Java 多线程开发之 Callable 与线程池（三）

一、前言

我们常见的创建线程的方式有 2 种：继承 Thread 和 实现 Runnable 接口。

其实，在 JDK 中还提供了另外 2 种 API 让开发者使用。

二、简单介绍

2.1 Callable

Java 5.0 在 java.util.concurrent 提供了一个新的创建执行线程的方式： 实现 Callable 接口。

Callable 接口类似于 Runnable，但是 Runnable 不会返回结果，并且无法抛出经过检查的异常，而 Callable 依赖 FutureTask 类获取返回结果。

代码演示：

```
public class CallableTest {

    public static void main(String[] args) throws Exception {

        MyThread mt = new MyThread();

        FutureTask<Integer> result = new FutureTask<Integer>(mt);

        new Thread(result).start();

        // 获取运算结果是同步过程，即 call 方法执行完成，才能获取结果
        Integer sum = result.get();

        System.out.println(sum);
    }
}

class MyThread implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        int sum = 0;

        for (int i = 1; i <= 100; i++) {
            sum += i;
        }

        return sum;
    }

}
```

当某个请求需要在后端完成 N 次统计结果时，我们就可以使用该方式创建 N 个线程进行（并行）统计，而不需要同步等待其他统计操作完成后才统计另一个结果。

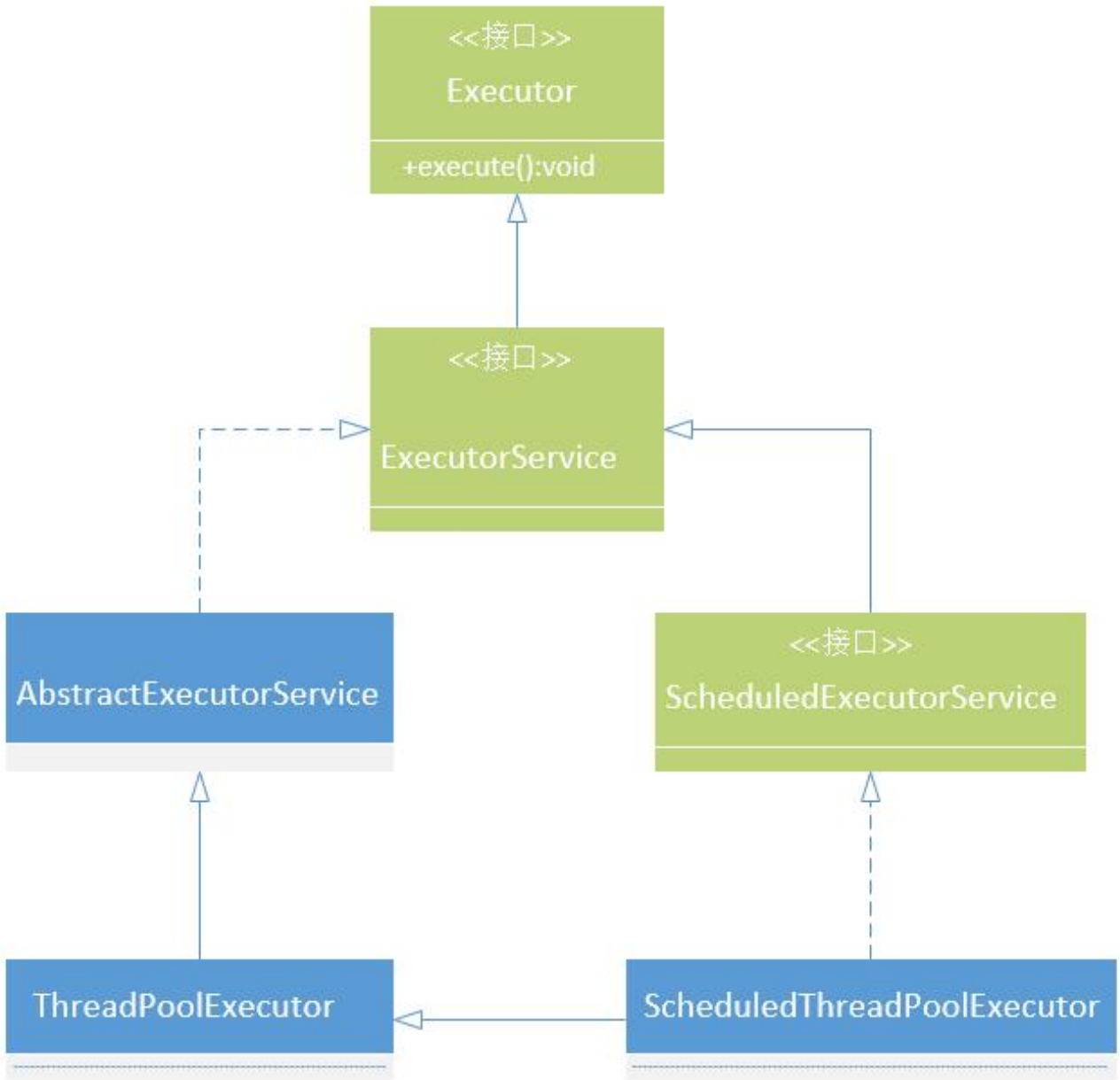
2.2 线程池

第四种获取线程的方法：线程池。

通过重用现有的线程而不是创建新的线程可以在处理多个请求时分摊在线程创建和销毁过程中产生的巨大开销，同时当请求到达时，工作线程已经存在，因此不会由于等待创建线程而延迟任何的执行，从而提高系统的响应性。

