https://www.cnblogs.com/wanqieddy/p/3853863.html

[**ExecutorService中submit和execute的区别**](https://www.cnblogs.com/wanqieddy/p/3853863.html)

在Java5之后，并发线程这块发生了根本的变化，最重要的莫过于新的启动、调度、管理线程的一大堆API了。在Java5以后，通过Executor来启动线程比用Thread的start()更好。在新特征中，可以很容易控制线程的启动、执行和关闭过程，还可以很容易使用线程池的特性。

**一、创建任务**

任务就是一个实现了Runnable接口的类。

创建的时候实run方法即可。

**二、执行任务**

通过java.util.concurrent.ExecutorService接口对象来执行任务，该接口对象通过工具类java.util.concurrent.Executors的静态方法来创建。

Executors此包中所定义的 Executor、ExecutorService、ScheduledExecutorService、ThreadFactory 和 Callable 类的工厂和实用方法。

ExecutorService提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。 可以关闭 ExecutorService，这将导致其停止接受新任务。关闭后，执行程序将最后终止，这时没有任务在执行，也没有任务在等待执行，并且无法提交新任务。

            executorService.execute(new TestRunnable());

1、创建ExecutorService

通过工具类java.util.concurrent.Executors的静态方法来创建。

Executors此包中所定义的 Executor、ExecutorService、ScheduledExecutorService、ThreadFactory 和 Callable 类的工厂和实用方法。

比如，创建一个ExecutorService的实例，ExecutorService实际上是一个线程池的管理工具：

        ExecutorService executorService = Executors.newCachedThreadPool();

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        ExecutorService executorService = Executors.newSingleThreadExecutor();

2、将任务添加到线程去执行

当将一个任务添加到线程池中的时候，线程池会为每个任务创建一个线程，该线程会在之后的某个时刻自动执行。

**三、关闭执行服务对象**

        executorService.shutdown();

**四、综合实例**

package concurrent;   
  
import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
  
/\*\*   
\* Created by IntelliJ IDEA.   
\*   
\* @author leizhimin 2008-11-25 14:28:59   
\*/  
publicclass TestCachedThreadPool {   
        publicstaticvoid main(String[] args) {   
//                ExecutorService executorService = Executors.newCachedThreadPool();  
                ExecutorService executorService = Executors.newFixedThreadPool(5);

//         ExecutorService executorService = Executors.newSingleThreadExecutor();  
  
                for (int i = 0; i < 5; i++) {   
                        executorService.execute(new TestRunnable());   
                        System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* a" + i + " \*\*\*\*\*\*\*\*\*\*\*\*\*");   
                }   
                executorService.shutdown();   
        }   
}   
  
class TestRunnable implements Runnable {   
        publicvoid run() {   
                System.out.println(Thread.currentThread().getName() + "线程被调用了。");   
                while (true) {   
                        try {   
                                Thread.sleep(5000);   
                                System.out.println(Thread.currentThread().getName());   
                        } catch (InterruptedException e) {   
                                e.printStackTrace();   
                        }   
                }   
        }   
}

运行结果：

\*\*\*\*\*\*\*\*\*\*\*\*\* a0 \*\*\*\*\*\*\*\*\*\*\*\*\*   
\*\*\*\*\*\*\*\*\*\*\*\*\* a1 \*\*\*\*\*\*\*\*\*\*\*\*\*   
pool-1-thread-2线程被调用了。   
\*\*\*\*\*\*\*\*\*\*\*\*\* a2 \*\*\*\*\*\*\*\*\*\*\*\*\*   
pool-1-thread-3线程被调用了。   
pool-1-thread-1线程被调用了。   
\*\*\*\*\*\*\*\*\*\*\*\*\* a3 \*\*\*\*\*\*\*\*\*\*\*\*\*   
\*\*\*\*\*\*\*\*\*\*\*\*\* a4 \*\*\*\*\*\*\*\*\*\*\*\*\*   
pool-1-thread-4线程被调用了。   
pool-1-thread-5线程被调用了。   
pool-1-thread-2   
pool-1-thread-1   
pool-1-thread-3   
pool-1-thread-5   
pool-1-thread-4   
pool-1-thread-2   
pool-1-thread-1   
pool-1-thread-3   
pool-1-thread-5   
pool-1-thread-4   
     ......

