

## 28 | launch的背后到底发生了什么？

2022-03-28 朱涛

《朱涛 · Kotlin编程第一课》

[课程介绍 >](#)



讲述：朱涛

时长 16:18 大小 14.94M



你好，我是朱涛。

在前面的课程里，我们一直在研究如何使用 Kotlin 协程，比如，如何启动协程，如何使用挂起函数，如何使用 Channel、Flow 等 API。但到目前为止，我们只知道该怎么用协程，对它内部的运行机制并没有深究。

现在我们都知，launch、async 可以创建、启动新的协程，但我们只能通过调试参数，通过 log 看到协程。比如我们可以回过头来看下 [第 13 讲](#) 当中的代码：

```
1 // 代码段1
2
3 // 代码中一共启动了两个协程
4 fun main() = runBlocking {
5     println(Thread.currentThread().name)
6 }
```

[复制代码](#)

```

7      launch {
8          println(Thread.currentThread().name)
9          delay(100L)
10     }
11
12     Thread.sleep(1000L)
13 }
14
15 /*
16 输出结果:
17 main @coroutine#1
18 main @coroutine#2
19 */

```

现在回过头来看，这段代码无疑是非常简单的，`runBlocking{}` 启动了第一个协程，`launch{}` 启动了第二个协程。可是，有一个问题，我们一直都没有找到答案：**协程到底是如何创建的？它对应的源代码，到底在哪个类？具体在哪一行？**

我们常说 Java 线程的源代码是 `Thread.java`，这样说虽然不一定准确，但我们起码能看到几个暴露出来的方法。那么，在 Kotlin 协程当中，有没有类似 `Coroutine.kt` 的类呢？对于这些问题，我们唯有去阅读 Kotlin 协程的源码、去分析 `launch` 的启动流程，才能找到答案。

这节课，我就将从 [第 26 讲](#) 当中提到的 `createCoroutine{}` 、`startCoroutine{}`  这两个函数开始说起，在认识了这两个协程基础元素以后，我们就会进入协程的“中间层”，开始分析 `launch` 的源代码。我相信，学完这节课以后，你一定会对 Kotlin 协程有一个更加透彻的认识。

## 协程启动的基础 API

在第 26 讲里，我给你留了一个思考题，在 [Continuation.kt](#) 这个文件当中，还有两个重要的扩展函数：

 复制代码

```

1  // 代码段2
2
3  public fun <T> (suspend () -> T).createCoroutine(
4      completion: Continuation<T>
5  ): Continuation<Unit> =
6      SafeContinuation(createCoroutineUnintercepted(completion).intercepted(), CC
7
8  public fun <T> (suspend () -> T).startCoroutine(
9      completion: Continuation<T>
10 ) {

```

```
11     createCoroutineUnintercepted(completion).intercepted().resume(Unit)
12 }
```

其实，`createCoroutine{}、startCoroutine{}`这两个函数，就是 Kotlin 协程当中最基础的两个创建协程的 API。

我们在 [第 14 讲](#) 里曾经提到过，启动协程有三种常见的方式：`launch、runBlocking、async`。它们其实属于协程中间层提供的 API，而它们的底层都在某种程度上调用了“基础层”的协程 API。

那么，这是不是就意味着：**我们使用协程的基础层 API，也可以创建协程呢？**

答案当然是肯定的。我们可以来分析一下代码段 2 当中的函数签名。

`createCoroutine{}、startCoroutine{}、`它们都是扩展函数，其扩展接收者类型是一个函数类型：`suspend () -> T`，代表了“无参数，返回值为 T 的挂起函数或者 Lambda”。而对于函数本身，它们两个都接收一个 `Continuation<T>` 类型的参数，其中一个函数，还会返回一个 `Continuation<Unit>` 类型的返回值。

也许你对于“给函数类型增加扩展”这样的行为会感到不太适应。不过，在 Kotlin 当中，**函数就是一等公民**，普通的类型可以有扩展，那么，函数类型自然也可以有扩展。因此，我们完全可以写出像下面这样的代码：

