪费时间和效率，更重要的是浪费内存，因为正常来说线程执行完毕后死亡，线程对象变成垃圾！那么有没有一种方法能让线程运行完后不利己销毁，而是让线程重复使用，可以使用线程池就能很好的解决这个问题。

线程池概要

线程池最上层接口是Executor,这个接口定义了一个核心方法`void execute(Runnable var1);`这个方法最后被ThreadPoolExecutor类实现，这个方法时用来传入任务的，而且ThreadPoolExecutor是线程池的核心类，此类的构造方法如下：

1.第一种

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

2.第二种

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

3.第三种

```
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              RejectedExecutionHandler handler) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
```

```
            Executors.defaultThreadFactory(), handler);
    }
```

4.第四种

```
    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.acc = System.getSecurityManager() == null ?
                null :
                AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

构造方法的参数及意思：

1.corePoolSize： 核心线程池的大小，如果核心线程池有空闲位置，这时新的任务就会被核心线程池新建一个线程执行，执行完毕后不会销毁，线程会进入缓冲队列等待再次被运行。

2.maximumPoolSize： 线程池能创建最大的线程数量，如果核心线程池和缓冲队列都已经满了，新的任务进来就会创建新的线程来执行，但是数量不能超过maximumPoolSize，否则会采取拒绝接受任务策略。

3.keepAliveTime： 非核心线程池能够空闲的最长时间，超过时间，线程终止，这个参数默认只有在线程数量超过核心线程池时才起作用，只要线程数量不超过核心线程大小，就不会起作用。

4.unit: 时间单位，和keepAliveTime配合使用。线程休眠 `TimeUnit.SECONDS.sleep( 5 );`

5.workQueue: 缓冲队列，用来存放等待被执行的任务。

6.threadFactory: 线程工厂，用来创建线程，一般有三种选择策略：

