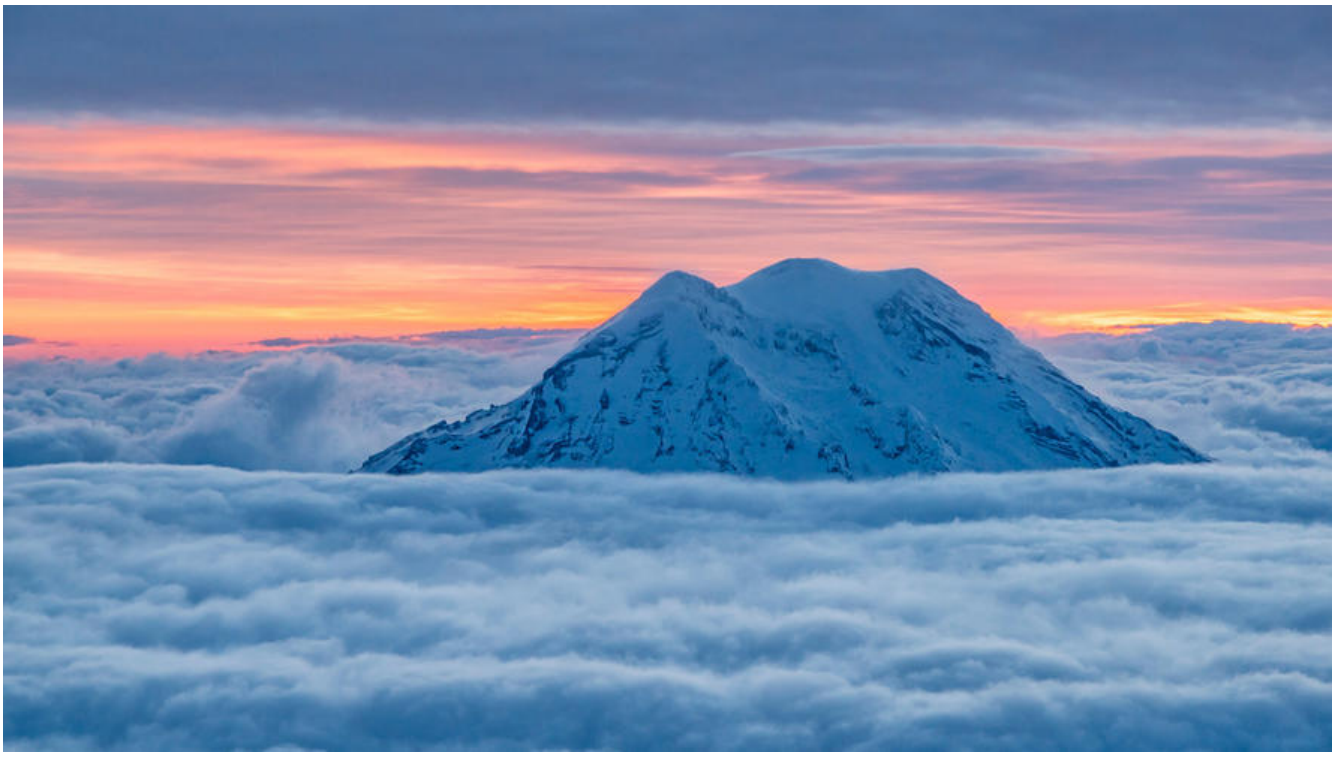
Java 并发编程（五）线程池

作者：wlgzs-sjl
原文链接：https://ld246.com/article/1574727499919
来源网站：链滴



线程池顶级接口Executor

   Executor接口为线程池的顶级接口,其executor()方法接收一个Runnable实类对象,定义了在使用线程池时,如何调用线程中的业务逻辑。

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}

class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

线程池接口ExecutorService

- 其 submit()方法接收一个Callable或Runnable对象,用于指定线程的行为.返回一个Future对象,用取消任务或取得Callable对象call()方法的返回值。
- 其 shutdown()方法和shutdownNow()方法都可以关闭线程池,调用此方法后,线程池不能再调用submit()
  - shutdown()方法会等待当前线程池内所有任务全部完成再关闭线程池。
  - shutdownNow()方法会尝试终止池内正在运行的线程且放弃正在等待的任务,马上关闭线程池。
  - awaitTermination()方法可以使当前线程阻塞一段时间等待线程池被关闭完.返回一个boolean示该线程池是否被关闭完。

```
void shutdownAndAwaitTermination(ExecutorService pool) {
   pool.shutdown(); // Disable new tasks from being submitted
    try {
    // Wait a while for existing tasks to terminate
    if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
       pool.shutdownNow(); // Cancel currently executing tasks
       // Wait a while for tasks to respond to being cancelled
       if (!pool.awaitTermination(60, TimeUnit.SECONDS))
           System.err.println("Pool did not terminate");
   }
} catch (InterruptedException ie) {
    // (Re-)Cancel if current thread also interrupted
    pool.shutdownNow();
    // Preserve interrupt status
    Thread.currentThread().interrupt();
}
```

一个线程池的状态有以下三种:

1. Running: 活动状态,线程池中的线程正在运行且可以通过调用submit()方法接收新任务。
2. ShuttingDown: 线程池正在关闭过程中,线程池中有线程在执行任务,但不会接收新任务。
3. Terminated: 线程池已经关闭,此案城池中没有线程在运行,且不接受新任务。

与线程池相关的类

Callable- 用于定义任务及其返回值

   Callable类用于创建一个任务,类似于Runnable接口,其call()方法定义任务具执行的行为.与Runnalbe类的不同之处在于Callable的call()方法可以有返回值且支持泛型。

要注意Callable和Runnable定义的都是任务而不是线程,要将其传入一个线程或线程池后才可以执行。

Executors-线程池的工厂类和工具类

   Executors为线程池的工厂类和工具类,我们使用其newXXXPool()方法创建种封装好的线程池。

```
newCachedThreadPool()                    ExecutorService
newCachedThreadPool(Thread…              ExecutorService
newFixedThreadPool(int nTh…              ExecutorService