 复制代码

```
1 // 代码段3
2
3 fun main() {
4     testStartCoroutine()
5     Thread.sleep(2000L)
6 }
7
8 val block = suspend {
9     println("Hello!")
10    delay(1000L)
11    println("World!")
12    "Result"
13 }
14
15 private fun testStartCoroutine() {
16
17     val continuation = object : Continuation<String> {
```

```

18         override val context: CoroutineContext
19             get() = EmptyCoroutineContext
20
21         override fun resumeWith(result: Result<String>) {
22             println("Result is: ${result.getOrNull()}")
23         }
24     }
25
26     block.startCoroutine(continuation)
27 }
28
29 /*
30 输出结果
31 Hello!
32 World!
33 Result is: Result
34 */

```

在这段代码中，我们定义了一个 Lambda 表达式 `block`，它的类型就是 `suspend () -> T`。这样一来，我们就可以用 `block.startCoroutine()` 来启动协程了。这里，我们还创建了一个匿名内部类对象 `continuation`，作为 `startCoroutine()` 的参数。

在 [🔗 加餐](#) 里，我们提到过 `Continuation` 主要有两种用法，一种是在实现挂起函数的时候，用于 **传递挂起函数的执行结果**；另一种是在调用挂起函数的时候，以匿名内部类的方式，用于 **接收挂起函数的执行结果**。而代码段 3 中 `Continuation` 的作用，则明显属于后者。

从代码段 3 的执行结果中，我们可以看出来，`startCoroutine()` 的作用其实就是创建一个新的协程，并且执行 `block` 当中的逻辑，等协程执行完毕以后，将结果返回给 `Continuation` 对象。而这个逻辑，我们使用 `createCoroutine()` 这个方法其实也可以实现。

 复制代码

```

1  代码段4
2
3  private fun testCreateCoroutine() {
4
5      val continuation = object : Continuation<String> {
6          override val context: CoroutineContext
7              get() = EmptyCoroutineContext
8
9          override fun resumeWith(result: Result<String>) {
10              println("Result is: ${result.getOrNull()}")
11          }
12      }
13

```

```

14     val coroutine = block.createCoroutine(continuation)
15
16     coroutine.resume(Unit)
17 }
18
19 /*
20 输出结果
21 Hello!
22 World!
23 Result is: Result
24 */

```

根据以上代码，我们可以看到，`createCoroutine()` 的作用其实就是创建一个协程，并暂时先不启动它。等我们想要启动它的时候，直接调用 `resume()` 即可。如果我们再进一步分析代码段 2 当中的源代码，会发现 `createCoroutine()`、`startCoroutine()` 的源代码差别也并不大，只是前者没有调用 `resume()`，而后者调用了 `resume()`。

换句话说，`startCoroutine()` 之所以可以创建并同时启动协程的原因就在于，它在源码中直接调用了 `resume(Unit)`，所以，我们在代码段 3 当中就不需要自己调用 `resume()` 方法了。

那么下面，我们就以 `startCoroutine()` 为例，来研究下它的实现原理。我们把代码段 3 反编译成 Java，看看它会变成什么样子：

 复制代码

```

1 // 代码段5
2
3 public final class LaunchUnderTheHoodKt {
4     // 1
5     public static final void main() {
6         testStartCoroutine();
7         Thread.sleep(2000L);
8     }
9
10    // 2
11    private static final Function1<Continuation<? super String>, Object> block
12
13    // 3
14    public static final Function1<Continuation<? super String>, Object> getBlock
15        return block;
16    }
17    // 4
18    static final class LaunchUnderTheHoodKt$block$1 extends SuspendLambda implements
19        int label;
20
21    LaunchUnderTheHoodKt$block$1(Continuation $completion) {

