讲真，Kotlin 协程的挂起没那么神秘(原理篇)

小鱼人爱编程 2022-06-20 8,168 阅读12分钟 关注

前言

协程系列文章：

- 一个小故事讲明白进程、线程、Kotlin 协程到底啥关系?
- 少年，你可知 Kotlin 协程最初的样子?
- 讲真，Kotlin 协程的挂起/恢复没那么神秘(故事篇)
- 讲真，Kotlin 协程的挂起/恢复没那么神秘(原理篇)
- Kotlin 协程调度切换线程是时候解开真相了
- Kotlin 协程之线程池探索之旅(与Java线程池PK)
- Kotlin 协程之取消与异常处理探索之旅(上)
- Kotlin 协程之取消与异常处理探索之旅(下)
- 来，跟我一起撸Kotlin runBlocking/launch/join/async/delay 原理&使用
- 继续来，同我一起撸Kotlin Channel 深水区
- Kotlin 协程 Select：看我如何多路复用
- Kotlin Sequence 是时候派上用场了
- Kotlin Flow啊，你将流向何方?
- Kotlin Flow 背压和线程切换竟然如此相似
- Kotlin SharedFlow&StateFlow 热流到底有多热?
- 狂飙吧，Lifecycle与协程、Flow的化学反应
- 来吧！接受Kotlin 协程--线程池的7个灵魂拷问
- 当，Kotlin Flow与Channel相逢
- 这一次，让Kotlin Flow 操作符真正好用起来

上篇从拟物的角度阐述了协程挂起/恢复的场景，相信大家对此应该有了一个感性的的认识。上上篇分析了如何开启一个原始的协程，相信大家也知道协程内部执行原理了。本篇将重点分析协程挂起与恢复的原理， 探究协程凭什么能挂起？它又为何能够在原地恢复?
通过本篇文章，你将了解到：

> 3、withContext 靠什么恢复协程?
> 4、不用withContext 如何挂起协程?
> 5、协程执行、挂起、恢复的全流程。

1、suspend 函数该怎么写?

suspend 写法初探

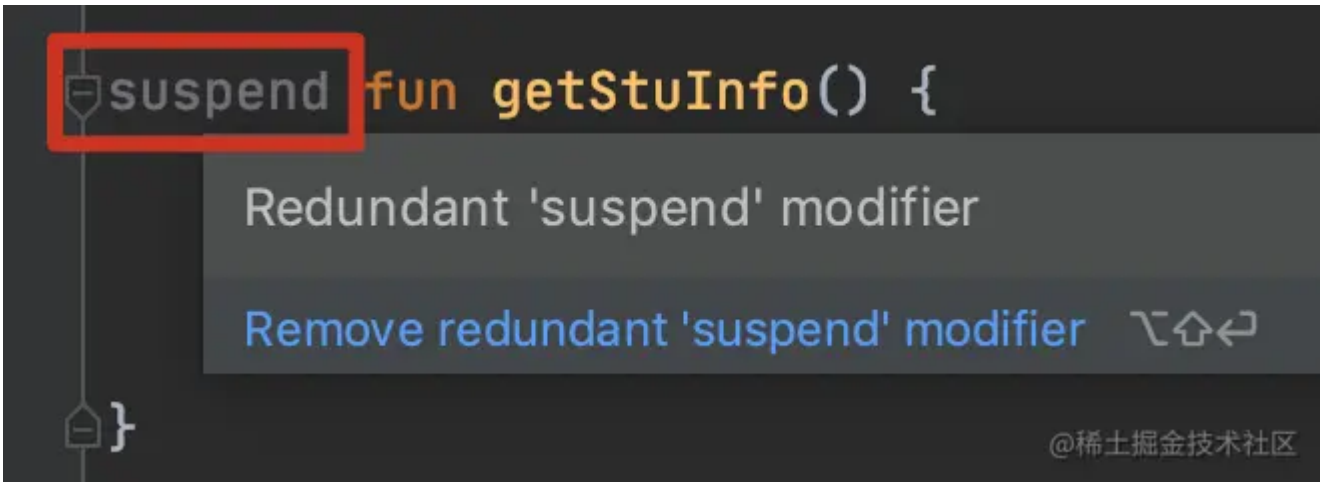
古有两小儿辩日，今有俩码农论协程。

小明说："挂起函数当然很容易写，不就是加个suspend吗？"

kotlin 复制代码

```kotlin
suspend fun getStuInfo() {
    println("after sleep")
}
```

小红说："你这样写不对，编译器会提示："



意思是挂起函数毫无意义，可以删除suspend 关键字。

小明说："那我这写的到底是不是挂起函数呢？"
小红："简单，遇事不决反编译。"

```
        String var1 = "after sleep";
        boolean var2 = false;
        System.out.println(var1);
        return Unit.INSTANCE;
    }
```

