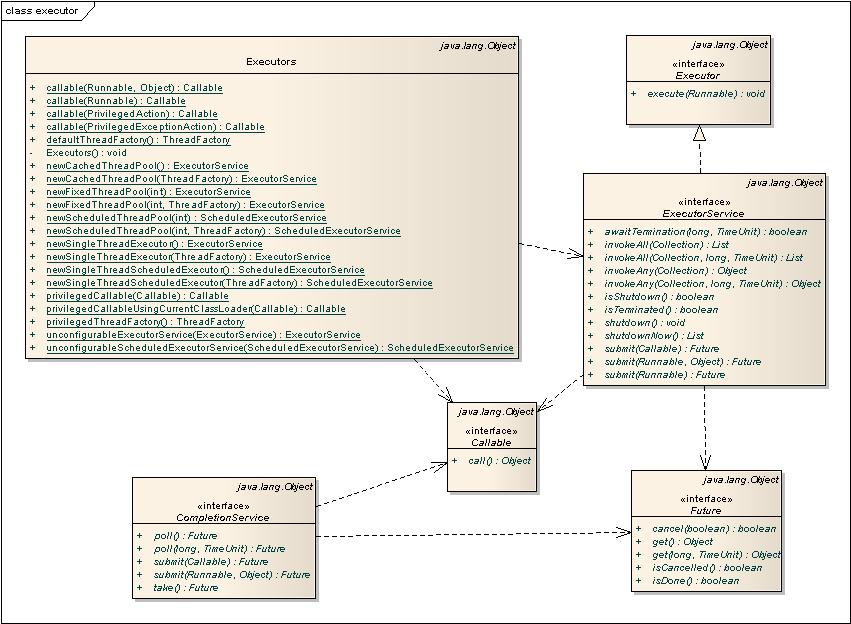
Executor框架是指java 5中引入的一系列并发库中与executor相关的一些功能类，其中包括线程池，Executor，Executors，ExecutorService，CompletionService，Future，Callable等。他们的关系为：



并发编程的一种编程方式是把任务拆分为一些列的小任务，即Runnable，然后在提交给一个Executor执行，**Executor.execute(Runnalbe)** 。Executor在执行时使用内部的线程池完成操作。

**一、创建线程池**

Executors类，提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

**public static ExecutorService newFixedThreadPool(int nThreads)**

创建固定数目线程的线程池。

**public static ExecutorService newCachedThreadPool()**

创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

**public static ExecutorService newSingleThreadExecutor()**

创建一个单线程化的Executor。

**public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

**Java代码  [收藏代码](javascript:void())**

1. Executor executor = Executors.newFixedThreadPool(10);
2. Runnable task = **new** Runnable() {
3. @Override
4. **public** **void** run() {
5. System.out.println("task over");
6. }
7. };
8. executor.execute(task);
10. executor = Executors.newScheduledThreadPool(10);
11. ScheduledExecutorService scheduler = (ScheduledExecutorService) executor;
12. scheduler.scheduleAtFixedRate(task, 10, 10, TimeUnit.SECONDS);

**二、ExecutorService与生命周期**

ExecutorService扩展了Executor并添加了一些生命周期管理的方法。一个Executor的生命周期有三种状态，**运行** ，**关闭** ，**终止** 。Executor创建时处于运行状态。当调用ExecutorService.shutdown()后，处于关闭状态，isShutdown()方法返回true。这时，不应该再想Executor中添加任务，所有已添加的任务执行完毕后，Executor处于终止状态，isTerminated()返回true。

如果Executor处于关闭状态，往Executor提交任务会抛出unchecked exception RejectedExecutionException。

**Java代码  [收藏代码](javascript:void())**

1. ExecutorService executorService = (ExecutorService) executor;
2. **while** (!executorService.isShutdown()) {
3. **try** {
4. executorService.execute(task);
5. } **catch** (RejectedExecutionException ignored) {
7. }
8. }
9. executorService.shutdown();

**三、使用Callable，Future返回结果**

Future<V>代表一个异步执行的操作，通过get()方法可以获得操作的结果，如果异步操作还没有完成，则，get()会使当前线程阻塞。FutureTask<V>实现了Future<V>和Runable<V>。Callable代表一个有返回值得操作。

**Java代码  [收藏代码](javascript:void())**

1. Callable<Integer> func = **new** Callable<Integer>(){
2. **public** Integer call() **throws** Exception {
3. System.out.println("inside callable");
4. Thread.sleep(1000);
5. **return** **new** Integer(8);
6. }
7. };
8. FutureTask<Integer> futureTask  = **new** FutureTask<Integer>(func);
9. Thread newThread = **new** Thread(futureTask);
10. newThread.start();
12. **try** {
13. System.out.println("blocking here");
14. Integer result = futureTask.get();
15. System.out.println(result);
16. } **catch** (InterruptedException ignored) {
17. } **catch** (ExecutionException ignored) {
18. }

 ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

**例子：并行计算数组的和。**

**Java代码  [收藏代码](javascript:void())**

1. **package** executorservice;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
5. **import** java.util.concurrent.Callable;
6. **import** java.util.concurrent.ExecutionException;
7. **import** java.util.concurrent.ExecutorService;
8. **import** java.util.concurrent.Executors;
9. **import** java.util.concurrent.Future;
10. **import** java.util.concurrent.FutureTask;
12. **public** **class** ConcurrentCalculator {
14. **private** ExecutorService exec;
15. **private** **int** cpuCoreNumber;
16. **private** List<Future<Long>> tasks = **new** ArrayList<Future<Long>>();
18. // 内部类
19. **class** SumCalculator **implements** Callable<Long> {
20. **private** **int**[] numbers;
21. **private** **int** start;
22. **private** **int** end;
24. **public** SumCalculator(**final** **int**[] numbers, **int** start, **int** end) {
25. **this**.numbers = numbers;
26. **this**.start = start;
27. **this**.end = end;
28. }
30. **public** Long call() **throws** Exception {
31. Long sum = 0l;
32. **for** (**int** i = start; i < end; i++) {
33. sum += numbers[i];
34. }
35. **return** sum;
36. }
37. }
39. **public** ConcurrentCalculator() {
40. cpuCoreNumber = Runtime.getRuntime().availableProcessors();
41. exec = Executors.newFixedThreadPool(cpuCoreNumber);
42. }
44. **public** Long sum(**final** **int**[] numbers) {
45. // 根据CPU核心个数拆分任务，创建FutureTask并提交到Executor
46. **for** (**int** i = 0; i < cpuCoreNumber; i++) {
47. **int** increment = numbers.length / cpuCoreNumber + 1;
48. **int** start = increment \* i;
49. **int** end = increment \* i + increment;
50. **if** (end > numbers.length)
51. end = numbers.length;
52. SumCalculator subCalc = **new** SumCalculator(numbers, start, end);
53. FutureTask<Long> task = **new** FutureTask<Long>(subCalc);
54. tasks.add(task);
55. **if** (!exec.isShutdown()) {
56. exec.submit(task);
57. }
58. }
59. **return** getResult();
60. }
62. /\*\*
63. \* 迭代每个只任务，获得部分和，相加返回
64. \*
65. \* @return
66. \*/
67. **public** Long getResult() {
68. Long result = 0l;
69. **for** (Future<Long> task : tasks) {
70. **try** {
71. // 如果计算未完成则阻塞
72. Long subSum = task.get();
73. result += subSum;
74. } **catch** (InterruptedException e) {
75. e.printStackTrace();
76. } **catch** (ExecutionException e) {
77. e.printStackTrace();
78. }
79. }
80. **return** result;
81. }
83. **public** **void** close() {
84. exec.shutdown();
85. }
86. }

**Main**

**Java代码  [收藏代码](javascript:void())**

1. **int**[] numbers = **new** **int**[] { 1, 2, 3, 4, 5, 6, 7, 8, 10, 11 };
2. ConcurrentCalculator calc = **new** ConcurrentCalculator();
3. Long sum = calc.sum(numbers);
4. System.out.println(sum);
5. calc.close();

**四、CompletionService**

在刚在的例子中，getResult()方法的实现过程中，迭代了FutureTask的数组，如果任务还没有完成则当前线程会阻塞，如果我们希望任意字任务完成后就把其结果加到result中，而不用依次等待每个任务完成，可以使CompletionService。生产者submit()执行的任务。使用者take()已完成的任务，**并按照完成这些任务的顺序处理它们的结果** 。也就是调用CompletionService的take方法是，会返回按完成顺序放回任务的结果，CompletionService内部维护了一个阻塞队列BlockingQueue，如果没有任务完成，take()方法也会阻塞。修改刚才的例子使用CompletionService：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** ConcurrentCalculator2 {
3. **private** ExecutorService exec;
4. **private** CompletionService<Long> completionService;

7. **private** **int** cpuCoreNumber;
9. // 内部类
10. **class** SumCalculator **implements** Callable<Long> {
11. ......
12. }
14. **public** ConcurrentCalculator2() {
15. cpuCoreNumber = Runtime.getRuntime().availableProcessors();
16. exec = Executors.newFixedThreadPool(cpuCoreNumber);
17. completionService = **new** ExecutorCompletionService<Long>(exec);

20. }
22. **public** Long sum(**final** **int**[] numbers) {
23. // 根据CPU核心个数拆分任务，创建FutureTask并提交到Executor
24. **for** (**int** i = 0; i < cpuCoreNumber; i++) {
25. **int** increment = numbers.length / cpuCoreNumber + 1;
26. **int** start = increment \* i;
27. **int** end = increment \* i + increment;
28. **if** (end > numbers.length)
29. end = numbers.length;
30. SumCalculator subCalc = **new** SumCalculator(numbers, start, end);
31. **if** (!exec.isShutdown()) {
32. completionService.submit(subCalc);

35. }
37. }
38. **return** getResult();
39. }
41. /\*\*
42. \* 迭代每个只任务，获得部分和，相加返回
43. \*
44. \* @return
45. \*/
46. **public** Long getResult() {
47. Long result = 0l;
48. **for** (**int** i = 0; i < cpuCoreNumber; i++) {
49. **try** {
50. Long subSum = completionService.take().get();
51. result += subSum;
52. } **catch** (InterruptedException e) {
53. e.printStackTrace();
54. } **catch** (ExecutionException e) {
55. e.printStackTrace();
56. }
57. }
58. **return** result;
59. }
61. **public** **void** close() {
62. exec.shutdown();
63. }
64. }

**五、例子HtmlRender**

该例子模拟浏览器的Html呈现过程，先呈现文本，再异步下载图片，下载完毕每个图片即显示，见附件eclipse项目htmlreander包。

所有代码见附件，Eclipse项目。本文参考《**Java并发编程实践** 》。

* [concurrent.rar](http://dl.iteye.com/topics/download/1a59e915-5945-3795-a6ec-ab85cfea0244) (21.2 KB)
* 下载次数: 1482
* [查看图片附件](http://www.iteye.com/topic/366591)