```

```

22     super(1, $completion);
23 }
24
25 @Nullable
26 public final Object invokeSuspend(@NotNull Object $result) {
27     Object object = IntrinsicsKt.getCOROUTINE_SUSPENDED();
28     switch (this.label) {
29         case 0:
30             ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
31             System.out
32                 .println("Hello!");
33             this.label = 1;
34             if (DelayKt.delay(1000L, (Continuation)this) == object)
35                 return object;
36             DelayKt.delay(1000L, (Continuation)this);
37             System.out
38                 .println("World!");
39             return "Result";
40         case 1:
41             ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
42             System.out.println("World!");
43             return "Result";
44     }
45     throw new IllegalStateException("call to 'resume' before 'invoke' with");
46 }
47
48 @NotNull
49 public final Continuation<Unit> create(@NotNull Continuation<? super La
50     return (Continuation<Unit>)new LaunchUnderTheHoodKt$block$1($completi
51 }
52
53 @Nullable
54 public final Object invoke(@Nullable Continuation<?> p1) {
55     return ((LaunchUnderTheHoodKt$block$1)create(p1)).invokeSuspend(Unit.
56 }
57 }
58
59 // 5
60 private static final void testStartCoroutine() {
61     LaunchUnderTheHoodKt$testStartCoroutine$continuation$1 continuation = n
62     ContinuationKt.startCoroutine(block, continuation);
63 }
64
65 // 6
66 public static final class LaunchUnderTheHoodKt$testStartCoroutine$continuat
67     @NotNull
68     public CoroutineContext getContext() {
69         return (CoroutineContext)EmptyCoroutineContext.INSTANCE;
70     }
71
72     public void resumeWith(@NotNull Object result) {
73         System.out.println(Intrinsics.stringPlus("Result is: ", Result.isFail

```



```

74     }
75 }
76 }
77
78
79 internal abstract class SuspendLambda(
80     public override val arity: Int,
81     completion: Continuation<Any??>
82 ) : ContinuationImpl<Any>(), FunctionBase<Any>() {

```

上面的反编译代码中，一共有 6 个注释，我们一个个来看：

- 注释 1，是我们的 `main()` 函数。由于它本身只是一个普通的函数，因此反编译之后，逻辑并没有什么变化。
- 注释 2、3，它们是 Kotlin 为 block 变量生成的静态变量以及方法。
- 注释 4，`LaunchUnderTheHoodKt$block$1`，其实就是 `block` 具体的实现类。这个类继承自 `SuspendLambda`，而 `SuspendLambda` 是 `ContinuationImpl` 的子类，因此它也间接实现了 `Continuation` 接口。其中的 `invokeSuspend()`，也就是我们在上节课分析过的**协程状态机逻辑**。除此之外，它还有一个 `create()` 方法，我们在后面会来分析它。
- 注释 5，它对应了 `testStartCoroutine()` 这个方法，原本的 `block.startCoroutine(continuation)` 变成了 `ContinuationKt.startCoroutine(block, continuation)`”，这其实就体现出了扩展函数的原理。
- 注释 6，其实就是 `continuation` 变量对应的匿名内部类。

那么接下来，我们就可以对照着反编译代码，来分析整个代码的执行流程了。

首先，`main()` 函数会调用 `testStartCoroutine()` 函数，接着，就会调用 `startCoroutine()` 方法。

 复制代码

```

1 // 代码段6
2
3 public fun <T> (suspend () -> T).startCoroutine(
4     completion: Continuation<T>
5 ) {
6 //     注意这里
7 //     ↓
8 createCoroutineUnintercepted(completion).intercepted().resume(Unit)
9 }

```

从代码段 6 里，我们可以看到，在 `startCoroutine()` 当中，首先会调用 `createCoroutineUnintercepted()` 方法。如果我们直接去看它的源代码，会发现它只存在一个声明，并没有具体实现：

 复制代码

```
1 // 代码段7
2
3 //    注意这里
4 //    ↓
5 public expect fun <T> (suspend () -> T).createCoroutineUnintercepted(
6     completion: Continuation<T>
7 ): Continuation<Unit>
```

上面代码中的 `expect`，我们可以把它理解为一种**声明**，由于 Kotlin 是面向多个平台的，具体的实现，就需要在特定的平台实现。所以在这里，我们就需要打开 Kotlin 的源代码，找到 JVM 平台对应的实现：