虽然带了Continuation 参数，但这个参数没有用武之地。
并且调用getStuInfo()的地方反编译查看：

java 复制代码

```java
    public final Object invokeSuspend(@NotNull Object $result) {
        //挂起标记
        Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
        String var2;
        boolean var3;
        switch(this.label) {
            case 0:
                ResultKt.throwOnFailure($result);
                var2 = "before suspend";
                var3 = false;
                System.out.println(var2);
                this.label = 1;
                //此处判断结果为false，因为getStuInfo 永远不会挂起
                if (CoroutineSuspendKt.getStuInfo(this) == var4) {
                    return var4;
                }
                break;
            case 1:
                ResultKt.throwOnFailure($result);
                break;
            default:
                throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine
        }

        var2 = "after suspend";
        var3 = false;
        System.out.println(var2);
        return Unit.INSTANCE;
    }
```

可以看出，在调用的地方会判断getStuInfo()是否会挂起，但结果是永远不会挂起。
综合定义与调用处可知：

小明计上心头说："挂起嘛，顾名思义就是阻塞。"

kotlin 复制代码

```kotlin
suspend fun getStuInfo() {
    Thread.sleep(2000)
    println("after sleep")
}
```

小红："然而这是线程的阻塞而非协程的挂起。"
小明："额，不阻塞原来的线程，那我再开个线程做耗时任务。"

kotlin 复制代码

```kotlin
suspend fun getStuInfo1() {
    thread {
        Thread.sleep(2000)
        println("after sleep")
    }
    println("after thread")
}
```

小红："这次虽然不阻塞原来的线程了，但是线程还是往下执行了(after thread 先于 after sleep 打印)，并不能挂起协程。"
小明："到底要我怎样？既不能阻塞线程又不能让线程继续执行后续的代码，这触及了我的知识盲区，我需要研究研究。"

delay 挂起协程

小明恶补了协程相关的知识点，信心满满找到小红展示成果。

kotlin 复制代码

```kotlin
suspend fun getStuInfo() {
    delay(5000)
    Log.d("fish", "after delay thread:${Thread.currentThread()}")
}
```

> - suspend 编译器不会再提示是冗余的了。
> - 反编译结果展示getStuInfo()的Continuation参数也有用了。
> - 反编译结果展示调用getStuInfo()有机会被挂起了。

小红看完称赞道："不错哦，有进步，看来是做了功课的。那我再问你个问题：你怎么证明调用getStuInfo()函数的线程没有被阻塞的呢？"
小明胸有成竹的说："这个我早有准备，且看我完整代码。"

kotlin 复制代码

```kotlin
        //点击UI
        binding.btnDelay.setOnClickListener {
            GlobalScope.launch(Dispatchers.Main) {
                //在主线程执行协程
                Log.d("fish", "before suspend thread:${Thread.currentThread()}")
                //执行挂起函数
                getStuInfo()
            }
            binding.btnDelay.postDelayed({
                //延迟2s在主线程执行打印
                Log.d("fish", "post thread:${Thread.currentThread()}")
            }, 2000)
        }

       suspend fun getStuInfo() {
          delay(5000)
          Log.d("fish", "after delay thread:${Thread.currentThread()}")
      }
```

最后打印结果如下：

```
17:32:38.217 21840-21840/com.fish.kotlindemo D/fish: before suspend thread:Thread[main,5,main]
17:32:40.218 21840-21840/com.fish.kotlindemo D/fish: post thread:Thread[main,5,main]
17:32:43.221 21840-21840/com.fish.kotlindemo D/fish: after delay thread:Thread[main,5,main]
```


getStuInfo()运行在主线程，该函数里将协程挂起5s，而在2s后在主线程里打印。

> 1、第三条语句5s后打印说明delay(5000)有效果，主线程在执行delay()后没有继续往下执行了。
> 2、第二条语句2s后打印说明主线程并没有被阻塞。

小红："理解很到位，我又有个问题了：挂起函数是在主线程执行的，那能否让它在子线程执行呢？在大部分的场景下，我们都需要在子线程执行耗时操作，子线程执行完毕后，主线程刷新UI。"
小明："容我三思..."

