## 线程池之 Executors + 面试题

线程池的创建分为两种方式：ThreadPoolExecutor 和 Executors，上一节学习了 ThreadPoolExecutor 的使用方式，本节重点来看 Executors 是如何创建线程池的。  
Executors 可以创建以下六种线程池。

* FixedThreadPool(n)：创建一个数量固定的线程池，超出的任务会在队列中等待空闲的线程，可用于控制程序的最大并发数。
* CachedThreadPool()：短时间内处理大量工作的线程池，会根据任务数量产生对应的线程，并试图缓存线程以便重复使用，如果限制 60 秒没被使用，则会被移除缓存。
* SingleThreadExecutor()：创建一个单线程线程池。
* ScheduledThreadPool(n)：创建一个数量固定的线程池，支持执行定时性或周期性任务。
* SingleThreadScheduledExecutor()：此线程池就是单线程的 newScheduledThreadPool。
* WorkStealingPool(n)：Java 8 新增创建线程池的方法，创建时如果不设置任何参数，则以当前机器处理器个数作为线程个数，此线程池会并行处理任务，不能保证执行顺序。

下面分别来看以上六种线程池的具体代码使用。

### FixedThreadPool 使用

创建固定个数的线程池，具体示例如下：

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(2);  
for (int i = 0; i < 3; i++) {  
 fixedThreadPool.execute(() -> {  
 System.out.println("CurrentTime - " + LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

以上程序执行结果如下：

CurrentTime - 2019-06-27 20:58:58

CurrentTime - 2019-06-27 20:58:58

CurrentTime - 2019-06-27 20:58:59

根据执行结果可以看出，newFixedThreadPool(2) 确实是创建了两个线程，在执行了一轮（2 次）之后，停了一秒，有了空闲线程，才执行第三次。

### CachedThreadPool 使用

根据实际需要自动创建带缓存功能的线程池，具体代码如下：

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();  
for (int i = 0; i < 10; i++) {  
 cachedThreadPool.execute(() -> {  
 System.out.println("CurrentTime - " +  
 LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

以上程序执行结果如下：

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

CurrentTime - 2019-06-27 21:24:46

根据执行结果可以看出，newCachedThreadPool 在短时间内会创建多个线程来处理对应的任务，并试图把它们进行缓存以便重复使用。

### SingleThreadExecutor 使用

创建单个线程的线程池，具体代码如下：

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();  
for (int i = 0; i < 3; i++) {  
 singleThreadExecutor.execute(() -> {  
 System.out.println("CurrentTime - " +  
 LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

以上程序执行结果如下：

CurrentTime - 2019-06-27 21:43:34

CurrentTime - 2019-06-27 21:43:35

CurrentTime - 2019-06-27 21:43:36

### ScheduledThreadPool 使用

创建一个可以执行周期性任务的线程池，具体代码如下：

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(2);  
scheduledThreadPool.schedule(() -> {  
 System.out.println("ThreadPool：" + LocalDateTime.now());  
}, 1L, TimeUnit.SECONDS);  
System.out.println("CurrentTime：" + LocalDateTime.now());

以上程序执行结果如下：

CurrentTime：2019-06-27T21:54:21.881

ThreadPool：2019-06-27T21:54:22.845

根据执行结果可以看出，我们设置的 1 秒后执行的任务生效了。

### SingleThreadScheduledExecutor 使用

创建一个可以执行周期性任务的单线程池，具体代码如下：

ScheduledExecutorService singleThreadScheduledExecutor = Executors.newSingleThreadScheduledExecutor();  
singleThreadScheduledExecutor.schedule(() -> {  
 System.out.println("ThreadPool：" + LocalDateTime.now());  
}, 1L, TimeUnit.SECONDS);  
System.out.println("CurrentTime：" + LocalDateTime.now());

### WorkStealingPool 使用

Java 8 新增的创建线程池的方式，可根据当前电脑 CPU 处理器数量生成相应个数的线程池，使用代码如下：

ExecutorService workStealingPool = Executors.newWorkStealingPool();  
for (int i = 0; i < 5; i++) {  
 int finalNumber = i;  
 workStealingPool.execute(() -> {  
 System.out.println("I：" + finalNumber);  
 });  
}  
Thread.sleep(5000);

以上程序执行结果如下：

I：0

I：3

I：2

I：1

I：4

根据执行结果可以看出，newWorkStealingPool 是并行处理任务的，并不能保证执行顺序。

### ThreadPoolExecutor VS Executors

ThreadPoolExecutor 和 Executors 都是用来创建线程池的，其中 ThreadPoolExecutor 创建线程池的方式相对传统，而 Executors 提供了更多的线程池类型（6 种），但很不幸的消息是在实际开发中并不推荐使用 Executors 的方式来创建线程池。

无独有偶《阿里巴巴 Java 开发手册》中对于线程池的创建也是这样规定的，内容如下：

线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的读者更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。

2）CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

OOM 是 OutOfMemoryError 的缩写，指内存溢出的意思。

#### 为什么不允许使用 Executors？

我们先来看一个简单的例子：

ExecutorService maxFixedThreadPool = Executors.newFixedThreadPool(10);  
for (int i = 0; i < Integer.MAX\_VALUE; i++) {  
 maxFixedThreadPool.execute(()->{  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