 复制代码

```
1 // 代码段8
2
3 //    1, 注意这里
4 //    ↓
5 public actual fun <T> (suspend () -> T).createCoroutineUnintercepted(
6     completion: Continuation<T>
7 ): Continuation<Unit> {
8     val probeCompletion = probeCoroutineCreated(completion)
9     //    2, 注意这里
10    //    ↓
11    return if (this is BaseContinuationImpl)
12        create(probeCompletion)
13    else
14        createCoroutineFromSuspendFunction(probeCompletion) {
15            (this as Function1<Continuation<T>, Any?>).invoke(it)
16        }
17 }
```

请留意这里的注释 1，这个 `actual`，代表了 `createCoroutineUnintercepted()` 在 JVM 平台的实现。



另外，我们可以看到，`createCoroutineUnintercepted()` 仍然还是一个扩展函数，注释 2 处的 `this`，其实就代表了前面代码段 3 当中的 `block` 变量。我们结合代码段 5 反编译出来的 `LaunchUnderTheHoodKt$block$1`，可以知道 `block` 其实就是 `SuspendLambda` 的子类，而 `SuspendLambda` 则是 `ContinuationImpl` 的子类。

因此，注释 2 处的 `(this is BaseContinuationImpl)` 条件一定是为 **true** 的。这时候，它就会调用 `create(probeCompletion)`。

然后，如果你去查看 `create()` 的源代码，会看到这样的代码：

 复制代码

```
1 // 代码段9
2
3 public open fun create(completion: Continuation<*>): Continuation<Unit> {
4     throw UnsupportedOperationException("create(Continuation) has not been over
5 }
```

可以看到，在默认情况下，这个 `create()` 方法是会抛出异常的，它的提示信息是：`create()` 方法没有被重写！潜台词就是，`create()` 方法应该被重写！如果不被重写，就会抛出异常。

那么，**`create()` 方法是在哪里被重写的呢？** 答案其实就在代码段 5 的“`LaunchUnderTheHoodKt$block$1`”这个 `block` 的实现类当中。

 复制代码

```
1 // 代码段10
2
3 static final class LaunchUnderTheHoodKt$block$1 extends SuspendLambda implement
4     int label;
5
6     LaunchUnderTheHoodKt$block$1(Continuation $completion) {
7         super(1, $completion);
8     }
9
10    @Nullable
11    public final Object invokeSuspend(@NotNull Object $result) {
12        Object object = IntrinsicsKt.getCOROUTINE_SUSPENDED();
13        switch (this.label) {
14            case 0:
15                ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
16                System.out
17                    .println("Hello!");
```

```

18         this.label = 1;
19         if (DelayKt.delay(1000L, (Continuation)this) == object)
20             return object;
21         DelayKt.delay(1000L, (Continuation)this);
22         System.out
23             .println("World!");
24         return "Result";
25     case 1:
26         ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
27         System.out.println("World!");
28         return "Result";
29     }
30     throw new IllegalStateException("call to 'resume' before 'invoke' with co
31 }
32
33 // 1, 注意这里
34 public final Continuation<Unit> create(@NotNull Continuation<? super Launch
35     return (Continuation<Unit>)new LaunchUnderTheHoodKt$block$1($completion);
36 }
37
38 @Nullable
39 public final Object invoke(@Nullable Continuation<?> p1) {
40     return ((LaunchUnderTheHoodKt$block$1)create(p1)).invokeSuspend(Unit.INST
41 }
42 }

```

这里，你可以留意下代码里的注释 1，这个其实就是重写之后的 `create()` 方法。换句话说，代码段 8 当中的 `create(probeCompletion)`，最终会调用代码段 10 的 `create()` 方法，它最终会返回“`LaunchUnderTheHoodKt$block$1`”这个 block 实现类，对应的 `Continuation` 对象。