2、withContext 凭什么能挂起协程?

withContext 使用

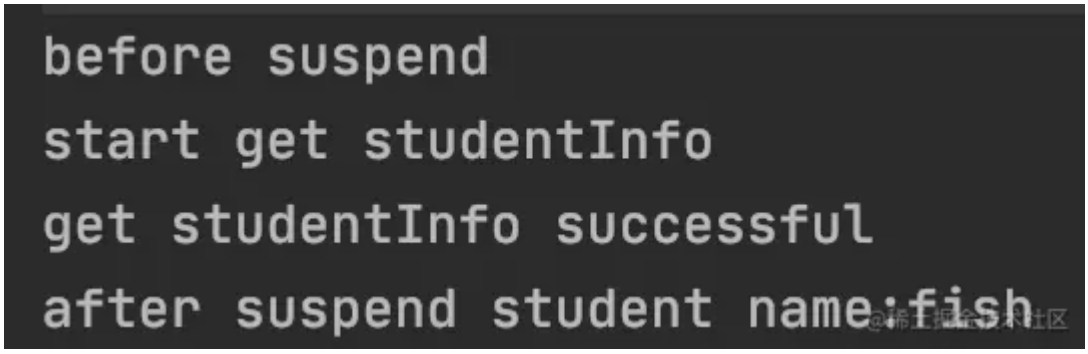
不用小明思考了，我们直接开撸源码。协程使用过程中除了launch/asyc/runBlocking/delay之外，想必还有一个函数比较熟悉：withContext。
刚接触时大家都使用它来切换线程用以执行新的协程(子协程)，而原来的协程(父协程)则被挂起。当子协程执行完毕后将会恢复父协程的运行。

kotlin 复制代码

```kotlin
fun main(array: Array<String>) {
    GlobalScope.launch() {
        println("before suspend")
        //挂起函数
        var studentInfo = getStuInfo2()
        //挂起函数执行返回
        println("after suspend student name:${studentInfo?.name}")
    }
    //防止进程退出
    Thread.sleep(1000000)
}

suspend fun getStuInfo2():StudentInfo {
    return withContext(Dispatchers.IO) {
        println("start get studentInfo")
        //模拟耗时操作
        Thread.sleep(3000)
        println("get studentInfo successful")
        //返回学生信息
        StudentInfo()
    }
}
```

查看打印结果：



从结果上来看，明明是异步调用，代码里却是用同步的方式表达出来，这就是协程的魅力所在。

withContext 原理

suspendCoroutineUninterceptedOrReturn 的理解

父协程为啥能挂起呢？这得从withContext 函数源码说起。

kotlin 复制代码

```kotlin
#Builders.common.kt
suspend fun <T> withContext(
    context: CoroutineContext,
    block: suspend CoroutineScope.() -> T
): T {
    return suspendCoroutineUninterceptedOrReturn sc@ { uCont ->
        val oldContext = uCont.context
        val newContext = oldContext + context
        //...
        //构造分发的协程
        val coroutine = DispatchedCoroutine(newContext, uCont)
        //开启协程
        block.startCoroutineCancellable(coroutine, coroutine)
        //获取协程结果
        coroutine.getResult()
    }
}
```

不管来源如何，先看它的参数，发现是Continuation类型的，这个参数是从哪来的呢？仔细看，原来是withContext 被suspend 修饰的，而suspend 修饰的函数会默认带一个Continuation类型的形参，这样就能关联起来了：

> suspendCoroutineUninterceptedOrReturn 传入的uCount 实参即为父协程的协程体。

将父协程的协程体存储到DispatchedCoroutine里，最后通过DispatchedCoroutine 分发。

getResult 的理解

直接看代码：

kotlin 复制代码

```kotlin
#DispatchedCoroutine类里
fun getResult(): Any? {
    //先判断是否需要挂起
    if (trySuspend()) return COROUTINE_SUSPENDED
    //如果无需挂起，说明协程已经执行完毕
    //那么需要将返回值返回
    val state = this.state.unboxState()
    if (state is CompletedExceptionally) throw state.cause
    //强转返回值到对应的类型
    return state as T
}

private fun trySuspend(): Boolean {
    //_decision 原子变量，三种值可选
    //private const val UNDECIDED = 0 默认值，未确定是1还是2
    //private const val SUSPENDED = 1 挂起
    //private const val RESUMED = 2   恢复
    _decision.loop { decision ->
        when (decision) {
            //若当前值为默认值，则修改为挂起，并且返回ture，表示需要挂起
            UNDECIDED -> if (this._decision.compareAndSet(UNDECIDED, SUSPENDED)) return true
            //当前已经是恢复状态，无需挂起，返回false
            RESUMED -> return false
            else -> error("Already suspended")
        }
```

也就是说当调用了coroutine.getResult() 后，该函数执行的返回值即为suspendCoroutineUninterceptedOrReturn 的返回值，进而是withContext 的返回值。

此时withContext 的返回值为：COROUTINE_SUSPENDED，它是个枚举值，表示协程执行到该函数需要挂起协程，也即是调用了withContext()函数的协程需要被挂起。

小结挂起逻辑：

> 1. withContext()函数记录当前调用它的协程，并开启一个新的协程。
> 2. 开启的新协程在指定的线程执行（提交给线程池或是提交给主线程执行任务）。
> 3. 判断新协程当前的状态，若是挂起则返回挂起状态，若是恢复状态则返回具体的返回值。