newFixedThreadPool(int nTh…              ExecutorService
newScheduledThreadPool                   ScheduledExecutorSe…
newScheduledThreadPool                   ScheduledExecutorSe…
newSingleThreadExecutor()                ExecutorService
newSingleThreadExecutor(Th…              ExecutorService
newSingleThreadScheduledExecutor         Schedule…
newSingleThreadScheduledExecutor         Schedule…
newWorkStealingPool()                    ExecutorService
newWorkStealingPool(int pa…              ExecutorService
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards >>
```

Future-用于获取任务返回值

  每将一个任务(Callable或Runnable对象)被剑入线程池后,返回一个Future对象.其主要方法和属性如下:

```
cancel(boolean mayInterruptIfRunning)    boolean
get()                                    Integer
get(long timeout, TimeUnit unit)         Integer
isCancelled()                            boolean
isDone()                                 boolean
```

- cancel()方法可以取消该任务。
- get()方法可以阻塞当前线程直到该任务执行完毕并获取其返回值。
- isDone属性指示任务是否执行完毕,isCancelled属性指示任务是否被取消。

```
public class Future {

    public static void main(String[] args) throws ExecutionException, InterruptedException {


        // 未来任务, 既是Runnable 也是 Future
        FutureTask<Integer> task = new FutureTask<>(() -> {
            TimeUnit.MILLISECONDS.sleep(500);
            return 100;
        });
        new Thread(task).start();

        System.out.println(task.get()); // 阻塞等待任务执行完成, 获取到返回值100

        System.out.println("-------------------------------");
```

```
        // 使用ExecutorService的submit替代FutureTask
        ExecutorService service = Executors.newFixedThreadPool(5);
        Future<Integer> result = service.submit(() -> {
            TimeUnit.MILLISECONDS.sleep(500);
            return 1;
        });
        System.out.println(result.isDone()); // false 执行未完毕
        System.out.println(result.get()); // 1
        System.out.println(result.isDone()); // true  执行已完毕
        System.out.println(result.get()); // 一直等待
        System.out.println(service.shutdownNow()); // 立即停止

    }
}
```