这行代码，其实就对应着协程被创建的时刻。

好，到这里，协程创建的逻辑就分析完了，我们再回到 `startCoroutine()` 的源码，看看它后续的逻辑。

 复制代码

```

1 // 代码段11
2
3 public fun <T> (suspend () -> T).startCoroutine(
4     completion: Continuation<T>
5 ) {
6     //
7     //
8     createCoroutineUnintercepted(completion).intercepted().resume(Unit)

```

注意这里



类似的，`intercepted()` 这个方法的源代码，我们也需要去 Kotlin 的源代码当中找到对应的 JVM 实现。

[复制代码](#)

```
1 // 代码段12
2
3 public actual fun <T> Continuation<T>.intercepted(): Continuation<T> =
4     (this as? ContinuationImpl)?.intercepted() ?: this
```

它的逻辑很简单，只是将 `Continuation` 强转成了 `ContinuationImpl`，调用了它的 `intercepted()`。这里有个细节，由于 `this` 的类型是“`LaunchUnderTheHoodKt$block$1`”，它是 `ContinuationImpl` 的子类，所以这个类型转换一定可以成功。

接下来，我们看看 `ContinuationImpl` 的源代码。

[复制代码](#)

```
1 // 代码段13
2
3 internal abstract class ContinuationImpl(
4     completion: Continuation<Any?>?,
5     private val _context: CoroutineContext?
6 ) : BaseContinuationImpl(completion) {
7
8     @Transient
9     private var intercepted: Continuation<Any?>? = null
10
11     public fun intercepted(): Continuation<Any?> =
12         intercepted
13         ?: (context[ContinuationInterceptor]?.interceptContinuation(this) ?
14             .also { intercepted = it }
15     }
```

这里其实就是通过 `ContinuationInterceptor`，对 `Continuation` 进行拦截，从而将程序的执行逻辑派发到特定的线程之上，这部分的逻辑我们在下一讲会再展开。

让我们回到 `startCoroutine()` 的源码，看看它的最后一步 `resume(Unit)`。

```

1 // 代码段14
2
3 public fun <T> (suspend () -> T).startCoroutine(
4     completion: Continuation<T>
5 ) {
6     //
7     //
8     createCoroutineUnintercepted(completion).intercepted().resume(Unit)
9 }

```

注意这里

↓

这里的 `resume(Unit)`，作用其实就相当于启动了协程。

好，现在我们已经弄清楚了 `startCoroutine()` 这个协程的基础 API 是如何启动协程的了。接下来，我们来看看中间层的 `launch{}` 函数是如何启动协程的。

## launch 是如何启动协程的？

在研究 `launch` 的源代码之前，我们先来写一个简单的 Demo：

```

1 // 代码段15
2
3 fun main() {
4     testLaunch()
5     Thread.sleep(2000L)
6 }
7
8 private fun testLaunch() {
9     val scope = CoroutineScope(Job())
10    scope.launch {
11        println("Hello!")
12        delay(1000L)
13        println("World!")
14    }
15 }
16
17 /*
18 输出结果：
19 Hello!
20 World!
21 */