其中第2点只负责提交任务，耗时可以忽略。 第3点则是挂起与否的关键所在。

协程状态机

withContext()函数已经返回了，它的使命已经结束，关键是看谁在使用它的返回值做文章。

kotlin 复制代码

```kotlin
GlobalScope.launch() {
    println("before suspend")
    //挂起函数
    var studentInfo = getStuInfo2() //①
    //挂起函数执行返回
    println("after suspend student name:${studentInfo?.name}")//②
}
```

我们通俗的理解是：getStuInfo2()里调用了withContext()，而withContext() 返回了，那么getStuInfo2()也应当返回啊？而实际结果却是②的打印3s后才显示，说明实际情况是②的语句是3s后才执行。
luanch(){...}花括号里的内容我们称为协程体，而该协程体比较特殊，看起来是同步的写法，实际内部并不是同步执行，这部分在上上篇文章有分析，此处简单过一下。
老规矩，还是反编译看看花括号里的是啥内容。

```
        //状态机状态的值
        int label;
        @Nullable
        public final Object invokeSuspend(@NotNull Object $result) {
            //挂起状态
            Object var5 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
            Object var10000;
            switch(this.label) {
                case 0:
                    //默认为label == 0，第一次进来时
                    String var2 = "before suspend";
                    System.out.println(var2);
                    //状态流转为下一个状态
                    this.label = 1;
                    //执行挂起函数
                    var10000 = CoroutineSuspendKt.getStuInfo2(this);
                    if (var10000 == var5) {
                        //若是挂起，直接返回挂起值
                        return var5;
                    }
                    break;
                case 1:
                    //第二次进来时，走这，没有return，只是退出循环
                    ...
                    break;
                default:
                    throw new IllegalStateException("call to 'resume' before 'invoke' with c
            }
            //第二次进入走这，执行打印语句
            StudentInfo studentInfo = (StudentInfo)var10000;
            String var7 = "after suspend student name:" + (studentInfo != null ? studentInfo
            boolean var4 = false;
            System.out.println(var7);
            return Unit.INSTANCE;
        }
        @NotNull
        public final Continuation create(@Nullable Object value, @NotNull Continuation compl
            ...
            return var3;
        }
        public final Object invoke(Object var1, Object var2) {
            ...
        }
    }), 3, (Object)null);
```

第一次进入时默认为0，因此会调用 getStuInfo2()，而之前的分析表明该函数会返回挂起状态，因此此处检测到挂起状态后直接return 了，invokeSuspend() 执行结束。

invokeSuspend()返回值谁关注?

```kotlin
#BaseContinuationImpl 类成员方法
override fun resumeWith(result: Result<Any?>) {
    var current = this
    var param = result
    while (true) {
        ...
        with(current) {
            val completion = completion!! // fail fast when trying to resume continuation withou
            val outcome: Result<Any?> =
                try {
                    //执行协程体
                    val outcome = invokeSuspend(param)
                    //若是挂起，则直接return
                    if (outcome === kotlin.coroutines.intrinsics.COROUTINE_SUSPENDED) return
                    kotlin.Result.success(outcome)
                } catch (exception: Throwable) {
                    kotlin.Result.failure(exception)
                }
            //恢复逻辑
            //...
        }
    }
}
```

GlobalScope.launch() 函数本身会执行resumeWith()函数，该函数里执行invokeSuspend()，invokeSuspend()里会执行协程体，也即是GlobalScope.launch()花括号里的内容。

至此就比较明了了：

> GlobalScope.launch() 最终会执行闭包(协程体)，遇到挂起函数getStuInfo2()时将不会再执行挂起函数后的代码直到被恢复。

3、withContext 靠什么恢复协程?

GlobalScope.launch() 启动的协程在调用getStuInfo2()后就挂起了，它啥时候会恢复执行呢？也就是说协程状态机啥时候会走到label=1的分支？
从上节分析可知，withContext(){} 花括号里的内容(协程体)将会被调度执行，既然是协程体当然还是要反编译查看。

java 复制代码

```java
public static final Object getStuInfo2(@NotNull Continuation $completion) {
    return BuildersKt.withContext((CoroutineContext)Dispatchers.getIO(), (Function2)(new Fun
        //状态机的值
        int label;

        @Nullable
        public final Object invokeSuspend(@NotNull Object var1) {
            //挂起值
            Object var4 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
            switch(this.label) {
                case 0:
                    //默认走这，正常协程体里的内容
                    String var2 = "start get studentInfo";
                    System.out.println(var2);
                    Thread.sleep(3000L);
                    var2 = "get studentInfo successful";
                    System.out.println(var2);
                    //返回对象
                    return new StudentInfo();
                    //...
            }
        }
        ...
    }), $completion);
}
```