Java线程池的具体实现

ThreadPoolExecutor实现的线程池

ThreadPoolExecutor为最常见的线程池类,其构造函数如下:

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

各参数代表的意义如下:

- corePoolSize: 核心线程数量。
- maximumPoolSize: 最大线程数量。
- keepAliveTime: 线程的最大存活时间。一个线程超过keepAliveTime时间不工作则会被销毁。(除当前池中线程个数小于corePoolSize)
- unit: 枚举类型,表示keepAliveTime的单位。
- workQueue: 存放任务的队列。
- handler: 拒绝策略(添加任务失败后如何处理该任务)

线程池的运行策略如下:

1. 线程池刚创建时,里面没有任何线程。
2. 当调用 execute()方法添加一个任务时,会根据**当前线程池中运行线程个数**做出以下不同行为:

- 若 **当前线程池中运行线程数**小于corePoolSize,则在线程池中添加一个新线程执行该任务,即使当

线程池中有空闲线程。

- 若 **当前线程池中运行线程数**大于等于corePoolSize,则尝试将这个任务存入任务队列workQueue。
  - 若任务队列满了且 **当前线程池中运行线程数**小于等于maximumPoolSize,则在线程池中添加个新线程执行该任务。
  - 若任务队列满了且 **当前线程池中运行线程数**大于maximumPoolSize,则任务将被拒绝并执行handler中的拒绝策略。

3. 当一个线程完成任务时,它会从队列中取下一个任务来执行。

4. 当一个线程超过 keepAliveTime时间未执行任务时,线程池根据**当前线程池中运行线程个数**判断否销毁该线程.若**当前线程池中运行线程数**大于corePoolSize,就会销毁该线程,直到线程池收缩到corePolSize大小。

FixedThreadPool:固定容量的线程池

   FixedThreadPool线程池内的最大线程个数是固定的,是一种最常见的线程实现.通过Executors类的ExecutorService newFixedThreadPool(int nThreads)方法创建,其中nThreds参数指定最大线程数。

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

下面程序通过一个容量为4的FixedThreadPool并行寻找质数:

```
public static void main(String[] args) {
    ExecutorService pool = Executors.newFixedThreadPool(3);

    // 向线程池中添加四个任务,分别对四个区间进行查找
    Future<List<Integer>> future1 = pool.submit(new ComputeTask(1, 8_0000));
    Future<List<Integer>> future2 = pool.submit(new ComputeTask(8_0001, 13_0000));
    Future<List<Integer>> future3 = pool.submit(new ComputeTask(13_0001, 17_0000));
    Future<List<Integer>> future4 = pool.submit(new ComputeTask(17_0001, 20_0000));

     // 主程序阻塞等待四个任务执行完成并获取结果
    List<Integer> primes = new LinkedList<>();
    primes.addAll(future1.get());
    primes.addAll(future2.get());
    primes.addAll(future3.get());
    primes.addAll(future4.get());

    // 关闭线程池
    pool.shutdown();
}

// 定义计算任务
static class ComputeTask implements Callable<List<Integer>> {

    private int start, end;
```

```
    ComputeTask(int start, int end) {this.start = start; this.end = end; }

    @Override
    public List<Integer> call()  {
        System.out.println(Thread.currentThread().getName() + "start");
        List<Integer> returnValue = getPrime(start, end);
        System.out.println(Thread.currentThread().getName() + "end");
        return returnValue;
    }
}

static boolean isPrime(int num) {
        for (int i = 2; i < num / 2; i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }

/**
* 返回指定范围的质数列表
*/
static List<Integer> getPrime(int start, int end) {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < end; i++) {
            if (isPrime(i)) {
                list.add(i);
            }
        }
        return list;
    }
```

  我们向容量为3的线程池中加入4个任务,则同一时刻只有3个任务并行执行.
序输出如下,我们发现线程pool-1-thread-3执行了两个任务。

```
pool-1-thread-1start
pool-1-thread-2start
pool-1-thread-3start
pool-1-thread-3end
pool-1-thread-1end
pool-1-thread-2end
pool-1-thread-3start
pool-1-thread-3end
```

