二

01 为何说只有 1 种实现线程的方法?

在本课时我们主要学习为什么说本质上只有一种实现线程的方式？实现 Runnable 接口究竟比继承 Thread 类实现线程好在哪里?

实现线程是并发编程中基础中的基础，因为我们必须要先实现多线程，才可以继续后续的一系列操作。所以本课时就先从并发编程的基础如何实现线程开始讲起，希望你能够夯实基础，虽然实现线程看似简单、基础，但实际上却暗藏玄机。首先，我们来看下为什么说本质上实现线程只有一种方式?

实现线程的方式到底有几种？大部分人会说有 2 种、3 种或是 4 种，很少有人会说有 1 种。我们接下来看看它们具体指什么？2 种实现方式的描述是最基本的，也是最为大家熟知的，我们就先来看看 2 种线程实现方式的源码。

实现 Runnable 接口

```
public class RunnableThread implements Runnable {

    @Override

    public void run() {

        System.out.println('用实现Runnable接口实现线程');

    }

}
```

第 1 种方式是通过实现 Runnable 接口实现多线程，如代码所示，首先通过 RunnableThread 类实现 Runnable 接口，然后重写 run() 方法，之后只需要把这个实现了 run() 方法的实例传到 Thread 类中就可以实现多线程。

继承 Thread 类

```
public class ExtendsThread extends Thread {
```

```
    @Override
    public void run() {
        System.out.println('用Thread类实现线程');
    }
}
```

第 2 种方式是继承 Thread 类，如代码所示，与第 1 种方式不同的是它没有实现接口，而是继承 Thread 类，并重写了其中的 run() 方法。相信上面这两种方式你一定非常熟悉，并且经常在工作中使用它们。

线程池创建线程

那么为什么说还有第 3 种或第 4 种方式呢？我们先来看看第 3 种方式：通过线程池创建线程。线程池确实实现了多线程，比如我们给线程池的线程数量设置成 10，那么就会有 10 个子线程来为我们工作，接下来，我们深入解析线程池中的源码，来看看线程池是怎么实现线程的？

```
static class DefaultThreadFactory implements ThreadFactory {
    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null) ? s.getThreadGroup() :
            Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" +
            poolNumber.getAndIncrement() +
            "-thread-";
    }
    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
0);
        if (t.isDaemon())
            t.setDaemon(false);
```

```
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

对于线程池而言，本质上是通过线程工厂创建线程的，默认采用 DefaultThreadFactory，它会给线程池创建的线程设置一些默认值，比如：线程的名字、是否是守护线程，以及线程的优先级等。但是无论怎么设置这些属性，最终它还是通过 new Thread() 创建线程的，只不过这里的构造函数传入的参数要多一些，由此可以看出通过线程池创建线程并没有脱离最开始的那两种基本的创建方式，因为本质上还是通过 new Thread() 实现的。

在面试中，如果你只是知道这种方式可以创建线程但不了解其背后的实现原理，就会在面试的过程中举步维艰，想更好的表现自己却给自己挖了“坑”。

所以我们在回答线程实现的问题时，描述完前两种方式，可以进一步引申说“我还知道线程池和Callable 也是可以创建线程的，但是它们本质上也是通过前两种基本方式实现的线程创建。”这样的回答会成为面试中的加分项。然后面试官大概率会追问线程池的构成及原理，这部分内容会在后面的课时中详细分析。

有返回值的 Callable 创建线程

```
class CallableTask implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        return new Random().nextInt();
    }
}
//创建线程池
ExecutorService service = Executors.newFixedThreadPool(10);
//提交任务，并用 Future提交返回结果
Future<Integer> future = service.submit(new CallableTask());
```

第 4 种线程创建方式是通过有返回值的 Callable 创建线程，Runnable 创建线程是无返回值的，而 Callable 和与之相关的 Future、FutureTask，它们可以把线程执行的结果作为返回值返回，如代码所示，实现了 Callable 接口，并且给它的泛型设置成 Integer，然后它会返回一个随机数。

但是，无论是 Callable 还是 FutureTask，它们首先和 Runnable 一样，都是一个任务，是需要被执行的，而不是说它们本身就是线程。它们可以放到线程池中执行，如代码所示，submit() 方法把任务放到线程池中，并由线程池创建线程，不管用什么方法，最终都是靠线程来执行的，而子线程的创建方式仍脱离不了最开始讲的两种基本方式，也就是实现 Runnable 接口和继承 Thread 类。

其他创建方式

定时器 Timer

```
class TimerThread extends Thread {
//具体实现
}
```

讲到这里你可能会说，我还知道一些其他的实现线程的方式。比如，定时器也可以实现线程，如果新建一个 Timer，令其每隔 10 秒或设置两个小时之后，执行一些任务，那么这时它确实也创建了线程并执行了任务，但如果我们深入分析定时器的源码会发现，本质上它还是会有一个继承自 Thread 类的 TimerThread，所以定时器创建线程最后又绕回到最开始说的两种方式。

其他方法

```
/**
 *描述：匿名内部类创建线程
 */
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
```

