```
Thread(Runnable {
    println("Thread is running")
}).start()
```

这段代码明显简化了很多，既可以实现同样的功能，又不会造成任何歧义。因为 Runnable 类中只有一个待实现方法，即使这里没有显式地重写 run()方法，Kotlin 也能自动明白 Runnable 后面的 Lambda 表达式就是要在 run()方法中实现的内容。

另外，如果一个 Java 方法的参数列表中有且仅有一个 Java 单抽象方法接口参数，我们还可以将接口名进行省略，这样代码就变得更加精简了：

```
Thread({
    println("Thread is running")
}).start()
```

不过到这里还没有结束，和之前 Kotlin 中函数式 API 的用法类似，当 Lambda 表达式是方法的最后一个参数时，可以将 Lambda 表达式移到方法括号的外面。同时，如果 Lambda 表达式还是方法的唯一一个参数，还可以将方法的括号省略，最终简化结果如下：

```
Thread {
    println("Thread is running")
}.start()
```

如果你将上述代码写到 main()函数中并执行，就会得到如图 2.29 所示的结果。

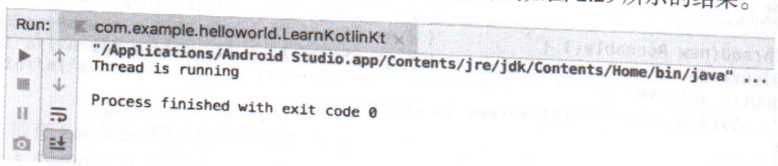

图 2.29　Java 函数式 API 的运行结果

或许你会觉得，既然本书中所有的代码都是使用 Kotlin 编写的，这种 Java 函数式 API 应该并不常用吧？其实并不是这样的，因为我们后面要经常打交道的 Android SDK 还是使用 Java 语言编写的，当我们在 Kotlin 中调用这些 SDK 接口时，就很可能会用到这种 Java 函数式 API 的写法。

举个例子，Android 中有一个极为常用的点击事件接口 OnClickListener，其定义如下：

```
public interface OnClickListener {
    void onClick(View v);
}
```

可以看到，这又是一个单抽象方法接口。假设现在我们拥有一个按钮 button 的实例，然后使用 Java 代码去注册这个按钮的点击事件，需要这么写：

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
```