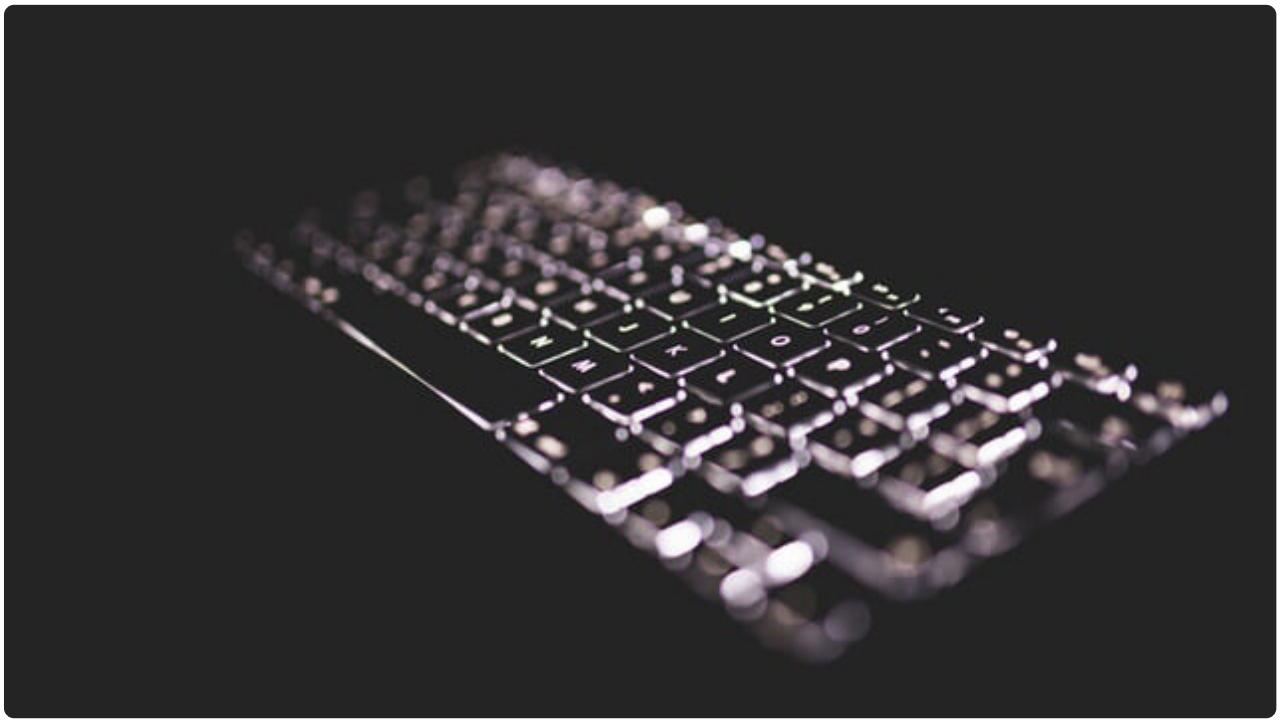
19 自己动手丰衣足食—简单线程池实现

更新时间：2019-10-31 10:26:12



> 学习要注意到细处，不是粗枝大叶的，这样可以逐步学习、摸索，找到客观规律。
>
> —— 徐特立

专栏写到这里，已经完成了前四章的内容。前四章主要围绕线程基础概念在做讲解。比如如何创建线程，多线程并发的问题等等。从本章开始我们会开始讲解 JDK 提供给我们的并发工具类，我们在做多线程开发时经常会借助这些工具类，不但节省了工作量，而且程序也更为健壮。