```
}).start();

}

}
```

或许你还会说，我还知道一些其他方式，比如匿名内部类或 lambda 表达式方式，实际上，匿名内部类或 lambda 表达式创建线程，它们仅仅是在语法层面上实现了线程，并不能把它归结于实现多线程的方式，如匿名内部类实现线程的代码所示，它仅仅是用一个匿名内部类把需要传入的 Runnable 给实例出来。

```
new Thread(() -> System.out.println(Thread.currentThread().getName())).start();

}
```

我们再来看下 lambda 表达式方式。如代码所示，最终它们依然符合最开始所说的那两种实现线程的方式。

实现线程只有一种方式

关于这个问题，我们先不聚焦为什么说创建线程只有一种方式，先认为有两种创建线程的方式，而其他的创建方式，比如线程池或是定时器，它们仅仅是在 new Thread() 外做了一层封装，如果我们把这些都叫作一种新的方式，那么创建线程的方式便会千变万化、层出不穷，比如 JDK 更新了，它可能会多出几个类，会把 new Thread() 重新封装，表面上看又会是一种新的实现线程的方式，透过现象看本质，打开封装后，会发现它们最终都是基于 Runnable 接口或继承 Thread 类实现的。

接下来，我们进行更深层次的探讨，为什么说这两种方式本质上是一种呢？

```
@Override
public void run() {
    if (target != null) {
        target.run();
    }
}
```

首先，启动线程需要调用 start() 方法，而 start() 方法最终还会调用 run() 方法，我们先来看看第一种方式中 run() 方法究竟是怎么实现的，可以看出 run() 方法的代码非常短小精

悍，第 1 行代码 **if (target != null)**，判断 target 是否等于 null，如果不等于 null，就执行第 2 行代码 target.run()，而 target 实际上就是一个 Runnable，即使用 Runnable 接口实现线程时传给Thread类的对象。

然后，我们来看第二种方式，也就是继承 Thread 方式，实际上，继承 Thread 类之后，会把上述的 run() 方法重写，重写后 run() 方法里直接就是所需要执行的任务，但它最终还是需要调用 thread.start() 方法来启动线程，而 start() 方法最终也会调用这个已经被重写的 run() 方法来执行它的任务，这时我们就可以彻底明白了，事实上创建线程只有一种方式，就是构造一个 Thread 类，这是创建线程的唯一方式。

我们上面已经了解了两种创建线程方式本质上是一样的，它们的不同点仅仅在于**实现线程运行内容的不同**，那么运行内容来自于哪里呢？

运行内容主要来自于两个地方，要么来自于 target，要么来自于重写的 run() 方法，在此基础上我们进行拓展，可以这样描述：本质上，实现线程只有一种方式，而要想实现线程执行的内容，却有两种方式，也就是可以通过 实现 Runnable 接口的方式，或是继承 Thread 类重写 run() 方法的方式，把我们想要执行的代码传入，让线程去执行，在此基础上，如果我们还想有更多实现线程的方式，比如线程池和 Timer 定时器，只需要在此基础上进行封装即可。

实现 Runnable 接口比继承 Thread 类实现线程要好

下面我们来对刚才说的两种实现线程内容的方式进行对比，也就是为什么说实现 Runnable 接口比继承 Thread 类实现线程要好？好在哪里呢？

首先，我们从代码的架构考虑，实际上，Runnable 里只有一个 run() 方法，它定义了需要执行的内容，在这种情况下，实现了 Runnable 与 Thread 类的解耦，Thread 类负责线程启动和属性设置等内容，权责分明。

第二点就是在某些情况下可以提高性能，使用继承 Thread 类方式，每次执行一次任务，都需要新建一个独立的线程，执行完任务后线程走到生命周期的尽头被销毁，如果还想执行这个任务，就必须再新建一个继承了 Thread 类的类，如果此时执行的内容比较少，比如只是在 run() 方法里简单打印一行文字，那么它所带来的开销并不大，相比于整个线程从开始创建到执行完毕被销毁，这一系列的操作比 run() 方法打印文字本身带来的开销要大得多，相当于捡了芝麻丢了西瓜，得不偿失。如果我们使用实现 Runnable 接口的方式，就可以把任务直接传入线程池，使用一些固定的线程来完成任务，不需要每次新建销毁线程，大大降低了性能开销。

第三点好处在于 Java 语言不支持双继承，如果我们的类一旦继承了 Thread 类，那么它后续就没有办法再继承其他的类，这样一来，如果未来这个类需要继承其他类实现一些功能上

的拓展，它就没有办法做到了，相当于限制了代码未来的可拓展性。

综上所述，我们应该优先选择通过实现 Runnable 接口的方式来创建线程。

好啦，本课时的全部内容就讲完了，在这一课时我们主要学习了 通过 Runnable 接口和继承 Thread 类等几种方式创建线程，又详细分析了为什么说本质上只有一种实现线程的方式，以及实现 Runnable 接口究竟比继承 Thread 类实现线程好在哪里？学习完本课时相信你一定对创建线程有了更深入的理解。

上一页 下一页