此时的状态机只有一个状态，说明withContext() 协程体里没有调用挂起的函数。
继续查看是谁关注了invokeSuspend()的返回值，也就是谁调用了它。

协程体调用

协程的恢复离不开 resumeWith()函数

```
override fun resumeWith(result: Result<Any?>) {
    var current = this
    var param = result
    while (true) {
        with(current) {
            //completion 可能是父协程的协程体(或是包装后的)，也即是当前协程体执行完成后
            //需要通知之前的协程体
            val completion = completion!! // fail fast when trying to resume continuation withou
            val outcome: Result<Any?> =
                try {
                    //调用协程体
                    val outcome = invokeSuspend(param)
                    //如果是挂起则直接返回
                    if (outcome === kotlin.coroutines.intrinsics.COROUTINE_SUSPENDED) return
                    kotlin.Result.success(outcome)
                } catch (exception: Throwable) {
                    kotlin.Result.failure(exception)
                }
            if (completion is BaseContinuationImpl) {
                // 仅仅记录 ①
                current = completion
                param = outcome
            } else {
                //执行恢复逻辑 ②
                completion.resumeWith(outcome)
                return
            }
        }
    }
}
```

对于Demo 里的withContext()函数的协程体来说，因为它没有调用任何挂起的函数，因此此处invokeSuspend(param) 返回的结果将是对象，"outcome === kotlin.coroutines.intrinsics.COROUTINE_SUSPENDED" 判断不满足，继续往下执行。completion 的类型至关重要，而我们的Demo里completion 是DispatchedCoroutine(包装后的)，它的成员变量uCont表示的即是父协程的协程体，最终uCont 分发任务会执行：协程体的resumeWith(outcome)，outCome 为 StudentInfo 对象。

到这就比较有趣了，我们之前分析过GlobalScope.launch()的协程体执行是因为调用了resueWith()，而此处也是调用了resumeWith()，最终都会调用到invokeSuspend()，而该函数就是真正执行了协程体。
此次调用已经属于第二次调用invokeSuspend()，之前第一次调用后label=0变为label=1，因

最后再小结一下withContext()函数恢复父协程的原理：

> 1. 调用withContext()时传入父协程的协程体。
> 2. 当withContext()的协程体执行完毕后会判断completion。
> 3. completion 即为1的协程体包装类：DispatchedCoroutine。
> 4. completion.resumeWith() 最后执行invokeSuspend()，通过状态机流转执行之前挂起逻辑之后的代码。
> 5. 整个父协程体就执行完毕了。

协程恢复关键的俩字：**回调。**
协程表面上写法很简洁，云淡风轻，实际内部将回调利用起来，这就是协程原理的冰山之下的内容。

4、不用withContext 如何挂起协程?

上个小结只是关心协程挂起与恢复的核心原理，有意避开了launch/withContext里有关协程调度器的问题(这部分下篇分析），可能有的小伙伴觉得没有完全弄明白，没关系，和启动原始协程一样，这次我们也通过原始的方法挂起协程，这样就摒除调度器逻辑的影响，专注于挂起的本身。

协程挂起的核心要点

回过头看看delay的实现：

kotlin 复制代码

```kotlin
suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // don't delay
    return suspendCancellableCoroutine sc@ { cont: CancellableContinuation<Unit> ->
        if (timeMillis < Long.MAX_VALUE) {
            //提交给Loop进行超时任务的调度
            cont.context.delay.scheduleResumeAfterDelay(timeMillis, cont)
        }
    }
}
```

```
): T =
    suspendCoroutineUninterceptedOrReturn { uCont ->
        val cancellable = CancellableContinuationImpl(uCont.intercepted(), resumeMode = MODE_CAN
        cancellable.initCancellability()
        //开始调度
        block(cancellable)
        //返回结果
        cancellable.getResult()
    }
```

你发现了和withContext()函数的共同点了吗?
没错，就是：suspendCoroutineUninterceptedOrReturn 函数。
它的作用就是将父协程的协程体传递给其它协程/调度器。
想当然地我们也可以模仿withContext、delay利用它来做文章。

原始协程的挂起

复用之前的原始协程的创建：

kotlin 复制代码

```kotlin
fun <T> launchFish(block: suspend () -> T) {
    //创建协程，返回值为SafeContinuation(实现了Continuation 接口)
    //入参为Continuation 类型，参数名为completion，顾名思义就是
    //协程结束后(正常返回&抛出异常）将会调用它。
    var coroutine = block.createCoroutine(object : Continuation<T> {
        override val context: CoroutineContext
            get() = EmptyCoroutineContext

        //协程结束后调用该函数
        override fun resumeWith(result: Result<T>) {
            println("result:$result")
        }
    })
    //开启协程
    coroutine.resume(Unit)
}
```