1、创建线程的问题

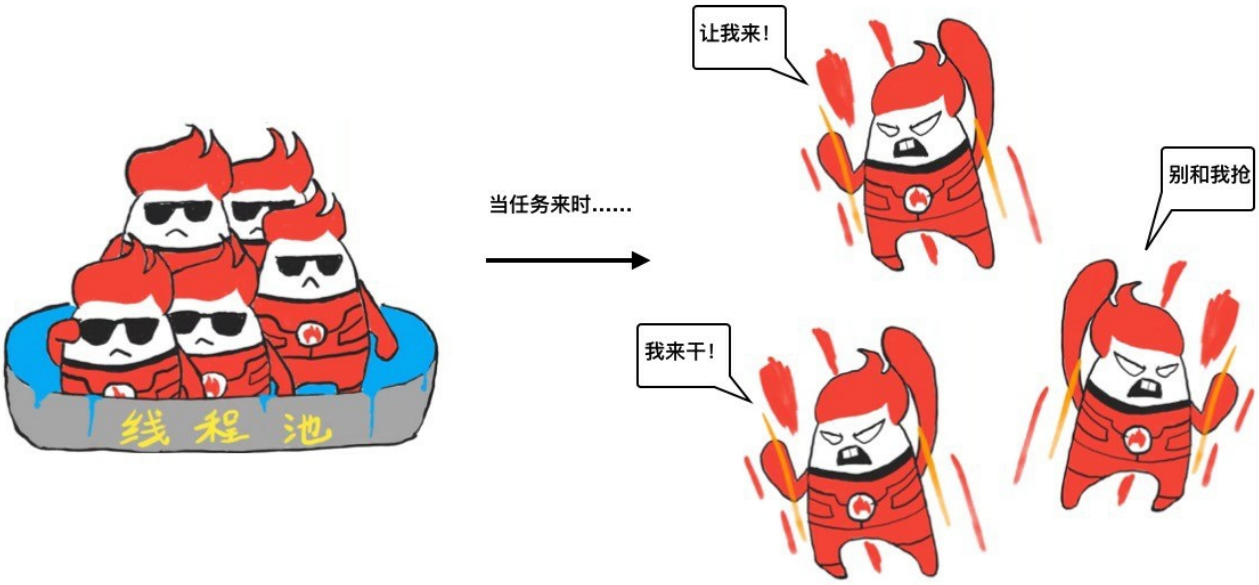
并发的本质其实就是任务的并行处理。绝大多数的并发程序都是围绕离散的任务执行来进行构建。我们在设计此类多线程程序时，首要任务就是对任务进行划分，使得各个不同类型的任务之间相互独立，没有依赖。这样我们就可以并行处理任意的任务。基于我们之前所学习的知识，我们可以为每一个任务建立一个线程来执行。不过我们知道电脑的资源是有限的，无止境的创建线程，性能并不会一直提升，反而会达到峰值后开始衰减。为每个任务都去创建线程存在如下的问题：

1. 线程创建需要消耗资源。通过前面的学习，我们知道线程的创建和启动都需要消耗资源，需要 JVM 和操作系统提供支持。如果线程运行的任务十分轻量级，那么会造成创建线程的时间开销比任务逻辑运行时间还要长；
2. CPU 性能有限。当活跃的线程超过了 CPU 的承载限度，那么会有大量线程参与竞争 CPU，造成系统额外的开销，但是永远都会有很多线程无法竞争到 CPU，造成了资源的浪费；
3. 系统能够支持的线程存在上限。如果超出上限，整个应用就会崩溃。

