05 看若兄弟，实如父子—Thread和Runnable详解

更新时间：2019-09-12 09:40:02

> 我们有力的道德就是通过奋斗取得物质上的成功；这种道德既适用于国家，也适用于个人。
>
> ——罗素

上篇文章，我们学习了Java中实现多线程的两种基本方式：继承Thread类和实现Runnable接口。从实现的编程手法来看，认为这是两种实现方式并无不妥。但是究其实现根源，这么讲其实并不准确。在本篇文章中，我们将撤底搞懂这两种实现方式。

相信大家之前已经对多线程的实现方式烂熟于心：继承Thread和实现Runnable接口，这么听起来好像两种实现方式是并列关系，就像文章标题所讲的–“看若兄弟”。但其实多线程从根本上讲只有一种实现方式，就是实例化Thread，并且提供其执行的run方法。无论你是通过继承thread还是实现runnable接口，最终都是重写或者实现了run方法。而你真正启动线程都是通过实例化Thread，调用其start方法。我们看下前文中不同实现方式的例子：

1、继承thread方式

```
Student xiaoming = new Student("小明",punishment);
xiaoming.start();
```

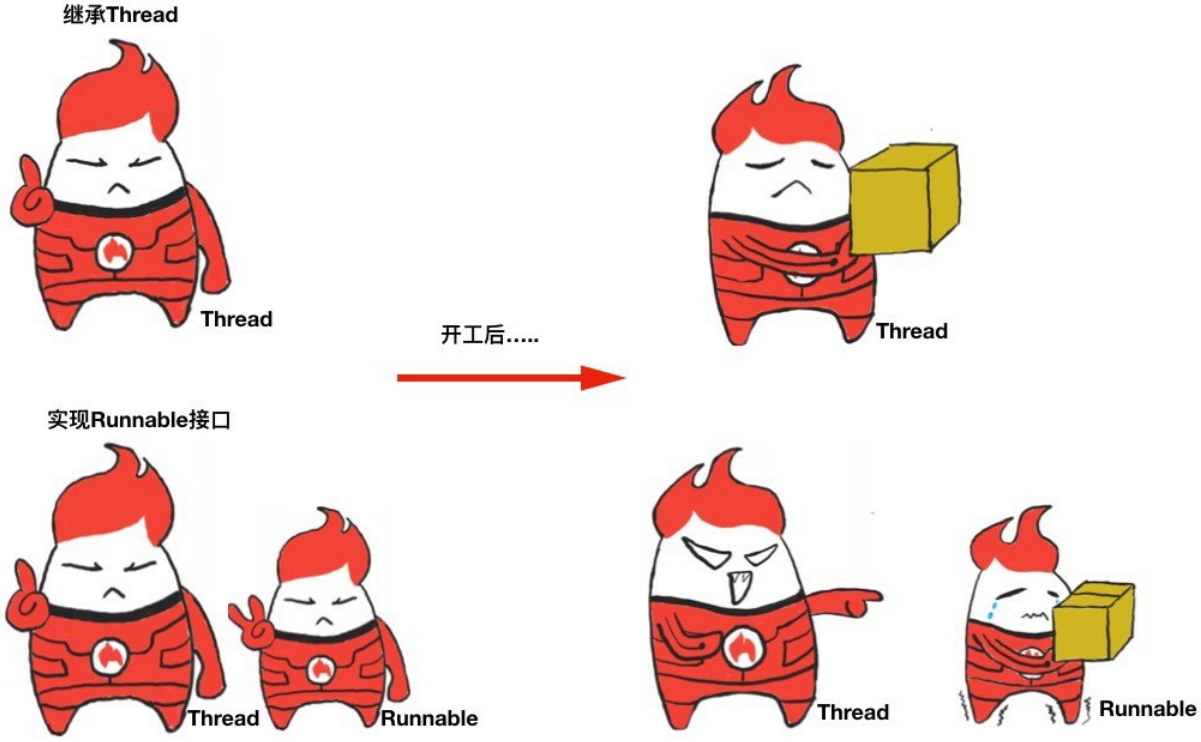
2、实现runnable方式

```
Thread xiaoming = new Thread(new Student("小明",punishment),"小明");
xiaoming.start();
```