CachedThreadPool:容量自动调整的线程池

CachedThreadPool线程池中存活的线程数可以根据实际情况自动调整

- 向线程池添加新任务时,优先使用线程池中存活的可用线程;若线程池当前没有可用线程,则向线程池添加一个新线程。
- 若线程池中的线程超过60秒未使用,则回收该线程。

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

SingleThreadExecutor:线程数为1的线程池

SingleThreadExecutor中的线程个数为1,可以用来保证任务同步执行。

```
public classSingleThreadPool {

    public static void main(String[] args) {
        ExecutorService service = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 5; i++) {
            final int j = i;
            service.execute(() -> {
                System.out.println(j + " " + Thread.currentThread().getName());
            });
        }
    }
}
```

程序输出如下:

```
0 pool-1-thread-1
1 pool-1-thread-1
2 pool-1-thread-1
3 pool-1-thread-1
4 pool-1-thread-1
```

ScheduledThreadPoolExecutor实现的线程池

ScheduledThreadPool:定时执行任务的线程池

   ScheduledThreadPool线程池可以定时执行任务。通过Executors类的ExecutorService newSingleThreadExecutor(int corePoolSize)方法创建,其corePoolSize参数指定线程池核心线程数。

  其主要方法有 schedule(),scheduleAtFixedRate(),schedulWithFixedDelay(),可以设定任务的执行计划.可以根据 当前任务的到期情况 自动调整线程池中的线程.示例程序如下:

```
public class ScheduledPool {

    public static void main(String[] args) {
        ScheduledExecutorService service = Executors.newScheduledThreadPool(4);
        // 使用固定的频率执行某个任务
        // 四个参数
        // command: 执行的任务
        // initialDelay: 第一次执行延时多久执行
        // period: 每隔多久执行一次这个任务
        // unit: 时间单位
```

```
        service.scheduleAtFixedRate(() -> {
            try {
                TimeUnit.MILLISECONDS.sleep(new Random().nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName());
        }, 0, 500, TimeUnit.MILLISECONDS);  // 每隔500ms打印一下线程名称
        // 线程执行1000ms,而每sleep 500 就要新启动一个线程
        // 上个线程未执行完毕,会启用新的线程执行
        // 如果线程池已满,只有延时
    }
}
```

ForkJoinPool实现的线程池

ForkJoinPool:执行递归任务的线程池

ForkJoinPool线程池的submit()方法接收ForkJoinTask类的任务,该类支持fork(),join()成员方法:

- fork()方法会将该子任务加入线程池异步执行.
- join()方法会阻塞当前线程直到子任务执行完成并获取其返回值

   ForkJoinTask有两个子类: RecursiveAction和RecursiveTask,其中RecursiveAction任务没有返回值,而RecursiveTask任务有返回值.它们的任务执行逻辑均写在其compute()方法.

