Medusar's Blog

> *問題描述：*
> *假設我們有一個線程池，由於程序需要，我們向該線程池中提交了好多任務，但是這些任務都沒有對異常進行try catch處理，並且運行的時候都拋出了異常。這會對線程池的運行帶來什麼影響？*

正確答案是：沒有影響。
想一下，如果是你開發了一個線程池供開發者使用，你會不會對這種情況做處理？想想也是肯定的，不然你提供給別人使用的東西就是有問題的，欠考慮的。而且java線程池的主要開發人員是大名鼎鼎的Doug Lea，你覺得他開發的代碼怎麼會允許出現這種問題？

問題分析

接下來我們來看一下java中的線程池是如何運行我們提交的任務的，詳細流程比較複雜，這裡我們不關注，我們只關注任務執行的部分。java中的線程池用的是ThreadPoolExecutor，真正執行代碼的部分是runWorker方法：`final void runWorker(Worker w)`

```
1.  //省略無關部分
2.  try {
3.      beforeExecute(wt, task);
4.      Throwable thrown = null;
5.      try {
6.          task.run();  //執行程序邏輯
7.      } catch (RuntimeException x) { //捕獲RuntimeException
8.          thrown = x; throw x;
9.      } catch (Error x) { //捕獲Error
0.          thrown = x; throw x;
1.      } catch (Throwable x) {    //捕獲Throwable
2.          thrown = x; throw new Error(x);
3.      } finally {
4.          afterExecute(task, thrown);  //運行完成，進行後續處理
5.      }
6.  } finally {
7.      task = null;
8.      w.completedTasks++;
9.      w.unlock();
0.  }
1.  //省略無關部分
```

可以看到，程序會捕獲包括Error在內的所有異常，並且在程序最後，將出現過的異常和當前任務傳遞給afterExecute方法。

而ThreadPoolExecutor中的afterExecute方法是沒有任何實現的：

```
1.  protected void afterExecute(Runnable r, Throwable t) { }
```

也就是說，默認情況下，線程池會捕獲任務拋出的所有異常，但是不做任何處理。

存在問題

想像下ThreadPoolExecutor這種處理方式會有什麼問題？
這樣做能夠保證我們提交的任務拋出了異常不會影響其他任務的執行，同時也不會對用來執行該任務的線程產生任何影響。
問題就在`afterExecute`方法上，這個方法沒有做任何處理，所以如果我們的任務拋出了異常，我們也無法立刻感知到。即使感知到了，也無法查看異常信息。

所以，作為一名好的開發者，是不應該允許這種情況出現的。

如何避免這種問題

思路很簡單。
1. 在提交的任務中將異常捕獲並處理，不拋給線程池。
2. 異常拋給線程池，但是我們要及時處理拋出的異常。

第一種思路很簡單，就是我們提交任務的時候，將所有可能的異常都Catch住，並且自己處理，任務的大致代碼如下：

```
1.  @Override
2.      public void run() {
3.          try {
4.              //處理所有的業務邏輯
5.          } catch (Throwable e) {
6.              //打印日誌等
7.          } finally {
8.              //其他處理
9.          }
0.      }
```

說白了就是把業務邏輯都trycatch起來。
但是這種思路的缺點就是：1）所有的不同任務類型都要trycatch，增加了代碼量。2）不存在checkedexception的地方也需要都trycatch起來，代碼醜陋。

第二種思路就可以避免上面的兩個問題。
第二種思路又有以下幾種實現方式：
1. 自定義線程池，繼承ThreadPoolExecutor並複寫其`afterExecute(Runnable r, Throwable t)`方法。
2. 實現Thread.UncaughtExceptionHandler接口，實現`void uncaughtException(Thread t, Throwable e);`方法，並將該handler傳遞給線程池的ThreadFactory
3. 採用Future模式，將返回結果以及異常放到Future中，在Future中處理

4. 繼承ThreadGroup，覆蓋其uncaughtException方法。（與第二種方式類似，因為ThreadGroup類本身就實現了Thread.UncaughtExceptionHandler接口）

下面是以上幾種方式的代碼

方式1

自定義線程池：

```
1.  final class PoolService {
2.    // The values have been hard-coded for brevity
3.    ExecutorService pool = new CustomThreadPoolExecutor(
4.        10, 10, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(10));
5.    // ...
6.  }
7.  class CustomThreadPoolExecutor extends ThreadPoolExecutor {
8.    // ... Constructor ...
9.    public CustomThreadPoolExecutor(
0.        int corePoolSize, int maximumPoolSize, long keepAliveTime,
1.        TimeUnit unit, BlockingQueue<Runnable> workQueue) {
2.      super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
3.    }
4.    @Override
5.    public void afterExecute(Runnable r, Throwable t) {
6.      super.afterExecute(r, t);
7.      if (t != null) {
8.        // Exception occurred, forward to handler
9.      }
0.      // ... Perform task-specific cleanup actions
1.    }
2.    @Override
3.    public void terminated() {
4.      super.terminated();
5.      // ... Perform final clean-up actions
6.    }
7.  }
```