第一种方式中，Student继承了Thread类，启动时调用的start方法，其实还是他父类Thread的start方法。并最终触发执行Student重写的run方法。

第二种方式中，Student实现Runnable接口，作为参数传递给Thread构造函数。接下来还是调用了Thread的start方法。最后则会触发传入的runnable实现的run方法。

两种方式都是创建 Thread 或者 Thread 的子类，通过 Thread 的 start 方法启动。唯一不同是第一种 run 方法实现在 Thread 子类中。第二种则是把run方法逻辑转移到 Runnable 的实现类中。线程启动后，第一种方式是 thread 对象运行自己的 run 方法逻辑，第二种方式则是调用 Runnable 实现的 run 方法逻辑。如下图所示：



相比较来说，第二种方式是更好的实践，原因如下：

1. java语言中只能单继承，通过实现接口的方式，可以让实现类去继承其它类。而直接继承thread就不能再继承其它类了；
2. 线程控制逻辑在Thread类中，业务运行逻辑在Runnable实现类中。解耦更为彻底；
3. 实现Runnable的实例，可以被多个线程共享并执行。而实现thread是做不到这一点的。

看到这里，你是不是很好奇，为什么程序中调用的是Thread的start方法，而不是run方法？为什么线程在调用start方法后会执行run方法的逻辑呢？接下来我们通过开始start方法的源代码来找到答案。

Thread start方法源代码分析

我们先看Thread类start方法源代码，如下：

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    group.add(this);

    boolean started = false;
    try {
        start0();
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            }
    }
}
```

这段代码足够简单，简单到没什么内容。主要逻辑如下：

1. 检查线程的状态，是否可以启动；
2. 把线程加入到线程group中；
3. 调用了start0()方法。

可以看到Start方法中最终调用的是start0方法，并不是run方法。那么我们再看start0方法源代码：

```
private native void start0();
```

什么也没有，因为start0是一个native方法，也称为JNI（Java Native Interface）方法。JNI方法是java和其它语言交互的方式。同样也是java代码和虚拟机交互的方式，虚拟机就是由C++和汇编所编写。

由于start0是一个native方法，所以后面的执行会进入到JVM中。那么run方法到底是何时被调用的呢？这里似乎找不到答案了。

难道我们错过了什么？回过头来我们再看看Start方法的注解。其实读源代码的时候，要先读注解，否则直接进入代码逻辑，容易陷进去，出不来。原来答案就在start方法的注解里，我们可以看到：

```
* Causes this thread to begin execution; the Java Virtual Machine
* calls the <code>run</code> method of this thread.
* <p>
* The result is that two threads are running concurrently: the
* current thread (which returns from the call to the
* <code>start</code> method) and the other thread (which executes its
* <code>run</code> method).
* <p>
* It is never legal to start a thread more than once.
* In particular, a thread may not be restarted once it has completed
* execution.
```

最关键一句*the Java Virtual Machine calls the run method of this thread。*由此我们可以推断出整个执行流程如下：

start方法调用了start0方法，start0方法在JVM中，start0中的逻辑会调用run方法。

至此，我们已经分析清楚从线程创建到run方法被执行的逻辑。但是通过实现Runnbale的方式实现多线程时，Runnable的run方法是如何被调用的呢？

Thread Run方法分析

对于上面提出的问题，我们先从Thread的构造函数入手。原因是Runnable的实现对象通过构造函数传入Thread。