那么有没有一种方法，既能得到多线程的好处，又能避免以上的问题呢？

2、线程池简介

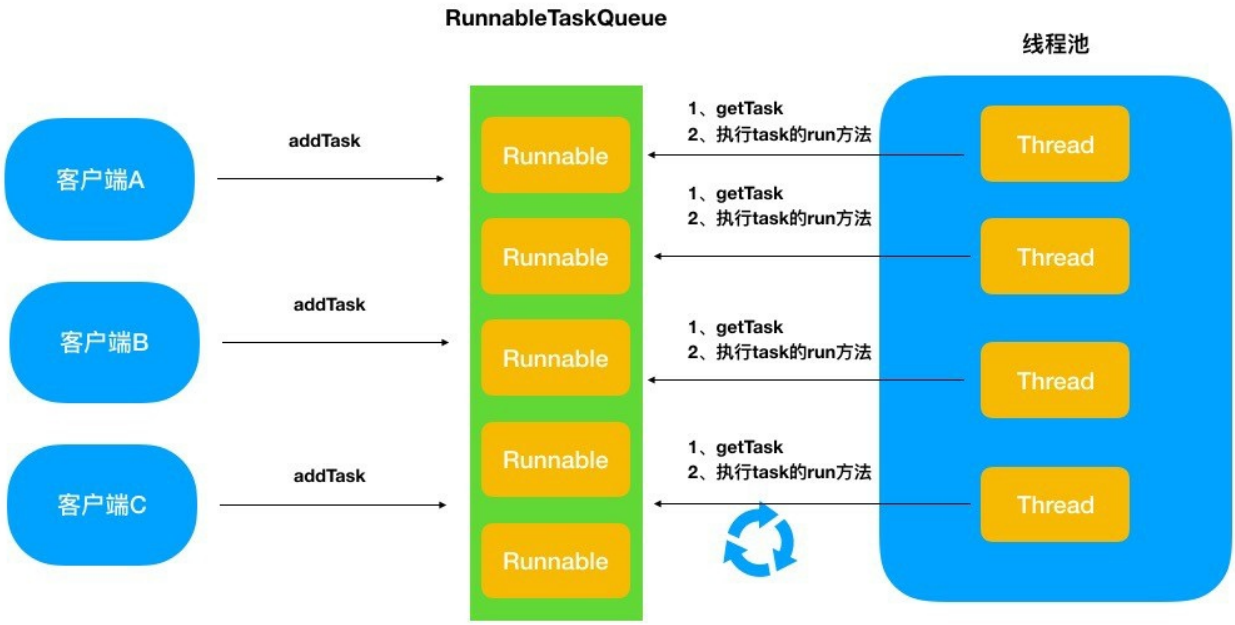
说了那么多，其实答案你肯定已经知道，那就是线程池。线程池的作用是维护一定数量的线程，接收任意数量的任务，这些任务被线程池中的线程并发执行。看到这是不是很像前面讲道德生产者 / 消费者模式？没错，线程池就是基于生产者 / 消费者模式来实现的。客户端调用线程池暴露的方法，向任务列表中生产任务，而线程池中的线程并发消费任务，执行任务的逻辑。



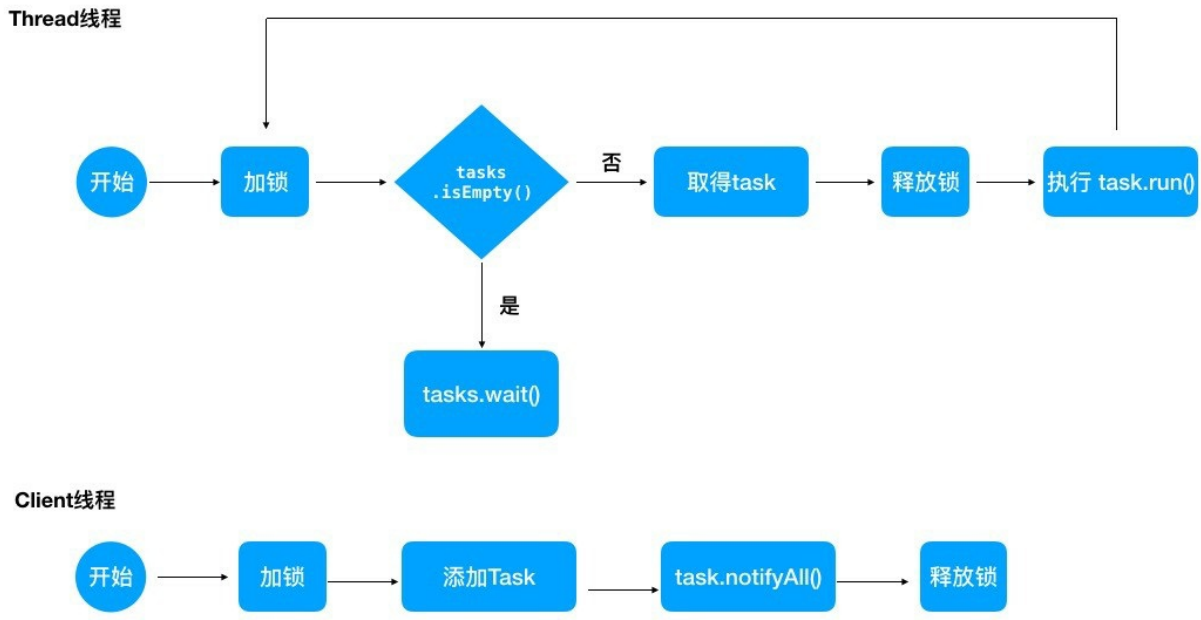
Java 提供了 Excutor 来实现线程池。不过为了加深对线程池的理解，本节我们先不介绍 Excutor，而是自己动手来实现一个线程池。