```

然后，我们还是通过反编译，来看看它对应的 Java 代码长什么样：

```
1 // 代码段16
2
3 public final class LaunchUnderTheHoodKt {
4     public static final void main() {
5         testLaunch();
6         Thread.sleep(2000L);
7     }
8
9     private static final void testLaunch() {
10         CoroutineScope scope = CoroutineScopeKt.CoroutineScope((CoroutineContext)Jc
11             BuildersKt.launch$default(scope, null, null, new LaunchUnderTheHoodKt$testL
12     }
13
14     static final class LaunchUnderTheHoodKt$testLaunch$1 extends SuspendLambda ir
15         int label;
16
17     LaunchUnderTheHoodKt$testLaunch$1(Continuation $completion) {
18         super(2, $completion);
19     }
20
21     @Nullable
22     public final Object invokeSuspend(@NotNull Object $result) {
23         Object object = IntrinsicsKt.getCOROUTINE_SUSPENDED();
24         switch (this.label) {
25             case 0:
26                 ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
27                 System.out
28                     .println("Hello!");
29                 this.label = 1;
30                 if (DelayKt.delay(1000L, (Continuation)this) == object)
31                     return object;
32                 DelayKt.delay(1000L, (Continuation)this);
33                 System.out
34                     .println("World!");
35                 return Unit.INSTANCE;
36             case 1:
37                 ResultKt.throwOnFailure(SYNTHETIC_LOCAL_VARIABLE_1);
38                 System.out.println("World!");
39                 return Unit.INSTANCE;
40         }
41         throw new IllegalStateException("call to 'resume' before 'invoke' with co
42     }
43
44     @NotNull
45     public final Continuation<Unit> create(@Nullable Object value, @NotNull Con
46         return (Continuation<Unit>)new LaunchUnderTheHoodKt$testLaunch$1($complet
47     }
48
49     @Nullable
50     public final Object invoke(@NotNull CoroutineScope p1, @Nullable Continuati
51     return ((LaunchUnderTheHoodKt$testLaunch$1)create(p1, p2)).invokeSuspend(
```

```
52     }  
53 }  
54 }
```

有了前面的经验，上面的代码对我们来说就很简单了。唯一需要注意的是“LaunchUnderTheHoodKt\$testLaunch\$1”这个类，它其实对应的就是我们 `launch` 当中的 `Lambda`。

为了让它们之间的对应关系更加明显，我们可以换一种写法：

 复制代码

```
1 // 代码段17  
2  
3 private fun testLaunch() {  
4     val scope = CoroutineScope(Job())  
5     val block: suspend CoroutineScope.() -> Unit = {  
6         println("Hello!")  
7         delay(1000L)  
8         println("World!")  
9     }  
10    scope.launch(block = block)  
11 }
```

这段代码中的 `block`，其实就对应着“LaunchUnderTheHoodKt\$testLaunch\$1”这个类。这里的 `block`，本质上仍然是一个 **Continuation**。

好，接下来，我们来看看 `launch{}` 的源代码。

 复制代码

```
1 public fun CoroutineScope.launch(  
2     context: CoroutineContext = EmptyCoroutineContext,  
3     start: CoroutineStart = CoroutineStart.DEFAULT,  
4     block: suspend CoroutineScope.() -> Unit  
5 ): Job {  
6     // 1  
7     val newContext = newCoroutineContext(context)  
8     // 2  
9     val coroutine = if (start.isLazy)  
10        LazyStandaloneCoroutine(newContext, block) else  
11        StandaloneCoroutine(newContext, active = true)  
12    // 3  
13    coroutine.start(start, coroutine, block)
```

```
14     return coroutine
15 }
```

上面的代码一共有三个注释，我们也来分析一下：

- 注释 1，`launch` 会根据传入的 `CoroutineContext` 创建出新的 `Context`。
- 注释 2，`launch` 会根据传入的启动模式来创建对应的协程对象。这里有两种，一种是标准的，一种是懒加载的。
- 注释 3，尝试启动协程。

我们跟进 `coroutine.start()` 这个方法，会进入 `AbstractCoroutine` 这个抽象类：

 复制代码

```
1 public abstract class AbstractCoroutine<in T>(  
2     parentContext: CoroutineContext,  
3     initParentJob: Boolean,  
4     active: Boolean  
5 ) : JobSupport(active), Job, Continuation<T>, CoroutineScope {  
6  
7     // 省略  
8  
9     public fun <R> start(start: CoroutineStart, receiver: R, block: suspend R.()  
10         start(block, receiver, this)  
11     }  
12 }
```

到这里，我们其实就能看到，Java 当中有 `Thread.java` 对应线程的逻辑，而 Kotlin 协程当中，也有 `AbstractCoroutine.kt` 这个类对应协程的抽象逻辑。`AbstractCoroutine` 有一个 `start()` 方法，专门用于启动协程。