```
ArrayBlockingQueue;
LinkedBlockingQueue;
SynchronousQueue;
```

7.handler: 拒绝处理策略，线程数量大于最大线程数就会采用决绝处理策略，四种策略为：

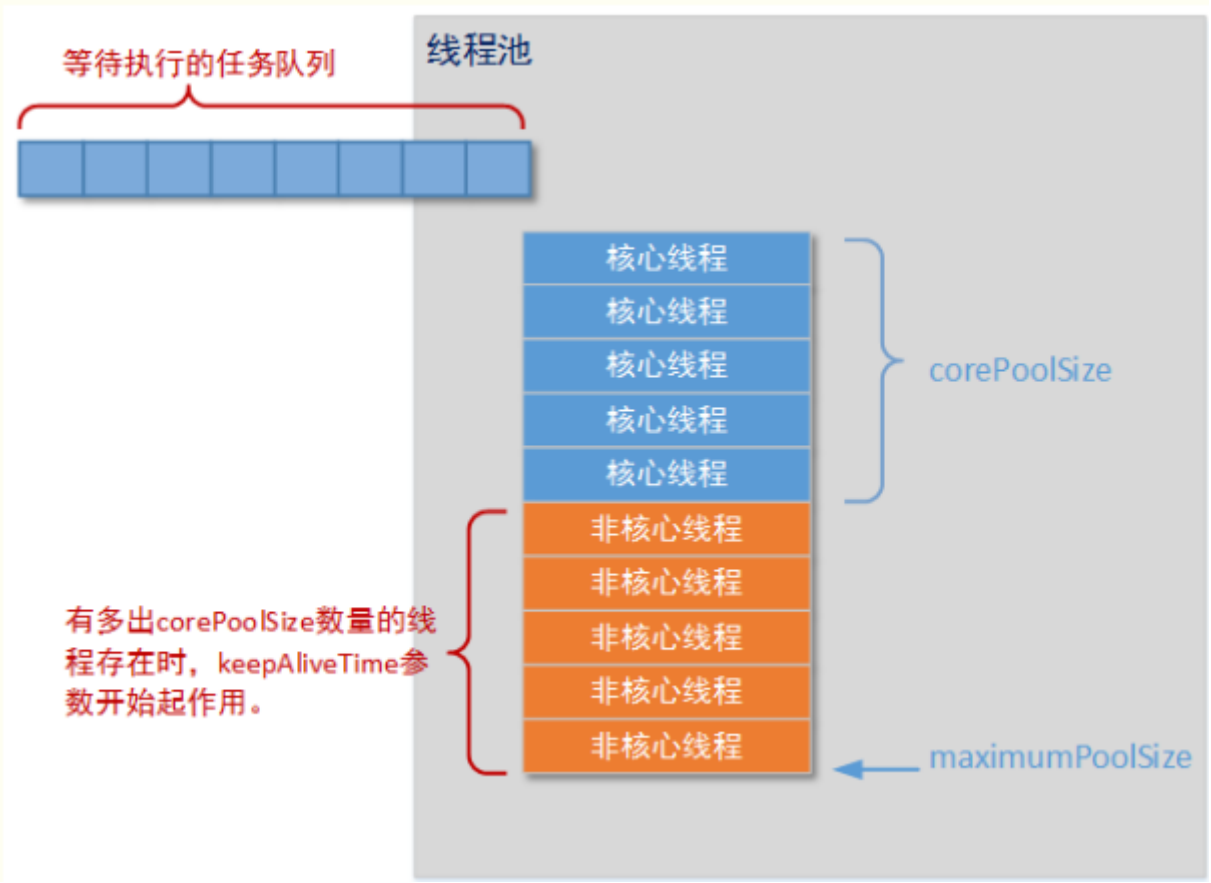
```
ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务
```

ThreadPoolExecutor的源头

`Executor` 接口有一个子接口 `ExecutorService`，`ExecutorService` 的实现类有一个为 `AbstractExecutorService`，`ThreadPoolExecutor` 是 `AbstractExecutorService` 的子类，`ThreadPoolExecutor` 还有两个常用的方法shutdown和submit,两者都用来关闭线程池，但是后者有一个结果返回。

线程池实现原理

线程池图：



1.线程池状态

RUNNING: Accept new tasks and process queued tasks 线程池创建后就一直是RUNNING,接受新的任务，并执行缓冲队列的任务

SHUTDOWN: Don't accept new tasks, but process queued tasks
调用SHUTDOWN后处于SHUTDOWN状态，线程池不能接受新的任务，但是会等待缓冲队列的任务

完成。

STOP: Don't accept new tasks, don't process queued tasks, and interrupt in-progress tasks

调用shutdownNow后处于stop状态，线程池不能接受新的任务，不执行缓冲队列的任务，并尝试终止正在执行的任务。

TIDYING: All tasks have terminated, workerCount is zero, the thread transitioning to state TIDYING will run the terminated() hook method
当前所有的任务已经终止，任务数量为0，线程池状态会变成TIDYING，并且执行terminated()函数。

TERMINATED: terminated() has completed 当线程池处于shutdown或者stop状态，并且所有工作线程已经销毁，任务缓冲队列已经清空或执行结束后，线程池被置为TERMINATED。

线程池总结

1.如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

2.如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

3.如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

4.如果线程池中的线程数量大于corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

线程池实例代码

1.Task类

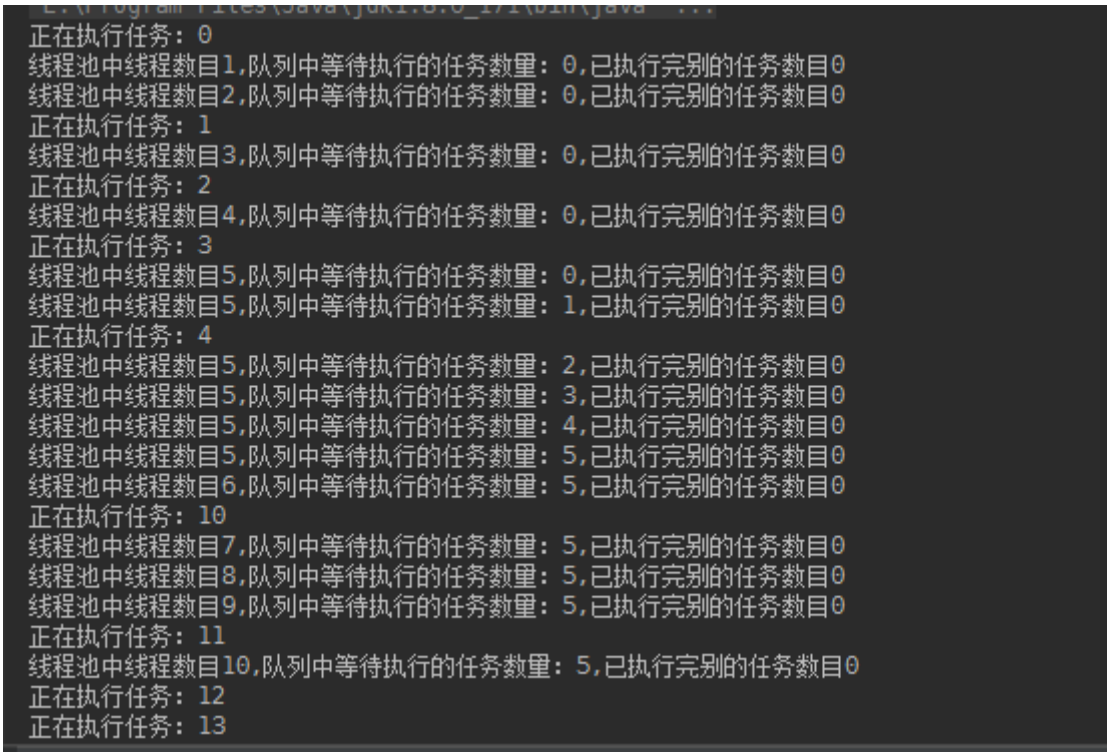
```
package com.hh.threadDemo;