3、自开发线程池设计

接下来我们将开发一个简单的线程池程序 MyExecutor。正如前文所述，我们的线程池基于生产者 / 消费者模式设计。线程池中维护一个任务对列，线程池接收到的任务放入此队列中。另外还有一个线程队列，其实就是消费者队列，会轮询取得任务队列中的任务，进行执行。如下图所示。



MyExecutor 持有任务队列 RunnableTaskQueue 及固定数量的线程。客户端调用 MyExecutor 对外暴露的 execute 方法，像 RunnableTaskQueue 中添加任务。而 MyExecutor 维护的每个 Thread，其实只做一件事情 —— 不断从 RunnableTaskQueue 中取得 Runable 的实现，调用其 run 方法。run 方法的逻辑就是要执行的任务。而 RunnableTaskQueue 一旦任务被取完，就会开始 wait，线程阻塞。而一旦有新的任务被客户端添加进来，线程池中线程则被唤醒继续拉取任务并执行。如下图所示：



我们实现的这个简单的线程池主要有两个类

1. MyExecutor；
2. RunnableTaskQueue 。

另外还有个测试用的 Client 类。我们逐一讲解。

3.1 RunnableTaskQueue

先看 RunnableTaskQueue 类。这个类中维护了一个 Runnable 实现对象的 LinkedList。并且提供线程安全的 add 和 get 方法，用来添加任务和获取任务。利用 LinkedList 的特性，在获取任务的同时会从队列中移除。代码如下：

```
public class RunnableTaskQueue {
    private final LinkedList<Runnable> tasks = new LinkedList<>();

    public Runnable getTask() throws InterruptedException {
        synchronized (tasks) {
            while (tasks.isEmpty()) {
                System.out.println(Thread.currentThread().getName() + " says task queue is empty. i will wait");
                tasks.wait();
            }
            return tasks.removeFirst();
        }
    }

    public void addTask(Runnable runnable) {
        synchronized (tasks) {
            tasks.add(runnable);
            tasks.notifyAll();
        }
    }
}
```

RunnableTaskQueue 是一个阻塞队列，这保证了线程池中的线程能够不断从中取得任务执行，没有任务时线程也能停下来等待。getTask 和 setTask 都会以同步的方式执行，确保线程安全，并且采用 wait 和 nofityAll 的方式让线程在一定条件下等待和继续运行。

3.2 MyExecutor

接下来我们看 MyExecutor 代码：

```
public class MyExecutor {
    private final int poolSize;

    private final RunnableTaskQueue runnableTaskQueue;

    private final List<Thread> threads = new ArrayList<>();

    public MyExecutor(int poolSize) {
        this.poolSize = poolSize;
        this.runnableTaskQueue = new RunnableTaskQueue();
        Stream.iterate(1, item -> item + 1).limit(poolSize).forEach(item -> {
            initThread();
        });
    }

    private void initThread() {
        if (threads.size() <= poolSize) {
            Thread thread = new Thread(() -> {
                while (true) {
                    try {
                        Runnable task = runnableTaskQueue.getTask();
                        task.run();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            threads.add(thread);
            thread.start();
        }
    }

    public void execute(Runnable runnable) {
        runnableTaskQueue.addTask(runnable);
    }
}
```

poolSize 是线程池的容量，在 MyExecutor 的构造函数中，我们会创建 poolSize 个 Thread。创建 Thread 的方法为 initThread。此方法中先比较已有线程数量是否达到 poolSize。未达到的话，则创建 thread，并且提供 run 的逻辑。这里采用 lambda 表达式的方式，传入 runnable。可以看到线程的 run 方法很简单，就是不断从 runnableTaskQueue 中取得 task，然后运行 task 的 run 方法。回忆下刚刚讲过的 runnableTaskQueue 的 getTask 方法，在没有 task 的时候，会让此线程陷入等待中。

execute 方法是对外暴露的执行任务的方法，方法中向 runnableTaskQueue 添加 task。addTask 方法中，在添加完 task 后，会 nofity 所有等待 task 的线程。

是不是很丝滑，getTask 时可能触发 wait，而一旦 addTask 则会 notifyAll。这一来一往，线程池就能顺畅地工作起来。

3.3 运行你的线程池

方式一：

接下来我们看看客户端代码，对我们刚刚编写线程池做一下测试。我们看下面客户端的代码：

```
public class Client {
    public static void main(String[] args) {
        MyExecutor executor = new MyExecutor(5);

        Stream.iterate(1, item -> item + 1).limit(10).forEach(
                item -> {
                    executor.execute(() -> {
                        try {
                            System.out.println(Thread.currentThread().getName() + " execute this task");
                            TimeUnit.SECONDS.sleep(2);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    });
                }
        );
    }
}
```