   ForkJoinPool是一个较底层的线程池,因而Executor中没有与其对应的构造法,需要我们显示调用其构造函数获得该类型的线程池。

```
public class T12_ForkJoinPool {

    static int[] nums = new int[100_0000];
    static final int MAX_NUM = 5_0000; // 每个线程最多可以运行5万个数字相加
    static Random random = new Random();

    // 初始化这100_000个数字, 每个数字范围在100之内
    static {

        for (int i = 0; i < nums.length; i++) {
            nums[i] = random.nextInt(100);
        }
        // 所有数字和, 事先计算:
        //System.out.println(Arrays.stream(nums).sum()); // 使用单线程stream api 进行求和
    }

    /**
     * RecursiveAction: 递归操作 没有返回值
     * RecursiveTask: 递归操作,有返回值
     */
    static class AddTask extends RecursiveAction {
```

```
        int start, end;

        AddTask(int start, int end) {
            this.start = start;
            this.end = end;
        }

        @Override
        protected void compute() {

            // 进行计算
            // 如果计算的数的和的范围 小于 MAX_NUM, 进行计算,否则进行 fork
            if (end - start <= MAX_NUM) {
                long sum = 0;
                for (int i = start; i < end; i++) {
                    sum += nums[i];
                }
                System.out.println("sum = " + sum);
            } else {
                int middle = (end - start) / 2;
                AddTask subTask1 = new AddTask(start, middle);
                AddTask subTask2 = new AddTask(middle, end);
                subTask1.fork();
                subTask2.fork();
            }
        }
    }

    static class AddTask2 extends RecursiveTask<Long> {

        int start, end;

        AddTask2(int start, int end) {
            this.start = start;
            this.end = end;
        }

        @Override
        protected Long compute() {
            // 进行计算
            // 如果计算的数的和的范围 小于 MAX_NUM, 进行计算,否则进行 fork
            if (end - start <= MAX_NUM) {
                long sum = 0;
                for (int i = start; i < end; i++) {
                    sum += nums[i];
                }
                return sum;
            } else {
                int middle = start + (end - start) / 2; // 注意这里，如果有问题，会抛出java.lang.NoClassDefFoundError: Could not initialize class java.util.concurrent.locks.AbstractQueuedSynchronizer$Node 异常
                AddTask2 subTask1 = new AddTask2(start, middle);
                AddTask2 subTask2 = new AddTask2(middle, end);
                subTask1.fork();
```

```
            subTask2.fork();
            return subTask1.join() + subTask2.join();
        }
    }
}

// 运行
public static void main(String[] args) throws IOException {
    ForkJoinPool fjp = new ForkJoinPool();
    AddTask2 task = new AddTask2(0, nums.length);
    fjp.execute(task);
    System.out.println(task.join());

    //System.in.read();
}

}
```

WorkStealingPool:工作窃取线程池

   WorkStealingPool线程池通过Executors类的ExecutorService newWorkStealingPool()方法创建,其核心线程数为机器的核心数。

```
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),
                            ForkJoinPool.defaultForkJoinWorkerThreadFactory,
                            null, true);
}
```

   WorkStealingPool线程池采用工作窃取模式,相比于一般的线程池实现,工作取模式的优势体现在对递归任务的处理方式上。

- 在一般的线程池中,若一个线程正在执行的任务由于某些原因无法继续运行,那么该线程会处于等待态。
- 而在 工作窃取模式中,若某个子问题由于等待另外一个子问题的完成而无法继续运行,则处理该子题的线程会主动寻找其他尚未运行的子问题(窃取过来)来执行.这种方式减少了线程的等待时间,提高了能。

```
public class WorkStealingPool {

    public static void main(String[] args) throws IOException {
        // CPU 核数
        System.out.println(Runtime.getRuntime().availableProcessors());

        // workStealingPool 会自动启动cpu核数个线程去执行任务
        ExecutorService service = Executors.newWorkStealingPool();
        service.execute(new R(1000));  // 我的cpu核数为4 启动5个线程,其中第一个是1s执行完毕,其
都是2s执行完毕,
                                       // 有一个任务会进行等待,当第一个执行完毕后,会再次偷取第5个任
执行
        for (int i = 0; i < Runtime.getRuntime().availableProcessors(); i++) {
            service.execute(new R(2000));
        }
```

```
        // 因为work stealing 是deamon线程,即后台线程,精灵线程,守护线程
        // 所以当main方法结束时, 此方法虽然还在后台运行,但是无输出
        // 可以通过对主线程阻塞解决
        System.in.read();
    }

    static class R implements Runnable {

        int time;

        R(int time) {
            this.time = time;
        }

        @Override
        public void run() {
            try {
                TimeUnit.MILLISECONDS.sleep(time);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "  " + time);
        }
    }
}
```

程序输出如下：

```
4
ForkJoinPool-1-worker-2  1000
ForkJoinPool-1-worker-1  2000
ForkJoinPool-1-worker-3  2000
ForkJoinPool-1-worker-0  2000
ForkJoinPool-1-worker-2  2000
```