public class Task implements Runnable{

    private int num;

    public Task(int num) {
        this.num = num;
    }

    @Override
    public void run() {
    System.out.println("正在执行任务："+num);
    try{

        Thread.currentThread().sleep(4000);
    }catch(Exception e){
```

```
            e.printStackTrace();
        }
        System.out.println("线程"+num+"执行完毕");
        }
}
```

2.ThreadPoolTest类

```
package com.hh.threadDemo;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolTest {
  public static void main(String[] args) {
      ThreadPoolExecutor pool = new ThreadPoolExecutor(5,10,200,
TimeUnit.MICROSECONDS,new ArrayBlockingQueue<Runnable>(5));

      for (int i =0;i<15;i++){
          Task t = new Task(i);
          pool.execute(t);
      System.out.println("线程池中线程数目"+pool.getPoolSize()+",队列中等待执行的任务数
量："+pool.getQueue().size()+",已执行完别的任务数目"+pool.getCompletedTaskCount());
      }
      pool.shutdown();
  }

}
```

3.运行结果：



我们并不提倡直接使用`ThreadPoolExecutor`，而是使用Executors类中提供的几个静态方法来创建线程池：

new`FixedThreadPool`创建的线程池`corePoolSize`和`maximumPoolSize`值是相等的，它使用的`LinkedBlockingQueue`；

new`SingleThreadExecutor`将`corePoolSize`和`maximumPoolSize`都设置为`1`，也使用的`LinkedBlockingQueue`；

new`CachedThreadPool`将`corePoolSize`设置为`0`，将`maximumPoolSize`设置为`Integer.MAX_VALUE`，使用的`SynchronousQueue`，也就是说来了任务就创建线程运行，当线程空闲超过`60`秒，就销毁线程。