之后设置 JVM（Java 虚拟机）的启动参数： -Xmx10m -Xms10m （设置 JVM 最大运行内存等于 10M）运行程序，会抛出 OOM 异常，信息如下：

Exception in thread “main” java.lang.OutOfMemoryError: GC overhead limit exceeded

at java.util.concurrent.LinkedBlockingQueue.offer(LinkedBlockingQueue.java:416)

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1371)

at xxx.main(xxx.java:127)

#### 为什么 Executors 会存在 OOM 的缺陷？

通过以上代码，找到了 FixedThreadPool 的源码，代码如下：

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

可以看到创建 FixedThreadPool 使用了 LinkedBlockingQueue 作为任务队列，继续查看 LinkedBlockingQueue 的源码就会发现问题的根源，源码如下：

public LinkedBlockingQueue() {  
 this(Integer.MAX\_VALUE);  
}

当使用 LinkedBlockingQueue 并没有给它指定长度的时候，默认长度为 Integer.MAX\_VALUE，这样就会导致程序会给线程池队列添加超多个任务，因为任务量太大就有造成 OOM 的风险。

### 相关面试题

#### 1.以下程序会输出什么结果？

public static void main(String[] args) {  
 ExecutorService workStealingPool = Executors.newWorkStealingPool();  
 for (int i = 0; i < 5; i++) {  
 int finalNumber = i;  
 workStealingPool.execute(() -> {  
 System.out.print(finalNumber);  
 });  
 }  
}

A：不输出任何结果  
B：输出 0 到 9 有序数字  
C：输出 0 到 9 无需数字  
D：以上全对

答：A  
题目解析：newWorkStealingPool 内部实现是 ForkJoinPool，它会随着主程序的退出而退出，因为主程序没有任何休眠和等待操作，程序会一闪而过，不会执行任何信息，所以也就不会输出任何结果。

#### 2.Executors 能创建单线程的线程池吗？怎么创建？

答：Executors 可以创建单线程线程池，创建分为两种方式：

* Executors.newSingleThreadExecutor()：创建一个单线程线程池。
* Executors.newSingleThreadScheduledExecutor()：创建一个可以执行周期性任务的单线程池。

#### 3.Executors 中哪个线程适合执行短时间内大量任务？

答：newCachedThreadPool() 适合处理大量短时间工作任务。它会试图缓存线程并重用，如果没有缓存任务就会新创建任务，如果线程的限制时间超过六十秒，则会被移除线程池，因此它比较适合短时间内处理大量任务。

#### 4.可以执行周期性任务的线程池都有哪些？

答：可执行周期性任务的线程池有两个，分别是：newScheduledThreadPool() 和 newSingleThreadScheduledExecutor()，其中 newSingleThreadScheduledExecutor() 是 newScheduledThreadPool() 的单线程版本。

#### 5.JDK 8 新增了什么线程池？有什么特点？

答：JDK 8 新增的线程池是 newWorkStealingPool(n)，如果不指定并发数（也就是不指定 n），newWorkStealingPool() 会根据当前 CPU 处理器数量生成相应个数的线程池。它的特点是并行处理任务的，不能保证任务的执行顺序。

#### 6.newFixedThreadPool 和 ThreadPoolExecutor 有什么关系？

答：newFixedThreadPool 是 ThreadPoolExecutor 包装，newFixedThreadPool 底层也是通过 ThreadPoolExecutor 实现的。

newFixedThreadPool 的实现源码如下：

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

#### 7.单线程的线程池存在的意义是什么？

答：单线程线程池提供了队列功能，如果有多个任务会排队执行，可以保证任务执行的顺序性。单线程线程池也可以重复利用已有线程，减低系统创建和销毁线程的性能开销。

#### 8.线程池为什么建议使用 ThreadPoolExecutor 创建，而非 Executors？

答：使用 ThreadPoolExecutor 能让开发者更加明确线程池的运行规则，避免资源耗尽的风险。

Executors 返回线程池的缺点如下：

* FixedThreadPool 和 SingleThreadPool 允许请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量请求，可能会导致内存溢出；
* CachedThreadPool 和 ScheduledThreadPool 允许创建线程数量为 Integer.MAX\_VALUE，创建大量线程，可能会导致内存溢出。

### 总结

Executors 可以创建 6 种不同类型的线程池，其中 newFixedThreadPool() 适合执行单位时间内固定的任务数，newCachedThreadPool() 适合短时间内处理大量任务，newSingleThreadExecutor() 和 newSingleThreadScheduledExecutor() 为单线程线程池，而 newSingleThreadScheduledExecutor() 可以执行周期性的任务，是 newScheduledThreadPool(n) 的单线程版本，而 newWorkStealingPool() 为 JDK 8 新增的并发线程池，可以根据当前电脑的 CPU 处理数量生成对比数量的线程池，但它的执行为并发执行不能保证任务的执行顺序。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

更多java学习资料如下：

[Java高级架构师课程(总共122门课程，1460GB)](https://www.consultdog.com/index.php/front/44/)

[最强java面试视频课程(21门课程，126GB)](https://www.consultdog.com/index.php/front/78/)