方式2

實現Thread.UncaughtExceptionHandler接口，實現`void uncaughtException(Thread t, Throwable e);`方法，並將該handler傳遞給線程池的ThreadFactory

```
1.  final class PoolService {
2.    private static final ThreadFactory factory =
3.        new ExceptionThreadFactory(new MyExceptionHandler());
4.    private static final ExecutorService pool =
5.        Executors.newFixedThreadPool(10, factory);
6.    public void doSomething() {
7.      pool.execute(new Task()); // Task is a runnable class
8.    }
9.    public static class ExceptionThreadFactory implements ThreadFactory  {
0.      private static final ThreadFactory defaultFactory =
1.          Executors.defaultThreadFactory();
2.      private final Thread.UncaughtExceptionHandler handler;
3.      public ExceptionThreadFactory(
4.          Thread.UncaughtExceptionHandler handler)
5.      {
6.        this.handler = handler;
7.      }
8.      @Override public Thread newThread(Runnable run) {
9.        Thread thread = defaultFactory.newThread(run);
0.        thread.setUncaughtExceptionHandler(handler);
1.        return thread;
2.      }
3.    }
4.    public static class MyExceptionHandler extends ExceptionReporter
5.        implements Thread.UncaughtExceptionHandler {
6.      // ...
7.      @Override public void uncaughtException(Thread thread, Throwable t) {
8.        // Recovery or logging code
9.      }
0.    }
1.  }
```

方式3

繼承ThreadGroup，覆蓋其uncaughtException方法

```
1.  public class ThreadGroupExample {
2.      public static class MyThreadGroup extends ThreadGroup {
3.          public MyThreadGroup(String s) {
```

```
4.              super(s);
5.          }
6.          public void uncaughtException(Thread thread, Throwable throwable) {
7.              System.out.println("Thread " + thread.getName()
8.                + " died, exception was: ");
9.              throwable.printStackTrace();
0.          }
1.      }
2.      public static ThreadGroup workerThreads =
3.        new MyThreadGroup("Worker Threads");
4.      public static class WorkerThread extends Thread {
5.          public WorkerThread(String s) {
6.              super(workerThreads, s);
7.          }
8.          public void run() {
9.              throw new RuntimeException();
0.          }
1.      }
2.      public static void main(String[] args) {
3.          Thread t = new WorkerThread("Worker Thread");
4.          t.start();
5.      }
6.  }
```

確實這種方式與上面通過ThreadFactory來指定UncaughtExceptionHandler是一樣的，只是代碼邏輯不同，但原理上都是一樣的，即給線程池中的每個線程都指定一個UncaughtExceptionHandler。

** 注意：上面三種方式針對的都是通過execute(xx)的方式提交任務，如果你提交任務用的是submit()方法，那麼上面的三種方式都將不起作用,而應該使用下面的方式 **

方式4

如果提交任務的時候使用的方法是submit，那麼該方法將返回一個Future對象，所有的異常以及處理結果都可以通過future對象獲取。
採用Future模式，將返回結果以及異常放到Future中，在Future中處理

```
1.  final class PoolService {
2.    private final ExecutorService pool = Executors.newFixedThreadPool(10);
3.    public void doSomething() {
4.      Future<?> future = pool.submit(new Task());
5.      // ...
```

```
6.      try {
7.        future.get();
8.      } catch (InterruptedException e) {
9.        Thread.currentThread().interrupt(); // Reset interrupted status
0.      } catch (ExecutionException e) {
1.        Throwable exception = e.getCause();
2.        // Forward to exception reporter
3.      }
4.    }
5.  }
```

總結

1. java線程池會捕獲任務拋出的異常和錯誤，但不做任何處理
2. 好的程序設計應該考慮到對於類異常的處理
3. 處理線程池中的異常有兩種思路：
   1）提交到線程池中的任務自己捕獲異常並處理，不拋給線程池
   2）由線程池統一處理
4. 對於execute方法提交的線程，有兩種處理方式
   1）自定義線程池並實現afterExecute方法
   2）給線程池中的每個線程指定一個UncaughtExceptionHandler,由handler來統一處理異常。
5. 對於submit方法提交的任務，異常處理是通過返回的Future對象進行的。