再编写协程挂起函数：

```
        thread {
            //开启线程执行耗时任务
            Thread.sleep(3000)
            var studentInfo = StudentInfo()
            println("resume coroutine")
            //恢复协程,it指代 Continuation
            it.resumeWith(Result.success(studentInfo))
        }
        println("suspendCoroutine end")
    }
}
```

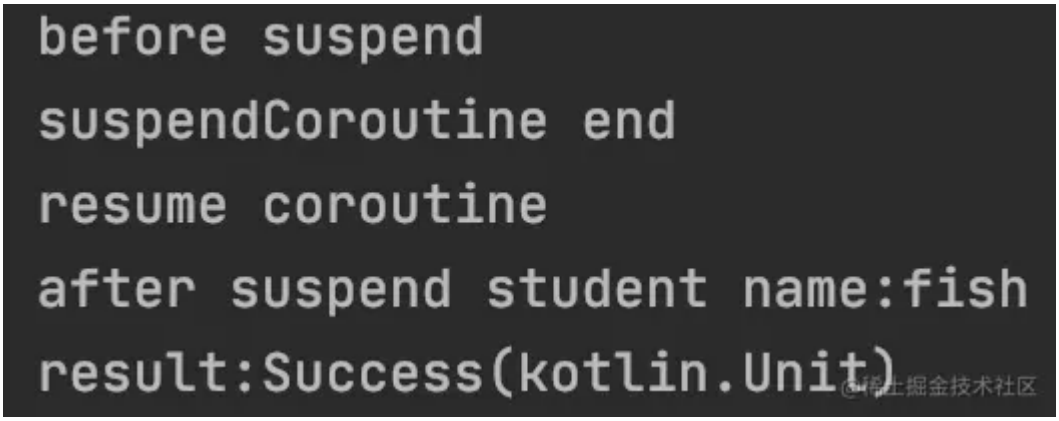
getStuInfo3()即为一个有效的挂起函数，它通过开启子线程执行耗时任务，执行完毕后恢复协程。

最后创建和挂起结合使用：

kotlin 复制代码

```kotlin
fun main(array: Array<String>) {
    launchFish {
        println("before suspend")
        var studentInfo = getStuInfo3()
        //挂起函数执行返回
        println("after suspend student name:${studentInfo?.name}")
    }
    //防止进程退出
    Thread.sleep(1000000)
}
```

运行效果：



原始协程挂起原理

重点看suspendCoroutine()函数：

kotlin 复制代码

```kotlin
#Continuation.kt
psuspend inline fun <T> suspendCoroutine(crossinline block: (Continuation<T>) -> Unit): T {
    contract { callsInPlace(block, InvocationKind.EXACTLY_ONCE) }
    return suspendCoroutineUninterceptedOrReturn { c: Continuation<T> ->
        //传入的c 为父协程的协程体
        //c.intercepted() 为检测拦截器，demo里没有拦截器用自身，也就是c
        val safe = SafeContinuation(c.intercepted())
        //执行函数，也就是子协程体
        block(safe)
        //检测返回值
        safe.getOrThrow()
    }
```

与withContext()函数相似，最终都调用了 suspendCoroutineUninterceptedOrReturn() 函数。

kotlin 复制代码

```kotlin
#SafeContinuationJvm.kt
internal actual fun getOrThrow(): Any? {
    //原子变量
    var result = this.result
    //如果是默认值，则将它修改为挂起状态，并返回挂起状态
    if (result === CoroutineSingletons.UNDECIDED) {
        if (SafeContinuation.RESULT.compareAndSet(this,
                CoroutineSingletons.UNDECIDED, COROUTINE_SUSPENDED)) return COROUTINE_SUSPENDED
        result = this.result // reread volatile var
    }
    return when {
        result === CoroutineSingletons.RESUMED -> COROUTINE_SUSPENDED // already called continua
        result is Result.Failure -> throw result.exception
        //挂起或者正常数据返回走这
        else -> result // either COROUTINE_SUSPENDED or data
    }
}
```