我们继续跟进 `start(block, receiver, this)`，就会进入 `CoroutineStart.invoke()`。

 复制代码

```
1 public enum class CoroutineStart {  
2     public operator fun <T> invoke(block: suspend () -> T, completion: Continua  
3     when (this) {  
4         DEFAULT -> block.startCoroutineCancellable(completion)  
5         ATOMIC -> block.startCoroutine(completion)  
6         UNDISPATCHED -> block.startCoroutineUndispatched(completion)  
7         LAZY -> Unit // will start lazily
```



```
8         }
9     }
```

在这个 `invoke()` 方法当中，它会根据 `launch` 传入的启动模式，以不同的方式启动协程。当我们的启动模式是 `ATOMIC` 的时候，就会调用 `block.startCoroutine(completion)`。而这个，其实就是我们在课程最开始研究过的 `startCoroutine()` 这个协程基础 API。

而另外两个方法，`startCoroutineUndispatched(completion)` 和 `startCoroutineCancellable(completion)`，我们从名字上也能判断出，它们只是在 `startCoroutine()` 的基础上增加了一些额外的功能而已。前者代表启动协程以后就不会被分发，后者代表启动以后可以响应取消。

然后，对于代码段 15 的 `launch` 逻辑而言，由于我们没有传入特定的启动模式，因此，这里会执行默认的模式，也就是调用“`startCoroutineCancellable(completion)`”这个方法。

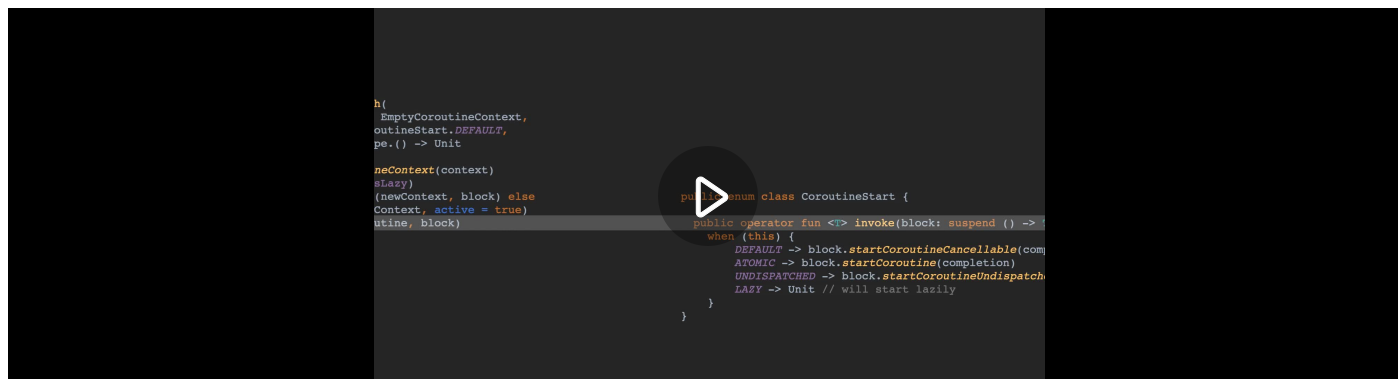
 复制代码

```
1 public fun <T> (suspend () -> T).startCoroutineCancellable(completion: Continua
2     // 1
3     createCoroutineUnintercepted(completion).intercepted().resumeCancellableWit
4 }
5
6 public actual fun <T> (suspend () -> T).createCoroutineUnintercepted(
7     completion: Continuation<T>
8 ): Continuation<Unit> {
9     val probeCompletion = probeCoroutineCreated(completion)
10
11     return if (this is BaseContinuationImpl)
12         // 2
13         create(probeCompletion)
14     else
15         createCoroutineFromSuspendFunction(probeCompletion) {
16             (this as Function1<Continuation<T>, Any??>).invoke(it)
17         }
18 }
```

那么，通过查看 `startCoroutineCancellable()` 的源代码，我们能发现，它最终还是会调用我们之前分析过的 `createCoroutineUnintercepted()`，而在它的内部，仍然会像我们之前分析过的，去调用 `create(probeCompletion)`，然后最终会调用代码段 16 当中“`LaunchUnderTheHoodKt$testLaunch$1`”的 `create()` 方法。