2.2.1 线程池体系结构

线程池体系结构：



2.2.2 ThreadPoolExecutor API

ThreadPoolExecutor 用于创建线程池，它有 4 个重载构造器，我们以最多参数的构造器讲解：

```
ThreadPoolExecutor(int corePoolSize,                    # 线程池核心线程个
数，默认线程池线程个数为 0，只有接到任务才新建线程
                   int maximumPoolSize,                 # 线程池最大线程数量
                   long keepAliveTime,                  # 线程池空闲时，线程
存活的时间，当线程池中的线程数大于 corePoolSize 时才会起作用
                   TimeUnit unit,                       # 时间单位
                   BlockingQueue<Runnable> workQueue,   # 阻塞队列，当达到线
程数达到 corePoolSize 时，将任务放入队列等待线程处理
                   ThreadFactory threadFactory,         # 线程工厂
                   RejectedExecutionHandler handler)    # 线程拒绝策略，当队
列满了并且线程个数达到 maximumPoolSize 后采取的策略
```

阻塞队列有以下 4 种：

```
ArrayBlockingQueue：基于数组、有界，按 FIFO（先进先出）原则对元素进行排序；
LinkedBlockingQueue：基于链表，按FIFO （先进先出） 排序元素，吞吐量通常要高于
ArrayBlockingQueue；
SynchronousQueue：每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处
于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue；
PriorityBlockingQueue：具有优先级的、无限阻塞队列。
```

线程拒绝策略有以下 4 种：

```
CallerRunsPolicy：如果发现线程池还在运行，就直接运行这个线程；
DiscardOldestPolicy：在线程池的等待队列中，将头取出一个抛弃，然后将当前线程放进
去；
DiscardPolicy：默默丢弃,不抛出异常；
AbortPolicy：java默认，抛出一个异常（RejectedExecutionException）。
```

实现原则：

```
如果当前池大小 poolSize 小于 corePoolSize ，则创建新线程执行任务；
如果当前池大小 poolSize 大于 corePoolSize ，且等待队列未满，则进入等待队列；
如果当前池大小 poolSize 大于 corePoolSize 且小于 maximumPoolSize ，且等待队列
已满，则创建新线程执行任务；
如果当前池大小 poolSize 大于 corePoolSize 且大于 maximumPoolSize ，且等待队列
已满，则调用拒绝策略来处理该任务；
线程池里的每个线程执行完任务后不会立刻退出，而是会去检查下等待队列里是否还有线程任
务需要执行，如果在 keepAliveTime 里等不到新的任务了，那么线程就会退出。
```

2.2.3 内置线程池

在 java.util.concurrent 包中已经提供为大多数使用场景的内置线程池：

```
Executors.newSingleThreadExecutor()      # 单条线程
Executors.newFixedThreadPool(int n)      # 固定数目线程的线程池
Executors.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。
Executors.newScheduledThreadPool(int n) # 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。
```

上述 4 种线程池底层都是通过创建 ThreadPoolExecutor 获取线程池。

1) newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor() {
        return new Executors.FinalizableDelegatedExecutorService(
                    new ThreadPoolExecutor(1, 1, 0L,
TimeUnit.MILLISECONDS, new LinkedBlockingQueue())
            );
    }
```

主要用于串行（顺序执行）操作场景。

2) newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int var0) {
    return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS,
new LinkedBlockingQueue());
}
```

主要用于负载比较重的场景，为了资源的合理利用，需要限制当前线程数量。

3) newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS,
new SynchronousQueue());
}
```

主要用于并发执行大量短期的小任务，或者是负载较轻的服务器。

4) newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int var0) {
        return new ScheduledThreadPoolExecutor(var0);
    }
```

其中，ScheduledExecutorService 继承 ThreadPoolExecutor。

2.2.4 提交任务

获取 ExecutorService 对象后，我们需要提交任务来让线程池中的线程执行，提交任务的方法有 2 种：

```
void execute()：提交不需要返回值的任务
Future<T> submit()：  提交需要返回值的任务
```

代码演示：

```
// 创建 1 个线程
ExecutorService service = Executors.newSingleThreadExecutor();
service.execute(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ":hello
world");
    }
});
// 关闭线程池
service.shutdown();
```

```
// 创建 5 个线程
ExecutorService service = Executors.newFixedThreadPool(5);
List<Future<String>> list = new ArrayList<>(5);
for (int i = 0; i < 5; i++) {

    Future<String> future = service.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
            return Thread.currentThread().getName() + ":hello world";
        }
    });

    list.add(future);
}

// 打印结果
for (Future<String> future : list) {
    System.out.println(future.get());
}

// 关闭线程池
service.shutdown();
```

```
// 创建 1 个线程
ScheduledExecutorService service = Executors.newScheduledThreadPool(1);
service.schedule(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ":hello
world");
    }
}, 2, TimeUnit.SECONDS);
```

其他线程池使用方式类似，此处不再列举代码。

三、参考资料

线程池相关