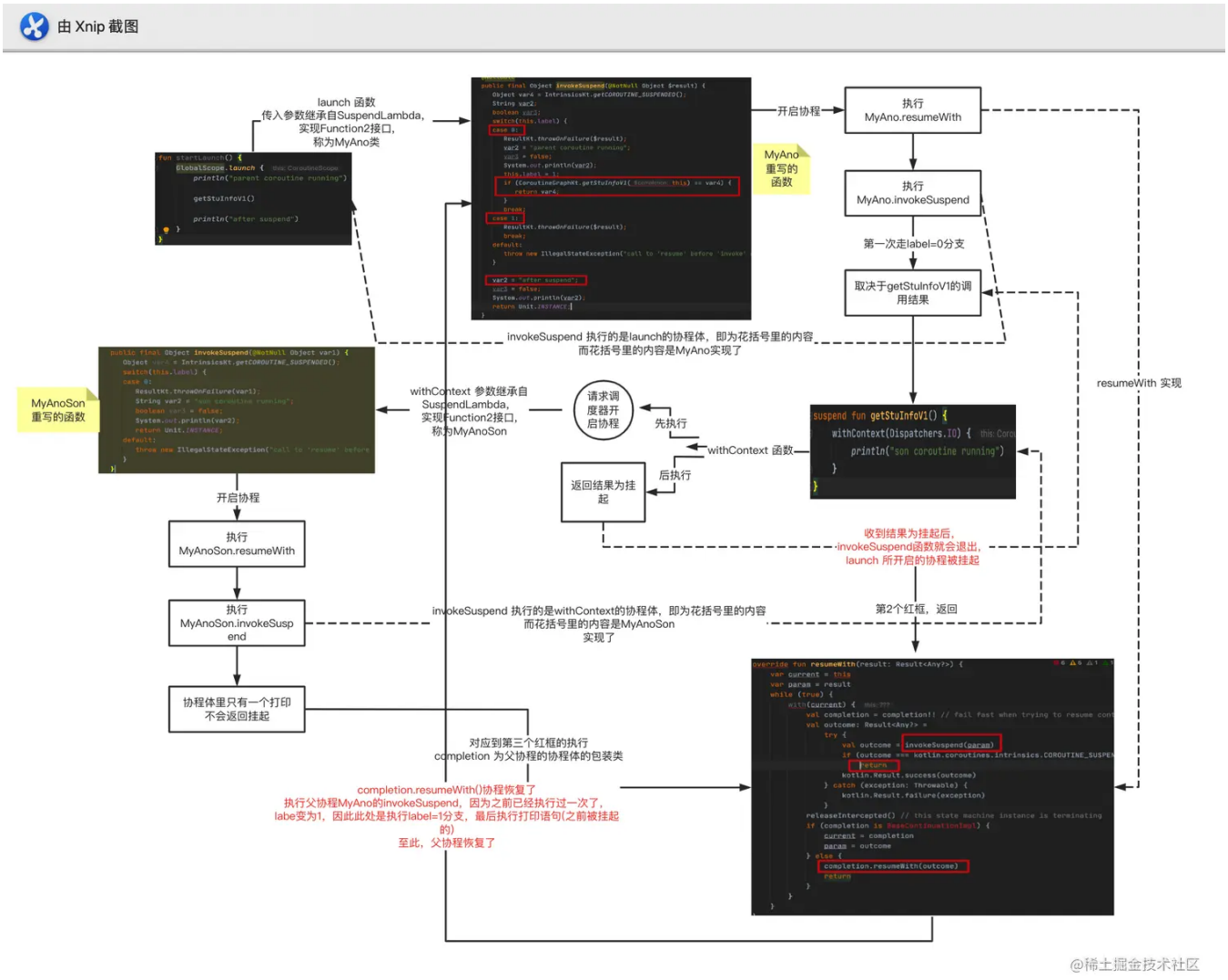
对比withContext、delay、suspendCoroutine 的返回值：

```kotlin
coroutine.getResult() //withContext
cancellable.getResult()//delay
safe.getOrThrow()//suspendCoroutine
```

都是判断当前协程的状态，用来给外部协程确定是否需要挂起自身。

5、协程执行、挂起、恢复的全流程

行文至此，相信大家对协程的挂起与恢复原理有了一定的认识，将这些点串联起来，用图表示：



图上对应的代码：

```kotlin
fun startLaunch() {
    GlobalScope.launch {
        println("parent coroutine running")

        getStuInfoV1()

        println("after suspend")
    }
}
suspend fun getStuInfoV1() {
    withContext(Dispatchers.IO) {
        println("son coroutine running")
    }
}
```

至于反编译结果，此处就不展示了，使用Android Studio 可以很方便展示。
代码和图对着看，相信大家一定会对协程开启、挂起、恢复有个全局的认识。

下篇我们将会深入分析协程提供的一些易用API，launch/async/runBlocking 等的使用及其原理。

本文基于Kotlin 1.5.3，文中完整Demo请点击

您若喜欢，请点赞、关注，您的鼓励是我前进的动力

持续更新中，和我一起步步为营系统、深入学习Android/Kotlin

1、Android各种Context的前世今生
2、Android DecorView 必知必会
3、Window/WindowManager 不可不知之事
4、View Measure/Layout/Draw 真明白了
5、Android事件分发全套服务

8、Android事件驱动Handler-Message-Looper解析
9、Android 键盘一招搞定
10、Android 各种坐标彻底明了
11、Android Activity/Window/View 的background
12、Android Activity创建到View的显示过
13、Android IPC 系列
14、Android 存储系列
15、Java 并发系列不再疑惑
16、Java 线程池系列
17、Android Jetpack 前置基础系列
18、Android Jetpack 易学易懂系列
19、Kotlin 轻松入门系列

标签： Android Kotlin 面试

本文收录于以下专栏

Kotlin 从现在开始学 专栏目录
系统学习Kotlin，由浅入深，环环相扣
166 订阅 · 26 篇文章

订阅

上一篇 讲真，Kotlin 协程的挂起没那...
下一篇 Kotlin 协程调度切换线程是时...