五、获取任务的执行的返回值

在Java5之后，任务分两类：一类是实现了Runnable接口的类，一类是实现了Callable接口的类。两者都可以被ExecutorService执行，但是Runnable任务没有返回值，而Callable任务有返回值。并且Callable的call()方法只能通过ExecutorService的(<T> task) 方法来执行，并且返回一个 <T><T>，是表示任务等待完成的 Future。

public interface **Callable<V>**

返回结果并且可能抛出异常的任务。实现者定义了一个不带任何参数的叫做 call 的方法。

Callable 接口类似于，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常。

类包含一些从其他普通形式转换成 Callable 类的实用方法。

Callable中的call()方法类似Runnable的run()方法，就是前者有返回值，后者没有。

当将一个Callable的对象传递给ExecutorService的submit方法，则该call方法自动在一个线程上执行，并且会返回执行结果Future对象。

同样，将Runnable的对象传递给ExecutorService的submit方法，则该run方法自动在一个线程上执行，并且会返回执行结果Future对象，但是在该Future对象上调用get方法，将返回null。

遗憾的是，在Java API文档中，这块介绍的很糊涂，估计是翻译人员还没搞清楚的缘故吧。或者说是注释不到位。下面看个例子：

import java.util.ArrayList;   
import java.util.List;   
import java.util.concurrent.\*;   
  
/\*\*   
\* Callable接口测试   
\*   
\* @author leizhimin 2008-11-26 9:20:13   
\*/  
publicclass CallableDemo {   
        publicstaticvoid main(String[] args) {   
                ExecutorService executorService = Executors.newCachedThreadPool();   
                List<Future<String>> resultList = new ArrayList<Future<String>>();   
  
                //创建10个任务并执行  
                for (int i = 0; i < 10; i++) {   
                        //使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中  
                        Future<String> future = executorService.submit(new TaskWithResult(i));   
                        //将任务执行结果存储到List中  
                        resultList.add(future);   
                }   
  
                //遍历任务的结果  
                for (Future<String> fs : resultList) {   
                        try {   
                                System.out.println(fs.get());     //打印各个线程（任务）执行的结果  
                        } catch (InterruptedException e) {   
                                e.printStackTrace();   
                        } catch (ExecutionException e) {   
                                e.printStackTrace();   
                        } finally {   
                                //启动一次顺序关闭，执行以前提交的任务，但不接受新任务。如果已经关闭，则调用没有其他作用。  
                                executorService.shutdown();   
                        }   
                }   
        }   
}   
  
  
class TaskWithResult implements Callable<String> {   
        privateint id;   
  
        public TaskWithResult(int id) {   
                this.id = id;   
        }   
  
        /\*\*   
         \* 任务的具体过程，一旦任务传给ExecutorService的submit方法，则该方法自动在一个线程上执行。   
         \*   
         \* @return   
         \* @throws Exception   
         \*/  
        public String call() throws Exception {   
                System.out.println("call()方法被自动调用,干活！！！             " + Thread.currentThread().getName());   
                //一个模拟耗时的操作  
                for (int i = 999999; i > 0; i--) ;   
                return"call()方法被自动调用，任务的结果是：" + id + "    " + Thread.currentThread().getName();   
        }   
}

运行结果：

call()方法被自动调用,干活！！！             pool-1-thread-1   
call()方法被自动调用,干活！！！             pool-1-thread-3   
call()方法被自动调用,干活！！！             pool-1-thread-4   
call()方法被自动调用,干活！！！             pool-1-thread-6   
call()方法被自动调用,干活！！！             pool-1-thread-2   
call()方法被自动调用,干活！！！             pool-1-thread-5   
call()方法被自动调用，任务的结果是：0    pool-1-thread-1   
call()方法被自动调用，任务的结果是：1    pool-1-thread-2   
call()方法被自动调用,干活！！！             pool-1-thread-2   
call()方法被自动调用,干活！！！             pool-1-thread-6   
call()方法被自动调用,干活！！！             pool-1-thread-4   
call()方法被自动调用，任务的结果是：2    pool-1-thread-3   
call()方法被自动调用,干活！！！             pool-1-thread-3   
call()方法被自动调用，任务的结果是：3    pool-1-thread-4   
call()方法被自动调用，任务的结果是：4    pool-1-thread-5   
call()方法被自动调用，任务的结果是：5    pool-1-thread-6   
call()方法被自动调用，任务的结果是：6    pool-1-thread-2   
call()方法被自动调用，任务的结果是：7    pool-1-thread-6   
call()方法被自动调用，任务的结果是：8    pool-1-thread-4   
call()方法被自动调用，任务的结果是：9    pool-1-thread-3   
  
Process finished with exit code 0  
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

因为之前一直是用的execute方法，最近有个情况需要用到submit方法，所以研究了下。

三个区别：

1、接收的参数不一样

2、submit有返回值，而execute没有

Method submit extends base method Executor.execute by creating and returning a Future that can be used to cancel execution and/or wait for completion.

用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。然后我就可以把所有失败的原因综合起来发给调用者。

个人觉得cancel execution这个用处不大，很少有需要去取消执行的。

而最大的用处应该是第二点。

3、submit方便Exception处理

There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit **any** thrown exception, checked or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will rethrow this exception, wrapped in an ExecutionException.

