Java：多線程，線程池，ThreadPoolExecutor詳解

cnblogs.com/nayitian/p/3262031.html

1. ThreadPoolExecutor的一個常用的構造方法

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

參數說明：

-corePoolSize　　線程池中所保存的核心線程數。線程池啟動後默認是空的，只有任務來臨時才會創建線程以處理請求。prestartAllCoreThreads方法可以在線程池啟動後即啟動所有核心線程以等待任務。

-maximumPoolSize　　線程池允許創建的最大線程數。當workQueue使用無界隊列時（如：LinkedBlockingQueue），則此參數無效。

-keepAliveTime　　當前線程池線程總數大於核心線程數時，終止多餘的空閒線程的時間。

-unit　　keepAliveTime參數的時間單位。

-workQueue　　工作隊列，如果當前線程池達到核心線程數時（corePoolSize），且當前所有線程都處於活動狀態，則將新加入的任務放到此隊列中。下面僅列幾個常用的：

- ArrayBlockingQueue：　　基於數組結構的有界隊列，此隊列按FIFO原則對任務進行排序。如果隊列滿了還有任務進來，則調用拒絕策略。
- LinkedBlockingQueue：　基於鏈表結構的無界隊列，此隊列按FIFO原則對任務進行排序。因為它是無界的，根本不會滿，所以採用此隊列後線程池將忽略拒絕策略（handler）參數；同時還將忽略最大線程數（maximumPoolSize）等參數。
- SynchronousQueue：　　直接將任務提交給線程而不是將它加入到隊列，實際上此隊列是空的。每個插入的操作必須等到另一個調用移除的操作；如果新任務來了線程池沒有任何可用線程處理的話，則調用拒絕策略。其實要是把maximumPoolSize設置成無界（Integer.MAX_VALUE）的，加上SynchronousQueue隊列，就等同於Executors.newCachedThreadPool()。
- PriorityBlockingQueue：　具有優先級的隊列的有界隊列，可以自定義優先級；默認是按自然排序，可能很多場合並不合適。

-handler　　拒絕策略，當線程池與workQueue隊列都滿了的情況下，對新加任務採取的策略。

- AbortPolicy：　　拒絕任務，拋出`RejectedExecutionException`異常。默認值。
- CallerRunsPolicy：　A handler for rejected tasks that runs the rejected task directly in the calling thread of the execute method, unless the executor has been shut down, in which case the task is discarded（沒太弄懂意思，看不太懂，程序模擬半天也沒得出啥結論。）
- DiscardOldestPolicy：　如果執行程序尚未關閉，則位於工作隊列頭部的任務將被刪除，然後重試執行程序（如果再次失敗，則重複此過程）。這樣的結果是最後加入的任務反而有可能被執行到，先前加入的都被拋棄了。
- DiscardPolicy：　　加不進的任務都被拋棄了，同時沒有異常拋出。

2. 詳解及示範

1. 一個任務進來（Runnable）時，如果核心線程數（corePoolSize）未達到，則直接創建線程處理該任務；如果核心線程數已經達到則該任務進入工作隊列（workQueue）。如果工作隊列滿了（只能是有界隊列），則檢查最大線程數（maximumPoolSize）是否達到，如果沒達到則創建線程處理任務（FIFO）；如果最大線程數據也達到了，則調用拒絕策略（handler）。
2. 如果workQueue使用LinkedBlockingQueue隊列，因為它是無界的，隊列永遠不會滿，所以maximumPoolSize參數是沒有意義的，同樣keepAliveTime、unit、handler三個參數都無意義。
3. 如果workQueue使用ArrayBlockingQueue隊列，那麼小心，因為此隊列是有界的，必須小心處理拒絕策略。你看人家Executors類，壓根就不使用ArrayBlockingQueue隊列。
4. 正常情況下，如果使用Executors靜態工廠生成的幾種常用線程池能夠滿足要求，建議就用它們吧。自己控制所有細節挺不容易的。

```
package com.clzhang.sample.thread;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolTest3 {
    static class MyThread implements Runnable {
        private String name;

        public MyThread(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            // 做點事情
            try {
                Thread.sleep(1000);

                System.out.println(name + " finished job!") ;
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        // 創建線程池，為了更好的明白運行流程，增加了一些額外的代碼
//        BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(2);
        BlockingQueue<Runnable> queue = new LinkedBlockingQueue<Runnable>();
//        BlockingQueue<Runnable> queue = new PriorityBlockingQueue<Runnable>();
```

```
//        BlockingQueue<Runnable> queue = new SynchronousQueue<Runnable>();


        // AbortPolicy/CallerRunsPolicy/DiscardOldestPolicy/DiscardPolicy
        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(2, 4, 5,
TimeUnit.SECONDS,
                queue, new ThreadPoolExecutor.AbortPolicy());

        // 向線程池裡面扔任務
        for (int i = 0; i < 10; i++) {
            System.out.println("當前線程池大小[" + threadPool.getPoolSize() + "],當前隊
列大小[" + queue.size() + "]");

            threadPool.execute(new MyThread("Thread" + i));
        }
        // 關閉線程池
        threadPool.shutdown();
    }
}
```

輸出(採用LinkedBlockingQueue隊列)：

當前線程池大小[0],當前隊列大小[0]
當前線程池大小[1],當前隊列大小[0]
當前線程池大小[2],當前隊列大小[0]
當前線程池大小[2],當前隊列大小[1]
當前線程池大小[2],當前隊列大小[2]
當前線程池大小[2],當前隊列大小[3]
當前線程池大小[2],當前隊列大小[4]
當前線程池大小[2],當前隊列大小[5]
當前線程池大小[2],當前隊列大小[6]
當前線程池大小[2],當前隊列大小[7]
Thread1 finished job!
Thread0 finished job!
Thread3 finished job!
Thread2 finished job!
Thread4 finished job!
Thread5 finished job!
Thread6 finished job!
Thread7 finished job!
Thread8 finished job!
Thread9 finished job!

3. 回頭看看Executors靜態工廠方法生成線程池的源代碼

Executors.newSingleThreadExecutor()

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

Executors.newFixedThreadPool()

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

Executors.newCachedThreadPool()

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

通過上述代碼可以發現，用Executors靜態工廠生成的幾種常用線程池，都可以向裡面插入n多任務：要麼workQueue是無界的，要麼maximumPoolSize是無界的。