评论 29

RebornXXW Android

终uCont 分发任务会执行：协程体的resumeWith(outcome)，outCome 为 StudentInfo 对象。

对于这一块感觉可以更清楚一些~，因为withContext中resumeWith发现没有挂起以...

展开

2月前 点赞 1

小鱼人爱编程 作者：可以的

2月前 点赞 回复

Rookie_ LV.3 散人 @落云宗

该如何像你一样查看反编译后的代码呢

3月前 点赞 1

小鱼人爱编程 作者：juejin.cn

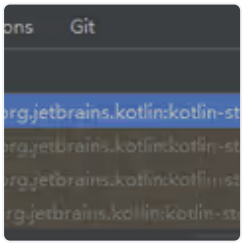
3月前 点赞 回复

Airoure LV.3

SafeContinuationJvm为啥我全局搜索找不到这个类呢

4月前 点赞 2

小鱼人爱编程 作者：看图



4月前 点赞 回复

Airoure 回复 小鱼人爱编程 作者：找到了，感谢！

4月前 点赞 回复

查看全部 29 条评论 >

目录 收起

您若喜欢，请点赞、关注，您的鼓励是我前进的动力

持续更新中，和我一起步步为营系统、深入学习Android/Kotlin

相关推荐

讲真，Kotlin 协程的挂起没那么神秘(故事篇)

4.7k阅读 · 24点赞

Kotlin 协程调度切换线程是时候解开真相了

7.4k阅读 · 47点赞

Epoxy - 在RecyclerView中构建复杂界面 - 7

154阅读 · 0点赞

RxJava 沉思录（二）：空间维度

14k阅读 · 230点赞

Kotlin契约（Contract）

精选内容

【网络安全】「漏洞复现」（四）NodeBB 被爆未授权拒绝服务攻击
sidiot · 766阅读 · 20点赞

JSON
忧郁的大喷菇 · 748阅读 · 1点赞

客服发送一条消息背后的技术和思考
得物技术 · 879阅读 · 1点赞

MySQL查询百万级数据分页查询优化实验记录
sumAll · 827阅读 · 3点赞

jenkins实践篇(2)—— 自动打tag的可回滚发布模式
蓝胖子的编程梦 · 720阅读 · 0点赞

为你推荐

Android Binder 原理换个姿势就顿悟了(图文版)
小鱼人爱编程 | 1年前 | 15k 188 27 Android 面试 APP

来吧！接受Kotlin 协程--线程池的7个灵魂拷问
小鱼人爱编程 | 8月前 | 12k 150 27 Kotlin Android ... 面试

一个小故事讲明白进程、线程、Kotlin 协程到底啥关系?
小鱼人爱编程 | 1年前 | 9.7k 121 29 Android Kotlin 面试

狂飙吧，Lifecycle与协程、Flow的化学反应
小鱼人爱编程 | 8月前 | 9.6k 109 10 Android Kotlin 面试

这一次，让Kotlin Flow 操作符真正好用起来
小鱼人爱编程 | 6月前 | 7.4k 101 14 Kotlin 源码 面试

Kotlin SharedFlow&StateFlow 热流到底有多热?
小鱼人爱编程 | 9月前 | 6.4k 119 28 Android Kotlin 面试

Android-软键盘一招搞定(实践篇)
小鱼人爱编程 | 2年前 | 8.4k 30 6 前端 Android

Kotlin 协程调度切换线程是时候解开真相了

小鱼人爱编程 | 1年前 | 7.4k 47 11 Android Kotlin 面试

少年，你可知 Kotlin 协程最初的样子?

小鱼人爱编程 | 1年前 | 6.6k 52 30 Android Kotlin 面试

Java切换到Kotlin，Crash率上升了?

小鱼人爱编程 | 1月前 | 6.4k 56 7 面试 Kotlin Java

Kotlin Flow啊，你将流向何方?

小鱼人爱编程 | 11月前 | 6.5k 48 10 Android Kotlin 架构

Kotlin Sequence 是时候派上用场了

小鱼人爱编程 | 1年前 | 5.9k 53 10 Kotlin Android 面试

当，Kotlin Flow与Channel相逢

小鱼人爱编程 | 6月前 | 5.7k 60 7 Kotlin 源码阅读 面试

Kotlin 协程 Select：看我如何多路复用

小鱼人爱编程 | 1年前 | 5.6k 50 4 Android Kotlin 面试