```
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

可以看到Runnable实现作为target对象传递进来。再次调用了init方法，init　方法有多个重载，最终调用的是如下方法：

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals)
```

此方法里有一行代码：

```
this.target = target;
```

原来target是Thread的成员变量：

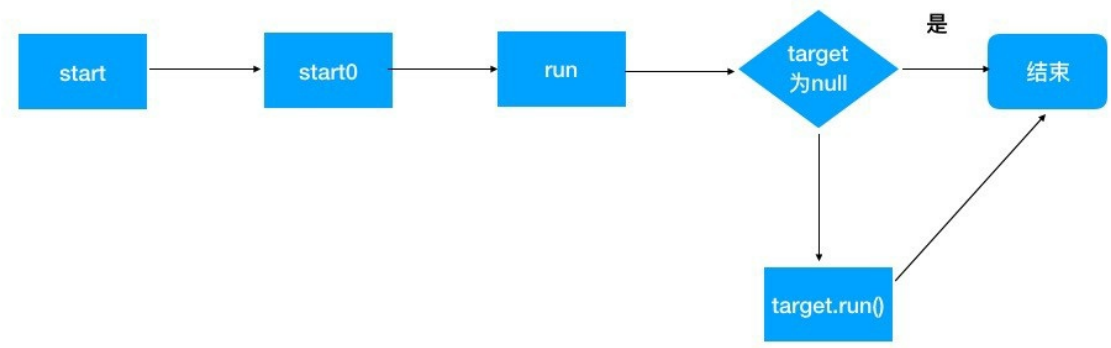
```
/* What will be run. */
private Runnable target;
```

此时，Thread的target被设置为你实现业务逻辑的Runnable实现。

我们再看下run方法的代码：

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

看到这里是不是已经很清楚了，当你传入了target，则会执行target的run方法。也就是执行你实现业务逻辑的方法。整体执行流程如下：



如果你是通过继承Thread，重写run方法的方式实现多线程。那么在第三步执行的就是你重写的run方法。

我们回过头看看Thread类的定义：

```
public class Thread implements Runnable
```

原来Thread也实现了Runnable接口。怪不得Thread类的run方法上有@Override注解。所以继承thread类实现多线程，其实也相当于是实现runnable接口的run方法。只不过此时，不需要再传入一个Thread类去启动。它自己已具备了thread的功能，自己就可以运转起来。既然Thread类也实现了Runnable接口，那么thread子类对象是不是也可以传入另外的thread对象，让其执行自己的run方法呢？答案是可行的，你可以亲手试一下。

总结

至此，我们已经从理论到代码，把多线程的两种实现方式做了分析。在学习多线程的同时，我们也应该学习源代码中优秀的设计模式。Java中多线程的实现采用了模版模式。Thread是模版对象，负责线程相关的逻辑，比如线程的创建、运行以及各种操作。而线程真正的业务逻辑则被剥离出来，交由Runnable的实现类去实现。线程操作和业务逻辑完全解耦，普通开发者只需要聚焦在业务逻辑实现。

执行业务逻辑，是Thread对象的生命周期中的重要一环。这一步通过调用传入Runnable的run方法实现。thread线程整体逻辑就是一个模版，把其中一个步骤剥离出来由其他类实现，这就是模版方法模式。讲到这里，我们回到标题–“实如父子”。没错，其实线程自身的逻辑都在thread类中，而runnable实现类只是线程执行流程中的一小步而已。所以Thread和Runnable更像是父子关系。

讲到最后，抛出一个问题，你还记得start方法中如下两行代码吗？

```
if (threadStatus != 0)
    throw new IllegalThreadStateException();
```

这段代码判断了线程状态属性threadStatus的值。如果不是0，则直接抛出异常，不会向下执行start0方法。那么Thread有几种状态，几种状态之间是如何转换的呢？start之后，run方法立即就会被调用吗？在下一篇专栏中，将会一一为你解答。

}

04 人多力量未必大—并发可能会遇到的问题

06 线程什么时候开始真正执行？—线程的状态详解