意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

比如说，我有很多更新各种数据的task，我希望如果其中一个task失败，其它的task就不需要执行了。那我就需要catch Future.get抛出的异常，然后终止其它task的执行，代码如下：

51cto上有一篇非常好的文章“Java5并发学习”（<http://lavasoft.blog.51cto.com/62575/115112>），下面的代码是基于它之上修改的。

**[java]** [view plaincopy](http://blog.csdn.net/peachpi/article/details/6771946)

1. **import java.util.ArrayList;**
2. **import java.util.List;**
3. **import java.util.Random;**
4. **import java.util.concurrent.Callable;**
5. **import java.util.concurrent.ExecutionException;**
6. **import java.util.concurrent.ExecutorService;**
7. **import java.util.concurrent.Executors;**
8. **import java.util.concurrent.Future;**
10. **public class ExecutorServiceTest {**
11. **public static void main(String[] args) {**
12. ExecutorService executorService = Executors.newCachedThreadPool();
13. List<Future<String>> resultList = **new ArrayList<Future<String>>();**
15. // 创建10个任务并执行
16. **for (int i = 0; i < 10; i++) {**
17. // 使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中
18. Future<String> future = executorService.submit(**new TaskWithResult(i));**
19. // 将任务执行结果存储到List中
20. resultList.add(future);
21. }
22. executorService.shutdown();
24. // 遍历任务的结果
25. **for (Future<String> fs : resultList) {**
26. **try {**
27. System.out.println(fs.get()); // 打印各个线程（任务）执行的结果
28. } **catch (InterruptedException e) {**
29. e.printStackTrace();
30. } **catch (ExecutionException e) {**
31. executorService.shutdownNow();
32. e.printStackTrace();
33. **return;**
34. }
35. }
36. }
37. }
39. **class TaskWithResult implements Callable<String> {**
40. **private int id;**
42. **public TaskWithResult(int id) {**
43. **this.id = id;**
44. }
46. /\*\*
47. \* 任务的具体过程，一旦任务传给ExecutorService的submit方法，则该方法自动在一个线程上执行。
48. \*
49. \* @return
50. \* @throws Exception
51. \*/
52. **public String call() throws Exception {**
53. System.out.println("call()方法被自动调用,干活！！！             " + Thread.currentThread().getName());
54. **if (new Random().nextBoolean())**
55. **throw new TaskException("Meet error in task." + Thread.currentThread().getName());**
56. // 一个模拟耗时的操作
57. **for (int i = 999999999; i > 0; i--)**
58. ;
59. **return "call()方法被自动调用，任务的结果是：" + id + "    " + Thread.currentThread().getName();**
60. }
61. }
63. **class TaskException extends Exception {**
64. **public TaskException(String message) {**
65. **super(message);**
66. }
67. }

执行的结果类似于：

**[java]** [view plaincopy](http://blog.csdn.net/peachpi/article/details/6771946)

1. call()方法被自动调用,干活！！！             pool-1-thread-1
2. call()方法被自动调用,干活！！！             pool-1-thread-2
3. call()方法被自动调用,干活！！！             pool-1-thread-3
4. call()方法被自动调用,干活！！！             pool-1-thread-5
5. call()方法被自动调用,干活！！！             pool-1-thread-7
6. call()方法被自动调用,干活！！！             pool-1-thread-4
7. call()方法被自动调用,干活！！！             pool-1-thread-6
8. call()方法被自动调用,干活！！！             pool-1-thread-7
9. call()方法被自动调用,干活！！！             pool-1-thread-5
10. call()方法被自动调用,干活！！！             pool-1-thread-8
11. call()方法被自动调用，任务的结果是：0    pool-1-thread-1
12. call()方法被自动调用，任务的结果是：1    pool-1-thread-2
13. java.util.concurrent.ExecutionException: com.cicc.pts.TaskException: Meet error in task.pool-1-thread-3
14. at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:222)
15. at java.util.concurrent.FutureTask.get(FutureTask.java:83)
16. at com.cicc.pts.ExecutorServiceTest.main(ExecutorServiceTest.java:29)
17. Caused by: com.cicc.pts.TaskException: Meet error in task.pool-1-thread-3
18. at com.cicc.pts.TaskWithResult.call(ExecutorServiceTest.java:57)
19. at com.cicc.pts.TaskWithResult.call(ExecutorServiceTest.java:1)
20. at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
21. at java.util.concurrent.FutureTask.run(FutureTask.java:138)
22. at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
23. at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
24. at java.lang.Thread.run(Thread.java:619)

可以看见一旦某个task出错，其它的task就停止执行。