首先我们声明了一个 5 个线程的线程池。然后以 lambda 形式向线程池添加了 10 个任务。任务的内容很简单，只是打印执行任务线程的名称，然后 sleep 2 毫秒就结束了。这里大家可以先自己思考下程序运行的结果，再看下面的程序输出：

```
Thread-0 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
Thread-4 execute this task
Thread-3 execute this task
Thread-0 execute this task
Thread-2 execute this task
Thread-1 execute this task
Thread-4 execute this task
Thread-0 execute this task
Thread-3 execute this task
Thread-2 execute this task
Thread-1 execute this task
Thread-2 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
```

以上输出是和程序执行过程保持一致的。下面我们分析下程序执行过程。

1、首先声明 5 个线程的线程池后，这 5 个线程会立即启动，然后从 RunnableTaskQueue 中 getTask；

2、由于还没有添加任务，所以 5 个线程全部开始 wait；

3、然后 10 个任务几乎同时被添加进线程池；

4、每添加一个 task，就会触发 task.notifyAll ()。使得所有线程从从 task 的 waitSet 中被弹出；

5、其中一个线程会取得锁，进入同步的 getTask 方法中获取一个 task；

6、获取 task 后释放锁；

7、执行这个 task 的 run 方法；

8、与此同时其他某个线程会获得锁，然后从 RunnableTaskQueue 获取任务。由于 10 个任务几乎同时被添加进来，所以 RunnableTaskQueue 中此时还有 9 个 task，第二个线程也可以顺利拿到 task。以此类推 5 个线程都能顺利取得 task 执行；

9、第一轮执行完毕后，RunnableTaskQueue 中还剩 5 个 task。于是 5 个线程在第二轮中又各自成功取得一个 task 执行；

10、当 5 个线程第三轮再去 getTask 时，发现 RunnableTaskQueue 已经没有任务了，所以 5 个线程全部开始 wait。

以上分析的执行过程和我们的输出完全吻合。

下面我们换一种执行方式。

方式二：

```
public class Client {
    public static void main(String[] args) {
        MyExecutor executor = new MyExecutor(5);

        Stream.iterate(1, item -> item + 1).limit(10).forEach(
                item -> {
                    try {
                        if(item%2==0){
                            TimeUnit.SECONDS.sleep(2);
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    executor.execute(() -> {
                        System.out.println(Thread.currentThread().getName() + " execute this task");

                    });
                }
        );
    }
}
```

和方式一的区别是，客户端在 2 的整数倍时，sleep2 毫秒再创建。另外任务中不再 sleep。这样会造成生产得慢，消费得快，我们看下程序输出：

```
Thread-0 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-4 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-2 says task queue is empty. i will wait
Thread-0 execute this task
Thread-1 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-0 execute this task
Thread-4 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-2 says task queue is empty. i will wait
Thread-0 execute this task
Thread-1 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-4 says task queue is empty. i will wait
Thread-0 execute this task
Thread-1 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-0 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
Thread-3 execute this task
Thread-0 says task queue is empty. i will wait
Thread-2 says task queue is empty. i will wait
Thread-1 says task queue is empty. i will wait
Thread-4 says task queue is empty. i will wait
Thread-3 says task queue is empty. i will wait
```

可以看到由于消费得快，每产生一个 task 会被迅速消费掉，所以绝大多是时间，大多睡线程都在 wait。另外我们注意看除了第一个 task 和最后一个 task，中间的 task 基本上都是成对被执行的，这是因为双数的任务被添加前要 sleep 2 毫秒，而单数 task 会被立即创建，这就造成双数的 task 产生和上一个 task 有时间间隔。10 个 task 就像被分成了 5 组，分别是 1、2 和 3、4 和 5、6 和 7、8 和 9、10。所以会呈现以上日志中的情况。

4、总结

本节我们自己实现了一个很简单的线程池，提供了非常有限的功能，并且线程池是固定大小。不过这已经足以体会线程池设计的核心思想。就是以固定数量的线程来轮询执行任务队列中的任务。有了这一节的学习，我相信下一节学习 JDK 提供的 Excutor 不会有任何障碍。

}