至此，launch 启动协程的整个过程，我们就已经分析完了。其实，launch 这个 API，只是对协程的基础元素 startCoroutine() 等方法进行了一些封装而已。

看完这么多的代码和文字，相信你可能已经有一些感觉了，不过可能对整个流程还是有些模糊。这里我做了一个视频，描述了 launch 的执行流程。



## 小结

createCoroutine{}、startCoroutine{}，它们是 Kotlin 提供的两个底层 API，前者是用来创建协程的，后者是用来创建并同时启动协程的。

通过反编译，我们发现，startCoroutine{} 最终会调用 createCoroutineUnintercepted() 这个函数，而它在 JVM 平台的实现，就是调用 Lambda 对应的实现类“LaunchUnderTheHoodKt\$block\$1”当中的 create() 方法。

另外，Kotlin 协程框架在**中间层**实现了 launch、async 之类的协程构建器（Builder），你要知道，它们只是对协程底层 API 进行了更好的封装而已。它们除了拥有启动协程的基础能力，还支持传入 CoroutineContext、CoroutineStart 等参数，前者可以帮我们实现结构化并发，后者可以支持更灵活的启动模式。

## 思考题

在代码段 3 当中，我们使用的是 suspend {} 启动的协程，它的类型是 suspend () -> String。那么，我们是否可以使用挂起函数启动协程呢？就像下面这样：

 复制代码

```
1 private suspend fun func(): String {
2     println("Hello!")
3     delay(1000L)
```

```


4     println("World!")
5     return "Result"
6 }
7
8 private fun testStartCoroutineForSuspend() {
9     val block = ::func
10
11     val continuation = object : Continuation<String> {
12         override val context: CoroutineContext
13             get() = EmptyCoroutineContext
14
15         override fun resumeWith(result: Result<String>) {
16             println("Result is: ${result.getOrNull()}")
17         }
18     }
19
20     block.startCoroutine(continuation)
21 }

```

如果使用这种方式启动协程，它的整体执行流程和代码段 3 会有什么不一样吗？欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 加餐五 | 深入理解协程基础元素

[下一篇](#) 29 | Dispatchers是如何工作的？

## 精选留言 (6)





杨小妞

2022-04-06

createCoroutineUnintercepted这个函数的JVM实现在哪个包，哪个类下呢？



辉哥

2022-03-29

startCoroutine -> createCoroutineUnintercepted -> createCoroutineFromSuspendFunction, 最终返回一个RestrictedContinuationImpl对象,然后调用其resume方法,从而调用block的invoke方法.最终调起协程.

作者回复: 很棒的解答!



L先生

2022-03-28

反编译了一下, block最终会转成function1。(this as Function1, Any?>).invoke(it)中的invoke是指的这个Function1中的invoke吗

作者回复: 接近了~~



L先生

2022-03-28

打印没啥区别啊。应该是走这里了。createCoroutineFromSuspendFunction(probeCompletion) { (this as Function1, Any?>).invoke(it) }。但是我看不太懂。this指什么, it又指什么参数

作者回复: 嗯, 方向已经对了。



Allen

2022-03-28

关于思考题的思考:

我认为执行流程及结果和代码段 3 中是完全一样的。因为

```
private suspend fun func(): String {  
    println("Hello!")  
    delay(1000L)  
    println("World!")  
    return "Result"  
} 和
```

```
val block = suspend {  
    println("Hello!")  
    delay(1000L)  
    println("World!")  
    "Result"  
}
```

完全是等价的写法。

作者回复: 写法是等价的，那么执行流程有变化吗？



**Paul Shan**

2022-03-28

思考题：调试了一下，结果是一样的。唯一的区别可能在于`block`原来被反编译成一个函数对象直接用实现状态机的`Continuation`对象赋值。加入函数赋值以后，`block`对象被实现为一个简单的内部类，这个内部类的`invoke`函数再去调用`Continuation`对象。

作者回复: 是的。

